CJKspace

GNU 과학 계산 Library

GNU 과학계산 라이브러리

릴리스 2.7

Mark Galassi

Jim Davies

James Theiler

Brian Gough

Gerard Jungman

Patrick Alken

Michael Booth

Fabrice Rossi

Rhys Ulerich

저

번역: 김현성

2022년 04월 18일

차례

서문

참고: 이 번역 프로젝트는 현재 진행중에 있습니다.

본 서적은 GNU Scientific Library(GSL) 2.7 manual과 관련 자료들을 한글로 번역한 책입니다. python api를 제공하는 라이브러리(예: `numpy`, `scipy` 등)들은 한글로 된 자료들도 많지만, C의 경우 많은 입문서에도 불구하고 과학 계산 라이브러리 쪽에서 한글 자료들이 빈약해 번역을 시작하게 되었습니다.

GSL 2.7 Manual을 기준으로 해서 번역합니다. GSL이 차후 버전으로 갱신될 시 해당 버전의 변경점들도 같이 번역할 예정입니다. 번역은 최대한 한글로 풀어씀을 목적으로 합니다.

구체적으로 번역자 본인이 사용에 참고하려고 번역하는 책이기에 참고와 사용에 유용한 정보들도 같이 포함합니다. 원본 사용 설명서는 라이브러리의 사용에 필요한 상세한 내용을 전부 기술하고 있지는 않습니다. 필요에 따라 본래 설명서에 없는 추가적인 정보들을 문장과 단원에 넣을 것 입니다. 그러한 정보들은 끝에 다음과 같은 기호 '(*)'를 써서 나타낼 것입니다(*).

```
Additional Contents (*)
#.# Additional Contents Chapter (*)
    Additional sentence (*).

footnote: additional information (*)
```

이 번역서의 원문인 GSL 2.7 Manual은 다음 사이트에서 확인할 수 있습니다.

https://www.gnu.org/software/gsl/doc/html/index.html

다음 사이트 주소에서 해당 문서를 PDF로 다운로드할 수도 있습니다.

https://www.gnu.org/software/gsl/doc/latex/gsl-ref.pdf

해당 웹 문서들과 pdf 문서들은 Sphinx 프로그램을 이용해 만들어졌습니다. Sphinx를 위한 사용 설명서의 원본 문서들은 `.rst`[1] 파일 형식으로 GSL의 베포 파일 내부 `/doc/` 디렉토리에 예제 코드와 그림 등과 함께 저장되어 있습니다.

Sphinx에 관한 자세한 내용은 공식 홈페이지인 https://www.sphinx-doc.org/en/master/ 를 참고할 수 있습니다. 해당 프로그램에서 쓰는 rst 파일 형식은 https://docutils.sourceforge.io/rst.html 를 참고하길 바랍니다.

본 번역서의 원본은 https://github.com/HYUNSEONG-KIM/GSL_KOR_MANUAL.git 에서 찾을 수 있습니다. 디지털 문서와 PDF 파일은 Readthedocs 호스팅 서비스를 이용하고 있습니다. 다음 주소로 접근할 수 있습니다.

https://gsl-kor-manual.readthedocs.io/ko/latest/

[1] ReStructuredText의 약자입니다.

문서에서 오류가 발견되거나 번역에 기여하고 싶다면 번역서의 Git 저장소에 branch로 만들어 수정을 하거나 다음 이메일로 알려주길 바랍니다.

qwqwhsnote@gm.gist.ac.kr

저작권 고지

제 1 장

소개

GNU Scientific Library (GSL)은 수치 계산을 위한 라이브러리로 C로 작성되었습니다. 라이브러리에서 C-api를 제공하며, 다른 고급언어를 위한 인터페이스도 존재합니다. 또 이러한 인터페이스의 작성도 허용됩니다. 이 라이브러리의 코드는 **GNU 일반 공중 사용 허가서** 아래에서 베포됩니다.

1.1 GSL의 기능

본 라이브러리는 수치 계산의 다양한 분야를 지원합니다. 다음은 GSL에서 제공하는 수치해석 기능들입니다.

복소수	다항식의 근 탐색	특수 함수들
벡터와 행렬	순열	조합
정렬	BLAS 지원	선형 대수학
CBLAS 라이브러리	고속 푸리에 변환	고유계 문제의 해답
난수	Quadrature	난수 분포
Quasi-Random Sequences	히스토그램	통계
몬테 카를로 적분	N-튜플	미분 방정식의 해
담금질 기법	수치 미분	보간법
급수 가속	체비쇼프 근사	근 탐색
이산 한켈 변환	최소 제곱법	최소 지점 탐색
IEEE 부동 소수 대수	물리 상수	B-스플라인
웨이블렛 변환	희소 BLAS 지원	희소 선형 대수학

이 책의 각 단원들은 해당 단원에서 제공하는 기능들에 대해 함수의 상세한 정의, 예시 프로그램과 기반이 되는 알고리즘의 참고 문헌들을 제공합니다.

몇몇 구현체들은 **FFTPACK** 이나 **QUADPACK** 등과 같이 공공 패키지에 기반하기도합니다. GSL의 개발자들은 이러한 패키지들을 현대적인 코딩 규약으로 재구성하기도 했습니다.

1.2 자유 소프트웨어

GNU Scientific Library의 하위 프로그램들은 "자유 소프트웨어"입니다. 이 뜻은 프로그램들의 사용과 배포를 자유롭게 할 수 있다는 뜻입니다. 그러나 이 라이브러리는 **자유 이용 저작물**[1] 이 아닙니다. 엄연히 저작권이 존재하며, 배포와 사용에서 특정한 조건을 만족해야 합니다.

GSL의 저작권은 여러 기여자들의 협동과 같은 긍정적인 행위들을 보호하고 권장할 수 있도록 고안되었습니다. 이 저작권은 다른 사람들이 당신이 GSL을 이용해 작성한 소프트웨어의 어느 특정 버전을 얻지 못하도록 하는 행위를 금지합니다.

구체적으로 GNU 과학 라이브러리를 사용하는 프로그램의 복사본의 공유, 소스 코드가 필요할 시의 입수 가능성, 프로그램의 변경 가능성, 새로운 무료 프로그램에서의 사용등을 의미합니다.

이 행위들이 허가서 아래에서 허용되는 작업들입니다. 소프트웨어의 개발자는 해당 행위들이 가능하도록 해야합니다.

모두가 이러한 권리를 공유하기 위해, 어느 특정인이 타인에게 해당 권리를 박탈하는 행위를 금지합니다. 예시로 GNU 과학 라이브러리를 사용하는 모든 코드의 복사본을 베포하는 경우, 당신이 작성한 코드를 받는 사람은 당신이 GNU 과학 라이브러리를 사용할 때 받은 모든 권한을 그대로 타인에게 주어야하고(소스코드의 제공을 포함합니다), 해당 권리를 공지해야 함을 의미합니다. 다시말해, 이 라이브러리는 독점 프로그램에 재분배 되어서는 안된다는 것을 의미합니다[2] .

그리고 분명히 하고 넘어가야 할 점은 GNU Scientific Library는 제공하는 기능, 문서들에 대해 어느 보증도 존재하지 않습니다. 만약 프로그램들이 누군가에 의해 수정되고 배포될 경우, 그 작업물을 받는 사람들은 GNU 공식 베포가 아닌 것을 알고 있어야 합니다. 그러므로 다른 수정에 의한 문제들은 본 라이브러리에 반영되지 않을 것입니다.

GNU Scientific Library와 관련된 소프트웨어의 배포에 관한 자세한 조건은 **GNU General Public License** 의 원문을 참고할 수 있습니다. 좀 더 자세한 정보를 얻고 싶다면 GNU 프로젝트의 웹페이지에서 **GNU GPL에 자주 묻는 질문들** 을 참고할 수 있습니다. 다음의 주소로 이 둘에 대한 자세한 정보를 얻을 수 있습니다.

- **GNU General Public License**

  https://www.gnu.org/software/gsl/doc/html/#GNU-General-Public-License

- **GNU GPL에 자주 묻는 질문들**

  https://www.gnu.org/licenses/gpl-faq.html

[1] 자유 이용 저작물(Public domain)은 저작권이 소멸 되었거나 저작자가 저작권을 포기한 저작물을 말합니다.

[2] 상업적 이용은 이 문건과 관계 없습니다. 많은 상업 프로그램들이 GNU 공중 사용 허가서로 베포되는 코드를 포함하고 있으며, 그들의 경우 소스 코드를 다운받을 수 있는 방안들을 제공해 GNU 공중 사용 허가서를 따르고 있습니다.

자유 소프트웨어 재단 또한 상업 이용을 위한 저작권 컨설팅을 제공합니다. 자세한 사항은 **FSF** 에서 확인할 수 있습니다.

<https://www.fsf.org/>

1.3 GSL 얻기

라이브러리의 소스코드는 다양한 방법으로 얻을 수 있습니다. 친구로부터 복사하거나, CDROM을 구입, 아니면 인터넷에서 다운로드 할 수도 있습니다.

GNU 홈페이지에서 이러한 소스코드들을 다운 받을 수 있는 **FTP** 서버들의 목록을 제공하고 있습니다.

http://www.gnu.org/software/gsl/

라이브러리를 사용할 때, GNU 시스템을 이용하는 것을 권장합니다. GNU C complier와 GNU C Library에서 본 라이브러리는 몇몇 추가 기능들을 활용할 수 있습니다. 하지만, 이 라이브러리는 완전히 이식 가능하도록 구현되었기 때문에, 굳이 해당하는 시스템을 사용하지 않더라도, C 컴파일러가 있는 대부분의 시스템에서 사용가능합니다.

새로운 배포, 갱신 사항 그리고 다른 관련 소식들은 <info-gsl@gnu.org>에서 알림을 받을 수 있습니다. 이메일을 통해 알림을 받고 싶다면 다음과 같은 형식으로 이메일을 보내면 됩니다.

```
To: info-gsl-request@gnu.org
Subject: subscribe
```

보내고 나면 구독 요청 확인 답장이 올 것입니다.

1.4 비보증성

이 사용 설명서에서 설명하고 있는 소프트웨어는 어떤 보증도 존재하지 않습니다. 이 소프트웨어는 “있는 그대로” 제공됩니다. 상업 베포자로부터의 유료 보증 서비스의 이용, 제공된 코드의 정확한 사용, 그리고 기능의 확인등은 모두 사용자 본인의 책임입니다.

자세한 내용은 **GNU 일반 공중 사용 허가서** 를 참고하길 바랍니다.

1.5 버그 보고

알려진 버그들의 목록은 GSL 배포판의 BUGS 파일이나 온라인 GSL 버그 추척기에서 찾을 수 있습니다[3] . 컴파일 문제에 대한 자세한 내용은 INSTALL 파일에서 찾을 수 있습니다.

만약, 이러한 버그 목록에 없는 버그를 발견 했을 경우 <bug-gsl@gnu.org>로 보고해주시기를 바랍니다.

모든 버그 보고는 다음을 포함하고 있어야합니다.

- GSL의 버전 숫자
- 하드웨어와 운영체제
- 사용된 컴파일러와 컴파일러의 버전, 컴파일 옵션
- 버그를 만드는 짧은 프로그램

같은 문제가 라이브러리를 최적화 없이 컴파일 할 때 발생하는지, 아닌지를 확인해 준다면 더욱 유용합니다.

이 설명서의 오류나 누락도 같은 주소로 보고할 수 있습니다.

1.6 자세한 정보

이 사용 설명서의 온라인 복사본과 더 자세한 정보, 관련된 프로젝트, 이메일 리스트들은 언급한 사이트들에서 찾을 수 있습니다.

어떤 질문이나 라이브러리 설치에 관한 것들은 <help-gsl@gnu.org>에서 주는 목록에 물어 볼 수 있습니다. 본 리스트를 구독하려면 다음과 같이 이메일을 보내면 됩니다.

```
To: help-gsl-request@gnu.org
Subject: subscribe
```

이 이메일 리스트는 이 사용 설명서에서 다루지 않는 내용에 대해 묻거나 라이브러리 개발자들과 연락하는데 쓸 수 있습니다.

GNU Scientific Library를 저널의 문서에 참고 문헌으로 넣고 싶다면, 이 사용 설명서를 넣는 것을 추천합니다. 예시로 다음과 같이 인용할 수 있습니다.

```
M. Galassi et al, GNU Scientific Library Reference Manual (3rd Ed.), ISBN 0954612078
```

만약 주소를 넣고 싶다면,

```
http://www.gnu.org/software/gsl/
```

를 사용하면 됩니다.

[3] <http://savannah.gnu.org/bugs/?group=gsl>

1.7 규약

이 사용 설명서에서는 키보드로 작성해야하는 많은 예시들을 포함하고 있습니다. 터미널에서 작성해야하는 경우 다음과 같이 작성됩니다.

```
$command
```

줄의 첫번째 문자는 터미널 프롬프트를 나타내고 명령어를 작성할 때 작성하지 말아야 할 부분입니다. 어떤 시스템에서는 다른 기호를 사용하기도 하지만, **$** 는 터미널 프롬프트의 표준 기호로 본 사용 설명서에서 쓰입니다.

이 사용 설명서에서 GNU Scientific Library는 앞으로 **GSL** 이란 단어로 쓰입니다.

제 2 장

라이브러리의 사용

이 단원에서는 GSL의 설치와 환경 구성, GSL을 사용한 프로그램을 어떻게 컴파일 하는지 그리고 규약들에 대해 기술합니다.

2.1 예제 프로그램

다음 프로그램 코드는 베셀 함수 $J_0(x)$ 일 때의 값을 구하는 프로그램 입니다.

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
  double x = 5.0;
  double y = gsl_sf_bessel_J0 (x);
  printf ("J0(%g) = %.18e\n", x, y);
  return 0;
}
```

결과는 다음과 같습니다. 이는 배정밀도의 정확도를 가집니다[1] .

```
J0(5) = -1.77596771314338264e-01
```

[1] 끝의 자리값들은 컴파일러와 환경에 따라 다양하게 나올 수 있습니다.

2.2 컴파일과 링킹

이 라이브러리의 헤더 파일들은 gsl 디렉토리 내에 존재합니다. 라이브러리를 사용하려면 전처리기가 gsl/ 디렉토리를 포함하도록 다음과 같이 작성해야합니다.

```
#include <gsl/gsl_math.h>
```

만약 표준 경로에 디렉토리가 있지 않다면 전처리기에 옵션으로 이 위치를 넘겨주어야 합니다. gsl 의 기본 디렉토리는 /usr/local/include/gsl 입니다. 전 단락의 프로그램 코드파일을 example.c 로 저장했다면 gcc 에서 다음과 같이 쓸 수 있습니다.

```
$gcc -Wall -I/usr/local/include -c example.c
```

결과물은 example.o 파일로 나올 것입니다. gcc 의 include 경로는 기본적으로 /usr/local/include 이므로 위 명령어의 -I 옵션은 없이 쓰는 것과 차이가 없습니다. 다른 경로에 설치된 라이브러리를 사용할 경우 -I 옵션으로 추가해주면 됩니다.

2.2.1 라이브러리 링킹

라이브러리는 libgsl.a 라는 한개의 파일로 설치되어있습니다. 공유 버전의 경우 libgsl.so 로 같이 설치되었있습니다. 이 파일들의 기본 위치는 /usr/local/lib 입니다. 이 디렉토리가 링커의 표준 검색 경로에 포함되어 있지 않다면, 커멘드 라인 명령어로 설정해 주어야 합니다.

```
$gcc -L/usr/local/lib example.o -lgsl -lgslcblas -lm
```

gcc 의 기본 검색 경로는 /usr/local/lib 입니다. 따라서 GSL이 기본 경로에 설치되어 있다면, -L 옵션은 무시 가능합니다.

-lm 옵션은 시스템의 수학 라이브러리를 링크합니다. 다른 시스템에서는 필요 없을 수도 있습니다[2] .

GNU C Complier와 관련 프로그램의 튜토리얼을 보고 싶다면 An Introduction to GCC (ISBN:0954161793)를 참고 하시기를 바랍니다.

참고: 해당 GCC 문서는 2004년도에 작성된 문서입니다. 큰 틀은 비슷하지만 GCC 또한 18여년 동안 많은 발전이 있어왔습니다. 최신 GCC 사용 설명서와 문서는 **GCC 온라인 문서** 의 문서들을 참고할 수 있습니다(*).

<https://gcc.gnu.org/online/docs/>

---

[2] 예를 들어 Mac OS system에서는 필요 없습니다.

2.2.2 대체 BLAS 라이브러리 링킹

다음의 명령어는 다른 BLAS라이브러리(libcblas.a)를 어떻게 프로그램과 링크하는 지 보여줍니다.

```
$gcc example.o -lgsl -lcblas -lm
```

최고의 효율을 위해서는 -lcblas 를 통해 최적화된 특정 플랫폼을 위한 CBLAS 라이브러리를 사용해야 합니다. 이때, 해당 라이브러리는 반드시 CBLAS 표준을 준수해야 합니다. ATLAS 패키지는 고효율의 BLAS 라이브러리를 CBLAS 인터페이스를 통해 제공합니다. 이 패키지는 자유 소프트웨어이고 빠른 벡터와 매트릭스 연산이 필요할 때, 설치되어있어야 합니다. 다음 명령줄은 ATLAS 라이브러리와 CBLAS 인터페이스를 링크합니다.

```
$gcc example.o -lgsl -lcblas -latlas -lm
```

만약 ATLAS 라이브러리가 비표준 경로에 설치되어있다면, 전 단계들에서 보였다시피 -L 옵션으로 검색 경로에 추가시켜주어야 합니다.

BLAS에 대한 더 자세한 정보를 알고 싶다면, BLAS 지원 을 참고할 수 있습니다.

2.3 공유 라이브러리

프로그램이 공유 라이브러리를 사용하려면, 운영체제가 대응되는 .so 파일을 구동 과정에서 제공해야 합니다. 만약 해당 라이브러리를 찾을 수 없다면 다음의 오류 메세지가 나옵니다.

```
$./a.out
./a.out: error while loadinng shared libraries:
libgsl.so.0: canot open shared object file: No such file or directory
```

이러한 오류를 피하기 위해서 시스템의 동적 링커의 설정[3] 을 바꾸거나 쉘 변수 LD_LIBRARY_PATH 를 정의해 라이브러리가 설치된 디렉토리를 포함 시키게 할 수 있습니다. (둘 다 동시에 할 수도 있습니다.)

예를 들어서, Bourne shell(/bin/sh 이나 /bin/bash)의 경우, 라이브러리 검색 경로는 다음과 같은 명령어로 설정할 수 있습니다.

```
$LD_LIBRARY_PATH=/usr/local/lib
$export LD_LIBRARY_PATH
$./example
```

C-shell(/bin/csh 이나 /bin/tcsh)의 경우 동일한 기능을 하는 다음 명령어를 쓸 수 있습니다.

[3] GNU/Linux 시스템의 /etc/ld.so.conf

```
% setenv LD_LIBRARY_PATH /usr/local/lib
```

C-shell의 표준 프롬프트 기호는 % 입니다. 이 기호는 명령어를 입력할 때, 제외하고 쳐야합니다.

각 세션에서 이러한 명령을 재입력하기 위해서, 해당 명령어들은 시스템 전체나 각각의 계정 로그인 파일에 저장할 수 있습니다.

프로그램의 정적 링크 버전을 원한다면 gcc 에서 -static 플래그를 사용하면 됩니다.

```
$gcc -static example.o -lgsl -lgslcblas -lm
```

공유, 정적, 동적 라이브러리에 관한 정보는 Program Library HOWTO 를 추천합니다(*).

2.4 ANSI C 규격

본 라이브러리는 **ANSI C** 로 작성되었고, ANSI C 표준으로(C89)로 쓰여지는 것을 의도하고 있습니다. ANSI C 컴파일러를 지원하는 모든 시스템에서 사용가능합니다.

본 라이브러리는 사용자에게 보이는 어떠한 비 ANSI C 확장기능에도 의존하지 않습니다. GSL을 사용하는 프로그램은 ANSI 표준을 준수해야 합니다. 하지만, 순수 ANSI C와 호환되는 확장 기능은 조건부 컴파일을 이용해서 지원할 수 있습니다. 때문에, 이러한 조건부 컴파일 기능을 지원하는 시스템에서 GSL 라이브러리는 컴파일러 확장 기능과 함께 활용할 수 있습니다.

특정 시스템에서 ANSI C의 기능이 손상되었다면, 라이브러리는 컴파일 과정에서 관련 기능들을 제외하고 컴파일합니다. 이런 경우 해당 기능을 사용하는 프로그램의 링크가 불가능하고 의도치 않은 결과를 얻을 수 있습니다.

네임스페이스 충돌을 방지하기 위해서 모든 함수와 변수들은 앞에 접두사로 gsl_ 이 붙게 됩니다. 매크로의 경우 GSL_ 접두사가 붙습니다.

2.5 inline 함수

inline 기능[4] 은 ANSI C 표준(C89)에서 지원하는 기능은 아니라 라이브러리에서 inline 함수를 기본적으로 지원하지 않습니다. inline 함수는 C99 표준에서 공식적으로 지원하기 시작했습니다. 하지만 대다수의 C89 표준 컴파일러에서도 오랫동안 inline 기능을 확장기능으로 제공해왔습니다.

[4] 기본적으로 정의 된 함수를 사용하기 위해 코드 내에서 함수를 부르면, 플랫폼별, 언어별 호출 규약(Calling consvention)에 의해 정해진 절차에 따라 함수를 부르게 됩니다. 이러한 과정으로 인해 특정한 기능을 함수로 사용하는 경우 단순히 해당 코드를 안에 넣는 것보다 호출 과정이 추가되어 실행 시간이 늘어나는 제약이 있습니다. 해당 이유로 인해 재귀 함수 기능은 일반적으로 실용적인 프로그래밍 과정에서 권장되지 않습니다. 인라인 기능은 이를 개선할 수 있는 방법 중 하나로, 매크로와 비슷하게 인라인으로 정의된 함수의 내부 코드를 해당 함수가 호출된 부분에 그대로 넣어 컴파일을 해 호출 과정에서의 간극을 개선할 수 있습니다(*).

inline 기능의 사용을 위해, 라이브러리의 외부 헤더 파일에서는 조건부 컴파일 기능을 이용해, 성능 개선이 가능한 몇몇 기능들에 대해 inline 버전을 제공합니다. 이러한 함수들의 inline 버전은 응용 프로그램을 컴파일 할 때, 매크로 HAVE_INLINE 을 정의해 포함시킬 수 있습니다.

```
$gcc -Wall -c DHAVE_INLINE example.c
```

만약 autoconf 라는 매크로를 사용한다면, 자동으로 정의됩니다. HAVE_INLINE 매크로를 정의하지 않는다면, inline 함수가 아닌 일반 함수가 대신 사용됩니다.

기본적으로 extern inline[5] 가 inline 함수를 정의하기 위한 키워드(keyword)로 사용됩니다. 이는 gcc 에서 불명확한 함수 정의를 막기위한 확장기능입니다. 만약 다른 컴파일러에서 extern inline 이 문제가 생긴다면, autoconf 검사를 사용해볼 수 있습니다. Autoconf 매크로

gcc 를 C99로 컴파일한다면(gcc -std=c99`) 헤더파일들은 자동으로 extern inline 에서 C99 호환 inline 함수 정의들로 바뀝니다. 다른 C99 컴파일러를 사용한다면, GSL_C99_INLINE 매크로를 넣어볼 수 있습니다.

2.6 Long double

일반적으로, 이 라이브러리에서 사용된 알고리즘들은 배 정밀도를 기반으로 쓰였습니다. long double 데이터형은 실제 계산에서 지원되지 않습니다.

이러한 선택의 이유는 long double 의 정밀도가 기기에 의존하기 때문입니다. IEEE 표준은 각각의 기기들에서 확장된 숫자형들이 가져야 하는 최소 정밀도만을 정해두었습니다. 반면, 배정밀도 double 의 정밀도는 기기에 관계 없이 모두 동일한 정밀도를 가집니다.

그러나, 실제 계산을 할때는, long double 형의 데이터를 사용해야 할 때도 있습니다. vector 와 matrix 데이터형은 long double 을 지원하는 데이터형을 포함하고 있습니다.

한가지 알아두어야 할 점은 어떤 시스템의 표준 라이브러리 stdio.h 에 정의된 printf 와 scanf 같은 입출력 함수들은 long double 형을 정확히 포함하지 않는 경우도 있습니다. 라이브러리의 configure 단계에서 이러한 기능을 확인하고 필요한 경우 이에 의존하는 특정 GSL 명령어를 제거해서, 정의되지 않거나 잘못된 결과가 나오는 경우를 피할 수 있습니다. long double 을 지원하지 않을 경우 configure 단계에서 출력 결과는 다음과 같습니다.:

```
checking whether printf works with long double... no
```

long double 데이터 형의 입/출력이 사용하고자 하는 시스템에서 지원하지 않는다면, 이에 의존하는 GSL 함수들은 결과적으로, 프로그램에 link할 수 없습니다.

만약, long double 을 지원하지 않는 시스템에서 작업해야 한다면, 이진 형태(binary format)을 사용하거나 long double 을 double 로 변환해 읽고 쓰는 방법 등이 있습니다.

[5] extern inline 은 C89, ANSI C에서 확장으로 지원하는 인라인 함수 선언 방법입니다. C99에서는 간단히 inline 을 사용해 인라인 함수를 선언할 수 있습니다.

2.7 함수의 이식성

이식 가능한 프로그램의 작성을 지원하기 위해, GSL에서는 다른 라이브러리에 작성된 함수들을 이식해 제공하기도 합니다. 예를 들어 BSD 수학 라이브러리가 있습니다. 프로그램을 작성할 때, 원래 라이브러리에 있는 함수들을 사용하거나 아니면, GSL의 이식 버전을 사용해 볼 수 있습니다. 이 과정은 전처리기에서 매크로로 관리 가능하며, 원래 라이브러리가 존재하지 않는 다른 기기에서 사용할 때 유용합니다.

예를 들어서, 사용하는 기기에 BSD 라이브러리의 함수 hypot() 가 있다면, 다음의 메크로를 config.h 와 응용 프로그램에 정의할 수 있습니다.

```
/* Substitute gsl_hypot for missing system hypot */

#ifndef HAVE_HYPOT
#define hypot gsl_hypot
#endif
```

응용 프로그램의 소스 파일들에 include 명령어; #include <config.h> 를 사용해 hypot() 이 존재하지 않을 때, 소스 파일 내의 hypot() 을 gsl_hypot() 으로 교체할 수 있습니다. 이러한 교체는 autoconf 를 사용해서 자동으로 이루어지도록 할 수도 있습니다.

Autoconf 매크로 를 참고할 수 있습니다.

대부분의 경우에, 가장 좋은 방법은 본래 함수들이 존재한다는 가정하에, 그 함수들을 사용하고 존재하지 않는다면, 대신에 GSL 함수를 사용하는 것입니다. 이를 이용하면 시스템 별로 최적화된 라이브러리를 사용할 수 있습니다. 이설계 방법은 GSL 스스로도 사용하고 있습니다.

2.8 대체 최적화

라이브러리에 있는 대부분의 함수들은 모든 아키텍쳐들에 대해 최적화 되어있지 않습니다. 예를 들어서, 가우스 난수를 계산하는 방법이 여러가지가 있는데, 이들의 상대적 속도는 구동 기기의 종류에 따라 달라집니다. 이 경우에 라이브러리에서는 본래 함수랑 똑같은 인터페이스[6] 로 이식 함수를 구현해 제공합니다. 만약, 프로그램을 작성할 때, 표준 함수의 라이브러리 구현체를 사용했다면, 전처리기에서 대체 함수를 선택할 수 있습니다. 이러한 방법은 사용자가 최적화한 함수를 사용할 때도 이식성을 유지하기에 좋은 방법입니다. 다음 줄들은 가우스 분포)에서 표본을 뽑아오는 방법을 플래폼 의존 방식으로 구현한 것입니다.

```
#ifdef SPARC
#define gsl_ran_gaussian gsl_ran_gaussian_ratio_method
#endif
#ifdef INTEL
```

(다음 페이지에 계속)

[6] 같은 인터페이스라는 뜻은, 예를 들어서 본래 함수가 double f_get(int i, double, j) 형태로 되어있다면, 이러한 함수의 GSL이식 버전도 똑같은, 인자와 반환값으로 설계되었다는 뜻입니다. double gsl_f_get(int i, double j) 형태로 정의됩니다.

(이전 페이지에서 계속)

```
#define gsl_ran_gaussian my_gaussian
#endif
```

이러한 줄들은 응용 프로그램의 구성 헤더 파일 config.h 에 작성되어, 모든 소스파일에서 이 헤더파일을 포함해야 합니다. 주의할 점은 대체한 이식함수들은 비트 단위로 똑같은 결과를 내지는 않으며, 난수 분포의 경우 완전히 다른 난수들을 생성한다는 것에 유의해야합니다.

2.9 다양한 수치 자료형 지원

라이브러리에 정의된 많은 함수들은 다양한 자료형을 지원합니다. 한 함수의 지료형 구현체는 자료형을 이름으로 가지는 접사와 함수 이름이 붙은 형태로 구현되어 있습니다. 이러한 자료형의 이름은 C++ 원시 템플릿에 정의된 자료형을 기반으로 합니다. 구체적으로 해당 접사는 모듈의 이름으로 된 접두사와 함수의 이름 사이에 넣어집니다. 다음 표는 가상의 모듈 gsl_foo 자료형으로 정의된 fn() 의 모든 수치형 정의를 보여줍니다.

```
gsl_foo_fn              double
gsl_foo_long_double_fn  long double
gsl_foo_float_fn        float
gsl_foo_long_fn         long
gsl_foo_ulong_fn        unsigned long
gsl_foo_int_fn          int
gsl_foo_uint_fn         unsigned int
gsl_foo_short_fn        short
gsl_foo_ushort_fn       unsigned short
gsl_foo_char_fn         char
gsl_foo_uchar_fn        unsigned char
```

일반적으로 배정밀도 double 의 수치형이 기본으로 사용됩니다. 이 경우에는 접사가 필요 없습니다. 예를 들어서 함수 gsl_stats_mean() 는 double 자료형들의 평균값을 구해줍니다. 하지만, gsl_stats_int_mean() 의 경우 정수들의 평균값을 구해줍니다.

라이브러리에서 정의하는 여러 자료형들도 똑같은 규약을 사용합니다. 예를 들어 gsl_vector 나 gsl_matrix 가 있습니다. 이 경우 자료형의 이름 뒤에 붙는 형태로 구성됩니다. 예를 들어서 어느 모듈이 gsl_foo 라는 자료형을 정의하는 경우, 다음과 같은 방법으로 확장할 수 있습니다.

```
gsl_foo                 double
gsl_foo_long_double     long double
gsl_foo_float           float
gsl_foo_long            long
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
gsl_foo_ulong           unsigned long
gsl_foo_int             int
gsl_foo_uint            unsigned int
gsl_foo_short           short
gsl_foo_ushort          unsigned short
gsl_foo_char            char
gsl_foo_uchar           unsigned char
```

라이브러리에서 제공하는 모듈이 자료형에 의존해 정의되어 있다면, 이 라이브러리에서는 각각의 자료형을 위한 헤더 파일을 독립적으로 제공할 것입니다. 이러한 파일 이름들은 아래와 같이 작성되어 있습니다. 편의를 위해서 기본 헤더파일은 모든 자료형에 대한 정의를 담고 있습니다. 배정밀도로 정의된 함수만을 가져오거나 다른 특정한 자료형으로 정의된 함수만을 가져오고 싶다면 다음의 독립된 헤더 파일들을 포함시키면 됩니다.

```
#include <gsl/gsl_foo.h>              All types
#include <gsl/gsl_foo_double.h>       double
#include <gsl/gsl_foo_long_double.h>  long double
#include <gsl/gsl_foo_float.h>        float
#include <gsl/gsl_foo_long.h>         long
#include <gsl/gsl_foo_ulong.h>        unsigned long
#include <gsl/gsl_foo_int.h>          int
#include <gsl/gsl_foo_uint.h>         unsigned int
#include <gsl/gsl_foo_short.h>        short
#include <gsl/gsl_foo_ushort.h>       unsigned short
#include <gsl/gsl_foo_char.h>         char
#include <gsl/gsl_foo_uchar.h>        unsigned char
```

2.10 C++과의 호환성

이 라이브러리의 헤더 파일들은 직접 C++ 프로그램에 사용할 수 있도록, 함수들을 `extern "C"` 형태로 정의합니다. 이 방식은 라이브러리 내의 함수들을 C++에서 바로 불러올 수 있게 해줍니다.

라이브러리에 사용자 정의함수를 인자로 넘기는 경우에 C++ 예외 처리를 사용하고자 한다면, 라이브러리가 추가적인 CFLAFS 설정인 `-fexceptions` 로 빌드 되어야 합니다.

2.11 배열 별칭

이 라이브러리에서 배열, 벡터, 행렬들이 수정 가능한 인자로 전달 되었을 때, 각각의 자료형들이 별칭된 관계가 아니며, 겹치지도 않는다고 가정합니다. 이러한 방법은 라이브러리에서 중접 메모리 구역을 관리하지 않아도 되게 하고 추가적인 최적화 방법을 사용할 수 있게 해줍니다. 만약 중첩된 메모리 구역이 수정 가능한 인자로 전달 된다면, 함수의 결과가 정의되지 않습니다. 만약 인자가 수정되지 않게 할 경우, (예를 들어서 함수 원형에서 `const` 인자로 정의하는 경우가 있습니다) 중첩되거나 할당된 메모리 구역은 안전하게 사용할 수 있습니다.

2.12 스레드 안전성

이 라이브러리는 다중 스레드 프로그램에 사용할 수 있습니다. 모든 함수는 스레드 안전합니다. 이 말은 모든 함수가 정적 변수를 사용하지 않는다는 뜻입니다. 메모리는 항상 함수가 아니라 객체들에 연결되어 있습니다. 임시 공간에 있는 작업 공간 객체를 사용하는 함수의 경우, 작업 공간 객체는 각각의 스레드 기저에 할당되어야 합니다. 읽기 전용 메모리에 있는 표 객체를 사용하는 경우 여러 스레드에서 동시에 사용될 수 있습니다. 표 객체는 함수 원형에서 항상 상수로 정의되어야 합니다. 이는 다른 스레드에 의해 안전하게 접근할 수 있음을 나타냅니다.

라이브러리 안에 몇몇 정적 변수들이 존재합니다. 이 변수들은 라이브러리 전체의 행동을 제어하기 위해 사용됩니다. (예를 들어, 범위를 확인하고 함수가 치명적인 오류를 반환할 때 등이 있습니다.) 이 변수들은 사용자에 의해 직접적으로 설정됩니다. 따라서 프로그램이 시작될 때, 한번 초기화 되어야 하며, 다른 스레드들에 의해 수정하지 않도록 해야합니다.

2.13 제거 예정 함수

라이브러리의 개발 과정에서 라이브러리 내부의 함수들을 수정하거나 제거할 수도 있습니다. 이러한 상황에 있는 함수들은 처음에 deprecated 로 선언되고 다음 버전의 라이브러리에서 제거됩니다. 프로그래밍 과정에서 현재 베포판에서 제거 예정인 함수들 비활성화 할 수도 있습니다. 전처리기에서 `GSL_DISALBE_DEPRECATED` 를 선언해 주면 됩니다. 이는 다음 버전의 라이브러리와의 호환성 검사에 사용될 수 있습니다.

2.14 코드 재사용

라이브러리에 작성된 기능등은 가능한 한 다른 모듈이나 파일들에 의존하지 않도록 짜여져 있습니다. 이는 라이브러리 전체를 설치할 필요 없이 독립된 함수들을 추출해서 다른 응용 프로그램에 사용할 수 있게 합니다. `GSL_ERROR` 와 같은 매크로를 선언하고 `#include` 선언을 제거해 파일을 독립적으로 실행할 수 있게 컴파일할 수 있습니다. 이러한 방법의 코드 재사용은 GNU 일반 공중 사용 허가서의 규약에서 권장하고 있습니다.

제 3 장

오류 관리

이 단원에서는 GSL 함수들이 오류를 보고하는 방법과 이러한 오류들을 다루는 방법에 대해 서술합니다. 오류 관리 기능들은 함수들이 반환하는 상태 값을 보고, 해당 함수의 실행이 정상적으로 이루어졌는 지 판별할 수 있습니다. 오류가 발생했을 시, 실패한 원인이 무엇인지에 대한 판별도 제공할 수 있습니다. 또, 사용자 정의 오류 관리 함수를 추가하여 라이브러리의 기본 행동들에 변화를 줄 수도 있습니다.

이 단원에서 다루는 함수들은 gsl_errno.h 에 기술되어 있습니다.

3.1 오류 보고

이 라이브러리는 POSIX 스레드 라이브러리에 기술된 스레드-안전 오류 보고 규약을 따릅니다. 함수들은 오류 발생시 각각의 오류를 나타내는 0이 아닌 오류값을 반환하고, 성공적으로 실행되었을 시 0 값을 반환합니다.

```
int status = gsl_function (...)

 if (status) { /* an error occurred */
   .....
   /* status value specifies the type of error */
 }
```

라이브러리의 명령어 집합들은 필요한 작업을 더 이상 수행할 수 없을 때 오류를 보고합니다. 예를 들어서, 근을 찾는 함수는 요구되는 정확도 이상으로 근이 수렴하지 않을 때 0이 아닌 오류 값을 반환합니다. 이러한 상황은 수학 라이브러리를 사용할 때 빈번히 발생합니다. 따라서 호출한 함수들의 반환 값을 반드시 확인해야 합니다.

어떤 상황이든지 명령어가 오류를 보고한 상황의 반환 값은 오류의 종류를 나타냅니다. 반환 값은 표준 C 라이브러리의 errno 값과 유사합니다. 호출자는 반환 값을 보고 어떤 행동을 취할지 선택할 수 있습니다.

만약 중요한 오류가 아니라면 무시할 수도 있습니다.

코드가 반환하는 오류 보고 외에도 라이브러리에는 오류 관리 함수인 gsl_error() 가 있습니다. 이 함수는 다른 라이브러리의 다른 함수들에서 오류를 보고할 때 호출자에게 값을 반환하기 직전에 호출됩니다. 이 오류 관리 함수의 기본 행동은 메세지를 화면에 출력하고 프로그램을 정지시킵니다.

```
gsl: file.67: ERROR: invalid argument supplied by user
Default GSL error handler invoked.
Aborted
```

gsl_error() 관리 함수의 제공 목적은 디버거 아래에서 실행 되는 상황에서, 라이브러리의 오류를 잡을 수 있는 중단점을 설정하는 함수를 제공함에 있습니다. 이는 반환 값을 보고 오류를 잡는 상업용 프로그램에서는 사용해서는 안됩니다.

3.2 오류 값

라이브러리 함수들이 반환하는 오류 값 번호들은 gsl_errno.h 에 정의되어 있습니다. 모두 접두사 GSL 이 붙어있고 0이 아닌 정수 형태의 상수값으로 정의되어 있습니다. 1024 이상의 오류 값은 응용 프로그램용으로 예약되었으며 라이브러리에서 사용되지 않습니다. 대부분의 오류 값은 C 라이브러리에서 해당 오류 값와 동일한 기본 이름을 사용합니다. 다음은 가장 일반적인 오류 값들입니다.

int **GSL_EDOM**

도메인 오류: 수학 함수에서 인수 값이 함수가 정의된 정의역에 포함되지 않는 경우 사용됩니다. 표준 C 라이브러리의 EDOM 와 같은 상황에서 사용됩니다.

int **GSL_ERANGE**

범위 오류: 수학 함수에서 오버플로우나 언더플로우로 인해 결과 값을 나타낼 수 없는 경우 사용됩니다. 표준 C 라이브러리의 ERANGE 와 같은 상황에서 사용됩니다.

int **GSL_ENOMEM**

사용 가능한 메모리가 없음: 용량이 꽉 찼기 때문에 시스템이 가상 메모리를 더 할당할 수 없음을 나타냅니다. 표준 C 라이브러리의 ENOMEM 와 같은 상황에서 사용됩니다. 이 오류는 GSL 구현체에서 malloc() 계열의 함수로 메모리 할당이 불가능할 때 보고됩니다.

int **GSL_EINVAL**

잘못된 인수: 이는 잘못된 인수를 라이브러리 내 함수에 전달한 대다수의 문제를 나타내기 위해 사용됩니다. 표준 C 라이브러리의 EINVAL 와 유사합니다.

각각의 오류 값들은 함수 gsl_strerror() 를 사용해 오류 메세지로 바꿀 수 있습니다.

```
const char * gsl_strerror(const int gsl_errno)
```

이 함수는 각각의 오류 값 gsl_errorno 에 대한 설명을 문자열로 반환합니다. 예를 들어서:

```
printf ("error: %s\n", gsl_strerror (status));
```

이 코드는 `error: output range error` 와 같은 오류 메세지를 반환할 것입니다. 이는 오류 값 `GSL_ERANGE` 를 기술하는 메세지입니다.

3.3 오류 관리자

GSL 오류 관리자는 기본적으로 짧은 메세지를 출력하고, `abort()` 를 부릅니다. 이 기본 설정은 라이브러리 함수들이 오류를 보고할 때 코어 덤프(core-dump)를 일으키며 프로그램을 정지시킵니다. 페일-세이프(fail-safe) 방식을 기본 설정으로 하도록 의도되었기 때문에, 라이브러리 내장 기능들의 반환값을 확인하지 않습니다. 이러한 방식으로 프로그램을 짜는 것은 권장하지 않습니다.

만약 기본 오류 관리자를 종료시키면, 프로그래머는 직접 각 함수들의 반환 값을 확인하고 다룰 책임이 생깁니다. 오류 발생시 행동할 절차를 새로운 오류 관리자를 제공해 수정할 수도 있습니다. 예를 들어서 오류 관리자가 파일에 모든 오류 기록을 저장하도록 하거나, 특정 오류 조건(언더플로우 같은)을 무시, 아니면 디버거를 실행시켜 현재 프로세스에 연결할 수도 있습니다.

모든 GSL 오류 관리자들은 `gsl_error_handler_t` 자료형으로 정의됩니다. 이는 `gsl_errno.h` 에 정의되어 있습니다.

type **gsl_error_handler_t**

> GSL 오류 관리자 함수의 자료형입니다. 오류 관리자는 4개의 인자를 넘겨 받습니다. 이 인자들은 각각 오류의 이유(문자열), 오류가 발생한 소스 파일의 이름(문자열), 그 파일에서 오류가 발생한 줄 숫자(정수), 그리고 오류 값(정수)을 나타냅니다. 소스 파일과 줄 숫자는 컴파일 시간에 전처리기에 의해 `__FILE__` 그리고 `__LINE__` 를 이용해 결정됩니다. 오류 관리자는 `void` 형태로 값을 반환합니다.

사용자 정의 오류 관리자는 다음과 같이 정의되어야 합니다.

void **handler**(const char *reason, const char *file, int line, int gsl_errno)

> 사용자 정의 오류 관리자를 사용하기 위해서는 `gsl_set_error_handler()` 함수를 호출해야합니다. 이 함수도 `gsl_errno.h` 에 정의되어 있습니다.

gsl_error_handler_t ***gsl_set_error_handler**(gsl_error_handler_t *new_handler)

> 이 함수는 GSL 명령어 집합을 위한 `new_handler` 는 새로운 오류 관리자를 설정합니다. 이전 관리자는 반환됩니다(나중에 복구할 수 있습니다.) 사용자 정의 오류 관리자 함수가 전역 변수로 저장된다는 사실을 유의해야합니다. 이로 인해 한 개의 프로그램에 1개의 오류 관리자만이 사용될 수 있습니다. 이 함수는 다중 스레드 프로그램에서 사용될 수 없습니다. 주 스레드에서 프로그램 전체 오류 관리를 하도록 하는 예외 상황에서는 사용할 수 있습니다. 다음 예제는 어떻게 새로운 오류 관리자를 설정하고 복구하는지 예시를 보여줍니다.

```
/* save original handler, install new handler */
old_handler = gsl_set_error_handler (&my_handler);
/* code uses new handler */
.....
/* restore original handler */
gsl_set_error_handler (old_handler);
```

기본 오류 관리자(오류 발생시 abort() 를 호출)를 사용하려면 오류 관리자에 NULL 을 넣어주면 됩니다.

```
old_handler = gsl_set_error_handler (NULL);
```

gsl_error_handler_t *__gsl_set_error_handler_off__()

오류 관리자 기능을 아무것도 하지 않도록 설정해 꺼버립니다. 어떤 오류가 발생해도 프로그램이 계속 작동하도록 하기 때문에, 라이브러리 함수들의 반환 값을 반드시 확인해야 합니다. 실제 상용 프로그램 단계에서 설정하는 것을 권장합니다. 이전 관리자가 반환되기 때문에 나중에 복구할 수도 있습니다.

gsl_errno.h 에 정의된 GSL_ERROR 매크로를 사용자가 정의해서 라이브러리를 재컴파일하면, 특정 응용 프로그램에서 오류의 행동을 수정할 수 있습니다.

3.4 사용자 정의 함수에서 GSL 오류 보고 사용하기

GSL 코드를 이용해 수치 계산 함수를 프로그램 안에 작성했다면 라이브러리와 같은 오류 보고 규약를 사용하는 것이 효율적입니다.

오류를 보고하기 위해서는 함수에서 gsl_error() 를 호출해서 오류를 설명하는 문자열과 gsl_errno.h 에 기술된 적절한 오류 값, 아니면 특정한 값(예: NaN)을 넘겨주어야 합니다. gsl_errno.h 에서는 이러한 과정을 효율적으로 처리해 줄 수 있는 두 개의 매크로를 제공합니다.

__GSL_ERROR__(reason, gsl_errno)

GSL 규약에 따라 오류를 보고하고 gsl_errno 상태 값을 반환합니다. 이는 다음과 같은 함수로 확장해 볼 수 있습니다.

```
gsl_error (reason, __FILE__, __LINE__, gsl_errno);
return gsl_errno;
```

gsl_errno.h 에 정의 되어있고, do {...} while(0) 로 감싸져 있습니다. 이는 구문 분석 문제를 방지하기 위함입니다.

다음은 함수가 요구하는 정밀도를 tolerance 까지 만족시키지 못했을 때, 매크로를 사용해 오류를 보고하는 예제를 나타냅니다. 오류를 보고를 위해 함수에서 오류 값 GSL_ETOL 를 반환해야 합니다.

```
    if (residual > tolerance){
    GSL_ERROR("residual exceeds tolerance", GSL_ETOL);
}
```

GSL_ERROR_VAL(reason, gsl_errno, value)

GSL_ERROR 매크로와 똑같습니다. 하지만 사용자 정의 값 value 를 오류 값 대신에 반환합니다. 이는 부동 소수점 값을 반환하는 수치 함수에서 쓰일 수 있습니다.

다음 예제는 GSL_ERROR_VAL 사용해 수학적 특이점에서 NaN 값을 반환하는 것을 보여줍니다.

```
    if (x == 0){
GSL_ERROR_VAL("argument lies on singularity", GSL_ERANGE, GSL_NAN);
}
```

3.5 예제

다음은 오류가 보고될 수 있는 함수의 반환 값을 확인하는 예제 코드입니다.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>
...
int status;
size_t n = 37;
gsl_set_error_handler_off();
status = gsl_fft_complex_radix2_forward (data, stride, n);

if (status) {
     if (status == GSL_EINVAL) {
          fprintf (stderr, "invalid argument, n=%d\n", n);
     } else {
          fprintf (stderr, "failed, gsl_errno=%d\n", status);
     }
     exit (-1);
}
...
```

함수 gsl_fft_complex_radix2_forward() 는 2의 거듭 제곱인 정수형 길이만을 인자로 받습니다. 만약 변수 n 가 2의 거듭 제곱이 아니라면, 함수는 GSL_EINVAL 값을 반환해 길이 인자가 부적절 하다고 알릴 것입니다. 함수 gsl_set_error_handler_off() 를 호출해 기본 오류 관리자를 멈추어 프로그램이 정지하는 것을 막습니다. else 구문은 다른 오류들을 감지합니다.

제 4 장

수학 함수

이 단원에서는 기초적인 수학 함수들을 다룹니다. 여기서 다루는 함수들의 일부는 이미 시스템 라이브러리에서 제공합니다[1] . 그러나 해당 시스템 함수들을 쓸 수 없는 경우를 대비해, `gsl` 에서 대체 함수들을 포함해 제공합니다.

본 단원에서 제공하는 함수와 매크로들은 `gsl_math.h` 헤더 파일에 포함되어 있습니다.

4.1 수학 상수

이 라이브러리에서는 표준 BSD 라이브러리의 수학 상수들을 제공합니다. 해당 자료를 기반으로 다음과 같은 상수들을 포함합니다.

[1] C에서 기본으로 제공하는 표준 헤더파일 중 math.h 를 말합니다(*).

M_E	지수함수의 밑, e
M_LOG2E	e 의 밑이 2인 로그 값, $\log_2(e)$
M_LOG10E	e 의 밑이 10인 로그 값, $\log_{10}(e)$
M_SQRT2	2의 제곱근, $\sqrt{2}$
M_SQRT1_2	1/2의 제곱근, $\sqrt{1/2}$
M_SQRT3	3의 제곱근, $\sqrt{3}$
M_PI	원주율 상수, π
M_PI_2	원주율의 절반 값, $\pi/2$
M_PI_4	원주율의 1/4 값, $\pi/4$
M_SQRTPI	원주율의 제곱근, $\sqrt{\pi}$
M_2_SQRTPI	2를 원주율의 제곱근으로 나눈 값, $2/\sqrt{\pi}$
M_1_PI	원주율의 역수, $1/\pi$
M_2_PI	원주율의 역수의 2배, $2/\pi$
M_LN10	10의 자연로그 값, $\ln(10)$
M_LN2	2의 자연로그 값, $\ln(2)$
M_LNPI	파이의 자연로그 값, $\ln(\pi)$
M_EULER	오일러 상수, γ

4.2 무한과 비정상 값(Not-A-Number)

GSL_POSINF

IEEE 표기 형식의 양의 무한대($+\infty$)를 나타냅니다. 이 값은 $+1.0/0.0$ 으로 표현될 수 있습니다.

GSL_NEGINF

IEEE 표기 형식의 음의 무한대($-\infty$)를 나타냅니다. 이 값은 $-1.0/0.0$ 으로 표현될 수 있습니다.

GSL_NAN

IEEE 표기 형식의 비정상 값(Not-a-Number; NAN)을 나타냅니다. $0.0/0,0$ 로 표현될 수 있습니다.

int **gsl_isnan**(const double x)

x 이 비정상 값(NaN)인지 아닌지 판단합니다. 비정상 값으로 판단되면 1 을 반환합니다.

int **gsl_isinf**(const double x)

양의 무한이면 $+1$ 음의 무한이면 -1 반환합니다. 나머지 경우에 0을 반환합니다[2] .

[2] C99 표준에서는 isinf() 함수가 무한대의 부호와 관계없이 0이 아닌 값을 반환합니다. GSL 이전 버전의 경우 시스템의 isinf() 함수를 사용했고, 어떤 기기에 따라 동일한 현상이 발생할 수도 있습니다. 따라서, 필요한 경우 $gsl_isinf()$ 반환값의 부호 보다는 x 의 부호를 별도로 판정하는 것이 현명합니다.

int **gsl_finite**(const double x)

x 실수면 1을, 만약 무한대거나 비정상 값이면 0을 반환합니다.

4.3 기초 함수들

다음 명령어 집합들은 BSD 수학 라이브러리의 이식을 기반으로 제공됩니다. 시스템 내장 기능이 없다면 다음의 함수들을 대신 사용할 수 있습니다. 만약, autoconf 를 사용해 프로그램을 컴파일한다면, 자동으로 치환이 일어납니다. 함수의 이식성 을 참고할 수 있습니다.

double **gsl_log1p**(const double x)

$\log(1+x)$ 의 값을 계산합니다. 정확도는 작은 x 값에 대해 보장됩니다. 이는 BSD 수학 함수 log1p(x) 대체 함수입니다.

double **gsl_expm1**(const double x)

$\exp(x)-1$ 의 값을 계산합니다. 정확도는 작은 x 값에 대해 보장됩니다. 이는 BSD 수학 함수 expm1(x) 대체 함수입니다.

double **gsl_hypot**(const double x, const double y)

$\sqrt{x^2+y^2}$ 의 값을 오버 플로우가 일어나지 않도록, 계산합니다. BSD 수학 함수 hypot(x,y) 대체 함수입니다.

double **gsl_hypot3**(const double x, const double y, const double z)

$\sqrt{x^2+y^2+z^2}$ 의 값을 오버 플로우가 일어나지 않도록, 계산합니다.

double **gsl_acosh**(const double x)

$\mathrm{arccosh}(x)$ 의 값을 계산합니다. 표준 수학 라이브러리 acosh(x) 대체함수입니다.

double **gsl_asinh**(const double x)

$\mathrm{arcsinh}(x)$ 의 값을 계산합니다. 표준 수학 라이브러리 asinh(x) 대체함수입니다.

double **gsl_atanh**(const double x)

$\mathrm{arctanh}(x)$ 의 값을 계산합니다. 표준 수학 라이브러리 atanh(x) 대체함수입니다.

double **gsl_ldexp**(double x, int e)

$x \cdot 2^e$ 의 값을 계산합니다. 표준 수학 라이브러리 ldexp(x) 대체함수입니다.

double **gsl_frexp**(double x, int *e)

숫자 x 정규화 분수 f 와 지수 e 로 분리합니다. $x = f \cdot 2^e$ 으로 쓸 수 있고, $0.5 <= f < 1$ 입니다. f 의 값을 반환하고 지수를 e 에 저장합니다. 만약 x 가 0이라면, f, e 모두 0으로 맞추어집니다. 표준 라이브러리 frexp(x,e) 대체함수입니다.

4.4 작은 정수 지수들

표준 C 라이브러리를 향한 많은 불만들 중 하나는 작은 정수 지수 계산이 없다는 점입니다. GSL에서는 해당 함수들을 제공해 이를 보완합니다. 효율성을 위해서 오버플로나 언더플로 조건을 계산하지 않습니다.

double **gsl_pow_int**(double x, int n)

double **gsl_pow_uint**(double x, unsigned int n)

n 값에 대해 x^n 의 값을 계산해줍니다. 이 지수 계산은 효율적으로 설계되었습니다. 예를 들어 x^8 을 계산하고자 하면, $((x^2)^2)^2$ 으로 3번의 계산만으로 구할 수 있습니다. 수치적 오류를 함께 계산하는 함수도 라이브러리 내에서 같이 제공합니다. `gsl_sf_pow_int_e()` 를 사용할 수 있습니다.

double **gsl_pow_2**(const double x)

double **gsl_pow_3**(const double x)

double **gsl_pow_4**(const double x)

double **gsl_pow_5**(const double x)

double **gsl_pow_6**(const double x)

double **gsl_pow_7**(const double x)

double **gsl_pow_8**(const double x)

double **gsl_pow_9**(const double x)

작은 정수 지수 $x^2, x^3, \ldots$ 값들을 효율적으로 계산해줍니다. 만약, `HAVE_INLINE` 가 정의되어 있다면, `inline` 함수로 작동합니다. 따라서 이러한 함수의 사용이 수식을 그대로 사용하는 만큼이나 효율적일 수 있습니다.

```
#include<gsl/gsl_math.h>
double =y = gsl_pow_4(3.141) /* compute 3.141**4 */
```

4.5 숫자의 부호 판별

GSL_SIGN(x)

x 부호를 반환합니다. `((x) >= 0 ? 1: -1)` 로 정의되어 있습니다. 유의할 점은 이 구현에서 0은 양수로 반환됩니다. (IEEE 부호 비트와 관계 없습니다.)

4.6 숫자의 홀짝 판별

GSL_IS_ODD(n)

만약, n 이 홀수면 1을, n 이 짝수면 0을 반환합니다. 인자 n 는 반드시 정수형이어야 합니다.

GSL_IS_EVEN(n)

GSL_IS_ODD 와 정반대로 작동합니다. 만약, n 이 홀수면 0을, n 이 짝수면 1을 반환합니다. 인자 n 는 반드시 정수형이어야 합니다.

4.7 최대, 최소 함수

여기서 서술한 매크로에서는 인수에 대한 여러 가지 평가를 수행하므로 부작용이 있는 인수(예: 난수 생성기에 대한 호출)와 함께 사용하지 않아야 합니다.

GSL_MAX(a, b)

a 와 b 중 최대값을 반환합니다. ((a) > (b) ? (a) : (b)) 로 정의되어 있습니다.

GSL_MIN(a, b)

a 와 b 중 최소값을 반환합니다. ((a) < (b) ? (a) : (b)) 정의되어 있습니다.

extern inline double **GSL_MAX_DBL**(double a, double b)

배정밀도(double) 자료형 변수 a 와 b 에 대해 인라인 함수를 사용해서 큰 값을 반환합니다. 함수를 사용함으로써 추가적인 안전 기능으로 인자의 형식 검사를 사용할 수 있습니다. 인라인 함수를 지원하지 않는 플랫폼에서는 자동으로 GSL_MAX 으로 대체됩니다.

extern inline double **GSL_MIN_DBL**(double a, double b)

배정밀도(double) 자료형 변수 a 와 b 에 대해 인라인 함수를 사용해서 작은 값을 반환합니다. 함수를 사용함으로써 추가적인 안전 기능으로 인자의 형식 검사를 사용할 수 있습니다. 인라인 함수를 지원하지 않는 플랫폼에서는 자동으로 GSL_MIN 으로 대체됩니다.

extern inline int **GSL_MAX_INT**(int a, int b)

extern inline int **GSL_MIN_INT**(int a, int b)

정수(integer) a 와 b 에 대해 인라인 함수를 사용해서 크거나 작은 값을 반환합니다. 인라인 함수를 지원하지 않는 플랫폼에서는 자동으로 GSL_MIN 으로 대체됩니다.

extern inline long double **GSL_MAX_LDBL**(long double a, long double b)

extern inline long double **GSL_MIN_LDBL**(long double a, long double b)

정수(integer) a 와 b 에 대해 인라인 함수를 사용해서 크거나 작은 값을 반환합니다. 인라인 함수를 지원하지 않는 플랫폼에서는 자동으로 GSL_MAX 나 GSL_MIN 으로 대체됩니다.

4.8 부동 소수점 숫자의 근사 비교

두 개의 부동소수점 숫자들을 반올림하거나 오차들을 절단해서 근사적으로 비교하는 건 많은 경우에 유용합니다. 다음 함수는 “D.E. Knuth in Section 4.2.2 of “Seminumerical Algorithms” (3rd edition)”의 부동 소수점 근사 비교 알고리즘을 이식한 것입니다.

int **gsl_fcmp**(double x, double y, double epsilon)

주어진 x 와 y 가 근사적으로 상대 정확도 `epsilon` 만큼 같은지 판별합니다.

상대 정확도는 구간 길이 2δ 로 측정됩니다. $\delta = 2^k\epsilon$ 으로 정의되고, k 는 $frexp()$ 함수에 의해 계산된, x 와 y 의 밑이 2인 최대 지수 값입니다.

만약, x 와 y 의 차가 이 구간 안에 있다면, 이 둘은 근사적으로 같다고 판정하고 0을 반환합니다. 다른 경우에 만약 $x < y$ 면 -1을, $x > y$ 면 1을 반환합니다.

명심할 점은 x 와 y 가 상대 정확도와 비교해서 결정된다는 점입니다. 따라서 주어진 값이 근사적으로 0에 가까운지 판정하는 것에는 부적절합니다.

이 구현체는 `fcmp` 패키지에 기반해 T.C Belding이 구현했습니다.

제 5 장

복소수

이 단원에서는 복소수를 다루기 위한 기능들을 제공합니다. 구현된 알고리즘들은 중간 과정에서 불필요한 언더플로우나 오버 플로우가 발생하지 않도록 검토함과 동시에, 가능한한 넓은 범위의 복소 평면상에서 값을 계산할 수 있도록 작성되었습니다.

다변수 함수는 Abramowitz & Stegun[1] 의 규약을 따라, 분지 절단이 결정됩니다. 이 함수들은 **GNU Calc** 과 같은 표준 값들을 반환합니다. 이들은 "Common Lisp, The Language (Second Edition)"과 HP-28/48 계산기 시리즈에서 구현된 값들과 같습니다.

복소수 자료형은 헤더 파일 `gsl_complex.h` 에 선언되어 있고, 대응되는 복소수 함수들은 `gsl_complex_math.h` 에 정의되어 있습니다.

5.1 복소수의 표현

복소수는 `gsl_complex` 자료형을 통해 표현됩니다. 이 자료형의 내부 표현은 플랫폼에 따라 다양하기 때문에 직접적으로 접근해서는 안됩니다. 아래에 기술된 함수와 매크로들을 이용해 간단하게 복소수들을 제어할 수 있습니다.

`gsl_complex` 기본적인 형태는 다음 구조체로 정의됩니다.

type **gsl_complex**

```
typedef struct
{
        double dat[2];
} gsl_complex;
```

[1] 미국 표준 기술 연구소에서 Milton Abramowitz와 Irene Stegun이 편집한 수학 문헌으로 수식, 그래프 및 수학표를 포함하는 수학 함수 핸드북입니다(*).

실수, 허수부는 각각 붙어 있는 배열의 두 원소에 저장됩니다. 이러한 구현은 실수, 허수부[2] 사이에 간격이 없어서, 이 구조체를 복소 배열에 직접 대응시킬 수 있습니다.

만약, C 컴파일러가 C11 표준을 지원하고 <complex.h> 헤더파일을 <gsl_complex.h> 보다 먼저 포함했다면, gsl_complex C의 복소수 자료 형으로 다음과 같이 정의됩니다.

```
typedef double complex gsl_complex
```

이 방법은 gsl_complex 구조체를 사용자들이 일반 연산으로 취급할 수 있게 해줍니다.

```
gsl_complex x = 2+ 5*I;
gsl_complex y = x_ (3-4*I);
```

중요: C에 내장된 복소수 기능들은 C11 표준 이후에 지원되기 시작했습니다. <complex.h> 가 gsl_complex.h 보다 먼저 포함되었다면, GSL 라이브러리에서는 C11의 새 기능들을 이용해 GSL_REAL 과 GSL_IMAG 를 정의합니다. C99 표준에서 이러한 기능들을 정의할 수 있는 적절한 기능은 없습니다. 따라서 C99 컴파일러에서는 gsl_complex 가 C에 내장된 복소수 기능들로 정의되지 않습니다.

gcc 4.8 계열 컴파일러와 같이 몇몇 컴파일러들은 C11의 일부 기능만을 지원하는 경우도 있습니다. 이러한 컴파일러들에서는 사용자가 C의 내장 복소수 기능들을 활성화 시켜 GSL을 컴파일 할 수 없습니다. 이러한 경우 포함하는 헤더 파일 목록에서 complex.h 를 제거하거나, 컴파일러에 -DGSL_COMPLEX_LEGACY 를 넣어서 구조체에 기반해 정의된 gsl_complex 를 사용하도록 해야합니다.

5.2 복소수 매크로

다음의 C 매크로들은 복소수를 편하게 조작할 수 있게 해줍니다.

GSL_REAL(z)

GSL_IMAG(z)

이 두 매크로는 각각 복소수 z 실수, 허수부의 **메모리 위치** 를 반환합니다. 이 매크로를 이용해서 일반적인 연산을 복소수에 대해, 사용할 수 있습니다.

```
gsl_complex x, y;

GSL_REAL(x) = 4;
GSL_IMAG(x) = 2;
```

(다음 페이지에 계속)

[2] dat[0] dat[1]

(이전 페이지에서 계속)

```
GSL_REAL(y) = GSL_REAL(x);
GSL_IMAG(y) = GSL_REAL(x);
```

다시 말해, 이 매크로를 이용해서 복소수의 실수, 허수부를 읽고, 쓸 수 있습니다.

GSL_SET_COMPLEX(zp, x, y)

데카르트 좌표 성분 (x, y)을 포인터 zp 가 가르키는 복소수의 실수, 허수부에 저장합니다. 예를 들어서,

```
GSL_SET_COMPLEX(&z, 3, 4)
```

는 z 를 $3 + 4i$ 로 초기화 합니다.

5.3 복소수 할당

gsl_complex **gsl_complex_react**(double x, double y)

직교 데카르트 좌표계 (x, y) 를 이용해 $z = x + iy$ 복소수를 반환합니다. HAVE_INLINE 을 정의하면, 인라인 형태의 함수를 사용할 수 있습니다.

gsl_complex **gsl_complex_polar**(double r, double theta)

극좌표 방식의 복소수 $z = r\exp(i\theta) = r(\cos(\theta) + i\sin(\theta))$ 를 주어진 (r theta 에 대해 반환합니다.

5.4 복소수의 성질

double **gsl_complex_arg**(gsl_complex z)

주어진 복소수 z 편각 $\arg(z)$ 을 반환합니다. 편각 $\arg(z)$ 는, $-\pi \leq \arg(z) \leq \pi$ 의 범주를 가집니다.

double **gsl_complex_abs**(gsl_complex z)

주어진 복소수 z 크기, $|z|$ 의 값을 반환합니다.

double **gsl_complex_abs2**(gsl_complex z)

주어진 복소수 z 크기의 제곱 $|z|^2$ 를 반환합니다.

double **gsl_complex_logabs**(gsl_complex z)

주어진 복소수 z 크기에 대한 자연로그 값 $\log(|z|)$ 을 반환합니다. $|z|$ 의 값이 1 에 가까울 때, 정확한 값을 얻을 수 있습니다. 직접 log(gsl_complexabs(z)) 계산하는 경우 정확도를 잃을 수 있습니다.

5.5 복소수 연산자

gsl_complex **gsl_complex_add**(gsl_complex a, gsl_complex b)

주어진 두 복소수 a b 합, $z = a + b$ 를 반환합니다.

gsl_complex **gsl_complex_sub**(gsl_complex a, gsl_complex b)

주어진 두 복소수 a b 차, $z = a - b$ 를 반환합니다.

gsl_complex **gsl_complex_mul**(gsl_complex a, gsl_complex b)

주어진 두 복소수 a b 곱, $z = a \cdot b$ 를 반환합니다.

gsl_complex **gsl_complex_div**(gsl_complex a, gsl_complex b)

주어진 두 복소수 a b 나눗셈, $z = a/b$ 를 반환합니다.

gsl_complex **gsl_complex_add_real**(gsl_complex a, double x)

주어진 복소수 a 실수 x 합, $z = a + x$ 를 반환합니다.

gsl_complex **gsl_complex_sub_real**(gsl_complex a, double x)

주어진 복소수 a 실수 x 차, $z = a - x$ 를 반환합니다.

gsl_complex **gsl_complex_mul_real**(gsl_complex a, double x)

주어진 복소수 a 실수 x 곱, $z = a \cdot x$ 를 반환합니다.

gsl_complex **gsl_complex_div_real**(gsl_complex a, double x)

주어진 복소수 a 실수 x 나눗셈, $z = a/x$ 를 반환합니다.

gsl_complex **gsl_complex_add_imag**(gsl_complex a, double y)

주어진 복소수 a 허수 y 합, $z = a + y$ 를 반환합니다.

gsl_complex **gsl_complex_sub_imag**(gsl_complex a, double y)

주어진 복소수 a 허수 y 차, $z = a - y$ 를 반환합니다.

gsl_complex **gsl_complex_mul_imag**(gsl_complex a, double y)

주어진 복소수 a 허수 y 곱, $z = a \cdot y$ 를 반환합니다.

gsl_complex **gsl_complex_div_imag**(gsl_complex a, double y)

주어진 복소수 a 허수 y 나눗셈, $z = a/y$ 를 반환합니다.

gsl_complex **gsl_complex_conjugate**(gsl_complex z)

주어진 복소수 z 컬레 복소수, $z^* = x - iy$ 를 반환합니다.

gsl_complex **gsl_complex_inverse**(gsl_complex z)

주어진 복소수 z 역수, $\frac{1}{z} = \frac{x-iy}{x^2+y^2}$ 를 반환합니다.

gsl_complex **gsl_complex_negative**(gsl_complex z)

주어진 복소수 z 덧셈 역원, $-z = (-x) + i(-y)$ 를 반환합니다.

5.6 기초 복소 함수들

gsl_complex **gsl_complex_sqrt**(gsl_complex z)

주어진 복소수 z 제곱근, $\sqrt{z}$ 의 값을 반환합니다. 분지 절단은 음의 실수축 에서 이루어집니다. 결과는 항상 복소 평면의 오른쪽 절반 영역에 위치합니다.

gsl_complex **gsl_complex_sqrt_real**(double x)

주어진 실수 x 복소수 제곱근을 반환합니다. x 음수일 수 있습니다.

gsl_complex **gsl_complex_pow**(gsl_complex z, gsl_complex a)

주어진 복소수 z a 대해, z^a 값을 반환합니다. 이 값은 복소수 로그와 지수 함수를 이용해 계산됩니다. $\exp(\log(z) \cdot a))$

gsl_complex **gsl_complex_pow_real**(gsl_complex z, double x)

주어진 복소수 z 대해 주어진 실수 x 승, z^x 값을 반환합니다.

gsl_complex **gsl_complex_exp**(gsl_complex z)

주어진 복소수 z 지수 값, $\exp(z)$ 를 반환합니다.

gsl_complex **gsl_complex_log**(gsl_complex z)

주어진 복소수 z 복소수 자연 로그(밑이 e 인) 값, $\log(z)$ 를 반환합니다. 분지 절단은 음의 실수축에서 이루어집니다.

gsl_complex **gsl_complex_log10**(gsl_complex z)

주어진 복소수 z 대해, 10 을 밑으로 가지는 로그값, $\log_1 0(z)$ 값을 반환합니다.

gsl_complex **gsl_complex_log_b**(gsl_complex z, gsl_complex b)

주어진 복소수 z b 대해, b 밑으로 하는 로그에 대한 z , $\log_b(z)$ 의 값을 반환 합니다. 이 값은 $\frac{\log(z)}{\log(b)}$ 를 반환합니다.

5.7 복소 삼각 함수

gsl_complex **gsl_complex_sin**(gsl_complex z)

주어진 복소수 z sine 값, $\sin(z) = \frac{(\exp(iz)-\exp(-iz))}{2i}$ 을 반환합니다.

gsl_complex **gsl_complex_cos**(gsl_complex z)

주어진 복소수 z cosine 값, $\cos(z) = \frac{(\exp(iz)+\exp(-iz))}{2}$ 을 반환합니다.

gsl_complex **gsl_complex_tan**(gsl_complex z)

주어진 복소수 z tangent 값, $\tan(z) = \frac{\sin(z)}{\cos(z)}$ 을 반환합니다.

gsl_complex **gsl_complex_sec**(gsl_complex z)

주어진 복소수 z secant 값, $\sec(z) = \frac{1}{\cos(z)}$ 을 반환합니다.

gsl_complex **gsl_complex_csc**(gsl_complex z)

주어진 복소수 z 복소수 cosecant 값, $\csc(z) = \frac{1}{\sin(z)}$ 을 반환합니다.

gsl_complex **gsl_complex_cot**(gsl_complex z)

주어진 복소수 z cotangent 값, $\cot(z) = \frac{1}{\tan(z)}$ 을 반환합니다.

5.8 복소 역삼각 함수

gsl_complex **gsl_complex_arcsin_real**(double z)

주어진 복소수 z arcsine 값, $\arcsin(z)$ 을 반환합니다. 분지 절단은 실수 축 위에서 이루어지며, 1 보다 크거나 -1 보다 작은 지점으로 이루어집니다.

gsl_complex **gsl_complex_arcsin**(gsl_complex z)

주어진 실수 z arcsine 값, $\arcsin(z)$ 의 값을 반환합니다. z 값이 -1 과 1 사이에 있을 때, $[-\frac{\pi}{2}, \frac{\pi}{2}]$ 사이의 값을 반환합니다. z 값이 -1 보다 작은 경우 반환 값은 실수부가 $-\frac{\pi}{2}$ 이고 양의 허수부를 가집니다. z 가 1 보다 큰 경우 반환값은 $\frac{\pi}{2}$ 의 실수부와 음의 허수부를 가집니다.

gsl_complex **gsl_complex_arccos**(gsl_complex z)

주어진 복소수 z arccosine 값 $\arccos(z)$ 의 값을 반환합니다. 분지 절단은 실수축 위에서 이루어지며, 1 보다 크거나 -1 보다 작은 지점으로 이루어집니다.

gsl_complex **gsl_complex_arccos_real**(double z)

주어진 실수 z arccosine 값, $\arcsin(z)$ 의 값을 반환합니다. z 값이 -1 과 1 사이에 있을 때, $[0, \pi]$ 사이의 값을 반환합니다. z 값이 -1 보다 작은 경우 반환 값은 실수부가 π 이고 음의 허수부를 가집니다. z 가 1 보다 큰 경우 반환값은 순허수 형태를 가집니다.

gsl_complex **gsl_complex_arctan**(gsl_complex z)

주어진 복소수 z arctan 값, $\arctan(z)$ 을 반환합니다. 분지 절단은 허수 축 위에서 이루어지며, i 보다 크거나 $-i$ 보다 작은 지점으로 이루어집니다.

gsl_complex **gsl_complex_arcsec**(gsl_complex z)

주어진 복소수 z arcsec 값, $\text{arcsec}(z) = \arccos(\frac{1}{z})$ 을 반환합니다.

gsl_complex **gsl_complex_arcsec_real**(double z)

주어진 실수 z arcsec 값, $\text{arcsec}(z) = \arccos(\frac{1}{z})$ 을 반환합니다.

gsl_complex **gsl_complex_arccsc**(gsl_complex z)

주어진 복소수 z arccsc 값, $\text{arccsc}(z) = \arcsin(\frac{1}{z})$ 을 반환합니다.

gsl_complex **gsl_complex_arccsc_real_real**(double z)

주어진 실수 z arccsc 값, $\text{arccsc}(z) = \arcsin(\frac{1}{z})$ 을 반환합니다.

gsl_complex **gsl_complex_arccot**(gsl_complex z)

주어진 복소수 z arccot 값, $\text{arccot}(z) = \arctan(\frac{1}{z})$ 을 반환합니다.

5.9 복소 쌍곡 함수

gsl_complex **gsl_complex_sinh**(gsl_complex z)

주어진 복소수 z sinh 값, $\sinh(z) = \frac{e^z - e^{-z}}{2}$ 을 반환합니다.

gsl_complex **gsl_complex_cosh**(gsl_complex z)

주어진 복소수 z cosh 값, $\cosh(z) = \frac{e^z + e^{-z}}{2}$ 을 반환합니다.

gsl_complex **gsl_complex_tanh**(gsl_complex z)

주어진 복소수 z tanh 값, $\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$ 을 반환합니다.

gsl_complex **gsl_complex_sech**(gsl_complex z)

주어진 복소수 z sech 값, $\text{sech}(z) = \frac{1}{\sinh(z)}$ 을 반환합니다.

gsl_complex **gsl_complex_csch**(gsl_complex z)

주어진 복소수 z csch 값, $\text{csch}(z) = \frac{1}{\cosh(z)}$ 을 반환합니다.

gsl_complex **gsl_complex_coth**(gsl_complex z)

주어진 복소수 z coth 값, $\coth(z) = \frac{\cosh(z)}{\sinh(z)}$ 을 반환합니다.

5.10 복소 역쌍곡 함수

gsl_complex **gsl_complex_arcsinh**(gsl_complex z)

주어진 복소수 z arcsinh 값, $\text{arcsinh}(z)$ 를 반환합니다. 분지 절단은 허수축 위에서 이루어지며, $-i$ 밑 그리고 i 위 입니다.

gsl_complex **gsl_complex_arccosh**(gsl_complex z)

주어진 복소수 z arccosh 값, $\text{arccosh}(z)$ 를 반환합니다. 분지 절단은 실수축 위에서 이루어지며, -1 밑입니다. 한가지 알아두어야 할 점은 Abramowitz & Stegun의 4.6.21식에 있는 음수 근을 사용한다는 점입니다. 해당식은 $\text{arccosh}(z) = \log(z - \sqrt{z^2 - 1})$ 입니다.

gsl_complex **gsl_complex_arccosh_real**(double z)

주어진 실수 z 대해, arccosh 값, $\text{arccosh}(z)$ 를 반환합니다.

gsl_complex **gsl_complex_arctanh**(gsl_complex z)

주어진 복소수 z arctanh 값, $\arctanh(z)$ 를 반환합니다. 분지 절단은 실수축 위에서 이루어지며, -1 밑 그리고 1 위 입니다.

gsl_complex **gsl_complex_arctanh_real**(double z)

주어진 실수 z 대해, arctanh 값, $\arctanh(z)$ 를 반환합니다.

gsl_complex **gsl_complex_arcsech**(gsl_complex z)

복소수 z 대해, arcsech 값 $\operatorname{arcsech}(z) = \arccosh(\frac{1}{z})$ 값을 반환합니다.

gsl_complex **gsl_complex_arccsch**(gsl_complex z)

복소수 z 대해, arccsch 값 $\operatorname{arccsch}(z) = \operatorname{arcsinh}(\frac{1}{z})$ 값을 반환합니다.

gsl_complex **gsl_complex_arccoth**(gsl_complex z)

복소수 z 대해, arccoth 값 $\operatorname{arccoth}(z) = \arctanh(\frac{1}{z})$ 값을 반환합니다.

5.11 참고 문헌과 추가 자료

기초 함수들과 삼각 함수들의 구현체들은 다음의 논문에 기반해 만들어졌습니다.

- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, “Implementing Complex Elementary Functions Using Exception Handling”, ACM Transactions on Mathematical Software, Volume 20 (1994), pp 215-244, Corrigenda, p553
- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, “Implementing the complex arcsin and arccosine functions using exception handling”, ACM Transactions on Mathematical Software, Volume 23 (1997) pp 299-335

일반 식들과 분지점은 다음의 책들을 참고할 수 있습니다.

- Abramowitz & Stegun, Handbook of Mathematical Functions, “Circular Functions in Terms of Real and Imaginary Parts”, Formulas 4.3.55-58, “Inverse Circular Functions in Terms of Real and Imaginary Parts”, Formulas 4.4.37-39, “Hyperbolic Functions in Terms of Real and Imaginary Parts”, Formulas 4.5.49-52, “Inverse Hyperbolic Functions —relation to Inverse Circular Functions”, Formulas 4.6.14-19.
- Dave Gillespie, Calc Manual, Free Software Foundation, ISBN 1-882114-18-3

제 6 장

다항식

이 단원에서는 다항식들의 값을 계산하거나 푸는 함수들을 기술합니다. 해석적 방법을 통해, 2차 3차 식에 대해 실수, 허수 근을 찾는 기능들을 제공합니다. 일반적인 실수 계수의 다항식들에 대해서도 반복적인 방법을 통해 근을 찾는 방법 또한 제공합니다.

이 함수들은 헤더 파일 `gsl_poly.h` 정의되어 있습니다.

6.1 다항식의 계산

다항식 값의 계산은 안전성을 위해 호너 방법(Horner's method)을 사용합니다.

$$P(x) = c[0] + c[1]x + c[2]x^2 + \cdots + c[len-1]x^{len-1}$$

`HAVE_INLINE` 가 정의되어 있으면 inline 함수가 사용됩니다.

double **gsl_poly_eval**(const double c[], const int len, const double x)

실수 값 `x` 대해, 실수 계수를 가지는 다항식의 값을 계산합니다.

gsl_complex **gsl_poly_complex_eval**(const double c[], const int len, const gsl_complex z)

허수 값 `z` 대해, 실수 계수를 가지는 다항식의 값을 계산합니다.

gsl_complex **gsl_complex_poly_complex_eval**(const gsl_complex c[], const int len, const gsl_complex z)

허수 값 `z` 대해, 허수 계수를 가지는 다항식의 값을 계산합니다.

int **gsl_poly_eval_derivs**(const double c[], const size_t lenc, const double x, double res[], const size_t lenres)

다항식과 그 도함수를 계산해 길이 `lenres` 의 배열 `res` 에 저장합니다. 이 배열은 주어진 `x` 값에 대한, $d^k P(x)/dx^k$ 값을 각각의 원소로 가지고 있습니다. k 는 0 부터 시작합니다.

6.2 다항식의 분할 차분 표현

이 단락에서 기술하는 함수들은 뉴턴의 분할 차분법으로 표현된 다항식들을 다룹니다. 분할 차분법은 Abramowitz & Stegun의 25.1.4와 25.2.26 단원, 그리고 Burden and Faires, 3단원에 기술되어 있습니다. 이 단락에서는 분할 차분법의 개요를 서술합니다.

주어진 함수 $f(x)$ 와 n 차수의 다항식 $P_n(x)$ 는 함수 f 와 $n+1$ 개의 서로 다른 지점 $x_0, x_1, \ldots, x_n$ 들로 표현할 수 있습니다. 다음과 같은 다항식의 표현법을 뉴턴의 분할 차분 표현이라 합니다.

$$P_n(x) = f(x_0) + \sum_{k=1}^{n} [x_0, x_1, \ldots, x_k](x - x_0)(x - x_1)\cdots(x - x_{k-1})$$

분할 차분 $[x_0, x_1, \ldots, x_k]$ 은 Abramowitz & Stegun 25.1.4에 정의되어 있습니다. 추가적으로 $x_0, x_1, \ldots, x_k$ 에서 f 의 1 계 도함수 값이 주어졌다면, 차수 $2n+1$ 의 보간 다항식을 만드는 데 사용할 수도 있습니다. 이 보간 다항식은 에르미트 보간 다항식으로 불리며 다음과 같이 정의됩니다.

$$H_{2n+1}(x) = f(z_0) + \sum_{k=1}^{2n+1} [z_0, z_1, \ldots, z_k](x - z_0)(x - z_1)\cdots(x - z_{k-1})$$

$z = \{x_0, x_0, x_1, x_1, \ldots, x_n, x_n\}$ 는 $z_{2k} = z_{2k+1} = x_k$ 로 정의됩니다. 분할 차분 $[z_0, z_1, \ldots, z_k]$ 는 Burden and Faires의 3.4 단원을 참고할 수 있습니다.

int **gsl_poly_dd_init**(double dd[], const double xa[], const double ya[], size_t size)

길이 size 의 배열 xa 와 ya 에 저장된 지점 (x, y) 에 대해, 보간 다항식의 분할 차분 표현을 계산합니다. (xa ya)의 분할 차분 계산 결과는 길이 size 배열 dd 저장됩니다. 개요에서 설명한 서술법을 따라 $dd[k] = [x_0, x_1, \ldots, x_k]$ 입니다.

double **gsl_poly_dd_eval**(const double dd[], const double xa[], const size_t size, const double x)

길이 size 의 배열 xa 와 ya 에 저장된 분할 차분 표현된 다항식의 값을 주어진 변수 x 대해 계산합니다. 매크로 HAVE_INLINE 가 정의되어 있으면 인라인 버전이 사용됩니다.

int **gsl_poly_dd_taylor**(double c[], double xp, const double dd[], const double xa[], size_t size, double w[])

분할 차분 표현된 다항식을 테일러 전개로 바꾸어줍니다. 분할 차분 표현은 길이 size 배열 dd 와 xa 로 표현됩니다. xp 지점의 테일러 계수들은 배열 c 에 저장됩니다. c 배열도 size 크기의 길이를 가집니다. 배열 w 는 길이가 size 로 같습니다.

int **gsl_poly_dd_hermite_init**(double dd[], double za[], const double xa[], const double ya[], const double dya[], const size_t size)

길이 size 배열 xa 와 ya 에 저장된 지점 (x, y) 들에 대해, 에르미트 보간 다항식의 분할 차분 표현을 계산합니다. 에르미트 보간법으로 만들어지는 다항식은 1 계 도함수 dy/dx 의 값을 필요로 합니다. 이 값은 길이 size 의 배열 dya 로 주어져 있습니다. 1 계 도함수 값들은 새로운 자료 집합

$z = \{x_0, x_0, x_1, x_1 \ldots\}$ 를 정의해서, 일반적인 분할 차분법에 통합시킬 수 있습니다. 이 값들은 길이 2· size 의 배열 za 에 저장되어 있습니다. 계산 결과들은 2· size 길이를 가지는 배열 dd 저장됩니다. 개요에서 설명한 서술법을 따라 $dd[k] = [z_0, z_1, \ldots, z_k]$ 로 표현됩니다. 계산된 에르미트 다항식은 gsl_poly_dd_eval() 함수를 호출해 xa 에 대한 za 값을 넘겨 계산될 수 있습니다.

6.3 2 차 다항식 (Quadratic Equations)

int **gsl_poly_solve_quadratic**(double a, double b, double c, double *x0, double *x1)

2차 다항식

$$ax^2 + bx + c = 0$$

의 실수 근을 찾습니다. 근의 갯수(0,1,2)를 반환하며, 각 근의 위치는 x0 와 x 에 저장됩니다. 만약, 실수 근이 존재하지 않는다면 x0 와 x1 의 값을 수정하지 않습니다. 한 개의 근만이 있는 경우(예를 들어 $a = 0$)는 x0 에 저장됩니다. 두 개의 근이 존재하면 x0 와 x1 는 각각 오름차순으로 저장됩니다. 중근의 경우는 특별히 취급되지 않습니다. 예를 들어, $(x - 1)^2 = 0$ 은 값이 같은 두 개의 근을 가지는 방정식으로 취급됩니다.

근의 갯수는 $b^2 - 4ac$ 의 부호로 판별됩니다. 배 정밀도의 계산에서 이 방법은 반올림과 소거 오차의 영향을 받으며, 다항식의 계수가 정확하지 않을 때 비슷한 오류를 가질 수 있습니다. 하지만, 작은 정수 계수를 가지는 다항식에서는 정확하게 계산할 수 있습니다.

int **gsl_poly_complex_solve_quadratic**(double a, double b, double c, gsl_complex *z0, gsl_complex *z1)

2차 다항식

$$az^2 + bz + c = 0$$

의 복소수 근을 계산합니다.

함수의 반환 값은 복소수 근의 숫자를 의미합니다. (1 이거나 2 입니다.) 각 근의 위치는 z0 z1 에 저장됩니다. 저장되는 순서는 오름차순으로 저장되고, 실수부를 우선으로 판정하고 그 다음 허수부의 크기를 기준으로 배열합니다. 만약 한 개의 실수 근만 존재하면 (예를 들어 $a = 0$) z0 에 저장됩니다.

6.4 3 차 다항식 (Cubic Equations)

int **gsl_poly_solve_cubic**(double a, double b, double c, double *x0, double *x1, double *x2)

최고 차항의 계수가 1인 3차 다항식

$$x^3 + ax^2 + bx + c = 0$$

의 실수 근을 계산합니다. 실수 근의 숫자 (1-3)을 반환합니다. 이 근들의 위치는 x0 , x1 그리고 x2 에 저장됩니다. 만약 한 개의 실수 근만이 존재한다면, x0 에 저장됩니다. 세 개의 근이 존재한다면, 오름차순으로 x0 , x1 그리고 x2 에 저장됩니다. 중근은 특별하게 취급하지 않습니다. 예로 $(x-1)^3 = 0$ 인 경우, 같은 값을 가지는 세 개의 근을 가지는 것으로 취급됩니다.

2차 다항식의 경우와 같이, 유한한 정밀도로 인해 밀접한 실수 근들이 실수 축에서 복소수 평면으로 이동해 근의 숫자가 달라질 수 있습니다.

int **gsl_poly_complex_solve_cubic**(double a, double b, double c, gsl_complex *z0, gsl_complex *z1, gsl_complex *z2)

3차 다항식

$$z^3 + az^2 + bz + c = 0$$

의 복소수 근을 찾습니다. 복소수 근의 숫자를 의미합니다(항상 3 입니다). 각 근의 위치는 z0 , z1 그리고 z2 에 저장됩니다. 각 근은 오름차순으로 실수부를 우선 판정하고, 허수부를 판정해 결정합니다.

6.5 일반 다항식

일반적으로 2차, 3차 그리고 4차 다항식같은 특수한 경우를 제외하면, 다항식 근은 해석적으로 찾을 수 없습니다. 이 단원에서 서술하는 알고리즘은 이러한 고차 다항식들의 근들을 반복적인 방법을 이용해 근사적인 위치를 구해줍니다.

type **gsl_poly_complex_workspace**

일반적인 다항식의 근들을 찾기 위한 인자들을 가진 작업 공간입니다.

gsl_poly_complex_workspace ***gsl_poly_complex_workspace_alloc**(size_t n)

gsl_poly_complex_workspace 구조체를 할당합니다. 이 작업 공간은 n 개의 계수를 가지는 다항식을 푸는 함수 gsl_poly_complex_solve() 를 위한 공간입니다.

오류가 생기지 않는다면, 새로 할당된 gsl_poly_complex_workspace 를 가르키는 포인터를 반환하고, 오류가 생기면 NULL 포인터를 반환합니다.

void **gsl_poly_complex_workspace_free**(gsl_poly_complex_workspace *w)

작업 공간 w rk 할당된 모든 메모리를 해제합니다.

int **gsl_poly_complex_solve**(const double *a, size_t n, gsl_poly_complex_workspace *w,
gsl_complex_packed_ptr z)

일반 다항 함수

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

의 근들을 계산합니다. 동반 행렬(companion matrix)의 균형-QR 차원 감소를 이용합니다[1] . 인자 n 는 계수 배열의 길이를 나타냅니다. 가장 높은 차수의 계수는 반드시 0 이 아니여야 합니다. 적절한 크기의 작업 공간 w 를 필요로 합니다. 총, $n-1$ 개의 근들이 반환되며, 크기 $2(n-1)$ 의 복소수 배열 z 에 실수부-허수부 순서로 반복되어 저장됩니다.

모든 근들을 찾으면, GSL_SUCCESS 값으 반환합니다. 만약 QR 차원 감소가 수렴하지 않으면, 오류 관리자가 호출되고 GSL_EFAILED 오류 값를 전달합니다. 유의할 점은 유한한 정확도로 인해, 높은 차수의 중첩근은 저하된 정확도를 가지는 여러개의 단일 근들로 반환됩니다. 이러한 고-중첩근은 다중 구조를 고려하는 특별한 알고리즘이 필요합니다[2] .

6.6 예시

다음의 예시는 $P(x) = x^5 - 1$ 다항식을 이용해 주어진 일반적인 다항식의 해를 찾는 기능을 보여줍니다.

이 다항식은 다음의 5개 근을 가집니다.

$$1, e^{2\pi i/5}, e^{4\pi i/5}, e^{6\pi i/5}, e^{8\pi i/5}$$

다음 코드는 이 근들을 찾아줍니다.

```
#include <stdio.h>
#include <gsl/gsl_poly.h>

int
main (void)
{
  int i;
```

(다음 페이지에 계속)

[1] balanced-QR reduction

[2] 26. Zeng, Algorithm 835, ACM Transactions on Mathematical Software, Volume 30, Issue 2 (2004), pp 218-236 을 참고할 수 있습니다.

(이전 페이지에서 계속)

```
  /* coefficients of P(x) =  -1 +    x^5  */
  double a[6] = { -1, 0, 0, 0, 0, 1 }    ;
  double z[10];

  gsl_poly_complex_workspace * w
      =   gsl_poly_complex_workspace_allo  c (6);

  gsl_poly_complex_solve (a, 6, w, z)    ;

  gsl_poly_complex_workspace_free (w)    ;

  for (i = 0; i < 5; i++)
    {
      printf ("z%d = %+.18f %+.  18f\n",
              i, z[2*i], z[2*i+1]);
    }

  return 0;
}
```

프로그램의 결과 값은 다음과 같습니다.

```
z0 = -0.809016994374947673 +0.587785252292473359
z1 = -0.809016994374947673 -0.587785252292473359
z2 = +0.309016994374947507 +0.951056516295152976
z3 = +0.309016994374947507 -0.951056516295152976
z4 = +0.999999999999999889 +0.000000000000000000
```

이 결과는 $z_n = e^{2\pi n i/5}$ 의 해석적 값들과 일치합니다.

6.7 참고 문헌과 추가 자료

균형 QR 방법과 이 방법의 오차 분석은 다음의 논문들에 기술되어 있습니다.

- R.S. Martin, G. Peters and J.H. Wilkinson, “The QR Algorithm for Real Hessenberg Matrices”, Numerische Mathematik, 14 (1970), 219-231.
- B.N. Parlett and C. Reinsch, “Balancing a Matrix for Calculation of Eigenvalues and Eigenvectors”, Numerische Mathematik, 13 (1969), 293-304.
- A. Edelman and H. Murakami, “Polynomial roots from companion matrix eigenvalues”, Mathematics of Computation, Vol.: 64, No.: 210 (1995), 763-776.

분할 차분법 식들은 다음 문헌들에 기반합니다.

- Abramowitz and Stegun, Handbook of Mathematical Functions, Sections 25.1.4 and 25.2.26.
- R. L. Burden and J. D. Faires, Numerical Analysis, 9th edition, ISBN 0-538-73351-9, 2011.

제 7 장

특수 함수

이 단원에서는 GSL이 제공하는 특수 함수 구현체들을 기술합니다. 라이브러리에서 에어리 함수, 베셀 함수, 클라우센 함수, 쿨룽 파동 함수, 커플링 계수, 도슨 함수, 디바이 함수, 다이로그, 타원 적분, 자코비 타원적분, 오류 함수, 지수적 적분, 페르미-디렉 함수, 감마 함수, 구겐바우어 함수, 에르미트 다항식과 함수, 초기하 함수, 라게르 함수, 르장그르 함수, 구면 조화 함수, 프사이(디감마)함수, 싱크로트론 함수, 전송 함수, 삼각함수, 제타 함수 등의 구현체들을 제공합니다.

각각의 구현체는 수치적 오류와 함수의 값을 함께 계산합니다.

기술된 함수 구현체들은 각각 다른 헤더파일에 정의되어 있습니다. 예를 들어서 `gsl_sf_airy.h` 나 `gsl_sf_bessel.h` 로 각각 나뉘어져 있습니다. 모두 포함시켜서 계산하려면, `gsl_sf.h` 을 포함시키면 됩니다.

7.1 함수의 사용

함수 구현체들은 두 가지 형태로 호출할 수 있습니다. 하나는 함수의 값을 불러오는 호출입니다. 이는 주어진 인자에 대한 함수의 수치값을 반환합니다. 다른 하나는 오류 관리를 위한 호출 형태입니다. 이 호출은 함수 값과 함께 오류 값을 반환합니다. 이 두 가지 호출 형태는 모두 동일한 코드에 접근합니다. 즉, 오류 관리 호출에서 얻은 계산 값과 곧바로 계산 값을 받는 호출에서 얻은 계산 값은 정확히 동일합니다.

참고: 적어도 동일한 과정을 거친 계산입니다(*).

함수 값을 불러오는 호출은 말 그대로 인자에 대해 값을 계산한 수치 값만을 반환합니다. 이는 바로 수학 수식 표현에 사용할 수 있습니다. 예를 들어서 다음의 함수 호출은 베셀 함수 $J_0(x)$ 의 값을 계산합니다.

```
double y = gsl_sf_bessel_J0(x);
```

이 방법은 오류 값를 확인하거나 오차 값을 계산할 수 없습니다. 해당 정보를 얻기 위해서는 오류 관리 호출 형태로 대체해서 해당 값들을 저장할 변수를 인자에 하나 더 추가해야 합니다.

```
gsl_sf_result result;
int status = gsl_sf_bessel_J0_e (x, &result);
```

이러한 오류 관리 함수는 _e 접미사를 가지고 있습니다. 반환 값은 오류의 조건을 나타냅니다. 대표적으로 오버플로우, 언더플로우 아니면 오차를 나타냅니다. 만약, 오류가 없다면 오류 관리 함수는 GSL_SUCCESS 값을 반환합니다.

7.2 gsl_sf_result 구조체

특수 함수들의 구현체들은 결과 값을 계산하는 과정에서 오차를 함께 계산합니다. 이를 위해, 함수의 오차와 결과값을 함께 다룰 수 있는 구조체를 함께 제공합니다. 이 구조체는 헤더파일 gsl_sf_result.h 정의되어 있습니다.

이 구조체는 다음과 같이 함수 값과 오차 값으로 이루어져 있습니다.

type **gsl_sf_result**

```
typedef struct
{
  double val;
  double err;
} gsl_sf_result;
```

val 는 함수의 값을 저장하고, err 는 함수 값의 절대 오차를 나타냅니다.

몇몇 경우에, 오버플로우나 언더플로우가 함수에 의해 감지될 수 있습니다. 그러한 경우에 크기를 나타내는 지수 값을 오차/값 쌍과 함께 반환할 수도 있습니다. 이러한 방법은 내장된 변수의 동적 크기를 초과해서 결과를 나타낼 수 있는 장점이 있습니다. 이를 위해 다음과 같이 함수 값, 오차, 그리고 지수 값을 포함하는 구조체를 사용합니다. 실제 값은 result * 10^(e10) 을 계산해야 합니다.

type **gsl_sf_result_e10**

```
typedef struct
{
  double val;
  double err;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  int    e10;
} gsl_sf_result_e10;
```

7.3 모드

특수 함수 구현체들의 제공 목적은 배정밀도를 가지는 특수 함수 값의 제공을 위함입니다. 그러나 몇몇 특수 함수들의 경우, 배정밀도 수준의 정밀도를 얻기 위해서 매우 높은 항이 필요한 경우도 있습니다. 이러한 경우에 `gsl_mode_t` 자료형으로 정의된 `mode` 인자를 이용해 요구 정밀도를 낮추어 연산 효율을 높일 수 있습니다. 다음의 정밀도 값들이 `mode` 인자에 들어갈 수 있습니다.

type **gsl_mode_t**

GSL_PREC_DOUBLE

배정밀도, 근사적으로 $2 \cdot 10^{-16}$ 의 상대 정확도를 가짐.

GSL_PREC_SINGLE

단정밀도, 근사적으로 $1 \cdot 10^{-7}$ 의 상대 정확도를 가짐.

GSL_PREC_APPROX

근사 값, 근사적으로 $5 \cdot 10^{-4}$ 의 상대 정확도를 가짐.

근사 상태는 가장 빠른 계산 시간을 보장하지만 가장 낮은 정확도를 가집니다.

7.4 에어리 함수와 도함수(Airy function & Derivative)

에어리 함수(Airy function) $Ai(x)$ 와 $Bi(x)$ 는 적분표현으로 다음과 같이 정의됩니다.

$$Ai(x) = \frac{1}{\pi} \int_0^\infty \cos(t^3/3 + xt)\, dt$$
$$Bi(x) = \frac{1}{\pi} \int_0^\infty (e^{-t^3/3+xt} + \sin(t^3/3 + xt))\, dt$$

더 자세한 정보는 Abramowitz & Stengu, Section 10.4를 참고할 수 있습니다. 에어리 함수들은 헤더 파일 `gsl_sf_airy.h` 에 정의되어 있습니다.

7.4.1 에어리 함수

double **gsl_sf_airy_Ai**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Ai_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

에어리 함수 $Ai(x)$ 정밀도에 따라 계산합니다.

double **gsl_sf_airy_Bi**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Bi_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

에어리 함수 $Bi(x)$ 정밀도에 따라 계산합니다.

double **gsl_sf_airy_Ai_scaled**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Ai_scaled_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

조정된 에어리 함수 $S_A(x)Ai(x)$ 의 값을 계산합니다. $x > 0$ 의 값은 $exp(\frac{2}{3}x^{\frac{3}{2}})$ 이고, $x < 0$ 일 때는 1 입니다.

double **gsl_sf_airy_Bi_scaled**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Bi_scaled_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

조정된 에어리 함수 $S_A(x)Bi(x)$ 의 값을 계산합니다. $x > 0$ 일 때, 조정 계수 $S_A(x)$ 의 값은 $exp(\frac{2}{3}x^{\frac{3}{2}})$ 이고, $x < 0$ 일 때는 1 입니다.

7.4.2 에어리 함수의 도함수

double **gsl_sf_airy_Ai_deriv**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Ai_deriv_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

에어리 함수의 도함수 $Ai'(x)$ 의 값을 주어진 `mode` 정밀도에 따라 계산합니다.

double **gsl_sf_airy_Bi_deriv**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Bi_deriv_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

에어리 함수의 도함수 $Bi'(x)$ 의 값을 주어진 `mode` 정밀도에 따라 계산합니다.

double **gsl_sf_airy_Ai_deriv_scaled**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Ai_deriv_scaled_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

조정된 에어리 도함수 $S_A(x)Ai'(x)$ 의 값은 $exp(\frac{2}{3}x^{\frac{3}{2}})$ 이고, $x < 0$ 일 때는 1 입니다.

double **gsl_sf_airy_Bi_deriv_scaled**(double x, gsl_mode_t mode)

int **gsl_sf_airy_Bi_deriv_scaled_e**(double x, gsl_mode_t mode, gsl_sf_result *result)

조정된 에어리 도함수 $S_A(x)Bi'(x)$ 의 값을 계산합니다. $x > 0$ 의 값은 $exp(\frac{2}{3}x^{\frac{3}{2}})$ 이고, $x < 0$ 일 때는 1 입니다.

7.4.3 에어리 함수의 근

double **gsl_sf_airy_zero_Ai**(unsigned int s)

int **gsl_sf_airy_zero_Ai_e**(unsigned int s, gsl_sf_result *result)

에어리 함수 $Ai(x)$ 번째 근을 찾아 반환합니다.

double **gsl_sf_airy_zero_Bi**(unsigned int s)

int **gsl_sf_airy_zero_Bi_e**(unsigned int s, gsl_sf_result *result)

에어리 함수 $Bi(x)$ 번째 근을 찾아 반환합니다.

7.4.4 에어리 도함수의 근

double **gsl_sf_airy_zero_Ai_deriv**(unsigned int s)

int **gsl_sf_airy_zero_Ai_deriv_e**(unsigned int s, gsl_sf_result *result)

에어리 도함수 $Ai'(x)$ 번째 근을 찾아 반환합니다.

double **gsl_sf_airy_zero_Bi_deriv**(unsigned int s)

int **gsl_sf_airy_zero_Bi_deriv_e**(unsigned int s, gsl_sf_result *result)

에어리 도함수 $Bi'(x)$ 번째 근을 찾아 반환합니다.

7.5 베셀 함수(Bessel function)

이 단원은 원통형 베셀 함수 $J_n(x), Y_n(x)$, 수정 베셀 함수 $I_n(x), K_n(x)$, 구면 베셀 함수 $j_l(x), y_l(x)$ 그리고 수정 구면 베셀 함수 $i_l(x), k_l(x)$ 에 정의되어 있습니다.

7.5.1 1종 베셀 함수

동의어: Regular Bessel function

double **gsl_sf_bessel_J0**(double x)

int **gsl_sf_bessel_J0_e**(double x, gsl_sf_result *result)

0 차수의 1종 베셀 함수 $J_0(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_J1**(double x)

int **gsl_sf_bessel_J1_e**(double x, gsl_sf_result *result)

1 차수의 1종 베셀 함수 $J_1(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_Jn**(int n, double x)

int **gsl_sf_bessel_Jn_e**(int n, double x, gsl_sf_result *result)

주어진 n 차수의 1종 베셀 함수 $J_n(x)$ 의 값을 계산합니다.

int **gsl_sf_bessel_Jn_array**(int nmin, int nmax, double x, double result_array[])

nmin 에서 nmax 까지, 1종 베셀 함수 $J_n(x)$ 의 값을 계산합니다. 계산 결과값은 result_array 배열에 저장됩니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

7.5.2 2종 베셀 함수

동의어: 노이먼 함수(Neumann funtion), Irregular Bessel function

double **gsl_sf_bessel_Y0**(double x)

int **gsl_sf_bessel_Y0_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 0 차수의 2종 베셀 함수 $Y_0(x)$ 값을 계산합니다.

double **gsl_sf_bessel_Y1**(double x)

int **gsl_sf_bessel_Y1_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 1 차수의 2종 베셀 함수 $Y_1(x)$ 값을 계산합니다.

double **gsl_sf_bessel_Yn**(int n, double x)

int **gsl_sf_bessel_Yn_e**(int n, double x, gsl_sf_result *result)

$x > 0$ 에 대해, 주어진 n 차수의 2종 베셀 함수 $Y_1(x)$ 값을 계산합니다.

int **gsl_sf_bessel_Yn_array**(int nmin, int nmax, double x, double result_array[])

nmin 에서 nmax 까지, 2종 베셀 함수 $Y_n(x)$ 의 값을 계산합니다. 계산 결과값은 result_array 배열에 저장됩니다. 이 함수의 정의역은 $x > 0$ 입니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

7.5.3 1종 변형 베셀 함수(Regular Modified Cylindrical Bessel Functions)

double **gsl_sf_bessel_I0**(double x)

int **gsl_sf_bessel_I0_e**(double x, gsl_sf_result *result)

0 차수의 1종 변형 베셀 함수 $I_0(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_I1**(double x)

int **gsl_sf_bessel_I1_e**(double x, gsl_sf_result *result)

1 차수의 1종 변형 베셀 함수 $1_0(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_In**(int n, double x)

int **gsl_sf_bessel_In_e**(int n, double x, gsl_sf_result *result)

주어진 n 차수의 1종 변형 베셀 함수 $I_n(x)$ 의 값을 계산합니다.

int **gsl_sf_bessel_In_array**(int nmin, int nmax, double x, double result_array[])

nmin 에서 nmax 까지, 1종 변형 베셀 함수 $I_n(x)$ 의 값을 계산합니다. 계산 결과값은 result_array 배열에 저장됩니다. nmin 는 반드시 양수이거나 0 이어야 합니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

double **gsl_sf_bessel_I0_scaled**(double x)

int **gsl_sf_bessel_I0_scaled_e**(double x, gsl_sf_result *result)

조정 계수가 곱해진 0 차수의 1종 변형 베셀 함수 $\exp(-|x|)I_0(x)$ 를 계산합니다.

double **gsl_sf_bessel_I1_scaled**(double x)

int **gsl_sf_bessel_I1_scaled_e**(double x, gsl_sf_result *result)

조정 계수가 곱해진 1 차수의 1종 변형 베셀 함수 $\exp(-|x|)I_1(x)$ 를 계산합니다.

double **gsl_sf_bessel_In_scaled**(int n, double x)

int **gsl_sf_bessel_In_scaled_e**(int n, double x, gsl_sf_result *result)

조정 계수가 곱해진, n 차수의 1종 변형 베셀 함수 $\exp(-|x|)I_n(x)$ 를 계산합니다.

int **gsl_sf_bessel_In_scaled_array**(int nmin, int nmax, double x, double result_array[])

nmin 에서 nmax 까지, 조정 계수가 곱해진, 1종 변형 베셀 함수 $\exp(-|x|)I_n(x)$ 의 값을 계산합니다. 계산 결과값은 result_array 배열에 저장됩니다. nmin 반드시 양수이거나 0 이어야 합니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

7.5.4 2종 변형 베셀 함수

double **gsl_sf_bessel_K0**(double x)

int **gsl_sf_bessel_K0_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 0 차수의 2종 변형 베셀 함수 $K_0(x)$ 값을 계산합니다.

double **gsl_sf_bessel_K1**(double x)

int **gsl_sf_bessel_K1_e**(double x, gsl_sf_result *result)

$x > 0$ 차수의 2종 변형 베셀 함수 $K_1(x)$ 값을 계산합니다.

double **gsl_sf_bessel_Kn**(int n, double x)

int **gsl_sf_bessel_Kn_e**(int n, double x, gsl_sf_result *result)

$x > 0$ 차수의 2종 변형 베셀 함수 $K_n(x)$ 값을 계산합니다.

int **gsl_sf_bessel_Kn_array**(int nmin, int nmax, double x, double result_array[])

nmin 에서 nmax 까지, 2종 변형 베셀 함수 $K_n(x)$ 의 값을 계산합니다. 계산 결과값은 result_array 배열에 저장됩니다. nmin 반드시 양수이거나 0 이어야 합니다. 함수의 정의역은 $x > 0$ 입니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

double **gsl_sf_bessel_K0_scaled**(double x)

int **gsl_sf_bessel_K0_scaled_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 조정 계수가 곱해진 0 차수의 2종 변형 베셀 함수 $\exp(x)K_0(x)$ 를 계산합니다.

double **gsl_sf_bessel_K1_scaled**(double x)

int **gsl_sf_bessel_K1_scaled_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 조정 계수가 곱해진 1 차수의 2종 변형 베셀 함수 $\exp(x)K_1(x)$ 를 계산합니다.

double **gsl_sf_bessel_Kn_scaled**(int n, double x)

int **gsl_sf_bessel_Kn_scaled_e**(int n, double x, gsl_sf_result *result)

$x > 0$ 차수의 2종 변형 베셀 함수 $\exp(x)K_n(x)$ 를 계산합니다.

int **gsl_sf_bessel_Kn_scaled_array**(int nmin, int nmax, double x, double result_array[])

`nmin` 에서 `nmax` 까지, 조정 계수가 곱해진 2종 변형 베셀 함수 $\exp(x)K_n(x)$ 의 값을 계산합니다. 계산 결과값은 `result_array` 배열에 저장됩니다. `nmin` 는 반드시 양수이거나 0 이어야 합니다. 함수의 정의역은 $x > 0$ 입니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

7.5.5 1종 구면 베셀 함수

double **gsl_sf_bessel_j0**(double x)

int **gsl_sf_bessel_j0_e**(double x, gsl_sf_result *result)

0 차수의 1종 구면 베셀 함수 $j_0(x) = \sin(x)/x$ 의 값을 계산합니다.

double **gsl_sf_bessel_j1**(double x)

int **gsl_sf_bessel_j1_e**(double x, gsl_sf_result *result)

1 차수의 1종 구면 베셀 함수 $j_1(x) = (\sin(x)/x - \cos(x))/x$ 의 값을 계산합니다.

double **gsl_sf_bessel_j2**(double x)

int **gsl_sf_bessel_j2_e**(double x, gsl_sf_result *result)

2 차수의 1종 구면 베셀 함수 $j_2(x) = ((3/x^2 - 1)\sin(x) - 3\cos(x)/x)/x$ 의 값을 계산합니다.

double **gsl_sf_bessel_jl**(int l, double x)

int **gsl_sf_bessel_jl_e**(int l, double x, gsl_sf_result *result)

l 차수의 1종 구면 베셀 함수 $j_l(x)$ 의 값을 계산합니다. x, l 은 $l \geq 0, x \geq 0$ 이어야 합니다.

int **gsl_sf_bessel_jl_array**(int lmax, double x, double result_array[])

$lmax \geq 0, x \geq 0$ 에 대해, 1종 구면 베셀 함수 $j_l(x)$ 의 값을 $l = 0$ 에서 $l = lmax$ 까지 계산합니다. 계산 결과값은 `result_array` 배열에 저장됩니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

int **gsl_sf_bessel_jl_steed_array**(int lmax, double x, double *result_array)

Steed 방법을 이용해 1종 구면 베셀 함수 $j_l(x)$ 의 값을 $l = 0$ 에서 $l = lmax$ 까지 계산합니다. $lmax, x$

는 $lmax \geq 0, x \geq 0$ 이어야 합니다. 계산 결과값은 result_array 배열에 저장됩니다. Steed/Barnett 알고리즘은 Comp. Phys. Comm. 21, 297(1981)에 기술되어 있습니다. Steed 방법은 다른 함수의 재귀적 방법보다 더 안정적이지만, 그 대신 더 느립니다.

7.5.6 2종 구면 베셀 함수

double **gsl_sf_bessel_y0**(double x)

int **gsl_sf_bessel_y0_e**(double x, gsl_sf_result *result)

0 차수의 2종 구면 베셀 함수 $y_0(x) = -\cos(x)/x$ 의 값을 계산합니다.

double **gsl_sf_bessel_y1**(double x)

int **gsl_sf_bessel_y1_e**(double x, gsl_sf_result *result)

1 차수의 2종 구면 베셀 함수 $y_1(x) = -(\cos(x)/x + \sin(x))/x$ 의 값을 계산합니다.

double **gsl_sf_bessel_y2**(double x)

int **gsl_sf_bessel_y2_e**(double x, gsl_sf_result *result)

2 차수의 2종 구면 베셀 함수 $y_2(x) = (-3/x^3 + 1/x)\cos(x) - (3/x^2)\sin(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_yl**(int l, double x)

int **gsl_sf_bessel_yl_e**(int l, double x, gsl_sf_result *result)

$l \geq 0$ 에 대해, l 차수의 2종 구면 베셀 함수 $y_l(x)$ 의 값을 계산합니다.

int **gsl_sf_bessel_yl_array**(int lmax, double x, double result_array[])

$lmax \geq 0$ 에 대해, 2종 구면 베셀 함수 $y_l(x)$ 의 값을 $l = 0$ 에서 $l = lmax$ 까지 계산합니다. 계산 결과값은 result_array 배열에 저장됩니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

7.5.7 1종 변형 구면 베셀 함수

1종 변형 구면 베셀 함수 $i_l(x)$ 는 분수 차수의 1종 수정 베셀 함수와 다음과 같은 관계를 가집니다.

$$i_l(x) = \sqrt{\pi/(2x)} I_{l+1/2}(x)$$

double **gsl_sf_bessel_i0_scaled**(double x)

int **gsl_sf_bessel_i0_scaled_e**(double x, gsl_sf_result *result)

조정 계수가 곱해진, 0 차수의 1종 변형 구면 베셀 함수 $\exp(-|x|)i_0(x)$ 를 계산합니다.

double **gsl_sf_bessel_i1_scaled**(double x)

int **gsl_sf_bessel_i1_scaled_e**(double x, gsl_sf_result *result)

조정 계수가 곱해진, 1 차수의 1종 변형 구면 베셀 함수 $\exp(-|x|)i_1(x)$ 를 계산합니다.

double **gsl_sf_bessel_i2_scaled**(double x)

int **gsl_sf_bessel_i2_scaled_e**(double x, gsl_sf_result *result)

조정 계수가 곱해진, 2 차수의 1종 변형 구면 베셀 함수 $\exp(-|x|)i_2(x)$ 를 계산합니다.

double **gsl_sf_bessel_il_scaled**(int l, double x)

int **gsl_sf_bessel_il_scaled_e**(int l, double x, gsl_sf_result *result)

조정 계수가 곱해진, l 차수의 1종 변형 구면 베셀 함수 $\exp(-|x|)i_2(x)$ 를 계산합니다.

int **gsl_sf_bessel_il_scaled_array**(int lmax, double x, double result_array[])

$lmax \geq 0, x \geq 0$ 에 대해, 조정 계수가 곱해진 1종 변형 구면 베셀 함수 $\exp(-|x|)i_l(x)$ 의 값을 $l = 0$ 에서 $l = lmax$ 까지 계산합니다. 계산 결과값은 `result_array` 배열에 저장됩니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

7.5.8 2종 변형 구면 베셀 함수

2종 변형 구면 베셀 함수 $k_l(x)$ 는 분수 차수 2종 구면 베셀 함수와 다음과 같은 관계를 가집니다.

$$k_l(x) = \sqrt{\pi/(2x)K_{l+1/2}(x)}$$

double **gsl_sf_bessel_k0_scaled**(double x)

int **gsl_sf_bessel_k0_scaled_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 조정 계수가 곱해진 0 차수의 2종 변형 구면 베셀 함수 $\exp(x)k_0(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_k1_scaled**(double x)

int **gsl_sf_bessel_k1_scaled_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 조정 계수가 곱해진 1 차수의 2종 변형 구면 베셀 함수 $\exp(x)k_1(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_k2_scaled**(double x)

int **gsl_sf_bessel_k2_scaled_e**(double x, gsl_sf_result *result)

$x > 0$ 에 대해, 조정 계수가 곱해진 2 차수의 2종 변형 구면 베셀 함수 $\exp(x)k_2(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_kl_scaled**(int l, double x)

int **gsl_sf_bessel_kl_scaled_e**(int l, double x, gsl_sf_result *result)

$x > 0$ 에 대해, 조정 계수가 곱해진 l 차수의 2종 변형 구면 베셀 함수 $\exp(x)k_l(x)$ 의 값을 계산합니다.

int **gsl_sf_bessel_kl_scaled_array**(int lmax, double x, double result_array[])

$lmax \geq 0, x \geq 0$ 에 대해, 조정 계수가 곱해진 1종 변형 구면 베셀 함수 $\exp(x)k_l(x)$ 의 값을 $l = 0$ 에서 $l = lmax$ 까지 계산합니다. 계산 결과값은 `result_array` 배열에 저장됩니다. 재귀식을 이용해 계산 효율을 높여, 실제 값과는 조금 다를 수 있습니다.

7.5.9 1종 베셀 함수-분수 차수

double **gsl_sf_bessel_Jnu**(double nu, double x)

int **gsl_sf_bessel_Jnu_e**(double nu, double x, gsl_sf_result *result)

분수 차수 ν 에 대해, 1종 베셀 함수 $J_\nu(x)$ 의 값을 계산합니다.

int **gsl_sf_bessel_sequence_Jnu_e**(double nu, gsl_mode_t mode, size_t size, double v[])

분수 차수 ν 의 1종 배셀함수 $J_\nu(x)$ 의 값을 주어진 x 값 배열에 대해 계산합니다. $size$ 길이의 배열 v 은 x 값들을 담고있습니다. 함수는 이 배열이 양수가 순차적으로 배열되어 있다 가정합니다. v 배열을 수정해 $J_\nu(x_i)$ 의 값을 덮어 씌웁니다.

7.5.10 2종 베셀 함수-분수 차수

double **gsl_sf_bessel_Ynu**(double nu, double x)

int **gsl_sf_bessel_Ynu_e**(double nu, double x, gsl_sf_result *result)

분수 차수 ν 에 대해, 2종 베셀 함수 $Y_\nu(x)$ 의 값을 계산합니다.

7.5.11 1종 변형 베셀 함수-분수 차수

double **gsl_sf_bessel_Inu**(double nu, double x)

int **gsl_sf_bessel_Inu_e**(double nu, double x, gsl_sf_result *result)

$x > 0, \nu > 0$ 에 대해, 분수 차수 ν 의 1종 변형 베셀 함수 $I_\nu(x)$ 를 계산합니다.

double **gsl_sf_bessel_Inu_scaled**(double nu, double x)

int **gsl_sf_bessel_Inu_scaled_e**(double nu, double x, gsl_sf_result *result)

$x > 0, \nu > 0$ 에 대해, 조정 계수가 곱해진 분수 차수 ν 의 2종 변형 베셀 함수 $\exp(-|x|)I_\nu(x)$ 의 값을 계산합니다.

7.5.12 2종 변형 베셀 함수-분수 차수

double **gsl_sf_bessel_Knu**(double nu, double x)

int **gsl_sf_bessel_Knu_e**(double nu, double x, gsl_sf_result *result)

$x > 0, \nu > 0$ 에 대해, 분수 차수 ν 의 2종 변형 베셀 함수 $K_\nu(x)$ 의 값을 계산합니다.

double **gsl_sf_bessel_lnKnu**(double nu, double x)

int **gsl_sf_bessel_lnKnu_e**(double nu, double x, gsl_sf_result *result)

$x > 0, \nu > 0$ 에 대해, 로그가 씌워진, 분수 차수 ν 의 2종 변형 베셀 함수 $\ln(K_\nu(x))$ 의 값을 계산합니다.

double **gsl_sf_bessel_Knu_scaled**(double nu, double x)

int **gsl_sf_bessel_Knu_scaled_e**(double nu, double x, gsl_sf_result *result)

$x > 0, \nu > 0$ 에 대해, 조정 계수가 곱해진 분수 차수 ν 의 2종 변형 베셀 함수 $\exp(+|x|)K_\nu(x)$ 의 값을 계산합니다.

7.5.13 1종 베셀 함수의 근

double **gsl_sf_bessel_zero_J0**(unsigned int s)

int **gsl_sf_bessel_zero_J0_e**(unsigned int s, gsl_sf_result *result)

0 차수의 베셀 함수 $J_0(x)$ 번째, 양수 근의 위치를 찾습니다.

double **gsl_sf_bessel_zero_J1**(unsigned int s)

int **gsl_sf_bessel_zero_J1_e**(unsigned int s, gsl_sf_result *result)

1 차수의 베셀 함수 $J_1(x)$ 번째, 양수 근의 위치를 찾습니다.

double **gsl_sf_bessel_zero_Jnu**(double nu, unsigned int s)

int **gsl_sf_bessel_zero_Jnu_e**(double nu, unsigned int s, gsl_sf_result *result)

베셀 함수 $J_\nu(x)$ 번째, 양수 근의 위치를 찾습니다. 현재 구현체는 nu 가 음수일 때를 지원하지 않습니다.

7.6 클라우센 함수 (Clausen Functions)

클라우센 함수는 다음과 같은 적분으로 정의됩니다.

$$Cl_2(x) = -\int_0^x \log(2\sin(\frac{t}{2}))\,dt$$

이 함수는 다이로그 함수와 다음과 같은 관계를 가집니다.

$$Cl_2(\theta) = \Im[Li_2(e^{i\theta})]$$

클라우센 함수들은 헤더 파일 gsl_sf_clausen.h 에 정의되어 있습니다.

double **gsl_sf_clausen**(double x)

int **gsl_sf_clausen_e**(double x, gsl_sf_result *result)

이 함수들은 클라우센 적분 $Cl_2(x)$ 의 값을 계산합니다.

7.7 쿨롱 함수 (Coulomb Functions)

쿨롱 함수의 초안은 헤더 파일 gsl_sf_coulomb.h 에 정의되어 있습니다. 구속 상태와 산란해를 모두 포함합니다.

7.7.1 정규화 된 수소의 구속 상태

double **gsl_sf_hydrogenicR_1**(double Z, double r)

int **gsl_sf_hydrogenicR_1_e**(double Z, double r, gsl_sf_result *result)

가장 낮은 차수의 정규화 된 수소의 구속 방사형 상태 함수 $R_1 = 2Z\sqrt{Z}\exp(-Zr)$ 를 계산합니다.

double **gsl_sf_hydrogenicR**(int n, int l, double Z, double r)

int **gsl_sf_hydrogenicR_e**(int n, int l, double Z, double r, gsl_sf_result *result)

n 차수의 정규화 된 수소의 구속 방사형 상태 함수

$$R_n := \frac{2Z^{3/2}}{n^2}(\frac{2Zr}{n})^l \sqrt{\frac{(n-l-1)!}{n+l}} \exp(-Zr/n) L^{2l+1}_{n-l-1}(2Zr/n)$$

를 계산합니다. $L^a_b(x)$ 는 일반화 된 르장드르 다항식입니다. 이 정규화 방법은 수소의 파동 함수 ψ 가 $\psi(n,l,r) = R_n Y_{lm}$ 로 주어지도록 정해졌습니다.

7.7.2 쿨롱 파동 함수

쿨롱 파동함수 $F_L(\eta, x), G_L(\eta, x)$ 들은 Abramowitz & Stegun Chapter 14에 기술되어 있습니다. 다양하고 넓은 범위의 값들을 가지기 때문에, 오버플로우를 이용할 수 있도록 구현되었습니다. 만약, 오버플로우가 발생한다면, GSL_EOVERFLW 값을 반환하고 exponent(s) 는 수정 가능한 인자 exp_F 와 exp_G 를 반환합니다. 완전해는 다음과 같이 구할 수 있습니다.

$$F_L(\eta, x) = f_c[k_L] * \exp(exp_F)$$
$$G_L(\eta, x) = g_c[k_L] * \exp(exp_G)$$

$$F'_L(\eta, x) = f_{cp}[k_L] * \exp(exp_F)$$
$$G'_L(\eta, x) = g_{cp}[k_L] * \exp(exp_G)$$

int **gsl_sf_coulomb_wave_FG_e**(double eta, double x, double L_F, int k, gsl_sf_result *F, gsl_sf_result *Fp, gsl_sf_result *G, gsl_sf_result *Gp, double *exp_F, double *exp_G)

이 함수는 쿨롱 파동 함수 $F_L(\eta, x), G_{L-k}(\eta, x)$ 와 그 도함수 $F'_L(\eta, x), G'_{L-k}(\eta, x)$ 를 인자 x`에 대해 계산합니다. 각 계수들은 :math:`L 에 대해 다음의 제약조건을 가집니다.

$$L - k > -1/2, x > 0, k \in \mathbb{Z}$$

유의할 점은 L 이 반드시 정수로 있을 필요는 없다는 점입니다. 결과 값들은 인자 F 와 G 에 함수 값이 저장되고, 도함수의 값은 Fp 와 Gp 에 저장됩니다. 오버플로우가 발생하면, GSL_EOVERFLW 가 반환되고 조정된 지수값이 수정 가능한 인자 exp_F 와 exp_G 에 저장됩니다.

int **gsl_sf_coulomb_wave_F_array**(double L_min, int kmax, double eta, double x, double fc_array[], double *F_exponent)

이 함수는 $L = L_{min} \ldots L_{min} + k_{max}$ 에 대해, 함수 $F_L(\eta, x)$ 의 값을 계산합니다. 계산 결과값은 fc_array 배열에 저장됩니다. 오버플로우가 발생하면 지수값이 F_exponenet 에 저장됩니다.

int **gsl_sf_coulomb_wave_FG_array**(double L_min, int kmax, double eta, double x, double fc_array[], double gc_array[], double *F_exponent, double *G_exponent)

이 함수는 $L = L_{min} \ldots L_{min} + k_{max}$ 에 대해, 함수 $F_L(\eta, x), G_L(\eta, x)$ 의 값을 계산합니다. 계산 결과값은 각각 fc_array 와 gc_array 배열에 저장됩니다. 오버플로우가 발생하면 지수값이 F_exponenet 와 G_exponent 에 저장됩니다.

int **gsl_sf_coulomb_wave_FGp_array**(double L_min, int kmax, double eta, double x, double fc_array[], double fcp_array[], double gc_array[], double gcp_array[], double *F_exponent, double *G_exponent)

이 함수는 $L = L_{min} \ldots L_{min} + k_{max}$ 에 대해, 함수 $F_L(\eta, x), G_L(\eta, x)$ 와 그 도함수 $F'_L(\eta, x), G'_L(\eta, x)$ 의 값을 계산합니다. 계산 결과값은 각각 fc_array , gc_array , fcp_array 그리고 gcp_array 배열에 저장됩니다. 오버플로우가 발생하면 지수값이 F_exponenet 와 G_exponent 에 저장됩니다.

int **gsl_sf_coulomb_wave_sphF_array**(double L_min, int kmax, double eta, double x, double fc_array[], double F_exponent[])

이 함수는 $L = L_{min} \ldots L_{min} + k_{max}$ 에 대해, 인자로 나누어진 쿨롱 함수 $F_L(\eta, x)/x$ 값을 계산합니다. 계산 결과값은 fc_array 배열에 저장됩니다. 오버플로우가 발생하면 지수값이 F_exponenet 에 저장됩니다. $\eta \to 0$ 이 함수는 구면 베셀 함수로 수렴합니다.

7.7.3 쿨롱 파동함수의 정규화 계수

쿨롱 파동 함수의 정규화 상수들은 Abramowitz 14.1.7에 정의되어 있습니다.

int **gsl_sf_coulomb_CL_e**(double L, double eta, gsl_sf_result *result)

$L > -1$ 에 대해, 쿨롱 파동 함수의 정규화 계수 $C_L(\eta)$ 를 계산합니다.

int **gsl_sf_coulomb_CL_array**(double Lmin, int kmax, double eta, double cl[])

$L = L_{min} \ldots L_{min} + k_{max}, L_{min} > -1$ 에 대해, 쿨롱 파동 함수의 정규화 계수 $C_L(\eta)$ 를 계산합니다.

7.8 상호작용 계수 (Coupling Coefficients)

위그너 $3-j$, $6-j$, 그리고 $9-j$ 기호들은, 결합된 각 운동량 벡터들의 상호 작용 계수들을 나타냅니다. 표준 상호작용 계수 함수들의 인자가 정수거나 반-정수(half-integer)이기 때문에, 이를 계산하는 함수들의 인자들도 이와 같습니다. 이 정수들은 실제 스핀 값들의 두 배를 의미합니다. $3-j$ 계수들에 대한 더 자세한 정보는 Abramowitz & Stegun, Section 27.9를 참고할 수 있습니다. 이 단원의 함수들은 헤더 파일 `gsl_sf_coupling.h` 에 기술되어 있습니다.

7.8.1 3-j 기호

double **gsl_sf_coupling_3j**(int two_ja, int two_jb, int two_jc, int two_ma, int two_mb, int two_mc)

int **gsl_sf_coupling_3j_e**(int two_ja, int two_jb, int two_jc, int two_ma, int two_mb, int two_mc, gsl_sf_result *result)

위그너 $3-j$ 계수

$$\begin{pmatrix} ja & jb & jc \\ \\ ma & mb & mc \end{pmatrix}$$

를 계산합니다. 인자들은 반-정수 값을 가집니다. 예를 들어, $ja =$ `two_ja` /2 이고, $ma =$ `two_ma` /2 입니다.

7.8.2 6-j 기호(6-j Symbols)

double **gsl_sf_coupling_6j**(int two_ja, int two_jb, int two_jc, int two_jd, int two_je, int two_jf)

int **gsl_sf_coupling_6j_e**(int two_ja, int two_jb, int two_jc, int two_jd, int two_je, int two_jf, gsl_sf_result *result)

위그너 $6-j$ 계수

$$\left\{ \begin{matrix} ja & jb & jc \\ jd & je & jf \end{matrix} \right\}$$

를 계산합니다. 인자들은 반-정수 값을 가집니다. 예를 들어, $ja =$ `two_ja` /2 이고 $ma =$ `two_ma` /2 입니다.

7.8.3 9-j 기호(9-j Symbols)

double **gsl_sf_coupling_9j**(int two_ja, int two_jb, int two_jc, int two_jd, int two_je, int two_jf, int two_jg, int two_jh, int two_ji)

int **gsl_sf_coupling_9j_e**(int two_ja, int two_jb, int two_jc, int two_jd, int two_je, int two_jf, int two_jg, int two_jh, int two_ji, gsl_sf_result *result)

위그너 $9-j$ 계수

$$\left\{ \begin{matrix} ja & jb & jc \\ jd & je & jf \\ jg & jh & ji \end{matrix} \right\}$$

를 계산합니다. 인자들은 반-정수 값을 값을 가집니다. 예를 들어, $ja =$ `two_ja` /2 이고 $ma =$ `two_ma` /2 입니다.

7.9 도슨 함수 (Dawson Function)

도슨 적분은 다음의 적분을 의미합니다.

$$D_+(x) = e^{-x^2} \int_0^x e^{t^2}\, dt$$

도슨 적분표는 Abramowitz & Stegun의 표7.5에서 찾을 수 있습니다. 이 도슨 함수는 헤더 파일 `gsl_sf_dawson.h` 에 정의되어 있습니다.

double **gsl_sf_dawson**(double x)

int **gsl_sf_dawson_e**(double x, gsl_sf_result *result)

이 함수들은 주어진 값 x 대해, 도슨 적분 값을 계산합니다.

7.10 디바이 함수 (Debye Functions)

디바이 함수 $D_n(x)$ 는 다음과 같이 정의됩니다.

$$D_n(x) = \frac{n}{x^n} \int_0^x \frac{t^n}{e^t - 1}\, dt$$

더 자세한 정보는 Abramowitz & Stegun, Section 27.1을 참고할 수 있습니다. 디바이 함수는 헤더 파일 `gsl_sf_debye.h` 에 정의되어 있습니다.

double **gsl_sf_debye_1**(double x)

int **gsl_sf_debye_1_e**(double x, gsl_sf_result *result)

1 차 디바이 함수 $D_1(x)$ 의 값을 계산합니다.

double **gsl_sf_debye_2**(double x)

int **gsl_sf_debye_2_e**(double x, gsl_sf_result *result)

2 차 디바이 함수 $D_2(x)$ 의 값을 계산합니다.

double **gsl_sf_debye_3**(double x)

int **gsl_sf_debye_3_e**(double x, gsl_sf_result *result)

3 차 디바이 함수 $D_3(x)$ 의 값을 계산합니다.

double **gsl_sf_debye_4**(double x)

int **gsl_sf_debye_4_e**(double x, gsl_sf_result *result)

4 차 디바이 함수 $D_4(x)$ 의 값을 계산합니다.

double **gsl_sf_debye_5**(double x)

int **gsl_sf_debye_5_e**(double x, gsl_sf_result *result)

5 차 디바이 함수 $D_5(x)$ 의 값을 계산합니다.

double **gsl_sf_debye_6**(double x)

int **gsl_sf_debye_6_e**(double x, gsl_sf_result *result)

6 차 디바이 함수 $D_6(x)$ 의 값을 계산합니다.

7.11 다이로그 함수 (Dilogarithm)

다이 로그는 다음과 같이 정의됩니다.

$$Li_2(z) = -\int_0^z \frac{\log(1-s)}{s}\,ds$$

이 함수들은 헤더 파일 `gsl_sf_dilog.h` 에 정의되어 있습니다.

7.11.1 실수 인자 (Real Argument)

double **gsl_sf_dilog**(double x)

int **gsl_sf_dilog_e**(double x, gsl_sf_result *result)

이 함수들은 실수 값에 대한 다이 로그값을 계산합니다. 르윈의 표기법으로 $Li_2(x)$ 로 표기되며, 이는 실수 값 x 의 다이 로그 값을 나타냅니다. 이 값은 다음과 같이 적분형 표현으로 정의됩니다.

$$Li_2(x) = -\Re \int_0^x \log(1-s)/s\,ds$$

참고:

- $\Im(Li_2(x))$ 는 $x \leq 1$ 에서 0 , $x > 1$ 에서 $-\pi\log(x)$ 입니다.
- Abramowitz & Stegun에서는 스펜스 적분을 $Li_2(x)$ 대신 $S(x) = Li_2(1-x)$ 로 정의합니다.

7.11.2 복소수 인자 (Complex Argument)

int **gsl_sf_complex_dilog_e**(double r, double theta, gsl_sf_result *result_re, gsl_sf_result *result_im)

복소수 인자 $z = r\exp(i\theta)$ 에 대해, 복소수 값을 가지는 완전한 다이 로그 값을 계산합니다. 실수, 허수 값은 각각 `result_re` 와 `result_im` 에 반환됩니다.

7.12 기초 연산 (Elementary Operations)

다음 함수들은 곱셈 과정에서 오차의 전파를 같이 계산합니다. 이 함수들은 헤더 파일 `gsl_sf_elementary.h` 에 정의되어 있습니다.

double **gsl_sf_multiply**(double x, double y)

int **gsl_sf_multiply_e**(double x, double y, gsl_sf_result *result)

x 와 y 에 저장합니다.

int **gsl_sf_multiply_err_e**(double x, double dx, double y, double dy, gsl_sf_result *result)

x 와 y 의 곱셈을 절대 오차 dx 와 dy 와 함께 연산합니다. `result` 에는 $xy \pm xy\sqrt{(dx/x)^2 + (dy/y)^2}$ 가 저장됩니다.

7.13 타원 적분 (Elliptic Integrals)

이 단원에서 기술된 함수들은 헤더 파일 `gsl_sf_ellint.h` 에 정의되어 있습니다. 타원 적분에 관한 더 자세한 정보들은 Abramowitz & Stegun, Chpater 17를 참고해 볼 수 있습니다.

7.13.1 르장드르 형태 정의 (Definition of Legendre Forms)

타원 적분의 르장드르 형태 $F(\phi,k), E(\phi,k)$ 그리고 $\Pi(\phi,k,n)$ 는 다음과 같이 정의됩니다.

$$F(\phi,k) = \int_0^\phi dt \frac{1}{\sqrt{(1-k^2\sin^2(t))}}$$

$$E(\phi,k) = \int_0^\phi dt \sqrt{(1-k^2\sin^2(t))}$$

$$\Pi(\phi,k,n) = \int_0^\phi dt \frac{1}{(1+n\sin^2(t))\sqrt{1-k^2\sin^2(t)}}$$

르장드르 형태의 완전 타원 적분은 $K(k) = F(\pi/2,k)$ 와 $E(k) = E(\pi/2,k)$ 로 표기할 수 있습니다.

이 단원에서 사용하는 표기법은 Carlson, "Numerische Mathematik" 33 (1979) 1 에 기반해 있습니다. 이 표기는 Abramowitz & Stegun의 표기와 조금 다른데, 함수의 표기에서 인자가 $m = k^2$ 이고, n 이 $-n$ 으로 바뀌어 있습니다.

7.13.2 칼슨 형태 정의 (Definition of Carlson Forms)

킬슨 대칭 형태 함수 $RC(x,y), RD(x,y,z), RF(x,y,z)$ 와 $RJ(x,y,z,p)$ 는 다음과 같이 정의됩니다.

$$RC(x,y) = 1/2 \int_0^\infty dt(t+x)^{-1/2}(t+y)^{-1}$$
$$RD(x,y,z) = 3/2 \int_0^\infty dt(t+x)^{-1/2}(t+y)^{-1/2}(t+z)^{-3/2}$$
$$RF(x,y,z) = 1/2 \int_0^\infty dt(t+x)^{-1/2}(t+y)^{-1/2}(t+z)^{-1/2}$$
$$RJ(x,y,z,p) = 3/2 \int_0^\infty dt(t+x)^{-1/2}(t+y)^{-1/2}(t+z)^{-1/2}(t+p)^{-1}$$

7.13.3 르장드르 형태-완전 타원 적분 (Legendre Form of Complete Elliptic Integrals)

double **gsl_sf_ellint_Kcomp**(double k, gsl_mode_t mode)

int **gsl_sf_ellint_Kcomp_e**(double k, gsl_mode_t mode, gsl_sf_result *result)

완전 타원 적분의 르장드르 형태 $K(k)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다. Abramowitz & Stegun에서 이 함수는 $m = k^2$ 이라는 점에 유의해야 합니다.

double **gsl_sf_ellint_Ecomp**(double k, gsl_mode_t mode)

int **gsl_sf_ellint_Ecomp_e**(double k, gsl_mode_t mode, gsl_sf_result *result)

완전 타원 적분의 르장드르 형태 $E(k)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다. Abramowitz & Stegun에서 이 함수는 $m = k^2$ 이라는 점에 유의해야 합니다.

double **gsl_sf_ellint_Pcomp**(double k, double n, gsl_mode_t mode)

int **gsl_sf_ellint_Pcomp_e**(double k, double n, gsl_mode_t mode, gsl_sf_result *result)

완전 타원 적분의 르장드르 형태 $\Pi(k,n)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다. Abramowitz & Stegun에서 이 함수는 $m = k^2$ 이고 $\sin^2(\alpha) = k^2$ 이고, n 의 부호를 $-n$ 으로 바뀌었다는 점에 유의해야 합니다.

7.13.4 르장드르 형태-불완전 타원 적분 (Legendre Form of Incomplete Elliptic Integrals)

double **gsl_sf_ellint_F**(double phi, double k, gsl_mode_t mode)

int **gsl_sf_ellint_F_e**(double phi, double k, gsl_mode_t mode, gsl_sf_result *result)

타원 적분의 르장드르 형태 $K(\phi,k)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다. Abramowitz & Stegun에서 이 함수는 $m = k^2$ 이라는 점에 유의해야 합니다.

double **gsl_sf_ellint_E**(double phi, double k, gsl_mode_t mode)

int **gsl_sf_ellint_E_e**(double phi, double k, gsl_mode_t mode, gsl_sf_result *result)

완전 타원 적분의 르장드르 형태 $E(\phi, k)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다. Abramowitz & Stegun에서 이 함수는 $m = k^2$ 이라는 점에 유의해야 합니다.

double **gsl_sf_ellint_P**(double phi, double k, double n, gsl_mode_t mode)

int **gsl_sf_ellint_P_e**(double phi, double k, double n, gsl_mode_t mode, gsl_sf_result *result)

완전 타원 적분의 르장드르 형태 $\Pi(\phi, k, n)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다. Abramowitz & Stegun에서 이 함수는 $m = k^2$ 이고 $\sin^2(\alpha) = k^2$ 이고, n 의 부호를 $-n$ 으로 바뀌었다는 점에 유의해야 합니다.

double **gsl_sf_ellint_D**(double phi, double k, gsl_mode_t mode)

int **gsl_sf_ellint_D_e**(double phi, double k, gsl_mode_t mode, gsl_sf_result *result)

타원 적분 $D(\phi, k)$ 을 계산합니다. 이 함수는 칼슨 형태 타원 함수 $RD(x, y, z)$ 와 다음의 관계로 정의되어 있습니다.

$$D(\phi, k) = \frac{1}{3}(\sin\phi)^3 RD(1 - \sin^2(\phi), 1 - k^2\sin^2(\phi), 1)$$

7.13.5 칼슨 형태 (Carlson Forms)

double **gsl_sf_ellint_RC**(double x, double y, gsl_mode_t mode)

int **gsl_sf_ellint_RC_e**(double x, double y, gsl_mode_t mode, gsl_sf_result *result)

타원적분 $RC(x, y)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다.

double **gsl_sf_ellint_RD**(double x, double y, double z, gsl_mode_t mode)

int **gsl_sf_ellint_RD_e**(double x, double y, double z, gsl_mode_t mode, gsl_sf_result *result)

타원적분 $RD(x, y, z)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다.

double **gsl_sf_ellint_RF**(double x, double y, double z, gsl_mode_t mode)

int **gsl_sf_ellint_RF_e**(double x, double y, double z, gsl_mode_t mode, gsl_sf_result *result)

타원적분 $RF(x, y, z)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다.

double **gsl_sf_ellint_RJ**(double x, double y, double z, double p, gsl_mode_t mode)

int **gsl_sf_ellint_RJ_e**(double x, double y, double z, double p, gsl_mode_t mode, gsl_sf_result *result)

타원적분 $RJ(x, y, z, p)$ 를 mode 변수의 값에 따라 정확도를 결정해 계산합니다.

7.14 자코비 타원 적분 (Elliptic Functions (Jacobi))

자코비 타원 함수들은 Abramowitz & Stegun, 16 단원의 정의를 따릅니다. 이 함수들은 gsl_sf_elljac.h 에 정의되어 있습니다.

int **gsl_sf_elljac_e**(double u, double m, double *sn, double *cn, double *dn)

자코비 타원 함수 $sn(u|m), cn(u|m), dn(u|m)$ 을 내림차순 란덴 변환을 이용해 계산합니다.

7.15 오차 함수 (Error Functions)

오차 함수는 Abramowitz & Stegun, Chapter 7.에 기술되어 있습니다. 이 단원에서 기술된 함수들은 헤더파일 gs_sf_erf.h 에 기술되어 있습니다.

7.15.1 오차 함수(Error Functionn)

double **gsl_sf_erf**(double x)

int **gsl_sf_erf_e**(double x, gsl_sf_result *result)

오차함수 erf(x) 의 값을 계산합니다. erf(x) 는 다음과 같이 정의됩니다.

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x dt \exp(-t^2)$$

7.15.2 상보 오차 함수(Complementary Error Functionn)

double **gsl_sf_erfc**(double x)

int **gsl_sf_erfc_e**(double x, gsl_sf_result *result)

상보 오차 함수 erfc$(x) = 1 - \mathrm{erf}(x)$ 의 값을 계산합니다. erfc(x) 는 다음과 같이 정의됩니다.

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty dt \exp(-t^2)$$

7.15.3 로그 상보 오차 함수(Log Complementary Error Functionn)

double **gsl_sf_log_erfc**(double x)

int **gsl_sf_log_erfc_e**(double x, gsl_sf_result *result)

상보 오차 함수의 로그값 $\log(\mathrm{erfc}(x))$ 를 계산합니다.

7.15.4 확률 함수 (Probability functions)

표준/가우스 분포의 확률 함수들은 Abramowitz & Stegun, Section 26.2에 기술되어 있습니다.

double **gsl_sf_erf_Z**(double x)

int **gsl_sf_erf_Z_e**(double x, gsl_sf_result *result)

가우스 확률 밀도 함수 $Z(x) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{x^2}{2})$ 값을 계산합니다.

double **gsl_sf_erf_Q**(double x)

int **gsl_sf_erf_Q_e**(double x, gsl_sf_result *result)

가우스 확률 함수의 $Q(x) = (1/\sqrt{2\pi}) \int_x^\infty dt \exp(-t^2/2)$ 값을 계산합니다.

표준 분포의 **하자드 함수(Hazard function)** 는 Mills' 비의 역으로도 알려져 있습니다. 이는 다음과 같이 정의됩니다.

$$h(x) = \frac{Z(x)}{Q(x)} = \sqrt{\frac{2}{\pi}} \frac{\exp(-x^2/2)}{\mathrm{erfc}(x/\sqrt{2})}$$

x 가 $-\infty$ 에 가까우질 수록 급격히 감소하며, x 가 $+\infty$ 에 가까워질 수록 $h(x) \approx$ 로 점근합니다.

double **gsl_sf_hazard**(double x)

int **gsl_sf_hazard_e**(double x, gsl_sf_result *result)

표준 분포의 하자드 함수를 계산합니다.

7.16 지수 함수 (Exponential Functions)

이 단원에서 기술하는 함수들은 헤더 파일 `gsl_sf_exp.h` 에 정의되어 있습니다.

7.16.1 지수 함수 (Exponential Function)

double **gsl_sf_exp**(double x)

int **gsl_sf_exp_e**(double x, gsl_sf_result *result)

지수 함수 $\exp(x)$ 의 값을 계산합니다. GSL semantics와 오차 검사를 함께 진행합니다.

int **gsl_sf_exp_e10_e**(double x, gsl_sf_result_e10 *result)

지수 함수 $\exp(x)$ 의 값을 계산하는 데, 반환값의 자료형으로 `gsl_sf_result_e10` 을 사용해 확장된 크기의 반환값을 계산합니다. 이 함수는 *double* 자료형의 범주를 초과한 $\exp(x)$ 값을 구할 때, 사용할 수 있습니다.

double **gsl_sf_exp_mult**(double x, double y)

int **gsl_sf_exp_mult_e**(double x, double y, gsl_sf_result *result)

주어진 실수 x 지수 함수 값에 계수 y 곱한 값 $y\exp(x)$ 를 계산합니다.

int **gsl_sf_exp_mult_e10_e**(const double x, const double y, gsl_sf_result_e10 *result)

확장된 범위를 가지는 `gsl_sf_result_e10` 자료형을 반환 값에 사용해 $y\exp(x)$ 값을 계산합니다.

7.16.2 상대 지수 함수 (Relative Exponential Functions)

double **gsl_sf_expm1**(double x)

int **gsl_sf_expm1_e**(double x, gsl_sf_result *result)

$\exp(x) - 1$ 를 계산합니다. 이 함수에 사용된 알고리즘은 작은 x 에서만 정확합니다.

double **gsl_sf_exprel**(double x)

int **gsl_sf_exprel_e**(double x, gsl_sf_result *result)

작은 x 값에서 정확한 알고리즘을 이용해 $(\exp(x) - 1)/x$ 값을 계산합니다. 작은 x 값에 대해, 알고리즘은 다음과 같은 확장을 이용합니다.

$$(\exp(x) - 1)/x = 1 + x/2 + x^2/(2 \cdot 3) + x^3/(2 \cdot 3 \cdot 4) + \ldots$$

double **gsl_sf_exprel_2**(double x)

int **gsl_sf_exprel_2_e**(double x, gsl_sf_result *result)

작은 x 값에서 정확한 알고리즘을 이용해 $2(\exp(x) - 1 - x)/x^2$ 값을 계산합니다. 작은 x 값에 대해, 알고리즘은 다음과 같은 확장을 이용합니다.

$$2(\exp(x) - 1 - x)/x^2 = 1 + x/3 + x^2/(3 \cdot 4) + x^3/(3 \cdot 4 \cdot 5) + \ldots$$

double **gsl_sf_exprel_n**(int n, double x)

int **gsl_sf_exprel_n_e**(int n, double x, gsl_sf_result *result)

$N-$ 상대 지수 함수 값을 계산합니다. 이 함수는 gsl_sf_exprel() 와 gsl_sf_exprel_2() 의 n 차 일반화 함수입니다. 다음과 같이 주어집니다.

$$\mathrm{exprel}_N(x) = N!/x^N(\exp(x) - \sum_{k=0}^{N-1} x^k/k!) \tag{7.2}$$

$$= 1 + x/(N+1) + x^2/((N+1)(N+2)) + \ldots \tag{7.3}$$

$$= {}_1F_1(1, 1+N, x) \tag{7.4}$$

$$\tag{7.5}$$

7.16.3 오차 평가가 있는 지수 함수 (Exponentiation With Error Estimate)

int **gsl_sf_exp_err_e**(double x, double dx, gsl_sf_result *result)

주어진 x 의 지수함수 값을 절대 오차 dx 와 함께 반환합니다.

int **gsl_sf_exp_err_e10_e**(double x, double dx, gsl_sf_result_e10 *result)

주어진 x 의 지수함수 값을 절대 오차 dx 와 함께 반환합니다. 이때, 반환 값의 자료형을 gsl_sf_result_e10 을 사용해 확장된 범위의 결과를 계산할 수 있습니다.

int **gsl_sf_exp_mult_err_e**(double x, double dx, double y, double dy, gsl_sf_result *result)

x 와 y 대해, $y\exp(x)$ 값을 절대 오차 dx 와 dy 함께 계산합니다.

int **gsl_sf_exp_mult_err_e10_e**(double x, double dx, double y, double dy, gsl_sf_result_e10 *result)

x 와 y 대해, $y\exp(x)$ 값을 절대 오차 dx 와 dy 함께 계산합니다. 이때, 반환 값의 자료형을 gsl_sf_result_e10 을 사용해 확장된 범위의 결과를 계산할 수 있습니다.

7.17 지수 적분 함수 (Exponential Integrals)

지수 적분의 자세한 정보는 Abramowitz & Stegun, Chapter 5. 에서 찾아볼 수 있습니다. 이 함수들은 헤더 파일 gsl_sf_expint.h 정의되어 있습니다.

7.17.1 지수 적분 (Exponential Integral)

double **gsl_sf_expint_E1**(double x)

int **gsl_sf_expint_E1_e**(double x, gsl_sf_result *result)

지수 적분 함수 $E_1(x)$ 를 계산합니다.

$$E_1(x) := \Re \int_1^\infty \frac{\exp(-xt)}{t} dt$$

double **gsl_sf_expint_E2**(double x)

int **gsl_sf_expint_E2_e**(double x, gsl_sf_result *result)

2 차 지수 적분 함수 $E_2(x)$ 를 계산합니다.

$$E_2(x) := \Re \int_1^\infty \frac{\exp(-xt)}{t^2} dt$$

double **gsl_sf_expint_En**(int n, double x)

int **gsl_sf_expint_En_e**(int n, double x, gsl_sf_result *result)

n 차 지수 적분 함수 $E_2(x)$ 를 계산합니다.

$$E_n(x) := \Re \int_1^\infty \frac{\exp(-xt)}{t^n} dt$$

7.17.2 Ei(x)

double **gsl_sf_expint_Ei**(double x)

int **gsl_sf_expint_Ei_e**(double x, gsl_sf_result *result)

지수 적분 $Ei(x)$ 의 값을 계산합니다.

$$Ei(x) = -PV(\int_{-x}^\infty \frac{\exp(-t)}{t} dt)$$

PV 는 적분 주요값(Principal Value of Integral)입니다[1] .

[1] 코시 주요값(Cauchy principal value)이라고도 합니다(*).

7.17.3 초기하 적분 (Hyperbolic Integrals)

double **gsl_sf_Shi**(double x)

int **gsl_sf_Shi_e**(double x, gsl_sf_result *result)

다음의 적분값을 계산합니다.

$$\mathrm{Shi}(x) := \int_0^x \frac{\sinh(t)}{t} dt$$

double **gsl_sf_Chi**(double x)

int **gsl_sf_Chi_e**(double x, gsl_sf_result *result)

다음의 적분값을 계싼합니다.

$$\mathrm{Chi}(x) := \Re[\gamma_E + \log(x) + \int_0^x \frac{\cosh(t) - 1}{t} dt]$$

γ_E 는 오일러 상수입니다. 오일러 상수는 매크로 `M_EULER` 로 라이브러리 내에 있습니다.

7.17.4 Ei_3(x)

double **gsl_sf_expint_3**(double x)

int **gsl_sf_expint_3_e**(double x, gsl_sf_result *result)

다음의 3 차 지수 적분값을 $x \geq 0$ 에 대해 계산합니다.

$$\mathrm{Ei}_3(x) := \int_0^x \exp(-t^3) dt$$

7.17.5 삼각 적분 (Trigonometric Integrals)

double **gsl_sf_Si**(const double x)

int **gsl_sf_Si_e**(double x, gsl_sf_result *result)

다음의 적분값을 계산합니다.

$$\mathrm{Si}(x) := \int_0^x \frac{\sin(t)}{t} dt$$

double **gsl_sf_Ci**(const double x)

int **gsl_sf_Ci_e**(double x, gsl_sf_result *result)

다음의 적분값을 $x \geq 0$ 에 대해 계산합니다.

$$\mathrm{Ci}(x) := -\int_0^x \frac{\cos(t)}{t} dt$$

7.17.6 역탄젠트 적분 (Arctangent Integral)

double **gsl_sf_atanint**(double x)

int **gsl_sf_atanint_e**(double x, gsl_sf_result *result)

다음의 적분값을 계산합니다.

$$\mathrm{AtanInt}(x) := \int_0^x \frac{\arctan}{t} dt$$

7.18 페르미 디렉 함수 (Fermi-Dirac Function)

헤더 파일 `gsl_sf_fermi_dirac.h` 에 정의되어 있습니다.

7.18.1 완비 페르미-디렉 적분 (Complete Fermi-Dirac Integrals)

완비 페르미 디렉 적분 $F_j(x)$ 는 다음과 같이 정의됩니다.

$$F_j(x) := \frac{1}{\Gamma(j+1)} \int_0^\infty \frac{t^j}{(\exp(t-x)+1)} dt$$

다른 문헌에서 정규화 계수 없이 표현되기도 합니다.

double **gsl_sf_fermi_dirac_m1**(double x)

int **gsl_sf_fermi_dirac_m1_e**(double x, gsl_sf_result *result)

$j = -1$ 인 완비 페르미 디렉 적분 값을 계산합니다. 이 값은 $F_{-1}(x) = e^x/(1+e^x)$ 로 주어집니다.

double **gsl_sf_fermi_dirac_0**(double x)

int **gsl_sf_fermi_dirac_0_e**(double x, gsl_sf_result *result)

$j = 0$ 인 완비 페르미 디렉 적분 값을 계산합니다. 이 값은 $F_0(x) = \ln(1+e^x)$ 로 주어집니다.

double **gsl_sf_fermi_dirac_1**(double x)

int **gsl_sf_fermi_dirac_1_e**(double x, gsl_sf_result *result)

$j = -1$ 인 완비 페르미 디렉 적분 값을 계산합니다. 이 값은 $F_1(x) = \int_0^\infty (t/(\exp(t-x)+1) dt$ 로 주어집니다.

double **gsl_sf_fermi_dirac_2**(double x)

int **gsl_sf_fermi_dirac_2_e**(double x, gsl_sf_result *result)

$j = 2$ 인 완비 페르미 디렉 적분 값을 계산합니다. 이 값은 $F_2(x) = (1/2) \int_0^\infty (t^2/(\exp(t-x)+1) dt$ 로 주어집니다.

double **gsl_sf_fermi_dirac_int**(int j, double x)

int **gsl_sf_fermi_dirac_int_e**(int j, double x, gsl_sf_result *result)

일반 완비 페르미 디렉 적분 값을 계산합니다. 이 값은 $F_j(x) = ((1/\Gamma(j+1)) \int_0^\infty (t^j/(\exp(t-x)+1) dt$ 로 주어집니다.

double **gsl_sf_fermi_dirac_mhalf**(double x)

int **gsl_sf_fermi_dirac_mhalf_e**(double x, gsl_sf_result *result)

완비 페르미-디렉 적분 $F_{-1/2}(x)$ 값을 계산합니다.

double **gsl_sf_fermi_dirac_half**(double x)

int **gsl_sf_fermi_dirac_half_e**(double x, gsl_sf_result *result)

완비 페르미-디렉 적분 $F_{1/2}(x)$ 값을 계산합니다.

double **gsl_sf_fermi_dirac_3half**(double x)

int **gsl_sf_fermi_dirac_3half_e**(double x, gsl_sf_result *result)

완비 페르미-디렉 적분 $F_{3/2}(x)$ 값을 계산합니다.

7.18.2 비완비 페르미-디렉 적분 (Incomplete Fermi-Dirac Integrals)

비완비 페르미-디렉 함수 $F_j(x, b)$ 는 다음과 같이 정의됩니다.

$$F_j(x,b) := \frac{1}{\Gamma(j+1)} \int_b^\infty \frac{t^j}{\exp(t-x)+1} dt$$

double **gsl_sf_fermi_dirac_inc_0**(double x, double b)

int **gsl_sf_fermi_dirac_inc_0_e**(double x, double b, gsl_sf_result *result)

0 차수의 비완비 페르미-디렉 적분, $F_0(x,b) = \ln(1 + e^{b-x}) - (b-x)$ 값을 계산합니다.

7.19 감마와 베타 함수 (Gamma and Beta Functions)

다음 함수들은 완전/불완전 감마, 베타 함수를 다양한 팩토리얼에 대해 계산합니다. 이 단원의 함수들은 헤더 파일 `gsl_sf_gamma.h` 에 정의되어 있습니다.

7.19.1 감마 함수 (Gamma Functions)

감마 함수는 다음의 적분으로 정의되어 있습니다.

$$\Gamma(x) = \int_0^\infty t^{x-1} \exp(-t) dt$$

양의 정수 n 에 대해, 팩토리얼과 $\Gamma(n) = (n-1)!$ 의 관계를 가집니다. 감마 함수에 관한 더 자세한 설명은 Abramowitz & Stegun, Chapter 6를 참고할 수 있습니다.

double **gsl_sf_gamma**(double x)

int **gsl_sf_gamma_e**(double x, gsl_sf_result *result)

0 이나 음의 정수가 아닌 x 에 대해, 감마 함수 $\Gamma(x)$ 의 값을 계산합니다. 실수 Lanczos 방법을 사용합니다. x 의 최댓값은 $\Gamma(x)$ 가 오버 플로우 되지 않는 범위로 매크로 GSL_SF_GAMMA_XMAX 로 주어져 있습니다. 이 값은 171.0 입니다.

double **gsl_sf_lngamma**(double x)

int **gsl_sf_lngamma_e**(double x, gsl_sf_result *result)

0 이나 음의 정수가 아닌 x 에 대해, 로그 감마 함수 $\log(\Gamma(x))$ 의 값을 계산합니다. $x < 0$ 에 대해, $\log(\Gamma(x))$ 값이 반환되며, 이 값은 $\log(|\Gamma(x)|)$ 의 값과 같습니다. 실수 Lanczos 방법을 사용합니다.

int **gsl_sf_lngamma_sgn_e**(double x, gsl_sf_result *result_lg, double *sgn)

0 이나 음의 정수가 아닌 x 에 대해, 감마 함수의 부호와 그 크기의 로그 값을 계산합니다. 실수 Lanczos 방법을 사용합니다. 감마 함수의 값과 그 오차는 result_lg 두 요소와 다음의 관계를 사용해 계산합니다. $\Gamma(x) = sgn * \exp(result_{lg})$.

double **gsl_sf_gammastar**(double x)

int **gsl_sf_gammastar_e**(double x, gsl_sf_result *result)

regulated 감마 함수 $\Gamma^*(x)$ 를 $x > 0$ 에 대해 계산합니다. 다음과 같이 주어집니다.

$$\begin{aligned}\Gamma^*(x) &= \Gamma(x)/(\sqrt{2\pi} x^{(x-1/2)} \exp(-x)) \\ &= (1 + \frac{1}{12x} + \dots) \text{ for } x \to \infty\end{aligned}$$

double **gsl_sf_gammainv**(double x)

int **gsl_sf_gammainv_e**(double x, gsl_sf_result *result)

실수 Lanczos 방법을 사용해 감마 함수의 역수, $1/\Gamma(x)$ 값을 계산합니다.

int **gsl_sf_lngamma_complex_e**(double zr, double zi, gsl_sf_result *lnr, gsl_sf_result *arg)

복소수 Lanczos 방법을 사용해, 주어진 복소수 $z = z_r + iz_i$ 에 대해, $\log(\Gamma(z))$ 값을 계산합니다. 이때, z 는 0 과 음의 정수가 아닌 복소수입니다. lnr :math:` = log|Gamma(z)|` , arg :math:` = text{arg}(Gamma(z))` 이고, arg 는 $(-\pi, \pi]$ 범위의 값을 가집니다. $|z|$ 가 매우 클 때, arg 값이 정확히 정해지지 않을 수 있습니다. 이는 $(-\pi, \pi]$ 로 범위를 제약함으로써 생기는 절단점으로 인해

필연적으로 발생합니다. 이러한 상황은 GSL_ELOSS 오류에 속하는 상황입니다. 절대값 lnr 부분은 이러한 정밀도 저하를 격지 않습니다.

7.19.2 팩토리얼

양의 정수 n 에 대해, 감마 함수를 이용해 $n! = \Gamma(n+1)$ 팩토리얼을 계산할 수 있습니다. 하지만, 아래의 함수들을 이용하는 것이 작은 n 값들에 대해 더 효율적입니다. 팩토리얼 값들을 하드 코딩된 테이블에 보관하고 있습니다.

double **gsl_sf_fact**(unsigned int n)

int **gsl_sf_fact_e**(unsigned int n, gsl_sf_result *result)

팩토리얼 $n!$ 를 계산합니다. 팩토리얼은 감마 함수와 $n! = \Gamma(n+1)$ 의 관계를 가지고 있습니다. n 의 최댓값은 $n!$ 이 오버플로우되지 않는 값으로 정해집니다. 이는 매크로 GSL_SF_FACT_NMAX 에 정의되어 있고 170 입니다.

double **gsl_sf_doublefact**(unsigned int n)

int **gsl_sf_doublefact_e**(unsigned int n, gsl_sf_result *result)

더블 팩토리얼 $n!! = n(n-2)(n-4)\dots$ 을 계산합니다. n 의 최댓값은 $n!!$ 이 오버플로 되지 않는 값으로 정해집니다. 이는 매크로 GSL_SF_DOUBLEFACT_NMAX 에 정의되어 있고 297 입니다.

double **gsl_sf_lnfact**(unsigned int n)

int **gsl_sf_lnfact_e**(unsigned int n, gsl_sf_result *result)

n 팩토리얼의 로그 값, $\log(n!)$ 값을 계산합니다. 이 알고리즘은 $n < 170$ 에서 $\ln(\Gamma(n+1))$ 값을 계산하는 gsl_sf_lngamma 보다 빠릅니다. 하지만 큰 n 에 대해서는 빠르지 않습니다.

double **gsl_sf_lndoublefact**(unsigned int n)

int **gsl_sf_lndoublefact_e**(unsigned int n, gsl_sf_result *result)

n 대해, 더블 팩토리얼의 로그 값 $\log(n!!)$ 을 계산합니다.

double **gsl_sf_choose**(unsigned int n, unsigned int m)

int **gsl_sf_choose_e**(unsigned int n, unsigned int m, gsl_sf_result *result)

조합 계수 n choose m $= n!/(m!(n-m!))$ 의 값을 계산합니다.

double **gsl_sf_lnchoose**(unsigned int n, unsigned int m)

int **gsl_sf_lnchoose_e**(unsigned int n, unsigned int m, gsl_sf_result *result)

조합 계수 $nchoosem$ 의 로그 값을 계산합니다. 이 값은 $\log(n!) - \log(m!) - \log((n-m)!)$ 과 같습니다.

double **gsl_sf_taylorcoeff**(int n, double x)

int **gsl_sf_taylorcoeff_e**(int n, double x, gsl_sf_result *result)

$x \geq 0$, $n \geq 0$ 에 대해, 테일러 계수 $x^n/n!$ 값을 계산합니다.

7.19.3 포흐하머 기호

double **gsl_sf_poch**(double a, double x)

int **gsl_sf_poch_e**(double a, double x, gsl_sf_result *result)

포흐하머 기호 $(a)_x = \Gamma(a_x)/\Gamma(a)$ 를 계산합니다. 포흐하머 기호는 아펠 기호로도 알려져있으며, (a, x) 로 표기하기도 합니다. a 와 $a + x$ 가 음의 정수나 0 일때, 해당 비의 극한 값이 반환됩니다.

double **gsl_sf_lnpoch**(double a, double x)

int **gsl_sf_lnpoch_e**(double a, double x, gsl_sf_result *result)

포흐하머 기호의 로그값 $\log((a)_x) = \log(\Gamma(a + x)/\Gamma(a))$ 을 계산합니다.

int **gsl_sf_lnpoch_sgn_e**(double a, double x, gsl_sf_result *result, double *sgn)

포흐하머 기호의 부호와 그 크기의 로그값을 계산합니다. 계산되는 계수들은 `result` $= \log(|(a)_x|)$ 가 오차 값과 함께 계산되고, $(a)_x = \Gamma(a + x)/\Gamma(a)$ 에 대해, `sgn` $= \mathrm{sgn}((a)_x)$ 을 계산합니다.

double **gsl_sf_pochrel**(double a, double x)

int **gsl_sf_pochrel_e**(double a, double x, gsl_sf_result *result)

$(a)_x = \Gamma(a + x)/\Gamma(a)$ 에 대해, $((a)_x - 1)/x$ 값을 계산합니다.

7.19.4 불완전 감마 함수

double **gsl_sf_gamma_inc**(double a, double x)

int **gsl_sf_gamma_inc_e**(double a, double x, gsl_sf_result *result)

실수 a 와 $x \geq 0$ 에 대해, 비정규화된 불완전 감마 함수 $\Gamma(a, x) = \int_x^\infty t^{(a-1)} \exp(-t)dt$ 값을 계산합니다.

double **gsl_sf_gamma_inc_Q**(double a, double x)

int **gsl_sf_gamma_inc_Q_e**(double a, double x, gsl_sf_result *result)

$a > 0$ 과 $x \leq 0$ 에 대해, 정규화된 불완전 감마 함수 $Q(a, x) = 1.\Gamma(a) \int_x^\infty t^{(a-1)} \exp(-t)dt$ 의 값을 계산합니다.

double **gsl_sf_gamma_inc_P**(double a, double x)

int **gsl_sf_gamma_inc_P_e**(double a, double x, gsl_sf_result *result)

$a > 0$ 과 $x \geq 0$ 에 대해, $P(a, x) = 1 - Q(a, x) = 1/\Gamma(a) \int_0^x t^{(a-1)} \exp(-t)dt$ 값을 계산합니다.

Abramowtz & Stegun의 6.5단원, 불완전 감마 함수에서 $P(a, x)$ 표기를 씁니다.

7.19.5 베타 함수

double **gsl_sf_beta**(double a, double b)

int **gsl_sf_beta_e**(double a, double b, gsl_sf_result *result)

베타함수 $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$ 값을 계산합니다. a, b 는 음의 정수가 아니여야 합니다.

double **gsl_sf_lnbeta**(double a, double b)

int **gsl_sf_lnbeta_e**(double a, double b, gsl_sf_result *result)

베타 함수의 로그 값 $\log(B(a,b))$ 를 계산합니다. a, b 는 음의 정수가 아니여야 합니다.

7.19.6 불완전 베타 함수

double **gsl_sf_beta_inc**(double a, double b, double x)

int **gsl_sf_beta_inc_e**(double a, double b, double x, gsl_sf_result *result)

정규화된 불완전 베타함수 $I_x(a,b) = B_x(a,b)/B(a,b)$ 를 계산합니다. $B_x(a,b)$ 는 $0 \leq x \leq 1$ 에 대해 다음과 같이 정해집니다.

$$B_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1} dt$$

이 값은 $a > 0, b > 0$ 에 대해, 연속 분수 전개를 이용해 계산됩니다. 다른 경우에는 다음의 관계를 이용해 계산합니다.

$$I_x(a,b,x) = (\frac{1}{a})x^a \frac{{}_2F_1(a, 1-b, a+1, x)}{B(a,b)}$$

7.20 구겐바우어 함수 (Gegenbauer Functions)

구겐바우어 다항식은 Abramowitz & Stgun의 22단원에 정의되어 있습니다. 이 다항식은 또 Ultraspherical 다항식으로도 알려져있습니다. 이 함수들은 헤더 파일 `gsl_sf_gegenbauer.h` 에 정의되어 있습니다.

double **gsl_sf_gegenpoly_1**(double lambda, double x)

double **gsl_sf_gegenpoly_2**(double lambda, double x)

double **gsl_sf_gegenpoly_3**(double lambda, double x)

int **gsl_sf_gegenpoly_1_e**(double lambda, double x, gsl_sf_result *result)

int **gsl_sf_gegenpoly_2_e**(double lambda, double x, gsl_sf_result *result)

int **gsl_sf_gegenpoly_3_e**(double lambda, double x, gsl_sf_result *result)

구겐바우어 다항식 $C_n^{(\lambda)}(x)$ 을 $n = 1, 2, 3$ 인 경우에 대해, 정의식을 이용해 계산합니다.

double **gsl_sf_gegenpoly_n**(int n, double lambda, double x)

int **gsl_sf_gegenpoly_n_e**(int n, double lambda, double x, gsl_sf_result *result)

구겐바우어 다항식 $C_n^{(\lambda)}(x)$ 을 주어진 n , lambda , 그리고 x 에 대해 계산합니다. 이때, $\lambda > -\frac{1}{2}.n \geq 0$ 이어야 합니다.

int **gsl_sf_gegenpoly_array**(int nmax, double lambda, double x, double result_array[])

구겐바우어 다항식 $C_n^{(\lambda)}(x)$ 배열 값을 계산합니다. $n = 0, 1, 2, \ldots, nmax$ 의 값을 계산하며, $\lambda > -\frac{1}{2}.nmax \geq 0$ 의 제약을 가집니다.

7.21 에르미트 다항식과 함수 (Hermite Polynomials and Functions)

에르미트 다항식과 함수는 Abramowitz & Stegunm Chapter 22 와 Szego, Gabor (1939, 1957, 1967) Orthogonal Polynomials, American Mathematical Society에 기술되어 있습니다. 본 단원의 함수들은 헤더 파일 gsl_sf_hermite.h 에 정의되어 있습니다.

7.21.1 에르미트 다항식(Hermite Polynomials)

에르미트 다항식은 두 가지 형태가 존재합니다. $H_n(x)$ 는 물리학에서 사용하는 형태이고, $H_{e_n}(x)$ 는 확률론에서 사용하는 형태입니다.

$$H_n(x) = (-1)^n e^{x^2} (\frac{d}{dx})^n e^{-x^2}$$
$$H_{e_n}(x) = (-1)^n e^{x^2/2} (\frac{d}{dx})^n e^{-x^2/2}$$

이 둘은 다음의 관계를 가지고,

$$H_n(x) = 2^{\frac{n}{2}} H_{e_n}(\sqrt{2}x)$$
$$H_{e_n}(x) = 2^{-\frac{n}{2}} H_n(\frac{x}{\sqrt{2}})$$

다음과 같은 미분 방정식을 만족합니다.

$$H_n''(x) - 2xH_n'(x) + 2nH_n(x) = 0$$
$$H_{e_n}''(x) - xH_{e_n}'(x) + nH_{e_n}(x) = 0$$

double **gsl_sf_hermite**(const int n, const double x)

int **gsl_sf_hermite_e**(const int n, const double x, gsl_sf_result *result)

이 함수는 :math"H_n (x) 형태의 에르미트 다항식을 주어진 차수 n 과 변수 x 에 대해 계산합니다. 오버플로우가 감지되면, 오류 관리자를 호출하지 않고 GSL_EOVERFLW 를 반환합니다.

int **gsl_sf_hermite_array**(const int nmax, const double x, double *result_array)

$nmax$ 이하의 차수를 가지는 모든 $H_n(x)$ 형태의 에르미트 다항식을 주어진 변수 x 에 대해 계산합니다. 결과는 result_array 에 저장됩니다.

double **gsl_sf_hermite_series**(const int n, const double x, const double *a)

int **gsl_sf_hermite_series_e**(const int n, const double x, const double *a, gsl_sf_result *result)

$\sum_{j=0}^{n} a_j H_j(x)$ 급수 값을 Clenshaw 알고리즘을 이용해 계산합니다.

double **gsl_sf_hermite_prob**(const int n, const double x)

int **gsl_sf_hermite_prob_e**(const int n, const double x, gsl_sf_result *result)

$H_{e_n}(x)$ 형태의 에르미트 다항식을 주어진 차수 n 변수 x 대해 계산합니다. 오버플로우가 감지되면, 오류 관리자를 호출하지 않고 GSL_EOVERFLW 를 반환합니다.

int **gsl_sf_hermite_prob_array**(const int nmax, const double x, double *result_array)

$nmax$ 이하의 차수를 가지는 모든 $H_{e_n}(x)$ 형태의 에르미트 다항식을 주어진 변수 x 에 대해 계산합니다. 결과는 result_array 에 저장됩니다.

double **gsl_sf_hermite_prob_series**(const int n, const double x, const double *a)

int **gsl_sf_hermite_prob_series_e**(const int n, const double x, const double *a, gsl_sf_result *result)

$\sum_{j=0}^{n} a_j H_{e_j}(x)$ 급수 값을 Clenshaw 알고리즘을 이용해 계산합니다.

7.21.2 에르미트 다항식의 도함수 (Derivatives of Hermite Polynomials)

double **gsl_sf_hermite_deriv**(const int m, const int n, const double x)

int **gsl_sf_hermite_deriv_e**(const int m, const int n, const double x, gsl_sf_result *result)

n 차수의 에르미트 다항식 $H_n(x)$ 의 m 차 도함수 값을 주어진 변수 x 대해 계산합니다.

int **gsl_sf_hermite_array_deriv**(const int m, const int nmax, const double x, double *result_array)

$0, \ldots, nmax$ 차수의 모든 에르미트 다항식 $H_n(x)$ 의 m 차 도함수 값을 주어진 변수 x 대해 계산합니다. $d^m/dx^m H_n(x)$ 의 값은 result_array[n] 에 저장됩니다. 계산 결과가 저장되는 result_array 는 최소 $nmax + 1$ 이상의 길이를 가져야 합니다.

int **gsl_sf_hermite_deriv_array**(const int mmax, const int n, const double x, double *result_array)

n 차수를 가지는 에르미트 다항식 $H_n(x)$ 의 모든 $0, \ldots, mmax$ 차 도함수 값을 주어진 변수 x 대해 계산합니다. $d^m/dx^m H_n(x)$ 의 값은 result_array[m] 에 저장됩니다. 계산 결과가 저장되는 result_array 는 최소 mmax+1 이상의 길이를 가져야 합니다.

double **gsl_sf_hermite_prob_deriv**(const int m, const int n, const double x)

int **gsl_sf_hermite_prob_deriv_e**(const int m, const int n, const double x, gsl_sf_result *result)

n 차수의 에르미트 다항식 $H_{e_n}(x)$ 의 m 차 도함수 값을 주어진 변수 x 대해 계산합니다.

int **gsl_sf_hermite_prob_array_deriv**(const int m, const int nmax, const double x, double *result_array)

n 차수를 가지는 에르미트 다항식 $H_{e_n}(x)$ 의 모든 $0, \ldots, mmax$ 차 도함수 값을 주어진 변수 x 대해 계산합니다. $d^m/dx^m H_{e_n}(x)$ 의 값은 result_array[m] 에 저장됩니다. 계산 결과가 저장되는 result_array 는 최소 mmax+1 이상의 길이를 가져야 합니다.

int **gsl_sf_hermite_prob_deriv_array**(const int mmax, const int n, const double x, double *result_array)

n 차수를 가지는 에르미트 다항식 $H_{e_n}(x)$ 의 모든 $0, \ldots, mmax$ 차 도함수 값을 주어진 변수 x 대해 계산합니다. $d^m/dx^m H_{e_n}(x)$ 의 값은 result_array[m] 저장됩니다. 계산 결과가 저장되는 result_array 는 최소 mmax+1 이상의 길이를 가져야 합니다.

7.21.3 에르미트 함수 (Hermite Functions)

에르미트 함수는 다음과 같이 정의됩니다.

$$\psi_n(x) = \frac{1}{(2^n n! \sqrt{\pi})^{\frac{1}{2}}} e^{-\frac{x^2}{2}} H_n(x)$$

그리고 이는 양자 역학에 나오는 슈뢰딩거 방정식의 조화 진동자 형태를 만족합니다.

$$\psi_n''(x) + (2n + 1 - x^2)\psi_n(x) = 0$$

이 들은 서로 직교하는 함수고

$$\int_{-\infty}^{\infty} \psi_m(x)\psi_n(x)\, dx = \delta_{mn}$$

$L^2(\mathbb{R})$ 공간의 직교 기저를 형성합니다. 에르미트 함수들은 연속 푸리에 변환의 고유 함수이기도 합니다. GSL은 에르미트 함수를 계산하는 두 가지 방법을 제공합니다. 첫 번째는 수학적으로 정의된 3 개 항의 재귀 관계를 이용합니다. 이 방법은 $O(n)$ 의 계산 복잡도를 가지고 가장 정확합니다. 두 번째는 코시 적분 접근 방법을 이용한 방법입니다. 이는 (Bunck, 2009)에 소개 되었으며, $O(\sqrt{n})$ 의 계산복잡도를 가집니다. 정확도를 조금 희생하지만 n 값이 클 수록, 기존 방법에 비해 속도에 큰 이점이 있습니다.

double **gsl_sf_hermite_func**(const int n, const double x)

int **gsl_sf_hermite_func_e**(const int n, const double x, gsl_sf_result *result)

차수 n 에르미트 함수 $\psi_n(x)$ 를 주어진 변수 x 에 대해 계산합니다. 이 방법은 재귀 관계를 이용하며, $O(n)$ 의 계산 복잡도를 가집니다.

double **gsl_sf_hermite_func_fast**(const int n, const double x)

int **gsl_sf_hermite_func_fast_e**(const int n, const double x, gsl_sf_result *result)

차수 n 에르미트 함수 $\psi_n(x)$ 를 주어진 변수 x 에 대해 계산합니다. 이 방법은 (Bunck, 2009)의 코시 적분을 이용하며, $O(\sqrt{n})$ 의 계산 복잡도를 가집니다.

int **gsl_sf_hermite_func_array**(const int nmax, const double x, double *result_array)

$n = 0, \ldots, nmax$ 의 차수를 가지는 에르미트 함수 $\psi_n(x)$ 를 주어진 변수 x 대해, 재귀적 방법을 이용해 계산합니다. 계산 결과는 result_array 저장되며 최소 $nmax + 1$ 이상의 길이를 가져야 합니다.

double **gsl_sf_hermite_func_series**(const int n, const double x, const double *a)

int **gsl_sf_hermite_func_series_e**(const int n, const double x, const double *a, gsl_sf_result *result)

$\sum_{j=0}^{n} a_j \psi_j(x)$ 급수를 계산합니다. ψ_j 는 j 의 차수를 가지는 에르미트 함수를 의미하며, Clenshaw 알고리즘을 이용합니다.

7.21.4 에르미트 함수의 도함수 (Derivatives of Hermite Functions)

double **gsl_sf_hermite_func_der**(const int m, const int n, const double x)

int **gsl_sf_hermite_func_der_e**(const int m, const int n, const double x, gsl_sf_result *result)

n 차수의 에르미트 함수 $\psi_n(x)$ 의 m 차 도함수를 주어진 x 대해 계산합니다.

7.21.5 에르미트 함수와 다항식의 근 (Zeros of Hermite Polynomials and Hermite Functions)

이 함수들은 차수 n 을 가지는 에르미트 함수와 다항식의 s 번째 근을 계산합니다. 각 근들이 원점을 기준으로 대칭이기 때문에, 양수인 근들만 계산됩니다. 인덱스는 1 부터 시작해서 오름차순으로 배열됩니다. 홀수 차수의 다항식 만이 0 에서 0 번째 근을 가집니다. 해당 값은 항상 0 입니다.

double **gsl_sf_hermite_zero**(const int n, const int s)

int **gsl_sf_hermite_zero_e**(const int n, const int s, gsl_sf_result *result)

n 차수의 에르미트 다항식 $H_n(x)$ 의 s 번째 근을 계산합니다.

double **gsl_sf_hermite_prob_zero**(const int n, const int s)

int **gsl_sf_hermite_prob_zero_e**(const int n, const int s, gsl_sf_result *result)

n 차수의 에르미트 다항식 $H_{e_n}(x)$ 의 s 번째 근을 계산합니다.

double **gsl_sf_hermite_func_zero**(const int n, const int s)

int **gsl_sf_hermite_func_zero_e**(const int n, const int s, gsl_sf_result *result)

n 차수의 에르미트 함수 $\psi_n(x)$ 의 s 번째 근을 계산합니다.

7.22 초기하 함수 (Hypergeometric Functions)

초기하 함수들은 Abramowitz& Stegun의 13, 15 단원의 기술을 기반으로 작성되었습니다. 헤더 파일 gsl_sf_hyperg.h 에 정의 되어있습니다,

double **gsl_sf_hyperg_0F1**(double c, double x)

int **gsl_sf_hyperg_0F1_e**(double c, double x, gsl_sf_result *result)

초기하 함수

$$ {}_0F_1(c,x) $$

를 계산합니다.

double **gsl_sf_hyperg_1F1_int**(int m, int n, double x)

int **gsl_sf_hyperg_1F1_int_e**(int m, int n, double x, gsl_sf_result *result)

합류 초기하 함수(confluent hypergeometric)

$$ {}_1F_1(m,n,x) = M(m,n,x) $$

를 정수 인자 m 과 n 에 따라 계산합니다.

double **gsl_sf_hyperg_1F1**(double a, double b, double x)

int **gsl_sf_hyperg_1F1_e**(double a, double b, double x, gsl_sf_result *result)

합류 초기하 함수

$$ {}_1F_1(a,b,x) = M(a,b,x) $$

를 일반 인자 a b 따라 계산합니다.

double **gsl_sf_hyperg_U_int**(int m, int n, double x)

int **gsl_sf_hyperg_U_int_e**(int m, int n, double x, gsl_sf_result *result)

합류 초기하 함수 $U(m,n,x)$ 를 정수 인자 m 과 n 에 대해 계산합니다.

int **gsl_sf_hyperg_U_int_e10_e**(int m, int n, double x, gsl_sf_result_e10 *result)

합류 초기하 함수 $U(m,n,x)$ 를 정수 인자 m 과 n 에 대해 계산하고 확장된 구간에 대해, gsl_sf_result_e10 형의 값을 반환합니다.

double **gsl_sf_hyperg_U**(double a, double b, double x)

int **gsl_sf_hyperg_U_e**(double a, double b, double x, gsl_sf_result *result)

합류 초기하 함수 $U(a,b,x)$ 를 계산합니다.

int **gsl_sf_hyperg_U_e10_e**(double a, double b, double x, gsl_sf_result_e10 *result)

합류 초기하 함수 $U(a,b,x)$ 를 계산하고 확장된 구간에 대해 gsl_sf_result_e10 형의 값을 반환합니다.

double **gsl_sf_hyperg_2F1**(double a, double b, double c, double x)

int **gsl_sf_hyperg_2F1_e**(double a, double b, double c, double x, gsl_sf_result *result)

구간 $\|x\| < 1$ 에 대해, 가우스 초기하 함수

$$ {}_2F_1(a,b,c,x) = F(a,b,c,x) $$

의 값을 계산합니다. 만약, 인자 (a,b,c,x) 가 특이점(singular point)에 너무 가깝다면, 급수 근사가 너무 느려지게 되고 함수는 오류 값 GSL_EMAXITER 반환합니다. 이러한 지점은 $x = 1$, $c - a - b = m, m \in \mathbf{Z}$ 구간에서 발생합니다.

double **gsl_sf_hyperg_2F1_conj**(double aR, double aI, double c, double x)

int **gsl_sf_hyperg_2F1_conj_e**(double aR, double aI, double c, double x, gsl_sf_result *result)

구간 $\|x\| < 1$ 에 대해, 가우스 초기하 함수

$$ {}_2F_1(a_R + ia_I, aR - iaI, c, x) $$

의 복소수 인자 값을 계산합니다.

double **gsl_sf_hyperg_2F1_renorm**(double a, double b, double c, double x)

int **gsl_sf_hyperg_2F1_renorm_e**(double a, double b, double c, double x, gsl_sf_result *result)

구간 $\|x\| < 1$ 에 대해, 재규격화 된 가우스 초기하 함수

$$ \frac{{}_2F_1(a,b,c,x)}{\Gamma(c)} $$

의 값을 계산합니다.

double **gsl_sf_hyperg_2F1_conj_renorm**(double aR, double aI, double c, double x)

int **gsl_sf_hyperg_2F1_conj_renorm_e**(double aR, double aI, double c, double x, gsl_sf_result *result)

구간 $\|x\| < 1$ 에 대해, 재규격화 된 가우스 초기하 함수

$$ \frac{{}_2F_1(a_R + ia_I, a_R - ia_I, c, x)}{\Gamma(c)} $$

의 값을 계산합니다.

double **gsl_sf_hyperg_2F0**(double a, double b, double x)

int **gsl_sf_hyperg_2F0_e**(double a, double b, double x, gsl_sf_result *result)

초기하 함수

$$ {}_2F_0(a,b,x) $$

를 계산합니다.

급수 표현은 발산하는 초기하 급수입니다. 하지만, $x < 0$ 이라면 다음을 얻을 수 있습니다.

$$ {}_2F_0(a,b,x) = (-\frac{1}{x})^a U(a, 1+a, -b, -\frac{1}{x}) $$

7.23 라게르 함수 (Laguerre Functions)

일반화 된 라게르 다항식(다른 이름으로 버금 라게르 다항식이 있습니다)은 합류 초기하 함수(confluent hypergeometric function)로 정의됩니다.

$$ L_n^a(x) = \frac{(a+1)_n}{n!} {}_1F_1(-n, a+1, x) $$

$(a)_n$ 는 포흐하머 기호(Pochhammer symbol)입니다. 이들은 일반적인 라게르 다항식 $L_n(x)$ 과 다음의 관계를 가집니다.

$$ L_n^0(x) = L_n(x) $$
$$ L_n^k(x) = (-1)^k (d^k/dx^k) L_{(n+k)}(x) $$

더 자세한 정보는 Abramowitz & Strgun, Chapter 22를 참고할 수 있습니다.

이 단원에서 기술된 함수들은 헤더 파일 `gsl_sf_laguerre.h` 에 정의되어 있습니다.

double **gsl_sf_laguerre_1**(double a, double x)

double **gsl_sf_laguerre_3**(double a, double x)

double **gsl_sf_laguerre_2**(double a, double x)

int **gsl_sf_laguerre_1_e**(double a, double x, gsl_sf_result *result)

int **gsl_sf_laguerre_2_e**(double a, double x, gsl_sf_result *result)

int **gsl_sf_laguerre_3_e**(double a, double x, gsl_sf_result *result)

일반화된 라게르 다항식 $L_1^a(x), L_2^a(x), L_3^a(x)$ 을 수학 정의식을 이용해 계산합니다.

double **gsl_sf_laguerre_n**(const int n, const double a, const double x)

일반화된 라게르 다항식 $L_n^a(x)$ 를 $a > -1, n \geq 0$ 인 경우를 계산합니다.

7.24 람베르트 W 함수 (Lambert W Functions)

람베르트 W 함수 $W(x)$ 는 $W(x)\exp(W(x)) = x$ 방정식의 해로 정의됩니다. 이 함수는 $x < 0$ 에서 다양한 부분 함수들로 나뉘어 집니다. 하지만, 2 개의 실수 함수들이 존재합니다. 일반적으로 $W_0(x)$ 를 주 함수로 사용합니다. 이 함수는 $x < 0$ 에 대해, $W > -1$ 값을 가집니다. 그리고, $w_{-1}(x)$ 는 또다른 실수 함수로 $x < 0$ 에 대해, $W < -1$ 값을 가집니다.

람베르트 함수들은 헤더 파일 `gsl_sf_lambert.h` 에 정의되어 있습니다.

double **gsl_sf_lambert_W0**(double x)

int **gsl_sf_lambert_W0_e**(double x, gsl_sf_result *result)

주 람베르트 W 함수 $W_0(x)$ 값을 계산합니다.

double **gsl_sf_lambert_Wm1**(double x)

int **gsl_sf_lambert_Wm1_e**(double x, gsl_sf_result *result)

두 번째 실수 람베르트 W 함수 $W_{-1](x)}$ 값을 계산합니다.

7.25 르장드르 함수와 구면조화 함수 (Legendre Functions and Spherical Harmonics)

르장드르 다항식과 함수들은 Abramowitz & Stegun Chapter 8. 에 기술되어있습니다. `gsl_sf_legnedre.h` 헤더 파일에 정의되어 있습니다.

7.25.1 르장드르 다항식(Legendre Polynomials)

double **gsl_sf_legendre_P1**(double x)

double **gsl_sf_legendre_P3**(double x)

double **gsl_sf_legendre_P2**(double x)

int **gsl_sf_legendre_P1_e**(double x, gsl_sf_result *result)

int **gsl_sf_legendre_P2_e**(double x, gsl_sf_result *result)

int **gsl_sf_legendre_P3_e**(double x, gsl_sf_result *result)

르장드르 다항식 $P_l(x)$ 을 $l = 1, 2, 3$ 에 대해 계산합니다. 이 계산은 해당 함수들의 수학적 정의식을 사용합니다.

double **gsl_sf_legendre_Pl**(int l, double x)

int **gsl_sf_legendre_Pl_e**(int l, double x, gsl_sf_result *result)

르장드르 다항식 $P_l(x)$ 을 주어진 l 과 x 에 대해 계산합니다. 이때, l 과 x 는 $l \geq 0$ 과 $|x| \leq 1$ 을 만족해야 합니다.

int **gsl_sf_legendre_Pl_array**(int lmax, double x, double result_array[])

int **gsl_sf_legendre_Pl_deriv_array**(int lmax, double x, double result_array[], double result_deriv_array[])

$l = 0, \ldots lmax$ 와 $|x| \leq 1$ 에 대해, 르장드르 다항식 $P_l(x)$ 과 그 도함수 $dP_l(x)/dx$ 를 주어진 배열 `result_array[]` 에 계산합니다.

double **gsl_sf_legendre_Q0**(double x)

int **gsl_sf_legendre_Q0_e**(double x, gsl_sf_result *result)

$x > -1$ 과 $x \neq 1$ 에 대해, 르장드르 함수 $Q_0(x)$ 를 계산합니다.

double **gsl_sf_legendre_Q1**(double x)

int **gsl_sf_legendre_Q1_e**(double x, gsl_sf_result *result)

$x > -1$ 과 $x \neq 1$ 에 대해, 르장드르 함수 $Q_1(x)$ 를 계산합니다.

double **gsl_sf_legendre_Ql**(int l, double x)

int **gsl_sf_legendre_Ql_e**(int l, double x, gsl_sf_result *result)

$x > -1$, $x \neq 1$ 과 $l \geq 0$ 에 대해 ,르장드르 함수 $Q_l(x)$ 를 계산합니다.

7.25.2 버금 르장드르 함수와 구면 조화 함수 (Associated Legendre Polynomials and Spherical Harmonics)

이 단원의 함수들은 버금 르장드르 함수 $P_l^m(x)$ 의 값을 계산합니다. 다음 미분 방정식의 해입니다.

$$(1 - x^2)\frac{d^2}{dx^2}P_l^m(x) - 2x\frac{d}{dx}P_l^m(x) + (l(l+1) - \frac{m^2}{1-x^2})P_l^m(x) = 0$$

l 과 m 은 $0 \leq l$ 과 $0 \leq m \leq l$ 을 만족합니다. 함수 $P_l^m(x)$ 는 조합적으로 상승하며 l 이 150 보다 클 경우 오버플로우가 발생합니다. 이를 대신해서 정규화 된 버금 르장드르 함수를 계산할 수 있습니다. 다양한 종류의 정규화 표현이 존재하고, degree와 order가 2700 인 함수까지 안정적으로 계산할 수 있습니다. 이 라이브러리에서는 다음의 정규화 표현을 제공합니다.

- 슈미트 반 정규화 (Schmidt semi-normalization)

  슈미트 반 정규화 버금 르장드르 함수는 자기 교환 계산에 빈번히 사용됩니다. 이는 다음과 같이 계산할 수 있습니다.

$$S_l^0(x) = P_l^0(x)$$

$$S_l^m(x) = (-1)^m\sqrt{2\frac{(l-m)!}{(l+m)!}}P_l^m(x), m > 0$$

  계수 $(-1)^m$ 은 Condon-Shortley 위상 계수로 불리며, 필요시 함수에서 $csphase = 1$ 인자를 설정해 무시할 수 있습니다.

- 구면 조화 정규화 (Spherical Harmonic Normalization)

버금 르장드르 함수는 다음과 같이 구면 조화 함수를 계산하는 데 쓸 수 있습니다.

$$Y_l^m(x) = (-1)^m \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} P_l^m(x)$$

계수 $(-1)^m$ 은 필요시 계산에서 제외할 수 있습니다.

- 완전 정규화 (Full Normalization)

  완전 정규화된 버금 르장드르 다항식은 다음과 같이 정의됩니다.

$$N_l^m(x) = (-1)^m \sqrt{(1 + \frac{1}{2} \frac{(l-m)!}{(l+m)!}} P_l^m(x)$$

  이 때, 다음과 같은 성질을 가집니다.

$$\int_{-1}^{1} N_l^m(x)\, dx = 1$$

아래에 나올 정규화된 버금 르장드르 함수를 계산하는 함수들은 재귀적 방법을 사용합니다. 이 방법은 degree l 과 order m 이 2700 이하라면, 안정적으로 계산할 수 있습니다. 이 값을 넘어서면 계산 함수들은 언더플로우를 일으켜 부정확한 값을 반환합니다. 각 함수들은 1 -계 도함수 $dP_l^m(x)/dx$ 와 2 -계 도함수 $d^2P_l^m(x)/dx^2$ 을 제공하며, 이와 함께 $dP_l^m(\cos\theta)/d\theta$ 와 $d^2P_l^m(\cos\theta)/d\theta^2$ 도 같이 제공합니다. 이 두 종류의 도함수들은 단순한 비례 관계를 가집니다. θ 에 관한 미분은 구면 조화 함수에서 매우 빈번히 사용되기 때문에 이 기능 또한 같이 제공하고 있습니다.

아래의 함수에서 `gsl_sf_legendre_t` 인자를 이용해 정규화 방법을 선택할 수 있습니다. 가능한 값들은 다음과 같습니다.

type **gsl_sf_legendre_t**

값	설명
GSL_SF_LEGENDRE_NONE	비 정규화된 버금 르장드르 다항식 $P_l^m(x)$
GSL_SF_LEGENDRE_SCHMIDT	슈미트 반 정규화된 버금 르장드르 다항식 $S_l^m(x)$
GSL_SF_LEGENDRE_SPHARM	구면 조화 버금 르장드르 다항식 $Y_l^m(x)$
GSL_SF_LEGENDRE_FULL	완전 정규화 버금 르장드르 다항식 $N_l^m(x)$

int **gsl_sf_legendre_array**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, double result_array[])

int **gsl_sf_legendre_array_e**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, const double csphase, double result_array[])

정규화된 버금 르장드르 다항식들을, $0 \le l \le lmax$, $0 \ge m \ge l$ 그리고 $|x| \le 1$ 에 대해 계산합니다. `norm` 인자는 어느 정규화 방법을 사용할지 결정합니다. 정규화된 $P_l^m(x)$ 값들은 `result_array` 에 저장됩니다. 이 값은 $gsl_sf_legendre_array_n()$ 를 호출해 최소 크기를 결정할 수 있습니다.

배열 $P_l^m(x)$ 의 지수는 gsl_sf_legendre_array_index(l, m) 를 호출해 얻을 수 있습니다. _e 붙은 함수에서 Condon-Shortly 위상 계수 $(-1)^m$ 의 포함 유무를 조정하려면 csphase 을 -1 이나 1 로 설정해 주면 됩니다. 이 계수는 기본적으로 비활성화 되어 있습니다.

int **gsl_sf_legendre_deriv_array**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, double result_array[], double result_deriv_array[])

int **gsl_sf_legendre_deriv_array_e**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, const double csphase, double result_array[], double result_deriv_array[])

$|x| \leq 1$ 값에 대해, 정규화된 버금 르장드르 함수들의 1 차에서 $lmax$ 차 까지의 도함수 값을 계산합니다. norm 인자는 어느 정규화 방법을 사용할지 결정합니다. 르장드르 함수 $P_l^m(x)$ 와 $dP_l^m(x)/dx$ 값은 각각 result_array 와 result_deriv_array 에 저장됩니다. _e 붙은 함수에서 Condon-Shortly 위상 계수 $(-1)^m$ 의 포함 유무를 조정하려면 csphase 을 -1 이나 1 로 설정해 주면 됩니다. 이 계수는 기본적으로 비활성화 되어 있습니다.

int **gsl_sf_legendre_deriv_alt_array**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, double result_array[], double result_deriv_array[])

int **gsl_sf_legendre_deriv_alt_array_e**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, const double csphase, double result_array[], double result_deriv_array[])

$|x| \leq 1$ 값에 대해, 1 차에서 $lmax$ 까지의 정규화된 버금 르장드르 함수의 값과 대체된 도함수 값을 계산합니다. 르장드르 함수 $P_l^m(x)$ 와 $dP_l^m(\cos(\theta))/d\theta$ 의 값들은 각각 result_array 와 result_deriv_array 에 저장됩니다. _e 붙은 함수에서 Condon-Shortly 위상 계수 $(-1)^m$ 의 포함 유무를 조정하려면 csphase 을 -1 이나 1 로 설정해 주면 됩니다. 이 계수는 기본적으로 비활성화 되어 있습니다.

int **gsl_sf_legendre_deriv2_array**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, double result_array[], double result_deriv_array[], double result_deriv2_array[])

int **gsl_sf_legendre_deriv2_array_e**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, const double csphase, double result_array[], double result_deriv_array[], double result_deriv2_array[])

$|x| \leq 1$ 값에 대해, 1 차에서 $lmax$ 까지의 정규화된 버금 르장드르 함수들, 그 도함수들과 2 계 도함수 값들을 계산합니다. norm 인자는 어느 정규화 방법을 사용할지 결정합니다. 르장드르 함수 $P_l^m(x)$ 와 $dP_l^m(x)/dx$, 그리고 2 계 도함수 $d^2P_l^m(x)/dx^2$ 의 값은 각각 result_array , result_deriv_array , 그리고 result_deriv2_array 에 저장됩니다. _e 붙은 함수에서 Condon-Shortly 위상 계수 $(-1)^m$ 의 포함 유무를 조정하려면 csphase 을 -1 이나 1 로 설정해 주면 됩니다. 이 계수는 기본적으로 비활성화 되어 있습니다.

int **gsl_sf_legendre_deriv2_alt_array**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, double result_array[], double result_deriv_array[], double result_deriv2_array[])

int **gsl_sf_legendre_deriv2_alt_array_e**(const gsl_sf_legendre_t norm, const size_t lmax, const double x, const double csphase, double result_array[], double result_deriv_array[], double result_deriv2_array[])

$|x| \leq 1$ 값에 대해, 1 차에서 $lmax$ 까지의 정규화된 버금 르장드르 함수들, 그 대체 도함수들과 2 계 도함수 값들을 계산합니다. norm 인자는 어느 정규화 방법을 사용할지 결정합니다. 르장드르 함수 $P_l^m(x)$ 와 $dP_l^m(\cos(\theta))/d\theta$, 그리고 2 계 도함수 $d^2P_l^m(\cos(\theta))/d\theta^2$ 의 값은 각각 result_array , result_deriv_array, 그리고 result_deriv2_array 에 저장됩니다. _e 붙은 함수에서 Condon-Shortly 위상 계수 $(-1)^m$ 의 포함 유무를 조정하려면 csphase 을 -1 이나 1 로 설정해 주면 됩니다. 이 계수는 기본적으로 비활성화 되어 있습니다.

size_t **gsl_sf_legendre_nlm**(const size_t lmax)

$lmax$ 까지의 버금 르장드르 함수 $P_l^m(x)$ 의 갯수를 반환합니다. 해당 값은 $(lmax_1) * (lmax + 2)/2$ 입니다.

size_t **gsl_sf_legendre_array_n**(const size_t lmax)

최대 차수 $lmax$ 까지의 배열 버전 버금 르장드르 함수에 필요한 최소 배열의 크기를 반환합니다. 이 값은 $P_l^m(x)$ 의 최댓값과 재귀식을 계산할 때 필요한, 곱 계수 계산을 위한 공간을 더한 값입니다.

size_t **gsl_sf_legendre_array_index**(const size_t l, const size_t m)

result_array result_deriv_array 그리고 result_deriv2_array 배열의 색인 값을 반환합니다. 해당 값은 $P_l^m(x)$, $P_l^{'m}(x)$, 그리고 $P_l^{''m}(x)$ 에 대응되고 주어진 l m 대해, $l(l_1)//2+m$ 으로 정해집니다.

$HAVE_INLINE$ 를 사용하면 인라인 버전의 함수를 사용할 수 있습니다.

double **gsl_sf_legendre_Plm**(int l, int m, double x)

int **gsl_sf_legendre_Plm_e**(int l, int m, double x, gsl_sf_result *result)

$m \geq 0, l \geq m$ 그리고 $|x| \leq 1$ 에 대해, 버금 르장드르 함수 $P_l^m(x)$ 의 값을 계산합니다.

double **gsl_sf_legendre_sphPlm**(int l, int m, double x)

int **gsl_sf_legendre_sphPlm_e**(int l, int m, double x, gsl_sf_result *result)

구면 조화 함수에서 사용하기 위한, 정규화된 버금 르장드르 다항식 $\sqrt{(2l+1)/(4\pi)}\sqrt{(l-m)!/(l+m)!}P_l^m(x)$ 값을 계산합니다. 앞의 계수는 $m \geq 0, l \geq m$ 그리고 $|x| \leq 1$ 를 만족해야합니다. 표준 정규화과정에서 일어나는 오버플로우를 피할 수 있습니다.

int **gsl_sf_legendre_Plm_array**(int lmax, int m, double x, double result_array[])

int **gsl_sf_legendre_Plm_deriv_array**(int lmax, int m, double x, double result_array[], double result_deriv_array[])

현재 비활성화 되어 있고 차후 버전에서 삭제될 예정입니다. gsl_sf_legendre_array() 와

gsl_sf_legendre_deriv_array() 를 참고하길 바랍니다.

int **gsl_sf_legendre_sphPlm_array**(int lmax, int m, double x, double result_array[])

int **gsl_sf_legendre_sphPlm_deriv_array**(int lmax, int m, double x, double result_array[], double result_deriv_array[])

현재 비활성화 되어 있고 차후 버전에서 삭제될 예정입니다. gsl_sf_legendre_array() 와 gsl_sf_legendre_deriv_array() 를 참고하길 바랍니다.

int **gsl_sf_legendre_array_size**(const int lmax, const int m)

현재 비활성화 되어 있고 차후 버전에서 삭제될 예정입니다.

7.25.3 원뿔 함수 (Conial Functions)

원통 함수 $P^{\mu}_{-(1/2)+i\lambda}$ 와 $Q^{\mu}_{-(1/2)+i\lambda}$ 는 Abramowitz & Stegun 8.12 단원에 기술되어 있습니다.

double **gsl_sf_conicalP_half**(double lambda, double x)

int **gsl_sf_conicalP_half_e**(double lambda, double x, gsl_sf_result *result)

$x > -1$ 에 대해, 비정칙 구면 원뿔 함수 $P^{1/2}_{-1/2+i\lambda}(x)$ 값을 계산합니다.

double **gsl_sf_conicalP_mhalf**(double lambda, double x)

int **gsl_sf_conicalP_mhalf_e**(double lambda, double x, gsl_sf_result *result)

$x > -1$ 에 대해, 정칙 구면 원뿔 함수 $P^{1/2}_{-1/2+i\lambda}(x)$ 값을 계산합니다.

double **gsl_sf_conicalP_0**(double lambda, double x)

int **gsl_sf_conicalP_0_e**(double lambda, double x, gsl_sf_result *result)

$x > -1$ 에 대해, 원뿔 함수 $P^{0}_{-1/2+i\lambda}(x)$ 값을 계산합니다.

double **gsl_sf_conicalP_1**(double lambda, double x)

int **gsl_sf_conicalP_1_e**(double lambda, double x, gsl_sf_result *result)

$x > -1$ 에 대해, 원뿔 함수 $P^{1}_{-1/2+i\lambda}(x)$ 값을 계산합니다.

double **gsl_sf_conicalP_sph_reg**(int l, double lambda, double x)

int **gsl_sf_conicalP_sph_reg_e**(int l, double lambda, double x, gsl_sf_result *result)

$x > -1$, $l \geq -1$ 에 대해, 정칙 구면 원뿔 함수 $P^{-1/2-l}_{1/2+i\lambda}(x)$ 값을 계산합니다.

double **gsl_sf_conicalP_cyl_reg**(int m, double lambda, double x)

int **gsl_sf_conicalP_cyl_reg_e**(int m, double lambda, double x, gsl_sf_result *result)

$x > -1$, $m \geq -1$ 에 대해, 정칙 원통 원뿔 함수 $P^{-m}_{1/2+i\lambda}(x)$ 값을 계산합니다.

7.25.4 쌍곡 공간에서의 방사 함수 (Radial Functions for Hyperbolic Space)

다음의 구면 함수들은 3 차원 쌍곡 공간 H^3 속 라플라시안의 고유 함수들인 르장드르 함수들입니다. 특히 중점으로 다루는 부분은 평평한 극한(flat limit)으로 $\lambda \to \infty, \eta \to 0$ 이고, $\lambda\eta$ 가 상수로 고정된 상황입니다.

double **gsl_sf_legendre_H3d_0**(double lambda, double eta)

int **gsl_sf_legendre_H3d_0_e**(double lambda, double eta, gsl_sf_result *result)

3 차원 쌍곡 공간 라플라시안의 0 차 고유 함수를 계산합니다. 다음과 같이 정의되어 있습니다. $\eta \geq 0$ 에 대해,

$$L_0^{H3d}(\lambda, \eta) := \frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)}$$

평평한 극한값은 $L_0^{H3d}(\lambda, \eta) = j_0(\lambda\eta)$ 의 형태를 가집니다.

double **gsl_sf_legendre_H3d_1**(double lambda, double eta)

int **gsl_sf_legendre_H3d_1_e**(double lambda, double eta, gsl_sf_result *result)

3 차원 쌍곡 공간 라플라시안의 1 차 고유 함수를 계산합니다. 다음과 같이 정의되어 있습니다. $\eta \geq 0$ 에 대해,

$$L_1^{H3d}(\lambda, \eta) := \frac{1}{\sqrt{\lambda^2 + 1}}\left(\frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)}\right)(\coth(\eta) - \lambda \cot(\lambda\eta))$$

평평한 극한값은 $L_1^{H3d}(\lambda, \eta) = j_1(\lambda\eta)$ 의 형태를 가집니다.

double **gsl_sf_legendre_H3d**(int l, double lambda, double eta)

int **gsl_sf_legendre_H3d_e**(int l, double lambda, double eta, gsl_sf_result *result)

$\eta \geq 0, l \geq 0$ 에 대해, l 차수의 방사 고유 함수값을 계산합니다. 이 고유 함수들은 3 차원 쌍곡 공간 속 라플라시안의 고유 함수들입니다. 평평한 극한값은 $L_l^{H3d}(\lambda, \eta) = j_l(\lambda\eta)$ 형태를 가집니다.

int **gsl_sf_legendre_H3d_array**(int lmax, double lambda, double eta, double result_array[])

$0 \leq l \leq lmax$ 방사 고유 함수 $L_l^{H3d}(\lambda, \eta)$ 의 값을 계산합니다.

7.26 로그 함수 (Logarithm and Related Functions)

로그 함수에 관한 정보와 그 성질들은 Abramowitz & Stegun, Chapter 4를 참고할 수 있습니다. 이 단원에서 기술된 함수들은 헤더 파일 `gsl_sf_log.h` 에 기술되어 있습니다.

double **gsl_sf_log**(double x)

int **gsl_sf_log_e**(double x, gsl_sf_result *result)

$x > 0$ 인 x 대해, $\log(x)$ 의 값을 계산합니다.

double **gsl_sf_log_abs**(double x)

int **gsl_sf_log_abs_e**(double x, gsl_sf_result *result)

$x \neq 0$ 인 x 대해, $\log(|x|)$ 을 계산합니다.

int **gsl_sf_complex_log_e**(double zr, double zi, gsl_sf_result *lnr, gsl_sf_result *theta)

복소수 $z = z_r + iz_i$ 의 로그값을 계산합니다. 결과 값은 `lnr` 와 `theta` 에 각각 저장되며, 다음과 같은 관계를 가집니다. $\exp(lnr + i\theta) = z_r + iz_i$, θ 는 $[-\pi, \pi]$ 의 범위를 가집니다.

double **gsl_sf_log_1plusx**(double x)

int **gsl_sf_log_1plusx_e**(double x, gsl_sf_result *result)

$x > -1$ 에 대해, $\log(1 + x)$ 의 값을 계산합니다. 이 함수에서 사용한 알고리즘은 작은 x 대해 정확합니다.

double **gsl_sf_log_1plusx_mx**(double x)

int **gsl_sf_log_1plusx_mx_e**(double x, gsl_sf_result *result)

$x > -1$ 에 대해, $\log(1 + x) - x$ 의 값을 계산합니다. 이 함수에서 사용한 알고르즘은 작은 x 대해 정확합니다.

7.27 마티유 함수 (Mathieu Functions)

이 단원에서는 극 마티유 함수와 방사형 마티유 함수를 계산하는 함수들을 서술합니다. 이 함수들의 특성값의 계산 기능도 같이 제공합니다. 마트유 함수는 다음 두 미분 방정식의 해로 주어집니다.

$$\frac{d^2y}{dv^2} + (a - 2q\cos 2v)y = 0$$
$$\frac{d^2f}{du^2} - (a - 2q\cosh 2u)f = 0$$

극 마티유 함수 $ce_r(x, q)$ 와 $se_r(x, q)$ 은 각각 첫번째 미분 방정식의 짝수, 홀수 번째 주기 함수해 입니다. 이 첫번째 미분 방정식은 마티유 방정식으로도 알려져 있습니다. 이 해들은 짝수와 홀수에 대해 각각 특성값의 이산 배열값들 $a = a_r(q)$ 와 $a = b_r(q)$ 에 대해서만 존재합니다.

방사형 마티유 함수 $Mc_r^{(j)}(z, q)$ 와 $Ms_r^{(j)}(z, q)$ 는 두번째 미분 방정식의 해로 이 미분 방성직은 수정 마티유 방정식으로 알려져 있습니다. 1, 2, 3, 그리고 4차 방사 마티유 함수들은 계수 j 로 표현됩니다. j 는 1, 2, 3, 4 값을 가집니다.

마티유 함수에 대한 더 자세한 정보를 얻고 싶다면 Abramowitz and Stegunn, Chapter 20을 참고하길 바랍니다. 이 함수들은 헤더 파일 `gsl_sf_mathieu.h` 에 정의되어 있습니다.

7.27.1 마티유 함수 작업 공간 (Mathieu Function Workspace)

마티유 함수들은 단일 차수나 복수 차수의 함수가 있으며, 배열에 기반한 방법을 통해 계산할 수 있습니다. 배열 기반 함수들은 사전 정의된 작업 공간이 필요합니다.

type **gsl_sf_mathieu_workspace**

배열 기반 방법을 위한 작업 공간입니다.

gsl_sf_mathieu_workspace ***gsl_sf_mathieu_alloc**(size_t n, double qmax)

마티유 함수들의 배열 기반 방법을 위한 작업 공간을 반환합니다. 인자 `n` 는 `qmax` 최대 계수와 공간에서 계산되는 마티유 함수의 q -값을 결정합니다.

void **gsl_sf_mathieu_free**(gsl_sf_mathieu_workspace *work)

주어진 작업 공간 `work` 를 해제합니다.

7.27.2 마티유 함수 특성 값 (Mathieu Function Characteristic Values)

int **gsl_sf_mathieu_a**(int n, double q)

int **gsl_sf_mathieu_a_e**(int n, double q, gsl_sf_result *result)

int **gsl_sf_mathieu_b**(int n, double q)

int **gsl_sf_mathieu_b_e**(int n, double q, gsl_sf_result *result)

각각, 마티유 함수 $ce_n(q,x)$ 와 $se_n(q,x)$ 의 특성 값 $a_n(q)$ 와 $b_n(q)$ 을 계산합니다.

int **gsl_sf_mathieu_a_array**(int order_min, int order_max, double q,
gsl_sf_mathieu_workspace *work, double result_array[])

int **gsl_sf_mathieu_b_array**(int order_min, int order_max, double q,
gsl_sf_mathieu_workspace *work, double result_array[])

마티유 함수의 특성 값 $a_n(q)$ 와 $b_n(q)$ 을 주어진 `oreder_min` 와 `order_max` 사이 범위에 있는 `n` 에 대해 계산합니다. 계산 결과는 `result_array` 에 저장됩니다.

7.27.3 각 운동량 마티유 함수 (Angular Mathieu Functions)

int **gsl_sf_mathieu_ce**(int n, double q, double x)

int **gsl_sf_mathieu_ce_e**(int n, double q, double x, gsl_sf_result *result)

int **gsl_sf_mathieu_se**(int n, double q, double x)

int **gsl_sf_mathieu_se_e**(int n, double q, double x, gsl_sf_result *result)

각각, 각 운동량 마티유 함수 $ce_n(q,x)$ 와 $se_n(q,x)$ 를 계산합니다.

int **gsl_sf_mathieu_ce_array**(int nmin, int nmax, double q, double x,
gsl_sf_mathieu_workspace *work, double result_array[])

int **gsl_sf_mathieu_se_array**(int nmin, int nmax, double q, double x,
gsl_sf_mathieu_workspace *work, double result_array[])

각각, 각 운동량 마티유 함수 $ce_n(q,x)$ 와 $se_n(q,x)$ 의 값을 주어진 nmin 와 nmax 범위에 있는 n 에 대해 계산합니다. 계산 결과는 result_array 에 저장됩니다.

7.27.4 방사 마티유 함수 (Radial Mathieu Functions)

int **gsl_sf_mathieu_Mc**(int j, int n, double q, double x)

int **gsl_sf_mathieu_Mc_e**(int j, int n, double q, double x, gsl_sf_result *result)

int **gsl_sf_mathieu_Ms**(int j, int n, double q, double x)

int **gsl_sf_mathieu_Ms_e**(int j, int n, double q, double x, gsl_sf_result *result)

각각, j 종 n 차수의 마티유 함수 $Mc_n^{(j)}(q,x)$ 와 $Ms_n^{(j)}(q,x)$ 를 계산합니다.

j 값은 $1,2$ 로 한정됩니다. $j=3,4$ 는 다음의 관계를 이용해 계산할 수 있습니다. $M_n^{(j)} = Mc_n^{(j)}$ 나 $Ms_n^{(j)}$ 에 대해, $M_n^{(3)} = M_n^{(1)} + iM_n^{(2)}$ 와 $M_n^{(4)} = M_n^{(1)} - iM_n^{(2)}$.

int **gsl_sf_mathieu_Mc_array**(int j, int nmin, int nmax, double q, double x,
gsl_sf_mathieu_workspace *work, double result_array[])

int **gsl_sf_mathieu_Ms_array**(int j, int nmin, int nmax, double q, double x,
gsl_sf_mathieu_workspace *work, double result_array[])

j 메티유 함수의 값을 주어진 nmin 와 nmax 범위에 있는 n 에 대해 계산합니다. 계산 결과는 result_array 에 저장됩니다.

7.28 멱함수 (Power Function)

다음의 함수들은 오차 계산과 함께라면, 함수 gsl_pow_int() 같습니다. 이 함수들은 헤더 파일 $gsl_sf_pow_int.h$ 에 정의되어 있습니다.

double **gsl_sf_pow_int**(double x, int n)

int **gsl_sf_pow_int_e**(double x, int n, gsl_sf_result *result)

정수 n 대해 x^n 의 값을 계산합니다. 이 때, 거듭 제곱은 곱셈 횟수를 최소화해 계산됩니다. 예를 들어서, x^8 을 계산하고자 하면, 3 번의 곱셈을 이용해 $((x^2)^2)^2$ 으로 계산합니다. 효율성을 위해서 이 함수는 오버 플로우나 언더 플로우 조건을 검사하지 않습니다.

다음은 간단한 예제입니다.

```
#include <gsl/gsl_sf_pow_int.h>
/* compute 3.0**12 */
double y = gsl_sf_pow_int(3.0, 12);
```

7.29 프사이(디감마) 함수 (Psi (Digamma) Function)

차수 n 의 극 감마 함수는 다음과 같이 정의됩니다.

$$\psi^{(n)}(x) = \left(\frac{d}{dx}\right)^n \psi(x) = \left(\frac{d}{dx}\right)^{n+1} \log(\Gamma(x))$$

$\psi(x) = \Gamma'(x)/\Gamma(x)$ 는 Digamma 함수로 알려져 있습니다. 이 단원의 함수들은 헤더 파일 $gsl_sf_psi.h$ 에 기술되어 있습니다.

7.29.1 디감마 함수 (Digamma Function)

double **gsl_sf_psi_int**(int n)

int **gsl_sf_psi_int_e**(int n, gsl_sf_result *result)

디감마 함수 $\psi(n)$ 값을 주어진 양의 정수 n 대해 계산합니다. 디감마 함수는 프사이(Psi) 함수라고도 불립니다.

double **gsl_sf_psi**(double x)

int **gsl_sf_psi_e**(double x, gsl_sf_result *result)

디감마 함수 $\psi(x)$ 를 주어진 값 x 대해 계산합니다. 이때, $x \neq 0$ 입니다.

double **gsl_sf_psi_1piy**(double y)

int **gsl_sf_psi_1piy_e**(double y, gsl_sf_result *result)

$1 + iy$ 선 위에서의 디감마 함수의 실수부 $\Re[\psi(1 + iy)]$ 를 계산합니다.

7.29.2 Trigamma 함수 (Trigamma Function)

double **gsl_sf_psi_1_int**(int n)

int **gsl_sf_psi_1_int_e**(int n, gsl_sf_result *result)

Trigamma 함수 $\psi'(n)$ 을 주어진 양의 정수 n 대해 계산합니다.

double **gsl_sf_psi_1**(double x)

int **gsl_sf_psi_1_e**(double x, gsl_sf_result *result)

Trigamma 함수 $\psi'(x)$ 를 주어진 값 x 대해 계산합니다.

7.29.3 Polygamma 함수 (Polygamma Function)

double **gsl_sf_psi_n**(int n, double x)

int **gsl_sf_psi_n_e**(int n, double x, gsl_sf_result *result)

$n \geq 0, x > 0$ 인 n, x 에 대해, polygamma 함수 $\psi^{(n)}(x)$ 를 계산합니다.

7.30 싱크트론 함수 (Synchrotron Functions)

이 단원의 함수들은 gsl_sf_sychrotron.h 해더 파일에 기술되어 있습니다.

double **gsl_sf_synchrotron_1**(double x)

int **gsl_sf_synchrotron_1_e**(double x, gsl_sf_result *result)

$x \geq 0$ 에 대해, 1차 싱크로트론 함수 $x \int_x^\infty K_{5/3}\, dt$ 를 계산합니다.

double **gsl_sf_synchrotron_2**(double x)

int **gsl_sf_synchrotron_2_e**(double x, gsl_sf_result *result)

$x \geq 0$ 에 대해, 2차 싱크로트론 함수 $x \int_x^\infty K_{2/3}\, dt$ 를 계산합니다.

7.31 운송 함수 (Transport Functions)

운송 함수 $J(n, x)$ 는 다음과 같이 적분으로 정의됩니다.

$$J(n, x) = \int_0^x \frac{t^n e^t}{(e^t - 1)^2}\, dx$$

헤더파일 : $code$: `$gsl_s f_t ransport.h$`에정의되어있습니다.

double **gsl_sf_transport_2**(double x)

int **gsl_sf_transport_2_e**(double x, gsl_sf_result *result)

운송 함수 $J(2, x)$ 의 값을 계산합니다.

double **gsl_sf_transport_3**(double x)

int **gsl_sf_transport_3_e**(double x, gsl_sf_result *result)

운송 함수 $J(3, x)$ 의 값을 계산합니다.

double **gsl_sf_transport_4**(double x)

int **gsl_sf_transport_4_e**(double x, gsl_sf_result *result)

운송 함수 $J(4, x)$ 의 값을 계산합니다.

double **gsl_sf_transport_5**(double x)

int **gsl_sf_transport_5_e**(double x, gsl_sf_result *result)

운송 함수 $J(5, x)$ 의 값을 계산합니다.

7.32 삼각 함수 (Trigonometric Functions)

본 라이브러리는 여러 플랫폼에서 일관적인 코드 작성과 신뢰할 수 있는 오류 측정을 위해 자체적인 삼각 함수를 포함하고

헤더파일 `gsl_sf_trig.h` 에 정의되어 있습니다.

7.32.1 삼각 함수 (Circular Trigonometric Functions)

double **gsl_sf_sin**(double x)

int **gsl_sf_sin_e**(double x, gsl_sf_result *result)

sine 함수 $\sin(x)$ 를 계산합니다.

double **gsl_sf_cos**(double x)

int **gsl_sf_cos_e**(double x, gsl_sf_result *result)

cosine 함수 $\cos(x)$ 를 계산합니다.

double **gsl_sf_hypot**(double x, double y)

int **gsl_sf_hypot_e**(double x, double y, gsl_sf_result *result)

오버 플로우와 언더 플로우 없는 $\sqrt{x^2 + y^2}$ 값을 계산합니다.

double **gsl_sf_sinc**(double x)

int **gsl_sf_sinc_e**(double x, gsl_sf_result *result)

x 에 대해, $\mathrm{sinc}(x) = \sin(\pi x)/(\pi x)$ 값을 계산합니다.

7.32.2 복소 삼각 함수 (Trigonometric Functions for Complex Arguments)

int **gsl_sf_complex_sin_e**(double zr, double zi, gsl_sf_result *szr, gsl_sf_result *szi)

복소 sine 함수 $\sin(z_r + iz_i)$ 값을 계산합니다. 계산 값의 실수부는 szr 에 허수부는 szi 에 저장됩니다.

int **gsl_sf_complex_cos_e**(double zr, double zi, gsl_sf_result *czr, gsl_sf_result *czi)

복소 cosine 함수 $\cos(z_r + iz_i)$ 값을 계산합니다. 계산 값의 실수부는 czr 에, 허수부는 czi 에 저장됩니다.

int **gsl_sf_complex_logsin_e**(double zr, double zi, gsl_sf_result *lszr, gsl_sf_result *lszi)

sine 함수의 로그 값 $\log(\sin(z_r + iz_i))$ 값을 계산합니다. 계산 값의 실수부는 $lszr$ 에 허수부는 $lszi$ 에 저장됩니다.

7.32.3 쌍곡 함수 (Hyperbolic Trigonometric Functions)

double **gsl_sf_lnsinh**(double x)

int **gsl_sf_lnsinh_e**(double x, gsl_sf_result *result)

log(sinh(x)) 값을, $x > 0$ 에 대해 계산합니다.

double **gsl_sf_lncosh**(double x)

int **gsl_sf_lncosh_e**(double x, gsl_sf_result *result)

주어진 값 x 대해, log(cosh(x)) 값을 계산합니다.

7.32.4 좌표 변환 함수 (Conversion Functions)

int **gsl_sf_polar_to_rect**(double r, double theta, gsl_sf_result *x, gsl_sf_result *y)

극 좌표(r $theta$) 를 직교 좌표(x y)로 변환합니다. $x = r\cos(\theta), y = r\sin(\theta)$ 관계를 이용합니다.

int **gsl_sf_rect_to_polar**(double x, double y, gsl_sf_result *r, gsl_sf_result *theta)

직교 좌표(x y)를 극 좌표(r $theta$)로 변환합니다. $x = r\cos(\theta), y = r\sin(\theta)$ 관계를 이용합니다. $theta$ 는 $[-\pi, \pi]$ 의 범위를 가집니다.

7.32.5 각 제한 함수 (Restriction Functions)

double **gsl_sf_angle_restrict_symm**(double theta)

int **gsl_sf_angle_restrict_symm_e**(double *theta)

각 $theta$ 값을 $(-\pi, pi]$ 범위 내에 있도록 변환합니다.

참고: 실제 π 값은 M_PI 보다 조금 큽니다.따라서, M_PI $-M_PI$ 이 범위에 포함되어 있습니다.

---

double **gsl_sf_angle_restrict_pos**(double theta)

int **gsl_sf_angle_restrict_pos_e**(double *theta)

각 $theta$ 값을 $(0, 2pi]$ 범위 내에 있도록 변환합니다.

참고: 실제 2π 값은 $2M_PI$ 보다 조금 큽니다. 따라서, $2 * M_PI$ 이 범위에 포함되어 있습니다.

7.32.6 오차 분석을 포함한 삼각 함수 (Trigonometric Functions With Error Estimates)

int **gsl_sf_sin_err_e**(double x, double dx, gsl_sf_result *result)

각 x 대해, 버금 절대 오차 dx 가 포함된 sine 값 $\sin(x \pm dx)$ 을 계산합니다.

참고: 오차 전파를 계산하기 위해 제공하는 것이기 때문에 오차 관리 함수 형태만으로 제공됩니다.

---

int **gsl_sf_cos_err_e**(double x, double dx, gsl_sf_result *result)

각 x 대해, 버금 절대 오차 dx 포함된 cosine 값 $\cos(x \pm dx)$ 을 계산합니다.

참고: 오차 전파를 계산하기 위해 제공하는 것이기 때문에 오차 관리 함수 형태만으로 제공됩니다.

7.33 제타 함수 (Zeta Functions)

7.33.1 리만 제타 함수 (Riemann Zeta Function)

리만 제타 함수는 다음과 같이 무한 급수로 정의됩니다.

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$$

double **gsl_sf_zeta_int**(int n)

int **gsl_sf_zeta_int_e**(int n, gsl_sf_result *result)

정수 n 대한 리만 제타 함수 $\zeta(n)$ 의 값을 계산합니다. $n \neq 1$ 이어야 합니다.

double **gsl_sf_zeta**(double s)

int **gsl_sf_zeta_e**(double s, gsl_sf_result *result)

임의의 수 s 대해 리만 제타 함수 $\zeta(s)$ 값을 계산합니다. 위 함수와 마찬가지로 $s \neq 1$ 이어야 합니다.

7.33.2 리만 제타 함수 -1 (Riemann Zeta Function Minus One)

큰 양수 값에 대해, 리만 제타함수는 1로 수렴하게 됩니다. 이 경우에 1이 아닌 부분의 값이 빈번히 사용되므로, 라이브러리에서는 이를 위한 함수를 제공합니다.

double **gsl_sf_zetam1_int**(int n)

int **gsl_sf_zetam1_int_e**(int n, gsl_sf_result *result)

정수 n 대한 $\zeta(n) - 1$ 의 값을 계산합니다. $n \neq 1$ 이어야 합니다.

double **gsl_sf_zetam1**(double s)

int **gsl_sf_zetam1_e**(double s, gsl_sf_result *result)

임의의 수 s 대해 $\zeta(s) - 1$ 값을 계산합니다. 위 함수와 마찬가지로 $s \neq 1$ 이어야 합니다.

7.33.3 후르비츠(Hurwitz) 제타 함수 (Hurwitz Zeta Function)

후르비츠 제타함수는 다음과 같이 정의됩니다.

$$\zeta(s,q) = \sum_0^\infty (k+q)^{-s}$$

double **gsl_sf_hzeta**(double s, double q)

int **gsl_sf_hzeta_e**(double s, double q, gsl_sf_result *result)

후르비츠 제타 함수 $\zeta(s,q)$ 의 값을 계산합니다. $s > 1, q > 0$ 이어야 합니다.

7.33.4 에타(Eta) 함수 (Eta Function)

에타 함수는 다음과 같이 정의됩니다.

$$\eta(s) = (1 - 2^{1-s})\zeta(s)$$

double **gsl_sf_eta_int**(int n)

int **gsl_sf_eta_int_e**(int n, gsl_sf_result *result)

정수 n 대해 에타 함수 $\eta(n)$ 의 값을 계산합니다.

double **gsl_sf_eta**(double s)

int **gsl_sf_eta_e**(double s, gsl_sf_result *result)

임의의 수 s 대해 에타 함수 $\eta(s)$ 의 값을 계산합니다.

7.34 예제 (Examples)

다음 예제는 오차 관리 형태의 특수 함수 구현체를 사용하는 방법을 보여줍니다. 베셀 함수 $J_0(5.0)$ 값을 계산합니다.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
  double *x* = 5.0;
  gsl_sf_result result;

  double expected = -0.17759677131433830434739701;

  int status = gsl_sf_bessel_J0_e (x, &result);

  printf ("status  = %s\n", gsl_strerror(status));
  printf ("J0(5.0) = %.18f\n"
          "      +/- % .18f\n",
          result.val, result.err);
  printf ("exact   = %.18f\n", expected);
  return status;
}
```

다음은 프로그램의 출력 결과입니다.

```
status  = success
J0(5.0) = -0.    177596771314338264
      +/-  0.    00000000000000019   3
exact   = -0.    177596771314338292
```

다음 프로그램은 같은 함수(베셀 함수)로 똑같은 값($J_0(5.0)$)을 계산합니다. 이 때, $result.err$ 반환 상태는 존재하지 않습니다.

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
{
  double *x* = 5.0;
  double expected = -0.  177596771314338304347     39701;

  double y =     gsl_sf_bessel_J0 (x);

  printf ("J0(5.0) = %.  18f\n", y);
  printf ("exact   = %.  18f\n", expected);
  return 0;
}
```

출력 결과는 다음과 같습니다.

```
J0(5.0) = -0.177596771314338264
exact   = -0.177596771314338292
```

7.35 참고 문헌과 추가 자료

이 라이브러리는 다음 책의 규약을 따릅니다.

- Handbook of Mathematical Functions, edited by Abramowitz & Stegun, Dover, ISBN 0486612724.

다음 논문들은 특수 함수들을 계산하기 위한 알고리즘에 관한 내용입니다.

- Allan J. MacLeod, MISCFUN: A software package to compute uncommon special functions. ACM Trans. Math. Soft., vol.: 22, 1996, 288–301
- Bunck, B. F., A fast algorithm for evaluation of normalized Hermite functions, BIT Numer. Math, 49: 281-295, 2009.
- G.N. Watson, A Treatise on the Theory of Bessel Functions, 2nd Edition (Cambridge University Press, 1944).
- G. Nemeth, Mathematical Approximations of Special Functions, Nova Science Publishers, ISBN 1-56072-052-2
- B.C. Carlson, Special Functions of Applied Mathematics (1977)
- N. M. Temme, Special Functions: An Introduction to the Classical Functions of Mathematical Physics (1996), ISBN 978-0471113133.
- W.J. Thompson, Atlas for Computing Mathematical Functions, John Wiley & Sons, New York (1997).

- Y.Y. Luke, Algorithms for the Computation of Mathematical Functions, Academic Press, New York (1977).
- S. A. Holmes and W. E. Featherstone, A unified approach to the Clenshaw summation and the recursive computation of very high degree and order normalised associated Legendre functions, Journal of Geodesy, 76, pg. 279-299, 2002.

제 8 장

벡터와 행렬

이 단원에서 기술하는 함수들은 일반적인 C 배열에 대한 간단한 벡터와 행렬 인터페이스를 제공합니다. 이러한 배열의 메모리 관리는 후술할 '블럭(Block)'이라는 구현체로 이루어집니다. 사용자 정의 함수를 벡터와 행렬을 이용해서 작성하면, 별도의 추가 인자를 구현할 필요 없이 값과 차원을 모두 포함하는 단일 자료형을 인자로 전달할 수 있습니다. 이 구조는 BLAS에서 사용하는 벡터와 행렬과 호환됩니다.

8.1 자료형

이 단원에서 기술하는 자료형들은 대응되는 표준 자료형들이 존재합니다. 배정밀도 `double` 의 정밀도를 가지는 `gsl_block` , `gsl_vector` , 그리고 `gsl_matrix` 자료형이 존재하며, 단정밀도 `float` 의 경우 `gsl_block_float` , `gsl_vector_float` , 그리고 `gsl_matrix_float` 로 존재합니다.

사용 가능한 전체 자료형은 다음과 같습니다.

GSL 자료형	Type
gsl_block	double
gsl_block_float	float
gsl_block_long_double	long double
gsl_block_int	int
gsl_block_uint	unsigned int
gsl_block_long	long
gsl_block_ulong	unsigned long
gsl_block_short	short
gsl_block_ushort	unsigned short
gsl_block_char	char
gsl_block_uchar	unsigned char
gsl_block_complex	complex double
gsl_block_complex_float	complex float
gsl_block_complex_long_double	complex long double

벡터 gsl_vector 와 행렬 gsl_matrix 에 대해서도 동일한 형태로 존재합니다.

8.2 블럭

일관성을 위해 모든 메모리는 gsl_block 구조체를 통해 할당됩니다. 이 구조체는 2가지 변수를 포함합니다. 메모리의 크기를 담는 size 와 실제 데이터가 담긴 메모리를 가르키는 포인터 data 구성됩니다.(*) gsl_block 구조체는 다음과 같이 정의됩니다.

type **gsl_block**

```
typedef struct
{
    size_t size;
    double * data;
} gsl_block;
```

벡터와 행렬들은 이러한 블럭을 자른 조각들로 구현됩니다. 각 조각은 스텝 크기와 인덱스들의 조합, 그리고 시작 설정으로 구성됩니다. 행렬의 경우 열 인덱스의 스텝 크기는 행의 길이를 나타냅니다. 벡터의 스텝 크기는 보폭 stride 로 표기합니다.

이러한 블럭을 메모리에 할당하고, 해제하는 함수들은 gsl_block.h 로 정의되어 있습니다.

8.2.1 블럭 할당

블럭을 메모리에 할당하는 함수들은 malloc 와 free 형태로 이루어 집니다. 여기에 더해, 각자 고유의 오류 검사 기능을 수행합니다. 만약 블럭을 할당하기에 메모리가 충분하지 않다면 각각의 함수들은 GSL 오류 관리자를 호출해 GSL_ENOMEM 를 넘기고 함수는 NULL 포인터를 반환합니다. 따라서 라이브러리의 오류 관리자이 프로그램을 정지하는 형태로 사용한다면 모든 alloc 과정을 검사할 필요가 없습니다.

gsl_block *__gsl_block_alloc__(size_t n)

> n 의 배정밀도 크기의 메모리를 블럭에 할당하고, 해당 블럭을 가르키는 포인터를 반환합니다. 이 블럭이 가진 메모리는 초기화가 이루어지지 않은 상태입니다. gsl_block_calloc() 를 사용하면 모든 메모리가 0 으로 초기화 된 상태의 블럭을 얻을 수 있습니다.

0 **크기의 블록을 반환하라는 것도 유효합니다.** NULL 포인터가 아닌 정상적인 포인터를 반환합니다. NULL 포인터는 메모리를 할당하지 못했을 때 반환됩니다.

gsl_block *__gsl_block_calloc__(size_t n)

> 모든 원소를 0 으로 초기화한 블록을 가르키는 포인터를 반환합니다.

void __gsl_block_free__(gsl_block *b)

> 블럭 포인터 b 가르키는 메모리를 메모리에서 해제합니다.

8.2.2 블럭 읽고 쓰기 (Reading and writing blocks)

이 라이브러리에서는 이진 값이나 규격화 된 문서 파일들을 블럭에 읽고 쓸 수 있는 함수들을 제공합니다.

int __gsl_block_fwrite__(FILE *stream, const gsl_block *b)

> 블럭 b 원소 값들을 스트림 stream 바이너리 형태로 작성 합니다. 성공할 경우 0 값을 반환하고 파일을 작성하는 데 문제가 생길 경우 GSL_EFILED 값을 반환합니다. 데이터가 플랫폼의 기본 바이너리 규격으로 작성되기 때문에 다른 아키텍처에서는 호환이 안될 수 있습니다.

int __gsl_block_fread__(FILE *stream, gsl_block *b)

> 열려진 스트림 stream 로 부터 바이너리 규격 값을 읽어 블럭 b 작성합니다. 블럭 b 반드시 사전에 적절한 길이로 할당되어 있어야합니다. 왜냐하면, b 길이를 기반으로 얼마나 많은 데이터를 읽을 지 결정하기 때문입니다. 성공할 경우 0 값을 반환하고 값을 읽는 데 문제가 생길 경우 GSL_EFILED 값을 반환합니다. 읽는 값들은 동일한 플랫폼의 기본 바이너리 규격으로 작성되었다고 가정합니다.

int __gsl_block_fprintf__(FILE *stream, const gsl_block *b, const char *format)

> 블럭 b 원소들을 줄마다 스트림 stream 작성합니다. 이때, 형 결정자 format 로 값들이 작성됩니다. 이 결정자는 부동 소수점을 위한 %g , %e , 그리고 %f 가 존재하며, 정수를 위한 %d 이 있습니다. 성공할 경우 0 값을 반환하고 파일을 작성하는 데 문제가 생길 경우 GSL_EFILED 값을 반환합니다.

int **gsl_block_fscanf**(FILE *stream, gsl_block *b)

스트림 stream 로 부터 규격화 된 값을 읽어 블럭 b 에 저장합니다. 블럭 b 반드시 사전에 적절한 길이로 할당되어 있어야합니다. 왜냐하면, 이 함수는 b 길이를 기반으로 얼마나 많은 데이터를 읽을지 결정하기 때문입니다. 성공할 경우 0 값을 반환하고 값을 읽는 데 문제가 생길 경우 GSL_EFILED 값을 반환합니다.

8.2.3 블럭 예제

다음 프로그램은 블럭을 할당하는 방법을 보여줍니다.

```
#include <stdio.h>
#include <gsl/gsl_block.h>

int
main (void)
{
  gsl_block * b = gsl_block_alloc (100);

  printf ("length of block = %zu\n", b->size);
  printf ("block data address = %p\n", b->data);

  gsl_block_free (b);
  return 0;
}
```

다음은 프로그램의 출력 결과입니다.

```
length of block = 100
block data address = 0x804b0d8
```

8.3 벡터

벡터는 gsl_vector 구조체로 블럭을 쪼갠 형태로 정의되어있습니다. 각기 다른 벡터가 같은 블록을 가르킬 수도 있습니다. 벡터 슬라이스는 동일한 크기 메모리 영역들의 집합입니다.

gsl_vector 구조체는 5 개의 성분을 가지고 있습니다. size, stride, 원소들이 저장되어 있는 메모리를 가리키는 포인터 data 벡터가 소유하고 있는 블럭을 나타내는 포인터 block 그리고 소유 상태를 나타내는 owner 가 있습니다. 이 구조체는 간단히 다음과 같이 정의되어 있습니다.

type **gsl_vector**

```
typedef struct
{
  size_t size;
  size_t stride;
  double * data;
  gsl_block * block;
  int owner;
} gsl_vector;
```

size 간단히 벡터의 원소 수를 나타냅니다. 원소에 접근할 수 있는 인덱스의 범주는 0 에서 size-1 까지입니다. stride 물리적 메모리에서 한 개의 원소에서 다음 원소로 접근할 때의 보폭 값입니다. 이는 적절한 데이터 형의 단위로 정해집니다. 포인터 data 는 메모리에서 첫번째 벡터 원소를 가리킵니다. 포인터 block 은 벡터 원소들이 위치한 메모리 블럭의 주소를 가리킵니다. 벡터가 해당 블럭을 소유하고 있다면, owner 는 1 로 설정됩니다. 그리고 벡터가 해제될 때, 블럭도 해제됩니다. 다른 객체가 블럭을 소유하고 있다면, 벡터의 owner 는 0 으로 설정되고 메모리에서 벡터가 해제되어도, 블럭은 해제되지 않습니다.

벡터를 할당하고 접근하는 함수들은 gsl_vector.h 에 정의되어 있습니다.

8.3.1 벡터 할당 (Vector allocation)

벡터를 메모리에 할당하는 함수들은 malloc , free 와 같은 형식을 따릅니다. 이에 더해 각자 고유의 오류 확인 기능을 수행합니다. 메모리 공간이 충분하지 않아 벡터를 할당할 수 없다면, 함수들은 GSL 오류 관리자를 불러 GSL_ENOMEM 오류 값를 넘기고 NULL 포인터를 반환합니다. 때문에 프로그램을 멈추는 데 오류 관리자를 사용한다면 모든 alloc 과정을 확인할 필요가 없습니다.

gsl_vector ***gsl_vector_alloc**(size_t n)

길이 n 의 벡터를 만들고, 벡터 구조체의 시작 지점을 가르키는 포인터를 반환합니다. 벡터의 원소로 새로운 블럭이 할당되고 벡터 구조체의 block 변수에 저장됩니다. 블럭은 벡터에 "소유된" 상태입니다. 만약, 벡터가 해제된다면 블럭도 같이 해제됩니다. 크기가 0 인 요청도 수행되며 non-null 결과를 반환합니다.

gsl_vector ***gsl_vector_calloc**(size_t n)

길이 n 의 벡터를 만들고, 벡터 구조체의 시작 지점을 가르키는 포인터를 반환합니다. 벡터의 모든 원소들은 0 으로 초기화 되어있습니다.

void **gsl_vector_free**(gsl_vector *v)

할당된 벡터 v 를 해제합니다. 벡터가 gsl_vector_alloc() 를 통해 만들어졌다면 벡터 안의 블럭도 같이 해제됩니다. 만약, 벡터가 메모리의 다른 객체로부터 만들어졌고 다른 객체가 소유권을 가지고 있다면 블럭은 해제되지 않습니다.

8.3.2 벡터 원소에 접근

포트란 컴파일러들과는 다르게 C 컴파일러들은 벡터와 행렬들에 대해 범위 확인 기능을 지원하지 않습니다[1]. 함수 gsl_vector_get() gsl_vector_set() 간단한 범위 확인 기능을 제공하고, 범위 밖 요소에 접근할 경우 오류를 보고합니다.

벡터와 행렬의 원소에 접근하는 함수들은 gsl_vector.h 에 정의 되어 있으며, extern inline 로 인라인 선언 되어 있습니다. 이러한 구조는 함수 호출 과정에서 오버헤드를 줄여줍니다. 이를 위해서는 프로그램을 컴파일 할 때, 전처리기에서 매크로 HAVE_INLINE 처리 하도록 선언해야합니다.

GSL_RANGE_CHECK_OFF

필요한 경우 GSL_RANGE_CHECK_OFF 정의해 재컴파일하면, 코드의 수정 없이 모든 범위 확인 기능을 무시할 수 있습니다. 인라인 함수를 지원하는 컴파일러에서 이러한 범위 확인 기능을 끄는 것은 gsl_vector_get(v,i) 와 gsl_vector_set(v,i,x) 를 각각 v->data[i*v->stride] 와 v->data[i*v->stride]=x 로 바꾸는 것과 같습니다. 따라서, 범위 확인 기능을 해제한 상황에서 범위 확인 기능이 있는 함수를 사용해도 성능상의 하락은 없습니다.

GSL_C99_INLINE

C99 컴파일러를 사용한다면 extern inline 대신 inline 키워드를 사용해 인라인 함수를 정의합니다. 이 경우 매크로 GSL_C99_INLINE 를 정의해야합니다. GCC의 C99 모드 -std=c99 에서는 자동으로 선택됩니다.

int **gsl_check_range**

만약 인라인 함수들을 사용하지 않으면, gsl_vector_get() 와 gsl_vector_set() 함수는 라이브러리에 존재하는 컴파일된 버전으로 연결됩니다. 이러한 경우, 범위 확인 기능은 전역 정수 변수 gsl_check_range 에 의해 제어됩니다. 기본적으로 이 옵션은 활성화 되어있으며 gsl_check_range 0 으로 설정해 기능을 끌 수 있습니다. 함수 호출 과정에서 오버헤드의 문제 때문에, 인라인 함수와 비교해서 범위 확인 기능을 끄는 것은 권장되지 않습니다.

double **gsl_vector_get**(const gsl_vector *v, const size_t i)

벡터 v 의 i 번째 원소를 반환합니다. 만약, i 가 0 에서 size-1 밖에 있다면, 오류 관리자가 실행되고 0 이 반환됩니다. HAVE_INLINE 이 정의된 경우, 인라인 함수가 사용됩니다.

void **gsl_vector_set**(gsl_vector *v, const size_t i, double x)

벡터 v 의 i 번째 원소를 x 로 설정합니다. 만약, i 가 0 에서 size-1 밖에 있다면, 오류 관리자가 실행됩니다. HAVE_INLINE 이 정의된 경우, 인라인 함수가 사용됩니다.

double ***gsl_vector_ptr**(gsl_vector *v, size_t i)

const double ***gsl_vector_const_ptr**(const gsl_vector *v, size_t i)

벡터 v 의 i 번째 원소를 가르키는 포인터를 반환합니다. 만약, i 가 0 에서 size-1 밖에 있다면,

[1] GNC C 컴파일러에서는 확장기능을 통해 범주 확인 기능을 지원합니다. 하지만, GCC의 기본 설치 목록에 포함되어 있지는 않습니다. 메모리 접근은 GCC의 gcc -fmudflap 라는 메모리 보호 옵션과 Valgrind로 확인할 수 있습니다.

오류 관리자가 실행되고 NULL 포인터가 반환됩니다. HAVE_INLINE 이 정의된 경우, 인라인 함수가 사용됩니다.

8.3.3 벡터 원소 초기화

void **gsl_vector_set_all**(gsl_vector *v, double x)

벡터 v 의 모든 원소를 주어진 값 x 으로 초기화합니다.

void **gsl_vector_set_zero**(gsl_vector *v)

벡터 v 의 모든 원소를 0 으로 초기화합니다.

int **gsl_vector_set_basis**(gsl_vector *v, size_t i)

벡터 v 의 원소중 i 번째 원소를 1 로, 나머지 원소를 0 으로 초기화해, 기저 벡터를 만듭니다.

8.3.4 벡터 읽고 쓰기

이 라이브러리에서는 이진 값이나 규격화 된 문서 파일들을 벡터에 읽고 쓸 수 있는 함수들을 제공합니다.

int **gsl_vector_fwrite**(FILE *stream, const gsl_vector *v)

벡터 v 의 원소들을 스트림 stream 에 이진 형식의 내용으로 작성합니다. 작성에 오류가 없으면 0 값을 반환하고, 작성에서 오류가 발생했을 시 GSL_EFAILED 값을 반환합니다. 해당 값들은 작동하는 시스템의 이진 형식을 따르기 때문에, 다른 시스템끼리의 호환성은 보장되지 않습니다.

int **gsl_vector_fread**(FILE *stream, gsl_vector *v)

열린 스트림 stream 의 값들을 읽어 주어진 벡터 v 에 이진 형태의 값으로 저장합니다. 벡터 v 는 사전에 할당되어 있어야하며, 읽을 값의 크기에 맞추어 적절한 길이를 가지고 있어야합니다. 왜냐하면, 이 함수가 값을 읽는 길이는 v 의 길이에 의존하기 때문입니다. 작성에 오류가 없으면 0 값을 반환하고, 작성에서 오류가 발생했을 시 GSL_EFAILED 값을 반환합니다. 읽을 값은 같은 시스템에서 사용하는 이진 형식으로 작성되었다라 가정합니다.

int **gsl_vector_fprintf**(FILE *stream, const gsl_vector *v, const char *format)

주어진 벡터 v 의 원소들을 정해진 형식 format 로 스트림 stream 에 한 줄씩 기록합니다. format 으로 부동 소수점을 위한 %g , %e , 그리고 %f 가 있고, 정수형은 %d 를 사용할 수 있습니다. 작성에 오류가 없으면 0 값을 반환하고, 작성에서 오류가 발생했을 시 GSL_EFAILED 값을 반환합니다.

int **gsl_vector_fscanf**(FILE *stream, gsl_vector *v)

stream 로부터 형식화된 값을 읽어 벡터 v 에 저장합니다. 벡터 v 는 사전에 할당되어 있어야하며, 읽을 값의 크기에 맞추어 적절한 길이를 가지고 있어야합니다. 왜냐하면, 이 함수가 값을 읽는 길이는 v 의 길이에 의존하기 때문입니다. 작성에 오류가 없으면 0 값을 반환하고, 작성에서 오류가 발생했을 시 GSL_EFAILED 값을 반환합니다.

8.3.5 벡터 view

블럭을 나누어 벡터를 만들었다시피, 벡터를 나누어 벡터 view를 만들 수도 있습니다. 예를 들어, 다른 벡터의 부분 벡터를 벡터 view로 표현할 수 있으며, 벡터의 홀수 번째 원소들과 짝수 번째 원소들을 두 개의 벡터 view로 표현할 수도 있습니다.

type **gsl_vector_view**

type **gsl_vector_const_view**

벡터 view는 stack에 저장되는 임시 개체로 벡터 원소들의 부분 집합에 적용할 수 있는 연산입니다. 이러한 벡터 view는 상수 벡터와 비상수 벡터 모두에 적용할 수 있습니다. 각각의 성질 보존을 위해 자료형을 분리해 제공합니다. 일반적인 벡터는 `gsl_vector_view`, 상수 벡터는 `gsl_vector_const_view` 자료형을 사용합니다. 두 경우 모두, view의 원소는 view 객체의 `vector` 성분을 사용해 `gsl_vector`에 접근할 수 있습니다. `gsl_vector *` 와 `const gsl_vector *` 포인터는 이 성분들에 `&` 연산을 적용해 얻을 수 있습니다.

이 포인터를 사용할 때는, view가 스코프 안에 있는지 확인하는 것이 매우 중요합니다. 가장 간단한 방법은 포인터를 `&view.vector` 형태로 사용하고, 다른 변수에 해당 값을 저장하지 않는 것입니다.

gsl_vector_view **gsl_vector_subvector**(gsl_vector *v, size_t offset, size_t n)

gsl_vector_const_view **gsl_vector_const_subvector**(const gsl_vector *v, size_t offset, size_t n)

벡터 `v` 의 부분 벡터에 대한 벡터 view를 반환합니다. 새로운 벡터의 시작 값은 원본 벡터의 시작 지점으로부터 거리 `offset` 을 이용해 정해집니다. 새 벡터는 `n` 의 원소를 가지고 있습니다. 수학적으로 새로운 벡터 `v'` 의 `i` 번째 원소는 다음과 같이 주어집니다.

```
v'(i) = v->data[(offset +i)*v -> stride]
```

`i` 는 `0` 에서 `n-1` 까지 입니다.

반환된 벡터 구조체의 `data` 포인터는 `offset` `n` 계수가 원본 벡터의 끝을 넘을 경우 `null` 초기화됩니다.

새로운 벡터는 원본 벡터 `v` 이루는 블럭들의 view일 뿐입니다. `b` 의 원소를 이루는 블럭들은 새로운 벡터가 소유하고 있지 않습니다. view가 스코프 밖으로 나간다 하더라도, 원본 벡터 `v` 의 블럭들은 여전히 존재합니다. 원본 벡터가 담긴 메모리는 오직 원본 벡터의 해제로만 해제가 가능합니다. 물론, 원본 벡터도 만약 해당 벡터의 view가 사용되고 있을 때에는 해제가 불가능합니다.

함수 `gsl_vector_const_subvector()` 는 `gsl_vector_subvector()` 동일한 기능을 제공하고, `const` 로 선언된 벡터에 대해서 사용가능합니다.

gsl_vector_view **gsl_vector_subvector_with_stride**(gsl_vector *v, size_t offset, size_t stride, size_t n)

gsl_vector_const_view **gsl_vector_const_subvector_with_stride**(const gsl_vector *v, size_t offset, size_t stride, size_t n)

보폭 값 `stride` 이용해 다른 벡터 `v` 서 생성된 부분 벡터의 벡터 view를 반환합니다. 이 부분 벡터는

gsl_vector_subvector() 와 같은 방식으로 생성되고 n 의 원소를 가지게 됩니다. 이는 원래 벡터의 한 원소에서 다음 원소까지 stride 간격을 가지도록 추출된 원소들입니다. 수학적으로 새 벡터 v' 의 i 째 원소는 다음과 같이 주어집니다.

```
v'(i) = v->data[(offset + i*stride)*v->stride]
```

i 는 0 부터 n-1 까지의 범위를 가집니다.

유의할 점은 부분 벡터의 view들은 원래 벡터의 원소들에대한 직접 접근을 제공한다는 점입니다.

예를 들어서, 다음의 예제 코드는 길이 n 벡터 b 의 짝수번째 원소들을 0 으로 만듭니다. 홀수 번째 원소들은 건드리지 않습니다.

```
gsl_vector_view **v_even = gsl_vector_subvector_with_stride (v, 0, 2, n/2);
gsl_vector_set_zero (&v_even.vector);
```

벡터 view들은 &view.vector 를 사용해 벡터를 인자로 가지는 모든 함수들에 할당된 다른 벡터처럼 인자로 넣어줄 수 있습니다. 예를 들어 다음의 코드는 벡터 v 홀수 번째 원소들에 대해 BLAS 함수 dnrm2 사용해 노름을 계산합니다.

```
gsl_vector_view **v_odd = gsl_vector_subvector_with_stride (v, 1, 2, n/2);
double r = gsl_blas_dnrm2 (&v_odd.vector);
```

함수 gsl_vector_const_subvector_with_stride() 는 gsl_vector_subvector_with_stride() 와 동일한 기능을 제공하고, const 로 정의된 벡터들에서 사용가능합니다.

gsl_vector_view **gsl_vector_complex_real**(gsl_vector_complex *v)

gsl_vector_const_view **gsl_vector_complex_const_real**(const gsl_vector_complex *v)

복소수 벡터 v 대해 실수부로 이루어진 벡터 view를 반환합니다.

함수 gsl_vector_complex_const_real() 는 gsl_vector_complex_real() 와 동일한 기능을 제공하고, const 로 정의된 벡터들에서 사용가능합니다.

gsl_vector_view **gsl_vector_complex_imag**(gsl_vector_complex *v)

gsl_vector_const_view **gsl_vector_complex_const_imag**(const gsl_vector_complex *v)

복소수 벡터 v 대해 허수부로 이루어진 벡터 view를 반환합니다.

gsl_vector_complex_const_imag() 는 gsl_vector_complex_imag() 와 동일한 기능을 제공하고, const 로 정의된 벡터들에서 사용가능합니다.

gsl_vector_view **gsl_vector_view_array**(double *base, size_t n)

gsl_vector_const_view **gsl_vector_const_view_array**(const double *base, size_t n)

배열에 대한 벡터 view를 반환합니다. 새 벡터의 값들은 n 의 원소를 가지는 배열 base 의해 결정됩니다. 수학적으로 새 벡터 v' i 번째 원소는 다음과 같이 주어집니다.

```
v'(i) = base[i];
```

i 는 0 부터 n-1 까지의 범위를 가집니다. v 원소들을 가지는 배열은 새로운 벡터 view에 속해져있지 않습니다. view가 소멸하더라도 배열은 여전히 존재합니다. 해당 배열이 속해있는 메모리는 해당 배열을 가르키는 base 포인터가 소멸할 때 해제됩니다. 물론 반대로 배열이 소멸한 상황에서도 view는 여전히 사용할 수 있습니다.

함수 gsl_vector_const_view_array() 는 gsl_vector_view_array() 와 동일한 기능을 제공하고, const 로 정의된 벡터들에서만 사용가능합니다.

gsl_vector_view **gsl_vector_view_array_with_stride**(double *base, size_t stride, size_t n)

gsl_vector_const_view **gsl_vector_const_view_array_with_stride**(const double *base, size_t stride, size_t n)

보폭 값 stride 이용해 배열 base 에서 생성된 벡터 view를 반환합니다. 이 부분 벡터는 gsl_vector_view_array() 같은 방식으로 생성되고 n 의 원소를 가지게 됩니다. 이는 원래 배열의 한 원소에서 다음 원소까지 stride 간격을 가지도록 추출된 원소들입니다. 수학적으로 새 벡터 v' i 째 원소는 다음과 같이 주어집니다.

```
v'(i) = base[i*stride];
```

i 는 0 부터 n-1 까지 범위를 가집니다.

유의할 점은 해당 벡터 view가 본래 배열에 대해 직접적인 접근 방법을 제공한다는 점입니다. 벡터 view들은 &view.vector 를 사용해 벡터를 인자로 가지는 모든 함수들에 할당된 다른 벡터처럼 인자로 넣어줄 수 있습니다.

함수 gsl_vector_const_view_array_with_stride() 는 gsl_vector_view_array_with_stride() 와 동일한 기능을 제공하고, const 로 정의된 벡터들에서만 사용가능합니다.

8.3.6 벡터 복사

덧셈이나 곱셈 등의 일반적인 벡터 연산들은 BLAS 라이브러리에 구현되어 있습니다(BLAS 지원 을 참고할 수 있습니다). 하지만, BLAS 전체 코드가 필요 없는 몇몇 유용한 기능들이 있을 수 있습니다. 다음의 함수들은 이러한 범주에 드는 기능들을 구현한 함수들입니다.

int **gsl_vector_memcpy**(gsl_vector *dest, const gsl_vector *src)

벡터 src 원소들을 dest 벡터로 복사합니다. 이 두 벡터들은 같은 길이를 가져야합니다.

int **gsl_vector_swap**(gsl_vector *v, gsl_vector *w)

두 벡터 v 와 w 의 원소들을 교환합니다. 이 두 벡터들은 같은 길이를 가져야합니다.

8.3.7 원소 교환

다음의 함수들은 벡터의 원소들에 대해 교환, 순열 연산을 취하는 데 이용할 수 있습니다.

int **gsl_vector_swap_elements**(gsl_vector *v, size_t i, size_t j)

벡터 v 의 i 번째와 j 번째 원소들을 교환합니다.

int **gsl_vector_reverse**(gsl_vector *v)

벡터 v 의 원소 순서를 역으로 바꿉니다.

8.3.8 벡터 연산

int **gsl_vector_add**(gsl_vector *a, const gsl_vector *b)

벡터 a 와 벡터 b 의 원소를 합합니다. $a_i \leftarrow a_i + b_i$ 값이 a 벡터에 저장되며 b 는 변하지 않습니다. 이 두 벡터들은 같은 길이를 가져야합니다.

int **gsl_vector_sub**(gsl_vector *a, const gsl_vector *b)

벡터 a 와 벡터 b 의 원소를 뺍니다. $a_i \leftarrow a_i - b_i$ 값이 a 벡터에 저장되며 b 는 변하지 않습니다. 이 두 벡터들은 같은 길이를 가져야합니다.

int **gsl_vector_mul**(gsl_vector *a, const gsl_vector *b)

벡터 a 와 벡터 b 의 원소를 곱합니다. $a_i \leftarrow a_i * b_i$ 값이 a 벡터에 저장되며 b 는 변하지 않습니다. 이 두 벡터들은 같은 길이를 가져야합니다.

int **gsl_vector_div**(gsl_vector *a, const gsl_vector *b)

벡터 a 와 벡터 b 의 원소를 나눕니다. $a_i \leftarrow a_i / b_i$ 값이 a 벡터에 저장되며 b 는 변하지 않습니다. 이 두 벡터들은 같은 길이를 가져야합니다.

int **gsl_vector_scale**(gsl_vector *a, const double x)

벡터 a 원소들에 상수 계수 x 를 곱합니다. $a_i \leftarrow x a_i$ 값이 a 에 저장됩니다.

int **gsl_vector_add_constant**(gsl_vector *a, const double x)

벡터 a 원소들에 상수 x 를 더합니다. $a_i \leftarrow a_i + x$ 값이 a 에 저장됩니다.

double **gsl_vector_sum**(const gsl_vector *a)

주어진 벡터 a 원소들의 합 $\sum_{i=1}^{n} a_i$ 을 반환합니다.

int **gsl_vector_axpby**(const double alpha, const gsl_vector *x, const double beta, gsl_vector *y)

$y \leftarrow \alpha x + \beta y$ 연산을 수행합니다. 벡터 x 와 y 는 반드시 같은 길이를 가져야합니다.

8.3.9 벡터의 최대, 최소 원소 찾기

다음 연산들은 실수 벡터들에 대해서만 사용 가능합니다.

double **gsl_vector_max**(const gsl_vector *v)

벡터 v 최댓값을 반환합니다.

double **gsl_vector_min**(const gsl_vector *v)

벡터 v 최솟값을 반환합니다.

void **gsl_vector_minmax**(const gsl_vector *v, double *min_out, double *max_out)

벡터 v 최댓값과 최솟값을 반환합니다. 해당 값들은 각각 min_out 과 max_out 포인터가 가르키는 공간에 저장됩니다.

size_t **gsl_vector_max_index**(const gsl_vector *v)

벡터 v 최댓값인 원소의 위치를 반환합니다. 동일한 최댓값이 여럿 존재하는 경우 가장 작은 값이 반환됩니다.

size_t **gsl_vector_min_index**(const gsl_vector *v)

벡터 v 최솟값인 원소의 위치를 반환합니다. 동일한 최솟값이 여럿 존재하는 경우 가장 작은 값이 반환됩니다.

void **gsl_vector_minmax_index**(const gsl_vector *v, size_t *imin, size_t *imax)

벡터 v 최댓값과 최솟값의 위치들을 반환합니다. 해당 값들은 각각 imin 과 imax 포인터가 가르키는 공간에 저장됩니다. 최댓값과 최솟값이 여럿 존재하는 경우 각각 가장 작은 값이 반환됩니다.

8.3.10 벡터 성질

다음 함수들은 실수, 복소수 벡터들에대해 모두 정의되어 있습니다. 복소수 벡터들은 실수, 허수부 모두 각각의 함수들의 조건들을 만족해야 합니다. 예를 들어서 복소수 벡터가 뒤의 나오는 함수 gsl_vector_ispos() 에서 참 값이 1 이 나왔다면 이는 해당 복소수 벡터의 실수부와 허수부 모두 양수라는 것을 의미합니다. 허수부나 실수부 둘 중 하나라도 음수 값이 있다면 해당 함수는 0 을 반환할 것입니다. (*)

int **gsl_vector_isnull**(const gsl_vector *v)

int **gsl_vector_ispos**(const gsl_vector *v)

int **gsl_vector_isneg**(const gsl_vector *v)

int **gsl_vector_isnonneg**(const gsl_vector *v)

벡터 v 의 모든 원소들이 0 이거나, 양수이거나, 음수이거나, 음수가 아닌 상황들을 판정해 참일 경우 1 을 반환하고 아니면 0 을 반환합니다.

int **gsl_vector_equal**(const gsl_vector *u, const gsl_vector *v)

주어진 두 벡터 u v 에 대해, 원소들을 비교해 이 둘이 같으면 1 을 아니면 0 을 반환합니다.

8.3.11 벡터 예제

다음 프로그램은 어떻게 벡터를 할당, 초기화하고 읽는 지를 보여줍니다. gsl_vector_alloc() 와 gsl_vector_set() 그리고 gsl_vector_get() 를 사용합니다.

```
#include <stdio.h>
#include <gsl/gsl_vector.h>

int
main (void)
{
  int i;
  gsl_vector * v = gsl_vector_alloc (3);

  for (i = 0; i < 3; i++)
    {
      gsl_vector_set (v, i, 1.23 + i);
    }

  for (i = 0; i < 100; i++) /* OUT OF RANGE ERROR */
    {
      printf ("v_%d = %g\n", i, gsl_vector_get (v, i));
    }

  gsl_vector_free (v);
  return 0;
}
```

다음은 프로그램의 출력 결과입니다. 반복문의 마지막 단계에서 함수는 벡터 v 의 길이 이상을 읽으려 시도하고 해당 시도는 gsl_vector_get() 함수의 길이 확인 코드에 의해 감지되어 오류를 반환합니다.

```
$./a.out
v_0 = 1.23
v_1 = 2.23
v_2 = 3.23
gsl: vector_source.12: ERROR: index out of range
Default GSL error handler invoked.
Aborted (core dumped)
```

다음 프로그램은 벡터를 어떻게 파일에 기록하는지를 보여줍니다.

```
#include <stdio.h>
#include <gsl/gsl_vector.h>

int
main (void)
{
  int i;
  gsl_vector * v = gsl_vector_alloc (100);

  for (i = 0; i < 100; i++)
    {
      gsl_vector_set (v, i, 1.23 + i);
    }

  {
     FILE * f = fopen ("test.dat", "w");
     gsl_vector_fprintf (f, v, "%.5g");
     fclose (f);
  }

  gsl_vector_free (v);
  return 0;
}
```

프로그램을 실행하고 나면 test.dat 파일이 생성됩니다. 이 파일은 v 원소들을 값으로 가지고 있습니다. 해당 값들은 %.5g 형식으로 저장되어져 있습니다. gsl_vector_fscanf(f,v) 를 사용해 다음과 같이 읽을 수 있습니다.

```
#include <stdio.h>
#include <gsl/gsl_vector.h>

int
main (void)
{
  int i;
  gsl_vector * v = gsl_vector_alloc (10);

  {
     FILE * f = fopen ("test.dat", "r");
     gsl_vector_fscanf (f, v);
     fclose (f);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  }

  for (i = 0; i < 10; i++)
    {
      printf ("%g\n", gsl_vector_get(v, i));
    }

  gsl_vector_free (v);
  return 0;
}
```

8.4 행렬 (Matrices)

참고: 번역중

행렬은 gsl_matrix 구조체로 정의됩니다. 이 구조체는 일반화된 블록들의 분할체 입니다. 벡터와 같이 이 구조체는 메모리의 특정 원소 집합을 나타냅니다. 그러나 벡터와는 달리 2개의 인덱스를 사용합니다.

type **gsl_matrix**

gsl_matrix 구조체는 여섯개의 변수를 가지고 있습니다. 행렬의 두 차원, 물리적 차원, 행렬 원소들이 저장된 메모리를 가르키는 포인터 data , 행렬이 소유한 블럭 block 그리고 소유권을 나타내는 owner 입니다. 물리적 차원은 메모리의 배치를 나타내며 행렬의 차원과는 다를 수 있습니다. 이는 부분 행렬의 사용을 위함합니다. gsl_matrix 구조체는 다음과 같이 간단하게 정의되어 있습니다.

```
typedef struct
{
  size_t size1;
  size_t size2;
  size_t tda;
  double * data;
  gsl_block * block;
  int owner;
} gsl_matrix;
```

행렬은 행-우선 순서(row-major order)로 되어 있습니다. 이 의미는 한 행의 원소들은 메모리 내에 연속적으로 배치되어 있음을 의미합니다. 이 방법은 2차원 배열에 대한 표준 “C 언어 순서”입니다. 포트란의 경우 열-우선 순서(column-major order)로 되어 있습니다. 행의 갯수는 size1 로 표기됩니다. 사용가능한 행 인덱스는 0 에서 size1-1 까지 입니다. 같은 방식으로 열의 갯수는 size2 표기됩니다. 열 인덱스는 0 에서

size2-1 지 입니다. 물리적인 행 차원은 tda 로 표기됩니다. 이는 trailing dimension 이라고도 불리며, 메모리 내 행렬의 배치에서 행의 크기를 나타냅니다.

예를 들어서 다음 행렬은 size1 가 3 , size2 가 4 , 그리고 tda 기 8 인 행렬입니다. 메모리 내의 행렬의 물리적 배치는 좌상단에서 부터 시작해 좌에서 우로 각 행들을 따라 값들이 배치됩니다.

```
00 01 02 03 XX XX XX XX
10 11 12 13 XX XX XX XX
20 21 22 23 XX XX XX XX
```

메모리 내에서 사용되지 않는 영역은 " XX "로 표기됩니다. 포인터 data 메모리 내 행렬의 첫번째 원소의 위치를 가르킵니다. 포인터 block 행렬의 원소들이 저장된 메모리 블럭의 위치를 나타냅니다. 만약 행렬이 해당 블럭을 소유하고 있다면, owner 는 1 로 설정됩니다. 이 경우 블럭은 행렬이 해제되면 같이 해제됩니다. 만약, 행렬이 다른 객체가 소유한 블럭의 조각이라면 owner 는 0 으로 설정되고 해당 행렬이 가르키는 블럭은 해제되지 않습니다.

행렬을 할당하고 접근하는 함수들은 gsl_matrix.h 헤더 파일에 정의되어 있습니다.

8.4.1 행렬 할당

메모리에 행렬을 할당하는 함수들은 malloc 과 free 형식을 따릅니다. 그 자체로 오류 검증 기능을 가지고 있습니다. 행렬을 할당하기에 적절한 메모리가 없는 경우 GSL 오류 관리자를 호출해 GSL_ENOMEM 오류 값를 전달하고 Null 포인터를 반환합니다. 따라서, 프로그램을 정지시키기기 위해 라이브러리의 오류 관리자를 사용한다면 모든 alloc 과정을 검사할 필요가 없습니다.

gsl_matrix ***gsl_matrix_alloc**(size_t n1, size_t n2)

This function creates a matrix of size n1 rows by n2 columns, returning a pointer to a newly initialized matrix struct. A new block is allocated for the elements of the matrix, and stored in the block component of the matrix struct. The block is "owned" by the matrix, and will be deallocated when the matrix is deallocated. Requesting zero for n1 or n2 is valid and returns a non-null result.

gsl_matrix ***gsl_matrix_calloc**(size_t n1, size_t n2)

This function allocates memory for a matrix of size n1 rows by n2 columns and initializes all the elements of the matrix to zero.

void **gsl_matrix_free**(gsl_matrix *m)

This function frees a previously allocated matrix m. If the matrix was created using :fun`gsl_matrix_alloc` then the block underlying the matrix will also be deallocated. If the matrix has been created from another object then the memory is still owned by that object and will not be deallocated.

8.4.2 행렬 원소 접근

The functions for accessing the elements of a matrix use the same range checking system as vectors. You can turn off range checking by recompiling your program with the preprocessor definition `GSL_RANGE_CHECK_OFF` .

The elements of the matrix are stored in "C-order", where the second index moves continuously through memory. More precisely, the element accessed by the function `gsl_matrix_get(m,i,j)` and `gsl_matrix_set(m,i,j,x)` is:

```
m->data[i * m->tda + j]
```

where `tda` is the physical row-length of the matrix.

double **gsl_matrix_get**(const gsl_matrix *m, const size_t i, const size_t j)

This function returns the (i,j)-th element of a matrix `m`. If `i` or `j` lie outside the allowed range of 0 to `n1 - 1` and 0 to `n2 - 1` then the error handler is invoked and 0 is returned. An inline version of this function is used when `HAVE_INLINE` is defined.

void **gsl_matrix_set**(gsl_matrix *m, const size_t i, const size_t j, double x)

This function sets the value of the (i,j)-th element of a matrix `m` to `x`. If `i` or `j` lies outside the allowed range of 0 to `n1 - 1` and 0 to `n2 - 1` then the error handler is invoked. An inline version of this function is used when `HAVE_INLINE` is defined.

double ***gsl_matrix_ptr**(gsl_matrix *m, size_t i, size_t j)

const double ***gsl_matrix_const_ptr**(const gsl_matrix *m, size_t i, size_t j)

These functions return a pointer to the (i,j)-th element of a matrix `m`. If `i` or `j` lie outside the allowed range of 0 to `n1 - 1` and 0 to `n2 - 1` then the error handler is invoked and a null pointer is returned. Inline versions of these functions are used when `HAVE_INLINE` is defined.

8.4.3 행렬 원소 초기화

void **gsl_matrix_set_all**(gsl_matrix *m, double x)

This function sets all the elements of the matrix `m` to the value `x`.

void **gsl_matrix_set_zero**(gsl_matrix *m)

This function sets all the elements of the matrix `m` to zero.

void **gsl_matrix_set_identity**(gsl_matrix *m)

This function sets the elements of the matrix `m` to the corresponding elements of the

identity matrix, $m(i,j) = \delta(i,j)$, i.e. a unit diagonal with all off-diagonal elements zero. This applies to both square and rectangular matrices.

8.4.4 행렬 읽고 쓰기

The library provides functions for reading and writing matrices to a file as binary data or formatted text.

int **gsl_matrix_fwrite**(FILE *stream, const gsl_matrix *m)

This function writes the elements of the matrix `m` to the stream `stream` in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_matrix_fread**(FILE *stream, gsl_matrix *m)

This function reads into the matrix `m` from the open stream `stream` in binary format. The matrix `m` must be preallocated with the correct dimensions since the function uses the size of `m` to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl_matrix_fprintf**(FILE *stream, const gsl_matrix *m, const char *format)

This function writes the elements of the matrix `m` line-by-line to the stream `stream` using the format specifier `format`, which should be one of the `%g`, `%e` or `%f` formats for floating point numbers and `%d` for integers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file.

int **gsl_matrix_fscanf**(FILE *stream, gsl_matrix *m)

This function reads formatted data from the stream `stream` into the matrix `m`. The matrix `m` must be preallocated with the correct dimensions since the function uses the size of `m` to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

8.4.5 행렬 view

Writing {: .label .label-red}

type **gsl_matrix_view**

type **gsl_matrix_const_view**

A matrix view is a temporary object, stored on the stack, which can be used to operate

on a subset of matrix elements. Matrix views can be defined for both constant and non-constant matrices using separate types that preserve constness. A matrix view has the type `gsl_matrix_view` and a constant matrix view has the type `gsl_matrix_const_view` . In both cases the elements of the view can by accessed using the `matrix` component of the view object. A pointer `gsl_matrix *` or `const gsl_matrix *` can be obtained by taking the address of the `matrix` component with the `&` operator. In addition to matrix views it is also possible to create vector views of a matrix, such as row or column views.

gsl_matrix_view **gsl_matrix_submatrix**(gsl_matrix *m, size_t k1, size_t k2, size_t n1, size_t n2)

gsl_matrix_const_view **gsl_matrix_const_submatrix**(const gsl_matrix *m, size_t k1, size_t k2, size_t n1, size_t n2)

These functions return a matrix view of a submatrix of the matrix `m`. The upper-left element of the submatrix is the element (`k1`, `k2`) of the original matrix. The submatrix has `n1` rows and `n2` columns. The physical number of columns in memory given by `tda` is unchanged. Mathematically, the (i, j) -th element of the new matrix is given by:

```
m'(i,j) = m->data[(k1*m->tda + k2) + i*m->tda + j]
```

where the index `i` runs from 0 to `n1 - 1` and the index `j` runs from 0 to `n2 - 1`.

The `data` pointer of the returned matrix struct is set to null if the combined parameters (`i`, `j`, `n1`, `n2`, `tda`) overrun the ends of the original matrix.

The new matrix view is only a view of the block underlying the existing matrix, `m`. The block containing the elements of `m` is not owned by the new matrix view. When the view goes out of scope the original matrix `m` and its block will continue to exist. The original memory can only be deallocated by freeing the original matrix. Of course, the original matrix should not be deallocated while the view is still in use.

The function :fun`gsl_matrix_const_submatrix` is equivalent to :fun`gsl_matrix_submatrix` but can be used for matrices which are declared `const`.

gsl_matrix_view **gsl_matrix_view_array**(double *base, size_t n1, size_t n2)

gsl_matrix_const_view **gsl_matrix_const_view_array**(const double *base, size_t n1, size_t n2)

These functions return a matrix view of the array `base` . The matrix has `n1` rows and `n2` columns. The physical number of columns in memory is also given by `n2` . Mathematically, the (i, j) -th element of the new matrix is given by:

```
m'(i,j) = base[i*n2 + j]
```

where the index `i` runs from 0 to `n1 - 1` and the index `j` runs from 0 to `n2 - 1` .

The new matrix is only a view of the array `base` . When the view goes out of scope the original array `base` will continue to exist. The original memory can only be deallocated by

freeing the original array. Of course, the original array should not be deallocated while the view is still in use.

The function :fun`gsl_matrix_const_view_array` is equivalent to :fun`gsl_matrix_view_array` but can be used for matrices which are declared `const` .

gsl_matrix_view **gsl_matrix_view_array_with_tda**(double *base, size_t n1, size_t n2, size_t tda)

gsl_matrix_const_view **gsl_matrix_const_view_array_with_tda**(const double *base, size_t n1, size_t n2, size_t tda)

These functions return a matrix view of the array `base` with a physical number of columns `tda` which may differ from the corresponding dimension of the matrix. The matrix has `n1` rows and `n2` columns, and the physical number of columns in memory is given by `tda` . Mathematically, the (i,j)-th element of the new matrix is given by:

```
m'(i,j) = base[i*tda + j]
```

where the index `i` runs from 0 to `n1 - 1` and the index `j` runs from 0 to `n2 - 1` .

The new matrix is only a view of the array `base`. When the view goes out of scope the original array `base` will continue to exist. The original memory can only be deallocated by freeing the original array. Of course, the original array should not be deallocated while the view is still in use.

The function :fun`gsl_matrix_const_view_array_with_tda` is equivalent to :fun`gsl_matrix_view_array_with_tda` but can be used for matrices which are declared `const` .

gsl_matrix_view **gsl_matrix_view_vector**(gsl_vector *v, size_t n1, size_t n2)

gsl_matrix_const_view **gsl_matrix_const_view_vector**(const gsl_vector *v, size_t n1, size_t n2)

These functions return a matrix view of the vector `v`. The matrix has `n1` rows and `n2` columns. The vector must have unit stride. The physical number of columns in memory is also given by `n2`. Mathematically, the (i,j)-th element of the new matrix is given by:

```
m'(i,j) = v->data[i*n2 + j]
```

where the index `i` runs from 0 to `n1 - 1` and the index `j` runs from 0 to `n2 - 1` .

The new matrix is only a view of the vector `v`. When the view goes out of scope the original vector `v` will continue to exist. The original memory can only be deallocated by freeing the original vector. Of course, the original vector should not be deallocated while the view is still in use.

The function :fun`gsl_matrix_const_view_vector` is equivalent to :fun`gsl_matrix_view_vector` but can be used for matrices which are declared `const` .

gsl_matrix_view **gsl_matrix_view_vector_with_tda**(gsl_vector *v, size_t n1, size_t n2, size_t tda)

gsl_matrix_const_view **gsl_matrix_const_view_vector_with_tda**(const gsl_vector *v, size_t n1, size_t n2, size_t tda)

These functions return a matrix view of the vector `v` with a physical number of columns `tda` which may differ from the corresponding matrix dimension. The vector must have unit stride. The matrix has `n1` rows and `n2` columns, and the physical number of columns in memory is given by `tda`. Mathematically, the (i,j) -th element of the new matrix is given by:

```
m'(i,j) = v->data[i*tda + j]
```

where the index `i` runs from 0 to `n1 - 1` and the index `j` runs from 0 to `n2 - 1`.

The new matrix is only a view of the vector `v`. When the view goes out of scope the original vector `v` will continue to exist. The original memory can only be deallocated by freeing the original vector. Of course, the original vector should not be deallocated while the view is still in use.

The function :fun`gsl_matrix_const_view_vector_with_tda` is equivalent to :fun`gsl_matrix_view_vector_with_tda` but can be used for matrices which are declared `const`.

8.4.6 행, 열 view 생성

Writing {: .label .label-red}

In general there are two ways to access an object, by reference or by copying. The functions described in this section create vector views which allow access to a row or column of a matrix by reference. Modifying elements of the view is equivalent to modifying the matrix, since both the vector view and the matrix point to the same memory block.

gsl_vector_view **gsl_matrix_row**(gsl_matrix *m, size_t i)

gsl_vector_const_view **gsl_matrix_const_row**(const gsl_matrix *m, size_t i)

These functions return a vector view of the `i`-th row of the matrix `m`. The `data` pointer of the new vector is set to null if `i` is out of range.

The function :fun`gsl_matrix_const_row` is equivalent to :fun`gsl_matrix_row` but can be used for matrices which are declared `const`.

gsl_vector_view **gsl_matrix_column**(gsl_matrix *m, size_t j)

gsl_vector_const_view **gsl_matrix_const_column**(const gsl_matrix *m, size_t j)

These functions return a vector view of the `j`-th column of the matrix `m`. The `data` pointer of the new vector is set to null if `j` is out of range.

The function :fun`gsl_matrix_const_column` is equivalent to :fun`gsl_matrix_column` but can be used for matrices which are declared `const`.

gsl_vector_view **gsl_matrix_subrow**(gsl_matrix *m, size_t i, size_t offset, size_t n)

gsl_vector_const_view **gsl_matrix_const_subrow**(const gsl_matrix *m, size_t i, size_t offset, size_t n)

These functions return a vector view of the `i`-th row of the matrix `m` beginning at `offset` elements past the first column and containing `n` elements. The `data` pointer of the new vector is set to null if `i`, `offset`, or `n` are out of range.

The function :fun`gsl_matrix_const_subrow` is equivalent to :fun`gsl_matrix_subrow` but can be used for matrices which are declared `const`.

gsl_vector_view **gsl_matrix_subcolumn**(gsl_matrix *m, size_t j, size_t offset, size_t n)

gsl_vector_const_view **gsl_matrix_const_subcolumn**(const gsl_matrix *m, size_t j, size_t offset, size_t n)

These functions return a vector view of the `j`-th column of the matrix `m` beginning at `offset` elements past the first row and containing `n` elements. The `data` pointer of the new vector is set to null if `j`, `offset`, or `n` are out of range.

The function :fun`gsl_matrix_const_subcolumn` is equivalent to :fun`gsl_matrix_subcolumn` but can be used for matrices which are declared `const`.

gsl_vector_view **gsl_matrix_diagonal**(gsl_matrix *m)

gsl_vector_const_view **gsl_matrix_const_diagonal**(const gsl_matrix *m)

These functions return a vector view of the diagonal of the matrix `m`. The matrix `m` is not required to be square. For a rectangular matrix the length of the diagonal is the same as the smaller dimension of the matrix.

The function :fun`gsl_matrix_const_diagonal` is equivalent to :fun`gsl_matrix_diagonal` but can be used for matrices which are declared `const`.

gsl_vector_view **gsl_matrix_subdiagonal**(gsl_matrix *m, size_t k)

gsl_vector_const_view **gsl_matrix_const_subdiagonal**(const gsl_matrix *m, size_t k)

These functions return a vector view of the `k`-th subdiagonal of the matrix `m`. The matrix `m` is not required to be square. The diagonal of the matrix corresponds to $k = 0$.

The function :fun`gsl_matrix_const_subdiagonal` is equivalent to :fun`gsl_matrix_subdiagonal` but can be used for matrices which are declared `const`.

gsl_vector_view **gsl_matrix_superdiagonal**(gsl_matrix *m, size_t k)

gsl_vector_const_view **gsl_matrix_const_superdiagonal**(const gsl_matrix *m, size_t k)

These functions return a vector view of the `k`-th superdiagonal of the matrix `m`. The matrix `m` is not required to be square. The diagonal of the matrix corresponds to $k = 0$.

The function :fun`gsl_matrix_const_superdiagonal` is equivalent to :fun`gsl_matrix_superdiagonal` but can be used for matrices which are declared `const`.

8.4.7 행렬 복사

Writing {: .label .label-red}

int **gsl_matrix_memcpy**(gsl_matrix *dest, const gsl_matrix *src)

This function copies the elements of the matrix `src` into the matrix `dest`. The two matrices must have the same size.

int **gsl_matrix_swap**(gsl_matrix *m1, gsl_matrix *m2)

This function exchanges the elements of the matrices `m1` and :da

8.4.8 행, 열 복사

Writing {: .label .label-red}

The functions described in this section copy a row or column of a matrix into a vector. This allows the elements of the vector and the matrix to be modified independently. Note that if the matrix and the vector point to overlapping regions of memory then the result will be undefined. The same effect can be achieved with more generality using :fun`gsl_vector_memcpy` with vector views of rows and columns.

int **gsl_matrix_get_row**(gsl_vector *v, const gsl_matrix *m, size_t i)

This function copies the elements of the `i`-th row of the matrix `m` into the vector `v`. The length of the vector must be the same as the length of the row.

int **gsl_matrix_get_col**(gsl_vector *v, const gsl_matrix *m, size_t j)

This function copies the elements of the `j`-th column of the matrix `m` into the vector `v`. The length of the vector must be the same as the length of the column.

int **gsl_matrix_set_row**(gsl_matrix *m, size_t i, const gsl_vector *v)

This function copies the elements of the vector `v` into the `i`-th row of the matrix `m`. The length of the vector must be the same as the length of the row.

int **gsl_matrix_set_col**(gsl_matrix *m, size_t j, const gsl_vector *v)

This function copies the elements of the vector `v` into the `j`-th column of the matrix `m`. The length of the vector must be the same as the length of the column.

8.4.9 행과 열 교환하기

Writing {: .label .label-red}

The following functions can be used to exchange the rows and columns of a matrix.

int **gsl_matrix_swap_rows**(gsl_matrix *m, size_t i, size_t j)

This function exchanges the `i`-th and `j`-th rows of the matrix `m` in-place.

int **gsl_matrix_swap_columns**(gsl_matrix *m, size_t i, size_t j)

This function exchanges the `i`-th and `j`-th columns of the matrix `m` in-place.

int **gsl_matrix_swap_rowcol**(gsl_matrix *m, size_t i, size_t j)

This function exchanges the `i`-th row and `j`-th column of the matrix `m` in-place. The matrix must be square for this operation to be possible.

int **gsl_matrix_transpose_memcpy**(gsl_matrix *dest, const gsl_matrix *src)

This function makes the matrix `dest` the transpose of the matrix `src` by copying the elements of `src` into `dest`. This function works for all matrices provided that the dimensions of the matrix `dest` match the transposed dimensions of the matrix `src`.

int **gsl_matrix_transpose**(gsl_matrix *m)

This function replaces the matrix `m` by its transpose by copying the elements of the matrix in-place. The matrix must be square for this operation to be possible.

int **gsl_matrix_complex_conjtrans_memcpy**(gsl_matrix *dest, const gsl_matrix *src)

This function makes the matrix `dest` the conjugate transpose of the matrix `src` by copying the complex conjugate elements of `src` into `dest`. This function works for all complex matrices provided that the dimensions of the matrix `dest` match the transposed dimensions of the matrix `src`.

8.4.10 행렬 연산

Writing {: .label .label-red}

The following operations are defined for real and complex matrices.

int **gsl_matrix_add**(gsl_matrix *a, const gsl_matrix *b)

This function adds the elements of matrix `b` to the elements of matrix `a`. The result $a(i,j) \leftarrow a(i,j) + b(i,j)$ is stored in `a` and `b` remains unchanged. The two matrices must have the same dimensions.

int **gsl_matrix_sub**(gsl_matrix *a, const gsl_matrix *b)

This function subtracts the elements of matrix `b` from the elements of matrix `a`. The result $a(i,j) \leftarrow a(i,j) - b(i,j)$ is stored in `a` and `b` remains unchanged. The two matrices must have the same dimensions.

int **gsl_matrix_mul_elements**(gsl_matrix *a, const gsl_matrix *b)

This function multiplies the elements of matrix `a` by the elements of matrix `b`. The result $a(i,j) \leftarrow a(i,j) * b(i,j)$ is stored in `a` and `b` remains unchanged. The two matrices must have the same dimensions.

int **gsl_matrix_div_elements**(gsl_matrix *a, const gsl_matrix *b)

This function divides the elements of matrix `a` by the elements of matrix `b`. The result $a(i,j) \leftarrow a(i,j)/b(i,j)$ is stored in `a` and `b` remains unchanged. The two matrices must have the same dimensions.

int **gsl_matrix_scale**(gsl_matrix *a, const double x)

This function multiplies the elements of matrix `a` by the constant factor `x`. The result $a(i,j) \leftarrow xa(i,j)$ is stored in `a`.

int **gsl_matrix_scale_columns**(gsl_matrix *A, const gsl_vector *x)

This function scales the columns of the M-by-N matrix `A` by the elements of the vector `x`, of length N. The j -th column of `A` is multiplied by x_j. This is equivalent to forming

$$A \rightarrow AX$$

where $X = \text{diag}(x)$.

int **gsl_matrix_scale_rows**(gsl_matrix *A, const gsl_vector *x)

This function scales the rows of the M-by-N matrix `A` by the elements of the vector `x`, of length M. The i -th row of `A` is multiplied by x_i. This is equivalent to forming

$$A \rightarrow XA$$

where $X = \text{diag}(x)$.

int **gsl_matrix_add_constant**(gsl_matrix *a, const double x)

This function adds the constant value `x` to the elements of the matrix `a`. The result $a(i,j) \leftarrow a(i,j) + x$ is stored in `a`.

8.4.11 행렬 원소의 최대, 최소 찾기

Writing {: .label .label-red}

The following operations are only defined for real matrices.

double **gsl_matrix_max**(const gsl_matrix *m)

This function returns the maximum value in the matrix `m`.

double **gsl_matrix_min**(const gsl_matrix *m)

This function returns the minimum value in the matrix `m`.

void **gsl_matrix_minmax**(const gsl_matrix *m, double *min_out, double *max_out)

This function returns the minimum and maximum values in the matrix `m`, storing them in `min_out` and `max_out`.

void **gsl_matrix_max_index**(const gsl_matrix *m, size_t *imax, size_t *jmax)

This function returns the indices of the maximum value in the matrix `m`, storing them in `imax` and `jmax`. When there are several equal maximum elements then the first element found is returned, searching in row-major order.

void **gsl_matrix_min_index**(const gsl_matrix *m, size_t *imin, size_t *jmin)

This function returns the indices of the minimum value in the matrix `m`, storing them in `imin` and `jmin`. When there are several equal minimum elements then the first element found is returned, searching in row-major order.

void **gsl_matrix_minmax_index**(const gsl_matrix *m, size_t *imin, size_t *jmin, size_t *imax, size_t *jmax)

This function returns the indices of the minimum and maximum values in the matrix `m`, storing them in (`imin`, `jmin`) and (`imax`, `jmax`). When there are several equal minimum or maximum elements then the first elements found are returned, searching in row-major order.

8.4.12 행렬의 성질

다음 함수들은 실수와 복소수 행렬 모두 사용할 수 있습니다. 복소수 행렬의 경우 각 함수에서 검사하는 성질이 실수부와 허수부가 모두 일치해야 해당 행렬이 함수의 검사 조건을 만족한다 간주합니다.

int **gsl_matrix_isnull**(const gsl_matrix *m)

int **gsl_matrix_ispos**(const gsl_matrix *m)

int **gsl_matrix_isneg**(const gsl_matrix *m)

int **gsl_matrix_isnonneg**(const gsl_matrix *m)

각각 행렬의 모든 원소들이 0 이거나, 양수이거나, 음수이거나, 음수가 아니면 1 을 반환하고 아니면 0 을 반환합니다. 행렬이 positive-definite임을 검사하려면 숄레스키 분해(Cholesky decomposition)를 사용해야 합니다.

int **gsl_matrix_equal**(const gsl_matrix *a, const gsl_matrix *b)

행렬 a b 원소를 비교해서 같은 행렬이면 1 을 아니면 0 을 반환합니다.

double **gsl_matrix_norm1**(const gsl_matrix *A)

m - n 행렬 A 대해, 1 -노름값을 반환합니다. 이는 다음과 같이 정의됩니다.

$$||A||_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |A_{ij}|$$

8.4.13 행렬 예제

아래의 프로그램은 함수 gsl_matrix_alloc() 와 gsl_mtrix_set() 그리고 gsl_matrix_get() 를 이용해 행렬의 할당, 초기화, 그리고 값을 읽는 방법을 보여줍니다.

```
#include <stdio.h>
#include <gsl/gsl_matrix.h>

int
main (void)
{
  int i, j;
  gsl_matrix * m = gsl_matrix_alloc (10, 3);

  for (i = 0; i < 10; i++)
    for (j = 0; j < 3; j++)
      gsl_matrix_set (m, i, j, 0.23 + 100*i + j);

  for (i = 0; i < 100; i++)  /* OUT OF RANGE ERROR */
    for (j = 0; j < 3; j++)
      printf ("m(%d,%d) = %g\n", i, j,
              gsl_matrix_get (m, i, j));

  gsl_matrix_free (m);

  return 0;
}
```

다음은 프로그램의 출력 결과입니다. 마지막 반복문은 행렬 m 범위를 넘어 읽으려 시도하고, gsl_matrix_get() 의 범위 확인 코드가 이를 감지했습니다.

```
$./a.out
 m(0,0) = 0.23
 m(0,1) = 1.23
 m(0,2) = 2.23
 m(1,0) = 100.23
 m(1,1) = 101.23
 m(1,2) = 102.23
 ...
 m(9,2) = 902.23
 gsl: matrix_source.13: ERROR: first index out of range
 Default GSL error handler invoked.
 Aborted (core dumped)
```

다음 프로그램은 행렬을 어떻게 파일로 저장하는지 보여줍니다.

#include <stdio.h> #include <gsl/gsl_matrix.h>

int main (void) {

int i, j, k = 0; gsl_matrix * m = gsl_matrix_alloc (100, 100); gsl_matrix * a = gsl_matrix_alloc (100, 100);

for (i = 0; i < 100; i++)

for (j = 0; j < 100; j++) gsl_matrix_set (m, i, j, 0.23 + i + j);

{

FILE * f = fopen (“test.dat”, “wb”); gsl_matrix_fwrite (f, m); fclose (f);

}

{

FILE * f = fopen (“test.dat”, “rb”); gsl_matrix_fread (f, a); fclose (f);

}

for (i = 0; i < 100; i++)

for (j = 0; j < 100; j++) {

double mij = gsl_matrix_get (m, i, j); double aij = gsl_matrix_get (a, i, j); if (mij != aij) k++;

}

gsl_matrix_free (m); gsl_matrix_free (a);

printf (“differences = %d (should be zero)n”, k); return (k > 0);

}

프로그램의 실행이 끝나면, m 행렬의 원소들을 담고 있는 파일 test.dat 파일이 생성됩니다. 이 파일은 바이너리 형식으로 작성됩니다. 함수 gsl_matrix_fread() 사용해 파일을 읽어 본래 행렬과 같은 행렬을 얻을 수 있습니다.

다음 프로그램은 벡터 view를 사용하는 법을 기술합니다. 이 프로그램은 행렬의 열 노름을 계산합니다.

```
#include <math.h>
#include <stdio.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

int
main (void)
{
  size_t i,j;

  gsl_matrix *m = gsl_matrix_alloc (10, 10);

  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      gsl_matrix_set (m, i, j, sin (i) + cos (j));

  for (j = 0; j < 10; j++)
    {
      gsl_vector_view column = gsl_matrix_column (m, j);
      double d;

      d = gsl_blas_dnrm2 (&column.vector);

      printf ("matrix column %zu, norm = %g\n", j, d);
    }

  gsl_matrix_free (m);

  return 0;
}
```

결과는 다음과 같습니다.

```
matrix column 0, norm = 4.31461
matrix column 1, norm = 3.1205
matrix column 2, norm = 2.19316
matrix column 3, norm = 3.26114
matrix column 4, norm = 2.53416
matrix column 5, norm = 2.57281
matrix column 6, norm = 4.20469
matrix column 7, norm = 3.65202
matrix column 8, norm = 2.08524
matrix column 9, norm = 3.07313
```

GNU octave를 이용해 이 결과를 검증해볼 수 있습니다.

```
$octave
GNU Octave, version 2.0.16.92
octave> m = sin(0:9)' * ones(1,10)
               + ones(10,1) * cos(0:9);
octave> sqrt(sum(m.^2))
ans =
  4.3146  3.1205  2.1932  3.2611  2.5342  2.5728
  4.2047  3.6520  2.0852  3.0731
```

8.5 참고 문헌과 추가 자료

GSL의 블럭, 벡터 그리고 행렬 객체들은 C++의 valarraay 모델을 따릅니다. 이 모델의 자세한 기술은 다음을 참고할 수 있습니다.

- B. Stroustrup, The C++ Programming Language (3rd Ed), Section 22.4 Vector Arithmetic. Addison-Wesley 1997, ISBN 0-201-88954-4.

제 9 장

순열

참고: 번역중

이 단원에서는 순열을 만들고 제어하는 함수들을 서술합니다. 순열 p 는 0 에서 $n-1$ 의 범주를 가지는 n 길이의 정수 배열을 의미합니다. 각각의 값 p_i 는 배열 내에서 한번만 나옵니다. 순열 p 를 벡터 v 에 적용하면 새로운 벡터 v' 를 얻을 수 있습니다.

$$v'_i = v_{p_i}$$

예를 들어서, 배열 $(0, 1, 3, 2)$ 는 길이 4의 벡터의 원소 중 마지막 두 원소를 바꾸는 순열을 나타냅니다. 항등 순열은 $(0, 1, 2, 3)$ 로 나타낼 수 있습니다.

Note that the permutations produced by the linear algebra routines correspond to the exchange of matrix columns, and so should be considered as applying to row-vectors in the form $v' = vP$ rather than column-vectors, when permuting the elements of a vector.

이 단원에 기술된 함수들은 헤더 파일 `gsl_permutation.h` 에 기술되어 있습니다.

9.1 순열 구조체

type **gsl_permutation**

순열은 2 개의 원소를 가지는 구조체로 정의됩니다. 이 두 변수는 각각 순열의 크기와 순열 배열을 가르키는 포인터를 의미합니다. 순열 배열의 원소는 모두 `size_t` 입니다. `gsl_permutation` 구조체는 다음과 같이 정의되어 있습니다.

```
typedef struct
  {
    size_t size;
    size_t * data;
  } gsl_permutation;
```

9.2 순열 할당

gsl_permutation *gsl_permutation_alloc(size_t n)

크기 n 의 순열을 위한 메모리 공간을 할당합니다. 이 함수는 순열의 원소들을 초기화하지 않기 때문에 반환된 순열에서 각 원소들을 특정 값으로 정의되지 않은 상태입니다. gsl_permutation_calloc 함수를 사용하면 항등 순열로 초기화 된 순열을 얻을 수 있습니다. 순열을 할당하기에 메모리가 충분하지 않다면 NULL 포인터를 반환합니다.

gsl_permutation *gsl_permutation_calloc(size_t n)

크기 n 의 순열을 위한 메모리 공간을 할당합니다. 이 함수에서 반환하는 순열은 항등 순열입니다. 순열을 할당하기에 메모리가 충분하지 않다면 NULL 포인터를 반환합니다.

void gsl_permutation_init(gsl_permutation *p)

주어진 순열 p 를 항등 순열로 초기화합니다. n 크기의 순열에 대해, $(0, 1, 2, \ldots, n-1)$ 로 초기화합니다.

void gsl_permutation_free(gsl_permutation *p)

순열 p 의 메모리를 해제합니다.

int gsl_permutation_memcpy(gsl_permutation *dest, const gsl_permutation *src)

주어진 순열 src 의 원소들을 dest 로 복사합니다. 이 두 순열은 반드시 같은 크기를 가져야 합니다.

9.3 순열 접근

다음 함수들은 순열 속 값에 접근하거나 이 값들을 제어하는 기능들을 제공합니다.

size_t gsl_permutation_get(const gsl_permutation *p, const size_t i)

주어진 순열 p 의 i 번째 원소의 값을 반환합니다. i 가 0에서 $n-1$ 범위 밖으로 주어지면, 오류 관리자가 호출되고 0이 반환됩니다. HAVE_INLINE 이 정의된 경우, 인라인 함수가 사용됩니다.

int gsl_permutation_swap(gsl_permutation *p, const size_t i, const size_t j)

주어진 순열 p 의 i 번째 원소와 j 번째 원소의 값을 교환합니다.

9.4 순열의 성질

size_t **gsl_permutation_size**(const gsl_permutation *p)

주어진 순열 `p` 의 크기를 반환합니다.

size_t ***gsl_permutation_data**(const gsl_permutation *p)

주어진 순열 `p` 의 순열 배열을 가르키는 포인터를 반환합니다.

int **gsl_permutation_valid**(const gsl_permutation *p)

주어진 순열 `p` 가 타당한지 검사합니다. 구체적으로 크기 `n` 인 순열 `p` 가 0에서 `n-1` 까지의 값을 모두, 그리고 한번만 가지고 있는지 검사합니다.

9.5 순열 함수

void **gsl_permutation_reverse**(gsl_permutation *p)

주어진 순열 `p` 의 원소들을 역순으로 배열합니다.

int **gsl_permutation_inverse**(gsl_permutation *inv, const gsl_permutation *p)

주어진 순열 `p` 의 역순열을 계산해 `inv` 에 저장합니다.

int **gsl_permutation_next**(gsl_permutation *p)

This function advances the permutation `p` to the next permutation in lexicographic order and returns `GSL_SUCCESS`. If no further permutations are available it returns `GSL_FAILURE` and leaves `p` unmodified. Starting with the identity permutation and repeatedly applying this function will iterate through all possible permutations of a given order.

int **gsl_permutation_prev**(gsl_permutation *p)

This function steps backwards from the permutation `p` to the previous permutation in lexicographic order, returning `GSL_SUCCESS`. If no previous permutation is available it returns `GSL_FAILURE` and leaves `p` unmodified.

9.6 순열 적용

다음 함수들은 헤더 파일 `gsl_permute.h` 와 `gsl_permute_vector.h` 에 정의되어 있습니다.

int **gsl_permute**(const size_t *p, double *data, size_t stride, size_t n)

순열 `p` 를 길이 `n` , `stride` 크기의 걸음을 가지는 값 배열 `data` 에 적용합니다.

int **gsl_permute_inverse**(const size_t *p, double *data, size_t stride, size_t n)

순열 `p` 의 역순열을 길이 `n` , `stride` 크기의 걸음을 가지는 값 배열 `data` 에 적용합니다.

int **gsl_permute_vector**(const gsl_permutation *p, gsl_vector *v)

This function applies the permutation `p` to the elements of the vector `v`, considered as a row-vector acted on by a permutation matrix from the right, $v' = vP$. The j-th column of the permutation matrix P is given by the p_j-th column of the identity matrix. The permutation `p` and the vector `v` must have the same length.

int **gsl_permute_vector_inverse**(const gsl_permutation *p, gsl_vector *v)

This function applies the inverse of the permutation `p` to the elements of the vector `v`, considered as a row-vector acted on by an inverse permutation matrix from the right, $v' = vP^T$. Note that for permutation matrices the inverse is the same as the transpose. The j-th column of the permutation matrix P is given by the p_j-th column of the identity matrix. The permutation `p` and the vector `v` must have the same length.

int **gsl_permute_matrix**(const gsl_permutation *p, gsl_matrix *A)

This function applies the permutation `p` to the matrix `A` from the right, $A' = AP$. The j-th column of the permutation matrix P is given by the p_j-th column of the identity matrix. This effectively permutes the columns of `A` according to the permutation `p`, and so the number of columns of `A` must equal the size of the permutation `p`.

int **gsl_permutation_mul**(gsl_permutation *p, const gsl_permutation *pa, const gsl_permutation *pb)

This function combines the two permutations `pa` and `pb` into a single permutation `p`, where $p = pa * pb$ The permutation `p` is equivalent to applying `pb` first and then `pa`.

9.7 순열 읽고 쓰기

라이브러리에서 형식화 된 문자열이나 이진 파일로 순열을 읽고 쓸 수 있는 함수들을 제공합니다.

int **gsl_permutation_fwrite**(FILE *stream, const gsl_permutation *p)

This function writes the elements of the permutation `p` to the stream `stream` in binary format. The function returns `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_permutation_fread**(FILE *stream, gsl_permutation *p)

This function reads into the permutation `p` from the open stream `stream` in binary format. The permutation `p` must be preallocated with the correct length since the function uses the size of `p` to determine how many bytes to read. The function returns `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl_permutation_fprintf**(FILE *stream, const gsl_permutation *p, const char *format)

This function writes the elements of the permutation `p` line-by-line to the stream `stream` using the format specifier `format`, which should be suitable for a type of `size_t`. In ISO C99 the type modifier `z` represents `size_t`, so `"%zu\n"` is a suitable format[1]. The function returns `GSL_EFAILED` if there was a problem writing to the file.

int **gsl_permutation_fscanf**(FILE *stream, gsl_permutation *p)

This function reads formatted data from the stream `stream` into the permutation `p`. The permutation `p` must be preallocated with the correct length since the function uses the size of `p` to determine how many numbers to read. The function returns `GSL_EFAILED` if there was a problem reading from the file.

9.8 원순열

A permutation can be represented in both linear and cyclic notations. The functions described in this section convert between the two forms. The linear notation is an index mapping, and has already been described above. The cyclic notation expresses a permutation as a series of circular rearrangements of groups of elements, or cycles.

For example, under the cycle (1 2 3), 1 is replaced by 2, 2 is replaced by 3 and 3 is replaced by 1 in a circular fashion. Cycles of different sets of elements can be combined independently, for example (1 2 3) (4 1) combines the cycle (1 2 3) with the cycle (4 5), which is an exchange of elements 4 and 5. A cycle of length one represents an element which is unchanged by the permutation and is referred to as a singleton.

It can be shown that every permutation can be decomposed into combinations of cycles. The decomposition is not unique, but can always be rearranged into a standard canonical form by a reordering of elements. The library uses the canonical form defined in Knuth's Art of Computer Programming (Vol 1, 3rd Ed, 1997) Section 1.3.3, p.178.

The procedure for obtaining the canonical form given by Knuth is,

1. Write all singleton cycles explicitly
2. Within each cycle, put the smallest number first
3. Order the cycles in decreasing order of the first number in the cycle.

For example, the linear representation (2 4 3 0 1) is represented as (1 4) (0 2 3) in canonical form. The permutation corresponds to an exchange of elements 1 and 4, and rotation of elements 0, 2 and 3.

[1] In versions of the GNU C library prior to the ISO C99 standard, the type modifier `Z` was used instead.

The important property of the canonical form is that it can be reconstructed from the contents of each cycle without the brackets. In addition, by removing the brackets it can be considered as a linear representation of a different permutation. In the example given above the permutation (2 4 3 0 1) would become (1 4 0 2 3). This mapping has many applications in the theory of permutations.

int **gsl_permutation_linear_to_canonical**(gsl_permutation *q, const gsl_permutation *p)

This function computes the canonical form of the permutation `p` and stores it in the output argument `q`.

int **gsl_permutation_canonical_to_linear**(gsl_permutation *p, const gsl_permutation *q)

This function converts a permutation `q` in canonical form back into linear form storing it in the output argument `p`.

size_t **gsl_permutation_inversions**(const gsl_permutation *p)

This function counts the number of inversions in the permutation `p`. An inversion is any pair of elements that are not in order. For example, the permutation 2031 has three inversions, corresponding to the pairs (2,0) (2,1) and (3,1). The identity permutation has no inversions.

size_t **gsl_permutation_linear_cycles**(const gsl_permutation *p)

This function counts the number of cycles in the permutation `p`, given in linear form.

size_t **gsl_permutation_canonical_cycles**(const gsl_permutation *q)

This function counts the number of cycles in the permutation `q`, given in canonical form.

9.9 예제

이 예제에서는 항드원의 원소를 섞는 방법을 사용해 임의의 순열을 생성하고 그 순열의 역순열을 찾는 방법을 보여줍니다.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_permutation.h>

int
main (void)
{
  const size_t N = 10;
  const gsl_rng_type * T;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_rng * r;

  gsl_permutation * p = gsl_permutation_alloc (N);
  gsl_permutation * q = gsl_permutation_alloc (N);

  gsl_rng_env_setup();
  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  printf ("initial permutation:");
  gsl_permutation_init (p);
  gsl_permutation_fprintf (stdout, p, " %u");
  printf ("\n");

  printf (" random permutation:");
  gsl_ran_shuffle (r, p->data, N, sizeof(size_t));
  gsl_permutation_fprintf (stdout, p, " %u");
  printf ("\n");

  printf ("inverse permutation:");
  gsl_permutation_inverse (q, p);
  gsl_permutation_fprintf (stdout, q, " %u");
  printf ("\n");

  gsl_permutation_free (p);
  gsl_permutation_free (q);
  gsl_rng_free (r);

  return 0;
}
```

프로그램의 출력 결과는 다음과 같습니다.

```
$./a.out
initial permutation: 0 1 2 3 4 5 6 7 8 9
 random permutation: 1 3 5 2 7 6 0 4 9 8
inverse permutation: 6 0 3 1 7 2 5 4 9 8
```

The random permutation `p[i]` and its inverse `q[i]` are related through the identity `p[q[i]] = i`, which can be verified from the output.

The next example program steps forwards through all possible third order permutations,

starting from the identity,

```
#include <stdio.h>
#include <gsl/gsl_permutation.h>

int
main (void)
{
  gsl_permutation * p = gsl_permutation_alloc (3);

  gsl_permutation_init (p);

  do
    {
      gsl_permutation_fprintf (stdout, p, " %u");
      printf ("\n");
    }
  while (gsl_permutation_next(p) == GSL_SUCCESS);

  gsl_permutation_free (p);

  return 0;
}
```

프로그램의 출력 결과는 다음과 같습니다.

```
$./a.out
 0 1 2
 0 2 1
 1 0 2
 1 2 0
 2 0 1
 2 1 0
```

The permutations are generated in lexicographic order. To reverse the sequence, begin with the final permutation (which is the reverse of the identity) and replace :fun`gsl_permutation_next` with :fun`gsl_permutation_prev`.

9.10 참고 문헌과 추가 자료

다음의 문헌에서 순열과 관련된 광범위한 내용을 참고할 수 있습니다.

- Donald E. Knuth, The Art of Computer Programming: Sorting and Searching (Vol 3, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850.

정규형 순열에 관한 정의는 다음을 참고할 수 있습니다.

- Donald E. Knuth, The Art of Computer Programming: Fundamental Algorithms (Vol 1, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850. Section 1.3.3, An Unusual Correspondence, p.178–179.

제 10 장

조합

참고: 번역중

This chapter describes functions for creating and manipulating combinations. A combination c is represented by an array of k integers in the range 0 to $n-1$, where each value c_i occurs at most once. The combination c corresponds to indices of k elements chosen from an n element vector. Combinations are useful for iterating over all k-element subsets of a set.

The functions described in this chapter are defined in the header file `gsl_combination.h`.

10.1 The Combination struct

type **`gsl_combination`**

A combination is defined by a structure containing three components, the values of n and k, and a pointer to the combination array. The elements of the combination array are all of type `size_t`, and are stored in increasing order. The `gsl_combination` structure looks like this:

```
typedef struct
{
  size_t n;
  size_t k;
  size_t *data;
} gsl_combination;
```

10.2 Combination allocation

gsl_combination *`gsl_combination_alloc`(size_t n, size_t k)

This function allocates memory for a new combination with parameters `n`, `k`. The combination is not initialized and its elements are undefined. Use the function `gsl_combination_calloc()` if you want to create a combination which is initialized to the lexicographically first combination. A null pointer is returned if insufficient memory is available to create the combination.

gsl_combination *`gsl_combination_calloc`(size_t n, size_t k)

This function allocates memory for a new combination with parameters `n`, `k` and initializes it to the lexicographically first combination. A null pointer is returned if insufficient memory is available to create the combination.

void `gsl_combination_init_first`(gsl_combination *c)

This function initializes the combination `c` to the lexicographically first combination, i.e. $(0, 1, 2, \ldots, k-1)$.

void `gsl_combination_init_last`(gsl_combination *c)

This function initializes the combination `c` to the lexicographically last combination, i.e. $(n-k, n-k+1, \ldots, n-1)$.

void `gsl_combination_free`(gsl_combination *c)

This function frees all the memory used by the combination `c`.

int `gsl_combination_memcpy`(gsl_combination *dest, const gsl_combination *src)

This function copies the elements of the combination `src` into the combination `dest`. The two combinations must have the same size.

10.3 Accessing combination elements

The following function can be used to access the elements of a combination.

size_t `gsl_combination_get`(const gsl_combination *c, const size_t i)

This function returns the value of the `i`-th element of the combination `c`. If `i` lies outside the allowed range of 0 to $k-1$ then the error handler is invoked and 0 is returned. HAVE_INLINE 이 정의된 경우, 인라인 함수가 사용됩니다.

10.4 Combination properties

size_t **gsl_combination_n**(const gsl_combination *c)

This function returns the range (n) of the combination c.

size_t **gsl_combination_k**(const gsl_combination *c)

This function returns the number of elements (k) in the combination `c`.

size_t ***gsl_combination_data**(const gsl_combination *c)

This function returns a pointer to the array of elements in the combination `c`.

int **gsl_combination_valid**(gsl_combination *c)

This function checks that the combination `c` is valid. The `k` elements should lie in the range 0 to $n - 1$, with each value occurring once at most and in increasing order.

10.5 Combination functions

int **gsl_combination_next**(gsl_combination *c)

This function advances the combination `c` to the next combination in lexicographic order and returns `GSL_SUCCESS`. If no further combinations are available it returns `GSL_FAILURE` and leaves `c` unmodified. Starting with the first combination and repeatedly applying this function will iterate through all possible combinations of a given order.

int **gsl_combination_prev**(gsl_combination *c)

This function steps backwards from the combination `c` to the previous combination in lexicographic order, returning `GSL_SUCCESS`. If no previous combination is available it returns `GSL_FAILURE` and leaves `c` unmodified.

10.6 Reading and writing combinations

The library provides functions for reading and writing combinations to a file as binary data or formatted text.

int **gsl_combination_fwrite**(FILE *stream, const gsl_combination *c)

This function writes the elements of the combination `c` to the stream `stream` in binary format. The function returns `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_combination_fread**(FILE *stream, gsl_combination *c)

This function reads elements from the open stream `stream` into the combination `c` in binary format. The combination `c` must be preallocated with correct values of n and k since the function uses the size of `c` to determine how many bytes to read. The function returns `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl_combination_fprintf**(FILE *stream, const gsl_combination *c, const char *format)

This function writes the elements of the combination `c` line-by-line to the stream `stream` using the format specifier `format`, which should be suitable for a type of `size_t`. In ISO C99 the type modifier `z` represents `size_t`, so `"%zu\n"` is a suitable format[1]. The function returns `GSL_EFAILED` if there was a problem writing to the file.

int **gsl_combination_fscanf**(FILE *stream, gsl_combination *c)

This function reads formatted data from the stream `stream` into the combination `c`. The combination `c` must be preallocated with correct values of n and k since the function uses the size of `c` to determine how many numbers to read. The function returns `GSL_EFAILED` if there was a problem reading from the file.

10.7 Examples

The example program below prints all subsets of the set $0, 1, 2, 3$ ordered by size. Subsets of the same size are ordered lexicographically.

```
#include <stdio.h>
#include <gsl/gsl_combination.h>

int
main (void)
{
  gsl_combination * c;
  size_t i;

  printf ("All subsets of {0,1,2,3} by size:\n") ;
  for (i = 0; i <= 4; i++)
    {
      c = gsl_combination_calloc (4, i);
      do
        {
```

(다음 페이지에 계속)

[1] In versions of the GNU C library prior to the ISO C99 standard, the type modifier `Z` was used instead.

(이전 페이지에서 계속)

```
          printf ("{");
          gsl_combination_fprintf (stdout, c, " %u");
          printf (" }\n");
        }
      while (gsl_combination_next (c) == GSL_SUCCESS);
      gsl_combination_free (c);
    }

  return 0;
}
```

Here is the output from the program,

```
All subsets of {0,1,2,3} by size:
{ }
{ 0 }
{ 1 }
{ 2 }
{ 3 }
{ 0 1 }
{ 0 2 }
{ 0 3 }
{ 1 2 }
{ 1 3 }
{ 2 3 }
{ 0 1 2 }
{ 0 1 3 }
{ 0 2 3 }
{ 1 2 3 }
{ 0 1 2 3 }
```

All 16 subsets are generated, and the subsets of each size are sorted lexicographically.

10.8 References and Further Reading

Further information on combinations can be found in,

- Donald L. Kreher, Douglas R. Stinson, Combinatorial Algorithms: Generation, Enumeration and Search, 1998, CRC Press LLC, ISBN 084933988X

제 11 장

중복 집합

참고: 번역중

This chapter describes functions for creating and manipulating multisets. A multiset c is represented by an array of k integers in the range 0 to $n-1$, where each value c_i may occur more than once. The multiset c corresponds to indices of k elements chosen from an n element vector with replacement. In mathematical terms, n is the cardinality of the multiset while k is the maximum multiplicity of any value. Multisets are useful, for example, when iterating over the indices of a k-th order symmetric tensor in n-space.

The functions described in this chapter are defined in the header file `gsl_multiset.h`.

11.1 The Multiset struct

type **gsl_multiset**

A multiset is defined by a structure containing three components, the values of n and k, and a pointer to the multiset array. The elements of the multiset array are all of type `size_t`, and are stored in increasing order. The `gsl_multiset` structure looks like this:

```
typedef struct
{
  size_t n;
  size_t k;
  size_t *data;
} gsl_multiset;
```

11.2 Multiset allocation

gsl_multiset *__gsl_multiset_alloc__(size_t n, size_t k)

This function allocates memory for a new multiset with parameters `n`, `k`. The multiset is not initialized and its elements are undefined. Use the function `gsl_multiset_calloc()` if you want to create a multiset which is initialized to the lexicographically first multiset element. A null pointer is returned if insufficient memory is available to create the multiset.

gsl_multiset *__gsl_multiset_calloc__(size_t n, size_t k)

This function allocates memory for a new multiset with parameters `n`, `k` and initializes it to the lexicographically first multiset element. A null pointer is returned if insufficient memory is available to create the multiset.

void __gsl_multiset_init_first__(gsl_multiset *c)

This function initializes the multiset `c` to the lexicographically first multiset element, i.e. 0 repeated k times.

void __gsl_multiset_init_last__(gsl_multiset *c)

This function initializes the multiset `c` to the lexicographically last multiset element, i.e. $n-1$ repeated k times.

void __gsl_multiset_free__(gsl_multiset *c)

This function frees all the memory used by the multiset `c`.

int __gsl_multiset_memcpy__(gsl_multiset *dest, const gsl_multiset *src)

This function copies the elements of the multiset `src` into the multiset `dest`. The two multisets must have the same size.

11.3 Accessing multiset elements

The following function can be used to access the elements of a multiset.

size_t __gsl_multiset_get__(const gsl_multiset *c, const size_t i)

This function returns the value of the `i`-th element of the multiset `c`. If `i` lies outside the allowed range of 0 to $k-1$ then the error handler is invoked and 0 is returned. `HAVE_INLINE` 이 정의된 경우, 인라인 함수가 사용됩니다.

11.4 Multiset properties

size_t **gsl_multiset_n**(const gsl_multiset *c)

This function returns the range (n) of the multiset `c`.

size_t **gsl_multiset_k**(const gsl_multiset *c)

This function returns the number of elements (k) in the multiset `c`.

size_t ***gsl_multiset_data**(const gsl_multiset *c)

This function returns a pointer to the array of elements in the multiset `c`.

int **gsl_multiset_valid**(gsl_multiset *c)

This function checks that the multiset `c` is valid. The `k` elements should lie in the range 0 to $n - 1$, with each value occurring in nondecreasing order.

11.5 Multiset functions

int **gsl_multiset_next**(gsl_multiset *c)

This function advances the multiset `c` to the next multiset element in lexicographic order and returns `GSL_SUCCESS`. If no further multisets elements are available it returns `GSL_FAILURE` and leaves `c` unmodified. Starting with the first multiset and repeatedly applying this function will iterate through all possible multisets of a given order.

int **gsl_multiset_prev**(gsl_multiset *c)

This function steps backwards from the multiset `c` to the previous multiset element in lexicographic order, returning `GSL_SUCCESS`. If no previous multiset is available it returns `GSL_FAILURE` and leaves `c` unmodified.

11.6 Reading and writing multisets

The library provides functions for reading and writing multisets to a file as binary data or formatted text.

int **gsl_multiset_fwrite**(FILE *stream, const gsl_multiset *c)

This function writes the elements of the multiset `c` to the stream `stream` in binary format. The function returns `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_multiset_fread**(FILE *stream, gsl_multiset *c)

This function reads elements from the open stream `stream` into the multiset `c` in binary format. The multiset `c` must be preallocated with correct values of n and k since the function uses the size of `c` to determine how many bytes to read. The function returns `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl_multiset_fprintf**(FILE *stream, const gsl_multiset *c, const char *format)

This function writes the elements of the multiset `c` line-by-line to the stream `stream` using the format specifier `format`, which should be suitable for a type of `size_t`. In ISO C99 the type modifier `z` represents `size_t`, so `"%zu\n"` is a suitable format[1]. The function returns `GSL_EFAILED` if there was a problem writing to the file.

int **gsl_multiset_fscanf**(FILE *stream, gsl_multiset *c)

This function reads formatted data from the stream `stream` into the multiset `c`. The multiset `c` must be preallocated with correct values of n and k since the function uses the size of `c` to determine how many numbers to read. The function returns `GSL_EFAILED` if there was a problem reading from the file.

11.7 Examples

The example program below prints all multisets elements containing the values $0, 1, 2, 3$ ordered by size. Multiset elements of the same size are ordered lexicographically.

```
#include <stdio.h>
#include <gsl/gsl_multiset.h>

int
main (void)
{
  gsl_multiset * c;
  size_t i;

  printf ("All multisets of {0,1,2,3} by size:\n") ;
  for (i = 0; i <= 4; i++)
    {
      c = gsl_multiset_calloc (4, i);
      do
        {
```

(다음 페이지에 계속)

[1] In versions of the GNU C library prior to the ISO C99 standard, the type modifier `Z` was used instead.

(이전 페이지에서 계속)

```
          printf ("{");
          gsl_multiset_fprintf (stdout, c, " %u");
          printf (" }\n");
        }
      while (gsl_multiset_next (c) == GSL_SUCCESS);
      gsl_multiset_free (c);
    }

  return 0;
}
```

Here is the output from the program,

```
All multisets of {0,1,2,3} by size:
{ }
{ 0 }
{ 1 }
{ 2 }
{ 3 }
{ 0 0 }
{ 0 1 }
{ 0 2 }
{ 0 3 }
{ 1 1 }
{ 1 2 }
{ 1 3 }
{ 2 2 }
{ 2 3 }
{ 3 3 }
{ 0 0 0 }
{ 0 0 1 }
{ 0 0 2 }
{ 0 0 3 }
{ 0 1 1 }
{ 0 1 2 }
{ 0 1 3 }
{ 0 2 2 }
{ 0 2 3 }
{ 0 3 3 }
{ 1 1 1 }
{ 1 1 2 }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
{ 1 1 3 }
{ 1 2 2 }
{ 1 2 3 }
{ 1 3 3 }
{ 2 2 2 }
{ 2 2 3 }
{ 2 3 3 }
{ 3 3 3 }
{ 0 0 0 0 }
{ 0 0 0 1 }
{ 0 0 0 2 }
{ 0 0 0 3 }
{ 0 0 1 1 }
{ 0 0 1 2 }
{ 0 0 1 3 }
{ 0 0 2 2 }
{ 0 0 2 3 }
{ 0 0 3 3 }
{ 0 1 1 1 }
{ 0 1 1 2 }
{ 0 1 1 3 }
{ 0 1 2 2 }
{ 0 1 2 3 }
{ 0 1 3 3 }
{ 0 2 2 2 }
{ 0 2 2 3 }
{ 0 2 3 3 }
{ 0 3 3 3 }
{ 1 1 1 1 }
{ 1 1 1 2 }
{ 1 1 1 3 }
{ 1 1 2 2 }
{ 1 1 2 3 }
{ 1 1 3 3 }
{ 1 2 2 2 }
{ 1 2 2 3 }
{ 1 2 3 3 }
{ 1 3 3 3 }
{ 2 2 2 2 }
{ 2 2 2 3 }
{ 2 2 3 3 }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
{ 2 3 3 3 }
{ 3 3 3 3 }
```

All 70 multisets are generated and sorted lexicographically.

제 12 장

정렬

참고: 번역중

This chapter describes functions for sorting data, both directly and indirectly (using an index). All the functions use the heapsort algorithm. Heapsort is an $O(N \log N)$ algorithm which operates in-place and does not require any additional storage. It also provides consistent performance, the running time for its worst-case (ordered data) being not significantly longer than the average and best cases. Note that the heapsort algorithm does not preserve the relative ordering of equal elements—it is an unstable sort. However the resulting order of equal elements will be consistent across different platforms when using these functions.

12.1 Sorting objects

The following function provides a simple alternative to the standard library function `qsort()`. It is intended for systems lacking `qsort()`, not as a replacement for it. The function `qsort()` should be used whenever possible, as it will be faster and can provide stable ordering of equal elements. Documentation for `qsort()` is available in the GNU C Library Reference Manual.

The functions described in this section are defined in the header file `gsl_heapsort.h`.

void **gsl_heapsort**(void *array, size_t count, size_t size, gsl_comparison_fn_t compare)

This function sorts the `count` elements of the array `array`, each of size `size`, into ascending order using the comparison function `compare`. The type of the comparison function is defined by

type **gsl_comparison_fn_t**

```
int (*gsl_comparison_fn_t) (const void * a, const void * b)
```

A comparison function should return a negative integer if the first argument is less than the second argument, `0` if the two arguments are equal and a positive integer if the first argument is greater than the second argument.

For example, the following function can be used to sort doubles into ascending numerical order.

```
int
compare_doubles (const double * a, const double * b)
{
  if (*a > *b)
    return 1;
  else if (*a < *b)
    return -1;
  else
    return 0;
}
```

The appropriate function call to perform the sort is:

```
gsl_heapsort (array, count, sizeof(double), compare_doubles);
```

Note that unlike `qsort()` the heapsort algorithm cannot be made into a stable sort by pointer arithmetic. The trick of comparing pointers for equal elements in the comparison function does not work for the heapsort algorithm. The heapsort algorithm performs an internal rearrangement of the data which destroys its initial ordering.

int **gsl_heapsort_index**(size_t *p, const void *array, size_t count, size_t size,
gsl_comparison_fn_t compare)

This function indirectly sorts the `count` elements of the array `array`, each of size `size`, into ascending order using the comparison function `compare`. The resulting permutation is stored in `p`, an array of length `n`. The elements of `p` give the index of the array element which would have been stored in that position if the array had been sorted in place. The first element of `p` gives the index of the least element in `array`, and the last element of `p` gives the index of the greatest element in `array`. The array itself is not changed.

12.2 Sorting vectors

The following functions will sort the elements of an array or vector, either directly or indirectly. They are defined for all real and integer types using the normal suffix rules. For example, the `float` versions of the array functions are `gsl_sort_float()` and `gsl_sort_float_index()`. The corresponding vector functions are `gsl_sort_vector_float()` and `gsl_sort_vector_float_index()`. The prototypes are available in the header files `gsl_sort_float.h` `gsl_sort_vector_float.h`. The complete set of prototypes can be included using the header files `gsl_sort.h` and `gsl_sort_vector.h`.

There are no functions for sorting complex arrays or vectors, since the ordering of complex numbers is not uniquely defined. To sort a complex vector by magnitude compute a real vector containing the magnitudes of the complex elements, and sort this vector indirectly. The resulting index gives the appropriate ordering of the original complex vector.

void **gsl_sort**(double *data, const size_t stride, size_t n)

This function sorts the `n` elements of the array `data` with stride `stride` into ascending numerical order.

void **gsl_sort2**(double *data1, const size_t stride1, double *data2, const size_t stride2, size_t n)

This function sorts the `n` elements of the array `data1` with stride `stride1` into ascending numerical order, while making the same rearrangement of the array `data2` with stride `stride2`, also of size `n`.

void **gsl_sort_vector**(gsl_vector *v)

This function sorts the elements of the vector `v` into ascending numerical order.

void **gsl_sort_vector2**(gsl_vector *v1, gsl_vector *v2)

This function sorts the elements of the vector `v1` into ascending numerical order, while making the same rearrangement of the vector `v2`.

void **gsl_sort_index**(size_t *p, const double *data, size_t stride, size_t n)

This function indirectly sorts the `n` elements of the array `data` with stride `stride` into ascending order, storing the resulting permutation in `p`. The array `p` must be allocated with a sufficient length to store the `n` elements of the permutation. The elements of `p` give the index of the array element which would have been stored in that position if the array had been sorted in place. The array `data` is not changed.

int **gsl_sort_vector_index**(gsl_permutation *p, const gsl_vector *v)

This function indirectly sorts the elements of the vector `v` into ascending order, storing the resulting permutation in `p`. The elements of `p` give the index of the vector element

which would have been stored in that position if the vector had been sorted in place. The first element of `p` gives the index of the least element in `v`, and the last element of `p` gives the index of the greatest element in `v`. The vector `v` is not changed.

12.3 Selecting the k smallest or largest elements

The functions described in this section select the k smallest or largest elements of a data set of size N. The routines use an $O(kN)$ direct insertion algorithm which is suited to subsets that are small compared with the total size of the dataset. For example, the routines are useful for selecting the 10 largest values from one million data points, but not for selecting the largest 100,000 values. If the subset is a significant part of the total dataset it may be faster to sort all the elements of the dataset directly with an $O(N \log N)$ algorithm and obtain the smallest or largest values that way.

int **gsl_sort_smallest**(double *dest, size_t k, const double *src, size_t stride, size_t n)

This function copies the `k` smallest elements of the array `src`, of size `n` and stride `stride`, in ascending numerical order into the array `dest`. The size `k` of the subset must be less than or equal to `n`. The data `src` is not modified by this operation.

int **gsl_sort_largest**(double *dest, size_t k, const double *src, size_t stride, size_t n)

This function copies the `k` largest elements of the array `src`, of size `n` and stride `stride`, in descending numerical order into the array `dest`. `k` must be less than or equal to `n`. The data `src` is not modified by this operation.

int **gsl_sort_vector_smallest**(double *dest, size_t k, const gsl_vector *v)

int **gsl_sort_vector_largest**(double *dest, size_t k, const gsl_vector *v)

These functions copy the `k` smallest or largest elements of the vector `v` into the array `dest`. `k` must be less than or equal to the length of the vector `v`.

The following functions find the indices of the k smallest or largest elements of a dataset.

int **gsl_sort_smallest_index**(size_t *p, size_t k, const double *src, size_t stride, size_t n)

This function stores the indices of the `k` smallest elements of the array `src`, of size `n` and stride `stride`, in the array `p`. The indices are chosen so that the corresponding data is in ascending numerical order. `k` must be less than or equal to `n`. The data `src` is not modified by this operation.

int **gsl_sort_largest_index**(size_t *p, size_t k, const double *src, size_t stride, size_t n)

This function stores the indices of the `k` largest elements of the array `src`, of size `n` and stride `stride`, in the array `p`. The indices are chosen so that the corresponding data is

in descending numerical order. k must be less than or equal to n. The data src is not modified by this operation.

int **gsl_sort_vector_smallest_index**(size_t *p, size_t k, const gsl_vector *v)

int **gsl_sort_vector_largest_index**(size_t *p, size_t k, const gsl_vector *v)

These functions store the indices of the k smallest or largest elements of the vector v in the array p. k must be less than or equal to the length of the vector v.

12.4 Computing the rank

The rank of an element is its order in the sorted data. The rank is the inverse of the index permutation, p. It can be computed using the following algorithm:

```
for (i = 0; i < p->size; i++)
  {
    size_t pi = p->data[i];
    rank->data[pi] = i;
  }
```

This can be computed directly from the function gsl_permutation_inverse(rank,p).

The following function will print the rank of each element of the vector v:

```
void
print_rank (gsl_vector * v)
{
  size_t i;
  size_t n = v->size;
  gsl_permutation * perm = gsl_permutation_alloc(n);
  gsl_permutation * rank = gsl_permutation_alloc(n);

  gsl_sort_vector_index (perm, v);
  gsl_permutation_inverse (rank, perm);

  for (i = 0; i < n; i++)
    {
      double vi = gsl_vector_get(v, i);
      printf ("element = %d, value = %g, rank = %d\n",
              i, vi, rank->data[i]);
    }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_permutation_free (perm);
  gsl_permutation_free (rank);
}
```

12.5 Examples

The following example shows how to use the permutation p to print the elements of the vector v in ascending order:

```
gsl_sort_vector_index (p, v);

for (i = 0; i < v->size; i++)
  {
    double vpi = gsl_vector_get (v, p->data[i]);
    printf ("order = %d, value = %g\n", i, vpi);
  }
```

The next example uses the function `gsl_sort_smallest()` to select the 5 smallest numbers from 100000 uniform random variates stored in an array,

```
#include <gsl/gsl_rng.h>
#include <gsl/gsl_sort_double.h>

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;

  size_t i, k = 5, N = 100000;

  double * x = malloc (N * sizeof(double));
  double * small = malloc (k * sizeof(double));

  gsl_rng_env_setup();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  for (i = 0; i < N; i++)
    {
      x[i] = gsl_rng_uniform(r);
    }

  gsl_sort_smallest (small, k, x, 1, N);

  printf ("%zu smallest values from %zu\n", k, N);

  for (i = 0; i < k; i++)
    {
      printf ("%zu: %.18f\n", i, small[i]);
    }

  free (x);
  free (small);
  gsl_rng_free (r);
  return 0;
}
```

The output lists the 5 smallest values, in ascending order,

```
5 smallest values from 100000
0: 0.000003489200025797
1: 0.000008199829608202
2: 0.000008953968062997
3: 0.000010712770745158
4: 0.000033531803637743
```

12.6 References and Further Reading

The subject of sorting is covered extensively in the following,

- Donald E. Knuth, The Art of Computer Programming: Sorting and Searching (Vol 3, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850.

The Heapsort algorithm is described in the following book,

- Robert Sedgewick, Algorithms in C, Addison-Wesley, ISBN 0201514257.

제 13 장

BLAS 지원

기본 선형 대수학 서브 프로그램(Basic Linear Algebra Subprograms; BLAS)에서는 최적화된 고차원 선형 대수학 기능을 위한 행렬-벡터 연산들을 제공합니다.

이 라이브러리는 C 언어 BLAS 표준(CBLAS라고 합니다)에 해당하는 저수준의 기능들과 고수준의 GSL 벡터, 행렬들을 위한 기능들을 제공합니다. GSL 벡터와 행렬의 연산을 하고자 한다면, 여기서 기술하는 고차원 기능들을 사용할 수 있습니다. 이 단원에서 서술하는 함수들은 `gsl_blas.h` 헤더 파일에 정의되어 있고 대부분의 사용자들의 요구를 충족시킬 수 있습니다.

알아둘 점은 GSL 행렬들은 밀집 저장(dense-storage)을 사용하여, 해당하는 BLAS 함수들만이 구현되었다는 점입니다. 밴드 형식(band-format)이나 팩 형식(packed-format))의 기능을 포함하는 모든 BLAS 기능은 저 수준의 CBLAS 인터페이스를 통해 사용할 수 있습니다. 비슷하게 GSL 벡터는 양수 크기의 걸음을 가지지만, 저 수준의 CBLAS 인터페이스는 BLAS 표준에 정의된 음수 크기의 걸음을 지원합니다.

BLAS 연산은 3가지로 분류됩니다.

BLAS 연산 단계	설명
Level 1	벡터 연산, $y = \alpha x + y$
Level 2	행렬-벡터 연산. $y = \alpha A x + \beta y$
Level 3	행렬-행렬 연산, $C = \alpha A B + C$

각각의 단계의 함수들은 연산을 분류하는 특별한 이름들을 가지고 있습니다. 행렬의 형태와 정확도 등을 포함합니다. 가장 흔히 쓰이는 연산들은 아래와 같습니다.

이름	설명
DOT	스칼라곱, $x^T y$
AXPY	벡터 합, $\alpha x + y$
MV	행렬-벡터 곱, Ax
SV	행렬-벡터 풀이, $A^{-1}x$
MM	행렬-행렬 곱, AB
SM	행렬-행렬 풀이, $A^{-1}B$

행렬의 특성은,

이름	설명
GE	일반 행렬
GB	일반 밴드(band)
SY	대칭 행렬
SB	대칭 밴드
SP	대칭 팩
HE	에르미트 행렬
HB	에르미트 밴드
HP	에르미트 팩
TR	삼각 행렬
TB	삼각 밴드
TP	삼각 팩

각각의 연산은 4가지 정밀도로 구분됩니다.

이름	설명
S	단 정밀도 실수
D	배 정밀도 실수
C	단 정밀도 복소수
Z	배 정밀도 복소수

예를 들어서, $SGEMM$ 는 "단 정밀도 실수 일반 행렬-행렬 곱"을 의미하고 `ZGEMM` 는 "배 정밀도 복소수 일반 행렬-행렬 곱"을 의미합니다.

참고: BLAS 함수들에 인자로 들어가는 벡터와 행렬들은 별칭(alias)된 상태가 아니여야합니다. 이러한 중첩 상태에 있을 시 결과가 정의되지 않습니다(배열 별칭).

13.1 GSL BLAS 인터페이스

GSL은 관련된 자료형에 기반한 밀집 벡터, 행렬 객체들을 제공합니다. 라이브러리는 이러한 객체들에 적용할 수 있는 BLAS 연산자들에 대한 인터페이스를 제공합니다. 이러한 기능들의 인처페이스들은 `gsl_blas.h` 에 정의되어 있습니다.

13.1.1 Level 1 BLAS 인터페이스

int **gsl_blas_sdsdot**(float alpha, const gsl_vector_float *x, const gsl_vector_float *y, float *result)

$\alpha + x^T y$ 값을 주어진 벡터 x 와 y 대해 계산하고 결과를 $result$ 에 반환합니다.

int **gsl_blas_sdot**(const gsl_vector_float *x, const gsl_vector_float *y, float *result)

int **gsl_blas_dsdot**(const gsl_vector_float *x, const gsl_vector_float *y, double *result)

int **gsl_blas_ddot**(const gsl_vector *x, const gsl_vector *y, double *result)

스칼라 곱 $x^T y$ 를 주어진 벡터 x 와 y 대해 계산하고 결과를 $result$ 에 반환합니다.

int **gsl_blas_cdotu**(const gsl_vector_complex_float *x, const gsl_vector_complex_float *y, gsl_complex_float *dotu)

int **gsl_blas_zdotu**(const gsl_vector_complex *x, const gsl_vector_complex *y, gsl_complex *dotu)

복소수 스칼라 곱 $x^T y$ 를 주어진 벡터 x 와 y 대해 계산하고 결과를 $result$ 에 반환합니다.

int **gsl_blas_cdotc**(const gsl_vector_complex_float *x, const gsl_vector_complex_float *y, gsl_complex_float *dotc)

int **gsl_blas_zdotc**(const gsl_vector_complex *x, const gsl_vector_complex *y, gsl_complex *dotc)

켤레 복소 스클라 곱 $x^H y$ 를 주어진 벡터 x 와 y 대해 계산하고 결과를 $dotc$ 에 반환합니다.

float **gsl_blas_snrm2**(const gsl_vector_float *x)

double **gsl_blas_dnrm2**(const gsl_vector *x)

유클리드 노름 $\|x\|_2 = \sqrt{\sum x_i^2}$ 의 값을 주어진 벡터 x 에 대해 계산합니다.

float **gsl_blas_scnrm2**(const gsl_vector_complex_float *x)

double **gsl_blas_dznrm2**(const gsl_vector_complex *x)

유클리드 노름 $\|x\|_2 = \sqrt{\sum \Re(x_i)^2 + \Im(x_i)^2}$ 의 값을 주어진 복소수 벡터 x 에 대해 계산합니다.

float **gsl_blas_sasum**(const gsl_vector_float *x)

double **gsl_blas_dasum**(const gsl_vector *x)

절대값 급수 $\sum |x_i|$ 의 값을 주어진 벡터 x 에 대해 계산합니다.

float **gsl_blas_scasum**(const gsl_vector_complex_float *x)

double **gsl_blas_dzasum**(const gsl_vector_complex *x)

실수, 허수의 크기 급수 $\sum(|\Re(x_i)| + |\Im(x_i)|)$ 의 값을 주어진 복소수 벡터 x 에 대해 계산합니다.

CBLAS_INDEX_t **gsl_blas_isamax**(const gsl_vector_float *x)

CBLAS_INDEX_t **gsl_blas_idamax**(const gsl_vector *x)

CBLAS_INDEX_t **gsl_blas_icamax**(const gsl_vector_complex_float *x)

CBLAS_INDEX_t **gsl_blas_izamax**(const gsl_vector_complex *x)

주어진 벡터 x 원소중 가장 큰 원소의 인덱스를 반환합니다. 가장 큰 원소는 실수 벡터의 경우 원소의 절대값의 크기가 가장 큰 원소를, 복소수 벡터의 경우 실, 허수 부분의 크기의 합 $|\Re(x_i)| + |\Im(x_i)|$ 이 가장 큰 원소로 결정됩니다. 만약— 가장 큰 원소가 여러개 있다면, 그 중 첫번째 원소가 반환됩니다.

int **gsl_blas_sswap**(gsl_vector_float *x, gsl_vector_float *y)

int **gsl_blas_dswap**(gsl_vector *x, gsl_vector *y)

int **gsl_blas_cswap**(gsl_vector_complex_float *x, gsl_vector_complex_float *y)

int **gsl_blas_zswap**(gsl_vector_complex *x, gsl_vector_complex *y)

주어진 벡터 x 와 y 의 원소들을 교환합니다.

int **gsl_blas_scopy**(const gsl_vector_float *x, gsl_vector_float *y)

int **gsl_blas_dcopy**(const gsl_vector *x, gsl_vector *y)

int **gsl_blas_ccopy**(const gsl_vector_complex_float *x, gsl_vector_complex_float *y)

int **gsl_blas_zcopy**(const gsl_vector_complex *x, gsl_vector_complex *y)

주어진 벡터 x 원소들을 y 로 복사합니다.

int **gsl_blas_saxpy**(float alpha, const gsl_vector_float *x, gsl_vector_float *y)

int **gsl_blas_daxpy**(double alpha, const gsl_vector *x, gsl_vector *y)

int **gsl_blas_caxpy**(const gsl_complex_float alpha, const gsl_vector_complex_float *x, gsl_vector_complex_float *y)

int **gsl_blas_zaxpy**(const gsl_complex alpha, const gsl_vector_complex *x, gsl_vector_complex *y)

$y = \alpha x + y$ 의 값을 주어진 벡터 x 와 y 에 대해 계산합니다.

void **gsl_blas_sscal**(float alpha, gsl_vector_float *x)

void **gsl_blas_dscal**(double alpha, gsl_vector *x)

void **gsl_blas_cscal**(const gsl_complex_float alpha, gsl_vector_complex_float *x)

void **gsl_blas_zscal**(const gsl_complex alpha, gsl_vector_complex *x)

void **gsl_blas_csscal**(float alpha, gsl_vector_complex_float *x)

void **gsl_blas_zdscal**(double alpha, gsl_vector_complex *x)

벡터 x 원소들의 크기를 주어진 값 $alpha$ 를 곱해 변경합니다.

int **gsl_blas_srotg**(float a[], float b[], float c[], float s[])

int **gsl_blas_drotg**(double a[], double b[], double c[], double s[])

벡터 (a, b) 를 0으로 만드는 기븐스(Givens) 회전 (c, s) 를 계산합니다.

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r' \\ 0 \end{pmatrix}$$

변수 a 와 b 는 명령어에 의해 계산과정에서 여러번 초기화됩니다.

int **gsl_blas_srot**(gsl_vector_float *x, gsl_vector_float *y, float c, float s)

int **gsl_blas_drot**(gsl_vector *x, gsl_vector *y, const double c, const double s)

회전 $(x', y') = (cx + sy, -sx + cy)$ 을 주어진 벡터 x 와 y 에 대해 적용합니다.

int **gsl_blas_srotmg**(float d1[], float d2[], float b1[], float b2, float P[])

int **gsl_blas_drotmg**(double d1[], double d2[], double b1[], double b2, double P[])

수정 기븐스 변환을 계산합니다. 수정 기븐스 변환은 Level-1 BLAS specification에 정의되어 있습니다.

int **gsl_blas_srotm**(gsl_vector_float *x, gsl_vector_float *y, const float P[])

int **gsl_blas_drotm**(gsl_vector *x, gsl_vector *y, const double P[])

수정 기븐슨 변환을 적용합니다.

13.1.2 Level 2 BLAS 인터페이스

int **gsl_blas_sgemv**(CBLAS_TRANSPOSE_t TransA, float alpha, const gsl_matrix_float *A, const gsl_vector_float *x, float beta, gsl_vector_float *y)

int **gsl_blas_dgemv**(CBLAS_TRANSPOSE_t TransA, double alpha, const gsl_matrix *A, const gsl_vector *x, double beta, gsl_vector *y)

int **gsl_blas_cgemv**(CBLAS_TRANSPOSE_t TransA, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_vector_complex_float *x, const gsl_complex_float beta, gsl_vector_complex_float *y)

int **gsl_blas_zgemv**(CBLAS_TRANSPOSE_t TransA, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_vector_complex *x, const gsl_complex beta, gsl_vector_complex *y)

행렬-벡터 사이의 곱, 덧셈인 $y = \alpha op(A)x + \beta y$ 을 계산합니다. `op(A) = A, A^T, A.h` 이고 `TransA = CblasNoTrans , CblasTrans , CblasConjTrans` 가 가능합니다.

int **gsl_blas_strmv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_float *A, gsl_vector_float *x)

int **gsl_blas_dtrmv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix *A, gsl_vector *x)

int **gsl_blas_ctrmv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex_float *A, gsl_vector_complex_float *x)

int **gsl_blas_ztrmv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex *A, gsl_vector_complex *x)

행렬-벡터 곱 $x = op(A)x$ 를 삼각 행렬 A 대해 계산합니다. `op(A) = A, A^T, A.h` 이고 `TransA = CblasNoTrans , CblasTrans , CblasConjTrans` 가 가능합니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 의 상삼각 행렬이 사용되고, `CblasLower` 라면, A 의 하삼각 행렬이 사용됩니다. 만약, `Dig` 가 `CblasNonUnit` 라면 행렬의 대각 성분들이 사용됩니다. `CblasUnit` 라면 행렬 A 의 대각 성분들은 유니터리 행렬의 대각성분으로 취급되기 때문에 참고되지 않습니다.

int **gsl_blas_strsv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_float *A, gsl_vector_float *x)

int **gsl_blas_dtrsv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix *A, gsl_vector *x)

int **gsl_blas_ctrsv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex_float *A, gsl_vector_complex_float *x)

int **gsl_blas_ztrsv**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex *A, gsl_vector_complex *x)

주어진 벡터 x 대해, $inv(op(A))x$ 를 계산합니다. `op(A) = A, A^T, A.h` 이고 `TransA = CblasNoTrans , CblasTrans , CblasConjTrans` 가 가능합니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 의 상삼각 행렬이 사용되고, `CblasLower` 라면, 행렬 A 의 하삼각 행렬이 사용됩니다. 만약, `Dig`가 :code:`CblasNonUnit 라면 행렬의 대각 성분들이 사용됩니다. `CblasUnit` 라면 행렬 A 의 대각 성분들은 유니터리 행렬의 대각성분으로 취급되기 때문에 참고되지 않습니다.

int **gsl_blas_ssymv**(CBLAS_UPLO_t Uplo, float alpha, const gsl_matrix_float *A, const gsl_vector_float *x, float beta, gsl_vector_float *y)

int **gsl_blas_dsymv**(CBLAS_UPLO_t Uplo, double alpha, const gsl_matrix *A, const gsl_vector *x, double beta, gsl_vector *y)

행렬-벡터 곱과 합 $y = \alpha Ax + \beta y$ 을 대칭 행렬 A 에 대해 계산합니다. 행렬 A 는 대칭 행렬이기 때문에, 상삼각 부분이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 의 상삼각 행렬이 사용되고, `CblasLower` 라면, 행렬 A 의 하삼각 행렬이 사용됩니다.

int **gsl_blas_chemv**(CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_vector_complex_float *x, const gsl_complex_float beta, gsl_vector_complex_float *y)

int **gsl_blas_zhemv**(CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_vector_complex *x, const gsl_complex beta, gsl_vector_complex *y)

행렬-벡터 곱과 합 $y = \alpha A x + \beta y$ 을 에르미트 행렬 A 에 대해 계산합니다. 행렬 A 는 대칭 행렬이기 때문에, 상삼각 부분이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 의 상삼각 행렬이 사용되고, `CblasLower` 라면, 행렬 A 의 하삼각 행렬이 사용됩니다. 대각 성분의 복소 성분들은 자동적으로 0으로 가정하고 참고되지 않습니다.

int **gsl_blas_sger**(float alpha, const gsl_vector_float *x, const gsl_vector_float *y, gsl_matrix_float *A)

int **gsl_blas_dger**(double alpha, const gsl_vector *x, const gsl_vector *y, gsl_matrix *A)

int **gsl_blas_cgeru**(const gsl_complex_float alpha, const gsl_vector_complex_float *x, const gsl_vector_complex_float *y, gsl_matrix_complex_float *A)

int **gsl_blas_zgeru**(const gsl_complex alpha, const gsl_vector_complex *x, const gsl_vector_complex *y, gsl_matrix_complex *A)

행렬 A 대해, 랭크-1 갱신 $A = \alpha x y^T + A$ 를 계산합니다.

int **gsl_blas_cgerc**(const gsl_complex_float alpha, const gsl_vector_complex_float *x, const gsl_vector_complex_float *y, gsl_matrix_complex_float *A)

int **gsl_blas_zgerc**(const gsl_complex alpha, const gsl_vector_complex *x, const gsl_vector_complex *y, gsl_matrix_complex *A)

행렬 A 대해, 컬레 랭크-1 갱신 $A = \alpha x y^H + A$ 를 계산합니다.

int **gsl_blas_ssyr**(CBLAS_UPLO_t Uplo, float alpha, const gsl_vector_float *x, gsl_matrix_float *A)

int **gsl_blas_dsyr**(CBLAS_UPLO_t Uplo, double alpha, const gsl_vector *x, gsl_matrix *A)

대칭 행렬 A 대해, 대칭 랭크-1 갱신 $A = \alpha x x^T + A$ 를 계산합니다. 행렬 A 는 대칭 행렬이기 때문에, 상삼각 부분이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 는 상삼각 행렬과 대각 성분들이 사용되고, `CblasLower` 라면, A 는 하삼각 행렬과 대각 성분들이 사용됩니다.

int **gsl_blas_cher**(CBLAS_UPLO_t Uplo, float alpha, const gsl_vector_complex_float *x, gsl_matrix_complex_float *A)

int **gsl_blas_zher**(CBLAS_UPLO_t Uplo, double alpha, const gsl_vector_complex *x, gsl_matrix_complex *A)

에르미트 행렬 A 대해, 에르미트 랭크-1 갱신 $A = \alpha x x^H + A$ 를 계산합니다. 행렬 A 는 에르미트 행렬이기 때문에, 상삼각 부분이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 는 상삼각 행렬과 대각 성분들이 사용되고, `CblasLower` 라면, A 는 하삼각 행렬과 대각 성분들이 사용됩니다. 대각 성분의 복소 성분들은 자동적으로 0으로 가정하고 참고되지 않습니다.

int **gsl_blas_ssyr2**(CBLAS_UPLO_t Uplo, float alpha, const gsl_vector_float *x, const gsl_vector_float *y, gsl_matrix_float *A)

int **gsl_blas_dsyr2**(CBLAS_UPLO_t Uplo, double alpha, const gsl_vector *x, const gsl_vector *y, gsl_matrix *A)

대칭 행렬 A 대해, 대칭 랭크-2 갱신 $A = \alpha xy^T + \alpha yx^T + A$ 를 계산합니다. 행렬 A 는 대칭 행렬이기 때문에, 상삼각 부분이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 는 상삼각 행렬과 대각 성분들이 사용되고, `CblasLower` 라면, A 는 하삼각 행렬과 대각 성분들이 사용됩니다.

int **gsl_blas_cher2**(CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_vector_complex_float *x, const gsl_vector_complex_float *y, gsl_matrix_complex_float *A)

int **gsl_blas_zher2**(CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_vector_complex *x, const gsl_vector_complex *y, gsl_matrix_complex *A)

에르미트 행렬 A 대해, 에르미트 랭크-2갱신 $A = \alpha xy^H + \alpha yx^H + A$ 를 계산합니다. 행렬 A 는 에르미트 행렬이기 때문에, 상삼각 부분이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 일 때, 행렬 A 는 상삼각 행렬과 대각 성분들이 사용되고, `CblasLower` 라면, A 는 하삼각 행렬과 대각 성분들이 사용됩니다. 대각 성분의 복소 성분들은 자동으로 0으로 가정하고 참고되지 않습니다.

13.1.3 Level 3 BLAS 인터페이스

int **gsl_blas_sgemm**(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, float alpha, const gsl_matrix_float *A, const gsl_matrix_float *B, float beta, gsl_matrix_float *C)

int **gsl_blas_dgemm**(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, double alpha, const gsl_matrix *A, const gsl_matrix *B, double beta, gsl_matrix *C)

int **gsl_blas_cgemm**(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, const gsl_complex_float beta, gsl_matrix_complex_float *C)

int **gsl_blas_zgemm**(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_matrix_complex *B, const gsl_complex beta, gsl_matrix_complex *C)

행렬-행렬 사이의 곱과 합 $C = \alpha op(A) op(B) + \beta C$ 를 계산합니다. `op(A) = A, A^T, A.h` 이고 `TransA = CblasNoTrans , CblasTrans , CblasConjTrans` 가 가능합니다. $TransB$ 같은 인자를 사용가능합니다.

int **gsl_blas_ssymm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, float alpha, const gsl_matrix_float *A, const gsl_matrix_float *B, float beta, gsl_matrix_float *C)

int **gsl_blas_dsymm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, double alpha, const gsl_matrix *A, const gsl_matrix *B, double beta, gsl_matrix *C)

int **gsl_blas_csymm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, const gsl_complex_float beta, gsl_matrix_complex_float *C)

int **gsl_blas_zsymm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_matrix_complex *B, const gsl_complex beta, gsl_matrix_complex *C)

행렬-행렬 사이의 곱과 합을 계산합니다. `Side` 가 $CblasLeft$ 일 때 $C = \alpha AB + \beta C$ 를, $CblsRight$ 면 $C = \alpha BA + \beta C$ 를 계산합니다. 행렬 A 는 대칭 행렬이어야 합니다. `Uplo` 가 `CblasUpper` 때, 행렬 A 는 상삼각 행렬과 대각 성분들이 사용되고, `CblasLower` 때, 행렬 A 는 하삼각 행렬과 대각 성분들이 사용됩니다. 대각 성분의 복소 성분들은 자동적으로 0으로 가정하고 참고되지 않습니다.

int **gsl_blas_chemm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, const gsl_complex_float beta, gsl_matrix_complex_float *C)

int **gsl_blas_zhemm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_matrix_complex *B, const gsl_complex beta, gsl_matrix_complex *C)

행렬-행렬 곱을 계산합니다. `Side` 가 $CblasLeft$ 일 때 $B = \alpha op(A)B$ 를, `CblsRight` 면 $B = \alpha Bop(A)$ 를 계산합니다. 행렬 A 는 에르미트 행렬이어야 합니다. `Uplo` 가 `CblasUpper` 때, 행렬 A 는 상삼각 부분과 대각 부분이 사용되고, `CblasLower` 때, 하삼각 부분과 대각 부분이 사용됩니다. 허수 성분의 대각 성분은 자동으로 0으로 취급됩니다.

int **gsl_blas_strmm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, float alpha, const gsl_matrix_float *A, gsl_matrix_float *B)

int **gsl_blas_dtrmm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, double alpha, const gsl_matrix *A, gsl_matrix *B)

int **gsl_blas_ctrmm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, gsl_matrix_complex_float *B)

int **gsl_blas_ztrmm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex alpha, const gsl_matrix_complex *A, gsl_matrix_complex *B)

행렬-행렬 곱을 계산합니다. Side 가 $CblasLeft$ 일 때 $B = \alpha op(A)B$ 를, $CblasRight$ 면 $B = \alpha Bop(A)$ 를 계산합니다. 행렬 A 는 삼각행렬이어야 하고, TransA 가 CblasNoTrans , CblasTrans , CblasConkTrans 인 경우 각각 op(A) = A, A^T, A.h 를 의미합니다. Uplo 가 CblasUpper 인 경우 행렬 A 는 상삼각 부분이 사용되고, CblasLower 인 경우 A 는 하삼각 부분이 사용됩니다. 만약, $Diag$ 가 CblasNonUnit 라면 행렬 A 는 대각 성분이 사용되고, CblasUnit 라면 유니터리 행렬로 취급하여 대각 성분은 참고되지 않습니다.

int **gsl_blas_strsm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, float alpha, const gsl_matrix_float *A, gsl_matrix_float *B)

int **gsl_blas_dtrsm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, double alpha, const gsl_matrix *A, gsl_matrix *B)

int **gsl_blas_ctrsm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, gsl_matrix_complex_float *B)

int **gsl_blas_ztrsm**(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex alpha, const gsl_matrix_complex *A, gsl_matrix_complex *B)

역행렬의 행렬곱을 계산합니다. Side 가 $CblasLeft$ 일 때 $B = \alpha op(inv(A))B$ 를, $CblasRight$ 경우에는 $B = \alpha Bop(inv(A))$ 를 계산합니다. 행렬 A 는 삼각 행렬이어야 하고 TransA 가 CblasNoTrans , CblasTrans , CblasConkTrans 인 경우 각각 op(A) = A, A^T, A.h 를 의미합니다. Uplo 가 CblasUpper 인 경우 행렬 A 는 상삼각 부분이 사용되고, CblasLower 인 경우 A 는 하삼각 부분이 사용됩니다. 만약, $Diag$ 가 CblasNonUnit 라면 행렬 A 는 대각 성분이 사용되고, CblasUnit 라면 유니터리 행렬로 취급하여 취급하여 대각 성분은 참고되지 않습니다.

int **gsl_blas_ssyrk**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_float *A, float beta, gsl_matrix_float *C)

int **gsl_blas_dsyrk**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix *A, double beta, gsl_matrix *C)

int **gsl_blas_csyrk**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_complex_float beta, gsl_matrix_complex_float *C)

int **gsl_blas_zsyrk**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_complex beta, gsl_matrix_complex *C)

대칭 행렬 C 랭크-k 갱신을 계산합니다. Trans 가 CblasNoTrans 라면 $C = \alpha AA^T + \beta C$ 을, CblasTrans 면 $C = \alpha A^T A + \beta C$ 를 계산합니다. 행렬 C 대칭 행렬이기 때문에 상삼각이나 하삼각 부분만 저장해도 됩니다. Uplo 가 CblasUpper 인 경우 행렬 C 상삼각 부분과 대각 부분이 사용되고, CblasLower 라면 C 의 하삼각 부분과 대각 부분이 사용됩니다.

int **gsl_blas_cherk**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_complex_float *A, float beta, gsl_matrix_complex_float *C)

int **gsl_blas_zherk**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix_complex *A, double beta, gsl_matrix_complex *C)

에르미트 행렬 C 랭크-k 갱신을 계산합니다. `Trans` 가 `CblasNoTrans` 라면 $C = \alpha A A^H + \beta C$ 을, `CblasTrans` 면 $C = \alpha A^H A + \beta C$ 를 계산합니다. 행렬 C 에르미트 행렬이기 때문에 상삼각이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 행렬 C 상삼각 부분과 대각 부분이 사용되고, `CblasLower` 라면 C 의 하삼각 부분과 대각 부분이 사용됩니다.

int **gsl_blas_ssyr2k**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_float *A, const gsl_matrix_float *B, float beta, gsl_matrix_float *C)

int **gsl_blas_dsyr2k**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix *A, const gsl_matrix *B, double beta, gsl_matrix *C)

int **gsl_blas_csyr2k**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, const gsl_complex_float beta, gsl_matrix_complex_float *C)

int **gsl_blas_zsyr2k**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_matrix_complex *B, const gsl_complex beta, gsl_matrix_complex *C)

대칭 행렬 C 랭크-2k 갱신을 계산합니다. `Trans` 가 `CblasNoTrans` 라면 $C = \alpha A B^T + \alpha B A^T + \beta C$ 을, `CblasTrans` 면 $C = \alpha A^T B + \alpha B^T A + \beta C$ 를 계산합니다. 행렬 C 대칭 행렬이기 때문에 상삼각이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 인 경우 행렬 C 상삼각 부분과 대각 부분이 사용되고, `CblasLower` 라면 C 의 하삼각 부분과 대각 부분이 사용됩니다.

int **gsl_blas_cher2k**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, float beta, gsl_matrix_complex_float *C)

int **gsl_blas_zher2k**(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const gsl_matrix_complex *A, const gsl_matrix_complex *B, double beta, gsl_matrix_complex *C)

에르미트 행렬 C 의 랭크-2k 갱신을 계산합니다. `Trans` 가 `CblasNoTrans` 라면 $C = \alpha A B^T + \alpha B A^T + \beta C$ 을, `CblasTrans` 면 $C = \alpha A^T B + \alpha B^T A + \beta C$ 를 계산합니다. 행렬 C 에르미트 행렬이기 때문에 상삼각이나 하삼각 부분만 저장해도 됩니다. `Uplo` 가 `CblasUpper` 라면 행렬 C 의 상삼각 부분과 대각 부분이 사용되고, `CblasLower` 라면 C 의 하삼각 부분과 대각 부분이 사용됩니다.

13.2 예제

다음 프로그램은 Level-3 BLAS 함수 DGEMM를 사용해 두 개의 행렬의 곱을 계산합니다.

$$\begin{pmatrix} 0.11 & 0.12 & 0.13 \\ 0.21 & 0.22 & 0.23 \end{pmatrix} \begin{pmatrix} 1011 & 1012 \\ 1021 & 1022 \\ 1031 & 1031 \end{pmatrix} = \begin{pmatrix} 367.76 & 368.12 \\ 674.06 & 674.72 \end{pmatrix}$$

행렬은 배열의 C 표준 순서에 따라, 열을 기준으로 배열합니다.

```
#include <stdio.h>
#include <gsl/gsl_blas.h>

int
main (void)
{
  double a[] = { 0.11, 0.12, 0.13,
                 0.21, 0.22, 0.23 };

  double b[] = { 1011, 1012,
                 1021, 1022,
                 1031, 1032 };

  double c[] = { 0.00, 0.00,
                 0.00, 0.00 };

  gsl_matrix_view A = gsl_matrix_view_array(a, 2, 3);
  gsl_matrix_view B = gsl_matrix_view_array(b, 3, 2);
  gsl_matrix_view C = gsl_matrix_view_array(c, 2, 2);

  /* Compute C = A B */

  gsl_blas_dgemm (CblasNoTrans, CblasNoTrans,
                  1.0, &A.matrix, &B.matrix,
                  0.0, &C.matrix);

  printf ("[ %g, %g\n", c[0], c[1]);
  printf ("  %g, %g ]\n", c[2], c[3]);

  return 0;
}
```

다음은 프로그램의 출력 결과입니다.

```
[ 367.76, 368.12
  674.06, 674.72 ]
```

13.2.1 참고 문헌과 추가 자료

기존 인터페이스 표준과 업데이트 된 인터페이스 표준을 모두 포함한 BLAS 표준에 대한 자료는 BLAS 홈페이지 및 BLAS 기술 포럼 웹 사이트에서 온라인으로 확인할 수 있습니다.

- BLAS Homepage
- BLAS Technical Forum

BLAS의 Level 1,2,3의 자세한 사양을 확안하고 싶다면 다음 문서를 확인하시길 바랍니다.

- C. Lawson, R. Hanson, D. Kincaid, F. Krogh, “Basic Linear Algebra Subprograms for Fortran Usage”, ACM Transactions on Mathematical Software, Vol.: 5 (1979), Pages 308-325.
- J.J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson, “An Extended Set of Fortran Basic Linear Algebra Subprograms”, ACM Transactions on Mathematical Software, Vol.: 14, No.: 1 (1988), Pages 1-32.
- J.J. Dongarra, I. Duff, J. DuCroz, S. Hammarling, “A Set of Level 3 Basic Linear Algebra Subprograms”, ACM Transactions on Mathematical Software, Vol.: 16 (1990), Pages 1-28.

마지막 두 개 문서의 postscript 버전은 http://www.netlib.org/blas/ 에서 확인할 수 있습니다. Fortran BLAS 라이브러리의 CBLAS 이식 버전의 정보도 동일한 곳에서 사용할 수 있습니다.

제 14 장

선형 대수학

참고: 번역중

This chapter describes functions for solving linear systems. The library provides linear algebra operations which operate directly on the `gsl_vector` and `gsl_matrix` objects. These routines use the standard algorithms from Golub & Van Loan's Matrix Computations with Level-1 and Level-2 BLAS calls for efficiency.

The functions described in this chapter are declared in the header file `gsl_linalg.h`.

14.1 LU Decomposition

A general M-by-N matrix A has an LU decomposition

$$PA = LU$$

where P is an M-by-M permutation matrix, L is M-by-$\min(M,N)$ and U is $\min(M,N)$-by-N. For square matrices, L is a lower unit triangular matrix and U is upper triangular. For $M > N$, L is a unit lower trapezoidal matrix of size M-by-N. For $M < N$, U is upper trapezoidal of size M-by-N. For square matrices this decomposition can be used to convert the linear system $Ax = b$ into a pair of triangular systems ($Ly = Pb$, $Ux = y$), which can be solved by forward and back-substitution. Note that the LU decomposition is valid for singular matrices.

int **gsl_linalg_LU_decomp**(gsl_matrix *A, gsl_permutation *p, int *signum)

int **gsl_linalg_complex_LU_decomp**(gsl_matrix_complex *A, gsl_permutation *p, int *signum)

These functions factorize the matrix `A` into the LU decomposition $PA = LU$. On output

the diagonal and upper triangular (or trapezoidal) part of the input matrix `A` contain the matrix U. The lower triangular (or trapezoidal) part of the input matrix (excluding the diagonal) contains L. The diagonal elements of L are unity, and are not stored.

The permutation matrix P is encoded in the permutation `p` on output. The j-th column of the matrix P is given by the k-th column of the identity matrix, where $k = p_j$ the j-th element of the permutation vector. The sign of the permutation is given by `signum`. It has the value $(-1)^n$, where n is the number of interchanges in the permutation.

The algorithm used in the decomposition is Gaussian Elimination with partial pivoting (Golub & Van Loan, Matrix Computations, Algorithm 3.4.1), combined with a recursive algorithm based on Level 3 BLAS (Peise and Bientinesi, 2016).

int **gsl_linalg_LU_solve**(const gsl_matrix *LU, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x)

int **gsl_linalg_complex_LU_solve**(const gsl_matrix_complex *LU, const gsl_permutation *p, const gsl_vector_complex *b, gsl_vector_complex *x)

These functions solve the square system $Ax = b$ using the LU decomposition of A into (`LU`, `p`) given by `gsl_linalg_LU_decomp()` or `gsl_linalg_complex_LU_decomp()` as input.

int **gsl_linalg_LU_svx**(const gsl_matrix *LU, const gsl_permutation *p, gsl_vector *x)

int **gsl_linalg_complex_LU_svx**(const gsl_matrix_complex *LU, const gsl_permutation *p, gsl_vector_complex *x)

These functions solve the square system $Ax = b$ in-place using the precomputed LU decomposition of A into (`LU`, `p`). On input `x` should contain the right-hand side b, which is replaced by the solution on output.

int **gsl_linalg_LU_refine**(const gsl_matrix *A, const gsl_matrix *LU, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x, gsl_vector *work)

int **gsl_linalg_complex_LU_refine**(const gsl_matrix_complex *A, const gsl_matrix_complex *LU, const gsl_permutation *p, const gsl_vector_complex *b, gsl_vector_complex *x, gsl_vector_complex *work)

These functions apply an iterative improvement to `x`, the solution of $Ax = b$, from the precomputed LU decomposition of A into (`LU`, `p`). Additional workspace of length `N` is required in `work`.

int **gsl_linalg_LU_invert**(const gsl_matrix *LU, const gsl_permutation *p, gsl_matrix *inverse)

int **gsl_linalg_complex_LU_invert**(const gsl_matrix_complex *LU, const gsl_permutation *p, gsl_matrix_complex *inverse)

These functions compute the inverse of a matrix A from its LU decomposition (`LU`, `p`),

storing the result in the matrix `inverse`. The inverse is computed by computing the inverses U^{-1}, L^{-1} and finally forming the product $A^{-1} = U^{-1}L^{-1}P$. Each step is based on Level 3 BLAS calls.

It is preferable to avoid direct use of the inverse whenever possible, as the linear solver functions can obtain the same result more efficiently and reliably (consult any introductory textbook on numerical linear algebra for details).

int **gsl_linalg_LU_invx**(gsl_matrix *LU, const gsl_permutation *p)

int **gsl_linalg_complex_LU_invx**(gsl_matrix_complex *LU, const gsl_permutation *p)

These functions compute the inverse of a matrix A from its LU decomposition (`LU`, `p`), storing the result in-place in the matrix `LU`. The inverse is computed by computing the inverses U^{-1}, L^{-1} and finally forming the product $A^{-1} = U^{-1}L^{-1}P$. Each step is based on Level 3 BLAS calls.

It is preferable to avoid direct use of the inverse whenever possible, as the linear solver functions can obtain the same result more efficiently and reliably (consult any introductory textbook on numerical linear algebra for details).

double **gsl_linalg_LU_det**(gsl_matrix *LU, int signum)

gsl_complex **gsl_linalg_complex_LU_det**(gsl_matrix_complex *LU, int signum)

These functions compute the determinant of a matrix A from its LU decomposition, `LU`. The determinant is computed as the product of the diagonal elements of U and the sign of the row permutation `signum`.

double **gsl_linalg_LU_lndet**(gsl_matrix *LU)

double **gsl_linalg_complex_LU_lndet**(gsl_matrix_complex *LU)

These functions compute the logarithm of the absolute value of the determinant of a matrix A, $\ln|\det(A)|$, from its LU decomposition, `LU`. This function may be useful if the direct computation of the determinant would overflow or underflow.

int **gsl_linalg_LU_sgndet**(gsl_matrix *LU, int signum)

gsl_complex **gsl_linalg_complex_LU_sgndet**(gsl_matrix_complex *LU, int signum)

These functions compute the sign or phase factor of the determinant of a matrix A, $\det(A)/|\det(A)|$, from its LU decomposition, `LU`.

14.2 QR Decomposition

A general rectangular M-by-N matrix A has a QR decomposition into the product of a unitary M-by-M square matrix Q (where $Q^\dagger Q = I$) and an M-by-N right-triangular matrix R,

$$A = QR$$

This decomposition can be used to convert the square linear system $Ax = b$ into the triangular system $Rx = Q^\dagger b$, which can be solved by back-substitution. Another use of the QR decomposition is to compute an orthonormal basis for a set of vectors. The first N columns of Q form an orthonormal basis for the range of A, $ran(A)$, when A has full column rank.

When $M > N$, the bottom $M - N$ rows of R are zero, and so A can be naturally partioned as

$$A = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = Q_1 R_1$$

where R_1 is N-by-N upper triangular, Q_1 is M-by-N, and Q_2 is M-by-$(M-N)$. $Q_1 R_1$ is sometimes called the thin or reduced QR decomposition. The solution of the least squares problem $\min_x ||b - Ax||^2$ when A has full rank is:

$$x = R_1^{-1} c_1$$

where c_1 is the first N elements of $Q^\dagger b$. If A is rank deficient, see QR Decomposition with Column Pivoting and Complete Orthogonal Decomposition.

GSL offers two interfaces for the QR decomposition. The first proceeds by zeroing out columns below the diagonal of A, one column at a time using Householder transforms. In this method, the factor Q is represented as a product of Householder reflectors:

$$Q = H_n \cdots H_2 H_1$$

where each $H_i = I - \tau_i v_i v_i^\dagger$ for a scalar τ_i and column vector v_i. In this method, functions which compute the full matrix Q or apply $Q^\dagger$ to a right hand side vector operate by applying the Householder matrices one at a time using Level 2 BLAS.

The second interface is based on a Level 3 BLAS block recursive algorithm developed by Elmroth and Gustavson. In this case, Q is written in block form as

$$Q = I - VTV^\dagger$$

where V is an M-by-N matrix of the column vectors v_i and T is an N-by-N upper triangular

matrix, whose diagonal elements are the τ_i. Computing the full T, while requiring more flops than the Level 2 approach, offers the advantage that all standard operations can take advantage of cache efficient Level 3 BLAS operations, and so this method often performs faster than the Level 2 approach. The functions for the recursive block algorithm have a `_r` suffix, and it is recommended to use this interface for performance critical applications.

int **gsl_linalg_QR_decomp_r**(gsl_matrix *A, gsl_matrix *T)

int **gsl_linalg_complex_QR_decomp_r**(gsl_matrix_complex *A, gsl_matrix_complex *T)

These functions factor the M-by-N matrix `A` into the QR decomposition $A = QR$ using the recursive Level 3 BLAS algorithm of Elmroth and Gustavson. On output the diagonal and upper triangular part of `A` contain the matrix R. The N-by-N matrix `T` stores the upper triangular factor appearing in Q. The matrix Q is given by $Q = I - VTV^\dagger$, where the elements below the diagonal of `A` contain the columns of V on output.

This algorithm requires $M \geq N$ and performs best for "tall-skinny" matrices, i.e. $M \gg N$.

int **gsl_linalg_QR_solve_r**(const gsl_matrix *QR, const gsl_matrix *T, const gsl_vector *b, gsl_vector *x)

int **gsl_linalg_complex_QR_solve_r**(const gsl_matrix_complex *QR, const gsl_matrix_complex *T, const gsl_vector_complex *b, gsl_vector_complex *x)

These functions solve the square system $Ax = b$ using the QR decomposition of A held in (`QR`, `T`). The least-squares solution for rectangular systems can be found using `gsl_linalg_QR_lssolve_r()` or `gsl_linalg_complex_QR_lssolve_r()`.

int **gsl_linalg_QR_lssolve_r**(const gsl_matrix *QR, const gsl_matrix *T, const gsl_vector *b, gsl_vector *x, gsl_vector *work)

int **gsl_linalg_complex_QR_lssolve_r**(const gsl_matrix_complex *QR, const gsl_matrix_complex *T, const gsl_vector_complex *b, gsl_vector_complex *x, gsl_vector_complex *work)

These functions find the least squares solution to the overdetermined system $Ax = b$ where the matrix `A` has more rows than columns. The least squares solution minimizes the Euclidean norm of the residual, $||b - Ax||$. The routine requires as input the QR decomposition of A into (`QR`, `T`) given by `gsl_linalg_QR_decomp_r()` or `gsl_linalg_complex_QR_decomp_r()`. The parameter `x` is of length M. The solution x is returned in the first N rows of `x`, i.e. $x =$ `x[0], x[1], ..., x[N-1]`. The last $M - N$ rows of `x` contain a vector whose norm is equal to the residual norm $||b - Ax||$. This similar to the behavior of LAPACK DGELS. Additional workspace of length N is required in `work`.

int **gsl_linalg_QR_QTvec_r**(const gsl_matrix *QR, const gsl_matrix *T, gsl_vector *v, gsl_vector *work)

int **gsl_linalg_complex_QR_QHvec_r**(const gsl_matrix_complex *QR, const gsl_matrix_complex *T, gsl_vector_complex *v, gsl_vector_complex *work)

These functions apply the matrix Q^T (or $Q^\dagger$) encoded in the decomposition (QR, T) to the vector v, storing the result $Q^T v$ (or $Q^\dagger v$) in v. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q^T (or $Q^\dagger$). Additional workspace of size N is required in work.

int **gsl_linalg_QR_QTmat_r**(const gsl_matrix *QR, const gsl_matrix *T, gsl_matrix *B, gsl_matrix *work)

This function applies the matrix Q^T encoded in the decomposition (QR, T) to the M-by-K matrix B, storing the result $Q^T B$ in B. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q^T. Additional workspace of size N-by-K is required in work.

int **gsl_linalg_QR_unpack_r**(const gsl_matrix *QR, const gsl_matrix *T, gsl_matrix *Q, gsl_matrix *R)

int **gsl_linalg_complex_QR_unpack_r**(const gsl_matrix_complex *QR, const gsl_matrix_complex *T, gsl_matrix_complex *Q, gsl_matrix_complex *R)

These functions unpack the encoded QR decomposition (QR, T) as output from gsl_linalg_QR_decomp_r() or gsl_linalg_complex_QR_decomp_r() into the matrices Q and R, where Q is M-by-M and R is N-by-N. Note that the full R matrix is M-by-N, however the lower trapezoidal portion is zero, so only the upper triangular factor is stored.

int **gsl_linalg_QR_rcond**(const gsl_matrix *QR, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the R factor, stored in the upper triangle of QR. The reciprocal condition number estimate, defined as $1/(||R||_1 \cdot ||R^{-1}||_1)$, is stored in rcond. Additional workspace of size $3N$ is required in work.

14.2.1 Level 2 Interface

The functions below are for the slower Level 2 interface to the QR decomposition. It is recommended to use these functions only for $M < N$, since the Level 3 interface above performs much faster for $M \geq N$.

int **gsl_linalg_QR_decomp**(gsl_matrix *A, gsl_vector *tau)

int **gsl_linalg_complex_QR_decomp**(gsl_matrix_complex *A, gsl_vector_complex *tau)

These functions factor the M-by-N matrix A into the QR decomposition $A = QR$. On output the diagonal and upper triangular part of the input matrix contain the matrix R. The vector tau and the columns of the lower triangular part of the matrix A contain the Householder coefficients and Householder vectors which encode the orthogonal matrix

Q. The vector tau must be of length N. The matrix Q is related to these components by the product of $k = min(M, N)$ reflector matrices, $Q = H_k ... H_2 H_1$ where $H_i = I - \tau_i v_i v_i^\dagger$ and v_i is the Householder vector $v_i = (0, ..., 1, A(i+1, i), A(i+2, i), ..., A(m, i))$. This is the same storage scheme as used by LAPACK.

The algorithm used to perform the decomposition is Householder QR (Golub & Van Loan, "Matrix Computations", Algorithm 5.2.1).

int **gsl_linalg_QR_solve**(const gsl_matrix *QR, const gsl_vector *tau, const gsl_vector *b, gsl_vector *x)

int **gsl_linalg_complex_QR_solve**(const gsl_matrix_complex *QR, const gsl_vector_complex *tau, const gsl_vector_complex *b, gsl_vector_complex *x)

These functions solve the square system $Ax = b$ using the QR decomposition of A held in (QR, tau). The least-squares solution for rectangular systems can be found using gsl_linalg_QR_lssolve().

int **gsl_linalg_QR_svx**(const gsl_matrix *QR, const gsl_vector *tau, gsl_vector *x)

int **gsl_linalg_complex_QR_svx**(const gsl_matrix_complex *QR, const gsl_vector_complex *tau, gsl_vector_complex *x)

These functions solve the square system $Ax = b$ in-place using the QR decomposition of A held in (QR, tau). On input x should contain the right-hand side b, which is replaced by the solution on output.

int **gsl_linalg_QR_lssolve**(const gsl_matrix *QR, const gsl_vector *tau, const gsl_vector *b, gsl_vector *x, gsl_vector *residual)

int **gsl_linalg_complex_QR_lssolve**(const gsl_matrix_complex *QR, const gsl_vector_complex *tau, const gsl_vector_complex *b, gsl_vector_complex *x, gsl_vector_complex *residual)

These functions find the least squares solution to the overdetermined system $Ax = b$ where the matrix A has more rows than columns. The least squares solution minimizes the Euclidean norm of the residual, $||Ax - b||$.The routine requires as input the QR decomposition of A into (QR, tau) given by gsl_linalg_QR_decomp() or gsl_linalg_complex_QR_decomp(). The solution is returned in x. The residual is computed as a by-product and stored in residual.

int **gsl_linalg_QR_QTvec**(const gsl_matrix *QR, const gsl_vector *tau, gsl_vector *v)

int **gsl_linalg_complex_QR_QHvec**(const gsl_matrix_complex *QR, const gsl_vector_complex *tau, gsl_vector_complex *v)

These functions apply the matrix Q^T (or $Q^\dagger$) encoded in the decomposition (QR, tau) to the vector v, storing the result $Q^T v$ (or $Q^\dagger v$) in v. The matrix multiplication is carried out

directly using the encoding of the Householder vectors without needing to form the full matrix Q^T (or $Q^\dagger$).

int **gsl_linalg_QR_Qvec**(const gsl_matrix *QR, const gsl_vector *tau, gsl_vector *v)

int **gsl_linalg_complex_QR_Qvec**(const gsl_matrix_complex *QR, const gsl_vector_complex *tau, gsl_vector_complex *v)

These functions apply the matrix Q encoded in the decomposition (QR, tau) to the vector v, storing the result Qv in v. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q.

int **gsl_linalg_QR_QTmat**(const gsl_matrix *QR, const gsl_vector *tau, gsl_matrix *B)

This function applies the matrix Q^T encoded in the decomposition (QR, tau) to the M-by-K matrix B, storing the result $Q^T B$ in B. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q^T.

int **gsl_linalg_QR_Rsolve**(const gsl_matrix *QR, const gsl_vector *b, gsl_vector *x)

This function solves the triangular system $Rx = b$ for x. It may be useful if the product $b' = Q^T b$ has already been computed using gsl_linalg_QR_QTvec().

int **gsl_linalg_QR_Rsvx**(const gsl_matrix *QR, gsl_vector *x)

This function solves the triangular system $Rx = b$ for x in-place. On input x should contain the right-hand side b and is replaced by the solution on output. This function may be useful if the product $b' = Q^T b$ has already been computed using gsl_linalg_QR_QTvec().

int **gsl_linalg_QR_unpack**(const gsl_matrix *QR, const gsl_vector *tau, gsl_matrix *Q, gsl_matrix *R)

This function unpacks the encoded QR decomposition (QR, tau) into the matrices Q and R, where Q is M-by-M and R is M-by-N.

int **gsl_linalg_QR_QRsolve**(gsl_matrix *Q, gsl_matrix *R, const gsl_vector *b, gsl_vector *x)

This function solves the system $Rx = Q^T b$ for x. It can be used when the QR decomposition of a matrix is available in unpacked form as (Q, R).

int **gsl_linalg_QR_update**(gsl_matrix *Q, gsl_matrix *R, gsl_vector *w, const gsl_vector *v)

This function performs a rank-1 update wv^T of the QR decomposition (Q, R). The update is given by $Q'R' = Q(R + wv^T)$ where the output matrices Q and R are also orthogonal and right triangular. Note that w is destroyed by the update.

int **gsl_linalg_R_solve**(const gsl_matrix *R, const gsl_vector *b, gsl_vector *x)

This function solves the triangular system $Rx = b$ for the N-by-N matrix R.

int **gsl_linalg_R_svx**(const gsl_matrix *R, gsl_vector *x)

This function solves the triangular system $Rx = b$ in-place. On input x should contain the right-hand side b, which is replaced by the solution on output.

14.2.2 Triangle on Top of Rectangle

This section provides routines for computing the QR decomposition of the specialized matrix

$$\begin{pmatrix} U \\ A \end{pmatrix} = QR$$

where U is an N-by-N upper triangular matrix, and A is an M-by-N dense matrix. This type of matrix arises, for example, in the sequential TSQR algorithm. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. Due to the upper triangular factor, the Q matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and Y is dense and of the same dimensions as A.

int **gsl_linalg_QR_UR_decomp**(gsl_matrix *U, gsl_matrix *A, gsl_matrix *T)

This function computes the QR decomposition of the matrix $(U; A)$, where U is N-by-N upper triangular and A is M-by-N dense. On output, U is replaced by the R factor, and A is replaced by Y. The N-by-N upper triangular block reflector is stored in T on output.

14.2.3 Triangle on Top of Triangle

This section provides routines for computing the QR decomposition of the specialized matrix

$$\begin{pmatrix} U_1 \\ U_2 \end{pmatrix} = QR$$

where U_1, U_2 are N-by-N upper triangular matrices. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. The Q matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and Y is N-by-N upper triangular.

int **gsl_linalg_QR_UU_decomp**(gsl_matrix *U1, gsl_matrix *U2, gsl_matrix *T)

This function computes the QR decomposition of the matrix $(U_1; U_2)$, where U_1, U_2 are N-by-N upper triangular. On output, `U1` is replaced by the R factor, and `U2` is replaced by Y. The N-by-N upper triangular block reflector is stored in `T` on output.

int **gsl_linalg_QR_UU_lssolve**(const gsl_matrix *R, const gsl_matrix *Y, const gsl_matrix *T, const gsl_vector *b, gsl_vector *x, gsl_vector *work)

This function find the least squares solution to the overdetermined system,

$$\min_x \left|\left| b - \begin{pmatrix} U_1 \\ U_2 \end{pmatrix} x \right|\right|^2$$

where U_1, U_2 are N-by-N upper triangular matrices. The routine requires as input the QR decomposition of $(U_1; U_2)$ into (`R`, `Y`, `T`) given by `gsl_linalg_QR_UU_decomp()`. The parameter `x` is of length $2N$. The solution x is returned in the first N rows of `x`, i.e. $x =$ `x[0], x[1], ..., x[N-1]`. The last N rows of `x` contain a vector whose norm is equal to the residual norm $||b - (U_1; U_2)x||$. This similar to the behavior of LAPACK DGELS. Additional workspace of length N is required in `work`.

int **gsl_linalg_QR_UU_QTec**(const gsl_matrix *Y, const gsl_matrix *T, gsl_vector *b, gsl_vector *work)

This function computes $Q^T b$ using the decomposition (`Y`, `T`) previously computed by `gsl_linalg_QR_UU_decomp()`. On input, `b` contains the vector b, and on output it will contain $Q^T b$. Additional workspace of length N is required in `work`.

14.2.4 Triangle on Top of Trapezoidal

This section provides routines for computing the QR decomposition of the specialized matrix

$$\begin{pmatrix} U \\ A \end{pmatrix} = QR$$

where U is an N-by-N upper triangular matrix, and A is an M-by-N upper trapezoidal matrix with $M \geq N$. A has the structure,

$$A = \begin{pmatrix} A_d \\ A_u \end{pmatrix}$$

where A_d is $(M - N)$-by-N dense, and A_u is N-by-N upper triangular. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. The Q matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and Y is upper trapezoidal and of the same dimensions as A.

int **gsl_linalg_QR_UZ_decomp**(gsl_matrix *U, gsl_matrix *A, gsl_matrix *T)

This function computes the QR decomposition of the matrix $(U; A)$, where U is N-by-N upper triangular and A is M-by-N upper trapezoidal. On output, U is replaced by the R factor, and A is replaced by Y. The N-by-N upper triangular block reflector is stored in T on output.

14.2.5 Triangle on Top of Diagonal

This section provides routines for computing the QR decomposition of the specialized matrix

$$\begin{pmatrix} U \\ D \end{pmatrix} = QR$$

where U is an N-by-N upper triangular matrix and D is an N-by-N diagonal matrix. This type of matrix arises in regularized least squares problems. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. The Q matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and Y is N-by-N upper triangular.

int **gsl_linalg_QR_UD_decomp**(gsl_matrix *U, const gsl_vector *D, gsl_matrix *Y, gsl_matrix *T)

This function computes the QR decomposition of the matrix $(U; D)$, where U is N-by-N upper triangular and D is N-by-N diagonal. On output, `U` is replaced by the R factor and Y is stored in `Y`. The N-by-N upper triangular block reflector is stored in `T` on output.

int **gsl_linalg_QR_UD_lssolve**(const gsl_matrix *R, const gsl_matrix *Y, const gsl_matrix *T, const gsl_vector *b, gsl_vector *x, gsl_vector *work)

This function find the least squares solution to the overdetermined system,

$$\min_x \left\| b - \begin{pmatrix} U \\ D \end{pmatrix} x \right\|^2$$

where U is N-by-N upper triangular and D is N-by-N diagonal. The routine requires as input the QR decomposition of $(U; D)$ into (`R`, `Y`, `T`) given by `gsl_linalg_QR_UD_decomp()`. The parameter `x` is of length $2N$. The solution x is returned in the first N rows of `x`, i.e. $x =$ `x[0], x[1], ..., x[N-1]`. The last N rows of `x` contain a vector whose norm is equal to the residual norm $||b - (U; D)x||$. This similar to the behavior of LAPACK DGELS. Additional workspace of length N is required in `work`.

14.3 QR Decomposition with Column Pivoting

The QR decomposition of an M-by-N matrix A can be extended to the rank deficient case by introducing a column permutation P,

$$AP = QR$$

The first r columns of Q form an orthonormal basis for the range of A for a matrix with column rank r. This decomposition can also be used to convert the square linear system $Ax = b$ into the triangular system $Ry = Q^T b, x = Py$, which can be solved by back-substitution and permutation. We denote the QR decomposition with column pivoting by QRP^T since $A = QRP^T$. When A is rank deficient with $r = \text{rank}(A)$, the matrix R can be partitioned as

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} \approx \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

where R_{11} is r-by-r and nonsingular. In this case, a basic least squares solution for the overdetermined system $Ax = b$ can be obtained as

$$x = P \begin{pmatrix} R_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

where c_1 consists of the first r elements of $Q^T b$. This basic solution is not guaranteed to be the minimum norm solution unless $R_{12} = 0$ (see Complete Orthogonal Decomposition).

int **gsl_linalg_QRPT_decomp**(gsl_matrix *A, gsl_vector *tau, gsl_permutation *p, int *signum, gsl_vector *norm)

This function factorizes the M-by-N matrix `A` into the QRP^T decomposition $A = QRP^T$. On output the diagonal and upper triangular part of the input matrix contain the matrix R. The permutation matrix P is stored in the permutation `p`. The sign of the permutation is given by `signum`. It has the value $(-1)^n$, where n is the number of interchanges in the permutation. The vector `tau` and the columns of the lower triangular part of the matrix `A` contain the Householder coefficients and vectors which encode the orthogonal matrix `Q`. The vector `tau` must be of length $k = \min(M, N)$. The matrix Q is related to these components by, $Q = Q_k ... Q_2 Q_1$ where $Q_i = I - \tau_i v_i v_i^T$ and v_i is the Householder vector

$$v_i = (0, ..., 1, A(i+1, i), A(i+2, i), ..., A(m, i))$$

This is the same storage scheme as used by LAPACK. The vector `norm` is a workspace of length `N` used for column pivoting.

The algorithm used to perform the decomposition is Householder QR with column pivoting (Golub & Van Loan, "Matrix Computations", Algorithm 5.4.1).

int **gsl_linalg_QRPT_decomp2**(const gsl_matrix *A, gsl_matrix *q, gsl_matrix *r, gsl_vector *tau, gsl_permutation *p, int *signum, gsl_vector *norm)

This function factorizes the matrix `A` into the decomposition $A = QRP^T$ without modifying `A` itself and storing the output in the separate matrices `q` and `r`.

int **gsl_linalg_QRPT_solve**(const gsl_matrix *QR, const gsl_vector *tau, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x)

This function solves the square system $Ax = b$ using the QRP^T decomposition of A held in (`QR`, `tau`, `p`) which must have been computed previously by `gsl_linalg_QRPT_decomp()`.

int **gsl_linalg_QRPT_svx**(const gsl_matrix *QR, const gsl_vector *tau, const gsl_permutation *p, gsl_vector *x)

This function solves the square system $Ax = b$ in-place using the QRP^T decomposition of A held in (`QR`, `tau`, `p`). On input `x` should contain the right-hand side b, which is replaced

by the solution on output.

int **gsl_linalg_QRPT_lssolve**(const gsl_matrix *QR, const gsl_vector *tau, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x, gsl_vector *residual)

This function finds the least squares solution to the overdetermined system $Ax = b$ where the matrix `A` has more rows than columns and is assumed to have full rank. The least squares solution minimizes the Euclidean norm of the residual, $||b - Ax||$. The routine requires as input the QR decomposition of A into (`QR`, `tau`, `p`) given by `gsl_linalg_QRPT_decomp()`. The solution is returned in `x`. The residual is computed as a by-product and stored in `residual`. For rank deficient matrices, `gsl_linalg_QRPT_lssolve2()` should be used instead.

int **gsl_linalg_QRPT_lssolve2**(const gsl_matrix *QR, const gsl_vector *tau, const gsl_permutation *p, const gsl_vector *b, const size_t rank, gsl_vector *x, gsl_vector *residual)

This function finds the least squares solution to the overdetermined system $Ax = b$ where the matrix `A` has more rows than columns and has rank given by the input `rank`. If the user does not know the rank of A, the routine `gsl_linalg_QRPT_rank()` can be called to estimate it. The least squares solution is the so-called "basic" solution discussed above and may not be the minimum norm solution. The routine requires as input the QR decomposition of A into (`QR`, `tau`, `p`) given by `gsl_linalg_QRPT_decomp()`. The solution is returned in `x`. The residual is computed as a by-product and stored in `residual`.

int **gsl_linalg_QRPT_QRsolve**(const gsl_matrix *Q, const gsl_matrix *R, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x)

This function solves the square system $RP^T x = Q^T b$ for `x`. It can be used when the QR decomposition of a matrix is available in unpacked form as (`Q`, `R`).

int **gsl_linalg_QRPT_update**(gsl_matrix *Q, gsl_matrix *R, const gsl_permutation *p, gsl_vector *w, const gsl_vector *v)

This function performs a rank-1 update wv^T of the QRP^T decomposition (`Q`, `R`, `p`). The update is given by $Q'R' = Q(R + wv^T P)$ where the output matrices Q' and R' are also orthogonal and right triangular. Note that `w` is destroyed by the update. The permutation `p` is not changed.

int **gsl_linalg_QRPT_Rsolve**(const gsl_matrix *QR, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x)

This function solves the triangular system $RP^T x = b$ for the N-by-N matrix R contained in `QR`.

int **gsl_linalg_QRPT_Rsvx**(const gsl_matrix *QR, const gsl_permutation *p, gsl_vector *x)

This function solves the triangular system $RP^T x = b$ in-place for the N-by-N matrix R contained in `QR`. On input `x` should contain the right-hand side b, which is replaced by the solution on output.

size_t **gsl_linalg_QRPT_rank**(const gsl_matrix *QR, const double tol)

This function estimates the rank of the triangular matrix R contained in `QR`. The algorithm simply counts the number of diagonal elements of R whose absolute value is greater than the specified tolerance `tol`. If the input `tol` is negative, a default value of $20(M + N)eps(max(|diag(R)|))$ is used.

int **gsl_linalg_QRPT_rcond**(const gsl_matrix *QR, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the R factor, stored in the upper triangle of `QR`. The reciprocal condition number estimate, defined as $1/(||R||_1 \cdot ||R^{-1}||_1)$, is stored in `rcond`. Additional workspace of size $3N$ is required in `work`.

14.4 LQ Decomposition

A general rectangular M-by-N matrix A has a LQ decomposition into the product of a lower trapezoidal M-by-N matrix L and an orthogonal N-by-N square matrix Q:

$$A = LQ$$

If $M \leq N$, then L can be written as $L = (L_1 \quad 0)$ where L_1 is M-by-M lower triangular, and

$$A = \begin{pmatrix} L_1 & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = L_1 Q_1$$

where Q_1 consists of the first M rows of Q, and Q_2 contains the remaining $N - M$ rows. The LQ factorization of A is essentially the same as the QR factorization of A^T.

The LQ factorization may be used to find the minimum norm solution of an underdetermined system of equations $Ax = b$, where A is M-by-N and $M \leq N$. The solution is

$$x = Q^T \begin{pmatrix} L_1^{-1} b \\ 0 \end{pmatrix}$$

int **gsl_linalg_LQ_decomp**(gsl_matrix *A, gsl_vector *tau)

This function factorizes the M-by-N matrix `A` into the LQ decomposition $A = LQ$. On

output the diagonal and lower trapezoidal part of the input matrix contain the matrix L. The vector `tau` and the elements above the diagonal of the matrix `A` contain the Householder coefficients and Householder vectors which encode the orthogonal matrix `Q`. The vector `tau` must be of length $k = \min(M, N)$. The matrix Q is related to these components by, $Q = Q_k ... Q_2 Q_1$ where $Q_i = I - \tau_i v_i v_i^T$ and v_i is the Householder vector $v_i = (0, ..., 1, A(i, i+1), A(i, i+2), ..., A(i, N))$. This is the same storage scheme as used by LAPACK.

int **gsl_linalg_LQ_lssolve**(const gsl_matrix *LQ, const gsl_vector *tau, const gsl_vector *b, gsl_vector *x, gsl_vector *residual)

This function finds the minimum norm least squares solution to the underdetermined system $Ax = b$, where the M-by-N matrix `A` has $M \leq N$. The routine requires as input the LQ decomposition of A into (`LQ`, `tau`) given by `gsl_linalg_LQ_decomp()`. The solution is returned in `x`. The residual, $b - Ax$, is computed as a by-product and stored in `residual`.

int **gsl_linalg_LQ_unpack**(const gsl_matrix *LQ, const gsl_vector *tau, gsl_matrix *Q, gsl_matrix *L)

This function unpacks the encoded LQ decomposition (`LQ`, `tau`) into the matrices `Q` and `L`, where `Q` is N-by-N and `L` is M-by-N.

int **gsl_linalg_LQ_QTvec**(const gsl_matrix *LQ, const gsl_vector *tau, gsl_vector *v)

This function applies Q^T to the vector `v`, storing the result $Q^T v$ in `v` on output.

14.5 QL Decomposition

A general rectangular M-by-N matrix A has a QL decomposition into the product of an orthogonal M-by-M square matrix Q (where $Q^T Q = I$) and an M-by-N left-triangular matrix L.

When $M \geq N$, the decomposition is given by

$$A = Q \begin{pmatrix} 0 \\ L_1 \end{pmatrix}$$

where L_1 is N-by-N lower triangular. When $M \leq N$, the decomposition is given by

$$A = Q \begin{pmatrix} L_1 & L_2 \end{pmatrix}$$

where L_1 is a dense M-by-$N - M$ matrix and L_2 is a lower triangular M-by-M matrix.

int **gsl_linalg_QL_decomp**(gsl_matrix *A, gsl_vector *tau)

This function factorizes the M-by-N matrix `A` into the QL decomposition $A = QL$. The

vector `tau` must be of length N and contains the Householder coefficients on output. The matrix Q is stored in packed form in `A` on output, using the same storage scheme as LAPACK.

int **gsl_linalg_QL_unpack**(const gsl_matrix *QL, const gsl_vector *tau, gsl_matrix *Q, gsl_matrix *L)

This function unpacks the encoded QL decomposition (`QL`, `tau`) into the matrices `Q` and `L`, where `Q` is M-by-M and `L` is M-by-N.

14.6 Complete Orthogonal Decomposition

The complete orthogonal decomposition of a M-by-N matrix A is a generalization of the QR decomposition with column pivoting, given by

$$AP = Q \begin{pmatrix} R_{11} & 0 \\ 0 & 0 \end{pmatrix} Z^T$$

where P is a N-by-N permutation matrix, Q is M-by-M orthogonal, R_{11} is r-by-r upper triangular, with $r = \text{rank}(A)$, and Z is N-by-N orthogonal. If A has full rank, then $R_{11} = R$, $Z = I$ and this reduces to the QR decomposition with column pivoting.

For a rank deficient least squares problem, $\min_x ||b - Ax||^2$, the solution vector x is not unique. However if we further require that $||x||^2$ is minimized, then the complete orthogonal decomposition gives us the ability to compute the unique minimum norm solution, which is given by

$$x = PZ \begin{pmatrix} R_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

and the vector c_1 is the first r elements of $Q^T b$.

The COD also enables a straightforward solution of regularized least squares problems in Tikhonov standard form, written as

$$\min_x ||b - Ax||^2 + \lambda^2 ||x||^2$$

where $\lambda > 0$ is a regularization parameter which represents a tradeoff between minimizing the residual norm $||b - Ax||$ and the solution norm $||x||$. For this system, the solution is given by

$$x = PZ \begin{pmatrix} y_1 \\ 0 \end{pmatrix}$$

where y_1 is a vector of length r which is found by solving

$$\begin{pmatrix} R_{11} \\ \lambda I_r \end{pmatrix} y_1 = \begin{pmatrix} c_1 \\ 0 \end{pmatrix}$$

and c_1 is defined above. The equation above can be solved efficiently for different values of λ using QR factorizations of the left hand side matrix.

int **gsl_linalg_COD_decomp**(gsl_matrix *A, gsl_vector *tau_Q, gsl_vector *tau_Z, gsl_permutation *p, size_t *rank, gsl_vector *work)

int **gsl_linalg_COD_decomp_e**(gsl_matrix *A, gsl_vector *tau_Q, gsl_vector *tau_Z, gsl_permutation *p, double tol, size_t *rank, gsl_vector *work)

These functions factor the M-by-N matrix `A` into the decomposition $A = QRZP^T$. The rank of `A` is computed as the number of diagonal elements of R greater than the tolerance `tol` and output in `rank`. If `tol` is not specified, a default value is used (see `gsl_linalg_QRPT_rank()`). On output, the permutation matrix P is stored in `p`. The matrix R_{11} is stored in the upper `rank`-by-`rank` block of `A`. The matrices Q and Z are encoded in packed storage in `A` on output. The vectors `tau_Q` and `tau_Z` contain the Householder scalars corresponding to the matrices Q and Z respectively and must be of length $k = \min(M, N)$. The vector `work` is additional workspace of length N.

int **gsl_linalg_COD_lssolve**(const gsl_matrix *QRZT, const gsl_vector *tau_Q, const gsl_vector *tau_Z, const gsl_permutation *p, const size_t rank, const gsl_vector *b, gsl_vector *x, gsl_vector *residual)

This function finds the unique minimum norm least squares solution to the overdetermined system $Ax = b$ where the matrix `A` has more rows than columns. The least squares solution minimizes the Euclidean norm of the residual, $||b - Ax||$ as well as the norm of the solution $||x||$. The routine requires as input the $QRZT$ decomposition of A into (`QRZT`, `tau_Q`, `tau_Z`, `p`, `rank`) given by `gsl_linalg_COD_decomp()`. The solution is returned in `x`. The residual, $b - Ax$, is computed as a by-product and stored in `residual`.

int **gsl_linalg_COD_lssolve2**(const double lambda, const gsl_matrix *QRZT, const gsl_vector *tau_Q, const gsl_vector *tau_Z, const gsl_permutation *p, const size_t rank, const gsl_vector *b, gsl_vector *x, gsl_vector *residual, gsl_matrix *S, gsl_vector *work)

This function finds the solution to the regularized least squares problem in Tikhonov standard form, $\min_x ||b - Ax||^2 + \lambda^2 ||x||^2$. The routine requires as input the $QRZT$ decomposition of A into (`QRZT`, `tau_Q`, `tau_Z`, `p`, `rank`) given by `gsl_linalg_COD_decomp()`. The parameter λ is supplied in `lambda`. The solution is returned in `x`. The residual, $b - Ax$, is stored in `residual` on output. `S` is additional workspace of size `rank`-by-`rank`. `work` is additional workspace of length `rank`.

int **gsl_linalg_COD_unpack**(const gsl_matrix *QRZT, const gsl_vector *tau_Q, const gsl_vector *tau_Z, const size_t rank, gsl_matrix *Q, gsl_matrix *R, gsl_matrix *Z)

This function unpacks the encoded $QRZT$ decomposition (`QRZT`, `tau_Q`, `tau_Z`, `rank`) into the matrices `Q`, `R`, and `Z`, where `Q` is M-by-M, `R` is M-by-N, and `Z` is N-by-N.

int **gsl_linalg_COD_matZ**(const gsl_matrix *QRZT, const gsl_vector *tau_Z, const size_t rank, gsl_matrix *A, gsl_vector *work)

This function multiplies the input matrix `A` on the right by `Z`, $A' = AZ$ using the encoded $QRZT$ decomposition (`QRZT`, `tau_Z`, `rank`). `A` must have N columns but may have any number of rows. Additional workspace of length M is provided in `work`.

14.7 Singular Value Decomposition

A general rectangular M-by-N matrix A has a singular value decomposition (SVD) into the product of an M-by-N orthogonal matrix U, an N-by-N diagonal matrix of singular values S and the transpose of an N-by-N orthogonal square matrix V,

$$A = USV^T$$

The singular values $\sigma_i = S_{ii}$ are all non-negative and are generally chosen to form a non-increasing sequence

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_N \geq 0$$

The singular value decomposition of a matrix has many practical uses. The condition number of the matrix is given by the ratio of the largest singular value to the smallest singular value. The presence of a zero singular value indicates that the matrix is singular. The number of non-zero singular values indicates the rank of the matrix. In practice singular value decomposition of a rank-deficient matrix will not produce exact zeroes for singular values, due to finite numerical precision. Small singular values should be edited by choosing a suitable tolerance.

For a rank-deficient matrix, the null space of A is given by the columns of V corresponding to the zero singular values. Similarly, the range of A is given by columns of U corresponding to the non-zero singular values.

Note that the routines here compute the "thin" version of the SVD with U as M-by-N orthogonal matrix. This allows in-place computation and is the most commonly-used form in practice. Mathematically, the "full" SVD is defined with U as an M-by-M orthogonal matrix and S as an M-by-N diagonal matrix (with additional rows of zeros).

int **gsl_linalg_SV_decomp**(gsl_matrix *A, gsl_matrix *V, gsl_vector *S, gsl_vector *work)

This function factorizes the M-by-N matrix `A` into the singular value decomposition $A = USV^T$ for $M \geq N$. On output the matrix `A` is replaced by U. The diagonal elements of the singular value matrix S are stored in the vector `S`. The singular values are non-negative and form a non-increasing sequence from S_1 to S_N. The matrix `V` contains the elements of V in untransposed form. To form the product USV^T it is necessary to take the transpose of `V`. A workspace of length `N` is required in `work`.

This routine uses the Golub-Reinsch SVD algorithm.

int **gsl_linalg_SV_decomp_mod**(gsl_matrix *A, gsl_matrix *X, gsl_matrix *V, gsl_vector *S, gsl_vector *work)

This function computes the SVD using the modified Golub-Reinsch algorithm, which is faster for $M \gg N$. It requires the vector `work` of length `N` and the N-by-N matrix `X` as additional working space.

int **gsl_linalg_SV_decomp_jacobi**(gsl_matrix *A, gsl_matrix *V, gsl_vector *S)

This function computes the SVD of the M-by-N matrix `A` using one-sided Jacobi orthogonalization for $M \geq N$. The Jacobi method can compute singular values to higher relative accuracy than Golub-Reinsch algorithms (see references for details).

int **gsl_linalg_SV_solve**(const gsl_matrix *U, const gsl_matrix *V, const gsl_vector *S, const gsl_vector *b, gsl_vector *x)

This function solves the system $Ax = b$ using the singular value decomposition (`U`, `S`, `V`) of A which must have been computed previously with `gsl_linalg_SV_decomp()`.

Only non-zero singular values are used in computing the solution. The parts of the solution corresponding to singular values of zero are ignored. Other singular values can be edited out by setting them to zero before calling this function.

In the over-determined case where `A` has more rows than columns the system is solved in the least squares sense, returning the solution `x` which minimizes $||Ax - b||_2$.

int **gsl_linalg_SV_leverage**(const gsl_matrix *U, gsl_vector *h)

This function computes the statistical leverage values h_i of a matrix A using its singular value decomposition (`U`, `S`, `V`) previously computed with `gsl_linalg_SV_decomp()`. h_i are the diagonal values of the matrix $A(A^TA)^{-1}A^T$ and depend only on the matrix `U` which is the input to this function.

14.8 Cholesky Decomposition

A symmetric, positive definite square matrix A has a Cholesky decomposition into a product of a lower triangular matrix L and its transpose L^T,

$$A = LL^T$$

This is sometimes referred to as taking the square-root of a matrix. The Cholesky decomposition can only be carried out when all the eigenvalues of the matrix are positive. This decomposition can be used to convert the linear system $Ax = b$ into a pair of triangular systems ($Ly = b$, $L^T x = y$), which can be solved by forward and back-substitution.

If the matrix A is near singular, it is sometimes possible to reduce the condition number and recover a more accurate solution vector x by scaling as

$$(SAS)\left(S^{-1}x\right) = Sb$$

where S is a diagonal matrix whose elements are given by $S_{ii} = 1/\sqrt{A_{ii}}$. This scaling is also known as Jacobi preconditioning. There are routines below to solve both the scaled and unscaled systems.

int **gsl_linalg_cholesky_decomp1**(gsl_matrix *A)

int **gsl_linalg_complex_cholesky_decomp**(gsl_matrix_complex *A)

These functions factorize the symmetric, positive-definite square matrix `A` into the Cholesky decomposition $A = LL^T$ (or $A = LL^\dagger$ for the complex case). On input, the values from the diagonal and lower-triangular part of the matrix `A` are used (the upper triangular part is ignored). On output the diagonal and lower triangular part of the input matrix `A` contain the matrix L, while the upper triangular part contains the original matrix. If the matrix is not positive-definite then the decomposition will fail, returning the error code `GSL_EDOM`.

When testing whether a matrix is positive-definite, disable the error handler first to avoid triggering an error. These functions use Level 3 BLAS to compute the Cholesky factorization (Peise and Bientinesi, 2016).

int **gsl_linalg_cholesky_decomp**(gsl_matrix *A)

This function is now deprecated and is provided only for backward compatibility.

int **gsl_linalg_cholesky_solve**(const gsl_matrix *cholesky, const gsl_vector *b, gsl_vector *x)

int **gsl_linalg_complex_cholesky_solve**(const gsl_matrix_complex *cholesky, const gsl_vector_complex *b, gsl_vector_complex *x)

These functions solve the system $Ax = b$ using the Cholesky decomposition of A held in the matrix `cholesky` which must have been previously computed by `gsl_linalg_cholesky_decomp()` or `gsl_linalg_complex_cholesky_decomp()`.

int **gsl_linalg_cholesky_svx**(const gsl_matrix *cholesky, gsl_vector *x)

int **gsl_linalg_complex_cholesky_svx**(const gsl_matrix_complex *cholesky, gsl_vector_complex *x)

These functions solve the system $Ax = b$ in-place using the Cholesky decomposition of A held in the matrix `cholesky` which must have been previously computed by `gsl_linalg_cholesky_decomp()` or `gsl_linalg_complex_cholesky_decomp()`. On input `x` should contain the right-hand side b, which is replaced by the solution on output.

int **gsl_linalg_cholesky_invert**(gsl_matrix *cholesky)

int **gsl_linalg_complex_cholesky_invert**(gsl_matrix_complex *cholesky)

These functions compute the inverse of a matrix from its Cholesky decomposition `cholesky`, which must have been previously computed by `gsl_linalg_cholesky_decomp()` or `gsl_linalg_complex_cholesky_decomp()`. On output, the inverse is stored in-place in `cholesky`.

int **gsl_linalg_cholesky_decomp2**(gsl_matrix *A, gsl_vector *S)

This function calculates a diagonal scaling transformation S for the symmetric, positive-definite square matrix `A`, and then computes the Cholesky decomposition $SAS = LL^T$. On input, the values from the diagonal and lower-triangular part of the matrix `A` are used (the upper triangular part is ignored). On output the diagonal and lower triangular part of the input matrix `A` contain the matrix L, while the upper triangular part of the input matrix is overwritten with L^T (the diagonal terms being identical for both L and L^T). If the matrix is not positive-definite then the decomposition will fail, returning the error code `GSL_EDOM`. The diagonal scale factors are stored in `S` on output.

When testing whether a matrix is positive-definite, disable the error handler first to avoid triggering an error.

int **gsl_linalg_cholesky_solve2**(const gsl_matrix *cholesky, const gsl_vector *S, const gsl_vector *b, gsl_vector *x)

This function solves the system $(SAS)(S^{-1}x) = Sb$ using the Cholesky decomposition of SAS held in the matrix `cholesky` which must have been previously computed by `gsl_linalg_cholesky_decomp2()`.

int **gsl_linalg_cholesky_svx2**(const gsl_matrix *cholesky, const gsl_vector *S, gsl_vector *x)

This function solves the system $(SAS)(S^{-1}x) = Sb$ in-place using the Cholesky decomposition of SAS held in the matrix `cholesky` which must have been previously

computed by `gsl_linalg_cholesky_decomp2()`. On input `x` should contain the right-hand side b, which is replaced by the solution on output.

int **gsl_linalg_cholesky_scale**(const gsl_matrix *A, gsl_vector *S)

This function calculates a diagonal scaling transformation of the symmetric, positive definite matrix `A`, such that SAS has a condition number within a factor of N of the matrix of smallest possible condition number over all possible diagonal scalings. On output, `S` contains the scale factors, given by $S_i = 1/\sqrt{A_{ii}}$. For any $A_{ii} \leq 0$, the corresponding scale factor S_i is set to 1.

int **gsl_linalg_cholesky_scale_apply**(gsl_matrix *A, const gsl_vector *S)

This function applies the scaling transformation `S` to the matrix `A`. On output, `A` is replaced by SAS.

int **gsl_linalg_cholesky_rcond**(const gsl_matrix *cholesky, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric positive definite matrix A, using its Cholesky decomposition provided in `cholesky`. The reciprocal condition number estimate, defined as $1/(||A||_1 \cdot ||A^{-1}||_1)$, is stored in `rcond`. Additional workspace of size $3N$ is required in `work`.

14.9 Pivoted Cholesky Decomposition

A symmetric positive semi-definite square matrix A has an alternate Cholesky decomposition into a product of a lower unit triangular matrix L, a diagonal matrix D and L^T, given by LDL^T. For postive definite matrices, this is equivalent to the Cholesky formulation discussed above, with the standard Cholesky lower triangular factor given by $LD^{\frac{1}{2}}$. For ill-conditioned matrices, it can help to use a pivoting strategy to prevent the entries of D and L from growing too large, and also ensure $D_1 \geq D_2 \geq \cdots \geq D_n > 0$, where D_i are the diagonal entries of D. The final decomposition is given by

$$PAP^T = LDL^T$$

where P is a permutation matrix.

int **gsl_linalg_pcholesky_decomp**(gsl_matrix *A, gsl_permutation *p)

This function factors the symmetric, positive-definite square matrix `A` into the Pivoted Cholesky decomposition $PAP^T = LDL^T$. On input, the values from the diagonal and lower-triangular part of the matrix `A` are used to construct the factorization. On output the diagonal of the input matrix `A` stores the diagonal elements of D, and the lower triangular portion of `A` contains the matrix L. Since L has ones on its diagonal these do

not need to be explicitely stored. The upper triangular portion of A is unmodified. The permutation matrix P is stored in p on output.

int **gsl_linalg_pcholesky_solve**(const gsl_matrix *LDLT, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x)

This function solves the system $Ax = b$ using the Pivoted Cholesky decomposition of A held in the matrix LDLT and permutation p which must have been previously computed by gsl_linalg_pcholesky_decomp().

int **gsl_linalg_pcholesky_svx**(const gsl_matrix *LDLT, const gsl_permutation *p, gsl_vector *x)

This function solves the system $Ax = b$ in-place using the Pivoted Cholesky decomposition of A held in the matrix LDLT and permutation p which must have been previously computed by gsl_linalg_pcholesky_decomp(). On input, x contains the right hand side vector b which is replaced by the solution vector on output.

int **gsl_linalg_pcholesky_decomp2**(gsl_matrix *A, gsl_permutation *p, gsl_vector *S)

This function computes the pivoted Cholesky factorization of the matrix SAS, where the input matrix A is symmetric and positive definite, and the diagonal scaling matrix S is computed to reduce the condition number of A as much as possible. See Cholesky Decomposition for more information on the matrix S. The Pivoted Cholesky decomposition satisfies $PSASP^T = LDL^T$. On input, the values from the diagonal and lower-triangular part of the matrix A are used to construct the factorization. On output the diagonal of the input matrix A stores the diagonal elements of D, and the lower triangular portion of A contains the matrix L. Since L has ones on its diagonal these do not need to be explicitely stored. The upper triangular portion of A is unmodified. The permutation matrix P is stored in p on output. The diagonal scaling transformation is stored in S on output.

int **gsl_linalg_pcholesky_solve2**(const gsl_matrix *LDLT, const gsl_permutation *p, const gsl_vector *S, const gsl_vector *b, gsl_vector *x)

This function solves the system $(SAS)(S^{-1}x) = Sb$ using the Pivoted Cholesky decomposition of SAS held in the matrix LDLT, permutation p, and vector S, which must have been previously computed by gsl_linalg_pcholesky_decomp2().

int **gsl_linalg_pcholesky_svx2**(const gsl_matrix *LDLT, const gsl_permutation *p, const gsl_vector *S, gsl_vector *x)

This function solves the system $(SAS)(S^{-1}x) = Sb$ in-place using the Pivoted Cholesky decomposition of SAS held in the matrix LDLT, permutation p and vector S, which must have been previously computed by gsl_linalg_pcholesky_decomp2(). On input, x contains the right hand side vector b which is replaced by the solution vector on output.

int **gsl_linalg_pcholesky_invert**(const gsl_matrix *LDLT, const gsl_permutation *p, gsl_matrix *Ainv)

This function computes the inverse of the matrix A, using the Pivoted Cholesky decomposition stored in `LDLT` and `p`. On output, the matrix `Ainv` contains A^{-1}.

int **gsl_linalg_pcholesky_rcond**(const gsl_matrix *LDLT, const gsl_permutation *p, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric positive definite matrix A, using its pivoted Cholesky decomposition provided in `LDLT`. The reciprocal condition number estimate, defined as $1/(||A||_1 \cdot ||A^{-1}||_1)$, is stored in `rcond`. Additional workspace of size $3N$ is required in `work`.

14.10 Modified Cholesky Decomposition

The modified Cholesky decomposition is suitable for solving systems $Ax = b$ where A is a symmetric indefinite matrix. Such matrices arise in nonlinear optimization algorithms. The standard Cholesky decomposition requires a positive definite matrix and would fail in this case. Instead of resorting to a method like QR or SVD, which do not take into account the symmetry of the matrix, we can instead introduce a small perturbation to the matrix A to make it positive definite, and then use a Cholesky decomposition on the perturbed matrix. The resulting decomposition satisfies

$$P(A + E)P^T = LDL^T$$

where P is a permutation matrix, E is a diagonal perturbation matrix, L is unit lower triangular, and D is diagonal. If A is sufficiently positive definite, then the perturbation matrix E will be zero and this method is equivalent to the pivoted Cholesky algorithm. For indefinite matrices, the perturbation matrix E is computed to ensure that $A + E$ is positive definite and well conditioned.

int **gsl_linalg_mcholesky_decomp**(gsl_matrix *A, gsl_permutation *p, gsl_vector *E)

This function factors the symmetric, indefinite square matrix `A` into the Modified Cholesky decomposition $P(A+E)P^T = LDL^T$. On input, the values from the diagonal and lower-triangular part of the matrix `A` are used to construct the factorization. On output the diagonal of the input matrix `A` stores the diagonal elements of D, and the lower triangular portion of `A` contains the matrix L. Since L has ones on its diagonal these do not need to be explicitely stored. The upper triangular portion of `A` is unmodified. The permutation matrix P is stored in `p` on output. The diagonal perturbation matrix is stored in `E` on output. The parameter `E` may be set to NULL if it is not required.

int **gsl_linalg_mcholesky_solve**(const gsl_matrix *LDLT, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x)

This function solves the perturbed system $(A+E)x = b$ using the Cholesky decomposition of $A + E$ held in the matrix `LDLT` and permutation `p` which must have been previously computed by `gsl_linalg_mcholesky_decomp()`.

int **gsl_linalg_mcholesky_svx**(const gsl_matrix *LDLT, const gsl_permutation *p, gsl_vector *x)

This function solves the perturbed system $(A + E)x = b$ in-place using the Cholesky decomposition of $A + E$ held in the matrix `LDLT` and permutation `p` which must have been previously computed by `gsl_linalg_mcholesky_decomp()`. On input, `x` contains the right hand side vector b which is replaced by the solution vector on output.

int **gsl_linalg_mcholesky_rcond**(const gsl_matrix *LDLT, const gsl_permutation *p, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the perturbed matrix $A + E$, using its pivoted Cholesky decomposition provided in `LDLT`. The reciprocal condition number estimate, defined as $1/(||A+E||_1 \cdot ||(A+E)^{-1}||_1)$, is stored in `rcond`. Additional workspace of size $3N$ is required in `work`.

14.11 LDLT Decomposition

If A is a symmetric, nonsingular square matrix, then it has a unique factorization of the form

$$A = LDL^T$$

where L is a unit lower triangular matrix and D is diagonal. If A is positive definite, then this factorization is equivalent to the Cholesky factorization, where the lower triangular Cholesky factor is $LD^{\frac{1}{2}}$. Some indefinite matrices for which no Cholesky decomposition exists have an LDL^T decomposition with negative entries in D. The LDL^T algorithm is sometimes referred to as the square root free Cholesky decomposition, as the algorithm does not require the computation of square roots. The algorithm is stable for positive definite matrices, but is not guaranteed to be stable for indefinite matrices.

int **gsl_linalg_ldlt_decomp**(gsl_matrix *A)

This function factorizes the symmetric, non-singular square matrix `A` into the decomposition $A = LDL^T$. On input, the values from the diagonal and lower-triangular part of the matrix `A` are used. The upper triangle of `A` is used as temporary workspace. On output the diagonal of `A` contains the matrix D and the lower triangle of `A` contains

the unit lower triangular matrix L. The matrix 1-norm, $||A||_1$ is stored in the upper right corner on output, for later use by `gsl_linalg_ldlt_rcond()`.

If the matrix is detected to be singular, the function returns the error code `GSL_EDOM`.

int **gsl_linalg_ldlt_solve**(const gsl_matrix *LDLT, const gsl_vector *b, gsl_vector *x)

This function solves the system $Ax = b$ using the LDL^T decomposition of A held in the matrix `LDLT` which must have been previously computed by `gsl_linalg_ldlt_decomp()`.

int **gsl_linalg_ldlt_svx**(const gsl_matrix *LDLT, gsl_vector *x)

This function solves the system $Ax = b$ in-place using the LDL^T decomposition of A held in the matrix `LDLT` which must have been previously computed by `gsl_linalg_ldlt_decomp()`. On input `x` should contain the right-hand side b, which is replaced by the solution on output.

int **gsl_linalg_ldlt_rcond**(const gsl_matrix *LDLT, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric nonsingular matrix A, using its LDL^T decomposition provided in `LDLT`. The reciprocal condition number estimate, defined as $1/(||A||_1 \cdot ||A^{-1}||_1)$, is stored in `rcond`. Additional workspace of size $3N$ is required in `work`.

14.12 Tridiagonal Decomposition of Real Symmetric Matrices

A symmetric matrix A can be factorized by similarity transformations into the form,

$$A = QTQ^T$$

where Q is an orthogonal matrix and T is a symmetric tridiagonal matrix.

int **gsl_linalg_symmtd_decomp**(gsl_matrix *A, gsl_vector *tau)

This function factorizes the symmetric square matrix `A` into the symmetric tridiagonal decomposition QTQ^T. On output the diagonal and subdiagonal part of the input matrix `A` contain the tridiagonal matrix T. The remaining lower triangular part of the input matrix contains the Householder vectors which, together with the Householder coefficients `tau`, encode the orthogonal matrix Q. This storage scheme is the same as used by LAPACK. The upper triangular part of `A` is not referenced.

int **gsl_linalg_symmtd_unpack**(const gsl_matrix *A, const gsl_vector *tau, gsl_matrix *Q, gsl_vector *diag, gsl_vector *subdiag)

This function unpacks the encoded symmetric tridiagonal decomposition (`A`, `tau`) obtained from `gsl_linalg_symmtd_decomp()` into the orthogonal matrix `Q`, the vector of diagonal elements `diag` and the vector of subdiagonal elements `subdiag`.

int **gsl_linalg_symmtd_unpack_T**(const gsl_matrix *A, gsl_vector *diag, gsl_vector *subdiag)

This function unpacks the diagonal and subdiagonal of the encoded symmetric tridiagonal decomposition (`A`, `tau`) obtained from `gsl_linalg_symmtd_decomp()` into the vectors `diag` and `subdiag`.

14.13 Tridiagonal Decomposition of Hermitian Matrices

A hermitian matrix A can be factorized by similarity transformations into the form,

$$A = UTU^T$$

where U is a unitary matrix and T is a real symmetric tridiagonal matrix.

int **gsl_linalg_hermtd_decomp**(gsl_matrix_complex *A, gsl_vector_complex *tau)

This function factorizes the hermitian matrix `A` into the symmetric tridiagonal decomposition UTU^T. On output the real parts of the diagonal and subdiagonal part of the input matrix `A` contain the tridiagonal matrix T. The remaining lower triangular part of the input matrix contains the Householder vectors which, together with the Householder coefficients `tau`, encode the unitary matrix U. This storage scheme is the same as used by LAPACK. The upper triangular part of `A` and imaginary parts of the diagonal are not referenced.

int **gsl_linalg_hermtd_unpack**(const gsl_matrix_complex *A, const gsl_vector_complex *tau, gsl_matrix_complex *U, gsl_vector *diag, gsl_vector *subdiag)

This function unpacks the encoded tridiagonal decomposition (`A`, `tau`) obtained from `gsl_linalg_hermtd_decomp()` into the unitary matrix `U`, the real vector of diagonal elements `diag` and the real vector of subdiagonal elements `subdiag`.

int **gsl_linalg_hermtd_unpack_T**(const gsl_matrix_complex *A, gsl_vector *diag, gsl_vector *subdiag)

This function unpacks the diagonal and subdiagonal of the encoded tridiagonal decomposition (`A`, `tau`) obtained from the `gsl_linalg_hermtd_decomp()` into the real vectors `diag` and `subdiag`.

14.14 Hessenberg Decomposition of Real Matrices

A general real matrix A can be decomposed by orthogonal similarity transformations into the form

$$A = UHU^T$$

where U is orthogonal and H is an upper Hessenberg matrix, meaning that it has zeros below the first subdiagonal. The Hessenberg reduction is the first step in the Schur decomposition for the nonsymmetric eigenvalue problem, but has applications in other areas as well.

int **gsl_linalg_hessenberg_decomp**(gsl_matrix *A, gsl_vector *tau)

This function computes the Hessenberg decomposition of the matrix `A` by applying the similarity transformation $H = U^T AU$. On output, H is stored in the upper portion of `A`. The information required to construct the matrix U is stored in the lower triangular portion of `A`. U is a product of $N-2$ Householder matrices. The Householder vectors are stored in the lower portion of `A` (below the subdiagonal) and the Householder coefficients are stored in the vector `tau`. `tau` must be of length `N`.

int **gsl_linalg_hessenberg_unpack**(gsl_matrix *H, gsl_vector *tau, gsl_matrix *U)

This function constructs the orthogonal matrix U from the information stored in the Hessenberg matrix `H` along with the vector `tau`. `H` and `tau` are outputs from `gsl_linalg_hessenberg_decomp()`.

int **gsl_linalg_hessenberg_unpack_accum**(gsl_matrix *H, gsl_vector *tau, gsl_matrix *V)

This function is similar to `gsl_linalg_hessenberg_unpack()`, except it accumulates the matrix `U` into `V`, so that $V' = VU$. The matrix `V` must be initialized prior to calling this function. Setting `V` to the identity matrix provides the same result as `gsl_linalg_hessenberg_unpack()`. If `H` is order `N`, then `V` must have `N` columns but may have any number of rows.

int **gsl_linalg_hessenberg_set_zero**(gsl_matrix *H)

This function sets the lower triangular portion of `H`, below the subdiagonal, to zero. It is useful for clearing out the Householder vectors after calling `gsl_linalg_hessenberg_decomp()`.

14.15 Hessenberg-Triangular Decomposition of Real Matrices

A general real matrix pair (A, B) can be decomposed by orthogonal similarity transformations into the form

$$A = UHV^T$$
$$B = URV^T$$

where U and V are orthogonal, H is an upper Hessenberg matrix, and R is upper triangular. The Hessenberg-Triangular reduction is the first step in the generalized Schur decomposition for the generalized eigenvalue problem.

int **gsl_linalg_hesstri_decomp**(gsl_matrix *A, gsl_matrix *B, gsl_matrix *U, gsl_matrix *V, gsl_vector *work)

This function computes the Hessenberg-Triangular decomposition of the matrix pair (`A`, `B`). On output, H is stored in `A`, and R is stored in `B`. If `U` and `V` are provided (they may be null), the similarity transformations are stored in them. Additional workspace of length N is needed in `work`.

14.16 Bidiagonalization

A general matrix A can be factorized by similarity transformations into the form,

$$A = UBV^T$$

where U and V are orthogonal matrices and B is a N-by-N bidiagonal matrix with non-zero entries only on the diagonal and superdiagonal. The size of `U` is M-by-N and the size of `V` is N-by-N.

int **gsl_linalg_bidiag_decomp**(gsl_matrix *A, gsl_vector *tau_U, gsl_vector *tau_V)

This function factorizes the M-by-N matrix `A` into bidiagonal form UBV^T. The diagonal and superdiagonal of the matrix B are stored in the diagonal and superdiagonal of `A`. The orthogonal matrices U and `V` are stored as compressed Householder vectors in the remaining elements of `A`. The Householder coefficients are stored in the vectors `tau_U` and `tau_V`. The length of `tau_U` must equal the number of elements in the diagonal of `A` and the length of `tau_V` should be one element shorter.

int **gsl_linalg_bidiag_unpack**(const gsl_matrix *A, const gsl_vector *tau_U, gsl_matrix *U, const gsl_vector *tau_V, gsl_matrix *V, gsl_vector *diag, gsl_vector *superdiag)

This function unpacks the bidiagonal decomposition of `A` produced by `gsl_linalg_bidiag_decomp()`, (`A`, `tau_U`, `tau_V`) into the separate orthogonal matrices `U`, `V` and the diagonal vector `diag` and superdiagonal `superdiag`. Note that `U` is stored as a compact M-by-N orthogonal matrix satisfying $U^T U = I$ for efficiency.

int **gsl_linalg_bidiag_unpack2**(gsl_matrix *A, gsl_vector *tau_U, gsl_vector *tau_V, gsl_matrix *V)

This function unpacks the bidiagonal decomposition of `A` produced by `gsl_linalg_bidiag_decomp()`, (`A`, `tau_U`, `tau_V`) into the separate orthogonal matrices `U`, `V` and the diagonal vector `diag` and superdiagonal `superdiag`. The matrix `U` is stored in-place in `A`.

int **gsl_linalg_bidiag_unpack_B**(const gsl_matrix *A, gsl_vector *diag, gsl_vector *superdiag)

This function unpacks the diagonal and superdiagonal of the bidiagonal decomposition of `A` from `gsl_linalg_bidiag_decomp()`, into the diagonal vector `diag` and superdiagonal vector `superdiag`.

14.17 Givens Rotations

A Givens rotation is a rotation in the plane acting on two elements of a given vector. It can be represented in matrix form as

$$G(i,j,\theta) = \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & \cos\theta & \dots & -\sin\theta & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & \sin\theta & \dots & \cos\theta & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix}$$

where the $\cos\theta$ and $\sin\theta$ appear at the intersection of the i-th and j-th rows and columns. When acting on a vector x, $G(i,j,\theta)x$ performs a rotation of the (i,j) elements of x. Givens rotations are typically used to introduce zeros in vectors, such as during the QR decomposition of a matrix. In this case, it is typically desired to find c and s such that

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

with $r = \sqrt{a^2 + b^2}$.

void **gsl_linalg_givens**(const double a, const double b, double *c, double *s)

This function computes $c = \cos\theta$ and $s = \sin\theta$ so that the Givens matrix $G(\theta)$ acting on the vector (a, b) produces $(r, 0)$, with $r = \sqrt{a^2 + b^2}$.

void **gsl_linalg_givens_gv**(gsl_vector *v, const size_t i, const size_t j, const double c, const double s)

This function applies the Givens rotation defined by $c = \cos\theta$ and $s = \sin\theta$ to the `i` and `j` elements of `v`. On output, $(v(i), v(j)) \leftarrow G(\theta)(v(i), v(j))$.

14.18 Householder Transformations

A Householder transformation is a rank-1 modification of the identity matrix which can be used to zero out selected elements of a vector. A Householder matrix H takes the form,

$$H = I - \tau v v^T$$

where v is a vector (called the Householder vector) and $\tau = 2/(v^T v)$. The functions described in this section use the rank-1 structure of the Householder matrix to create and apply Householder transformations efficiently.

double **gsl_linalg_householder_transform**(gsl_vector *w)

gsl_complex **gsl_linalg_complex_householder_transform**(gsl_vector_complex *w)

This function prepares a Householder transformation $H = I - \tau v v^T$ which can be used to zero all the elements of the input vector `w` except the first. On output the Householder vector `v` is stored in `w` and the scalar τ is returned. The householder vector `v` is normalized so that `v[0] = 1`, however this 1 is not stored in the output vector. Instead, `w[0]` is set to the first element of the transformed vector, so that if $u = Hw$, `w[0] = u[0]` on output and the remainder of u is zero.

int **gsl_linalg_householder_hm**(double tau, const gsl_vector *v, gsl_matrix *A)

int **gsl_linalg_complex_householder_hm**(gsl_complex tau, const gsl_vector_complex *v, gsl_matrix_complex *A)

This function applies the Householder matrix H defined by the scalar `tau` and the vector `v` to the left-hand side of the matrix `A`. On output the result HA is stored in `A`.

int **gsl_linalg_householder_mh**(double tau, const gsl_vector *v, gsl_matrix *A)

int **gsl_linalg_complex_householder_mh**(gsl_complex tau, const gsl_vector_complex *v, gsl_matrix_complex *A)

This function applies the Householder matrix H defined by the scalar `tau` and the vector `v` to the right-hand side of the matrix `A`. On output the result AH is stored in `A`.

int **gsl_linalg_householder_hv**(double tau, const gsl_vector *v, gsl_vector *w)

int **gsl_linalg_complex_householder_hv**(gsl_complex tau, const gsl_vector_complex *v, gsl_vector_complex *w)

This function applies the Householder transformation H defined by the scalar `tau` and the vector `v` to the vector `w`. On output the result Hw is stored in `w`.

14.19 Householder solver for linear systems

int **gsl_linalg_HH_solve**(gsl_matrix *A, const gsl_vector *b, gsl_vector *x)

This function solves the system $Ax = b$ directly using Householder transformations. On output the solution is stored in `x` and `b` is not modified. The matrix `A` is destroyed by the Householder transformations.

int **gsl_linalg_HH_svx**(gsl_matrix *A, gsl_vector *x)

This function solves the system $Ax = b$ in-place using Householder transformations. On input `x` should contain the right-hand side b, which is replaced by the solution on output. The matrix `A` is destroyed by the Householder transformations.

14.20 Tridiagonal Systems

The functions described in this section efficiently solve symmetric, non-symmetric and cyclic tridiagonal systems with minimal storage. Note that the current implementations of these functions use a variant of Cholesky decomposition, so the tridiagonal matrix must be positive definite. For non-positive definite matrices, the functions return the error code `GSL_ESING`.

int **gsl_linalg_solve_tridiag**(const gsl_vector *diag, const gsl_vector *e, const gsl_vector *f, const gsl_vector *b, gsl_vector *x)

This function solves the general N-by-N system $Ax = b$ where `A` is tridiagonal ($N \geq 2$). The super-diagonal and sub-diagonal vectors `e` and `f` must be one element shorter than the diagonal vector `diag`. The form of `A` for the 4-by-4 case is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ 0 & 0 & f_2 & d_3 \end{pmatrix}$$

int **gsl_linalg_solve_symm_tridiag**(const gsl_vector *diag, const gsl_vector *e, const gsl_vector *b, gsl_vector *x)

This function solves the general N-by-N system $Ax = b$ where `A` is symmetric tridiagonal ($N \geq 2$). The off-diagonal vector `e` must be one element shorter than the diagonal vector `diag`. The form of `A` for the 4-by-4 case is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ 0 & 0 & e_2 & d_3 \end{pmatrix}$$

int **gsl_linalg_solve_cyc_tridiag**(const gsl_vector *diag, const gsl_vector *e, const gsl_vector *f, const gsl_vector *b, gsl_vector *x)

This function solves the general N-by-N system $Ax = b$ where `A` is cyclic tridiagonal ($N \geq 3$). The cyclic super-diagonal and sub-diagonal vectors `e` and `f` must have the same number of elements as the diagonal vector `diag`. The form of `A` for the 4-by-4 case is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & f_3 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ e_3 & 0 & f_2 & d_3 \end{pmatrix}$$

int **gsl_linalg_solve_symm_cyc_tridiag**(const gsl_vector *diag, const gsl_vector *e, const gsl_vector *b, gsl_vector *x)

This function solves the general N-by-N system $Ax = b$ where `A` is symmetric cyclic tridiagonal ($N \geq 3$). The cyclic off-diagonal vector `e` must have the same number of elements as the diagonal vector `diag`. The form of `A` for the 4-by-4 case is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & e_3 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ e_3 & 0 & e_2 & d_3 \end{pmatrix}$$

14.21 Triangular Systems

int **gsl_linalg_tri_invert**(CBLAS_UPLO_t Uplo, CBLAS_DIAG_t Diag, gsl_matrix *T)

int **gsl_linalg_complex_tri_invert**(CBLAS_UPLO_t Uplo, CBLAS_DIAG_t Diag, gsl_matrix_complex *T)

These functions compute the in-place inverse of the triangular matrix `T`, stored in the lower triangle when `Uplo` = `CblasLower` and upper triangle when `Uplo` = `CblasUpper`. The parameter `Diag` = `CblasUnit`, `CblasNonUnit` specifies whether the matrix is unit triangular.

int **gsl_linalg_tri_LTL**(gsl_matrix *L)

int **gsl_linalg_complex_tri_LHL**(gsl_matrix_complex *L)

These functions compute the product $L^T L$ (or $L^\dagger L$) in-place and stores it in the lower triangle of `L` on output.

int **gsl_linalg_tri_UL**(gsl_matrix *LU)

int **gsl_linalg_complex_tri_UL**(gsl_matrix_complex *LU)

These functions compute the product UL where U is upper triangular and L is unit lower triangular, stored in `LU`, as computed by `gsl_linalg_LU_decomp()` or `gsl_linalg_complex_LU_decomp()`. The product is computed in-place using Level 3 BLAS.

int **gsl_linalg_tri_rcond**(CBLAS_UPLO_t Uplo, const gsl_matrix *A, double *rcond, gsl_vector *work)

This function estimates the 1-norm reciprocal condition number of the triangular matrix `A`, using the lower triangle when `Uplo` is `CblasLower` and upper triangle when `Uplo` is `CblasUpper`. The reciprocal condition number $1/\left(||A||_1 \left|\left|A^{-1}\right|\right|_1\right)$ is stored in `rcond` on output. Additional workspace of size $3N$ is required in `work`.

14.22 Banded Systems

Band matrices are sparse matrices whose non-zero entries are confined to a diagonal band. From a storage point of view, significant savings can be achieved by storing only the non-zero diagonals of a banded matrix. Algorithms such as LU and Cholesky factorizations preserve the band structure of these matrices. Computationally, working with compact banded matrices is preferable to working on the full dense matrix with many zero entries.

14.22.1 General Banded Format

An example of a general banded matrix is given below.

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & 0 \\ \delta_1 & \alpha_2 & \beta_2 & \gamma_2 & 0 & 0 \\ 0 & \delta_2 & \alpha_3 & \beta_3 & \gamma_3 & 0 \\ 0 & 0 & \delta_3 & \alpha_4 & \beta_4 & \gamma_4 \\ 0 & 0 & 0 & \delta_4 & \alpha_5 & \beta_5 \\ 0 & 0 & 0 & 0 & \delta_5 & \alpha_6 \end{pmatrix}$$

This matrix has a lower bandwidth of 1 and an upper bandwidth of 2. The lower bandwidth is the number of non-zero subdiagonals, and the upper bandwidth is the number of non-zero superdiagonals. A (p, q) banded matrix has a lower bandwidth p and upper bandwidth q. For example, diagonal matrices are $(0, 0)$, tridiagonal matrices are $(1, 1)$, and upper triangular matrices are $(0, N - 1)$ banded matrices.

The corresponding 6-by-4 packed banded matrix looks like

$$AB = \begin{pmatrix} * & * & \alpha_1 & \delta_1 \\ * & \beta_1 & \alpha_2 & \delta_2 \\ \gamma_1 & \beta_2 & \alpha_3 & \delta_3 \\ \gamma_2 & \beta_3 & \alpha_4 & \delta_4 \\ \gamma_3 & \beta_4 & \alpha_5 & \delta_5 \\ \gamma_4 & \beta_5 & \alpha_6 & * \end{pmatrix}$$

where the superdiagonals are stored in columns, followed by the diagonal, followed by the subdiagonals. The entries marked by $*$ are not referenced by the banded routines. With this format, each row of AB corresponds to the non-zero entries of the corresponding column of A. For an N-by-N matrix A, the dimension of AB will be N-by-$(p + q + 1)$.

14.22.2 Symmetric Banded Format

Symmetric banded matrices allow for additional storage savings. As an example, consider the following 6×6 symmetric banded matrix with lower bandwidth $p = 2$:

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & 0 \\ \beta_1 & \alpha_2 & \beta_2 & \gamma_2 & 0 & 0 \\ \gamma_1 & \beta_2 & \alpha_3 & \beta_3 & \gamma_3 & 0 \\ 0 & \gamma_2 & \beta_3 & \alpha_4 & \beta_4 & \gamma_4 \\ 0 & 0 & \gamma_3 & \beta_4 & \alpha_5 & \beta_5 \\ 0 & 0 & 0 & \gamma_4 & \beta_5 & \alpha_6 \end{pmatrix}$$

The packed symmetric banded 6×3 matrix will look like:

$$AB = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 \\ \alpha_2 & \beta_2 & \gamma_2 \\ \alpha_3 & \beta_3 & \gamma_3 \\ \alpha_4 & \beta_4 & \gamma_4 \\ \alpha_5 & \beta_5 & * \\ \alpha_6 & * & * \end{pmatrix}$$

The entries marked by $*$ are not referenced by the symmetric banded routines. The relationship between the packed format and original matrix is,

$$AB(i,j) = A(i, i+j) = A(i+j, i)$$

for $i = 0, \ldots, N-1, j = 0, \ldots, p$. Conversely,

$$A(i,j) = AB(j, i-j)$$

for $i = 0, \ldots, N-1, j = \max(0, i-p), \ldots, i$.

경고: Note that this format is the transpose of the symmetric banded format used by LAPACK. In order to develop efficient routines for symmetric banded matrices, it helps to have the nonzero elements in each column in contiguous memory locations. Since C uses row-major order, GSL stores the columns in the rows of the packed banded format, while LAPACK, written in Fortran, uses the transposed format.

14.22.3 Banded LU Decomposition

The routines in this section are designed to factor banded M-by-N matrices with an LU factorization, $PA = LU$. The matrix A is banded of type (p, q), i.e. a lower bandwidth of p and an upper bandwidth of q. See LU Decomposition for more information on the factorization. For banded (p, q) matrices, the U factor will have an upper bandwidth of $p + q$, while the L factor will have a lower bandwidth of at most p. Therefore, additional storage is needed to store the p additional bands of U.

As an example, consider the $M = N = 7$ matrix with lower bandwidth $p = 3$ and upper bandwidth $q = 2$,

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & 0 & 0 \\ \delta_1 & \alpha_2 & \beta_2 & \gamma_2 & 0 & 0 & 0 \\ \epsilon_1 & \delta_2 & \alpha_3 & \beta_3 & \gamma_3 & 0 & 0 \\ \zeta_1 & \epsilon_2 & \delta_3 & \alpha_4 & \beta_4 & \gamma_4 & 0 \\ 0 & \zeta_2 & \epsilon_3 & \delta_4 & \alpha_5 & \beta_5 & \gamma_5 \\ 0 & 0 & \zeta_3 & \epsilon_4 & \delta_5 & \alpha_6 & \beta_6 \\ 0 & 0 & 0 & \zeta_4 & \epsilon_5 & \delta_6 & \alpha_7 \end{pmatrix}$$

The corresponding N-by-$2p + q + 1$ packed banded matrix looks like

$$AB = \begin{pmatrix} * & * & * & * & * & \alpha_1 & \delta_1 & \epsilon_1 & \zeta_1 \\ * & * & * & * & \beta_1 & \alpha_2 & \delta_2 & \epsilon_2 & \zeta_2 \\ * & * & * & \gamma_1 & \beta_2 & \alpha_3 & \delta_3 & \epsilon_3 & \zeta_3 \\ * & * & - & \gamma_2 & \beta_3 & \alpha_4 & \delta_4 & \epsilon_4 & \zeta_4 \\ * & - & - & \gamma_3 & \beta_4 & \alpha_5 & \delta_5 & \epsilon_5 & * \\ - & - & - & \gamma_4 & \beta_5 & \alpha_6 & \delta_6 & * & * \\ p- & - & - & q\gamma_5 & \beta_6 & \alpha_7 & p* & * & * \end{pmatrix}$$

Entries marked with $-$ are used to store the additional p diagonals of the U factor. Entries marked with $*$ are not referenced by the banded routines.

int **gsl_linalg_LU_band_decomp**(const size_t M, const size_t lb, const size_t ub, gsl_matrix *AB, gsl_vector_uint *piv)

This function computes the LU factorization of the banded matrix `AB` which is stored in packed band format (see above) and has dimension N-by-$2p + q + 1$. The number of rows M of the original matrix is provided in `M`. The lower bandwidth p is provided in `lb` and the upper bandwidth q is provided in `ub`. The vector `piv` has length $\min(M, N)$ and stores the pivot indices on output (for $0 \leq i < \min(M, N)$, row i of the matrix was interchanged with row `piv[i]`). On output, `AB` contains both the L and U factors in packed format.

int **gsl_linalg_LU_band_solve**(const size_t lb, const size_t ub, const gsl_matrix *LUB, const gsl_vector_uint *piv, const gsl_vector *b, gsl_vector *x)

This function solves the square system $Ax = b$ using the banded LU factorization (`LUB`, `piv`) computed by `gsl_linalg_LU_band_decomp()`. The lower and upper bandwidths are provided in `lb` and `ub` respectively. The right hand side vector is provided in `b`. The solution vector is stored in `x` on output.

int **gsl_linalg_LU_band_svx**(const size_t lb, const size_t ub, const gsl_matrix *LUB, const gsl_vector_uint *piv, gsl_vector *x)

This function solves the square system $Ax = b$ in-place, using the banded LU factorization (`LUB`, `piv`) computed by `gsl_linalg_LU_band_decomp()`. The lower and upper bandwidths are provided in `lb` and `ub` respectively. On input, the right hand side vector b is provided in `x`, which is replaced by the solution vector x on output.

int **gsl_linalg_LU_band_unpack**(const size_t M, const size_t lb, const size_t ub, const gsl_matrix *LUB, const gsl_vector_uint *piv, gsl_matrix *L, gsl_matrix *U)

This function unpacks the banded LU factorization (`LUB`, `piv`) previously computed by `gsl_linalg_LU_band_decomp()` into the matrices `L` and `U`. The matrix `U` has dimension $\min(M,N)$-by-N and stores the upper triangular factor on output. The matrix `L` has dimension M-by-$\min(M,N)$ and stores the matrix $P^T L$ on output.

14.22.4 Banded Cholesky Decomposition

The routines in this section are designed to factor and solve N-by-N linear systems of the form $Ax = b$ where A is a banded, symmetric, and positive definite matrix with lower bandwidth p. See Cholesky Decomposition for more information on the factorization. The lower triangular factor of the Cholesky decomposition preserves the same banded structure as the matrix A, enabling an efficient algorithm which overwrites the original matrix with the L factor.

int **gsl_linalg_cholesky_band_decomp**(gsl_matrix *A)

This function factorizes the symmetric, positive-definite square matrix `A` into the Cholesky decomposition $A = LL^T$. The input matrix `A` is given in symmetric banded format, and has dimensions N-by-$(p+1)$, where p is the lower bandwidth of the matrix. On output, the entries of `A` are replaced by the entries of the matrix L in the same format. In addition, the lower right element of `A` is used to store the matrix 1-norm, used later by `gsl_linalg_cholesky_band_rcond()` to calculate the reciprocal condition number.

If the matrix is not positive-definite then the decomposition will fail, returning the error code `GSL_EDOM`. When testing whether a matrix is positive-definite, disable the error handler first to avoid triggering an error.

int **gsl_linalg_cholesky_band_solve**(const gsl_matrix *LLT, const gsl_vector *b, gsl_vector *x)

int **gsl_linalg_cholesky_band_solvem**(const gsl_matrix *LLT, const gsl_matrix *B, gsl_matrix *X)

This function solves the symmetric banded system $Ax = b$ (or $AX = B$) using the Cholesky decomposition of A held in the matrix `LLT` which must have been previously computed by `gsl_linalg_cholesky_band_decomp()`.

int **gsl_linalg_cholesky_band_svx**(const gsl_matrix *LLT, gsl_vector *x)

int **gsl_linalg_cholesky_band_svxm**(const gsl_matrix *LLT, gsl_matrix *X)

This function solves the symmetric banded system $Ax = b$ (or $AX = B$) in-place using the Cholesky decomposition of A held in the matrix `LLT` which must have been previously computed by `gsl_linalg_cholesky_band_decomp()`. On input `x` (or `X`) should contain the right-hand side b (or B), which is replaced by the solution on output.

int **gsl_linalg_cholesky_band_invert**(const gsl_matrix *LLT, gsl_matrix *Ainv)

This function computes the inverse of a symmetric banded matrix from its Cholesky decomposition `LLT`, which must have been previously computed by `gsl_linalg_cholesky_band_decomp()`. On output, the inverse is stored in `Ainv`, using both the lower and upper portions.

int **gsl_linalg_cholesky_band_unpack**(const gsl_matrix *LLT, gsl_matrix *L)

This function unpacks the lower triangular Cholesky factor from `LLT` and stores it in the lower triangular portion of the N-by-N matrix `L`. The upper triangular portion of `L` is not referenced.

int **gsl_linalg_cholesky_band_scale**(const gsl_matrix *A, gsl_vector *S)

This function calculates a diagonal scaling transformation of the symmetric, positive definite banded matrix `A`, such that SAS has a condition number within a factor of N of the matrix of smallest possible condition number over all possible diagonal scalings. On output, `S` contains the scale factors, given by $S_i = 1/\sqrt{A_{ii}}$. For any $A_{ii} \leq 0$, the corresponding scale factor S_i is set to 1.

int **gsl_linalg_cholesky_band_scale_apply**(gsl_matrix *A, const gsl_vector *S)

This function applies the scaling transformation `S` to the banded symmetric positive definite matrix `A`. On output, `A` is replaced by SAS.

int **gsl_linalg_cholesky_band_rcond**(const gsl_matrix *LLT, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric banded positive definite matrix A, using its Cholesky decomposition provided in `LLT`. The reciprocal condition number estimate, defined as $1/(||A||_1 \cdot ||A^{-1}||_1)$, is stored in `rcond`. Additional workspace of size $3N$ is required in `work`.

14.22.5 Banded LDLT Decomposition

The routines in this section are designed to factor and solve N-by-N linear systems of the form $Ax = b$ where A is a banded, symmetric, and non-singular matrix with lower bandwidth p. See LDLT Decomposition for more information on the factorization. The lower triangular factor of the LDL^T decomposition preserves the same banded structure as the matrix A, enabling an efficient algorithm which overwrites the original matrix with the L and D factors.

int **gsl_linalg_ldlt_band_decomp**(gsl_matrix *A)

This function factorizes the symmetric, non-singular square matrix `A` into the decomposition $A = LDL^T$. The input matrix `A` is given in symmetric banded format, and has dimensions N-by-$(p + 1)$, where p is the lower bandwidth of the matrix. On output, the entries of `A` are replaced by the entries of the matrices D and L in the same format.

If the matrix is singular then the decomposition will fail, returning the error code `GSL_EDOM`.

int **gsl_linalg_ldlt_band_solve**(const gsl_matrix *LDLT, const gsl_vector *b, gsl_vector *x)

This function solves the symmetric banded system $Ax = b$ using the LDL^T decomposition of A held in the matrix `LDLT` which must have been previously computed by `gsl_linalg_ldlt_band_decomp()`.

int **gsl_linalg_ldlt_band_svx**(const gsl_matrix *LDLT, gsl_vector *x)

This function solves the symmetric banded system $Ax = b$ in-place using the LDL^T decomposition of A held in the matrix `LDLT` which must have been previously computed by `gsl_linalg_ldlt_band_decomp()`. On input `x` should contain the right-hand side b, which is replaced by the solution on output.

int **gsl_linalg_ldlt_band_unpack**(const gsl_matrix *LDLT, gsl_matrix *L, gsl_vector *D)

This function unpacks the unit lower triangular factor L from `LDLT` and stores it in the lower triangular portion of the N-by-N matrix `L`. The upper triangular portion of `L` is not referenced. The diagonal matrix D is stored in the vector `D`.

int **gsl_linalg_ldlt_band_rcond**(const gsl_matrix *LDLT, double *rcond, gsl_vector *work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric banded nonsingular matrix A, using its LDL^T decomposition provided in `LDLT`. The reciprocal condition number estimate, defined as $1/(||A||_1 \cdot ||A^{-1}||_1)$, is stored in `rcond`. Additional workspace of size $3N$ is required in `work`.

14.23 Balancing

The process of balancing a matrix applies similarity transformations to make the rows and columns have comparable norms. This is useful, for example, to reduce roundoff errors in the solution of eigenvalue problems. Balancing a matrix A consists of replacing A with a similar matrix

$$A' = D^{-1} A D$$

where D is a diagonal matrix whose entries are powers of the floating point radix.

int **gsl_linalg_balance_matrix**(gsl_matrix *A, gsl_vector *D)

This function replaces the matrix `A` with its balanced counterpart and stores the diagonal elements of the similarity transformation into the vector `D`.

14.24 Examples

The following program solves the linear system $Ax = b$. The system to be solved is,

$$\begin{pmatrix} 0.18 & 0.60 & 0.57 & 0.96 \\ 0.41 & 0.24 & 0.99 & 0.58 \\ 0.14 & 0.30 & 0.97 & 0.66 \\ 0.51 & 0.13 & 0.19 & 0.85 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}$$

and the solution is found using LU decomposition of the matrix A.

```
#include <stdio.h>
#include <gsl/gsl_linalg.h>

int
main (void)
{
  double a_data[] = { 0.18, 0.60, 0.57, 0.96,
                      0.41, 0.24, 0.99, 0.58,
                      0.14, 0.30, 0.97, 0.66,
                      0.51, 0.13, 0.19, 0.85 };

  double b_data[] = { 1.0, 2.0, 3.0, 4.0 };

  gsl_matrix_view m
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    = gsl_matrix_view_array (a_data, 4, 4);

  gsl_vector_view b
    = gsl_vector_view_array (b_data, 4);

  gsl_vector *x = gsl_vector_alloc (4);

  int s;

  gsl_permutation * p = gsl_permutation_alloc (4);

  gsl_linalg_LU_decomp (&m.matrix, p, &s);

  gsl_linalg_LU_solve (&m.matrix, p, &b.vector, x);

  printf ("x = \n");
  gsl_vector_fprintf (stdout, x, "%g");

  gsl_permutation_free (p);
  gsl_vector_free (x);
  return 0;
}
```

Here is the output from the program,

```
x =
-4.05205
-12.6056
1.66091
8.69377
```

This can be verified by multiplying the solution x by the original matrix A using GNU octave,

```
octave> A = [ 0.18, 0.60, 0.57, 0.96;
              0.41, 0.24, 0.99, 0.58;
              0.14, 0.30, 0.97, 0.66;
              0.51, 0.13, 0.19, 0.85 ];

octave> x = [ -4.05205; -12.6056; 1.66091; 8.69377];

octave> A * x
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
ans =
  1.0000
  2.0000
  3.0000
  4.0000
```

This reproduces the original right-hand side vector, b, in accordance with the equation $Ax = b$.

14.25 References and Further Reading

Further information on the algorithms described in this section can be found in the following book,

- G. H. Golub, C. F. Van Loan, "Matrix Computations" (3rd Ed, 1996), Johns Hopkins University Press, ISBN 0-8018-5414-8.

The LAPACK library is described in the following manual,

- LAPACK Users' Guide (Third Edition, 1999), Published by SIAM, ISBN 0-89871-447-8

The LAPACK source code can be found at http://www.netlib.org/lapack, along with an online copy of the users guide.

Further information on recursive Level 3 BLAS algorithms may be found in the following paper,

- E. Peise and P. Bientinesi, "Recursive algorithms for dense linear algebra: the ReLAPACK collection", http://arxiv.org/abs/1602.06763, 2016.

The recursive Level 3 BLAS QR decomposition is described in the following paper,

- E. Elmroth and F. G. Gustavson, 2000. Applying recursion to serial and parallel QR factorization leads to better performance. IBM Journal of Research and Development, 44(4), pp.605-624.

The Modified Golub-Reinsch algorithm is described in the following paper,

- T.F. Chan, "An Improved Algorithm for Computing the Singular Value Decomposition", ACM Transactions on Mathematical Software, 8 (1982), pp 72–83.

The Jacobi algorithm for singular value decomposition is described in the following papers,

- J.C. Nash, "A one-sided transformation method for the singular value decomposition and algebraic eigenproblem", Computer Journal, Volume 18, Number 1 (1975), p 74–76

- J.C. Nash and S. Shlien “Simple algorithms for the partial singular value decomposition”, Computer Journal, Volume 30 (1987), p 268–275.
- J. Demmel, K. Veselic, “Jacobi’s Method is more accurate than QR”, Lapack Working Note 15 (LAWN-15), October 1989. Available from netlib, http://www.netlib.org/lapack/ in the `lawns` or `lawnspdf` directories.

The algorithm for estimating a matrix condition number is described in the following paper,

- N. J. Higham, “FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation”, ACM Trans. Math. Soft., vol. 14, no. 4, pp. 381-396, December 1988.

제 15 장

고유 공간

이 단원에서는 행렬의 고유 값과 고유 벡터를 계산하는 함수들에 대해 다룹니다. 실수 위 대칭, 비대칭, 일반화된 준대칭, 일반화된 비대칭 복소수 위 에르미트, 일반화된 준 에르미트, 그리고 행렬들의 풀이 기능들을 제공합니다. 고유 값들은 고유 벡터들과 함께 계산되거나 단독으로 계산될 수도 있습니다. 에르미트와 실수 대칭 행렬을 계산하는 알고리즘들은 대칭 bidigonalization 과 QR 감소을 이용합니다. 비대칭 알고리즘은 Francis-QR 이중 전이 방법을, 일반화된 비대칭 알고리즘은 Moler와 Strwart의 QZ 방법을 사용합니다.

15.1 실수 위 대칭 행렬

실수 위 대칭 행렬의 고유 값, 벡터 문제는 bidiagonalization과 QR-감소 방법을 사용합니다. 이 방법들은 Golub & van Loan, section 8.3 에 기술되어 있습니다. 고유 값은 $\epsilon||A||_2$ 의 절대 정확도를 가집니다. ϵ 은 계산기 정밀도입니다.

type **gsl_eigen_symm_workspace**

실수 위 대칭 행렬의 고유 값을 계산하기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_symm_workspace ***gsl_eigen_symm_alloc**(const size_t n)

n-by-n 크기의 실수 위 대칭 행렬의 고유값 문제를 풀기위한 작업 공간을 할당합니다. 작업 공간의 크기는 $O(2n)$ 입니다.

void **gsl_eigen_symm_free**(gsl_eigen_symm_workspace *w)

할당된 작업 공간 w 를 해제합니다.

int **gsl_eigen_symm**(gsl_matrix *A, gsl_vector *eval, gsl_eigen_symm_workspace *w)

대칭 행렬 A 의 고유값을 계산합니다. 함수를 호출과정에서 반드시 적절한 크기의 작업 공간을 w 에 전해주어야 합니다. 행렬 A 의 대각, 하삼각 성분들은 계산과정에서 수정됩니다. 반면, 상삼각 성분들은 참조되지 않습니다. 고유 값들은 벡터 eval 에 무작위 순서로 저장됩니다.

type **gsl_eigen_symmv_workspace**

실수 대칭 행렬의 고유 값과 고유 벡터를 계산하기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_symmv_workspace *__gsl_eigen_symmv_alloc__(const size_t n)

크기 n-by-n 의 실수 대칭 행렬의 고유 값, 벡터 문제를 풀기 위한 작업공간을 할당합니다. 공간의 크기는 $O(4n)$ 입니다.

void __gsl_eigen_symmv_free__(gsl_eigen_symmv_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

int __gsl_eigen_symmv__(gsl_matrix *A, gsl_vector *eval, gsl_matrix *evec,
gsl_eigen_symmv_workspace *w)

실수 대칭 행렬 A 의 고유값, 고유 벡터를 계산합니다. 반드시, 함수 호출과정에서 적절한 크기의 작업 공간을 w 에 제공해 주어야합니다. 행렬 A 의 대각, 하삼각 성분들은 계산과정에서 수정됩니다. 반면, 상삼각 성분들은 참조되지 않습니다. 고유값들은 벡터 eval 에 무작위 순서로 저장됩니다. 대응 되는 고유 벡터들은 행렬 evec 의 열에 고유값과 동일 순서로 저장됩니다. 예를 들어, 행렬의 첫번째 열은 첫번째 고유값에 해당하는 고유 벡터입니다. 계산된 고유 벡터들은 서로 직교하고 크기 1로 정규화 되어있습니다.

15.2 복소수 위 에르미트 행렬

에르미트 행렬의 계산은 대칭 bidiagonalization과 QR-감소 방법을 사용합니다.

type **gsl_eigen_herm_workspace**

에르미트 행렬의 고유 값을 계산하기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_herm_workspace *__gsl_eigen_herm_alloc__(const size_t n)

n × n 크기의 복소수 에르미트 행렬의 고유 값을 찾기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(3n)$ 입니다.

void __gsl_eigen_herm_free__(gsl_eigen_herm_workspace *w)

T:data:w 가 가르키는 작업 공간 영역을 해제합니다.

int __gsl_eigen_herm__(gsl_matrix_complex *A, gsl_vector *eval, gsl_eigen_herm_workspace *w)

복소수 에르미트 행렬 A 의 고유 값을 계산합니다. 반드시, 함수 호출과정에서 적절한 크기의 작업 공간을 w 에 제공해 주어야합니다. 행렬 A 의 대각 성분과 하 삼각 성분은 계산 과정에서 수정됩니다. 하지만, 상 삼각 성분은 계산 과정에서 수정되지 않습니다. 대각 성분의 허수부는 0으로 간주되고 사용되지 않습니다. 고유 값들은 벡터 eval 에 무작위로 저장됩니다.

type **gsl_eigen_hermv_workspace**

에르미트 행렬의 고유 값과 고유 벡터를 계산하기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_hermv_workspace *__gsl_eigen_hermv_alloc__(const size_t n)

n × n 크기의 복소수 에르미트 행렬의 고유 값과 고유 벡터를 찾기 위한 작업 공간을 메모리에 할당합니다. 작업 공간의 크기는 $O(5n)$ 입니다.

void __gsl_eigen_hermv_free__(gsl_eigen_hermv_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

int __gsl_eigen_hermv__(gsl_matrix_complex *A, gsl_vector *eval, gsl_matrix_complex *evec,
gsl_eigen_hermv_workspace *w)

복소수 에르미트 행렬 A 의 고유 벡터와 고유 값을 계산합니다. 반드시, 함수 호출과정에서 적절한 크기의 작업 공간을 w 에 제공해 주어야합니다. 행렬 A 의 대각 성분과 하 삼각 성분은 계산 과정에서 수정됩니다. 대각 성분의 허수부는 0으로 간주되며 계산 과정에서 사용되지 않습니다. 고유 값들은 eval 이 가르키는 벡터에 무자구이 순서로 저장됩니다. 이 고유 값들에 대응되는 고유 벡터들은 행렬 evec 에 열 벡터로 저장됩니다. 예로 행렬 evec 의 첫번째 열 벡터는 eval 의 첫번째 값에 대응되는 고유 벡터입니다. 이 고유 벡터들은 서로 직교하고 크기 1로 정규화 되어 있습니다.

15.3 실수 위 비대칭 행렬

실수 비대칭 행렬 A 의 고유 값과 고유 벡터들을 찾는 문제는 슈어 분해(Schur decomposition)를 이용해 계산할 수 있습니다.

$$A = ZTZ^T$$

Z 는 슈어 벡터들의 직교 행렬, T 은 Schur 형태로 Quasi-상 삼각 행렬입니다. Quasi-상 삼각 행렬은 블럭 상 삼각 행렬로 대각 성분이 1×1 이나 2×2 크기의 블럭 행렬인 행렬을 의미합니다. 이 때, 2×2 크기 행렬의 고유 값들은 A 행렬의 고유 값들의 켤레 복소수입니다. 이 과정에서 사용되는 알고리즘에는 이중 전이 Francis 방법(double-shiftted Francis method)이 사용됩니다.

type __gsl_eigen_nonsymm_workspace__

비대칭 행렬의 고유 값 문제를 풀기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_nonsymm_workspace *__gsl_eigen_nonsymm_alloc__(const size_t n)

n × n 크기의 실수 비대칭 행렬의 고유 값을 찾기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(2n)$ 입니다.

void __gsl_eigen_nonsymm_free__(gsl_eigen_nonsymm_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

void __gsl_eigen_nonsymm_params__(const int compute_t, const int balance,
gsl_eigen_nonsymm_workspace *w)

gsl_eigen_nonsymm() 함수를 사용하기 위한 기본 계수들을 설정합니다.

compute_t 가 1로 정해지면 슈어 형태 T 가 gsl_eigen_nonsymm() 를 이용해 계산됩니다. 0으로 정해지면 T 가 0인 상황입니다. 완전한 슈어 형태 T 의 계산은 근사적으로 1.5-2 Flops의 시간을 필요로 합니다.

balance 가 1로 정해지면 고유 값을 계산하기 전 균형 변환이 행렬에 적용됩니다. 이 변환은 행렬의 행과 열들이 comparable 노름을 가지도록 만들어 더 정확한 고유 값들을 얻을 수 있습니다. 더 자세한 내용은 Balancing 을 참고할 수 있습니다. 유의할 점은 균형 변환이 슈어 벡터들의 직교성을 보존하지 않는다는 점입니다. 따라서 gsl_eigen_nonsymm_Z() 를 이용해 슈어 벡터를 계산하고자 한다면, 원본 행렬의 슈어 벡터가 아니라 균형 변환된 행렬의 슈어 벡터를 얻게됩니다. 이 두 벡터들 사이의 관계는 다음과 같습니다.

$$T = Q^T D^{-1} A D Q$$

Q 는 행렬로 균형 변환된 행렬에 대한 슈어 벡터들을 나타냅니다. D 는 균형 변환에 대응되는 행렬입니다. gsl_eigen_nonsymm_Z() 함수는 다음을 만족하는 행렬 Z 를 계산합니다.

$$T = Z^{-1} A Z$$

$Z = DQ$ 를 의미합니다. Z 는 직교하지 않음을 유의해야 합니다. 이러한 이유로 균형 변환을 포함하면 더 정확한 고유 값들을 계산할 수 있음에도 불구하고 기본 설정에서 제거하고 선택 사항으로 제공합니다.

int **gsl_eigen_nonsymm**(gsl_matrix *A, gsl_vector_complex *eval,
gsl_eigen_nonsymm_workspace *w)

실수 위 비대칭 행렬 A 의 고유 값을 계산합니다. 계산된 고유 값들은 벡터 eval 에 저장됩니다. T 값이 필요한 경우, 해당 값들은 꼐산이 끝난 후 행렬 A 의 대각 성분을 포함한 상 삼각 부분에 저장됩니다. 계산 후 A 의 대각 성분은 1×1 크기의 고유 값과 고유 값이 A 고유 값의 켤레 복소수 값을 가지는 크기 :math:` 2 times 2` 행렬들을 가지고 있습니다. A 의 나머지 부분들의 값은 계산 과정에서 훼손됩니다. 드믄 일이지만 이 함수가 모든 고유 값을 찾지 못하는 경우가 있습니다. 이 경우 오류 값이 반환되고 수렴한 고유 값들의 숫자가 w -> n_evals 에 저장됩니다. 수렴된 고유 값들은 eval 에 앞 자리부터 채워져 저장됩니다.

int **gsl_eigen_nonsymm_Z**(gsl_matrix *A, gsl_vector_complex *eval, gsl_matrix *Z,
gsl_eigen_nonsymm_workspace *w)

gsl_eigen_nonsymm() 와 동일합니다. 다만 추가로 슈어 벡터를 계산하고 Z 에 저장한다는 차이가 있습니다.

type **gsl_eigen_nonsymmv_workspace**

비대칭 행렬의 고유 값과 고유 벡터를 계산하기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_nonsymmv_workspace ***gsl_eigen_nonsymmv_alloc**(const size_t n)

n × n 크기의 실수 비대칭 행렬의 고유 값과 고유 벡터를 계산하기 위한 작업 공간을 메모리에 할당

합니다. 공간의 크기는 $O(5n)$ 입니다.

void **gsl_eigen_nonsymmv_free**(gsl_eigen_nonsymmv_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

void **gsl_eigen_nonsymmv_params**(const int balance, gsl_eigen_nonsymm_workspace *w)

gsl_eigen_nonsymmv() 을 사용하기 위한 계수들을 설정합니다.

balance 가 1 로 정해지면 행렬에 균형 변환이 적용됩니다. 자세한 내용은 gsl_eigen_nonsymm_params() 를 참고하길 바랍니다. 균형 변환이 슈어 벡터들의 직교성을 보존하지 않기 때문에 기본적으로 비활성화 되어 있습니다.

int **gsl_eigen_nonsymmv**(gsl_matrix *A, gsl_vector_complex *eval, gsl_matrix_complex *evec, gsl_eigen_nonsymmv_workspace *w)

$n \times n$ 크기의 실수 위 비대칭 행렬 A 의 고유 값과 고유 벡터들을 계산합니다. 먼저 gsl_eigen_nonsymm() 를 호출해 고유 값, 슈어 형태 T 그리고 슈어 벡터를 계산합니다. 그 다음 T 의 고유 값을 계산하고 슈어 벡터들을 이용해 이들을 역변환합니다. 계산 과정에서 슈어 벡터들의 값은 유지되지 않습니다. 하지만, 함수 gsl_eigen_nonsymmv_Z() 를 이용해 저장할 수 있습니다. 계산이 끝난 후 A 의 상 삼각부는 슈어 형태 T 와 같습니다. gsl_eigen_nonsymm() 에서 계산이 실패하면 고유 벡터들이 계산되지 않고 오류 값이 반환됩니다.

int **gsl_eigen_nonsymmv_Z**(gsl_matrix *A, gsl_vector_complex *eval, gsl_matrix_complex *evec, gsl_matrix *Z, gsl_eigen_nonsymmv_workspace *w)

gsl_eigen_nonsymmv() 와 동일합니다. 다만 슈어 벡터를 Z 에 저장합니다.

15.4 실수 위 일반화된 대칭-정부호 고유계

실수 위에서 일반화된 대칭-정부호 행렬의 고유 문제는 다음과 같은 고유 값과 고유 벡터들을 계산하는 문제입니다.

$$Ax = \lambda Bx$$

A 와 B 는 대칭 행렬이고, B 는 양의 정부호 행렬입니다. 이 문제는 B 에 촘스키 분해를 적용해 표준적인 대칭 행렬의 고유 값 문제로 바꿀 수 있습니다.

$$\begin{aligned} Ax &= \lambda Bx \\ Ax &= \lambda LL^T x \\ \left(L^{-1}AL^{-T}\right) L^T x &= \lambda L^T x \end{aligned}$$

따라서, 원래 문제는 대칭 행렬 $C = L^{-1}AL^{-T}$ 와 $y = L^T x$ 에 대해 $Cy = \lambda y$ 형태로 표현할 수 있습니다. 이 경우 표준적인 대칭 행렬의 고유 문제 풀이 기능을 행렬 C 에 적용할 수 있습니다. 해당 계산 결과로

나온 고유 값들을 역변환해 원래 행렬의 고유 값들을 찾을 수 있습니다. 일반화된 대칭-정부호 행렬의 고유 값과 고유 벡터들은 항상 실수 있습니다.

type **gsl_eigen_gensymm_workspace**

일반화된 대칭 행렬의 고유 값 문제를 풀기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_gensymm_workspace ***gsl_eigen_gensymm_alloc**(const size_t n)

n × n 크기의 일반화된 대칭-정부호 행렬의 고유 값을 찾기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(2n)$ 입니다.

void **gsl_eigen_gensymm_free**(gsl_eigen_gensymm_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

int **gsl_eigen_gensymm**(gsl_matrix *A, gsl_matrix *B, gsl_vector *eval,
gsl_eigen_gensymm_workspace *w)

일반화된 대칭-정부호 행렬쌍 (A , B)의 고유 값을 계산해 eval 에 저장합니다. 이 함수는 위에 기술된 일반 대칭 행렬 문제로 변환하는 방법을 사용합니다. 계산후 B 에는 B 의 촘스키 분해가 저장되고 A 는 본래 값을 잃습니다.

type **gsl_eigen_gensymmv_workspace**

일반화된 대칭 행렬의 고유 값과 고유 벡터를 계산하기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_gensymmv_workspace ***gsl_eigen_gensymmv_alloc**(const size_t n)

n × n 크기의 실수 일반화된 대칭-정부호 행렬의 고유 값과 고유 벡터를 계산하기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(4n)$ 입니다.

void **gsl_eigen_gensymmv_free**(gsl_eigen_gensymmv_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

int **gsl_eigen_gensymmv**(gsl_matrix *A, gsl_matrix *B, gsl_vector *eval, gsl_matrix *evec,
gsl_eigen_gensymmv_workspace *w)

실수 위 일반화된 대칭-정부호 행렬 쌍(A, B)의 고유 값과 고유 벡터들을 계산합니다. 계산된 고유 값과 벡터들은 각각 eval 과 evec 에 저장됩니다. 계산된 고유 벡터들은 서로 직교하고 크기 1로 정규화 되어있습니다. 계산 후 B 에는 B 의 촘스키 분해가 저장되어 있고 A 는 계산 과정에서 훼손됩니다.

15.5 복소수 위 일반화된 에르미트-정부호 고유계

복소수 위 일반화딘 에르미트-정부호 행렬의 고유 값 문제는 다음을 만족하는 고유 값 λ 와 고유 벡터 x 를 찾는 문제입니다.

$$Ax = \lambda Bx$$

A 와 B 는 에르미트 행렬이고, B 는 추가로 정부호 행렬이기도 합니다. 실수 위와 비슷하게 이 문제는 에르미트 행렬 $C = L^{-1}AL^{-\dagger}$ 와 $y = L^{\dagger}x$ 로 $Cy = \lambda y$ 형태로 변환할 수 있습니다. 에르미트 고유 문제의 표준 해결 방법을 C 에 적용해 이 문제를 풀수 있고, 풀이로 나온 고유 벡터들에 역변환을 적용해 원래 문제의 해답을 얻을 수 있습니다. 일반화된 에르미트-정부호 고유 문제의 답은 언제나 실수입니다.

type **gsl_eigen_genherm_workspace**

일반화된 에르미트-정부호 행렬의 고유 값 문제를 풀기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_genherm_workspace ***gsl_eigen_genherm_alloc**(const size_t n)

복소수 위 `n` × `n` 크기의 일반화된 에르미트-정부호 행렬의 고유 값을 찾기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(3n)$ 입니다.

void **gsl_eigen_genherm_free**(gsl_eigen_genherm_workspace *w)

`w` 가 가르키는 작업 공간 영역을 해제합니다.

int **gsl_eigen_genherm**(gsl_matrix_complex *A, gsl_matrix_complex *B, gsl_vector *eval, gsl_eigen_genherm_workspace *w)

복소수 위에서 일반화된 에르미트-정부호 행렬 쌍 (`A`, `B`)의 고유 값을 계산해 `eval` 에 저장합니다. 사용하는 방법은 위에 기술되어 있습니다. 계산 후 `B` 에는 `B` 의 촘스키 분해가 저장되고 `A` 는 계산과정에서 훼손됩니다.

type **gsl_eigen_genhermv_workspace**

일반화된 에르미트-정부호 행렬의 고유 값가 고유 벡터 문제를 풀기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_genhermv_workspace ***gsl_eigen_genhermv_alloc**(const size_t n)

복소수 위 `n` × `n` 크기의 일반화된 에르미트-정부호 행렬의 고유 값과 고유 벡터를 찾기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(5n)$ 입니다.

void **gsl_eigen_genhermv_free**(gsl_eigen_genhermv_workspace *w)

`w` 가 가르키는 작업 공간 영역을 해제합니다.

int **gsl_eigen_genhermv**(gsl_matrix_complex *A, gsl_matrix_complex *B, gsl_vector *eval, gsl_matrix_complex *evec, gsl_eigen_genhermv_workspace *w)

복소수 위 일반화된 에르미트-정부호 행렬 쌍 (`A`, `B`)의 고유 값과 고유 벡터를 계산하고 이들을 각각

eval 와 evec 에 저장합니다. 계산된 고유 벡터들은 서로 직교하고 크기 1로 정규화 되어있습니다. 계산 후 B 에는 B 의 촘스키 분해가 저장되고 A 는 계산과정에서 훼손됩니다.

15.6 실수 위 일반화된 비대칭 고유계

주어진 두 정사각 행렬 (A, B) 에 대해, 일반화된 비대칭 계의 풀이는 다음을 만족하는 고유 값 λ 와 고유 벡터 x 를 찾는 문제입니다.

$$Ax = \lambda Bx$$

고유 값 mu 와 고유 벡터 y 에 대해 다음과 같은 형식도 가능합니다.

$$\mu Ay = By$$

이 두 문제는 $\lambda = 1/\mu$ 식을 이용하면 동일한 식임을 알 수 있습니다. 이때, $lambda$ 와 mu 는 모두 0이 아니여야 합니다. $lambda$ 가 0이여도 첫번째 식은 여전히 잘 정의된 고유계를 형성하지만 두번째 식은 mu 가 정의되지 않습니다. 고유 값으로 0과 무한대를 사용하기 위해 라이브러리에서 제공하는 함수들은 다음과 같은 형태의 고유 계를 풀게 됩니다.

$$\beta Ax = \alpha Bx$$

풀이 함수들은 α 와 β 두 값을 반환해 사용자가 λ 나 μ 를 계산해 사용할 수 있게 합니다. 각각은 나눗셈을 이용해서 $\lambda = \alpha/\beta$ 와 $\mu = \beta/\alpha$ 로 계산할 수 있습니다.

선형 행렬 pencil $A - \lambda B$ 의 행렬식이 모든 λ 에 대해 0이면 해당 고유 계는 특이성을 가집니다. 특이성을 가지는 고유 계는 $\alpha = \beta = 0$ 를 가지게 됩니다. 이는 해당 고유 계가 제대로 정의된 고유 값과 벡터를 가지지 않음을 의미합니다. 이 단원의 함수들은 특이성을 가지지 않는 일반적인 문제들에 대한 풀이를 제공합니다. 이러한 특이 pencil 값을 가지는 문제에 함수들을 적용할 경우 예기치 못한 결과가 나올 수 있습니다.

행렬 쌍 (A, B) 의 실수 위의 일반화된 비 대칭 고유 계 문제는 일반화된 슈어 분해와 관련되어 있습니다.

$$A = QSZ^T$$
$$B = QTZ^T$$

Q 와 Z 는 각각 좌, 우 슈어 벡터들의 직교 행렬입니다. (S, T) 는 일반화된 슈어 형태로 이들의 대각 성분은 각각 α 와 β 값들을 제공해줍니다. 이 알고리즘들은 Moler & StewartQZ 의 QZ 방법에 기반해 있습니다. (참고 문헌을 확인할 수 있습니다.)

type **gsl_eigen_gen_workspace**

일반화된 비대칭 행렬의 고유 값 문제를 풀기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

problems.

gsl_eigen_gen_workspace ***gsl_eigen_gen_alloc**(const size_t n)

n × n 크기의 실수 일반화된 비대칭 행렬의 고유 값을 찾기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(n)$ 입니다.

void **gsl_eigen_gen_free**(gsl_eigen_gen_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

void **gsl_eigen_gen_params**(const int compute_s, const int compute_t, const int balance, gsl_eigen_gen_workspace *w)

gsl_eigen_gen() 을 사용하기 위한 계수들을 설정합니다.

compute_s 가 1로 정해지면 완전한 슈어 형태 S 함수 gsl_eigen_gen() 에 의해 계산됩니다. 0이 주어지면 S 는 계산되지 않습니다. 0이 기본 설정입니다. S 는 Schur 형태로 Quasi-상 삼각 행렬입니다. Quasi-상 삼각 행렬은 블럭 상 삼각 행렬로 대각 성분이 1×1 이나 2×2 크기의 블럭 행렬인 행렬을 의미합니다. 이 때, 2×2 크기 행렬의 고유 값들은 A 행렬의 고유 값들의 켤레 복소수입니다. 1×1 행렬은 실수 고유 값과 대응됩니다.

compute_t 가 1로 정해지면 완전한 슈어 형태 T 가 함수 gsl_eigen_gen() 에 의해 계산됩니다. 0이 주어지면 T 는 계산되지 않습니다. 0이 기본 설정입니다. T 는 상 삼각 행렬로 대각선에 음수 값이 없는 행렬입니다. S 의 2×2 크기 블럭 행렬은 T 의 2×2 크기 블럭 행렬에 대응됩니다.

balance 값은 현재 사용되지 않습니다. 잃반화된 균형 행렬기능은 아직 구현되지 않았습니다.

int **gsl_eigen_gen**(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta, gsl_eigen_gen_workspace *w)

실수 위 일반화된 비대칭 행렬 쌍 (A, B) 의 고유 값을 계산해 (alpha, beta)에 저장합니다. alpha 는 복소수, beta 는 실수입니다. β_i 가 0이 아니라면 $\lambda = \alpha_i/\beta_i$ 이 고유 값이 됩니다. 비슷하게 α_i 가 0이 아니라면 $\mu = \beta_i/\alpha_i$ 이 $\mu Ay = By$ 의 고유 값이 됩니다. beta 의 원소들은 모두 음수가 되지 않도록 정규화되어 있습니다.

S 가 필요하다면 이 행렬은 계산이 끝난 후 A 에 저장되어 있습니다. T 도 계산이 끝난 후 B 에 저장됩니다. (alpha, beta) 의 고유 값 순서는 슈어 형태 S 와 T 의 대각선 블럭들 순서를 따릅니다. 드문 일이지만 함수에서 모든 고유 값들을 찾지 못하는 경우가 있습니다. 이 경우 오류 값이 반환됩니다.

int **gsl_eigen_gen_QZ**(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta, gsl_matrix *Q, gsl_matrix *Z, gsl_eigen_gen_workspace *w)

gsl_eigen_gen() 함수와 동일합니다. 다만 좌, 우 슈어 벡터들을 계산해 각각 Q 와 Z 에 저장합니다.

type **gsl_eigen_genv_workspace**

일반화된 고유 값과 고유 벡터를 계산하기 위한 내부 인자들을 가지고 있는 작업 공간입니다.

gsl_eigen_genv_workspace *gsl_eigen_genv_alloc(const size_t n)

실수 위 크기 $n \times n$ 의 일반화된 비 대칭 고유 계의 고유 값과 고유 벡터를 찾기 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(7n)$ 입니다.

void gsl_eigen_genv_free(gsl_eigen_genv_workspace *w)

w 가 가르키는 작업 공간 영역을 해제합니다.

int gsl_eigen_genv(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta, gsl_matrix_complex *evec, gsl_eigen_genv_workspace *w)

실수 위 크기 $n \times n$ 의 일반화된 비 대칭 행렬 쌍(A , B)의 고유 값과 우 고유 벡터를 찾습니다. 고유 벡터들은 (alpha , beta)에 저장되고 고유 벡터들은 evec 에 저장됩니다. 이 함수는 먼저 함수 gsl_eigen_gen() 를 호출해 고유 값들과 슈어 형태 그리고 슈어 벡터를 계산합니다. 그 후 슈어 형태에 대한 고유 벡터를 찾고 이들을 슈어 벡터를 이용해 역변환 시켜 원래 계의 고유 벡터들을 얻습니다. 슈어 벡터들은 계산과정에서 사용되 훼손됩니다. 함수 gsl_eigen_genv_QZ() 를 사용하면 별도로 저장해둘 수 있습니다. 계산된 고유 벡터는 크기 1을 가지도록 정규화 되어 있습니다. 계산 후 (A , B) 는 일반화된 슈어 형태 (S , T)가 됩니다. 함수 gsl_eigen_gen() 의 구동이 실패하면 고유 벡터들은 계산되지 않고 오류 코드가 반환됩니다.

int gsl_eigen_genv_QZ(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta, gsl_matrix_complex *evec, gsl_matrix *Q, gsl_matrix *Z, gsl_eigen_genv_workspace *w)

gsl_eigen_genv() 와 동일합니다. 다만, 좌우 슈어 벡터들을 계산해 각각 Q 와 Z 에 저장합니다.

15.7 고유 값과 벡터의 정렬

int gsl_eigen_symmv_sort(gsl_vector *eval, gsl_matrix *evec, gsl_eigen_sort_t sort_type)

eval 벡터에 저장된 고유 값과, 대응되는 실수 고유 벡터들을 동시에 정렬합니다. 해당 벡터들은 행렬 evec 에 열 벡터로 저장되어 있습니다. sort_type 의 값에 따라 내림, 오름차순 정렬이 결정됩니다.

type gsl_eigen_sort_t

GSL_EIGEN_SORT_VAL_ASC	수치 값에 따른 오름차순 정렬
GSL_EIGEN_SORT_VAL_DESC	수치 값에 따른 내림차순 정렬
GSL_EIGEN_SORT_ABS_ASC	크기에 따른 오름차순 정렬
GSL_EIGEN_SORT_ABS_DESC	크기에 따른 내림차순 정렬

int gsl_eigen_hermv_sort(gsl_vector *eval, gsl_matrix_complex *evec, gsl_eigen_sort_t sort_type)

eval 벡터에 저장된 고유 값과, 행렬 evec 에 열 벡터로 저장된 복소수 고유 벡터들을 동시에 정렬합

니다. sort_type 의 값에 따라 내림, 오름차순 정렬이 결정됩니다. 해당 값들은 위 표에 있습니다.

int **gsl_eigen_nonsymmv_sort**(gsl_vector_complex *eval, gsl_matrix_complex *evec, gsl_eigen_sort_t sort_type)

eval 벡터에 저장된 고유 값과, 대응되는 복소수 고유 벡터들을 동시에 정렬합니다. 해당 벡터들은 행렬 evec 에 열 벡터로 저장되어 있습니다. sort_type 의 값에 따라 내림, 오름차순 정렬이 결정됩니다. 해당 값들은 위 표에 있습니다. 고유 값들이 복소수이기 때문에 GSL_EIGEN_SORT_ABS_ASC 와 GSL_EIGEN_SORT_ABS_DESC 매크로만 사용 가능합니다.

int **gsl_eigen_gensymmv_sort**(gsl_vector *eval, gsl_matrix *evec, gsl_eigen_sort_t sort_type)

eval 벡터에 저장된 고유 값과, 대응되는 실수 고유 벡터들을 동시에 정렬합니다. 해당 벡터들은 행렬 evec 에 열 벡터로 저장되어 있습니다. sort_type 의 값에 따라 내림, 오름차순 정렬이 결정됩니다. sort_type 값은 이전 표에 기술되어 있습니다.

int **gsl_eigen_genhermv_sort**(gsl_vector *eval, gsl_matrix_complex *evec, gsl_eigen_sort_t sort_type)

eval 벡터에 저장된 고유 값과, 대응되는 복소수 고유 벡터들을 동시에 정렬합니다. 해당 벡터들은 행렬 evec 에 열 벡터로 저장되어 있습니다. sort_type 의 값에 따라 내림, 오름차순 정렬이 결정됩니다. sort_type 값은 이전 표에 기술되어 있습니다.

int **gsl_eigen_genv_sort**(gsl_vector_complex *alpha, gsl_vector *beta, gsl_matrix_complex *evec, gsl_eigen_sort_t sort_type)

두 벡터 alpha, beta 에 저장된 고유 값들과 대응 되는 복소수 고유 벡터들을 동시에 정렬합니다. 대응되는 고유 벡터들은 행렬 evec 에 열 벡터로 저장되어 있습니다. sort_type 의 값에 따라 내림, 오름차순 정렬이 결정됩니다. sort_type 값은 이전 표에 기술되어 있습니다. 고유 값들이 복소수이기 때문에 GSL_EIGEN_SORT_ABS_ASC 와 GSL_EIGEN_SORT_ABS_DESC 매크로만 사용 가능합니다.

15.8 예제

다음 프로그램은 4차 힐베르트 행렬 $H(i,j) = 1/(i+j+1)$ 의 고유 값과 고유 벡터를 계산합니다.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>

int
main (void)
{
  double data[] = { 1.0   , 1/2.0, 1/3.0, 1/4.0,
                    1/2.0, 1/3.0, 1/4.0, 1/5.0,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                  1/3.0, 1/4.0, 1/5.0, 1/6.0,
                  1/4.0, 1/5.0, 1/6.0, 1/7.0 };

  gsl_matrix_view m
    = gsl_matrix_view_array (data, 4, 4);

  gsl_vector *eval = gsl_vector_alloc (4);
  gsl_matrix *evec = gsl_matrix_alloc (4, 4);

  gsl_eigen_symmv_workspace * w =
    gsl_eigen_symmv_alloc (4);

  gsl_eigen_symmv (&m.matrix, eval, evec, w);

  gsl_eigen_symmv_free (w);

  gsl_eigen_symmv_sort (eval, evec,
                        GSL_EIGEN_SORT_ABS_ASC);

  {
    int i;

    for (i = 0; i < 4; i++)
      {
        double eval_i
           = gsl_vector_get (eval, i);
        gsl_vector_view evec_i
           = gsl_matrix_column (evec, i);

        printf ("eigenvalue = %g\n", eval_i);
        printf ("eigenvector = \n");
        gsl_vector_fprintf (stdout,
                            &evec_i.vector, "%g");
      }
  }

  gsl_vector_free (eval);
  gsl_matrix_free (evec);

  return 0;
}
```

다음은 프로그램의 실행 결과입니다.

```
$ ./a.out
eigenvalue = 9.67023e-05
eigenvector =
-0.0291933
0.328712
-0.791411
0.514553
...
```

GNU octave 의 결과 값과 비교해 볼 수 있습니다.

```
octave> [v,d] = eig(hilb(4));
octave> diag(d)
ans =

   9.6702e-05
   6.7383e-03
   1.6914e-01
   1.5002e+00

octave> v
v =

   0.029193   0.179186  -0.582076   0.792608
  -0.328712  -0.741918   0.370502   0.451923
   0.791411   0.100228   0.509579   0.322416
  -0.514553   0.638283   0.514048   0.252161
```

참고: 고유 벡터들은 부호에 의해 달라질 수 있습니다. 고유 벡터들의 부호는 무작위로 정해집니다.

다음 프로그램은 비대칭 행렬의 고유문제를 계산합니다. 이 프로그램은 $x = (-1, -2, 3, 4)$ 에 대해, Vandermonde 행렬 $V(x; i, j) = x_i^{n-j}$ 을 사용했습니다.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>

int
main (void)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
{
  double data[] = { -1.0, 1.0, -1.0, 1.0,
                    -8.0, 4.0, -2.0, 1.0,
                    27.0, 9.0, 3.0, 1.0,
                    64.0, 16.0, 4.0, 1.0 };

  gsl_matrix_view m
    = gsl_matrix_view_array (data, 4, 4);

  gsl_vector_complex *eval = gsl_vector_complex_alloc (4);
  gsl_matrix_complex *evec = gsl_matrix_complex_alloc (4, 4);

  gsl_eigen_nonsymmv_workspace * w =
    gsl_eigen_nonsymmv_alloc (4);

  gsl_eigen_nonsymmv (&m.matrix, eval, evec, w);

  gsl_eigen_nonsymmv_free (w);

  gsl_eigen_nonsymmv_sort (eval, evec,
                           GSL_EIGEN_SORT_ABS_DESC);

  {
    int i, j;

    for (i = 0; i < 4; i++)
      {
        gsl_complex eval_i
           = gsl_vector_complex_get (eval, i);
        gsl_vector_complex_view evec_i
           = gsl_matrix_complex_column (evec, i);

        printf ("eigenvalue = %g + %gi\n",
                GSL_REAL(eval_i), GSL_IMAG(eval_i));
        printf ("eigenvector = \n");
        for (j = 0; j < 4; ++j)
          {
            gsl_complex z =
              gsl_vector_complex_get(&evec_i.vector, j);
            printf("%g + %gi\n", GSL_REAL(z), GSL_IMAG(z));
          }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      }
  }

  gsl_vector_complex_free(eval);
  gsl_matrix_complex_free(evec);

  return 0;
}
```

다음은 프로그램의 실행 결과입니다.

```
$ ./a.out
eigenvalue = -6.41391 + 0i
eigenvector =
-0.0998822 + 0i
-0.111251 + 0i
0.292501 + 0i
0.944505 + 0i
eigenvalue = 5.54555 + 3.08545i
eigenvector =
-0.043487 + -0.0076308i
0.0642377 + -0.142127i
-0.515253 + 0.0405118i
-0.840592 + -0.00148565i
...
```

GNU octave 의 결과 값과 비교해 볼 수 있습니다.

```
octave> [v,d] = eig(vander([-1 -2 3 4]));
octave> diag(d)
ans =

  -6.4139 + 0.0000i
   5.5456 + 3.0854i
   5.5456 - 3.0854i
   2.3228 + 0.0000i

octave> v
v =

 Columns 1 through 3:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 -0.09988 + 0.00000i  -0.04350 - 0.00755i  -0.04350 + 0.00755i
 -0.11125 + 0.00000i   0.06399 - 0.14224i   0.06399 + 0.14224i
  0.29250 + 0.00000i  -0.51518 + 0.04142i  -0.51518 - 0.04142i
  0.94451 + 0.00000i  -0.84059 + 0.00000i  -0.84059 - 0.00000i

Column 4:

 -0.14493 + 0.00000i
  0.35660 + 0.00000i
  0.91937 + 0.00000i
  0.08118 + 0.00000i
```

참고: 고유 값 $5.54555 + 3.08545i$ 의 대응 고유 벡터는 곱 계수 $0.9999984 + 0.0017674i$ 에 의해 다릅니다. 이 계수는 크기 1의 임의의 위상 계수입니다.

15.9 참고 자료와 추가 문헌

이 단원에 기술된 알고리즘들은 다음 책에 자세히 기술되어 있습니다.

- G. H. Golub, C. F. Van Loan, “Matrix Computations” (3rd Ed, 1996), Johns Hopkins University Press, ISBN 0-8018-5414-8.

일반화된 고유계의 QZ 알고리즘은 다음 논문을 참고할 수 있습니다.

- C. Moler, G. Stewart, “An Algorithm for Generalized Matrix Eigenvalue Problems”, SIAM J. Numer. Anal., Vol 10, No 2, 1973.

큰 규모의 행렬들에 대한 고유 문제 풀이 기능들은 포트란 라이브러리 LAPACK 에서 찾을 수 있습니다. LAPACK 은 다음 문헌에 기술되어 있습니다.

- LAPACK Users’ Guide (Third Edition, 1999), Published by SIAM, ISBN 0-89871-447-8.

LAPACK 의 소스코드는 사용자 설명서와 함께 http://www.netlib.org/lapack 에서 찾을 수 있습니다.

제 16 장

고속 푸리에 변환

이 단원에서는 고속 푸리에 변환(FFTs)을 위한 함수들을 기술합니다. 라이브러리에서는 radix-2 방법과 mixed-radix 방법을 제공합니다. 효율성을 위해서 각 기능들은 복소수용 함수와 실수용 함수로 나뉘어져 있습니다. Mixed-radix 함수들은 FFTPACK 라이브러리를 재구현한 형태입니다. FFTPACK의 포트란 코드는 Netlib에서 확인할 수 있습니다(FFTPACK에는 sine, cosine 변환 기능도 제공하지만 GSL 에서 더이상 제공하지 않습니다.). 본 문서에서 기술되는 알고리즘들에 대한 더 자세한 정보와 파생사항들은 "GSL FFT Algorithms" 문서를 참고할 수 있습니다. (참고 문헌과 추가 자료 참고)

16.1 수학적 정의

고속 푸리에 변환은 이산 푸리에 변환 (DFT)를 효율적으로 계산하기 위한 알고리즘들을 의미합니다.

$$x_j = \sum_{k=0}^{n-1} z_k \exp(-2\pi i j k / n)$$

DFT는 연속 푸리에 변환을 근사할 필요가 있을 때 사용됩니다. 일반적으로 공간과 시간에 대해 이산 간격으로 샘플들이 주어졌을 때 사용합니다. 기초적인 이산 푸리에 변환은 행렬-벡터곱 $W\vec{z}$ 으로 계산됩니다. 일반적인 행렬-곱의 계산 복잡도는 n 개의 데이터에 대해 $O(n^2)$ 의 복잡도를 가집니다. 고속 푸리에 변환은 분할 정복 전략을 사용해 행렬 W 를 여러개의 작은 하위-행렬들과 그에 상응하는 길이 n 의 정수들로 분할해 연산을 수행합니다. n 이 여러개의 정수들의 곱 $f_1 f_2 \ldots fm$ 으로 표현될 수 있다면 DFT는 $O(n \sum f_i)$ 의 시간 복잡도를 가집니다. radix-2 FFT 방법은 $O(n \log_2 n)$ 의 복잡도를 가집니다.

모든 FFT 함수들은 3가지 형태의 변환을 제공합니다. 변환, 역변환, 그리고 조정 계수가 없는 역변환입니다. 이들은 같은 수학적 정의에 기반해 있습니다.

고속 푸리에 변환은 $x = \mathrm{FFT}(z)$ 로 다음과 같이 정의됩니다.

$$x_j = \sum_{k=0}^{n-1} z_k \exp(-2\pi i j k / n)$$

역변환은 $x = \mathrm{IFFT}(z)$ 로 다음과 같이 정의됩니다.

$$z_j = \sum_{k=0}^{n-1} x_k \exp(2\pi i j k / n)$$

계수 $1/n$ 로 완전한 역변환을 얻을 수 있습니다.

예를 들어서, gsl_fft_complex_forward() 와 gsl_fft_complex_inverse() 를 합성하면 완전히 같은 값을 얻을 수 있습니다(수치적 오류가 같이 나올 수 있습니다).

일반적으로, 변환-역변환 과정에서 지수 함수의 지수 부호를 $\pm$ 중 자유롭게 선택할 수 있습니다. GSL 에서는 FFTPACK과 같은 규약을 따릅니다. 이 규약은 변환에서 음의 부호를 역변환에서는 양의 부호를 사용합니다. 이러한 부호 규칙의 이점은 역변환으로 본래의 함수를 다시 만들 수 있다는 점입니다. 반면, Numerical Recipes 부호 규약이 반대입니다. 변환에 양의 부호를 역변환에 음의 부호를 사용합니다.

조정 계수 없는 역 변환은 backwards FFT 로 불립니다. 이는 말 그대로 역 변환에서 크기를 조정하는 계수를 생략한 변환입니다.

$$z_j^{backwards} = \sum_{k=0}^{n-1} x_k \exp(2\pi i j k / n)$$

결과물의 구체적인 크기가 중요하지 않을 때 조정계수가 없는 변환은 역변환에서 불필요한 나눗셈을 줄일 수 있습니다.

16.2 복소수 FFTs 개요

복소수 FFT 함수들은 부동 소수점 배열을 입출력에 사용합니다. 이러한 배열은 실수-허수부가 번갈아 할당된 형태로 사용됩니다. 예를 들어서, 다음은 길이 6 의 복소 FFT 함수 입력용 배열입니다.

```
double x[3*2];
gsl_compleX_packed_array data = x;
```

이 배열은 길이 3 의 복소수 배열 z[3] 와 같이 사용할 수 있습니다.

```
data[0] = Re(z[0])
data[1] = Re(z[0])
data[2] = Re(z[1])
data[3] = Re(z[1])
data[4] = Re(z[2])
data[5] = Re(z[2])
```

배열의 인덱스는 DFT의 수학 정의에 사용된 인덱스와 동일한 순서를 가집니다. 별도의 인덱스 변환을 할 필요가 없습니다.

stride 인자는 사용자 함수를 사용할 때, z[i] 대신 z[strode*i] 형태로 변환을 할 수 있게 해줍니다. 1 보다 큰 이 값은 행렬의 열들에 대해 FFT를 적용할 수 있게 해줍니다. 1 값을 가지면, 배열 사이에 건너뛰는 값 없이 변환을 할 수 있습니다.

벡터 인자들에 대해 FFT를 적용할 수 있습니다. 예를 들어 gsl_vector_complex * v 사용하면 다음과 같은 형태로 이 단원의 함수들을 사용할 수 있습니다.

```
gsl_complex_packed_array data = v -> data;
size_t *stride* = v->stride;
size_t *n* = v->size;
```

물리학등의 응용에서 기억해 두어야 할 점은, DFT의 인덱스들이 물리적 주파수에 그대로 대응되지는 않는다는 점입니다. 만약 DFT의 시간 간격이 Δ 라면, 주파수 공간은 $-1/(2\Delta)$ 에서 $1/(2\Delta)$ 로 주파수의 부호가 $\pm$ 을 모두 가집니다. $+$ 주파수는 배열의 시작 지점에서 중간지점까지, $-$ 의 주파수는 배열의 끝에서 중간까지 공간에 저장됩니다.

다음은 배열 data 자세한 정보와 대응되는 시간-공간의 z , 주파수-공간의 x 를 나타낸 표입니다.

```
index    z              x = FFT(z)

0        z(t = 0)       x(f = 0)
1        z(t = 1)       x(f = 1/(n Delta))
2        z(t = 2)       x(f = 2/(n Delta))
.        ........       ..................
n/2      z(t = n/2)     x(f = +1/(2 Delta),
                                -1/(2 Delta))
.        ........       ..................
n-3      z(t = n-3)     x(f = -3/(n Delta))
n-2      z(t = n-2)     x(f = -2/(n Delta))
n-1      z(t = n-1)     x(f = -1/(n Delta))
```

n 이 짝수일 때, $n/2$ 지점은 양수와 음수 부호의 주파수 $(+1/(2\Delta), -1/(2\Delta))$ 값을 모두 가집니다. 이 둘은 같습니다. n 이 홀수라면, 위의 표와 같은 구조를 가집니다. 하지만 $n/2$ 는 존재하지 않습니다.

16.3 복소수 Radix-2 FFTs

이 단원에서 서술하는 radix-2 알고리즘은 간단하고 작지만, 가장 효율적인 방법은 아닙니다. 이 방법은 Cooley-Tukey 알고리즘을 이용해서 길이가 $2^m, m \in \mathbb{N}$ 인 복소수 데이터의 FFTs를 계산합니다. 계산 과정에서 해당 배열 외에 다른 저장소가 필요 없습니다.

대응되는 자기 정렬 mixed-radix 방법은 계산 과정에서 별도의 공간을 사용해 더 좋은 효율을 보여줍니다.

모든 함수들은 헤더파일 gsl_fft_complex.h 에 기술되어 있습니다.

int **gsl_fft_complex_radix2_forward**(gsl_complex_packed_array data, size_t stride, size_t n)

int **gsl_fft_complex_radix2_transform**(gsl_complex_packed_array data, size_t stride, size_t n, gsl_fft_direction sign)

int **gsl_fft_complex_radix2_backward**(gsl_complex_packed_array data, size_t stride, size_t n)

int **gsl_fft_complex_radix2_inverse**(gsl_complex_packed_array data, size_t stride, size_t n)

변환, 역변환, 그리고 조정계수 없는 역변환을 주어진 길이 `n` 과 `stride` 이용해 복소수 배열 `data` 에 적용합니다. 이 함수들은 실시간 선택 알고리즘인 radix-2를 사용하며 결과값은 `data` 그대로 저장됩니다. 배열의 길이 `n` 은 반드시 $2^m, m \in \mathbb{N}$ 여야 합니다. `transform` 이 붙은 함수에 관해, `sign` 인자는 `forward` $(+1)$ 나 `backward` (-1)가 될 수 있습니다.

오류 없이 변환이 정상적으로 완료되면 각 함수들은 `GSL_SUCESS` 값을 반환합니다. 만약 `n` 2 의 배수가 아닐 때에는 `GSL_ECOM` 값을 반환합니다.

int **gsl_fft_complex_radix2_dif_forward**(gsl_complex_packed_array data, size_t stride, size_t n)

int **gsl_fft_complex_radix2_dif_transform**(gsl_complex_packed_array data, size_t stride, size_t n, gsl_fft_direction sign)

int **gsl_fft_complex_radix2_dif_backward**(gsl_complex_packed_array data, size_t stride, size_t n)

int **gsl_fft_complex_radix2_dif_inverse**(gsl_complex_packed_array data, size_t stride, size_t n)

Radix-2 변환들의 decimation-in-frequency 형태 입니다.

다음은 짧은 펄스파의 128 길이 샘플 데이터를 고속 푸리에 변환하는 예제 프로그램입니다. 푸리에 변환 결과를 실수로 얻기 위해 펄스는 시간 공간에서 원점을 중심으로 대칭인 범주 $(-10 \ldots 10)$에 대해 정의되어 있습니다. 여기서 음수인 시간은 배열의 끝에서 반올림 됩니다.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

#define REAL(z,i) ((z)[2*(i)])
#define IMAG(z,i) ((z)[2*(i)+1])

int
main (void)
{
  int i; double data[2*128];
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  for (i = 0; i < 128; i++)
    {
       REAL(data,i) = 0.0; IMAG(data,i) = 0.0;
    }

  REAL(data,0) = 1.0;

  for (i = 1; i <= 10; i++)
    {
       REAL(data,i) = REAL(data,128-i) = 1.0;
    }

  for (i = 0; i < 128; i++)
    {
      printf ("%d %e %e\n", i,
              REAL(data,i), IMAG(data,i));
    }
  printf ("\n\n");

  gsl_fft_complex_radix2_forward (data, 1, 128);

  for (i = 0; i < 128; i++)
    {
      printf ("%d %e %e\n", i,
              REAL(data,i)/sqrt(128),
              IMAG(data,i)/sqrt(128));
    }

  return 0;
}
```

프로그램에서 암묵적으로 기본 오류 관리자(오류를 발견하면 프로그램을 정지합니다)를 사용함을 가정함을 알고 있어야합니다. 만약 안전한 오류 관리자를 사용하지 않는다면, `gsl_fft_complex_radix2_forward()`의 반환값을 검사해야합니다.

변환된 값들은 $1/\sqrt{n}$ 로 크기 조정이 이루어집니다. 따라서 변환된 결과 값은 입력된 초기 값과 잘 맞아떨어집니다. 입력 값을 적절히 조정해 허수부를 0 으로 만들었기 때문에 실수 값만을 볼 수 있습니다. $t = 128$ 단위로 변환이 이루어져, DFT는 연속 푸리에 변환을 근사해 변형된 sine 함수를 보여줍니다.

$$\int_{-a}^{+a} e^{-2\pi i k x} dx = \frac{\sin(2\pi k a)}{\pi k}$$

예제 프로그램의 결과는 다음과 같이 그래프로 그릴 수 있습니다 그림 16.1.

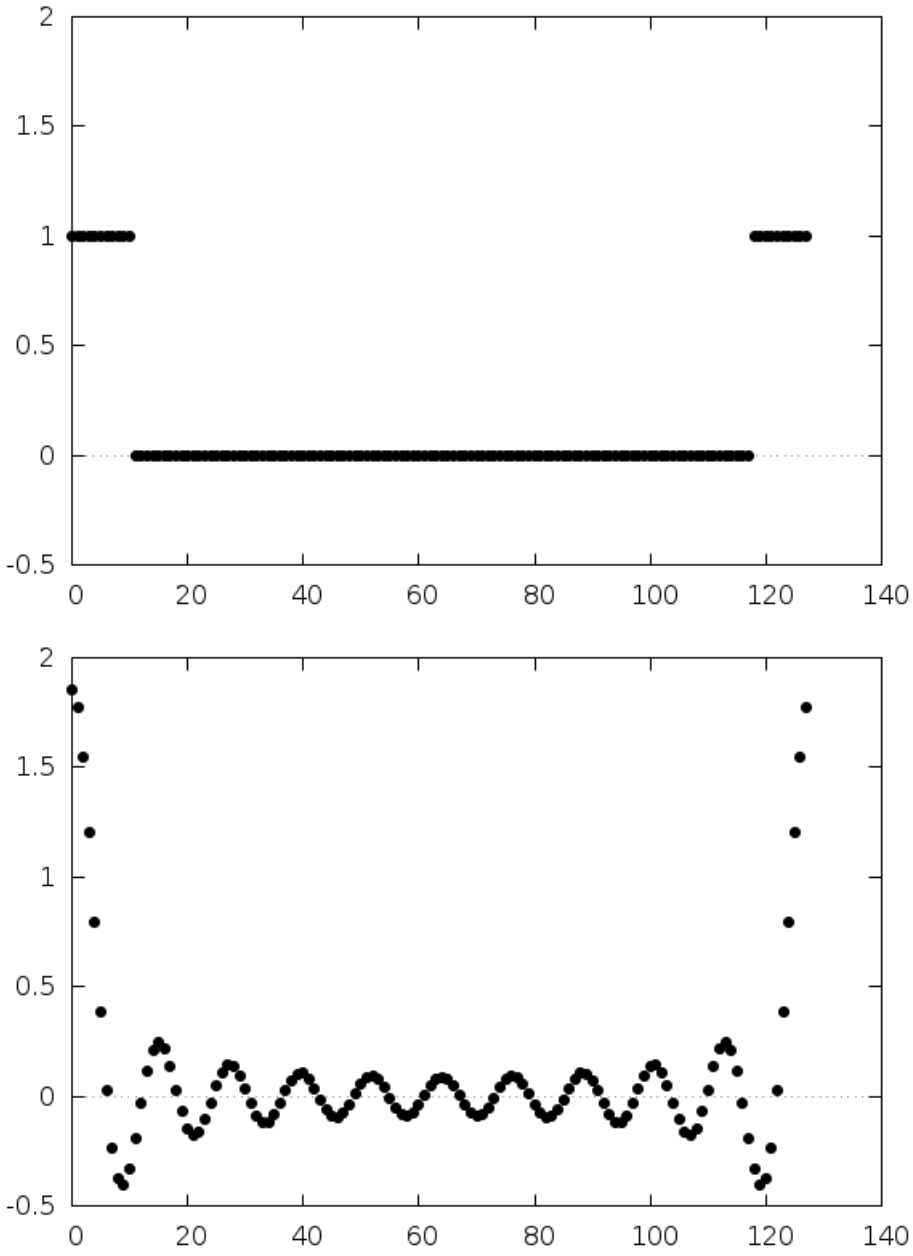


그림 16.1: 펄스 신호와 그 신호의 이산 푸리에 변환, 예제 프로그램의 결과

16.4 복소수 mixed-radix FFT

이 단원은 복소수 자료에 대한 mixed-radix FFT 알고리즘에 대해 서술합니다. Mixed-radix 함수들을 이용해 임의의 길이를 가지는 복소수 자료들의 고속 푸리에 변환을 얻을 수 있습니다. Paul Swartzrauber 가 작성한 포트란 라이브러리 FFTPACK의 재구현체를 포함하고 있습니다. 이 방법에 관한 이론은 Clive Temperton 이 작성한 리뷰 논문 “Self-sorting Mixed-radix FFTs”에서 설명하고 있습니다. 함수들은 FFTPACK과 인덱스 표기와 기초 알고리즘을 공유합니다.

Mixed-radix 알고리즘은 부분 변환 모듈에 기반해 있습니다. 잘 최적화된, 소규모 자료를 대상으로한 고속 푸리에 변환 모듈을 사용해서 더 큰 규모의 고속 푸리에 변환을 만들어내는 방법입니다. 2, 3, 4, 5, 6 그리고 7 단계에 대한 최적화된 연산 모듈을 포함하고 있습니다. 4와 6 규모의 고속 푸리에 변환 모듈은 $2 * 2$ 과 $2 * 3$ 으로 2, 3 단계의 변환 모듈을 합성한 것보다 빠른 속도를 보여줍니다.

모듈 내에 구현되지 않은 길이는 일반적인 길이 n 구현체로 계산해야 합니다. 해당 구현체는 Singleton 방법을 사용해 이산 푸리에 변환 효율적으로 계산합니다. 해당 모듈은 $O(n^2)$ 의 길이를 가지는 일반적인 자료를 사용하더라도 낮은 단계로 분해 할 수 있습니다. 예를 들어서 143 길이의 변환을 하고자 한다면 이는 $11 * 13$ 의 두 단계로 분해할 수 있습니다. 큰 소수를 인수로 가질 때가 가장 피해야하는 경우입니다. 예로, $2 * 3 * 99991$ 등이 있습니다. 이러한 경우 $O(n^2)$ 복잡도를 가지는 알고리즘의 연산에 대부분의 구동 시간을 소모하게 됩니다. 이러한 상황에 빠졌다면 GSL 베포 파일에 들어있는 “GSL FFT Algorithms” 문서를 참고하십시오.

Mixed-radix의 초기화 함수 `gsl_fft_complex_wavetable_alloc` 는 주어진 길이 n 의 크기를 가지게 됩니다. f_i 는 n 의 인자들입니다. 사용자가 작성한 프로그램에서 이러한 분해가 잘 이루어지지 않았을 때, 변환의 속도 저하를 경고하고 싶을 수도 있습니다. 라이브러리 내부에 있는 작은 규모의 소인수 분해 모듈에서 분해할 수 없는 길이의 모듈을 자주 사용한다면, “GSL FFT Algorithms” 문서를 참고해서 다른 인자들에 대한 지원을 추가할 수 있습니다.

이 단원에 기술된 함수들은 `gsl_fft_complex.h` 에 기술되어 있습니다.

gsl_fft_complex_wavetable *__gsl_fft_complex_wavetable_alloc__(size_t n)

> 길이 `n` 의 복소수 고속 푸리에 변환을 위한 삼각 함수 참고표를 생성합니다. 오류가 없다면 새로 할당된 `gsl_fft_complex_wavetable` 의 포인터를 반환합니다. 오류가 발생하면 `Null` 포인터를 반환합니다. 길이 `n` 은 여러 부분 변환들로 분해되고, 파동 참고표에 대응되는 삼각 함수 계수들이 저장됩니다. 삼각 함수 계수들은 정확도를 위해 `sin` 와 `cos` 를 사용해 직접 계산됩니다. 빠른 계산을 위해 재귀적 방법을 사용해 참고표를 작성할 수도 있습니다. 하지만, 프로그램에서 동일한 길이의 고속 푸리에 변환을 반복해서 수행한다면, 이 계산은 한번만 수행해도 최종 결과에 영향을 미치지 않습니다.
>
> 파동 참고표 구조체는 같은 길이를 사용하는 모든 변환에 반복적으로 사용할 수 있습니다. 이 표는 다른 고속 푸리에 변환 함수에 사용된다고 해서 값들이 변하지 않습니다. 같은 파동 참고표를 같은 길이를 가지는 값들의 변환, 역변환, 그리고 조정 계수가 없는 역변환 에 모두 사용할 수 있습니다.

void __gsl_fft_complex_wavetable_free__(gsl_fft_complex_wavetable *wavetable)

> 파동 참고표 `wavetable` 가 할당된 메모리를 해제합니다. 이 파동 참고표는 해당하는 길이의 고속 푸리에 변환이 더 이상 필요 없을 때 해제될 수 있습니다.

이 함수들의 연산은 `gsl_fft_complex_wavetable` 구조체에 적용됩니다. 이 구조체는 고속 푸리에 변환을 위한 내부 계수들을 가지고 있습니다. 해당 구조체의 내부 변수들을 직접 설정하는 것이 필수적이지는 않으나 해당 값들을 알고 있으면 몇몇 상황에서 유용할 수 있습니다. 예를 들어서 고속 푸리에 변환 길이에 대한 소인수 분해가 주어져 있으면, 실행 시간의 추정치나 수치적 오차를 계산하는 데 사용할 수 있습니다.

파동 참고표 구조체는 헤더 파일 `gsl_fft_complex.h` 에 정의되어 있습니다.

type **gsl_fft_complex_wavetable**

이 구조체는 mixed-radix fft 알고리즘을 위한 인수 분해와 삼각 함수 참고표를 담고 있습니다. 다음의 구성으로 이루어져 있습니다.

size_t n	복소수 지점들의 갯수
size_t nf	길이:code:n 이 분해된 인수들의 갯수
size_t factor[64]	인수들의 배열. 첫번째 nf 원소만이 사용됩니다.
gsl_complex * trig	길이 n 변환에 대해 사전 할당된 삼각 함수 참고표를 가르키는 포인터
gsl_complex * twiddle[64]	This is an array of pointers into trig , giving the twiddle factors for each pass.

type **gsl_fft_complex_workspace**

mixed-radix 알고리즘은 변환 중간 단계를 유지하기 위한 별도의 작업 공간이 필요합니다.

gsl_fft_complex_workspace ***gsl_fft_complex_workspace_alloc**(size_t n)

길이 n 의 복소수 값들의 변환을 위한 작업 공간을 반환합니다.

void **gsl_fft_complex_workspace_free**(gsl_fft_complex_workspace *workspace)

작업 공간 workspace 가 할당된 메모리를 해제합니다. 이 작업 공간은 동일한 길이의 변환이 더 이상 필요 없을 때 해제할 수 있습니다.

다음 함수들은 실제 변환을 계산합니다.

int **gsl_fft_complex_forward**(gsl_complex_packed_array data, size_t stride, size_t n, const gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work)

int **gsl_fft_complex_transform**(gsl_complex_packed_array data, size_t stride, size_t n, const gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work, gsl_fft_direction sign)

int **gsl_fft_complex_backward**(gsl_complex_packed_array data, size_t stride, size_t n, const gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work)

int **gsl_fft_complex_inverse**(gsl_complex_packed_array data, size_t stride, size_t n, const gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work)

이 함수들은 각각 고속 푸리에 변환, 역변환 그리고 조정 계수 없는 역변환을 주어진 길이 n 과 stride 걸음 크기를 가지는 복소수 값들 data 에 대해 계산합니다. mixed-radix decimation-in-frequency 알고리즘을 사용합니다. 길이 n 에 대한 제약은 존재하지 않습니다. 길이 2, 3, 4, 5, 6 그리고 7 에 대해 최적화된 부분 변환 모듈이 있습니다. 나머지 길이들은 $O(n^2)$ 의 시간 복잡도를 가지는 상대적으로 느린 방법으로 계산됩니다. 호출 과정에서 변환하는 값들의 길이에 맞는 파동 참고표 wavetable 와

작업 공간 work 를 인자로 제공해야합니다. wavetable 에는 참고용 삼각 함수 계수들을 포함하고 있습니다. transform 이 붙은 함수에 관해, sign 인자는 forward ($+1$) 나 backward (-1)가 될 수 있습니다.

오류가 없으면 0 값을 반환합니다. 다음은 이 함수를 위해 정의된 gsl_errno 의 오류 조건 입니다.

GSL_EDOM	자료의 길이 n 이 양의 정수가 아닐 때, (예 n=0).
GSL_EINVAL	자료의 길이 n 와 계산을 위해 사용하는 파동 참고표 wavetable 가 맞지 않을 때.

다음 프로그램은 Mixed-radix 알고리즘을 이용해 짧은 펄스에 대해, 길이 630 ($1 = 2 * 3 * 3 * 5 * 7$)의 고속 푸리에 변환을 계산하는 예제를 보여줍니다.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

#define REAL(z,i) ((z)[2*(i)])
#define IMAG(z,i) ((z)[2*(i)+1])

int
main (void)
{
  int i;
  const int n = 630;
  double data[2*n];

  gsl_fft_complex_wavetable * wavetable;
  gsl_fft_complex_workspace * workspace;

  for (i = 0; i < n; i++)
    {
      REAL(data,i) = 0.0;
      IMAG(data,i) = 0.0;
    }

  data[0] = 1.0;

  for (i = 1; i <= 10; i++)
    {
      REAL(data,i) = REAL(data,n-i) = 1.0;
    }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  for (i = 0; i < n; i++)
    {
      printf ("%d: %e %e\n", i, REAL(data,i),
                                 IMAG(data,i));
    }
  printf ("\n");

  wavetable = gsl_fft_complex_wavetable_alloc (n);
  workspace = gsl_fft_complex_workspace_alloc (n);

  for (i = 0; i < (int) wavetable->nf; i++)
    {
       printf ("# factor %d: %zu\n", i,
               wavetable->factor[i]);
    }

  gsl_fft_complex_forward (data, 1, n,
                           wavetable, workspace);

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e %e\n", i, REAL(data,i),
                                 IMAG(data,i));
    }

  gsl_fft_complex_wavetable_free (wavetable);
  gsl_fft_complex_workspace_free (workspace);
  return 0;
}
```

프로그램에서 암묵적으로 기본 오류 관리자(오류를 발견하면 프로그램을 정지합니다)를 사용함을 가정함을 알고 있어야합니다. 만약, 안전한 오류 관리자를 사용하지 않는다면 gsl 속 모든 함수들의 반환값을 검사해야합니다.

16.5 실수 FFTs 개요

실수 값들에 적용하는 고속 푸리에 변환은 복소수인 경우와 비슷하지만, 변환과 역변환에서 중요한 차이가 있습니다. 바로 실수 값들의 푸리에 변환은 실수 값이 아니라는 점입니다. 변환 된 값들은 다음과 같은 대칭성을 가지는 복소 수열로 나타나게 됩니다.

$$z_k = z^*_{n-k}$$

이러한 대칭성을 가지는 수열을 **컬레 복소** 나 반 복소성을 가지고 있다고 부릅니다. 이러한 구조적 차이점은 변환(실수->반 복소), 역변환(반 복소 -> 실수)에 대해 이전의 복소 푸리에 변환과 다른 설계를 필요로 합니다. 때문에, 실수 푸리에 변환의 함수들은 두 개의 범주로 나뉘어집니다. 실수 배열을 처리하는 `gsl_fft_real` 함수들과 반 복소 배열을 처리하는 `gsl_fft_halfcomplex` 함수들입니다.

`gsl_fft_real` 함수들은 실수 배열의 주파수 계수를 계산합니다. 실수 배열 x 에 대한, 반 복소 계수 c 는 푸리에 분석에 따라 다음과 같이 주어집니다.

$$c_k = \sum_{j=0}^{n-1} x_j \exp(-2\pi i j k / n)$$

`gsl_fft_halfcomplex` 함수들은 역변환과 조정 계수 없는 역변환을 계산합니다. 이 함수들은 반 복소 주파수 계수 c 로 부터 실수 배열을 재구성합니다.

$$x_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \exp(2\pi i j k / n)$$

반 복소 배열의 대칭성으로 인해 배열 전체의 절반만 저장해도 됩니다. 나머지 절반은 반 복소 대칭성으로 계산 과정에서 재구성 됩니다. 이 방법은 짝수, 홀수를 가리지 않고 모든 크기에 대해 사용할 수 있습니다. 짝수 크기의 배열에 대해, 중간값 $k/2$ 도 실수가 됩니다. 따라서 반 복소 배열의 계산에 필요한 것은 `n` 크기의 실수 저장 공간뿐입니다. 계산 결과인 실수 배열도 입력 배열과 같은 크기의 배열에 저장됩니다.

정확한 저장 공간의 관리는 알고리즘에 따라 다릅니다. radix-2와 mixed-radix에 따라 차이가 있습니다. radix-2 방법은 실수와 허수부가 저장되는 위치를 가능한 한 멀리 떨어지게 저장하도록 강제합니다. 반면, mixed-radix 방법은 이러한 제약이 없으며, 실수와 허수부가 인접 위치에 저장합니다. 이는 메모리 접근에서 인접성 향상에 좋습니다.

16.6 실수 Radix-2 FFTs

이 단원은 radix-2 고속 푸리에 변환 알고리즘을 실수 값들에 적용시키는 함수들에 대해 다룹니다. 이들은 Cooley-Tukey 알고리즘을 사용해 2 의 거듭 제곱 크기의 길이를 가지는 값들에 대해 계산합니다.

이 radix-2 고속 푸리에 변환 함수들은 헤더 파일 `gsl_fft_real.h` 에 저장되어 있습니다.

int **gsl_fft_real_radix2_transform**(double data[], size_t stride, size_t n)

radix-2 고속 푸리에 변환을 주어진 `stride` 간격과 길이 `n` 를 가지는 실수 배열 `data` 대해 계산합니다. 계산 결과는 반 복소 배열로 각 실수-허수 값들은 정해진 위치 규약에 따라 저장됩니다. 길이 n 의 배열에 대해, $k < n/2$ 대해, k 번째 복소수의 실수 값은 k 번째 배열에 저장되고, 대응 되는 허수 값은 $n-k$ 번째 배열에 저장됩니다. $k > n/2$ 인 항들은 대칭성 $z_k = z^*_{n-k}$ 에 의해 재구성 됩니다. $k = 0$ 이나 $k = n/2$ 인 경우는 항상 실수가 되며, 따로 처리됩니다. 실수 값은 0 과 $n/2$ 위치에 저장되며, 허수 값은 0 이므로 따로 저장되지 않습니다.

다음 표는 어느 실수 배열의 계산 값 `data` 허수부가 0 인 동일한 복소수 배열을 이용해 얻은 결과를 비교한 표입니다. (`stride` $= 1$ 로 가정합니다.)

```
complex[0].real    =    data[0]
complex[0].imag    =    0
complex[1].real    =    data[1]
complex[1].imag    =    data[n-1]
.............          .............
complex[k].real    =    data[k]
complex[k].imag    =    data[n-k]
.............          .............
complex[n/2].real  =    data[n/2]
complex[n/2].imag  =    0
.............          .............
complex[k'].real   =    data[k]       k' = n - k
complex[k'].imag   =   -data[n-k]
.............          .............
complex[n-1].real  =    data[1]
complex[n-1].imag  =   -data[n-1]
```

계산 결과는 나중에 기술될 함수 `gsl_fft_halfcomplex_radix2_unpack()` 로 완전한 복소수 배열로 바꿀 수 있습니다.

반 복소 배열을 위한 radix-2 고속 푸리에 변환 함수들은 헤더 파일 `gsl_fft_halfcomplex.h` 에 기술되어 있습니다.

int **gsl_fft_halfcomplex_radix2_inverse**(double data[], size_t stride, size_t n)

int **gsl_fft_halfcomplex_radix2_backward**(double data[], size_t stride, size_t n)

radix-2 고속 푸리에 변환의 역변환과 조정 계수 없는 역변환을 주어진 stride 간격과 길이 n 가지는 반 복소 배열 data 대해 계산합니다. 이때, data gsl_fft_real_radix2() 서 사용된 결과 배열과 동일한 저장 규칙을 가지고 있습니다. 결과로 나오는 실수 배열은 기본 배열 순서를 따릅니다.

int **gsl_fft_halfcomplex_radix2_unpack**(const double halfcomplex_coefficient[], gsl_complex_packed_array complex_coefficient, size_t stride, size_t n)

gsl_fft_real_radix2_transform() 함수의 계산 결과로 나오는 halfcomplex_coefficient 배열을 일반 복소수 complex_coefficient 배열로 변환합니다. 이 함수는 복소수 배열을 대칭성 $z_k = z^*_{n-k}$ 를 이용해, 중복되는 요소들을 재구성합니다.

알고리즘은 다음과 같습니다.

```
complex_coefficient[0].real = halfcomplex_coefficient[0];
complex_coefficient[0].imag = 0.0;

for (i = 1; i < n - i; i++)
  {
    double hc_real = halfcomplex_coefficient[i*stride];
    double hc_imag = halfcomplex_coefficient[(n-i)*stride];
    complex_coefficient[i*stride].real = hc_real;
    complex_coefficient[i*stride].imag = hc_imag;
    complex_coefficient[(n - i)*stride].real = hc_real;
    complex_coefficient[(n - i)*stride].imag = -hc_imag;
  }

if (i == n - i)
  {
    complex_coefficient[i*stride].real = halfcomplex_coefficient[(n - 1)*stride];
    complex_coefficient[i*stride].imag = 0.0;
  }
```

16.7 실수 mixed-radix FFTs

이 단원에서는 실수 값들을 위한 mixed-radix 고속 푸리에 알고리즘을 기술합니다. 고속 푸리에 변환을 위한 mixed-radix 함수들은 모든 크기의 값에 대해 적용할 수 있습니다. 이 함수들은 Paul Swarztrauber 가 포트란 FFRPACK 라이브러리에 작성한 실수 고속 푸리에 변환 기능들을 재구현한 형태입니다. 이 알고리즘에 사용된 이론들은 Clive Temperton이 작성한 “Fast Mixed-Radix Real Fourier Transforms”을 참고할 수 있습니다. 이 단원에 작성된 함수들은 FFTPACK에 있는 기반 알고리즘과 같은 인덱스 규약을

가집니다.

이 단원의 함수들은 FFTPACK의 반 복소 배열 저장 규약을 사용합니다. 이 규약에서 실수 배열의 반 복소 변환은 0 에서 증가하는 순서로, 실수-허수부가 번갈아 이웃해 가며 저장됩니다. 실수로 확인된 값에 대해서는 허수부가 저장되지 않습니다. 0 의 주파수를 가지는 부분에는 허수부가 저장되지 않습니다(). 크기가 짝수인 입력 배열에서는, $n/2$ 의 허수부가 저장되지 않습니다. 이는 $z_k = z^*_{n-k}$ 로 인해 자명하게 실수이기 때문입니다.

저장 규약은 다음 예시로 보는 것이 가장 정확합니다. 아래의 표는 $n = 5$ 의 홀수 길이 배열에 대한 계산 결과를 나타냅니다. 각 두 열은 대응되는 반-복소열 값 5 개를 나타내고 있습니다. 이 값들은 각각 gsl_fft_real_transform() 반환값 halfcomplex[] 나태납니다. complex[] 동일한 실수열이 gsl_fft_complex_backward() 복소수열로 전달 되었을 때 반환됩니다. 이 실수열은 허수부가 0 복소수열로 취급됩니다.

```
complex[0].real  =  halfcomplex[0]
complex[0].imag  =  0
complex[1].real  =  halfcomplex[1]
complex[1].imag  =  halfcomplex[2]
complex[2].real  =  halfcomplex[3]
complex[2].imag  =  halfcomplex[4]
complex[3].real  =  halfcomplex[3]
complex[3].imag  = -halfcomplex[4]
complex[4].real  =  halfcomplex[1]
complex[4].imag  = -halfcomplex[2]
```

complex 배열의 뒷 부분 , complex[3] complex[4] 는 대칭 조건으로 채워집니다. 대칭성에 의해 주파수 성분이 0 부분의 허수부 complex[0].imag 0 으로 정해집니다.

다음 표는 길이가 짝수인 배열의 결과를 나타내줍니다. $n = 6$ 인 경우입니다. 짝수 길이를 가지는 경우 완전히 실수인 성분이 두 개 생기게 됩니다.

```
complex[0].real  =  halfcomplex[0]
complex[0].imag  =  0
complex[1].real  =  halfcomplex[1]
complex[1].imag  =  halfcomplex[2]
complex[2].real  =  halfcomplex[3]
complex[2].imag  =  halfcomplex[4]
complex[3].real  =  halfcomplex[5]
complex[3].imag  =  0
complex[4].real  =  halfcomplex[3]
complex[4].imag  = -halfcomplex[4]
complex[5].real  =  halfcomplex[1]
complex[5].imag  = -halfcomplex[2]
```

complex 배열의 뒷부분, complex[4] complex[5] 대칭 조건에 의해 결정됩니다. complex[0].imag complex[3].imag 0 으로 정해집니다.

이 함수들은 헤더 파일 gsl_fft_real.h gsl_fft_halfcomplex.h 정의되어 있습니다.

type **gsl_fft_real_wavetable**

type **gsl_fft_halfcomplex_wavetable**

FFT를 위한 고정된 크기의 파동 자료표를 위한 구조체입니다.

gsl_fft_real_wavetable ***gsl_fft_real_wavetable_alloc**(size_t n)

gsl_fft_halfcomplex_wavetable ***gsl_fft_halfcomplex_wavetable_alloc**(size_t n)

math:n 길이의 실수 원소들에 대한 FFT에 쓰이는 고정된 크기의 삼각함수 파동 자료표를 제공합니다. 이 함수들은 새로 할당된 구조체를 가르키는 포인터를 반환하고 오류가 발견되면 Null 인터를 반환합니다. 길이 n 부분 변환들의 곱으로 분해되며, 인자와 삼각함수 계수들은 파동 자료표에 저장됩니다. 삼각함수 계수들은 정확도를 위해 sin cos 호출해 직접 계산됩니다. 속도를 위해 재귀적 관계를 사용해 계산할 수도 있습니다. 하지만, 응용 프로그램에서 많은 횟수의 FFT를 같은 길이의 자료들에 대해 계산한다면 이 파동 자료표는 최종 결과에 영향을 미치지 않습니다.

파동 자료표 구조체는 같은 크기의 자료를 변환하는데 반복적으로 사용될 수 있습니다. 이 표는 다른 FFT 함수들의 호출에서 변하지 않습니다. 순방향의 실수, 반복소 변환에 대해 적절한 유형의 파동 자료표를 사용해야 합니다.

void **gsl_fft_real_wavetable_free**(gsl_fft_real_wavetable *wavetable)

void **gsl_fft_halfcomplex_wavetable_free**(gsl_fft_halfcomplex_wavetable *wavetable)

파동 자료표 wavetable 에 할당된 메모리를 해제합니다. 파동 자료표의 해제는 동일한 크기의 FFT가 더 이상 필요 없을 때 해제될 수 있습니다.

Mixed-radix 알고리즘은 변환의 중간 단계를 유지하기 위해 추가적인 작업 공간을 필요로 합니다.

type **gsl_fft_real_workspace**

실수 FFT를 계산하기 위한 계수들을 가지고 있는 작업 공간입니다.

gsl_fft_real_workspace ***gsl_fft_real_workspace_alloc**(size_t n)

길이 n 의 실수 변환을 위한 작업 공간을 할당합니다. 같은 작업공간이 순방향 실수나 역방향 반 복소 변환에 대해 사용될 수 있습니다.

void **gsl_fft_real_workspace_free**(gsl_fft_real_workspace *workspace)

작업 공간 workspace 에 할당된 메모리를 해제합니다. 작업 공간은 동일한 크기의 FFT가 더이상 필요 없을 때 해제될 수 있습니다.

다음 함수들은 실수, 반복소 자료들에 대한 변환을 계산합니다.

int **gsl_fft_real_transform**(double data[], size_t stride, size_t n, const gsl_fft_real_wavetable *wavetable, gsl_fft_real_workspace *work)

int **gsl_fft_halfcomplex_transform**(double data[], size_t stride, size_t n, const gsl_fft_halfcomplex_wavetable *wavetable, gsl_fft_real_workspace *work)

주어진 길이 n 의 실수나 반복소 배열 값 data 대한 FFT를 계산합니다. mixed-radix decimation-in-frequency 알고리즘을 사용합니다. gsl_fft_real_transform() 에 대해, data 시간 순으로 배열된 실수 자료로 취급됩니다. gsl_fft_halfcomplex_transform() 에 대해, data 는 위에 서술한 반-복소 순서로 배열된 푸리에 계수를 가지고 있습니다. n 대한 제약은 없습니다. 효율을 위해 모듈들은 길이 $2, 3, 4$ 와 5 에 대한 부분 변환을 제공합니다. 일반적인 n 길이 모듈에 대해 나머지 값들은 $O(n^2)$ 의 속도로 느리게 계산됩니다. 호출시 삼각함수 파동 자료표를 포함하는 wavetable 작업 공간 work 와 같이 제공해야합니다.

int **gsl_fft_real_unpack**(const double real_coefficient[], gsl_complex_packed_array complex_coefficient, size_t stride, size_t n)

실수 배열 real_coefficient 을 동일한 복소수 배열 complex_coefficient 로 변환합니다. 이 복소수 배열은 허수부가 모두 0 이고 실수부가 실수 배열과 같은 배열이며, gsl_fft_complex 함수들에 대해 사용할 수 있습니다.

다음과 같이 간단하게 구현되어 있습니다.

```
for (i = 0; i < n; i++)
{
  complex_coefficient[i*stride].real = real_coefficient[i*stride];
  complex_coefficient[i*stride].imag = 0.0;
}
```

int **gsl_fft_halfcomplex_unpack**(const double halfcomplex_coefficient[], gsl_complex_packed_array complex_coefficient, size_t stride, size_t n)

gsl_fft_real_transform() 서 반환된 반복소 계수 배열 halfcompelex_coefficient 와 일반적인 복소수 배열 complex_coefficient 을 변환합니다. 이 함수는 복소수 배열을 $z_k = z^*_{n-k}$ 대칭성을 사용해 중복된 요소들을 재 구성합니다. 이 변환에 사용된 알고리즘은 다음과 같습니다.

```
complex_coefficient[0].real = halfcomplex_coefficient[0];
complex_coefficient[0].imag = 0.0;

for (i = 1; i < n - i; i++)
  {
    double hc_real = halfcomplex_coefficient[(2 * i - 1)*stride];
    double hc_imag = halfcomplex_coefficient[(2 * i)*stride];
    complex_coefficient[i*stride].real = hc_real;
    complex_coefficient[i*stride].imag = hc_imag;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    complex_coefficient[(n - i)*stride].real = hc_real;
    complex_coefficient[(n - i)*stride].imag = -hc_imag;
  }

if (i == n - i)
  {
    complex_coefficient[i*stride].real = halfcomplex_coefficient[(n - 1)*stride];
    complex_coefficient[i*stride].imag = 0.0;
  }
```

gsl_fft_real_transform() 과 gsl_fft_halfcomplex_inverse() 의 사용을 보여주는 예제 프로그램이 있습니다. 이 프로그램은 사각형 모양의 실수 신호를 생성합니다. 신호는 푸리에 변환을 통해 주파수 공간으로 변환돼고, 가장 낮은 10 개의 주파수는 푸리에 계수 배열에서 제거되어 gsl_fft_real_transform() 에 의해 반환됩니다.

나머지 푸리에 계수들은 다시 역변환되어 시간 공간으로 되돌아옵니다. 이는 필터를 거친 사각 신호를 나타냅니다. 푸리에 계수는 반 복소 대칭성을 사용해 저장되므로 양의 신호와 음의 신호는 모두 제거되고 최종 필터링된 신호도 실수가 됩니다.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_real.h>
#include <gsl/gsl_fft_halfcomplex.h>

int
main (void)
{
  int i, n = 100;
  double data[n];

  gsl_fft_real_wavetable * real;
  gsl_fft_halfcomplex_wavetable * hc;
  gsl_fft_real_workspace * work;

  for (i = 0; i < n; i++)
    {
      data[i] = 0.0;
    }

  for (i = n / 3; i < 2 * n / 3; i++)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    {
      data[i] = 1.0;
    }

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e\n", i, data[i]);
    }
  printf ("\n");

  work = gsl_fft_real_workspace_alloc (n);
  real = gsl_fft_real_wavetable_alloc (n);

  gsl_fft_real_transform (data, 1, n,
                          real, work);

  gsl_fft_real_wavetable_free (real);

  for (i = 11; i < n; i++)
    {
      data[i] = 0;
    }

  hc = gsl_fft_halfcomplex_wavetable_alloc (n);

  gsl_fft_halfcomplex_inverse (data, 1, n,
                               hc, work);
  gsl_fft_halfcomplex_wavetable_free (hc);

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e\n", i, data[i]);
    }

  gsl_fft_real_workspace_free (work);
  return 0;
}
```

이 프로그램의 결과는 다음 그림 그림 16.2 과 같습니다.

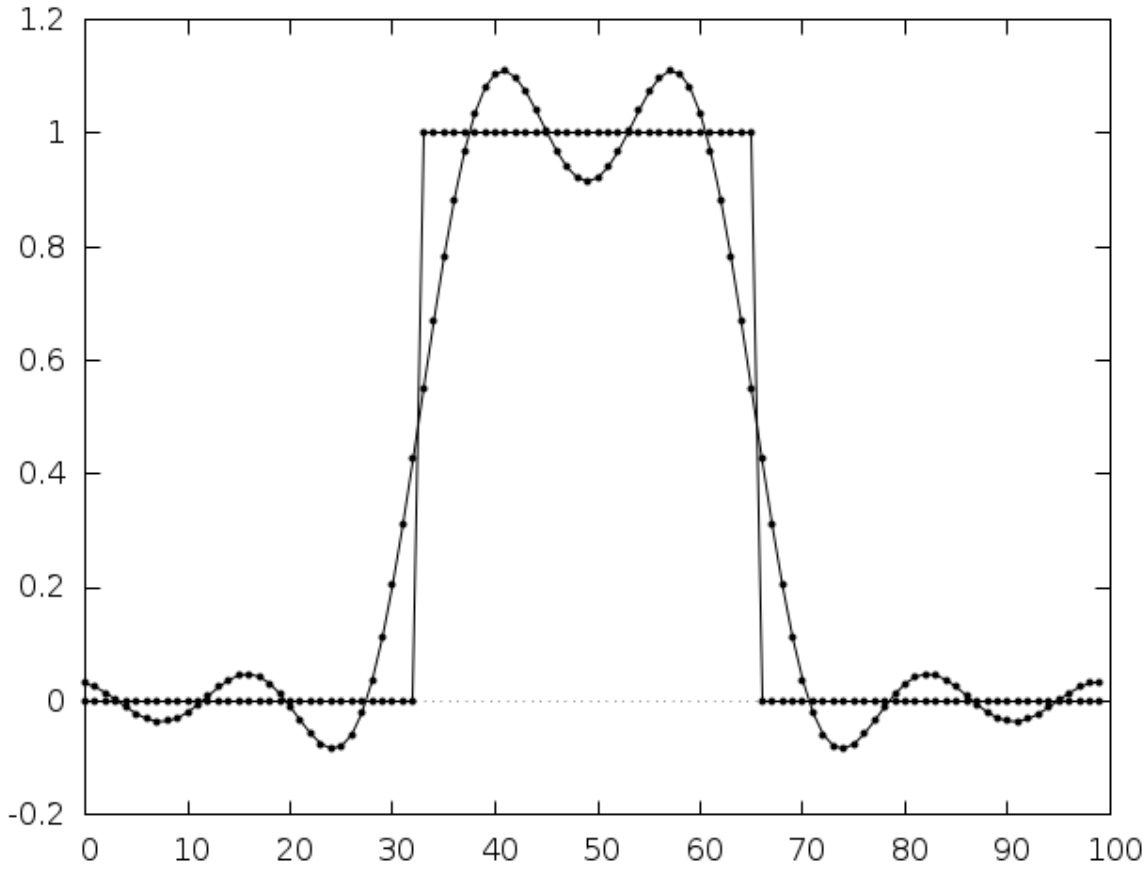


그림 16.2: 실수 펄스 신호의 저주파 통과 필터의 적용, 예제 프로그램의 결과.

16.8 참고 문헌과 추가 자료

FFT 입문에 다음의 리뷰 논문을 추천합니다.

- P. Duhamel and M. Vetterli. Fast Fourier transforms: A tutorial review and a state of the art. Signal Processing, 19:259-299, 1990.

GSL 구현체에 쓰인 알고리즘을 참고하기 위해서 "GSL FFT Algorithms" 문서를 참고할 수 있습니다[1] . 이 문서는 FFT에 관한 일반적인 정보들과 각각의 구현체와 그 알고리즘에 관한 여러 파생 정보들을 기술하고 있습니다. 이에 더해, 관련 참고 문헌들도 함께 제공하고 있습니다. 편의를 위해 중요한 참고문헌들을 이곳에 함께 제공합니다.

FFT를 예시 프로그램과 함께 제공하는 몇몇 입문 서적들이 있습니다. Brigham 저 "The Fast Fourier Transform"과 Burrus and Parks 저 "DFT/FFT and Convolution Algorithms"이 있습니다.

- 마. Oran Brigham. "The Fast Fourier Transform". Prentice Hall, 1974.
- 다. . Burrus and T. W. Parks. "DFT/FFT and Convolution Algorithms", Wiley, 1984.

이 입문 서적들은 radix-2 FFT 방법을 자세히 서술하고 있습니다. FFTPACK 구현체의 핵심 알고리즘인 Mixed-radix 알고리즘은 Clive Temperton의 논문에서 잘 다루어져 있습니다.

- Clive Temperton, Self-sorting mixed-radix fast Fourier transforms, Journal of Computational Physics, 52(1):1-23, 1983.

실수 데이터에 관한 FFTs의 파생 정보들은 다음의 두 논문에서 다루고 있습니다.

- Henrik V. Sorenson, Douglas L. Jones, Michael T. Heideman, and C. Sidney Burrus. Real-valued fast Fourier transform algorithms. "IEEE Transactions on Acoustics, Speech, and Signal Processing", ASSP-35(6):849-863, 1987.
- Clive Temperton. Fast mixed-radix real Fourier transforms. "Journal of Computational Physics", 52:340-350, 1983.

1979년도에, IEEE에서는 포트란 FFT 프로그램을 중점적으로 리뷰한 개괄 문서을 출판했습니다. 해당 문서는 "Programs for Digital Sigal Processinng"이란 제목으로 여러 다른 FFT 알고리즘들의 구현에 좋은 참고 문헌이 되어줍니다.

- Digital Signal Processing Committee and IEEE Acoustics, Speech, and Signal Processing Committee, editors. Programs for Digital Signal Processing. IEEE Press, 1979.

대규모 데이터에 FFT를 적용할 때, 이러한 작업 전용으로 만들어진 FFTW 라이브러리를 쓰는 것을 추천합니다. 이 라이브러리는 Frigo와 Johnson이 작성했습니다. FFTW 라이브러리는 구동 하드웨어에 맞추어 최고의 효율을 내도록 스스로를 최적화 합니다. 이 라이브러리는 GNU GPL하에서 사용가능합니다.

- FFTW Website, http://www.fftw.org/

[1] 이 문서는 GSL 라이브러리에 포함되어 있습니다. doc/fftalgorithms.tex 경로에 tex 문서로 되어있으며, pdf는 이곳 에서 내려받을 수 있습니다.

FFTPACK의 소스 코드는 http://www.netlib.org/fftpack/ 에서 확인할 수 있습니다.

제 17 장

수치 적분

참고: 번역중

This chapter describes routines for performing numerical integration (quadrature) of a function in one dimension. There are routines for adaptive and non-adaptive integration of general functions, with specialised routines for specific cases. These include integration over infinite and semi-infinite ranges, singular integrals, including logarithmic singularities, computation of Cauchy principal values and oscillatory integrals. The library reimplements the algorithms used in QUADPACK, a numerical integration package written by Piessens, de Doncker-Kapenga, Ueberhuber and Kahaner. Fortran code for QUADPACK is available on Netlib. Also included are non-adaptive, fixed-order Gauss-Legendre integration routines with high precision coefficients, as well as fixed-order quadrature rules for a variety of weighting functions from IQPACK.

The functions described in this chapter are declared in the header file `gsl_integration.h`.

17.1 Introduction

Each algorithm computes an approximation to a definite integral of the form,

$$I = \int_a^b f(x)w(x)dx$$

where $w(x)$ is a weight function (for general integrands $w(x) = 1$). The user provides absolute and relative error bounds $(epsabs, epsrel)$ which specify the following accuracy requirement,

$$|RESULT - I| \leq \max(epsabs, epsrel|I|)$$

where $RESULT$ is the numerical approximation obtained by the algorithm. The algorithms attempt to estimate the absolute error $ABSERR = |RESULT - I|$ in such a way that the following inequality holds,

$$|RESULT - I| \leq ABSERR \leq \max(epsabs, epsrel|I|)$$

In short, the routines return the first approximation which has an absolute error smaller than $epsabs$ or a relative error smaller than $epsrel$.

Note that this is an either-or constraint, not simultaneous. To compute to a specified absolute error, set $epsrel$ to zero. To compute to a specified relative error, set $epsabs$ to zero. The routines will fail to converge if the error bounds are too stringent, but always return the best approximation obtained up to that stage.

The algorithms in QUADPACK use a naming convention based on the following letters:

```
Q - quadrature routine

N - non-adaptive integrator
A - adaptive integrator

G - general integrand (user-defined)
W - weight function with integrand

S - singularities can be more readily integrated
P - points of special difficulty can be supplied
I - infinite range of integration
O - oscillatory weight function, cos or sin
F - Fourier integral
C - Cauchy principal value
```

The algorithms are built on pairs of quadrature rules, a higher order rule and a lower order rule. The higher order rule is used to compute the best approximation to an integral over a small range. The difference between the results of the higher order rule and the lower order rule gives an estimate of the error in the approximation.

17.1.1 Integrands without weight functions

The algorithms for general functions (without a weight function) are based on Gauss-Kronrod rules.

A Gauss-Kronrod rule begins with a classical Gaussian quadrature rule of order m. This is extended with additional points between each of the abscissae to give a higher order Kronrod rule of order $2m+1$. The Kronrod rule is efficient because it reuses existing function evaluations from the Gaussian rule.

The higher order Kronrod rule is used as the best approximation to the integral, and the difference between the two rules is used as an estimate of the error in the approximation.

17.1.2 Integrands with weight functions

For integrands with weight functions the algorithms use Clenshaw-Curtis quadrature rules.

A Clenshaw-Curtis rule begins with an n-th order Chebyshev polynomial approximation to the integrand. This polynomial can be integrated exactly to give an approximation to the integral of the original function. The Chebyshev expansion can be extended to higher orders to improve the approximation and provide an estimate of the error.

17.1.3 Integrands with singular weight functions

The presence of singularities (or other behavior) in the integrand can cause slow convergence in the Chebyshev approximation. The modified Clenshaw-Curtis rules used in QUADPACK separate out several common weight functions which cause slow convergence.

These weight functions are integrated analytically against the Chebyshev polynomials to precompute modified Chebyshev moments. Combining the moments with the Chebyshev approximation to the function gives the desired integral. The use of analytic integration for the singular part of the function allows exact cancellations and substantially improves the overall convergence behavior of the integration.

17.2 QNG non-adaptive Gauss-Kronrod integration

The QNG algorithm is a non-adaptive procedure which uses fixed Gauss-Kronrod-Patterson abscissae to sample the integrand at a maximum of 87 points. It is provided for fast integration of smooth functions.

int **gsl_integration_qng**(const gsl_function *f, double a, double b, double epsabs, double epsrel, double *result, double *abserr, size_t *neval)

This function applies the Gauss-Kronrod 10-point, 21-point, 43-point and 87-point integration rules in succession until an estimate of the integral of f over (a, b) is achieved within the desired absolute and relative error limits, `epsabs` and `epsrel`. The function returns the final approximation, `result`, an estimate of the absolute error, `abserr` and the number of function evaluations used, `neval`. The Gauss-Kronrod rules are designed in such a way that each rule uses all the results of its predecessors, in order to minimize the total number of function evaluations.

17.3 QAG adaptive integration

The QAG algorithm is a simple adaptive integration procedure. The integration region is divided into subintervals, and on each iteration the subinterval with the largest estimated error is bisected. This reduces the overall error rapidly, as the subintervals become concentrated around local difficulties in the integrand. These subintervals are managed by the following struct,

type **gsl_integration_workspace**

This workspace handles the memory for the subinterval ranges, results and error estimates.

gsl_integration_workspace *__gsl_integration_workspace_alloc__(size_t n)

This function allocates a workspace sufficient to hold `n` double precision intervals, their integration results and error estimates. One workspace may be used multiple times as all necessary reinitialization is performed automatically by the integration routines.

void **gsl_integration_workspace_free**(gsl_integration_workspace *w)

This function frees the memory associated with the workspace `w`.

int **gsl_integration_qag**(const gsl_function *f, double a, double b, double epsabs, double epsrel, size_t limit, int key, gsl_integration_workspace *workspace, double *result, double *abserr)

This function applies an integration rule adaptively until an estimate of the integral of f

over (a, b) is achieved within the desired absolute and relative error limits, `epsabs` and `epsrel`. The function returns the final approximation, `result`, and an estimate of the absolute error, `abserr`. The integration rule is determined by the value of `key`, which should be chosen from the following symbolic names,

Symbolic Name	Key
GSL_INTEG_GAUSS15	1
GSL_INTEG_GAUSS21	2
GSL_INTEG_GAUSS31	3
GSL_INTEG_GAUSS41	4
GSL_INTEG_GAUSS51	5
GSL_INTEG_GAUSS61	6

corresponding to the 15, 21, 31, 41, 51 and 61 point Gauss-Kronrod rules. The higher-order rules give better accuracy for smooth functions, while lower-order rules save time when the function contains local difficulties, such as discontinuities.

On each iteration the adaptive integration strategy bisects the interval with the largest error estimate. The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace.

17.4 QAGS adaptive integration with singularities

The presence of an integrable singularity in the integration region causes an adaptive routine to concentrate new subintervals around the singularity. As the subintervals decrease in size the successive approximations to the integral converge in a limiting fashion. This approach to the limit can be accelerated using an extrapolation procedure. The QAGS algorithm combines adaptive bisection with the Wynn epsilon-algorithm to speed up the integration of many types of integrable singularities.

int **gsl_integration_qags**(const gsl_function *f, double a, double b, double epsabs, double epsrel, size_t limit, gsl_integration_workspace *workspace, double *result, double *abserr)

This function applies the Gauss-Kronrod 21-point integration rule adaptively until an estimate of the integral of f over (a, b) is achieved within the desired absolute and relative error limits, `epsabs` and `epsrel`. The results are extrapolated using the epsilon-algorithm, which accelerates the convergence of the integral in the presence of discontinuities and integrable singularities. The function returns the final approximation from the

extrapolation, `result`, and an estimate of the absolute error, `abserr`. The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace.

17.5 QAGP adaptive integration with known singular points

int **gsl_integration_qagp**(const gsl_function *f, double *pts, size_t npts, double epsabs, double epsrel, size_t limit, gsl_integration_workspace *workspace, double *result, double *abserr)

This function applies the adaptive integration algorithm QAGS taking account of the user-supplied locations of singular points. The array `pts` of length `npts` should contain the endpoints of the integration ranges defined by the integration region and locations of the singularities. For example, to integrate over the region (a,b) with break-points at x_1, x_2, x_3 (where $a < x_1 < x_2 < x_3 < b$) the following `pts` array should be used:

```
pts[0] = a
pts[1] = x_1
pts[2] = x_2
pts[3] = x_3
pts[4] = b
```

with `npts` = 5.

If you know the locations of the singular points in the integration region then this routine will be faster than `gsl_integration_qags()`.

17.6 QAGI adaptive integration on infinite intervals

int **gsl_integration_qagi**(gsl_function *f, double epsabs, double epsrel, size_t limit, gsl_integration_workspace *workspace, double *result, double *abserr)

This function computes the integral of the function `f` over the infinite interval $(-\infty, +\infty)$. The integral is mapped onto the semi-open interval $(0,1]$ using the transformation $x = (1-t)/t$,

$$\int_{-\infty}^{+\infty} dx f(x) = \int_0^1 dt (f((1-t)/t) + f(-(1-t)/t))/t^2.$$

It is then integrated using the QAGS algorithm. The normal 21-point Gauss-Kronrod rule

of QAGS is replaced by a 15-point rule, because the transformation can generate an integrable singularity at the origin. In this case a lower-order rule is more efficient.

int **gsl_integration_qagiu**(gsl_function *f, double a, double epsabs, double epsrel, size_t limit, gsl_integration_workspace *workspace, double *result, double *abserr)

This function computes the integral of the function `f` over the semi-infinite interval $(a, +\infty)$. The integral is mapped onto the semi-open interval $(0, 1]$ using the transformation $x = a + (1-t)/t$,

$$\int_a^{+\infty} dx f(x) = \int_0^1 dt f(a + (1-t)/t)/t^2$$

and then integrated using the QAGS algorithm.

int **gsl_integration_qagil**(gsl_function *f, double b, double epsabs, double epsrel, size_t limit, gsl_integration_workspace *workspace, double *result, double *abserr)

This function computes the integral of the function `f` over the semi-infinite interval $(-\infty, b)$. The integral is mapped onto the semi-open interval $(0, 1]$ using the transformation $x = b - (1-t)/t$,

$$\int_{-\infty}^b dx f(x) = \int_0^1 dt f(b - (1-t)/t)/t^2$$

and then integrated using the QAGS algorithm.

17.7 QAWC adaptive integration for Cauchy principal values

int **gsl_integration_qawc**(gsl_function *f, double a, double b, double c, double epsabs, double epsrel, size_t limit, gsl_integration_workspace *workspace, double *result, double *abserr)

This function computes the Cauchy principal value of the integral of f over (a, b), with a singularity at `c`,

$$I = \int_a^b dx \frac{f(x)}{x-c} = \lim_{\epsilon \to 0} \left\{ \int_a^{c-\epsilon} dx \frac{f(x)}{x-c} + \int_{c+\epsilon}^b dx \frac{f(x)}{x-c} \right\}$$

The adaptive bisection algorithm of QAG is used, with modifications to ensure that subdivisions do not occur at the singular point $x = c$. When a subinterval contains the point $x = c$ or is close to it then a special 25-point modified Clenshaw-Curtis rule is used to control the singularity. Further away from the singularity the algorithm uses an ordinary

15-point Gauss-Kronrod integration rule.

17.8 QAWS adaptive integration for singular functions

The QAWS algorithm is designed for integrands with algebraic-logarithmic singularities at the end-points of an integration region. In order to work efficiently the algorithm requires a precomputed table of Chebyshev moments.

type **gsl_integration_qaws_table**

This structure contains precomputed quantities for the QAWS algorithm.

gsl_integration_qaws_table *__gsl_integration_qaws_table_alloc__(double alpha, double beta, int mu, int nu)

This function allocates space for a `gsl_integration_qaws_table` struct describing a singular weight function $w(x)$ with the parameters $(\alpha, \beta, \mu, \nu)$,

$$w(x) = (x-a)^\alpha (b-x)^\beta \log^\mu (x-a) \log^\nu (b-x)$$

where $\alpha > -1$, $\beta > -1$, and $\mu = 0, 1$, $\nu = 0, 1$. The weight function can take four different forms depending on the values of μ and ν,

Weight function $w(x)$	(μ, ν)
$(x-a)^\alpha (b-x)^\beta$	$(0, 0)$
$(x-a)^\alpha (b-x)^\beta \log(x-a)$	$(1, 0)$
$(x-a)^\alpha (b-x)^\beta \log(b-x)$	$(0, 1)$
$(x-a)^\alpha (b-x)^\beta \log(x-a) \log(b-x)$	$(1, 1)$

The singular points (a, b) do not have to be specified until the integral is computed, where they are the endpoints of the integration range.

The function returns a pointer to the newly allocated table `gsl_integration_qaws_table` if no errors were detected, and 0 in the case of error.

int __gsl_integration_qaws_table_set__(gsl_integration_qaws_table *t, double alpha, double beta, int mu, int nu)

This function modifies the parameters $(\alpha, \beta, \mu, \nu)$ of an existing `gsl_integration_qaws_table` struct `t`.

void __gsl_integration_qaws_table_free__(gsl_integration_qaws_table *t)

This function frees all the memory associated with the `gsl_integration_qaws_table` struct `t`.

int **gsl_integration_qaws**(gsl_function *f, const double a, const double b, gsl_integration_qaws_table *t, const double epsabs, const double epsrel, const size_t limit, gsl_integration_workspace *workspace, double *result, double *abserr)

This function computes the integral of the function $f(x)$ over the interval (a, b) with the singular weight function $(x-a)^\alpha (b-x)^\beta \log^\mu (x-a) \log^\nu (b-x)$. The parameters of the weight function $(\alpha, \beta, \mu, \nu)$ are taken from the table `t`. The integral is,

$$I = \int_a^b dx f(x) (x-a)^\alpha (b-x)^\beta \log^\mu (x-a) \log^\nu (b-x).$$

The adaptive bisection algorithm of QAG is used. When a subinterval contains one of the endpoints then a special 25-point modified Clenshaw-Curtis rule is used to control the singularities. For subintervals which do not include the endpoints an ordinary 15-point Gauss-Kronrod integration rule is used.

17.9 QAWO adaptive integration for oscillatory functions

The QAWO algorithm is designed for integrands with an oscillatory factor, $\sin(\omega x)$ or $\cos(\omega x)$. In order to work efficiently the algorithm requires a table of Chebyshev moments which must be pre-computed with calls to the functions below.

gsl_integration_qawo_table ***gsl_integration_qawo_table_alloc**(double omega, double L, enum gsl_integration_qawo_enum sine, size_t n)

This function allocates space for a `gsl_integration_qawo_table` struct and its associated workspace describing a sine or cosine weight function $w(x)$ with the parameters (ω, L),

$$w(x) = \left\{ \begin{array}{c} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

The parameter `L` must be the length of the interval over which the function will be integrated $L = b - a$. The choice of sine or cosine is made with the parameter `sine` which should be chosen from one of the two following symbolic values:

GSL_INTEG_COSINE

GSL_INTEG_SINE

The `gsl_integration_qawo_table` is a table of the trigonometric coefficients required in the integration process. The parameter `n` determines the number of levels of coefficients

that are computed. Each level corresponds to one bisection of the interval L, so that `n` levels are sufficient for subintervals down to the length $L/2^n$. The integration routine `gsl_integration_qawo()` returns the error `GSL_ETABLE` if the number of levels is insufficient for the requested accuracy.

int **gsl_integration_qawo_table_set**(gsl_integration_qawo_table *t, double omega, double L, enum gsl_integration_qawo_enum sine)

This function changes the parameters `omega`, `L` and `sine` of the existing workspace `t`.

int **gsl_integration_qawo_table_set_length**(gsl_integration_qawo_table *t, double L)

This function allows the length parameter `L` of the workspace `t` to be changed.

void **gsl_integration_qawo_table_free**(gsl_integration_qawo_table *t)

This function frees all the memory associated with the workspace `t`.

int **gsl_integration_qawo**(gsl_function *f, const double a, const double epsabs, const double epsrel, const size_t limit, gsl_integration_workspace *workspace, gsl_integration_qawo_table *wf, double *result, double *abserr)

This function uses an adaptive algorithm to compute the integral of f over (a, b) with the weight function $\sin(\omega x)$ or $\cos(\omega x)$ defined by the table `wf`,

$$I = \int_a^b dx f(x) \left\{ \begin{array}{c} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

The results are extrapolated using the epsilon-algorithm to accelerate the convergence of the integral. The function returns the final approximation from the extrapolation, `result`, and an estimate of the absolute error, `abserr`. The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace.

Those subintervals with "large" widths d where $d\omega > 4$ are computed using a 25-point Clenshaw-Curtis integration rule, which handles the oscillatory behavior. Subintervals with a "small" widths where $d\omega < 4$ are computed using a 15-point Gauss-Kronrod integration.

17.10 QAWF adaptive integration for Fourier integrals

int **gsl_integration_qawf**(gsl_function *f, const double a, const double epsabs, const size_t limit, gsl_integration_workspace *workspace, gsl_integration_workspace *cycle_workspace, gsl_integration_qawo_table *wf, double *result, double *abserr)

This function attempts to compute a Fourier integral of the function `f` over the semi-infinite interval $[a, +\infty)$

$$I = \int_a^{+\infty} dx f(x) \left\{ \begin{array}{c} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

The parameter ω and choice of sin or cos is taken from the table `wf` (the length `L` can take any value, since it is overridden by this function to a value appropriate for the Fourier integration). The integral is computed using the QAWO algorithm over each of the subintervals,

$$C_1 = [a, a + c]$$
$$C_2 = [a + c, a + 2c]$$
$$\ldots = \ldots$$
$$C_k = [a + (k - 1)c, a + kc]$$

where $c = (2 floor(|\omega|) + 1)\pi/|\omega|$. The width c is chosen to cover an odd number of periods so that the contributions from the intervals alternate in sign and are monotonically decreasing when `f` is positive and monotonically decreasing. The sum of this sequence of contributions is accelerated using the epsilon-algorithm.

This function works to an overall absolute tolerance of `abserr`. The following strategy is used: on each interval C_k the algorithm tries to achieve the tolerance

$$TOL_k = u_k abserr$$

where $u_k = (1 - p)p^{k-1}$ and $p = 9/10$. The sum of the geometric series of contributions from each interval gives an overall tolerance of `abserr`.

If the integration of a subinterval leads to difficulties then the accuracy requirement for subsequent intervals is relaxed,

$$TOL_k = u_k \max(abserr, \max_{i<k}(E_i))$$

where E_k is the estimated error on the interval C_k.

The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace. The integration over each subinterval uses the memory provided by `cycle_workspace` as workspace for the QAWO algorithm.

17.11 CQUAD doubly-adaptive integration

CQUAD is a new doubly-adaptive general-purpose quadrature routine which can handle most types of singularities, non-numerical function values such as `Inf` or `NaN`, as well as some divergent integrals. It generally requires more function evaluations than the integration routines in QUADPACK, yet fails less often for difficult integrands.

The underlying algorithm uses a doubly-adaptive scheme in which Clenshaw-Curtis quadrature rules of increasing degree are used to compute the integral in each interval. The L_2-norm of the difference between the underlying interpolatory polynomials of two successive rules is used as an error estimate. The interval is subdivided if the difference between two successive rules is too large or a rule of maximum degree has been reached.

gsl_integration_cquad_workspace *__gsl_integration_cquad_workspace_alloc__(size_t n)

This function allocates a workspace sufficient to hold the data for `n` intervals. The number `n` is not the maximum number of intervals that will be evaluated. If the workspace is full, intervals with smaller error estimates will be discarded. A minimum of 3 intervals is required and for most functions, a workspace of size 100 is sufficient.

void __gsl_integration_cquad_workspace_free__(gsl_integration_cquad_workspace *w)

This function frees the memory associated with the workspace `w`.

int __gsl_integration_cquad__(const gsl_function *f, double a, double b, double epsabs, double epsrel, gsl_integration_cquad_workspace *workspace, double *result, double *abserr, size_t *nevals)

This function computes the integral of f over (a, b) within the desired absolute and relative error limits, `epsabs` and `epsrel` using the CQUAD algorithm. The function returns the final approximation, `result`, an estimate of the absolute error, `abserr`, and the number of function evaluations required, `nevals`.

The CQUAD algorithm divides the integration region into subintervals, and in each iteration, the subinterval with the largest estimated error is processed. The algorithm uses Clenshaw-Curtis quadrature rules of degree 4, 8, 16 and 32 over 5, 9, 17 and 33 nodes respectively. Each interval is initialized with the lowest-degree rule. When an interval is processed, the next-higher degree rule is evaluated and an error estimate is computed

based on the L_2-norm of the difference between the underlying interpolating polynomials of both rules. If the highest-degree rule has already been used, or the interpolatory polynomials differ significantly, the interval is bisected.

The subintervals and their results are stored in the memory provided by `workspace`. If the error estimate or the number of function evaluations is not needed, the pointers `abserr` and `nevals` can be set to `NULL`.

17.12 Romberg integration

The Romberg integration method estimates the definite integral

$$I = \int_a^b f(x)dx$$

by applying Richardson extrapolation on the trapezoidal rule, using equally spaced points with spacing

$$h_k = (b-a)2^{-k}$$

for $k = 1, \ldots, n$. For each k, Richardson extrapolation is used $k-1$ times on previous approximations to improve the order of accuracy as much as possible. Romberg integration typically works well (and converges quickly) for smooth integrands with no singularities in the interval or at the end points.

gsl_integration_romberg_workspace ***gsl_integration_romberg_alloc**(const size_t n)

This function allocates a workspace for Romberg integration, specifying a maximum of n iterations, or divisions of the interval. Since the number of divisions is $2^n + 1$, n can be kept relatively small (i.e. 10 or 20). It is capped at a maximum value of 30 to prevent overflow. The size of the workspace is $O(2n)$.

void **gsl_integration_romberg_free**(gsl_integration_romberg_workspace *w)

This function frees the memory associated with the workspace `w`.

int **gsl_integration_romberg**(const gsl_function *f, const double a, const double b, const double epsabs, const double epsrel, double *result, size_t *neval, gsl_integration_romberg_workspace *w)

This function integrates $f(x)$, specified by `f`, from `a` to `b`, storing the answer in `result`. At each step in the iteration, convergence is tested by checking:

$$|I_k - I_{k-1}| \leq \max(epsabs, epsrel \times |I_k|)$$

where I_k is the current approximation and I_{k-1} is the approximation of the previous iteration. If the method does not converge within the previously specified n iterations, the function stores the best current estimate in `result` and returns `GSL_EMAXITER`. If the method converges, the function returns `GSL_SUCCESS`. The total number of function evaluations is returned in `neval`.

17.13 Gauss-Legendre integration

The fixed-order Gauss-Legendre integration routines are provided for fast integration of smooth functions with known polynomial order. The n-point Gauss-Legendre rule is exact for polynomials of order $2n-1$ or less. For example, these rules are useful when integrating basis functions to form mass matrices for the Galerkin method. Unlike other numerical integration routines within the library, these routines do not accept absolute or relative error bounds.

gsl_integration_glfixed_table *__gsl_integration_glfixed_table_alloc__(size_t n)

This function determines the Gauss-Legendre abscissae and weights necessary for an n-point fixed order integration scheme. If possible, high precision precomputed coefficients are used. If precomputed weights are not available, lower precision coefficients are computed on the fly.

double __gsl_integration_glfixed__(const gsl_function *f, double a, double b, const gsl_integration_glfixed_table *t)

This function applies the Gauss-Legendre integration rule contained in table `t` and returns the result.

int __gsl_integration_glfixed_point__(double a, double b, size_t i, double *xi, double *wi, const gsl_integration_glfixed_table *t)

For `i` in $[0, \ldots, n-1]$, this function obtains the `i`-th Gauss-Legendre point `xi` and weight `wi` on the interval [`a`, `b`]. The points and weights are ordered by increasing point value. A function f may be integrated on [`a`, `b`] by summing $wi * f(xi)$ over `i`.

void __gsl_integration_glfixed_table_free__(gsl_integration_glfixed_table *t)

This function frees the memory associated with the table `t`.

17.14 Fixed point quadratures

The routines in this section approximate an integral by the sum

$$\int_a^b w(x)f(x)dx = \sum_{i=1}^{n} w_i f(x_i)$$

where $f(x)$ is the function to be integrated and $w(x)$ is a weighting function. The n weights w_i and nodes x_i are carefully chosen so that the result is exact when $f(x)$ is a polynomial of degree $2n-1$ or less. Once the user chooses the order n and weighting function $w(x)$, the weights w_i and nodes x_i can be precomputed and used to efficiently evaluate integrals for any number of functions $f(x)$.

This method works best when $f(x)$ is well approximated by a polynomial on the interval (a,b), and so is not suitable for functions with singularities. Since the user specifies ahead of time how many quadrature nodes will be used, these routines do not accept absolute or relative error bounds. The table below lists the weighting functions currently supported.

Name	Interval	Weighting function $w(x)$	Constraints
Legendre	(a,b)	1	$b > a$
Chebyshev Type 1	(a,b)	$1/\sqrt{(b-x)(x-a)}$	$b > a$
Gegenbauer	(a,b)	$((b-x)(x-a))^\alpha$	$\alpha > -1, b > a$
Jacobi	(a,b)	$(b-x)^\alpha (x-a)^\beta$	$\alpha, \beta > -1, b > a$
Laguerre	(a,∞)	$(x-a)^\alpha \exp(-b(x-a))$	$\alpha > -1, b > 0$
Hermite	$(-\infty,\infty)$	$\lvert x-a\rvert^\alpha \exp(-b(x-a)^2)$	$\alpha > -1, b > 0$
Exponential	(a,b)	$\lvert x-(a+b)/2\rvert^\alpha$	$\alpha > -1, b > a$
Rational	(a,∞)	$(x-a)^\alpha (x+b)^\beta$	$\alpha > -1, \alpha+\beta+2n < 0, a+b > 0$
Chebyshev Type 2	(a,b)	$\sqrt{(b-x)(x-a)}$	$b > a$

The fixed point quadrature routines use the following workspace to store the nodes and weights, as well as additional variables for intermediate calculations:

type **gsl_integration_fixed_workspace**

This workspace is used for fixed point quadrature rules and looks like this:

```
typedef struct
{
  size_t n;        /* number of nodes/weights */
  double *weights; /* quadrature weights */
  double *x;       /* quadrature nodes */
  double *diag;    /* diagonal of Jacobi matrix */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  double *subdiag; /* subdiagonal of Jacobi matrix */
  const gsl_integration_fixed_type * type;
} gsl_integration_fixed_workspace;
```

gsl_integration_fixed_workspace ***gsl_integration_fixed_alloc**(const gsl_integration_fixed_type *T, const size_t n, const double a, const double b, const double alpha, const double beta)

This function allocates a workspace for computing integrals with interpolating quadratures using `n` quadrature nodes. The parameters `a`, `b`, `alpha`, and `beta` specify the integration interval and/or weighting function for the various quadrature types. See the table above for constraints on these parameters. The size of the workspace is $O(4n)$.

type **gsl_integration_fixed_type**

The type of quadrature used is specified by `T` which can be set to the following choices:

gsl_integration_fixed_type ***gsl_integration_fixed_legendre**

This specifies Legendre quadrature integration. The parameters `alpha` and `beta` are ignored for this type.

gsl_integration_fixed_type ***gsl_integration_fixed_chebyshev**

This specifies Chebyshev type 1 quadrature integration. The parameters `alpha` and `beta` are ignored for this type.

gsl_integration_fixed_type ***gsl_integration_fixed_gegenbauer**

This specifies Gegenbauer quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_type ***gsl_integration_fixed_jacobi**

This specifies Jacobi quadrature integration.

gsl_integration_fixed_type ***gsl_integration_fixed_laguerre**

This specifies Laguerre quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_type ***gsl_integration_fixed_hermite**

This specifies Hermite quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_type ***gsl_integration_fixed_exponential***

This specifies exponential quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_type ***gsl_integration_fixed_rational***

This specifies rational quadrature integration.

gsl_integration_fixed_type ***gsl_integration_fixed_chebyshev2***

This specifies Chebyshev type 2 quadrature integration. The parameters `alpha` and `beta` are ignored for this type.

void **gsl_integration_fixed_free**(gsl_integration_fixed_workspace *w)

This function frees the memory assocated with the workspace `w`

size_t **gsl_integration_fixed_n**(const gsl_integration_fixed_workspace *w)

This function returns the number of quadrature nodes and weights.

double ***gsl_integration_fixed_nodes**(const gsl_integration_fixed_workspace *w)

This function returns a pointer to an array of size `n` containing the quadrature nodes x_i.

double ***gsl_integration_fixed_weights**(const gsl_integration_fixed_workspace *w)

This function returns a pointer to an array of size `n` containing the quadrature weights w_i.

int **gsl_integration_fixed**(const gsl_function *func, double *result, const gsl_integration_fixed_workspace *w)

This function integrates the function $f(x)$ provided in `func` using previously computed fixed quadrature rules. The integral is approximated as

$$\sum_{i=1}^{n} w_i f(x_i)$$

where w_i are the quadrature weights and x_i are the quadrature nodes computed previously by `gsl_integration_fixed_alloc()`. The sum is stored in `result` on output.

17.15 Error codes

In addition to the standard error codes for invalid arguments the functions can return the following values,

GSL_EMAXITER	the maximum number of subdivisions was exceeded.
GSL_EROUND	cannot reach tolerance because of roundoff error, or roundoff error was detected in the extrapolation table.
GSL_ESING	a non-integrable singularity or other bad integrand behavior was found in the integration interval.
GSL_EDIVERGE	the integral is divergent, or too slowly convergent to be integrated numerically.
GSL_EDOM	error in the values of the input arguments

17.16 Examples

17.16.1 Adaptive integration example

The integrator QAGS will handle a large class of definite integrals. For example, consider the following integral, which has an algebraic-logarithmic singularity at the origin,

$$\int_0^1 x^{-1/2} \log(x) dx = -4$$

The program below computes this integral to a relative accuracy bound of 1e-7.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void * params) {
  double alpha = *(double *) params;
  double f = log(alpha*x) / sqrt(x);
  return f;
}

int
main (void)
{
  gsl_integration_workspace * w
    = gsl_integration_workspace_alloc (1000);

  double result, error;
  double expected = -4.0;
  double alpha = 1.0;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_function F;
  F.function = &f;
  F.params = &alpha;

  gsl_integration_qags (&F, 0, 1, 0, 1e-7, 1000,
                        w, &result, &error);

  printf ("result          = % .18f\n", result);
  printf ("exact result    = % .18f\n", expected);
  printf ("estimated error = % .18f\n", error);
  printf ("actual error    = % .18f\n", result - expected);
  printf ("intervals       = %zu\n", w->size);

  gsl_integration_workspace_free (w);

  return 0;
}
```

The results below show that the desired accuracy is achieved after 8 subdivisions.

```
result          = -4.000000000000085265
exact result    = -4.000000000000000000
estimated error =  0.000000000000135447
actual error    = -0.000000000000085265
intervals       = 8
```

In fact, the extrapolation procedure used by `QAGS` produces an accuracy of almost twice as many digits. The error estimate returned by the extrapolation procedure is larger than the actual error, giving a margin of safety of one order of magnitude.

17.16.2 Fixed-point quadrature example

In this example, we use a fixed-point quadrature rule to integrate the integral

$$\int_{-\infty}^{\infty} e^{-x^2} (x^m + 1)\, dx = \begin{cases} \sqrt{\pi} + \Gamma\left(\frac{m+1}{2}\right), & m \text{even} \\ \sqrt{\pi}, & m \text{odd} \end{cases}$$

for integer m. Consulting our table of fixed point quadratures, we see that this integral can be evaluated with a Hermite quadrature rule, setting $\alpha = 0, a = 0, b = 1$. Since we are integrating a polynomial of degree m, we need to choose the number of nodes $n \geq (m+1)/2$ to achieve

the best results.

First we will try integrating for $m = 10, n = 5$, which does not satisfy our criteria above:

```
$ ./integration2 10 5
```

The output is,

```
m               = 10
intervals       = 5
result          =  47.468529694563351029
exact result    =  54.115231635459025483
actual error    = -6.646701940895674454
```

So, we find a large error. Now we try integrating for $m = 10, n = 6$ which does satisfy the criteria above:

```
$ ./integration2 10 6
```

The output is,

```
m               = 10
intervals       = 6
result          =  54.115231635459096537
exact result    =  54.115231635459025483
actual error    =  0.000000000000071054
```

The program is given below.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_sf_gamma.h>

double
f(double x, void * params)
{
  int m = *(int *) params;
  double f = gsl_pow_int(x, m) + 1.0;
  return f;
}

int
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
main (int argc, char *argv[])
{
  gsl_integration_fixed_workspace * w;
  const gsl_integration_fixed_type * T = gsl_integration_fixed_hermite;
  int m = 10;
  int n = 6;
  double expected, result;
  gsl_function F;

  if (argc > 1)
    m = atoi(argv[1]);

  if (argc > 2)
    n = atoi(argv[2]);

  w = gsl_integration_fixed_alloc(T, n, 0.0, 1.0, 0.0, 0.0);

  F.function = &f;
  F.params = &m;

  gsl_integration_fixed(&F, &result, w);

  if (m % 2 == 0)
    expected = M_SQRTPI + gsl_sf_gamma(0.5*(1.0 + m));
  else
    expected = M_SQRTPI;

  printf ("m               = %d\n", m);
  printf ("intervals       = %zu\n", gsl_integration_fixed_n(w));
  printf ("result          = % .18f\n", result);
  printf ("exact result    = % .18f\n", expected);
  printf ("actual error    = % .18f\n", result - expected);

  gsl_integration_fixed_free (w);

  return 0;
}
```

17.17 References and Further Reading

The following book is the definitive reference for QUADPACK, and was written by the original authors. It provides descriptions of the algorithms, program listings, test programs and examples. It also includes useful advice on numerical integration and many references to the numerical integration literature used in developing QUADPACK.

- R. Piessens, E. de Doncker-Kapenga, C.W. Ueberhuber, D.K. Kahaner. QUADPACK A subroutine package for automatic integration Springer Verlag, 1983.

The CQUAD integration algorithm is described in the following paper:

- P. Gonnet, “Increasing the Reliability of Adaptive Quadrature Using Explicit Interpolants”, ACM Transactions on Mathematical Software, Volume 37 (2010), Issue 3, Article 26.

The fixed-point quadrature routines are based on IQPACK, described in the following papers:

- S. Elhay, J. Kautsky, Algorithm 655: IQPACK, FORTRAN Subroutines for the Weights of Interpolatory Quadrature, ACM Transactions on Mathematical Software, Volume 13, Number 4, December 1987, pages 399-415.
- J. Kautsky, S. Elhay, Calculation of the Weights of Interpolatory Quadratures, Numerische Mathematik, Volume 40, Number 3, October 1982, pages 407-422.

제 18 장

난수 생성기

참고: 번역중

The library provides a large collection of random number generators which can be accessed through a uniform interface. Environment variables allow you to select different generators and seeds at runtime, so that you can easily switch between generators without needing to recompile your program. Each instance of a generator keeps track of its own state, allowing the generators to be used in multi-threaded programs. Additional functions are available for transforming uniform random numbers into samples from continuous or discrete probability distributions such as the Gaussian, log-normal or Poisson distributions.

These functions are declared in the header file `gsl_rng.h`.

18.1 General comments on random numbers

In 1988, Park and Miller wrote a paper entitled "Random number generators: good ones are hard to find." [Commun.: ACM, 31, 1192–1201]. Fortunately, some excellent random number generators are available, though poor ones are still in common use. You may be happy with the system-supplied random number generator on your computer, but you should be aware that as computers get faster, requirements on random number generators increase. Nowadays, a simulation that calls a random number generator millions of times can often finish before you can make it down the hall to the coffee machine and back.

A very nice review of random number generators was written by Pierre L'Ecuyer, as Chapter 4 of the book: Handbook on Simulation, Jerry Banks, ed. (Wiley, 1997). The chapter is available in postscript from L'Ecuyer's ftp site (see references). Knuth's volume on Seminumerical

Algorithms (originally published in 1968) devotes 170 pages to random number generators, and has recently been updated in its 3rd edition (1997). It is brilliant, a classic. If you don't own it, you should stop reading right now, run to the nearest bookstore, and buy it.

A good random number generator will satisfy both theoretical and statistical properties. Theoretical properties are often hard to obtain (they require real math!), but one prefers a random number generator with a long period, low serial correlation, and a tendency **not** to "fall mainly on the planes." Statistical tests are performed with numerical simulations. Generally, a random number generator is used to estimate some quantity for which the theory of probability provides an exact answer. Comparison to this exact answer provides a measure of "randomness".

18.2 The Random Number Generator Interface

It is important to remember that a random number generator is not a "real" function like sine or cosine. Unlike real functions, successive calls to a random number generator yield different return values. Of course that is just what you want for a random number generator, but to achieve this effect, the generator must keep track of some kind of "state" variable. Sometimes this state is just an integer (sometimes just the value of the previously generated random number), but often it is more complicated than that and may involve a whole array of numbers, possibly with some indices thrown in. To use the random number generators, you do not need to know the details of what comprises the state, and besides that varies from algorithm to algorithm.

type **gsl_rng_type**

type **gsl_rng**

The random number generator library uses two special structs, `gsl_rng_type` which holds static information about each type of generator and `gsl_rng` which describes an instance of a generator created from a given `gsl_rng_type`.

The functions described in this section are declared in the header file `gsl_rng.h`.

18.3 Random number generator initialization

gsl_rng *__gsl_rng_alloc__(const gsl_rng_type *T)

This function returns a pointer to a newly-created instance of a random number generator of type `T`. For example, the following code creates an instance of the Tausworthe generator:

```
gsl_rng * r = gsl_rng_alloc (gsl_rng_taus);
```

If there is insufficient memory to create the generator then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

The generator is automatically initialized with the default seed, `gsl_rng_default_seed`. This is zero by default but can be changed either directly or by using the environment variable `GSL_RNG_SEED`.

The details of the available generator types are described later in this chapter.

void **gsl_rng_set**(const gsl_rng *r, unsigned long int s)

This function initializes (or "seeds") the random number generator. If the generator is seeded with the same value of `s` on two different runs, the same stream of random numbers will be generated by successive calls to the routines below. If different values of $s \geq 1$ are supplied, then the generated streams of random numbers should be completely different. If the seed `s` is zero then the standard seed from the original implementation is used instead. For example, the original Fortran source code for the `ranlux` generator used a seed of 314159265, and so choosing `s` equal to zero reproduces this when using `gsl_rng_ranlux`.

When using multiple seeds with the same generator, choose seed values greater than zero to avoid collisions with the default setting.

Note that the most generators only accept 32-bit seeds, with higher values being reduced modulo 2^{32}. For generators with smaller ranges the maximum seed value will typically be lower.

void **gsl_rng_free**(gsl_rng *r)

This function frees all the memory associated with the generator `r`.

18.4 Sampling from a random number generator

The following functions return uniformly distributed random numbers, either as integers or double precision floating point numbers. `HAVE_INLINE` 이 정의된 경우, 각 함수들의 인라인 함수가 사용됩니다. To obtain non-uniform distributions, see 난수 분포.

unsigned long int **gsl_rng_get**(const gsl_rng *r)

This function returns a random integer from the generator `r`. The minimum and maximum values depend on the algorithm used, but all integers in the range [`min`, `max`] are equally likely. The values of `min` and `max` can be determined using the auxiliary functions `gsl_rng_max()` and `gsl_rng_min()`.

double **gsl_rng_uniform**(const gsl_rng *r)

This function returns a double precision floating point number uniformly distributed in the range [0,1). The range includes 0.0 but excludes 1.0. The value is typically obtained by dividing the result of `gsl_rng_get(r)` by `gsl_rng_max(r) + 1.0` in double precision. Some generators compute this ratio internally so that they can provide floating point numbers with more than 32 bits of randomness (the maximum number of bits that can be portably represented in a single `unsigned long int`).

double **gsl_rng_uniform_pos**(const gsl_rng *r)

This function returns a positive double precision floating point number uniformly distributed in the range (0,1), excluding both 0.0 and 1.0. The number is obtained by sampling the generator with the algorithm of `gsl_rng_uniform()` until a non-zero value is obtained. You can use this function if you need to avoid a singularity at 0.0.

unsigned long int **gsl_rng_uniform_int**(const gsl_rng *r, unsigned long int n)

This function returns a random integer from 0 to $n-1$ inclusive by scaling down and/or discarding samples from the generator `r`. All integers in the range $[0, n-1]$ are produced with equal probability. For generators with a non-zero minimum value an offset is applied so that zero is returned with the correct probability.

Note that this function is designed for sampling from ranges smaller than the range of the underlying generator. The parameter `n` must be less than or equal to the range of the generator `r`. If `n` is larger than the range of the generator then the function calls the error handler with an error code of `GSL_EINVAL` and returns zero.

In particular, this function is not intended for generating the full range of unsigned integer values $[0, 2^{32}-1]$. Instead choose a generator with the maximal integer range and zero minimum value, such as `gsl_rng_ranlxd1`, `gsl_rng_mt19937` or `gsl_rng_taus`, and sample it directly using `gsl_rng_get()`. The range of each generator can be found using the auxiliary functions described in the next section.

18.5 Auxiliary random number generator functions

The following functions provide information about an existing generator. You should use them in preference to hard-coding the generator parameters into your own code.

const char ***gsl_rng_name**(const gsl_rng *r)

This function returns a pointer to the name of the generator. For example:

```
printf ("r is a '%s' generator\n", gsl_rng_name (r));
```

would print something like:

```
r is a 'taus' generator
```

unsigned long int **gsl_rng_max**(const gsl_rng *r)

This function returns the largest value that `gsl_rng_get()` can return.

unsigned long int **gsl_rng_min**(const gsl_rng *r)

This function returns the smallest value that `gsl_rng_get()` can return. Usually this value is zero. There are some generators with algorithms that cannot return zero, and for these generators the minimum value is 1.

void ***gsl_rng_state**(const gsl_rng *r)

size_t **gsl_rng_size**(const gsl_rng *r)

These functions return a pointer to the state of generator `r` and its size. You can use this information to access the state directly. For example, the following code will write the state of a generator to a stream:

```
void * state = gsl_rng_state (r);
size_t n = gsl_rng_size (r);
fwrite (state, n, 1, stream);
```

const gsl_rng_type ****gsl_rng_types_setup**(void)

This function returns a pointer to an array of all the available generator types, terminated by a null pointer. The function should be called once at the start of the program, if needed. The following code fragment shows how to iterate over the array of generator types to print the names of the available algorithms:

```
const gsl_rng_type **t, **t0;

t0 = gsl_rng_types_setup ();

printf ("Available generators:\n");

for (t = t0; *t != 0; t++)
  {
    printf ("%s\n", (*t)->name);
  }
```

18.6 Random number environment variables

The library allows you to choose a default generator and seed from the environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED` and the function `gsl_rng_env_setup()`. This makes it easy try out different generators and seeds without having to recompile your program.

GSL_RNG_TYPE

This environment variable specifies the default random number generator. It should be the name of a generator, such as `taus` or `mt19937`.

GSL_RNG_SEED

This environment variable specifies the default seed for the random number generator

gsl_rng_type ***gsl_rng_default**

This global library variable specifies the default random number generator, and can be initialized from `GSL_RNG_TYPE` using `gsl_rng_env_setup()`. It is defined as follows:

```
extern const gsl_rng_type *gsl_rng_default
```

unsigned long int **gsl_rng_default_seed**

This global library variable specifies the seed for the default random number generator, and can be initialized from `GSL_RNG_SEED` using `gsl_rng_env_setup()`. It is set to zero by default and is defined as follows:

```
extern unsigned long int gsl_rng_default_seed
```

const gsl_rng_type ***gsl_rng_env_setup**(void)

This function reads the environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED` and uses their values to set the corresponding library variables `gsl_rng_default` and `gsl_rng_default_seed`.

The value of `GSL_RNG_SEED` is converted to an `unsigned long int` using the C library function `strtoul()`.

If you don't specify a generator for `GSL_RNG_TYPE` then `gsl_rng_mt19937` is used as the default. The initial value of `gsl_rng_default_seed` is zero.

Here is a short program which shows how to create a global generator using the environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED`,

```
#include <stdio.h>
#include <gsl/gsl_rng.h>

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
gsl_rng * r;  /* global generator */

int
main (void)
{
  const gsl_rng_type * T;

  gsl_rng_env_setup();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  printf ("generator type: %s\n", gsl_rng_name (r));
  printf ("seed = %lu\n", gsl_rng_default_seed);
  printf ("first value = %lu\n", gsl_rng_get (r));

  gsl_rng_free (r);
  return 0;
}
```

Running the program without any environment variables uses the initial defaults, an `mt19937` generator with a seed of 0,

```
generator type: mt19937
seed = 0
first value = 4293858116
```

By setting the two variables on the command line we can change the default generator and the seed:

```
$ GSL_RNG_TYPE="taus" GSL_RNG_SEED=123 ./a.out
GSL_RNG_TYPE=taus
GSL_RNG_SEED=123
generator type: taus
seed = 123
first value = 2720986350
```

18.7 Copying random number generator state

The above methods do not expose the random number state which changes from call to call. It is often useful to be able to save and restore the state. To permit these practices, a few somewhat more advanced functions are supplied. These include:

int **gsl_rng_memcpy**(gsl_rng *dest, const gsl_rng *src)

This function copies the random number generator `src` into the pre-existing generator `dest`, making `dest` into an exact copy of `src`. The two generators must be of the same type.

gsl_rng ***gsl_rng_clone**(const gsl_rng *r)

This function returns a pointer to a newly created generator which is an exact copy of the generator `r`.

18.8 Reading and writing random number generator state

The library provides functions for reading and writing the random number state to a file as binary data.

int **gsl_rng_fwrite**(FILE *stream, const gsl_rng *r)

This function writes the random number state of the random number generator `r` to the stream `stream` in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_rng_fread**(FILE *stream, gsl_rng *r)

This function reads the random number state into the random number generator `r` from the open stream `stream` in binary format. The random number generator `r` must be preinitialized with the correct random number generator type since type information is not saved. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

18.9 Random number generator algorithms

The functions described above make no reference to the actual algorithm used. This is deliberate so that you can switch algorithms without having to change any of your application source code. The library provides a large number of generators of different types, including simulation quality generators, generators provided for compatibility with other libraries and historical generators from the past.

The following generators are recommended for use in simulation. They have extremely long periods, low correlation and pass most statistical tests. For the most reliable source of uncorrelated numbers, the second-generation RANLUX generators have the strongest proof of randomness.

gsl_rng_type ***gsl_rng_mt19937**

The MT19937 generator of Makoto Matsumoto and Takuji Nishimura is a variant of the twisted generalized feedback shift-register algorithm, and is known as the "Mersenne Twister" generator. It has a Mersenne prime period of $2^{19937} - 1$ (about 10^{6000}) and is equi-distributed in 623 dimensions. It has passed the DIEHARD statistical tests. It uses 624 words of state per generator and is comparable in speed to the other generators. The original generator used a default seed of 4357 and choosing `s` equal to zero in `gsl_rng_set()` reproduces this. Later versions switched to 5489 as the default seed, you can choose this explicitly via `gsl_rng_set()` instead if you require it.

For more information see,

- Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". ACM Transactions on Modeling and Computer Simulation, Vol.: 8, No.: 1 (Jan. 1998), Pages 3–30

The generator `gsl_rng_mt19937` uses the second revision of the seeding procedure published by the two authors above in 2002. The original seeding procedures could cause spurious artifacts for some seed values. They are still available through the alternative generators `gsl_rng_mt19937_1999` and `gsl_rng_mt19937_1998`.

gsl_rng_type ***gsl_rng_ranlxs0**

gsl_rng_type ***gsl_rng_ranlxs1**

gsl_rng_type ***gsl_rng_ranlxs2**

The generator `ranlxs0` is a second-generation version of the RANLUX algorithm of Luscher, which produces "luxury random numbers". This generator provides single precision output (24 bits) at three luxury levels `ranlxs0`, `ranlxs1` and `ranlxs2`, in increasing order of strength. It uses double-precision floating point arithmetic internally and can be significantly faster than the integer version of `ranlux`, particularly on 64-bit architectures.

The period of the generator is about 10^{171}. The algorithm has mathematically proven properties and can provide truly decorrelated numbers at a known level of randomness. The higher luxury levels provide increased decorrelation between samples as an additional safety margin.

Note that the range of allowed seeds for this generator is $[0, 2^{31} - 1]$. Higher seed values are wrapped modulo 2^{31}.

gsl_rng_type ***gsl_rng_ranlxd1**

gsl_rng_type ***gsl_rng_ranlxd2**

These generators produce double precision output (48 bits) from the RANLXS generator. The library provides two luxury levels `ranlxd1` and `ranlxd2`, in increasing order of strength.

gsl_rng_type ***gsl_rng_ranlux**

gsl_rng_type ***gsl_rng_ranlux389**

The `ranlux` generator is an implementation of the original algorithm developed by Luscher. It uses a lagged-fibonacci-with-skipping algorithm to produce "luxury random numbers". It is a 24-bit generator, originally designed for single-precision IEEE floating point numbers. This implementation is based on integer arithmetic, while the second-generation versions RANLXS and RANLXD described above provide floating-point implementations which will be faster on many platforms. The period of the generator is about 10^{171}. The algorithm has mathematically proven properties and it can provide truly decorrelated numbers at a known level of randomness. The default level of decorrelation recommended by Luscher is provided by `gsl_rng_ranlux`, while `gsl_rng_ranlux389` gives the highest level of randomness, with all 24 bits decorrelated. Both types of generator use 24 words of state per generator.

For more information see,

- M. Luscher, "A portable high-quality random number generator for lattice field theory calculations", Computer Physics Communications, 79 (1994) 100–110.
- F. James, "RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Luscher", Computer Physics Communications, 79 (1994) 111–114

gsl_rng_type ***gsl_rng_cmrg**

This is a combined multiple recursive generator by L'Ecuyer. Its sequence is,

$$z_n = (x_n - y_n) \mod m_1$$

where the two underlying generators x_n and y_n are,

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \mod m_1$$
$$y_n = (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \mod m_2$$

with coefficients $a_1 = 0$, $a_2 = 63308$, $a_3 = -183326$, $b_1 = 86098$, $b_2 = 0$, $b_3 = -539608$, and moduli $m_1 = 2^{31} - 1 = 2147483647$ and $m_2 = 2145483479$.

The period of this generator is $\mathrm{lcm}(m_1^3 - 1, m_2^3 - 1)$, which is approximately 2^{185} (about 10^{56}). It uses 6 words of state per generator. For more information see,

- P. L'Ecuyer, "Combined Multiple Recursive Random Number Generators", Operations Research, 44, 5 (1996), 816–822.

gsl_rng_type ***gsl_rng_mrg**

This is a fifth-order multiple recursive generator by L'Ecuyer, Blouin and Coutre. Its sequence is,

$$x_n = (a_1 x_{n-1} + a_5 x_{n-5}) \mod m$$

with $a_1 = 107374182$, $a_2 = a_3 = a_4 = 0$, $a_5 = 104480$ and $m = 2^{31} - 1$.

The period of this generator is about 10^{46}. It uses 5 words of state per generator. More information can be found in the following paper,

- P. L'Ecuyer, F. Blouin, and R. Coutre, "A search for good multiple recursive random number generators", ACM Transactions on Modeling and Computer Simulation 3, 87–98 (1993).

gsl_rng_type ***gsl_rng_taus**

gsl_rng_type ***gsl_rng_taus2**

This is a maximally equidistributed combined Tausworthe generator by L'Ecuyer. The sequence is,

$$x_n = (s_n^1 \oplus s_n^2 \oplus s_n^3)$$

where,

$$s_{n+1}^1 = (((s_n^1 \& 4294967294) \ll 12) \oplus (((s_n^1 \ll 13) \oplus s_n^1) \gg 19))$$
$$s_{n+1}^2 = (((s_n^2 \& 4294967288) \ll 4) \oplus (((s_n^2 \ll 2) \oplus s_n^2) \gg 25))$$
$$s_{n+1}^3 = (((s_n^3 \& 4294967280) \ll 17) \oplus (((s_n^3 \ll 3) \oplus s_n^3) \gg 11))$$

computed modulo 2^{32}. In the formulas above $\oplus$ denotes exclusive-or. Note that the algorithm relies on the properties of 32-bit unsigned integers and has been implemented

using a bitmask of 0xFFFFFFFF to make it work on 64 bit machines.

The period of this generator is 2^{88} (about 10^{26}). It uses 3 words of state per generator. For more information see,

- P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators", Mathematics of Computation, 65, 213 (1996), 203–213.

The generator gsl_rng_taus2 uses the same algorithm as gsl_rng_taus but with an improved seeding procedure described in the paper,

- P. L'Ecuyer, "Tables of Maximally Equidistributed Combined LFSR Generators", Mathematics of Computation, 68, 225 (1999), 261–269

The generator gsl_rng_taus2 should now be used in preference to gsl_rng_taus.

gsl_rng_type ***gsl_rng_gfsr4**

The gfsr4 generator is like a lagged-fibonacci generator, and produces each number as an xor'd sum of four previous values.

$$r_n = r_{n-A} \oplus r_{n-B} \oplus r_{n-C} \oplus r_{n-D}$$

Ziff (ref below) notes that "it is now widely known" that two-tap registers (such as R250, which is described below) have serious flaws, the most obvious one being the three-point correlation that comes from the definition of the generator. Nice mathematical properties can be derived for GFSR's, and numerics bears out the claim that 4-tap GFSR's with appropriately chosen offsets are as random as can be measured, using the author's test.

This implementation uses the values suggested the example on p392 of Ziff's article: $A = 471$, $B = 1586$, $C = 6988$, $D = 9689$.

If the offsets are appropriately chosen (such as the one ones in this implementation), then the sequence is said to be maximal; that means that the period is $2^D - 1$, where D is the longest lag. (It is one less than 2^D because it is not permitted to have all zeros in the ra[] array.) For this implementation with $D = 9689$ that works out to about 10^{2917}.

Note that the implementation of this generator using a 32-bit integer amounts to 32 parallel implementations of one-bit generators. One consequence of this is that the period of this 32-bit generator is the same as for the one-bit generator. Moreover, this independence means that all 32-bit patterns are equally likely, and in particular that 0 is an allowed random value. (We are grateful to Heiko Bauke for clarifying for us these properties of GFSR random number generators.)

For more information see,

- Robert M. Ziff, "Four-tap shift-register-sequence random-number generators", Computers in Physics, 12(4), Jul/Aug 1998, pp 385–392.

18.10 Unix random number generators

The standard Unix random number generators `rand`, `random` and `rand48` are provided as part of GSL. Although these generators are widely available individually often they aren't all available on the same platform. This makes it difficult to write portable code using them and so we have included the complete set of Unix generators in GSL for convenience. Note that these generators don't produce high-quality randomness and aren't suitable for work requiring accurate statistics. However, if you won't be measuring statistical quantities and just want to introduce some variation into your program then these generators are quite acceptable.

gsl_rng_type ***gsl_rng_rand**

This is the BSD `rand` generator. Its sequence is

$$x_{n+1} = (a x_n + c) \mod m$$

with $a = 1103515245$, $c = 12345$ and $m = 2^{31}$. The seed specifies the initial value, x_1. The period of this generator is 2^{31}, and it uses 1 word of storage per generator.

gsl_rng_type ***gsl_rng_random_bsd**

gsl_rng_type ***gsl_rng_random_libc5**

gsl_rng_type ***gsl_rng_random_glibc2**

These generators implement the `random` family of functions, a set of linear feedback shift register generators originally used in BSD Unix. There are several versions of `random` in use today: the original BSD version (e.g. on SunOS4), a libc5 version (found on older GNU/Linux systems) and a glibc2 version. Each version uses a different seeding procedure, and thus produces different sequences.

The original BSD routines accepted a variable length buffer for the generator state, with longer buffers providing higher-quality randomness. The `random` function implemented algorithms for buffer lengths of 8, 32, 64, 128 and 256 bytes, and the algorithm with the largest length that would fit into the user-supplied buffer was used. To support these algorithms additional generators are available with the following names:

```
gsl_rng_random8_bsd
gsl_rng_random32_bsd
gsl_rng_random64_bsd
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
gsl_rng_random128_bsd
gsl_rng_random256_bsd
```

where the numeric suffix indicates the buffer length. The original BSD `random` function used a 128-byte default buffer and so `gsl_rng_random_bsd` has been made equivalent to `gsl_rng_random128_bsd`. Corresponding versions of the `libc5` and `glibc2` generators are also available, with the names `gsl_rng_random8_libc5`, `gsl_rng_random8_glibc2`, etc.

gsl_rng_type ***gsl_rng_rand48**

This is the Unix `rand48` generator. Its sequence is

$$x_{n+1} = (ax_n + c) \mod m$$

defined on 48-bit unsigned integers with $a = 25214903917$, $c = 11$ and $m = 2^{48}$. The seed specifies the upper 32 bits of the initial value, x_1, with the lower 16 bits set to `0x330E`. The function `gsl_rng_get()` returns the upper 32 bits from each term of the sequence. This does not have a direct parallel in the original `rand48` functions, but forcing the result to type `long int` reproduces the output of `mrand48`. The function `gsl_rng_uniform()` uses the full 48 bits of internal state to return the double precision number x_n/m, which is equivalent to the function `drand48`. Note that some versions of the GNU C Library contained a bug in `mrand48` function which caused it to produce different results (only the lower 16-bits of the return value were set).

18.11 Other random number generators

The generators in this section are provided for compatibility with existing libraries. If you are converting an existing program to use GSL then you can select these generators to check your new implementation against the original one, using the same random number generator. After verifying that your new program reproduces the original results you can then switch to a higher-quality generator.

Note that most of the generators in this section are based on single linear congruence relations, which are the least sophisticated type of generator. In particular, linear congruences have poor properties when used with a non-prime modulus, as several of these routines do (e.g. with a power of two modulus, 2^{31} or 2^{32}). This leads to periodicity in the least significant bits of each number, with only the higher bits having any randomness. Thus if you want to produce a random bitstream it is best to avoid using the least significant bits.

gsl_rng_type ***gsl_rng_ranf**

This is the CRAY random number generator RANF. Its sequence is

$$x_{n+1} = (a x_n) \mod m$$

defined on 48-bit unsigned integers with $a = 44485709377909$ and $m = 2^{48}$. The seed specifies the lower 32 bits of the initial value, x_1, with the lowest bit set to prevent the seed taking an even value. The upper 16 bits of x_1 are set to 0. A consequence of this procedure is that the pairs of seeds 2 and 3, 4 and 5, etc.: produce the same sequences.

The generator compatible with the CRAY MATHLIB routine RANF. It produces double precision floating point numbers which should be identical to those from the original RANF.

There is a subtlety in the implementation of the seeding. The initial state is reversed through one step, by multiplying by the modular inverse of a mod m. This is done for compatibility with the original CRAY implementation.

Note that you can only seed the generator with integers up to 2^{32}, while the original CRAY implementation uses non-portable wide integers which can cover all 2^{48} states of the generator.

The function gsl_rng_get() returns the upper 32 bits from each term of the sequence. The function gsl_rng_uniform() uses the full 48 bits to return the double precision number x_n/m.

The period of this generator is 2^{46}.

gsl_rng_type ***gsl_rng_ranmar**

This is the RANMAR lagged-fibonacci generator of Marsaglia, Zaman and Tsang. It is a 24-bit generator, originally designed for single-precision IEEE floating point numbers. It was included in the CERNLIB high-energy physics library.

gsl_rng_type ***gsl_rng_r250**

This is the shift-register generator of Kirkpatrick and Stoll. The sequence is based on the recurrence

$$x_n = x_{n-103} \oplus x_{n-250}$$

where $\oplus$ denotes exclusive-or, defined on 32-bit words. The period of this generator is about 2^{250} and it uses 250 words of state per generator.

For more information see,

- S. Kirkpatrick and E. Stoll, "A very fast shift-register sequence random number generator", Journal of Computational Physics, 40, 517–526 (1981)

gsl_rng_type ***gsl_rng_tt800**

This is an earlier version of the twisted generalized feedback shift-register generator, and has been superseded by the development of MT19937. However, it is still an acceptable generator in its own right. It has a period of 2^{800} and uses 33 words of storage per generator.

For more information see,

- Makoto Matsumoto and Yoshiharu Kurita, "Twisted GFSR Generators II", ACM Transactions on Modelling and Computer Simulation, Vol.: 4, No.: 3, 1994, pages 254–266.

gsl_rng_type ***gsl_rng_vax**

This is the VAX generator `MTH$RANDOM`. Its sequence is,

$$x_{n+1} = (a x_n + c) \mod m$$

with $a = 69069$, $c = 1$ and $m = 2^{32}$. The seed specifies the initial value, x_1. The period of this generator is 2^{32} and it uses 1 word of storage per generator.

gsl_rng_type ***gsl_rng_transputer**

This is the random number generator from the INMOS Transputer Development system. Its sequence is,

$$x_{n+1} = (a x_n) \mod m$$

with $a = 1664525$ and $m = 2^{32}$. The seed specifies the initial value, x_1.

gsl_rng_type ***gsl_rng_randu**

This is the IBM `RANDU` generator. Its sequence is

$$x_{n+1} = (a x_n) \mod m$$

with $a = 65539$ and $m = 2^{31}$. The seed specifies the initial value, x_1. The period of this generator was only 2^{29}. It has become a textbook example of a poor generator.

gsl_rng_type ***gsl_rng_minstd**

This is Park and Miller's "minimal standard" MINSTD generator, a simple linear congruence which takes care to avoid the major pitfalls of such algorithms. Its sequence is,

$$x_{n+1} = (a x_n) \mod m$$

with $a = 16807$ and $m = 2^{31} - 1 = 2147483647$. The seed specifies the initial value, x_1. The period of this generator is about 2^{31}.

This generator was used in the IMSL Library (subroutine RNUN) and in MATLAB (the RAND function) in the past. It is also sometimes known by the acronym "GGL" (I'm not sure what that stands for).

For more information see,

- Park and Miller, "Random Number Generators: Good ones are hard to find", Communications of the ACM, October 1988, Volume 31, No 10, pages 1192–1201.

gsl_rng_type ***gsl_rng_uni**

gsl_rng_type ***gsl_rng_uni32**

This is a reimplementation of the 16-bit SLATEC random number generator RUNIF. A generalization of the generator to 32 bits is provided by `gsl_rng_uni32`. The original source code is available from NETLIB.

gsl_rng_type ***gsl_rng_slatec**

This is the SLATEC random number generator RAND. It is ancient. The original source code is available from NETLIB.

gsl_rng_type ***gsl_rng_zuf**

This is the ZUFALL lagged Fibonacci series generator of Peterson. Its sequence is,

$$t = u_{n-273} + u_{n-607}$$
$$u_n = t - \text{floor}(t)$$

The original source code is available from NETLIB. For more information see,

- W. Petersen, "Lagged Fibonacci Random Number Generators for the NEC SX-3", International Journal of High Speed Computing (1994).

gsl_rng_type ***gsl_rng_knuthran2**

This is a second-order multiple recursive generator described by Knuth in Seminumerical Algorithms, 3rd Ed., page 108. Its sequence is,

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2}) \mod m$$

with $a_1 = 271828183$, $a_2 = 314159269$, and $m = 2^{31} - 1$.

gsl_rng_type ***gsl_rng_knuthran2002**

gsl_rng_type ***gsl_rng_knuthran**

This is a second-order multiple recursive generator described by Knuth in Seminumerical Algorithms, 3rd Ed., Section 3.6. Knuth provides its C code. The updated routine

gsl_rng_knuthran2002 is from the revised 9th printing and corrects some weaknesses in the earlier version, which is implemented as gsl_rng_knuthran.

gsl_rng_type ***gsl_rng_borosh13**

gsl_rng_type ***gsl_rng_fishman18**

gsl_rng_type ***gsl_rng_fishman20**

gsl_rng_type ***gsl_rng_lecuyer21**

gsl_rng_type ***gsl_rng_waterman14**

These multiplicative generators are taken from Knuth's Seminumerical Algorithms, 3rd Ed., pages 106–108. Their sequence is,

$$x_{n+1} = (a x_n) \mod m$$

where the seed specifies the initial value, x_1. The parameters a and m are as follows, Borosh-Niederreiter: $a = 1812433253$, $m = 2^{32}$, Fishman18: $a = 62089911$, $m = 2^{31} - 1$, Fishman20: $a = 48271$, $m = 2^{31} - 1$, L'Ecuyer: $a = 40692$, $m = 2^{31} - 249$, Waterman: $a = 1566083941$, $m = 2^{32}$.

gsl_rng_type ***gsl_rng_fishman2x**

This is the L'Ecuyer–Fishman random number generator. It is taken from Knuth's Seminumerical Algorithms, 3rd Ed., page 108. Its sequence is,

$$z_{n+1} = (x_n - y_n) \mod m$$

with $m = 2^{31} - 1$. x_n and y_n are given by the fishman20 and lecuyer21 algorithms. The seed specifies the initial value, x_1.

gsl_rng_type ***gsl_rng_coveyou**

This is the Coveyou random number generator. It is taken from Knuth's Seminumerical Algorithms, 3rd Ed., Section 3.2.2. Its sequence is,

$$x_{n+1} = (x_n(x_n + 1)) \mod m$$

with $m = 2^{32}$. The seed specifies the initial value, x_1.

18.12 Performance

The following table shows the relative performance of a selection the available random number generators. The fastest simulation quality generators are `taus`, `gfsr4` and `mt19937`. The generators which offer the best mathematically-proven quality are those based on the RANLUX algorithm:

```
1754 k ints/sec,    870 k doubles/sec, taus
1613 k ints/sec,    855 k doubles/sec, gfsr4
1370 k ints/sec,    769 k doubles/sec, mt19937
 565 k ints/sec,    571 k doubles/sec, ranlxs0
 400 k ints/sec,    405 k doubles/sec, ranlxs1
 490 k ints/sec,    389 k doubles/sec, mrg
 407 k ints/sec,    297 k doubles/sec, ranlux
 243 k ints/sec,    254 k doubles/sec, ranlxd1
 251 k ints/sec,    253 k doubles/sec, ranlxs2
 238 k ints/sec,    215 k doubles/sec, cmrg
 247 k ints/sec,    198 k doubles/sec, ranlux389
 141 k ints/sec,    140 k doubles/sec, ranlxd2
```

18.13 Examples

The following program demonstrates the use of a random number generator to produce uniform random numbers in the range [0.0, 1.0),

```
#include <stdio.h>
#include <gsl/gsl_rng.h>

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;

  int i, n = 10;

  gsl_rng_env_setup();

  T = gsl_rng_default;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  r = gsl_rng_alloc (T);

  for (i = 0; i < n; i++)
    {
      double u = gsl_rng_uniform (r);
      printf ("%.5f\n", u);
    }

  gsl_rng_free (r);

  return 0;
}
```

Here is the output of the program,

```
0.99974
0.16291
0.28262
0.94720
0.23166
0.48497
0.95748
0.74431
0.54004
0.73995
```

The numbers depend on the seed used by the generator. The default seed can be changed with the `GSL_RNG_SEED` environment variable to produce a different stream of numbers. The generator itself can be changed using the environment variable `GSL_RNG_TYPE`. Here is the output of the program using a seed value of 123 and the multiple-recursive generator `mrg`:

```
$ GSL_RNG_SEED=123 GSL_RNG_TYPE=mrg ./a.out
```

```
0.33050
0.86631
0.32982
0.67620
0.53391
0.06457
0.16847
0.70229
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
0.04371
0.86374
```

18.14 References and Further Reading

The subject of random number generation and testing is reviewed extensively in Knuth's Seminumerical Algorithms.

- Donald E. Knuth, The Art of Computer Programming: Seminumerical Algorithms (Vol 2, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896842.

Further information is available in the review paper written by Pierre L'Ecuyer,

- P. L'Ecuyer, "Random Number Generation", Chapter 4 of the Handbook on Simulation, Jerry Banks Ed., Wiley, 1998, 93–137.
- http://www.iro.umontreal.ca/~lecuyer/papers.html in the file `handsim.ps`.

The source code for the DIEHARD random number generator tests is also available online,

- DIEHARD source code, G. Marsaglia, http://stat.fsu.edu/pub/diehard/

A comprehensive set of random number generator tests is available from NIST,

- NIST Special Publication 800-22, "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications".
- http://csrc.nist.gov/rng/

18.15 Acknowledgements

Thanks to Makoto Matsumoto, Takuji Nishimura and Yoshiharu Kurita for making the source code to their generators (MT19937, MM&TN; TT800, MM&YK) available under the GNU General Public License. Thanks to Martin Luscher for providing notes and source code for the RANLXS and RANLXD generators.

제 19 장

Quasi-연속 난수 배열

참고: 번역중

This chapter describes functions for generating quasi-random sequences in arbitrary dimensions. A quasi-random sequence progressively covers a d-dimensional space with a set of points that are uniformly distributed. Quasi-random sequences are also known as low-discrepancy sequences. The quasi-random sequence generators use an interface that is similar to the interface for random number generators, except that seeding is not required—each generator produces a single sequence.

The functions described in this section are declared in the header file `gsl_qrng.h`.

19.1 Quasi-random number generator initialization

type **`gsl_qrng`**

This is a workspace for computing quasi-random sequences.

gsl_qrng *__`gsl_qrng_alloc`__(const gsl_qrng_type *T, unsigned int d)

This function returns a pointer to a newly-created instance of a quasi-random sequence generator of type `T` and dimension `d`. If there is insufficient memory to create the generator then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

void __`gsl_qrng_free`__(gsl_qrng *q)

This function frees all the memory associated with the generator `q`.

void **gsl_qrng_init**(gsl_qrng *q)

This function reinitializes the generator `q` to its starting point. Note that quasi-random sequences do not use a seed and always produce the same set of values.

19.2 Sampling from a quasi-random number generator

int **gsl_qrng_get**(const gsl_qrng *q, double x[])

This function stores the next point from the sequence generator `q` in the array `x`. The space available for `x` must match the dimension of the generator. The point `x` will lie in the range $0 < x_i < 1$ for each x_i. `HAVE_INLINE` 이 정의된 경우, 인라인 함수가 사용됩니다.

19.3 Auxiliary quasi-random number generator functions

const char ***gsl_qrng_name**(const gsl_qrng *q)

This function returns a pointer to the name of the generator.

size_t **gsl_qrng_size**(const gsl_qrng *q)

void ***gsl_qrng_state**(const gsl_qrng *q)

These functions return a pointer to the state of generator `r` and its size. You can use this information to access the state directly. For example, the following code will write the state of a generator to a stream:

```
void * state = gsl_qrng_state (q);
size_t n = gsl_qrng_size (q);
fwrite (state, n, 1, stream);
```

19.4 Saving and restoring quasi-random number generator state

int **gsl_qrng_memcpy**(gsl_qrng *dest, const gsl_qrng *src)

This function copies the quasi-random sequence generator `src` into the pre-existing generator `dest`, making `dest` into an exact copy of `src`. The two generators must be of the same type.

gsl_qrng *gsl_qrng_clone(const gsl_qrng *q)

This function returns a pointer to a newly created generator which is an exact copy of the generator q.

19.5 Quasi-random number generator algorithms

The following quasi-random sequence algorithms are available,

type gsl_qrng_type

gsl_qrng_type *gsl_qrng_niederreiter_2

This generator uses the algorithm described in Bratley, Fox, Niederreiter, ACM Trans. Model. Comp. Sim. 2, 195 (1992). It is valid up to 12 dimensions.

gsl_qrng_type *gsl_qrng_sobol

This generator uses the Sobol sequence described in Antonov, Saleev, USSR Comput. Maths. Math. Phys. 19, 252 (1980). It is valid up to 40 dimensions.

gsl_qrng_type *gsl_qrng_halton

gsl_qrng_type *gsl_qrng_reversehalton

These generators use the Halton and reverse Halton sequences described in J.H. Halton, Numerische Mathematik, 2, 84-90 (1960) and B. Vandewoestyne and R. Cools Computational and Applied Mathematics, 189, 1&2, 341-361 (2006). They are valid up to 1229 dimensions.

19.6 Examples

The following program prints the first 1024 points of the 2-dimensional Sobol sequence.

```
#include <stdio.h>
#include <gsl/gsl_qrng.h>

int
main (void)
{
  int i;
  gsl_qrng * q = gsl_qrng_alloc (gsl_qrng_sobol, 2);

  for (i = 0; i < 1024; i++)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    {
      double v[2];
      gsl_qrng_get (q, v);
      printf ("%.5f %.5f\n", v[0], v[1]);
    }

  gsl_qrng_free (q);
  return 0;
}
```

Here is the output from the program:

```
$ ./a.out
0.50000 0.50000
0.75000 0.25000
0.25000 0.75000
0.37500 0.37500
0.87500 0.87500
0.62500 0.12500
0.12500 0.62500
....
```

It can be seen that successive points progressively fill-in the spaces between previous points.

그림 19.1 shows the distribution in the x-y plane of the first 1024 points from the Sobol sequence,

19.7 References

The implementations of the quasi-random sequence routines are based on the algorithms described in the following paper,

- P. Bratley and B.L. Fox and H. Niederreiter, “Algorithm 738: Programs to Generate Niederreiter’s Low-discrepancy Sequences”, ACM Transactions on Mathematical Software, Vol.: 20, No.: 4, December, 1994, p.: 494–495.

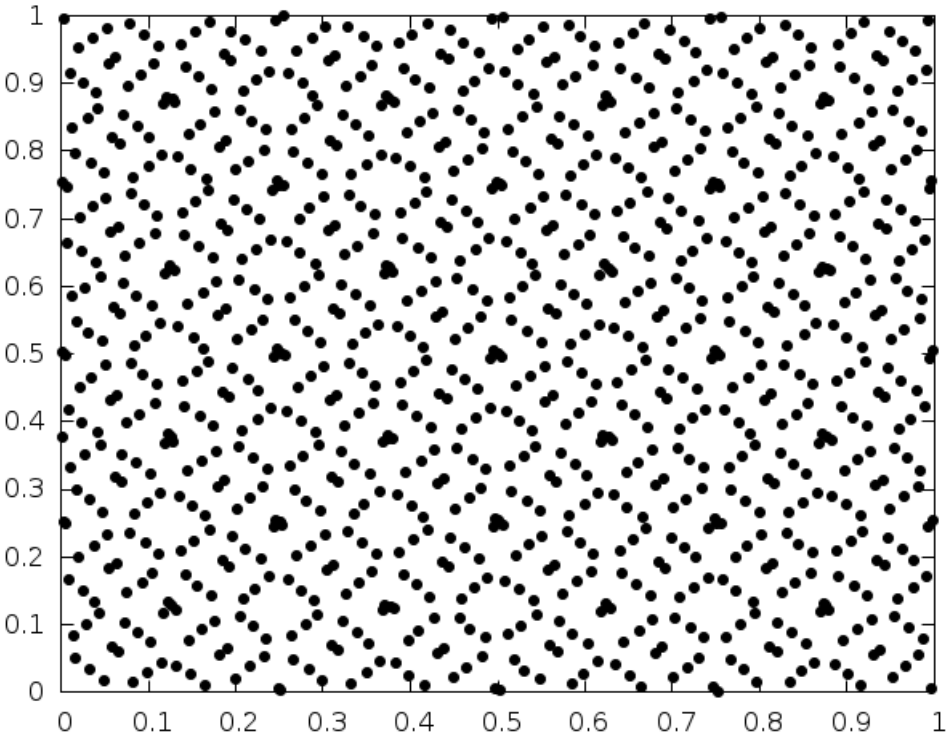


그림 19.1: Distribution of the first 1024 points from the quasi-random Sobol sequence

제 20 장

난수 분포

참고: 번역중

이 단원에서는 난수 분포를 생성하고 이들의 확률 분포를 계산하는 함수들에 대해 기술합니다. 이 단원에서 기술된 분포들의 특정 샘플은 라이브러리의 난수 발생기를 실행해 얻을 수 있습니다.

This chapter describes functions for generating random variates and computing their probability distributions. Samples from the distributions described in this chapter can be obtained using any of the random number generators in the library as an underlying source of randomness.

In the simplest cases a non-uniform distribution can be obtained analytically from the uniform distribution of a random number generator by applying an appropriate transformation. This method uses one call to the random number generator. More complicated distributions are created by the acceptance-rejection method, which compares the desired distribution against a distribution which is similar and known analytically. This usually requires several samples from the generator.

The library also provides cumulative distribution functions and inverse cumulative distribution functions, sometimes referred to as quantile functions. The cumulative distribution functions and their inverses are computed separately for the upper and lower tails of the distribution, allowing full accuracy to be retained for small results.

The functions for random variates and probability density functions described in this section are declared in `gsl_randist.h`. The corresponding cumulative distribution functions are declared in `gsl_cdf.h`.

Note that the discrete random variate functions always return a value of type `unsigned int`,

and on most platforms this has a maximum value of

$$2^{32} - 1 \approx 4.29 \times 10^9$$

They should only be called with a safe range of parameters (where there is a negligible probability of a variate exceeding this limit) to prevent incorrect results due to overflow.

20.1 개요

연속 난수 분포는 확률 밀도 함수로 정의됩니다. 확률 밀도 함수는 $p(x)$ 로 표기되며 x 가 x 에서 $x + dx$ 까지의 극소 범위에서 발생할 확률은 $p(x)dx$ 로 표현할 수 있습니다.

Continuous random number distributions are defined by a probability density function, $p(x)$, such that the probability of x occurring in the infinitesimal range x to $x + dx$ is $p(x)dx$.

누적 분포 함수 $P(x)$ 는 다음의 적분으로 정의되고

$$P(x) = \int_{-\infty}^{x} dx' p(x')$$

확률 변수가 x 보다 작을 확률을 나타냅니다.

보완(complementary) 누적 분포 함수 $Q(x)$ 는 다음 적분으로 정의되며

$$Q(x) = \int_{x}^{+\infty} dx' p(x')$$

확률 변수가 x 보다 클 확률을 나타냅니다..

이 두 분포 함수는 $P(x) + Q(x) = 1$ 의 관계를 가지고 $0 \leq P(x) \leq 1$, $0 \leq Q(x) \leq 1$ 를 만족합니다.

역 누적 분포 $x = P^{-1}(P)$ 와 $x = Q^{-1}(Q)$ 는 특정 P 와 Q 값에 대한 해당 x 값을 나타냅니다. 이는 확률 변수들로부터 적절한 범위를 계산하는 데 사용될 수 있습니다.

이상 확률 분포에서 정수값 k 를 뽑아내는 확률은 $p(k)$ 로 주어지고 $p(k)$sum_k p(k) = 1` 의 성질을 가집니다. 이산 누적 분포 $P(k)$ 는 다음과 같이 정의됩니다.

$$P(k) = \sum_{i \leq k} p(i)$$

이 정의의 분포 값 급수는 k 와 같거나 작을 확률을 나타냅니다.

똑같이 보완 누적 분포의 이산 변수 형태 $Q(k)$ 는 다음과 같이 정의되며

$$Q(k) = \sum_{i > k} p(i)$$

k 보다 큰 샘플들의 확률 합을 나타냅니다. 이 두 정의는 $P(k) + Q(k) = 1$ 관계를 만족합니다. 예를 들어

이산 분포의 범주가 1 에서 n 로 주어졌다면, $P(n) = 1$, $Q(n) = 0$ 이고 $P(1) = p(1)$, $Q(1) = 1 - p(1)$ 가 됩니다.

20.2 가우스 분포(Gaussian distribution)

double **gsl_ran_gaussian**(const gsl_rng *r, double sigma)

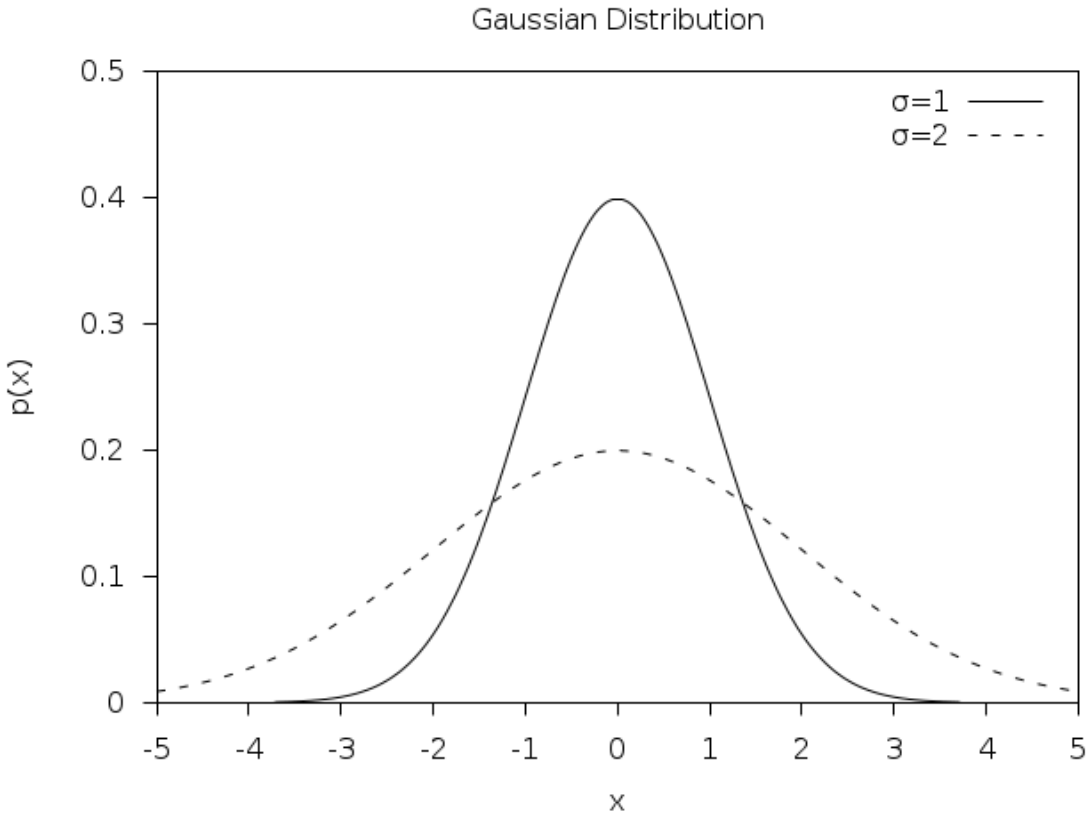
평균이 0 이고 표준 편차가 sigma 인 가우스 난수를 반환합니다. 가우스 난수의 확률 분포는 가우스 분포라 불리며 $x \in (-\infty, \infty)$ 에 대해 다음과 같이 정의됩니다.

$$p(x)dx = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$$

$z = \mu + x$ 변환을 이용해 평균이 μ 인 가우스 분포를 얻을 수도 있습니다. 이 함수는 Box-Muller 알고리즘을 사용합니다. 이 알고리즘은 난수 생성자 r 을 두 번에 걸쳐 호출합니다.

double **gsl_ran_gaussian_pdf**(double x, double sigma)

표준편차가 sigma 가우스 분포에 대해 x 의 확률 밀도 함수 $p(x)$ 를 계산합니다. 위에 기술된 수식을 사용합니다.



double **gsl_ran_gaussian_ziggurat**(const gsl_rng *r, double sigma)

double **gsl_ran_gaussian_ratio_method**(const gsl_rng *r, double sigma)

가우스 난수를 Marsaglia-Tsang 지구라트(ziggurat) 알고리즘과 Kinderman-Monahan-Leva 비율 방법을 사용해 계싼합니다. 지구라트 알고리즘은 대부분의 경우 가장 빠른 속도를 보여줍니다.

double **gsl_ran_ugaussian**(const gsl_rng *r)

double **gsl_ran_ugaussian_pdf**(double x)

double **gsl_ran_ugaussian_ratio_method**(const gsl_rng *r)

단위-가우스 분포(unit Gaussian distribution)를 계산합니다. 이들은 위의 함수들이 sigma = 1 인 표준 편차와 같습니다.

double **gsl_cdf_gaussian_P**(double x, double sigma)

double **gsl_cdf_gaussian_Q**(double x, double sigma)

double **gsl_cdf_gaussian_Pinv**(double P, double sigma)

double **gsl_cdf_gaussian_Qinv**(double Q, double sigma)

표준 편차가 sigma 인 가우스 분포에 대해 누적 분포 함수 $P(x)$, $Q(x)$ 와 이들의 역함수를 계산합니다.

double **gsl_cdf_ugaussian_P**(double x)

double **gsl_cdf_ugaussian_Q**(double x)

double **gsl_cdf_ugaussian_Pinv**(double P)

double **gsl_cdf_ugaussian_Qinv**(double Q)

단위-가우스 분포에 대해 누적 분포 함수 $P(x)$, $Q(x)$ 와 이들의 역함수를 계산합니다.

20.3 The Gaussian Tail Distribution

double **gsl_ran_gaussian_tail**(const gsl_rng *r, double a, double sigma)

This function provides random variates from the upper tail of a Gaussian distribution with standard deviation `sigma`. The values returned are larger than the lower limit `a`, which must be positive. The method is based on Marsaglia's famous rectangle-wedge-tail algorithm (Ann. Math. Stat. 32, 894–899 (1961)), with this aspect explained in Knuth, v2, 3rd ed, p139,586 (exercise 11).

The probability distribution for Gaussian tail random variates is,

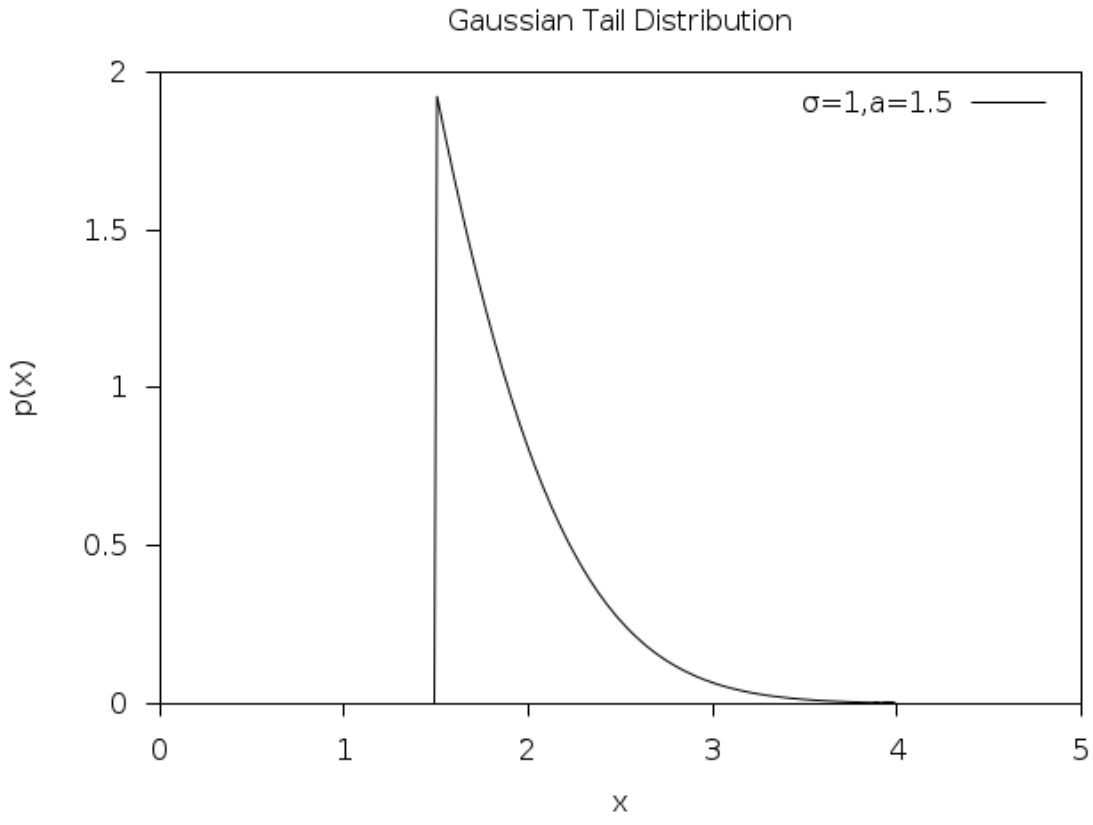
$$p(x)dx = \frac{1}{N(a;\sigma)\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$$

for $x > a$ where $N(a;\sigma)$ is the normalization constant,

$$N(a;\sigma) = \frac{1}{2}\mathrm{erfc}\left(\frac{a}{\sqrt{2\sigma^2}}\right).$$

double **gsl_ran_gaussian_tail_pdf**(double x, double a, double sigma)

This function computes the probability density $p(x)$ at `x` for a Gaussian tail distribution with standard deviation `sigma` and lower limit `a`, using the formula given above.



double **gsl_ran_ugaussian_tail**(const gsl_rng *r, double a)

double **gsl_ran_ugaussian_tail_pdf**(double x, double a)

These functions compute results for the tail of a unit Gaussian distribution. They are equivalent to the functions above with a standard deviation of one, `sigma` = 1.

20.4 The Bivariate Gaussian Distribution

void **gsl_ran_bivariate_gaussian**(const gsl_rng *r, double sigma_x, double sigma_y, double rho, double *x, double *y)

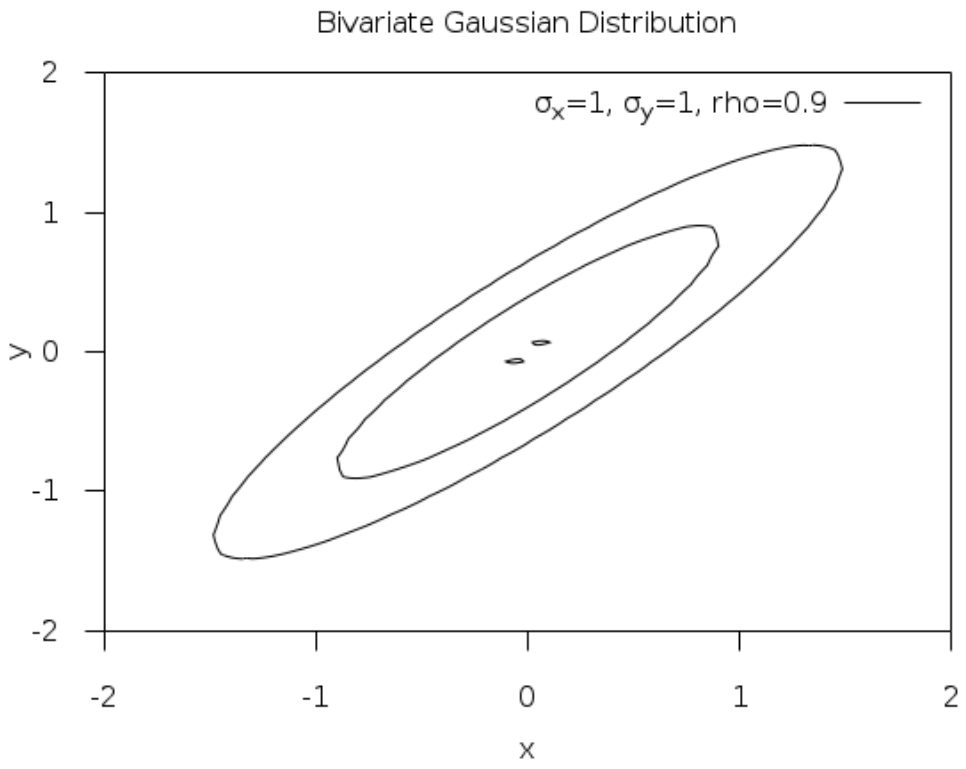
This function generates a pair of correlated Gaussian variates, with mean zero, correlation coefficient `rho` and standard deviations `sigma_x` and `sigma_y` in the x and y directions. The probability distribution for bivariate Gaussian random variates is,

$$p(x,y)dxdy = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left(-\frac{(x^2/\sigma_x^2 + y^2/\sigma_y^2 - 2\rho xy/(\sigma_x\sigma_y))}{2(1-\rho^2)}\right) dxdy$$

for x, y in the range $-\infty$ to $+\infty$. The correlation coefficient `rho` should lie between 1 and -1.

double **gsl_ran_bivariate_gaussian_pdf**(double x, double y, double sigma_x, double sigma_y, double rho)

This function computes the probability density $p(x,y)$ at (`x`, `y`) for a bivariate Gaussian distribution with standard deviations `sigma_x`, `sigma_y` and correlation coefficient `rho`, using the formula given above.



20.5 The Multivariate Gaussian Distribution

int **gsl_ran_multivariate_gaussian**(const gsl_rng *r, const gsl_vector *mu, const gsl_matrix *L, gsl_vector *result)

This function generates a random vector satisfying the k-dimensional multivariate Gaussian distribution with mean μ and variance-covariance matrix Σ. On input, the k-vector μ is given in `mu`, and the Cholesky factor of the k-by-k matrix $\Sigma = LL^T$ is given in the lower triangle of `L`, as output from `gsl_linalg_cholesky_decomp()`. The random vector is stored in `result` on output. The probability distribution for multivariate Gaussian random variates is

$$p(x_1, \ldots, x_k) dx_1 \ldots dx_k = \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right) dx_1 \ldots dx_k$$

int **gsl_ran_multivariate_gaussian_pdf**(const gsl_vector *x, const gsl_vector *mu, const gsl_matrix *L, double *result, gsl_vector *work)

int **gsl_ran_multivariate_gaussian_log_pdf**(const gsl_vector *x, const gsl_vector *mu, const gsl_matrix *L, double *result, gsl_vector *work)

These functions compute $p(x)$ or $\log p(x)$ at the point `x`, using mean vector `mu` and variance-covariance matrix specified by its Cholesky factor `L` using the formula above. Additional workspace of length k is required in `work`.

int **gsl_ran_multivariate_gaussian_mean**(const gsl_matrix *X, gsl_vector *mu_hat)

Given a set of n samples X_j from a k-dimensional multivariate Gaussian distribution, this function computes the maximum likelihood estimate of the mean of the distribution, given by

$$\hat{\mu} = \frac{1}{n} \sum_{j=1}^{n} X_j$$

The samples $X_1, X_2, \ldots, X_n$ are given in the n-by-k matrix `X`, and the maximum likelihood estimate of the mean is stored in `mu_hat` on output.

int **gsl_ran_multivariate_gaussian_vcov**(const gsl_matrix *X, gsl_matrix *sigma_hat)

Given a set of n samples X_j from a k-dimensional multivariate Gaussian distribution, this function computes the maximum likelihood estimate of the variance-covariance matrix of the distribution, given by

$$\hat{\Sigma} = \frac{1}{n} \sum_{j=1}^{n} (X_j - \hat{\mu})(X_j - \hat{\mu})^T$$

The samples $X_1, X_2, \ldots, X_n$ are given in the n-by-k matrix `X` and the maximum likelihood estimate of the variance-covariance matrix is stored in `sigma_hat` on output.

20.6 지수 분포(Exponential distribution)

double **gsl_ran_exponential**(const gsl_rng *r, double mu)

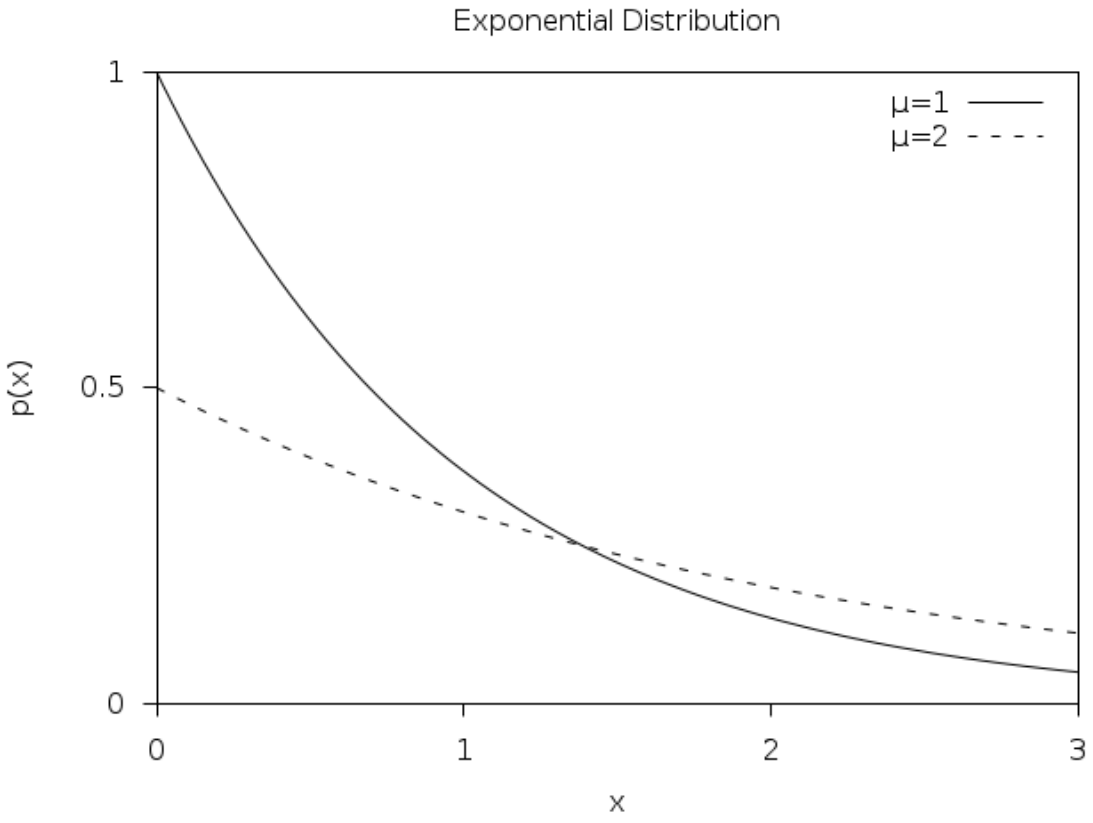
This function returns a random variate from the exponential distribution with mean `mu`. The distribution is,

$$p(x)dx = \frac{1}{\mu}\exp(-x/\mu)dx$$

for $x \geq 0$.

double **gsl_ran_exponential_pdf**(double x, double mu)

This function computes the probability density $p(x)$ at `x` for an exponential distribution with mean `mu`, using the formula given above.



double **gsl_cdf_exponential_P**(double x, double mu)

double **gsl_cdf_exponential_Q**(double x, double mu)

double **gsl_cdf_exponential_Pinv**(double P, double mu)

double **gsl_cdf_exponential_Qinv**(double Q, double mu)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the exponential distribution with mean `mu`.

20.7 The Laplace Distribution

double **gsl_ran_laplace**(const gsl_rng *r, double a)

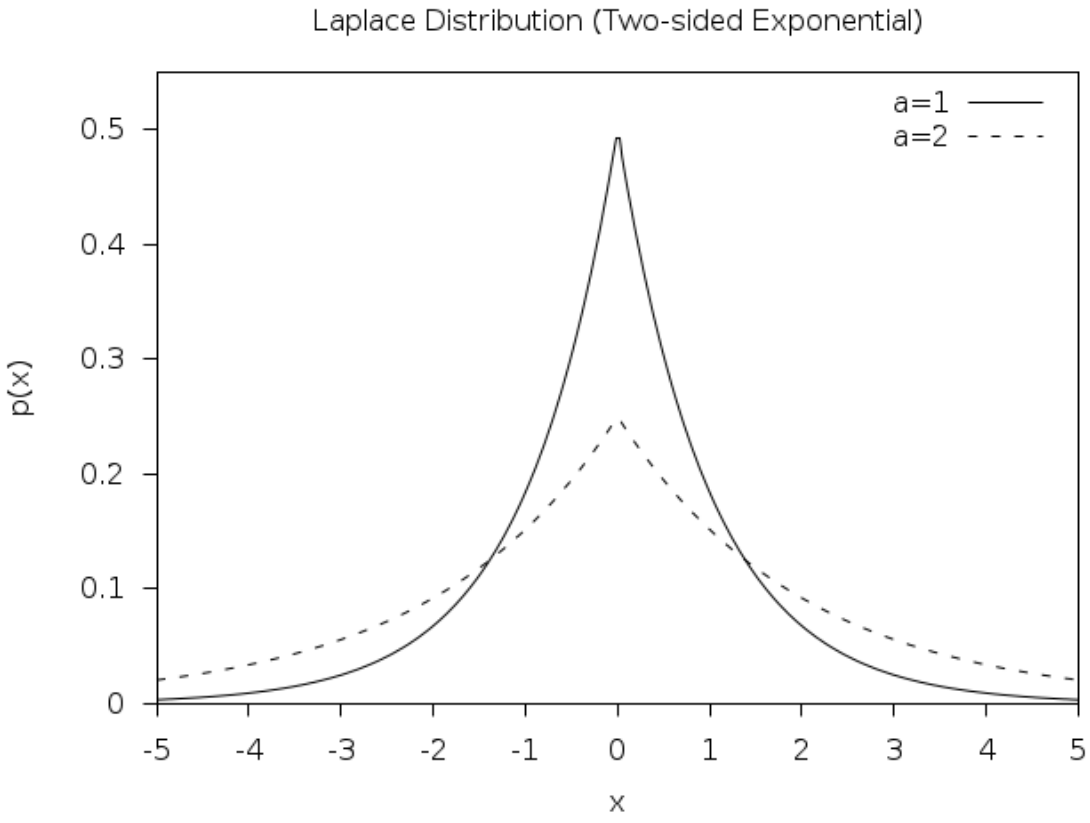
This function returns a random variate from the Laplace distribution with width a. The distribution is,

$$p(x)dx = \frac{1}{2a}\exp(-|x/a|)dx$$

for $-\infty < x < \infty$.

double **gsl_ran_laplace_pdf**(double x, double a)

This function computes the probability density $p(x)$ at x for a Laplace distribution with width a, using the formula given above.



double **gsl_cdf_laplace_P**(double x, double a)

double **gsl_cdf_laplace_Q**(double x, double a)

double **gsl_cdf_laplace_Pinv**(double P, double a)

double **gsl_cdf_laplace_Qinv**(double Q, double a)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the Laplace distribution with width a.

20.8 The Exponential Power Distribution

double **gsl_ran_exppow**(const gsl_rng *r, double a, double b)

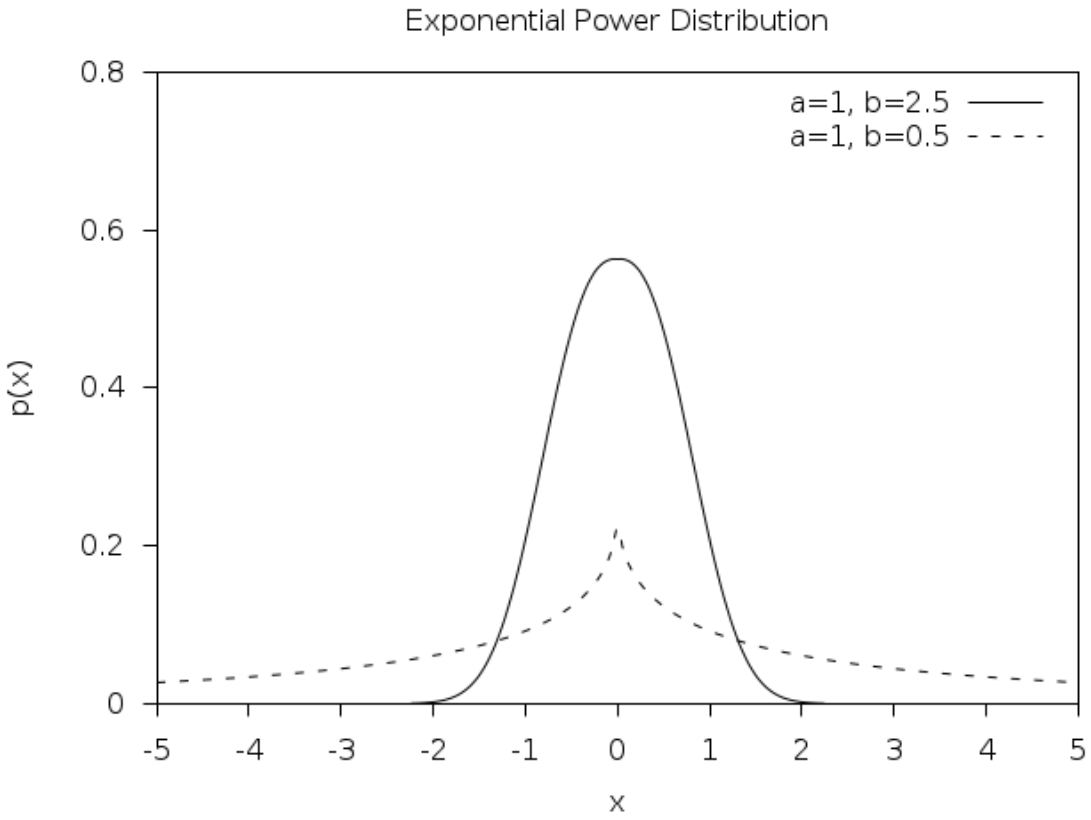
This function returns a random variate from the exponential power distribution with scale parameter `a` and exponent `b`. The distribution is,

$$p(x)dx = \frac{1}{2a\Gamma(1+1/b)} \exp(-|x/a|^b)dx$$

for $x \geq 0$. For $b = 1$ this reduces to the Laplace distribution. For $b = 2$ it has the same form as a Gaussian distribution, but with $a = \sqrt{2}\sigma$.

double **gsl_ran_exppow_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at `x` for an exponential power distribution with scale parameter `a` and exponent `b`, using the formula given above.



double **gsl_cdf_exppow_P**(double x, double a, double b)

double **gsl_cdf_exppow_Q**(double x, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ for the exponential power distribution with parameters `a` and `b`.

20.9 The Cauchy Distribution

double **gsl_ran_cauchy**(const gsl_rng *r, double a)

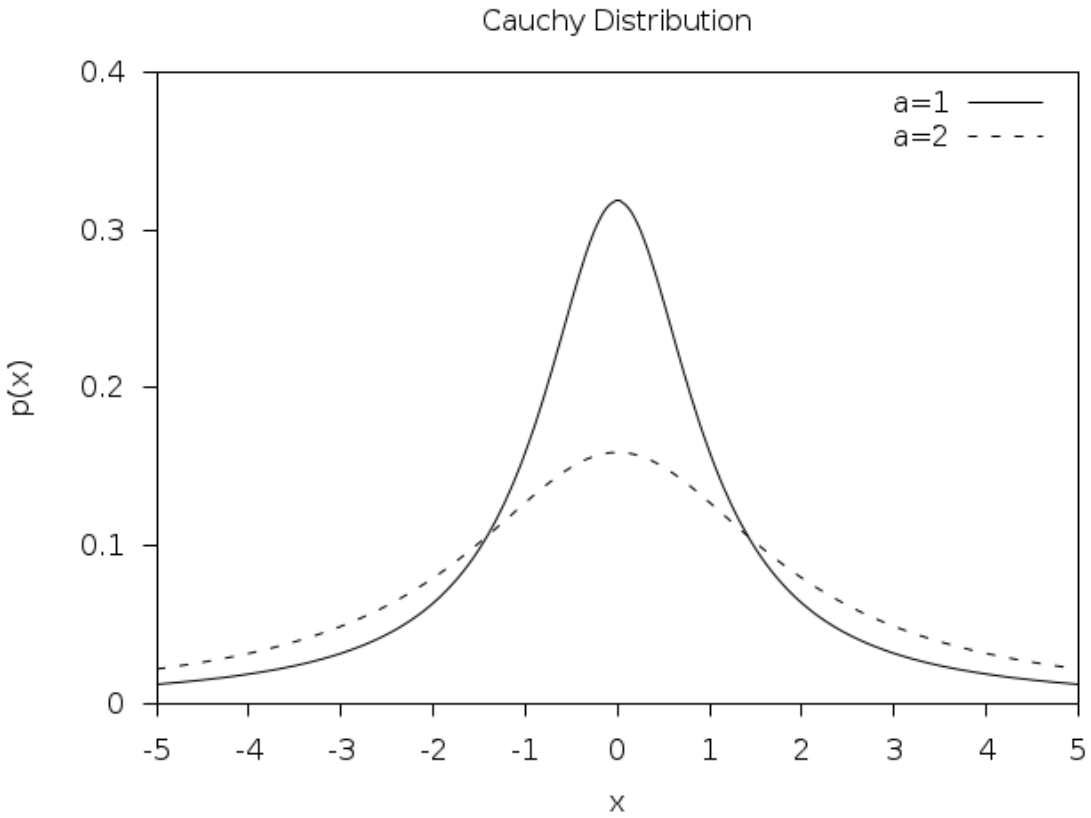
This function returns a random variate from the Cauchy distribution with scale parameter a. The probability distribution for Cauchy random variates is,

$$p(x)dx = \frac{1}{a\pi(1 + (x/a)^2)}dx$$

for x in the range $-\infty$ to $+\infty$. The Cauchy distribution is also known as the Lorentz distribution.

double **gsl_ran_cauchy_pdf**(double x, double a)

This function computes the probability density $p(x)$ at x for a Cauchy distribution with scale parameter a, using the formula given above.



double **gsl_cdf_cauchy_P**(double x, double a)

double **gsl_cdf_cauchy_Q**(double x, double a)

double **gsl_cdf_cauchy_Pinv**(double P, double a)

double **gsl_cdf_cauchy_Qinv**(double Q, double a)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the Cauchy distribution with scale parameter a.

20.10 The Rayleigh Distribution

double **gsl_ran_rayleigh**(const gsl_rng *r, double sigma)

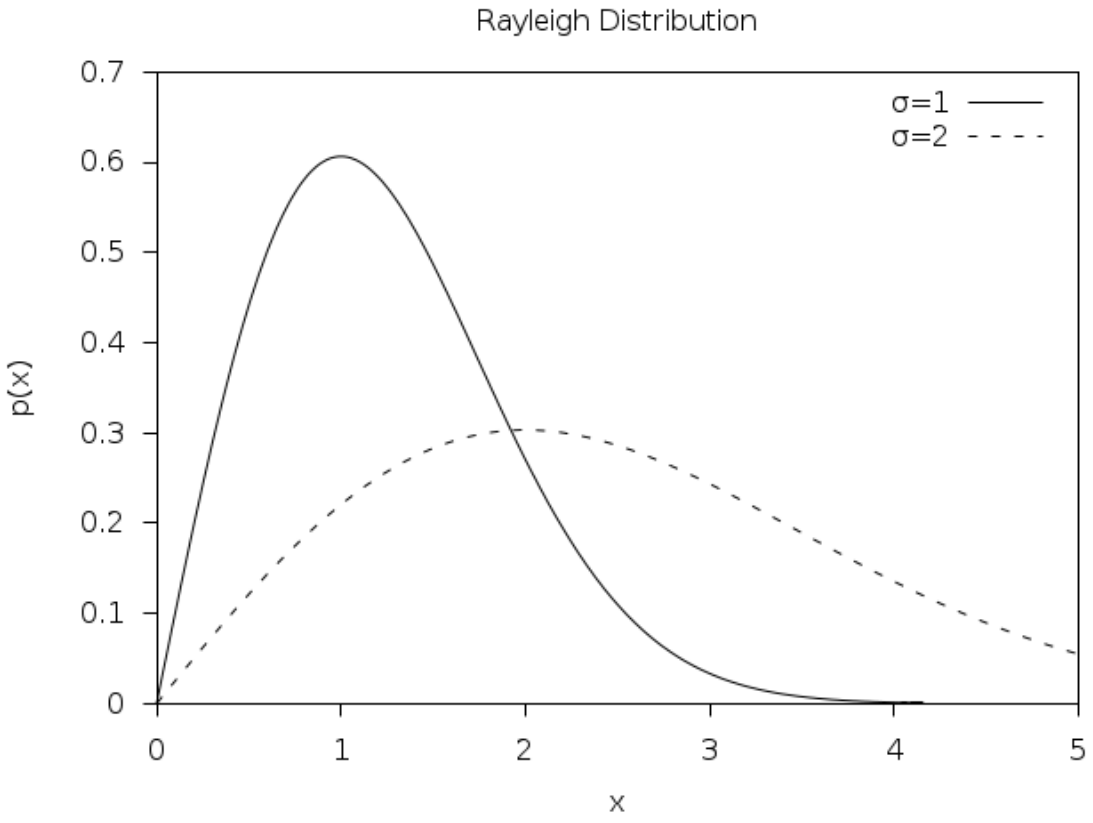
This function returns a random variate from the Rayleigh distribution with scale parameter `sigma`. The distribution is,

$$p(x)dx = \frac{x}{\sigma^2} \exp(-x^2/(2\sigma^2))dx$$

for $x > 0$.

double **gsl_ran_rayleigh_pdf**(double x, double sigma)

This function computes the probability density $p(x)$ at `x` for a Rayleigh distribution with scale parameter `sigma`, using the formula given above.



double **gsl_cdf_rayleigh_P**(double x, double sigma)

double **gsl_cdf_rayleigh_Q**(double x, double sigma)

double **gsl_cdf_rayleigh_Pinv**(double P, double sigma)

double **gsl_cdf_rayleigh_Qinv**(double Q, double sigma)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the Rayleigh distribution with scale parameter `sigma`.

20.11 The Rayleigh Tail Distribution

double **gsl_ran_rayleigh_tail**(const gsl_rng *r, double a, double sigma)

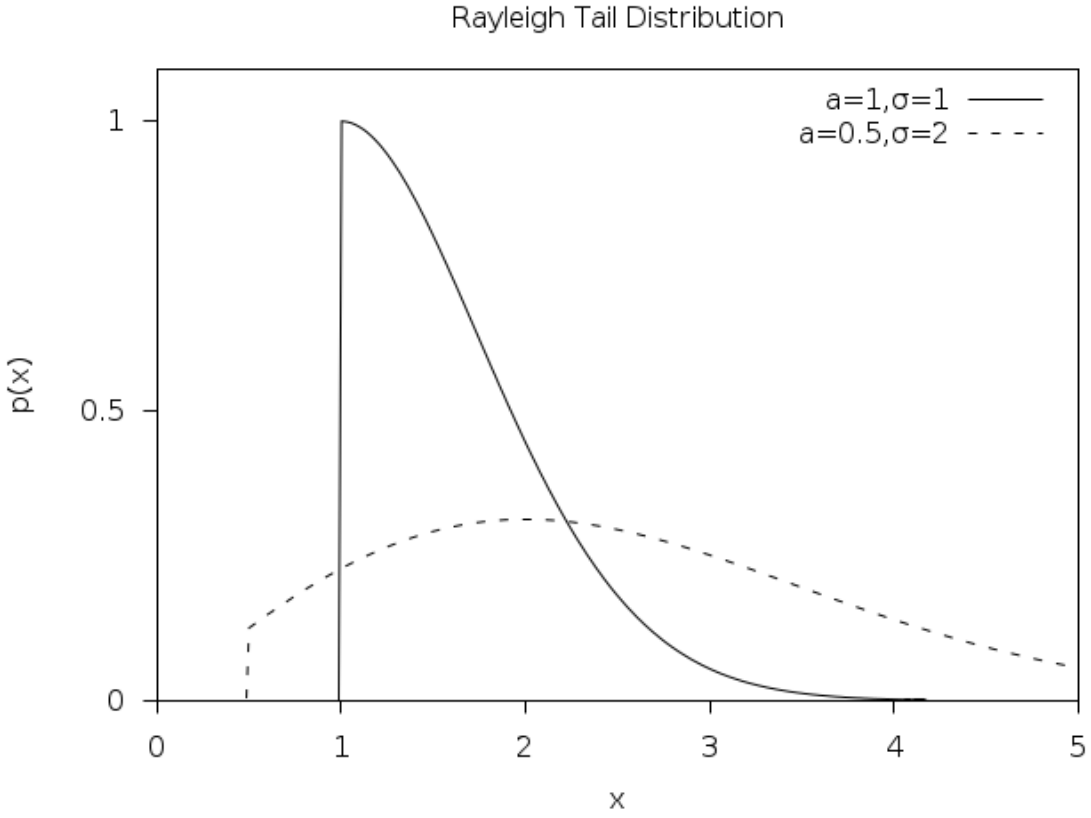
This function returns a random variate from the tail of the Rayleigh distribution with scale parameter `sigma` and a lower limit of `a`. The distribution is,

$$p(x)dx = \frac{x}{\sigma^2} \exp((a^2 - x^2)/(2\sigma^2))dx$$

for $x > a$.

double **gsl_ran_rayleigh_tail_pdf**(double x, double a, double sigma)

This function computes the probability density $p(x)$ at `x` for a Rayleigh tail distribution with scale parameter `sigma` and lower limit `a`, using the formula given above.



20.12 The Landau Distribution

double **gsl_ran_landau**(const gsl_rng *r)

This function returns a random variate from the Landau distribution. The probability distribution for Landau random variates is defined analytically by the complex integral,

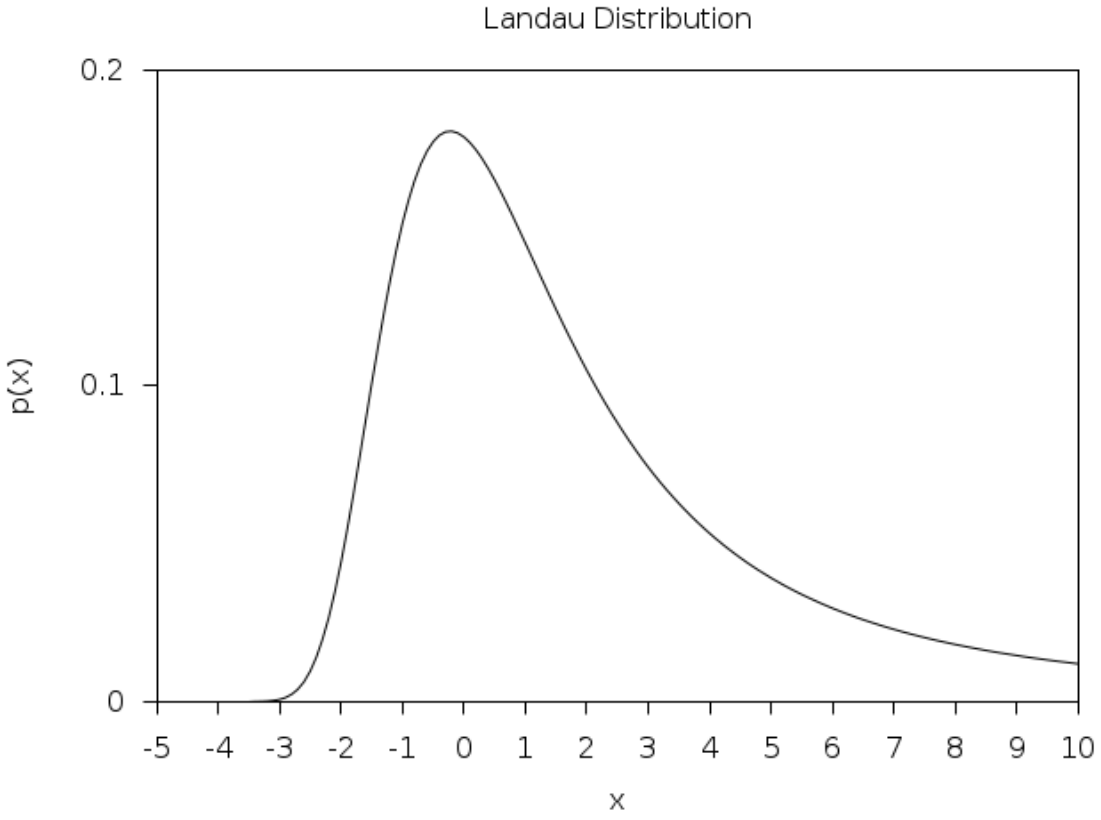
$$p(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} ds\ \exp(s\log(s) + xs)$$

For numerical purposes it is more convenient to use the following equivalent form of the integral,

$$p(x) = (1/\pi) \int_0^\infty dt \exp(-t\log(t) - xt)\sin(\pi t).$$

double **gsl_ran_landau_pdf**(double x)

This function computes the probability density $p(x)$ at x for the Landau distribution using an approximation to the formula given above.



20.13 The Levy alpha-Stable Distributions

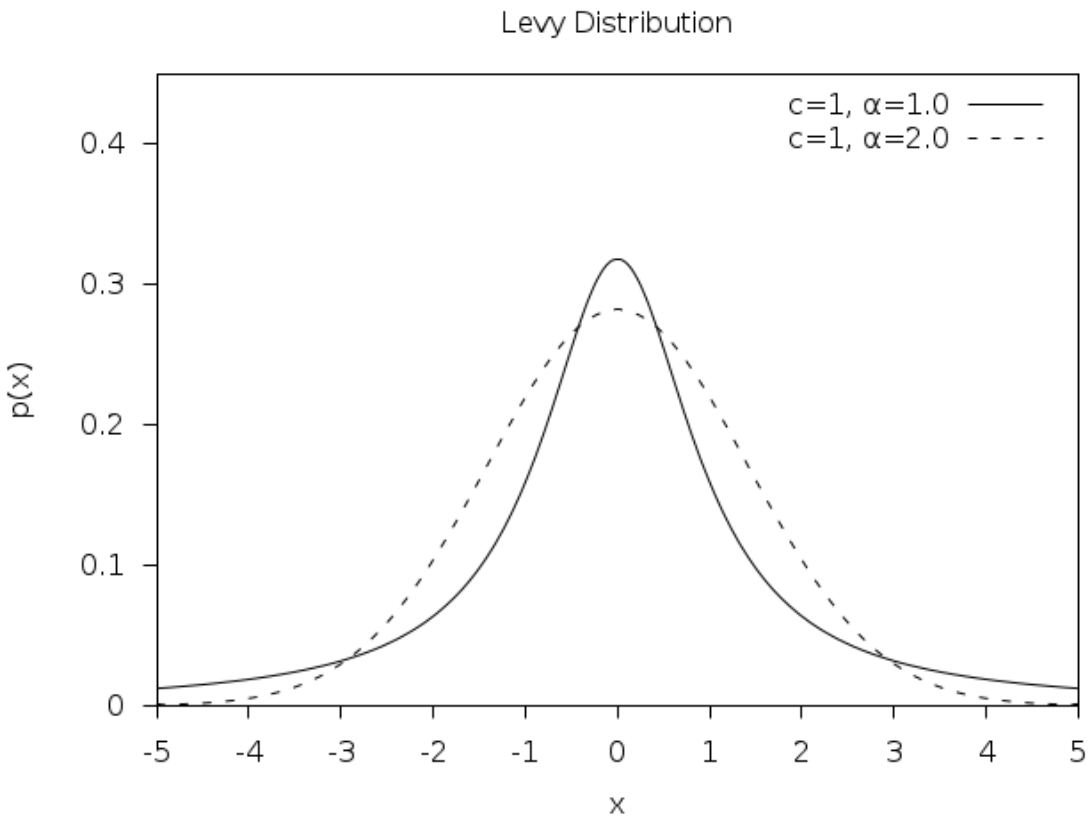
double **gsl_ran_levy**(const gsl_rng *r, double c, double alpha)

This function returns a random variate from the Levy symmetric stable distribution with scale `c` and exponent `alpha`. The symmetric stable probability distribution is defined by a Fourier transform,

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dt \exp(-itx - |ct|^{\alpha})$$

There is no explicit solution for the form of $p(x)$ and the library does not define a corresponding `pdf` function. For $\alpha = 1$ the distribution reduces to the Cauchy distribution. For $\alpha = 2$ it is a Gaussian distribution with $\sigma = \sqrt{2}c$. For $\alpha < 1$ the tails of the distribution become extremely wide.

The algorithm only works for $0 < \alpha \leq 2$.



20.14 The Levy skew alpha-Stable Distribution

double **gsl_ran_levy_skew**(const gsl_rng *r, double c, double alpha, double beta)

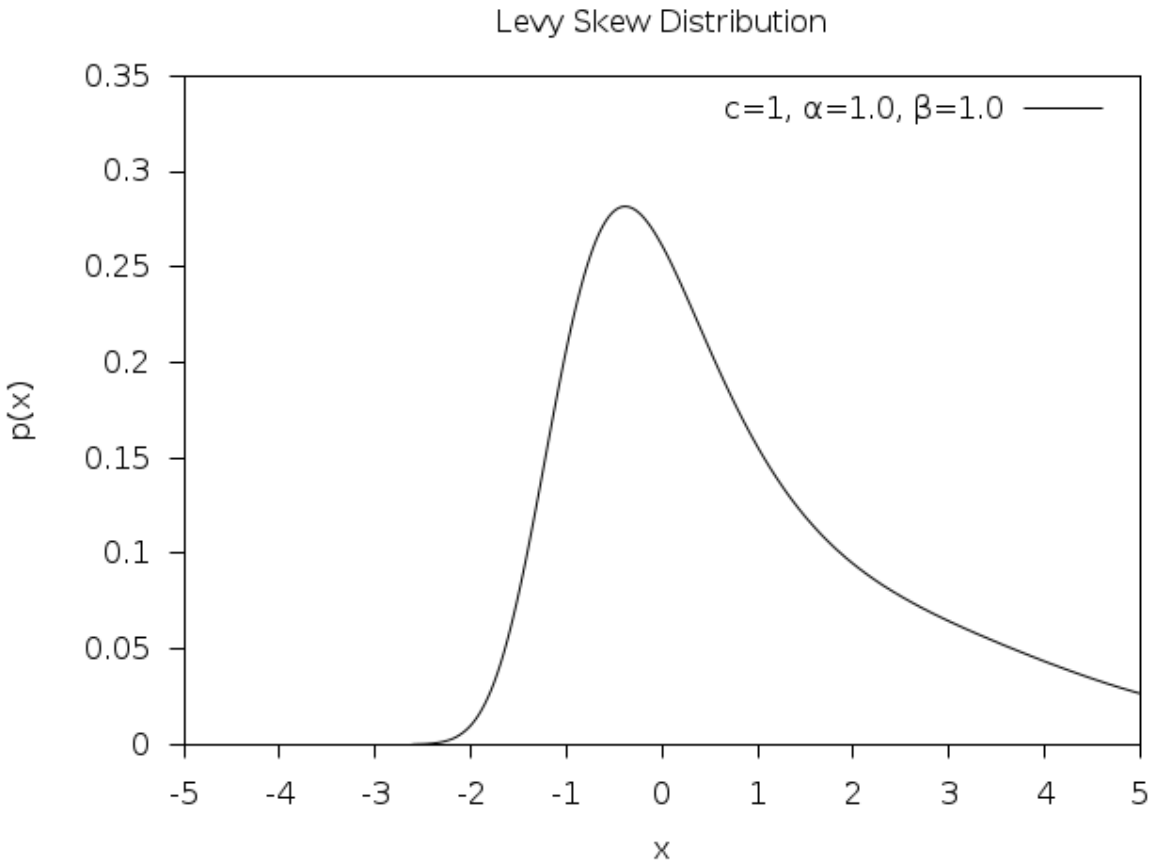
This function returns a random variate from the Levy skew stable distribution with scale `c`, exponent `alpha` and skewness parameter `beta`. The skewness parameter must lie in the range $[-1, 1]$. The Levy skew stable probability distribution is defined by a Fourier transform,

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dt \exp(-itx - |ct|^\alpha (1 - i\beta(t) \tan(\pi\alpha/2)))$$

When $\alpha = 1$ the term $\tan(\pi\alpha/2)$ is replaced by $-(2/\pi)\log|t|$. There is no explicit solution for the form of $p(x)$ and the library does not define a corresponding `pdf` function. For $\alpha = 2$ the distribution reduces to a Gaussian distribution with $\sigma = \sqrt{2}c$ and the skewness parameter has no effect. For $\alpha < 1$ the tails of the distribution become extremely wide. The symmetric distribution corresponds to $\beta = 0$.

The algorithm only works for $0 < \alpha \leq 2$.

The Levy alpha-stable distributions have the property that if N alpha-stable variates are drawn from the distribution $p(c, \alpha, \beta)$ then the sum $Y = X_1 + X_2 + \cdots + X_N$ will also be distributed as an alpha-stable variate, $p(N^{1/\alpha}c, \alpha, \beta)$.


Levy Skew Distribution
c=1, α=1.0, β=1.0
p(x)
x
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-5
-4
-3
-2
-1
0
1
2
3
4
5

20.15 The Gamma Distribution

double **gsl_ran_gamma**(const gsl_rng *r, double a, double b)

This function returns a random variate from the gamma distribution. The distribution function is,

$$p(x)dx = \frac{1}{\Gamma(a)b^a} x^{a-1} e^{-x/b} dx$$

for $x > 0$.

The gamma distribution with an integer parameter a is known as the Erlang distribution.

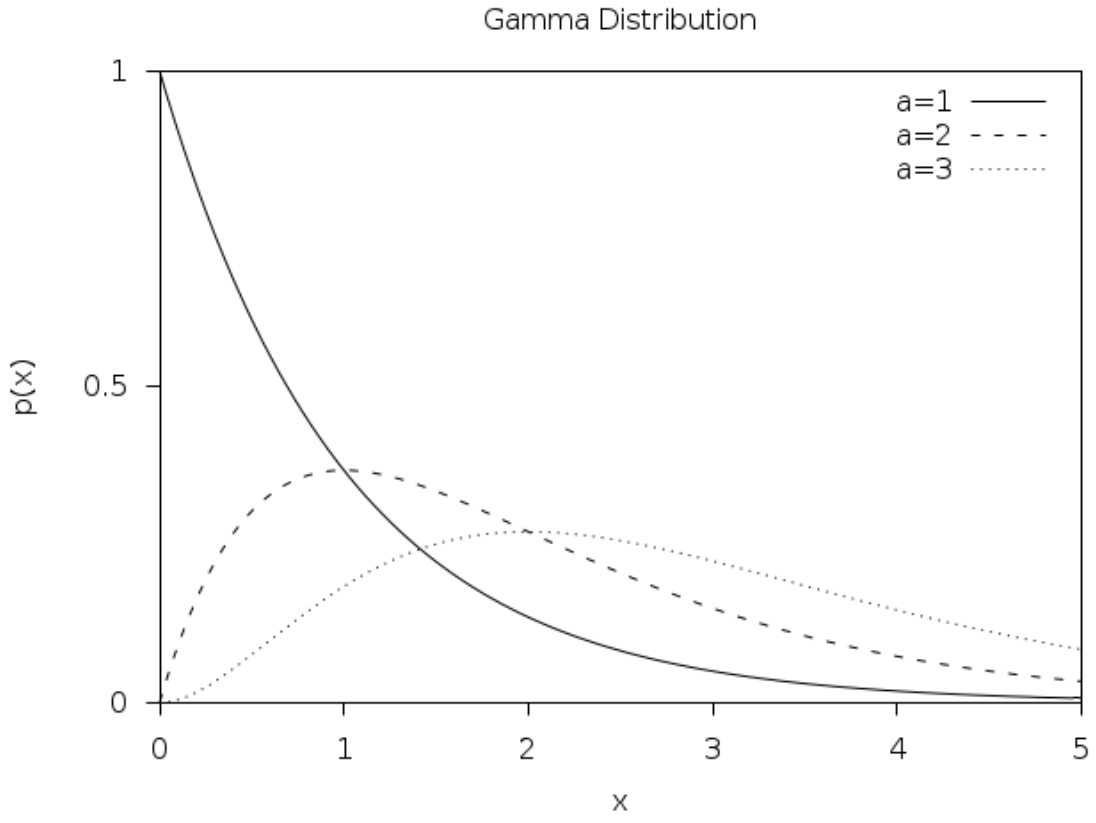
The variates are computed using the Marsaglia-Tsang fast gamma method. This function for this method was previously called gsl_ran_gamma_mt() and can still be accessed using this name.

double **gsl_ran_gamma_knuth**(const gsl_rng *r, double a, double b)

This function returns a gamma variate using the algorithms from Knuth (vol 2).

double **gsl_ran_gamma_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at x for a gamma distribution with parameters a and b, using the formula given above.



double **gsl_cdf_gamma_P**(double x, double a, double b)

double **gsl_cdf_gamma_Q**(double x, double a, double b)

double **gsl_cdf_gamma_Pinv**(double P, double a, double b)

double **gsl_cdf_gamma_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the gamma distribution with parameters `a` and `b`.

20.16 The Flat (Uniform) Distribution

double **gsl_ran_flat**(const gsl_rng *r, double a, double b)

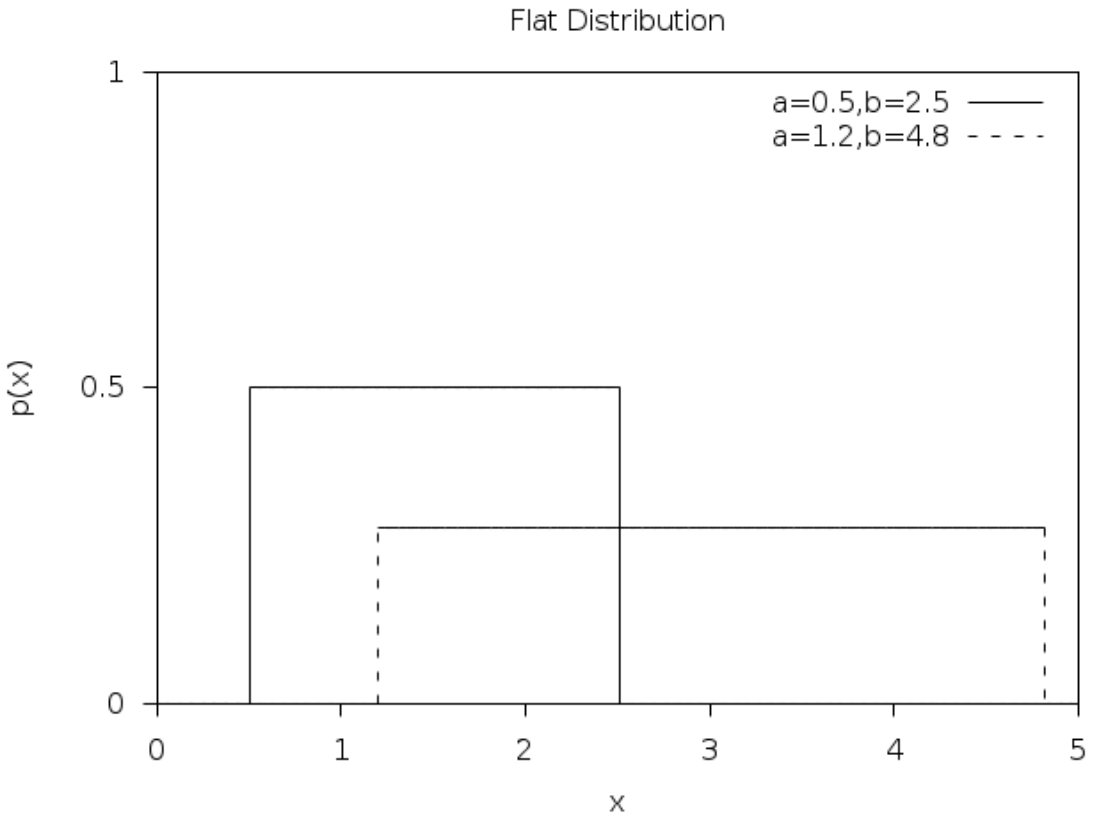
This function returns a random variate from the flat (uniform) distribution from `a` to `b`. The distribution is,

$$p(x)dx = \frac{1}{(b-a)}dx$$

if $a \leq x < b$ and 0 otherwise.

double **gsl_ran_flat_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at `x` for a uniform distribution from `a` to `b`, using the formula given above.



double **gsl_cdf_flat_P**(double x, double a, double b)

double **gsl_cdf_flat_Q**(double x, double a, double b)

double **gsl_cdf_flat_Pinv**(double P, double a, double b)

double **gsl_cdf_flat_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for a uniform distribution from `a` to `b`.

20.17 The Lognormal Distribution

double **gsl_ran_lognormal**(const gsl_rng *r, double zeta, double sigma)

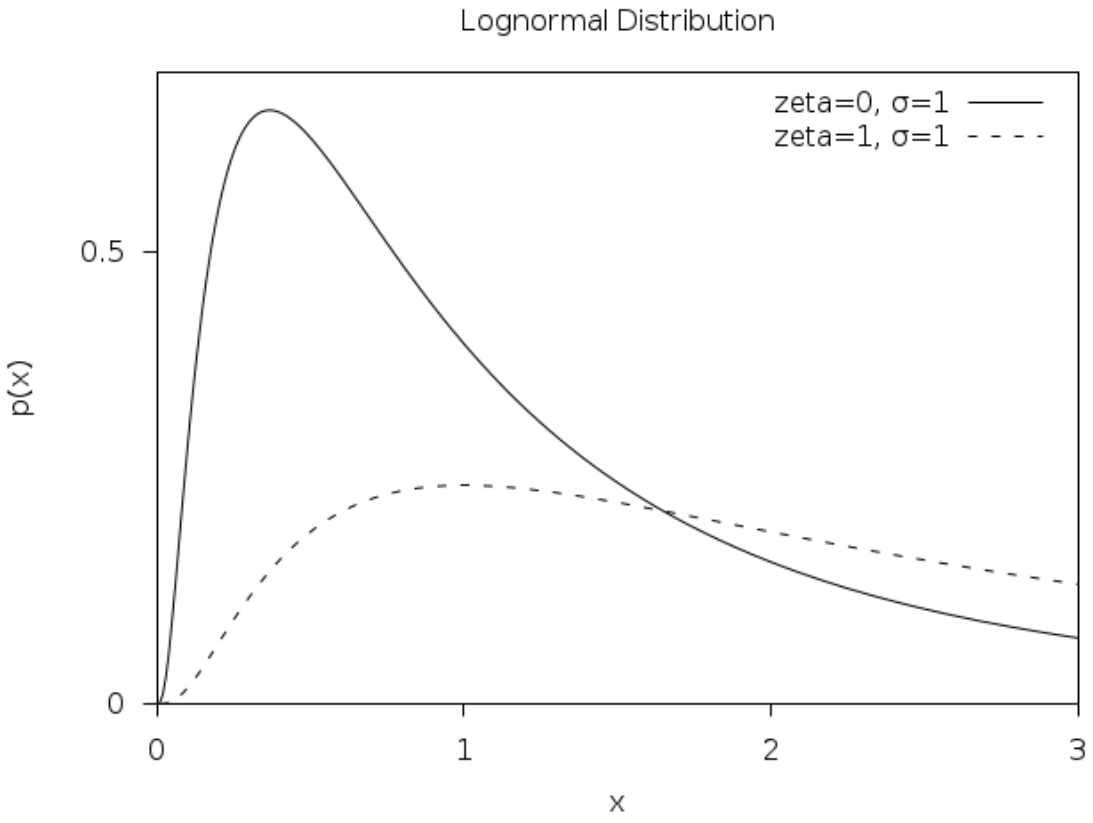
This function returns a random variate from the lognormal distribution. The distribution function is,

$$p(x)dx = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp(-(\ln(x) - \zeta)^2/2\sigma^2)dx$$

for $x > 0$.

double **gsl_ran_lognormal_pdf**(double x, double zeta, double sigma)

This function computes the probability density $p(x)$ at x for a lognormal distribution with parameters zeta and sigma, using the formula given above.



double **gsl_cdf_lognormal_P**(double x, double zeta, double sigma)

double **gsl_cdf_lognormal_Q**(double x, double zeta, double sigma)

double **gsl_cdf_lognormal_Pinv**(double P, double zeta, double sigma)

double **gsl_cdf_lognormal_Qinv**(double Q, double zeta, double sigma)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the lognormal distribution with parameters zeta and sigma.

20.18 The Chi-squared Distribution

The chi-squared distribution arises in statistics. If Y_i are n independent Gaussian random variates with unit variance then the sum-of-squares,

$$X_i = \sum_i Y_i^2$$

has a chi-squared distribution with n degrees of freedom.

double **gsl_ran_chisq**(const gsl_rng *r, double nu)

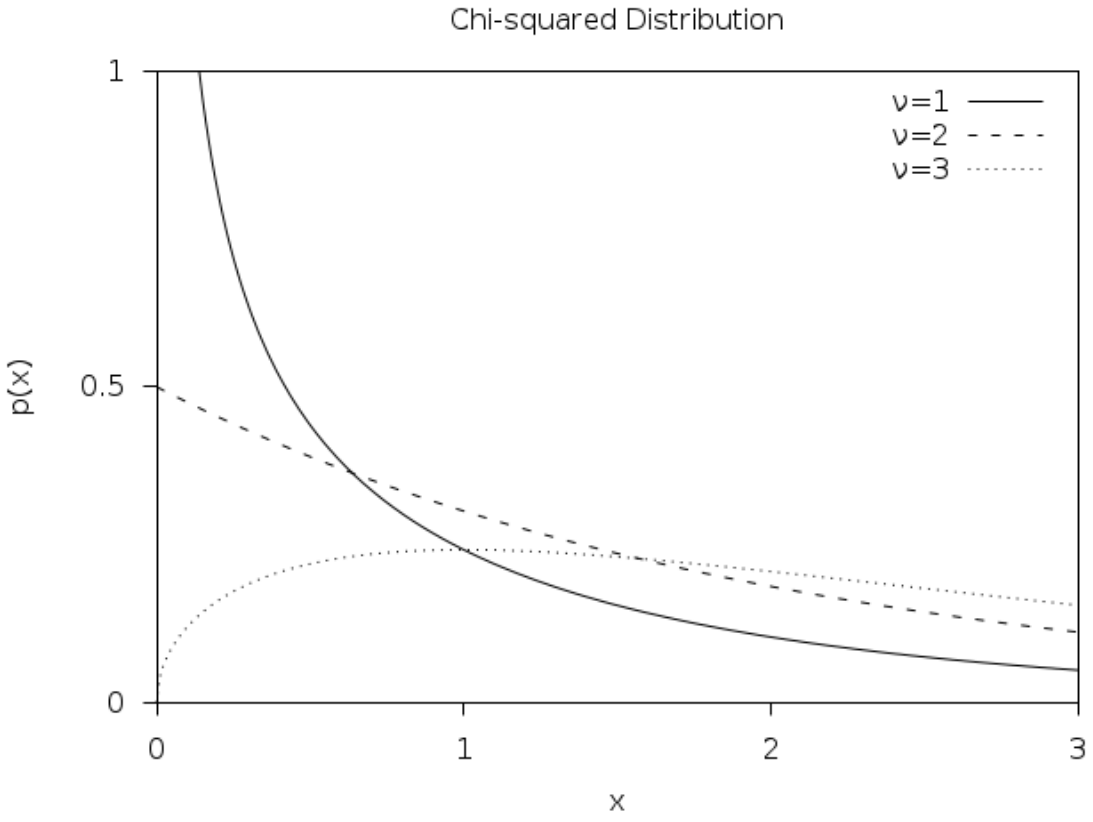
This function returns a random variate from the chi-squared distribution with `nu` degrees of freedom. The distribution function is,

$$p(x)dx = \frac{1}{2\Gamma(\nu/2)}(x/2)^{\nu/2-1}\exp(-x/2)dx$$

for $x \geq 0$.

double **gsl_ran_chisq_pdf**(double x, double nu)

This function computes the probability density $p(x)$ at `x` for a chi-squared distribution with `nu` degrees of freedom, using the formula given above.



double **gsl_cdf_chisq_P**(double x, double nu)

double **gsl_cdf_chisq_Q**(double x, double nu)

double **gsl_cdf_chisq_Pinv**(double P, double nu)

double **gsl_cdf_chisq_Qinv**(double Q, double nu)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the chi-squared distribution with `nu` degrees of freedom.

20.19 The F-distribution

The F-distribution arises in statistics. If Y_1 and Y_2 are chi-squared deviates with ν_1 and ν_2 degrees of freedom then the ratio,

$$X = \frac{(Y_1/\nu_1)}{(Y_2/\nu_2)}$$

has an F-distribution $F(x; \nu_1, \nu_2)$.

double **gsl_ran_fdist**(const gsl_rng *r, double nu1, double nu2)

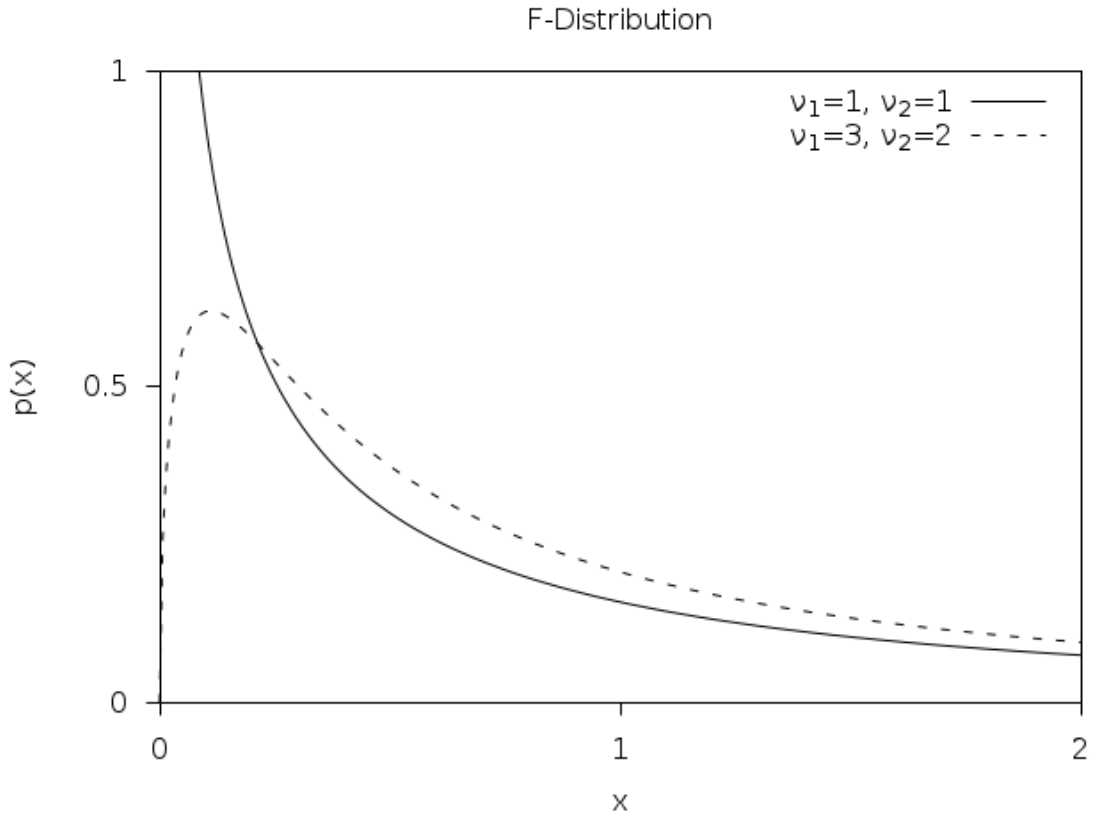
This function returns a random variate from the F-distribution with degrees of freedom `nu1` and `nu2`. The distribution function is,

$$p(x)dx = \frac{\Gamma((\nu_1 + \nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} x^{\nu_1/2-1} (\nu_2 + \nu_1 x)^{-\nu_1/2-\nu_2/2}$$

for $x \geq 0$.

double **gsl_ran_fdist_pdf**(double x, double nu1, double nu2)

This function computes the probability density $p(x)$ at `x` for an F-distribution with `nu1` and `nu2` degrees of freedom, using the formula given above.



double **gsl_cdf_fdist_P**(double x, double nu1, double nu2)

double **gsl_cdf_fdist_Q**(double x, double nu1, double nu2)

double **gsl_cdf_fdist_Pinv**(double P, double nu1, double nu2)

double **gsl_cdf_fdist_Qinv**(double Q, double nu1, double nu2)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the F-distribution with `nu1` and `nu2` degrees of freedom.

20.20 The t-distribution

The t-distribution arises in statistics. If Y_1 has a normal distribution and Y_2 has a chi-squared distribution with ν degrees of freedom then the ratio,

$$X = \frac{Y_1}{\sqrt{Y_2/\nu}}$$

has a t-distribution $t(x;\nu)$ with ν degrees of freedom.

double **gsl_ran_tdist**(const gsl_rng *r, double nu)

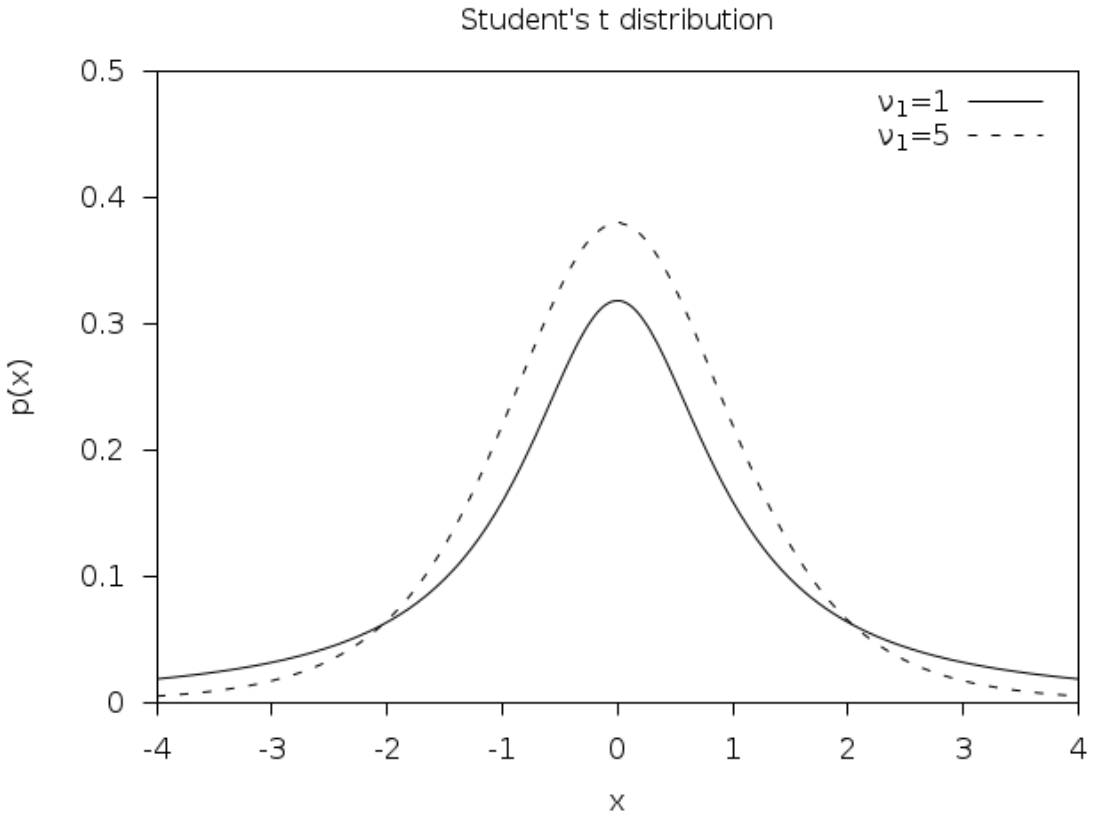
This function returns a random variate from the t-distribution. The distribution function is,

$$p(x)dx = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}(1+x^2/\nu)^{-(\nu+1)/2}dx$$

for $-\infty < x < +\infty$.

double **gsl_ran_tdist_pdf**(double x, double nu)

This function computes the probability density $p(x)$ at `x` for a t-distribution with `nu` degrees of freedom, using the formula given above.



double **gsl_cdf_tdist_P**(double x, double nu)

double **gsl_cdf_tdist_Q**(double x, double nu)

double **gsl_cdf_tdist_Pinv**(double P, double nu)

double **gsl_cdf_tdist_Qinv**(double Q, double nu)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the t-distribution with `nu` degrees of freedom.

20.21 The Beta Distribution

double **gsl_ran_beta**(const gsl_rng *r, double a, double b)

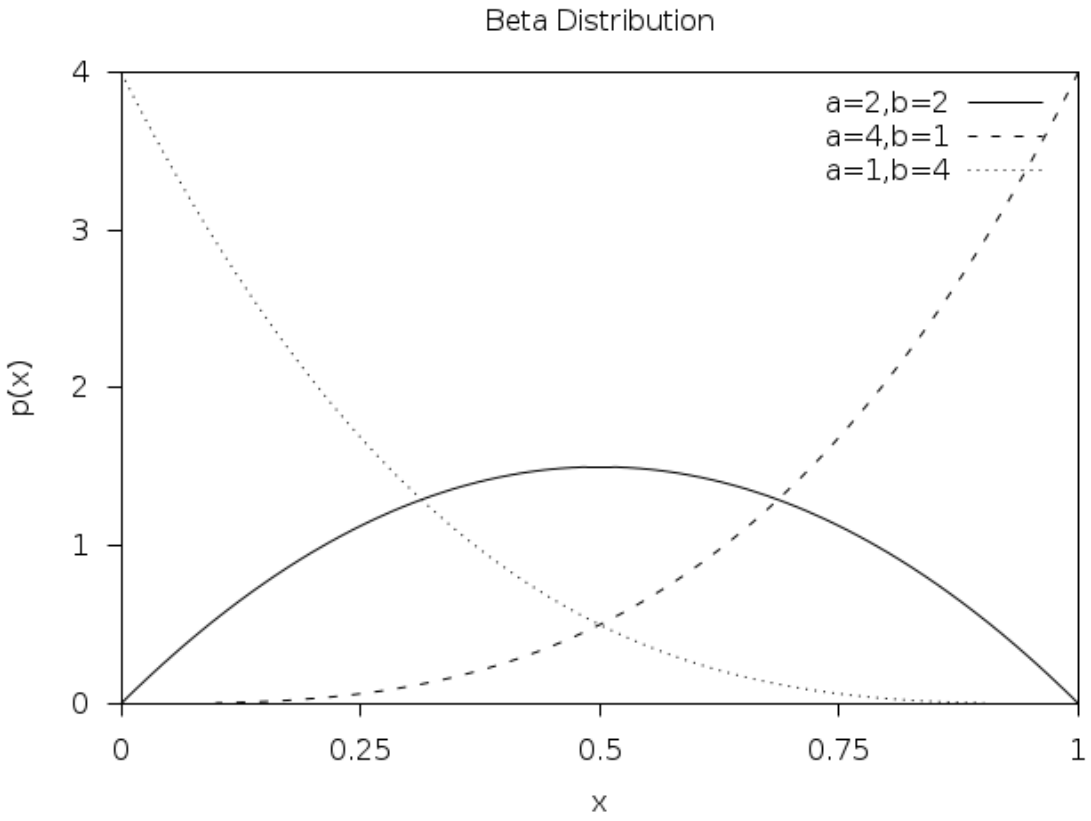
This function returns a random variate from the beta distribution. The distribution function is,

$$p(x)dx = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}dx$$

for $0 \leq x \leq 1$.

double **gsl_ran_beta_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at `x` for a beta distribution with parameters `a` and `b`, using the formula given above.



double **gsl_cdf_beta_P**(double x, double a, double b)

double **gsl_cdf_beta_Q**(double x, double a, double b)

double **gsl_cdf_beta_Pinv**(double P, double a, double b)

double **gsl_cdf_beta_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the beta distribution with parameters `a` and `b`.

20.22 The Logistic Distribution

double **gsl_ran_logistic**(const gsl_rng *r, double a)

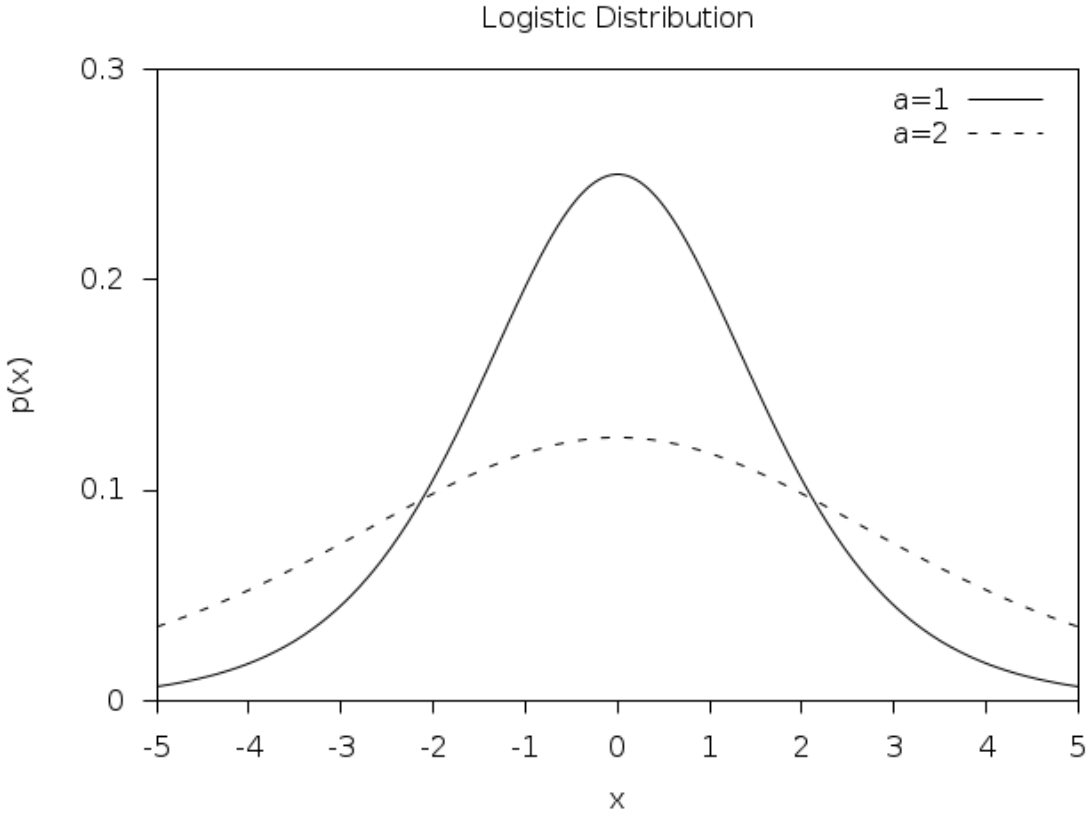
This function returns a random variate from the logistic distribution. The distribution function is,

$$p(x)dx = \frac{\exp(-x/a)}{a(1+\exp(-x/a))^2}dx$$

for $-\infty < x < +\infty$.

double **gsl_ran_logistic_pdf**(double x, double a)

This function computes the probability density $p(x)$ at x for a logistic distribution with scale parameter a, using the formula given above.



double **gsl_cdf_logistic_P**(double x, double a)

double **gsl_cdf_logistic_Q**(double x, double a)

double **gsl_cdf_logistic_Pinv**(double P, double a)

double **gsl_cdf_logistic_Qinv**(double Q, double a)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the logistic distribution with scale parameter a.

20.23 The Pareto Distribution

double **gsl_ran_pareto**(const gsl_rng *r, double a, double b)

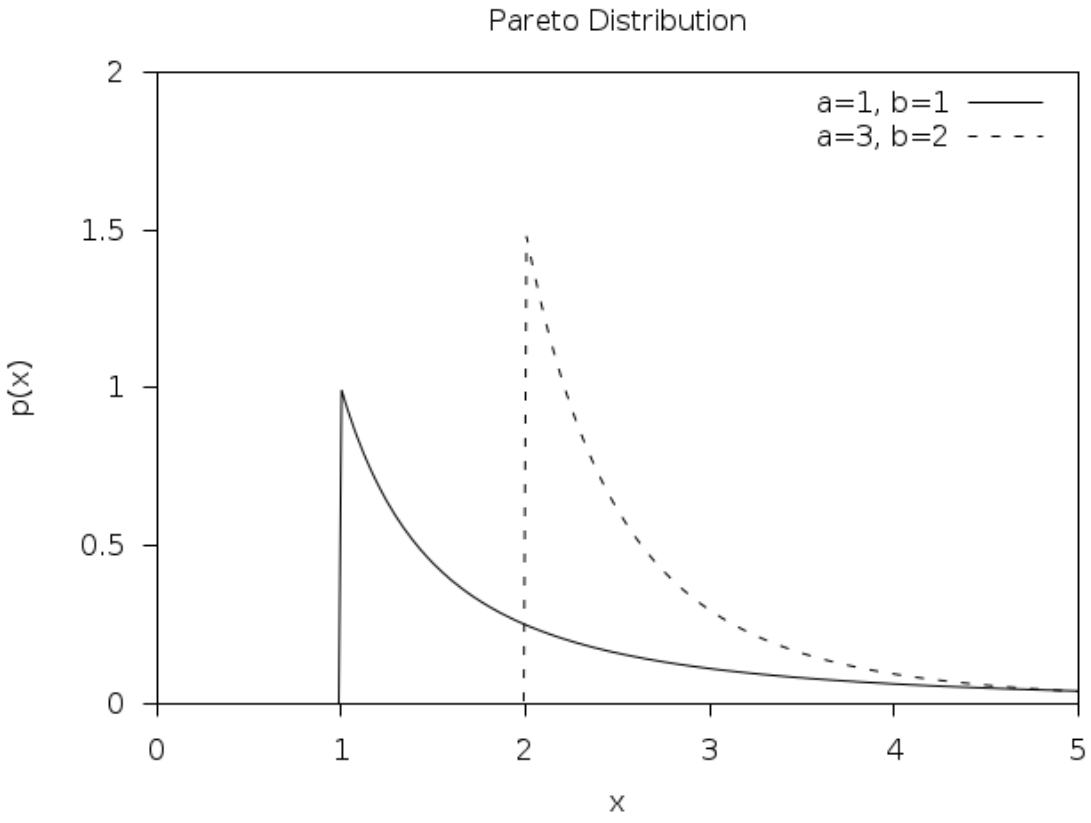
This function returns a random variate from the Pareto distribution of order `a`. The distribution function is,

$$p(x)dx = (a/b)/(x/b)^{a+1}dx$$

for $x \geq b$.

double **gsl_ran_pareto_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at `x` for a Pareto distribution with exponent `a` and scale `b`, using the formula given above.



double **gsl_cdf_pareto_P**(double x, double a, double b)

double **gsl_cdf_pareto_Q**(double x, double a, double b)

double **gsl_cdf_pareto_Pinv**(double P, double a, double b)

double **gsl_cdf_pareto_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the Pareto distribution with exponent `a` and scale `b`.

20.24 Spherical Vector Distributions

The spherical distributions generate random vectors, located on a spherical surface. They can be used as random directions, for example in the steps of a random walk.

void **gsl_ran_dir_2d**(const gsl_rng *r, double *x, double *y)

void **gsl_ran_dir_2d_trig_method**(const gsl_rng *r, double *x, double *y)

This function returns a random direction vector v = (x, y) in two dimensions. The vector is normalized such that $|v|^2 = x^2 + y^2 = 1$. The obvious way to do this is to take a uniform random number between 0 and 2π and let x and y be the sine and cosine respectively. Two trig functions would have been expensive in the old days, but with modern hardware implementations, this is sometimes the fastest way to go. This is the case for the Pentium (but not the case for the Sun Sparcstation). One can avoid the trig evaluations by choosing x and y in the interior of a unit circle (choose them at random from the interior of the enclosing square, and then reject those that are outside the unit circle), and then dividing by $\sqrt{x^2 + y^2}$. A much cleverer approach, attributed to von Neumann (See Knuth, v2, 3rd ed, p140, exercise 23), requires neither trig nor a square root. In this approach, u and v are chosen at random from the interior of a unit circle, and then $x = (u^2 - v^2)/(u^2 + v^2)$ and $y = 2uv/(u^2 + v^2)$.

void **gsl_ran_dir_3d**(const gsl_rng *r, double *x, double *y, double *z)

This function returns a random direction vector v = (x, y, z) in three dimensions. The vector is normalized such that $|v|^2 = x^2 + y^2 + z^2 = 1$. The method employed is due to Robert E. Knop (CACM 13, 326 (1970)), and explained in Knuth, v2, 3rd ed, p136. It uses the surprising fact that the distribution projected along any axis is actually uniform (this is only true for 3 dimensions).

void **gsl_ran_dir_nd**(const gsl_rng *r, size_t n, double *x)

This function returns a random direction vector $v = (x_1, x_2, \ldots, x_n)$ in n dimensions. The vector is normalized such that $|v|^2 = x_1^2 + x_2^2 + \cdots + x_n^2 = 1$. The method uses the fact that a multivariate Gaussian distribution is spherically symmetric. Each component is generated to have a Gaussian distribution, and then the components are normalized. The method is described by Knuth, v2, 3rd ed, p135–136, and attributed to G. W. Brown, Modern Mathematics for the Engineer (1956).

20.25 The Weibull Distribution

double **gsl_ran_weibull**(const gsl_rng *r, double a, double b)

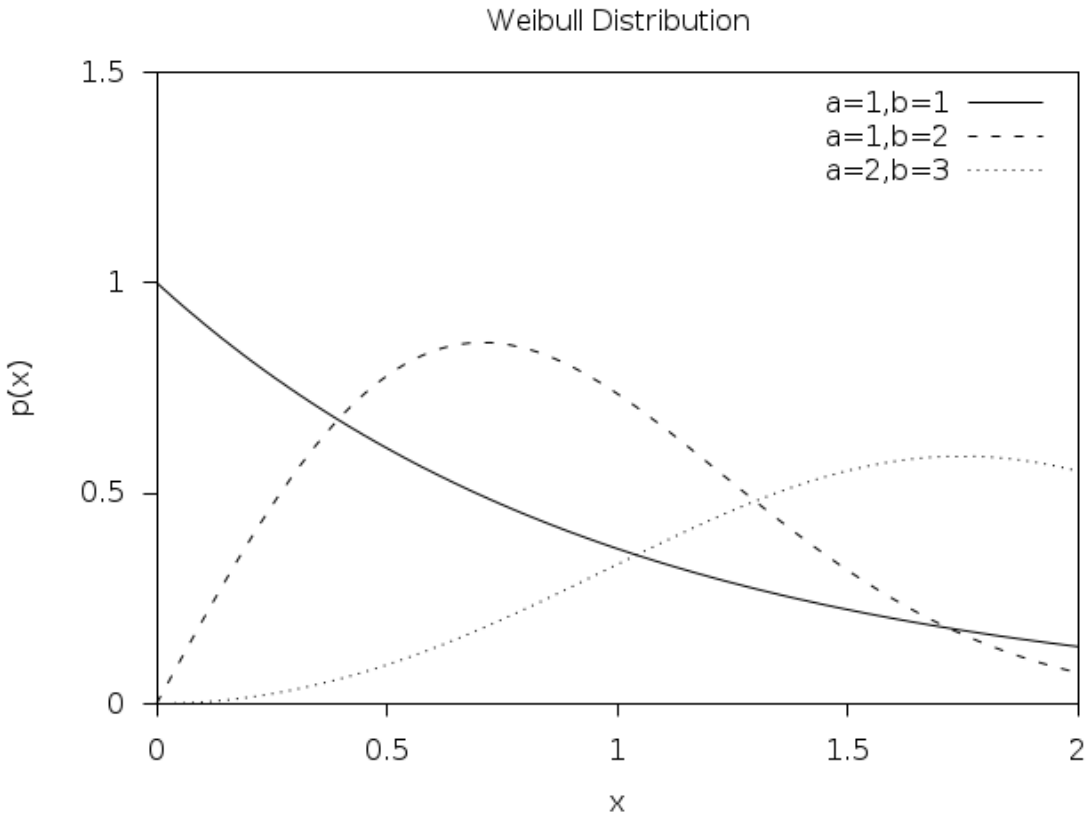
This function returns a random variate from the Weibull distribution. The distribution function is,

$$p(x)dx = \frac{b}{a^b} x^{b-1} \exp(-(x/a)^b) dx$$

for $x \geq 0$.

double **gsl_ran_weibull_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at `x` for a Weibull distribution with scale `a` and exponent `b`, using the formula given above.



double **gsl_cdf_weibull_P**(double x, double a, double b)

double **gsl_cdf_weibull_Q**(double x, double a, double b)

double **gsl_cdf_weibull_Pinv**(double P, double a, double b)

double **gsl_cdf_weibull_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the Weibull distribution with scale `a` and exponent `b`.

20.26 The Type-1 Gumbel Distribution

double **gsl_ran_gumbel1**(const gsl_rng *r, double a, double b)

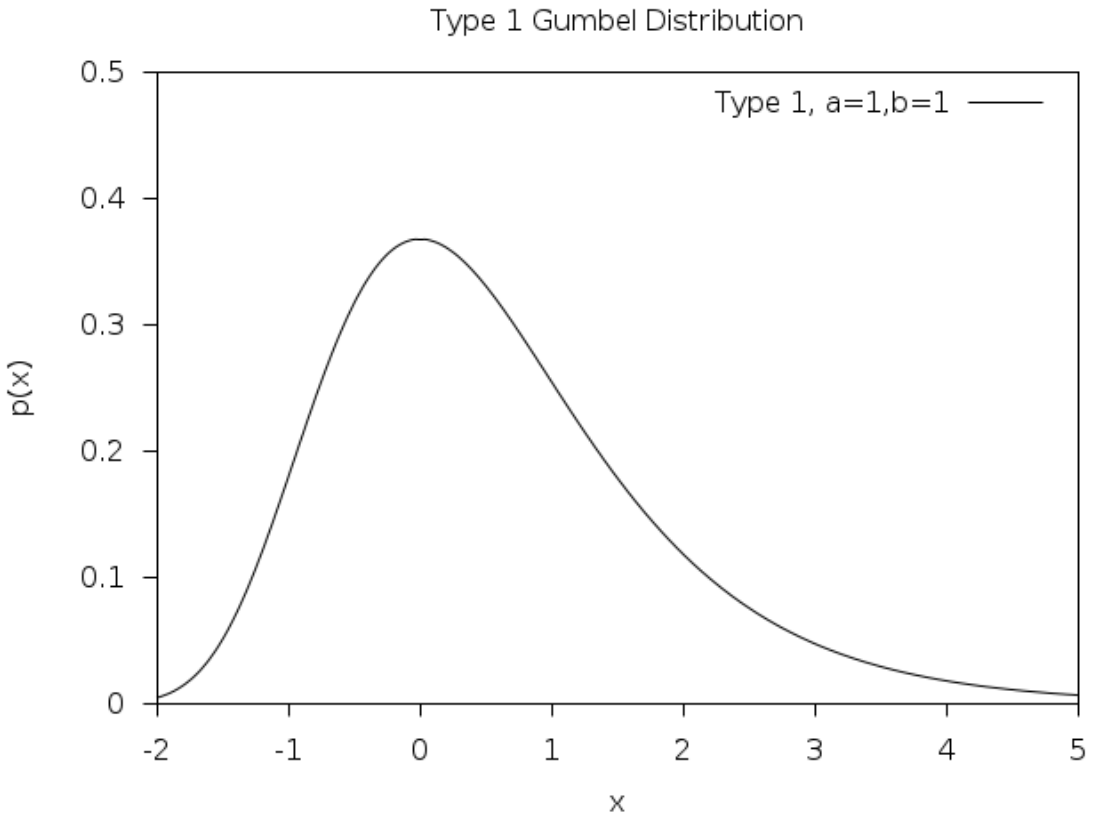
This function returns a random variate from the Type-1 Gumbel distribution. The Type-1 Gumbel distribution function is,

$$p(x)dx = ab\exp(-(b\exp(-ax) + ax))dx$$

for $-\infty < x < \infty$.

double **gsl_ran_gumbel1_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at `x` for a Type-1 Gumbel distribution with parameters `a` and `b`, using the formula given above.



double **gsl_cdf_gumbel1_P**(double x, double a, double b)

double **gsl_cdf_gumbel1_Q**(double x, double a, double b)

double **gsl_cdf_gumbel1_Pinv**(double P, double a, double b)

double **gsl_cdf_gumbel1_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the Type-1 Gumbel distribution with parameters `a` and `b`.

20.27 The Type-2 Gumbel Distribution

double **gsl_ran_gumbel2**(const gsl_rng *r, double a, double b)

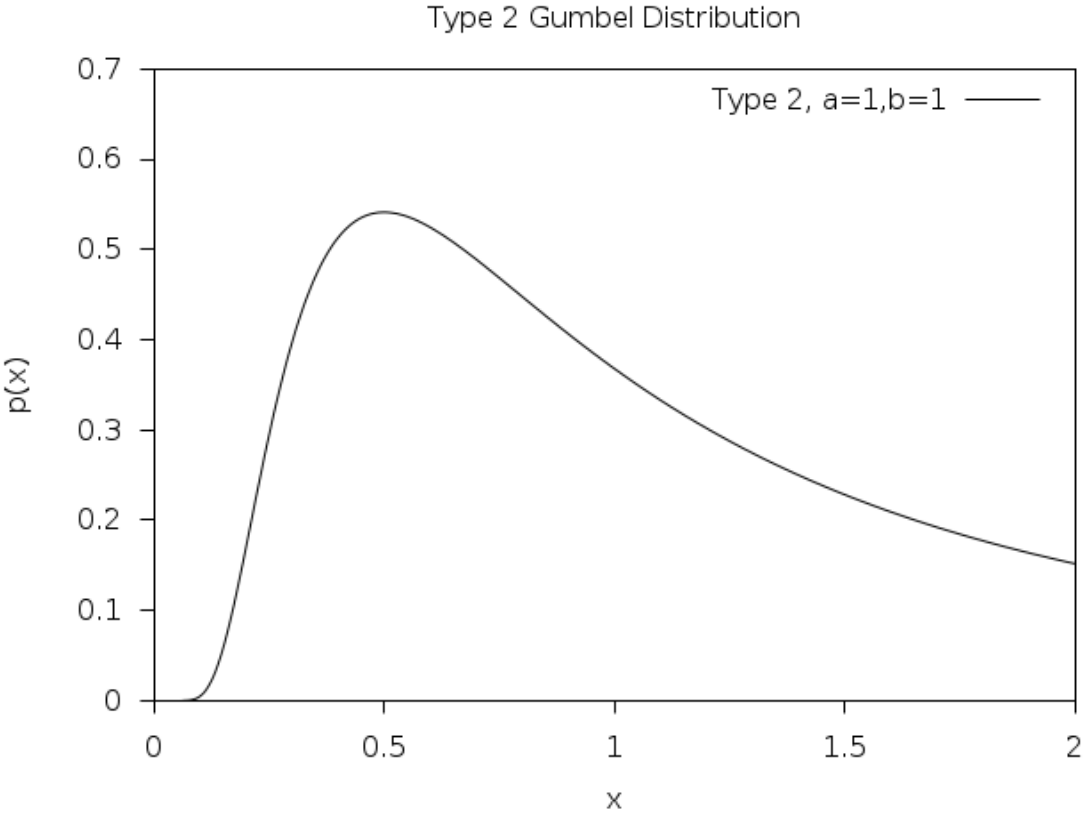
This function returns a random variate from the Type-2 Gumbel distribution. The Type-2 Gumbel distribution function is,

$$p(x)dx = abx^{-a-1}\exp(-bx^{-a})dx$$

for $0 < x < \infty$.

double **gsl_ran_gumbel2_pdf**(double x, double a, double b)

This function computes the probability density $p(x)$ at `x` for a Type-2 Gumbel distribution with parameters `a` and `b`, using the formula given above.



double **gsl_cdf_gumbel2_P**(double x, double a, double b)

double **gsl_cdf_gumbel2_Q**(double x, double a, double b)

double **gsl_cdf_gumbel2_Pinv**(double P, double a, double b)

double **gsl_cdf_gumbel2_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the Type-2 Gumbel distribution with parameters `a` and `b`.

20.28 The Dirichlet Distribution

void **gsl_ran_dirichlet**(const gsl_rng *r, size_t K, const double alpha[], double theta[])

This function returns an array of `K` random variates from a Dirichlet distribution of order `K`-1. The distribution function is

$$p(\theta_1, \ldots, \theta_K)\, d\theta_1 \cdots d\theta_K = \frac{1}{Z} \prod_{i=1}^{K} \theta_i^{\alpha_i - 1}\ \delta(1 - \sum_{i=1}^{K} \theta_i) d\theta_1 \cdots d\theta_K$$

for $\theta_i \geq 0$ and $\alpha_i > 0$. The delta function ensures that $\sum \theta_i = 1$. The normalization factor Z is

$$Z = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{K} \alpha_i)}$$

The random variates are generated by sampling `K` values from gamma distributions with parameters $a = \alpha_i$, $b = 1$, and renormalizing. See A.M. Law, W.D. Kelton, Simulation Modeling and Analysis (1991).

double **gsl_ran_dirichlet_pdf**(size_t K, const double alpha[], const double theta[])

This function computes the probability density $p(\theta_1, \ldots, \theta_K)$ at `theta[K]` for a Dirichlet distribution with parameters `alpha[K]`, using the formula given above.

double **gsl_ran_dirichlet_lnpdf**(size_t K, const double alpha[], const double theta[])

This function computes the logarithm of the probability density $p(\theta_1, \ldots, \theta_K)$ for a Dirichlet distribution with parameters `alpha[K]`.

20.29 General Discrete Distributions

Given K discrete events with different probabilities $P[k]$, produce a random value k consistent with its probability.

The obvious way to do this is to preprocess the probability list by generating a cumulative probability array with $K+1$ elements:

$$C[0] = 0$$
$$C[k+1] = C[k] + P[k]$$

Note that this construction produces $C[K] = 1$. Now choose a uniform deviate u between 0 and 1, and find the value of k such that $C[k] \leq u < C[k+1]$. Although this in principle requires of order $\log K$ steps per random number generation, they are fast steps, and if you use something like $\lfloor uK \rfloor$ as a starting point, you can often do pretty well.

But faster methods have been devised. Again, the idea is to preprocess the probability list, and save the result in some form of lookup table; then the individual calls for a random discrete event can go rapidly. An approach invented by G. Marsaglia (Generating discrete random variables in a computer, Comm ACM 6, 37–38 (1963)) is very clever, and readers interested in examples of good algorithm design are directed to this short and well-written paper. Unfortunately, for large K, Marsaglia's lookup table can be quite large.

A much better approach is due to Alastair J. Walker (An efficient method for generating discrete random variables with general distributions, ACM Trans on Mathematical Software 3, 253–256 (1977); see also Knuth, v2, 3rd ed, p120–121,139). This requires two lookup tables, one floating point and one integer, but both only of size K. After preprocessing, the random numbers are generated in O(1) time, even for large K. The preprocessing suggested by Walker requires $O(K^2)$ effort, but that is not actually necessary, and the implementation provided here only takes $O(K)$ effort. In general, more preprocessing leads to faster generation of the individual random numbers, but a diminishing return is reached pretty early. Knuth points out that the optimal preprocessing is combinatorially difficult for large K.

This method can be used to speed up some of the discrete random number generators below, such as the binomial distribution. To use it for something like the Poisson Distribution, a modification would have to be made, since it only takes a finite set of K outcomes.

type **gsl_ran_discrete_t**

This structure contains the lookup table for the discrete random number generator.

gsl_ran_discrete_t *__gsl_ran_discrete_preproc__(size_t K, const double *P)

This function returns a pointer to a structure that contains the lookup table for the discrete random number generator. The array `P` contains the probabilities of the discrete

events; these array elements must all be positive, but they needn't add up to one (so you can think of them more generally as "weights")—the preprocessor will normalize appropriately. This return value is used as an argument for the `gsl_ran_discrete()` function below.

size_t **gsl_ran_discrete**(const gsl_rng *r, const gsl_ran_discrete_t *g)

After the preprocessor, above, has been called, you use this function to get the discrete random numbers.

double **gsl_ran_discrete_pdf**(size_t k, const gsl_ran_discrete_t *g)

Returns the probability $P[k]$ of observing the variable `k`. Since $P[k]$ is not stored as part of the lookup table, it must be recomputed; this computation takes $O(K)$, so if `K` is large and you care about the original array $P[k]$ used to create the lookup table, then you should just keep this original array $P[k]$ around.

void **gsl_ran_discrete_free**(gsl_ran_discrete_t *g)

De-allocates the lookup table pointed to by `g`.

20.30 The Poisson Distribution

unsigned int **gsl_ran_poisson**(const gsl_rng *r, double mu)

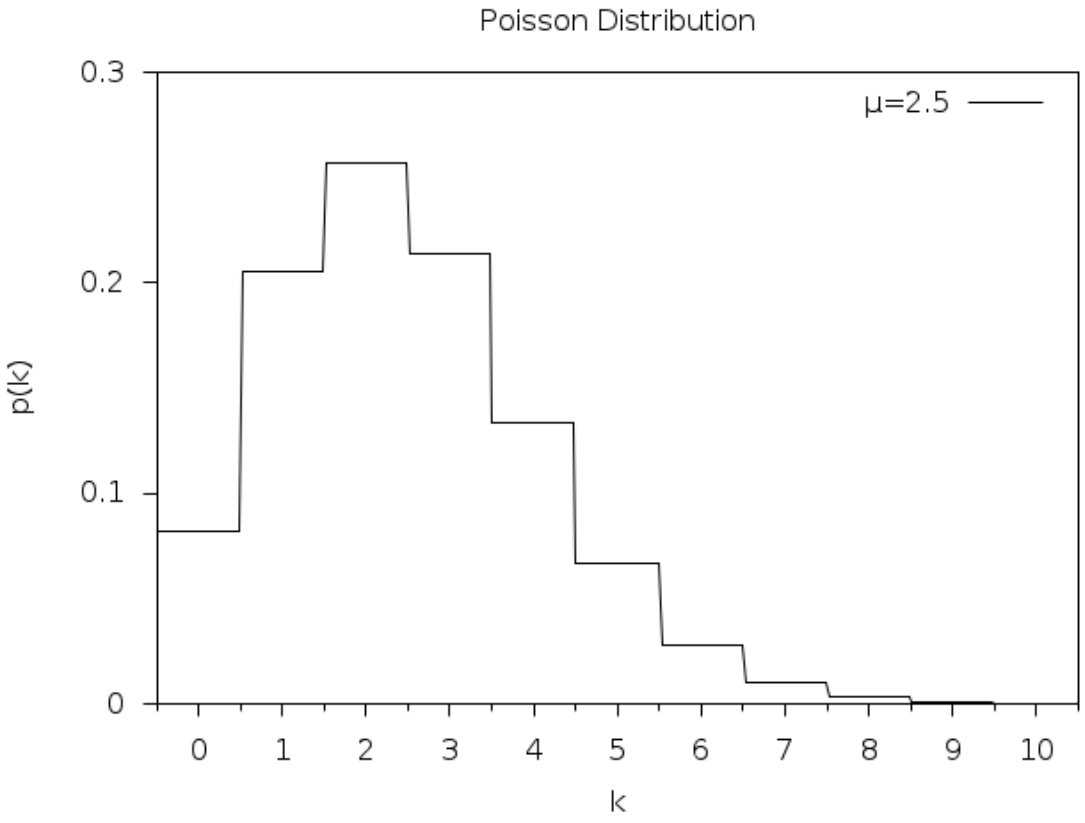
This function returns a random integer from the Poisson distribution with mean `mu`. The probability distribution for Poisson variates is,

$$p(k) = \frac{\mu^k}{k!} \exp(-\mu)$$

for $k \geq 0$.

double **gsl_ran_poisson_pdf**(unsigned int k, double mu)

This function computes the probability $p(k)$ of obtaining `k` from a Poisson distribution with mean `mu`, using the formula given above.



double **gsl_cdf_poisson_P**(unsigned int k, double mu)

double **gsl_cdf_poisson_Q**(unsigned int k, double mu)

These functions compute the cumulative distribution functions $P(k)$, $Q(k)$ for the Poisson distribution with parameter `mu`.

20.31 The Bernoulli Distribution

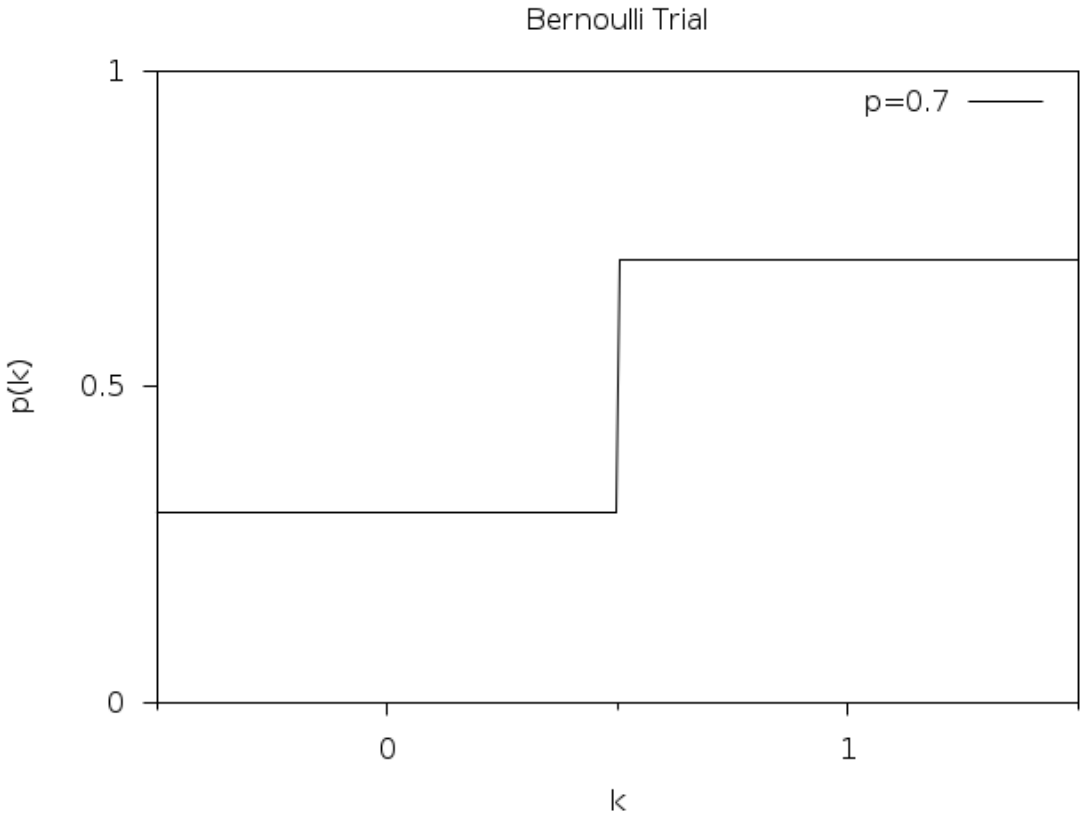
unsigned int **gsl_ran_bernoulli**(const gsl_rng *r, double p)

This function returns either 0 or 1, the result of a Bernoulli trial with probability `p`. The probability distribution for a Bernoulli trial is,

$$p(0) = 1 - p$$
$$p(1) = p$$

double **gsl_ran_bernoulli_pdf**(unsigned int k, double p)

This function computes the probability $p(k)$ of obtaining `k` from a Bernoulli distribution with probability parameter `p`, using the formula given above.



20.32 The Binomial Distribution

unsigned int **gsl_ran_binomial**(const gsl_rng *r, double p, unsigned int n)

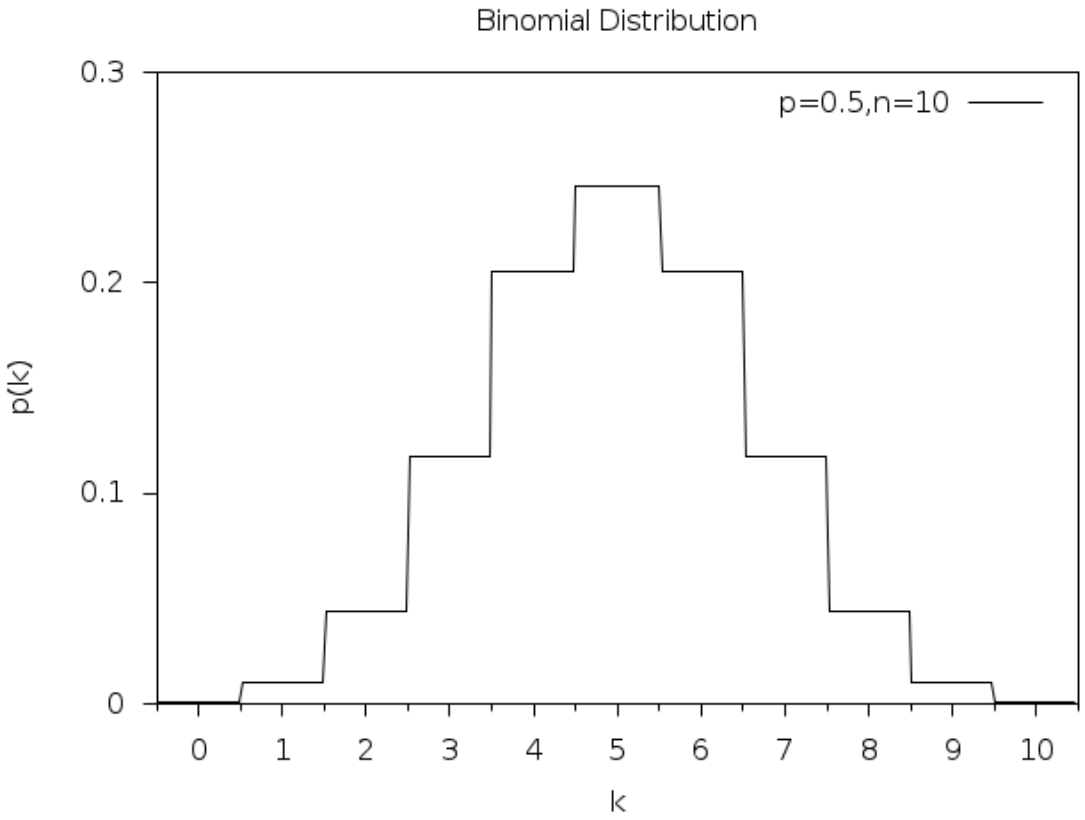
This function returns a random integer from the binomial distribution, the number of successes in `n` independent trials with probability `p`. The probability distribution for binomial variates is,

$$p(k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

for $0 \leq k \leq n$.

double **gsl_ran_binomial_pdf**(unsigned int k, double p, unsigned int n)

This function computes the probability $p(k)$ of obtaining `k` from a binomial distribution with parameters `p` and `n`, using the formula given above.



double **gsl_cdf_binomial_P**(unsigned int k, double p, unsigned int n)

double **gsl_cdf_binomial_Q**(unsigned int k, double p, unsigned int n)

These functions compute the cumulative distribution functions $P(k)$, $Q(k)$ for the binomial distribution with parameters `p` and `n`.

20.33 The Multinomial Distribution

void **gsl_ran_multinomial**(const gsl_rng *r, size_t K, unsigned int N, const double p[], unsigned int n[])

This function computes a random sample `n` from the multinomial distribution formed by `N` trials from an underlying distribution `p[K]`. The distribution function for `n` is,

$$P(n_1, n_2, \cdots, n_K) = \frac{N!}{n_1! n_2! \cdots n_K!} p_1^{n_1} p_2^{n_2} \cdots p_K^{n_K}$$

where $(n_1, n_2, \ldots, n_K)$ are nonnegative integers with $\sum_{k=1}^{K} n_k = N$, and $(p_1, p_2, \ldots, p_K)$ is a probability distribution with $\sum p_i = 1$. If the array `p[K]` is not normalized then its entries will be treated as weights and normalized appropriately. The arrays `n` and `p` must both be of length `K`.

Random variates are generated using the conditional binomial method (see C.S. Davis, The computer generation of multinomial random variates, Comp. Stat. Data Anal. 16 (1993) 205–217 for details).

double **gsl_ran_multinomial_pdf**(size_t K, const double p[], const unsigned int n[])

This function computes the probability $P(n_1, n_2, \ldots, n_K)$ of sampling `n[K]` from a multinomial distribution with parameters `p[K]`, using the formula given above.

double **gsl_ran_multinomial_lnpdf**(size_t K, const double p[], const unsigned int n[])

This function returns the logarithm of the probability for the multinomial distribution $P(n_1, n_2, \ldots, n_K)$ with parameters `p[K]`.

20.34 The Negative Binomial Distribution

unsigned int **gsl_ran_negative_binomial**(const gsl_rng *r, double p, double n)

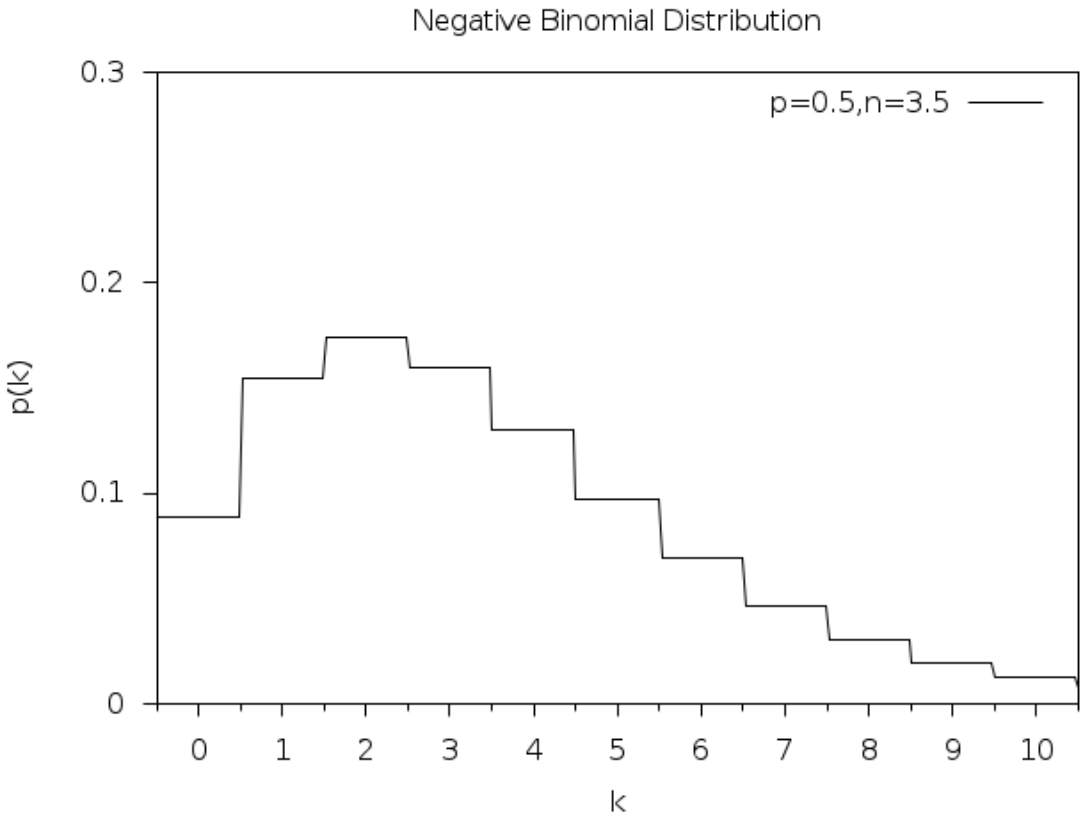
This function returns a random integer from the negative binomial distribution, the number of failures occurring before `n` successes in independent trials with probability `p` of success. The probability distribution for negative binomial variates is,

$$p(k) = \frac{\Gamma(n+k)}{\Gamma(k+1)\Gamma(n)} p^n (1-p)^k$$

Note that n is not required to be an integer.

double **gsl_ran_negative_binomial_pdf**(unsigned int k, double p, double n)

This function computes the probability $p(k)$ of obtaining `k` from a negative binomial distribution with parameters `p` and `n`, using the formula given above.



double **gsl_cdf_negative_binomial_P**(unsigned int k, double p, double n)

double **gsl_cdf_negative_binomial_Q**(unsigned int k, double p, double n)

These functions compute the cumulative distribution functions $P(k)$, $Q(k)$ for the negative binomial distribution with parameters `p` and `n`.

20.35 The Pascal Distribution

unsigned int **gsl_ran_pascal**(const gsl_rng *r, double p, unsigned int n)

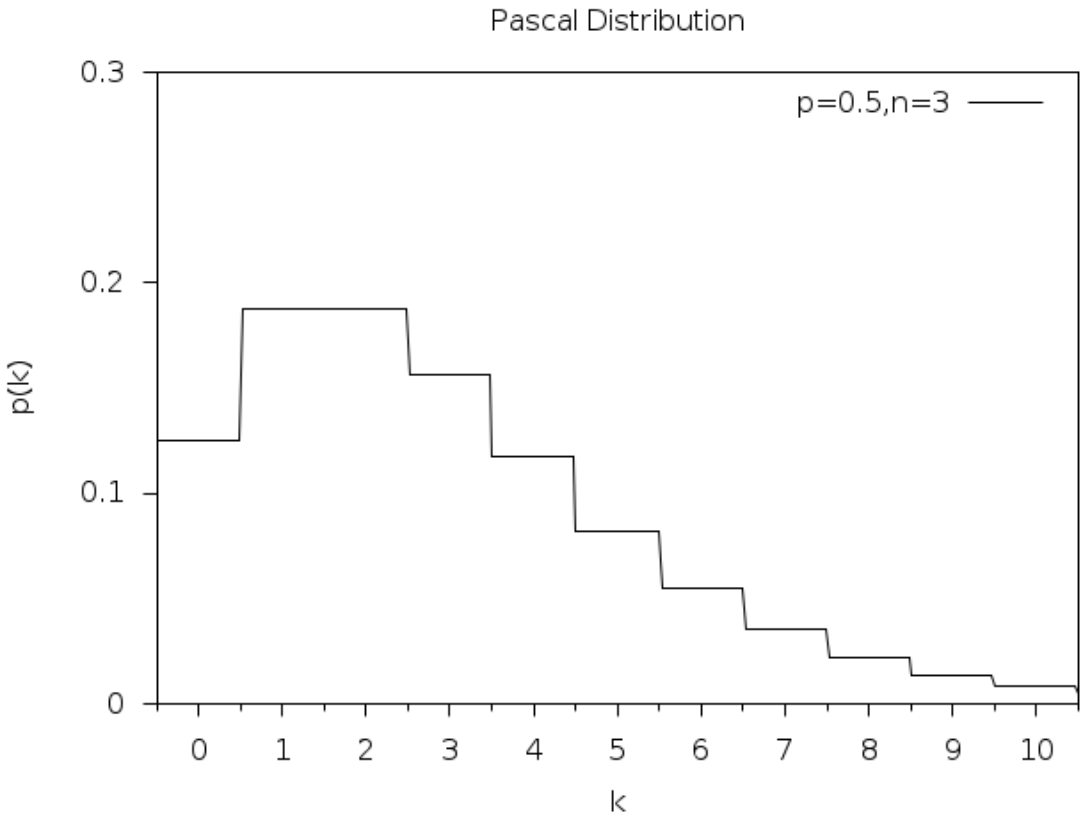
This function returns a random integer from the Pascal distribution. The Pascal distribution is simply a negative binomial distribution with an integer value of n.

$$p(k) = \frac{(n+k-1)!}{k!(n-1)!} p^n (1-p)^k$$

for $k \geq 0$.

double **gsl_ran_pascal_pdf**(unsigned int k, double p, unsigned int n)

This function computes the probability $p(k)$ of obtaining `k` from a Pascal distribution with parameters `p` and `n`, using the formula given above.



double **gsl_cdf_pascal_P**(unsigned int k, double p, unsigned int n)

double **gsl_cdf_pascal_Q**(unsigned int k, double p, unsigned int n)

These functions compute the cumulative distribution functions $P(k)$, $Q(k)$ for the Pascal distribution with parameters `p` and `n`.

20.36 The Geometric Distribution

unsigned int **gsl_ran_geometric**(const gsl_rng *r, double p)

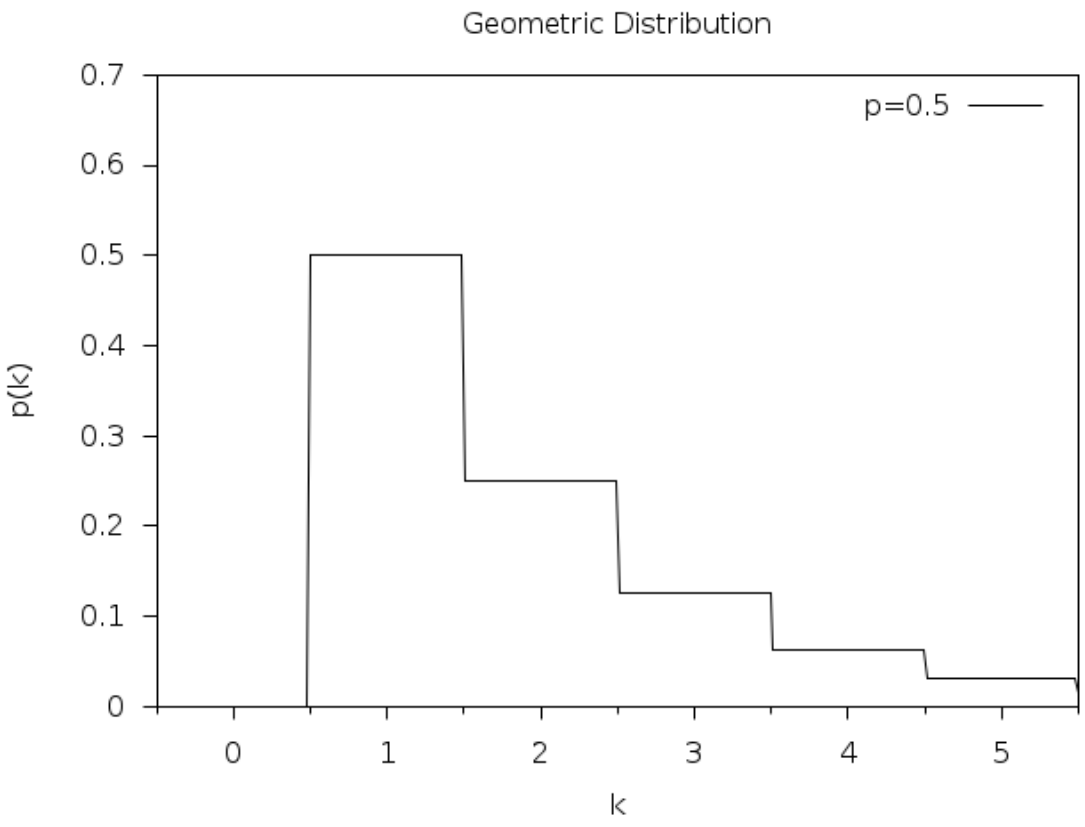
This function returns a random integer from the geometric distribution, the number of independent trials with probability `p` until the first success. The probability distribution for geometric variates is,

$$p(k) = p(1-p)^{k-1}$$

for $k \geq 1$. Note that the distribution begins with $k = 1$ with this definition. There is another convention in which the exponent $k-1$ is replaced by k.

double **gsl_ran_geometric_pdf**(unsigned int k, double p)

This function computes the probability $p(k)$ of obtaining `k` from a geometric distribution with probability parameter `p`, using the formula given above.



double **gsl_cdf_geometric_P**(unsigned int k, double p)

double **gsl_cdf_geometric_Q**(unsigned int k, double p)

These functions compute the cumulative distribution functions $P(k)$, $Q(k)$ for the geometric distribution with parameter `p`.

20.37 The Hypergeometric Distribution

unsigned int **gsl_ran_hypergeometric**(const gsl_rng *r, unsigned int n1, unsigned int n2, unsigned int t)

This function returns a random integer from the hypergeometric distribution. The probability distribution for hypergeometric random variates is,

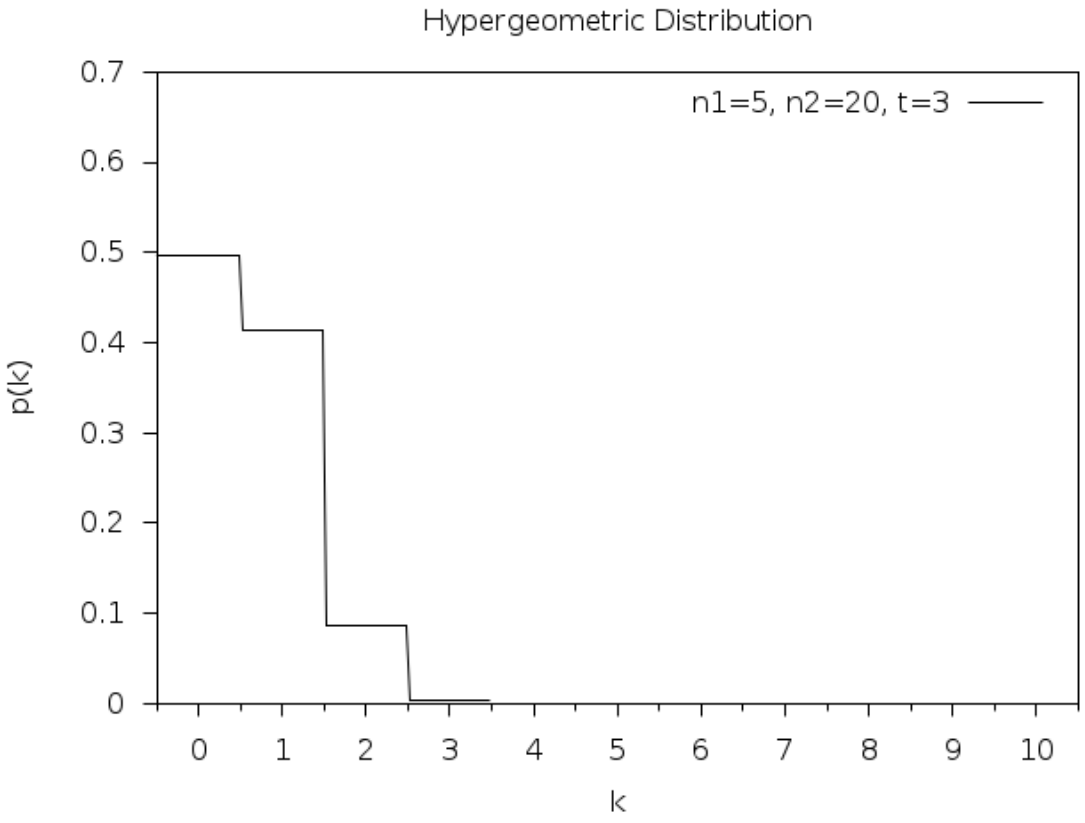
$$p(k) = C(n_1, k)C(n_2, t-k)/C(n_1+n_2, t)$$

where $C(a,b) = a!/(b!(a-b)!)$ and $t \leq n_1+n_2$. The domain of k is $\max(0, t-n_2), \ldots, \min(t, n_1)$

If a population contains n_1 elements of "type 1" and n_2 elements of "type 2" then the hypergeometric distribution gives the probability of obtaining k elements of "type 1" in t samples from the population without replacement.

double **gsl_ran_hypergeometric_pdf**(unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

This function computes the probability $p(k)$ of obtaining `k` from a hypergeometric distribution with parameters `n1`, `n2`, `t`, using the formula given above.



double **gsl_cdf_hypergeometric_P**(unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

double **gsl_cdf_hypergeometric_Q**(unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

These functions compute the cumulative distribution functions $P(k)$, $Q(k)$ for the hypergeometric distribution with parameters `n1`, `n2` and `t`.

20.38 The Logarithmic Distribution

unsigned int **gsl_ran_logarithmic**(const gsl_rng *r, double p)

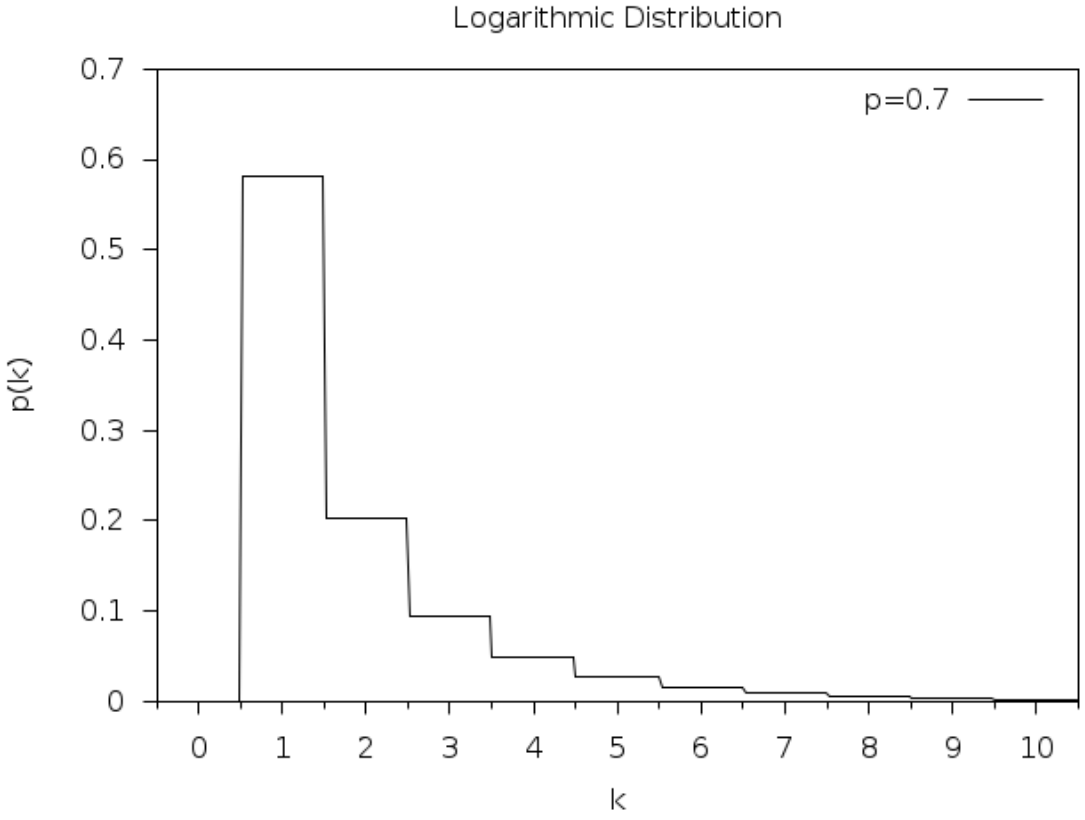
This function returns a random integer from the logarithmic distribution. The probability distribution for logarithmic random variates is,

$$p(k) = \frac{-1}{\log(1-p)} \left(\frac{p^k}{k} \right)$$

for $k \geq 1$.

double **gsl_ran_logarithmic_pdf**(unsigned int k, double p)

This function computes the probability $p(k)$ of obtaining `k` from a logarithmic distribution with probability parameter `p`, using the formula given above.



20.39 The Wishart Distribution

int **gsl_ran_wishart**(const gsl_rng *r, const double n, const gsl_matrix *L, gsl_matrix *result, gsl_matrix *work)

This function computes a random symmetric p-by-p matrix from the Wishart distribution. The probability distribution for Wishart random variates is,

$$p(X) = \frac{|X|^{(n-p-1)/2} e^{-\mathrm{tr}\left(V^{-1}X\right)/2}}{2^{\frac{np}{2}} |V|^{n/2} \Gamma_p(\frac{n}{2})}$$

Here, $n > p - 1$ is the number of degrees of freedom, V is a symmetric positive definite p-by-p scale matrix, whose Cholesky factor is specified by `L`, and `work` is p-by-p workspace. The p-by-p Wishart distributed matrix X is stored in `result` on output.

int **gsl_ran_wishart_pdf**(const gsl_matrix *X, const gsl_matrix *L_X, const double n, const gsl_matrix *L, double *result, gsl_matrix *work)

int **gsl_ran_wishart_log_pdf**(const gsl_matrix *X, const gsl_matrix *L_X, const double n, const gsl_matrix *L, double *result, gsl_matrix *work)

These functions compute $p(X)$ or $\log p(X)$ for the p-by-p matrix `X`, whose Cholesky factor is specified in `L_X`. The degrees of freedom is given by `n`, the Cholesky factor of the scale matrix V is specified in `L`, and `work` is p-by-p workspace. The probably density value is returned in `result`.

20.40 Shuffling and Sampling

The following functions allow the shuffling and sampling of a set of objects. The algorithms rely on a random number generator as a source of randomness and a poor quality generator can lead to correlations in the output. In particular it is important to avoid generators with a short period. For more information see Knuth, v2, 3rd ed, Section 3.4.2, "Random Sampling and Shuffling".

void **gsl_ran_shuffle**(const gsl_rng *r, void *base, size_t n, size_t size)

This function randomly shuffles the order of `n` objects, each of size `size`, stored in the array `base[0..n-1]`. The output of the random number generator `r` is used to produce the permutation. The algorithm generates all possible $n!$ permutations with equal probability, assuming a perfect source of random numbers.

The following code shows how to shuffle the numbers from 0 to 51:

```
int a[52];

for (i = 0; i < 52; i++)
  {
    a[i] = i;
  }

gsl_ran_shuffle (r, a, 52, sizeof (int));
```

int **gsl_ran_choose**(const gsl_rng *r, void *dest, size_t k, void *src, size_t n, size_t size)

This function fills the array `dest[k]` with `k` objects taken randomly from the `n` elements of the array `src[0..n-1]`. The objects are each of size `size`. The output of the random number generator `r` is used to make the selection. The algorithm ensures all possible samples are equally likely, assuming a perfect source of randomness.

The objects are sampled **without** replacement, thus each object can only appear once in `dest`. It is required that `k` be less than or equal to `n`. The objects in `dest` will be in the same relative order as those in `src`. You will need to call `gsl_ran_shuffle(r, dest, n, size)` if you want to randomize the order.

The following code shows how to select a random sample of three unique numbers from the set 0 to 99:

```
double a[3], b[100];

for (i = 0; i < 100; i++)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  {
    b[i] = (double) i;
  }

gsl_ran_choose (r, a, 3, b, 100, sizeof (double));
```

void **gsl_ran_sample**(const gsl_rng *r, void *dest, size_t k, void *src, size_t n, size_t size)

This function is like `gsl_ran_choose()` but samples `k` items from the original array of `n` items `src` with replacement, so the same object can appear more than once in the output sequence `dest`. There is no requirement that `k` be less than `n` in this case.

20.41 Examples

The following program demonstrates the use of a random number generator to produce variates from a distribution. It prints 10 samples from the Poisson distribution with a mean of 3.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;

  int i, n = 10;
  double mu = 3.0;

  /* create a generator chosen by the
     environment variable GSL_RNG_TYPE */

  gsl_rng_env_setup();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  /* print n random variates chosen from
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
     the poisson distribution with mean
     parameter mu */

  for (i = 0; i < n; i++)
    {
      unsigned int k = gsl_ran_poisson (r, mu);
      printf (" %u", k);
    }

  printf ("\n");
  gsl_rng_free (r);
  return 0;
}
```

If the library and header files are installed under `/usr/local` (the default location) then the program can be compiled with these options:

```
$ gcc -Wall demo.c -lgsl -lgslcblas -lm
```

Here is the output of the program,

```
 2 5 5 2 1 0 3 4 1 1
```

The variates depend on the seed used by the generator. The seed for the default generator type `gsl_rng_default` can be changed with the `GSL_RNG_SEED` environment variable to produce a different stream of variates:

```
$ GSL_RNG_SEED=123 ./a.out
```

giving output

```
 4 5 6 3 3 1 4 2 5 5
```

The following program generates a random walk in two dimensions.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

int
main (void)
{
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  int i;
  double x = 0, y = 0, dx, dy;

  const gsl_rng_type * T;
  gsl_rng * r;

  gsl_rng_env_setup();
  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  printf ("%g %g\n", x, y);

  for (i = 0; i < 10; i++)
    {
      gsl_ran_dir_2d (r, &dx, &dy);
      x += dx; y += dy;
      printf ("%g %g\n", x, y);
    }

  gsl_rng_free (r);
  return 0;
}
```

그림 20.1 shows the output from the program.

The following program computes the upper and lower cumulative distribution functions for the standard normal distribution at $x = 2$.

```
#include <stdio.h>
#include <gsl/gsl_cdf.h>

int
main (void)
{
  double P, Q;
  double x = 2.0;

  P = gsl_cdf_ugaussian_P (x);
  printf ("prob(x < %f) = %f\n", x, P);

  Q = gsl_cdf_ugaussian_Q (x);
```

(다음 페이지에 계속)

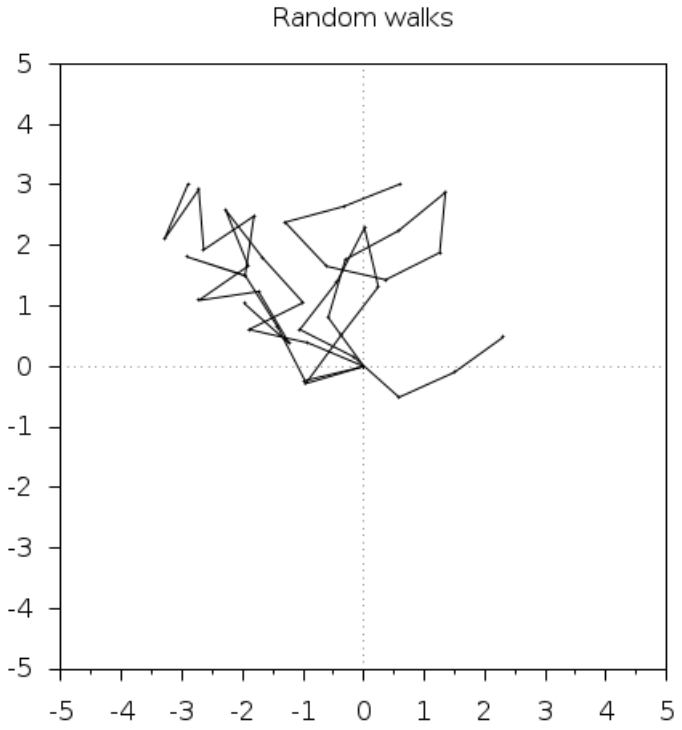


그림 20.1: Four 10-step random walks from the origin.

(이전 페이지에서 계속)

```
  printf ("prob(x > %f) = %f\n", x, Q);

  x = gsl_cdf_ugaussian_Pinv (P);
  printf ("Pinv(%f) = %f\n", P, x);

  x = gsl_cdf_ugaussian_Qinv (Q);
  printf ("Qinv(%f) = %f\n", Q, x);

  return 0;
}
```

Here is the output of the program,

```
prob(x < 2.000000) = 0.977250
prob(x > 2.000000) = 0.022750
Pinv(0.977250) = 2.000000
Qinv(0.022750) = 2.000000
```

20.42 References and Further Reading

For an encyclopaedic coverage of the subject readers are advised to consult the book "Non-Uniform Random Variate Generation" by Luc Devroye. It covers every imaginable distribution and provides hundreds of algorithms.

- Luc Devroye, "Non-Uniform Random Variate Generation", Springer-Verlag, ISBN 0-387-96305-7. Available online at http://cg.scs.carleton.ca/~luc/rnbookindex.html.

The subject of random variate generation is also reviewed by Knuth, who describes algorithms for all the major distributions.

- Donald E. Knuth, "The Art of Computer Programming: Seminumerical Algorithms" (Vol 2, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896842.

The Particle Data Group provides a short review of techniques for generating distributions of random numbers in the "Monte Carlo" section of its Annual Review of Particle Physics.

- Review of Particle Properties, R.M. Barnett et al., Physical Review D54, 1 (1996) http://pdg.lbl.gov/.

The Review of Particle Physics is available online in postscript and pdf format.

An overview of methods used to compute cumulative distribution functions can be found in Statistical Computing by W.J. Kennedy and J.E. Gentle. Another general reference is Elements of Statistical Computing by R.A. Thisted.

- William E. Kennedy and James E. Gentle, Statistical Computing (1980), Marcel Dekker, ISBN 0-8247-6898-1.
- Ronald A. Thisted, Elements of Statistical Computing (1988), Chapman & Hall, ISBN 0-412-01371-1.

The cumulative distribution functions for the Gaussian distribution are based on the following papers,

- Rational Chebyshev Approximations Using Linear Equations, W.J. Cody, W. Fraser, J.F. Hart. Numerische Mathematik 12, 242–251 (1968).
- Rational Chebyshev Approximations for the Error Function, W.J. Cody. Mathematics of Computation 23, n107, 631–637 (July 1969).

제 21 장

통계

참고: 번역중

이 단원은 라이브러리 내의 통계 함수들에 대해 기술합니다. 기본 통계 함수들은 자료의 평균, 분산, 그리고 표준 편차를 계산하는 기능들을 포함하고 있습니다. 좀 더 심화적인 함수들에는 절대 편차, 비대칭도, 그리고 첨도 의 계산 함수들이 구현되어 있고, 중앙값과 임의의 백분위수 계산 기능도 포함하고 있습니다. 사용된 알고리즘들은 재귀 관계를 이용해 평균 값을 계산합니다. 이 방법들은 안정적인 방법을 사용해 오버플로우를 야기할 수 있는 큰 중간 단계의 값을 사용하지 않습니다.

The functions are available in versions for datasets in the standard floating-point and integer types. The versions for double precision floating-point data have the prefix `gsl_stats` and are declared in the header file `gsl_statistics_double.h`. The versions for integer data have the prefix `gsl_stats_int` and are declared in the header file `gsl_statistics_int.h`. All the functions operate on C arrays with a `stride` parameter specifying the spacing between elements.

21.1 평균, 표준 편차, 분산

double **gsl_stats_mean**(const double data[], size_t stride, size_t n)

This function returns the arithmetic mean of `data`, a dataset of length `n` with stride `stride`. The arithmetic mean, or sample mean, is denoted by $\hat{\mu}$ and defined as,

$$\hat{\mu} = \frac{1}{N} \sum x_i$$

where x_i are the elements of the dataset `data`. For samples drawn from a gaussian distribution the variance of $\hat{\mu}$ is σ^2/N.

double **gsl_stats_variance**(const double data[], size_t stride, size_t n)

This function returns the estimated, or sample, variance of `data`, a dataset of length `n` with stride `stride`. The estimated variance is denoted by $\hat{\sigma}^2$ and is defined by,

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - \hat{\mu})^2$$

where x_i are the elements of the dataset `data`. Note that the normalization factor of $1/(N-1)$ results from the derivation of $\hat{\sigma}^2$ as an unbiased estimator of the population variance σ^2. For samples drawn from a Gaussian distribution the variance of $\hat{\sigma}^2$ itself is $2\sigma^4/N$.

This function computes the mean via a call to `gsl_stats_mean()`. If you have already computed the mean then you can pass it directly to `gsl_stats_variance_m()`.

double **gsl_stats_variance_m**(const double data[], size_t stride, size_t n, double mean)

This function returns the sample variance of `data` relative to the given value of `mean`. The function is computed with $\hat{\mu}$ replaced by the value of `mean` that you supply,

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - mean)^2$$

double **gsl_stats_sd**(const double data[], size_t stride, size_t n)

double **gsl_stats_sd_m**(const double data[], size_t stride, size_t n, double mean)

The standard deviation is defined as the square root of the variance. These functions return the square root of the corresponding variance functions above.

double **gsl_stats_tss**(const double data[], size_t stride, size_t n)

double **gsl_stats_tss_m**(const double data[], size_t stride, size_t n, double mean)

These functions return the total sum of squares (TSS) of `data` about the mean. For `gsl_stats_tss_m()` the user-supplied value of `mean` is used, and for `gsl_stats_tss()` it is computed using `gsl_stats_mean()`.

$$\mathrm{TSS} = \sum (x_i - mean)^2$$

double **gsl_stats_variance_with_fixed_mean**(const double data[], size_t stride, size_t n, double mean)

This function computes an unbiased estimate of the variance of `data` when the population mean `mean` of the underlying distribution is known a priori. In this case the estimator for the variance uses the factor $1/N$ and the sample mean $\hat{\mu}$ is replaced by the known

population mean μ,

$$\hat{\sigma}^2 = \frac{1}{N}\sum (x_i - \mu)^2$$

double **gsl_stats_sd_with_fixed_mean**(const double data[], size_t stride, size_t n, double mean)

This function calculates the standard deviation of `data` for a fixed population mean `mean`. The result is the square root of the corresponding variance function.

21.2 Absolute deviation

double **gsl_stats_absdev**(const double data[], size_t stride, size_t n)

This function computes the absolute deviation from the mean of `data`, a dataset of length `n` with stride `stride`. The absolute deviation from the mean is defined as,

$$absdev = \frac{1}{N}\sum |x_i - \hat{\mu}|$$

where x_i are the elements of the dataset `data`. The absolute deviation from the mean provides a more robust measure of the width of a distribution than the variance. This function computes the mean of `data` via a call to `gsl_stats_mean()`.

double **gsl_stats_absdev_m**(const double data[], size_t stride, size_t n, double mean)

This function computes the absolute deviation of the dataset `data` relative to the given value of `mean`,

$$absdev = \frac{1}{N}\sum |x_i - mean|$$

This function is useful if you have already computed the mean of `data` (and want to avoid recomputing it), or wish to calculate the absolute deviation relative to another value (such as zero, or the median).

21.3 Higher moments (skewness and kurtosis)

double **gsl_stats_skew**(const double data[], size_t stride, size_t n)

This function computes the skewness of `data`, a dataset of length `n` with stride `stride`. The skewness is defined as,

$$skew = \frac{1}{N}\sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}}\right)^3$$

where x_i are the elements of the dataset `data`. The skewness measures the asymmetry of the tails of a distribution.

The function computes the mean and estimated standard deviation of `data` via calls to `gsl_stats_mean()` and `gsl_stats_sd()`.

double **gsl_stats_skew_m_sd**(const double data[], size_t stride, size_t n, double mean, double sd)

This function computes the skewness of the dataset `data` using the given values of the mean `mean` and standard deviation `sd`,

$$skew = \frac{1}{N} \sum \left(\frac{x_i - mean}{sd} \right)^3$$

These functions are useful if you have already computed the mean and standard deviation of `data` and want to avoid recomputing them.

double **gsl_stats_kurtosis**(const double data[], size_t stride, size_t n)

This function computes the kurtosis of `data`, a dataset of length `n` with stride `stride`. The kurtosis is defined as,

$$kurtosis = \left(\frac{1}{N} \sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^4 \right) - 3$$

The kurtosis measures how sharply peaked a distribution is, relative to its width. The kurtosis is normalized to zero for a Gaussian distribution.

double **gsl_stats_kurtosis_m_sd**(const double data[], size_t stride, size_t n, double mean, double sd)

This function computes the kurtosis of the dataset `data` using the given values of the mean `mean` and standard deviation `sd`,

$$kurtosis = \frac{1}{N} \left(\sum \left(\frac{x_i - mean}{sd} \right)^4 \right) - 3$$

This function is useful if you have already computed the mean and standard deviation of `data` and want to avoid recomputing them.

21.4 Autocorrelation

double **gsl_stats_lag1_autocorrelation**(const double data[], const size_t stride, const size_t n)

This function computes the lag-1 autocorrelation of the dataset `data`.

$$a_1 = \frac{\sum_{i=2}^{n}(x_i - \hat{\mu})(x_{i-1} - \hat{\mu})}{\sum_{i=1}^{n}(x_i - \hat{\mu})(x_i - \hat{\mu})}$$

double **gsl_stats_lag1_autocorrelation_m**(const double data[], const size_t stride, const size_t n, const double mean)

This function computes the lag-1 autocorrelation of the dataset `data` using the given value of the mean `mean`.

21.5 Covariance

double **gsl_stats_covariance**(const double data1[], const size_t stride1, const double data2[], const size_t stride2, const size_t n)

This function computes the covariance of the datasets `data1` and `data2` which must both be of the same length `n`.

$$covar = \frac{1}{(n-1)}\sum_{i=1}^{n}(x_i - \hat{x})(y_i - \hat{y})$$

double **gsl_stats_covariance_m**(const double data1[], const size_t stride1, const double data2[], const size_t stride2, const size_t n, const double mean1, const double mean2)

This function computes the covariance of the datasets `data1` and `data2` using the given values of the means, `mean1` and `mean2`. This is useful if you have already computed the means of `data1` and `data2` and want to avoid recomputing them.

21.6 Correlation

double **gsl_stats_correlation**(const double data1[], const size_t stride1, const double data2[], const size_t stride2, const size_t n)

This function efficiently computes the Pearson correlation coefficient between the

datasets `data1` and `data2` which must both be of the same length `n`.

$$r = \frac{cov(x,y)}{\hat{\sigma}_x \hat{\sigma}_y} = \frac{\frac{1}{n-1}\sum(x_i - \hat{x})(y_i - \hat{y})}{\sqrt{\frac{1}{n-1}\sum(x_i - \hat{x})^2}\sqrt{\frac{1}{n-1}\sum(y_i - \hat{y})^2}}$$

double **gsl_stats_spearman**(const double data1[], const size_t stride1, const double data2[], const size_t stride2, const size_t n, double work[])

This function computes the Spearman rank correlation coefficient between the datasets `data1` and `data2` which must both be of the same length `n`. Additional workspace of size 2 * `n` is required in `work`. The Spearman rank correlation between vectors x and y is equivalent to the Pearson correlation between the ranked vectors x_R and y_R, where ranks are defined to be the average of the positions of an element in the ascending order of the values.

21.7 Weighted Samples

The functions described in this section allow the computation of statistics for weighted samples. The functions accept an array of samples, x_i, with associated weights, w_i. Each sample x_i is considered as having been drawn from a Gaussian distribution with variance σ_i^2. The sample weight w_i is defined as the reciprocal of this variance, $w_i = 1/\sigma_i^2$. Setting a weight to zero corresponds to removing a sample from a dataset.

double **gsl_stats_wmean**(const double w[], size_t wstride, const double data[], size_t stride, size_t n)

This function returns the weighted mean of the dataset `data` with stride `stride` and length `n`, using the set of weights `w` with stride `wstride` and length `n`. The weighted mean is defined as,

$$\hat{\mu} = \frac{\sum w_i x_i}{\sum w_i}$$

double **gsl_stats_wvariance**(const double w[], size_t wstride, const double data[], size_t stride, size_t n)

This function returns the estimated variance of the dataset `data` with stride `stride` and length `n`, using the set of weights `w` with stride `wstride` and length `n`. The estimated variance of a weighted dataset is calculated as,

$$\hat{\sigma}^2 = \frac{\sum w_i}{(\sum w_i)^2 - \sum(w_i^2)} \sum w_i (x_i - \hat{\mu})^2$$

Note that this expression reduces to an unweighted variance with the familiar $1/(N-1)$

factor when there are N equal non-zero weights.

double **gsl_stats_wvariance_m**(const double w[], size_t wstride, const double data[], size_t stride, size_t n, double wmean)

This function returns the estimated variance of the weighted dataset `data` using the given weighted mean `wmean`.

double **gsl_stats_wsd**(const double w[], size_t wstride, const double data[], size_t stride, size_t n)

The standard deviation is defined as the square root of the variance. This function returns the square root of the corresponding variance function `gsl_stats_wvariance()` above.

double **gsl_stats_wsd_m**(const double w[], size_t wstride, const double data[], size_t stride, size_t n, double wmean)

This function returns the square root of the corresponding variance function `gsl_stats_wvariance_m()` above.

double **gsl_stats_wvariance_with_fixed_mean**(const double w[], size_t wstride, const double data[], size_t stride, size_t n, const double mean)

This function computes an unbiased estimate of the variance of the weighted dataset `data` when the population mean `mean` of the underlying distribution is known a priori. In this case the estimator for the variance replaces the sample mean $\hat{\mu}$ by the known population mean μ,

$$\hat{\sigma}^2 = \frac{\sum w_i (x_i - \mu)^2}{\sum w_i}$$

double **gsl_stats_wsd_with_fixed_mean**(const double w[], size_t wstride, const double data[], size_t stride, size_t n, const double mean)

The standard deviation is defined as the square root of the variance. This function returns the square root of the corresponding variance function above.

double **gsl_stats_wtss**(const double w[], const size_t wstride, const double data[], size_t stride, size_t n)

double **gsl_stats_wtss_m**(const double w[], const size_t wstride, const double data[], size_t stride, size_t n, double wmean)

These functions return the weighted total sum of squares (TSS) of `data` about the weighted mean. For `gsl_stats_wtss_m()` the user-supplied value of `wmean` is used, and for `gsl_stats_wtss()` it is computed using `gsl_stats_wmean()`.

$$\mathrm{TSS} = \sum w_i (x_i - wmean)^2$$

double **gsl_stats_wabsdev**(const double w[], size_t wstride, const double data[], size_t stride, size_t n)

This function computes the weighted absolute deviation from the weighted mean of `data`. The absolute deviation from the mean is defined as,

$$absdev = \frac{\sum w_i |x_i - \hat{\mu}|}{\sum w_i}$$

double **gsl_stats_wabsdev_m**(const double w[], size_t wstride, const double data[], size_t stride, size_t n, double wmean)

This function computes the absolute deviation of the weighted dataset `data` about the given weighted mean `wmean`.

double **gsl_stats_wskew**(const double w[], size_t wstride, const double data[], size_t stride, size_t n)

This function computes the weighted skewness of the dataset `data`.

$$skew = \frac{\sum w_i ((x_i - \hat{x})/\hat{\sigma})^3}{\sum w_i}$$

double **gsl_stats_wskew_m_sd**(const double w[], size_t wstride, const double data[], size_t stride, size_t n, double wmean, double wsd)

This function computes the weighted skewness of the dataset `data` using the given values of the weighted mean and weighted standard deviation, `wmean` and `wsd`.

double **gsl_stats_wkurtosis**(const double w[], size_t wstride, const double data[], size_t stride, size_t n)

This function computes the weighted kurtosis of the dataset `data`.

$$kurtosis = \frac{\sum w_i ((x_i - \hat{x})/\hat{\sigma})^4}{\sum w_i} - 3$$

double **gsl_stats_wkurtosis_m_sd**(const double w[], size_t wstride, const double data[], size_t stride, size_t n, double wmean, double wsd)

This function computes the weighted kurtosis of the dataset `data` using the given values of the weighted mean and weighted standard deviation, `wmean` and `wsd`.

21.8 Maximum and Minimum values

The following functions find the maximum and minimum values of a dataset (or their indices). If the data contains NaN-s then a NaN will be returned, since the maximum or minimum value is undefined. For functions which return an index, the location of the first NaN in the array is returned.

double **gsl_stats_max**(const double data[], size_t stride, size_t n)

This function returns the maximum value in data, a dataset of length n with stride stride. The maximum value is defined as the value of the element x_i which satisfies $x_i \geq x_j$ for all j.

If you want instead to find the element with the largest absolute magnitude you will need to apply fabs() or abs() to your data before calling this function.

double **gsl_stats_min**(const double data[], size_t stride, size_t n)

This function returns the minimum value in data, a dataset of length n with stride stride. The minimum value is defined as the value of the element x_i which satisfies $x_i \leq x_j$ for all j.

If you want instead to find the element with the smallest absolute magnitude you will need to apply fabs() or abs() to your data before calling this function.

void **gsl_stats_minmax**(double *min, double *max, const double data[], size_t stride, size_t n)

This function finds both the minimum and maximum values min, max in data in a single pass.

size_t **gsl_stats_max_index**(const double data[], size_t stride, size_t n)

This function returns the index of the maximum value in data, a dataset of length n with stride stride. The maximum value is defined as the value of the element x_i which satisfies $x_i \geq x_j$ for all j. When there are several equal maximum elements then the first one is chosen.

size_t **gsl_stats_min_index**(const double data[], size_t stride, size_t n)

This function returns the index of the minimum value in data, a dataset of length n with stride stride. The minimum value is defined as the value of the element x_i which satisfies $x_i \geq x_j$ for all j. When there are several equal minimum elements then the first one is chosen.

void **gsl_stats_minmax_index**(size_t *min_index, size_t *max_index, const double data[], size_t stride, size_t n)

This function returns the indexes min_index, max_index of the minimum and maximum values in data in a single pass.

21.9 Median and Percentiles

The median and percentile functions described in this section operate on sorted data in $O(1)$ time. There is also a routine for computing the median of an unsorted input array in average $O(n)$ time using the quickselect algorithm. For convenience we use quantiles, measured on a scale of 0 to 1, instead of percentiles (which use a scale of 0 to 100).

double **gsl_stats_median_from_sorted_data**(const double sorted_data[], const size_t stride, const size_t n)

This function returns the median value of `sorted_data`, a dataset of length `n` with stride `stride`. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function `gsl_sort()` should always be used first.

When the dataset has an odd number of elements the median is the value of element $(n-1)/2$. When the dataset has an even number of elements the median is the mean of the two nearest middle values, elements $(n-1)/2$ and $n/2$. Since the algorithm for computing the median involves interpolation this function always returns a floating-point number, even for integer data types.

double **gsl_stats_median**(double data[], const size_t stride, const size_t n)

This function returns the median value of `data`, a dataset of length `n` with stride `stride`. The median is found using the quickselect algorithm. The input array does not need to be sorted, but note that the algorithm rearranges the array and so the input is not preserved on output.

double **gsl_stats_quantile_from_sorted_data**(const double sorted_data[], size_t stride, size_t n, double f)

This function returns a quantile value of `sorted_data`, a double-precision array of length `n` with stride `stride`. The elements of the array must be in ascending numerical order. The quantile is determined by the `f`, a fraction between 0 and 1. For example, to compute the value of the 75th percentile `f` should have the value 0.75.

There are no checks to see whether the data are sorted, so the function `gsl_sort()` should always be used first.

The quantile is found by interpolation, using the formula

$$\text{quantile} = (1-\delta)x_i + \delta x_{i+1}$$

where i is `floor((n - 1)f)` and δ is $(n-1)f - i$.

Thus the minimum value of the array (`data[0*stride]`) is given by `f` equal to zero, the

maximum value (`data[(n-1)*stride]`) is given by `f` equal to one and the median value is given by `f` equal to 0.5. Since the algorithm for computing quantiles involves interpolation this function always returns a floating-point number, even for integer data types.

21.10 Order Statistics

The k-th order statistic of a sample is equal to its k-th smallest value. The k-th order statistic of a set of n values $x = \{x_i\}, 1 \leq i \leq n$ is denoted $x_{(k)}$. The median of the set x is equal to $x_{\left(\frac{n}{2}\right)}$ if n is odd, or the average of $x_{\left(\frac{n}{2}\right)}$ and $x_{\left(\frac{n}{2}+1\right)}$ if n is even. The k-th smallest element of a length n vector can be found in average $O(n)$ time using the quickselect algorithm.

double **gsl_stats_select**(double data[], const size_t stride, const size_t n, const size_t k)

This function finds the `k`-th smallest element of the input array `data` of length `n` and stride `stride` using the quickselect method. The algorithm rearranges the elements of `data` and so the input array is not preserved on output.

21.11 Robust Location Estimates

A location estimate refers to a typical or central value which best describes a given dataset. The mean and median are both examples of location estimators. However, the mean has a severe sensitivity to data outliers and can give erroneous values when even a small number of outliers are present. The median on the other hand, has a strong insensitivity to data outliers, but due to its non-smoothness it can behave unexpectedly in certain situations. GSL offers the following alternative location estimators, which are robust to the presence of outliers.

21.11.1 Trimmed Mean

The trimmed mean, or truncated mean, discards a certain number of smallest and largest samples from the input vector before computing the mean of the remaining samples. The amount of trimming is specified by a factor $\alpha \in [0, 0.5]$. Then the number of samples discarded from both ends of the input vector is $\lfloor \alpha n \rfloor$, where n is the length of the input. So to discard 25% of the samples from each end, one would set $\alpha = 0.25$.

double **gsl_stats_trmean_from_sorted_data**(const double alpha, const double sorted_data[], const size_t stride, const size_t n)

This function returns the trimmed mean of `sorted_data`, a dataset of length `n` with stride `stride`. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function `gsl_sort()` should always be

used first. The trimming factor α is given in `alpha`. If $\alpha \geq 0.5$, then the median of the input is returned.

21.11.2 Gastwirth Estimator

Gastwirth's location estimator is a weighted sum of three order statistics,

$$gastwirth = 0.3 \times Q_{\frac{1}{3}} + 0.4 \times Q_{\frac{1}{2}} + 0.3 \times Q_{\frac{2}{3}}$$

where $Q_{\frac{1}{3}}$ is the one-third quantile, $Q_{\frac{1}{2}}$ is the one-half quantile (i.e. median), and $Q_{\frac{2}{3}}$ is the two-thirds quantile.

double **gsl_stats_gastwirth_from_sorted_data**(const double sorted_data[], const size_t stride, const size_t n)

This function returns the Gastwirth location estimator of `sorted_data`, a dataset of length `n` with stride `stride`. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function `gsl_sort()` should always be used first.

21.12 Robust Scale Estimates

A robust scale estimate, also known as a robust measure of scale, attempts to quantify the statistical dispersion (variability, scatter, spread) in a set of data which may contain outliers. In such datasets, the usual variance or standard deviation scale estimate can be rendered useless by even a single outlier.

21.12.1 Median Absolute Deviation (MAD)

The median absolute deviation (MAD) is defined as

$$MAD = 1.4826 \times \text{median}\left\{\left|x_i - \text{median}\,(x)\right|\right\}$$

In words, first the median of all samples is computed. Then the median is subtracted from all samples in the input to find the deviation of each sample from the median. The median of all absolute deviations is then the MAD. The factor 1.4826 makes the MAD an unbiased estimator of the standard deviation for Gaussian data. The median absolute deviation has an asymptotic efficiency of 37%.

double **gsl_stats_mad0**(const double data[], const size_t stride, const size_t n, double work[])

double **gsl_stats_mad**(const double data[], const size_t stride, const size_t n, double work[])

These functions return the median absolute deviation of `data`, a dataset of length `n` and stride `stride`. The `mad0` function calculates median $\{|x_i - \text{median}\,(x)|\}$ (i.e. the MAD statistic without the bias correction scale factor). These functions require additional workspace of size `n` provided in `work`.

21.12.2 S_n Statistic

The S_n statistic developed by Croux and Rousseeuw is defined as

$$S_n = 1.1926 \times c_n \times \text{median}_i \left\{\text{median}_j \left(|x_i - x_j|\right)\right\}$$

For each sample $x_i, 1 \le i \le n$, the median of the values $|x_i - x_j|$ is computed for all $x_j, 1 \le j \le n$. This yields n values, whose median then gives the final S_n. The factor 1.1926 makes S_n an unbiased estimate of the standard deviation for Gaussian data. The factor c_n is a correction factor to correct bias in small sample sizes. S_n has an asymptotic efficiency of 58%.

double **gsl_stats_Sn0_from_sorted_data**(const double sorted_data[], const size_t stride, const size_t n, double work[])

double **gsl_stats_Sn_from_sorted_data**(const double sorted_data[], const size_t stride, const size_t n, double work[])

These functions return the S_n statistic of `sorted_data`, a dataset of length `n` with stride `stride`. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function `gsl_sort()` should always be used first. The `Sn0` function calculates $\text{median}_i \left\{\text{median}_j \left(|x_i - x_j|\right)\right\}$ (i.e. the S_n statistic without the bias correction scale factors). These functions require additional workspace of size `n` provided in `work`.

21.12.3 Q_n Statistic

The Q_n statistic developed by Croux and Rousseeuw is defined as

$$Q_n = 2.21914 \times d_n \times \{|x_i - x_j|, i < j\}_{(k)}$$

The factor 2.21914 makes Q_n an unbiased estimate of the standard deviation for Gaussian data. The factor d_n is a correction factor to correct bias in small sample sizes. The order statistic is

$$k = \begin{pmatrix} \lfloor \frac{n}{2} \rfloor + 1 \\ 2 \end{pmatrix}$$

Q_n has an asymptotic efficiency of 82%.

double **gsl_stats_Qn0_from_sorted_data**(const double sorted_data[], const size_t stride, const size_t n, double work[], int work_int[])

double **gsl_stats_Qn_from_sorted_data**(const double sorted_data[], const size_t stride, const size_t n, double work[], int work_int[])

These functions return the Q_n statistic of `sorted_data`, a dataset of length `n` with stride `stride`. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function `gsl_sort()` should always be used first. The `Qn0` function calculates $\{|x_i - x_j|, i < j\}_{(k)}$ (i.e. Q_n without the bias correction scale factors). These functions require additional workspace of size `3n` provided in `work` and integer workspace of size `5n` provided in `work_int`.

21.13 Examples

Here is a basic example of how to use the statistical functions:

```
#include <stdio.h>
#include <gsl/gsl_statistics.h>

int
main(void)
{
  double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
  double mean, variance, largest, smallest;

  mean     = gsl_stats_mean(data, 1, 5);
  variance = gsl_stats_variance(data, 1, 5);
  largest  = gsl_stats_max(data, 1, 5);
  smallest = gsl_stats_min(data, 1, 5);

  printf ("The dataset is %g, %g, %g, %g, %g\n",
          data[0], data[1], data[2], data[3], data[4]);

  printf ("The sample mean is %g\n", mean);
  printf ("The estimated variance is %g\n", variance);
  printf ("The largest value is %g\n", largest);
  printf ("The smallest value is %g\n", smallest);
  return 0;
}
```

The program should produce the following output,

```
The dataset is 17.2, 18.1, 16.5, 18.3, 12.6
The sample mean is 16.54
The estimated variance is 5.373
The largest value is 18.3
The smallest value is 12.6
```

Here is an example using sorted data,

```
#include <stdio.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_statistics.h>

int
main(void)
{
  double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
  double median, upperq, lowerq;

  printf ("Original dataset:  %g, %g, %g, %g, %g\n",
          data[0], data[1], data[2], data[3], data[4]);

  gsl_sort (data, 1, 5);

  printf ("Sorted dataset: %g, %g, %g, %g, %g\n",
          data[0], data[1], data[2], data[3], data[4]);

  median
    = gsl_stats_median_from_sorted_data (data,
                                         1, 5);

  upperq
    = gsl_stats_quantile_from_sorted_data (data,
                                           1, 5,
                                           0.75);
  lowerq
    = gsl_stats_quantile_from_sorted_data (data,
                                           1, 5,
                                           0.25);

  printf ("The median is %g\n", median);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  printf ("The upper quartile is %g\n", upperq);
  printf ("The lower quartile is %g\n", lowerq);
  return 0;
}
```

This program should produce the following output,

```
Original dataset: 17.2, 18.1, 16.5, 18.3, 12.6
Sorted dataset: 12.6, 16.5, 17.2, 18.1, 18.3
The median is 17.2
The upper quartile is 18.1
The lower quartile is 16.5
```

21.14 References and Further Reading

The standard reference for almost any topic in statistics is the multi-volume Advanced Theory of Statistics by Kendall and Stuart.

- Maurice Kendall, Alan Stuart, and J. Keith Ord. The Advanced Theory of Statistics (multiple volumes) reprinted as Kendall's Advanced Theory of Statistics. Wiley, ISBN 047023380X.

Many statistical concepts can be more easily understood by a Bayesian approach. The following book by Gelman, Carlin, Stern and Rubin gives a comprehensive coverage of the subject.

- Andrew Gelman, John B. Carlin, Hal S. Stern, Donald B. Rubin. Bayesian Data Analysis. Chapman & Hall, ISBN 0412039915.

For physicists the Particle Data Group provides useful reviews of Probability and Statistics in the "Mathematical Tools" section of its Annual Review of Particle Physics.

- Review of Particle Properties, R.M. Barnett et al., Physical Review D54, 1 (1996)

The Review of Particle Physics is available online at the website http://pdg.lbl.gov/.

The following papers describe robust scale estimation,

- C. Croux and P. J. Rousseeuw, Time-Efficient algorithms for two highly robust estimators of scale, Comp. Stat., Physica, Heidelberg, 1992.
- P. J. Rousseeuw and C. Croux, Explicit scale estimators with high breakdown point, L1-Statistical Analysis and Related Methods, pp. 77-92, 1992.

제 22 장

통계 실행

참고: 번역중

This chapter describes routines for computing running statistics, also known as online statistics, of data. These routines are suitable for handling large datasets for which it may be inconvenient or impractical to store in memory all at once. The data can be processed in a single pass, one point at a time. Each time a data point is added to the accumulator, internal parameters are updated in order to compute the current mean, variance, standard deviation, skewness, and kurtosis. These statistics are exact, and are updated with numerically stable single-pass algorithms. The median and arbitrary quantiles are also available, however these calculations use algorithms which provide approximations, and grow more accurate as more data is added to the accumulator.

The functions described in this chapter are declared in the header file `gsl_rstat.h`.

22.1 Initializing the Accumulator

type **gsl_rstat_workspace**

This workspace contains parameters used to calculate various statistics and are updated after each data point is added to the accumulator.

gsl_rstat_workspace ***gsl_rstat_alloc**(void)

This function allocates a workspace for computing running statistics. The size of the workspace is $O(1)$.

void **gsl_rstat_free**(gsl_rstat_workspace *w)

This function frees the memory associated with the workspace `w`.

int **gsl_rstat_reset**(gsl_rstat_workspace *w)

This function resets the workspace `w` to its initial state, so it can begin working on a new set of data.

22.2 Adding Data to the Accumulator

int **gsl_rstat_add**(const double x, gsl_rstat_workspace *w)

This function adds the data point `x` to the statistical accumulator, updating calculations of the mean, variance, standard deviation, skewness, kurtosis, and median.

size_t **gsl_rstat_n**(const gsl_rstat_workspace *w)

This function returns the number of data so far added to the accumulator.

22.3 Current Statistics

double **gsl_rstat_min**(const gsl_rstat_workspace *w)

This function returns the minimum value added to the accumulator.

double **gsl_rstat_max**(const gsl_rstat_workspace *w)

This function returns the maximum value added to the accumulator.

double **gsl_rstat_mean**(const gsl_rstat_workspace *w)

This function returns the mean of all data added to the accumulator, defined as

$$\hat{\mu} = \frac{1}{N} \sum x_i$$

double **gsl_rstat_variance**(const gsl_rstat_workspace *w)

This function returns the variance of all data added to the accumulator, defined as

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - \hat{\mu})^2$$

double **gsl_rstat_sd**(const gsl_rstat_workspace *w)

This function returns the standard deviation of all data added to the accumulator, defined as the square root of the variance given above.

double **gsl_rstat_sd_mean**(const gsl_rstat_workspace *w)

This function returns the standard deviation of the mean, defined as

$$\hat{\sigma}_{\hat{\mu}} = \frac{\hat{\sigma}}{\sqrt{N}}$$

double **gsl_rstat_rms**(const gsl_rstat_workspace *w)

This function returns the root mean square of all data added to the accumulator, defined as

$$rms = \sqrt{\frac{1}{N}\sum x_i^2}$$

double **gsl_rstat_skew**(const gsl_rstat_workspace *w)

This function returns the skewness of all data added to the accumulator, defined as

$$skew = \frac{1}{N}\sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}}\right)^3$$

double **gsl_rstat_kurtosis**(const gsl_rstat_workspace *w)

This function returns the kurtosis of all data added to the accumulator, defined as

$$kurtosis = \left(\frac{1}{N}\sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}}\right)^4\right) - 3$$

double **gsl_rstat_median**(gsl_rstat_workspace *w)

This function returns an estimate of the median of the data added to the accumulator.

22.4 Quantiles

The functions in this section estimate quantiles dynamically without storing the entire dataset, using the algorithm of Jain and Chlamtec, 1985. Only five points (markers) are stored which represent the minimum and maximum of the data, as well as current estimates of the $p/2$-, p-, and $(1+p)/2$-quantiles. Each time a new data point is added, the marker positions and heights are updated.

type **gsl_rstat_quantile_workspace**

This workspace contains parameters for estimating quantiles of the current dataset

gsl_rstat_quantile_workspace *__gsl_rstat_quantile_alloc__(const double p)

This function allocates a workspace for the dynamic estimation of `p`-quantiles, where `p` is between 0 and 1. The median corresponds to $p = 0.5$. The size of the workspace is $O(1)$.

void __gsl_rstat_quantile_free__(gsl_rstat_quantile_workspace *w)

This function frees the memory associated with the workspace `w`.

int __gsl_rstat_quantile_reset__(gsl_rstat_quantile_workspace *w)

This function resets the workspace `w` to its initial state, so it can begin working on a new set of data.

int __gsl_rstat_quantile_add__(const double x, gsl_rstat_quantile_workspace *w)

This function updates the estimate of the p-quantile with the new data point `x`.

double __gsl_rstat_quantile_get__(gsl_rstat_quantile_workspace *w)

This function returns the current estimate of the p-quantile.

22.5 Examples

Here is a basic example of how to use the statistical functions:

```
#include <stdio.h>
#include <gsl/gsl_rstat.h>

int
main(void)
{
  double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
  double mean, variance, largest, smallest, sd,
         rms, sd_mean, median, skew, kurtosis;
  gsl_rstat_workspace *rstat_p = gsl_rstat_alloc();
  size_t i, n;

  /* add data to rstat accumulator */
  for (i = 0; i < 5; ++i)
    gsl_rstat_add(data[i], rstat_p);

  mean     = gsl_rstat_mean(rstat_p);
  variance = gsl_rstat_variance(rstat_p);
  largest  = gsl_rstat_max(rstat_p);
  smallest = gsl_rstat_min(rstat_p);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  median   = gsl_rstat_median(rstat_p);
  sd       = gsl_rstat_sd(rstat_p);
  sd_mean  = gsl_rstat_sd_mean(rstat_p);
  skew     = gsl_rstat_skew(rstat_p);
  rms      = gsl_rstat_rms(rstat_p);
  kurtosis = gsl_rstat_kurtosis(rstat_p);
  n        = gsl_rstat_n(rstat_p);

  printf ("The dataset is %g, %g, %g, %g, %g\n",
          data[0], data[1], data[2], data[3], data[4]);

  printf ("The sample mean is %g\n", mean);
  printf ("The estimated variance is %g\n", variance);
  printf ("The largest value is %g\n", largest);
  printf ("The smallest value is %g\n", smallest);
  printf( "The median is %g\n", median);
  printf( "The standard deviation is %g\n", sd);
  printf( "The root mean square is %g\n", rms);
  printf( "The standard devation of the mean is %g\n", sd_mean);
  printf( "The skew is %g\n", skew);
  printf( "The kurtosis %g\n", kurtosis);
  printf( "There are %zu items in the accumulator\n", n);

  gsl_rstat_reset(rstat_p);
  n = gsl_rstat_n(rstat_p);
  printf( "There are %zu items in the accumulator\n", n);

  gsl_rstat_free(rstat_p);

  return 0;
}
```

The program should produce the following output,

```
The dataset is 17.2, 18.1, 16.5, 18.3, 12.6
The sample mean is 16.54
The estimated variance is 5.373
The largest value is 18.3
The smallest value is 12.6
The median is 17.2
The standard deviation is 2.31797
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
The root mean square is 16.6694
The standard devation of the mean is 1.03663
The skew is -0.829058
The kurtosis -1.2217
There are 5 items in the accumulator
There are 0 items in the accumulator
```

This next program estimates the lower quartile, median and upper quartile from 10,000 samples of a random Rayleigh distribution, using the P^2 algorithm of Jain and Chlamtec. For comparison, the exact values are also computed from the sorted dataset.

```
#include <stdio.h>
#include <stdlib.h>
#include <gsl/gsl_rstat.h>
#include <gsl/gsl_statistics.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sort.h>

int
main(void)
{
  const size_t N = 10000;
  double *data = malloc(N * sizeof(double));
  gsl_rstat_quantile_workspace *work_25 = gsl_rstat_quantile_alloc(0.25);
  gsl_rstat_quantile_workspace *work_50 = gsl_rstat_quantile_alloc(0.5);
  gsl_rstat_quantile_workspace *work_75 = gsl_rstat_quantile_alloc(0.75);
  gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
  double exact_p25, exact_p50, exact_p75;
  double val_p25, val_p50, val_p75;
  size_t i;

  /* add data to quantile accumulators; also store data for exact
   * comparisons */
  for (i = 0; i < N; ++i)
    {
      data[i] = gsl_ran_rayleigh(r, 1.0);
      gsl_rstat_quantile_add(data[i], work_25);
      gsl_rstat_quantile_add(data[i], work_50);
      gsl_rstat_quantile_add(data[i], work_75);
    }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  /* exact values */
  gsl_sort(data, 1, N);
  exact_p25 = gsl_stats_quantile_from_sorted_data(data, 1, N, 0.25);
  exact_p50 = gsl_stats_quantile_from_sorted_data(data, 1, N, 0.5);
  exact_p75 = gsl_stats_quantile_from_sorted_data(data, 1, N, 0.75);

  /* estimated values */
  val_p25 = gsl_rstat_quantile_get(work_25);
  val_p50 = gsl_rstat_quantile_get(work_50);
  val_p75 = gsl_rstat_quantile_get(work_75);

  printf ("The dataset is %g, %g, %g, %g, %g, ...\n",
          data[0], data[1], data[2], data[3], data[4]);

  printf ("0.25 quartile: exact = %.5f, estimated = %.5f, error = %.6e\n",
           exact_p25, val_p25, (val_p25 - exact_p25) / exact_p25);
  printf ("0.50 quartile: exact = %.5f, estimated = %.5f, error = %.6e\n",
           exact_p50, val_p50, (val_p50 - exact_p50) / exact_p50);
  printf ("0.75 quartile: exact = %.5f, estimated = %.5f, error = %.6e\n",
           exact_p75, val_p75, (val_p75 - exact_p75) / exact_p75);

  gsl_rstat_quantile_free(work_25);
  gsl_rstat_quantile_free(work_50);
  gsl_rstat_quantile_free(work_75);
  gsl_rng_free(r);
  free(data);

  return 0;
}
```

The program should produce the following output,

```
The dataset is 0.00645272, 0.0074002, 0.0120706, 0.0207256, 0.0227282, ...
0.25 quartile: exact = 0.75766, estimated = 0.75580, error = -2.450209e-03
0.50 quartile: exact = 1.17508, estimated = 1.17438, error = -5.995912e-04
0.75 quartile: exact = 1.65347, estimated = 1.65696, error = 2.110571e-03
```

22.6 References and Further Reading

The algorithm used to dynamically estimate p-quantiles is described in the paper,

- R. Jain and I. Chlamtac. The P^2 algorithm for dynamic calculation of quantiles and histograms without storing observations, Communications of the ACM, Volume 28 (October), Number 10, 1985, p. 1076-1085.

제 23 장

디지털 필터링

참고: 번역중

23.1 Introduction

The filters discussed in this chapter are based on the following moving data window which is centered on i-th sample:

$$W_i^H = \{x_{i-H}, \ldots, x_i, \ldots, x_{i+H}\}$$

Here, H is a non-negative integer called the window half-length, which represents the number of samples before and after sample i. The total window length is $K = 2H + 1$.

23.2 Handling Endpoints

When processing samples near the ends of the input signal, there will not be enough samples to fill the window W_i^H defined above. Therefore the user must specify how to construct the windows near the end points. This is done by passing an input argument of type `gsl_filter_end_t`:

type **gsl_filter_end_t**

This data type specifies how to construct windows near end points and can be selected from the following choices:

GSL_FILTER_END_PADZERO

With this option, a full window of length K will be constructed by inserting zeros into the window near the signal end points. Effectively, the input signal is modified to

$$\tilde{x} = \{\underbrace{0, \dots, 0}_{H \text{zeros}}, x_1, x_2, \dots, x_{n-1}, x_n, \underbrace{0, \dots, 0}_{H \text{zeros}}\}$$

to ensure a well-defined window for all x_i.

GSL_FILTER_END_PADVALUE

With this option, a full window of length K will be constructed by padding the window with the first and last sample in the input signal. Effectively, the input signal is modified to

$$\tilde{x} = \{\underbrace{x_1, \dots, x_1}_{H}, x_1, x_2, \dots, x_{n-1}, x_n, \underbrace{x_n, \dots, x_n}_{H}\}$$

GSL_FILTER_END_TRUNCATE

With this option, no padding is performed, and the windows are simply truncated as the end points are approached.

23.3 Linear Digital Filters

23.3.1 Gaussian Filter

The Gaussian filter convolves the input signal with a Gaussian kernel or window. This filter is often used as a smoothing or noise reduction filter. The Gaussian kernel is defined by

$$G(k) = e^{-\frac{1}{2}\left(\alpha \frac{k}{(K-1)/2}\right)^2} = e^{-k^2/2\sigma^2}$$

for $-(K-1)/2 \leq k \leq (K-1)/2$, and K is the size of the kernel. The parameter α specifies the number of standard deviations σ desired in the kernel. So for example setting $\alpha = 3$ would define a Gaussian window of length K which spans $\pm 3\sigma$. It is often more convenient to specify the parameter α rather than the standard deviation σ when constructing the kernel, since a fixed value of α would correspond to the same shape of Gaussian regardless of the size K. The appropriate value of the standard deviation depends on K and is related to α as

$$\sigma = \frac{K-1}{2\alpha}$$

The routines below accept α as an input argument instead of σ.

The Gaussian filter offers a convenient way of differentiating and smoothing an input signal in a single pass. Using the derivative property of a convolution,

$$\frac{d}{dt}(G * x) = \frac{dG}{dt} * x$$

the input signal $x(t)$ can be smoothed and differentiated at the same time by convolution with a derivative Gaussian kernel, which can be readily computed from the analytic expression above. The same principle applies to higher order derivatives.

gsl_filter_gaussian_workspace ***gsl_filter_gaussian_alloc**(const size_t K)

This function initializes a workspace for Gaussian filtering using a kernel of size `K`. Here, $H = K/2$. If K is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is $O(K)$.

void **gsl_filter_gaussian_free**(gsl_filter_gaussian_workspace *w)

This function frees the memory associated with `w`.

int **gsl_filter_gaussian**(const gsl_filter_end_t endtype, const double alpha, const size_t order, const gsl_vector *x, gsl_vector *y, gsl_filter_gaussian_workspace *w)

This function applies a Gaussian filter parameterized by `alpha` to the input vector `x`, storing the output in `y`. The derivative order is specified by `order`, with `0` corresponding to a Gaussian, `1` corresponding to a first derivative Gaussian, and so on. The parameter `endtype` specifies how the signal end points are handled. It is allowed for `x` = `y` for an in-place filter.

int **gsl_filter_gaussian_kernel**(const double alpha, const size_t order, const int normalize, gsl_vector *kernel)

This function constructs a Gaussian kernel parameterized by `alpha` and stores the output in `kernel`. The parameter `order` specifies the derivative order, with `0` corresponding to a Gaussian, `1` corresponding to a first derivative Gaussian, and so on. If `normalize` is set to `1`, then the kernel will be normalized to sum to one on output. If `normalize` is set to `0`, no normalization is performed.

23.4 Nonlinear Digital Filters

The nonlinear digital filters described below are based on the window median, which is given by

$$m_i = \text{median}\,\{W_i^H\} = \text{median}\,\{x_{i-H}, \ldots, x_i, \ldots, x_{i+H}\}$$

The median is considered robust to local outliers, unlike the mean. Median filters can preserve sharp edges while at the same removing signal noise, and are used in a wide range of applications.

23.4.1 Standard Median Filter

The standard median filter (SMF) simply replaces the sample x_i by the median m_i of the window W_i^H: This filter has one tuning parameter given by H. The standard median filter is considered highly resistant to local outliers and local noise in the data sequence $\{x_i\}$.

gsl_filter_median_workspace ***gsl_filter_median_alloc**(const size_t K)

This function initializes a workspace for standard median filtering using a symmetric centered moving window of size `K`. Here, $H = K/2$. If K is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is $O(7K)$.

void **gsl_filter_median_free**(gsl_filter_median_workspace *w)

This function frees the memory associated with `w`.

int **gsl_filter_median**(const gsl_filter_end_t endtype, const gsl_vector *x, gsl_vector *y, gsl_filter_median_workspace *w)

This function applies a standard median filter to the input `x`, storing the output in `y`. The parameter `endtype` specifies how the signal end points are handled. It is allowed to have `x` = `y` for an in-place filter.

23.4.2 Recursive Median Filter

The recursive median filter (RMF) is a modification of the SMF to include previous filter outputs in the window before computing the median. The filter's response is

$$y_i = \text{median}\,(y_{i-H}, \ldots, y_{i-1}, x_i, x_{i+1}, \ldots, x_{i+H})$$

Sometimes, the SMF must be applied several times in a row to achieve adequate smoothing (i.e. a cascade filter). The RMF, on the other hand, converges to a root sequence in one pass,

and can sometimes provide a smoother result than several passes of the SMF. A root sequence is an input which is left unchanged by the filter. So there is no need to apply a recursive median filter twice to an input vector.

gsl_filter_rmedian_workspace ***gsl_filter_rmedian_alloc**(const size_t K)

This function initializes a workspace for recursive median filtering using a symmetric centered moving window of size `K`. Here, $H = K/2$. If K is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is $O(K)$.

void **gsl_filter_rmedian_free**(gsl_filter_rmedian_workspace *w)

This function frees the memory associated with `w`.

int **gsl_filter_rmedian**(const gsl_filter_end_t endtype, const gsl_vector *x, gsl_vector *y, gsl_filter_rmedian_workspace *w)

This function applies a recursive median filter to the input `x`, storing the output in `y`. The parameter `endtype` specifies how the signal end points are handled. It is allowed to have `x` = `y` for an in-place filter.

23.4.3 Impulse Detection Filter

Impulsive noise is characterized by short sequences of data points distinct from those in the surrounding neighborhood. This section describes a powerful class of filters, also known as impulse rejection filters and decision-based filters, designed to detect and remove such outliers from data. The filter's response is given by

$$y_i = \begin{cases} x_i, & |x_i - m_i| \le tS_i \\ m_i, & |x_i - m_i| > tS_i \end{cases}$$

where m_i is the median value of the window W_i^H, S_i is a robust estimate of the scatter or dispersion for the window W_i^H, and t is a tuning parameter specifying the number of scale factors needed to determine that a point is an outlier. The main idea is that the median m_i will be unaffected by a small number of outliers in the window, and so a given sample x_i is tested to determine how far away it is from the median in terms of the local scale estimate S_i. Samples which are more than t scale estimates away from the median are labeled as outliers and replaced by the window median m_i. Samples which are less than t scale estimates from the median are left unchanged by the filter.

Note that when $t = 0$, the impulse detection filter is equivalent to the standard median filter. When $t \to \infty$, it becomes the identity filter. This means the impulse detection filter can be viewed as a "less aggressive" version of the standard median filter, becoming less aggressive as t is increased. Note that this filter modifies only samples identified as outliers, while the standard median filter changes all samples to the local median, regardless of whether they are

outliers. This fact, plus the additional flexibility offered by the additional tuning parameter t can make the impulse detection filter a better choice for some applications.

It is important to have a robust and accurate scale estimate S_i in order to detect impulse outliers even in the presence of noise. The window standard deviation is not typically a good choice, as it can be significantly perturbed by the presence of even one outlier. GSL offers the following choices (specified by a parameter of type `gsl_filter_scale_t`) for computing the scale estimate S_i, all of which are robust to the presence of impulse outliers.

type **gsl_filter_scale_t**

This type specifies how the scale estimate S_i of the window W_i^H is calculated.

GSL_FILTER_SCALE_MAD

This option specifies the median absolute deviation (MAD) scale estimate, defined by

$$S_i = 1.4826 \times \text{median} \left\{ |W_i^H - m_i| \right\}$$

This choice of scale estimate is also known as the Hampel filter in the statistical literature. See here for more information.

GSL_FILTER_SCALE_IQR

This option specifies the interquartile range (IQR) scale estimate, defined as the difference between the 75th and 25th percentiles of the window W_i^H,

$$S_i = 0.7413 \left(Q_{0.75} - Q_{0.25} \right)$$

where Q_p is the p-quantile of the window W_i^H. The idea is to throw away the largest and smallest 25% of the window samples (where the outliers would be), and estimate a scale from the middle 50%. The factor 0.7413 provides an unbiased estimate of the standard deviation for Gaussian data.

GSL_FILTER_SCALE_SN

This option specifies the so-called S_n statistic proposed by Croux and Rousseeuw. See here for more information.

GSL_FILTER_SCALE_QN

This option specifies the so-called Q_n statistic proposed by Croux and Rousseeuw. See here for more information.

경고: While the scale estimates defined above are much less sensitive to outliers than the standard deviation, they can suffer from an effect called implosion. The standard

deviation of a window W_i^H will be zero if and only if all samples in the window are equal. However, it is possible for the MAD of a window to be zero even if all the samples in the window are not equal. For example, if $K/2+1$ or more of the K samples in the window are equal to some value x^*, then the window median will be equal to x^*. Consequently, at least $K/2+1$ of the absolute deviations $|x_j - x^*|$ will be zero, and so the MAD will be zero. In such a case, the Hampel filter will act like the standard median filter regardless of the value of t. Caution should also be exercised if dividing by S_i.

gsl_filter_impulse_workspace ***gsl_filter_impulse_alloc**(const size_t K)

This function initializes a workspace for impulse detection filtering using a symmetric moving window of size `K`. Here, $H = K/2$. If K is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is $O(6K)$.

void **gsl_filter_impulse_free**(gsl_filter_impulse_workspace *w)

This function frees the memory associated with `w`.

int **gsl_filter_impulse**(const gsl_filter_end_t endtype, const gsl_filter_scale_t scale_type, const double t, const gsl_vector *x, gsl_vector *y, gsl_vector *xmedian, gsl_vector *xsigma, size_t *noutlier, gsl_vector_int *ioutlier, gsl_filter_impulse_workspace *w)

These functions apply an impulse detection filter to the input vector `x`, storing the filtered output in `y`. The tuning parameter t is provided in `t`. The window medians m_i are stored in `xmedian` and the S_i are stored in `xsigma` on output. The number of outliers detected is stored in `noutlier` on output, while the locations of flagged outliers are stored in the boolean array `ioutlier`. The input `ioutlier` may be `NULL` if not desired. It is allowed to have `x` = `y` for an in-place filter.

23.5 Examples

23.5.1 Gaussian Example 1

This example program illustrates the Gaussian filter applied to smoothing a time series of length $N = 500$ with a kernel size of $K = 51$. Three filters are applied with parameters $\alpha = 0.5, 3, 10$. The results are shown in 그림 23.1.

We see that the filter corresponding to $\alpha = 0.5$ applies the most smoothing, while $\alpha = 10$ corresponds to the least amount of smoothing. The program is given below.

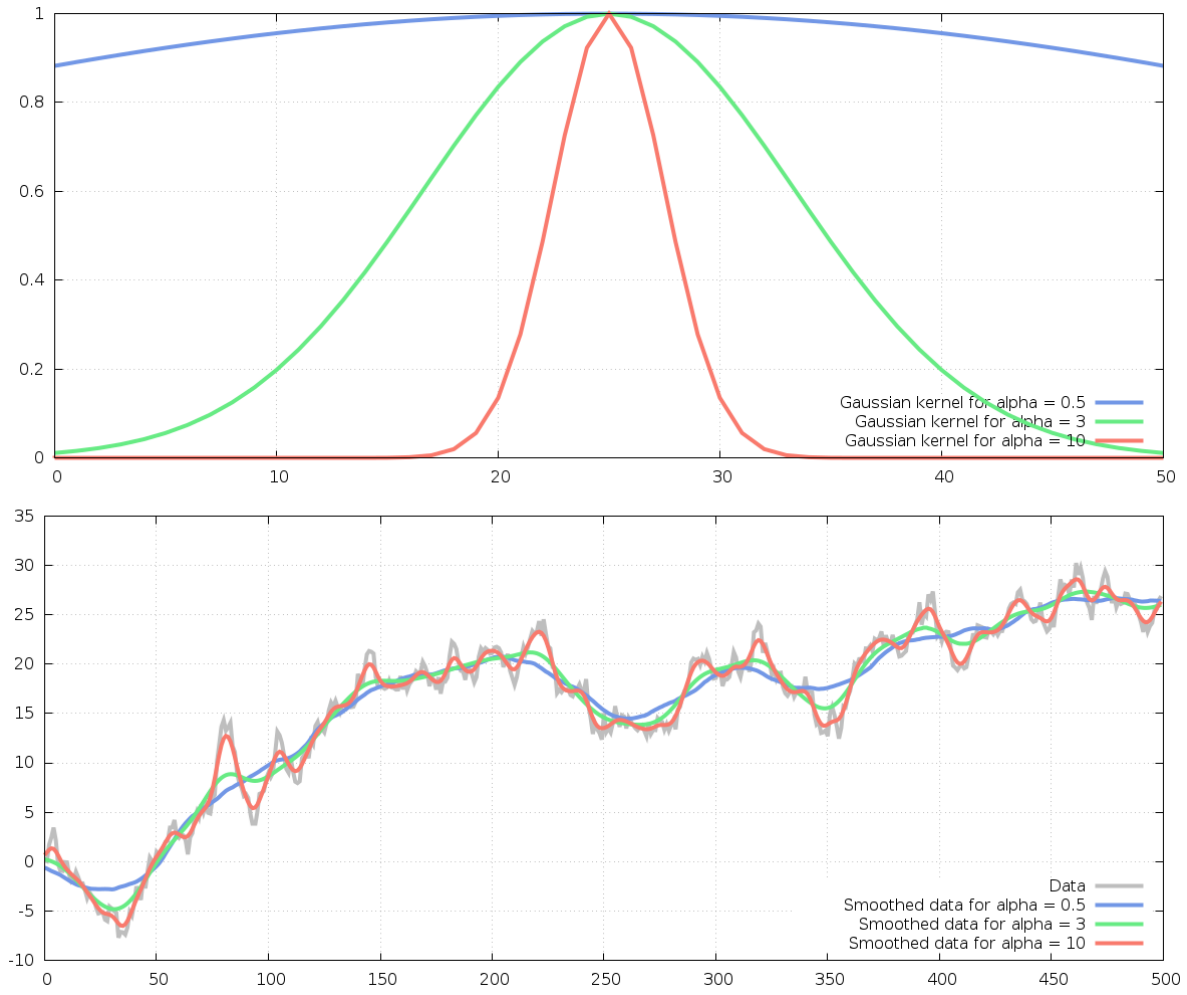


그림 23.1: Top panel: Gaussian kernels (unnormalized) for $\alpha = 0.5, 3, 10$. Bottom panel: Time series (gray) with Gaussian filter output for same α values.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
  const size_t N = 500;                       /* length of time series */
  const size_t K = 51;                        /* window size */
  const double alpha[3] = { 0.5, 3.0, 10.0 }; /* alpha values */
  gsl_vector *x = gsl_vector_alloc(N);        /* input vector */
  gsl_vector *y1 = gsl_vector_alloc(N);       /* filtered output vector for alpha1 */
  gsl_vector *y2 = gsl_vector_alloc(N);       /* filtered output vector for alpha2 */
  gsl_vector *y3 = gsl_vector_alloc(N);       /* filtered output vector for alpha3 */
  gsl_vector *k1 = gsl_vector_alloc(K);       /* Gaussian kernel for alpha1 */
  gsl_vector *k2 = gsl_vector_alloc(K);       /* Gaussian kernel for alpha2 */
  gsl_vector *k3 = gsl_vector_alloc(K);       /* Gaussian kernel for alpha3 */
  gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
  gsl_filter_gaussian_workspace *gauss_p = gsl_filter_gaussian_alloc(K);
  size_t i;
  double sum = 0.0;

  /* generate input signal */
  for (i = 0; i < N; ++i)
    {
      double ui = gsl_ran_gaussian(r, 1.0);
      sum += ui;
      gsl_vector_set(x, i, sum);
    }

  /* compute kernels without normalization */
  gsl_filter_gaussian_kernel(alpha[0], 0, 0, k1);
  gsl_filter_gaussian_kernel(alpha[1], 0, 0, k2);
  gsl_filter_gaussian_kernel(alpha[2], 0, 0, k3);

  /* apply filters */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha[0], 0, x, y1, gauss_p);
  gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha[1], 0, x, y2, gauss_p);
  gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha[2], 0, x, y3, gauss_p);

  /* print kernels */
  for (i = 0; i < K; ++i)
    {
      double k1i = gsl_vector_get(k1, i);
      double k2i = gsl_vector_get(k2, i);
      double k3i = gsl_vector_get(k3, i);

      printf("%e %e %e\n", k1i, k2i, k3i);
    }

  printf("\n\n");

  /* print filter results */
  for (i = 0; i < N; ++i)
    {
      double xi = gsl_vector_get(x, i);
      double y1i = gsl_vector_get(y1, i);
      double y2i = gsl_vector_get(y2, i);
      double y3i = gsl_vector_get(y3, i);

      printf("%.12e %.12e %.12e %.12e\n", xi, y1i, y2i, y3i);
    }

  gsl_vector_free(x);
  gsl_vector_free(y1);
  gsl_vector_free(y2);
  gsl_vector_free(y3);
  gsl_vector_free(k1);
  gsl_vector_free(k2);
  gsl_vector_free(k3);
  gsl_rng_free(r);
  gsl_filter_gaussian_free(gauss_p);

  return 0;
}
```

23.5.2 Gaussian Example 2

A common application of the Gaussian filter is to detect edges, or sudden jumps, in a noisy input signal. It is used both for 1D edge detection in time series, as well as 2D edge detection in images. Here we will examine a noisy time series of length $N = 1000$ with a single edge. The input signal is defined as

$$x(n) = e(n) + \begin{cases} 0, & n \leq N/2 \\ 0.5, & n > N/2 \end{cases}$$

where $e(n)$ is Gaussian random noise. The program smooths the input signal with order $0, 1,$ and 2 Gaussian filters of length $K = 61$ with $\alpha = 3$. For comparison, the program also computes finite differences of the input signal without smoothing. The results are shown in 그림 23.2.

The finite difference approximation of the first derivative (second row) shows the common problem with differentiating a noisy signal. The noise is amplified and makes it extremely difficult to detect the sharp gradient at sample 500. The third row shows the first order Gaussian smoothed signal with a clear peak at the location of the edge. Alternatively, one could examine the second order Gaussian smoothed signal (fourth row) and look for zero crossings to determine the edge location.

The program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
  const size_t N = 1000;                    /* length of time series */
  const size_t K = 61;                      /* window size */
  const double alpha = 3.0;                 /* Gaussian kernel has +/- 3 standard deviations */
  gsl_vector *x = gsl_vector_alloc(N);      /* input vector */
  gsl_vector *y = gsl_vector_alloc(N);      /* filtered output vector */
  gsl_vector *dy = gsl_vector_alloc(N);     /* first derivative filtered vector */
  gsl_vector *d2y = gsl_vector_alloc(N);    /* second derivative filtered vector */
```

(다음 페이지에 계속)

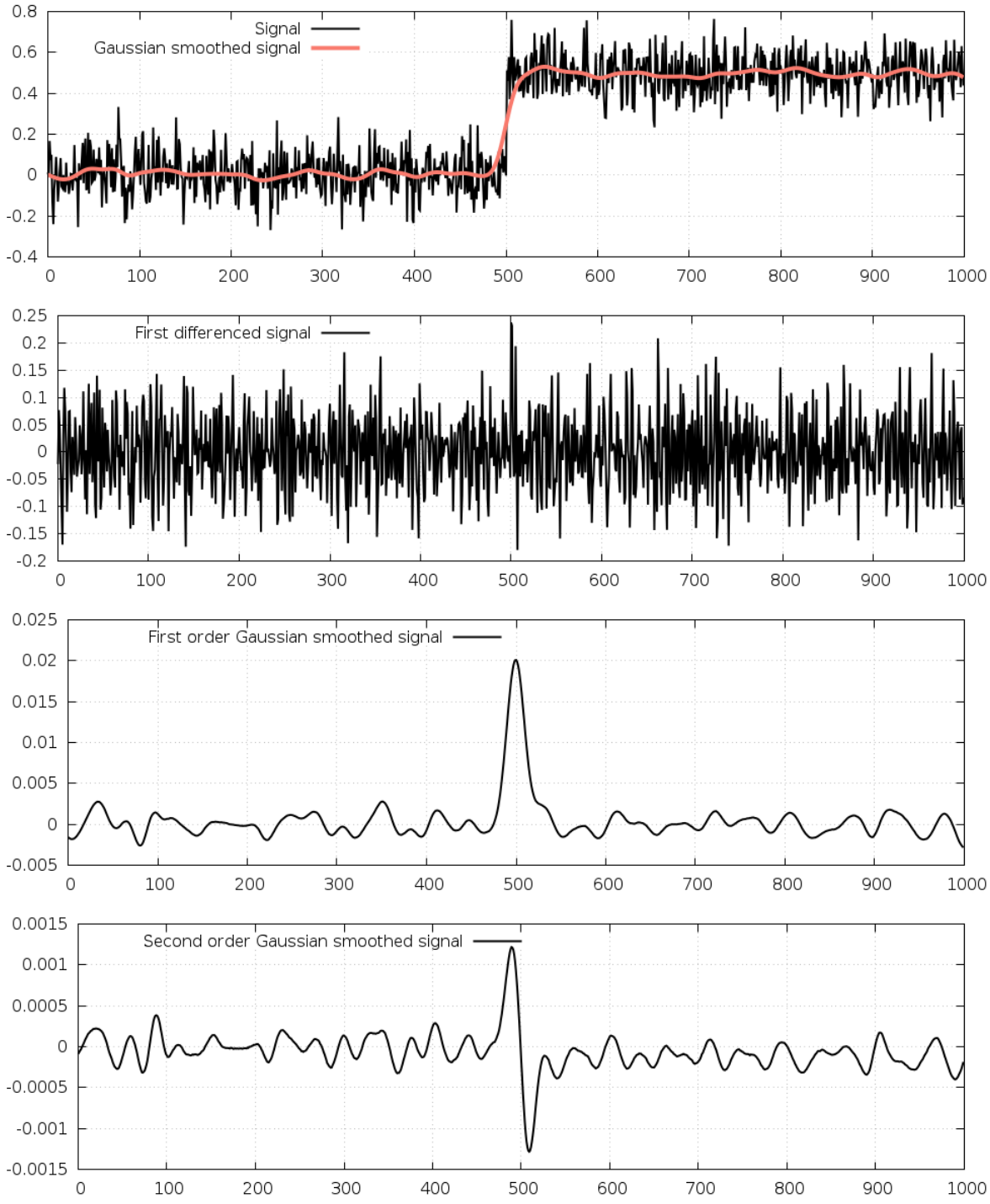


그림 23.2: Top row: original input signal $x(n)$ (black) with Gaussian smoothed signal in red. Second row: First finite differences of input signal. Third row: Input signal smoothed with a first order Gaussian filter. Fourth row: Input signal smoothed with a second order Gaussian filter.

(이전 페이지에서 계속)

```
gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
gsl_filter_gaussian_workspace *gauss_p = gsl_filter_gaussian_alloc(K);
size_t i;

/* generate input signal */
for (i = 0; i < N; ++i)
  {
    double xi = (i > N / 2) ? 0.5 : 0.0;
    double ei = gsl_ran_gaussian(r, 0.1);

    gsl_vector_set(x, i, xi + ei);
  }

/* apply filters */
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha, 0, x, y, gauss_p);
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha, 1, x, dy, gauss_p);
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha, 2, x, d2y, gauss_p);

/* print results */
for (i = 0; i < N; ++i)
  {
    double xi = gsl_vector_get(x, i);
    double yi = gsl_vector_get(y, i);
    double dyi = gsl_vector_get(dy, i);
    double d2yi = gsl_vector_get(d2y, i);
    double dxi;

    /* compute finite difference of x vector */
    if (i == 0)
      dxi = gsl_vector_get(x, i + 1) - xi;
    else if (i == N - 1)
      dxi = gsl_vector_get(x, i) - gsl_vector_get(x, i - 1);
    else
      dxi = 0.5 * (gsl_vector_get(x, i + 1) - gsl_vector_get(x, i - 1));

    printf("%.12e %.12e %.12e %.12e %.12e\n",
           xi,
           yi,
           dxi,
           dyi,
           d2yi);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    }

  gsl_vector_free(x);
  gsl_vector_free(y);
  gsl_vector_free(dy);
  gsl_vector_free(d2y);
  gsl_rng_free(r);
  gsl_filter_gaussian_free(gauss_p);

  return 0;
}
```

23.5.3 Square Wave Signal Example

The following example program illustrates the median filters on a noisy square wave signal. Median filters are well known for preserving sharp edges in the input signal while reducing noise. The program constructs a 5 Hz square wave signal with Gaussian noise added. Then the signal is filtered with a standard median filter and recursive median filter using a symmetric window of length $K = 7$. The results are shown in 그림 23.3.

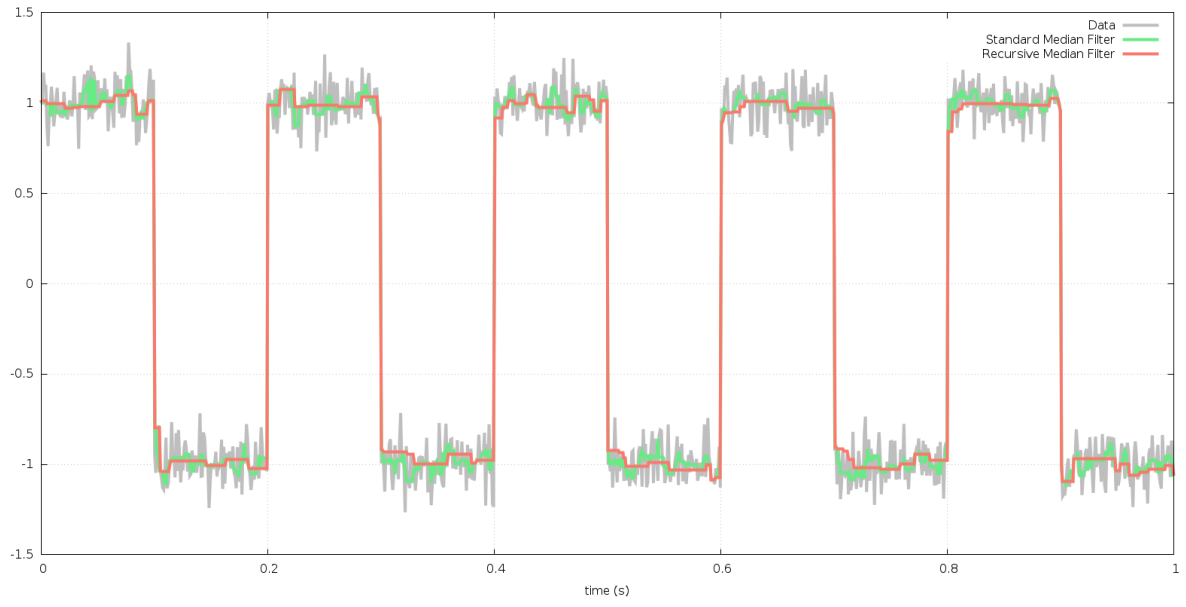


그림 23.3: Original time series is in gray. The standard median filter output is in green and the recursive median filter output is in red.

Both filters preserve the sharp signal edges while reducing the noise. The recursive median filter achieves a smoother result than the standard median filter. The "blocky" nature of the output is characteristic of all median filters. The program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
  const size_t N = 1000;                              /* length of time series */
  const size_t K = 7;                                 /* window size */
  const double f = 5.0;                               /* frequency of square wave in Hz */
  gsl_filter_median_workspace *median_p = gsl_filter_median_alloc(K);
  gsl_filter_rmedian_workspace *rmedian_p = gsl_filter_rmedian_alloc(K);
  gsl_vector *t = gsl_vector_alloc(N);                /* time */
  gsl_vector *x = gsl_vector_alloc(N);                /* input vector */
  gsl_vector *y_median = gsl_vector_alloc(N);         /* median filtered output */
  gsl_vector *y_rmedian = gsl_vector_alloc(N);        /* recursive median filtered output */
  gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
  size_t i;

  /* generate input signal */
  for (i = 0; i < N; ++i)
    {
      double ti = (double) i / (N - 1.0);
      double tmp = sin(2.0 * M_PI * f * ti);
      double xi = (tmp >= 0.0) ? 1.0 : -1.0;
      double ei = gsl_ran_gaussian(r, 0.1);

      gsl_vector_set(t, i, ti);
      gsl_vector_set(x, i, xi + ei);
    }

  gsl_filter_median(GSL_FILTER_END_PADVALUE, x, y_median, median_p);
  gsl_filter_rmedian(GSL_FILTER_END_PADVALUE, x, y_rmedian, rmedian_p);

  /* print results */
  for (i = 0; i < N; ++i)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    {
      double ti = gsl_vector_get(t, i);
      double xi = gsl_vector_get(x, i);
      double medi = gsl_vector_get(y_median, i);
      double rmedi = gsl_vector_get(y_rmedian, i);

      printf("%f %f %f %f\n",
             ti,
             xi,
             medi,
             rmedi);
    }

  gsl_vector_free(t);
  gsl_vector_free(x);
  gsl_vector_free(y_median);
  gsl_vector_free(y_rmedian);
  gsl_rng_free(r);
  gsl_filter_median_free(median_p);

  return 0;
}
```

23.5.4 Impulse Detection Example

The following example program illustrates the impulse detection filter. First, it constructs a sinusoid signal of length $N = 1000$ with Gaussian noise added. Then, about 1% of the data are perturbed to represent large outliers. An impulse detecting filter is applied with a window size $K = 25$ and tuning parameter $t = 4$, using the Q_n statistic as the robust measure of scale. The results are plotted in 그림 23.4.

The program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

(다음 페이지에 계속)

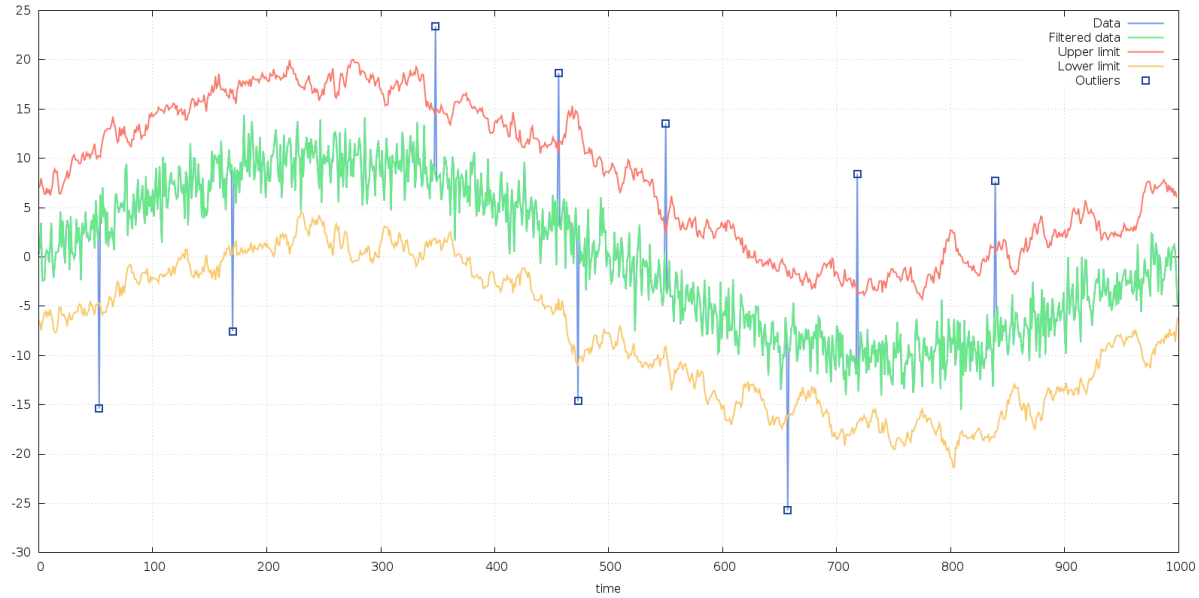


그림 23.4: Original time series is in blue, filter output is in green, upper and lower intervals for detecting outliers are in red and yellow respectively. Detected outliers are marked with squares.

(이전 페이지에서 계속)

```
#include <gsl/gsl_vector.h>

int
main(void)
{
  const size_t N = 1000;                          /* length of time series */
  const size_t K = 25;                            /* window size */
  const double t = 4.0;                           /* number of scale factors for outlier␣
↪detection */
  gsl_vector *x = gsl_vector_alloc(N);            /* input vector */
  gsl_vector *y = gsl_vector_alloc(N);            /* output (filtered) vector */
  gsl_vector *xmedian = gsl_vector_alloc(N);      /* window medians */
  gsl_vector *xsigma = gsl_vector_alloc(N);       /* window scale estimates */
  gsl_vector_int *ioutlier = gsl_vector_int_alloc(N);  /* outlier detected? */
  gsl_filter_impulse_workspace * w = gsl_filter_impulse_alloc(K);
  gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
  size_t noutlier;
  size_t i;

  /* generate input signal */
  for (i = 0; i < N; ++i)
    {
      double xi = 10.0 * sin(2.0 * M_PI * i / (double) N);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      double ei = gsl_ran_gaussian(r, 2.0);
      double u = gsl_rng_uniform(r);
      double outlier = (u < 0.01) ? 15.0*GSL_SIGN(ei) : 0.0;

      gsl_vector_set(x, i, xi + ei + outlier);
    }

  /* apply impulse detection filter */
  gsl_filter_impulse(GSL_FILTER_END_TRUNCATE, GSL_FILTER_SCALE_QN, t, x, y,
                     xmedian, xsigma, &noutlier, ioutlier, w);

  /* print results */
  for (i = 0; i < N; ++i)
    {
      double xi = gsl_vector_get(x, i);
      double yi = gsl_vector_get(y, i);
      double xmedi = gsl_vector_get(xmedian, i);
      double xsigmai = gsl_vector_get(xsigma, i);
      int outlier = gsl_vector_int_get(ioutlier, i);

      printf("%zu %f %f %f %f %d\n",
             i,
             xi,
             yi,
             xmedi + t * xsigmai,
             xmedi - t * xsigmai,
             outlier);
    }

  gsl_vector_free(x);
  gsl_vector_free(y);
  gsl_vector_free(xmedian);
  gsl_vector_free(xsigma);
  gsl_vector_int_free(ioutlier);
  gsl_filter_impulse_free(w);
  gsl_rng_free(r);

  return 0;
}
```

23.6 References and Further Reading

The following publications are relevant to the algorithms described in this chapter,

- F. J. Harris, On the use of windows for harmonic analysis with the discrete Fourier transform, Proceedings of the IEEE, 66 (1), 1978.
- S-J. Ko, Y-H. Lee, and A. T. Fam. Efficient implementation of one-dimensional recursive median filters, IEEE transactions on circuits and systems 37.11 (1990): 1447-1450.
- R. K. Pearson and M. Gabbouj, Nonlinear Digital Filtering with Python: An Introduction. CRC Press, 2015.

제 24 장

히스토그램

참고: 번역중

This chapter describes functions for creating histograms. Histograms provide a convenient way of summarizing the distribution of a set of data. A histogram consists of a set of bins which count the number of events falling into a given range of a continuous variable x. In GSL the bins of a histogram contain floating-point numbers, so they can be used to record both integer and non-integer distributions. The bins can use arbitrary sets of ranges (uniformly spaced bins are the default). Both one and two-dimensional histograms are supported.

Once a histogram has been created it can also be converted into a probability distribution function. The library provides efficient routines for selecting random samples from probability distributions. This can be useful for generating simulations based on real data.

The functions are declared in the header files `gsl_histogram.h` and `gsl_histogram2d.h`.

24.1 The histogram struct

A histogram is defined by the following struct,

type **`gsl_histogram`**

size_t n	This is the number of histogram bins
double * range	The ranges of the bins are stored in an array of n+1 elements pointed to by range.
double * bin	The counts for each bin are stored in an array of n elements pointed to by bin. The bins are floating-point numbers, so you can increment them by non-integer values if necessary.

The range for bin[i] is given by range[i] to range[i+1]. For n bins there are n+1 entries in the array range. Each bin is inclusive at the lower end and exclusive at the upper end. Mathematically this means that the bins are defined by the following inequality,

$$\text{bin[i] corresponds to range[i]} \leq x < \text{range[i+1]}$$

Here is a diagram of the correspondence between ranges and bins on the number-line for x:

```
   [ bin[0] )[ bin[1] )[ bin[2] )[ bin[3] )[ bin[4] )
---|---------|---------|---------|---------|---------|---  x
 r[0]      r[1]      r[2]      r[3]      r[4]      r[5]
```

In this picture the values of the range array are denoted by r. On the left-hand side of each bin the square bracket [denotes an inclusive lower bound ($r \leq x$), and the round parentheses) on the right-hand side denote an exclusive upper bound ($x < r$). Thus any samples which fall on the upper end of the histogram are excluded. If you want to include this value for the last bin you will need to add an extra bin to your histogram.

The gsl_histogram struct and its associated functions are defined in the header file gsl_histogram.h.

24.2 Histogram allocation

The functions for allocating memory to a histogram follow the style of malloc() and free(). In addition they also perform their own error checking. If there is insufficient memory available to allocate a histogram then the functions call the error handler (with an error number of GSL_ENOMEM) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every histogram alloc.

gsl_histogram ***gsl_histogram_alloc**(size_t n)

This function allocates memory for a histogram with `n` bins, and returns a pointer to a newly created `gsl_histogram` struct. If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`. The bins and ranges are not initialized, and should be prepared using one of the range-setting functions below in order to make the histogram ready for use.

int **gsl_histogram_set_ranges**(gsl_histogram *h, const double range[], size_t size)

This function sets the ranges of the existing histogram `h` using the array `range` of size `size`. The values of the histogram bins are reset to zero. The `range` array should contain the desired bin limits. The ranges can be arbitrary, subject to the restriction that they are monotonically increasing.

The following example shows how to create a histogram with logarithmic bins with ranges [1,10), [10,100) and [100,1000):

```
gsl_histogram * h = gsl_histogram_alloc (3);

/* bin[0] covers the range 1 <= x < 10 */
/* bin[1] covers the range 10 <= x < 100 */
/* bin[2] covers the range 100 <= x < 1000 */

double range[4] = { 1.0, 10.0, 100.0, 1000.0 };

gsl_histogram_set_ranges (h, range, 4);
```

Note that the size of the `range` array should be defined to be one element bigger than the number of bins. The additional element is required for the upper value of the final bin.

int **gsl_histogram_set_ranges_uniform**(gsl_histogram *h, double xmin, double xmax)

This function sets the ranges of the existing histogram `h` to cover the range `xmin` to `xmax` uniformly. The values of the histogram bins are reset to zero. The bin ranges are shown in the table below,

$$\begin{array}{lll} \text{bin[0]} & \text{corresponds to} & xmin \le x < xmin + d \\ \text{bin[1]} & \text{corresponds to} & xmin + d \le x < xmin + 2d \\ \dots & \dots & \dots \\ \text{bin[n-1]} & \text{corresponds to} & xmin + (n-1)d \le x < xmax \end{array}$$

where d is the bin spacing, $d = (xmax - xmin)/n$.

void **gsl_histogram_free**(gsl_histogram *h)

This function frees the histogram `h` and all of the memory associated with it.

24.3 Copying Histograms

int **gsl_histogram_memcpy**(gsl_histogram *dest, const gsl_histogram *src)

This function copies the histogram `src` into the pre-existing histogram `dest`, making `dest` into an exact copy of `src`. The two histograms must be of the same size.

gsl_histogram ***gsl_histogram_clone**(const gsl_histogram *src)

This function returns a pointer to a newly created histogram which is an exact copy of the histogram `src`.

24.4 Updating and accessing histogram elements

There are two ways to access histogram bins, either by specifying an x coordinate or by using the bin-index directly. The functions for accessing the histogram through x coordinates use a binary search to identify the bin which covers the appropriate range.

int **gsl_histogram_increment**(gsl_histogram *h, double x)

This function updates the histogram `h` by adding one (1.0) to the bin whose range contains the coordinate `x`.

If `x` lies in the valid range of the histogram then the function returns zero to indicate success. If `x` is less than the lower limit of the histogram then the function returns `GSL_EDOM`, and none of bins are modified. Similarly, if the value of `x` is greater than or equal to the upper limit of the histogram then the function returns `GSL_EDOM`, and none of the bins are modified. The error handler is not called, however, since it is often necessary to compute histograms for a small range of a larger dataset, ignoring the values outside the range of interest.

int **gsl_histogram_accumulate**(gsl_histogram *h, double x, double weight)

This function is similar to `gsl_histogram_increment()` but increases the value of the appropriate bin in the histogram `h` by the floating-point number `weight`.

double **gsl_histogram_get**(const gsl_histogram *h, size_t i)

This function returns the contents of the `i`-th bin of the histogram `h`. If `i` lies outside the valid range of indices for the histogram then the error handler is called with an error code of `GSL_EDOM` and the function returns 0.

int **gsl_histogram_get_range**(const gsl_histogram *h, size_t i, double *lower, double *upper)

This function finds the upper and lower range limits of the `i`-th bin of the histogram `h`. If the index `i` is valid then the corresponding range limits are stored in `lower` and `upper`. The lower limit is inclusive (i.e. events with this coordinate are included in the

bin) and the upper limit is exclusive (i.e. events with the coordinate of the upper limit are excluded and fall in the neighboring higher bin, if it exists). The function returns 0 to indicate success. If `i` lies outside the valid range of indices for the histogram then the error handler is called and the function returns an error code of `GSL_EDOM`.

double **gsl_histogram_max**(const gsl_histogram *h)

double **gsl_histogram_min**(const gsl_histogram *h)

size_t **gsl_histogram_bins**(const gsl_histogram *h)

These functions return the maximum upper and minimum lower range limits and the number of bins of the histogram `h`. They provide a way of determining these values without accessing the `gsl_histogram` struct directly.

void **gsl_histogram_reset**(gsl_histogram *h)

This function resets all the bins in the histogram `h` to zero.

24.5 Searching histogram ranges

The following functions are used by the access and update routines to locate the bin which corresponds to a given x coordinate.

int **gsl_histogram_find**(const gsl_histogram *h, double x, size_t *i)

This function finds and sets the index `i` to the bin number which covers the coordinate `x` in the histogram `h`. The bin is located using a binary search. The search includes an optimization for histograms with uniform range, and will return the correct bin immediately in this case. If `x` is found in the range of the histogram then the function sets the index `i` and returns `GSL_SUCCESS`. If `x` lies outside the valid range of the histogram then the function returns `GSL_EDOM` and the error handler is invoked.

24.6 Histogram Statistics

double **gsl_histogram_max_val**(const gsl_histogram *h)

This function returns the maximum value contained in the histogram bins.

size_t **gsl_histogram_max_bin**(const gsl_histogram *h)

This function returns the index of the bin containing the maximum value. In the case where several bins contain the same maximum value the smallest index is returned.

double **gsl_histogram_min_val**(const gsl_histogram *h)

This function returns the minimum value contained in the histogram bins.

size_t **gsl_histogram_min_bin**(const gsl_histogram *h)

This function returns the index of the bin containing the minimum value. In the case where several bins contain the same minimum value the smallest index is returned.

double **gsl_histogram_mean**(const gsl_histogram *h)

This function returns the mean of the histogrammed variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation. The accuracy of the result is limited by the bin width.

double **gsl_histogram_sigma**(const gsl_histogram *h)

This function returns the standard deviation of the histogrammed variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation. The accuracy of the result is limited by the bin width.

double **gsl_histogram_sum**(const gsl_histogram *h)

This function returns the sum of all bin values. Negative bin values are included in the sum.

24.7 Histogram Operations

int **gsl_histogram_equal_bins_p**(const gsl_histogram *h1, const gsl_histogram *h2)

This function returns 1 if the all of the individual bin ranges of the two histograms are identical, and 0 otherwise.

int **gsl_histogram_add**(gsl_histogram *h1, const gsl_histogram *h2)

This function adds the contents of the bins in histogram `h2` to the corresponding bins of histogram `h1`, i.e. $h_1'(i) = h_1(i) + h_2(i)$. The two histograms must have identical bin ranges.

int **gsl_histogram_sub**(gsl_histogram *h1, const gsl_histogram *h2)

This function subtracts the contents of the bins in histogram `h2` from the corresponding bins of histogram `h1`, i.e. $h_1'(i) = h_1(i) - h_2(i)$. The two histograms must have identical bin ranges.

int **gsl_histogram_mul**(gsl_histogram *h1, const gsl_histogram *h2)

This function multiplies the contents of the bins of histogram `h1` by the contents of the corresponding bins in histogram `h2`, i.e. $h_1'(i) = h_1(i) * h_2(i)$. The two histograms must have identical bin ranges.

int **gsl_histogram_div**(gsl_histogram *h1, const gsl_histogram *h2)

This function divides the contents of the bins of histogram `h1` by the contents of the

corresponding bins in histogram `h2`, i.e. $h'_1(i) = h_1(i)/h_2(i)$. The two histograms must have identical bin ranges.

int **gsl_histogram_scale**(gsl_histogram *h, double scale)

This function multiplies the contents of the bins of histogram `h` by the constant `scale`, i.e.

$$h'_1(i) = h_1(i) * \text{scale}$$

int **gsl_histogram_shift**(gsl_histogram *h, double offset)

This function shifts the contents of the bins of histogram `h` by the constant `offset`, i.e.

$$h'_1(i) = h_1(i) + \text{offset}$$

24.8 Reading and writing histograms

The library provides functions for reading and writing histograms to a file as binary data or formatted text.

int **gsl_histogram_fwrite**(FILE *stream, const gsl_histogram *h)

This function writes the ranges and bins of the histogram `h` to the stream `stream` in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_histogram_fread**(FILE *stream, gsl_histogram *h)

This function reads into the histogram `h` from the open stream `stream` in binary format. The histogram `h` must be preallocated with the correct size since the function uses the number of bins in `h` to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl_histogram_fprintf**(FILE *stream, const gsl_histogram *h, const char *range_format, const char *bin_format)

This function writes the ranges and bins of the histogram `h` line-by-line to the stream `stream` using the format specifiers `range_format` and `bin_format`. These should be one of the `%g`, `%e` or `%f` formats for floating point numbers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file. The histogram output is formatted in three columns, and the columns are separated by spaces, like this:

```
range[0] range[1] bin[0]
range[1] range[2] bin[1]
range[2] range[3] bin[2]
....
range[n-1] range[n] bin[n-1]
```

The values of the ranges are formatted using `range_format` and the value of the bins are formatted using `bin_format`. Each line contains the lower and upper limit of the range of the bins and the value of the bin itself. Since the upper limit of one bin is the lower limit of the next there is duplication of these values between lines but this allows the histogram to be manipulated with line-oriented tools.

int **gsl_histogram_fscanf**(FILE *stream, gsl_histogram *h)

This function reads formatted data from the stream `stream` into the histogram `h`. The data is assumed to be in the three-column format used by `gsl_histogram_fprintf()`. The histogram `h` must be preallocated with the correct length since the function uses the size of `h` to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

24.9 Resampling from histograms

A histogram made by counting events can be regarded as a measurement of a probability distribution. Allowing for statistical error, the height of each bin represents the probability of an event where the value of x falls in the range of that bin. The probability distribution function has the one-dimensional form $p(x)dx$ where,

$$p(x) = n_i/(N w_i)$$

In this equation n_i is the number of events in the bin which contains x, w_i is the width of the bin and N is the total number of events. The distribution of events within each bin is assumed to be uniform.

24.10 The histogram probability distribution struct

The probability distribution function for a histogram consists of a set of bins which measure the probability of an event falling into a given range of a continuous variable x. A probability distribution function is defined by the following struct, which actually stores the cumulative probability distribution function. This is the natural quantity for generating samples via the inverse transform method, because there is a one-to-one mapping between the cumulative probability distribution and the range [0,1]. It can be shown that by taking a uniform random number in this range and finding its corresponding coordinate in the cumulative probability distribution we obtain samples with the desired probability distribution.

type **gsl_histogram_pdf**

`size_t n`	This is the number of bins used to approximate the probability distribution function.
`double * range`	The ranges of the bins are stored in an array of $n+1$ elements pointed to by `range`.
`double * sum`	The cumulative probability for the bins is stored in an array of `n` elements pointed to by `sum`.

The following functions allow you to create a `gsl_histogram_pdf` struct which represents this probability distribution and generate random samples from it.

gsl_histogram_pdf ***gsl_histogram_pdf_alloc**(size_t n)

This function allocates memory for a probability distribution with `n` bins and returns a pointer to a newly initialized `gsl_histogram_pdf` struct. If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`.

int **gsl_histogram_pdf_init**(gsl_histogram_pdf *p, const gsl_histogram *h)

This function initializes the probability distribution `p` with the contents of the histogram `h`. If any of the bins of `h` are negative then the error handler is invoked with an error code of `GSL_EDOM` because a probability distribution cannot contain negative values.

void **gsl_histogram_pdf_free**(gsl_histogram_pdf *p)

This function frees the probability distribution function `p` and all of the memory associated with it.

double **gsl_histogram_pdf_sample**(const gsl_histogram_pdf *p, double r)

This function uses `r`, a uniform random number between zero and one, to compute a single random sample from the probability distribution `p`. The algorithm used to compute

the sample s is given by the following formula,

$$s = \text{range}[i] + \delta * (\text{range}[i+1] - \text{range}[i])$$

where i is the index which satisfies $sum[i] \leq r < sum[i+1]$ and $delta$ is $(r - sum[i])/(sum[i+1] - sum[i])$.

24.11 Example programs for histograms

The following program shows how to make a simple histogram of a column of numerical data supplied on `stdin`. The program takes three arguments, specifying the upper and lower bounds of the histogram and the number of bins. It then reads numbers from `stdin`, one line at a time, and adds them to the histogram. When there is no more data to read it prints out the accumulated histogram using `gsl_histogram_fprintf()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <gsl/gsl_histogram.h>

int
main (int argc, char **argv)
{
  double a, b;
  size_t n;

  if (argc != 4)
    {
      printf ("Usage: gsl-histogram xmin xmax n\n"
              "Computes a histogram of the data "
              "on stdin using n bins from xmin "
              "to xmax\n");
      exit (0);
    }

  a = atof (argv[1]);
  b = atof (argv[2]);
  n = atoi (argv[3]);

  {
    double x;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    gsl_histogram * h = gsl_histogram_alloc (n);
    gsl_histogram_set_ranges_uniform (h, a, b);

    while (fscanf (stdin, "%lg", &x) == 1)
      {
        gsl_histogram_increment (h, x);
      }
    gsl_histogram_fprintf (stdout, h, "%g", "%g");
    gsl_histogram_free (h);
  }
  exit (0);
}
```

Here is an example of the program in use. We generate 10000 random samples from a Cauchy distribution with a width of 30 and histogram them over the range -100 to 100, using 200 bins:

```
$ gsl-randist 0 10000 cauchy 30
   | gsl-histogram -- -100 100 200 > histogram.dat
```

그림 24.1 shows the familiar shape of the Cauchy distribution and the fluctuations caused by the finite sample size.

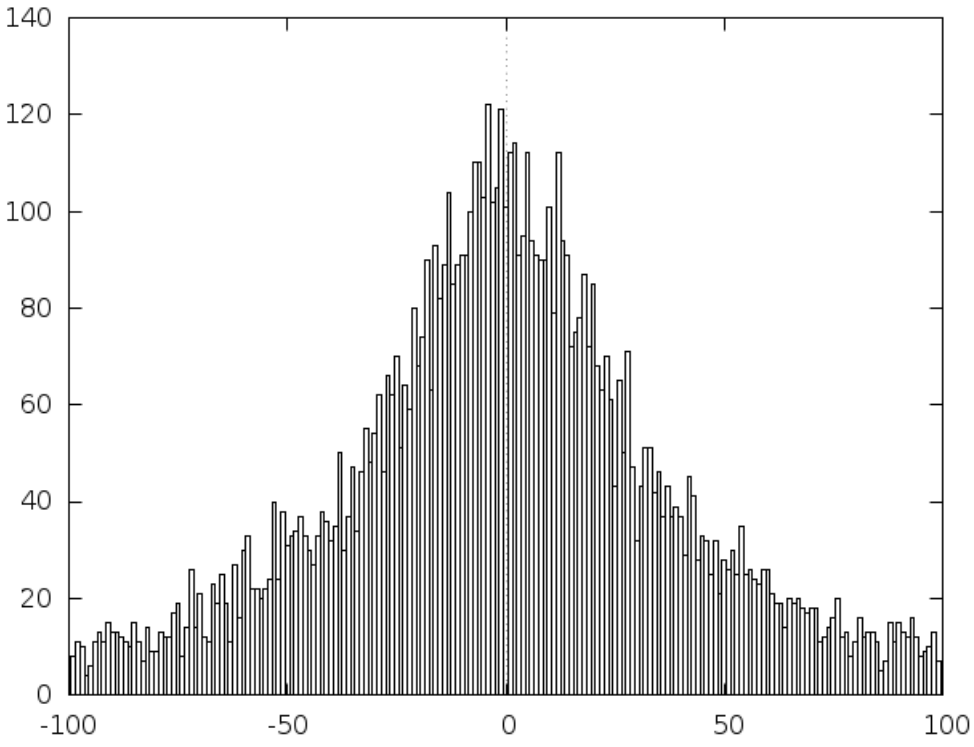


그림 24.1: Histogram output from example program

24.12 Two dimensional histograms

A two dimensional histogram consists of a set of bins which count the number of events falling in a given area of the (x, y) plane. The simplest way to use a two dimensional histogram is to record two-dimensional position information, $n(x, y)$. Another possibility is to form a joint distribution by recording related variables. For example a detector might record both the position of an event (x) and the amount of energy it deposited E. These could be histogrammed as the joint distribution $n(x, E)$.

24.13 The 2D histogram struct

Two dimensional histograms are defined by the following struct,

type **gsl_histogram2d**

`size_t nx, ny`	This is the number of histogram bins in the x and y directions.
`double * xrange`	The ranges of the bins in the x-direction are stored in an array of `nx + 1` elements pointed to by `xrange`.
`double * yrange`	The ranges of the bins in the y-direction are stored in an array of `ny + 1` elements pointed to by `yrange`.
`double * bin`	The counts for each bin are stored in an array pointed to by `bin`. The bins are floating-point numbers, so you can increment them by non-integer values if necessary. The array `bin` stores the two dimensional array of bins in a single block of memory according to the mapping `bin(i,j) = bin[i * ny + j]`.

The range for `bin(i,j)` is given by `xrange[i]` to `xrange[i+1]` in the x-direction and `yrange[j]` to `yrange[j+1]` in the y-direction. Each bin is inclusive at the lower end and exclusive at the upper end. Mathematically this means that the bins are defined by the following inequality,

$$\begin{aligned} \text{bin(i,j) corresponds to} \quad & \text{xrange}[i] \leq x < \text{xrange}[i+1] \\ \text{and} \quad & \text{yrange}[j] \leq y < \text{yrange}[j+1] \end{aligned}$$

Note that any samples which fall on the upper sides of the histogram are excluded. If you want to include these values for the side bins you will need to add an extra row or column to your histogram.

The gsl_histogram2d struct and its associated functions are defined in the header file gsl_histogram2d.h.

24.14 2D Histogram allocation

The functions for allocating memory to a 2D histogram follow the style of malloc() and free(). In addition they also perform their own error checking. If there is insufficient memory available to allocate a histogram then the functions call the error handler (with an error number of GSL_ENOMEM) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every 2D histogram alloc.

gsl_histogram2d *gsl_histogram2d_alloc(size_t nx, size_t ny)

This function allocates memory for a two-dimensional histogram with nx bins in the x direction and ny bins in the y direction. The function returns a pointer to a newly created gsl_histogram2d struct. If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of GSL_ENOMEM. The bins and ranges must be initialized with one of the functions below before the histogram is ready for use.

int gsl_histogram2d_set_ranges(gsl_histogram2d *h, const double xrange[], size_t xsize, const double yrange[], size_t ysize)

This function sets the ranges of the existing histogram h using the arrays xrange and yrange of size xsize and ysize respectively. The values of the histogram bins are reset to zero.

int gsl_histogram2d_set_ranges_uniform(gsl_histogram2d *h, double xmin, double xmax, double ymin, double ymax)

This function sets the ranges of the existing histogram h to cover the ranges xmin to xmax and ymin to ymax uniformly. The values of the histogram bins are reset to zero.

void gsl_histogram2d_free(gsl_histogram2d *h)

This function frees the 2D histogram h and all of the memory associated with it.

24.15 Copying 2D Histograms

int gsl_histogram2d_memcpy(gsl_histogram2d *dest, const gsl_histogram2d *src)

This function copies the histogram src into the pre-existing histogram dest, making dest into an exact copy of src. The two histograms must be of the same size.

gsl_histogram2d *gsl_histogram2d_clone(const gsl_histogram2d *src)

This function returns a pointer to a newly created histogram which is an exact copy of the histogram src.

24.16 Updating and accessing 2D histogram elements

You can access the bins of a two-dimensional histogram either by specifying a pair of (x, y) coordinates or by using the bin indices (i, j) directly. The functions for accessing the histogram through (x, y) coordinates use binary searches in the x and y directions to identify the bin which covers the appropriate range.

int gsl_histogram2d_increment(gsl_histogram2d *h, double x, double y)

This function updates the histogram h by adding one (1.0) to the bin whose x and y ranges contain the coordinates (x, y).

If the point (x, y) lies inside the valid ranges of the histogram then the function returns zero to indicate success. If (x, y) lies outside the limits of the histogram then the function returns GSL_EDOM, and none of the bins are modified. The error handler is not called, since it is often necessary to compute histograms for a small range of a larger dataset, ignoring any coordinates outside the range of interest.

int gsl_histogram2d_accumulate(gsl_histogram2d *h, double x, double y, double weight)

This function is similar to gsl_histogram2d_increment() but increases the value of the appropriate bin in the histogram h by the floating-point number weight.

double gsl_histogram2d_get(const gsl_histogram2d *h, size_t i, size_t j)

This function returns the contents of the (i, j)-th bin of the histogram h. If (i, j) lies outside the valid range of indices for the histogram then the error handler is called with an error code of GSL_EDOM and the function returns 0.

int gsl_histogram2d_get_xrange(const gsl_histogram2d *h, size_t i, double *xlower, double *xupper)

int gsl_histogram2d_get_yrange(const gsl_histogram2d *h, size_t j, double *ylower, double *yupper)

These functions find the upper and lower range limits of the i-th and j-th bins in the x and y directions of the histogram h. The range limits are stored in xlower and xupper or ylower and yupper. The lower limits are inclusive (i.e. events with these coordinates are included in the bin) and the upper limits are exclusive (i.e. events with the value of the upper limit are not included and fall in the neighboring higher bin, if it exists). The functions return 0 to indicate success. If i or j lies outside the valid range of indices for the histogram then the error handler is called with an error code of GSL_EDOM.

double **gsl_histogram2d_xmax**(const gsl_histogram2d *h)

double **gsl_histogram2d_xmin**(const gsl_histogram2d *h)

size_t **gsl_histogram2d_nx**(const gsl_histogram2d *h)

double **gsl_histogram2d_ymax**(const gsl_histogram2d *h)

double **gsl_histogram2d_ymin**(const gsl_histogram2d *h)

size_t **gsl_histogram2d_ny**(const gsl_histogram2d *h)

These functions return the maximum upper and minimum lower range limits and the number of bins for the x and y directions of the histogram `h`. They provide a way of determining these values without accessing the `gsl_histogram2d` struct directly.

void **gsl_histogram2d_reset**(gsl_histogram2d *h)

This function resets all the bins of the histogram `h` to zero.

24.17 Searching 2D histogram ranges

The following functions are used by the access and update routines to locate the bin which corresponds to a given (x, y) coordinate.

int **gsl_histogram2d_find**(const gsl_histogram2d *h, double x, double y, size_t *i, size_t *j)

This function finds and sets the indices `i` and `j` to the bin which covers the coordinates (`x`, `y`). The bin is located using a binary search. The search includes an optimization for histograms with uniform ranges, and will return the correct bin immediately in this case. If (x, y) is found then the function sets the indices (`i`, `j`) and returns `GSL_SUCCESS`. If (x, y) lies outside the valid range of the histogram then the function returns `GSL_EDOM` and the error handler is invoked.

24.18 2D Histogram Statistics

double **gsl_histogram2d_max_val**(const gsl_histogram2d *h)

This function returns the maximum value contained in the histogram bins.

void **gsl_histogram2d_max_bin**(const gsl_histogram2d *h, size_t *i, size_t *j)

This function finds the indices of the bin containing the maximum value in the histogram `h` and stores the result in (`i`, `j`). In the case where several bins contain the same maximum value the first bin found is returned.

double **gsl_histogram2d_min_val**(const gsl_histogram2d *h)

This function returns the minimum value contained in the histogram bins.

void **gsl_histogram2d_min_bin**(const gsl_histogram2d *h, size_t *i, size_t *j)

This function finds the indices of the bin containing the minimum value in the histogram `h` and stores the result in (`i`, `j`). In the case where several bins contain the same maximum value the first bin found is returned.

double **gsl_histogram2d_xmean**(const gsl_histogram2d *h)

This function returns the mean of the histogrammed x variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl_histogram2d_ymean**(const gsl_histogram2d *h)

This function returns the mean of the histogrammed y variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl_histogram2d_xsigma**(const gsl_histogram2d *h)

This function returns the standard deviation of the histogrammed x variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl_histogram2d_ysigma**(const gsl_histogram2d *h)

This function returns the standard deviation of the histogrammed y variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl_histogram2d_cov**(const gsl_histogram2d *h)

This function returns the covariance of the histogrammed x and y variables, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl_histogram2d_sum**(const gsl_histogram2d *h)

This function returns the sum of all bin values. Negative bin values are included in the sum.

24.19 2D Histogram Operations

int **gsl_histogram2d_equal_bins_p**(const gsl_histogram2d *h1, const gsl_histogram2d *h2)

This function returns 1 if all the individual bin ranges of the two histograms are identical, and 0 otherwise.

int **gsl_histogram2d_add**(gsl_histogram2d *h1, const gsl_histogram2d *h2)

This function adds the contents of the bins in histogram `h2` to the corresponding bins of histogram `h1`, i.e. $h_1'(i,j) = h_1(i,j) + h_2(i,j)$. The two histograms must have identical bin ranges.

int **gsl_histogram2d_sub**(gsl_histogram2d *h1, const gsl_histogram2d *h2)

This function subtracts the contents of the bins in histogram `h2` from the corresponding bins of histogram `h1`, i.e. $h_1'(i,j) = h_1(i,j) - h_2(i,j)$. The two histograms must have identical bin ranges.

int **gsl_histogram2d_mul**(gsl_histogram2d *h1, const gsl_histogram2d *h2)

This function multiplies the contents of the bins of histogram `h1` by the contents of the corresponding bins in histogram `h2`, i.e. $h_1'(i,j) = h_1(i,j) * h_2(i,j)$. The two histograms must have identical bin ranges.

int **gsl_histogram2d_div**(gsl_histogram2d *h1, const gsl_histogram2d *h2)

This function divides the contents of the bins of histogram `h1` by the contents of the corresponding bins in histogram `h2`, i.e. $h_1'(i,j) = h_1(i,j) / h_2(i,j)$. The two histograms must have identical bin ranges.

int **gsl_histogram2d_scale**(gsl_histogram2d *h, double scale)

This function multiplies the contents of the bins of histogram `h` by the constant `scale`, i.e.

$$h_1'(i,j) = h_1(i,j) * \text{scale}$$

int **gsl_histogram2d_shift**(gsl_histogram2d *h, double offset)

This function shifts the contents of the bins of histogram `h` by the constant `offset`, i.e.

$$h_1'(i,j) = h_1(i,j) + \text{offset}$$

24.20 Reading and writing 2D histograms

The library provides functions for reading and writing two dimensional histograms to a file as binary data or formatted text.

int **gsl_histogram2d_fwrite**(FILE *stream, const gsl_histogram2d *h)

This function writes the ranges and bins of the histogram `h` to the stream `stream` in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing

to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_histogram2d_fread**(FILE *stream, gsl_histogram2d *h)

This function reads into the histogram `h` from the stream `stream` in binary format. The histogram `h` must be preallocated with the correct size since the function uses the number of x and y bins in `h` to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl_histogram2d_fprintf**(FILE *stream, const gsl_histogram2d *h, const char *range_format, const char *bin_format)

This function writes the ranges and bins of the histogram `h` line-by-line to the stream `stream` using the format specifiers `range_format` and `bin_format`. These should be one of the `%g`, `%e` or `%f` formats for floating point numbers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file. The histogram output is formatted in five columns, and the columns are separated by spaces, like this:

```
xrange[0] xrange[1] yrange[0] yrange[1] bin(0,0)
xrange[0] xrange[1] yrange[1] yrange[2] bin(0,1)
xrange[0] xrange[1] yrange[2] yrange[3] bin(0,2)
....
xrange[0] xrange[1] yrange[ny-1] yrange[ny] bin(0,ny-1)

xrange[1] xrange[2] yrange[0] yrange[1] bin(1,0)
xrange[1] xrange[2] yrange[1] yrange[2] bin(1,1)
xrange[1] xrange[2] yrange[1] yrange[2] bin(1,2)
....
xrange[1] xrange[2] yrange[ny-1] yrange[ny] bin(1,ny-1)


....

xrange[nx-1] xrange[nx] yrange[0] yrange[1] bin(nx-1,0)
xrange[nx-1] xrange[nx] yrange[1] yrange[2] bin(nx-1,1)
xrange[nx-1] xrange[nx] yrange[1] yrange[2] bin(nx-1,2)
....
xrange[nx-1] xrange[nx] yrange[ny-1] yrange[ny] bin(nx-1,ny-1)
```

Each line contains the lower and upper limits of the bin and the contents of the bin. Since the upper limits of the each bin are the lower limits of the neighboring bins there is duplication of these values but this allows the histogram to be manipulated with line-oriented tools.

int **gsl_histogram2d_fscanf**(FILE *stream, gsl_histogram2d *h)

This function reads formatted data from the stream `stream` into the histogram `h`. The data is assumed to be in the five-column format used by `gsl_histogram2d_fprintf()`. The histogram `h` must be preallocated with the correct lengths since the function uses the sizes of `h` to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

24.21 Resampling from 2D histograms

As in the one-dimensional case, a two-dimensional histogram made by counting events can be regarded as a measurement of a probability distribution. Allowing for statistical error, the height of each bin represents the probability of an event where (x, y) falls in the range of that bin. For a two-dimensional histogram the probability distribution takes the form $p(x,y)dxdy$ where,

$$p(x,y) = n_{ij}/(NA_{ij})$$

In this equation n_{ij} is the number of events in the bin which contains (x,y), A_{ij} is the area of the bin and N is the total number of events. The distribution of events within each bin is assumed to be uniform.

type **gsl_histogram2d_pdf**

`size_t nx, ny`	This is the number of histogram bins used to approximate the probability distribution function in the x and y directions.
`double * xrange`	The ranges of the bins in the x-direction are stored in an array of `nx + 1` elements pointed to by `xrange`.
`double * yrange`	The ranges of the bins in the y-direction are stored in an array of `ny + 1` pointed to by `yrange`.
`double * sum`	The cumulative probability for the bins is stored in an array of `nx * ny` elements pointed to by `sum`.

The following functions allow you to create a `gsl_histogram2d_pdf` struct which represents a two dimensional probability distribution and generate random samples from it.

gsl_histogram2d_pdf ***gsl_histogram2d_pdf_alloc**(size_t nx, size_t ny)

This function allocates memory for a two-dimensional probability distribution of size `nx`-by-`ny` and returns a pointer to a newly initialized `gsl_histogram2d_pdf` struct. If insufficient

memory is available a null pointer is returned and the error handler is invoked with an error code of GSL_ENOMEM.

int **gsl_histogram2d_pdf_init**(gsl_histogram2d_pdf *p, const gsl_histogram2d *h)

This function initializes the two-dimensional probability distribution calculated p from the histogram h. If any of the bins of h are negative then the error handler is invoked with an error code of GSL_EDOM because a probability distribution cannot contain negative values.

void **gsl_histogram2d_pdf_free**(gsl_histogram2d_pdf *p)

This function frees the two-dimensional probability distribution function p and all of the memory associated with it.

int **gsl_histogram2d_pdf_sample**(const gsl_histogram2d_pdf *p, double r1, double r2, double *x, double *y)

This function uses two uniform random numbers between zero and one, r1 and r2, to compute a single random sample from the two-dimensional probability distribution p.

24.22 Example programs for 2D histograms

This program demonstrates two features of two-dimensional histograms. First a 10-by-10 two-dimensional histogram is created with x and y running from 0 to 1. Then a few sample points are added to the histogram, at (0.3,0.3) with a height of 1, at (0.8,0.1) with a height of 5 and at (0.7,0.9) with a height of 0.5. This histogram with three events is used to generate a random sample of 1000 simulated events, which are printed out.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_histogram2d.h>

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;

  gsl_histogram2d * h = gsl_histogram2d_alloc (10, 10);

  gsl_histogram2d_set_ranges_uniform (h,
                                      0.0, 1.0,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                                    0.0, 1.0);

  gsl_histogram2d_accumulate (h, 0.3, 0.3, 1);
  gsl_histogram2d_accumulate (h, 0.8, 0.1, 5);
  gsl_histogram2d_accumulate (h, 0.7, 0.9, 0.5);

  gsl_rng_env_setup ();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  {
    int i;
    gsl_histogram2d_pdf * p
      = gsl_histogram2d_pdf_alloc (h->nx, h->ny);

    gsl_histogram2d_pdf_init (p, h);

    for (i = 0; i < 1000; i++) {
      double x, y;
      double u = gsl_rng_uniform (r);
      double v = gsl_rng_uniform (r);

      gsl_histogram2d_pdf_sample (p, u, v, &x, &y);

      printf ("%g %g\n", x, y);
    }

    gsl_histogram2d_pdf_free (p);
  }

  gsl_histogram2d_free (h);
  gsl_rng_free (r);

  return 0;
}
```

The following plot shows the distribution of the simulated events. Using a higher resolution grid we can see the original underlying histogram and also the statistical fluctuations caused by the events being uniformly distributed over the area of the original bins.

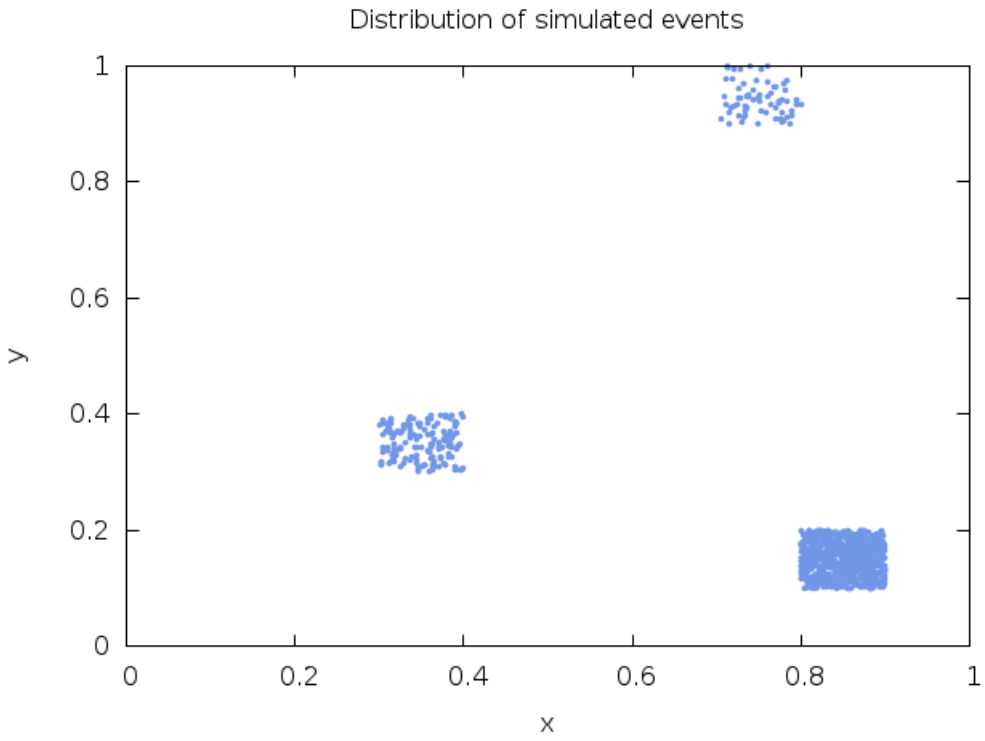


그림 24.2: Distribution of simulated events from example program

제 25 장

N-튜플

참고: 번역중

This chapter describes functions for creating and manipulating ntuples , sets of values associated with events. The ntuples are stored in files. Their values can be extracted in any combination and booked in a histogram using a selection function.

The values to be stored are held in a user-defined data structure, and an ntuple is created associating this data structure with a file. The values are then written to the file (normally inside a loop) using the ntuple functions described below.

A histogram can be created from ntuple data by providing a selection function and a value function. The selection function specifies whether an event should be included in the subset to be analyzed or not. The value function computes the entry to be added to the histogram for each event.

All the ntuple functions are defined in the header file `gsl_ntuple.h`.

25.1 The ntuple struct

type **`gsl_ntuple`**

Ntuples are manipulated using the `gsl_ntuple` struct. This struct contains information on the file where the ntuple data is stored, a pointer to the current ntuple data row and the size of the user-defined ntuple data struct:

```
typedef struct
  {
    FILE * file;
    void * ntuple_data;
    size_t size;
  } gsl_ntuple;
```

25.2 Creating ntuples

gsl_ntuple *__gsl_ntuple_create__(char *filename, void *ntuple_data, size_t size)

This function creates a new write-only ntuple file `filename` for ntuples of size `size` and returns a pointer to the newly created ntuple struct. Any existing file with the same name is truncated to zero length and overwritten. A pointer to memory for the current ntuple row `ntuple_data` must be supplied—this is used to copy ntuples in and out of the file.

25.3 Opening an existing ntuple file

gsl_ntuple *__gsl_ntuple_open__(char *filename, void *ntuple_data, size_t size)

This function opens an existing ntuple file `filename` for reading and returns a pointer to a corresponding ntuple struct. The ntuples in the file must have size `size`. A pointer to memory for the current ntuple row `ntuple_data` must be supplied—this is used to copy ntuples in and out of the file.

25.4 Writing ntuples

int __gsl_ntuple_write__(gsl_ntuple *ntuple)

This function writes the current ntuple `ntuple->ntuple_data` of size `ntuple->size` to the corresponding file.

int __gsl_ntuple_bookdata__(gsl_ntuple *ntuple)

This function is a synonym for `gsl_ntuple_write()`.

25.5 Reading ntuples

int **gsl_ntuple_read**(gsl_ntuple *ntuple)

This function reads the current row of the ntuple file for `ntuple` and stores the values in `ntuple->data`.

25.6 Closing an ntuple file

int **gsl_ntuple_close**(gsl_ntuple *ntuple)

This function closes the ntuple file `ntuple` and frees its associated allocated memory.

25.7 Histogramming ntuple values

Once an ntuple has been created its contents can be histogrammed in various ways using the function `gsl_ntuple_project()`. Two user-defined functions must be provided, a function to select events and a function to compute scalar values. The selection function and the value function both accept the ntuple row as a first argument and other parameters as a second argument.

type **gsl_ntuple_select_fn**

The selection function determines which ntuple rows are selected for histogramming. It is defined by the following struct:

```
typedef struct
  {
    int (* function) (void * ntuple_data, void * params);
    void * params;
  } gsl_ntuple_select_fn;
```

The struct component `function` should return a non-zero value for each ntuple row that is to be included in the histogram.

type **gsl_ntuple_value_fn**

The value function computes scalar values for those ntuple rows selected by the selection function:

```
typedef struct
  {
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  double (* function) (void * ntuple_data, void * params);
  void * params;
} gsl_ntuple_value_fn;
```

In this case the struct component `function` should return the value to be added to the histogram for the ntuple row.

int **gsl_ntuple_project**(gsl_histogram *h, gsl_ntuple *ntuple, gsl_ntuple_value_fn *value_func, gsl_ntuple_select_fn *select_func)

This function updates the histogram `h` from the ntuple `ntuple` using the functions `value_func` and `select_func`. For each ntuple row where the selection function `select_func` is non-zero the corresponding value of that row is computed using the function `value_func` and added to the histogram. Those ntuple rows where `select_func` returns zero are ignored. New entries are added to the histogram, so subsequent calls can be used to accumulate further data in the same histogram.

25.8 Examples

The following example programs demonstrate the use of ntuples in managing a large dataset. The first program creates a set of 10,000 simulated "events", each with 3 associated values (x, y, z). These are generated from a Gaussian distribution with unit variance, for demonstration purposes, and written to the ntuple file `test.dat`.

```
#include <gsl/gsl_ntuple.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

struct data
{
  double x;
  double y;
  double z;
};

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  struct data ntuple_row;
  int i;

  gsl_ntuple *ntuple
    = gsl_ntuple_create ("test.dat", &ntuple_row,
                         sizeof (ntuple_row));

  gsl_rng_env_setup ();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  for (i = 0; i < 10000; i++)
    {
      ntuple_row.x = gsl_ran_ugaussian (r);
      ntuple_row.y = gsl_ran_ugaussian (r);
      ntuple_row.z = gsl_ran_ugaussian (r);

      gsl_ntuple_write (ntuple);
    }

  gsl_ntuple_close (ntuple);
  gsl_rng_free (r);

  return 0;
}
```

The next program analyses the ntuple data in the file `test.dat`. The analysis procedure is to compute the squared-magnitude of each event, $E^2 = x^2 + y^2 + z^2$, and select only those which exceed a lower limit of 1.5. The selected events are then histogrammed using their E^2 values.

```
#include <math.h>
#include <gsl/gsl_ntuple.h>
#include <gsl/gsl_histogram.h>

struct data
{
  double x;
  double y;
  double z;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
};

int sel_func (void *ntuple_data, void *params);
double val_func (void *ntuple_data, void *params);

int
main (void)
{
  struct data ntuple_row;

  gsl_ntuple *ntuple
    = gsl_ntuple_open ("test.dat", &ntuple_row,
                       sizeof (ntuple_row));
  double lower = 1.5;

  gsl_ntuple_select_fn S;
  gsl_ntuple_value_fn V;

  gsl_histogram *h = gsl_histogram_alloc (100);
  gsl_histogram_set_ranges_uniform(h, 0.0, 10.0);

  S.function = &sel_func;
  S.params = &lower;

  V.function = &val_func;
  V.params = 0;

  gsl_ntuple_project (h, ntuple, &V, &S);
  gsl_histogram_fprintf (stdout, h, "%f", "%f");
  gsl_histogram_free (h);
  gsl_ntuple_close (ntuple);

  return 0;
}

int
sel_func (void *ntuple_data, void *params)
{
  struct data * data = (struct data *) ntuple_data;
  double x, y, z, E2, scale;
  scale = *(double *) params;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  x = data->x;
  y = data->y;
  z = data->z;

  E2 = x * x + y * y + z * z;

  return E2 > scale;
}

double
val_func (void *ntuple_data, void *params)
{
  (void)(params); /* avoid unused parameter warning */
  struct data * data = (struct data *) ntuple_data;
  double x, y, z;

  x = data->x;
  y = data->y;
  z = data->z;

  return x * x + y * y + z * z;
}
```

그림 25.1 shows the distribution of the selected events. Note the cut-off at the lower bound.

25.9 References and Further Reading

Further information on the use of ntuples can be found in the documentation for the CERN packages PAW and HBOOK (available online).

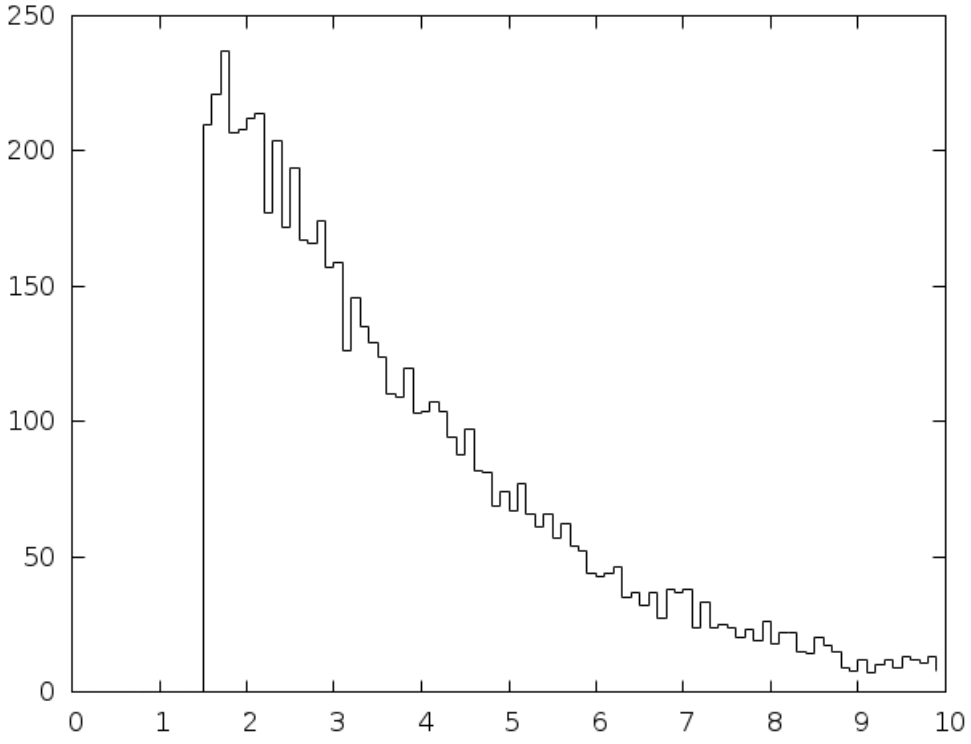


그림 25.1: Distribution of selected events

제 26 장

몬테카를로 적분

참고: 번역중

This chapter describes routines for multidimensional Monte Carlo integration. These include the traditional Monte Carlo method and adaptive algorithms such as VEGAS and MISER which use importance sampling and stratified sampling techniques. Each algorithm computes an estimate of a multidimensional definite integral of the form,

$$I = \int_{x_l}^{x_u} dx \int_{y_l}^{y_u} dy ... f(x, y, ...)$$

over a hypercubic region $((x_l, x_u), (y_l, y_u), ...)$ using a fixed number of function calls. The routines also provide a statistical estimate of the error on the result. This error estimate should be taken as a guide rather than as a strict error bound—random sampling of the region may not uncover all the important features of the function, resulting in an underestimate of the error.

The functions are defined in separate header files for each routine, `gsl_monte_plain.h`, `gsl_monte_miser.h` and `gsl_monte_vegas.h`.

26.1 Interface

All of the Monte Carlo integration routines use the same general form of interface. There is an allocator to allocate memory for control variables and workspace, a routine to initialize those control variables, the integrator itself, and a function to free the space when done.

Each integration function requires a random number generator to be supplied, and returns an estimate of the integral and its standard deviation. The accuracy of the result is determined by the number of function calls specified by the user. If a known level of accuracy is required this can be achieved by calling the integrator several times and averaging the individual results until the desired accuracy is obtained.

Random sample points used within the Monte Carlo routines are always chosen strictly within the integration region, so that endpoint singularities are automatically avoided.

The function to be integrated has its own datatype, defined in the header file `gsl_monte.h`.

type **gsl_monte_function**

This data type defines a general function with parameters for Monte Carlo integration.

`double (* f) (double * x, size_t dim, void * params)`	this function should return the value $f(x, params)$ for the argument `x` and parameters `params`, where `x` is an array of size `dim` giving the coordinates of the point where the function is to be evaluated.
`size_t dim`	the number of dimensions for `x`.
`void * params`	a pointer to the parameters of the function.

Here is an example for a quadratic function in two dimensions,

$$f(x, y) = ax^2 + bxy + cy^2$$

with $a = 3$, $b = 2$, $c = 1$. The following code defines a `gsl_monte_function F` which you could pass to an integrator:

```
struct my_f_params { double a; double b; double c; };

double
my_f (double x[], size_t dim, void * p) {
   struct my_f_params * fp = (struct my_f_params *)p;

   if (dim != 2)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      {
        fprintf (stderr, "error: dim != 2");
        abort ();
      }

   return  fp->a * x[0] * x[0]
            + fp->b * x[0] * x[1]
              + fp->c * x[1] * x[1];
}

gsl_monte_function F;
struct my_f_params params = { 3.0, 2.0, 1.0 };

F.f = &my_f;
F.dim = 2;
F.params = &params;
```

The function $f(x)$ can be evaluated using the following macro:

```
#define GSL_MONTE_FN_EVAL(F,x)
    (*((F)->f))(x,(F)->dim,(F)->params)
```

26.2 PLAIN Monte Carlo

The plain Monte Carlo algorithm samples points randomly from the integration region to estimate the integral and its error. Using this algorithm the estimate of the integral $E(f;N)$ for N randomly distributed points x_i is given by,

$$E(f;N) = V\langle f\rangle = \frac{V}{N}\sum_{i}^{N} f(x_i)$$

where V is the volume of the integration region. The error on this estimate $\sigma(E;N)$ is calculated from the estimated variance of the mean,

$$\sigma^2(E;N) = \frac{V^2}{N^2}\sum_{i}^{N}(f(x_i) - \langle f\rangle)^2.$$

For large N this variance decreases asymptotically as $(f)/N$, where (f) is the true variance of the function over the integration region. The error estimate itself should decrease as $\sigma(f)/\sqrt{N}$. The familiar law of errors decreasing as $1/\sqrt{N}$ applies—to reduce the error by a factor of 10

requires a 100-fold increase in the number of sample points.

The functions described in this section are declared in the header file `gsl_monte_plain.h`.

type **gsl_monte_plain_state**

This is a workspace for plain Monte Carlo integration

gsl_monte_plain_state ***gsl_monte_plain_alloc**(size_t dim)

This function allocates and initializes a workspace for Monte Carlo integration in `dim` dimensions.

int **gsl_monte_plain_init**(gsl_monte_plain_state *s)

This function initializes a previously allocated integration state. This allows an existing workspace to be reused for different integrations.

int **gsl_monte_plain_integrate**(gsl_monte_function *f, const double xl[], const double xu[], size_t dim, size_t calls, gsl_rng *r, gsl_monte_plain_state *s, double *result, double *abserr)

This routines uses the plain Monte Carlo algorithm to integrate the function `f` over the `dim`-dimensional hypercubic region defined by the lower and upper limits in the arrays `xl` and `xu`, each of size `dim`. The integration uses a fixed number of function calls `calls`, and obtains random sampling points using the random number generator `r`. A previously allocated workspace `s` must be supplied. The result of the integration is returned in `result`, with an estimated absolute error `abserr`.

void **gsl_monte_plain_free**(gsl_monte_plain_state *s)

This function frees the memory associated with the integrator state `s`.

26.3 MISER

The MISER algorithm of Press and Farrar is based on recursive stratified sampling. This technique aims to reduce the overall integration error by concentrating integration points in the regions of highest variance.

The idea of stratified sampling begins with the observation that for two disjoint regions a and b with Monte Carlo estimates of the integral $E_a(f)$ and $E_b(f)$ and variances $\sigma_a^2(f)$ and $\sigma_b^2(f)$, the variance (f) of the combined estimate $E(f) = \frac{1}{2}(E_a(f) + E_b(f))$ is given by,

$$(f) = \frac{\sigma_a^2(f)}{4N_a} + \frac{\sigma_b^2(f)}{4N_b}.$$

It can be shown that this variance is minimized by distributing the points such that,

$$\frac{N_a}{N_a + N_b} = \frac{\sigma_a}{\sigma_a + \sigma_b}.$$

Hence the smallest error estimate is obtained by allocating sample points in proportion to the standard deviation of the function in each sub-region.

The MISER algorithm proceeds by bisecting the integration region along one coordinate axis to give two sub-regions at each step. The direction is chosen by examining all d possible bisections and selecting the one which will minimize the combined variance of the two sub-regions. The variance in the sub-regions is estimated by sampling with a fraction of the total number of points available to the current step. The same procedure is then repeated recursively for each of the two half-spaces from the best bisection. The remaining sample points are allocated to the sub-regions using the formula for N_a and N_b. This recursive allocation of integration points continues down to a user-specified depth where each sub-region is integrated using a plain Monte Carlo estimate. These individual values and their error estimates are then combined upwards to give an overall result and an estimate of its error.

The functions described in this section are declared in the header file `gsl_monte_miser.h`.

type **gsl_monte_miser_state**

This workspace is used for MISER Monte Carlo integration

gsl_monte_miser_state ***gsl_monte_miser_alloc**(size_t dim)

This function allocates and initializes a workspace for Monte Carlo integration in `dim` dimensions. The workspace is used to maintain the state of the integration.

int **gsl_monte_miser_init**(gsl_monte_miser_state *s)

This function initializes a previously allocated integration state. This allows an existing workspace to be reused for different integrations.

int **gsl_monte_miser_integrate**(gsl_monte_function *f, const double xl[], const double xu[], size_t dim, size_t calls, gsl_rng *r, gsl_monte_miser_state *s, double *result, double *abserr)

This routines uses the MISER Monte Carlo algorithm to integrate the function `f` over the `dim`-dimensional hypercubic region defined by the lower and upper limits in the arrays `xl` and `xu`, each of size `dim`. The integration uses a fixed number of function calls `calls`, and obtains random sampling points using the random number generator `r`. A previously allocated workspace `s` must be supplied. The result of the integration is returned in `result`, with an estimated absolute error `abserr`.

void **gsl_monte_miser_free**(gsl_monte_miser_state *s)

This function frees the memory associated with the integrator state `s`.

The MISER algorithm has several configurable parameters which can be changed using the following two functions[1].

void **gsl_monte_miser_params_get**(const gsl_monte_miser_state *s, gsl_monte_miser_params *params)

This function copies the parameters of the integrator state into the user-supplied `params` structure.

void **gsl_monte_miser_params_set**(gsl_monte_miser_state *s, const gsl_monte_miser_params *params)

This function sets the integrator parameters based on values provided in the `params` structure.

Typically the values of the parameters are first read using `gsl_monte_miser_params_get()`, the necessary changes are made to the fields of the `params` structure, and the values are copied back into the integrator state using `gsl_monte_miser_params_set()`. The functions use the `gsl_monte_miser_params` structure which contains the following fields:

type **gsl_monte_miser_params**

double **estimate_frac**

This parameter specifies the fraction of the currently available number of function calls which are allocated to estimating the variance at each recursive step. The default value is 0.1.

size_t **min_calls**

This parameter specifies the minimum number of function calls required for each estimate of the variance. If the number of function calls allocated to the estimate using `estimate_frac` falls below `min_calls` then `min_calls` are used instead. This ensures that each estimate maintains a reasonable level of accuracy. The default value of `min_calls` is `16 * dim`.

size_t **min_calls_per_bisection**

This parameter specifies the minimum number of function calls required to proceed with a bisection step. When a recursive step has fewer calls available than `min_calls_per_bisection` it performs a plain Monte Carlo estimate of the current sub-region and terminates its branch of the recursion. The default value of this parameter is `32 * min_calls`.

[1] The previous method of accessing these fields directly through the `gsl_monte_miser_state` struct is now deprecated.

double **alpha**

This parameter controls how the estimated variances for the two sub-regions of a bisection are combined when allocating points. With recursive sampling the overall variance should scale better than $1/N$, since the values from the sub-regions will be obtained using a procedure which explicitly minimizes their variance. To accommodate this behavior the MISER algorithm allows the total variance to depend on a scaling parameter α,

$$(f) = \frac{\sigma_a}{N_a^\alpha} + \frac{\sigma_b}{N_b^\alpha}.$$

The authors of the original paper describing MISER recommend the value $\alpha = 2$ as a good choice, obtained from numerical experiments, and this is used as the default value in this implementation.

double **dither**

This parameter introduces a random fractional variation of size `dither` into each bisection, which can be used to break the symmetry of integrands which are concentrated near the exact center of the hypercubic integration region. The default value of dither is zero, so no variation is introduced. If needed, a typical value of `dither` is 0.1.

26.4 VEGAS

The VEGAS algorithm of Lepage is based on importance sampling. It samples points from the probability distribution described by the function $|f|$, so that the points are concentrated in the regions that make the largest contribution to the integral.

In general, if the Monte Carlo integral of f is sampled with points distributed according to a probability distribution described by the function g, we obtain an estimate $E_g(f; N)$,

$$E_g(f; N) = E(f/g; N)$$

with a corresponding variance,

$$_g(f; N) = (f/g; N)$$

If the probability distribution is chosen as $g = |f|/I(|f|)$ then it can be shown that the variance $V_g(f; N)$ vanishes, and the error in the estimate will be zero. In practice it is not possible to sample from the exact distribution g for an arbitrary function, so importance sampling algorithms aim to produce efficient approximations to the desired distribution.

The VEGAS algorithm approximates the exact distribution by making a number of passes over the integration region while histogramming the function f. Each histogram is used to define a sampling distribution for the next pass. Asymptotically this procedure converges to the desired distribution. In order to avoid the number of histogram bins growing like K^d the probability distribution is approximated by a separable function: $g(x_1, x_2, \ldots) = g_1(x_1)g_2(x_2)\ldots$ so that the number of bins required is only Kd. This is equivalent to locating the peaks of the function from the projections of the integrand onto the coordinate axes. The efficiency of VEGAS depends on the validity of this assumption. It is most efficient when the peaks of the integrand are well-localized. If an integrand can be rewritten in a form which is approximately separable this will increase the efficiency of integration with VEGAS.

VEGAS incorporates a number of additional features, and combines both stratified sampling and importance sampling. The integration region is divided into a number of "boxes", with each box getting a fixed number of points (the goal is 2). Each box can then have a fractional number of bins, but if the ratio of bins-per-box is less than two, Vegas switches to a kind variance reduction (rather than importance sampling).

type **gsl_monte_vegas_state**

This workspace is used for VEGAS Monte Carlo integration

gsl_monte_vegas_state ***gsl_monte_vegas_alloc**(size_t dim)

This function allocates and initializes a workspace for Monte Carlo integration in `dim` dimensions. The workspace is used to maintain the state of the integration.

int **gsl_monte_vegas_init**(gsl_monte_vegas_state *s)

This function initializes a previously allocated integration state. This allows an existing workspace to be reused for different integrations.

int **gsl_monte_vegas_integrate**(gsl_monte_function *f, double xl[], double xu[], size_t dim, size_t calls, gsl_rng *r, gsl_monte_vegas_state *s, double *result, double *abserr)

This routines uses the VEGAS Monte Carlo algorithm to integrate the function `f` over the `dim`-dimensional hypercubic region defined by the lower and upper limits in the arrays `xl` and `xu`, each of size `dim`. The integration uses a fixed number of function calls `calls`, and obtains random sampling points using the random number generator `r`. A previously allocated workspace `s` must be supplied. The result of the integration is returned in `result`, with an estimated absolute error `abserr`. The result and its error estimate are based on a weighted average of independent samples. The chi-squared per degree of freedom for the weighted average is returned via the state struct component, `s->chisq`, and must be consistent with 1 for the weighted average to be reliable.

void **gsl_monte_vegas_free**(gsl_monte_vegas_state *s)

This function frees the memory associated with the integrator state `s`.

The VEGAS algorithm computes a number of independent estimates of the integral internally, according to the `iterations` parameter described below, and returns their weighted average. Random sampling of the integrand can occasionally produce an estimate where the error is zero, particularly if the function is constant in some regions. An estimate with zero error causes the weighted average to break down and must be handled separately. In the original Fortran implementations of VEGAS the error estimate is made non-zero by substituting a small value (typically `1e-30`). The implementation in GSL differs from this and avoids the use of an arbitrary constant—it either assigns the value a weight which is the average weight of the preceding estimates or discards it according to the following procedure,

- current estimate has zero error, weighted average has finite error

  The current estimate is assigned a weight which is the average weight of the preceding estimates.

- current estimate has finite error, previous estimates had zero error

  The previous estimates are discarded and the weighted averaging procedure begins with the current estimate.

- current estimate has zero error, previous estimates had zero error

  The estimates are averaged using the arithmetic mean, but no error is computed.

The convergence of the algorithm can be tested using the overall chi-squared value of the results, which is available from the following function:

double **gsl_monte_vegas_chisq**(const gsl_monte_vegas_state *s)

This function returns the chi-squared per degree of freedom for the weighted estimate of the integral. The returned value should be close to 1. A value which differs significantly from 1 indicates that the values from different iterations are inconsistent. In this case the weighted error will be under-estimated, and further iterations of the algorithm are needed to obtain reliable results.

void **gsl_monte_vegas_runval**(const gsl_monte_vegas_state *s, double *result, double *sigma)

This function returns the raw (unaveraged) values of the integral `result` and its error `sigma` from the most recent iteration of the algorithm.

The VEGAS algorithm is highly configurable. Several parameters can be changed using the following two functions.

void **gsl_monte_vegas_params_get**(const gsl_monte_vegas_state *s, gsl_monte_vegas_params *params)

This function copies the parameters of the integrator state into the user-supplied `params` structure.

void **gsl_monte_vegas_params_set**(gsl_monte_vegas_state *s, const gsl_monte_vegas_params *params)

This function sets the integrator parameters based on values provided in the `params` structure.

Typically the values of the parameters are first read using `gsl_monte_vegas_params_get()`, the necessary changes are made to the fields of the `params` structure, and the values are copied back into the integrator state using `gsl_monte_vegas_params_set()`. The functions use the `gsl_monte_vegas_params` structure which contains the following fields:

type **gsl_monte_vegas_params**

double **alpha**

The parameter `alpha` controls the stiffness of the rebinning algorithm. It is typically set between one and two. A value of zero prevents rebinning of the grid. The default value is 1.5.

size_t **iterations**

The number of iterations to perform for each call to the routine. The default value is 5 iterations.

int **stage**

Setting this determines the stage of the calculation. Normally, `stage = 0` which begins with a new uniform grid and empty weighted average. Calling VEGAS with `stage = 1` retains the grid from the previous run but discards the weighted average, so that one can "tune" the grid using a relatively small number of points and then do a large run with `stage = 1` on the optimized grid. Setting `stage = 2` keeps the grid and the weighted average from the previous run, but may increase (or decrease) the number of histogram bins in the grid depending on the number of calls available. Choosing `stage = 3` enters at the main loop, so that nothing is changed, and is equivalent to performing additional iterations in a previous call.

int **mode**

The possible choices are `GSL_VEGAS_MODE_IMPORTANCE`, `GSL_VEGAS_MODE_STRATIFIED`, `GSL_VEGAS_MODE_IMPORTANCE_ONLY`. This determines whether VEGAS will use importance sampling or stratified sampling, or whether it can pick on its own. In low dimensions VEGAS uses strict stratified sampling (more precisely, stratified sampling is chosen if there are fewer than 2 bins per box).

int **verbose**

FILE ***ostream**

These parameters set the level of information printed by VEGAS. All information is written to the stream `ostream`. The default setting of `verbose` is `-1`, which turns off all output. A `verbose` value of `0` prints summary information about the weighted average and final result, while a value of `1` also displays the grid coordinates. A value of `2` prints information from the rebinning procedure for each iteration.

The above fields and the `chisq` value can also be accessed directly in the `gsl_monte_vegas_state` but such use is deprecated.

26.5 Examples

The example program below uses the Monte Carlo routines to estimate the value of the following 3-dimensional integral from the theory of random walks,

$$I = \int_{-\pi}^{+\pi} \frac{dk_x}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_y}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_z}{2\pi} \frac{1}{(1 - \cos(k_x)\cos(k_y)\cos(k_z))}.$$

The analytic value of this integral can be shown to be $I = \Gamma(1/4)^4/(4\pi^3) = 1.393203929685676859\ldots$. The integral gives the mean time spent at the origin by a random walk on a body-centered cubic lattice in three dimensions.

For simplicity we will compute the integral over the region $(0,0,0)$ to (π,π,π) and multiply by 8 to obtain the full result. The integral is slowly varying in the middle of the region but has integrable singularities at the corners $(0,0,0)$, $(0,\pi,\pi)$, $(\pi,0,\pi)$ and $(\pi,\pi,0)$. The Monte Carlo routines only select points which are strictly within the integration region and so no special measures are needed to avoid these singularities.

```
#include <stdlib.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_monte.h>
#include <gsl/gsl_monte_plain.h>
#include <gsl/gsl_monte_miser.h>
#include <gsl/gsl_monte_vegas.h>

/* Computation of the integral,

     I = int (dx dy dz)/(2pi)^3  1/(1-cos(x)cos(y)cos(z))

   over (-pi,-pi,-pi) to (+pi, +pi, +pi).  The exact answer
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   is Gamma(1/4)^4/(4 pi^3).  This example is taken from
   C.Itzykson, J.M.Drouffe, "Statistical Field Theory -
   Volume 1", Section 1.1, p21, which cites the original
   paper M.L.Glasser, I.J.Zucker, Proc.Natl.Acad.Sci.USA 74
   1800 (1977) */

/* For simplicity we compute the integral over the region
   (0,0,0) -> (pi,pi,pi) and multiply by 8 */

double exact = 1.393203929685676859184246260325 5;

double
g (double *k, size_t dim, void *params)
{
  (void)(dim); /* avoid unused parameter warnings */
  (void)(params);
  double A = 1.0 / (M_PI * M_PI * M_PI);
  return A / (1.0 - cos (k[0]) * cos (k[1]) * cos (k[2]));
}

void
display_results (char *title, double result, double error)
{
  printf ("%s ==================\n", title);
  printf ("result = % .6f\n", result);
  printf ("sigma  = % .6f\n", error);
  printf ("exact  = % .6f\n", exact);
  printf ("error  = % .6f = %.2g sigma\n", result - exact,
          fabs (result - exact) / error);
}

int
main (void)
{
  double res, err;

  double xl[3] = { 0, 0, 0 };
  double xu[3] = { M_PI, M_PI, M_PI };

  const gsl_rng_type *T;
  gsl_rng *r;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
gsl_monte_function G = { &g, 3, 0 };

size_t calls = 500000;

gsl_rng_env_setup ();

T = gsl_rng_default;
r = gsl_rng_alloc (T);

{
  gsl_monte_plain_state *s = gsl_monte_plain_alloc (3);
  gsl_monte_plain_integrate (&G, xl, xu, 3, calls, r, s,
                             &res, &err);
  gsl_monte_plain_free (s);

  display_results ("plain", res, err);
}

{
  gsl_monte_miser_state *s = gsl_monte_miser_alloc (3);
  gsl_monte_miser_integrate (&G, xl, xu, 3, calls, r, s,
                             &res, &err);
  gsl_monte_miser_free (s);

  display_results ("miser", res, err);
}

{
  gsl_monte_vegas_state *s = gsl_monte_vegas_alloc (3);

  gsl_monte_vegas_integrate (&G, xl, xu, 3, 10000, r, s,
                             &res, &err);
  display_results ("vegas warm-up", res, err);

  printf ("converging...\n");

  do
    {
      gsl_monte_vegas_integrate (&G, xl, xu, 3, calls/5, r, s,
                                 &res, &err);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        printf ("result = % .6f sigma = % .6f "
                "chisq/dof = %.1f\n", res, err, gsl_monte_vegas_chisq (s));
      }
    while (fabs (gsl_monte_vegas_chisq (s) - 1.0) > 0.5);

    display_results ("vegas final", res, err);

    gsl_monte_vegas_free (s);
  }

  gsl_rng_free (r);

  return 0;
}
```

With 500,000 function calls the plain Monte Carlo algorithm achieves a fractional error of 1%. The estimated error sigma is roughly consistent with the actual error–the computed result differs from the true result by about 1.4 standard deviations:

```
plain ==================
result =  1.412209
sigma  =  0.013436
exact  =  1.393204
error  =  0.019005 = 1.4 sigma
```

The MISER algorithm reduces the error by a factor of four, and also correctly estimates the error:

```
miser ==================
result =  1.391322
sigma  =  0.003461
exact  =  1.393204
error  = -0.001882 = 0.54 sigma
```

In the case of the VEGAS algorithm the program uses an initial warm-up run of 10,000 function calls to prepare, or "warm up", the grid. This is followed by a main run with five iterations of 100,000 function calls. The chi-squared per degree of freedom for the five iterations are checked for consistency with 1, and the run is repeated if the results have not converged. In this case the estimates are consistent on the first pass:

```
vegas warm-up ==================
result =  1.392673
sigma  =  0.003410
exact  =  1.393204
error  = -0.000531 = 0.16 sigma
converging...
result =  1.393281 sigma =  0.000362 chisq/dof = 1.5
vegas final ================
result =  1.393281
sigma  =  0.000362
exact  =  1.393204
error  =  0.000077 = 0.21 sigma
```

If the value of `chisq` had differed significantly from 1 it would indicate inconsistent results, with a correspondingly underestimated error. The final estimate from VEGAS (using a similar number of function calls) is significantly more accurate than the other two algorithms.

26.6 References and Further Reading

The MISER algorithm is described in the following article by Press and Farrar,

- W.H. Press, G.R. Farrar, Recursive Stratified Sampling for Multidimensional Monte Carlo Integration, Computers in Physics, v4 (1990), pp190–195.

The VEGAS algorithm is described in the following papers,

- G.P. Lepage, A New Algorithm for Adaptive Multidimensional Integration, Journal of Computational Physics 27, 192–203, (1978)
- G.P. Lepage, VEGAS: An Adaptive Multi-dimensional Integration Program, Cornell preprint CLNS 80-447, March 1980

제 27 장

담금질 기법

어떤 문제를 풀고자 할 때, 풀고자 하는 공간의 정보가 부족하거나 연속적이지 않으면, 뉴턴 방법과 같은 방법은 사용할 수 없습니다(야코비안 같은 미분 행렬 계산이 필요합니다.). 이런 경우에 확률적 탐색 기법를 시도해 볼 수 있습니다. 특히 이 방법들은, 조합적 최적화 문제를 푸는 데 빈번히 사용됩니다. 이러한 문제의 예로 외판원 문제가 있습니다.

문제 공간에서 해를 구한다는 것은 실수 값을 가지는 에너지 함수 (비용 함수라고도 합니다)의 값이 최소화 되는 지점을 찾는 것을 의미합니다. 담금질 기법(Simlated annealing)은 지역적 최소화를 피할 수 있어 일반적으로 좋은 결과를 내놓는 최소화 기법입니다. 이 방법은 온도(temperature)를 낮출 수 있는 무작위 전이를 공간 위에서 진행합니다. 각 걸음의 진행 확률은 볼츠만 분포에 의해 결정됩니다.

이 단원에서 기술된 함수들은 `gsl_siman.h` 헤더파일에 기록되어 있습니다.

27.1 담금질 알고리즘

담금질 알고리즘은 기본적으로 문제 공간에서 무작위 방향 중 에너지를 최소화하는 경향으로 상태를 확률적으로 바꾸며 해를 탐색합니다. 이러한 전이 확률은 볼츠만 분포을 기반으로 결정됩니다.

$$p = e^{-(E_{i+1} - E_i)/(kT)}$$

$E_{i+1} > E_i$ 인 조건에서 성립하며, $E_{i+1} \leq E_i$ 인 경우 $p = 1$ 로 정의합니다.

다시 말해, 새로운 에너지 단계가 낮으면 전이가 일어나고, 새로운 에너지 단계가 높으면 확률에 기반해 전이가 결정됩니다. 이 확률은 온도 T 에 비례하고, 에너지 차이 $E_{i+1} - E_i$ 에 반비례합니다.

온도 T 는 초기 단계에서 결정되며 일반적으로 적절한 높은 값을 가집니다. 무작위 전이는 이 온도 값에 의존하게 됩니다. 전이 과정에서 온도는 설계된 냉각 절차에 따라 천천히 낮아집니다. 예를 들어 $T -> T/\mu_T$ 으로 μ_T 를 1보다 조금 더 큰 수로 설정해 볼 수 있습니다.

이러한 높은 에너지 상태로 전이할 수 있는 확률의 존재는 다양한 상황에서 담금질 기법이 지역적 최소값에 빠지지 않도록 해, 최적의 값을 찾을 수 있도록 도와줍니다.

27.2 담금질 함수

void **gsl_siman_solve**(const gsl_rng *r, void *x0_p, gsl_siman_Efunc_t Ef, gsl_siman_step_t take_step, gsl_siman_metric_t distance, gsl_siman_print_t print_position, gsl_siman_copy_t copyfunc, gsl_siman_copy_construct_t copy_constructor, gsl_siman_destroy_t destructor, size_t element_size, gsl_siman_params_t params)

주어진 공간에서 담금질 기법을 이용해 해를 찾습니다. 공간은 `Ef` 와 `distance` 로 정의됩니다. 각 전이 단계는 주어진 인자 중 난수 생성자 `r` 과 함수 `take_step` 에 의해 결정됩니다.

이 계의 시작 조건 설정은 `x0_p` 에 의해 주어집니다. 이 명령어 집합은 2개의 방법으로 현재 설정을 갱신하는데, **고정 크기 방법** (fixed-size mode)과 **동적 크기 방법** (variable-size mode)이 있습니다. 고정 크기 방법에서는 설정이 메모리 상의 `element_size` 크기의 한 공간에 저장되고, C의 표준 함수 `malloc()`, `memcpy()`, `free()` 들에 의해 내부에서 이 설정의 복사본들의 생성, 복사, 소멸이 진행됩니다. 이 고정 크기 방법 상태에서는 함수 포인터 `copyfunc`, `copy_constructor` 그리고 `destructor` 이 `null` 값을 가지게 됩니다. 반면, 동적 크기 방법에서는 `copyfunc`, `copy_constructor` 그리고 `destructor` 들이 `malloc()`, `memcpy()`, `free()` 대신에 쓰입니다. 이 경우 `element_size` 값은 0으로 주어집니다.

`params` 구조체 변수(세부 구조는 후술)는 제공된 온도 계획과 수정 가능한 변수들로 이루어지는 실행을 제어합니다.

종료 시, 알고리즘 과정을 통해 얻어진 최적의 결과가 `x0_p` 에 저장됩니다. 만약, 담금질 기법이 성공적이었다면, 이는 문제 공간의 최적점에 대해 좋은 근사를 나타내줍니다.

만약, 함수 포인터 `print_position` 가 `NULL` 값이 아니라면, 디버그 기록이 `stdout` 에 출력될 것입니다. 이는 다음과 같은 행을 가집니다:

```
#-iter  #-evals  temperature position energy   best_energy
```

함수 `print_position` 의 출력 값도 같이 출력됩니다. 만약 `print_position` 가 `NULL` 이라면, 아무 값도 출력 되지 않습니다.

이 라이브러리의 담금질 알고리즘 명령어 집합은 사용자 정의 함수들을 필요로 합니다. 해당 함수들은 풀고자 하는 문제 공간과 에너지 함수를 정의합니다. 이러한 함수들을 사용자가 직접 다음의 초안의 형태로 정의해 풀고자하는 문제에 적절한 함수들을 스스로 선택해 알고리즘에 사용할 수 있습니다.

type **gsl_siman_Efunc_t**

현재 상태 `xp` 에 대한 에너지를 반환해야 합니다.

```
double (*gsl_siman_Efunc_t) (void *xp)
```

type **gsl_siman_step_t**

현재 상태 xp 를 난수 발생자 r 와 단계별 최대 이동거리 step_size 를 이용해 수정해야 합니다.

```
void (*gsl_siman_step_t) (const gsl_rng *r, void *xp, double step_size)
```

type **gsl_siman_metric_t**

주어진 두 개의 상태 xp 와 yp 사이의 거리를 반환해야 합니다.

```
double (*gsl_siman_metric_t) (void * xp, void * yp)
```

type **gsl_siman_print_t**

주어진 상태 xp 의 설정값을 출력해야 합니다.

```
void (*gsl_siman_print_t) (void *xp)
```

type **gsl_siman_copy_t**

설정 값을 source 에서 dest 로 복사해야 합니다.

```
void (*gsl_siman_copy_t) (voi d *souirce, coid *dest)
```

type **gsl_siman_copy_construct_t**

설정 xp 의 새 복사본을 만들어야 합니다.

```
void (*gsl_siman_copy_construct_t) (void *xp)
```

type **gsl_siman_destroy_t**

설정 xp 를 삭제해야 합니다. 다시말해, xp 가 들어있는 메모리를 해제해야 합니다.

```
void (*gsl_siman_destroy_t) (void *xp)
```

type **gsl_siman_params_t**

gsl_siman_solve() 의 실행을 제어하는 인자들입니다. 이 구조체는 탐색, 에너지 함수, 단계 함수와 초기 가정을 제어하는 데 필요한 모든 정보를 담고 있습니다.

int n_tries	각 단계에서 시도할 지점의 수.
int iters_fixed_T	각 온도별 반복 횟수.
double step_size	각 무작위 전이에서 최대 크기.
double k, t_initial, mu_t, t_min	볼츠만 분포 인자들과 냉각 절차 인자.

27.3 예제

GSL의 담금질 알고리즘 구현은 그다지 세련된 구현체가 아닙니다. 상당히 원시적인 형태로 구현되어 있습니다. 이는 의도된 사항입니다. 이 구현체는 C로 작성해 C에서 호출함을 염두에 두고 개발되었으며, 동시에 다양한 응용 가능성에 목적을 두고 있기 때문입니다. 그렇기에 여기서 라이브러리의 사용자들이 개발하는 응용 프로그램에 약간의 수정을 거쳐 적용할 수 있는 여러 예시들을 제공할 것입니다. 이는 다양한 구현을 좀 더 쉽게 할 수 있도록 도와줄 것입니다.

27.3.1 Trivial 예제

첫번째 예제로, 1차원 직교 좌표계에서 감쇠하는 sine 함수를 에너지 함수로 둔 상황을 살펴봅시다. 이 공간은 많은 지역적 최소값이 존재합니다. 하지만 전역 최소값은 1개만이 존재합니다. 1.0과 1.5사이에 이 값이 존재합니다. 초기 추정은 15.5입니다. 전역 최소값으로 부터 사이에 여러 지역적 최소값이 존재하는 지점입니다.

```
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <gsl/gsl_siman.h>

/* set up parameters for this simulated annealing run */

/* how many points do we try before stepping */
#define N_TRIES 200

/* how many iterations for each T? */
#define ITERS_FIXED_T 1000

/* max step size in random walk */
#define STEP_SIZE 1.0

/* Boltzmann constant */
#define K 1.0

/* initial temperature */
#define T_INITIAL 0.008

/* damping factor for temperature */
#define MU_T 1.003
#define T_MIN 2.0e-6
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
gsl_siman_params_t params
  = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,
     K, T_INITIAL, MU_T, T_MIN};

/* now some functions to test in one dimension */
double E1(void *xp)
{
  double x = * ((double *) xp);

  return exp(-pow((x-1.0),2.0))*sin(8*x);
}

double M1(void *xp, void *yp)
{
  double x = *((double *) xp);
  double y = *((double *) yp);

  return fabs(x - y);
}

void S1(const gsl_rng * r, void *xp, double step_size)
{
  double old_x = *((double *) xp);
  double new_x;

  double u = gsl_rng_uniform(r);
  new_x = u * 2 * step_size - step_size + old_x;

  memcpy(xp, &new_x, sizeof(new_x));
}

void P1(void *xp)
{
  printf ("%12g", *((double *) xp));
}

int
main(void)
{
  const gsl_rng_type * T;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_rng * r;

  double x_initial = 15.5;

  gsl_rng_env_setup();

  T = gsl_rng_default;
  r = gsl_rng_alloc(T);

  gsl_siman_solve(r, &x_initial, E1, S1, M1, P1,
                  NULL, NULL, NULL,
                  sizeof(double), params);

  gsl_rng_free (r);
  return 0;
}
```

그림 27.1 는 siman_test 를 다음과 같이 실행한 결과입니다.

```
$./siman_test | awk '!/^#/ {print $1, $4}'
 | graph -y 1.34 1.4 -W0 -X generation -Y position
 | plot -Tps > siman-test.eps
```

그림 27.2 는 siman_test 다음과 같이 실행한 결과입니다.

```
$./siman_test | awk '!/^#/ {print $1, $5}'
 | graph -y -0.88 -0.83 -W0 -X generation -Y energy
 | plot -Tps > siman-energy.eps
```

27.3.2 외판원 문제

외판원 문제는 고전적인 조합적 최적화 문제입니다. Mark Galassi는 미국 남부 도시 12개의 좌표를 기반으로 아주 간단한 형태의 외판원 문제를 제시했습니다. 이 문제는 엄밀히 말해 비행기를 탄 외판원 문제입니다. 도시와 도시 사이의 거리를 기반으로 하고, 실질적인 차량의 이동거리를 기반으로 하지 않았기 때문입니다. 때문에, 지오이드 거리[1] 를 적용하지 않았습니다.

전체 코드는 다음과 같습니다.

[1] 해발고도를 재는 기준이 되는 지구의 구형 높이 입니다(*).

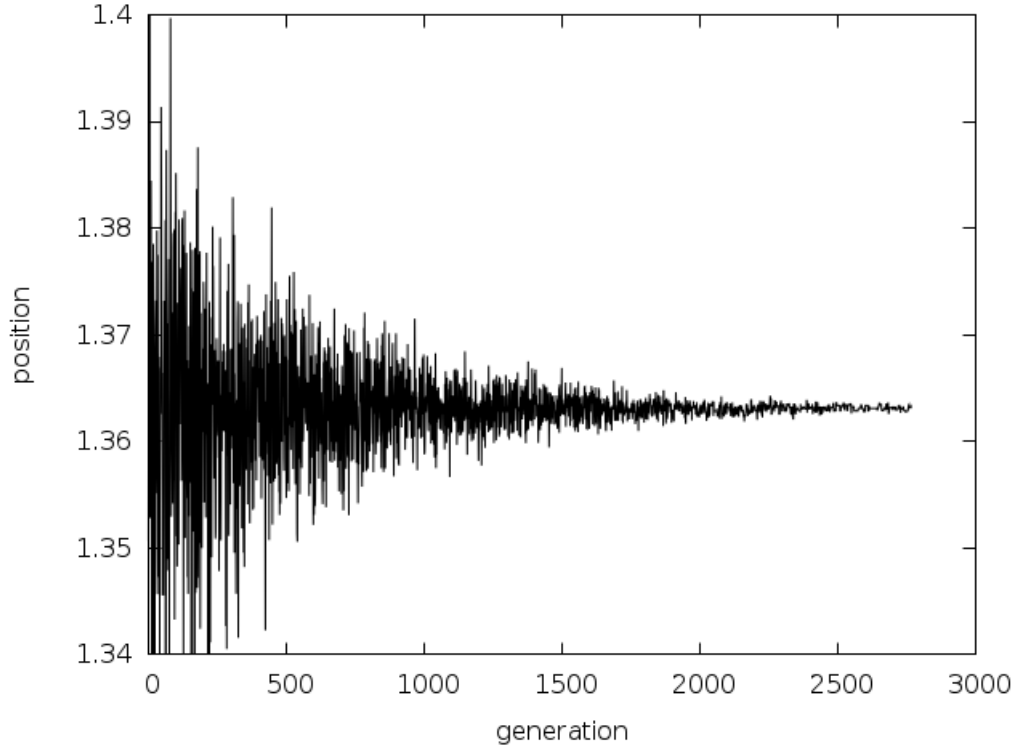


그림 27.1: 담금질 방법 실행 예시: 높은 온도에서(그래프 시작지점)는 해가 여러 값으로 발산하는 것을 볼 수 있습니다. 하지만 낮은 온도에서는 수렴합니다.

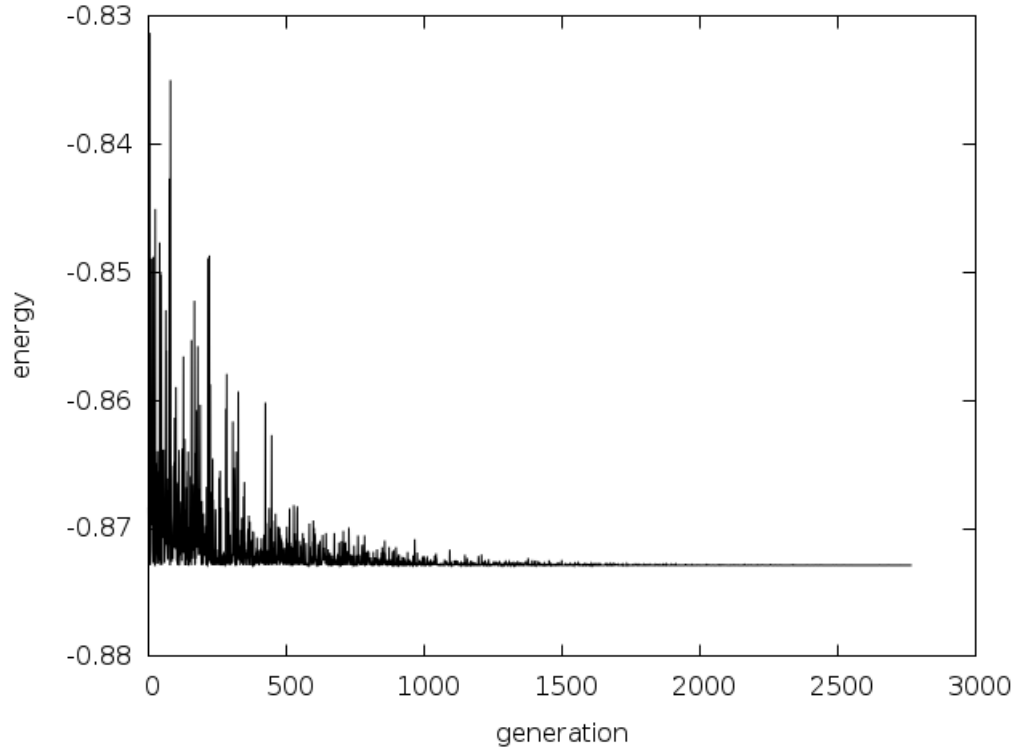


그림 27.2: 담금질 방법 에너지 vs 시행 횟수

```
/* siman/siman_tsp.c
 *
 * Copyright (C) 1996, 1997, 1998, 1999, 2000 Mark Galassi
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
 */

#include <config.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_siman.h>
#include <gsl/gsl_ieee_utils.h>

/* set up parameters for this simulated annealing run */

#define N_TRIES 200             /* how many points do we try before stepping */
#define ITERS_FIXED_T 2000      /* how many iterations for each T? */
#define STEP_SIZE 1.0           /* max step size in random walk */
#define K 1.0                   /* Boltzmann constant */
#define T_INITIAL 5000.0        /* initial temperature */
#define MU_T 1.002              /* damping factor for temperature */
#define T_MIN 5.0e-1

gsl_siman_params_t params = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,
                             K, T_INITIAL, MU_T, T_MIN};
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
struct s_tsp_city {
  const char * name;
  double lat, longitude;        /* coordinates */
};
typedef struct s_tsp_city Stsp_city;

void prepare_distance_matrix(void);
void exhaustive_search(void);
void print_distance_matrix(void);
double city_distance(Stsp_city c1, Stsp_city c2);
double Etsp(void *xp);
double Mtsp(void *xp, void *yp);
void Stsp(const gsl_rng * r, void *xp, double step_size);
void Ptsp(void *xp);

/* in this table, latitude and longitude are obtained from the US
   Census Bureau, at http://www.census.gov/cgi-bin/gazetteer */

Stsp_city cities[] = {{"Santa Fe",    35.68,   105.95},
                      {"Phoenix",     33.54,   112.07},
                      {"Albuquerque", 35.12,   106.62},
                      {"Clovis",      34.41,   103.20},
                      {"Durango",     37.29,   107.87},
                      {"Dallas",      32.79,    96.77},
                      {"Tesuque",     35.77,   105.92},
                      {"Grants",      35.15,   107.84},
                      {"Los Alamos",  35.89,   106.28},
                      {"Las Cruces",  32.34,   106.76},
                      {"Cortez",      37.35,   108.58},
                      {"Gallup",      35.52,   108.74}};

#define N_CITIES (sizeof(cities)/sizeof(Stsp_city))

double distance_matrix[N_CITIES][N_CITIES];

/* distance between two cities */
double city_distance(Stsp_city c1, Stsp_city c2)
{
  const double earth_radius = 6375.000; /* 6000KM approximately */
  /* sin and cos of lat and long; must convert to radians */
  double sla1 = sin(c1.lat*M_PI/180), cla1 = cos(c1.lat*M_PI/180),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    slo1 = sin(c1.longitude*M_PI/180), clo1 = cos(c1.longitude*M_PI/180);
  double sla2 = sin(c2.lat*M_PI/180), cla2 = cos(c2.lat*M_PI/180),
    slo2 = sin(c2.longitude*M_PI/180), clo2 = cos(c2.longitude*M_PI/180);

  double x1 = cla1*clo1;
  double x2 = cla2*clo2;

  double y1 = cla1*slo1;
  double y2 = cla2*slo2;

  double z1 = sla1;
  double z2 = sla2;

  double dot_product = x1*x2 + y1*y2 + z1*z2;

  double angle = acos(dot_product);

  /* distance is the angle (in radians) times the earth radius */
  return angle*earth_radius;
}

/* energy for the travelling salesman problem */
double Etsp(void *xp)
{
  /* an array of N_CITIES integers describing the order */
  int *route = (int *) xp;
  double E = 0;
  unsigned int i;

  for (i = 0; i < N_CITIES; ++i) {
    /* use the distance_matrix to optimize this calculation; it had
       better be allocated!! */
    E += distance_matrix[route[i]][route[(i + 1) % N_CITIES]];
  }

  return E;
}

double Mtsp(void *xp, void *yp)
{
  int *route1 = (int *) xp, *route2 = (int *) yp;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  double distance = 0;
  unsigned int i;

  for (i = 0; i < N_CITIES; ++i) {
    distance += ((route1[i] == route2[i]) ? 0 : 1);
  }

  return distance;
}

/* take a step through the TSP space */
void Stsp(const gsl_rng * r, void *xp, double step_size)
{
  int x1, x2, dummy;
  int *route = (int *) xp;

  step_size = 0 ; /* prevent warnings about unused parameter */

  /* pick the two cities to swap in the matrix; we leave the first
     city fixed */
  x1 = (gsl_rng_get (r) % (N_CITIES-1)) + 1;
  do {
    x2 = (gsl_rng_get (r) % (N_CITIES-1)) + 1;
  } while (x2 == x1);

  dummy = route[x1];
  route[x1] = route[x2];
  route[x2] = dummy;
}

void Ptsp(void *xp)
{
  unsigned int i;
  int *route = (int *) xp;
  printf("  [");
  for (i = 0; i < N_CITIES; ++i) {
    printf(" %d ", route[i]);
  }
  printf("]  ");
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
int main(void)
{
  int x_initial[N_CITIES];
  unsigned int i;

  const gsl_rng * r = gsl_rng_alloc (gsl_rng_env_setup()) ;

  gsl_ieee_env_setup ();

  prepare_distance_matrix();

  /* set up a trivial initial route */
  printf("# initial order of cities:\n");
  for (i = 0; i < N_CITIES; ++i) {
    printf("# \"%s\"\n", cities[i].name);
    x_initial[i] = i;
  }

  printf("# distance matrix is:\n");
  print_distance_matrix();

  printf("# initial coordinates of cities (longitude and latitude)\n");
  /* this can be plotted with */
  /* ./siman_tsp > hhh ; grep city_coord hhh | awk '{print $2 "   " $3}' | xyplot -ps -d "xy" > c.
↪eps */
  for (i = 0; i < N_CITIES+1; ++i) {
    printf("###initial_city_coord: %g %g \"%s\"\n",
           -cities[x_initial[i % N_CITIES]].longitude,
           cities[x_initial[i % N_CITIES]].lat,
           cities[x_initial[i % N_CITIES]].name);
  }

/*   exhaustive_search(); */

  gsl_siman_solve(r, x_initial, Etsp, Stsp, Mtsp, Ptsp, NULL, NULL, NULL,
                  N_CITIES*sizeof(int), params);

  printf("# final order of cities:\n");
  for (i = 0; i < N_CITIES; ++i) {
    printf("# \"%s\"\n", cities[x_initial[i]].name);
  }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  printf("# final coordinates of cities (longitude and latitude)\n");
  /* this can be plotted with */
  /* ./siman_tsp > hhh ; grep city_coord hhh | awk '{print $2 "   " $3}' | xyplot -ps -d "xy" > c.
↪eps */
  for (i = 0; i < N_CITIES+1; ++i) {
    printf("###final_city_coord: %g %g %s\n",
           -cities[x_initial[i % N_CITIES]].longitude,
           cities[x_initial[i % N_CITIES]].lat,
           cities[x_initial[i % N_CITIES]].name);
  }

  printf("# ");
  fflush(stdout);
#if 0
  system("date");
#endif /* 0 */
  fflush(stdout);

  return 0;
}

void prepare_distance_matrix()
{
  unsigned int i, j;
  double dist;

  for (i = 0; i < N_CITIES; ++i) {
    for (j = 0; j < N_CITIES; ++j) {
      if (i == j) {
        dist = 0;
      } else {
        dist = city_distance(cities[i], cities[j]);
      }
      distance_matrix[i][j] = dist;
    }
  }
}

void print_distance_matrix()
{
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  unsigned int i, j;

  for (i = 0; i < N_CITIES; ++i) {
    printf("# ");
    for (j = 0; j < N_CITIES; ++j) {
      printf("%15.8f   ", distance_matrix[i][j]);
    }
    printf("\n");
  }
}

/* [only works for 12] search the entire space for solutions */
static double best_E = 1.0e100, second_E = 1.0e100, third_E = 1.0e100;
static int best_route[N_CITIES];
static int second_route[N_CITIES];
static int third_route[N_CITIES];
static void do_all_perms(int *route, int n);

void exhaustive_search()
{
  static int initial_route[N_CITIES] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
  printf("\n# ");
  fflush(stdout);
#if 0
  system("date");
#endif
  fflush(stdout);
  do_all_perms(initial_route, 1);
  printf("\n# ");
  fflush(stdout);
#if 0
  system("date");
#endif /* 0 */
  fflush(stdout);

  printf("# exhaustive best route: ");
  Ptsp(best_route);
  printf("\n# its energy is: %g\n", best_E);

  printf("# exhaustive second_best route: ");
  Ptsp(second_route);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  printf("\n# its energy is: %g\n", second_E);

  printf("# exhaustive third_best route: ");
  Ptsp(third_route);
  printf("\n# its energy is: %g\n", third_E);
}

/* James Theiler's recursive algorithm for generating all routes */
static void do_all_perms(int *route, int n)
{
  if (n == (N_CITIES-1)) {
    /* do it! calculate the energy/cost for that route */
    double E;
    E = Etsp(route);            /* TSP energy function */
    /* now save the best 3 energies and routes */
    if (E < best_E) {
      third_E = second_E;
      memcpy(third_route, second_route, N_CITIES*sizeof(*route));
      second_E = best_E;
      memcpy(second_route, best_route, N_CITIES*sizeof(*route));
      best_E = E;
      memcpy(best_route, route, N_CITIES*sizeof(*route));
    } else if (E < second_E) {
      third_E = second_E;
      memcpy(third_route, second_route, N_CITIES*sizeof(*route));
      second_E = E;
      memcpy(second_route, route, N_CITIES*sizeof(*route));
    } else if (E < third_E) {
      third_E = E;
      memcpy(route, third_route, N_CITIES*sizeof(*route));
    }
  } else {
    int new_route[N_CITIES];
    unsigned int j;
    int swap_tmp;
    memcpy(new_route, route, N_CITIES*sizeof(*route));
    for (j = n; j < N_CITIES; ++j) {
      swap_tmp = new_route[j];
      new_route[j] = new_route[n];
      new_route[n] = swap_tmp;
      do_all_perms(new_route, n+1);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    }
  }
}
```

다음 코드는 시각 그래프를 만들어 줍니다.

```
$./siman_tsp > tsp.output
$grep -v "^#" tsp.output
 | awk '{print :math:`1, `  NF}'
 | graph -y 3300 6500 -W0 -X generation -Y distance
    -L "TSP - 12 southwest cities"
 | plot -Tps > 12-cities.eps
$grep initial_city_coord tsp.output
  | awk '{print :math:`2, `  3}'
  | graph -X "longitude (- means west)" -Y "latitude"
     -L "TSP - initial-order" -f 0.03 -S 1 0.1
  | plot -Tps > initial-route.eps
$grep final_city_coord tsp.output
  | awk '{print :math:`2, `  3}'
  | graph -X "longitude (- means west)" -Y "latitude"
     -L "TSP - final-order" -f 0.03 -S 1 0.1
  | plot -Tps > final-route.eps
```

다음은 초기 단계에서 방문하는 도시 순서입니다. 경도 값이 음수인 이유는, 서쪽에 위치한 도시들이고, 실제 지도처럼 그래프로 보일 수 있기를 바랬기 때문입니다.

```
# initial coordinates of cities (longitude and latitude)
##initial_city_coord: -105.95 35.68 Santa Fe
##initial_city_coord: -112.07 33.54 Phoenix
##initial_city_coord: -106.62 35.12 Albuquerque
##initial_city_coord: -103.2 34.41 Clovis
##initial_city_coord: -107.87 37.29 Durango
##initial_city_coord: -96.77 32.79 Dallas
##initial_city_coord: -105.92 35.77 Tesuque
##initial_city_coord: -107.84 35.15 Grants
##initial_city_coord: -106.28 35.89 Los Alamos
##initial_city_coord: -106.76 32.34 Las Cruces
##initial_city_coord: -108.58 37.35 Cortez
##initial_city_coord: -108.74 35.52 Gallup
##initial_city_coord: -105.95 35.68 Santa Fe
```

최적화 된 경로는 다음과 같습니다.

```
# final coordinates of cities (longitude and latitude)
##final_city_coord: -105.95 35.68 Santa Fe
##final_city_coord: -103.2 34.41 Clovis
##final_city_coord: -96.77 32.79 Dallas
##final_city_coord: -106.76 32.34 Las Cruces
##final_city_coord: -112.07 33.54 Phoenix
##final_city_coord: -108.74 35.52 Gallup
##final_city_coord: -108.58 37.35 Cortez
##final_city_coord: -107.87 37.29 Durango
##final_city_coord: -107.84 35.15 Grants
##final_city_coord: -106.62 35.12 Albuquerque
##final_city_coord: -106.28 35.89 Los Alamos
##final_city_coord: -105.92 35.77 Tesuque
##final_city_coord: -105.95 35.68 Santa Fe
```

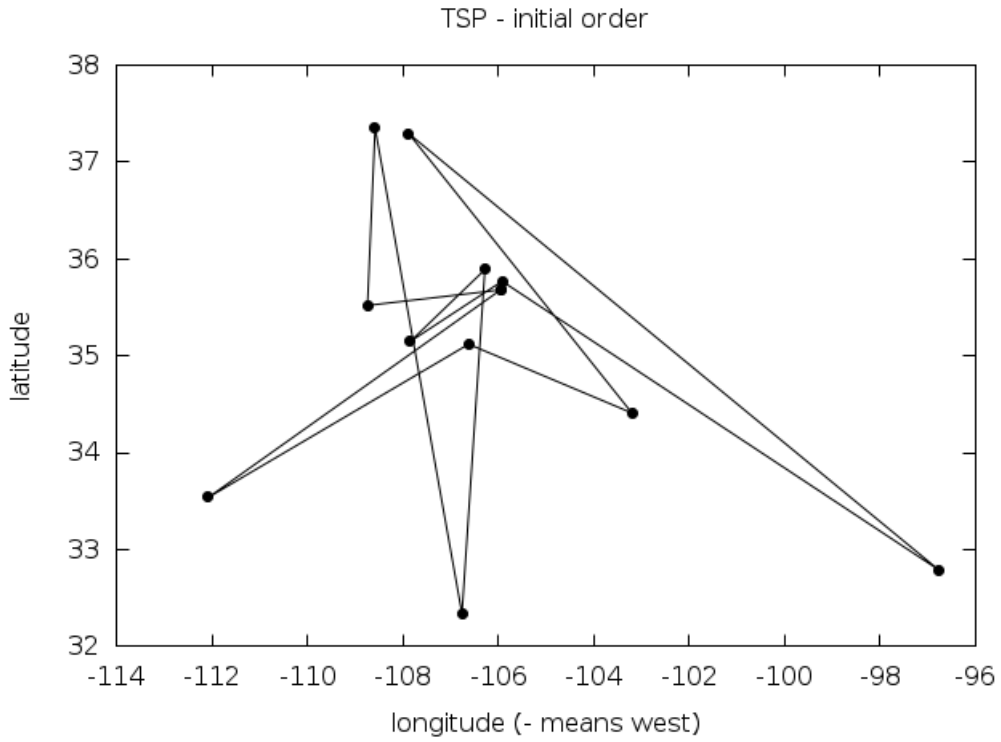


그림 27.3: 12개의 남서부 도시를 방문하는 비행기 외판원 문제의 초기 방문 경로.

다음은 비용 함수(에너지) vs 시행 횟수 그래프입니다. (각 지점은 새로운 온도가 설정되는 계산 지점을 의미합니다.)

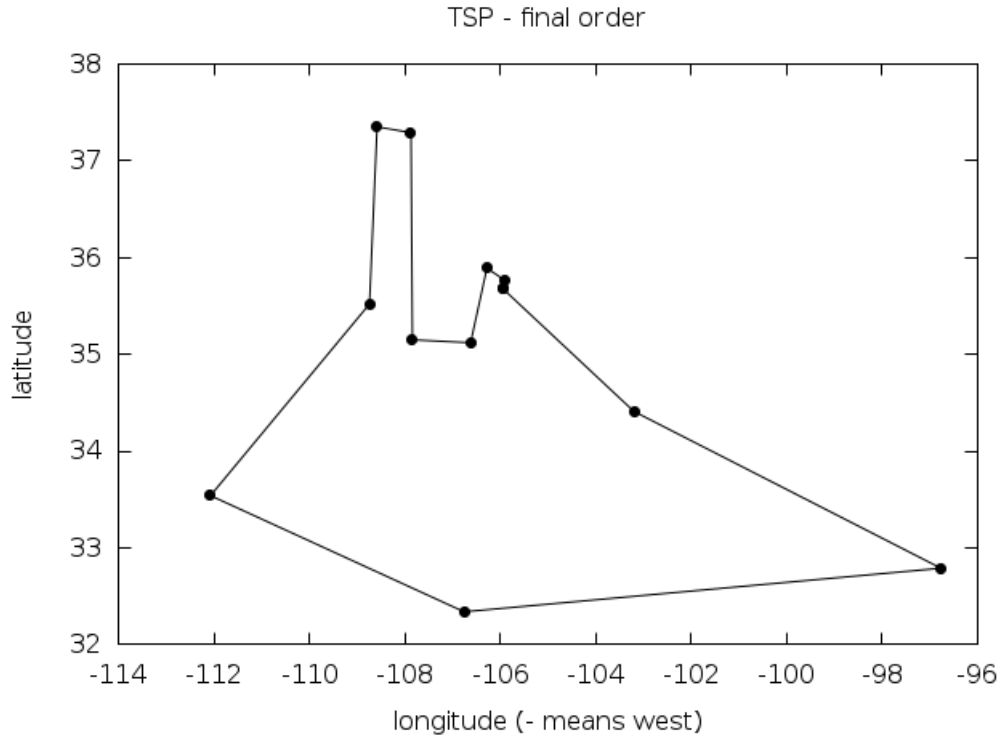


그림 27.4: 12개의 남서부 도시를 방문하는 비행기 외판원 문제의 최종(최적화 된) 방문 경로.

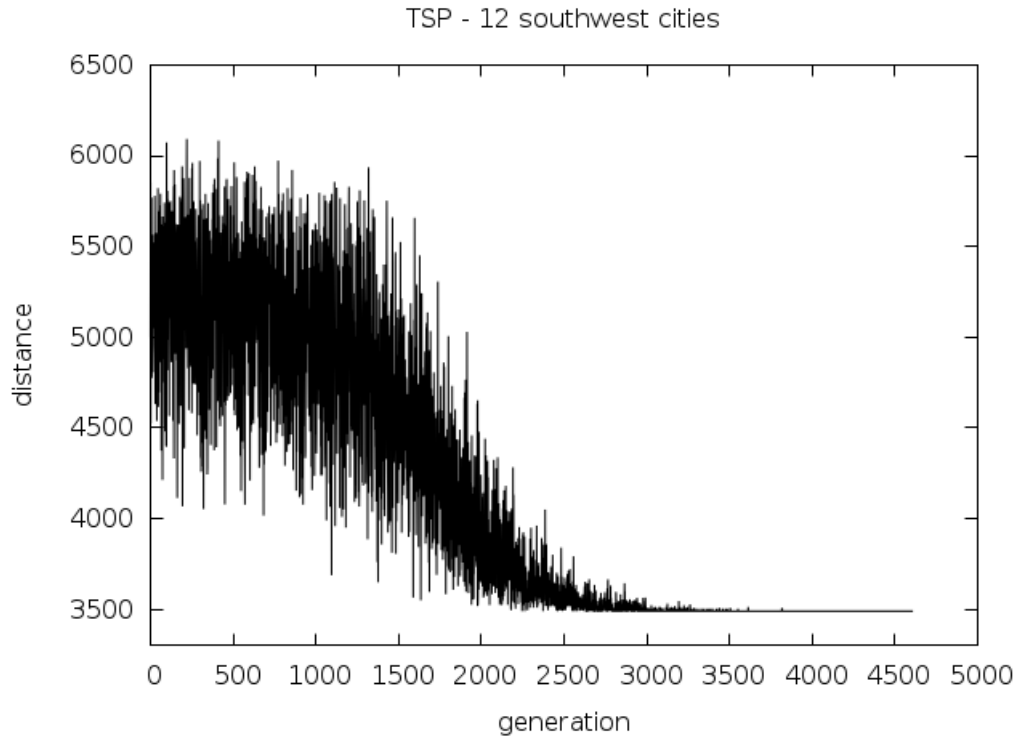


그림 27.5: 12개의 남서부 도시를 방문하는 비행기 외판원 문제의 담금질 방법 실행 결과.

27.4 참고 문헌과 추가 자료

더 자세한 사항은 다음의 책을 참고할 수 있습니다.

- Modern Heuristic Techniques for Combinatorial Problems, Colin R. Reeves (ed.), McGraw-Hill, 1995 (ISBN 0-07-709239-2).

제 28 장

상미분 방정식

참고: 번역중

This chapter describes functions for solving ordinary differential equation (ODE) initial value problems. The library provides a variety of low-level methods, such as Runge-Kutta and Bulirsch-Stoer routines, and higher-level components for adaptive step-size control. The components can be combined by the user to achieve the desired solution, with full access to any intermediate steps. A driver object can be used as a high level wrapper for easy use of low level functions.

These functions are declared in the header file `gsl_odeiv2.h`. This is a new interface in version 1.15 and uses the prefix `gsl_odeiv2` for all functions. It is recommended over the previous `gsl_odeiv` implementation defined in `gsl_odeiv.h` The old interface has been retained under the original name for backwards compatibility.

28.1 Defining the ODE System

The routines solve the general n-dimensional first-order system,

$$\frac{dy_i(t)}{dt} = f_i(t, y_1(t), \ldots y_n(t))$$

for $i = 1, \ldots, n$. The stepping functions rely on the vector of derivatives f_i and the Jacobian matrix,

$$J_{ij} = \partial f_i(t, y(t))/\partial y_j$$

A system of equations is defined using the gsl_odeiv2_system datatype.

type **gsl_odeiv2_system**

This data type defines a general ODE system with arbitrary parameters.

`int (* function) (double t, const double y[], double dydt[], void * params)`

This function should store the vector elements $f_i(t, y, params)$ in the array dydt, for arguments (t, y) and parameters params.

The function should return GSL_SUCCESS if the calculation was completed successfully. Any other return value indicates an error. A special return value GSL_EBADFUNC causes gsl_odeiv2 routines to immediately stop and return. If function is modified (for example contents of params), the user must call an appropriate reset function (gsl_odeiv2_driver_reset(), gsl_odeiv2_evolve_reset() or gsl_odeiv2_step_reset()) before continuing. Use return values distinct from standard GSL error codes to distinguish your function as the source of the error.

`int (* jacobian) (double t, const double y[], double * dfdy, double dfdt[], void * params)`

This function should store the vector of derivative elements

$$\partial f_i(t, y, params)/\partial t$$

in the array dfdt and the Jacobian matrix J_{ij} in the array dfdy, regarded as a row-ordered matrix J(i,j) = dfdy[i * dimension + j] where dimension is the dimension of the system.

Not all of the stepper algorithms of gsl_odeiv2 make use of the Jacobian matrix, so it may not be necessary to provide this function (the jacobian element of the struct can be replaced by a null pointer for those algorithms).

The function should return GSL_SUCCESS if the calculation was completed successfully. Any other return value indicates an error. A special return value GSL_EBADFUNC causes gsl_odeiv2 routines to immediately stop and return. If jacobian is modified (for example contents of params), the user must call an appropriate reset function (gsl_odeiv2_driver_reset(), gsl_odeiv2_evolve_reset() or gsl_odeiv2_step_reset()) before continuing. Use return values distinct from standard GSL error codes to distinguish your function as the source of the error.

`size_t dimension`

This is the dimension of the system of equations.

`void * params`

This is a pointer to the arbitrary parameters of the system.

28.2 Stepping Functions

The lowest level components are the stepping functions which advance a solution from time t to $t + h$ for a fixed step-size h and estimate the resulting local error.

type **gsl_odeiv2_step**

This contains internal parameters for a stepping function.

gsl_odeiv2_step ***gsl_odeiv2_step_alloc**(const gsl_odeiv2_step_type *T, size_t dim)

This function returns a pointer to a newly allocated instance of a stepping function of type `T` for a system of `dim` dimensions. Please note that if you use a stepper method that requires access to a driver object, it is advisable to use a driver allocation method, which automatically allocates a stepper, too.

int **gsl_odeiv2_step_reset**(gsl_odeiv2_step *s)

This function resets the stepping function `s`. It should be used whenever the next use of `s` will not be a continuation of a previous step.

void **gsl_odeiv2_step_free**(gsl_odeiv2_step *s)

This function frees all the memory associated with the stepping function `s`.

const char ***gsl_odeiv2_step_name**(const gsl_odeiv2_step *s)

This function returns a pointer to the name of the stepping function. For example:

```
printf ("step method is '%s'\n", gsl_odeiv2_step_name (s));
```

would print something like `step method is 'rkf45'`.

unsigned int **gsl_odeiv2_step_order**(const gsl_odeiv2_step *s)

This function returns the order of the stepping function on the previous step. The order can vary if the stepping function itself is adaptive.

int **gsl_odeiv2_step_set_driver**(gsl_odeiv2_step *s, const gsl_odeiv2_driver *d)

This function sets a pointer of the driver object `d` for stepper `s`, to allow the stepper to access control (and evolve) object through the driver object. This is a requirement for some steppers, to get the desired error level for internal iteration of stepper. Allocation of a driver object calls this function automatically.

int **gsl_odeiv2_step_apply**(gsl_odeiv2_step *s, double t, double h, double y[], double yerr[], const double dydt_in[], double dydt_out[], const gsl_odeiv2_system *sys)

This function applies the stepping function `s` to the system of equations defined by `sys`, using the step-size `h` to advance the system from time `t` and state `y` to time `t` + `h`. The new state of the system is stored in `y` on output, with an estimate of the absolute error in each component stored in `yerr`. If the argument `dydt_in` is not null it should point an array containing the derivatives for the system at time `t` on input. This is optional as the derivatives will be computed internally if they are not provided, but allows the reuse of existing derivative information. On output the new derivatives of the system at time `t` + `h` will be stored in `dydt_out` if it is not null.

The stepping function returns `GSL_FAILURE` if it is unable to compute the requested step. Also, if the user-supplied functions defined in the system `sys` return a status other than `GSL_SUCCESS` the step will be aborted. In that case, the elements of `y` will be restored to their pre-step values and the error code from the user-supplied function will be returned. Failure may be due to a singularity in the system or too large step-size `h`. In that case the step should be attempted again with a smaller step-size, e.g. `h` / 2.

If the driver object is not appropriately set via `gsl_odeiv2_step_set_driver()` for those steppers that need it, the stepping function returns `GSL_EFAULT`. If the user-supplied functions defined in the system `sys` returns `GSL_EBADFUNC`, the function returns immediately with the same return code. In this case the user must call `gsl_odeiv2_step_reset()` before calling this function again.

The following algorithms are available. Please note that algorithms which use step doubling for error estimation apply the more accurate values from two half steps instead of values from a single step for the new state `y`.

type **gsl_odeiv2_step_type**

gsl_odeiv2_step_type ***gsl_odeiv2_step_rk2**

Explicit embedded Runge-Kutta (2, 3) method.

gsl_odeiv2_step_type ***gsl_odeiv2_step_rk4**

Explicit 4th order (classical) Runge-Kutta. Error estimation is carried out by the step doubling method. For more efficient estimate of the error, use the embedded methods described below.

gsl_odeiv2_step_type ***gsl_odeiv2_step_rkf45**

Explicit embedded Runge-Kutta-Fehlberg (4, 5) method. This method is a good general-purpose integrator.

gsl_odeiv2_step_type *`gsl_odeiv2_step_rkck`

Explicit embedded Runge-Kutta Cash-Karp (4, 5) method.

gsl_odeiv2_step_type *`gsl_odeiv2_step_rk8pd`

Explicit embedded Runge-Kutta Prince-Dormand (8, 9) method.

gsl_odeiv2_step_type *`gsl_odeiv2_step_rk1imp`

Implicit Gaussian first order Runge-Kutta. Also known as implicit Euler or backward Euler method. Error estimation is carried out by the step doubling method. This algorithm requires the Jacobian and access to the driver object via `gsl_odeiv2_step_set_driver()`.

gsl_odeiv2_step_type *`gsl_odeiv2_step_rk2imp`

Implicit Gaussian second order Runge-Kutta. Also known as implicit mid-point rule. Error estimation is carried out by the step doubling method. This stepper requires the Jacobian and access to the driver object via `gsl_odeiv2_step_set_driver()`.

gsl_odeiv2_step_type *`gsl_odeiv2_step_rk4imp`

Implicit Gaussian 4th order Runge-Kutta. Error estimation is carried out by the step doubling method. This algorithm requires the Jacobian and access to the driver object via `gsl_odeiv2_step_set_driver()`.

gsl_odeiv2_step_type *`gsl_odeiv2_step_bsimp`

Implicit Bulirsch-Stoer method of Bader and Deuflhard. The method is generally suitable for stiff problems. This stepper requires the Jacobian.

gsl_odeiv2_step_type *`gsl_odeiv2_step_msadams`

A variable-coefficient linear multistep Adams method in Nordsieck form. This stepper uses explicit Adams-Bashforth (predictor) and implicit Adams-Moulton (corrector) methods in $P(EC)^m$ functional iteration mode. Method order varies dynamically between 1 and 12. This stepper requires the access to the driver object via `gsl_odeiv2_step_set_driver()`.

gsl_odeiv2_step_type *`gsl_odeiv2_step_msbdf`

A variable-coefficient linear multistep backward differentiation formula (BDF) method in Nordsieck form. This stepper uses the explicit BDF formula as predictor and implicit BDF formula as corrector. A modified Newton iteration method is used to solve the system of non-linear equations. Method order varies dynamically between 1 and 5. The method is generally suitable for stiff problems. This stepper requires the Jacobian and the access to the driver object via `gsl_odeiv2_step_set_driver()`.

28.3 Adaptive Step-size Control

The control function examines the proposed change to the solution produced by a stepping function and attempts to determine the optimal step-size for a user-specified level of error.

type **gsl_odeiv2_control**

This is a workspace for controlling step size.

type **gsl_odeiv2_control_type**

This specifies the control type.

gsl_odeiv2_control ***gsl_odeiv2_control_standard_new**(double eps_abs, double eps_rel, double a_y, double a_dydt)

The standard control object is a four parameter heuristic based on absolute and relative errors `eps_abs` and `eps_rel`, and scaling factors `a_y` and `a_dydt` for the system state $y(t)$ and derivatives $y'(t)$ respectively.

The step-size adjustment procedure for this method begins by computing the desired error level D_i for each component,

$$D_i = \epsilon_{abs} + \epsilon_{rel} * (a_y|y_i| + a_{dydt} h|y\prime_i|)$$

and comparing it with the observed error $E_i = |yerr_i|$. If the observed error `E` exceeds the desired error level `D` by more than 10% for any component then the method reduces the step-size by an appropriate factor,

$$h_{new} = h_{old} * S * (E/D)^{-1/q}$$

where q is the consistency order of the method (e.g. $q = 4$ for 4(5) embedded RK), and S is a safety factor of 0.9. The ratio E/D is taken to be the maximum of the ratios E_i/D_i.

If the observed error E is less than 50% of the desired error level `D` for the maximum ratio E_i/D_i then the algorithm takes the opportunity to increase the step-size to bring the error in line with the desired level,

$$h_{new} = h_{old} * S * (E/D)^{-1/(q+1)}$$

This encompasses all the standard error scaling methods. To avoid uncontrolled changes in the stepsize, the overall scaling factor is limited to the range $1/5$ to 5.

gsl_odeiv2_control ***gsl_odeiv2_control_y_new**(double eps_abs, double eps_rel)

This function creates a new control object which will keep the local error on each step

within an absolute error of `eps_abs` and relative error of `eps_rel` with respect to the solution $y_i(t)$. This is equivalent to the standard control object with `a_y` = 1 and `a_dydt` = 0.

gsl_odeiv2_control ***gsl_odeiv2_control_yp_new**(double eps_abs, double eps_rel)

This function creates a new control object which will keep the local error on each step within an absolute error of `eps_abs` and relative error of `eps_rel` with respect to the derivatives of the solution $y_i'(t)$. This is equivalent to the standard control object with `a_y` = 0 and `a_dydt` = 1.

gsl_odeiv2_control ***gsl_odeiv2_control_scaled_new**(double eps_abs, double eps_rel, double a_y, double a_dydt, const double scale_abs[], size_t dim)

This function creates a new control object which uses the same algorithm as `gsl_odeiv2_control_standard_new()` but with an absolute error which is scaled for each component by the array `scale_abs`. The formula for D_i for this control object is,

$$D_i = \epsilon_{abs} s_i + \epsilon_{rel} * (a_y |y_i| + a_{dydt} h |y\prime_i|)$$

where s_i is the i-th component of the array `scale_abs`. The same error control heuristic is used by the Matlab ODE suite.

gsl_odeiv2_control ***gsl_odeiv2_control_alloc**(const gsl_odeiv2_control_type *T)

This function returns a pointer to a newly allocated instance of a control function of type `T`. This function is only needed for defining new types of control functions. For most purposes the standard control functions described above should be sufficient.

int **gsl_odeiv2_control_init**(gsl_odeiv2_control *c, double eps_abs, double eps_rel, double a_y, double a_dydt)

This function initializes the control function `c` with the parameters `eps_abs` (absolute error), `eps_rel` (relative error), `a_y` (scaling factor for y) and `a_dydt` (scaling factor for derivatives).

void **gsl_odeiv2_control_free**(gsl_odeiv2_control *c)

This function frees all the memory associated with the control function `c`.

int **gsl_odeiv2_control_hadjust**(gsl_odeiv2_control *c, gsl_odeiv2_step *s, const double y[], const double yerr[], const double dydt[], double *h)

This function adjusts the step-size `h` using the control function `c`, and the current values of `y`, `yerr` and `dydt`. The stepping function `step` is also needed to determine the order of the method. If the error in the y-values `yerr` is found to be too large then the step-size `h` is reduced and the function returns `GSL_ODEIV_HADJ_DEC`. If the error is sufficiently

small then h may be increased and GSL_ODEIV_HADJ_INC is returned. The function returns GSL_ODEIV_HADJ_NIL if the step-size is unchanged. The goal of the function is to estimate the largest step-size which satisfies the user-specified accuracy requirements for the current point.

const char *gsl_odeiv2_control_name(const gsl_odeiv2_control *c)

This function returns a pointer to the name of the control function. For example:

```
printf ("control method is '%s'\n", gsl_odeiv2_control_name (c));
```

would print something like control method is 'standard'

int gsl_odeiv2_control_errlevel(gsl_odeiv2_control *c, const double y, const double dydt, const double h, const size_t ind, double *errlev)

This function calculates the desired error level of the ind-th component to errlev. It requires the value (y) and value of the derivative (dydt) of the component, and the current step size h.

int gsl_odeiv2_control_set_driver(gsl_odeiv2_control *c, const gsl_odeiv2_driver *d)

This function sets a pointer of the driver object d for control object c.

28.4 Evolution

The evolution function combines the results of a stepping function and control function to reliably advance the solution forward one step using an acceptable step-size.

type gsl_odeiv2_evolve

This workspace contains parameters for controlling the evolution function

gsl_odeiv2_evolve *gsl_odeiv2_evolve_alloc(size_t dim)

This function returns a pointer to a newly allocated instance of an evolution function for a system of dim dimensions.

int gsl_odeiv2_evolve_apply(gsl_odeiv2_evolve *e, gsl_odeiv2_control *con, gsl_odeiv2_step *step, const gsl_odeiv2_system *sys, double *t, double t1, double *h, double y[])

This function advances the system (e, sys) from time t and position y using the stepping function step. The new time and position are stored in t and y on output.

The initial step-size is taken as h. The control function con is applied to check whether the local error estimated by the stepping function step using step-size h exceeds the required error tolerance. If the error is too high, the step is retried by calling step with a decreased

step-size. This process is continued until an acceptable step-size is found. An estimate of the local error for the step can be obtained from the components of the array `e->yerr[]`.

If the user-supplied functions defined in the system `sys` returns `GSL_EBADFUNC`, the function returns immediately with the same return code. In this case the user must call `gsl_odeiv2_step_reset()` and `gsl_odeiv2_evolve_reset()` before calling this function again.

Otherwise, if the user-supplied functions defined in the system `sys` or the stepping function `step` return a status other than `GSL_SUCCESS`, the step is retried with a decreased step-size. If the step-size decreases below machine precision, a status of `GSL_FAILURE` is returned if the user functions returned `GSL_SUCCESS`. Otherwise the value returned by user function is returned. If no acceptable step can be made, `t` and `y` will be restored to their pre-step values and `h` contains the final attempted step-size.

If the step is successful the function returns a suggested step-size for the next step in `h`. The maximum time `t1` is guaranteed not to be exceeded by the time-step. On the final time-step the value of `t` will be set to `t1` exactly.

int **gsl_odeiv2_evolve_apply_fixed_step**(gsl_odeiv2_evolve *e, gsl_odeiv2_control *con, gsl_odeiv2_step *step, const gsl_odeiv2_system *sys, double *t, const double h, double y[])

This function advances the ODE-system (`e`, `sys`, `con`) from time `t` and position `y` using the stepping function `step` by a specified step size `h`. If the local error estimated by the stepping function exceeds the desired error level, the step is not taken and the function returns `GSL_FAILURE`. Otherwise the value returned by user function is returned.

int **gsl_odeiv2_evolve_reset**(gsl_odeiv2_evolve *e)

This function resets the evolution function `e`. It should be used whenever the next use of `e` will not be a continuation of a previous step.

void **gsl_odeiv2_evolve_free**(gsl_odeiv2_evolve *e)

This function frees all the memory associated with the evolution function `e`.

int **gsl_odeiv2_evolve_set_driver**(gsl_odeiv2_evolve *e, const gsl_odeiv2_driver *d)

This function sets a pointer of the driver object `d` for evolve object `e`.

If a system has discontinuous changes in the derivatives at known points, it is advisable to evolve the system between each discontinuity in sequence. For example, if a step-change in an external driving force occurs at times t_a, t_b and t_c then evolution should be carried out over the ranges (t_0, t_a), (t_a, t_b), (t_b, t_c), and (t_c, t_1) separately and not directly over the range (t_0, t_1).

28.5 Driver

The driver object is a high level wrapper that combines the evolution, control and stepper objects for easy use.

gsl_odeiv2_driver ***gsl_odeiv2_driver_alloc_y_new**(const gsl_odeiv2_system *sys, const gsl_odeiv2_step_type *T, const double hstart, const double epsabs, const double epsrel)

gsl_odeiv2_driver ***gsl_odeiv2_driver_alloc_yp_new**(const gsl_odeiv2_system *sys, const gsl_odeiv2_step_type *T, const double hstart, const double epsabs, const double epsrel)

gsl_odeiv2_driver ***gsl_odeiv2_driver_alloc_standard_new**(const gsl_odeiv2_system *sys, const gsl_odeiv2_step_type *T, const double hstart, const double epsabs, const double epsrel, const double a_y, const double a_dydt)

gsl_odeiv2_driver ***gsl_odeiv2_driver_alloc_scaled_new**(const gsl_odeiv2_system *sys, const gsl_odeiv2_step_type *T, const double hstart, const double epsabs, const double epsrel, const double a_y, const double a_dydt, const double scale_abs[])

These functions return a pointer to a newly allocated instance of a driver object. The functions automatically allocate and initialise the evolve, control and stepper objects for ODE system `sys` using stepper type `T`. The initial step size is given in `hstart`. The rest of the arguments follow the syntax and semantics of the control functions with same name (`gsl_odeiv2_control_*_new`).

int **gsl_odeiv2_driver_set_hmin**(gsl_odeiv2_driver *d, const double hmin)

The function sets a minimum for allowed step size `hmin` for driver `d`. Default value is 0.

int **gsl_odeiv2_driver_set_hmax**(gsl_odeiv2_driver *d, const double hmax)

The function sets a maximum for allowed step size `hmax` for driver `d`. Default value is `GSL_DBL_MAX`.

int **gsl_odeiv2_driver_set_nmax**(gsl_odeiv2_driver *d, const unsigned long int nmax)

The function sets a maximum for allowed number of steps `nmax` for driver `d`. Default value of 0 sets no limit for steps.

int **gsl_odeiv2_driver_apply**(gsl_odeiv2_driver *d, double *t, const double t1, double y[])

This function evolves the driver system `d` from `t` to `t1`. Initially vector `y` should contain the values of dependent variables at point `t`. If the function is unable to complete the calculation, an error code from `gsl_odeiv2_evolve_apply()` is returned, and `t` and `y` contain the values from last successful step.

If maximum number of steps is reached, a value of `GSL_EMAXITER` is returned. If the step size drops below minimum value, the function returns with `GSL_ENOPROG`. If the user-supplied functions defined in the system `sys` returns `GSL_EBADFUNC`, the function returns immediately with the same return code. In this case the user must call `gsl_odeiv2_driver_reset()` before calling this function again.

int **gsl_odeiv2_driver_apply_fixed_step**(gsl_odeiv2_driver *d, double *t, const double h, const unsigned long int n, double y[])

This function evolves the driver system `d` from `t` with `n` steps of size `h`. If the function is unable to complete the calculation, an error code from `gsl_odeiv2_evolve_apply_fixed_step()` is returned, and `t` and `y` contain the values from last successful step.

int **gsl_odeiv2_driver_reset**(gsl_odeiv2_driver *d)

This function resets the evolution and stepper objects.

int **gsl_odeiv2_driver_reset_hstart**(gsl_odeiv2_driver *d, const double hstart)

The routine resets the evolution and stepper objects and sets new initial step size to `hstart`. This function can be used e.g. to change the direction of integration.

int **gsl_odeiv2_driver_free**(gsl_odeiv2_driver *d)

This function frees the driver object, and the related evolution, stepper and control objects.

28.6 Examples

The following program solves the second-order nonlinear Van der Pol oscillator equation,

$$u''(t) + \mu u'(t)(u(t)^2 - 1) + u(t) = 0$$

This can be converted into a first order system suitable for use with the routines described in this chapter by introducing a separate variable for the velocity, $v = u'(t)$,

$$u' = v$$
$$v' = -u + \mu v(1 - u^2)$$

The program begins by defining functions for these derivatives and their Jacobian. The main function uses driver level functions to solve the problem. The program evolves the solution from $(u, v) = (1, 0)$ at $t = 0$ to $t = 100$. The step-size h is automatically adjusted by the controller to maintain an absolute accuracy of 10^{-6} in the function values (u, v). The loop in the example prints the solution at the points $t_i = 1, 2, \ldots, 100$.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv2.h>

int
func (double t, const double y[], double f[],
      void *params)
{
  (void)(t); /* avoid unused parameter warning */
  double mu = *(double *)params;
  f[0] = y[1];
  f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
  return GSL_SUCCESS;
}

int
jac (double t, const double y[], double *dfdy,
     double dfdt[], void *params)
{
  (void)(t); /* avoid unused parameter warning */
  double mu = *(double *)params;
  gsl_matrix_view dfdy_mat
    = gsl_matrix_view_array (dfdy, 2, 2);
  gsl_matrix * m = &dfdy_mat.matrix;
  gsl_matrix_set (m, 0, 0, 0.0);
  gsl_matrix_set (m, 0, 1, 1.0);
  gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
  gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
  dfdt[0] = 0.0;
  dfdt[1] = 0.0;
  return GSL_SUCCESS;
}

int
main (void)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
{
  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};

  gsl_odeiv2_driver * d =
    gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd,
                                   1e-6, 1e-6, 0.0);
  int i;
  double t = 0.0, t1 = 100.0;
  double y[2] = { 1.0, 0.0 };

  for (i = 1; i <= 100; i++)
    {
      double ti = i * t1 / 100.0;
      int status = gsl_odeiv2_driver_apply (d, &t, ti, y);

      if (status != GSL_SUCCESS)
        {
          printf ("error, return value=%d\n", status);
          break;
        }

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_driver_free (d);
  return 0;
}
```

The user can work with the lower level functions directly, as in the following example. In this case an intermediate result is printed after each successful step instead of equidistant time points.

```
int
main (void)
{
  const gsl_odeiv2_step_type * T
    = gsl_odeiv2_step_rk8pd;

  gsl_odeiv2_step * s
    = gsl_odeiv2_step_alloc (T, 2);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_odeiv2_control * c
    = gsl_odeiv2_control_y_new (1e-6, 0.0);
  gsl_odeiv2_evolve * e
    = gsl_odeiv2_evolve_alloc (2);

  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};

  double t = 0.0, t1 = 100.0;
  double h = 1e-6;
  double y[2] = { 1.0, 0.0 };

  while (t < t1)
    {
      int status = gsl_odeiv2_evolve_apply (e, c, s,
                                            &sys,
                                            &t, t1,
                                            &h, y);

      if (status != GSL_SUCCESS)
          break;

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_evolve_free (e);
  gsl_odeiv2_control_free (c);
  gsl_odeiv2_step_free (s);
  return 0;
}
```

For functions with multiple parameters, the appropriate information can be passed in through the `params` argument in `gsl_odeiv2_system` definition (`mu` in this example) by using a pointer to a struct.

It is also possible to work with a non-adaptive integrator, using only the stepping function itself, `gsl_odeiv2_driver_apply_fixed_step()` or `gsl_odeiv2_evolve_apply_fixed_step()`. The following program uses the driver level function, with fourth-order Runge-Kutta stepping function with a fixed stepsize of 0.001.

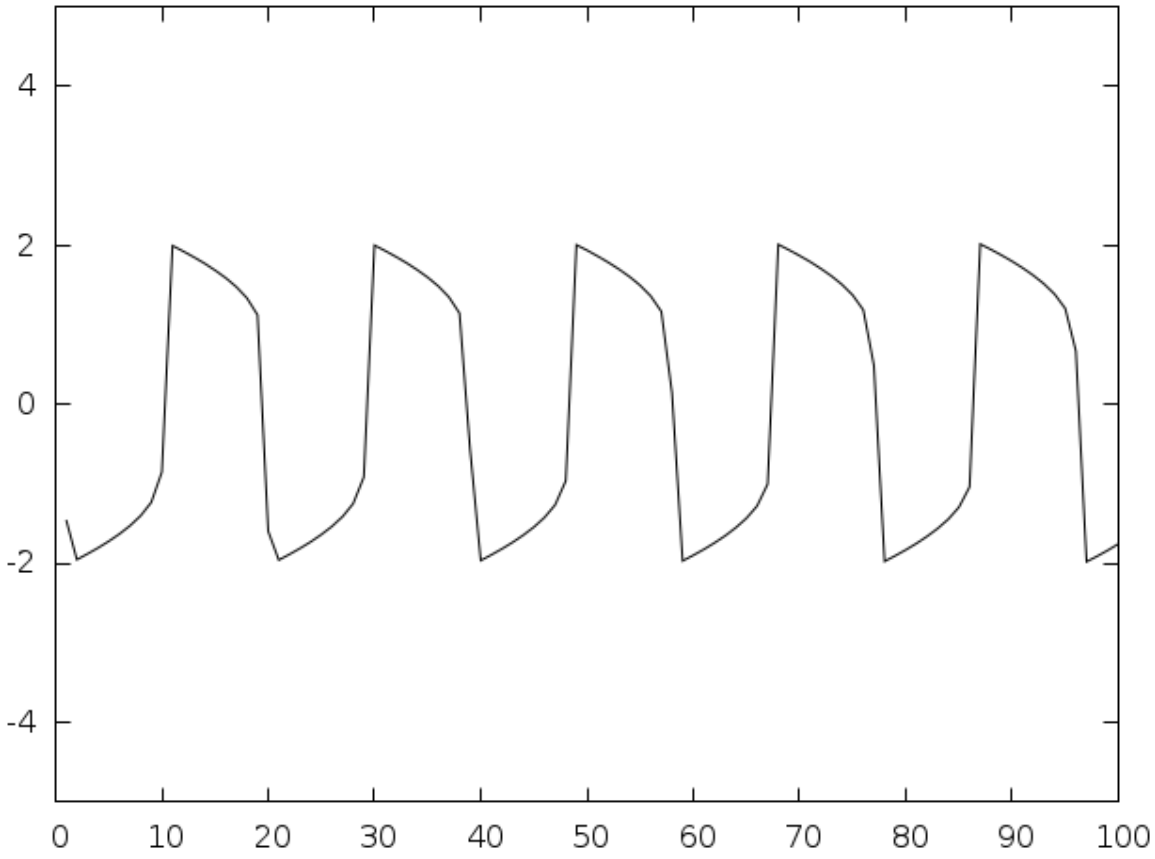


그림 28.1: Numerical solution of the Van der Pol oscillator equation using Prince-Dormand 8th order Runge-Kutta.

```
int
main (void)
{
  double mu = 10;
  gsl_odeiv2_system sys = { func, jac, 2, &mu };

  gsl_odeiv2_driver *d =
    gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk4,
                                   1e-3, 1e-8, 1e-8);

  double t = 0.0;
  double y[2] = { 1.0, 0.0 };
  int i, s;

  for (i = 0; i < 100; i++)
    {
      s = gsl_odeiv2_driver_apply_fixed_step (d, &t, 1e-3, 1000, y);

      if (s != GSL_SUCCESS)
        {
          printf ("error: driver returned %d\n", s);
          break;
        }

      printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }

  gsl_odeiv2_driver_free (d);
  return s;
}
```

28.7 References and Further Reading

- Ascher, U.M., Petzold, L.R., Computer Methods for Ordinary Differential and Differential-Algebraic Equations, SIAM, Philadelphia, 1998.
- Hairer, E., Norsett, S. P., Wanner, G., Solving Ordinary Differential Equations I: Nonstiff Problems, Springer, Berlin, 1993.
- Hairer, E., Wanner, G., Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, Springer, Berlin, 1996.

Many of the basic Runge-Kutta formulas can be found in the Handbook of Mathematical Functions,

- Abramowitz & Stegun (eds.), Handbook of Mathematical Functions, Section 25.5.

The implicit Bulirsch-Stoer algorithm `bsimp` is described in the following paper,

- G. Bader and P. Deuflhard, "A Semi-Implicit Mid-Point Rule for Stiff Systems of Ordinary Differential Equations.", Numer.: Math.: 41, 373–398, 1983.

The Adams and BDF multistep methods `msadams` and `msbdf` are based on the following articles,

- G. D. Byrne and A. C. Hindmarsh, "A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations.", ACM Trans. Math. Software, 1, 71–96, 1975.
- P. N. Brown, G. D. Byrne and A. C. Hindmarsh, "VODE: A Variable-coefficient ODE Solver.", SIAM J. Sci. Stat. Comput. 10, 1038–1051, 1989.
- A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker and C. S. Woodward, "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers.", ACM Trans. Math. Software 31, 363–396, 2005.

제 29 장

보간법

참고: 번역중

This chapter describes functions for performing interpolation. The library provides a variety of interpolation methods, including Cubic, Akima, and Steffen splines. The interpolation types are interchangeable, allowing different methods to be used without recompiling. Interpolations can be defined for both normal and periodic boundary conditions. Additional functions are available for computing derivatives and integrals of interpolating functions. Routines are provided for interpolating both one and two dimensional datasets.

These interpolation methods produce curves that pass through each datapoint. To interpolate noisy data with a smoothing curve see B-스플라인.

The functions described in this section are declared in the header files `gsl_interp.h` and `gsl_spline.h`.

29.1 Introduction to 1D Interpolation

Given a set of data points $(x_1, y_1) \dots (x_n, y_n)$ the routines described in this section compute a continuous interpolating function $y(x)$ such that $y(x_i) = y_i$. The interpolation is piecewise smooth, and its behavior at the end-points is determined by the type of interpolation used.

29.2 1D Interpolation Functions

The interpolation function for a given dataset is stored in a `gsl_interp` object. These are created by the following functions.

type **gsl_interp**

Workspace for 1D interpolation

gsl_interp ***gsl_interp_alloc**(const gsl_interp_type *T, size_t size)

This function returns a pointer to a newly allocated interpolation object of type `T` for `size` data-points.

int **gsl_interp_init**(gsl_interp *interp, const double xa[], const double ya[], size_t size)

This function initializes the interpolation object `interp` for the data (`xa`, `ya`) where `xa` and `ya` are arrays of size `size`. The interpolation object (`gsl_interp`) does not save the data arrays `xa` and `ya` and only stores the static state computed from the data. The `xa` data array is always assumed to be strictly ordered, with increasing x values; the behavior for other arrangements is not defined.

void **gsl_interp_free**(gsl_interp *interp)

This function frees the interpolation object `interp`.

29.3 1D Interpolation Types

The interpolation library provides the following interpolation types:

type **gsl_interp_type**

gsl_interp_type ***gsl_interp_linear**

Linear interpolation. This interpolation method does not require any additional memory.

gsl_interp_type ***gsl_interp_polynomial**

Polynomial interpolation. This method should only be used for interpolating small numbers of points because polynomial interpolation introduces large oscillations, even for well-behaved datasets. The number of terms in the interpolating polynomial is equal to the number of points.

gsl_interp_type ***gsl_interp_cspline**

Cubic spline with natural boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied

data-points. The second derivative is chosen to be zero at the first point and last point.

gsl_interp_type ***gsl_interp_cspline_periodic**

Cubic spline with periodic boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The derivatives at the first and last points are also matched. Note that the last point in the data must have the same y-value as the first point, otherwise the resulting periodic interpolation will have a discontinuity at the boundary.

gsl_interp_type ***gsl_interp_akima**

Non-rounded Akima spline with natural boundary conditions. This method uses the non-rounded corner algorithm of Wodicka.

gsl_interp_type ***gsl_interp_akima_periodic**

Non-rounded Akima spline with periodic boundary conditions. This method uses the non-rounded corner algorithm of Wodicka.

gsl_interp_type ***gsl_interp_steffen**

Steffen's method guarantees the monotonicity of the interpolating function between the given data points. Therefore, minima and maxima can only occur exactly at the data points, and there can never be spurious oscillations between data points. The interpolated function is piecewise cubic in each interval. The resulting curve and its first derivative are guaranteed to be continuous, but the second derivative may be discontinuous.

The following related functions are available:

const char ***gsl_interp_name**(const gsl_interp *interp)

This function returns the name of the interpolation type used by `interp`. For example:

```
printf ("interp uses '%s' interpolation.\n", gsl_interp_name (interp));
```

would print something like:

```
interp uses 'cspline' interpolation.
```

unsigned int **gsl_interp_min_size**(const gsl_interp *interp)

unsigned int **gsl_interp_type_min_size**(const gsl_interp_type *T)

These functions return the minimum number of points required by the interpolation object `interp` or interpolation type `T`. For example, Akima spline interpolation requires a minimum of 5 points.

29.4 1D Index Look-up and Acceleration

The state of searches can be stored in a `gsl_interp_accel` object, which is a kind of iterator for interpolation lookups.

type **gsl_interp_accel**

This workspace stores state variables for interpolation lookups. It caches the previous value of an index lookup. When the subsequent interpolation point falls in the same interval its index value can be returned immediately.

size_t **gsl_interp_bsearch**(const double x_array[], double x, size_t index_lo, size_t index_hi)

This function returns the index i of the array `x_array` such that `x_array[i] <= x < x_array[i+1]`. The index is searched for in the range [`index_lo`, `index_hi`]. `HAVE_INLINE` 이 정의된 경우, 인라인 함수가 사용됩니다.

gsl_interp_accel ***gsl_interp_accel_alloc**(void)

This function returns a pointer to an accelerator object, which is a kind of iterator for interpolation lookups. It tracks the state of lookups, thus allowing for application of various acceleration strategies. When multiple interpolants are in use, the same accelerator object may be used for all datasets with the same domain (`x_array`), but different accelerators should be used for data defined on different domains.

size_t **gsl_interp_accel_find**(gsl_interp_accel *a, const double x_array[], size_t size, double x)

This function performs a lookup action on the data array `x_array` of size `size`, using the given accelerator `a`. This is how lookups are performed during evaluation of an interpolation. The function returns an index i such that `x_array[i] <= x < x_array[i+1]`. `HAVE_INLINE` 이 정의된 경우, 인라인 함수가 사용됩니다.

int **gsl_interp_accel_reset**(gsl_interp_accel *acc);

This function reinitializes the accelerator object `acc`. It should be used when the cached information is no longer applicable—for example, when switching to a new dataset.

void **gsl_interp_accel_free**(gsl_interp_accel *acc)

This function frees the accelerator object `acc`.

29.5 1D Evaluation of Interpolating Functions

double **gsl_interp_eval**(const gsl_interp *interp, const double xa[], const double ya[], double x, gsl_interp_accel *acc)

int **gsl_interp_eval_e**(const gsl_interp *interp, const double xa[], const double ya[], double x, gsl_interp_accel *acc, double *y)

These functions return the interpolated value of `y` for a given point `x`, using the interpolation object `interp`, data arrays `xa` and `ya` and the accelerator `acc`. When `x` is outside the range of `xa`, the error code `GSL_EDOM` is returned with a value of `GSL_NAN` for `y`.

double **gsl_interp_eval_deriv**(const gsl_interp *interp, const double xa[], const double ya[], double x, gsl_interp_accel *acc)

int **gsl_interp_eval_deriv_e**(const gsl_interp *interp, const double xa[], const double ya[], double x, gsl_interp_accel *acc, double *d)

These functions return the derivative `d` of an interpolated function for a given point `x`, using the interpolation object `interp`, data arrays `xa` and `ya` and the accelerator `acc`.

double **gsl_interp_eval_deriv2**(const gsl_interp *interp, const double xa[], const double ya[], double x, gsl_interp_accel *acc)

int **gsl_interp_eval_deriv2_e**(const gsl_interp *interp, const double xa[], const double ya[], double x, gsl_interp_accel *acc, double *d2)

These functions return the second derivative `d2` of an interpolated function for a given point `x`, using the interpolation object `interp`, data arrays `xa` and `ya` and the accelerator `acc`.

double **gsl_interp_eval_integ**(const gsl_interp *interp, const double xa[], const double ya[], double a, double b, gsl_interp_accel *acc)

int **gsl_interp_eval_integ_e**(const gsl_interp *interp, const double xa[], const double ya[], double a, double b, gsl_interp_accel *acc, double *result)

These functions return the numerical integral `result` of an interpolated function over the range [`a`, `b`], using the interpolation object `interp`, data arrays `xa` and `ya` and the accelerator `acc`.

29.6 1D Higher-level Interface

The functions described in the previous sections required the user to supply pointers to the x and y arrays on each call. The following functions are equivalent to the corresponding `gsl_interp` functions but maintain a copy of this data in the `gsl_spline` object. This removes the need to pass both `xa` and `ya` as arguments on each evaluation. These functions are defined in the header file `gsl_spline.h`.

type **gsl_spline**

This workspace provides a higher level interface for the `gsl_interp` object

gsl_spline ***gsl_spline_alloc**(const gsl_interp_type *T, size_t size)

int **gsl_spline_init**(gsl_spline *spline, const double xa[], const double ya[], size_t size)

void **gsl_spline_free**(gsl_spline *spline)

const char ***gsl_spline_name**(const gsl_spline *spline)

unsigned int **gsl_spline_min_size**(const gsl_spline *spline)

double **gsl_spline_eval**(const gsl_spline *spline, double x, gsl_interp_accel *acc)

int **gsl_spline_eval_e**(const gsl_spline *spline, double x, gsl_interp_accel *acc, double *y)

double **gsl_spline_eval_deriv**(const gsl_spline *spline, double x, gsl_interp_accel *acc)

int **gsl_spline_eval_deriv_e**(const gsl_spline *spline, double x, gsl_interp_accel *acc, double *d)

double **gsl_spline_eval_deriv2**(const gsl_spline *spline, double x, gsl_interp_accel *acc)

int **gsl_spline_eval_deriv2_e**(const gsl_spline *spline, double x, gsl_interp_accel *acc, double *d2)

double **gsl_spline_eval_integ**(const gsl_spline *spline, double a, double b, gsl_interp_accel *acc)

int **gsl_spline_eval_integ_e**(const gsl_spline *spline, double a, double b, gsl_interp_accel *acc, double *result)

29.7 1D Interpolation Example Programs

The following program demonstrates the use of the interpolation and spline functions. It computes a cubic spline interpolation of the 10-point dataset (x_i, y_i) where $x_i = i + \sin(i)/2$ and $y_i = i + \cos(i^2)$ for $i = 0 \ldots 9$.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

int
main (void)
{
  int i;
  double xi, yi, x[10], y[10];

  printf ("#m=0,S=17\n");

  for (i = 0; i < 10; i++)
    {
      x[i] = i + 0.5 * sin (i);
      y[i] = i + cos (i * i);
      printf ("%g %g\n", x[i], y[i]);
    }

  printf ("#m=1,S=0\n");

  {
    gsl_interp_accel *acc
      = gsl_interp_accel_alloc ();
    gsl_spline *spline
      = gsl_spline_alloc (gsl_interp_cspline, 10);

    gsl_spline_init (spline, x, y, 10);

    for (xi = x[0]; xi < x[9]; xi += 0.01)
      {
        yi = gsl_spline_eval (spline, xi, acc);
        printf ("%g %g\n", xi, yi);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      }
    gsl_spline_free (spline);
    gsl_interp_accel_free (acc);
  }
  return 0;
}
```

The output is designed to be used with the GNU plotutils `graph` program:

```
$ ./a.out > interp.dat
$ graph -T ps < interp.dat > interp.ps
```

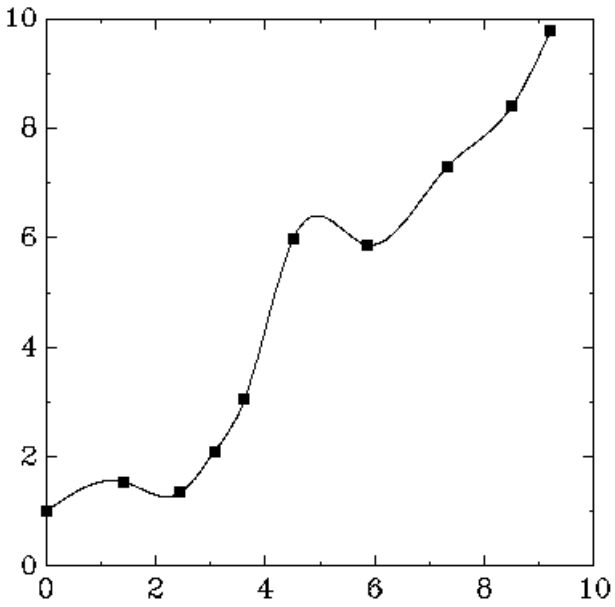


그림 29.1: Cubic spline interpolation

그림 29.1 shows a smooth interpolation of the original points. The interpolation method can be changed simply by varying the first argument of `gsl_spline_alloc()`.

The next program demonstrates a periodic cubic spline with 4 data points. Note that the first and last points must be supplied with the same y-value for a periodic spline.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

int
main (void)
{
  int N = 4;
  double x[4] = {0.00, 0.10,  0.27,  0.30};
  double y[4] = {0.15, 0.70, -0.10,  0.15};
              /* Note: y[0] == y[3] for periodic data */

  gsl_interp_accel *acc = gsl_interp_accel_alloc ();
  const gsl_interp_type *t = gsl_interp_cspline_periodic;
  gsl_spline *spline = gsl_spline_alloc (t, N);

  int i; double xi, yi;

  printf ("#m=0,S=5\n");
  for (i = 0; i < N; i++)
    {
      printf ("%g %g\n", x[i], y[i]);
    }

  printf ("#m=1,S=0\n");
  gsl_spline_init (spline, x, y, N);

  for (i = 0; i <= 100; i++)
    {
      xi = (1 - i / 100.0) * x[0] + (i / 100.0) * x[N-1];
      yi = gsl_spline_eval (spline, xi, acc);
      printf ("%g %g\n", xi, yi);
    }

  gsl_spline_free (spline);
  gsl_interp_accel_free (acc);
  return 0;
}
```

The output can be plotted with GNU graph:

```
$ ./a.out > interp.dat
$ graph -T ps < interp.dat > interp.ps
```

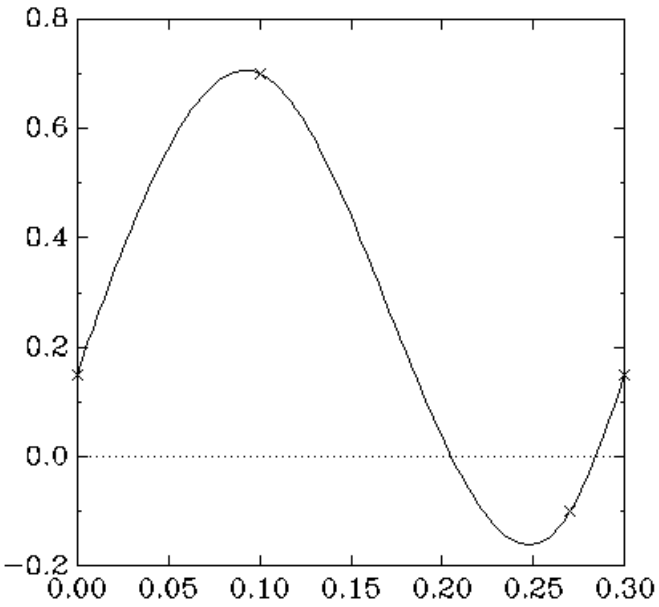


그림 29.2: Periodic cubic spline interpolation

그림 29.2 shows a periodic interpolation of the original points. The slope of the fitted curve is the same at the beginning and end of the data, and the second derivative is also.

The next program illustrates the difference between the cubic spline, Akima, and Steffen interpolation types on a difficult dataset.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_spline.h>

int
main(void)
{
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
size_t i;
const size_t N = 9;

/* this dataset is taken from
 * J. M. Hyman, Accurate Monotonicity preserving cubic interpolation,
 * SIAM J. Sci. Stat. Comput. 4, 4, 1983. */
const double x[] = { 7.99, 8.09, 8.19, 8.7, 9.2,
                     10.0, 12.0, 15.0, 20.0 };
const double y[] = { 0.0, 2.76429e-5, 4.37498e-2,
                     0.169183, 0.469428, 0.943740,
                     0.998636, 0.999919, 0.999994 };

gsl_interp_accel *acc = gsl_interp_accel_alloc();
gsl_spline *spline_cubic = gsl_spline_alloc(gsl_interp_cspline, N);
gsl_spline *spline_akima = gsl_spline_alloc(gsl_interp_akima, N);
gsl_spline *spline_steffen = gsl_spline_alloc(gsl_interp_steffen, N);

gsl_spline_init(spline_cubic, x, y, N);
gsl_spline_init(spline_akima, x, y, N);
gsl_spline_init(spline_steffen, x, y, N);

for (i = 0; i < N; ++i)
  printf("%g %g\n", x[i], y[i]);

printf("\n\n");

for (i = 0; i <= 100; ++i)
  {
    double xi = (1 - i / 100.0) * x[0] + (i / 100.0) * x[N-1];
    double yi_cubic = gsl_spline_eval(spline_cubic, xi, acc);
    double yi_akima = gsl_spline_eval(spline_akima, xi, acc);
    double yi_steffen = gsl_spline_eval(spline_steffen, xi, acc);

    printf("%g %g %g %g\n", xi, yi_cubic, yi_akima, yi_steffen);
  }

gsl_spline_free(spline_cubic);
gsl_spline_free(spline_akima);
gsl_spline_free(spline_steffen);
gsl_interp_accel_free(acc);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  return 0;
}
```

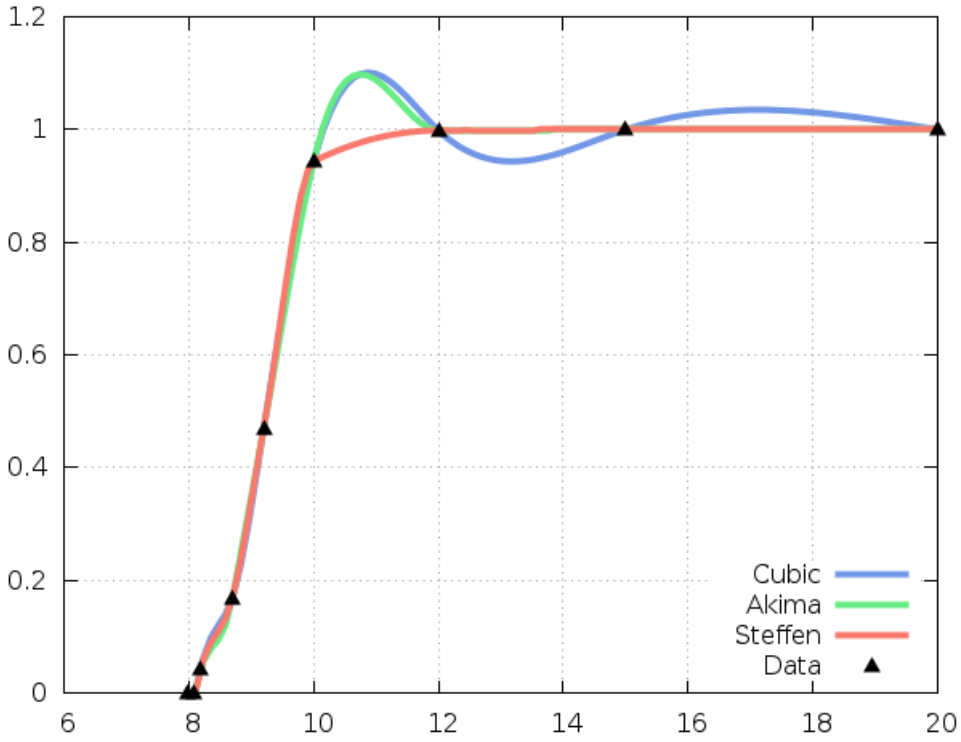


그림 29.3: Comparison of different 1D interpolation methods

The output is shown in 그림 29.3. The cubic method exhibits a local maxima between the 6th and 7th data points and continues oscillating for the rest of the data. Akima also shows a local maxima but recovers and follows the data well after the 7th grid point. Steffen preserves monotonicity in all intervals and does not exhibit oscillations, at the expense of having a discontinuous second derivative.

29.8 2D 보간법의 개요

Given a set of x coordinates $x_1, ..., x_m$ and a set of y coordinates $y_1, ..., y_n$, each in increasing order, plus a set of function values z_{ij} for each grid point (x_i, y_j), the routines described in this section compute a continuous interpolation function $z(x, y)$ such that $z(x_i, y_j) = z_{ij}$.

29.9 2D 보간 함수

The interpolation function for a given dataset is stored in a `gsl_interp2d` object. These are created by the following functions.

type **gsl_interp2d**

Workspace for 2D interpolation

gsl_interp2d ***gsl_interp2d_alloc**(const gsl_interp2d_type *T, const size_t xsize, const size_t ysize)

This function returns a pointer to a newly allocated interpolation object of type `T` for `xsize` grid points in the x direction and `ysize` grid points in the y direction.

int **gsl_interp2d_init**(gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const size_t xsize, const size_t ysize)

This function initializes the interpolation object `interp` for the data (`xa`, `ya`, `za`) where `xa` and `ya` are arrays of the x and y grid points of size `xsize` and `ysize` respectively, and `za` is an array of function values of size `xsize` * `ysize`. The interpolation object (`gsl_interp2d`) does not save the data arrays `xa`, `ya`, and `za` and only stores the static state computed from the data. The `xa` and `ya` data arrays are always assumed to be strictly ordered, with increasing x, y values; the behavior for other arrangements is not defined.

void **gsl_interp2d_free**(gsl_interp2d *interp)

This function frees the interpolation object `interp`.

29.10 2D Interpolation Grids

The 2D interpolation routines access the function values z_{ij} with the following ordering:

$$z_{ij} = za[j * xsize + i]$$

with $i = 0, ..., xsize - 1$ and $j = 0, ..., ysize - 1$. However, for ease of use, the following functions are provided to add and retrieve elements from the function grid without requiring knowledge of the internal ordering.

int **gsl_interp2d_set**(const gsl_interp2d *interp, double za[], const size_t i, const size_t j, const double z)

This function sets the value z_{ij} for grid point (`i`, `j`) of the array `za` to `z`.

double **gsl_interp2d_get**(const gsl_interp2d *interp, const double za[], const size_t i, const size_t j)

This function returns the value z_{ij} for grid point (i, j) stored in the array za.

size_t **gsl_interp2d_idx**(const gsl_interp2d *interp, const size_t i, const size_t j)

This function returns the index corresponding to the grid point (i, j). The index is given by $j * xsize + i$.

29.11 2D 보간법 유형

type **gsl_interp2d_type**

The interpolation library provides the following 2D interpolation types:

gsl_interp2d_type ***gsl_interp2d_bilinear**

Bilinear interpolation. This interpolation method does not require any additional memory.

gsl_interp2d_type ***gsl_interp2d_bicubic**

Bicubic interpolation.

const char ***gsl_interp2d_name**(const gsl_interp2d *interp)

This function returns the name of the interpolation type used by interp. For example:

```
printf ("interp uses '%s' interpolation.\n", gsl_interp2d_name (interp));
```

would print something like:

```
interp uses 'bilinear' interpolation.
```

unsigned int **gsl_interp2d_min_size**(const gsl_interp2d *interp)

unsigned int **gsl_interp2d_type_min_size**(const gsl_interp2d_type *T)

These functions return the minimum number of points required by the interpolation object interp or interpolation type T. For example, bicubic interpolation requires a minimum of 4 points.

29.12 2D Evaluation of Interpolating Functions

double **gsl_interp2d_eval**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_interp2d_eval_e**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *z)

These functions return the interpolated value of `z` for a given point (`x`, `y`), using the interpolation object `interp`, data arrays `xa`, `ya`, and `za` and the accelerators `xacc` and `yacc`. When `x` is outside the range of `xa` or `y` is outside the range of `ya`, the error code `GSL_EDOM` is returned.

double **gsl_interp2d_eval_extrap**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_interp2d_eval_extrap_e**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *z)

These functions return the interpolated value of `z` for a given point (`x`, `y`), using the interpolation object `interp`, data arrays `xa`, `ya`, and `za` and the accelerators `xacc` and `yacc`. The functions perform no bounds checking, so when `x` is outside the range of `xa` or `y` is outside the range of `ya`, extrapolation is performed.

double **gsl_interp2d_eval_deriv_x**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_interp2d_eval_deriv_x_e**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

These functions return the interpolated value `d` $= \partial z/\partial x$ for a given point (`x`, `y`), using the interpolation object `interp`, data arrays `xa`, `ya`, and `za` and the accelerators `xacc` and `yacc`. When `x` is outside the range of `xa` or `y` is outside the range of `ya`, the error code `GSL_EDOM` is returned.

double **gsl_interp2d_eval_deriv_y**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_interp2d_eval_deriv_y_e**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

These functions return the interpolated value `d` $= \partial z/\partial y$ for a given point (`x`, `y`), using the interpolation object `interp`, data arrays `xa`, `ya`, and `za` and the accelerators `xacc` and `yacc`. When `x` is outside the range of `xa` or `y` is outside the range of `ya`, the error code `GSL_EDOM` is returned.

double **gsl_interp2d_eval_deriv_xx**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_interp2d_eval_deriv_xx_e**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

These functions return the interpolated value `d` $= \partial^2 z/\partial x^2$ for a given point (`x`, `y`), using the interpolation object `interp`, data arrays `xa`, `ya`, and `za` and the accelerators `xacc` and `yacc`. When `x` is outside the range of `xa` or `y` is outside the range of `ya`, the error code `GSL_EDOM` is returned.

double **gsl_interp2d_eval_deriv_yy**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_interp2d_eval_deriv_yy_e**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

These functions return the interpolated value `d` $= \partial^2 z/\partial y^2$ for a given point (`x`, `y`), using the interpolation object `interp`, data arrays `xa`, `ya`, and `za` and the accelerators `xacc` and `yacc`. When `x` is outside the range of `xa` or `y` is outside the range of `ya`, the error code `GSL_EDOM` is returned.

double **gsl_interp2d_eval_deriv_xy**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_interp2d_eval_deriv_xy_e**(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

These functions return the interpolated value `d` $= \partial^2 z/\partial x \partial y$ for a given point (`x`, `y`), using the interpolation object `interp`, data arrays `xa`, `ya`, and `za` and the accelerators `xacc` and `yacc`. When `x` is outside the range of `xa` or `y` is outside the range of `ya`, the error code `GSL_EDOM` is returned.

29.13 2D 고수준 인터페이스

The functions described in the previous sections required the user to supply pointers to the x, y, and z arrays on each call. The following functions are equivalent to the corresponding `gsl_interp2d` functions but maintain a copy of this data in the `gsl_spline2d` object. This removes the need to pass `xa`, `ya`, and `za` as arguments on each evaluation. These functions are defined in the header file `gsl_spline2d.h`.

type **gsl_spline2d**

This workspace provides a higher level interface for the `gsl_interp2d` object

gsl_spline2d ***gsl_spline2d_alloc**(const gsl_interp2d_type *T, size_t xsize, size_t ysize)

int **gsl_spline2d_init**(gsl_spline2d *spline, const double xa[], const double ya[], const double za[], size_t xsize, size_t ysize)

void **gsl_spline2d_free**(gsl_spline2d *spline)

const char ***gsl_spline2d_name**(const gsl_spline2d *spline)

unsigned int **gsl_spline2d_min_size**(const gsl_spline2d *spline)

double **gsl_spline2d_eval**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_spline2d_eval_e**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *z)

double **gsl_spline2d_eval_extrap**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_spline2d_eval_extrap_e**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *z)

double **gsl_spline2d_eval_deriv_x**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_spline2d_eval_deriv_x_e**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

double **gsl_spline2d_eval_deriv_y**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_spline2d_eval_deriv_y_e**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

double **gsl_spline2d_eval_deriv_xx**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_spline2d_eval_deriv_xx_e**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

double **gsl_spline2d_eval_deriv_yy**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_spline2d_eval_deriv_yy_e**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

double **gsl_spline2d_eval_deriv_xy**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int **gsl_spline2d_eval_deriv_xy_e**(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *d)

int **gsl_spline2d_set**(const gsl_spline2d *spline, double za[], const size_t i, const size_t j, const double z)

double **gsl_spline2d_get**(const gsl_spline2d *spline, const double za[], const size_t i, const size_t j)

This function returns the value z_{ij} for grid point (i, j) stored in the array za.

29.14 2D 보간법의 예제

The following example performs bilinear interpolation on the unit square, using z values of $(0, 1, 0.5, 1)$ going clockwise around the square.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_interp2d.h>
#include <gsl/gsl_spline2d.h>

int
main()
{
  const gsl_interp2d_type *T = gsl_interp2d_bilinear;
  const size_t N = 100;             /* number of points to interpolate */
  const double xa[] = { 0.0, 1.0 }; /* define unit square */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  const double ya[] = { 0.0, 1.0 };
  const size_t nx = sizeof(xa) / sizeof(double); /* x grid points */
  const size_t ny = sizeof(ya) / sizeof(double); /* y grid points */
  double *za = malloc(nx * ny * sizeof(double));
  gsl_spline2d *spline = gsl_spline2d_alloc(T, nx, ny);
  gsl_interp_accel *xacc = gsl_interp_accel_alloc();
  gsl_interp_accel *yacc = gsl_interp_accel_alloc();
  size_t i, j;

  /* set z grid values */
  gsl_spline2d_set(spline, za, 0, 0, 0.0);
  gsl_spline2d_set(spline, za, 0, 1, 1.0);
  gsl_spline2d_set(spline, za, 1, 1, 0.5);
  gsl_spline2d_set(spline, za, 1, 0, 1.0);

  /* initialize interpolation */
  gsl_spline2d_init(spline, xa, ya, za, nx, ny);

  /* interpolate N values in x and y and print out grid for plotting */
  for (i = 0; i < N; ++i)
    {
      double xi = i / (N - 1.0);

      for (j = 0; j < N; ++j)
        {
          double yj = j / (N - 1.0);
          double zij = gsl_spline2d_eval(spline, xi, yj, xacc, yacc);

          printf("%f %f %f\n", xi, yj, zij);
        }
      printf("\n");
    }

  gsl_spline2d_free(spline);
  gsl_interp_accel_free(xacc);
  gsl_interp_accel_free(yacc);
  free(za);

  return 0;
}
```

The results of the interpolation are shown in 그림 29.4, where the corners are labeled with their fixed z values.

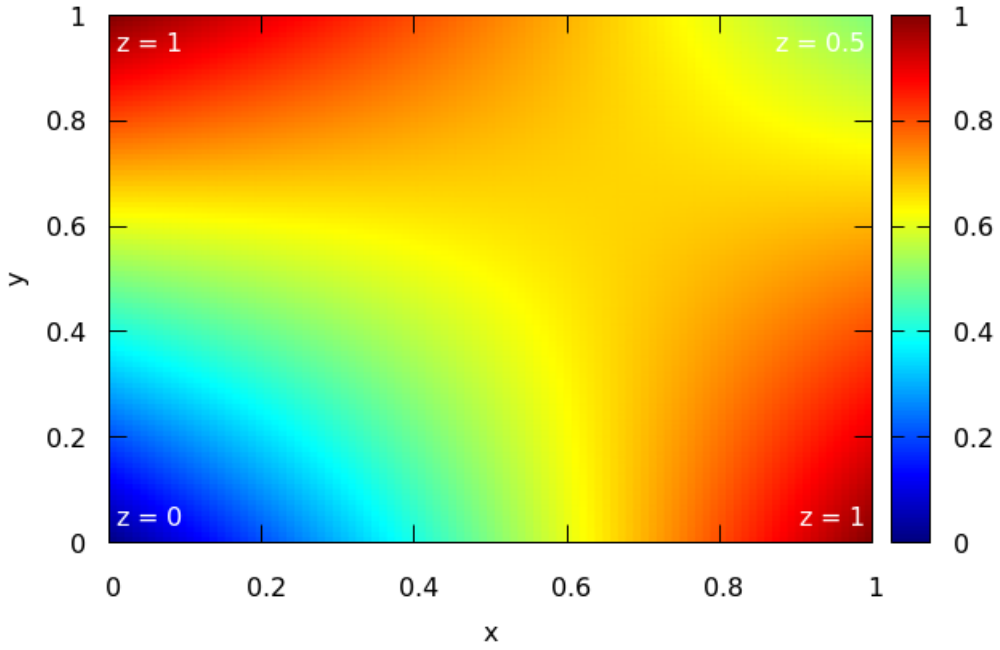


그림 29.4: 2D interpolation example

29.15 참고 문헌과 추가 자료

보간법 알고리즘에 대한 자세한 기술과 추가 자료들은 다음 출판물들에서 참고할 수 있습니다.

- C.W. Ueberhuber, Numerical Computation (Volume 1), Chapter 9 “Interpolation”, Springer (1997), ISBN 3-540-62058-3.
- D.M. Young, R.T. Gregory, A Survey of Numerical Mathematics (Volume 1), Chapter 6.8, Dover (1988), ISBN 0-486-65691-8.
- M. Steffen, A simple method for monotonic interpolation in one dimension, Astron. Astrophys. 239, 443-450, 1990.

제 30 장

수치 미분

이 단원에서 기술하는 함수는 유한 차분을 이용해 수치 미분을 계산합니다. 이 함수는 적응형 알고리즘을 이용해서 유한 차분들 중 최적값을 찾고 오류를 추정합니다. 이 단원에서 기술하는 함수들은 헤더 파일 `gsl_deriv.h` 에 기술되어 있습니다.

30.1 함수

int **gsl_deriv_central**(const gsl_function *f, double x, double h, double *result, double *abserr)

x 지점에서 함수 f 의 중간 미분 계수를 계산합니다. 이 방법은 구간의 크기를 나타내는 변수 h 를 이용하고, 결과는 $result$ 변수에 절대 오차는 $abser$ 저장됩니다.

초기값 h 는 계산에서 최적 구간을 얻기 위해 사용합니다. 이 과정은 절단 오차와 반올림 오차를 조절해 얻어집니다. 미분값은 5개 지점의 값을 사용해 계산합니다. 공간에서 각각 `x-h`, `x-\frac{h}{2}`, `x`, `x+ \frac{h}{2}`, `x+h` 인 지점들을 의미합니다. 미분 과정에서 오차 계산은 5개 지점 방법과 3개 지점 `x-h`, `x`, `x+h` 을 사용한 결과의 차이를 이용해 계산합니다. x 지점의 값은 미분을 계산하는데 영향을 미치지 못합니다. 따라서 실제로는 4개 지점만이 사용됩니다.

int **gsl_deriv_forward**(const gsl_function *f, double x, double h, double *result, double *abserr)

x 지점에서 함수 f 의 우미분 계수를 계산합니다. 이 방법은 구간의 크기를 나타내는 변수 h 를 이용합니다. x 보다 큰 지점들만을 이용하고 x 은 계산에 쓰이지 않습니다. 계산 결과는 $result$ 변수에 절대 오차는 $abser$ 저장됩니다. $f(x)$ 가 x 에서 연속적이지 않거나, x 보다 작은 값에서 정의되지 않은 경우에는 이 함수를 사용해야 합니다.

초기값 h 는 계산에서 최적 구간을 얻기 위해 사용합니다. 이 과정은 절단 오차와 반올림 오차를 조절해 얻어집니다. 미분은 열린 4개 지점 방법을 이용해 계산합니다. 공간에서 각각 `x+\frac{h}{4}`,

x+\frac{h}{2}, x+\frac{3h}{4}, x.h 인 지점을 의미합니다. 미분 과정에서 오차의 계산은 4개 지점 방법과 2개 지점 x+\frac{3h}{4}, x.h 을 사용한 방법의 차이를 이용해 계산합니다.

int **gsl_deriv_backward**(const gsl_function *f, double x, double h, double *result, double *abserr)

x 지점에서 함수 f 의 좌미분 계수를 계산합니다. 이 방법은 구간의 크기를 나타내는 변수 h 를 이용합니다. x 보다 작은 지점들만을 이용하고 x 은 계산에 쓰이지 않습니다. 계산 결과는 $result$ 변수에 절대 오차는 $abser$ 에 저장됩니다. $f(x)$ 가 x 에서 연속적이지 않거나, x 보다 큰 값에서 정의되지 않은 경우에는 이 함수를 사용해야 합니다.

이 함수는 단계의 크기를 음수로 했을 경우 $gsl_deriv_forward()$ 함수와 동일합니다.

30.2 예제

다음 코드는 $x = 2$ 와 $x = 0$ 지점에서 함수 $f(x) = x^{3/2}$ 의 미분값을 계산합니다. 함수 $f(x)$ 는 $x < 0$ 에서 정의되지 않았기 때문에 $x = 0$ 에서는 $gsl_derive_forward()$ 가 사용합니다.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_deriv.h>

double f (double x, void * params)
{
  (void)(params); /* avoid unused parameter warning */
  return pow (x, 1.5);
}

int
main (void)
{
  gsl_function F;
  double result, abserr;

  F.function = &f;
  F.params = 0;

  printf ("f(x) = x^(3/2)\n");

  gsl_deriv_central (&F, 2.0, 1e-8, &result, &abserr);
  printf ("x = 2.0\n");
  printf ("f'(x) = %.10f +/- %.10f\n", result, abserr);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  printf ("exact = %.10f\n\n", 1.5 * sqrt(2.0));

  gsl_deriv_forward (&F, 0.0, 1e-8, &result, &abserr);
  printf ("x = 0.0\n");
  printf ("f'(x) = %.10f +/- %.10f\n", result, abserr);
  printf ("exact = %.10f\n", 0.0);

  return 0;
}
```

다음은 프로그램의 실행 결과 입니다.

```
f(x) = x^(3/2)
x = 2.0
f'(x) = 2.1213203120 +/- 0.0000005006
exact = 2.1213203436

x = 0.0
f'(x) = 0.0000000160 +/- 0.0000000339
exact = 0.0000000000
```

30.3 참고 문헌과 추가 자료

이 함수들에 쓰인 알고리즘은 다음 문헌에서 찾을 수 있습니다.

- Abramowitz and Stegun, Handbook of Mathematical Functions, Section 25.3.4, and Table 25.5 (Coefficients for Differentiation).
- S.D. Conte and Carl de Boor, Elementary Numerical Analysis: An Algorithmic Approach, McGraw-Hill, 1972.

제 31 장

체비쇼프 근사

참고: 번역중

This chapter describes routines for computing Chebyshev approximations to univariate functions. A Chebyshev approximation is a truncation of the series $f(x) = \sum c_n T_n(x)$, where the Chebyshev polynomials $T_n(x) = \cos(n \arccos x)$ provide an orthogonal basis of polynomials on the interval $[-1, 1]$ with the weight function $1/\sqrt{1-x^2}$. The first few Chebyshev polynomials are, $T_0(x) = 1$, $T_1(x) = x$, $T_2(x) = 2x^2 - 1$. For further information see Abramowitz & Stegun, Chapter 22.

The functions described in this chapter are declared in the header file `gsl_chebyshev.h`.

31.1 Definitions

type **gsl_cheb_series**

A Chebyshev series is stored using the following structure:

```
typedef struct
{
  double * c;   /* coefficients  c[0] .. c[order] */
  int order;    /* order of expansion             */
  double a;     /* lower interval point           */
  double b;     /* upper interval point           */
  ...
} gsl_cheb_series
```

The approximation is made over the range $[a, b]$ using `order` + 1 terms, including the coefficient $c[0]$. The series is computed using the following convention,

$$f(x) = \frac{c_0}{2} + \sum_{n=1} c_n T_n(x)$$

which is needed when accessing the coefficients directly.

31.2 Creation and Calculation of Chebyshev Series

gsl_cheb_series *__gsl_cheb_alloc__(const size_t n)

This function allocates space for a Chebyshev series of order `n` and returns a pointer to a new `gsl_cheb_series` struct.

void __gsl_cheb_free__(gsl_cheb_series *cs)

This function frees a previously allocated Chebyshev series `cs`.

int __gsl_cheb_init__(gsl_cheb_series *cs, const gsl_function *f, const double a, const double b)

This function computes the Chebyshev approximation `cs` for the function `f` over the range (a, b) to the previously specified order. The computation of the Chebyshev approximation is an $O(n^2)$ process, and requires n function evaluations.

31.3 Auxiliary Functions

The following functions provide information about an existing Chebyshev series.

size_t __gsl_cheb_order__(const gsl_cheb_series *cs)

This function returns the order of Chebyshev series `cs`.

size_t __gsl_cheb_size__(const gsl_cheb_series *cs)

double *__gsl_cheb_coeffs__(const gsl_cheb_series *cs)

These functions return the size of the Chebyshev coefficient array `c[]` and a pointer to its location in memory for the Chebyshev series `cs`.

31.4 Chebyshev Series Evaluation

double **gsl_cheb_eval**(const gsl_cheb_series *cs, double x)

This function evaluates the Chebyshev series `cs` at a given point `x`.

int **gsl_cheb_eval_err**(const gsl_cheb_series *cs, const double x, double *result, double *abserr)

This function computes the Chebyshev series `cs` at a given point `x`, estimating both the series `result` and its absolute error `abserr`. The error estimate is made from the first neglected term in the series.

double **gsl_cheb_eval_n**(const gsl_cheb_series *cs, size_t order, double x)

This function evaluates the Chebyshev series `cs` at a given point `x`, to (at most) the given order `order`.

int **gsl_cheb_eval_n_err**(const gsl_cheb_series *cs, const size_t order, const double x, double *result, double *abserr)

This function evaluates a Chebyshev series `cs` at a given point `x`, estimating both the series `result` and its absolute error `abserr`, to (at most) the given order `order`. The error estimate is made from the first neglected term in the series.

31.5 Derivatives and Integrals

The following functions allow a Chebyshev series to be differentiated or integrated, producing a new Chebyshev series. Note that the error estimate produced by evaluating the derivative series will be underestimated due to the contribution of higher order terms being neglected.

int **gsl_cheb_calc_deriv**(gsl_cheb_series *deriv, const gsl_cheb_series *cs)

This function computes the derivative of the series `cs`, storing the derivative coefficients in the previously allocated `deriv`. The two series `cs` and `deriv` must have been allocated with the same order.

int **gsl_cheb_calc_integ**(gsl_cheb_series *integ, const gsl_cheb_series *cs)

This function computes the integral of the series `cs`, storing the integral coefficients in the previously allocated `integ`. The two series `cs` and `integ` must have been allocated with the same order. The lower limit of the integration is taken to be the left hand end of the range `a`.

31.6 Examples

The following example program computes Chebyshev approximations to a step function. This is an extremely difficult approximation to make, due to the discontinuity, and was chosen as an example where approximation error is visible. For smooth functions the Chebyshev approximation converges extremely rapidly and errors would not be visible.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_chebyshev.h>

double
f (double x, void *p)
{
  (void)(p); /* avoid unused parameter warning */

  if (x < 0.5)
    return 0.25;
  else
    return 0.75;
}

int
main (void)
{
  int i, n = 10000;

  gsl_cheb_series *cs = gsl_cheb_alloc (40);

  gsl_function F;

  F.function = f;
  F.params = 0;

  gsl_cheb_init (cs, &F, 0.0, 1.0);

  for (i = 0; i < n; i++)
    {
      double x = i / (double)n;
      double r10 = gsl_cheb_eval_n (cs, 10, x);
      double r40 = gsl_cheb_eval (cs, x);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      printf ("%g %g %g %g\n",
              x, GSL_FN_EVAL (&F, x), r10, r40);
    }

  gsl_cheb_free (cs);

  return 0;
}
```

그림 31.1 shows output from the program with the original function, 10-th order approximation and 40-th order approximation, all sampled at intervals of 0.001 in x.

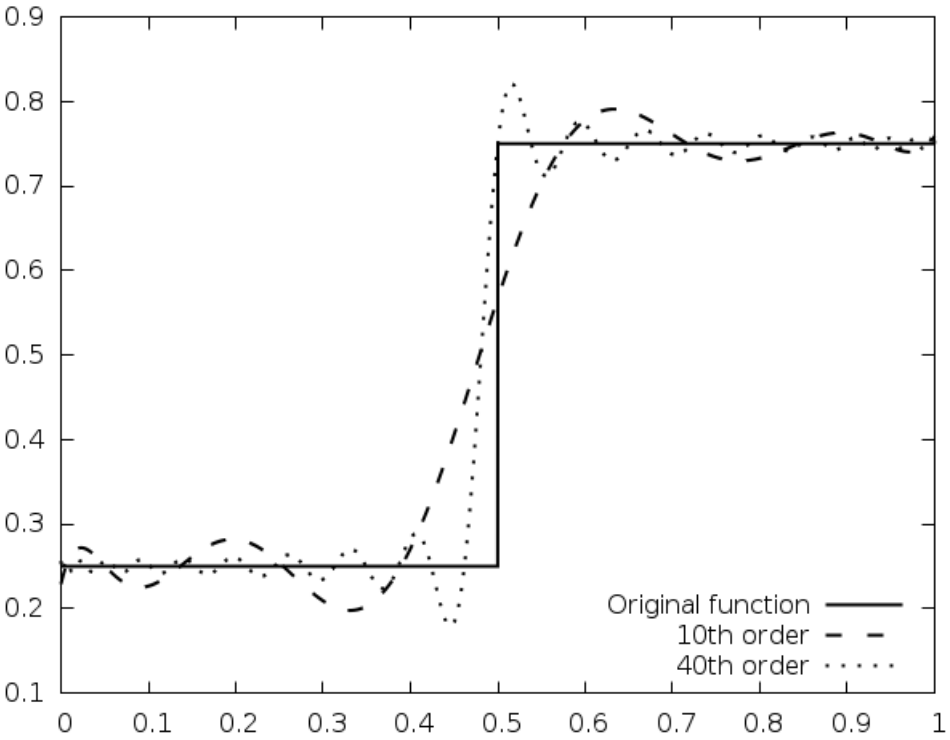


그림 31.1: Chebyshev approximations to a step function

31.7 References and Further Reading

The following paper describes the use of Chebyshev series,

- R. Broucke, "Ten Subroutines for the Manipulation of Chebyshev Series [C1] (Algorithm 446)". Communications of the ACM 16(4), 254–256 (1973)

제 32 장

급수 가속

참고: 번역중

이 단원은 Levin u 변환을 이용해 급수의 수렴을 가속하는 방법에 관해 기술합니다. 이 방법은 급수의 초기 항들에 외삽법(extrapolation)을 적용해 수렴값을 근사해 계산하고 근사 값의 오차도 같이 계산해 제공해 줍니다. u 변환은 수렴하는 급수, 발산하는 급수 그리고 점근적 확장을 표현하는 급수에도 사용할 수 있습니다.

헤더파일 `gsl_sum.h` 에 기술되어 있습니다.

32.1 가속 함수

다음 함수들은 급수의 Levin u 변환을 오차 값과 함께 계산합니다. 오차는 마지막 외삽항까지 각각 항의 오차 전파를 이용해 계산합니다. 외삽법의 마지막 항까지 각각의 항을 계산하는 과정에서 오차 전파를 이용해 추정합니다.

이 함수들은 해석적 급수들을 계산하기 위해 만들어졌습니다. 이러한 급수들은 각 항들이 높은 정확도의 수치로 알려져있고 반올림 오차는 한정된 정밀도에서 발생된다고 가정됩니다. 각 항들의 상대 오차는 `GSL_DBL_EPSILON` 수준의 정밀도로 정해집니다.

외삽된 값들의 오차 계산은 $O(N^2)$ 의 계산 복잡도를 가집니다. 이 복잡도는 시간과 자원을 매우 많이 소모합니다. 정확도는 낮지만 외삽법의 수렴 오차를 더 빠르게 추정할 수 있는 방법도 있습니다. 해당 방법은 다음 소단원에 기술되어 있습니다. 이 단원에서 기술하는 방법은 모든 중간값과 미분 값을 사용하는 방법으로 $O(N)$ 복잡도의 자원이 계산 과정에서 필요하지만 신뢰성 있는 오차 추정치를 제공해줍니다.

type **gsl_sum_levin_u_workspace**

Levin u 변환을 위한 작업 공간입니다.

gsl_sum_levin_u_workspace *__gsl_sum_levin_u_alloc__(size_t n)

Levin u 변환을 위한 작업 공간을 메모리에 할당합니다. 공간의 크기는 $O(2n^2 + 3n)$ 입니다.

void __gsl_sum_levin_u_free__(gsl_sum_levin_u_workspace *w)

w 가 가르키는 작업 공간을 메모리에서 해제합니다.

int __gsl_sum_levin_u_accel__(const double *array, size_t array_size, gsl_sum_levin_u_workspace *w, double *sum_accel, double *abserr)

크기 array_size 의 배열 array 로 주어진 급수 항들을 이용해 급수의 극한 값을 계산합니다. 이 방법은 외삽법의 일종으로 Levin u 변환을 이용합니다. 함수의 호출 과정에서 별도의 작업 공간을 반드시 w 에 제공해주어야 합니다.

외삽된 합들은 sum_accel 에 저장됩니다. 해당 값들에 대한 절대 오차는 abserr 에 저장됩니다. 실제 항들의 합은 w->sum_plain 에 반환됩니다. 이 함수에 사용된 알고리즘은 절단 오차와 반올림 오차를 계산해 외삽법을 위한 최적화된 항 갯수를 찾습니다. 절단 오차는 2개의 외삽법들의 차로 계산되고 반올림 오차는 각 항의 계산의 오차 전파를 이용합니다. array 를 통해 함수에 전달되는 모든 급수 항들은 반드시 0이 아니어야 합니다.

32.2 오차 추정이 없는 가속 함수

이 단락의 함수들은 Levin u 변환을 계산할 때, 오차 추정을 절단 오차(truncation error)를 이용해 계산합니다. 이 값은 마지막 두 근사 값의 차를 이용해 계산됩니다. 오차가 외삽법의 계산에서 추정되므로 중간에 미분 값 표를 계산할 필요가 없습니다. 결과적으로 이 알고리즘은 $O(n)$ 의 복잡도를 가지고 $O(n)$ 크기의 저장공간만을 필요로 합니다. 급수가 충분히 빠르게 수렴한다면 이 방법을 사용해 좋은 결과를 얻을 수 있습니다. 이 방법을 사용하기 적절한 상황은 비슷한 수렴 성질을 가지는 여러 급수들을 외삽법을 이용해 빠르게 계산하는 상황입니다. 예를 들어서, 어느 함수가 여러 매개 변수들을 가진 급수로 정의되었고 매개 변수들의 값 차이가 그리 크지 않은 여러 경우에 대한 수치 적분을 계산하는 상황이 있습니다. 전 단락에 쓰인 알고리즘을 이용해 먼저 적절한 오차 추정치를 계산해 계산 결과의 일관성을 검증해 볼 수도 있습니다.

type __gsl_sum_levin_utrunc_workspace__

오차 추정 계산을 하지 않는 Levin u 변환을 위한 작업 공간입니다.

gsl_sum_levin_utrunc_workspace *__gsl_sum_levin_utrunc_alloc__(size_t n)

오차 추정 계산이 없는 크기 n 의 Levin u 변환을 위한 작업 공간을 메모리에 할당합니다. 작업 공간의 크기는 $O(3n)$ 입니다.

void __gsl_sum_levin_utrunc_free__(gsl_sum_levin_utrunc_workspace *w)

w 가 가르키는 작업 공간을 메모리에서 해제합니다.

int __gsl_sum_levin_utrunc_accel__(const double *array, size_t array_size, gsl_sum_levin_utrunc_workspace *w, double *sum_accel, double *abserr_trunc)

크기 array_size 의 배열 array 로 주어진 급수 항들을 이용해 급수의 극한 값을 계산합니다. 이 방법은 외삽법의 일종으로 Levin u 변환을 이용합니다. 함수의 호출 과정에서 별도의 작업 공간을 반드시 w 에 제공해주어야 합니다.

외삽된 합들은 sum_accel 에 저장됩니다. 해당 값들에 대한 절대 오차는 abserr 에 저장됩니다. 실제 항들의 합은 w->sum_plain 에 반환됩니다.

이 알고리즘은 2개의 외삽법이 성공적으로 계산되었을 때, 해당 값들의 차이가 최소에 이르거나 충분히 작을 때 정지합니다. 차이는 오차를 추정하는 데 사용되며 abserr_trunc 에 저장됩니다. 알고리즘의 신뢰성을 개선하기 위해 외삽된 값들이 절단 오차를 계산할 때 이동 평균으로 대체되어 변동치를 줄여줍니다.

32.3 예제

다음 프로그램은 $\zeta(2) = \pi^2/6$ 의 값을 급수를 이용해 계산합니다.

The following code calculates an estimate of $\zeta(2) = \pi^2/6$ using the series,

$$\zeta(2) = 1 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$$

N 항 이후 급수의 오차는 $O(1/N)$ 를 따릅니다. 직접 급수를 계산하는 방법은 수렴 속도가 매우 느립니다.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_sum.h>

#define N 20

int
main (void)
{
  double t[N];
  double sum_accel, err;
  double sum = 0;
  int n;

  gsl_sum_levin_u_workspace * w
    = gsl_sum_levin_u_alloc (N);

  const double zeta_2 = M_PI * M_PI / 6.0;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  /* terms for zeta(2) = \sum_{n=1}^{\infty} 1/n^2 */

  for (n = 0; n < N; n++)
    {
      double np1 = n + 1.0;
      t[n] = 1.0 / (np1 * np1);
      sum += t[n];
    }

  gsl_sum_levin_u_accel (t, N, w, &sum_accel, &err);

  printf ("term-by-term sum = % .16f using %d terms\n",
          sum, N);

  printf ("term-by-term sum = % .16f using %zu terms\n",
          w->sum_plain, w->terms_used);

  printf ("exact value      = % .16f\n", zeta_2);
  printf ("accelerated sum  = % .16f using %zu terms\n",
          sum_accel, w->terms_used);

  printf ("estimated error  = % .16f\n", err);
  printf ("actual error     = % .16f\n",
          sum_accel - zeta_2);

  gsl_sum_levin_u_free (w);
  return 0;
}
```

아래의 결과는 Levin u 변환으로 처음 11개의 항을 이용해 10^{10} 자리 숫자의 정확도를 가지는 값을 계산할 수 있음을 보여줍니다. 함수가 반환한 오차 추정치도 참 값의 유효 숫자 갯수와 동일한 결과를 보여줍니다.

```
term-by-term sum =  1.5961632439130233 using 20 terms
term-by-term sum =  1.5759958390005426 using 13 terms
exact value      =  1.6449340668482264
accelerated sum  =  1.6449340669228176 using 13 terms
estimated error  =  0.0000000000888360
actual error     =  0.0000000000745912
```

직접 급수를 계산해 13개의 가속 변환값을 이용한 결과와 동일한 정밀도를 얻으려면 10^{10} 개 항의 합이

필요합니다.

32.4 참고 문헌과 추가 자료

함수들에 쓰인 알고리즘들은 다음의 논문들에 기반합니다.

- T. Fessler, W.F. Ford, D.A. Smith, HURRY: An acceleration algorithm for scalar sequences and series ACM Transactions on Mathematical Software, 9(3):346–354, 1983. and Algorithm 602 9(3):355–357, 1983.

u 변환은 Levin의 논문에서 소개되어 있습니다.

- D. Levin, Development of Non-Linear Transformations for Improving Convergence of Sequences, Intern.: J.: Computer Math. B3:371–388, 1973.

Levin 변환에 대한 리뷰 논문을 온라인에서 찾을 수 있습니다.

- Herbert H. H. Homeier, Scalar Levin-Type Sequence Transformations, http://arxiv.org/abs/math/0005209

제 33 장

웨이블렛 변환

참고: 번역중

This chapter describes functions for performing Discrete Wavelet Transforms (DWTs). The library includes wavelets for real data in both one and two dimensions. The wavelet functions are declared in the header files `gsl_wavelet.h` and `gsl_wavelet2d.h`.

33.1 Definitions

The continuous wavelet transform and its inverse are defined by the relations,

$$w(s,\tau) = \int_{-\infty}^{\infty} f(t) * \psi^*_{s,\tau}(t) dt$$

and,

$$f(t) = \int_0^{\infty} ds \int_{-\infty}^{\infty} w(s,\tau) * \psi_{s,\tau}(t) d\tau$$

where the basis functions $\psi_{s,\tau}$ are obtained by scaling and translation from a single function, referred to as the mother wavelet.

The discrete version of the wavelet transform acts on equally-spaced samples, with fixed scaling and translation steps (s, τ). The frequency and time axes are sampled dyadically on scales of 2^j through a level parameter j.

The resulting family of functions $\{\psi_{j,n}\}$ constitutes an orthonormal basis for square-integrable signals. The discrete wavelet transform is an $O(N)$ algorithm, and is also referred to as the fast

wavelet transform.

33.2 Initialization

type **gsl_wavelet**

This structure contains the filter coefficients defining the wavelet and any associated offset parameters.

gsl_wavelet *__gsl_wavelet_alloc__(const gsl_wavelet_type *T, size_t k)

This function allocates and initializes a wavelet object of type `T`. The parameter `k` selects the specific member of the wavelet family. A null pointer is returned if insufficient memory is available or if a unsupported member is selected.

The following wavelet types are implemented:

type **gsl_wavelet_type**

gsl_wavelet_type ***gsl_wavelet_daubechies**

gsl_wavelet_type ***gsl_wavelet_daubechies_centered**

This is the Daubechies wavelet family of maximum phase with $k/2$ vanishing moments. The implemented wavelets are $k = 4, 6, \ldots, 20$, with `k` even.

gsl_wavelet_type ***gsl_wavelet_haar**

gsl_wavelet_type ***gsl_wavelet_haar_centered**

This is the Haar wavelet. The only valid choice of k for the Haar wavelet is $k = 2$.

gsl_wavelet_type ***gsl_wavelet_bspline**

gsl_wavelet_type ***gsl_wavelet_bspline_centered**

This is the biorthogonal B-spline wavelet family of order (i, j). The implemented values of $k = 100 * i + j$ are 103, 105, 202, 204, 206, 208, 301, 303, 305 307, 309.

The centered forms of the wavelets align the coefficients of the various sub-bands on edges. Thus the resulting visualization of the coefficients of the wavelet transform in the phase plane is easier to understand.

const char ***gsl_wavelet_name**(const gsl_wavelet *w)

This function returns a pointer to the name of the wavelet family for `w`.

void **gsl_wavelet_free**(gsl_wavelet *w)

This function frees the wavelet object `w`.

type **gsl_wavelet_workspace**

This structure contains scratch space of the same size as the input data and is used to hold intermediate results during the transform.

gsl_wavelet_workspace ***gsl_wavelet_workspace_alloc**(size_t n)

This function allocates a workspace for the discrete wavelet transform. To perform a one-dimensional transform on `n` elements, a workspace of size `n` must be provided. For two-dimensional transforms of `n`-by-`n` matrices it is sufficient to allocate a workspace of size `n`, since the transform operates on individual rows and columns. A null pointer is returned if insufficient memory is available.

void **gsl_wavelet_workspace_free**(gsl_wavelet_workspace *work)

This function frees the allocated workspace `work`.

33.3 Transform Functions

This sections describes the actual functions performing the discrete wavelet transform. Note that the transforms use periodic boundary conditions. If the signal is not periodic in the sample length then spurious coefficients will appear at the beginning and end of each level of the transform.

33.3.1 Wavelet transforms in one dimension

int **gsl_wavelet_transform**(const gsl_wavelet *w, double *data, size_t stride, size_t n, gsl_wavelet_direction dir, gsl_wavelet_workspace *work)

int **gsl_wavelet_transform_forward**(const gsl_wavelet *w, double *data, size_t stride, size_t n, gsl_wavelet_workspace *work)

int **gsl_wavelet_transform_inverse**(const gsl_wavelet *w, double *data, size_t stride, size_t n, gsl_wavelet_workspace *work)

These functions compute in-place forward and inverse discrete wavelet transforms of length `n` with stride `stride` on the array `data`. The length of the transform `n` is restricted to powers of two. For the `transform` version of the function the argument `dir` can be either `forward` ($+1$) or `backward` (-1). A workspace `work` of length `n` must be provided.

For the forward transform, the elements of the original array are replaced by the discrete wavelet transform $f_i \to w_{j,k}$ in a packed triangular storage layout, where `j` is the index of the level $j = 0 \dots J-1$ and `k` is the index of the coefficient within each level, $k = 0 \dots 2^j - 1$.

The total number of levels is $J = \log_2(n)$. The output data has the following form,

$$(s_{-1,0}, d_{0,0}, d_{1,0}, d_{1,1}, d_{2,0}, \cdots, d_{j,k}, \cdots, d_{J-1,2^{J-1}-1})$$

where the first element is the smoothing coefficient $s_{-1,0}$, followed by the detail coefficients $d_{j,k}$ for each level j. The backward transform inverts these coefficients to obtain the original data.

These functions return a status of `GSL_SUCCESS` upon successful completion. `GSL_EINVAL` is returned if `n` is not an integer power of 2 or if insufficient workspace is provided.

33.3.2 Wavelet transforms in two dimension

The library provides functions to perform two-dimensional discrete wavelet transforms on square matrices. The matrix dimensions must be an integer power of two. There are two possible orderings of the rows and columns in the two-dimensional wavelet transform, referred to as the "standard" and "non-standard" forms.

The "standard" transform performs a complete discrete wavelet transform on the rows of the matrix, followed by a separate complete discrete wavelet transform on the columns of the resulting row-transformed matrix. This procedure uses the same ordering as a two-dimensional Fourier transform.

The "non-standard" transform is performed in interleaved passes on the rows and columns of the matrix for each level of the transform. The first level of the transform is applied to the matrix rows, and then to the matrix columns. This procedure is then repeated across the rows and columns of the data for the subsequent levels of the transform, until the full discrete wavelet transform is complete. The non-standard form of the discrete wavelet transform is typically used in image analysis.

The functions described in this section are declared in the header file `gsl_wavelet2d.h`.

int **gsl_wavelet2d_transform**(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2, gsl_wavelet_direction dir, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_transform_forward**(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_transform_inverse**(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2, gsl_wavelet_workspace *work)

These functions compute two-dimensional in-place forward and inverse discrete wavelet transforms in standard form on the array `data` stored in row-major form with dimensions `size1` and `size2` and physical row length `tda`. The dimensions must be equal (square matrix) and are restricted to powers of two. For the `transform` version of the function

the argument `dir` can be either `forward` (+1) or `backward` (−1). A workspace `work` of the appropriate size must be provided. On exit, the appropriate elements of the array `data` are replaced by their two-dimensional wavelet transform.

The functions return a status of `GSL_SUCCESS` upon successful completion. `GSL_EINVAL` is returned if `size1` and `size2` are not equal and integer powers of 2, or if insufficient workspace is provided.

int **gsl_wavelet2d_transform_matrix**(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_direction dir, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_transform_matrix_forward**(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_transform_matrix_inverse**(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_workspace *work)

These functions compute the two-dimensional in-place wavelet transform on a matrix `m`.

int **gsl_wavelet2d_nstransform**(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2, gsl_wavelet_direction dir, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_nstransform_forward**(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_nstransform_inverse**(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2, gsl_wavelet_workspace *work)

These functions compute the two-dimensional wavelet transform in non-standard form.

int **gsl_wavelet2d_nstransform_matrix**(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_direction dir, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_nstransform_matrix_forward**(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_workspace *work)

int **gsl_wavelet2d_nstransform_matrix_inverse**(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_workspace *work)

These functions compute the non-standard form of the two-dimensional in-place wavelet transform on a matrix `m`.

33.4 Examples

The following program demonstrates the use of the one-dimensional wavelet transform functions. It computes an approximation to an input signal (of length 256) using the 20 largest components of the wavelet transform, while setting the others to zero.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_wavelet.h>

int
main (int argc, char **argv)
{
  (void)(argc); /* avoid unused parameter warning */
  int i, n = 256, nc = 20;
  double *orig_data = malloc (n * sizeof (double));
  double *data = malloc (n * sizeof (double));
  double *abscoeff = malloc (n * sizeof (double));
  size_t *p = malloc (n * sizeof (size_t));

  FILE * f;
  gsl_wavelet *w;
  gsl_wavelet_workspace *work;

  w = gsl_wavelet_alloc (gsl_wavelet_daubechies, 4);
  work = gsl_wavelet_workspace_alloc (n);

  f = fopen (argv[1], "r");
  for (i = 0; i < n; i++)
    {
      fscanf (f, "%lg", &orig_data[i]);
      data[i] = orig_data[i];
    }
  fclose (f);

  gsl_wavelet_transform_forward (w, data, 1, n, work);

  for (i = 0; i < n; i++)
    {
      abscoeff[i] = fabs (data[i]);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    }

  gsl_sort_index (p, abscoeff, 1, n);

  for (i = 0; (i + nc) < n; i++)
    data[p[i]] = 0;

  gsl_wavelet_transform_inverse (w, data, 1, n, work);

  for (i = 0; i < n; i++)
    {
      printf ("%g %g\n", orig_data[i], data[i]);
    }

  gsl_wavelet_free (w);
  gsl_wavelet_workspace_free (work);

  free (data);
  free (orig_data);
  free (abscoeff);
  free (p);
  return 0;
}
```

The output can be used with the GNU plotutils `graph` program:

```
$ ./a.out ecg.dat > dwt.txt
$ graph -T ps -x 0 256 32 -h 0.3 -a dwt.txt > dwt.ps
```

그림 33.1 shows an original and compressed version of a sample ECG recording from the MIT-BIH Arrhythmia Database, part of the PhysioNet archive of public-domain of medical datasets.

33.5 References and Further Reading

The mathematical background to wavelet transforms is covered in the original lectures by Daubechies,

- Ingrid Daubechies. Ten Lectures on Wavelets. CBMS-NSF Regional Conference Series in Applied Mathematics (1992), SIAM, ISBN 0898712742.

An easy to read introduction to the subject with an emphasis on the application of the wavelet

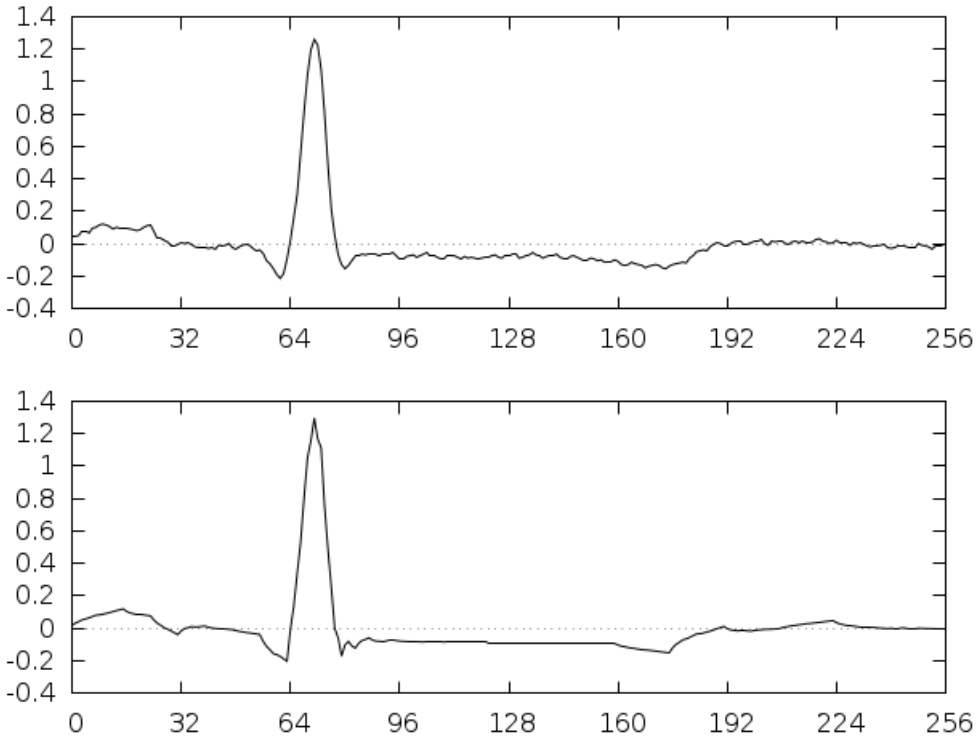


그림 33.1: Original (upper) and wavelet-compressed (lower) ECG signals, using the 20 largest components of the Daubechies(4) discrete wavelet transform.

transform in various branches of science is,

- Paul S. Addison. The Illustrated Wavelet Transform Handbook. Institute of Physics Publishing (2002), ISBN 0750306920.

For extensive coverage of signal analysis by wavelets, wavelet packets and local cosine bases see,

- S. G. Mallat. A wavelet tour of signal processing (Second edition). Academic Press (1999), ISBN 012466606X.

The concept of multiresolution analysis underlying the wavelet transform is described in,

- S. G. Mallat. Multiresolution Approximations and Wavelet Orthonormal Bases of L^2(R). Transactions of the American Mathematical Society, 315(1), 1989, 69–87.
- S. G. Mallat. A Theory for Multiresolution Signal Decomposition—The Wavelet Representation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11, 1989, 674–693.

The coefficients for the individual wavelet families implemented by the library can be found in the following papers,

- I. Daubechies. Orthonormal Bases of Compactly Supported Wavelets. Communications on Pure and Applied Mathematics, 41 (1988) 909–996.

- A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal Bases of Compactly Supported Wavelets. Communications on Pure and Applied Mathematics, 45 (1992) 485–560.

The PhysioNet archive of physiological datasets can be found online at http://www.physionet.org/ and is described in the following paper,

- Goldberger et al. PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals. Circulation 101(23):e215-e220 2000.

제 34 장

이산 한켈 변환

참고: 번역중

This chapter describes functions for performing Discrete Hankel Transforms (DHTs). The functions are declared in the header file `gsl_dht.h`.

34.1 Definitions

The discrete Hankel transform acts on a vector of sampled data, where the samples are assumed to have been taken at points related to the zeros of a Bessel function of fixed order; compare this to the case of the discrete Fourier transform, where samples are taken at points related to the zeroes of the sine or cosine function.

Starting with its definition, the Hankel transform (or Bessel transform) of order ν of a function f with $\nu > -1/2$ is defined as (see Johnson, 1987 and Lemoine, 1994)

$$F_\nu(u) = \int_0^\infty f(t) J_\nu(ut) t dt$$

If the integral exists, F_ν is called the Hankel transformation of f. The reverse transform is given by

$$f(t) = \int_0^\infty F_\nu(u) J_\nu(ut) u du$$

where $\int_0^\infty f(t) t^{1/2} dt$ must exist and be absolutely convergent, and where $f(t)$ satisfies Dirichlet's conditions (of limited total fluctuations) in the interval $[0, \infty]$.

Now the discrete Hankel transform works on a discrete function f, which is sampled on points $n = 1...M$ located at positions $t_n = (j_{\nu,n}/j_{\nu,M})X$ in real space and at $u_n = j_{\nu,n}/X$ in reciprocal space. Here, $j_{\nu,m}$ are the m-th zeros of the Bessel function $J_\nu(x)$ arranged in ascending order. Moreover, the discrete functions are assumed to be band limited, so $f(t_n) = 0$ and $F(u_n) = 0$ for $n > M$. Accordingly, the function f is defined on the interval $[0, X]$.

Following the work of Johnson, 1987 and Lemoine, 1994, the discrete Hankel transform is given by

$$F_\nu(u_m) = \frac{2X^2}{j_{\nu,M}^2} \sum_{k=1}^{M-1} f\left(\frac{j_{\nu,k}X}{j_{\nu,M}}\right) \frac{J_\nu(j_{\nu,m}j_{\nu,k}/j_{\nu,M})}{J_{\nu+1}(j_{\nu,k})^2}.$$

It is this discrete expression which defines the discrete Hankel transform calculated by GSL. In GSL, forward and backward transforms are defined equally and calculate $F_\nu(u_m)$. Following Johnson, the backward transform reads

$$f(t_k) = \frac{2}{X^2} \sum_{m=1}^{M-1} F\left(\frac{j_{\nu,m}}{X}\right) \frac{J_\nu(j_{\nu,m}j_{\nu,k}/j_{\nu,M})}{J_{\nu+1}(j_{\nu,m})^2}.$$

Obviously, using the forward transform instead of the backward transform gives an additional factor $X^4/j_{\nu,M}^2 = t_m^2/u_m^2$.

The kernel in the summation above defines the matrix of the ν-Hankel transform of size $M-1$. The coefficients of this matrix, being dependent on ν and M, must be precomputed and stored; the `gsl_dht` object encapsulates this data. The allocation function `gsl_dht_alloc()` returns a `gsl_dht` object which must be properly initialized with `gsl_dht_init()` before it can be used to perform transforms on data sample vectors, for fixed ν and M, using the `gsl_dht_apply()` function. The implementation allows to define the length X of the fundamental interval, for convenience, while discrete Hankel transforms are often defined on the unit interval instead of $[0, X]$.

Notice that by assumption $f(t)$ vanishes at the endpoints of the interval, consistent with the inversion formula and the sampling formula given above. Therefore, this transform corresponds to an orthogonal expansion in eigenfunctions of the Dirichlet problem for the Bessel differential equation.

34.2 Functions

type **gsl_dht**

Workspace for computing discrete Hankel transforms

gsl_dht ***gsl_dht_alloc**(size_t size)

This function allocates a Discrete Hankel transform object of size `size`.

int **gsl_dht_init**(gsl_dht *t, double nu, double xmax)

This function initializes the transform `t` for the given values of `nu` and `xmax`.

gsl_dht ***gsl_dht_new**(size_t size, double nu, double xmax)

This function allocates a Discrete Hankel transform object of size `size` and initializes it for the given values of `nu` and `xmax`.

void **gsl_dht_free**(gsl_dht *t)

This function frees the transform `t`.

int **gsl_dht_apply**(const gsl_dht *t, double *f_in, double *f_out)

This function applies the transform `t` to the array `f_in` whose size is equal to the size of the transform. The result is stored in the array `f_out` which must be of the same length.

Applying this function to its output gives the original data multiplied by $(X^2/j_{\nu,M})^2$, up to numerical errors.

double **gsl_dht_x_sample**(const gsl_dht *t, int n)

This function returns the value of the `n`-th sample point in the unit interval, $(j_{\nu,n+1}/j_{\nu,M})X$. These are the points where the function $f(t)$ is assumed to be sampled.

double **gsl_dht_k_sample**(const gsl_dht *t, int n)

This function returns the value of the `n`-th sample point in "k-space", $j_{\nu,n+1}/X$.

34.3 References and Further Reading

The algorithms used by these functions are described in the following papers,

- 아. Fisk Johnson, Comp.: Phys.: Comm.: 43, 181 (1987).
- 라. Lemoine, J. Chem.: Phys.: 101, 3936 (1994).

제 35 장

함수의 근 탐색

이 단원에서는 임의의 1 변수 함수의 근을 찾는 함수들을 기술합니다. 라이브러리에서는 다양한 반복 풀이와 수렴 테스트를 위한 저수준의 기능들을 제공합니다. 이 기능들은 반복 단계를 완전히 제어할 수 있어, 잘 이용하면 사용자가 요구하는 적절한 해를 찾을 수 있습니다. 각각의 풀이 방법은 같은 작업 환경을 사용하기 때문에, 프로그램을 재컴파일 할 필요없이 여러 풀이 방법을 구동 중에 바꿀 수 있습니다. 각각의 풀이 인스턴스들은 스스로 상태를 추적하기 때문에, 다중 스레드 프로그램에서도 사용이 가능합니다.

이 단원에서 기술하는 함수들의 원형은 `gsl_roots.h` 에 정의되어 있습니다.

35.1 개요

1차원 근 탐색 알고리즘은 2가지로 나눌수 있습니다. 괄호법[1] 과 기울기 연마 방법입니다.

괄호법의 경우 근이 존재하는 닫힌 구간에서 알고리즘이 실행되고, 반복적으로 구간의 크기를 줄여 허용치를 만족할 때까지 반복합니다. 이는 정밀한 오차 추정치를 제공해 줄 수 있습니다.

기울기 연마 방법 경우는 근에 대한 초기 추측을 개선해 나가면 진행합니다. 이러한 알고리즘의 경우는 근에 **충분히 가까운** 지점에서 시작한 경우에만 적절한 근을 찾을 수 있고, 속도를 위해서 엄격한 오차값의 측정을 희생합니다. 이 알고리즘은 근 근처에서 함수의 동작을 근사해, 더 높은 미분 차수의 개선점을 찾아 초기 조건을 개선합니다. 함수가 알고리즘과 잘 들어맞고, 좋은 초기 조건을 추측할 수 있을 때, 이러한 알고리즘은 빠르게 근으로 수렴합니다.

GSL에서는 이 두 가지 형태의 알고리즘을 비슷한 이름의 작업 공간으로 제공합니다. 사용자는 알고리즘을 위한 고수준의 작업공간을 제공하고, 라이브러리는 각 단계에서 필요한 개별 함수들을 제공합니다. 근 탐색 과정은 3가지로 나뉘어집니다.

- 알고리즘 T 대한, 근 풀이 공간 s 선언.

[1] 근의 존재 구간이 주어진 경우(*)

- T 를 수행해 s 갱신.
- s 의 수렴성 판별, 필요하면 계속 반복.

존재 구간이 주어진 알고리즘은 $gsl_root_fsolver$ 작업공간을 줄 수 있고, 갱신 과정은 함수 자체만을 이용합니다(도함수를 사용하지 않습니다). 존재 구간이 주어지지 않은 경우는 $gsl_root_fdfsolver$ 작업공간을 설정합니다. 이 경우 함수와 함수의 도함수(fdf 라 부릅니다)가 갱신에 사용됩니다.

35.2 주의 사항

명심해야 할 점은 여기서 제공하는 근 탐색 함수가 1번에 1개의 근만을 찾는다는 사실입니다. 탐색 공간에 여러 근들이 존재한다면 가장 먼저 찾아진 근을 반환합니다. 하지만, 어떤 근이 반환될 지 예측하는 것은 매우 어렵습니다. 근이 여러개 있는 공간에서 한 근을 찾아도 대부분의 경우 오류는 발생하지 않습니다.

다중근을 가지는 함수를 다룰 때는 주의해야 합니다. 예를 들어서 $f(x) = (x-x_0)^2$ 나 $f(x) = (x-x_0)^3$ 등의 경우가 있습니다. 괄호법 알고리즘의 경우 다중근 중에서 짝수 중근을 가지는 함수에 사용할 수 없습니다. 알고리즘의 초기 구간은 반드시 0을 지나는 지점을 포함해, 구간의 한쪽은 양수를 다른쪽은 음수를 가져야 합니다. 하지만, 짝수번 중첩된 다중근 지점은 0을 지나지 않고 접하므로 초기 조건을 만족하지 못합니다. 홀수번 중첩된 근에 대해서는 잘 작동합니다(e.g. 3중근, 5중근…). 도함수를 이용하는 기울기 연마 방법은 일반적으로 큰 중첩을 가지는 다중근에서 잘 작동하지만, 수렴 속도가 감소합니다. 이 경우에, **스테퍼슨 방법** (Steffensen's method)을 사용해 다중근에 대해 수렴 속도를 늘릴 수 있습니다.

f 가 탐색 공간에서 반드시 근을 가질 필요는 없습니다만, 수치 근 탐색 함수는 근의 **존재 여부를 판별** 하기 위해 무분별하게 사용하는 것은 권장하지 않습니다. 근의 존재 여부 판별에는 더 좋은 방법들이 있습니다. 왜냐하면, 수치적 근 탐색은 일반적으로 실패할 가능성이 높기 때문입니다. 따라서, 함수에 대한 정보가 없을 경우, 바로 근 탐색으로 넘어가는 것은 좋은 생각이 아닙니다. 일반적으로 가장 좋은 방법은 근 탐색을 시작하기 전에 그래프를 그려서 시각적으로 확인을 해보는 것입니다.

또한, 함수에 따라 초기 추정 위치를 적절하게 정해주어야 합니다. 만일, 구간 오류가 발생한다면 그래프로 함수의 개형을 판별하고 적절하게 초기 추정치를 수정해 주어야 하는 경우도 있습니다(*).

35.3 풀이 시작하기

type **gsl_root_fsolver**

괄호법을 사용하는 근 탐색 알고리즘을 위한 풀이 공간입니다.

type **gsl_root_fdfsolver**

도함수가 필요한 근 탐색 알고리즘(기울기 연마 방법)을 위한 풀이 공간입니다.

gsl_root_fsolver ***gsl_root_fsolver_alloc**(const gsl_root_fsolver_type *T)

T 풀이 방법을 위한 새 풀이 공간을 할당해 포인터를 반환합니다. 예를 들어서 다음 코드는 이분법 방법을(Bisection method)을 위한 풀이 공간을 생성합니다.

```
const gsl_root_fsolver_type * T = gsl_root_fsolver_bisection;
gsl_root_fsolver * s = gsl_root_fsolver_alloc (T);
```

만약, 이 공간을 할당하기에 메모리가 충분하지 않다면, 함수는 $NULL$ 포인터를 반환하고 오류 관리자가 GSL_ENOMEM 코드의 오류를 보고합니다.

gsl_root_fdfsolver *gsl_root_fdfsolver_alloc(const gsl_root_fdfsolver_type *T)

미분 기반 풀이형 T 위한 새 풀이 공간을 할당해 포인터를 반환합니다. 예를 들어서 다음 코드는 뉴턴-렙슨 방법(Newton-Raphson method)를 위한 풀이 공간을 생성합니다.

```
const gsl_root_fdfsolver_type * T = gsl_root_fdfsolver_newton;
gsl_root_fdfsolver * s = gsl_root_fdfsolver_alloc (T);
```

만약, 이 공간을 할당하기에 메모리가 충분하지 않다면, 함수는 $NULL$ 포인터를 반환하고 오류 관리자가 GSL_ENOMEM 코드의 오류를 보고합니다.

int **gsl_root_fsolver_set**(gsl_root_fsolver *s, gsl_function *f, double x_lower, double x_upper)

함수 f 를 이용해 해를 찾는 풀이 공간 s 에 대해 초기 탐색 구간을 [x_lower x_upper] 로 잡고 풀이를 시작/재시작 시킵니다.

int **gsl_root_fdfsolver_set**(gsl_root_fdfsolver *s, gsl_function_fdf *fdf, double root)

함수 f 와 그 도함수 fdf 를 이용해 해를 찾는 풀이 공간 s 에 대해 초기 추정 위치를 $root$ 로 잡고 풀이를 시작/재시작 시킵니다.

void **gsl_root_fsolver_free**(gsl_root_fsolver *s)

void **gsl_root_fdfsolver_free**(gsl_root_fdfsolver *s)

풀이 공간 s 할당된 메모리 영역을 해제합니다.

const char ***gsl_root_fsolver_name**(const gsl_root_fsolver *s)

const char ***gsl_root_fdfsolver_name**(const gsl_root_fdfsolver *s)

풀이 방법의 이름을 담은 포인터를 반환합니다. 예를 들어서,

```
printf("s is a '%s' solver\n", gsl_root_fsolver_name(s));
```

는 $sisa'bisection'solver$ 반환합니다.

35.4 근을 찾을 함수

근 탐색을 위해서는 1변수 연속함수를 제공해야하고, 알고리즘에 따라서는 1계 도함수도 제공해야 합니다.

다음과 같이 함수를 정의하면, 일반적인 함수의 계수처럼 함수를 쓸 수 있습니다.

type **gsl_function**

이 자료형은 계수를 갖는 일반적인 함수를 정의합니다.

`double (* function) (double x, void * params)`

이 함수는 인자 x 계수 $params$ 대해 $f(x, params)$ 값을 반환하도록 정의해야합니다.

`void * params`

함수의 계수를 나타내는 포인터입니다.

일반적인 2차 다항 함수를 이용해 예를 들어보도록 하겠습니다.

$$f(x) = ax^2 + bx + c$$

$a = 3, b = 2, c = 1$ 이면, 다음 코드와 같이 $gsl_function$ 로 F 를 정의할 수 있습니다. 이 F 는 근 탐색 공간에 쓸 함수로 함수 포인터로 전달할 수 있습니다.

```
struct my_f_params { double a; double b; double c; };

double
my_f (double x, void * p)
{
    struct my_f_params * params = (struct my_f_params *)p;
    double a = (params->a);
    double b = (params->b);
    double c = (params->c);

    return  (a * x + b) * x + c;
}

gsl_function F;
struct my_f_params params = { 3.0, 2.0, 1.0 };

F.function = &my_f;
F.params = &params;
```

함수 $f(x)$ 는 매크로 $GSL_FN_EVAL(F, x)$ 를 이용해 평가해 볼 수 있습니다. 이는 gsl_math.h 에 정의되어 있습니다.

type **gsl_function_fdf**

이 자료형은 계수를 갖는 일반적인 함수와 이 함수의 1계 도함수를 정의합니다.

`double (* f) (double x, void * params)`

이 함수는 인자 x 와 계수 $params$ 에 대해 $f(x, params)$ 값을 반환하도록 정의해야합니다.

`double (* df) (double x, void * params)`

이 함수는 인자 x 와 계수 $params$ 에 대해 f 1계 도함수 값; $f'(x, params)$ 을 반환하도록 정의해야합니다.

`void (* fdf) (double x, void * params, double * f, double * df)`

이 함수는 인자 x 와 계수 $params$ 에 대해 함수 f $f(x, params)$ 와 그 1계 도함수 df $f'(x, params)$ 를 설정합니다. 이러한 방식은 독립된 함수 $f(x, params)$ 와 $f'(x, params)$ 를 제공함으로써 최적화를 시키는 방법으로, 도함수를 동시에 계산하는 방식에 비해 항상 빠릅니다.

`void * params`

함수의 계수를 나타내는 포인터입니다.

다음은 $f(x) = \exp 2x)$ 인 경우의 예시입니다.

```
double my_f (double x, void * params){
        return exp (2 * x);
    }

double my_df (double x, void * params){
        return 2 * exp (2 * x);
     }

void my_fdf (double x, void * params, double * f, double * df){
        double t = exp (2 * x);

        *f = t;
        *df = 2 * t;   /* uses existing value */
     }

gsl_function_fdf FDF;

FDF.f = &my_f;
FDF.df = &my_df;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
FDF.fdf = &my_fdf;
FDF.params = 0;
```

함수 $f(x)$ 는 매크로 $GSL_FN_FDF_EVAL_F(FDF, x)$ 를 이용해서, 도함수 $f'(x)$ 는 $GSL_FN_FDF_EVAL_DF(FDF, x)$ 이용해 평가해 볼 수 있습니다. 함수 $y = f(x)$ 와 도함수 $dy = f'(x)$ 는 매크로 $GSL_FN_FDF_EVAL_F_DF(FDF, x, y, dy)$ 를 써서 동시에 평가해 볼 수도 있습니다. 이 매크로는 $f(x)$ 를 y 인자에 그리고 $f'(x)$ 를 dy 인자에 저장합니다. 이 둘은 반드시 *double* 포인터여야 합니다.

35.5 탐색 경계와 추측 값

라이브러리에 있는 두 종류의 알고리즘 모두 탐색 경계나 추측 값을 필요로 합니다. 이 단락에서는 이러한 탐색 경계와 초기 값의 작동여부 그리고 함수 인자들이 어떻게 이들을 제어하는지 알아봅시다.

추측 값은 단순히 x 값을 의미합니다. 이 값은 원하는 근 정밀도가 될 때까지 알고리즘 내에서 반복됩니다. 이 값은 *double* 자료형을 가집니다.

탐색 경계는 구간의 끝 지점들을 의미합니다. 이는 구간의 길이가 요구되는 정밀도보다 작을 때까지 반복됩니다. 이 구간은 두 개 값, 하한 값과 상한 값으로 정의됩니다. 경계 값들을 간격에 포함할지, 말지 여부는 구간이 사용되는 풀이 흐름에 따라 달라집니다.

35.6 반복

다음 함수들은 각각의 알고리즘을 되풀이합니다. 각각의 함수는 상응하는 풀이공간의 상태를 1번의 반복 단계마다 갱신합니다. 동일한 기능이 모든 풀이에 작동하므로 코드를 수정하지 않고 실행 단계에서 다른 방법을 대체할 수 있습니다.

int **gsl_root_fsolver_iterate**(gsl_root_fsolver *s)

int **gsl_root_fdfsolver_iterate**(gsl_root_fdfsolver *s)

풀이공간 s 대한 단일 반복을 수행합니다. 만약 반복 과정에서 예상치 못한 문제가 생기면, 다음의 오류 값가 반환됩니다.

GSL_EBADFUNC

이는 반복과정에서 함수나 도함수가 Inf 나 NaN 값이 되는 특이점에 도달했다는 뜻입니다.

GSL_EZERODIV

함수의 미분값이 반복 지점에서 소멸해, 알고리즘이 0으로 나누는 것을 막았다는 뜻입니다.

이들은 탐색과정에서 언제나 현재 가장 좋은 근사 값을 유지합니다. 존재 구간이 주어진 경우 이에 더해, 근이 존재하는 가장 좋은 구간 값을 추적합니다. 이 정보는 다음과 같은, 보조 함수들을 이용해 접근할 수 있습니다.

double **gsl_root_fsolver_root**(const gsl_root_fsolver *s)

double **gsl_root_fdfsolver_root**(const gsl_root_fdfsolver *s)

호출시점에 풀이 공간 s 현재 저장된 근의 근사 값을 반환합니다.

double **gsl_root_fsolver_x_lower**(const gsl_root_fsolver *s)

double **gsl_root_fsolver_x_upper**(const gsl_root_fsolver *s)

호출시점에 풀이 공간 s 현재 탐색 구간의 하한, 상한 값을 반환합니다.

35.7 탐색 정지 인자들

근 탐색 과정은 다음 조건들 중 한가지가 충족되면 정지합니다.

- 사용자가 정의한 정확도를 만족하는 범주에서 근이 찾아진 경우.
- 사용자가 정의한 최대 반복 횟수에 도달한 경우.
- 오류가 발생한 경우.

이 조건들은 사용자가 제어해 볼 수 있습니다. 아래의 함수들은 사용자가 현재 결과를 여러 표준 방법들을 사용해, 정확도를 검증해 볼 수 있게 해줍니다.

int **gsl_root_test_interval**(double x_lower, double x_upper, double epsabs, double epsrel)

구간 [x_lower x_upper] 의 수렴을 절대 오차 $epsabs$ 와 상대 오차 $epsrel$ 를 이용해 검증합니다. 만약, 다음 조건을 만족하면, 수렴한다고 보고 $GSL_SUCCESS$ 를 반환합니다.

$$|a - b| < epsabs + epsrel \cdot \min(|a|, |b|)$$

이 조건은 구간 $x = [a, b]$ 가 원점을 포함하지 않을 때, 사용됩니다. 만약 구간이 원점을 포함한다면, $\min(|a|, |b|)$ 는 0으로 대체됩니다. (구간의 $\|x\|$ 값 중에서 0이 자동으로 최소값이 됩니다.) 이런 방법은, 원점에 가까운 근에 대해 상대 오차가 정확하게 추정할 수 있습니다.

구간에서 이 조건은, 구간에서 측정된 근 r 이 실제 근 r^* 에 대해 같은 조건을 만족시키는 것을 의미합니다.

$$|r - r^*| < epsabs + epsrel \cdot r^*$$

이때, 실제 근 r^* 이 구간에 포함되어 있어야 합니다.

int **gsl_root_test_delta**(double x1, double x0, double epsabs, double epsrel)

수열 $x0$ $x1$ 수렴을 절대 오차 $epsabs$ 와 상대 오차 $epsrel$ 를 이용해 검증합니다. 이 판정은 다음 조건을 충족하면, $GSL_SUCCESS$ 를 반환합니다.

$$|x_1 - x_0| < epsabs + epsrel \cdot |x_1|$$

다른 경우에는 $GSL_CONTINUE$ 를 반환합니다.

int **gsl_root_test_residual**(double f, double epsabs)

잔존값(residual value) f 에 대해 검증합니다. 다음 조건이 충족되면 함수는 $GSL_SUCCESS$ 를 반환합니다.

$$|f| < epsabs``$$

그리고 다른 경우에는 $GSL_CONTINUE$ 를 반환합니다. 이 판정은 $|f(x)|$ 가 충분히 작고, x 의 정확한 값이 중요하지 않은 상황에서 사용하는 것이 적절합니다.

35.8 괄호법 알고리즘

이 단락에서 설명할 괄호법을 이용한 근 탐색 알고리즘은 근이 포함되어 있음을 보장하는 초기 구간 설정이 필요합니다. a 와 b 가 구간의 양 끝 값이라면, $f(a)$ 의 부호는 반드시 $f(b)$ 와 달라야합니다. 이는 구간에서 함수 값이 0인 지점을 가로지르는 것을 보장합니다. 만약, 적절한 초기 구간이 제공되고 주어진 함수가 잘 작동한다면, 이 알고리즘들은 적절한 값을 항상 제공할 수 있습니다.

알아두어야 할 점은 짝수번 중첩된 다중근은 괄호법을 통해 찾을 수 없다는 점입니다. 이 경우 근이 x 축을 지나지 않고 접하기 때문입니다.

type **gsl_root_fsolver_type**

gsl_root_fsolver_type ***gsl_root_fsolver_bisection**

이분법 (Bisection) 알고리즘은 가장 간단한 괄호법 근 탐색 알고리즘입니다. 이 라이브러리에서 제공하는 함수중 가장 느린 선형 수렴 알고리즘이기도 합니다.

각각의 반복 단계에서, 주어진 구간이 이분되어 중간점에서의 함수 값이 계산됩니다. 이 값의 부호를 이용해 절반의 구간 중 어느 구간이 근을 포함하고 있지 않은지 판정합니다. 그러한 절반의 구간은 다음 단계에서 사라지고 근을 포함하는 새로운 작은 구간이 넘겨집니다. 이 과정은 간격이 충분히 작아질 때까지 무한정으로 반복될 수 있습니다.

각 단계에서, 근의 추정치는 현재 구간의 중간점으로 간주됩니다.

gsl_root_fsolver_type ***gsl_root_fsolver_falsepos**

할선법 (false postion algorithm)은 선형 보간법에 기반해 근을 탐색합니다. 똑같이 선형으로

수렴하지만, 이분법보다 빠릅니다.

각 반복 단계에서, 구간의 끝지점의 함수 좌표 $(a, f(a))$ 와 $(b, f(b))$ 를 잇는 선이 그려지고, 그 선이 x 축과 만나는 지점을 **중간점** 으로 설정됩니다. 이 지점에서의 함수 값을 계산하고, 그 부호를 이용해 구간의 양 방향중 어느 부분 구간이 근을 가지지 않는지 판정합니다. 그러한 절반의 구간은 다음 단계에서 사라지고 근을 포함하는 새로운 작은 구간이 넘겨집니다. 이 과정은 간격이 충분히 작아질 때까지 무한정으로 반복될 수 있습니다.

근의 최적 추정치는 현재 반복에 대한 구간의 선형 보간값으로 간주됩니다.

gsl_root_fsolver_type ***gsl_root_fsolver_brent**

브렌트-데커 방법 (Brent-Dekker algorithm)은 이분법에 보간법을 같이 적용한 방법입니다. 이 방법은 강력하면서도 빠른 알고리즘입니다. 이 책에서는 **브렌트 방법** 이라고 부를 것입니다.

각 반복 단계에서, 브렌트 방법은 보간 곡선을 사용해 함수를 근사합니다. 첫번째, 반복 단계에서 이는 2개의 끝 지점의 선형 보간입니다. 이후의 반복에서는 이전 단계의 3 지점을 역 2차 보간으로 근사해 더 놓은 정확도의 근사를 계산합니다. 보간 곡선과 x 축과의 교점은 그 단계에서의 근으로 추정되고 이 지점을 이용해 다음 단계에서 더 작은 구간을 생성합니다. 만약 이 교점이 구간을 벗어 났다면, 다시 1차 선형 보간으로 되돌아가 이를 반복합니다.

각 단계의 최적 추정치는 가장 최근 시행된 보간이나 이분법의 결과로 주어집니다.

35.9 기울기 연마 알고리즘

이 단락에서 설명할 기울기 연마 알고리즘은 근의 위치에 대한 초기 추정이 필요합니다. 이 방법은 절대적인 수렴 보장을 하지 않습니다. 주어지는 함수는 반드시 기울기 연마 방법으로 풀 수 있는 형태여야 하고, 초기 추정 값은 근에 충분히 가까워야 합니다. 이러한 조건이 갖추어졌을 때, 이 알고리즘은 2 차수의 수렴 속도를 가집니다.

이 알고리즘은 함수와 그 도함수를 필요로 합니다.

type **gsl_root_fdfsolver_type**

gsl_root_fdfsolver_type ***gsl_root_fdfsolver_newton**

이 방법은 **뉴턴 방법(Newton's Method)** 으로 불리는 방법으로 표준적인 기울기 연마 알고리즘입니다. 이 알고리즘은 근에 대한 초기 추정으로 부터 시작합니다. 각각의 반복 단계에서, 현재 지점에서의 함수 f 의 접선이 그려집니다. 이 접선이 x 축과 만나는 지점이 새로운 추정값이 됩니다. 이러한 반복 과정은 다음과 같은 수열로 표현될 수 있습니다.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

뉴턴 방법은 단일 근에 대해 2차수의 수렴 속도를 가지며, 복수의 근에 대해서는 선형적인 수렴 속도를 가집니다.

gsl_root_fdfsolver_type *gsl_root_fdfsolver_secant

할선 방법(secant method) 은 뉴턴 방법과 흡사한 방법으로, 한가지 다른 점은 모든 단계에서 지점의 도함수의 값을 계산할 필요가 없습니다.

먼저, 첫번째 단계에서 뉴턴 방법으로 시작해 초기 추정값에서 다음 추정치를 얻습니다. .. math:

```
x_{1} = x_0 - \frac{f(x_0)}{f'(x_0)}
```

이 다음 단계에서는 도함수의 값을 계산하지 않고 현재 지점과 그 이전 지점을 이용해 수치적으로 근사합니다.

$$f'_{est} = \frac{f(x_i) - f(x_{i-1})}{(x_i - x_{i-1})}$$

따라서, $i > 0$ 일 때, 다음과 같습니다.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'_{est}}$$

도함수가 근 근처에서 변화가 크지 않을 경우 이러한 할선법은 속도 측면에서 도함수 값을 계산해야 하는 방법에 비해 빠른 속도를 제공해줍니다. 점근적으로 도함수의 값을 구하는 시간이 원래 함수 값을 구하는 시간의 0.44배 이상일 때, 이 방법이 뉴턴 방법보다 빠릅니다. 다른 수치 미분 방법이 그러하듯이 지점의 차이가 너무 작아지면 추정치가 취소되는 오류가 발생하기도 합니다.

단일근에서 이 방법은 $(1+\sqrt{5})/2$ 의 수렴 속도를 가집니다. 이는 대략 1.62 정도입니다. 다중근에 대해서는 선형적으로 수렴합니다.

gsl_root_fdfsolver_type *gsl_root_fdfsolver_steffenson

스테퍼슨 방법(Steffenson Method)은 라이브러리의 함수 중 가장 빠른 수렴 속도를 제공합니다. 이 방법은 기본적인 뉴턴 알고리즘에 아티켄(Aitken)의 **델타-제곱(delta-suared)** 가속을 추가한 방법입니다. 뉴턴 방법의 x_i 지점에서 가속 과정으로 새로운 수열 R_i 을 생성합니다.

$$R_i = x_i - \frac{(x_{i+1} - x_i)^2}{(x_{i+2} - 2x_{i+1} + x_i)}$$

이 수열은 적절한 조건 하에서 본래 수열보다 빠르게 수렴합니다. 새로운 수열은 1개의 값을 생성하기 위해 3개의 항을 필요로 합니다. 따라서, 첫번째 반복 단계에서는 일반적인 뉴턴 방법의 값을 두번 째 이후의 반복에서 부터 가속된 값을 반환합니다. 만약, 가속 수열의 생성 과정에서 분모가 0 일 경우, 뉴턴 방법의 값을 이용하게 됩니다.

다른 가속 방법들과 마찬가지로 이 방법은 함수가 적절하지 않으면 불안정한 값을 내놓습니다.

35.10 예제

어떤 근 탐색 알고리즘을 사용하건, 풀기 위한 함수를 준비해야 합니다. 예시로 이미 일반적인 2차 다항함수를 살펴보았습니다. 먼저 함수 계수들을 정의하기 위해 demo_fn.h 헤더 파일이 필요합니다.

```
struct quadratic_params
  {
    double a, b, c;
  };

double quadratic (double x, void *params);
double quadratic_deriv (double x, void *params);
void quadratic_fdf (double x, void *params,
                    double *y, double *dy);
```

함수의 정의는 분리된 다른 소스파일 $demo_fn.c$ 에 있습니다.

```
double
quadratic (double x, void *params)
{
  struct quadratic_params *p
    = (struct quadratic_params *) params;

  double a = p->a;
  double b = p->b;
  double c = p->c;

  return (a * x + b) * x + c;
}

double
quadratic_deriv (double x, void *params)
{
  struct quadratic_params *p
    = (struct quadratic_params *) params;

  double a = p->a;
  double b = p->b;

  return 2.0 * a * x + b;
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
void
quadratic_fdf (double x, void *params,
               double *y, double *dy)
{
  struct quadratic_params *p
    = (struct quadratic_params *) params;

  double a = p->a;
  double b = p->b;
  double c = p->c;

  *y = (a * x + b) * x + c;
  *dy = 2.0 * a * x + b;
}
```

첫 번째 프로그램은 $gsl_root_fsolver_brent$ 를 사용해 브랜트 방법을 써서, 일반화된 2차 다항 함수를 이용해 다음과 같은 방정식을 풀 것입니다.

$$x^2 - 5 = 0$$

이 방정식의 해는 $x = \sqrt{5} = 2.236068$ 입니다.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_roots.h>

#include "demo_fn.h"
#include "demo_fn.c"

int
main (void)
{
  int status;
  int iter = 0, max_iter = 100;
  const gsl_root_fsolver_type *T;
  gsl_root_fsolver *s;
  double r = 0, r_expected = sqrt (5.0);
  double x_lo = 0.0, x_hi = 5.0;
  gsl_function F;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  struct quadratic_params params = {1.0, 0.0, -5.0};

  F.function = &quadratic;
  F.params = &params;

  T = gsl_root_fsolver_brent;
  s = gsl_root_fsolver_alloc (T);
  gsl_root_fsolver_set (s, &F, x_lo, x_hi);

  printf ("using %s method\n",
          gsl_root_fsolver_name (s));

  printf ("%5s [%9s, %9s] %9s %10s %9s\n",
          "iter", "lower", "upper", "root",
          "err", "err(est)");

  do
    {
      iter++;
      status = gsl_root_fsolver_iterate (s);
      r = gsl_root_fsolver_root (s);
      x_lo = gsl_root_fsolver_x_lower (s);
      x_hi = gsl_root_fsolver_x_upper (s);
      status = gsl_root_test_interval (x_lo, x_hi,
                                       0, 0.001);

      if (status == GSL_SUCCESS)
        printf ("Converged:\n");

      printf ("%5d [%.7f, %.7f] %.7f %+.7f %.7f\n",
              iter, x_lo, x_hi,
              r, r - r_expected,
              x_hi - x_lo);
    }
  while (status == GSL_CONTINUE && iter < max_iter);

  gsl_root_fsolver_free (s);

  return status;
}
```

각각의 반복 시점에서 결과는 다음과 같습니다.

```
$./a.out
using brent method
 iter [    lower,     upper]      root        err  err(est)
    1 [1.0000000, 5.0000000] 1.0000000 -1.2360680 4.0000000
    2 [1.0000000, 3.0000000] 3.0000000 +0.7639320 2.0000000
    3 [2.0000000, 3.0000000] 2.0000000 -0.2360680 1.0000000
    4 [2.2000000, 3.0000000] 2.2000000 -0.0360680 0.8000000
    5 [2.2000000, 2.2366300] 2.2366300 +0.0005621 0.0366300
Converged:
    6 [2.2360634, 2.2366300] 2.2360634 -0.0000046 0.0005666
```

만약, 프로그램을 수정해서 브랜트 방법이 아닌 이분법 방법을 사용하면 ($gsl_root_fsolver_brent$ 를 $gsl_root_fsolver_bisection$ 로 수정하면 됩니다), 이분법은 브랜트 방법보다 느리게 수렴하는 것을 관찰할 수 있습니다.

```
$./a.out
using bisection method
 iter [    lower,     upper]      root        err  err(est)
    1 [0.0000000, 2.5000000] 1.2500000 -0.9860680 2.5000000
    2 [1.2500000, 2.5000000] 1.8750000 -0.3610680 1.2500000
    3 [1.8750000, 2.5000000] 2.1875000 -0.0485680 0.6250000
    4 [2.1875000, 2.5000000] 2.3437500 +0.1076820 0.3125000
    5 [2.1875000, 2.3437500] 2.2656250 +0.0295570 0.1562500
    6 [2.1875000, 2.2656250] 2.2265625 -0.0095055 0.0781250
    7 [2.2265625, 2.2656250] 2.2460938 +0.0100258 0.0390625
    8 [2.2265625, 2.2460938] 2.2363281 +0.0002601 0.0195312
    9 [2.2265625, 2.2363281] 2.2314453 -0.0046227 0.0097656
   10 [2.2314453, 2.2363281] 2.2338867 -0.0021813 0.0048828
   11 [2.2338867, 2.2363281] 2.2351074 -0.0009606 0.0024414
Converged:
   12 [2.2351074, 2.2363281] 2.2357178 -0.0003502 0.0012207
```

이번 프로그램은 똑같은 함수를 풀지만, 대신에 도함수 알고리즘을 사용해 볼 것입니다.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_roots.h>

#include "demo_fn.h"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
#include "demo_fn.c"

int
main (void)
{
  int status;
  int iter = 0, max_iter = 100;
  const gsl_root_fdfsolver_type *T;
  gsl_root_fdfsolver *s;
  double x0, x = 5.0, r_expected = sqrt (5.0);
  gsl_function_fdf FDF;
  struct quadratic_params params = {1.0, 0.0, -5.0};

  FDF.f = &quadratic;
  FDF.df = &quadratic_deriv;
  FDF.fdf = &quadratic_fdf;
  FDF.params = &params;

  T = gsl_root_fdfsolver_newton;
  s = gsl_root_fdfsolver_alloc (T);
  gsl_root_fdfsolver_set (s, &FDF, x);

  printf ("using %s method\n",
          gsl_root_fdfsolver_name (s));

  printf ("%-5s %10s %10s %10s\n",
          "iter", "root", "err", "err(est)");
  do
    {
      iter++;
      status = gsl_root_fdfsolver_iterate (s);
      x0 = x;
      x = gsl_root_fdfsolver_root (s);
      status = gsl_root_test_delta (x, x0, 0, 1e-3);

      if (status == GSL_SUCCESS)
        printf ("Converged:\n");

      printf ("%5d %10.7f %+10.7f %10.7f\n",
              iter, x, x - r_expected, x - x0);
    }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  while (status == GSL_CONTINUE && iter < max_iter);

  gsl_root_fdfsolver_free (s);
  return status;
}
```

다음은 위 프로그램에서 사용한 뉴턴 방법의 결과입니다.

```
$./a.out
using newton method
iter        root        err   err(est)
    1  3.0000000 +0.7639320 -2.0000000
    2  2.3333333 +0.0972654 -0.6666667
    3  2.2380952 +0.0020273 -0.0952381
Converged:
    4  2.2360689 +0.0000009 -0.0020263
```

알아두면 좋은 점은, 현재 값과 이전 값과의 차이보다, 현재 값과 다음 값과의 차이를 이용해 오차를 더 정확하게 계산할 수 있다는 점입니다. 다른 도함수 풀이 방법은 $gsl_root_fdfsolver_newton$ 를 $gsl_root_fdfsolver_secant$ 나 $gsl_root_fdfsolver_steffenson$ 로 교체해서 사용할 수 있습니다.

35.11 참고 문헌과 추가 자료

브렌트-데커 알고리즘 (Brent-Dekker algorithm)에 대한 추가 정보를 얻고 싶다면, 다음을 참고할 수 있습니다.

- R. P. Brent, “An algorithm with guaranteed convergence for finding a zero of a function”, Computer Journal, 14 (1971) 422-425
- J. C. P. Bus and T. J. Dekker, “Two Efficient Algorithms with Guaranteed Convergence for Finding a Zero of a Function”, ACM Transactions of Mathematical Software, Vol.: 1 No.: 4 (1975) 330-345

제 36 장

함수의 최솟값 탐색

이 단원에서는 임의의 1변수 함수의 최소값을 찾는 기능들을 기술합니다. 라이브러리에서는 다양한 반복 풀이와 수렴 테스트를 위한 저수준의 기능들을 제공합니다. 이 기능들은 반복 단계를 완전히 제어할 수 있어, 잘 이용하면 사용자가 요구하는 적절한 해를 찾을 수 있습니다. 각각의 풀이 방법은 같은 작업 환경을 사용하기 때문에, 프로그램을 재컴파일 할 필요없이 여러 풀이 방법을 구동 중에 바꿀 수 있습니다. 각각의 풀이 인스턴스들은 스스로 상태를 추적하기 때문에, 다중 스레드 프로그램에서도 사용이 가능합니다.

이 단원에서 기술하는 함수들의 원형은 `gsl_min.h` 에 기술, 정의되어 있습니다.

이를 이용해서 함수의 최대값을 찾으려 하는 경우, 간단히 함수의 부호를 반대로 바꾸어 적용하면 됩니다.

36.1 개요

최소화 알고리즘은 최소값을 가지고 있다고 여겨지는 제한된 구간에서 시작합니다. 이 구간은 x 의 하한 값 a 와 상한 값 b 로 구성됩니다.

Fig 27.을 참고하십시오.

x 지점에서의 함수 값은 반드시 구간의 끝에서의 함수 값들보다 작아야 합니다.

$$f(a) > f(x) < f(b)$$

이 조건은 최소값이 주어진 구간 속 어딘가에 속해있다는 것을 보장해 줍니다. 각각의 반복 과정에서 새로운 지점 x' 가 알고리즘에 의해 선택됩니다. 만약 새로운 지점이 최소값을 더 잘 근사한다면, 예를 들어 $f(x') < f(x)$ 일 경우, 현재 근사 최소값 x 를 x' 으로 갱신합니다. 새로운 지점은 $f(a) > f(x) < f(b)$ 조건을 만족하는 가장 작은 점 집합을 선택해, 주어진 닫힌 구간의 크기를 줄일 수 있게 합니다. 각 단계를 거치면서 주어진 구간의 크기는 함수의 최소 값이 적절한 상태에 이루기까지 계속 줄어듭니다. 이 과정에서 최소 지점의 근사값과 오차 값을 평가해 반환합니다.

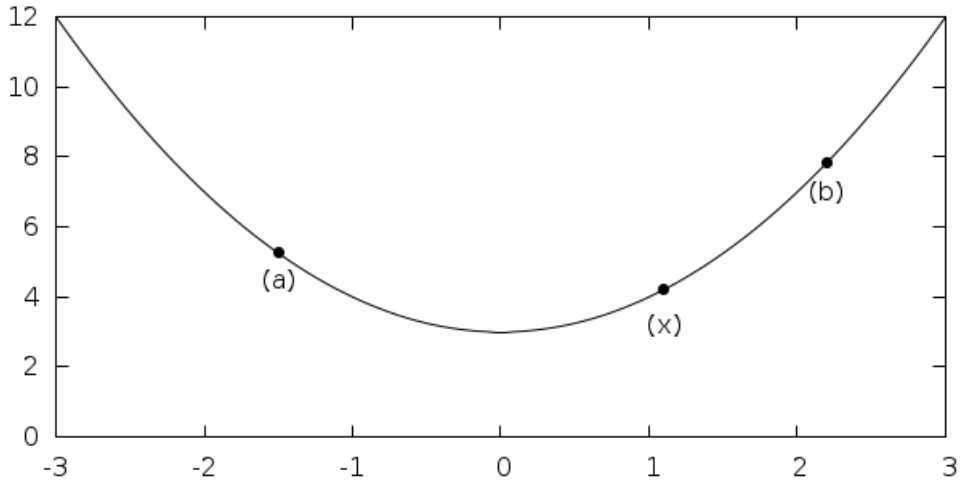


그림 36.1: 상하한 값이 주어진 닫힌 구간의 함수의 최소값 추정 장면.

이러한 최소화 과정 알고리즘은 괄호법(Bracket algorithm)에 속합니다. 라이브러리에서는 이러한 괄호법 알고리즘들을 동일한 작업공간에서 작동할 수 있게 구현했습니다. 사용자가 알고리즘을 사용할 수 있는 고수준의 프로그램을 작성하면, 라이브러리는 각각의 단계에 필요한 개별 함수들을 제공할 수 있습니다.

라이브러리에서 이러한 최소값 탐색 과정은 3가지로 나뉘어집니다.

- 알고리즘 T 대한, 최소화 공간 s 선언.
- T 를 수행해 s 갱신.
- s 가 수렴하는 지 판별, 필요하면 계속 반복.

최소화 공간의 상태는 $gsl_min_fminimizer$ 구조체로 주어집니다. 각 상태의 갱신은 도함수를 사용하지 않고 함수 값만을 사용해 이루어집니다.

36.2 주의 사항

유의해야 할 사항은 최소한 함수들이 한번에 한 개의 최소값을 찾는 다는 점입니다. 만약, 주어진 구간에 여러 최소값이 존재한다면 대부분의 경우 첫번째 최소값이 반환됩니다. 하지만 다양한 상황이 존재해 어떤 최소값이 반환될 지 예측하기는 어렵습니다. 대부분의 경우 여러 최소값이 존재하는 구간에서 근을 찾을 때, 오류가 반환되지는 않습니다.

제공하는 모든 최소화 알고리즘을 사용과정에서, 최소값의 위치를 자료형의 최대 정밀도로 파악하기는 어렵습니다. 주어진 구간의 최소값 x^* 에서 함수의 개형은 테일러 근사로 근사할 수 있습니다.

$$y = f(x^*) + \frac{1}{2} f''(x^*)(x - x^*)^2$$

여기서 두 번째 항은 자료형의 정밀도 한계로 인해 첫 번째 항에 더해질 때, 손실될 수 있습니다. 이때, x^* 에서 오차가 증폭되어, $\sqrt{\epsilon}$ 에 비례하게 됩니다. ϵ 은 부동 소수점의 상대 오차를 의미합니다. 고차항의 최소

지점을 가지는 함수들(예를 들어 x^4)은 이러한 오차 증폭 정도가 더 증가합니다. 가장 좋은 방법은 최솟값의 위치보다는 함수 값의 정밀도에 집중하는 것입니다.

36.3 최소화 설정하기

type **gsl_min_fminimizer**

최소화 함수를 위한 작업공간을 나타냅니다.

gsl_min_fminimizer *__gsl_min_fminimizer_alloc__(const gsl_min_fminimizer_type *T)

T 형태로 할당된 새 최소화 작업공간의 주소 포인터를 반환합니다. 예를 들어서, 다음 코드는 황금비 최소화 작업 공간을 할당해 반환합니다.

```
const gsl_min_fminimizer_type * T = gsl_min_fminimizer_goldensection;
gsl_min_fminimizer * s = gsl_min_fminimizer_alloc (T);
```

만약, 메모리 환경으로 인해 새로운 최소화 공간 할당이 불가능하다면, NULL 포인터를 반환하고, 오류 관리자가 호출되어 GSL_ENOMEM 코드를 전달합니다.

int __gsl_min_fminimizer_set__(gsl_min_fminimizer *s, gsl_function *f, double x_minimum, double x_lower, double x_upper)

최소화 공간 s 를 초기화 하거나 재설정합니다. 주어진 함수 f 초기 탐색 범위 [x_lower x_upper] 를 사용하도록 설정하며, 초기 최소 지점은 $x_minimum$ 로 추정합니다.

만약, 주어진 범위가 최소 지점을 포함하고 있지 않다면, 함수는 $GSLL_EINVAL$ 오류 값를 반환합니다.

int __gsl_min_fminimizer_set_with_values__(gsl_min_fminimizer *s, gsl_function *f, double x_minimum, double f_minimum, double x_lower, double f_lower, double x_upper, double f_upper)

$gsl_minnf minimizer_set()$ 함수와 같습니다. 하지만, $f_minimum$, f_lower , 그리고 f_upper 를 $f(x_minimum)$, $f(x_lower)$, 그리고 $f(x_upper)$ 대신 사용해 계산합니다.

void __gsl_min_fminimizer_free__(gsl_min_fminimizer *s)

최소화 공간 s 메모리를 해제합니다.

const char *__gsl_min_fminimizer_name__(const gsl_min_fminimizer *s)

최소화 공간의 최소화 방법 이름을 가리키는 포인터를 반환합니다. 예를 들어서,

```
printf("s is a '%s' minimizer\n", gsl_min_fminimizer_name(s));
```

은 $sisa'brent'minimizer$ 를 출력합니다.

36.4 최소화 함수

최소화를 시킬 함수는 반드시 1 변수 연속 함수여야 합니다. 일반적인 계수들을 지원하기 위해 $gsl_function$ 데이터 형을 이용합니다. (근을 찾을 함수 참고)

36.5 반복

다음 함수들은 주어진 알고리즘들을 반복 실행합니다. 각 함수는 최소화 공간의 상태를 주어진 최소화 알고리즘으로 갱신합니다. 모든 최소화 방법에 대해, 같은 함수를 사용가능합니다. 때문에, 별도의 코드 수정 없이 실행 도중에 다른 방법들을 선택할 수 있습니다.

int **gsl_min_fminimizer_iterate**(gsl_min_fminimizer *s)

최소화 공간 s 갱신합니다. 만약, 갱신 도중 예상치 못한 문제가 생긴다면 다음의 오류 값가 반환됩니다.

GSL_EBADFUNC

이 오류 값은 함수 값이 Inf 나 NaN 가 되는 특이점이 나올 경우 반환됩니다.

GSL_FAILURE

알고리즘이 현재 최적 근사 값이나 구간을 더 최적화 하지 못할 때 반환됩니다.

최소화 공간은 항상 현재의 최적 추정값을 유지하고 보유 구간은 항상 최소 지점을 포함하고 있습니다. 이 정보들은 다음의 보조 함수들로 접근할 수 있습니다.

double **gsl_min_fminimizer_x_minimum**(const gsl_min_fminimizer *s)

최소화 공간 s 의 현재 최적 추정치를 반환합니다.

double **gsl_min_fminimizer_x_upper**(const gsl_min_fminimizer *s)

double **gsl_min_fminimizer_x_lower**(const gsl_min_fminimizer *s)

최소화 공간 s 의 현재 구간 양 끝 값을 반환합니다.

double **gsl_min_fminimizer_f_minimum**(const gsl_min_fminimizer *s)

double **gsl_min_fminimizer_f_upper**(const gsl_min_fminimizer *s)

double **gsl_min_fminimizer_f_lower**(const gsl_min_fminimizer *s)

최소화 공간 s 의 현재 최적 추정값, 구간 양 끝 값에서의 함수 값들을 반환합니다.

36.6 탐색 정지 인자들

최소화 과정은 다음의 조건들 중 하나를 만족하면 멈추게 됩니다.

- 최소값이 사용자가 정의한 정밀도에 맞게 찾아진 경우,
- 사용자 정의 최대 반복 횟수를 넘어선 경우.
- 오류가 생긴 경우.

이러한 조건들은 사용자가 직접 설정할 수 있습니다. 아래의 함수는 현재 결과의 정밀도를 측정할 수 있게 해 줍니다.

int **gsl_min_test_interval**(double x_lower, double x_upper, double epsabs, double epsrel)

이 함수는 구간 [x_lower x_upper 의 수렴을 절대 오차 $epsabs$ 상대 오차 $epsrel$ 판별합니다. 판정 결과는 다음의 조건을 만족했을 때, GSL_SUCESS 반환합니다.

구간 $x = [a, b]$ 가 원점을 포함하지 않을 때, |a-b| < epsabs + epsrel min(|a|,|b|) $

만약 구간이 원점을 포함하고 있다면, $\min(|a|, |b|)$ 는 0으로 바뀝니다. 이런 과정을 통해 원점에 가까운 최소값에 대해 상대오차를 정확하게 추정할 수 있습니다.

추정 최소값 x_m 과 실제 최소값 x^*_m 에 대해서도 같은 조건을 쓸 수 있습니다. 구간에 x^*_m 이 존재한다는 가정하에 다음을 볼 수 있습니다.

$$|x_m - x^*_m| < epsabs + epsrel \cdot x^*_m \$$

36.7 최소화 알고리즘

이 단락에서 기술할 최소화 알고리즘은 최소값이 포함되어있음을 보장하는 초기 추정 구간이 주어져야합니다. 만약, a, b 가 구간의 끝 값들이고 x 가 최소값 지점을 나타내면, $f(a) > f(x) < f(b)$ 를 만족해야 합니다. 이 조건은 함수가 주어진 구간에서 적어도 1개의 최소값을 가지고 있음을 보장합니다. 만약, 적절한 시작 구간과 함수가 주어진다면, 알고리즘은 제대로 된 값을 반환합니다.

type **gsl_min_fminimizer_type**

gsl_min_fminimizer_type ***gsl_min_fminimizer_goldensect**

황금비 알고리즘(Golden section algorithm) 은 가장 간단한 괄호법을 이용하는 최소화 알고리즘입니다. 이는 선형 수렴하며 라이브러리에서 제공하는 알고리즘 중 가장 느린 알고리즘입니다.

각각의 반복 단계에서, 알고리즘은 먼저 끝 지점에서 현재 구해진 최소 지점으로 이루어진 부분 구간을 비교합니다. 더 큰 부분 구간은 황금비($(3 - \sqrt{5})/2 \approx 0.3819660$)로 분할합니다. 그리고 이렇게 만들어진 새로운 지점의 함수값을 계산합니다. 주어진 최소점 조건 :math:` f(a) > f(x) < f(b)` 을 이용해서, 나머지 지점을 제외하고 최소 지점을 포함하는 새로운 구간을 찾습니다. 이

과정은 구간이 충분히 작아질 때까지 계속 반복됩니다. 황금비로 구간을 이분하는 것은 이러한 알고리즘에서 가장 빠르게 수렴한다고 알려져 있습니다.

gsl_min_fminimizer_type ***gsl_min_fminimizer_brent**

브렌트 최소 알고리즘(Brent minimization algorithm) 은 포물선 보간법을 황금비 탐색 알고리즘과 함께 이용합니다. 이 방법은 충분히 빠르면서도 좋은 결과를 내놓습니다.

알고리즘의 진행과정은 다음과 같이 요약될 수 있습니다. 각각의 반복 단계에서 브렌트 알고리즘은 포물선과 주어진 구간의 세 지점을 이용해 함수를 근사합니다. 포물선의 최소 지점이 최소값으로 추정이 되고, 만약 이 값이 현재 구간에 포함된다면, 보간 지점이 받아들여지고 이를 이용해 더 작은 구간을 만들게 됩니다. 만약 보간 지점이 받아들여지지 않을 경우 기존의 황금비 방법을 사용해 새 구간을 형성합니다. 브렌트 방법은 수렴성을 높이기 위한 추가 검토 절차를 포함합니다.

gsl_min_fminimizer_type ***gsl_min_fminimizer_quad_golden**

이 알고리즘은 브랜트 알고리즘에 길과 머레이(Gill and Murray)가 만든 단계 길이 보호 알고리즘을 더한 방법입니다.

36.8 예제

다음 코드는 브랜트 알고리즘을 이용해서 함수 $f(x) = \cos(x) + 1$ 의 최소값을 찾는 프로그램입니다. 해당 함수의 최소값은 $x = \pi$ 지점입니다. 시작 구간은 $(0, 6)$, 초기 추정 최소값은 2 입니다.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_min.h>

double fn1 (double x, void * params)
{
  (void)(params); /* avoid unused parameter warning */
  return cos(x) + 1.0;
}

int
main (void)
{
  int status;
  int iter = 0, max_iter = 100;
  const gsl_min_fminimizer_type *T;
  gsl_min_fminimizer *s;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
double m = 2.0, m_expected = M_PI;
double a = 0.0, b = 6.0;
gsl_function F;

F.function = &fn1;
F.params = 0;

T = gsl_min_fminimizer_brent;
s = gsl_min_fminimizer_alloc (T);
gsl_min_fminimizer_set (s, &F, m, a, b);

printf ("using %s method\n",
        gsl_min_fminimizer_name (s));

printf ("%5s [%9s, %9s] %9s %10s %9s\n",
        "iter", "lower", "upper", "min",
        "err", "err(est)");

printf ("%5d [%.7f, %.7f] %.7f %+.7f %.7f\n",
        iter, a, b,
        m, m - m_expected, b - a);

do
  {
    iter++;
    status = gsl_min_fminimizer_iterate (s);

    m = gsl_min_fminimizer_x_minimum (s);
    a = gsl_min_fminimizer_x_lower (s);
    b = gsl_min_fminimizer_x_upper (s);

    status
      = gsl_min_test_interval (a, b, 0.001, 0.0);

    if (status == GSL_SUCCESS)
      printf ("Converged:\n");

    printf ("%5d [%.7f, %.7f] "
            "%.7f %+.7f %.7f\n",
            iter, a, b,
            m, m - m_expected, b - a);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    }
  while (status == GSL_CONTINUE && iter < max_iter);

  gsl_min_fminimizer_free (s);

  return status;
}
```

다음은 작성한 최소화 프로그램의 결과입니다.

```
using brent method
 iter [    lower,     upper]       min        err  err(est)
    0 [0.0000000, 6.0000000] 2.0000000 -1.1415927 6.0000000
    1 [2.0000000, 6.0000000] 3.5278640 +0.3862713 4.0000000
    2 [2.0000000, 3.5278640] 3.1748217 +0.0332290 1.5278640
    3 [2.0000000, 3.1748217] 3.1264576 -0.0151351 1.1748217
    4 [3.1264576, 3.1748217] 3.1414743 -0.0001183 0.0483641
    5 [3.1414743, 3.1748217] 3.1415930 +0.0000004 0.0333474
Converged:
    6 [3.1414743, 3.1415930] 3.1415927 +0.0000000 0.0001187
```

36.9 참고 문헌과 추가 자료

브렌트 알고리즘 (Brent Algorithm)에 대한 추가 정보를 얻고 싶다면, 음을 참고할 수 있습니다.

- Richard Brent, Algorithms for minimization without derivatives, Prentice-Hall (1973), republished by Dover in paperback (2002), ISBN 0-486-41998-3.

제 37 장

다변수 함수의 근 탐색

참고: 번역중

이 단원은 다변수 함수의 근을 찾는 함수들에 대해 기술합니다. 이는 n 개의 변수에 대해, n 개의 식을 가지는 비선형 계의 풀이를 의미합니다. 라이브러리 내에서 다양한 반복 기법을 통한 풀이 방법과 수렴 검증기능들을 제공합니다. 라이브러리에서 제공하는 반복 기능들의 중간 과정이 모두 직접 제어할 수 있기 때문에 다양한 풀이 방법과 수렴 검증 기능들을 적절한 해를 얻기 위해 조합해서 사용할 수 있습니다. 각 방법들은 모두 같은 프레임 워크를 사용해, 재 컴파일 과정 없이 구동 과정에서 다른 풀이 방법들을 교체해서 사용할 수 있습니다. 각 풀이 기능들은 구동 과정에서 풀이 상태를 지속적으로 추적하기 때문에 멀티 스레드 프로그램에서도 사용할 수 있습니다. 이 기능들은 포트란 라이브러리 MINPACK 에 기반해 있습니다.

`gsl_multiroots.h` 에 함수들의 초안과 필요한 기능들이 정의되어 있습니다.

37.1 개요

다변수 함수의 근 탐색 문제는 n 개의 변수 x_i 를 가지는 n 개의 f_i 방정식의 해를 필요로합니다.

$$f_i(x_1, \ldots, x_n) = 0 \qquad \text{for } i = 1 \ldots n.$$

일반적으로 이러한 n 차원 계를 위한 괄호법 알고리즘은 존재하지 않습니다. 또, 해가 존재하는 지를 판별할 수 있는 일반적인 방법도 없습니다. 라이브러리에서 제공하는 모든 알고리즘은 뉴턴 방법을 확장해 사용하고 초기 추정값에 의존합니다.

$$x \to x' = x - J^{-1} f(x)$$

x 와 f 는 벡터 값이고 J 는 야코비 행렬로 $J_{ij} = \partial f_i / \partial x_j$ 로 정의됩니다.

뉴턴 방법의 각 반복 과정에서 $|f|$ 노름을 감소시키 거나, $|f|$ 의 음수 그래디언트 값 방향으로 최대 경사 하강법을 사용하는 등의 방법을 사용할 수도 있습니다. 이 방법들은 뉴턴 방법보다 더 확장된 수렴 반경을 가집니다.

근 탐색 알고리즘들은 단일 프래임워크로 제공됩니다. 사용자는 알고리즘에 대한 고수준의 기능들을 작성해야하고, 라이브러리는 각 반복 과정에 필요한 개별 함수들을 제공해줍니다.

각 반복 과정은

- T 알고리즘을 위한 풀이 상태 s 초기화.
- T 알고리즘을 이용해 s 상태 갱신.
- s 의 수렴성 검사, 필요시 재 반복

야코비 행렬의 계산은 문제가 발생할 수 있습니다. 미분 값 계산이 불가능할 수도 있고 n^2 개의 항을 계산해야하는 행렬의 계산이 너무 많은 비용을 소모할 수도 있기 때문입니다. 이러한 이유로 라이브러리에서 제공하는 알고리즘은 미분 값을 제공하는 경우와 아닌 경우 2가지로 분리되어 있습니다.

해석적인 야코비 행렬을 이용한 풀이는 gsl_multiroot_fdfsolver 구조체를 이용해 풀이 상태를 정의합니다. 상태의 갱신은 사용자가 제공하는 함수와 그 미분값을 모두 계산해 이루어집니다.

해석적인 야코비 행렬을 이용하지 않는 풀이는 gsl_multiroot_fsolver 구조체를 이용합니다. 상태의 갱신 과정은 함수 값만을 이용합니다. 이 풀이는 행렬 J 나 J^{-1} 를 근사 방법을 이용해 계산합니다.

37.2 Initializing the Solver

The following functions initialize a multidimensional solver, either with or without derivatives. The solver itself depends only on the dimension of the problem and the algorithm and can be reused for different problems.

type **gsl_multiroot_fsolver**

This is a workspace for multidimensional root-finding without derivatives.

type **gsl_multiroot_fdfsolver**

This is a workspace for multidimensional root-finding with derivatives.

gsl_multiroot_fsolver ***gsl_multiroot_fsolver_alloc**(const gsl_multiroot_fsolver_type *T,
size_t n)

This function returns a pointer to a newly allocated instance of a solver of type T for a system of n dimensions. For example, the following code creates an instance of a hybrid solver, to solve a 3-dimensional system of equations:

```
const gsl_multiroot_fsolver_type * T = gsl_multiroot_fsolver_hybrid;
gsl_multiroot_fsolver * s = gsl_multiroot_fsolver_alloc (T, 3);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

gsl_multiroot_fdfsolver *__gsl_multiroot_fdfsolver_alloc__(const gsl_multiroot_fdfsolver_type *T, size_t n)

This function returns a pointer to a newly allocated instance of a derivative solver of type `T` for a system of `n` dimensions. For example, the following code creates an instance of a Newton-Raphson solver, for a 2-dimensional system of equations:

```
const gsl_multiroot_fdfsolver_type * T = gsl_multiroot_fdfsolver_newton;
gsl_multiroot_fdfsolver * s = gsl_multiroot_fdfsolver_alloc (T, 2);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

int __gsl_multiroot_fsolver_set__(gsl_multiroot_fsolver *s, gsl_multiroot_function *f, const gsl_vector *x)

int __gsl_multiroot_fdfsolver_set__(gsl_multiroot_fdfsolver *s, gsl_multiroot_function_fdf *fdf, const gsl_vector *x)

These functions set, or reset, an existing solver `s` to use the function `f` or function and derivative `fdf`, and the initial guess `x`. Note that the initial position is copied from `x`, this argument is not modified by subsequent iterations.

void __gsl_multiroot_fsolver_free__(gsl_multiroot_fsolver *s)

void __gsl_multiroot_fdfsolver_free__(gsl_multiroot_fdfsolver *s)

These functions free all the memory associated with the solver `s`.

const char *__gsl_multiroot_fsolver_name__(const gsl_multiroot_fsolver *s)

const char *__gsl_multiroot_fdfsolver_name__(const gsl_multiroot_fdfsolver *s)

These functions return a pointer to the name of the solver. For example:

```
printf ("s is a '%s' solver\n", gsl_multiroot_fdfsolver_name (s));
```

would print something like `s is a 'newton' solver`.

37.3 Providing the function to solve

You must provide n functions of n variables for the root finders to operate on. In order to allow for general parameters the functions are defined by the following data types:

type **gsl_multiroot_function**

This data type defines a general system of functions with parameters.

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`

this function should store the vector result $f(x, params)$ in `f` for argument `x` and parameters `params`, returning an appropriate error code if the function cannot be computed.

`size_t n`

the dimension of the system, i.e. the number of components of the vectors `x` and `f`.

`void * params`

a pointer to the parameters of the function.

Here is an example using Powell's test function,

$$f_1(x) = Ax_0x_1 - 1$$
$$f_2(x) = \exp(-x_0) + \exp(-x_1) - (1 + 1/A)$$

with $A = 10^4$. The following code defines a `gsl_multiroot_function` system `F` which you could pass to a solver:

```
struct powell_params { double A; };

int
powell (gsl_vector * x, void * p, gsl_vector * f) {
   struct powell_params * params
     = (struct powell_params *)p;
   const double A = (params->A);
   const double x0 = gsl_vector_get(x,0);
   const double x1 = gsl_vector_get(x,1);

   gsl_vector_set (f, 0, A * x0 * x1 - 1);
   gsl_vector_set (f, 1, (exp(-x0) + exp(-x1)
                          - (1.0 + 1.0/A)));
   return GSL_SUCCESS
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
}

gsl_multiroot_function F;
struct powell_params params = { 10000.0 };

F.f = &powell;
F.n = 2;
F.params = &params;
```

type **gsl_multiroot_function_fdf**

This data type defines a general system of functions with parameters and the corresponding Jacobian matrix of derivatives,

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`

this function should store the vector result $f(x, params)$ in `f` for argument `x` and parameters `params`, returning an appropriate error code if the function cannot be computed.

`int (* df) (const gsl_vector * x, void * params, gsl_matrix * J)`

this function should store the `n`-by-`n` matrix result

$$J_{ij} = \partial f_i(x, \text{params})/\partial x_j$$

in `J` for argument `x` and parameters `params`, returning an appropriate error code if the function cannot be computed.

`int (* fdf) (const gsl_vector * x, void * params, gsl_vector * f, gsl_matrix * J)`

This function should set the values of the `f` and `J` as above, for arguments `x` and parameters `params`. This function provides an optimization of the separate functions for $f(x)$ and $J(x)$—it is always faster to compute the function and its derivative at the same time.

`size_t n`

the dimension of the system, i.e. the number of components of the vectors `x` and `f`.

`void * params`

a pointer to the parameters of the function.

The example of Powell's test function defined above can be extended to include analytic derivatives using the following code:

```
int
powell_df (gsl_vector * x, void * p, gsl_matrix * J)
{
   struct powell_params * params
     = (struct powell_params *)p;
   const double A = (params->A);
   const double x0 = gsl_vector_get(x,0);
   const double x1 = gsl_vector_get(x,1);
   gsl_matrix_set (J, 0, 0, A * x1);
   gsl_matrix_set (J, 0, 1, A * x0);
   gsl_matrix_set (J, 1, 0, -exp(-x0));
   gsl_matrix_set (J, 1, 1, -exp(-x1));
   return GSL_SUCCESS
}

int
powell_fdf (gsl_vector * x, void * p,
            gsl_matrix * f, gsl_matrix * J) {
   struct powell_params * params
     = (struct powell_params *)p;
   const double A = (params->A);
   const double x0 = gsl_vector_get(x,0);
   const double x1 = gsl_vector_get(x,1);

   const double u0 = exp(-x0);
   const double u1 = exp(-x1);

   gsl_vector_set (f, 0, A * x0 * x1 - 1);
   gsl_vector_set (f, 1, u0 + u1 - (1 + 1/A));

   gsl_matrix_set (J, 0, 0, A * x1);
   gsl_matrix_set (J, 0, 1, A * x0);
   gsl_matrix_set (J, 1, 0, -u0);
   gsl_matrix_set (J, 1, 1, -u1);
   return GSL_SUCCESS
}

gsl_multiroot_function_fdf FDF;

FDF.f = &powell_f;
FDF.df = &powell_df;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
FDF.fdf = &powell_fdf;
FDF.n = 2;
FDF.params = 0;
```

Note that the function `powell_fdf` is able to reuse existing terms from the function when calculating the Jacobian, thus saving time.

37.4 Iteration

The following functions drive the iteration of each algorithm. Each function performs one iteration to update the state of any solver of the corresponding type. The same functions work for all solvers so that different methods can be substituted at runtime without modifications to the code.

int **gsl_multiroot_fsolver_iterate**(gsl_multiroot_fsolver *s)

int **gsl_multiroot_fdfsolver_iterate**(gsl_multiroot_fdfsolver *s)

These functions perform a single iteration of the solver `s`. If the iteration encounters an unexpected problem then an error code will be returned,

`GSL_EBADFUNC`

the iteration encountered a singular point where the function or its derivative evaluated to `Inf` or `NaN`.

`GSL_ENOPROG`

the iteration is not making any progress, preventing the algorithm from continuing.

The solver maintains a current best estimate of the root `s->x` and its function value `s->f` at all times. This information can be accessed with the following auxiliary functions,

gsl_vector *__gsl_multiroot_fsolver_root__(const gsl_multiroot_fsolver *s)

gsl_vector *__gsl_multiroot_fdfsolver_root__(const gsl_multiroot_fdfsolver *s)

These functions return the current estimate of the root for the solver `s`, given by `s->x`.

gsl_vector *__gsl_multiroot_fsolver_f__(const gsl_multiroot_fsolver *s)

gsl_vector *__gsl_multiroot_fdfsolver_f__(const gsl_multiroot_fdfsolver *s)

These functions return the function value $f(x)$ at the current estimate of the root for the solver `s`, given by `s->f`.

gsl_vector *__gsl_multiroot_fsolver_dx__(const gsl_multiroot_fsolver *s)

gsl_vector *gsl_multiroot_fdfsolver_dx(const gsl_multiroot_fdfsolver *s)

These functions return the last step dx taken by the solver `s`, given by `s->dx`.

37.5 Search Stopping Parameters

A root finding procedure should stop when one of the following conditions is true:

- A multidimensional root has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the precision of the current result in several standard ways.

int gsl_multiroot_test_delta(const gsl_vector *dx, const gsl_vector *x, double epsabs, double epsrel)

This function tests for the convergence of the sequence by comparing the last step `dx` with the absolute error `epsabs` and relative error `epsrel` to the current position `x`. The test returns `GSL_SUCCESS` if the following condition is achieved,

$$|dx_i| < \text{epsabs} + \text{epsrel}\,|x_i|$$

for each component of `x` and returns `GSL_CONTINUE` otherwise.

int gsl_multiroot_test_residual(const gsl_vector *f, double epsabs)

This function tests the residual value `f` against the absolute error bound `epsabs`. The test returns `GSL_SUCCESS` if the following condition is achieved,

$$\sum_i |f_i| < \text{epsabs}$$

and returns `GSL_CONTINUE` otherwise. This criterion is suitable for situations where the precise location of the root, x, is unimportant provided a value can be found where the residual is small enough.

37.6 Algorithms using Derivatives

The root finding algorithms described in this section make use of both the function and its derivative. They require an initial guess for the location of the root, but there is no absolute guarantee of convergence—the function must be suitable for this technique and the initial guess must be sufficiently close to the root for it to work. When the conditions are satisfied then convergence is quadratic.

type **gsl_multiroot_fdfsolver_type**

The following are available algorithms for minimizing functions using derivatives.

gsl_multiroot_fdfsolver_type ***gsl_multiroot_fdfsolver_hybridsj**

This is a modified version of Powell's Hybrid method as implemented in the HYBRJ algorithm in MINPACK. Minpack was written by Jorge J. Moré, Burton S. Garbow and Kenneth E. Hillstrom. The Hybrid algorithm retains the fast convergence of Newton's method but will also reduce the residual when Newton's method is unreliable.

The algorithm uses a generalized trust region to keep each step under control. In order to be accepted a proposed new position x' must satisfy the condition $|D(x' - x)| < \delta$, where D is a diagonal scaling matrix and δ is the size of the trust region. The components of D are computed internally, using the column norms of the Jacobian to estimate the sensitivity of the residual to each component of x. This improves the behavior of the algorithm for badly scaled functions.

On each iteration the algorithm first determines the standard Newton step by solving the system $Jdx = -f$. If this step falls inside the trust region it is used as a trial step in the next stage. If not, the algorithm uses the linear combination of the Newton and gradient directions which is predicted to minimize the norm of the function while staying inside the trust region,

$$dx = -\alpha J^{-1} f(x) - \beta \nabla |f(x)|^2$$

This combination of Newton and gradient directions is referred to as a dogleg step.

The proposed step is now tested by evaluating the function at the resulting point, x'. If the step reduces the norm of the function sufficiently then it is accepted and size of the trust region is increased. If the proposed step fails to improve the solution then the size of the trust region is decreased and another trial step is computed.

The speed of the algorithm is increased by computing the changes to the Jacobian approximately, using a rank-1 update. If two successive attempts fail to reduce

the residual then the full Jacobian is recomputed. The algorithm also monitors the progress of the solution and returns an error if several steps fail to make any improvement,

GSL_ENOPROG

the iteration is not making any progress, preventing the algorithm from continuing.

GSL_ENOPROGJ

re-evaluations of the Jacobian indicate that the iteration is not making any progress, preventing the algorithm from continuing.

gsl_multiroot_fdfsolver_type ***gsl_multiroot_fdfsolver_hybridj**

This algorithm is an unscaled version of HYBRIDSJ. The steps are controlled by a spherical trust region $|x' - x| < \delta$, instead of a generalized region. This can be useful if the generalized region estimated by HYBRIDSJ is inappropriate.

gsl_multiroot_fdfsolver_type ***gsl_multiroot_fdfsolver_newton**

Newton's Method is the standard root-polishing algorithm. The algorithm begins with an initial guess for the location of the solution. On each iteration a linear approximation to the function F is used to estimate the step which will zero all the components of the residual. The iteration is defined by the following sequence,

$$x \to x' = x - J^{-1} f(x)$$

where the Jacobian matrix J is computed from the derivative functions provided by `f`. The step dx is obtained by solving the linear system,

$$J dx = -f(x)$$

using LU decomposition. If the Jacobian matrix is singular, an error code of GSL_EDOM is returned.

gsl_multiroot_fdfsolver_type ***gsl_multiroot_fdfsolver_gnewton**

This is a modified version of Newton's method which attempts to improve global convergence by requiring every step to reduce the Euclidean norm of the residual, $|f(x)|$. If the Newton step leads to an increase in the norm then a reduced step of relative size,

$$t = (\sqrt{1 + 6r} - 1)/(3r)$$

is proposed, with r being the ratio of norms $|f(x')|^2/|f(x)|^2$. This procedure is

repeated until a suitable step size is found.

37.7 Algorithms without Derivatives

The algorithms described in this section do not require any derivative information to be supplied by the user. Any derivatives needed are approximated by finite differences. Note that if the finite-differencing step size chosen by these routines is inappropriate, an explicit user-supplied numerical derivative can always be used with the algorithms described in the previous section.

type **gsl_multiroot_fsolver_type**

The following are available algorithms for minimizing functions without derivatives.

gsl_multiroot_fsolver_type ***gsl_multiroot_fsolver_hybrids**

This is a version of the Hybrid algorithm which replaces calls to the Jacobian function by its finite difference approximation. The finite difference approximation is computed using `gsl_multiroots_fdjac()` with a relative step size of `GSL_SQRT_DBL_EPSILON`. Note that this step size will not be suitable for all problems.

gsl_multiroot_fsolver_type ***gsl_multiroot_fsolver_hybrid**

This is a finite difference version of the Hybrid algorithm without internal scaling.

gsl_multiroot_fsolver_type ***gsl_multiroot_fsolver_dnewton**

The discrete Newton algorithm is the simplest method of solving a multidimensional system. It uses the Newton iteration

$$x \to x - J^{-1} f(x)$$

where the Jacobian matrix J is approximated by taking finite differences of the function `f`. The approximation scheme used by this implementation is,

$$J_{ij} = (f_i(x + \delta_j) - f_i(x))/\delta_j$$

where δ_j is a step of size $\sqrt{\epsilon}|x_j|$ with ϵ being the machine precision ($\epsilon \approx 2.22 \times 10^{-16}$). The order of convergence of Newton's algorithm is quadratic, but the finite differences require n^2 function evaluations on each iteration. The algorithm may become unstable if the finite differences are not a good approximation to the true derivatives.

gsl_multiroot_fsolver_type ***gsl_multiroot_fsolver_broyden**

The Broyden algorithm is a version of the discrete Newton algorithm which attempts

to avoids the expensive update of the Jacobian matrix on each iteration. The changes to the Jacobian are also approximated, using a rank-1 update,

$$J^{-1} \to J^{-1} - (J^{-1}df - dx)dx^T J^{-1}/dx^T J^{-1}df$$

where the vectors dx and df are the changes in x and f. On the first iteration the inverse Jacobian is estimated using finite differences, as in the discrete Newton algorithm.

This approximation gives a fast update but is unreliable if the changes are not small, and the estimate of the inverse Jacobian becomes worse as time passes. The algorithm has a tendency to become unstable unless it starts close to the root. The Jacobian is refreshed if this instability is detected (consult the source for details).

This algorithm is included only for demonstration purposes, and is not recommended for serious use.

37.8 Examples

The multidimensional solvers are used in a similar way to the one-dimensional root finding algorithms. This first example demonstrates the HYBRIDS scaled-hybrid algorithm, which does not require derivatives. The program solves the Rosenbrock system of equations,

$$f_1(x, y) = a(1 - x)$$
$$f_2(x, y) = b(y - x^2)$$

with $a = 1, b = 10$. The solution of this system lies at $(x, y) = (1, 1)$ in a narrow valley.

The first stage of the program is to define the system of equations:

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_multiroots.h>

struct rparams
  {
    double a;
    double b;
  };
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
int
rosenbrock_f (const gsl_vector * x, void *params,
              gsl_vector * f)
{
  double a = ((struct rparams *) params)->a;
  double b = ((struct rparams *) params)->b;

  const double x0 = gsl_vector_get (x, 0);
  const double x1 = gsl_vector_get (x, 1);

  const double y0 = a * (1 - x0);
  const double y1 = b * (x1 - x0 * x0);

  gsl_vector_set (f, 0, y0);
  gsl_vector_set (f, 1, y1);

  return GSL_SUCCESS;
}
```

The main program begins by creating the function object `f`, with the arguments `(x, y)` and parameters `(a,b)`. The solver `s` is initialized to use this function, with the `gsl_multiroot_fsolver_hybrids` method:

```
int
main (void)
{
  const gsl_multiroot_fsolver_type *T;
  gsl_multiroot_fsolver *s;

  int status;
  size_t i, iter = 0;

  const size_t n = 2;
  struct rparams p = {1.0, 10.0};
  gsl_multiroot_function f = {&rosenbrock_f, n, &p};

  double x_init[2] = {-10.0, -5.0};
  gsl_vector *x = gsl_vector_alloc (n);

  gsl_vector_set (x, 0, x_init[0]);
  gsl_vector_set (x, 1, x_init[1]);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  T = gsl_multiroot_fsolver_hybrids;
  s = gsl_multiroot_fsolver_alloc (T, 2);
  gsl_multiroot_fsolver_set (s, &f, x);

  print_state (iter, s);

  do
    {
      iter++;
      status = gsl_multiroot_fsolver_iterate (s);

      print_state (iter, s);

      if (status)   /* check if solver is stuck */
        break;

      status =
        gsl_multiroot_test_residual (s->f, 1e-7);
    }
  while (status == GSL_CONTINUE && iter < 1000);

  printf ("status = %s\n", gsl_strerror (status));

  gsl_multiroot_fsolver_free (s);
  gsl_vector_free (x);
  return 0;
}
```

Note that it is important to check the return status of each solver step, in case the algorithm becomes stuck. If an error condition is detected, indicating that the algorithm cannot proceed, then the error can be reported to the user, a new starting point chosen or a different algorithm used.

The intermediate state of the solution is displayed by the following function. The solver state contains the vector `s->x` which is the current position, and the vector `s->f` with corresponding function values:

```
int
print_state (size_t iter, gsl_multiroot_fsolver * s)
{
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  printf ("iter = %3u x = % .3f % .3f "
          "f(x) = % .3e % .3e\n",
          iter,
          gsl_vector_get (s->x, 0),
          gsl_vector_get (s->x, 1),
          gsl_vector_get (s->f, 0),
          gsl_vector_get (s->f, 1));
}
```

Here are the results of running the program. The algorithm is started at $(-10, -5)$ far from the solution. Since the solution is hidden in a narrow valley the earliest steps follow the gradient of the function downhill, in an attempt to reduce the large value of the residual. Once the root has been approximately located, on iteration 8, the Newton behavior takes over and convergence is very rapid:

```
iter =  0 x = -10.000  -5.000  f(x) = 1.100e+01 -1.050e+03
iter =  1 x = -10.000  -5.000  f(x) = 1.100e+01 -1.050e+03
iter =  2 x =  -3.976  24.827  f(x) = 4.976e+00  9.020e+01
iter =  3 x =  -3.976  24.827  f(x) = 4.976e+00  9.020e+01
iter =  4 x =  -3.976  24.827  f(x) = 4.976e+00  9.020e+01
iter =  5 x =  -1.274  -5.680  f(x) = 2.274e+00 -7.302e+01
iter =  6 x =  -1.274  -5.680  f(x) = 2.274e+00 -7.302e+01
iter =  7 x =   0.249   0.298  f(x) = 7.511e-01  2.359e+00
iter =  8 x =   0.249   0.298  f(x) = 7.511e-01  2.359e+00
iter =  9 x =   1.000   0.878  f(x) = 1.268e-10 -1.218e+00
iter = 10 x =   1.000   0.989  f(x) = 1.124e-11 -1.080e-01
iter = 11 x =   1.000   1.000  f(x) = 0.000e+00  0.000e+00
status = success
```

Note that the algorithm does not update the location on every iteration. Some iterations are used to adjust the trust-region parameter, after trying a step which was found to be divergent, or to recompute the Jacobian, when poor convergence behavior is detected.

The next example program adds derivative information, in order to accelerate the solution. There are two derivative functions `rosenbrock_df` and `rosenbrock_fdf`. The latter computes both the function and its derivative simultaneously. This allows the optimization of any common terms. For simplicity we substitute calls to the separate `f` and `df` functions at this point in the code below:

```
int
rosenbrock_df (const gsl_vector * x, void *params,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                gsl_matrix * J)
{
  const double a = ((struct rparams *) params)->a;
  const double b = ((struct rparams *) params)->b;

  const double x0 = gsl_vector_get (x, 0);

  const double df00 = -a;
  const double df01 = 0;
  const double df10 = -2 * b  * x0;
  const double df11 = b;

  gsl_matrix_set (J, 0, 0, df00);
  gsl_matrix_set (J, 0, 1, df01);
  gsl_matrix_set (J, 1, 0, df10);
  gsl_matrix_set (J, 1, 1, df11);

  return GSL_SUCCESS;
}

int
rosenbrock_fdf (const gsl_vector * x, void *params,
                gsl_vector * f, gsl_matrix * J)
{
  rosenbrock_f (x, params, f);
  rosenbrock_df (x, params, J);

  return GSL_SUCCESS;
}
```

The main program now makes calls to the corresponding `fdfsolver` versions of the functions:

```
int
main (void)
{
  const gsl_multiroot_fdfsolver_type *T;
  gsl_multiroot_fdfsolver *s;

  int status;
  size_t i, iter = 0;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  const size_t n = 2;
  struct rparams p = {1.0, 10.0};
  gsl_multiroot_function_fdf f = {&rosenbrock_f,
                                  &rosenbrock_df,
                                  &rosenbrock_fdf,
                                  n, &p};

  double x_init[2] = {-10.0, -5.0};
  gsl_vector *x = gsl_vector_alloc (n);

  gsl_vector_set (x, 0, x_init[0]);
  gsl_vector_set (x, 1, x_init[1]);

  T = gsl_multiroot_fdfsolver_gnewton;
  s = gsl_multiroot_fdfsolver_alloc (T, n);
  gsl_multiroot_fdfsolver_set (s, &f, x);

  print_state (iter, s);

  do
    {
      iter++;

      status = gsl_multiroot_fdfsolver_iterate (s);

      print_state (iter, s);

      if (status)
        break;

      status = gsl_multiroot_test_residual (s->f, 1e-7);
    }
  while (status == GSL_CONTINUE && iter < 1000);

  printf ("status = %s\n", gsl_strerror (status));

  gsl_multiroot_fdfsolver_free (s);
  gsl_vector_free (x);
  return 0;
}
```

The addition of derivative information to the `gsl_multiroot_fsolver_hybrids` solver does not make any significant difference to its behavior, since it able to approximate the Jacobian numerically with sufficient accuracy. To illustrate the behavior of a different derivative solver we switch to `gsl_multiroot_fdfsolver_gnewton`. This is a traditional Newton solver with the constraint that it scales back its step if the full step would lead "uphill". Here is the output for the `gsl_multiroot_fdfsolver_gnewton` algorithm:

```
iter = 0 x = -10.000  -5.000 f(x) =  1.100e+01 -1.050e+03
iter = 1 x =  -4.231 -65.317 f(x) =  5.231e+00 -8.321e+02
iter = 2 x =   1.000 -26.358 f(x) = -8.882e-16 -2.736e+02
iter = 3 x =   1.000   1.000 f(x) = -2.220e-16 -4.441e-15
status = success
```

The convergence is much more rapid, but takes a wide excursion out to the point $(-4.23, -65.3)$. This could cause the algorithm to go astray in a realistic application. The hybrid algorithm follows the downhill path to the solution more reliably.

37.9 References and Further Reading

The original version of the Hybrid method is described in the following articles by Powell,

- M.J.D. Powell, "A Hybrid Method for Nonlinear Equations" (Chap 6, p 87–114) and "A Fortran Subroutine for Solving systems of Nonlinear Algebraic Equations" (Chap 7, p 115–161), in Numerical Methods for Nonlinear Algebraic Equations, P. Rabinowitz, editor. Gordon and Breach, 1970.

The following papers are also relevant to the algorithms described in this section,

- J.J. Moré, M.Y. Cosnard, "Numerical Solution of Nonlinear Equations", ACM Transactions on Mathematical Software, Vol 5, No 1, (1979), p 64–85
- C.G. Broyden, "A Class of Methods for Solving Nonlinear Simultaneous Equations", Mathematics of Computation, Vol 19 (1965), p 577–593
- J.J. Moré, B.S. Garbow, K.E. Hillstrom, "Testing Unconstrained Optimization Software", ACM Transactions on Mathematical Software, Vol 7, No 1 (1981), p 17–41

제 38 장

다변수 함수의 최솟값 탐색

참고: 번역중

This chapter describes routines for finding minima of arbitrary multidimensional functions. The library provides low level components for a variety of iterative minimizers and convergence tests. These can be combined by the user to achieve the desired solution, while providing full access to the intermediate steps of the algorithms. Each class of methods uses the same framework, so that you can switch between minimizers at runtime without needing to recompile your program. Each instance of a minimizer keeps track of its own state, allowing the minimizers to be used in multi-threaded programs. The minimization algorithms can be used to maximize a function by inverting its sign.

The header file `gsl_multimin.h` contains prototypes for the minimization functions and related declarations.

38.1 Overview

The problem of multidimensional minimization requires finding a point x such that the scalar function,

$$f(x_1, \ldots, x_n)$$

takes a value which is lower than at any neighboring point. For smooth functions the gradient $g = \nabla f$ vanishes at the minimum. In general there are no bracketing methods available for the minimization of n-dimensional functions. The algorithms proceed from an initial guess

using a search algorithm which attempts to move in a downhill direction.

Algorithms making use of the gradient of the function perform a one-dimensional line minimisation along this direction until the lowest point is found to a suitable tolerance. The search direction is then updated with local information from the function and its derivatives, and the whole process repeated until the true n-dimensional minimum is found.

Algorithms which do not require the gradient of the function use different strategies. For example, the Nelder-Mead Simplex algorithm maintains $n + 1$ trial parameter vectors as the vertices of a n-dimensional simplex. On each iteration it tries to improve the worst vertex of the simplex by geometrical transformations. The iterations are continued until the overall size of the simplex has decreased sufficiently.

Both types of algorithms use a standard framework. The user provides a high-level driver for the algorithms, and the library provides the individual functions necessary for each of the steps. There are three main phases of the iteration. The steps are,

- initialize minimizer state, `s`, for algorithm `T`
- update `s` using the iteration `T`
- test `s` for convergence, and repeat iteration if necessary

Each iteration step consists either of an improvement to the line-minimisation in the current direction or an update to the search direction itself. The state for the minimizers is held in a `gsl_multimin_fdfminimizer` struct or a `gsl_multimin_fminimizer` struct.

38.2 Caveats

Note that the minimization algorithms can only search for one local minimum at a time. When there are several local minima in the search area, the first minimum to be found will be returned; however it is difficult to predict which of the minima this will be. In most cases, no error will be reported if you try to find a local minimum in an area where there is more than one.

It is also important to note that the minimization algorithms find local minima; there is no way to determine whether a minimum is a global minimum of the function in question.

38.3 Initializing the Multidimensional Minimizer

The following function initializes a multidimensional minimizer. The minimizer itself depends only on the dimension of the problem and the algorithm and can be reused for different problems.

type **gsl_multimin_fdfminimizer**

This is a workspace for minimizing functions using derivatives.

type **gsl_multimin_fminimizer**

This is a workspace for minimizing functions without derivatives.

gsl_multimin_fdfminimizer ***gsl_multimin_fdfminimizer_alloc**(const gsl_multimin_fdfminimizer_type *T, size_t n)

gsl_multimin_fminimizer ***gsl_multimin_fminimizer_alloc**(const gsl_multimin_fminimizer_type *T, size_t n)

This function returns a pointer to a newly allocated instance of a minimizer of type `T` for an `n`-dimension function. If there is insufficient memory to create the minimizer then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

int **gsl_multimin_fdfminimizer_set**(gsl_multimin_fdfminimizer *s, gsl_multimin_function_fdf *fdf, const gsl_vector *x, double step_size, double tol)

int **gsl_multimin_fminimizer_set**(gsl_multimin_fminimizer *s, gsl_multimin_function *f, const gsl_vector *x, const gsl_vector *step_size)

The function `gsl_multimin_fdfminimizer_set()` initializes the minimizer `s` to minimize the function `fdf` starting from the initial point `x`. The size of the first trial step is given by `step_size`. The accuracy of the line minimization is specified by `tol`. The precise meaning of this parameter depends on the method used. Typically the line minimization is considered successful if the gradient of the function g is orthogonal to the current search direction p to a relative accuracy of `tol`, where $p \cdot g < tol|p||g|$. A `tol` value of 0.1 is suitable for most purposes, since line minimization only needs to be carried out approximately. Note that setting `tol` to zero will force the use of "exact" line-searches, which are extremely expensive.

The function `gsl_multimin_fminimizer_set()` initializes the minimizer `s` to minimize the function `f`, starting from the initial point `x`. The size of the initial trial steps is given in vector `step_size`. The precise meaning of this parameter depends on the method used.

void **gsl_multimin_fdfminimizer_free**(gsl_multimin_fdfminimizer *s)

void **gsl_multimin_fminimizer_free**(gsl_multimin_fminimizer *s)

This function frees all the memory associated with the minimizer `s`.

const char ***gsl_multimin_fdfminimizer_name**(const gsl_multimin_fdfminimizer *s)

const char ***gsl_multimin_fminimizer_name**(const gsl_multimin_fminimizer *s)

This function returns a pointer to the name of the minimizer. For example:

```
printf ("s is a '%s' minimizer\n", gsl_multimin_fdfminimizer_name (s));
```

would print something like `s is a 'conjugate_pr' minimizer`.

38.4 Providing a function to minimize

You must provide a parametric function of n variables for the minimizers to operate on. You may also need to provide a routine which calculates the gradient of the function and a third routine which calculates both the function value and the gradient together. In order to allow for general parameters the functions are defined by the following data types:

type **gsl_multimin_function_fdf**

This data type defines a general function of n variables with parameters and the corresponding gradient vector of derivatives,

`double (* f) (const gsl_vector * x, void * params)`

this function should return the result $f(x, params)$ for argument `x` and parameters `params`. If the function cannot be computed, an error value of `GSL_NAN` should be returned.

`void (* df) (const gsl_vector * x, void * params, gsl_vector * g)`

this function should store the `n`-dimensional gradient

$$g_i = \partial f(x, \text{params})/\partial x_i$$

in the vector `g` for argument `x` and parameters `params`, returning an appropriate error code if the function cannot be computed.

`void (* fdf) (const gsl_vector * x, void * params, double * f, gsl_vector * g)`

This function should set the values of the `f` and `g` as above, for arguments `x` and parameters `params`. This function provides an optimization of the separate

functions for $f(x)$ and $g(x)$—it is always faster to compute the function and its derivative at the same time.

`size_t n`

the dimension of the system, i.e. the number of components of the vectors `x`.

`void * params`

a pointer to the parameters of the function.

type **gsl_multimin_function**

This data type defines a general function of n variables with parameters,

`double (* f) (const gsl_vector * x, void * params)`

this function should return the result $f(x, params)$ for argument `x` and parameters `params`. If the function cannot be computed, an error value of `GSL_NAN` should be returned.

`size_t n`

the dimension of the system, i.e. the number of components of the vectors `x`.

`void * params`

a pointer to the parameters of the function.

The following example function defines a simple two-dimensional paraboloid with five parameters,

```
/* Paraboloid centered on (p[0],p[1]), with
   scale factors (p[2],p[3]) and minimum p[4] */

double
my_f (const gsl_vector *v, void *params)
{
  double x, y;
  double *p = (double *)params;

  x = gsl_vector_get(v, 0);
  y = gsl_vector_get(v, 1);

  return p[2] * (x - p[0]) * (x - p[0]) +
           p[3] * (y - p[1]) * (y - p[1]) + p[4];
}

/* The gradient of f, df = (df/dx, df/dy). */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
void
my_df (const gsl_vector *v, void *params,
       gsl_vector *df)
{
  double x, y;
  double *p = (double *)params;

  x = gsl_vector_get(v, 0);
  y = gsl_vector_get(v, 1);

  gsl_vector_set(df, 0, 2.0 * p[2] * (x - p[0]));
  gsl_vector_set(df, 1, 2.0 * p[3] * (y - p[1]));
}

/* Compute both f and df together. */
void
my_fdf (const gsl_vector *x, void *params,
        double *f, gsl_vector *df)
{
  *f = my_f(x, params);
  my_df(x, params, df);
}
```

The function can be initialized using the following code:

```
gsl_multimin_function_fdf my_func;

/* Paraboloid center at (1,2), scale factors (10, 20),
   minimum value 30 */
double p[5] = { 1.0, 2.0, 10.0, 20.0, 30.0 };

my_func.n = 2;  /* number of function components */
my_func.f = &my_f;
my_func.df = &my_df;
my_func.fdf = &my_fdf;
my_func.params = (void *)p;
```

38.5 Iteration

The following function drives the iteration of each algorithm. The function performs one iteration to update the state of the minimizer. The same function works for all minimizers so that different methods can be substituted at runtime without modifications to the code.

int **gsl_multimin_fdfminimizer_iterate**(gsl_multimin_fdfminimizer *s)

int **gsl_multimin_fminimizer_iterate**(gsl_multimin_fminimizer *s)

These functions perform a single iteration of the minimizer `s`. If the iteration encounters an unexpected problem then an error code will be returned. The error code `GSL_ENOPROG` signifies that the minimizer is unable to improve on its current estimate, either due to numerical difficulty or because a genuine local minimum has been reached.

The minimizer maintains a current best estimate of the minimum at all times. This information can be accessed with the following auxiliary functions,

gsl_vector ***gsl_multimin_fdfminimizer_x**(const gsl_multimin_fdfminimizer *s)

gsl_vector ***gsl_multimin_fminimizer_x**(const gsl_multimin_fminimizer *s)

double **gsl_multimin_fdfminimizer_minimum**(const gsl_multimin_fdfminimizer *s)

double **gsl_multimin_fminimizer_minimum**(const gsl_multimin_fminimizer *s)

gsl_vector ***gsl_multimin_fdfminimizer_gradient**(const gsl_multimin_fdfminimizer *s)

gsl_vector ***gsl_multimin_fdfminimizer_dx**(const gsl_multimin_fdfminimizer *s)

double **gsl_multimin_fminimizer_size**(const gsl_multimin_fminimizer *s)

These functions return the current best estimate of the location of the minimum, the value of the function at that point, its gradient, the last step increment of the estimate, and minimizer specific characteristic size for the minimizer `s`.

int **gsl_multimin_fdfminimizer_restart**(gsl_multimin_fdfminimizer *s)

This function resets the minimizer `s` to use the current point as a new starting point.

38.6 Stopping Criteria

A minimization procedure should stop when one of the following conditions is true:

- A minimum has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the precision of the current result.

int **gsl_multimin_test_gradient**(const gsl_vector *g, double epsabs)

This function tests the norm of the gradient `g` against the absolute tolerance `epsabs`. The gradient of a multidimensional function goes to zero at a minimum. The test returns `GSL_SUCCESS` if the following condition is achieved,

$$|g| < \text{epsabs}$$

and returns `GSL_CONTINUE` otherwise. A suitable choice of `epsabs` can be made from the desired accuracy in the function for small variations in x. The relationship between these quantities is given by $\delta f = g\,\delta x$.

int **gsl_multimin_test_size**(const double size, double epsabs)

This function tests the minimizer specific characteristic size (if applicable to the used minimizer) against absolute tolerance `epsabs`. The test returns `GSL_SUCCESS` if the size is smaller than tolerance, otherwise `GSL_CONTINUE` is returned.

38.7 Algorithms with Derivatives

There are several minimization methods available. The best choice of algorithm depends on the problem. The algorithms described in this section use the value of the function and its gradient at each evaluation point.

type **gsl_multimin_fdfminimizer_type**

This type specifies a minimization algorithm using gradients.

gsl_multimin_fdfminimizer_type ***gsl_multimin_fdfminimizer_conjugate_fr**

This is the Fletcher-Reeves conjugate gradient algorithm. The conjugate gradient algorithm proceeds as a succession of line minimizations. The sequence of search directions is used to build up an approximation to the curvature of the function in the neighborhood of the minimum.

An initial search direction `p` is chosen using the gradient, and line minimization is carried out in that direction. The accuracy of the line minimization is specified by the parameter `tol`. The minimum along this line occurs when the function gradient `g` and the search direction `p` are orthogonal. The line minimization terminates when $p \cdot g < tol|p||g|$. The search direction is updated using the Fletcher-Reeves formula $p' = g' - \beta p$ where $\beta = -|g'|^2/|g|^2$, and the line minimization is then repeated for the new search direction.

gsl_multimin_fdfminimizer_type ***gsl_multimin_fdfminimizer_conjugate_pr**

This is the Polak-Ribiere conjugate gradient algorithm. It is similar to the Fletcher-

Reeves method, differing only in the choice of the coefficient β. Both methods work well when the evaluation point is close enough to the minimum of the objective function that it is well approximated by a quadratic hypersurface.

gsl_multimin_fdfminimizer_type ***gsl_multimin_fdfminimizer_vector_bfgs2**

gsl_multimin_fdfminimizer_type ***gsl_multimin_fdfminimizer_vector_bfgs**

These methods use the vector Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. This is a quasi-Newton method which builds up an approximation to the second derivatives of the function f using the difference between successive gradient vectors. By combining the first and second derivatives the algorithm is able to take Newton-type steps towards the function minimum, assuming quadratic behavior in that region.

The `bfgs2` version of this minimizer is the most efficient version available, and is a faithful implementation of the line minimization scheme described in Fletcher's Practical Methods of Optimization, Algorithms 2.6.2 and 2.6.4. It supersedes the original `bfgs` routine and requires substantially fewer function and gradient evaluations. The user-supplied tolerance `tol` corresponds to the parameter σ used by Fletcher. A value of 0.1 is recommended for typical use (larger values correspond to less accurate line searches).

gsl_multimin_fdfminimizer_type ***gsl_multimin_fdfminimizer_steepest_descent**

The steepest descent algorithm follows the downhill gradient of the function at each step. When a downhill step is successful the step-size is increased by a factor of two. If the downhill step leads to a higher function value then the algorithm backtracks and the step size is decreased using the parameter `tol`. A suitable value of `tol` for most applications is 0.1. The steepest descent method is inefficient and is included only for demonstration purposes.

38.8 Algorithms without Derivatives

The algorithms described in this section use only the value of the function at each evaluation point.

type **gsl_multimin_fminimizer_type**

This type specifies minimization algorithms which do not use gradients.

gsl_multimin_fminimizer_type ***gsl_multimin_fminimizer_nmsimplex2**

gsl_multimin_fminimizer_type ***gsl_multimin_fminimizer_nmsimplex**

These methods use the Simplex algorithm of Nelder and Mead. Starting from the

initial vector $x = p_0$, the algorithm constructs an additional n vectors p_i using the step size vector $s = step\_size$ as follows:

$$
\begin{aligned}
p_0 &= (x_0, x_1, \cdots, x_n) \\
p_1 &= (x_0 + s_0, x_1, \cdots, x_n) \\
p_2 &= (x_0, x_1 + s_1, \cdots, x_n) \\
\ldots &= \ldots \\
p_n &= (x_0, x_1, \cdots, x_n + s_n)
\end{aligned}
$$

These vectors form the $n + 1$ vertices of a simplex in n dimensions. On each iteration the algorithm uses simple geometrical transformations to update the vector corresponding to the highest function value. The geometric transformations are reflection, reflection followed by expansion, contraction and multiple contraction. Using these transformations the simplex moves through the space towards the minimum, where it contracts itself.

After each iteration, the best vertex is returned. Note, that due to the nature of the algorithm not every step improves the current best parameter vector. Usually several iterations are required.

The minimizer-specific characteristic size is calculated as the average distance from the geometrical center of the simplex to all its vertices. This size can be used as a stopping criteria, as the simplex contracts itself near the minimum. The size is returned by the function `gsl_multimin_fminimizer_size()`.

The `gsl_multimin_fminimizer_nmsimplex2` version of this minimiser is a new $O(N)$ operations implementation of the earlier $O(N^2)$ operations `gsl_multimin_fminimizer_nmsimplex` minimiser. It uses the same underlying algorithm, but the simplex updates are computed more efficiently for high-dimensional problems. In addition, the size of simplex is calculated as the RMS distance of each vertex from the center rather than the mean distance, allowing a linear update of this quantity on each step. The memory usage is $O(N^2)$ for both algorithms.

gsl_multimin_fminimizer_type ***gsl_multimin_fminimizer_nmsimplex2rand**

This method is a variant of `gsl_multimin_fminimizer_nmsimplex2` which initialises the simplex around the starting point `x` using a randomly-oriented set of basis vectors instead of the fixed coordinate axes. The final dimensions of the simplex are scaled along the coordinate axes by the vector `step_size`. The randomisation uses a simple deterministic generator so that repeated calls to `gsl_multimin_fminimizer_set()` for a given solver object will vary the orientation in a well-defined way.

38.9 Examples

This example program finds the minimum of the paraboloid function defined earlier. The location of the minimum is offset from the origin in x and y, and the function value at the minimum is non-zero. The main program is given below, it requires the example function given earlier in this chapter.

```
int
main (void)
{
  size_t iter = 0;
  int status;

  const gsl_multimin_fdfminimizer_type *T;
  gsl_multimin_fdfminimizer *s;

  /* Position of the minimum (1,2), scale factors
     10,20, height 30. */
  double par[5] = { 1.0, 2.0, 10.0, 20.0, 30.0 };

  gsl_vector *x;
  gsl_multimin_function_fdf my_func;

  my_func.n = 2;
  my_func.f = my_f;
  my_func.df = my_df;
  my_func.fdf = my_fdf;
  my_func.params = par;

  /* Starting point, x = (5,7) */
  x = gsl_vector_alloc (2);
  gsl_vector_set (x, 0, 5.0);
  gsl_vector_set (x, 1, 7.0);

  T = gsl_multimin_fdfminimizer_conjugate_fr;
  s = gsl_multimin_fdfminimizer_alloc (T, 2);

  gsl_multimin_fdfminimizer_set (s, &my_func, x, 0.01, 1e-4);

  do
    {
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      iter++;
      status = gsl_multimin_fdfminimizer_iterate (s);

      if (status)
        break;

      status = gsl_multimin_test_gradient (s->gradient, 1e-3);

      if (status == GSL_SUCCESS)
        printf ("Minimum found at:\n");

      printf ("%5d %.5f %.5f %10.5f\n", iter,
              gsl_vector_get (s->x, 0),
              gsl_vector_get (s->x, 1),
              s->f);

    }
  while (status == GSL_CONTINUE && iter < 100);

  gsl_multimin_fdfminimizer_free (s);
  gsl_vector_free (x);

  return 0;
}
```

The initial step-size is chosen as 0.01, a conservative estimate in this case, and the line minimization parameter is set at 0.0001. The program terminates when the norm of the gradient has been reduced below 0.001. The output of the program is shown below,

```
        x       y         f
    1 4.99629 6.99072  687.84780
    2 4.98886 6.97215  683.55456
    3 4.97400 6.93501  675.01278
    4 4.94429 6.86073  658.10798
    5 4.88487 6.71217  625.01340
    6 4.76602 6.41506  561.68440
    7 4.52833 5.82083  446.46694
    8 4.05295 4.63238  261.79422
    9 3.10219 2.25548   75.49762
   10 2.85185 1.62963   67.03704
   11 2.19088 1.76182   45.31640
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   12 0.86892 2.02622   30.18555
Minimum found at:
   13 1.00000 2.00000   30.00000
```

Note that the algorithm gradually increases the step size as it successfully moves downhill, as can be seen by plotting the successive points in 그림 38.1.

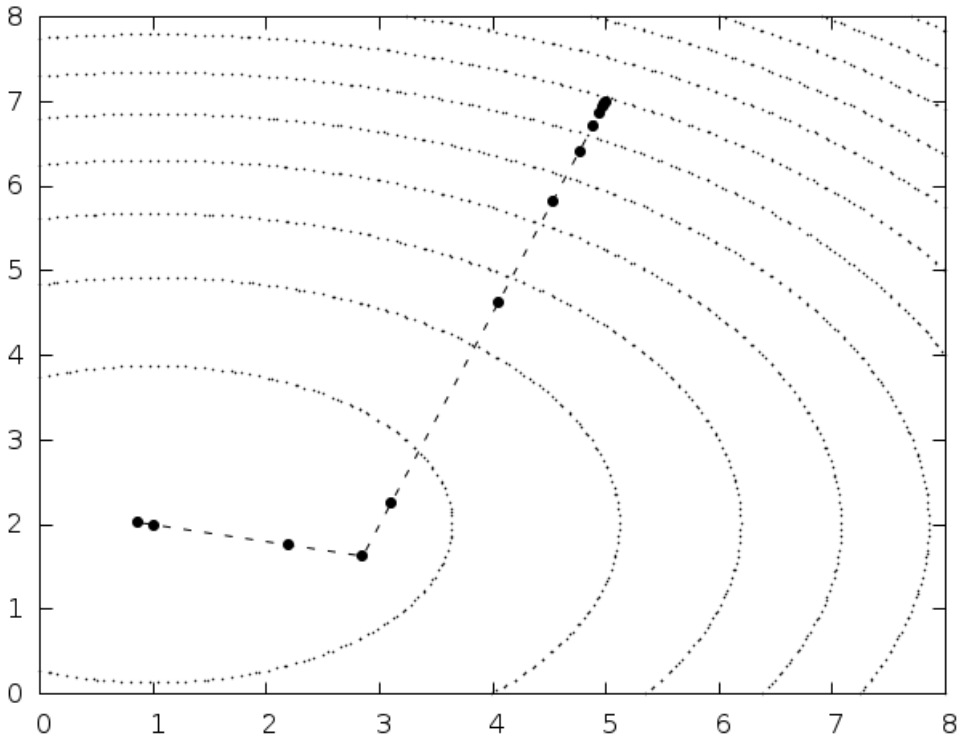


그림 38.1: Function contours with path taken by minimization algorithm

The conjugate gradient algorithm finds the minimum on its second direction because the function is purely quadratic. Additional iterations would be needed for a more complicated function.

Here is another example using the Nelder-Mead Simplex algorithm to minimize the same example object function, as above.

```
int
main(void)
{
  double par[5] = {1.0, 2.0, 10.0, 20.0, 30.0};

  const gsl_multimin_fminimizer_type *T =
    gsl_multimin_fminimizer_nmsimplex2;
  gsl_multimin_fminimizer *s = NULL;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
gsl_vector *ss, *x;
gsl_multimin_function minex_func;

size_t iter = 0;
int status;
double size;

/* Starting point */
x = gsl_vector_alloc (2);
gsl_vector_set (x, 0, 5.0);
gsl_vector_set (x, 1, 7.0);

/* Set initial step sizes to 1 */
ss = gsl_vector_alloc (2);
gsl_vector_set_all (ss, 1.0);

/* Initialize method and iterate */
minex_func.n = 2;
minex_func.f = my_f;
minex_func.params = par;

s = gsl_multimin_fminimizer_alloc (T, 2);
gsl_multimin_fminimizer_set (s, &minex_func, x, ss);

do
  {
    iter++;
    status = gsl_multimin_fminimizer_iterate(s);

    if (status)
      break;

    size = gsl_multimin_fminimizer_size (s);
    status = gsl_multimin_test_size (size, 1e-2);

    if (status == GSL_SUCCESS)
      {
        printf ("converged to minimum at\n");
      }

    printf ("%5d %10.3e %10.3e f() = %7.3f size = %.3f\n",
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
              iter,
              gsl_vector_get (s->x, 0),
              gsl_vector_get (s->x, 1),
              s->fval, size);
    }
  while (status == GSL_CONTINUE && iter < 100);

  gsl_vector_free(x);
  gsl_vector_free(ss);
  gsl_multimin_fminimizer_free (s);

  return status;
}
```

The minimum search stops when the Simplex size drops to 0.01. The output is shown below.

```
    1  6.500e+00  5.000e+00 f() = 512.500 size = 1.130
    2  5.250e+00  4.000e+00 f() = 290.625 size = 1.409
    3  5.250e+00  4.000e+00 f() = 290.625 size = 1.409
    4  5.500e+00  1.000e+00 f() = 252.500 size = 1.409
    5  2.625e+00  3.500e+00 f() = 101.406 size = 1.847
    6  2.625e+00  3.500e+00 f() = 101.406 size = 1.847
    7  0.000e+00  3.000e+00 f() =  60.000 size = 1.847
    8  2.094e+00  1.875e+00 f() =  42.275 size = 1.321
    9  2.578e-01  1.906e+00 f() =  35.684 size = 1.069
   10  5.879e-01  2.445e+00 f() =  35.664 size = 0.841
   11  1.258e+00  2.025e+00 f() =  30.680 size = 0.476
   12  1.258e+00  2.025e+00 f() =  30.680 size = 0.367
   13  1.093e+00  1.849e+00 f() =  30.539 size = 0.300
   14  8.830e-01  2.004e+00 f() =  30.137 size = 0.172
   15  8.830e-01  2.004e+00 f() =  30.137 size = 0.126
   16  9.582e-01  2.060e+00 f() =  30.090 size = 0.106
   17  1.022e+00  2.004e+00 f() =  30.005 size = 0.063
   18  1.022e+00  2.004e+00 f() =  30.005 size = 0.043
   19  1.022e+00  2.004e+00 f() =  30.005 size = 0.043
   20  1.022e+00  2.004e+00 f() =  30.005 size = 0.027
   21  1.022e+00  2.004e+00 f() =  30.005 size = 0.022
   22  9.920e-01  1.997e+00 f() =  30.001 size = 0.016
   23  9.920e-01  1.997e+00 f() =  30.001 size = 0.013
converged to minimum at
   24  9.920e-01  1.997e+00 f() =  30.001 size = 0.008
```

The simplex size first increases, while the simplex moves towards the minimum. After a while the size begins to decrease as the simplex contracts around the minimum.

38.10 References and Further Reading

The conjugate gradient and BFGS methods are described in detail in the following book,

- R. Fletcher, Practical Methods of Optimization (Second Edition) Wiley (1987), ISBN 0471915475.

A brief description of multidimensional minimization algorithms and more recent references can be found in,

- C.W. Ueberhuber, Numerical Computation (Volume 2), Chapter 14, Section 4.4 “Minimization Methods”, p.: 325–335, Springer (1997), ISBN 3-540-62057-5.

The simplex algorithm is described in the following paper,

- J.A. Nelder and R. Mead, A simplex method for function minimization, Computer Journal vol.: 7 (1965), 308–313.

제 39 장

선형 최소 제곱법

참고: 번역중

이 단원에서는 선형 최소 제곱법을 이용해 실험 값을 여러 함수들의 조합으로 표현하는 기능을 서술합니다. 각 계산은 알려진 오차를 포함하거나 포함하지 않을 수도 있습니다. 가중치가 있는 값들에 대해, 각 함수들은 최적의 계수를 계산하고 그들에 관한 공분산 행렬를 찾습니다. 가중치가 없는 값들은 각 지점의 분포를 이용해 공분산 행렬을 찾고, 분산-공분산 행렬을 제공합니다.

이 단원의 함수들은 여러 형태로 나누어져 있습니다. 단순한 1개, 2개의 계수를 받는 분석 함수와 여러 계수를 사용 가능한 분석 함수로 나누어져 있습니다.

39.1 개요

최소 제곱법은 χ^2 을 최소화 시켜 찾을 수 있습니다. 이 값은 모델 $Y(c, x)$ 에 대해, n 개의 실험 값들의 가중 잔차 제곱합을 의미합니다.

$$\chi^2 = \sum_i w_i (y_i - Y(c, x_i))^2$$

모델의 p 계수는 $c = \{c_0, c_1, \dots\}$ 입니다. 가중 계수 w_i 는 $w_i = 1/\sigma_i^2$ 로 주어집니다. σ_i 는 지점 y_i 의 실험 오차입니다. 이 오차들은 정규 분포(가우스 분포)를 따르고 비상관(uncorrelated) 관계에 있다고 가정합니다. 가중치가 없는 값들에 대해, χ^2 은 가중치 계수 없이 계산됩니다.

피팅 함수들은 가장 잘 맞는 계수 c 와 $p \times p$ 차원의 공분산 행렬을 반환합니다. 공분산 행렬은 값들에 있는 오차(σ_i)로 인해 생기는 계수 c 의 통계적 오차를 측정합니다. 이 행렬은 다음과 같이 정의됩니다.

$$C_{ab} = \delta c_a \delta c_b$$

은 해당 값 아래, 가우스 오차 분포 함수의 평균값을 나타냅니다.

공분산 행렬은 각 지점의 오차 σ_i 의 오차 전파로 계산할 수 있습니다. 값 σ_{y_i} 의 변화로 생기는 피팅 계수의 변화량 δ_{c_a} 은 다음과 같이 주어집니다.

$$\sigma c_a = \sum_i \frac{\partial c_a}{\partial y_i} \delta y_i$$

비상관 값들은, 다음을 만족합니다.

$$\delta y_i \delta y_j = \sigma_i^2 \delta_{ij}$$

이때, 해당하는 공분산 행렬은 다음과 같이 계산할 수 있습니다.

$$C_{ab} = \sum_i \frac{1}{w_i} \frac{\partial c_a}{\partial y_i} \frac{\partial c_b}{\partial y_i}$$

가중치가 없는 값들은 공분산 행렬을 계산할때, 급수에서 가중 계수 w_i 가 단일값 $w = \frac{1}{\sigma^2}$ 로 대체됩니다. σ^2 는 최적 추정 계수에 대한, 잔차 분포로 $\sigma^2 = \sum (y_i - Y(c, x_i))^2/(n-p)$ 와 같이 계산됩니다. 이를 분산-공분산 행렬이라 합니다.

최적 추정 계수들에 대한 표준 편차는 해당하는 공분산 함수의 대각 성분의 제곱근으로 계산됩니다. $\sigma_{c_a} = \sqrt{C_{aa}}$. 계수 c_a 와 c_b 의 상관 계수는 $\rho_{ab} = C_{ab}/\sqrt{C_{aa}C_{bb}}$ 로 주어집니다.

39.2 선형 회귀(Linear regression)

이 단원의 함수들은 계수가 1, 2개인 간단한 선형 회귀 모델을 계산합니다. 이 단원의 함수들은 헤더 파일 `gsl_fit.h` 에 정의되어 있습니다.

39.2.1 상수항을 가지는 선형 회귀 모델

이 단원의 함수들은 일직선 형태의 모델 $Y(c, x) = c_o + c_1 x$ 에 대한 선형 회귀 분석을 계산합니다.

39.2.2 상수항이 없는 선형 회귀 모델

39.2.3 여러 계수를 가지는 선형 모델

39.3 정규화된 회귀 분석(Regularized regression)

39.4 로버스트 회귀 분석(Robust regression)

39.5 대규모 선형계(Large dense liear systems)

39.6 1

39.7 예제

다음의 프로그램은 선형 최소 제곱법을 사용해 주어진 데이터에 맞는 직선 식을 찾습니다. 그리고 최적의 피팅선과 버금 표준 편차를 출력합니다.

```
#include <stdio.h>
#include <gsl/gsl_fit.h>

int
main (void)
{
  int i, n = 4;
  double x[4] = { 1970, 1980, 1990, 2000 };
  double y[4] = {   12,   11,   14,   13 };
  double w[4] = {  0.1,  0.2,  0.3,  0.4 };

  double c0, c1, cov00, cov01, cov11, chisq;

  gsl_fit_wlinear (x, 1, w, 1, y, 1, n,
                   &c0, &c1, &cov00, &cov01, &cov11,
                   &chisq);

  printf ("# best fit: Y = %g + %g X\n", c0, c1);
  printf ("# covariance matrix:\n");
  printf ("# [ %g, %g\n#   %g, %g]\n",
          cov00, cov01, cov01, cov11);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  printf ("# chisq = %g\n", chisq);

  for (i = 0; i < n; i++)
    printf ("data: %g %g %g\n",
                  x[i], y[i], 1/sqrt(w[i]));

  printf ("\n");

  for (i = -30; i < 130; i++)
    {
      double xf = x[0] + (i/100.0) * (x[n-1] - x[0]);
      double yf, yf_err;

      gsl_fit_linear_est (xf,
                          c0, c1,
                          cov00, cov01, cov11,
                          &yf, &yf_err);

      printf ("fit: %g %g\n", xf, yf);
      printf ("hi : %g %g\n", xf, yf + yf_err);
      printf ("lo : %g %g\n", xf, yf - yf_err);
    }
  return 0;
}
```

다음의 명령어들은 프로그램의 출력값으로부터 데이터를 뽑아내고 GNU plotutils “graph” 도구를 이용해 시각 그래프를 만들어줍니다.

```
$./demo > tmp
$more tmp
# best fit: Y = -106.6 + 0.06 X
# covariance matrix:
# [ 39602, -19.9
#   -19.9, 0.01]
# chisq = 0.8

$for n in data fit hi lo ;
   do
     grep "^ :math:`n" tmp | cut -d: -f2 > ` n ;
   done
$graph -T X -X x -Y y -y 0 20 -m 0 -S 2 -Ie data
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    -S 0 -I a -m 1 fit -m 2 hi -m 2 lo
```

결과는 다음과 같습니다.

39.8 참고 문헌과 추가 자료

최소 제곱법과 관련된 수식과 기법들은 Particle Data Group에서 출판한 The Review of Particle Physics의 “Statistics” 단원을 참고할 수 있습니다.

- Review of Particle Properties, R.M. Barnett et al., Physical Review D54, 1 (1996) http://pdg.lbl.gov

The Review of Particle Physics은 위의 링크에서 볼 수 있습니다.

이 단원에서 구현된 기능들을 검사하는 데 NIST Statistical Reference Datasets을 사용했습니다. 해당 값과 문서들은 NIST 사이트를 참고할 수 있습니다.

http://www.nist.gov/itl/div898/strd/index.html

Tikhonov regularization에 대한 자세한 정보는 다음을 참고할 수 있습니다.

- Hansen, P. C. (1998), Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion. SIAM Monogr. on Mathematical Modeling and Computation, Society for Industrial and Applied Mathematics
- M. Rezghi and S. M. Hosseini (2009), A new variant of L-curve for Tikhonov regularization, Journal of Computational and Applied Mathematics, Volume 231, Issue 2, pages 914-924.

GSL의 로버스트 선형 회귀 구현체는 다음 출판물에 기반해 있습니다.

- DuMouchel, W. and F. O’Brien (1989), “Integrating a robust option into a multiple regression computing environment,” Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface, American Statistical Association
- Street, J.O., R.J. Carroll, and D. Ruppert (1988), “A note on computing robust regression estimates via iteratively reweighted least squares”, The American Statistician, v. 42, pp. 152-154.

정규 방정식들과 TSQR을 이용한 대규모 선형 최소 제곱계의 풀이는 다음을 참고할 수 있습니다.

- Trefethen, L. N. and Bau, D. (1997), “Numerical Linear Algebra”, SIAM.
- Demmel, J., Grigori, L., Hoemmen, M. F., and Langou, J. “Communication-optimal parallel and sequential QR and LU factorizations”, UCB Technical Report No. UCB/EECS-2008-89, 2008.

제 40 장

비선형 최소 제곱법

참고: 번역중

This chapter describes functions for multidimensional nonlinear least-squares fitting. There are generally two classes of algorithms for solving nonlinear least squares problems, which fall under line search methods and trust region methods. GSL currently implements only trust region methods and provides the user with full access to intermediate steps of the iteration. The user also has the ability to tune a number of parameters which affect low-level aspects of the algorithm which can help to accelerate convergence for the specific problem at hand. GSL provides two separate interfaces for nonlinear least squares fitting. The first is designed for small to moderate sized problems, and the second is designed for very large problems, which may or may not have significant sparse structure.

The header file `gsl_multifit_nlinear.h` contains prototypes for the multidimensional nonlinear fitting functions and related declarations relating to the small to moderate sized systems.

The header file `gsl_multilarge_nlinear.h` contains prototypes for the multidimensional nonlinear fitting functions and related declarations relating to large systems.

40.1 Overview

The problem of multidimensional nonlinear least-squares fitting requires the minimization of the squared residuals of n functions, f_i, in p parameters, x_i,

$$\begin{aligned}\Phi(x) &= \frac{1}{2}||f(x)||^2 \\ &= \frac{1}{2}\sum_{i=1}^{n} f_i(x_1, \ldots, x_p)^2\end{aligned}$$

In trust region methods, the objective (or cost) function $\Phi(x)$ is approximated by a model function $m_k(\delta)$ in the vicinity of some point x_k. The model function is often simply a second order Taylor series expansion around the point x_k, ie:

$$\Phi(x_k + \delta) \approx m_k(\delta) = \Phi(x_k) + g_k^T\delta + \frac{1}{2}\delta^T B_k \delta$$

where $g_k = \nabla\Phi(x_k) = J^T f$ is the gradient vector at the point x_k, $B_k = \nabla^2\Phi(x_k)$ is the Hessian matrix at x_k, or some approximation to it, and J is the n-by-p Jacobian matrix

$$J_{ij} = \partial f_i/\partial x_j$$

In order to find the next step δ, we minimize the model function $m_k(\delta)$, but search for solutions only within a region where we trust that $m_k(\delta)$ is a good approximation to the objective function $\Phi(x_k + \delta)$. In other words, we seek a solution of the trust region subproblem (TRS)

$$\min_{\delta \in R^p} m_k(\delta) = \Phi(x_k) + g_k^T\delta + \frac{1}{2}\delta^T B_k \delta, \qquad \text{s.t.} \quad ||D_k\delta|| \le \Delta_k$$

where $\Delta_k > 0$ is the trust region radius and D_k is a scaling matrix. If $D_k = I$, then the trust region is a ball of radius Δ_k centered at x_k. In some applications, the parameter vector x may have widely different scales. For example, one parameter might be a temperature on the order of 10^3 K, while another might be a length on the order of 10^{-6} m. In such cases, a spherical trust region may not be the best choice, since if Φ changes rapidly along directions with one scale, and more slowly along directions with a different scale, the model function m_k may be a poor approximation to Φ along the rapidly changing directions. In such problems, it may be best to use an elliptical trust region, by setting D_k to a diagonal matrix whose entries are designed so that the scaled step $D_k\delta$ has entries of approximately the same order of magnitude.

The trust region subproblem above normally amounts to solving a linear least squares system (or multiple systems) for the step δ. Once δ is computed, it is checked whether or not it reduces

the objective function $\Phi(x)$. A useful statistic for this is to look at the ratio

$$\rho_k = \frac{\Phi(x_k) - \Phi(x_k + \delta_k)}{m_k(0) - m_k(\delta_k)}$$

where the numerator is the actual reduction of the objective function due to the step δ_k, and the denominator is the predicted reduction due to the model m_k. If ρ_k is negative, it means that the step δ_k increased the objective function and so it is rejected. If ρ_k is positive, then we have found a step which reduced the objective function and it is accepted. Furthermore, if ρ_k is close to 1, then this indicates that the model function is a good approximation to the objective function in the trust region, and so on the next iteration the trust region is enlarged in order to take more ambitious steps. When a step is rejected, the trust region is made smaller and the TRS is solved again. An outline for the general trust region method used by GSL can now be given.

Trust Region Algorithm

1. Initialize: given x_0, construct $m_0(\delta)$, D_0 and $\Delta_0 > 0$
2. For k = 0, 1, 2, ⋯
   ㄱ. If converged, then stop
   ㄴ. Solve TRS for trial step δ_k
   ㄷ. Evaluate trial step by computing ρ_k
      1). if step is accepted, set $x_{k+1} = x_k + \delta_k$ and increase radius, $\Delta_{k+1} = \alpha\Delta_k$
      2). if step is rejected, set $x_{k+1} = x_k$ and decrease radius, $\Delta_{k+1} = \frac{\Delta_k}{\beta}$; goto 2(b)
   ㄹ. Construct $m_{k+1}(\delta)$ and D_{k+1}

GSL offers the user a number of different algorithms for solving the trust region subproblem in 2(b), as well as different choices of scaling matrices D_k and different methods of updating the trust region radius Δ_k. Therefore, while reasonable default methods are provided, the user has a lot of control to fine-tune the various steps of the algorithm for their specific problem.

40.2 Solving the Trust Region Subproblem (TRS)

Below we describe the methods available for solving the trust region subproblem. The methods available provide either exact or approximate solutions to the trust region subproblem. In all algorithms below, the Hessian matrix B_k is approximated as $B_k \approx J_k^T J_k$, where $J_k = J(x_k)$. In all methods, the solution of the TRS involves solving a linear least squares system involving the Jacobian matrix. For small to moderate sized problems (`gsl_multifit_nlinear` interface), this is accomplished by factoring the full Jacobian matrix, which is provided by the user, with the

Cholesky, QR, or SVD decompositions. For large systems (`gsl_multilarge_nlinear` interface), the user has two choices. One is to solve the system iteratively, without needing to store the full Jacobian matrix in memory. With this method, the user must provide a routine to calculate the matrix-vector products Ju or $J^T u$ for a given vector u. This iterative method is particularly useful for systems where the Jacobian has sparse structure, since forming matrix-vector products can be done cheaply. The second option for large systems involves forming the normal equations matrix $J^T J$ and then factoring it using a Cholesky decomposition. The normal equations matrix is p-by-p, typically much smaller than the full n-by-p Jacobian, and can usually be stored in memory even if the full Jacobian matrix cannot. This option is useful for large, dense systems, or if the iterative method has difficulty converging.

40.2.1 Levenberg-Marquardt

There is a theorem which states that if δ_k is a solution to the trust region subproblem given above, then there exists $\mu_k \geq 0$ such that

$$\left(B_k + \mu_k D_k^T D_k\right) \delta_k = -g_k$$

with $\mu_k(\Delta_k - ||D_k \delta_k||) = 0$. This forms the basis of the Levenberg-Marquardt algorithm, which controls the trust region size by adjusting the parameter μ_k rather than the radius Δ_k directly. For each radius Δ_k, there is a unique parameter μ_k which solves the TRS, and they have an inverse relationship, so that large values of μ_k correspond to smaller trust regions, while small values of μ_k correspond to larger trust regions.

With the approximation $B_k \approx J_k^T J_k$, on each iteration, in order to calculate the step δ_k, the following linear least squares problem is solved:

$$\left[\begin{array}{c} J_k \\ \sqrt{\mu_k} D_k \end{array} \right] \delta_k = - \left[\begin{array}{c} f_k \\ 0 \end{array} \right]$$

If the step δ_k is accepted, then μ_k is decreased on the next iteration in order to take a larger step, otherwise it is increased to take a smaller step. The Levenberg-Marquardt algorithm provides an exact solution of the trust region subproblem, but typically has a higher computational cost per iteration than the approximate methods discussed below, since it may need to solve the least squares system above several times for different values of μ_k.

40.2.2 Levenberg-Marquardt with Geodesic Acceleration

This method applies a so-called geodesic acceleration correction to the standard Levenberg-Marquardt step δ_k (Transtrum et al, 2011). By interpreting δ_k as a first order step along a geodesic in the model parameter space (ie: a velocity $\delta_k = v_k$), the geodesic acceleration a_k is a second order correction along the geodesic which is determined by solving the linear least squares system

$$\left[\begin{array}{c} J_k \\ \sqrt{\mu_k} D_k \end{array} \right] a_k = - \left[\begin{array}{c} f_{vv}(x_k) \\ 0 \end{array} \right]$$

where f_{vv} is the second directional derivative of the residual vector in the velocity direction v, $f_{vv}(x) = D_v^2 f = \sum_{\alpha\beta} v_\alpha v_\beta \partial_\alpha \partial_\beta f(x)$, where α and β are summed over the p parameters. The new total step is then $\delta_k' = v_k + \frac{1}{2} a_k$. The second order correction a_k can be calculated with a modest additional cost, and has been shown to dramatically reduce the number of iterations (and expensive Jacobian evaluations) required to reach convergence on a variety of different problems. In order to utilize the geodesic acceleration, the user must supply a function which provides the second directional derivative vector $f_{vv}(x)$, or alternatively the library can use a finite difference method to estimate this vector with one additional function evaluation of $f(x + hv)$ where h is a tunable step size (see the `h_fvv` parameter description).

40.2.3 Dogleg

This is Powell's dogleg method, which finds an approximate solution to the trust region subproblem, by restricting its search to a piecewise linear "dogleg" path, composed of the origin, the Cauchy point which represents the model minimizer along the steepest descent direction, and the Gauss-Newton point, which is the overall minimizer of the unconstrained model. The Gauss-Newton step is calculated by solving

$$J_k \delta_{gn} = -f_k$$

which is the main computational task for each iteration, but only needs to be performed once per iteration. If the Gauss-Newton point is inside the trust region, it is selected as the step. If it is outside, the method then calculates the Cauchy point, which is located along the gradient direction. If the Cauchy point is also outside the trust region, the method assumes that it is still far from the minimum and so proceeds along the gradient direction, truncating the step at the trust region boundary. If the Cauchy point is inside the trust region, with the Gauss-Newton point outside, the method uses a dogleg step, which is a linear combination of the gradient direction and the Gauss-Newton direction, stopping at the trust region boundary.

40.2.4 Double Dogleg

This method is an improvement over the classical dogleg algorithm, which attempts to include information about the Gauss-Newton step while the iteration is still far from the minimum. When the Cauchy point is inside the trust region and the Gauss-Newton point is outside, the method computes a scaled Gauss-Newton point and then takes a dogleg step between the Cauchy point and the scaled Gauss-Newton point. The scaling is calculated to ensure that the reduction in the model m_k is about the same as the reduction provided by the Cauchy point.

40.2.5 Two Dimensional Subspace

The dogleg methods restrict the search for the TRS solution to a 1D curve defined by the Cauchy and Gauss-Newton points. An improvement to this is to search for a solution using the full two dimensional subspace spanned by the Cauchy and Gauss-Newton directions. The dogleg path is of course inside this subspace, and so this method solves the TRS at least as accurately as the dogleg methods. Since this method searches a larger subspace for a solution, it can converge more quickly than dogleg on some problems. Because the subspace is only two dimensional, this method is very efficient and the main computation per iteration is to determine the Gauss-Newton point.

40.2.6 Steihaug-Toint Conjugate Gradient

One difficulty of the dogleg methods is calculating the Gauss-Newton step when the Jacobian matrix is singular. The Steihaug-Toint method also computes a generalized dogleg step, but avoids solving for the Gauss-Newton step directly, instead using an iterative conjugate gradient algorithm. This method performs well at points where the Jacobian is singular, and is also suitable for large-scale problems where factoring the Jacobian matrix could be prohibitively expensive.

40.3 Weighted Nonlinear Least-Squares

Weighted nonlinear least-squares fitting minimizes the function

$$\begin{aligned}\Phi(x) &= \frac{1}{2}||f||_W^2 \\ &= \frac{1}{2}\sum_{i=1}^{n} w_i f_i(x_1, \ldots, x_p)^2\end{aligned}$$

where $W = (w_1, w_2, ..., w_n)$ is the weighting matrix, and $||f||_W^2 = f^T W f$. The weights w_i are commonly defined as $w_i = 1/\sigma_i^2$, where σ_i is the error in the i-th measurement. A simple change of variables $\tilde{f} = W^{\frac{1}{2}} f$ yields $\Phi(x) = \frac{1}{2}||\tilde{f}||^2$, which is in the same form as the unweighted case. The user can either perform this transform directly on their function residuals and Jacobian, or use the `gsl_multifit_nlinear_winit()` interface which automatically performs the correct scaling. To manually perform this transformation, the residuals and Jacobian should be modified according to

$$\tilde{f}_i = \sqrt{w_i} f_i = \frac{f_i}{\sigma_i}$$
$$\tilde{J}_{ij} = \sqrt{w_i} \frac{\partial f_i}{\partial x_j} = \frac{1}{\sigma_i} \frac{\partial f_i}{\partial x_j}$$

For large systems, the user must perform their own weighting.

40.4 Tunable Parameters

The user can tune nearly all aspects of the iteration at allocation time. For the `gsl_multifit_nlinear` interface, the user may modify the `gsl_multifit_nlinear_parameters` structure, which is defined as follows:

type **`gsl_multifit_nlinear_parameters`**

```
typedef struct
{
  const gsl_multifit_nlinear_trs *trs;        /* trust region subproblem method */
  const gsl_multifit_nlinear_scale *scale;    /* scaling method */
  const gsl_multifit_nlinear_solver *solver;  /* solver method */
  gsl_multifit_nlinear_fdtype fdtype;         /* finite difference method */
  double factor_up;                           /* factor for increasing trust radius */
  double factor_down;                         /* factor for decreasing trust radius */
  double avmax;                               /* max allowed |a|/|v| */
  double h_df;                                /* step size for finite difference Jacobian */
  double h_fvv;                               /* step size for finite difference fvv */
} gsl_multifit_nlinear_parameters;
```

For the `gsl_multilarge_nlinear` interface, the user may modify the `gsl_multilarge_nlinear_parameters` structure, which is defined as follows:

type **`gsl_multilarge_nlinear_parameters`**

```
typedef struct
{
  const gsl_multilarge_nlinear_trs *trs;       /* trust region subproblem method */
  const gsl_multilarge_nlinear_scale *scale;   /* scaling method */
  const gsl_multilarge_nlinear_solver *solver; /* solver method */
  gsl_multilarge_nlinear_fdtype fdtype;        /* finite difference method */
  double factor_up;                            /* factor for increasing trust radius */
  double factor_down;                          /* factor for decreasing trust radius */
  double avmax;                                /* max allowed |a|/|v| */
  double h_df;                                 /* step size for finite difference Jacobian */
  double h_fvv;                                /* step size for finite difference fvv */
  size_t max_iter;                             /* maximum iterations for trs method */
  double tol;                                  /* tolerance for solving trs */
} gsl_multilarge_nlinear_parameters;
```

Each of these parameters is discussed in further detail below.

type **gsl_multifit_nlinear_trs**

type **gsl_multilarge_nlinear_trs**

The parameter `trs` determines the method used to solve the trust region subproblem, and may be selected from the following choices,

gsl_multifit_nlinear_trs ***gsl_multifit_nlinear_trs_lm**

gsl_multilarge_nlinear_trs ***gsl_multilarge_nlinear_trs_lm**

This selects the Levenberg-Marquardt algorithm.

gsl_multifit_nlinear_trs ***gsl_multifit_nlinear_trs_lmaccel**

gsl_multilarge_nlinear_trs ***gsl_multilarge_nlinear_trs_lmaccel**

This selects the Levenberg-Marquardt algorithm with geodesic acceleration.

gsl_multifit_nlinear_trs ***gsl_multifit_nlinear_trs_dogleg**

gsl_multilarge_nlinear_trs ***gsl_multilarge_nlinear_trs_dogleg**

This selects the dogleg algorithm.

gsl_multifit_nlinear_trs ***gsl_multifit_nlinear_trs_ddogleg**

gsl_multilarge_nlinear_trs ***gsl_multilarge_nlinear_trs_ddogleg**

This selects the double dogleg algorithm.

gsl_multifit_nlinear_trs ***gsl_multifit_nlinear_trs_subspace2D**

gsl_multilarge_nlinear_trs ***gsl_multilarge_nlinear_trs_subspace2D**

This selects the 2D subspace algorithm.

gsl_multilarge_nlinear_trs *gsl_multilarge_nlinear_trs_cgst

This selects the Steihaug-Toint conjugate gradient algorithm. This method is available only for large systems.

type gsl_multifit_nlinear_scale

type gsl_multilarge_nlinear_scale

The parameter scale determines the diagonal scaling matrix D and may be selected from the following choices,

gsl_multifit_nlinear_scale *gsl_multifit_nlinear_scale_more

gsl_multilarge_nlinear_scale *gsl_multilarge_nlinear_scale_more

This damping strategy was suggested by Moré, and corresponds to $D^T D = \max((J^T J))$, in other words the maximum elements of $(J^T J)$ encountered thus far in the iteration. This choice of D makes the problem scale-invariant, so that if the model parameters x_i are each scaled by an arbitrary constant, $\tilde{x}_i = a_i x_i$, then the sequence of iterates produced by the algorithm would be unchanged. This method can work very well in cases where the model parameters have widely different scales (ie: if some parameters are measured in nanometers, while others are measured in degrees Kelvin). This strategy has been proven effective on a large class of problems and so it is the library default, but it may not be the best choice for all problems.

gsl_multifit_nlinear_scale *gsl_multifit_nlinear_scale_levenberg

gsl_multilarge_nlinear_scale *gsl_multilarge_nlinear_scale_levenberg

This damping strategy was originally suggested by Levenberg, and corresponds to $D^T D = I$. This method has also proven effective on a large class of problems, but is not scale-invariant. However, some authors (e.g. Transtrum and Sethna 2012) argue that this choice is better for problems which are susceptible to parameter evaporation (ie: parameters go to infinity)

gsl_multifit_nlinear_scale *gsl_multifit_nlinear_scale_marquardt

gsl_multilarge_nlinear_scale *gsl_multilarge_nlinear_scale_marquardt

This damping strategy was suggested by Marquardt, and corresponds to $D^T D = (J^T J)$. This method is scale-invariant, but it is generally considered inferior to both the Levenberg and Moré strategies, though may work well on certain classes of problems.

type gsl_multifit_nlinear_solver

type gsl_multilarge_nlinear_solver

Solving the trust region subproblem on each iteration almost always requires the solution

of the following linear least squares system

$$\left[\begin{array}{c} J \\ \sqrt{\mu} D \end{array} \right] \delta = - \left[\begin{array}{c} f \\ 0 \end{array} \right]$$

The `solver` parameter determines how the system is solved and can be selected from the following choices:

gsl_multifit_nlinear_solver ***gsl_multifit_nlinear_solver_qr**

This method solves the system using a rank revealing QR decomposition of the Jacobian J. This method will produce reliable solutions in cases where the Jacobian is rank deficient or near-singular but does require about twice as many operations as the Cholesky method discussed below.

gsl_multifit_nlinear_solver ***gsl_multifit_nlinear_solver_cholesky**

gsl_multilarge_nlinear_solver ***gsl_multilarge_nlinear_solver_cholesky**

This method solves the alternate normal equations problem

$$\left(J^T J + \mu D^T D\right) \delta = -J^T f$$

by using a Cholesky decomposition of the matrix $J^T J + \mu D^T D$. This method is faster than the QR approach, however it is susceptible to numerical instabilities if the Jacobian matrix is rank deficient or near-singular. In these cases, an attempt is made to reduce the condition number of the matrix using Jacobi preconditioning, but for highly ill-conditioned problems the QR approach is better. If it is known that the Jacobian matrix is well conditioned, this method is accurate and will perform faster than the QR approach.

gsl_multifit_nlinear_solver ***gsl_multifit_nlinear_solver_mcholesky**

gsl_multilarge_nlinear_solver ***gsl_multilarge_nlinear_solver_mcholesky**

This method solves the alternate normal equations problem

$$\left(J^T J + \mu D^T D\right) \delta = -J^T f$$

by using a modified Cholesky decomposition of the matrix $J^T J + \mu D^T D$. This is more suitable for the dogleg methods where the parameter $\mu = 0$, and the matrix $J^T J$ may be ill-conditioned or indefinite causing the standard Cholesky decomposition to fail. This method is based on Level 2 BLAS and is thus slower than the standard Cholesky decomposition, which is based on Level 3 BLAS.

gsl_multifit_nlinear_solver ***gsl_multifit_nlinear_solver_svd**

This method solves the system using a singular value decomposition of the Jacobian

J. This method will produce the most reliable solutions for ill-conditioned Jacobians but is also the slowest solver method.

type **gsl_multifit_nlinear_fdtype**

The parameter `fdtype` specifies whether to use forward or centered differences when approximating the Jacobian. This is only used when an analytic Jacobian is not provided to the solver. This parameter may be set to one of the following choices.

GSL_MULTIFIT_NLINEAR_FWDIFF

This specifies a forward finite difference to approximate the Jacobian matrix. The Jacobian matrix will be calculated as

$$J_{ij} = \frac{1}{\Delta_j} \left(f_i(x + \Delta_j e_j) - f_i(x) \right)$$

where $\Delta_j = h|x_j|$ and e_j is the standard j-th Cartesian unit basis vector so that $x + \Delta_j e_j$ represents a small (forward) perturbation of the j-th parameter by an amount Δ_j. The perturbation Δ_j is proportional to the current value $|x_j|$ which helps to calculate an accurate Jacobian when the various parameters have different scale sizes. The value of h is specified by the `h_df` parameter. The accuracy of this method is $O(h)$, and evaluating this matrix requires an additional p function evaluations.

GSL_MULTIFIT_NLINEAR_CTRDIFF

This specifies a centered finite difference to approximate the Jacobian matrix. The Jacobian matrix will be calculated as

$$J_{ij} = \frac{1}{\Delta_j} \left(f_i(x + \frac{1}{2}\Delta_j e_j) - f_i(x - \frac{1}{2}\Delta_j e_j) \right)$$

See above for a description of Δ_j. The accuracy of this method is $O(h^2)$, but evaluating this matrix requires an additional $2p$ function evaluations.

`double factor_up`

When a step is accepted, the trust region radius will be increased by this factor. The default value is 3.

`double factor_down`

When a step is rejected, the trust region radius will be decreased by this factor. The default value is 2.

`double avmax`

When using geodesic acceleration to solve a nonlinear least squares problem, an important

parameter to monitor is the ratio of the acceleration term to the velocity term,

$$\frac{||a||}{||v||}$$

If this ratio is small, it means the acceleration correction is contributing very little to the step. This could be because the problem is not "nonlinear" enough to benefit from the acceleration. If the ratio is large (> 1) it means that the acceleration is larger than the velocity, which shouldn't happen since the step represents a truncated series and so the second order term a should be smaller than the first order term v to guarantee convergence. Therefore any steps with a ratio larger than the parameter `avmax` are rejected. `avmax` is set to 0.75 by default. For problems which experience difficulty converging, this threshold could be lowered.

`double h_df`

This parameter specifies the step size for approximating the Jacobian matrix with finite differences. It is set to $\sqrt{\epsilon}$ by default, where ϵ is `GSL_DBL_EPSILON`.

`double h_fvv`

When using geodesic acceleration, the user must either supply a function to calculate $f_{vv}(x)$ or the library can estimate this second directional derivative using a finite difference method. When using finite differences, the library must calculate $f(x + hv)$ where h represents a small step in the velocity direction. The parameter `h_fvv` defines this step size and is set to 0.02 by default.

40.5 Initializing the Solver

type **gsl_multifit_nlinear_type**

This structure specifies the type of algorithm which will be used to solve a nonlinear least squares problem. It may be selected from the following choices,

gsl_multifit_nlinear_type ***gsl_multifit_nlinear_trust**

This specifies a trust region method. It is currently the only implemented nonlinear least squares method.

gsl_multifit_nlinear_workspace ***gsl_multifit_nlinear_alloc**(const gsl_multifit_nlinear_type *T, const gsl_multifit_nlinear_parameters *params, const size_t n, const size_t p)

gsl_multilarge_nlinear_workspace *gsl_multilarge_nlinear_alloc(const gsl_multilarge_nlinear_type *T, const gsl_multilarge_nlinear_parameters *params, const size_t n, const size_t p)

These functions return a pointer to a newly allocated instance of a derivative solver of type `T` for `n` observations and `p` parameters. The `params` input specifies a tunable set of parameters which will affect important details in each iteration of the trust region subproblem algorithm. It is recommended to start with the suggested default parameters (see `gsl_multifit_nlinear_default_parameters()` and `gsl_multilarge_nlinear_default_parameters()`) and then tune the parameters once the code is working correctly. See Tunable Parameters. for descriptions of the various parameters. For example, the following code creates an instance of a Levenberg-Marquardt solver for 100 data points and 3 parameters, using suggested defaults:

```
const gsl_multifit_nlinear_type * T = gsl_multifit_nlinear_trust;
gsl_multifit_nlinear_parameters params = gsl_multifit_nlinear_default_parameters();
gsl_multifit_nlinear_workspace * w = gsl_multifit_nlinear_alloc (T, &params, 100, 3);
```

The number of observations `n` must be greater than or equal to parameters `p`.

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

gsl_multifit_nlinear_parameters **gsl_multifit_nlinear_default_parameters**(void)

gsl_multilarge_nlinear_parameters **gsl_multilarge_nlinear_default_parameters**(void)

These functions return a set of recommended default parameters for use in solving nonlinear least squares problems. The user can tune each parameter to improve the performance on their particular problem, see Tunable Parameters.

int **gsl_multifit_nlinear_init**(const gsl_vector *x, gsl_multifit_nlinear_fdf *fdf, gsl_multifit_nlinear_workspace *w)

int **gsl_multifit_nlinear_winit**(const gsl_vector *x, const gsl_vector *wts, gsl_multifit_nlinear_fdf *fdf, gsl_multifit_nlinear_workspace *w)

int **gsl_multilarge_nlinear_init**(const gsl_vector *x, gsl_multilarge_nlinear_fdf *fdf, gsl_multilarge_nlinear_workspace *w)

These functions initialize, or reinitialize, an existing workspace `w` to use the system `fdf` and the initial guess `x`. See Providing the Function to be Minimized for a description of the `fdf` structure.

Optionally, a weight vector `wts` can be given to perform a weighted nonlinear regression. Here, the weighting matrix is $W = (w_1, w_2, ..., w_n)$.

void **gsl_multifit_nlinear_free**(gsl_multifit_nlinear_workspace *w)

void **gsl_multilarge_nlinear_free**(gsl_multilarge_nlinear_workspace *w)

These functions free all the memory associated with the workspace `w`.

const char ***gsl_multifit_nlinear_name**(const gsl_multifit_nlinear_workspace *w)

const char ***gsl_multilarge_nlinear_name**(const gsl_multilarge_nlinear_workspace *w)

These functions return a pointer to the name of the solver. For example:

```
printf ("w is a '%s' solver\n", gsl_multifit_nlinear_name (w));
```

would print something like `w is a 'trust-region' solver`.

const char ***gsl_multifit_nlinear_trs_name**(const gsl_multifit_nlinear_workspace *w)

const char ***gsl_multilarge_nlinear_trs_name**(const gsl_multilarge_nlinear_workspace *w)

These functions return a pointer to the name of the trust region subproblem method. For example:

```
printf ("w is a '%s' solver\n", gsl_multifit_nlinear_trs_name (w));
```

would print something like `w is a 'levenberg-marquardt' solver`.

40.6 Providing the Function to be Minimized

The user must provide n functions of p variables for the minimization algorithm to operate on. In order to allow for arbitrary parameters the functions are defined by the following data types:

type **gsl_multifit_nlinear_fdf**

This data type defines a general system of functions with arbitrary parameters, the corresponding Jacobian matrix of derivatives, and optionally the second directional derivative of the functions for geodesic acceleration.

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`

This function should store the n components of the vector $f(x)$ in `f` for argument `x` and arbitrary parameters `params`, returning an appropriate error code if the function cannot be computed.

`int (* df) (const gsl_vector * x, void * params, gsl_matrix * J)`

This function should store the n-by-p matrix result

$$J_{ij} = \partial f_i(x)/\partial x_j$$

in J for argument x and arbitrary parameters params, returning an appropriate error code if the matrix cannot be computed. If an analytic Jacobian is unavailable, or too expensive to compute, this function pointer may be set to NULL, in which case the Jacobian will be internally computed using finite difference approximations of the function f.

`int (* fvv) (const gsl_vector * x, const gsl_vector * v, void * params, gsl_vector * fvv)`

When geodesic acceleration is enabled, this function should store the n components of the vector $f_{vv}(x) = \sum_{\alpha\beta} v_\alpha v_\beta \frac{\partial}{\partial x_\alpha}\frac{\partial}{\partial x_\beta} f(x)$, representing second directional derivatives of the function to be minimized, into the output fvv. The parameter vector is provided in x and the velocity vector is provided in v, both of which have p components. The arbitrary parameters are given in params. If analytic expressions for $f_{vv}(x)$ are unavailable or too difficult to compute, this function pointer may be set to NULL, in which case $f_{vv}(x)$ will be computed internally using a finite difference approximation.

`size_t n`

the number of functions, i.e. the number of components of the vector f.

`size_t p`

the number of independent variables, i.e. the number of components of the vector x.

`void * params`

a pointer to the arbitrary parameters of the function.

`size_t nevalf`

This does not need to be set by the user. It counts the number of function evaluations and is initialized by the _init function.

`size_t nevaldf`

This does not need to be set by the user. It counts the number of Jacobian evaluations and is initialized by the _init function.

`size_t nevalfvv`

This does not need to be set by the user. It counts the number of $f_{vv}(x)$ evaluations and is initialized by the `_init` function.

type **gsl_multilarge_nlinear_fdf**

This data type defines a general system of functions with arbitrary parameters, a function to compute Ju or $J^T u$ for a given vector u, the normal equations matrix $J^T J$, and optionally the second directional derivative of the functions for geodesic acceleration.

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`

This function should store the n components of the vector $f(x)$ in `f` for argument `x` and arbitrary parameters `params`, returning an appropriate error code if the function cannot be computed.

`int (* df) (CBLAS_TRANSPOSE_t TransJ, const gsl_vector * x, const gsl_vector * u, void * params, gsl_vector * v, gsl_matrix * JTJ)`

If `TransJ` is equal to `CblasNoTrans`, then this function should compute the matrix-vector product Ju and store the result in `v`. If `TransJ` is equal to `CblasTrans`, then this function should compute the matrix-vector product $J^T u$ and store the result in `v`. Additionally, the normal equations matrix $J^T J$ should be stored in the lower half of `JTJ`. The input matrix `JTJ` could be set to `NULL`, for example by iterative methods which do not require this matrix, so the user should check for this prior to constructing the matrix. The input `params` contains the arbitrary parameters.

`int (* fvv) (const gsl_vector * x, const gsl_vector * v, void * params, gsl_vector * fvv)`

When geodesic acceleration is enabled, this function should store the n components of the vector $f_{vv}(x) = \sum_{\alpha\beta} v_\alpha v_\beta \frac{\partial}{\partial x_\alpha} \frac{\partial}{\partial x_\beta} f(x)$, representing second directional derivatives of the function to be minimized, into the output `fvv`. The parameter vector is provided in `x` and the velocity vector is provided in `v`, both of which have p components. The arbitrary parameters are given in `params`. If analytic expressions for $f_{vv}(x)$ are unavailable or too difficult to compute, this function pointer may be set to `NULL`, in which case $f_{vv}(x)$ will be computed internally using a finite difference approximation.

`size_t n`

the number of functions, i.e. the number of components of the vector `f`.

`size_t p`

the number of independent variables, i.e. the number of components of the vector `x`.

`void * params`

a pointer to the arbitrary parameters of the function.

`size_t nevalf`

This does not need to be set by the user. It counts the number of function evaluations and is initialized by the `_init` function.

`size_t nevaldfu`

This does not need to be set by the user. It counts the number of Jacobian matrix-vector evaluations (Ju or $J^T u$) and is initialized by the `_init` function.

`size_t nevaldf2`

This does not need to be set by the user. It counts the number of $J^T J$ evaluations and is initialized by the `_init` function.

`size_t nevalfvv`

This does not need to be set by the user. It counts the number of $f_{vv}(x)$ evaluations and is initialized by the `_init` function.

Note that when fitting a non-linear model against experimental data, the data is passed to the functions above using the `params` argument and the trial best-fit parameters through the `x` argument.

40.7 Iteration

The following functions drive the iteration of each algorithm. Each function performs one iteration of the trust region method and updates the state of the solver.

int **gsl_multifit_nlinear_iterate**(gsl_multifit_nlinear_workspace *w)

int **gsl_multilarge_nlinear_iterate**(gsl_multilarge_nlinear_workspace *w)

These functions perform a single iteration of the solver `w`. If the iteration encounters an unexpected problem then an error code will be returned. The solver workspace maintains a current estimate of the best-fit parameters at all times.

The solver workspace `w` contains the following entries, which can be used to track the progress of the solution:

`gsl_vector * x`

The current position, length p.

`gsl_vector * f`

The function residual vector at the current position $f(x)$, length n.

`gsl_matrix * J`

The Jacobian matrix at the current position $J(x)$, size n-by-p (only for `gsl_multifit_nlinear` interface).

`gsl_vector * dx`

The difference between the current position and the previous position, i.e. the last step δ, taken as a vector, length p.

These quantities can be accessed with the following functions,

gsl_vector ***gsl_multifit_nlinear_position**(const gsl_multifit_nlinear_workspace *w)

gsl_vector ***gsl_multilarge_nlinear_position**(const gsl_multilarge_nlinear_workspace *w)

These functions return the current position x (i.e. best-fit parameters) of the solver `w`.

gsl_vector ***gsl_multifit_nlinear_residual**(const gsl_multifit_nlinear_workspace *w)

gsl_vector ***gsl_multilarge_nlinear_residual**(const gsl_multilarge_nlinear_workspace *w)

These functions return the current residual vector $f(x)$ of the solver `w`. For weighted systems, the residual vector includes the weighting factor $\sqrt{W}$.

gsl_matrix ***gsl_multifit_nlinear_jac**(const gsl_multifit_nlinear_workspace *w)

This function returns a pointer to the n-by-p Jacobian matrix for the current iteration of the solver `w`. This function is available only for the `gsl_multifit_nlinear` interface.

size_t **gsl_multifit_nlinear_niter**(const gsl_multifit_nlinear_workspace *w)

size_t **gsl_multilarge_nlinear_niter**(const gsl_multilarge_nlinear_workspace *w)

These functions return the number of iterations performed so far. The iteration counter is updated on each call to the `_iterate` functions above, and reset to 0 in the `_init` functions.

int **gsl_multifit_nlinear_rcond**(double *rcond, const gsl_multifit_nlinear_workspace *w)

int **gsl_multilarge_nlinear_rcond**(double *rcond, const gsl_multilarge_nlinear_workspace *w)

This function estimates the reciprocal condition number of the Jacobian matrix at the current position x and stores it in `rcond`. The computed value is only an estimate to give the user a guideline as to the conditioning of their particular problem. Its calculation is based on which factorization method is used (Cholesky, QR, or SVD).

- For the Cholesky solver, the matrix $J^T J$ is factored at each iteration. Therefore this function will estimate the 1-norm condition number $rcond^2 = 1/(||J^T J||_1 \cdot ||(J^T J)^{-1}||_1)$
- For the QR solver, J is factored as $J = QR$ at each iteration. For simplicity, this function calculates the 1-norm conditioning of only the R factor, $rcond = 1/(||R||_1 \cdot ||R^{-1}||_1)$. This can be computed efficiently since R is upper triangular.

- For the SVD solver, in order to efficiently solve the trust region subproblem, the matrix which is factored is JD^{-1}, instead of J itself. The resulting singular values are used to provide the 2-norm reciprocal condition number, as $rcond = \sigma_{min}/\sigma_{max}$. Note that when using Moré scaling, $D \neq I$ and the resulting `rcond` estimate may be significantly different from the true `rcond` of J itself.

double **gsl_multifit_nlinear_avratio**(const gsl_multifit_nlinear_workspace *w)

double **gsl_multilarge_nlinear_avratio**(const gsl_multilarge_nlinear_workspace *w)

This function returns the current ratio $|a|/|v|$ of the acceleration correction term to the velocity step term. The acceleration term is computed only by the `gsl_multifit_nlinear_trs_lmaccel` and `gsl_multilarge_nlinear_trs_lmaccel` methods, so this ratio will be zero for other TRS methods.

40.8 Testing for Convergence

A minimization procedure should stop when one of the following conditions is true:

- A minimum has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the current estimate of the best-fit parameters in several standard ways.

int **gsl_multifit_nlinear_test**(const double xtol, const double gtol, const double ftol, int *info, const gsl_multifit_nlinear_workspace *w)

int **gsl_multilarge_nlinear_test**(const double xtol, const double gtol, const double ftol, int *info, const gsl_multilarge_nlinear_workspace *w)

These functions test for convergence of the minimization method using the following criteria:

- Testing for a small step size relative to the current parameter vector

$$|\delta_i| \leq xtol(|x_i| + xtol)$$

for each $0 <= i < p$. Each element of the step vector δ is tested individually in case the different parameters have widely different scales. Adding `xtol` to $|x_i|$ helps the test avoid breaking down in situations where the true solution value $x_i = 0$. If this test succeeds, `info` is set to 1 and the function returns `GSL_SUCCESS`.

A general guideline for selecting the step tolerance is to choose $xtol = 10^{-d}$ where d is the number of accurate decimal digits desired in the solution x. See Dennis and Schnabel for more information.

- Testing for a small gradient ($g = \nabla\Phi(x) = J^T f$) indicating a local function minimum:

$$\max_i |g_i \times \max(x_i, 1)| \leq gtol \times \max(\Phi(x), 1)$$

This expression tests whether the ratio $(\nabla\Phi)_i x_i/\Phi$ is small. Testing this scaled gradient is a better than $\nabla\Phi$ alone since it is a dimensionless quantity and so independent of the scale of the problem. The `max` arguments help ensure the test doesn't break down in regions where x_i or $\Phi(x)$ are close to 0. If this test succeeds, `info` is set to 2 and the function returns `GSL_SUCCESS`.

A general guideline for choosing the gradient tolerance is to set `gtol = GSL_DBL_EPSILON^(1/3)`. See Dennis and Schnabel for more information.

If none of the tests succeed, `info` is set to 0 and the function returns `GSL_CONTINUE`, indicating further iterations are required.

40.9 High Level Driver

These routines provide a high level wrapper that combines the iteration and convergence testing for easy use.

int **gsl_multifit_nlinear_driver**(const size_t maxiter, const double xtol, const double gtol, const double ftol, void (*callback)(const size_t iter, void *params, const gsl_multifit_linear_workspace *w), void *callback_params, int *info, gsl_multifit_nlinear_workspace *w)

int **gsl_multilarge_nlinear_driver**(const size_t maxiter, const double xtol, const double gtol, const double ftol, void (*callback)(const size_t iter, void *params, const gsl_multilarge_linear_workspace *w), void *callback_params, int *info, gsl_multilarge_nlinear_workspace *w)

These functions iterate the nonlinear least squares solver `w` for a maximum of `maxiter` iterations. After each iteration, the system is tested for convergence with the error tolerances `xtol`, `gtol` and `ftol`. Additionally, the user may supply a callback function `callback` which is called after each iteration, so that the user may save or print relevant quantities for each iteration. The parameter `callback_params` is passed to the `callback` function. The parameters `callback` and `callback_params` may be set to `NULL` to disable this

feature. Upon successful convergence, the function returns `GSL_SUCCESS` and sets `info` to the reason for convergence (see `gsl_multifit_nlinear_test()`). If the function has not converged after `maxiter` iterations, `GSL_EMAXITER` is returned. In rare cases, during an iteration the algorithm may be unable to find a new acceptable step δ to take. In this case, `GSL_ENOPROG` is returned indicating no further progress can be made. If your problem is having difficulty converging, see Troubleshooting for further guidance.

40.10 Covariance matrix of best fit parameters

int **gsl_multifit_nlinear_covar**(const gsl_matrix *J, const double epsrel, gsl_matrix *covar)

int **gsl_multilarge_nlinear_covar**(gsl_matrix *covar, gsl_multilarge_nlinear_workspace *w)

This function computes the covariance matrix of best-fit parameters using the Jacobian matrix `J` and stores it in `covar`. The parameter `epsrel` is used to remove linear-dependent columns when `J` is rank deficient.

The covariance matrix is given by,

$$C = (J^T J)^{-1}$$

or in the weighted case,

$$C = (J^T W J)^{-1}$$

and is computed using the factored form of the Jacobian (Cholesky, QR, or SVD). Any columns of R which satisfy

$$|R_{kk}| \leq epsrel|R_{11}|$$

are considered linearly-dependent and are excluded from the covariance matrix (the corresponding rows and columns of the covariance matrix are set to zero).

If the minimisation uses the weighted least-squares function $f_i = (Y(x, t_i) - y_i)/\sigma_i$ then the covariance matrix above gives the statistical error on the best-fit parameters resulting from the Gaussian errors σ_i on the underlying data y_i. This can be verified from the relation $\delta f = J\delta c$ and the fact that the fluctuations in f from the data y_i are normalised by σ_i and so satisfy

$$\langle \delta f \delta f^T \rangle = I$$

For an unweighted least-squares function $f_i = (Y(x, t_i) - y_i)$ the covariance matrix above

should be multiplied by the variance of the residuals about the best-fit $\sigma^2 = \sum(y_i - Y(x,t_i))^2/(n-p)$ to give the variance-covariance matrix $\sigma^2 C$. This estimates the statistical error on the best-fit parameters from the scatter of the underlying data.

For more information about covariance matrices see Linear Least-Squares Overview.

40.11 Troubleshooting

When developing a code to solve a nonlinear least squares problem, here are a few considerations to keep in mind.

1. The most common difficulty is the accurate implementation of the Jacobian matrix. If the analytic Jacobian is not properly provided to the solver, this can hinder and many times prevent convergence of the method. When developing a new nonlinear least squares code, it often helps to compare the program output with the internally computed finite difference Jacobian and the user supplied analytic Jacobian. If there is a large difference in coefficients, it is likely the analytic Jacobian is incorrectly implemented.
2. If your code is having difficulty converging, the next thing to check is the starting point provided to the solver. The methods of this chapter are local methods, meaning if you provide a starting point far away from the true minimum, the method may converge to a local minimum or not converge at all. Sometimes it is possible to solve a linearized approximation to the nonlinear problem, and use the linear solution as the starting point to the nonlinear problem.
3. If the various parameters of the coefficient vector x vary widely in magnitude, then the problem is said to be badly scaled. The methods of this chapter do attempt to automatically rescale the elements of x to have roughly the same order of magnitude, but in extreme cases this could still cause problems for convergence. In these cases it is recommended for the user to scale their parameter vector x so that each parameter spans roughly the same range, say $[-1, 1]$. The solution vector can be backscaled to recover the original units of the problem.

40.12 Examples

The following example programs demonstrate the nonlinear least squares fitting capabilities.

40.12.1 Exponential Fitting Example

The following example program fits a weighted exponential model with background to experimental data, $Y = A\exp(-\lambda t) + b$. The first part of the program sets up the functions `expb_f()` and `expb_df()` to calculate the model and its Jacobian. The appropriate fitting function is given by,

$$f_i = (A\exp(-\lambda t_i) + b) - y_i$$

where we have chosen $t_i = iT/(N-1)$, where N is the number of data points fitted, so that $t_i \in [0, T]$. The Jacobian matrix J is the derivative of these functions with respect to the three parameters (A, λ, b). It is given by,

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

where $x_0 = A$, $x_1 = \lambda$ and $x_2 = b$. The i-th row of the Jacobian is therefore

$$J_{i\cdot} = \begin{pmatrix} \exp(-\lambda t_i) & -t_i A\exp(-\lambda t_i) & 1 \end{pmatrix}$$

The main part of the program sets up a Levenberg-Marquardt solver and some simulated random data. The data uses the known parameters (5.0,1.5,1.0) combined with Gaussian noise (standard deviation = 0.1) with a maximum time $T = 3$ and $N = 100$ timesteps. The initial guess for the parameters is chosen as (1.0, 1.0, 0.0). The iteration terminates when the relative change in x is smaller than 10^{-8}, or when the magnitude of the gradient falls below 10^{-8}. Here are the results of running the program:

```
iter  0: A = 1.0000, lambda = 1.0000, b = 0.0000, cond(J) =      inf, |f(x)| = 88.4448
iter  1: A = 4.5109, lambda = 2.5258, b = 1.0704, cond(J) =  26.2686, |f(x)| = 24.0646
iter  2: A = 4.8565, lambda = 1.7442, b = 1.1669, cond(J) =  23.7470, |f(x)| = 11.9797
iter  3: A = 4.9356, lambda = 1.5713, b = 1.0767, cond(J) =  17.5849, |f(x)| = 10.7355
iter  4: A = 4.8678, lambda = 1.4838, b = 1.0252, cond(J) =  16.3428, |f(x)| = 10.5000
iter  5: A = 4.8118, lambda = 1.4481, b = 1.0076, cond(J) =  15.7925, |f(x)| = 10.4786
iter  6: A = 4.7983, lambda = 1.4404, b = 1.0041, cond(J) =  15.5840, |f(x)| = 10.4778
iter  7: A = 4.7967, lambda = 1.4395, b = 1.0037, cond(J) =  15.5396, |f(x)| = 10.4778
iter  8: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) =  15.5344, |f(x)| = 10.4778
iter  9: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) =  15.5339, |f(x)| = 10.4778
iter 10: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) =  15.5339, |f(x)| = 10.4778
iter 11: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) =  15.5339, |f(x)| = 10.4778
summary from method 'trust-region/levenberg-marquardt'
number of iterations: 11
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
function evaluations: 16
Jacobian evaluations: 12
reason for stopping: small gradient
initial |f(x)| = 88.444756
final   |f(x)| = 10.477801
chisq/dof = 1.1318
A      = 4.79653 +/- 0.18704
lambda = 1.43937 +/- 0.07390
b      = 1.00368 +/- 0.03473
status = success
```

The approximate values of the parameters are found correctly, and the chi-squared value indicates a good fit (the chi-squared per degree of freedom is approximately 1). In this case the errors on the parameters can be estimated from the square roots of the diagonal elements of the covariance matrix. If the chi-squared value shows a poor fit (i.e. $\chi^2/(n-p) \gg 1$ then the error estimates obtained from the covariance matrix will be too small. In the example program the error estimates are multiplied by $\sqrt{\chi^2/(n-p)}$ in this case, a common way of increasing the errors for a poor fit. Note that a poor fit will result from the use of an inappropriate model, and the scaled error estimates may then be outside the range of validity for Gaussian errors.

Additionally, we see that the condition number of $J(x)$ stays reasonably small throughout the iteration. This indicates we could safely switch to the Cholesky solver for speed improvement, although this particular system is too small to really benefit.

그림 40.1 shows the fitted curve with the original data.

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>

#define N      100    /* number of data points to fit */
#define TMAX   (3.0)  /* time variable in [0,TMAX] */

struct data {
  size_t n;
  double * t;
```

(다음 페이지에 계속)

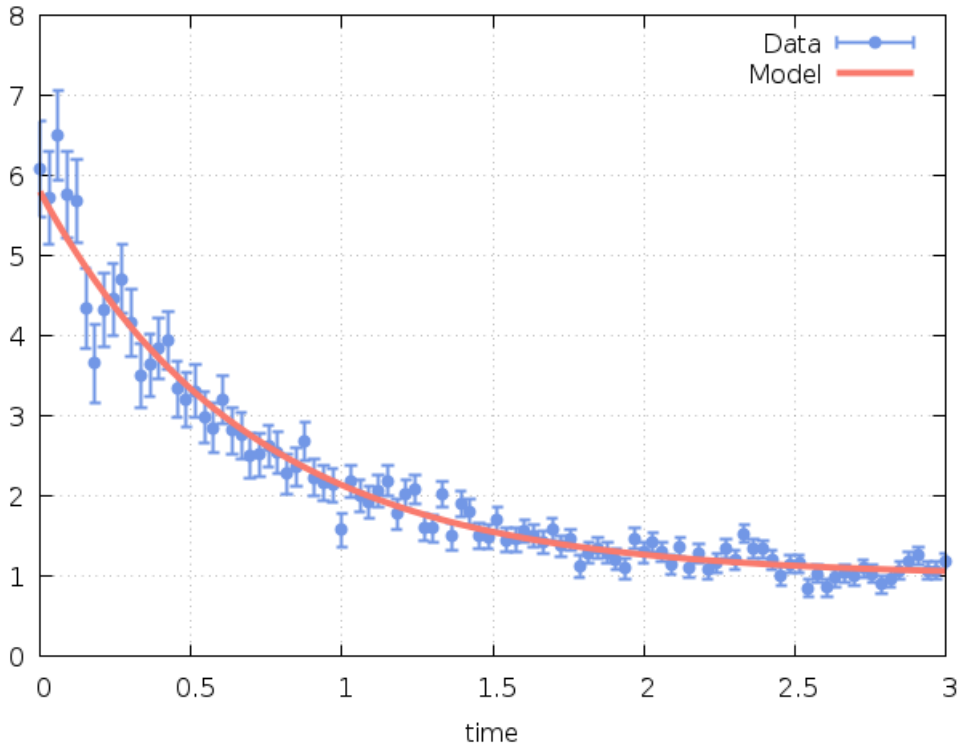


그림 40.1: Exponential fitted curve with data

(이전 페이지에서 계속)

```
  double * y;
};

int
expb_f (const gsl_vector * x, void *data,
        gsl_vector * f)
{
  size_t n = ((struct data *)data)->n;
  double *t = ((struct data *)data)->t;
  double *y = ((struct data *)data)->y;

  double A = gsl_vector_get (x, 0);
  double lambda = gsl_vector_get (x, 1);
  double b = gsl_vector_get (x, 2);

  size_t i;

  for (i = 0; i < n; i++)
    {
      /* Model Yi = A * exp(-lambda * t_i) + b */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      double Yi = A * exp (-lambda * t[i]) + b;
      gsl_vector_set (f, i, Yi - y[i]);
    }

  return GSL_SUCCESS;
}

int
expb_df (const gsl_vector * x, void *data,
         gsl_matrix * J)
{
  size_t n = ((struct data *)data)->n;
  double *t = ((struct data *)data)->t;

  double A = gsl_vector_get (x, 0);
  double lambda = gsl_vector_get (x, 1);

  size_t i;

  for (i = 0; i < n; i++)
    {
      /* Jacobian matrix J(i,j) = dfi / dxj, */
      /* where fi = (Yi - yi)/sigma[i],      */
      /*       Yi = A * exp(-lambda * t_i) + b  */
      /* and the xj are the parameters (A,lambda,b) */
      double e = exp(-lambda * t[i]);
      gsl_matrix_set (J, i, 0, e);
      gsl_matrix_set (J, i, 1, -t[i] * A * e);
      gsl_matrix_set (J, i, 2, 1.0);
    }

  return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
         const gsl_multifit_nlinear_workspace *w)
{
  gsl_vector *f = gsl_multifit_nlinear_residual(w);
  gsl_vector *x = gsl_multifit_nlinear_position(w);
  double rcond;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  /* compute reciprocal condition number of J(x) */
  gsl_multifit_nlinear_rcond(&rcond, w);

  fprintf(stderr, "iter %2zu: A = %.4f, lambda = %.4f, b = %.4f, cond(J) = %8.4f, |f(x)| = %.4f\n",
          iter,
          gsl_vector_get(x, 0),
          gsl_vector_get(x, 1),
          gsl_vector_get(x, 2),
          1.0 / rcond,
          gsl_blas_dnrm2(f));
}

int
main (void)
{
  const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
  gsl_multifit_nlinear_workspace *w;
  gsl_multifit_nlinear_fdf fdf;
  gsl_multifit_nlinear_parameters fdf_params =
    gsl_multifit_nlinear_default_parameters();
  const size_t n = N;
  const size_t p = 3;

  gsl_vector *f;
  gsl_matrix *J;
  gsl_matrix *covar = gsl_matrix_alloc (p, p);
  double t[N], y[N], weights[N];
  struct data d = { n, t, y };
  double x_init[3] = { 1.0, 1.0, 0.0 }; /* starting values */
  gsl_vector_view x = gsl_vector_view_array (x_init, p);
  gsl_vector_view wts = gsl_vector_view_array(weights, n);
  gsl_rng * r;
  double chisq, chisq0;
  int status, info;
  size_t i;

  const double xtol = 1e-8;
  const double gtol = 1e-8;
  const double ftol = 0.0;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
gsl_rng_env_setup();
r = gsl_rng_alloc(gsl_rng_default);

/* define the function to be minimized */
fdf.f = expb_f;
fdf.df = expb_df;   /* set to NULL for finite-difference Jacobian */
fdf.fvv = NULL;     /* not using geodesic acceleration */
fdf.n = n;
fdf.p = p;
fdf.params = &d;

/* this is the data to be fitted */
for (i = 0; i < n; i++)
  {
    double ti = i * TMAX / (n - 1.0);
    double yi = 1.0 + 5 * exp (-1.5 * ti);
    double si = 0.1 * yi;
    double dy = gsl_ran_gaussian(r, si);

    t[i] = ti;
    y[i] = yi + dy;
    weights[i] = 1.0 / (si * si);
    printf ("data: %g %g %g\n", ti, y[i], si);
  };

/* allocate workspace with default parameters */
w = gsl_multifit_nlinear_alloc (T, &fdf_params, n, p);

/* initialize solver with starting point and weights */
gsl_multifit_nlinear_winit (&x.vector, &wts.vector, &fdf, w);

/* compute initial cost function */
f = gsl_multifit_nlinear_residual(w);
gsl_blas_ddot(f, f, &chisq0);

/* solve the system with a maximum of 100 iterations */
status = gsl_multifit_nlinear_driver(100, xtol, gtol, ftol,
                                     callback, NULL, &info, w);

/* compute covariance of best fit parameters */
J = gsl_multifit_nlinear_jac(w);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_multifit_nlinear_covar (J, 0.0, covar);

  /* compute final cost */
  gsl_blas_ddot(f, f, &chisq);

#define FIT(i) gsl_vector_get(w->x, i)
#define ERR(i) sqrt(gsl_matrix_get(covar,i,i))

  fprintf(stderr, "summary from method '%s/%s'\n",
          gsl_multifit_nlinear_name(w),
          gsl_multifit_nlinear_trs_name(w));
  fprintf(stderr, "number of iterations: %zu\n",
          gsl_multifit_nlinear_niter(w));
  fprintf(stderr, "function evaluations: %zu\n", fdf.nevalf);
  fprintf(stderr, "Jacobian evaluations: %zu\n", fdf.nevaldf);
  fprintf(stderr, "reason for stopping: %s\n",
          (info == 1) ? "small step size" : "small gradient");
  fprintf(stderr, "initial |f(x)| = %f\n", sqrt(chisq0));
  fprintf(stderr, "final   |f(x)| = %f\n", sqrt(chisq));

  {
    double dof = n - p;
    double c = GSL_MAX_DBL(1, sqrt(chisq / dof));

    fprintf(stderr, "chisq/dof = %g\n", chisq / dof);

    fprintf (stderr, "A      = %.5f +/- %.5f\n", FIT(0), c*ERR(0));
    fprintf (stderr, "lambda = %.5f +/- %.5f\n", FIT(1), c*ERR(1));
    fprintf (stderr, "b      = %.5f +/- %.5f\n", FIT(2), c*ERR(2));
  }

  fprintf (stderr, "status = %s\n", gsl_strerror (status));

  gsl_multifit_nlinear_free (w);
  gsl_matrix_free (covar);
  gsl_rng_free (r);

  return 0;
}
```

40.12.2 Geodesic Acceleration Example 1

The following example program minimizes a modified Rosenbrock function, which is characterized by a narrow canyon with steep walls. The starting point is selected high on the canyon wall, so the solver must first find the canyon bottom and then navigate to the minimum. The problem is solved both with and without using geodesic acceleration for comparison. The cost function is given by

$$\Phi(x) = \frac{1}{2}(f_1^2 + f_2^2)$$
$$f_1 = 100\left(x_2 - x_1^2\right)$$
$$f_2 = 1 - x_1$$

The Jacobian matrix is

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} -200x_1 & 100 \\ -1 & 0 \end{pmatrix}$$

In order to use geodesic acceleration, the user must provide the second directional derivative of each residual in the velocity direction, $D_v^2 f_i = \sum_{\alpha\beta} v_\alpha v_\beta \partial_\alpha \partial_\beta f_i$. The velocity vector v is provided by the solver. For this example, these derivatives are

$$f_{vv} = D_v^2 \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} -200v_1^2 \\ 0 \end{pmatrix}$$

The solution of this minimization problem is

$$x^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
$$\Phi(x^*) = 0$$

The program output is shown below:

```
=== Solving system without acceleration ===
NITER         = 53
NFEV          = 56
NJEV          = 54
NAEV          = 0
initial cost  = 2.250225000000e+04
final cost    = 6.674986031430e-18
final x       = (9.999999974165e-01, 9.999999948328e-01)
final cond(J) = 6.000096055094e+02
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
=== Solving system with acceleration ===
NITER         = 15
NFEV          = 17
NJEV          = 16
NAEV          = 16
initial cost  = 2.250225000000e+04
final cost    = 7.518932873279e-19
final x       = (9.999999991329e-01, 9.999999982657e-01)
final cond(J) = 6.000097233278e+02
```

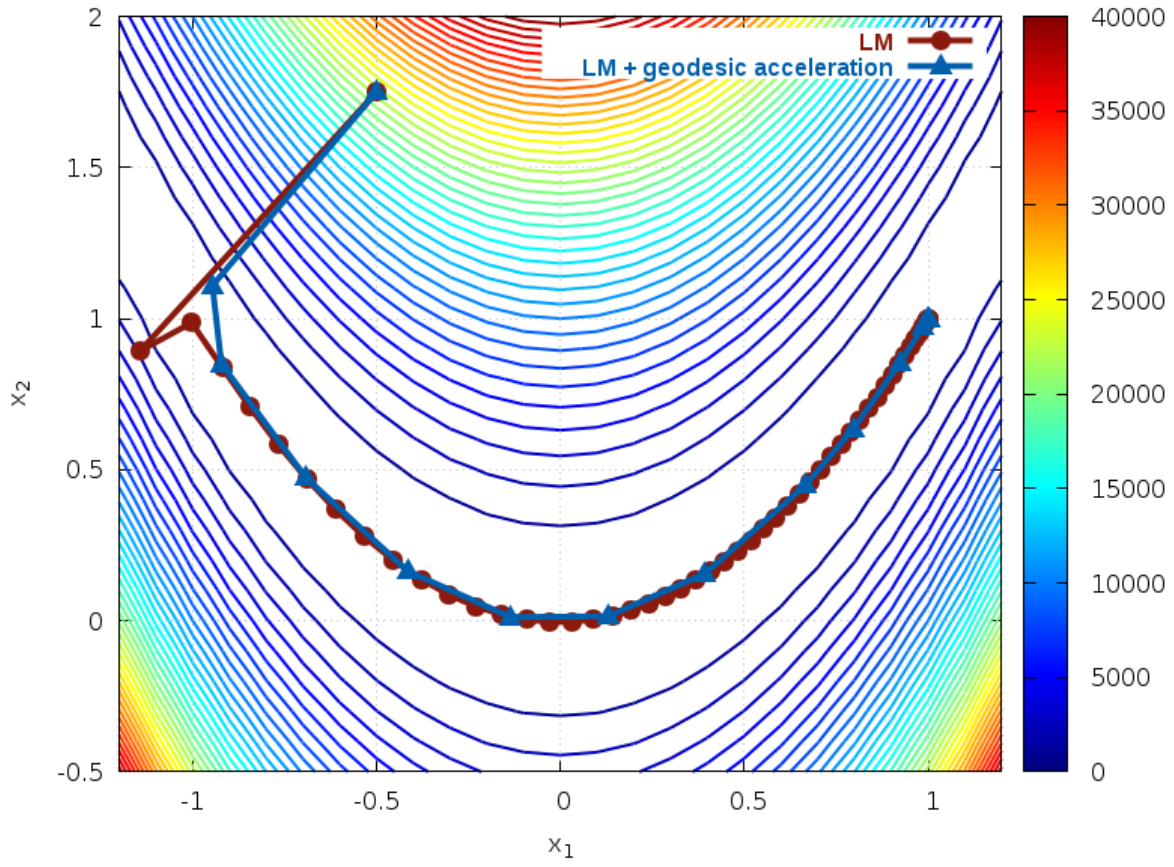


그림 40.2: Paths taken by solver for Rosenbrock function

We can see that enabling geodesic acceleration requires less than a third of the number of Jacobian evaluations in order to locate the minimum. The path taken by both methods is shown in 그림 40.2. The contours show the cost function $\Phi(x_1, x_2)$. We see that both methods quickly find the canyon bottom, but the geodesic acceleration method navigates along the bottom to the solution with significantly fewer iterations.

The program is given below.

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>

int
func_f (const gsl_vector * x, void *params, gsl_vector * f)
{
  double x1 = gsl_vector_get(x, 0);
  double x2 = gsl_vector_get(x, 1);

  gsl_vector_set(f, 0, 100.0 * (x2 - x1*x1));
  gsl_vector_set(f, 1, 1.0 - x1);

  return GSL_SUCCESS;
}

int
func_df (const gsl_vector * x, void *params, gsl_matrix * J)
{
  double x1 = gsl_vector_get(x, 0);

  gsl_matrix_set(J, 0, 0, -200.0*x1);
  gsl_matrix_set(J, 0, 1, 100.0);
  gsl_matrix_set(J, 1, 0, -1.0);
  gsl_matrix_set(J, 1, 1, 0.0);

  return GSL_SUCCESS;
}

int
func_fvv (const gsl_vector * x, const gsl_vector * v,
          void *params, gsl_vector * fvv)
{
  double v1 = gsl_vector_get(v, 0);

  gsl_vector_set(fvv, 0, -200.0 * v1 * v1);
  gsl_vector_set(fvv, 1, 0.0);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
         const gsl_multifit_nlinear_workspace *w)
{
  gsl_vector * x = gsl_multifit_nlinear_position(w);

  /* print out current location */
  printf("%f %f\n",
         gsl_vector_get(x, 0),
         gsl_vector_get(x, 1));
}

void
solve_system(gsl_vector *x0, gsl_multifit_nlinear_fdf *fdf,
             gsl_multifit_nlinear_parameters *params)
{
  const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
  const size_t max_iter = 200;
  const double xtol = 1.0e-8;
  const double gtol = 1.0e-8;
  const double ftol = 1.0e-8;
  const size_t n = fdf->n;
  const size_t p = fdf->p;
  gsl_multifit_nlinear_workspace *work =
    gsl_multifit_nlinear_alloc(T, params, n, p);
  gsl_vector * f = gsl_multifit_nlinear_residual(work);
  gsl_vector * x = gsl_multifit_nlinear_position(work);
  int info;
  double chisq0, chisq, rcond;

  /* initialize solver */
  gsl_multifit_nlinear_init(x0, fdf, work);

  /* store initial cost */
  gsl_blas_ddot(f, f, &chisq0);

  /* iterate until convergence */
  gsl_multifit_nlinear_driver(max_iter, xtol, gtol, ftol,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                           callback, NULL, &info, work);

  /* store final cost */
  gsl_blas_ddot(f, f, &chisq);

  /* store cond(J(x)) */
  gsl_multifit_nlinear_rcond(&rcond, work);

  /* print summary */

  fprintf(stderr, "NITER         = %zu\n", gsl_multifit_nlinear_niter(work));
  fprintf(stderr, "NFEV          = %zu\n", fdf->nevalf);
  fprintf(stderr, "NJEV          = %zu\n", fdf->nevaldf);
  fprintf(stderr, "NAEV          = %zu\n", fdf->nevalfvv);
  fprintf(stderr, "initial cost  = %.12e\n", chisq0);
  fprintf(stderr, "final cost    = %.12e\n", chisq);
  fprintf(stderr, "final x       = (%.12e, %.12e)\n",
          gsl_vector_get(x, 0), gsl_vector_get(x, 1));
  fprintf(stderr, "final cond(J) = %.12e\n", 1.0 / rcond);

  printf("\n\n");

  gsl_multifit_nlinear_free(work);
}

int
main (void)
{
  const size_t n = 2;
  const size_t p = 2;
  gsl_vector *f = gsl_vector_alloc(n);
  gsl_vector *x = gsl_vector_alloc(p);
  gsl_multifit_nlinear_fdf fdf;
  gsl_multifit_nlinear_parameters fdf_params =
    gsl_multifit_nlinear_default_parameters();

  /* print map of Phi(x1, x2) */
  {
    double x1, x2, chisq;
    double *f1 = gsl_vector_ptr(f, 0);
    double *f2 = gsl_vector_ptr(f, 1);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  for (x1 = -1.2; x1 < 1.3; x1 += 0.1)
    {
      for (x2 = -0.5; x2 < 2.1; x2 += 0.1)
        {
          gsl_vector_set(x, 0, x1);
          gsl_vector_set(x, 1, x2);
          func_f(x, NULL, f);

          chisq = (*f1) * (*f1) + (*f2) * (*f2);
          printf("%f %f %f\n", x1, x2, chisq);
        }
      printf("\n");
    }
  printf("\n\n");
}

/* define function to be minimized */
fdf.f = func_f;
fdf.df = func_df;
fdf.fvv = func_fvv;
fdf.n = n;
fdf.p = p;
fdf.params = NULL;

/* starting point */
gsl_vector_set(x, 0, -0.5);
gsl_vector_set(x, 1, 1.75);

fprintf(stderr, "=== Solving system without acceleration ===\n");
fdf_params.trs = gsl_multifit_nlinear_trs_lm;
solve_system(x, &fdf, &fdf_params);

fprintf(stderr, "=== Solving system with acceleration ===\n");
fdf_params.trs = gsl_multifit_nlinear_trs_lmaccel;
solve_system(x, &fdf, &fdf_params);

gsl_vector_free(f);
gsl_vector_free(x);

return 0;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
}
```

40.12.3 Geodesic Acceleration Example 2

The following example fits a set of data to a Gaussian model using the Levenberg-Marquardt method with geodesic acceleration. The cost function is

$$\Phi(x) = \frac{1}{2} \sum_i f_i^2$$
$$f_i = y_i - Y(a, b, c; t_i)$$

where y_i is the measured data point at time t_i, and the model is specified by

$$Y(a, b, c; t) = a \exp\left[-\frac{1}{2}\left(\frac{t-b}{c}\right)^2\right]$$

The parameters a, b, c represent the amplitude, mean, and width of the Gaussian respectively. The program below generates the y_i data on $[0, 1]$ using the values $a = 5$, $b = 0.4$, $c = 0.15$ and adding random noise. The i-th row of the Jacobian is

$$J_{i,:} = \begin{pmatrix} \frac{\partial f_i}{\partial a} & \frac{\partial f_i}{\partial b} & \frac{\partial f_i}{\partial c} \end{pmatrix} = \begin{pmatrix} -e_i & -\frac{a}{c} z_i e_i & -\frac{a}{c} z_i^2 e_i \end{pmatrix}$$

where

$$z_i = \frac{t_i - b}{c}$$
$$e_i = \exp\left(-\frac{1}{2} z_i^2\right)$$

In order to use geodesic acceleration, we need the second directional derivative of the residuals in the velocity direction, $D_v^2 f_i = \sum_{\alpha\beta} v_\alpha v_\beta \partial_\alpha \partial_\beta f_i$, where v is provided by the solver. To compute this, it is helpful to make a table of all second derivatives of the residuals f_i with respect to each combination of model parameters. This table is

	$\frac{\partial}{\partial a}$	$\frac{\partial}{\partial b}$	$\frac{\partial}{\partial c}$
$\frac{\partial}{\partial a}$	0	$-\frac{z_i}{c} e_i$	$-\frac{z_i^2}{c} e_i$
$\frac{\partial}{\partial b}$		$\frac{a}{c^2}\left(1 - z_i^2\right) e_i$	$\frac{a}{c^2} z_i \left(2 - z_i^2\right) e_i$
$\frac{\partial}{\partial c}$			$\frac{a}{c^2} z_i^2 \left(3 - z_i^2\right) e_i$

The lower half of the table is omitted since it is symmetric. Then, the second directional derivative of f_i is

$$D_v^2 f_i = v_a^2 \partial_a^2 f_i + 2 v_a v_b \partial_a \partial_b f_i + 2 v_a v_c \partial_a \partial_c f_i + v_b^2 \partial_b^2 f_i + 2 v_b v_c \partial_b \partial_c f_i + v_c^2 \partial_c^2 f_i$$

The factors of 2 come from the symmetry of the mixed second partial derivatives. The iteration is started using the initial guess $a = 1, b = 0, c = 1$. The program output is shown below:

```
iter  0: a = 1.0000, b = 0.0000, c = 1.0000, |a|/|v| = 0.0000 cond(J) =      inf, |f(x)| = 35.4785
iter  1: a = 1.5708, b = 0.5321, c = 0.5219, |a|/|v| = 0.3093 cond(J) =  29.0443, |f(x)| = 31.1042
iter  2: a = 1.7387, b = 0.4040, c = 0.4568, |a|/|v| = 0.1199 cond(J) =   3.5256, |f(x)| = 28.7217
iter  3: a = 2.2340, b = 0.3829, c = 0.3053, |a|/|v| = 0.3308 cond(J) =   4.5121, |f(x)| = 23.8074
iter  4: a = 3.2275, b = 0.3952, c = 0.2243, |a|/|v| = 0.2784 cond(J) =   8.6499, |f(x)| = 15.6003
iter  5: a = 4.3347, b = 0.3974, c = 0.1752, |a|/|v| = 0.2029 cond(J) =  15.1732, |f(x)| = 7.5908
iter  6: a = 4.9352, b = 0.3992, c = 0.1536, |a|/|v| = 0.1001 cond(J) =  26.6621, |f(x)| = 4.8402
iter  7: a = 5.0716, b = 0.3994, c = 0.1498, |a|/|v| = 0.0166 cond(J) =  34.6922, |f(x)| = 4.7103
iter  8: a = 5.0828, b = 0.3994, c = 0.1495, |a|/|v| = 0.0012 cond(J) =  36.5422, |f(x)| = 4.7095
iter  9: a = 5.0831, b = 0.3994, c = 0.1495, |a|/|v| = 0.0000 cond(J) =  36.6929, |f(x)| = 4.7095
iter 10: a = 5.0831, b = 0.3994, c = 0.1495, |a|/|v| = 0.0000 cond(J) =  36.6975, |f(x)| = 4.7095
iter 11: a = 5.0831, b = 0.3994, c = 0.1495, |a|/|v| = 0.0000 cond(J) =  36.6976, |f(x)| = 4.7095
NITER         = 11
NFEV          = 18
NJEV          = 12
NAEV          = 17
initial cost  = 1.258724737288e+03
final cost    = 2.217977560180e+01
final x       = (5.083101559156e+00, 3.994484109594e-01, 1.494898e-01)
final cond(J) = 3.669757713403e+01
```

We see the method converges after 11 iterations. For comparison the standard Levenberg-Marquardt method requires 26 iterations and so the Gaussian fitting problem benefits substantially from the geodesic acceleration correction. The column marked |a|/|v| above shows the ratio of the acceleration term to the velocity term as the iteration progresses. Larger values of this ratio indicate that the geodesic acceleration correction term is contributing substantial information to the solver relative to the standard LM velocity step.

The data and fitted model are shown in 그림 40.3.

The program is given below.

```
#include <stdlib.h>
#include <stdio.h>
```

(다음 페이지에 계속)

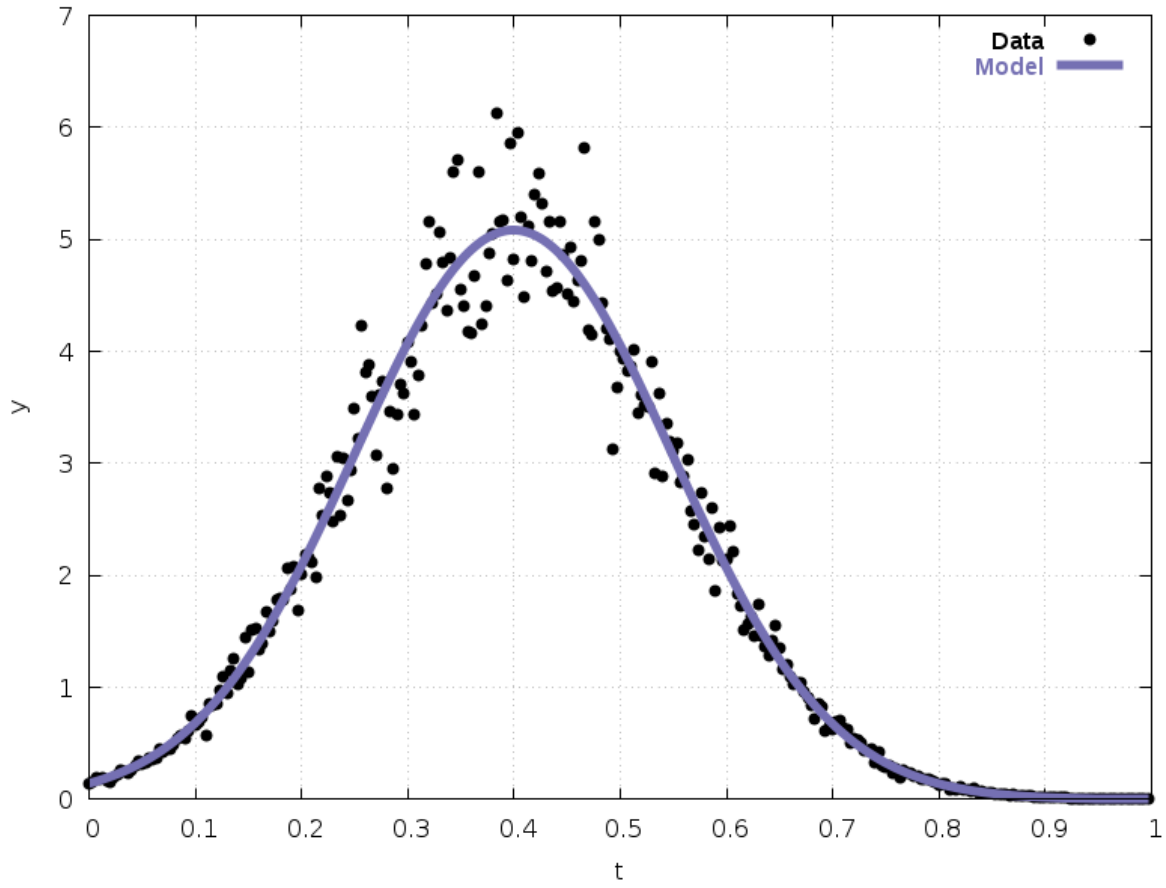


그림 40.3: Gaussian model fitted to data

(이전 페이지에서 계속)

```
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

struct data
{
  double *t;
  double *y;
  size_t n;
};

/* model function: a * exp( -1/2 * [ (t - b) / c ]^2 ) */
double
gaussian(const double a, const double b, const double c, const double t)
{
  const double z = (t - b) / c;
  return (a * exp(-0.5 * z * z));
}

int
func_f (const gsl_vector * x, void *params, gsl_vector * f)
{
  struct data *d = (struct data *) params;
  double a = gsl_vector_get(x, 0);
  double b = gsl_vector_get(x, 1);
  double c = gsl_vector_get(x, 2);
  size_t i;

  for (i = 0; i < d->n; ++i)
    {
      double ti = d->t[i];
      double yi = d->y[i];
      double y = gaussian(a, b, c, ti);

      gsl_vector_set(f, i, yi - y);
    }

  return GSL_SUCCESS;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
}

int
func_df (const gsl_vector * x, void *params, gsl_matrix * J)
{
  struct data *d = (struct data *) params;
  double a = gsl_vector_get(x, 0);
  double b = gsl_vector_get(x, 1);
  double c = gsl_vector_get(x, 2);
  size_t i;

  for (i = 0; i < d->n; ++i)
    {
      double ti = d->t[i];
      double zi = (ti - b) / c;
      double ei = exp(-0.5 * zi * zi);

      gsl_matrix_set(J, i, 0, -ei);
      gsl_matrix_set(J, i, 1, -(a / c) * ei * zi);
      gsl_matrix_set(J, i, 2, -(a / c) * ei * zi * zi);
    }

  return GSL_SUCCESS;
}

int
func_fvv (const gsl_vector * x, const gsl_vector * v,
          void *params, gsl_vector * fvv)
{
  struct data *d = (struct data *) params;
  double a = gsl_vector_get(x, 0);
  double b = gsl_vector_get(x, 1);
  double c = gsl_vector_get(x, 2);
  double va = gsl_vector_get(v, 0);
  double vb = gsl_vector_get(v, 1);
  double vc = gsl_vector_get(v, 2);
  size_t i;

  for (i = 0; i < d->n; ++i)
    {
      double ti = d->t[i];
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      double zi = (ti - b) / c;
      double ei = exp(-0.5 * zi * zi);
      double Dab = -zi * ei / c;
      double Dac = -zi * zi * ei / c;
      double Dbb = a * ei / (c * c) * (1.0 - zi*zi);
      double Dbc = a * zi * ei / (c * c) * (2.0 - zi*zi);
      double Dcc = a * zi * zi * ei / (c * c) * (3.0 - zi*zi);
      double sum;

      sum = 2.0 * va * vb * Dab +
            2.0 * va * vc * Dac +
                  vb * vb * Dbb +
            2.0 * vb * vc * Dbc +
                  vc * vc * Dcc;

      gsl_vector_set(fvv, i, sum);
    }

  return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
         const gsl_multifit_nlinear_workspace *w)
{
  gsl_vector *f = gsl_multifit_nlinear_residual(w);
  gsl_vector *x = gsl_multifit_nlinear_position(w);
  double avratio = gsl_multifit_nlinear_avratio(w);
  double rcond;

  (void) params; /* not used */

  /* compute reciprocal condition number of J(x) */
  gsl_multifit_nlinear_rcond(&rcond, w);

  fprintf(stderr, "iter %2zu: a = %.4f, b = %.4f, c = %.4f, |a|/|v| = %.4f cond(J) = %8.4f, |f(x)|
↪= %.4f\n",
          iter,
          gsl_vector_get(x, 0),
          gsl_vector_get(x, 1),
          gsl_vector_get(x, 2),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
          avratio,
          1.0 / rcond,
          gsl_blas_dnrm2(f));
}

void
solve_system(gsl_vector *x, gsl_multifit_nlinear_fdf *fdf,
             gsl_multifit_nlinear_parameters *params)
{
  const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
  const size_t max_iter = 200;
  const double xtol = 1.0e-8;
  const double gtol = 1.0e-8;
  const double ftol = 1.0e-8;
  const size_t n = fdf->n;
  const size_t p = fdf->p;
  gsl_multifit_nlinear_workspace *work =
    gsl_multifit_nlinear_alloc(T, params, n, p);
  gsl_vector * f = gsl_multifit_nlinear_residual(work);
  gsl_vector * y = gsl_multifit_nlinear_position(work);
  int info;
  double chisq0, chisq, rcond;

  /* initialize solver */
  gsl_multifit_nlinear_init(x, fdf, work);

  /* store initial cost */
  gsl_blas_ddot(f, f, &chisq0);

  /* iterate until convergence */
  gsl_multifit_nlinear_driver(max_iter, xtol, gtol, ftol,
                              callback, NULL, &info, work);

  /* store final cost */
  gsl_blas_ddot(f, f, &chisq);

  /* store cond(J(x)) */
  gsl_multifit_nlinear_rcond(&rcond, work);

  gsl_vector_memcpy(x, y);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  /* print summary */

  fprintf(stderr, "NITER         = %zu\n", gsl_multifit_nlinear_niter(work));
  fprintf(stderr, "NFEV          = %zu\n", fdf->nevalf);
  fprintf(stderr, "NJEV          = %zu\n", fdf->nevaldf);
  fprintf(stderr, "NAEV          = %zu\n", fdf->nevalfvv);
  fprintf(stderr, "initial cost  = %.12e\n", chisq0);
  fprintf(stderr, "final cost    = %.12e\n", chisq);
  fprintf(stderr, "final x       = (%.12e, %.12e, %12e)\n",
          gsl_vector_get(x, 0), gsl_vector_get(x, 1), gsl_vector_get(x, 2));
  fprintf(stderr, "final cond(J) = %.12e\n", 1.0 / rcond);

  gsl_multifit_nlinear_free(work);
}

int
main (void)
{
  const size_t n = 300;  /* number of data points to fit */
  const size_t p = 3;    /* number of model parameters */
  const double a = 5.0;  /* amplitude */
  const double b = 0.4;  /* center */
  const double c = 0.15; /* width */
  const gsl_rng_type * T = gsl_rng_default;
  gsl_vector *f = gsl_vector_alloc(n);
  gsl_vector *x = gsl_vector_alloc(p);
  gsl_multifit_nlinear_fdf fdf;
  gsl_multifit_nlinear_parameters fdf_params =
    gsl_multifit_nlinear_default_parameters();
  struct data fit_data;
  gsl_rng * r;
  size_t i;

  gsl_rng_env_setup ();
  r = gsl_rng_alloc (T);

  fit_data.t = malloc(n * sizeof(double));
  fit_data.y = malloc(n * sizeof(double));
  fit_data.n = n;

  /* generate synthetic data with noise */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  for (i = 0; i < n; ++i)
    {
      double t = (double)i / (double) n;
      double y0 = gaussian(a, b, c, t);
      double dy = gsl_ran_gaussian (r, 0.1 * y0);

      fit_data.t[i] = t;
      fit_data.y[i] = y0 + dy;
    }

  /* define function to be minimized */
  fdf.f = func_f;
  fdf.df = func_df;
  fdf.fvv = func_fvv;
  fdf.n = n;
  fdf.p = p;
  fdf.params = &fit_data;

  /* starting point */
  gsl_vector_set(x, 0, 1.0);
  gsl_vector_set(x, 1, 0.0);
  gsl_vector_set(x, 2, 1.0);

  fdf_params.trs = gsl_multifit_nlinear_trs_lmaccel;
  solve_system(x, &fdf, &fdf_params);

  /* print data and model */
  {
    double A = gsl_vector_get(x, 0);
    double B = gsl_vector_get(x, 1);
    double C = gsl_vector_get(x, 2);

    for (i = 0; i < n; ++i)
      {
        double ti = fit_data.t[i];
        double yi = fit_data.y[i];
        double fi = gaussian(A, B, C, ti);

        printf("%f %f %f\n", ti, yi, fi);
      }
  }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_vector_free(f);
  gsl_vector_free(x);
  gsl_rng_free(r);

  return 0;
}
```

40.12.4 Comparing TRS Methods Example

The following program compares all available nonlinear least squares trust-region subproblem (TRS) methods on the Branin function, a common optimization test problem. The cost function is

$$\Phi(x) = \frac{1}{2}(f_1^2 + f_2^2)$$
$$f_1 = x_2 + a_1 x_1^2 + a_2 x_1 + a_3$$
$$f_2 = \sqrt{a_4}\sqrt{1 + (1 - a_5)\cos x_1}$$

with $a_1 = -\frac{5.1}{4\pi^2}, a_2 = \frac{5}{\pi}, a_3 = -6, a_4 = 10, a_5 = \frac{1}{8\pi}$. There are three minima of this function in the range $(x_1, x_2) \in [-5, 15] \times [-5, 15]$. The program below uses the starting point $(x_1, x_2) = (6, 14.5)$ and calculates the solution with all available nonlinear least squares TRS methods. The program output is shown below:

```
Method                NITER  NFEV  NJEV  Initial Cost  Final cost   Final cond(J) Final x
levenberg-marquardt   20     27    21    1.9874e+02    3.9789e-01   6.1399e+07    (-3.14e+00, 1.
↪23e+01)
levenberg-marquardt+accel 27 36    28    1.9874e+02    3.9789e-01   1.4465e+07    (3.14e+00, 2.
↪27e+00)
dogleg                23     64    23    1.9874e+02    3.9789e-01   5.0692e+08    (3.14e+00, 2.
↪28e+00)
double-dogleg         24     69    24    1.9874e+02    3.9789e-01   3.4879e+07    (3.14e+00, 2.
↪27e+00)
2D-subspace           23     54    24    1.9874e+02    3.9789e-01   2.5142e+07    (3.14e+00, 2.
↪27e+00)
```

The first row of output above corresponds to standard Levenberg-Marquardt, while the second row includes geodesic acceleration. We see that the standard LM method converges to the minimum at $(-\pi, 12.275)$ and also uses the least number of iterations and Jacobian evaluations. All other methods converge to the minimum $(\pi, 2.275)$ and perform similarly in terms of

number of Jacobian evaluations. We see that J is fairly ill-conditioned at both minima, indicating that the QR (or SVD) solver is the best choice for this problem. Since there are only two parameters in this optimization problem, we can easily visualize the paths taken by each method, which are shown in 그림 40.4. The figure shows contours of the cost function $\Phi(x_1, x_2)$ which exhibits three global minima in the range $[-5, 15] \times [-5, 15]$. The paths taken by each solver are shown as colored lines.

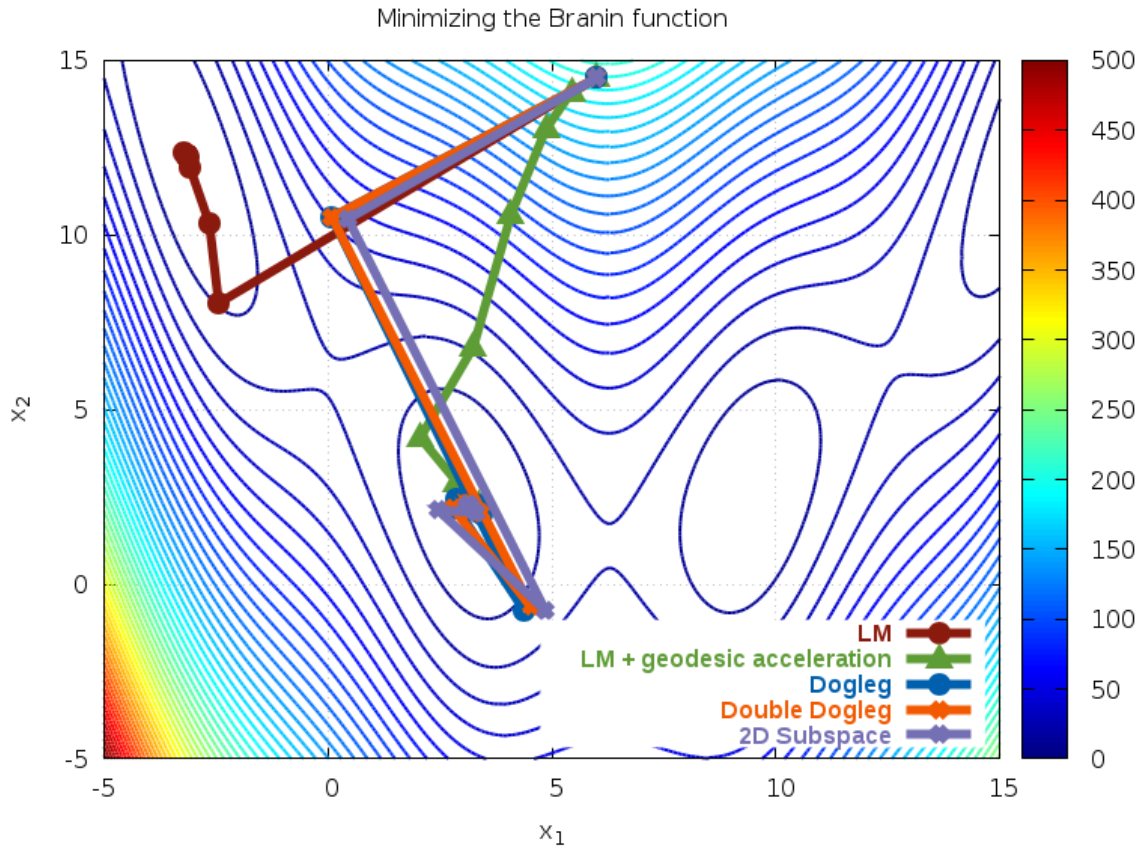


그림 40.4: Paths taken for different TRS methods for the Branin function

The program is given below.

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>

/* parameters to model */
struct model_params
{
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  double a1;
  double a2;
  double a3;
  double a4;
  double a5;
};

/* Branin function */
int
func_f (const gsl_vector * x, void *params, gsl_vector * f)
{
  struct model_params *par = (struct model_params *) params;
  double x1 = gsl_vector_get(x, 0);
  double x2 = gsl_vector_get(x, 1);
  double f1 = x2 + par->a1 * x1 * x1 + par->a2 * x1 + par->a3;
  double f2 = sqrt(par->a4) * sqrt(1.0 + (1.0 - par->a5) * cos(x1));

  gsl_vector_set(f, 0, f1);
  gsl_vector_set(f, 1, f2);

  return GSL_SUCCESS;
}

int
func_df (const gsl_vector * x, void *params, gsl_matrix * J)
{
  struct model_params *par = (struct model_params *) params;
  double x1 = gsl_vector_get(x, 0);
  double f2 = sqrt(par->a4) * sqrt(1.0 + (1.0 - par->a5) * cos(x1));

  gsl_matrix_set(J, 0, 0, 2.0 * par->a1 * x1 + par->a2);
  gsl_matrix_set(J, 0, 1, 1.0);

  gsl_matrix_set(J, 1, 0, -0.5 * par->a4 / f2 * (1.0 - par->a5) * sin(x1));
  gsl_matrix_set(J, 1, 1, 0.0);

  return GSL_SUCCESS;
}

int
func_fvv (const gsl_vector * x, const gsl_vector * v,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
          void *params, gsl_vector * fvv)
{
  struct model_params *par = (struct model_params *) params;
  double x1 = gsl_vector_get(x, 0);
  double v1 = gsl_vector_get(v, 0);
  double c = cos(x1);
  double s = sin(x1);
  double f2 = sqrt(par->a4) * sqrt(1.0 + (1.0 - par->a5) * c);
  double t = 0.5 * par->a4 * (1.0 - par->a5) / f2;

  gsl_vector_set(fvv, 0, 2.0 * par->a1 * v1 * v1);
  gsl_vector_set(fvv, 1, -t * (c + s*s/f2) * v1 * v1);

  return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
         const gsl_multifit_nlinear_workspace *w)
{
  gsl_vector * x = gsl_multifit_nlinear_position(w);
  double x1 = gsl_vector_get(x, 0);
  double x2 = gsl_vector_get(x, 1);

  /* print out current location */
  printf("%f %f\n", x1, x2);
}

void
solve_system(gsl_vector *x0, gsl_multifit_nlinear_fdf *fdf,
             gsl_multifit_nlinear_parameters *params)
{
  const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
  const size_t max_iter = 200;
  const double xtol = 1.0e-8;
  const double gtol = 1.0e-8;
  const double ftol = 1.0e-8;
  const size_t n = fdf->n;
  const size_t p = fdf->p;
  gsl_multifit_nlinear_workspace *work =
    gsl_multifit_nlinear_alloc(T, params, n, p);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_vector * f = gsl_multifit_nlinear_residual(work);
  gsl_vector * x = gsl_multifit_nlinear_position(work);
  int info;
  double chisq0, chisq, rcond;

  printf("# %s/%s\n",
         gsl_multifit_nlinear_name(work),
         gsl_multifit_nlinear_trs_name(work));

  /* initialize solver */
  gsl_multifit_nlinear_init(x0, fdf, work);

  /* store initial cost */
  gsl_blas_ddot(f, f, &chisq0);

  /* iterate until convergence */
  gsl_multifit_nlinear_driver(max_iter, xtol, gtol, ftol,
                              callback, NULL, &info, work);

  /* store final cost */
  gsl_blas_ddot(f, f, &chisq);

  /* store cond(J(x)) */
  gsl_multifit_nlinear_rcond(&rcond, work);

  /* print summary */
  fprintf(stderr, "%-25s %-6zu %-5zu %-5zu %-13.4e %-12.4e %-13.4e (%.2e, %.2e)\n",
          gsl_multifit_nlinear_trs_name(work),
          gsl_multifit_nlinear_niter(work),
          fdf->nevalf,
          fdf->nevaldf,
          chisq0,
          chisq,
          1.0 / rcond,
          gsl_vector_get(x, 0),
          gsl_vector_get(x, 1));

  printf("\n\n");

  gsl_multifit_nlinear_free(work);
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
int
main (void)
{
  const size_t n = 2;
  const size_t p = 2;
  gsl_vector *f = gsl_vector_alloc(n);
  gsl_vector *x = gsl_vector_alloc(p);
  gsl_multifit_nlinear_fdf fdf;
  gsl_multifit_nlinear_parameters fdf_params =
    gsl_multifit_nlinear_default_parameters();
  struct model_params params;

  params.a1 = -5.1 / (4.0 * M_PI * M_PI);
  params.a2 = 5.0 / M_PI;
  params.a3 = -6.0;
  params.a4 = 10.0;
  params.a5 = 1.0 / (8.0 * M_PI);

  /* print map of Phi(x1, x2) */
  {
    double x1, x2, chisq;

    for (x1 = -5.0; x1 < 15.0; x1 += 0.1)
      {
        for (x2 = -5.0; x2 < 15.0; x2 += 0.1)
          {
            gsl_vector_set(x, 0, x1);
            gsl_vector_set(x, 1, x2);
            func_f(x, &params, f);

            gsl_blas_ddot(f, f, &chisq);

            printf("%f %f %f\n", x1, x2, chisq);
          }
        printf("\n");
      }
    printf("\n\n");
  }

  /* define function to be minimized */
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  fdf.f = func_f;
  fdf.df = func_df;
  fdf.fvv = func_fvv;
  fdf.n = n;
  fdf.p = p;
  fdf.params = &params;

  /* starting point */
  gsl_vector_set(x, 0, 6.0);
  gsl_vector_set(x, 1, 14.5);

  fprintf(stderr, "%-25s %-6s %-5s %-5s %-13s %-12s %-13s %-15s\n",
          "Method", "NITER", "NFEV", "NJEV", "Initial Cost",
          "Final cost", "Final cond(J)", "Final x");

  fdf_params.trs = gsl_multifit_nlinear_trs_lm;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multifit_nlinear_trs_lmaccel;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multifit_nlinear_trs_dogleg;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multifit_nlinear_trs_ddogleg;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multifit_nlinear_trs_subspace2D;
  solve_system(x, &fdf, &fdf_params);

  gsl_vector_free(f);
  gsl_vector_free(x);

  return 0;
}
```

40.12.5 Large Nonlinear Least Squares Example

The following program illustrates the large nonlinear least squares solvers on a system with significant sparse structure in the Jacobian. The cost function is

$$\Phi(x) = \frac{1}{2}\sum_{i=1}^{p+1} f_i^2$$
$$f_i = \sqrt{\alpha}(x_i - 1), \quad 1 \le i \le p$$
$$f_{p+1} = ||x||^2 - \frac{1}{4}$$

with $\alpha = 10^{-5}$. The residual f_{p+1} imposes a constraint on the p parameters x, to ensure that $||x||^2 \approx \frac{1}{4}$. The $(p+1)$-by-p Jacobian for this system is

$$J(x) = \begin{pmatrix} \sqrt{\alpha} I_p \\ 2x^T \end{pmatrix}$$

and the normal equations matrix is

$$J^T J = \alpha I_p + 4xx^T$$

Finally, the second directional derivative of f for the geodesic acceleration method is

$$f_{vv} = D_v^2 f = \begin{pmatrix} 0 \\ 2||v||^2 \end{pmatrix}$$

Since the upper p-by-p block of J is diagonal, this sparse structure should be exploited in the nonlinear solver. For comparison, the following program solves the system for $p = 2000$ using the dense direct Cholesky solver based on the normal equations matrix $J^T J$, as well as the iterative Steihaug-Toint solver, based on sparse matrix-vector products Ju and $J^T u$. The program output is shown below:

```
Method                NITER NFEV NJUEV NJTJEV NAEV Init Cost  Final cost cond(J) Final |x|^2␣
↪Time (s)
levenberg-marquardt     25    31    26     26     0    7.1218e+18 1.9555e-02 447.50  2.5044e-01  46.
↪28
levenberg-marquardt+accel 22    23    45     23     22   7.1218e+18 1.9555e-02 447.64  2.5044e-01  33.
↪92
dogleg                  37    87    36     36     0    7.1218e+18 1.9555e-02 447.59  2.5044e-01  56.
↪05
double-dogleg           35    88    34     34     0    7.1218e+18 1.9555e-02 447.62  2.5044e-01  52.
↪65
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
2D-subspace              37    88    36    36     0    7.1218e+18 1.9555e-02 447.71  2.5044e-01  59.
↪75
steihaug-toint           35    88    345   0      0    7.1218e+18 1.9555e-02 inf     2.5044e-01  0.
↪09
```

The first five rows use methods based on factoring the dense $J^T J$ matrix while the last row uses the iterative Steihaug-Toint method. While the number of Jacobian matrix-vector products (NJUEV) is less for the dense methods, the added time to construct and factor the $J^T J$ matrix (NJTJEV) results in a much larger runtime than the iterative method (see last column).

The program is given below.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multilarge_nlinear.h>
#include <gsl/gsl_spblas.h>
#include <gsl/gsl_spmatrix.h>

/* parameters for functions */
struct model_params
{
  double alpha;
  gsl_spmatrix *J;
};

/* penalty function */
int
penalty_f (const gsl_vector * x, void *params, gsl_vector * f)
{
  struct model_params *par = (struct model_params *) params;
  const double sqrt_alpha = sqrt(par->alpha);
  const size_t p = x->size;
  size_t i;
  double sum = 0.0;

  for (i = 0; i < p; ++i)
    {
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      double xi = gsl_vector_get(x, i);

      gsl_vector_set(f, i, sqrt_alpha*(xi - 1.0));

      sum += xi * xi;
    }

  gsl_vector_set(f, p, sum - 0.25);

  return GSL_SUCCESS;
}

int
penalty_df (CBLAS_TRANSPOSE_t TransJ, const gsl_vector * x,
            const gsl_vector * u, void * params, gsl_vector * v,
            gsl_matrix * JTJ)
{
  struct model_params *par = (struct model_params *) params;
  const size_t p = x->size;
  size_t j;

  /* store 2*x in last row of J */
  for (j = 0; j < p; ++j)
    {
      double xj = gsl_vector_get(x, j);
      gsl_spmatrix_set(par->J, p, j, 2.0 * xj);
    }

  /* compute v = op(J) u */
  if (v)
    gsl_spblas_dgemv(TransJ, 1.0, par->J, u, 0.0, v);

  if (JTJ)
    {
      gsl_vector_view diag = gsl_matrix_diagonal(JTJ);

      /* compute J^T J = [ alpha*I_p + 4 x x^T ] */
      gsl_matrix_set_zero(JTJ);

      /* store 4 x x^T in lower half of JTJ */
      gsl_blas_dsyr(CblasLower, 4.0, x, JTJ);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      /* add alpha to diag(JTJ) */
      gsl_vector_add_constant(&diag.vector, par->alpha);
    }

  return GSL_SUCCESS;
}

int
penalty_fvv (const gsl_vector * x, const gsl_vector * v,
             void *params, gsl_vector * fvv)
{
  const size_t p = x->size;
  double normv = gsl_blas_dnrm2(v);

  gsl_vector_set_zero(fvv);
  gsl_vector_set(fvv, p, 2.0 * normv * normv);

  (void)params; /* avoid unused parameter warning */

  return GSL_SUCCESS;
}

void
solve_system(const gsl_vector *x0, gsl_multilarge_nlinear_fdf *fdf,
             gsl_multilarge_nlinear_parameters *params)
{
  const gsl_multilarge_nlinear_type *T = gsl_multilarge_nlinear_trust;
  const size_t max_iter = 200;
  const double xtol = 1.0e-8;
  const double gtol = 1.0e-8;
  const double ftol = 1.0e-8;
  const size_t n = fdf->n;
  const size_t p = fdf->p;
  gsl_multilarge_nlinear_workspace *work =
    gsl_multilarge_nlinear_alloc(T, params, n, p);
  gsl_vector * f = gsl_multilarge_nlinear_residual(work);
  gsl_vector * x = gsl_multilarge_nlinear_position(work);
  int info;
  double chisq0, chisq, rcond, xsq;
  struct timeval tv0, tv1;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gettimeofday(&tv0, NULL);

  /* initialize solver */
  gsl_multilarge_nlinear_init(x0, fdf, work);

  /* store initial cost */
  gsl_blas_ddot(f, f, &chisq0);

  /* iterate until convergence */
  gsl_multilarge_nlinear_driver(max_iter, xtol, gtol, ftol,
                                NULL, NULL, &info, work);

  gettimeofday(&tv1, NULL);

  /* store final cost */
  gsl_blas_ddot(f, f, &chisq);

  /* compute final ||x||^2 */
  gsl_blas_ddot(x, x, &xsq);

  /* store cond(J(x)) */
  gsl_multilarge_nlinear_rcond(&rcond, work);

  /* print summary */
  fprintf(stderr, "%-25s %-5zu %-4zu %-5zu %-6zu %-4zu %-10.4e %-10.4e %-7.2f %-11.4e %.2f\n",
          gsl_multilarge_nlinear_trs_name(work),
          gsl_multilarge_nlinear_niter(work),
          fdf->nevalf,
          fdf->nevaldfu,
          fdf->nevaldf2,
          fdf->nevalfvv,
          chisq0,
          chisq,
          1.0 / rcond,
          xsq,
          (tv1.tv_sec - tv0.tv_sec) + 1.0e-6 * (tv1.tv_usec - tv0.tv_usec));

  gsl_multilarge_nlinear_free(work);
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
int
main (void)
{
  const size_t p = 2000;
  const size_t n = p + 1;
  gsl_vector *f = gsl_vector_alloc(n);
  gsl_vector *x = gsl_vector_alloc(p);

  /* allocate sparse Jacobian matrix with 2*p non-zero elements in triplet format */
  gsl_spmatrix *J = gsl_spmatrix_alloc_nzmax(n, p, 2 * p, GSL_SPMATRIX_TRIPLET);

  gsl_multilarge_nlinear_fdf fdf;
  gsl_multilarge_nlinear_parameters fdf_params =
    gsl_multilarge_nlinear_default_parameters();
  struct model_params params;
  size_t i;

  params.alpha = 1.0e-5;
  params.J = J;

  /* define function to be minimized */
  fdf.f = penalty_f;
  fdf.df = penalty_df;
  fdf.fvv = penalty_fvv;
  fdf.n = n;
  fdf.p = p;
  fdf.params = &params;

  for (i = 0; i < p; ++i)
    {
      /* starting point */
      gsl_vector_set(x, i, i + 1.0);

      /* store sqrt(alpha)*I_p in upper p-by-p block of J */
      gsl_spmatrix_set(J, i, i, sqrt(params.alpha));
    }

  fprintf(stderr, "%-25s %-4s %-4s %-5s %-6s %-4s %-10s %-10s %-7s %-11s %-10s\n",
          "Method", "NITER", "NFEV", "NJUEV", "NJTJEV", "NAEV", "Init Cost",
          "Final cost", "cond(J)", "Final |x|^2", "Time (s)");
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  fdf_params.scale = gsl_multilarge_nlinear_scale_levenberg;

  fdf_params.trs = gsl_multilarge_nlinear_trs_lm;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multilarge_nlinear_trs_lmaccel;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multilarge_nlinear_trs_dogleg;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multilarge_nlinear_trs_ddogleg;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multilarge_nlinear_trs_subspace2D;
  solve_system(x, &fdf, &fdf_params);

  fdf_params.trs = gsl_multilarge_nlinear_trs_cgst;
  solve_system(x, &fdf, &fdf_params);

  gsl_vector_free(f);
  gsl_vector_free(x);
  gsl_spmatrix_free(J);

  return 0;
}
```

40.13 References and Further Reading

The following publications are relevant to the algorithms described in this section,

- J.J. Moré, The Levenberg-Marquardt Algorithm: Implementation and Theory, Lecture Notes in Mathematics, v630 (1978), ed G. Watson.
- H. B. Nielsen, “Damping Parameter in Marquardt’s Method”, IMM Department of Mathematical Modeling, DTU, Tech. Report IMM-REP-1999-05 (1999).
- K. Madsen and H. B. Nielsen, “Introduction to Optimization and Data Fitting”, IMM Department of Mathematical Modeling, DTU, 2010.
- J. E. Dennis and R. B. Schnabel, Numerical Methods for Unconstrained Optimization and

Nonlinear Equations, SIAM, 1996.

- M. K. Transtrum, B. B. Machta, and J. P. Sethna, Geometry of nonlinear least squares with applications to sloppy models and optimization, Phys. Rev. E 83, 036701, 2011.
- M. K. Transtrum and J. P. Sethna, Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization, arXiv:1201.5885, 2012.
- J.J. Moré, B.S. Garbow, K.E. Hillstrom, "Testing Unconstrained Optimization Software", ACM Transactions on Mathematical Software, Vol 7, No 1 (1981), p 17–41.
- H. B. Nielsen, "UCTP Test Problems for Unconstrained Optimization", IMM Department of Mathematical Modeling, DTU, Tech. Report IMM-REP-2000-17 (2000).

제 41 장

B-스플라인

참고: 번역중

This chapter describes functions for the computation of smoothing basis splines (B-splines). A smoothing spline differs from an interpolating spline in that the resulting curve is not required to pass through each datapoint. For information about interpolating splines, see 보간법.

The header file `gsl_bspline.h` contains the prototypes for the bspline functions and related declarations.

41.1 Overview

B-splines are commonly used as basis functions to fit smoothing curves to large data sets. To do this, the abscissa axis is broken up into some number of intervals, where the endpoints of each interval are called breakpoints. These breakpoints are then converted to knots by imposing various continuity and smoothness conditions at each interface. Given a nondecreasing knot vector

$$t = \{t_0, t_1, \ldots, t_{n+k-1}\}$$

the n basis splines of order k are defined by

$$B_{i,1}(x) = \begin{cases} 1, & t_i \le x < t_{i+1} \\ 0, & else \end{cases}$$
$$B_{i,k}(x) = \frac{(x - t_i)}{(t_{i+k-1} - t_i)} B_{i,k-1}(x) + \frac{(t_{i+k} - x)}{(t_{i+k} - t_{i+1})} B_{i+1,k-1}(x)$$

for $i = 0, \ldots, n-1$. The common case of cubic B-splines is given by $k = 4$. The above recurrence relation can be evaluated in a numerically stable way by the de Boor algorithm.

If we define appropriate knots on an interval $[a, b]$ then the B-spline basis functions form a complete set on that interval. Therefore we can expand a smoothing function as

$$f(x) = \sum_{i=0}^{n-1} c_i B_{i,k}(x)$$

given enough $(x_j, f(x_j))$ data pairs. The coefficients c_i can be readily obtained from a least-squares fit.

41.2 Initializing the B-splines solver

type **gsl_bspline_workspace**

The computation of B-spline functions requires a preallocated workspace.

gsl_bspline_workspace ***gsl_bspline_alloc**(const size_t k, const size_t nbreak)

This function allocates a workspace for computing B-splines of order `k`. The number of breakpoints is given by `nbreak`. This leads to $n = nbreak + k - 2$ basis functions. Cubic B-splines are specified by $k = 4$. The size of the workspace is $O(2k^2 + 5k + nbreak)$.

void **gsl_bspline_free**(gsl_bspline_workspace *w)

This function frees the memory associated with the workspace `w`.

41.3 Constructing the knots vector

int **gsl_bspline_knots**(const gsl_vector *breakpts, gsl_bspline_workspace *w)

This function computes the knots associated with the given breakpoints and stores them internally in `w->knots`.

int **gsl_bspline_knots_uniform**(const double a, const double b, gsl_bspline_workspace *w)

This function assumes uniformly spaced breakpoints on $[a, b]$ and constructs the corresponding knot vector using the previously specified `nbreak` parameter. The knots are stored in `w->knots`.

41.4 Evaluation of B-splines

int **gsl_bspline_eval**(const double x, gsl_vector *B, gsl_bspline_workspace *w)

This function evaluates all B-spline basis functions at the position `x` and stores them in the vector `B`, so that the i-th element is $B_i(x)$. The vector `B` must be of length $n = nbreak + k - 2$. This value may also be obtained by calling `gsl_bspline_ncoeffs()`. Computing all the basis functions at once is more efficient than computing them individually, due to the nature of the defining recurrence relation.

int **gsl_bspline_eval_nonzero**(const double x, gsl_vector *Bk, size_t *istart, size_t *iend, gsl_bspline_workspace *w)

This function evaluates all potentially nonzero B-spline basis functions at the position `x` and stores them in the vector `Bk`, so that the i-th element is $B_{(istart+i)}(x)$. The last element of `Bk` is $B_{iend}(x)$. The vector `Bk` must be of length k. By returning only the nonzero basis functions, this function allows quantities involving linear combinations of the $B_i(x)$ to be computed without unnecessary terms (such linear combinations occur, for example, when evaluating an interpolated function).

size_t **gsl_bspline_ncoeffs**(gsl_bspline_workspace *w)

This function returns the number of B-spline coefficients given by $n = nbreak + k - 2$.

41.5 Evaluation of B-spline derivatives

int **gsl_bspline_deriv_eval**(const double x, const size_t nderiv, gsl_matrix *dB, gsl_bspline_workspace *w)

This function evaluates all B-spline basis function derivatives of orders 0 through `nderiv` (inclusive) at the position `x` and stores them in the matrix `dB`. The (i,j)-th element of `dB` is $d^j B_i(x)/dx^j$. The matrix `dB` must be of size $n = nbreak + k - 2$ by $nderiv + 1$. The value n may also be obtained by calling `gsl_bspline_ncoeffs()`. Note that function evaluations are included as the zeroth order derivatives in `dB`. Computing all the basis function derivatives at once is more efficient than computing them individually, due to the nature of the defining recurrence relation.

int **gsl_bspline_deriv_eval_nonzero**(const double x, const size_t nderiv, gsl_matrix *dB, size_t *istart, size_t *iend, gsl_bspline_workspace *w)

This function evaluates all potentially nonzero B-spline basis function derivatives of orders 0 through `nderiv` (inclusive) at the position `x` and stores them in the matrix `dB`. The (i,j)-th element of `dB` is $d^j B_{(istart+i)}(x)/dx^j$. The last row of `dB` contains $d^j B_{iend}(x)/dx^j$.

The matrix **dB** must be of size k by at least $nderiv + 1$. Note that function evaluations are included as the zeroth order derivatives in **dB**. By returning only the nonzero basis functions, this function allows quantities involving linear combinations of the $B_i(x)$ and their derivatives to be computed without unnecessary terms.

41.6 Working with the Greville abscissae

The Greville abscissae are defined to be the mean location of $k - 1$ consecutive knots in the knot vector for each basis spline function of order k. With the first and last knots in the **gsl_bspline_workspace** knot vector excluded, there are **gsl_bspline_ncoeffs()** Greville abscissae for any given B-spline basis. These values are often used in B-spline collocation applications and may also be called Marsden-Schoenberg points.

double **gsl_bspline_greville_abscissa**(size_t i, gsl_bspline_workspace *w)

Returns the location of the i-th Greville abscissa for the given B-spline basis. For the ill-defined case when $k = 1$, the implementation chooses to return breakpoint interval midpoints.

41.7 Examples

The following program computes a linear least squares fit to data using cubic B-spline basis functions with uniform breakpoints. The data is generated from the curve $y(x) = \cos(x)\exp(-x/10)$ on the interval $[0, 15]$ with Gaussian noise added.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_statistics.h>

/* number of data points to fit */
#define N        200

/* number of fit coefficients */
#define NCOEFFS  12
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
/* nbreak = ncoeffs + 2 - k = ncoeffs - 2 since k = 4 */
#define NBREAK   (NCOEFFS - 2)

int
main (void)
{
  const size_t n = N;
  const size_t ncoeffs = NCOEFFS;
  const size_t nbreak = NBREAK;
  size_t i, j;
  gsl_bspline_workspace *bw;
  gsl_vector *B;
  double dy;
  gsl_rng *r;
  gsl_vector *c, *w;
  gsl_vector *x, *y;
  gsl_matrix *X, *cov;
  gsl_multifit_linear_workspace *mw;
  double chisq, Rsq, dof, tss;

  gsl_rng_env_setup();
  r = gsl_rng_alloc(gsl_rng_default);

  /* allocate a cubic bspline workspace (k = 4) */
  bw = gsl_bspline_alloc(4, nbreak);
  B = gsl_vector_alloc(ncoeffs);

  x = gsl_vector_alloc(n);
  y = gsl_vector_alloc(n);
  X = gsl_matrix_alloc(n, ncoeffs);
  c = gsl_vector_alloc(ncoeffs);
  w = gsl_vector_alloc(n);
  cov = gsl_matrix_alloc(ncoeffs, ncoeffs);
  mw = gsl_multifit_linear_alloc(n, ncoeffs);

  /* this is the data to be fitted */
  for (i = 0; i < n; ++i)
    {
      double sigma;
      double xi = (15.0 / (N - 1)) * i;
      double yi = cos(xi) * exp(-0.1 * xi);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    sigma = 0.1 * yi;
    dy = gsl_ran_gaussian(r, sigma);
    yi += dy;

    gsl_vector_set(x, i, xi);
    gsl_vector_set(y, i, yi);
    gsl_vector_set(w, i, 1.0 / (sigma * sigma));

    printf("%f %f\n", xi, yi);
  }

/* use uniform breakpoints on [0, 15] */
gsl_bspline_knots_uniform(0.0, 15.0, bw);

/* construct the fit matrix X */
for (i = 0; i < n; ++i)
  {
    double xi = gsl_vector_get(x, i);

    /* compute B_j(xi) for all j */
    gsl_bspline_eval(xi, B, bw);

    /* fill in row i of X */
    for (j = 0; j < ncoeffs; ++j)
      {
        double Bj = gsl_vector_get(B, j);
        gsl_matrix_set(X, i, j, Bj);
      }
  }

/* do the fit */
gsl_multifit_wlinear(X, w, y, c, cov, &chisq, mw);

dof = n - ncoeffs;
tss = gsl_stats_wtss(w->data, 1, y->data, 1, y->size);
Rsq = 1.0 - chisq / tss;

fprintf(stderr, "chisq/dof = %e, Rsq = %f\n",
                chisq / dof, Rsq);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  printf("\n\n");

  /* output the smoothed curve */
  {
    double xi, yi, yerr;

    for (xi = 0.0; xi < 15.0; xi += 0.1)
      {
        gsl_bspline_eval(xi, B, bw);
        gsl_multifit_linear_est(B, c, cov, &yi, &yerr);
        printf("%f %f\n", xi, yi);
      }
  }

  gsl_rng_free(r);
  gsl_bspline_free(bw);
  gsl_vector_free(B);
  gsl_vector_free(x);
  gsl_vector_free(y);
  gsl_matrix_free(X);
  gsl_vector_free(c);
  gsl_vector_free(w);
  gsl_matrix_free(cov);
  gsl_multifit_linear_free(mw);

  return 0;
} /* main() */
```

The output is shown below:

```
$ ./a.out > bspline.txt
chisq/dof = 1.118217e+00, Rsq = 0.989771
```

The data and fitted model are shown in 그림 41.1.

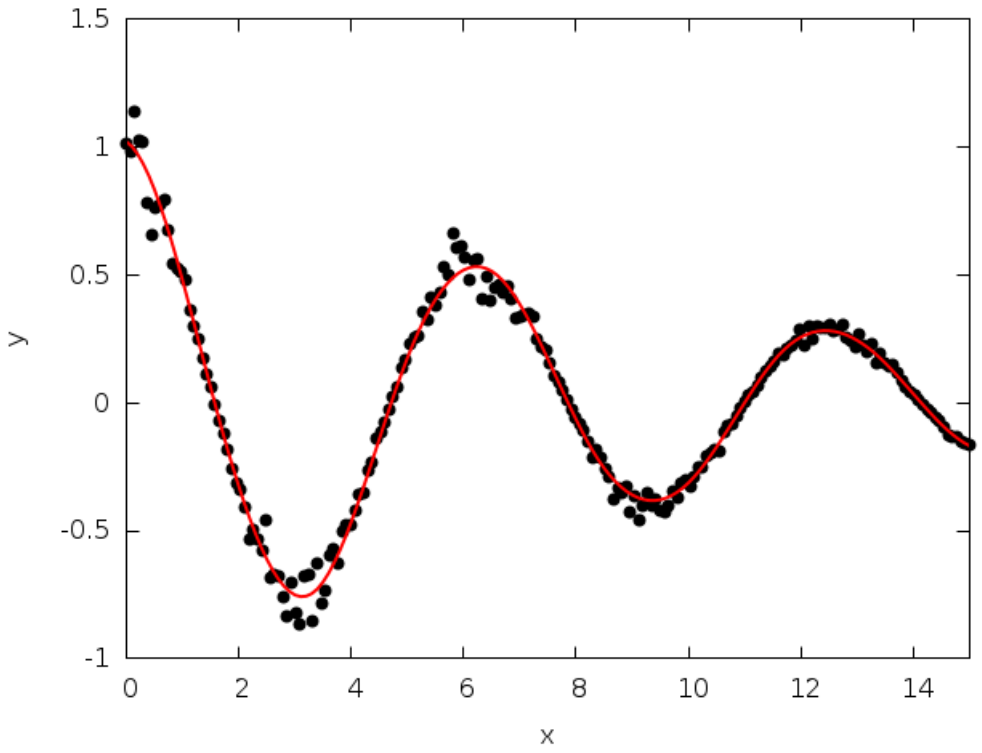


그림 41.1: Data (black) and fitted model (red)

41.8 References and Further Reading

Further information on the algorithms described in this section can be found in the following book,

- C. de Boor, A Practical Guide to Splines (1978), Springer-Verlag, ISBN 0-387-90356-9.

Further information of Greville abscissae and B-spline collocation can be found in the following paper,

- Richard W. Johnson, Higher order B-spline collocation at the Greville abscissae. Applied Numerical Mathematics. vol.: 52, 2005, 63–75.

A large collection of B-spline routines is available in the PPPACK library available at http://www.netlib.org/pppack, which is also part of SLATEC.

제 42 장

희소 행렬

참고: 번역중

This chapter describes functions for the construction and manipulation of sparse matrices, matrices which are populated primarily with zeros and contain only a few non-zero elements. Sparse matrices often appear in the solution of partial differential equations. It is beneficial to use specialized data structures and algorithms for storing and working with sparse matrices, since dense matrix algorithms and structures can be prohibitively slow and use huge amounts of memory when applied to sparse matrices.

The header file `gsl_spmatrix.h` contains the prototypes for the sparse matrix functions and related declarations.

42.1 Data types

All the functions are available for each of the standard data-types. The versions for `double` have the prefix `gsl_spmatrix`, Similarly the versions for single-precision `float` arrays have the prefix `gsl_spmatrix_float`. The full list of available types is given below,

Prefix	Type
gsl_spmatrix	double
gsl_spmatrix_float	float
gsl_spmatrix_long_double	long double
gsl_spmatrix_int	int
gsl_spmatrix_uint	unsigned int
gsl_spmatrix_long	long
gsl_spmatrix_ulong	unsigned long
gsl_spmatrix_short	short
gsl_spmatrix_ushort	unsigned short
gsl_spmatrix_char	char
gsl_spmatrix_uchar	unsigned char
gsl_spmatrix_complex	complex double
gsl_spmatrix_complex_float	complex float
gsl_spmatrix_complex_long_double	complex long double

42.2 Sparse Matrix Storage Formats

GSL currently supports three storage formats for sparse matrices: the coordinate (COO) representation, compressed sparse column (CSC) and compressed sparse row (CSR) formats. These are discussed in more detail below. In order to illustrate the different storage formats, the following sections will reference this M-by-N sparse matrix, with $M = 4$ and $N = 5$:

$$\begin{pmatrix} 9 & 0 & 0 & 0 & -3 \\ 4 & 7 & 0 & 0 & 0 \\ 0 & 8 & -1 & 8 & 0 \\ 4 & 0 & 5 & 6 & 0 \end{pmatrix}$$

The number of non-zero elements in the matrix, also abbreviated as `nnz` is equal to 10 in this case.

42.2.1 Coordinate Storage (COO)

The coordinate storage format, also known as triplet format, stores triplets (i, j, x) for each non-zero element of the matrix. This notation means that the (i, j) element of the matrix A is $A_{ij} = x$. The matrix is stored using three arrays of the same length, representing the row indices, column indices, and matrix data. For the reference matrix above, one possible storage format is:

data	9	7	4	8	-3	-1	8	5	6	4
row	0	1	1	2	0	2	2	3	3	3
col	0	1	0	1	4	2	3	2	3	0

Note that this representation is not unique - the coordinate triplets may appear in any ordering and would still represent the same sparse matrix. The length of the three arrays is equal to the number of non-zero elements in the matrix, `nnz`, which in this case is 10. The coordinate format is extremely convenient for sparse matrix assembly, the process of adding new elements, or changing existing elements, in a sparse matrix. However, it is generally not suitable for the efficient implementation of matrix-matrix products, or matrix factorization algorithms. For these applications it is better to use one of the compressed formats discussed below.

In order to faciliate efficient sparse matrix assembly, GSL stores the coordinate data in a balanced binary search tree, specifically an AVL tree, in addition to the three arrays discussed above. This allows GSL to efficiently determine whether an entry (i, j) already exists in the matrix, and to replace an existing matrix entry with a new value, without needing to search unsorted arrays.

42.2.2 Compressed Sparse Column (CSC)

Compressed sparse column storage stores each column of non-zero values in the sparse matrix in a continuous memory block, keeping pointers to the beginning of each column in that memory block, and storing the row indices of each non-zero element. For the reference matrix above, these arrays look like

data	9	4	4	7	8	-1	5	8	6	-3
row	0	1	3	1	2	2	3	2	3	0
col_ptr	0	3	5	7	9	10				

The `data` and `row` arrays are of length `nnz` and are the same as the COO storage format. The `col_ptr` array has length $N + 1$, and `col_ptr[j]` gives the index in `data` of the start of column `j`.

Therefore, the j-th column of the matrix is stored in `data[col_ptr[j]]`, `data[col_ptr[j] + 1]`, ⋯, `data[col_ptr[j+1] - 1]`. The last element of `col_ptr` is `nnz`.

42.2.3 Compressed Sparse Row (CSR)

Compressed row storage stores each row of non-zero values in a continuous memory block, keeping pointers to the beginning of each row in the block and storing the column indices of each non-zero element. For the reference matrix above, these arrays look like

data	9	-3	4	7	8	-1	8	4	5	6
col	0	4	0	1	1	2	3	0	2	3
row_ptr	0	2	4	7	10					

The `data` and `col` arrays are of length `nnz` and are the same as the COO storage format. The `row_ptr` array has length $M + 1$, and `row_ptr[i]` gives the index in `data` of the start of row `i`. Therefore, the i-th row of the matrix is stored in `data[row_ptr[i]]`, `data[row_ptr[i] + 1]`, ⋯, `data[row_ptr[i+1] - 1]`. The last element of `row_ptr` is `nnz`.

42.3 Overview

These routines provide support for constructing and manipulating sparse matrices in GSL, using an API similar to `gsl_matrix`. The basic structure is called `gsl_spmatrix`.

type **gsl_spmatrix**

This structure is defined as:

```
typedef struct
{
  size_t size1;
  size_t size2;
  int *i;
  double *data;
  int *p;
  size_t nzmax;
  size_t nz;
  [ ... variables for binary tree and memory management ... ]
  size_t sptype;
} gsl_spmatrix;
```

This defines a size1-by-size2 sparse matrix. The number of non-zero elements currently in the matrix is given by nz. For the triplet representation, i, p, and data are arrays of size nz which contain the row indices, column indices, and element value, respectively. So if $data[k] = A(i,j)$, then $i = i[k]$ and $j = p[k]$.

For compressed column storage, i and data are arrays of size nz containing the row indices and element values, identical to the triplet case. p is an array of size size2 + 1 where p[j] points to the index in data of the start of column j. Thus, if $data[k] = A(i,j)$, then $i = i[k]$ and $p[j] <= k < p[j+1]$.

For compressed row storage, i and data are arrays of size nz containing the column indices and element values, identical to the triplet case. p is an array of size size1 + 1 where p[i] points to the index in data of the start of row i. Thus, if $data[k] = A(i,j)$, then $j = i[k]$ and $p[i] <= k < p[i+1]$.

There are additional variables in the gsl_spmatrix structure related to binary tree storage and memory management. The GSL implementation of sparse matrices uses balanced AVL trees to sort matrix elements in the triplet representation. This speeds up element searches and duplicate detection during the matrix assembly process. The gsl_spmatrix structure also contains additional workspace variables needed for various operations like converting from triplet to compressed storage. sptype indicates the type of storage format being used (COO, CSC or CSR).

The compressed storage format defined above makes it very simple to interface with sophisticated external linear solver libraries which accept compressed storage input. The user can simply pass the arrays i, p, and data as the inputs to external libraries.

42.4 Allocation

The functions for allocating memory for a sparse matrix follow the style of malloc() and free(). They also perform their own error checking. If there is insufficient memory available to allocate a matrix then the functions call the GSL error handler with an error code of GSL_ENOMEM in addition to returning a null pointer.

gsl_spmatrix ***gsl_spmatrix_alloc**(const size_t n1, const size_t n2)

This function allocates a sparse matrix of size n1-by-n2 and initializes it to all zeros. If the size of the matrix is not known at allocation time, both n1 and n2 may be set to 1, and they will automatically grow as elements are added to the matrix. This function sets the matrix to the triplet representation, which is the easiest for adding and accessing matrix elements. This function tries to make a reasonable guess for the number of non-zero elements (nzmax) which will be added to the matrix by assuming a sparse density of

10%. The function gsl_spmatrix_alloc_nzmax() can be used if this number is known more accurately. The workspace is of size $O(nzmax)$.

gsl_spmatrix ***gsl_spmatrix_alloc_nzmax**(const size_t n1, const size_t n2, const size_t nzmax, const size_t sptype)

This function allocates a sparse matrix of size n1-by-n2 and initializes it to all zeros. If the size of the matrix is not known at allocation time, both n1 and n2 may be set to 1, and they will automatically grow as elements are added to the matrix. The parameter nzmax specifies the maximum number of non-zero elements which will be added to the matrix. It does not need to be precisely known in advance, since storage space will automatically grow using gsl_spmatrix_realloc() if nzmax is not large enough. Accurate knowledge of this parameter reduces the number of reallocation calls required. The parameter sptype specifies the storage format of the sparse matrix. Possible values are

GSL_SPMATRIX_COO

This flag specifies coordinate (triplet) storage.

GSL_SPMATRIX_CSC

This flag specifies compressed sparse column storage.

GSL_SPMATRIX_CSR

This flag specifies compressed sparse row storage.

The allocated gsl_spmatrix structure is of size $O(nzmax)$.

int **gsl_spmatrix_realloc**(const size_t nzmax, gsl_spmatrix *m)

This function reallocates the storage space for m to accomodate nzmax non-zero elements. It is typically called internally by gsl_spmatrix_set() if the user wants to add more elements to the sparse matrix than the previously specified nzmax.

Input matrix formats supported: COO, CSC, CSR

void **gsl_spmatrix_free**(gsl_spmatrix *m)

This function frees the memory associated with the sparse matrix m.

Input matrix formats supported: COO, CSC, CSR

42.5 Accessing Matrix Elements

double **gsl_spmatrix_get**(const gsl_spmatrix *m, const size_t i, const size_t j)

This function returns element (i, j) of the matrix m.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_set**(gsl_spmatrix *m, const size_t i, const size_t j, const double x)

This function sets element (i, j) of the matrix m to the value x.

Input matrix formats supported: COO

double ***gsl_spmatrix_ptr**(gsl_spmatrix *m, const size_t i, const size_t j)

This function returns a pointer to the (i, j) element of the matrix m. If the (i, j) element is not explicitly stored in the matrix, a null pointer is returned.

Input matrix formats supported: COO, CSC, CSR

42.6 Initializing Matrix Elements

Since the sparse matrix format only stores the non-zero elements, it is automatically initialized to zero upon allocation. The function `gsl_spmatrix_set_zero()` may be used to re-initialize a matrix to zero after elements have been added to it.

int **gsl_spmatrix_set_zero**(gsl_spmatrix *m)

This function sets (or resets) all the elements of the matrix m to zero. For CSC and CSR matrices, the cost of this operation is $O(1)$. For COO matrices, the binary tree structure must be dismantled, so the cost is $O(nz)$.

Input matrix formats supported: COO, CSC, CSR

42.7 Reading and Writing Matrices

int **gsl_spmatrix_fwrite**(FILE *stream, const gsl_spmatrix *m)

This function writes the elements of the matrix m to the stream `stream` in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_fread**(FILE *stream, gsl_spmatrix *m)

This function reads into the matrix `m` from the open stream `stream` in binary format. The matrix `m` must be preallocated with the correct storage format, dimensions and have a sufficiently large `nzmax` in order to read in all matrix elements, otherwise `GSL_EBADLEN` is returned. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_fprintf**(FILE *stream, const gsl_spmatrix *m, const char *format)

This function writes the elements of the matrix `m` line-by-line to the stream `stream` using the format specifier `format`, which should be one of the `%g`, `%e` or `%f` formats for floating point numbers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file. The input matrix `m` may be in any storage format, and the output file will be written in MatrixMarket format.

Input matrix formats supported: COO, CSC, CSR

gsl_spmatrix ***gsl_spmatrix_fscanf**(FILE *stream)

This function reads sparse matrix data in the MatrixMarket format from the stream `stream` and stores it in a newly allocated matrix which is returned in COO format. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The user should free the returned matrix when it is no longer needed.

42.8 Copying Matrices

int **gsl_spmatrix_memcpy**(gsl_spmatrix *dest, const gsl_spmatrix *src)

This function copies the elements of the sparse matrix `src` into `dest`. The two matrices must have the same dimensions and be in the same storage format.

Input matrix formats supported: COO, CSC, CSR

42.9 Exchanging Rows and Columns

int **gsl_spmatrix_transpose_memcpy**(gsl_spmatrix *dest, const gsl_spmatrix *src)

This function copies the transpose of the sparse matrix `src` into `dest`. The dimensions of `dest` must match the transpose of the matrix `src`. Also, both matrices must use the same sparse storage format.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_transpose**(gsl_spmatrix *m)

This function replaces the matrix `m` by its transpose, but changes the storage format for compressed matrix inputs. Since compressed column storage is the transpose of compressed row storage, this function simply converts a CSC matrix to CSR and vice versa. This is the most efficient way to transpose a compressed storage matrix, but the user should note that the storage format of their compressed matrix will change on output. For COO matrix inputs, the output matrix is also in COO storage.

Input matrix formats supported: COO, CSC, CSR

42.10 Matrix Operations

int **gsl_spmatrix_scale**(gsl_spmatrix *m, const double x)

This function scales all elements of the matrix `m` by the constant factor `x`. The result $m(i,j) \leftarrow x m(i,j)$ is stored in `m`.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_scale_columns**(gsl_spmatrix *A, const gsl_vector *x)

This function scales the columns of the M-by-N sparse matrix `A` by the elements of the vector `x`, of length N. The j-th column of `A` is multiplied by `x[j]`. This is equivalent to forming

$$A \rightarrow AX$$

where $X = \text{diag}(x)$.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_scale_rows**(gsl_spmatrix *A, const gsl_vector *x)

This function scales the rows of the M-by-N sparse matrix `A` by the elements of the vector `x`, of length M. The i-th row of `A` is multiplied by `x[i]`. This is equivalent to forming

$$A \rightarrow XA$$

where $X = \text{diag}(x)$.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_add**(gsl_spmatrix *c, const gsl_spmatrix *a, const gsl_spmatrix *b)

This function computes the sum $c = a + b$. The three matrices must have the same dimensions.

Input matrix formats supported: CSC, CSR

int **gsl_spmatrix_dense_add**(gsl_matrix *a, const gsl_spmatrix *b)

This function adds the elements of the sparse matrix `b` to the elements of the dense matrix `a`. The result $a(i,j) \leftarrow a(i,j) + b(i,j)$ is stored in `a` and `b` remains unchanged. The two matrices must have the same dimensions.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_dense_sub**(gsl_matrix *a, const gsl_spmatrix *b)

This function subtracts the elements of the sparse matrix `b` from the elements of the dense matrix `a`. The result $a(i,j) \leftarrow a(i,j) - b(i,j)$ is stored in `a` and `b` remains unchanged. The two matrices must have the same dimensions.

Input matrix formats supported: COO, CSC, CSR

42.11 Matrix Properties

const char *__gsl_spmatrix_type__(const gsl_spmatrix *m)

This function returns a string describing the sparse storage format of the matrix `m`. For example:

```
printf ("matrix is '%s' format.\n", gsl_spmatrix_type (m));
```

would print something like:

```
matrix is 'CSR' format.
```

Input matrix formats supported: COO, CSC, CSR

size_t **gsl_spmatrix_nnz**(const gsl_spmatrix *m)

This function returns the number of non-zero elements in `m`.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_equal**(const gsl_spmatrix *a, const gsl_spmatrix *b)

This function returns 1 if the matrices `a` and `b` are equal (by comparison of element values) and 0 otherwise. The matrices `a` and `b` must be in the same sparse storage format for comparison.

Input matrix formats supported: COO, CSC, CSR

double **gsl_spmatrix_norm1**(const gsl_spmatrix *A)

This function returns the 1-norm of the m-by-n matrix `A`, defined as the maximum column sum,

$$||A||_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |A_{ij}|$$

Input matrix formats supported: COO, CSC, CSR

42.12 Finding Maximum and Minimum Elements

int **gsl_spmatrix_minmax**(const gsl_spmatrix *m, double *min_out, double *max_out)

This function returns the minimum and maximum elements of the matrix `m`, storing them in `min_out` and `max_out`, and searching only the non-zero values.

Input matrix formats supported: COO, CSC, CSR

int **gsl_spmatrix_min_index**(const gsl_spmatrix *m, size_t *imin, size_t *jmin)

This function returns the indices of the minimum value in the matrix `m`, searching only the non-zero values, and storing them in `imin` and `jmin`. When there are several equal minimum elements then the first element found is returned.

Input matrix formats supported: COO, CSC, CSR

42.13 Compressed Format

These routines calculate a compressed matrix from a coordinate representation.

int **gsl_spmatrix_csc**(gsl_spmatrix *dest, const gsl_spmatrix *src)

This function creates a sparse matrix in compressed sparse column format from the input sparse matrix `src` which must be in COO format. The compressed matrix is stored in `dest`.

Input matrix formats supported: COO

int **gsl_spmatrix_csr**(gsl_spmatrix *dest, const gsl_spmatrix *src)

This function creates a sparse matrix in compressed sparse row format from the input sparse matrix `src` which must be in COO format. The compressed matrix is stored in `dest`.

Input matrix formats supported: COO

gsl_spmatrix *__gsl_spmatrix_compress__(const gsl_spmatrix *src, const int sptype)

This function allocates a new sparse matrix, and stores `src` into it using the format specified by `sptype`. The input `sptype` can be one of `GSL_SPMATRIX_COO`, `GSL_SPMATRIX_CSC`, or `GSL_SPMATRIX_CSR`. A pointer to the newly allocated matrix is returned, and must be freed by the caller when no longer needed.

42.14 Conversion Between Sparse and Dense Matrices

The `gsl_spmatrix` structure can be converted into the dense `gsl_matrix` format and vice versa with the following routines.

int __gsl_spmatrix_d2sp__(gsl_spmatrix *S, const gsl_matrix *A)

This function converts the dense matrix `A` into sparse COO format and stores the result in `S`.

Input matrix formats supported: COO

int __gsl_spmatrix_sp2d__(gsl_matrix *A, const gsl_spmatrix *S)

This function converts the sparse matrix `S` into a dense matrix and stores the result in `A`.

Input matrix formats supported: COO, CSC, CSR

42.15 Examples

The following example program builds a 5-by-4 sparse matrix and prints it in coordinate, compressed column, and compressed row format. The matrix which is constructed is

$$\begin{pmatrix} 0 & 0 & 3.1 & 4.6 \\ 1 & 0 & 7.2 & 0 \\ 0 & 0 & 0 & 0 \\ 2.1 & 2.9 & 0 & 8.5 \\ 4.1 & 0 & 0 & 0 \end{pmatrix}$$

The output of the program is:

```
printing all matrix elements:
A(0,0) = 0
A(0,1) = 0
A(0,2) = 3.1
A(0,3) = 4.6
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
A(1,0) = 1
.
.
.
A(4,0) = 4.1
A(4,1) = 0
A(4,2) = 0
A(4,3) = 0
matrix in triplet format (i,j,Aij):
(0, 2, 3.1)
(0, 3, 4.6)
(1, 0, 1.0)
(1, 2, 7.2)
(3, 0, 2.1)
(3, 1, 2.9)
(3, 3, 8.5)
(4, 0, 4.1)
matrix in compressed column format:
i = [ 1, 3, 4, 3, 0, 1, 0, 3, ]
p = [ 0, 3, 4, 6, 8, ]
d = [ 1, 2.1, 4.1, 2.9, 3.1, 7.2, 4.6, 8.5, ]
matrix in compressed row format:
i = [ 2, 3, 0, 2, 0, 1, 3, 0, ]
p = [ 0, 2, 4, 4, 7, 8, ]
d = [ 3.1, 4.6, 1, 7.2, 2.1, 2.9, 8.5, 4.1, ]
```

We see in the compressed column output, the data array stores each column contiguously, the array i stores the row index of the corresponding data element, and the array p stores the index of the start of each column in the data array. Similarly, for the compressed row output, the data array stores each row contiguously, the array i stores the column index of the corresponding data element, and the p array stores the index of the start of each row in the data array.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_spmatrix.h>

int
main()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
{
  gsl_spmatrix *A = gsl_spmatrix_alloc(5, 4); /* triplet format */
  gsl_spmatrix *B, *C;
  size_t i, j;

  /* build the sparse matrix */
  gsl_spmatrix_set(A, 0, 2, 3.1);
  gsl_spmatrix_set(A, 0, 3, 4.6);
  gsl_spmatrix_set(A, 1, 0, 1.0);
  gsl_spmatrix_set(A, 1, 2, 7.2);
  gsl_spmatrix_set(A, 3, 0, 2.1);
  gsl_spmatrix_set(A, 3, 1, 2.9);
  gsl_spmatrix_set(A, 3, 3, 8.5);
  gsl_spmatrix_set(A, 4, 0, 4.1);

  printf("printing all matrix elements:\n");
  for (i = 0; i < 5; ++i)
    for (j = 0; j < 4; ++j)
      printf("A(%zu,%zu) = %g\n", i, j,
             gsl_spmatrix_get(A, i, j));

  /* print out elements in triplet format */
  printf("matrix in triplet format (i,j,Aij):\n");
  gsl_spmatrix_fprintf(stdout, A, "%.1f");

  /* convert to compressed column format */
  B = gsl_spmatrix_ccs(A);

  printf("matrix in compressed column format:\n");
  printf("i = [ ");
  for (i = 0; i < B->nz; ++i)
    printf("%d, ", B->i[i]);
  printf("]\n");

  printf("p = [ ");
  for (i = 0; i < B->size2 + 1; ++i)
    printf("%d, ", B->p[i]);
  printf("]\n");

  printf("d = [ ");
  for (i = 0; i < B->nz; ++i)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    printf("%g, ", B->data[i]);
  printf("]\n");

  /* convert to compressed row format */
  C = gsl_spmatrix_crs(A);

  printf("matrix in compressed row format:\n");
  printf("i = [ ");
  for (i = 0; i < C->nz; ++i)
    printf("%d, ", C->i[i]);
  printf("]\n");

  printf("p = [ ");
  for (i = 0; i < C->size1 + 1; ++i)
    printf("%d, ", C->p[i]);
  printf("]\n");

  printf("d = [ ");
  for (i = 0; i < C->nz; ++i)
    printf("%g, ", C->data[i]);
  printf("]\n");

  gsl_spmatrix_free(A);
  gsl_spmatrix_free(B);
  gsl_spmatrix_free(C);

  return 0;
}
```

42.16 References and Further Reading

The algorithms used by these functions are described in the following sources,

- Davis, T. A., Direct Methods for Sparse Linear Systems, SIAM, 2006.
- CSparse software library, https://www.cise.ufl.edu/research/sparse/CSparse

제 43 장

희소 BLAS 지원

참고: 번역중

The Sparse Basic Linear Algebra Subprograms (BLAS) define a set of fundamental operations on vectors and sparse matrices which can be used to create optimized higher-level linear algebra functionality. GSL supports a limited number of BLAS operations for sparse matrices.

The header file `gsl_spblas.h` contains the prototypes for the sparse BLAS functions and related declarations.

43.1 Sparse BLAS operations

int **gsl_spblas_dgemv**(const CBLAS_TRANSPOSE_t TransA, const double alpha, const gsl_spmatrix *A, const gsl_vector *x, const double beta, gsl_vector *y)

This function computes the matrix-vector product and sum $y \leftarrow \alpha op(A)x + \beta y$, where $op(A) = A, A^T$ for `TransA` = `CblasNoTrans`, `CblasTrans`. In-place computations are not supported, so `x` and `y` must be distinct vectors. The matrix `A` may be in triplet or compressed format.

int **gsl_spblas_dgemm**(const double alpha, const gsl_spmatrix *A, const gsl_spmatrix *B, gsl_spmatrix *C)

This function computes the sparse matrix-matrix product $C = \alpha AB$. The matrices must be in compressed format.

43.2 References and Further Reading

The algorithms used by these functions are described in the following sources:

- Davis, T. A., Direct Methods for Sparse Linear Systems, SIAM, 2006.
- CSparse software library, https://www.cise.ufl.edu/research/sparse/CSparse

제 44 장

희소 선형 대수

참고: 번역중

이 단원에서 기술하는 함수들은 희소 선형계를 풀기위한 기능들을 제공합니다. 이 함수들은 `gsl_spmatrix` 와 `gsl_vector` 객체들에 직접 접근해 계산합니다.

헤더 파일 `gsl_splinalg.h` 에 정의되어 있습니다.

44.1 개요

이 단원에서 제공하는 기능들의 주된 목표는 다음의 선형계의 해답을 얻는 것입니다.

$$Ax = b$$

A 는 일반적인 $n \times n$ 정사각 행렬 where A is a general square n-by-n non-singular sparse matrix, x is an unknown n-by-1 vector, and b is a given n-by-1 right hand side vector. There exist many methods for solving such sparse linear systems, which broadly fall into either direct or iterative categories. Direct methods include LU and QR decompositions, while iterative methods start with an initial guess for the vector x and update the guess through iteration until convergence. GSL does not currently provide any direct sparse solvers.

44.2 Sparse Iterative Solvers

44.2.1 Overview

Many practical iterative methods of solving large n-by-n sparse linear systems involve projecting an approximate solution for x onto a subspace of $\mathbf{R}^n$. If we define a m-dimensional subspace $\mathcal{K}$ as the subspace of approximations to the solution x, then m constraints must be imposed to determine the next approximation. These m constraints define another m-dimensional subspace denoted by $\mathcal{L}$. The subspace dimension m is typically chosen to be much smaller than n in order to reduce the computational effort needed to generate the next approximate solution vector. The many iterative algorithms which exist differ mainly in their choice of $\mathcal{K}$ and $\mathcal{L}$.

44.2.2 Types of Sparse Iterative Solvers

The sparse linear algebra library provides the following types of iterative solvers:

type **gsl_splinalg_itersolve_type**

gsl_splinalg_itersolve_type ***gsl_splinalg_itersolve_gmres**

This specifies the Generalized Minimum Residual Method (GMRES). This is a projection method using $\mathcal{K} = \mathcal{K}_m$ and $\mathcal{L} = A\mathcal{K}_m$ where $\mathcal{K}_m$ is the m-th Krylov subspace

$$\mathcal{K}_m = span\left\{r_0, Ar_0, ..., A^{m-1}r_0\right\}$$

and $r_0 = b - Ax_0$ is the residual vector of the initial guess x_0. If m is set equal to n, then the Krylov subspace is $\mathbf{R}^n$ and GMRES will provide the exact solution x. However, the goal is for the method to arrive at a very good approximation to x using a much smaller subspace $\mathcal{K}_m$. By default, the GMRES method selects $m = MIN(n, 10)$ but the user may specify a different value for m. The GMRES storage requirements grow as $O(n(m+1))$ and the number of flops grow as $O(4m^2n - 4m^3/3)$.

In the below function gsl_splinalg_itersolve_iterate(), one GMRES iteration is defined as projecting the approximate solution vector onto each Krylov subspace $\mathcal{K}_1, ..., \mathcal{K}_m$, and so m can be kept smaller by "restarting" the method and calling gsl_splinalg_itersolve_iterate() multiple times, providing the updated approximation x to each new call. If the method is not adequately converging, the user may try increasing the parameter m.

GMRES is considered a robust general purpose iterative solver, however there are cases where the method stagnates if the matrix is not positive-definite and fails to

reduce the residual until the very last projection onto the subspace $\mathcal{K}_n = \mathbf{R}^n$. In these cases, preconditioning the linear system can help, but GSL does not currently provide any preconditioners.

44.2.3 Iterating the Sparse Linear System

The following functions are provided to allocate storage for the sparse linear solvers and iterate the system to a solution.

gsl_splinalg_itersolve ***gsl_splinalg_itersolve_alloc**(const gsl_splinalg_itersolve_type *T, const size_t n, const size_t m)

This function allocates a workspace for the iterative solution of `n`-by-`n` sparse matrix systems. The iterative solver type is specified by `T`. The argument `m` specifies the size of the solution candidate subspace $\mathcal{K}_m$. The dimension `m` may be set to 0 in which case a reasonable default value is used.

void **gsl_splinalg_itersolve_free**(gsl_splinalg_itersolve *w)

This function frees the memory associated with the workspace `w`.

const char ***gsl_splinalg_itersolve_name**(const gsl_splinalg_itersolve *w)

This function returns a string pointer to the name of the solver.

int **gsl_splinalg_itersolve_iterate**(const gsl_spmatrix *A, const gsl_vector *b, const double tol, gsl_vector *x, gsl_splinalg_itersolve *w)

This function performs one iteration of the iterative method for the sparse linear system specfied by the matrix `A`, right hand side vector `b` and solution vector `x`. On input, `x` must be set to an initial guess for the solution. On output, `x` is updated to give the current solution estimate. The parameter `tol` specifies the relative tolerance between the residual norm and norm of `b` in order to check for convergence. When the following condition is satisfied:

$$||Ax - b|| \leq tol \times ||b||$$

the method has converged, the function returns `GSL_SUCCESS` and the final solution is provided in `x`. Otherwise, the function returns `GSL_CONTINUE` to signal that more iterations are required. Here, $||\cdot||$ represents the Euclidean norm. The input matrix `A` may be in triplet or compressed format.

double **gsl_splinalg_itersolve_normr**(const gsl_splinalg_itersolve *w)

This function returns the current residual norm $||r|| = ||Ax - b||$, which is updated after each call to `gsl_splinalg_itersolve_iterate()`.

44.3 Examples

This example program demonstrates the sparse linear algebra routines on the solution of a simple 1D Poisson equation on $[0, 1]$:

$$\frac{d^2u(x)}{dx^2} = f(x) = -\pi^2 \sin(\pi x)$$

with boundary conditions $u(0) = u(1) = 0$. The analytic solution of this simple problem is $u(x) = \sin \pi x$. We will solve this problem by finite differencing the left hand side to give

$$\frac{1}{h^2}(u_{i+1} - 2u_i + u_{i-1}) = f_i$$

Defining a grid of N points, $h = 1/(N-1)$. In the finite difference equation above, $u_0 = u_{N-1} = 0$ are known from the boundary conditions, so we will only put the equations for $i = 1, ..., N-2$ into the matrix system. The resulting $(N-2) \times (N-2)$ matrix equation is

$$\frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 1 & -2 & 1 \\ 0 & \dots & \dots & \dots & 1 & -2 \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-3} \\ u_{N-2} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-3} \\ f_{N-2} \end{pmatrix}$$

An example program which constructs and solves this system is given below. The system is solved using the iterative GMRES solver. Here is the output of the program:

```
iter 0 residual = 4.297275996844e-11
Converged
```

showing that the method converged in a single iteration. The calculated solution is shown in 그림 44.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_spmatrix.h>
#include <gsl/gsl_splinalg.h>
```

(다음 페이지에 계속)

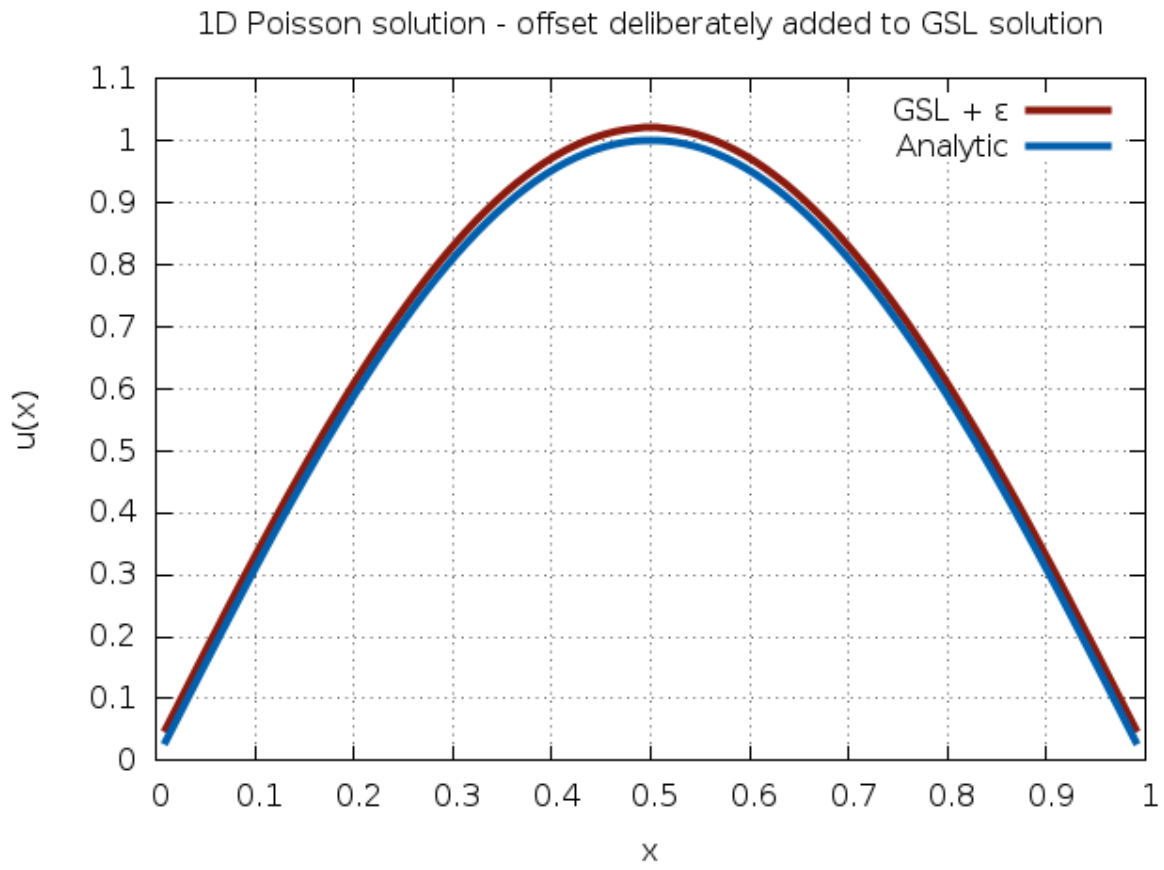


그림 44.1: Solution of PDE

(이전 페이지에서 계속)

```
int
main()
{
  const size_t N = 100;                     /* number of grid points */
  const size_t n = N - 2;                   /* subtract 2 to exclude boundaries */
  const double h = 1.0 / (N - 1.0);         /* grid spacing */
  gsl_spmatrix *A = gsl_spmatrix_alloc(n ,n); /* triplet format */
  gsl_spmatrix *C;                          /* compressed format */
  gsl_vector *f = gsl_vector_alloc(n);      /* right hand side vector */
  gsl_vector *u = gsl_vector_alloc(n);      /* solution vector */
  size_t i;

  /* construct the sparse matrix for the finite difference equation */

  /* construct first row */
  gsl_spmatrix_set(A, 0, 0, -2.0);
  gsl_spmatrix_set(A, 0, 1, 1.0);

  /* construct rows [1:n-2] */
  for (i = 1; i < n - 1; ++i)
    {
      gsl_spmatrix_set(A, i, i + 1, 1.0);
      gsl_spmatrix_set(A, i, i, -2.0);
      gsl_spmatrix_set(A, i, i - 1, 1.0);
    }

  /* construct last row */
  gsl_spmatrix_set(A, n - 1, n - 1, -2.0);
  gsl_spmatrix_set(A, n - 1, n - 2, 1.0);

  /* scale by h^2 */
  gsl_spmatrix_scale(A, 1.0 / (h * h));

  /* construct right hand side vector */
  for (i = 0; i < n; ++i)
    {
      double xi = (i + 1) * h;
      double fi = -M_PI * M_PI * sin(M_PI * xi);
      gsl_vector_set(f, i, fi);
    }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
/* convert to compressed column format */
C = gsl_spmatrix_ccs(A);

/* now solve the system with the GMRES iterative solver */
{
  const double tol = 1.0e-6;  /* solution relative tolerance */
  const size_t max_iter = 10; /* maximum iterations */
  const gsl_splinalg_itersolve_type *T = gsl_splinalg_itersolve_gmres;
  gsl_splinalg_itersolve *work =
    gsl_splinalg_itersolve_alloc(T, n, 0);
  size_t iter = 0;
  double residual;
  int status;

  /* initial guess u = 0 */
  gsl_vector_set_zero(u);

  /* solve the system A u = f */
  do
    {
      status = gsl_splinalg_itersolve_iterate(C, f, tol, u, work);

      /* print out residual norm ||A*u - f|| */
      residual = gsl_splinalg_itersolve_normr(work);
      fprintf(stderr, "iter %zu residual = %.12e\n", iter, residual);

      if (status == GSL_SUCCESS)
        fprintf(stderr, "Converged\n");
    }
  while (status == GSL_CONTINUE && ++iter < max_iter);

  /* output solution */
  for (i = 0; i < n; ++i)
    {
      double xi = (i + 1) * h;
      double u_exact = sin(M_PI * xi);
      double u_gsl = gsl_vector_get(u, i);

      printf("%f %.12e %.12e\n", xi, u_gsl, u_exact);
    }
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    gsl_splinalg_itersolve_free(work);
  }

  gsl_spmatrix_free(A);
  gsl_spmatrix_free(C);
  gsl_vector_free(f);
  gsl_vector_free(u);

  return 0;
} /* main() */
```

44.4 References and Further Reading

The implementation of the GMRES iterative solver closely follows the publications

- H. F. Walker, Implementation of the GMRES method using Householder transformations, SIAM J. Sci. Stat. Comput. 9(1), 1988.
- Y. Saad, Iterative methods for sparse linear systems, 2nd edition, SIAM, 2003.

제 45 장

물리 상수

이 단원에서는 물리 상수들을 위한 매크로들을 기술합니다. 예를 들어 빛의 속도 c 나, 중력 상수 G 등이 있습니다. 각각의 값들은 다른 단위계들로 존재합니다. 천체물리에서 빈번히 쓰이는 표준 MKSA 계 (meter, kilogransms, seconds, ampers)와 CGSM 계 (centimeters, grams, seconds, gauss)가 있습니다

MKSA 계로 정의된 함수들은 `gsl_const_mksa.h` 에 정의되어 있습니다. CGSM 계로 정의된 상수들은 `gsl_const_cgsm.h` 에 정의되어 있습니다.

차원이 없는 단위, 예를 들어서 미세 구조 상수와 같은 순수한 숫자들은 `gsl_const_num.h` 에 정의되어 있습니다

상수들의 완전한 목록은 다음 소단원들에 간단하게 기술되어있습니다. 라이브러리에서 사용되는 상수 값들은 헤더 파일을 참고할 수 있습니다

45.1 기초 상수

GSL_CONST_MKSA_SPEED_OF_LIGHT

진공에서의 빛의 속도 c

GSL_CONST_MKSA_VACUUM_PERMEABILITY

진공 투자율 μ_0 이 상수는 MKSA 계에서만 정의되었습니다

GSL_CONST_MKSA_VACUUM_PERMITTIVITY

진공 유전율 ϵ_0 이 상수는 MKSA 계에서만 정의되었습니다

GSL_CONST_MKSA_PLANKS_CONSTANT_H

플랑크 상수 h

GSL_CONST_MKSA_PLANKS_CONSTANT_HBAR

플랑크 상수를 2π 로 나눈 값 $\hbar$

GSL_CONST_NUM_AVOGADRO

아보가드로 수 N_α

GSL_CONST_MKSA_FARADAY

1 패러데이(F)

GSL_CONST_MKSA_BOLTZMANN

볼츠만 상수 k

GSL_CONST_MKSA_MOLAR_GAS

표준 기체 상수 R_0

GSL_CONST_MKSA_STANDARD_GAS_VOLUME

표준 기체 부피 V_0

GSL_CONST_MKSA_STEFAN_BOLTZMANN_CONSTANT

슈테판 볼츠만 방사 상수 σ

GSL_CONST_MKSA_GAUSS

1 가우스(Gauss) 크기의 자기장 세기

45.2 천문학과 천체물리

GSL_CONST_MKSA_ASTRONOMICAL_UNIT

천문 단위 (지구-태양 사이의 거리), au

GSL_CONST_MKSA_GRAVITATIONAL_CONSTANT

중력 상수, G

GSL_CONST_MKSA_LIGHT_YEAR

1 광년 거리, ly

GSL_CONST_MKSA_PASRSEC

1 파섹 거리, pc

GSL_CONST_MKSA_GRAV_ACCEL

지구 표면에서의 표준 중력 가속도, g

GSL_CONST_MKSA_SOLAR_MASS

태양의 질량

45.3 입자, 핵물리

GSL_CONST_MKSA_ELECTRON_CHARGE

전자의 전하, e

GSL_CONST_MKSA_ELECTORN_VOLT

1 전자 볼트, eV

GSL_CONST_MKSA_UNIFIED_ATOMIC_MASS

원자 질량 단위, amu

GSL_CONST_MKSA_MASS_ELECTRON

전자의 질량, m_e

GSL_CONST_MKSA_MASS_MUON

뮤온의 질량, m_μ

GSL_CONST_MKSA_MASS_PROTON

광자의 질량, m_p

GSL_CONST_MKSA_MASS_NEUTRON

중성자의 질량, m_n

GSL_CONST_NUM_FINE_STRUCTURE

미세 구조 상수, α

GSL_CONST_MKSA_RYDBERG

에너지 단위 리드버그 수, R_y 리드버그 상수와 다음과 같은 관계를 가집니다. $R_y = hcR_\infty$

GSL_CONST_MKSA_BOHR_RADIUS

보어 반지름, a_0

GSL_CONST_MKSA_ANGSTOM

1 옴스트롱, Å

GSL_CONST_MKSA_BARN

1 바른, b

GSL_CONST_MKSA_BOHR_MAGNETON

보어 마그네톤, μ_B

GSL_CONST_MKSA_NUCLEAR_MAGNERON

핵 마그네톤, μ_N

GSL_CONST_MKSA__ELECTRON_MAGNETIC_MOMENT

전자의 자기 모멘트 절대 값, μ_e 실제 물리적인 전자의 자기 모멘트는 음수 값을 가집니다

GSL_CONST_MKSA_PROTON_MAFNETIC_MOMENT

양성자의 자기 모멘트 값, μ_p

GSL_CONST_MKSA_THOMSON_CROSS_SECTION

톰슨 단면적, σ_T

GSL_CONST_MKSA_DEBYE

전기 쌍극자 모멘트 단위; 1 디바이, D

45.4 시간 측정

다음 상수들은 모두 초 단위 값을 가집니다.(*)

GSL_CONST_MKSA_MINUTE

1 분; 60초

GSL_CONST_MKSA_HOUR

1 시간; 360초

GSL_CONST_MKSA_DAY

1 일; 8,640초

GSL_CONST_MKSA_WEEK

1 주일; 60,480초

45.5 야드-파운드 단위

GSL_CONST_MKSA_INCH

1 인치 길이

GSL_CONST_MKSA_FOOT

1 피트 길이

GSL_CONST_MKSA_YARD

1 야드 길이

GSL_CONST_MKSA_MILE

1 마일 길이

GSL_CONST_MKSA_MIL

1 밀 길이 (인치의 1/10000 배)

45.6 속도, 해리 단위

GSL_CONST_MKSA_KILOMETERS_PER_HOUR

속 1 킬로미터

GSL_CONST_MKSA_MILES_PER_HOUR

시속 1 마일

GSL_CONST_MKSA_NAUTICAL_MILE

1 해리 길이

GSL_CONST_MKSA_FATHOM

1 패덤 길이

GSL_CONST_MKSA_KNOT

1 노트 속도

45.7 출력 단위

GSL_CONST_MKSA_POINT

1 출력 크기 (1/72 인치)

GSL_CONST_MKSA_TEXPOINT

1 Tex 크기 (1/72.27 인치)

45.8 부피, 면적 그리고 길이

GSL_CONST_MKSA_MICRON

1 마이크론 길이

GSL_CONST_MKSA_HECTARE

1 헥타르 크기

GSL_CONST_MKSA_ACRE

1 에이커 크기

GSL_CONST_MKSA_LITER

1 리터 부피

GSL_CONST_MKSA_US_GALLON

1 US 갤런 부피

GSL_CONST_MKSA_CANADIAN_GALLON

1 Canada 갤런 부피

GSL_CONST_MKSA_UK_GALLON

1 UK 갤런 부피

GSL_CONST_MKSA_QUART

1 쿼트 부피

GSL_CONST_MKSA_PINT

1 파인트 부피

45.9 질량과 무게

GSL_CONST_MKSA_POUND_MASS

1 파운드 질량

GSL_CONST_MKSA_OUNCE_MASS

1 온스 질량

GSL_CONST_MKSA_TON

1 톤 질량

GSL_CONST_MKSA_METRIC_TON

1 (metric) 톤 질량

GSL_CONST_MKSA_UK_TON

1 UK 톤 질량

GSL_CONST_MKSA_TROY_OUNCE

1 트로이 온스 질량

GSL_CONST_MKSA_CARAT

1 케럿 질량

GSL_CONST_MKSA_GRAM_FORCE

1 그램 무게

GSL_CONST_MKSA_POUND_FORCE

1 파운드 무게

GSL_CONST_MKSA_KILOPOUND_FORCE

1 킬로 파운드 무게

GSL_CONST_MKSA_POUNDAL

1 파운달 크기

45.10 열 에너지와 힘

GSL_CONST_MKSA_CALORIE

1 칼로리 에너지량

GSL_CONST_MKSA_BTU

1 영국 열 단위, btu

GSL_CONST_MKSA_THERM

1 섬

GSL_CONST_MKSA_HORSEPOWER

1 마력

45.11 압력

GSL_CONST_MKSA_BAR

1 바 압력

GSL_CONST_MKSA_STD_ATOMSPHERE

1 표준 대기압

GSL_CONST_MKSA_TORR

1 토르

GSL_CONST_MKSA_METER_OF_MERCURT

1 미터 높이 수은의 압력

GSL_CONST_MKSA_INCH_OF_MERCURY

1 인치 높이 수은의 압력

GSL_CONST_MKSA_INCH_OF_WATER

1 인치 높이 물의 압력

GSL_CONST_MKSA__PSI

1 파운드의 제곱 인치당 압력.

45.12 밀도

GSL_CONST_MKSA_POISE

1 푸아스

GSL_CONST_MKSA_STOKES

1 스토크스

45.13 빛과 광량

GSL_CONST_MKSA_STILB

1 스틸브 휘도

GSL_CONST_MKSA_LUMEN

1 루멘

GSL_CONST_MKSA_LUX

1 럭스

GSL_CONST_MKSA_PHOT

1 포트

GSL_CONST_MKSA_FOOTCANDLE

1 푸트캔들

GSL_CONST_MKSA_LAMBERT

1 람베르트

GSL_CONST_MKSA_FOOTLAMBERT

1 푸트 람베르트

45.14 방사능

GSL_CONST_MKSA_CURIE

1 퀴리

GSL_CONST_MKSA_ROENTGEN

1 뢴트겐

GSL_CONST_MKSA_RAD

1 라디. (방사선 흡수선량; Radiation Absorbed Dose)

45.15 힘과 에너지

GSL_CONST_MKSA_NEWTON

1 뉴턴

GSL_CONST_MKSA_DYNE

1 다인

GSL_CONST_MKSA_JOULE

1 줄

GSL_CONST_MKSA_ERG

1 에르그. (1 erg = 10^{-7} 줄)

45.16 접두사

GSL_CONST_NUM_YOTTA

10^{24}

GSL_CONST_NUM_ZETTA

10^{21}

GSL_CONST_NUM_EXA

10^{18}

GSL_CONST_NUM_PETA

10^{15}

GSL_CONST_NUM_TERA

10^{12}

GSL_CONST_NUM_GIGA

10^{9}

GSL_CONST_NUM_MEGA

10^{6}

GSL_CONST_NUM_KILO

10^{5}

GSL_CONST_NUM_MILLI

10^{-3}

GSL_CONST_NUM_MICRO

10^{-6}

GSL_CONST_NUM_NANO

10^{-9}

GSL_CONST_NUM_PICO

10^{-12}

GSL_CONST_NUM_FEMTO

10^{-15}

GSL_CONST_NUM_ATTO

10^{-18}

GSL_CONST_NUM_ZEPTO

10^{-21}

GSL_CONST_NUM_YOCTO

10^{-24}

45.17 예제

다음 예제는 계산에서 물리상수들을 사용하는 방법을 보여줍니다. 이 예제는, 지구에서 화성까지 빛이 이동하는 시간 범위를 구하는 방법을 보여줍니다.

이때, 행성이 태양으로 부터 떨어진 평균 거리를 천문 단위로 나타낸 데이터가 필요합니다. 궤도의 기울기와 타원 궤도의 편향은 무시됩니다. 화성의 평균 궤도 반지름은 천문 단위로 1.52입니다. 지구는 1입니다. 이 값들은 MKSA 단위 상수들과 함께 사용되어, 가장 짧은 이동 시간과 가장 긴 이동시간을 초단위로 계산할 수 있고, 화면에는 분으로 바뀌어 표시됩니다

```
#include <stdio.h>
#include <gsl/gsl_const_mksa.h>

int
main (void)
{
  double c  = GSL_CONST_MKSA_SPEED_OF_LIGHT;
  double au = GSL_CONST_MKSA_ASTRONOMICAL_UNIT;
  double minutes = GSL_CONST_MKSA_MINUTE;

  /* distance stored in meters */
  double r_earth = 1.00 * au;
  double r_mars  = 1.52 * au;

  double t_min, t_max;

  t_min = (r_mars - r_earth) / c;
  t_max = (r_mars + r_earth) / c;

  printf ("light travel time from Earth to Mars:\n");
  printf ("minimum = %.1f minutes\n", t_min / minutes);
  printf ("maximum = %.1f minutes\n", t_max / minutes);

  return 0;
}
```

다음은 프로그램의 실행 결과입니다

```
light travel time from Earth to Mars:
minimum = 4.3 minutes
maximum = 21.0 minutes
```

제 46 장

IEEE 부동 소수점 대수

이 단원에서는 부동 소수 표현을 검증하고, 제어할 수 있는 함수들에 대해 기술합니다. 함수들은 헤더 파일 `gsl_ieee_utils.h` 에 정의되어 있습니다

46.1 부동 소수점의 표현

IEEE 표준 이진 부동 소수점 대수에서는 단, 배정밀도의 이진 숫자 표현에 대해 정의하고 있습니다. 각 수는 3가지 부분으로 나뉘어 표현됩니다. 부호(s), 지수(E), 그리고 소수점(f)입니다. (s, E, f) 표현의 실제 실수 값은 다음과 같이 주어집니다.

$$(-1)^s (1 \cdot ffff\ldots) 2^E$$

부호부는 0이거나 1입니다. 지수부의 하한 E_{min} 과 상한 E_{max} 는 정밀도에 따라 달라집니다. 이 지수는 부호 없는 수 e 로 변환될 수 있습니다. 이는 바이어스 지수부라 불리며, 바이어스 계수 $bias$ 를 더해 저장됩니다.

$$e = E + bias$$

배열 $ffff\ldots$ 는 이진수의 자릿수를 나타냅니다. 이 이진 자릿수는 정규화된 형태로 저장되는데, 지수를 조정해 처음 자리에 1이 오게합니다. 정규화된 숫자는 처음 숫자가 항상 1이 온다고 암묵적으로 가정되기 때문에 저장되지 않습니다. 만일, 숫자가 $2^{E_{min}}$ 보다 작다면, 처음 숫자가 0인 비 정규화된 형태로 저장됩니다.

$$(-1)^s (0 \cdot ffff\ldots) 2^E$$

IEEE의 이진 부동 소수 대수 표준은 다음과 같이 정의됩니다.

p 비트의 정밀도에서 점진적 언더플로우 값으로 $2^{E_{min}-p}$ 를 반환합니다.

0은 $2^{E_{min}-1}$ 형태로 특정 지수 값으로 정의됩니다. 같은 방식으로 무한대 ∞ 도 $2^{E_{max}+1}$ 로 정의됩니다.

32bit를 사용하는 단정밀도 숫자 표현은 다음과 같습니다.

```
seeeeeeeeffffffffffffffffffffff

s = sign bit, 1 bit
e = exponent, 8 bits  (E_min=-126, E_max=127, bias=127)
f = fraction, 23 bits
```

64bit를 사용하는 배정밀도 숫자 표현은 다음과 같습니다.

```
seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff

s = sign bit, 1 bit
e = exponent, 11 bits  (E_min=-1022, E_max=1023, bias=1023)
f = fraction, 52 bits
```

이러한 bit 수준의 연산 단계를 확인할 수 있는 기능이 있으면 유용합니다. 이 라이브러리에서는 IEEE 표현을 읽기 쉬운 형태로 출력해주는 함수들을 제공합니다.

void **gsl_ieee_fprintf_float**(FILE *stream, const float *x)

void **gsl_ieee_fprintf_double**(FILE *stream, const double *x)

이 함수들은 주어진 값 `x` 의 IEEE 부동 소수점 숫자 형식을 스트림 `stream` 에 출력합니다. 포인터를 사용한 이유는 값을 간접적으로 넘겨 `float` 에서 `double` 의 형 변환을 방지하기 위함입니다. 출력값은 다음의 형태를 가질 수 있습니다.

`NaN`

비정상값(Not a number)

`Inf, -Inf`

양, 음의 무한대

`1.fffff...*2^E, -1.fffff...*2^E`

정규회된 부동 소수 숫자

`0.fffff...*2^E, -0.fffff...*2^E`

비정규회된 부동 소수 숫자

`0, -0`

양, 음수 0

결과값은 `2#` 이라는 이진 표현을 표식자를 붙여주면 GNU Emacs Calc에 바로 사용가능합니다.

void **gsl_ieee_printf_float**(const float *x)

void **gsl_ieee_printf_double**(const double *x)

x 가 가르키는 숫자를 형식화 된 IEEE 부동소수점 표현으로 stdout 에 출력합니다.

다음 프로그램은 이 함수들을 사용해 1/3 를 단, 배 정밀도 표현으로 출력하는 방법을 보여줍니다. 비교를 위해 단 정밀도 표현을 배 정밀도로 변환한 값도 같이 출력됩니다.

For comparison the representation of the value promoted from single to double precision is also printed.

```
#include <stdio.h>
#include <gsl/gsl_ieee_utils.h>

int
main (void)
{
  float f = 1.0/3.0;
  double d = 1.0/3.0;

  double fd = f; /* promote from float to double */

  printf (" f="); gsl_ieee_printf_float(&f);
  printf ("\n");

  printf ("fd="); gsl_ieee_printf_double(&fd);
  printf ("\n");

  printf (" d="); gsl_ieee_printf_double(&d);
  printf ("\n");

  return 0;
}
```

1/3 의 이진 표현은 0.01010101... 입니다. 아래의 결과는 IEEE 형식의 정규화 표현에서 이 분수 값의 앞 자릿수는 1이라는 사실을 보여줍니다.

```
 f= 1.01010101010101010101011*2^-2
fd= 1.01010101010101010101011100000000000000000000000000000*2^-2
 d= 1.01010101010101010101010101010101010101010101010101010*2^-2
```

이 결과는 단 정밀도 숫자가 배 정밀도로 변환될 때, 이진 표현의 나머지 자릿수에 0을 넣어서 변환함을 보여줍니다.

46.2 IEEE 환경 설정

IEEE 표준은 부동 소수점 연산을 제어하기 위해 몇가지 특정 환경 상태 들을 정의합니다. 이 환경들은 컴퓨터의 연산의 중요한 성질들을 특정 짓습니다. 절단의 방향(해당 숫자를 인접한 숫자로 올릴지, 내릴지 결정합니다.), 반올림 정밀도, 그리고 연산의 예외(예: 0으로 나누기)를 어떻게 처리할 지 등이 있습니다.

이러한 성질들의 상당수는 fpsetround() 와 같은 표준 함수들로 제어됩니다. 물론, 이러한 표준 함수들이 사용 가능할 때에 한정된 상황입니다. 불행히도, 옛날에는 이러한 행동들을 제어할만한 표준 API가 없었습니다. 각각의 시스템들은 이러한 행동들을 저수준에서 직접 접근해 사용했습니다. 프로그램의 이식성 구현을 돕기 위해 GSL에서는 플랫폼에 독립적으로 이러한 환경을 제어할 수 있는 환경 변수 GSL_IEEE_MODE 을 제공합니다. gsl_ieee_env_setup() 함수를 호출하면 라이브러리는 기계에 설정되어 있는 필요한 모든 사항들을 초기화합니다.

GSL_IEEE_MODE

IEEE 환경 상태를 특정짓는 환경 변수입니다.

void **gsl_ieee_env_setup**()

환경 변수 GSL_IEEE_MODE 를 읽고 대응 되는 특정 IEEE 환경으로 시스템을 설정합니다. 환경 변수는 다음의 목록에 있는 값들로 구성되어야 하며, 쉼표로 구분됩니다. 다음과 같이 작성할 수 있습니다.

```
GSL_IEEE_MODE = "keyword, keyword, ..."
```

keyword 는 다음의 상태 이름들로 작성해야 합니다.

```
single-precision
double-precision
extended-precision
round-to-nearest
round-down
round-up
round-to-zero
mask-all
mask-invalid
mask-denormalized
mask-division-by-zero
mask-overflow
mask-underflow
trap-inexact
trap-common
```

GSL_IEEE_MODE 가 비어있거나 정의되지 않으면, 이 함수는 즉시 종료되며 시스템의 IEEE 환경을 바꾸지 않습니다. GSL_IEEE_MODE 에 정의된 상태들이 함수에 의해 설정되면 이를 알리기 위해 짧은 알림 메세지가 출력되고 해당 IEEE 환경이 적용됩니다.

요구한 환경들을 플랫폼에서 지원하지 않는다면 이 함수는 오류 관리자를 부르고 GSL_EUNSUP 오류 값을 반환합니다.

이 방법을 통해 IEEE 환경을 설정할 경우, 설정된 환경 설정들은 기본적으로 round-to-nearest 환경에서 사용가능한 한 가장 높은 정밀도를 가지는 상태로 특정됩니다. (이 상태는 배정밀도거나 확장된 정밀도를 가지는 등 플랫폼에 따라 다양합니다.) 또한, INEXACT 예외를 제외한 모든 예외가 활성화 됩니다. INEXACT 예외는 절단이 발생하는 상황에서 발생하기 때문에 일반적인 과학 계산상황에서 비활성화 됩니다. 이들은 독립적으로 mask- 가 붙은 환경 설정으로 비활성화 하거나 mask-all 로 모두 비활성화 시킬 수 있습니다.

다음의 환경 변수 설정이 많은 상황에서 공통적으로 쓰입니다.

```
GSL_IEEE_MODE="double-precision,"\
              "mask-underflow,"\
                "mask-denormalized"
```

이 설정은 작은 숫자들에 관련된 오류들(비정규화나 0으로 언더플로우)을 무시합니다. 하지만, 오버플로우나 0으로 나누기 그리고 정의되지 않은 연산들을 잡아낼 수 있습니다.

참고: x86 계열의 프로세스들에서 이 함수는 본래의 x87환경과 새로운 MXCSR 환경을 모두 설정합니다. 이들은 SSE 부동 소수 연산을 제어합니다. SSE 부동 소수 유닛는 정밀도 제어 비트를 가지고 있지않고 항상 배 정밀도로 연산합니다. 단 정밀도와 확장된 정밀도 설정은 이러한 경우 어떠한 영향도 미치지 않습니다.

각기 다른 절단 환경 설정이 계산에 미치는 영향을 보여주기 위해 다음 프로그램은 e 를 계산하는 방법을 자연 로그와 급격히 감소하는 급수를 이용해 보여줍니다.

$$\begin{aligned} e &= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots \\ &= 2.71828182846\ldots \end{aligned}$$

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_ieee_utils.h>

int
main (void)
{
  double x = 1, oldsum = 0, sum = 0;
  int i = 0;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  gsl_ieee_env_setup (); /* read GSL_IEEE_MODE */

  do
    {
      i++;

      oldsum = sum;
      sum += x;
      x = x / i;

      printf ("i=%2d sum=%.18f error=%g\n",
              i, sum, sum - M_E);

      if (i > 30)
        break;
    }
  while (sum != oldsum);

  return 0;
}
```

다음 결과는 round-to-nearest 설정에서 나온 결과입니다. 이는 IEEE 대수의 기본 설정이기 때문에 달리 설명할 사항은 없습니다.

```
$ GSL_IEEE_MODE="round-to-nearest" ./a.out
i= 1 sum=1.000000000000000000 error=-1.71828
i= 2 sum=2.000000000000000000 error=-0.718282
....
i=18 sum=2.718281828459045535 error=4.44089e-16
i=19 sum=2.718281828459045535 error=4.44089e-16
```

19번째 항 이후로 이 급수는 참 값에 4×10^{-16} 내의 오차로 수렴합니다.

이제 절단 설정을 round-down 로 바꾸어 절단을 내림으로 정의합시다. 이 경우 최종 결과값이 이전 계산보다 덜 정확하게 나옵니다.

> $ GSL_IEEE_MODE=”round-down” ./a.out i= 1 sum=1.000000000000000000 error=-1.71828 …. i=19 sum=2.718281828459041094 error=-3.9968e-15

결과 값이 참 값에 4×10^{-15} 내의 오차로 수렴합니다. round-to-nearest 설정의 결과보다 크기 측면에서 더 안좋은 결과가 나왔습니다.

절단 설정을 round-up 로 바꾸어 올림으로 정의할 경우 최종 결과값은 참 값보다 크게 나옵니다. (각 항을

더할 때마다 계산 결과 값이 항상 올려집니다. 이는 추가된 항이 0으로 언더플로우 되기 전까지 적어도 합을 최소 단위의 1 이상 증가시킵니다.) 이를 피하기 위해 적절하게 정의된 epsilon 값을 이용해 while (fabs(sum - oldsum) > epsilon) 형태의 수렴 조건을 만들어야 합니다.

마지막으로, 단 정밀도 절단을 이용한 합 계산을 봅시다. 기본 설정은 round-to-nearest 환경입니다. 이 경우 프로그램은 여전이 배 정밀도로 숫자를 계산하고 있다고 간주하지만 CPU에서 각각의 부동 소수 연산을 단 정밀도의 정밀도로 절단합니다. 이는 프로그램을 double 변수들로 작성하는 대신 float 으로 작성했을 때를 시뮬레이션할 수 있습니다. 반복문이 절반의 횟수에서 종료되고 최종 계산 값은 더 낮은 정확도를 가집니다.

```
$ GSL_IEEE_MODE="single-precision" ./a.out
....
i=12 sum=2.718281984329223633 error=1.5587e-07
```

오차는 $O(10^{-7})$ 로 단 정밀도의 정확도(10^7 크기의 숫자)에 대응 되는 값입니다. 반복문을 계속 수행하는 행위는 의미가 없습니다. 이후의 모든 합이 동일한 값으로 절단되기 때문입니다.

46.3 참고 문헌과 추가 자료

IEEE 표준에 관한 참고 문헌은 다음을 볼 수 있습니다.

- ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

다음 논문들에서 표준에 관해 더 많은 교육 자료를 볼 수 있습니다.

- David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Computing Surveys, Vol.: 23, No.: 1 (March 1991), pages 5–48.
- Corrigendum: ACM Computing Surveys, Vol.: 23, No.: 3 (September 1991), page 413. and see also the sections by B. A. Wichmann and Charles B. Dunham in Surveyor's Forum: "What Every Computer Scientist Should Know About Floating-Point Arithmetic". ACM Computing Surveys, Vol.: 24, No.: 3 (September 1992), page 319.

SIAM 출판에서 IEEE 대수에 관한 내용과 실용적 사용의 예시 교재를 찾을 수 있습니다.

- Michael L. Overton, Numerical Computing with IEEE Floating Point Arithmetic, SIAM Press, ISBN 0898715717.

제 47 장

수치 해석 프로그램의 디버깅

참고: 번역중

이 단원은 GSL을 사용한 프로그램의 디버깅에 사용할 수 있는 몇가지 조언과 우회법을 소개합니다.

47.1 GDB 사용

라이브러리에서 생기는 모든 오류는 `gsl_error()` 함수로 전달됩니다. GDB를 사용한 디버그 모드에서 프로그램을 구동시키고 이 함수에 중단점을 지정핻면 라이브러리에서 생기는 모든 오류를 감지할 수 있습니다. 다음의 내용을 프로그램이 있는 디렉토리의 파일 `.gdbinit` 에 넣어서 모든 세션에 중단점을 넣을 수 있습니다.

You can add a breakpoint for every session by putting

```
break gsl_error
```

into your `.gdbinit` file in the directory where your program is started.

중단점이 오류를 감지하면 backtrace(`bt`)를 사용해 호출-트리와 오류를 유발한 인자를 확인할 수 있습니다. If the breakpoint catches an error then you can use a backtrace (`bt`) to see the call-tree, and the arguments which possibly caused the error. By moving up into the calling function you can investigate the values of variables at that point. Here is an example from the program `fft/test_trap`, which contains the following line:

```
status = gsl_fft_complex_wavetable_alloc (0, &complex_wavetable);
```

The function `gsl_fft_complex_wavetable_alloc()` takes the length of an FFT as its first argument. When this line is executed an error will be generated because the length of an FFT is not allowed to be zero.

To debug this problem we start `gdb`, using the file `.gdbinit` to define a breakpoint in `gsl_error()`:

```
$ gdb test_trap

GDB is free software and you are welcome to distribute copies
of it under certain conditions; type "show copying" to see
the conditions.  There is absolutely no warranty for GDB;
type "show warranty" for details.  GDB 4.16 (i586-debian-linux),
Copyright 1996 Free Software Foundation, Inc.

Breakpoint 1 at 0x8050b1e: file error.c, line 14.
```

When we run the program this breakpoint catches the error and shows the reason for it:

```
(gdb) run
Starting program: test_trap

Breakpoint 1, gsl_error (reason=0x8052b0d
    "length n must be positive integer",
    file=0x8052b04 "c_init.c", line=108, gsl_errno=1)
    at error.c:14
14        if (gsl_error_handler)
```

The first argument of `gsl_error()` is always a string describing the error. Now we can look at the backtrace to see what caused the problem:

```
(gdb) bt
#0  gsl_error (reason=0x8052b0d
    "length n must be positive integer",
    file=0x8052b04 "c_init.c", line=108, gsl_errno=1)
    at error.c:14
#1  0x8049376 in gsl_fft_complex_wavetable_alloc (n=0,
    wavetable=0xbffff778) at c_init.c:108
#2  0x8048a00 in main (argc=1, argv=0xbffff9bc)
    at test_trap.c:94
#3  0x80488be in ___crt_dummy__ ()
```

We can see that the error was generated in the function `gsl_fft_complex_wavetable_alloc()` when it was called with an argument of `n = 0`. The original call came from line 94 in the file

test_trap.c.

By moving up to the level of the original call we can find the line that caused the error:

```
(gdb) up
#1  0x8049376 in gsl_fft_complex_wavetable_alloc (n=0,
    wavetable=0xbffff778) at c_init.c:108
108   GSL_ERROR ("length n must be positive integer", GSL_EDOM);
(gdb) up
#2  0x8048a00 in main (argc=1, argv=0xbffff9bc)
  at test_trap.c:94
94    status = gsl_fft_complex_wavetable_alloc (0,
        &complex_wavetable);
```

Thus we have found the line that caused the problem. From this point we could also print out the values of other variables such as complex_wavetable.

47.2 Examining floating point registers

The contents of floating point registers can be examined using the command info float (on supported platforms):

```
(gdb) info float
     st0: 0xc4018b895aa17a945000  Valid Normal -7.838871e+308
     st1: 0x3ff9ea3f50e4d7275000  Valid Normal 0.0285946
     st2: 0x3fe790c64ce27dad4800  Valid Normal 6.7415931e-08
     st3: 0x3ffaa3ef0df6607d7800  Spec  Normal 0.0400229
     st4: 0x3c028000000000000000  Valid Normal 4.4501477e-308
     st5: 0x3ffef5412c22219d9000  Zero  Normal 0.9580257
     st6: 0x3fff8000000000000000  Valid Normal 1
     st7: 0xc4028b65a1f6d243c800  Valid Normal -1.566206e+309
   fctrl: 0x0272 53 bit; NEAR; mask DENOR UNDER LOS;
   fstat: 0xb9ba flags 0001; top 7; excep DENOR OVERF UNDER LOS
    ftag: 0x3fff
     fip: 0x08048b5c
     fcs: 0x051a0023
  fopoff: 0x08086820
  fopsel: 0x002b
```

Individual registers can be examined using the variables $reg, where reg is the register name:

```
(gdb) p $st1
$1 = 0.028594644542612103477 19
```

47.3 Handling floating point exceptions

It is possible to stop the program whenever a SIGFPE floating point exception occurs. This can be useful for finding the cause of an unexpected infinity or NaN. The current handler settings can be shown with the command info signal SIGFPE:

```
(gdb) info signal SIGFPE
Signal  Stop  Print  Pass to program Description
SIGFPE  Yes   Yes    Yes             Arithmetic exception
```

Unless the program uses a signal handler the default setting should be changed so that SIGFPE is not passed to the program, as this would cause it to exit. The command handle SIGFPE stop nopass prevents this:

```
(gdb) handle SIGFPE stop nopass
Signal  Stop  Print  Pass to program Description
SIGFPE  Yes   Yes    No              Arithmetic exception
```

Depending on the platform it may be necessary to instruct the kernel to generate signals for floating point exceptions. For programs using GSL this can be achieved using the GSL_IEEE_MODE environment variable in conjunction with the function gsl_ieee_env_setup() as described in IEEE 부동 소수점 대수:

```
(gdb) set env GSL_IEEE_MODE=double-precision
```

47.4 GCC warning options for numerical programs

Writing reliable numerical programs in C requires great care. The following GCC warning options are recommended when compiling numerical programs:

```
gcc -ansi -pedantic -Werror -Wall -W
  -Wmissing-prototypes -Wstrict-prototypes
  -Wconversion -Wshadow -Wpointer-arith
  -Wcast-qual -Wcast-align
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
-Wwrite-strings -Wnested-externs
-fshort-enums -fno-common -Dinline= -g -O2
```

For details of each option consult the manual Using and Porting GCC. The following table gives a brief explanation of what types of errors these options catch.

`-ansi -pedantic`

Use ANSI C, and reject any non-ANSI extensions. These flags help in writing portable programs that will compile on other systems.

`-Werror`

Consider warnings to be errors, so that compilation stops. This prevents warnings from scrolling off the top of the screen and being lost. You won't be able to compile the program until it is completely warning-free.

`-Wall`

This turns on a set of warnings for common programming problems. You need `-Wall`, but it is not enough on its own.

`-O2`

Turn on optimization. The warnings for uninitialized variables in `-Wall` rely on the optimizer to analyze the code. If there is no optimization then these warnings aren't generated.

`-W`

This turns on some extra warnings not included in `-Wall`, such as missing return values and comparisons between signed and unsigned integers.

`-Wmissing-prototypes -Wstrict-prototypes`

Warn if there are any missing or inconsistent prototypes. Without prototypes it is harder to detect problems with incorrect arguments.

`-Wconversion`

The main use of this option is to warn about conversions from signed to unsigned integers. For example, `unsigned int x = -1`. If you need to perform such a conversion you can use an explicit cast.

`-Wshadow`

This warns whenever a local variable shadows another local variable. If two variables have the same name then it is a potential source of confusion.

`-Wpointer-arith -Wcast-qual -Wcast-align`

These options warn if you try to do pointer arithmetic for types which don't have a size, such as `void`, if you remove a `const` cast from a pointer, or if you cast a pointer to a type which has a different size, causing an invalid alignment.

`-Wwrite-strings`

This option gives string constants a `const` qualifier so that it will be a compile-time error to attempt to overwrite them.

`-fshort-enums`

This option makes the type of `enum` as short as possible. Normally this makes an `enum` different from an `int`. Consequently any attempts to assign a pointer-to-int to a pointer-to-enum will generate a cast-alignment warning.

`-fno-common`

This option prevents global variables being simultaneously defined in different object files (you get an error at link time). Such a variable should be defined in one file and referred to in other files with an `extern` declaration.

`-Wnested-externs`

This warns if an `extern` declaration is encountered within a function.

`-Dinline=`

The `inline` keyword is not part of ANSI C. Thus if you want to use `-ansi` with a program which uses inline functions you can use this preprocessor definition to remove the `inline` keywords.

`-g`

It always makes sense to put debugging symbols in the executable so that you can debug it using `gdb`. The only effect of debugging symbols is to increase the size of the file, and you can use the `strip` command to remove them later if necessary.

47.5 References and Further Reading

The following books are essential reading for anyone writing and debugging numerical programs with `gcc` and `gdb`.

- R.M. Stallman, Using and Porting GNU CC, Free Software Foundation, ISBN 1882114388
- R.M. Stallman, R.H. Pesch, Debugging with GDB: The GNU Source-Level Debugger, Free Software Foundation, ISBN 1882114779

For a tutorial introduction to the GNU C Compiler and related programs, see

- B.J. Gough, http://www.network-theory.co.uk/gcc/intro/,' An Introduction to GCC, Network Theory Ltd, ISBN 0954161793

제 가 부록

Autoconf 매크로

autoconf 를 사용하는 응용 프로그램들은 표준 매크로 AC_CHECK_LIB 를 사용해 configure 스크립트에서 GSL를 자동으로 링크할 수 있습니다. 라이브러리는 CBLAS와 수학 라이브러리들에 대해서도 자체적으로 의존하고 있기 때문에 의존 라이브러리들을 libgsl 을 링크하기 전에 가져오도록 해야합니다. 이를 수행할 수 있는 명령어는 다음과 같습니다. 명령어들은 configure.ac 파일에 작성되어야합니다.

```
AC_CHECK_LIB([m],[cos])
AC_CHECK_LIB([gslcblas],[cblas_dgemm])
AC_CHECK_LIB([gsl],[gsl_blas_dgemm])
```

libm 와 libgslcblas 를 libgsl 이전에 배치시켜야 함을 기억해야합니다. 다른 순서로 배열하면 작동하지 않을 것입니다. 구성(configure) 단계에서 라이브러리를 찾을 수 있다면 다음과 같은 출력을 내보냅니다.

```
checking for cos in -lm... yes
checking for cblas_dgemm in -lgslcblas... yes
checking for gsl_blas_dgemm in -lgsl... yes
```

라이브러리를 찾을 수 있다면 검증 프로그램이 매크로 HAVE_LIBGSL, HAVE_LIBGSLCBLAS, HAVE_LIBM 를 정의하고 변수 LIBS 에 -lgsl -lgslcblas -lm 옵션들을 추가합니다.

위의 검증 과정은 어느 버전의 라이브러리에서도 사용 가능합니다. 이 과정은 보편적으로 사용할 수 있고, 함수의 버전은 중요하지 않습니다. 파일 내에는 대체 매크로 gsl.m4 가 있어 특정 라이브러리의 버전을 검사할 수도 있습니다. 이 매크로를 사용하기 위해서는 위의 검증 과정을 기술하는 명령어들 대신 다음의 명령어들을 configure.in 에 추가하면 됩니다.

```
AX_PATH_GSL(GSL_VERSION,
           [action-if-found],
           [action-if-not-found])
```

GSL_VERSION 인자는 major.minor 나 major.minor.micro 형식의 2-3 자리 정수 값을 가져야 합니다. 이는 필요한 라이브러리 베포판의 버전 숫자를 가르킵니다.

action-if-not-found 에 대해 일반적으로 많이 쓰이는 설정은

```
AC_MSG_ERROR(could not find required version of GSL)
```

Makefile.am 파일에 변수 GSL_LIBS 와 GSL_CFLAGS 를 추가해 제대로 된 컴파일러 옵션들을 얻을 수 있습니다. GSL_LIBS 는 gsl-config --libs 를 만들고, GSL_CFLAGS 는 gsl-config --cflags 를 만들어줍니다. 예를 들어서

```
libfoo_la_LDFLAGS = -lfoo $(GSL_LIBS) -lgslcblas
```

참고: 매크로 AX_PATH_GSL 는 C 컴파일러를 필요로 함을 유의해야 합니다. 따라서 이 매크로는 configure.in 파일 내에 매크로 AC_LANG_CPLUSPLUS 전에 기술되어야 합니다. 이 매크로는 C++로 쓰인 프로그램에 쓰입니다.

---

inline 기능을 검사하기 위해서는 다음의 명령어들을 configure.in 에 작성하고,

```
AC_C_INLINE

if test "$ac_cv_c_inline" != no ; then
  AC_DEFINE(HAVE_INLINE,1)
  AC_SUBST(HAVE_INLINE)
fi
```

이러면 매크로가 컴파일 옵션에 정의됩니다 아니면 다른 라이브러리 헤더들 보다 config.h 를 먼저 포함하는 형식으로 검사할 수도 있습니다.

다음 autoconf 검증 명령어들은 extern inline 를 검사합니다.

```
dnl Check for "extern inline", using a modified version
dnl of the test for AC_C_INLINE from acspecific.mt
dnl
AC_CACHE_CHECK([for extern inline], ac_cv_c_extern_inline,
[ac_cv_c_extern_inline=no
AC_TRY_COMPILE([extern $ac_cv_c_inline double foo(double x);
extern $ac_cv_c_inline double foo(double x) { return x+1.0; };
double foo (double x) { return x + 1.0; };],
[  foo(1.0)  ],
[ac_cv_c_extern_inline="yes"])
])
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
if test "$ac_cv_c_extern_inline" != no ; then
  AC_DEFINE(HAVE_INLINE,1)
  AC_SUBST(HAVE_INLINE)
fi
```

autoconf 를 사용하면 대체 함수들을 자동으로 적용 되도록 할 수 있습니다. 예를 들어, BSD 함수 hypot 이 사용가능한지 아닌지 확인하기 위해서 작성하는 응용프로그램의 설정파일 configure.in 에 다음 명령어를 넣을 수 있습니다.

```
AC_CHECK_FUNCS(hypot)
```

그리고 파일 config.h.in 에 다음 매크로를 정의해 줍시다.

```
/* Substitute gsl_hypot for missing system hypot */

#ifndef HAVE_HYPOT
#define hypot gsl_hypot
#endif
```

이 과정을 거치면 응용 프로그램 소스 파일들에서 #include <config.h> 를 사용해 :fun`hypot` 를 사용할 수 없는 상황에서 :fun`gsl_hypot` 로 :fun`hypot` 를 대체할 수 있습니다.

제 나 부록

GSL의 기여자들

최신 정보는 베포판의 AUTHORS 파일을 참고해야 합니다.

Mark Galassi

James Theiler와 함께 GSL 프로젝트 시작, 디자인 문서 작성, 담금질 패키지 구현과 사용 설명서의 관련 단원 작성.

James Theiler

Mark Galassi와 함께 GSL 프로젝트 시작, 난수 생성 패키지 구현과 매뉴얼의 관련 단원 작성.

Jim Davies

통계 처리 기능 구현과 사용 설명서의 관련 단원 작성.

Brian Gough

FFTs, 수치 적분, 난수 생성과 분포, 근 탐색, 최소화, 피팅, 다항식 풀이, 복소수, 물리 상수, 순열, 벡터와 행렬 함수, 히스토그램, 통계, ieee-유틸, CBLAS의 Level 2&3 단계 재작성, 행렬 분해, 고유계, 누적 분포 함수, 검사와 사용 설명서작성, 베포.

Reid Priedhorsky

Los Alamos National Laboratory, Mathematical Modeling and Analysis Group 재직 당시, 근 탐색 기능들의 초기형 구현과 문서화.

Gerard Jungman

특수 함수들, 급수 가속, ODEs, BLAS, 선형 대수, 고유계, 한켈 변환.

Patrick Alken

비 대칭, 일반화된 고유 공간, B-스플라인, 선형, 비선형 최소제곱 방법, 행렬 분해, 버금 르장드르 함수, 통계 실행, 희소 행렬, 희소 선형 대수학 기능들의 구현.

Mike Booth

몬테 카를로 라이브러리 작성.

Jorma Olavi Tähtinen

복소 대수 함수 초안 구현.

Thomas Walter

힙-정렬 초안 구현 및 촘스키 분해 작성.

Fabrice Rossi

다변수 함수 최소화 기능 구현.

Carlo Perassi

Knuth's Seminumerical Algorithms, 3rd Ed의 무작위 숫자 생성 기능 구현.

Szymon Jaroszewicz

조합 생성 기능 구현.

Nicolas Darnis

cyclic-함수와 canonical 순열의 초기화 함수 구현.

Jason H. Stover

주요 누적 분포 함수 기능 구현.

Ivo Alxneit

Wavelet 변환 구현.

Tuomo Keskitalo

상미분 방정식 구현체 개선 및 ode-initval2 기능 작성.

Lowell Johnson

마티유 함수 구현.

Rhys Ulerich

중복 집합 기능 구현.

Pavel Holoborodko

고정차수 가우스-르장드르 구적법 구현.

Pedro Gonnet

CQUAD 적분 기능들 구현.

사용 설명서를 검토해 준 **Nigel Lowry** 에게 감사드립니다.

비-대칭 고유 공간 함수들은 LAPACK 선형 대수 라이브러리에 기반해있습니다. LAPACK은 다음의 허가서 아래에 베포됩니다.

License for LAPACK

제 다 부록

GSL CBLAS 라이브러리

저수준의 CBLAS 함수들은 헤더 파일 `gsl_cblas.h` 에 저장되어 있습니다. 이 함수들의 자세한 정의는 Netlib의 문헌을 참고하길 바랍니다. BLAS 지원의 참고 문헌 에 있습니다.

다.1 Level 1

float **cblas_sdsdot**(const int N, const float alpha, const float *x, const int incx, const float *y, const int incy)

double **cblas_dsdot**(const int N, const float *x, const int incx, const float *y, const int incy)

float **cblas_sdot**(const int N, const float *x, const int incx, const float *y, const int incy)

double **cblas_ddot**(const int N, const double *x, const int incx, const double *y, const int incy)

void **cblas_cdotu_sub**(const int N, const void *x, const int incx, const void *y, const int incy, void *dotu)

void **cblas_cdotc_sub**(const int N, const void *x, const int incx, const void *y, const int incy, void *dotc)

void **cblas_zdotu_sub**(const int N, const void *x, const int incx, const void *y, const int incy, void *dotu)

void **cblas_zdotc_sub**(const int N, const void *x, const int incx, const void *y, const int incy, void *dotc)

float **cblas_snrm2**(const int N, const float *x, const int incx)

float **cblas_sasum**(const int N, const float *x, const int incx)

double **cblas_dnrm2**(const int N, const double *x, const int incx)

double **cblas_dasum**(const int N, const double *x, const int incx)

float **cblas_scnrm2**(const int N, const void *x, const int incx)

float **cblas_scasum**(const int N, const void *x, const int incx)

double **cblas_dznrm2**(const int N, const void *x, const int incx)

double **cblas_dzasum**(const int N, const void *x, const int incx)

CBLAS_INDEX **cblas_isamax**(const int N, const float *x, const int incx)

CBLAS_INDEX **cblas_idamax**(const int N, const double *x, const int incx)

CBLAS_INDEX **cblas_icamax**(const int N, const void *x, const int incx)

CBLAS_INDEX **cblas_izamax**(const int N, const void *x, const int incx)

void **cblas_sswap**(const int N, float *x, const int incx, float *y, const int incy)

void **cblas_scopy**(const int N, const float *x, const int incx, float *y, const int incy)

void **cblas_saxpy**(const int N, const float alpha, const float *x, const int incx, float *y, const int incy)

void **cblas_dswap**(const int N, double *x, const int incx, double *y, const int incy)

void **cblas_dcopy**(const int N, const double *x, const int incx, double *y, const int incy)

void **cblas_daxpy**(const int N, const double alpha, const double *x, const int incx, double *y, const int incy)

void **cblas_cswap**(const int N, void *x, const int incx, void *y, const int incy)

void **cblas_ccopy**(const int N, const void *x, const int incx, void *y, const int incy)

void **cblas_caxpy**(const int N, const void *alpha, const void *x, const int incx, void *y, const int incy)

void **cblas_zswap**(const int N, void *x, const int incx, void *y, const int incy)

void **cblas_zcopy**(const int N, const void *x, const int incx, void *y, const int incy)

void **cblas_zaxpy**(const int N, const void *alpha, const void *x, const int incx, void *y, const int incy)

void **cblas_srotg**(float *a, float *b, float *c, float *s)

void **cblas_srotmg**(float *d1, float *d2, float *b1, const float b2, float *P)

void **cblas_srot**(const int N, float *x, const int incx, float *y, const int incy, const float c, const float s)

void **cblas_srotm**(const int N, float *x, const int incx, float *y, const int incy, const float *P)

void **cblas_drotg**(double *a, double *b, double *c, double *s)

void **cblas_drotmg**(double *d1, double *d2, double *b1, const double b2, double *P)

void **cblas_drot**(const int N, double *x, const int incx, double *y, const int incy, const double c, const double s)

void **cblas_drotm**(const int N, double *x, const int incx, double *y, const int incy, const double *P)

void **cblas_sscal**(const int N, const float alpha, float *x, const int incx)

void **cblas_dscal**(const int N, const double alpha, double *x, const int incx)

void **cblas_cscal**(const int N, const void *alpha, void *x, const int incx)

void **cblas_zscal**(const int N, const void *alpha, void *x, const int incx)

void **cblas_csscal**(const int N, const float alpha, void *x, const int incx)

void **cblas_zdscal**(const int N, const double alpha, void *x, const int incx)

다.2 Level 2

void **cblas_sgemv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const float alpha, const float *A, const int lda, const float *x, const int incx, const float beta, float *y, const int incy)

void **cblas_sgbmv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const float alpha, const float *A, const int lda, const float *x, const int incx, const float beta, float *y, const int incy)

void **cblas_strmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *A, const int lda, float *x, const int incx)

void **cblas_stbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const float *A, const int lda, float *x, const int incx)

void **cblas_stpmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *Ap, float *x, const int incx)

void **cblas_strsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *A, const int lda, float *x, const int incx)

void **cblas_stbsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const float *A, const int lda, float *x, const int incx)

void **cblas_stpsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *Ap, float *x, const int incx)

void **cblas_dgemv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const double alpha, const double *A, const int lda, const double *x, const int incx, const double beta, double *y, const int incy)

void **cblas_dgbmv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const double alpha, const double *A, const int lda, const double *x, const int incx, const double beta, double *y, const int incy)

void **cblas_dtrmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *A, const int lda, double *x, const int incx)

void **cblas_dtbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const double *A, const int lda, double *x, const int incx)

void **cblas_dtpmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *Ap, double *x, const int incx)

void **cblas_dtrsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *A, const int lda, double *x, const int incx)

void **cblas_dtbsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const double *A, const int lda, double *x, const int incx)

void **cblas_dtpsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *Ap, double *x, const int incx)

void **cblas_cgemv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_cgbmv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_ctrmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int lda, void *x, const int incx)

void **cblas_ctbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void *A, const int lda, void *x, const int incx)

void **cblas_ctpmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *x, const int incx)

void **cblas_ctrsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int lda, void *x, const int incx)

void **cblas_ctbsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void *A, const int lda, void *x, const int incx)

void **cblas_ctpsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *x, const int incx)

void **cblas_zgemv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_zgbmv**(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_ztrmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int lda, void *x, const int incx)

void **cblas_ztbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void *A, const int lda, void *x, const int incx)

void **cblas_ztpmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *x, const int incx)

void **cblas_ztrsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int lda, void *x, const int incx)

void **cblas_ztbsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void *A, const int lda, void *x, const int incx)

void **cblas_ztpsv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *x, const int incx)

void **cblas_ssymv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const float *A, const int lda, const float *x, const int incx, const float beta, float *y, const int incy)

void **cblas_ssbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const int K, const float alpha, const float *A, const int lda, const float *x, const int incx, const float beta, float *y, const int incy)

void **cblas_sspmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const float *Ap, const float *x, const int incx, const float beta, float *y, const int incy)

void **cblas_sger**(const enum CBLAS_ORDER order, const int M, const int N, const float alpha, const float *x, const int incx, const float *y, const int incy, float *A, const int lda)

void **cblas_ssyr**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const float *x, const int incx, float *A, const int lda)

void **cblas_sspr**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const float *x, const int incx, float *Ap)

void **cblas_ssyr2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const float *x, const int incx, const float *y, const int incy, float *A, const int lda)

void **cblas_sspr2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const float *x, const int incx, const float *y, const int incy, float *A)

void **cblas_dsymv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const double *A, const int lda, const double *x, const int incx, const double beta, double *y, const int incy)

void **cblas_dsbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const int K, const double alpha, const double *A, const int lda, const double *x, const int incx, const double beta, double *y, const int incy)

void **cblas_dspmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const double *Ap, const double *x, const int incx, const double beta, double *y, const int incy)

void **cblas_dger**(const enum CBLAS_ORDER order, const int M, const int N, const double alpha, const double *x, const int incx, const double *y, const int incy, double *A, const int lda)

void **cblas_dsyr**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const double *x, const int incx, double *A, const int lda)

void **cblas_dspr**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const double *x, const int incx, double *Ap)

void **cblas_dsyr2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const double *x, const int incx, const double *y, const int incy, double *A, const int lda)

void **cblas_dspr2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const double *x, const int incx, const double *y, const int incy, double *A)

void **cblas_chemv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_chbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const int K, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_chpmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *Ap, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_cgeru**(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *A, const int lda)

void **cblas_cgerc**(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *A, const int lda)

void **cblas_cher**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const void *x, const int incx, void *A, const int lda)

void **cblas_chpr**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const float alpha, const void *x, const int incx, void *A)

void **cblas_cher2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *A, const int lda)

void **cblas_chpr2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *Ap)

void **cblas_zhemv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_zhbmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const int K, const void *alpha, const void *A, const int lda, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_zhpmv**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *Ap, const void *x, const int incx, const void *beta, void *y, const int incy)

void **cblas_zgeru**(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *A, const int lda)

void **cblas_zgerc**(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *A, const int lda)

void **cblas_zher**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const void *x, const int incx, void *A, const int lda)

void **cblas_zhpr**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const double alpha, const void *x, const int incx, void *A)

void **cblas_zher2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *A, const int lda)

void **cblas_zhpr2**(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const void *alpha, const void *x, const int incx, const void *y, const int incy, void *Ap)

다.3 Level 3

void **cblas_sgemm**(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const float alpha, const float *A, const int lda, const float *B, const int ldb, const float beta, float *C, const int ldc)

void **cblas_ssymm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const int N, const float alpha, const float *A, const int lda, const float *B, const int ldb, const float beta, float *C, const int ldc)

void **cblas_ssyrk**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const float *A, const int lda, const float beta, float *C, const int ldc)

void **cblas_ssyr2k**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const float *A, const int lda, const float *B, const int ldb, const float beta, float *C, const int ldc)

void **cblas_strmm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const float alpha, const float *A, const int lda, float *B, const int ldb)

void **cblas_strsm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const float alpha, const float *A, const int lda, float *B, const int ldb)

void **cblas_dgemm**(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const double alpha, const double *A, const int lda, const double *B, const int ldb, const double beta, double *C, const int ldc)

void **cblas_dsymm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const int N, const double alpha, const double *A, const int lda, const double *B, const int ldb, const double beta, double *C, const int ldc)

void **cblas_dsyrk**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const double *A, const int lda, const double beta, double *C, const int ldc)

void **cblas_dsyr2k**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const double *A, const int lda, const double *B, const int ldb, const double beta, double *C, const int ldc)

void **cblas_dtrmm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const double alpha, const double *A, const int lda, double *B, const int ldb)

void **cblas_dtrsm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const double alpha, const double *A, const int lda, double *B, const int ldb)

void **cblas_cgemm**(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_csymm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_csyrk**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const int lda, const void *beta, void *C, const int ldc)

void **cblas_csyr2k**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_ctrmm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb)

void **cblas_ctrsm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb)

void **cblas_zgemm**(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_zsymm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_zsyrk**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const int lda, const void *beta, void *C, const int ldc)

void **cblas_zsyr2k**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_ztrmm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb)

void **cblas_ztrsm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb)

void **cblas_chemm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_cherk**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const void *A, const int lda, const float beta, void *C, const int ldc)

void **cblas_cher2k**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const float beta, void *C, const int ldc)

void **cblas_zhemm**(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc)

void **cblas_zherk**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const void *A, const int lda, const double beta, void *C, const int ldc)

void **cblas_zher2k**(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const int lda, const void *B, const int ldb, const double beta, void *C, const int ldc)

void **cblas_xerbla**(int p, const char *rout, const char *form, ...)

다.4 예제

다음 은 3단계의 BLAS 함수 SGEMM 를 사용해 두 개의 행렬을 곱하는 예제를 보여줍니다.

$$\begin{pmatrix} 0.11 & 0.12 & 0.13 \\ 0.21 & 0.22 & 0.23 \end{pmatrix} \begin{pmatrix} 1011 & 1012 \\ 1021 & 1022 \\ 1031 & 1032 \end{pmatrix} = \begin{pmatrix} 367.76 & 368.12 \\ 674.06 & 674.72 \end{pmatrix}$$

행렬들은 행-기반 순서로 저장되어 있습니다. 하지만, 열-기반 순서로도 저장할 수 있습니다. cblas_sgemm() 의 첫번째 인자 값을 CblasColMajor 로 바꾸면 됩니다.

```
#include <stdio.h>
#include <gsl/gsl_cblas.h>

int
main (void)
{
  int lda = 3;

  float A[] = { 0.11, 0.12, 0.13,
                0.21, 0.22, 0.23 };

  int ldb = 2;

  float B[] = { 1011, 1012,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                1021, 1022,
                1031, 1032 };

  int ldc = 2;

  float C[] = { 0.00, 0.00,
                0.00, 0.00 };

  /* Compute C = A B */

  cblas_sgemm (CblasRowMajor,
               CblasNoTrans, CblasNoTrans, 2, 2, 3,
               1.0, A, lda, B, ldb, 0.0, C, ldc);

  printf ("[ %g, %g\n", C[0], C[1]);
  printf ("  %g, %g ]\n", C[2], C[3]);

  return 0;
}
```

프로그램의 컴파일은 다음 명령어를 사용면 됩니다.

```
$gcc -Wall demo.c -lgslcblas
```

이 프로그램은 컴파일 단계에서 메인 프로그램인 `-lgsl` 을 링크할 필요가 없습니다. 이는 CBLAS 라이브러리가 독립적으로 제공되기 때문입니다. 다음은 프로그램의 실행 결과입니다.

```
[ 367.76, 368.12
  674.06, 674.72 ]
```

제 라 부록

GNU 일반 공중 사용 허가서

경고: This is an unofficial translation of the GNU General Public License into Korean. It was not published by the Free Software Foundation, and does not legally state the distribution terms for software that uses the GNU GPL—only the original English text of the GNU GPL does that. However, we hope that this translation will help Korean speakers understand the GNU GPL better.

본 번역본은 GNU 일반 공중 사용 허가서의 비공식 한국어 번역본입니다. 이 문서는 자유 소프트웨어 재단에서 출판한 문서가 아니며 GNU GPL을 사용하는 소프트웨어들의 법적인 베포 규정을 기술하지 않습니다. 해당 사항들은 오직 원본 영어 GNU GPL 문서에 기반해 있어야 합니다. 그러나 이 번역본이 한국어 화자들에게 GNU GPL의 이해에 도움을 줄 수 있을 것이라 믿습니다.

자유 소프트웨어 재단은 공식적으로 어떠한 공식 번역본도 인정하지 않습니다. 실제 허가서를 사용하려면 변호사, 변리사등과 같은 전문적인 인력의 도움을 받아야합니다. 이 번역은 원본과 완전히 동일한 내용을 담았는 지에 대한 검증과 법률적 검토를 거치지 않았습니다. 이 번역은 이로 야기되는 모든 법적 문제에 대해 어떠한 보증도 하지 않습니다.

참고: 번역중

영어 원문은 이 단원의 끝 단락에 포함되어 있습니다.

라.1 영어 원문

이 전문은 다음의 GNU 공식 웹 사이트에서도 확인할 수 있습니다.

https://www.gnu.org/licenses/gpl-3.0.html

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<http://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<http://www.gnu.org/philosophy/why-not-lgpl.html>.
```

제 마 부록

GNU 자유 문서 사용 허가서

경고: This is an unofficial translation of the GNU Free Document License into Korean. It was not published by the Free Software Foundation, and does not legally state the distribution terms for software that uses the GNU GFDL—only the original English text of the GNU GFDL does that. However, we hope that this translation will help Korean speakers understand the GNU GFDL better.

본 번역본은 GNU 자유 문서 사용 허가서의 비공식 한국어 번역본입니다. 이 문서는 자유 소프트웨어 재단에서 발행한 문서가 아니며 GNU GFDL을 사용하는 소프트웨어들의 법적인 베포 규정을 기술하지 않습니다. 해당 사항들은 오직 원본 영어 GNU GFDL 문서에 기반해 있어야 합니다. 그러나 이 번역본이 한국어 화자들에게 GNU GFDL의 이해에 도움을 줄 수 있을 것이라 믿습니다.

자유 소프트웨어 재단은 공식적으로 어떠한 공식 번역본도 인정하지 않습니다. 실제 허가서를 사용하려면 변호사, 변리사등과 같은 전문적인 인력의 도움을 받아야합니다. 이 번역은 원본과 완전히 동일한 내용을 담았는 지에 대한 검증과 법률적 검토를 거치지 않았습니다. 이 번역은 이로 야기되는 모든 법적 문제에 대해 어떠한 보증도 하지 않습니다.

영어 원문은 이 단원의 끝 단락에 포함되어 있습니다.

마.1 0. 서문

이 허가서의 목적은 메뉴얼이나 서적, 혹은 다른 기능적 문서나 유용한 문서들을 “자유”롭게 만듬을 목표로 합니다. 이는 상업성과 수정의 여부와 관계 없이 자유롭게 이를 복사하고 재 배포할 수 있는 효과적인 자유를 보장한다는 뜻입니다. 이에 더해, 이 허가서는 저자와 발행자가 본인의 작업에 대한 이익과 타인의 수정으로 인한 사항에 책임이 없음을 보장합니다.

이 허가서는 “저좌권”의 한 종류입니다. 이는 이 문서의 파생 작업물들 또한 같은 의미로 자유로워야함을 의미합니다. 이는 GNU 일반 공중 사용 허가서와 같은 소프트웨어를 위한 저좌권을 보완합니다.

이 허가서는 자유 소프트웨어를 위한 메뉴얼에 사용되기 위해 설계되었습니다. 자유 문서는 자유 소프트웨어들에게 필수적이기 때문입니다. 자유 소프트웨어는 소프트웨어와 같은 자유도를 가지는 메뉴얼과 함께 제공되어야 합니다. 하지만, 이 허가서는 소프트웨어 메뉴얼에만 한정되지 않습니다. 주제를 불문하고 모든 발행물이나 문서에 사용할 수 있습니다. 교육이나 참조용 문서들에 본 허가서를 사용하는 것을 권장합니다.

마.2 1. 적용 대상과 정의

이 허가서는 저작권자가 본 허가서 아래에 베포된다는 내용을 작성할 수 있는 공간을 가진, 모든 메뉴얼이나 작업물에 사용될 수 있습니다. 이는 명시돤 조건 아래에서 전세계에서 기한의 제한없이, 저작물을 로얄티 없이 이용할 수 있는 허가를 부여합니다.

아래의 내용에서 “문서”는, 모든 메뉴얼이나 작업물들을 의미합니다. 허가서에 따른 문서를 사용하는 불특정 다수의 사람들은 “피양도자”라 표기될 것입니다. 저작권 법에 따른 허가가 필요한 행위로 저작물을 복사, 수정, 또는 배포하는 것은 피양도자가 허가서에 동의했다는 것을 의미합니다.

문서의 “수정본”은 원문서나 문서의 일부를 그대로 복제, 수정하거나 다른 언어로의 번역본을 포함하는 작업물을 의미합니다.

문서의 “2차 구성부”는 문서의 발행처나 저자가 문서가 다루는 전체 주제, 혹은 관련 내용 사이의 관계만을 설명하는 서두와 부록을 가르킵니다. 이 구성부는 문서의 전체 주제와 직접적인 관계를 서술하는 내용이 없어야합니다. (예를 들어서 문서가 수학 교과서일 경우 2차 구성부는 수학에 관한 어느 내용도 담아서는 안됩니다.) 2차 구성부가 담는 내용은 관련 내용과 주제에 대한 개정 이력이나, 법적, 상업적, 철학, 윤리 아니면 정치적 입장 등이 담길 수 있습니다.

“변경 불가 부분”(Invariant Section)은 2차 구성부의 한 형태로 이 허가서 아래에 베포되는 문서에서 저작권 고지에 서술된 변경 불가 부분이라 표기된 제목이 붙은 부분들을 의미합니다. 만약, 이 부분이 2차 구성부의 정의에 맞지 않는다면 변경 불가 부분으로 취급되지 않습니다. 문서는 변경 불가 부분이 없을 수도 있습니다. 문서에서 변경 불가 부분이 고지 되지 않았다면 해당 문서에 변경 불가 부분은 없습니다.

“표지 구절”(Cover Texts)은 이 허가서 아래에 베포되는 문서에서 저작권 고지에 서술된 앞 표지나 뒷 표지에 들어가야 할 짧은 문장을 말합니다. 앞 표지 구절은 적어도 최대 5개의 단어들로 이루어져야 하고, 뒷 표지 구절은 최대 25개의 단어들로 이루어질 수 있습니다.

문서의 “공개”(Transparent) 복제물이란 기기에서 읽을 수 있는 복제본으로, 공공에 열려있는 파일 형식으로 되어 있고, 일반적인 문서 편집기(그림의 경우 픽셀로 구성된 그림에서, 그리기 프로그램(일반적인 그림), 널리 사용되는 그림 편집기 등에서 출력, 수정이 용이하고 다양한 형태의 파일 형식으로 자동 변환하기 용이한 형태로 만들어진 복제물을 말합니다. 반면, 마크업 언어를 추가하거나 존재하는 마크업 언어를 생략하는 식의 변경을 가해 피양도자의 문서 수정을 방해 혹은 금지시키는 형식은 공개되지 않은 복제물로 간주합니다. 방대한 양의 문서를 그림 파일 형식으로 베포하는 경우 해당 문서는 공개되지 않았다 간주합니다. 그러한 복제물은 “공개”되지 않은 “비공개”(Opaque) 복제물입니다.

이러한 공개된 복제물의 적절한 형식의 예로, 마크업 언어가 포함 되지 않은 평범한 ASCII 형식 텍스트 파일, Texinfo 입력 파일, Latex 입력 파일, 공개적으로 사용 가능한 DTD에서 사용하는 SGML이나 XML 형식 파일, 표준 규약을 준수하는 HMTL 형식, 편집 가능한 PostScript 나 PDF 형식이 있습니다. 공개 그림 형식의 예로는 PNG, XCF, 그리고 JPG 등이 있습니다.

비공개 형식의 예로는 대응 되는 워드 프로세서에서만 읽고 쓸 수 있는 파일 형식이나 일반적으로 사용할 수 없는 DTD에서 쓰는 SGML과 XML 형식, 편집을 고려하지 않고 특정 워드 프로세서에서 출력 목적만을 위해 기기에서 만들어진 HTML, PostScript나 PDF 등이 있습니다.

"제목 페이지"(Title Page)는 발행된 서적에 대해, 서적의 제목 페이지 자체와 법적으로 허가서가 필요에 의해 제목 페이지에 귀속 시킨 모든 추가 페이지들을 의미합니다. 제목 페이지 형식의 페이지가 없는 문서의 경우 "제목 페이지"는 해당 문서에서 본문 앞에 나오는 문서의 제목에 가장 비슷한 페이지를 의미합니다.

"발행자"(Publisher)는 문서를 공공에 베포하는 개인이나 기관을 의미합니다.

"XYZ라 이름 붙은" 단원의 의미는 문서 안에서 이름이 붙은 부분으로 제목이 정확히 XYZ이거나 XYZ를 다른 언어로 번역한 문서 안에 XYZ가 괄호 안에 포함되어 있는 제목이 붙은 부분을 말합니다. (XYZ는 "감사의 글", "헌사", "추천사"나 "개정 이력" 등의 특정 제목이 붙은 부분을 의미합니다.) 문서를 수정할 때, 이러한 단원들의 "제목을 유지"함은 "XYZ라 이름 붙은" 단원이 정의에 따라 남아있어야 함을 의미합니다.

경우에 따라 보증 면책 선언이 이 허가서의 적용 고지 다음에 포함될 수도 있습니다. 이 보증 면책 선언 부분은 허가서에 포함되었다고 간주됩니다. 하지만, 이는 오직 보증에 관한 면책입니다. 보증 면책이 가질 수 있는 다른 암묵적 사항들은 모두 무효화되며 이 허가서에 어떠한 영향도 끼치지 않습니다.

마.3 2. 동일 복제

이 허가서에 따라 제공되는 문서는 저작권의 통지와 본 허가서에 어떠한 추가 사항이 가해지지 않은 상황에서 모든 재생산된 복사본에 이 허가서가 적용된다는 사실과 본 허가서를 함께 제공한다면, 피양도자는 모든 매체를 통해 문서를 상업/비상업적으로 베포할 수 있습니다. 이때, 피양도자는 피양도자가 만들거나 베포하는 작업물들의 복사나 열람을 제한하는 어느 기술적 제약도 적용해서는 안됩니다. 하지만, 복사본의 제공에 대가를 받을 수는 있습니다. 충분히 큰 양의 복사본을 베포하고자 한다면, 제 3조의 조건들을 따라야합니다.

피양도자는 위에 언급한 내용과 같은 조건 아래에서 복사본을 대여하거나 공개적으로 전시할 수도 있습니다.

마.4 3. 대량 복제

인쇄물이나 인쇄된 표지를 가진 매체의 형태로 복사본을 100부 이상 발행하고 해당 문서가 표지 구절을 명시하고 있다면, 반드시 인쇄물의 앞, 뒷 표지에 모든 표지 구절을 명확하고 주시하기 쉬운 형태로, 앞 표지 구절은 앞 표지에, 뒤 표지 구절은 뒤 표지에 기재해야 합니다. 두 표지 모두 반드시 이 복사본의 발행자를 명확하게 판별할 수 있도록 구성되어야 합니다. 앞 표지는 전체 제목을 모든 단어를 빠뜨리지 않고 동일한 주목성과 가시성을 유지해야 합니다. 추가로 표지에 다른 내용들을 추가할 수도 있습니다. 표지에 한정된 변경을 포함하는 복사는 원문을 그대로 복사한다고 간주 될 수 있습니다. 이때, 문서의 제목을 유지하고 이러한 조건들을 만족시켜야 합니다.

표지에 들어가야할 내용이 너무 많아서 가독성을 해치는 경우 적당히 가능한 내용으로 표지에 작성하고 인접한 페이지에 나머지 내용을 기재할 수도 있습니다.

비공개 복제물을 100부 이상 발행, 베포하는 경우 기계에서 읽을 수 있는 공개 복제물을 각각의 비공개 복제물에 포함시켜야 합니다. 아니면, 일반 대중이 표준 네트워크 프로토콜을 사용해 공개 복제물을 다운로드 할 수 있는 네트워크 상의 공개 복제물의 위치를 비공개 저작물 안에 명시하거나 포함시켜야 합니다. 이 공개 복제물은 비공개 복제물에서 추가한 내용들을 제거한 상태여야 합니다. 후자를 선택할 경우 주의해야 할 점이 있습니다. 비공개 복제물에 기재된 위치에서 공개 복제물의 접근이 마지막 비공개 복제물의 베포 후 1년까지는 직접적으로나 별도의 대리인, 소매업자를 통해서든 가능해야 합니다.

필수는 아니지만 문서를 대량으로 재 베포하기 전, 문서의 원본 저자와 연락해 저자가 최신 판본을 제공할 기회를 주는 행위가 적절합니다.

마.5 4. 수정

제 2와 3조의 내용에 따라 저작물을 수정하고 베포할 수 있습니다. 이는 정확히 이 허가서를 원 저작물의 역할을 하는 수정본에 적용해서 베포함을 의미합니다. 따라서, 베포와 수정 권한이 수정본의 복사본을 가지는 대상에게 그대로 부여됩니다. 이에 더해 수정본에 관해 다음의 내용을 준수해야 합니다.

가. 제목 페이지와 커버에 원본이나 이전 버전과 다른 제목을 사용해야 합니다. 이전 버전이 있을 경우 이력 부분에 나열되어야 합니다. 원본이나 이전 버전의 저작권자의 허락이 있다면 동일한 제목을 사용할 수 있습니다.

나. 제목 페이지에 저자로써 하나 혹은 여러 사람이나 기관의 목록이 올라가야 합니다. 이들은 수정본의 수정에 관한 저작권을 소유한 대상으로 최소 원문서의 주요 저자가 5명 이상은 올라가 있어야 합니다. (5명 보다 적다면 모든 주요 저자를 적어야합니다) 이는 저자들이 이 조항의 요구사항을 해제하지 않는 한 지켜저야 합니다.

다. 제목 페이지에 수정본의 발행자를 기재해야합니다.

라. 원문서의 모든 저작권 고지 부분을 유지해야 합니다.

마. 다른 저작권 고지에 인접부에 수정본의 적절한 저작권 고지를 추가해야 합니다.

바. 저작권 고지 바로 다음에, 이 수정본이 본 허가서 아래에서 사용될 수 있다는 허가 고지를 포함해야 합니다. 이 고지는 아래의 부록 부분의 형태로 사용해야합니다.

사. 원 문서의 사용 허가 고지에 포함된 모든 변경 불가 부분의 목록과 표지 구절을 수정판에 모두 유지시킵니다.

아. 이 사용 허가서를 변경 없이 그대로 포함시킵니다.

자. "개정 이력"으로 이름 붙은 단원을 그대로 유지시킵니다. 제목을 그대로 유지하고, 이 부분에 최소한 제목, 발행 년도, 수정본의 새로운 저자, 그리고 수정본의 발행자를 제목페이지에 포함된 대로 추가합니다. "개정 이력"으로 이름 붙은 부분이 없다면 새로 만들고 제목 페이지에 포함된 대로 문서의 제목, 발행 년도, 발행자를 기재합니다. 그리고, 이전 글에 서술된 대로 수정본의 정보를 추가합니다.

차. 네트워크 상에서의 위치를 유지해야합니다. 이는 문서의 공개 복제물을 대중에게 공개했을 시에 적용되며 문서의 기반이 된 이전 버전의 네트워크 위치도 마찮가지 로 취급됩니다. 이들은 "개정 이력" 부분에 포함되어 있을 수 있습니다. 문서가 발행된 시점에서 4년전 이상 전의 네트워크 위치는 포함하지 않을 수 있습니다. 아니면 참조하는 버전의 원 저작자가 허가할 경우 포함하지 않을 수 있습니다.

카. "감사의 글"이나 "헌사" 부분이 있다면 제목을 유지하고 기여자에 대한 감사와 헌사를 모든 부분, 내용 및 어조를 포함해서 수정판에도 모두 유지합니다.

타. 원본의 모든 변경 불가 부분을 수정본에도 유지. 이는 내용과 그 제목을 모두 포함합니다. 단원 순서를 나타내는 번호나 이와 동일한 사항들은 변경 불가 부분의 제목으로 간주되지 않습니다.

파. 원본의 추천사 부분을 삭제할 것, 해당 부분은 수정본에 관한 내용이 아닙니다.

하. 원본의 어떠한 단원도 "추천사"로 제목을 바꾸지 말고, 바꾸는 제목은 변경 불가 부분과 충돌하지 말아야합니다.

. 보증 면책 고지를 유지할 것

수정본이 문서에 없던 새로운 서두나 부록을 2차 구성부의 형태로 포함하고 있고 본 문서에 있는 어떤 내용도 가져오지 않았다면, 해당 부분을 변경 불가 부분으로 선언할 수 있습니다. 선언을 위해서는 수정본의 허가 고지부분에 해당 부분의 제목들을 불변 부분으로 선언하면 됩니다. 이때, 이 제목들이 다른 단락의 제목들과 구분되어야 합니다.

수정본에 대한 다양한 집단, 예를 들어서 동료 평가나 표준적으로 권위 있는 기관들이 승인한 내용이 추천사로 제공될 경우 "추천사"부분을 수정본에 추가할 수 있습니다.

수정본의 앞, 뒤 표지에 각각 5개, 25개의 단어 이하의 구절을 표지 구절로 추가할 수 있습니다. 한 개인, 단체 그리고 이들에 의해 이루진 협약은 각각 1개의 문장만을 각각 앞, 뒤 표지에 추가할 수 있습니다. 동일 주체에 의한 표지 구절이 이미 문서에 포함된 경우 새로운 구절을 동일 주체가 추가할 수 없습니다. 단, 오래된 표지구절을 문서의 발행인으로 부터 명시적으로 승인 받은 경우 새로운 표지 구절로 교체할 수 있습니다.

이 허가서는 원 문서의 저자(들)과 발행자(들)이 이러한 수정본의 선전이나 혹은 수정본의 추천에 명시적으로나 암묵적으로나 이름이 사용됨에 동의함을 뜻하지 않습니다.

마.6 5. 문서의 결합

이 허가서 아래에 베포되는 다른 문서들을 수정본에 관한 이 허가서의 4조 조항들에 따라 결합시킬 수 있습니다. 이 결합은 각각 문서들의 변경 불가 부분들을 과 그 목록들을 수정 없이 결합본의 허가 고지에 그대로 포함시켜야 합니다. 또, 모든 보증 면제 부분도 그대로 유지합니다.

결합물은 이 허가서의 복사본 1부만 포함기키면 됩니다. 여러 개의 동일한 변경 불가 부분들도 1개로 통합할 수 있습니다. 여러 개의 변경 불가 부분들이 동일한 제목을 가지고 내용이 다른 경우 각각에 대해 구분 가능한 표지를 각 제목 끝에 괄호 안으로 추가합니다. 원 문서의 저자나 해당 단락의 발행사를 알고있을 경우 이들을 표지에 넣고 아닐 경우 숫자를 사용할 수 있습니다. 각 원본 문서의 저작권 고지에 기재된 변경 불가 부분들 또한 이와 같은 방식을 사용해 결합합니다.

결합물에 관해, 각 문서의 “개정 이력”부분은 모두 통합해 새로운 1 개의 “개정 이력”으로 만들고 “감사의 글”과 “헌사” 등도 동일하게 통합해야 합니다. 결합시 “추천사”는 모두 삭제해야합니다.

마.7 6. 문서 규합

이 허가서를 따르는 여러 저작물을 묶고, 각각 저작물의 허가서 고지를 묶음본의 허가서 고지 1개로 대체할 수 있습니다. 단, 각 저작물의 내용 복사에 관한 모든 측면에서 이 허가서의 내용을 준수해야 합니다.

이 묶음본에서 단일 저작물을 추출해서, 이 허가서 아래에 다시 베포할 수도 있습니다. 이 때, 이 허가서의 고지를 추출된 저작물에 포함시켜야 하고, 저작물의 복사에 관련된 모든 측면에서 이 허가서의 내용을 준수해야 합니다.

마.8 7. 독립된 저작물과의 통합

저작물이나 2차 저작물들을 독립적인 문헌과 작업물들과 함께 저장, 베포 매체로 구성된 합본을 만들 경우 이는 “통합본”으로 불립니다. 이러한 통합 구성에 따른 저작권이 통합본의 이용자들의 향후 작업을 법적으로 제한하는 형태로 사용되지 않는한, 이 허가서에서 규정하는 수정본으로 취급하지 않습니다. 이 허가서가 적용된 저작물이 통합본안에 포함될 경우 이 허가서는 통합본 내 다른 문서들이 허가서가 적용된 저작물에서 파생되지 않는 한 적용되지 않습니다.

3조의 표지 구절이 통합본 내의 복제물에 적용되는 경우 해당 복제물이 통합본의 전체 분량의 절반 미만인 경우 복제물의 표지 구절은 해당 복제물이 위치한 곳의 표지에, 전자 문서일 경우 전자 문서에서 이에 해당하는 표지에 위치해도 무방합니다. 다른 경우에는 전체 통합본의 표지 부분에 나타나야 합니다.

마.9 8. 번역

번역은 수정의 일종으로 취급됩니다. 따라서 저작물의 번역본을 베포할 때는 4조의 내용을 따라 베포해야 합니다. 변경 불가 부분을 번역으로 대체할 때는 저작권자의 특별 허락이 필요합니다. 하지만, 변경 불가 부분의 일부나 전체의 번역본을 변경 불가 부분과 포함할 수는 있습니다. 이 허가서의 번역본, 저작물 내의 허가서와 보증 면제의 통지는 영어 원본을 함께 제공하는 조건 아래에서 번역본을 제공할 수 있습니다. 허가서, 통지, 보증 면제 내용에 관해 번역본과 원본이 다른 차이가 있다면 원본이 우선권을 가집니다.

만약, 저작물 내 단락중 1조에 기재된 “감사의 말”, “헌정” 혹은 “이력” 등의 이름이 붙은 단락이 있는 경우, 4조에 따른 제목의 보존 요건은 일반적으로 번역에 따라 실제 제목을 바꿀 수 있습니다.

마.10 9. 권리 소멸

이 허가서에서 아래에 명시된 경우를 제외하고, 문서를 복사, 수정 별도의 부가 허가서 적용, 그리고 베포하는 행위는 모두 금지됩니다. 그리고 이러한 복사, 수정, 별도 허가서 적용과 베포를 하려는 시도들 또한, 금지됩니다. 이 허가서 아래에서 해당 사항들에 관한 위반자의 권리는 정지됩니다.

그러나, 이러한 허가서의 위반을 모두 중단하면 특정 저작권자로 부터 받은 허가는 다시 회복됩니다. (a) 잠정적으로는 저작권 소유자가 명시적으로 이용자의 허가를 종료하지 않을 경우 종료할 때까지, (b) 영구적으로는 저작권 소유자가 중단 후 60일 이내로 적절한 이유로 위반 사실을 통지하지 않은 경우입니다.

또, 특정 저작권자로 부터 받은 허가는 다음의 경우 영구적으로 복원됩니다. 저작권자가 위반 사항에 대해 적절한 이유로 통지한 경우, 이 허가에 의해 해당 저작권자로 부터 위반 통지를 받은 사실이 처음일 때, 저작권자로 부터 위반 통보를 받고 해당 통보 이후 30일 이내로 위반 사항을 해결했을 때. (이는 모든 작업물에 적용됩니다.)

이 허가서 아래에서 위반으로 소멸된 권리는 해당 위반자로 부터 복제물과 권리를 받은 다른 집단의 권리를 소멸시키지 않습니다. 위반자의 권리가 종료 되고 영구적으로 복원되지 않은 경우, 동일한 자료의 일부 또는 전부의 수령은 해당 위반자에 대한 어떤 권리도 부여하지 않습니다.

마.11 10. 허가서의 향후 개정

자유 소프트웨어 재단에서 GNU 자유 문서 사용 허가서의 새로운 개정안을 차후에 발행할 수도 있습니다. 해당 새 개정본은 현재 판본과 동일한 정신을 공유하겠지만 새로운 문제나 고려 사항에 대해 세부적으로는 다를 수 있습니다. http://www.gnu.org/copyleft/ 를 참고하길 바랍니다.

허가서의 개정판은 판본의 숫자를 통해 구분합니다. 저작물이 이 허가서의 특정판본의 숫자와 “차후 갱신본”를 적용했다면, 해당 버전과 차후 발행된 갱신본의 중 하나를 선택해 따를 수 있습니다. 갱신본은 자유 소프트웨어 재단에서 정식 발행된 상황이어야 합니다. 초안은 해당하지 않습니다. 특정 판본이 명시되지 않았다면 자유 소프트웨어 재단에서 발행한 판본을 임의로 선택해 따를 수 있습니다. 만약, 문서에서

대리인이 차후의 허가서 판본을 결정할 수 있다고 명시할경우, 대리인의 차후 공공 고지의 판본 특정이 영구적으로 문서의 허가서 판본 규정으로 인정됩니다.

마.12 11. 재허가

"대규모 다중 저자 협업 사이트"("MMC 사이트")는 저작물을 베포하고 누구나 해당 저작물을 수정할 수 있게 하는 기능들을 제공하는 World Wide Web 서버를 의미합니다. 누구나 수정할 수 있는 공공 위키가 이러한 사이트의 예시입니다. "대규모 다중 저자 협업물"(MMC)은 이러한 MMC 사이트에서 발행되는 모든 종류의 저작물을 의미합니다.

"CC-BY-SA"는 Creative Commnos Attribution-Share Alike 3.0을 의미합니다. 이 허가서는 Creative Commons Corporation에 의해 발행되었습니다. 이 기관은 캘리포니아 주의 샌프란시스코에 위치한 비영리 기관입니다. 허가서의 차후 갱신판 또한 동일한 기관에서 베포됩니다.

MMC는 다음의 조건 아래에서 "재허가 가능 저작물"입니다. MMC의 이용 허가가 이 허가서를 따르고, MMC의 모든 내용이 MMC사이트가 아닌 다른 곳에서 먼저 발행된 문서가 이 허가서를 따르고 나중에 MMC에 통합되었을 때, (1) 표지 구절이이나 변경 불가 부분이 없고, (2) 2008년 11월 1일 전에 통합되었을 경우.

MMC 사이트의 운영자는 사이트 내부의 MMC를 다음의 조건 아래에서 재 발행할 수 있습니다. CC-BY-SA 허가를 따르고, 2009년 8월 1일 이전 시간이고, MMC가 재허가 가능 저작물 일 때.

마.13 부록: 문서에 허가서 적용하기

제목 페이지 바로 다음에 다음의 저작권과 허가서 통지를 명시하면 됩니다.

```
Copyright (c)  저작물의 발행년  저작권자의 이름.
GNU 자유 문서 사용 허가서 1.3판과 자유 소프트웨어 재단에서 발행한 차후의 갱신본의
규정에 따라, 본 저작물의 복제, 베포 및 수정을 허가합니다. 변경 불가 부분, 앞 표지 구절과
뒷 표지 구절은 없습니다. 이 허가서의 복사본은 "GNU 자유 문서 사용 허가서"로 지어진 단락에
포함되어 있습니다.
```

변경 불가 부분이 "LIST THEIR TITILES"로 주어져 있고, 앞/뒤 표지 "FRONT_LIST"와 "BACK_LIST"가 있다면, "변경 … 구절은 없습니다." 부분을 다음과 같이 변경합니다.

```
본 저작물의 변경 불가 부분은 "LIST THEIR TITILES"입니다.
앞 표지 구절에 다음과 같은 내용이 명시되어야 합니다. "FRONT_LIST"
뒷 표지 구절에 다음과 같은 내용이 명시되어야 합니다. "BACK_LIST"
```

만약, 변경 불가 부분만 있고 표지가 없거나, 이 세가지 경우중 여러가지가 같이 있다면, 상황에 맞게 두 대안을 병합해 사용하면 됩니다.

만약, 문서가 프로그램 코드 예시들을 포함하고 있다면, 해당 예제들을 GPL과 같은 자유 소프트웨어 허가서 아래에서 베포해, 다른 자유 소프트웨어에서 사용 가능하도록 함을 권장합니다.

마.14 영어 원문

이 전문은 다음의 GNU 공식 웹 사이트에서도 확인할 수 있습니다.

https://www.gnu.org/licenses/fdl-1.3.html

```
                GNU Free Documentation License
                 Version 1.3, 3 November 2008


 Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
     <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other
functional and useful document "free" in the sense of freedom: to
assure everyone the effective freedom to copy and redistribute it,
with or without modifying it, either commercially or noncommercially.
Secondarily, this License preserves for the author and publisher a way
to get credit for their work, while not being considered responsible
for modifications made by others.

This License is a kind of "copyleft", which means that derivative
works of the document must themselves be free in the same sense.  It
complements the GNU General Public License, which is a copyleft
license designed for free software.

We have designed this License in order to use it for manuals for free
software, because free software needs free documentation: a free
program should come with manuals providing the same freedoms that the
software does.  But this License is not limited to software manuals;
it can be used for any textual work, regardless of subject matter or
whether it is published as a printed book.  We recommend this License
principally for works whose purpose is instruction or reference.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that
contains a notice placed by the copyright holder saying it can be
distributed under the terms of this License.  Such a notice grants a
world-wide, royalty-free license, unlimited in duration, to use that
work under the conditions stated herein.  The "Document", below,
refers to any such manual or work.  Any member of the public is a
licensee, and is addressed as "you".  You accept the license if you
copy, modify or distribute the work in a way requiring permission
under copyright law.

A "Modified Version" of the Document means any work containing the
Document or a portion of it, either copied verbatim, or with
modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of
the Document that deals exclusively with the relationship of the
publishers or authors of the Document to the Document's overall
subject (or to related matters) and contains nothing that could fall
directly within that overall subject.  (Thus, if the Document is in
part a textbook of mathematics, a Secondary Section may not explain
any mathematics.)  The relationship could be a matter of historical
connection with the subject or with related matters, or of legal,
commercial, philosophical, ethical or political position regarding
them.

The "Invariant Sections" are certain Secondary Sections whose titles
are designated, as being those of Invariant Sections, in the notice
that says that the Document is released under this License.  If a
section does not fit the above definition of Secondary then it is not
allowed to be designated as Invariant.  The Document may contain zero
Invariant Sections.  If the Document does not identify any Invariant
Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed,
as Front-Cover Texts or Back-Cover Texts, in the notice that says that
the Document is released under this License.  A Front-Cover Text may
be at most 5 words, and a Back-Cover Text may be at most 25 words.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
A "Transparent" copy of the Document means a machine-readable copy,
represented in a format whose specification is available to the
general public, that is suitable for revising the document
straightforwardly with generic text editors or (for images composed of
pixels) generic paint programs or (for drawings) some widely available
drawing editor, and that is suitable for input to text formatters or
for automatic translation to a variety of formats suitable for input
to text formatters.  A copy made in an otherwise Transparent file
format whose markup, or absence of markup, has been arranged to thwart
or discourage subsequent modification by readers is not Transparent.
An image format is not Transparent if used for any substantial amount
of text.  A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain
ASCII without markup, Texinfo input format, LaTeX input format, SGML
or XML using a publicly available DTD, and standard-conforming simple
HTML, PostScript or PDF designed for human modification.  Examples of
transparent image formats include PNG, XCF and JPG.  Opaque formats
include proprietary formats that can be read and edited only by
proprietary word processors, SGML or XML for which the DTD and/or
processing tools are not generally available, and the
machine-generated HTML, PostScript or PDF produced by some word
processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself,
plus such following pages as are needed to hold, legibly, the material
this License requires to appear in the title page.  For works in
formats which do not have any title page as such, "Title Page" means
the text near the most prominent appearance of the work's title,
preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of
the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose
title either is precisely XYZ or contains XYZ in parentheses following
text that translates XYZ in another language.  (Here XYZ stands for a
specific section name mentioned below, such as "Acknowledgements",
"Dedications", "Endorsements", or "History".)  To "Preserve the Title"
of such a section when you modify the Document means that it remains a
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which
states that this License applies to the Document.  These Warranty
Disclaimers are considered to be included by reference in this
License, but only as regards disclaiming warranties: any other
implication that these Warranty Disclaimers may have is void and has
no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either
commercially or noncommercially, provided that this License, the
copyright notices, and the license notice saying this License applies
to the Document are reproduced in all copies, and that you add no
other conditions whatsoever to those of this License.  You may not use
technical measures to obstruct or control the reading or further
copying of the copies you make or distribute.  However, you may accept
compensation in exchange for copies.  If you distribute a large enough
number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and
you may publicly display copies.



3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have
printed covers) of the Document, numbering more than 100, and the
Document's license notice requires Cover Texts, you must enclose the
copies in covers that carry, clearly and legibly, all these Cover
Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on
the back cover.  Both covers must also clearly and legibly identify
you as the publisher of these copies.  The front cover must present
the full title with all words of the title equally prominent and
visible.  You may add other material on the covers in addition.
Copying with changes limited to the covers, as long as they preserve
the title of the Document and satisfy these conditions, can be treated
as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
legibly, you should put the first ones listed (as many as fit
reasonably) on the actual cover, and continue the rest onto adjacent
pages.

If you publish or distribute Opaque copies of the Document numbering
more than 100, you must either include a machine-readable Transparent
copy along with each Opaque copy, or state in or with each Opaque copy
a computer-network location from which the general network-using
public has access to download using public-standard network protocols
a complete Transparent copy of the Document, free of added material.
If you use the latter option, you must take reasonably prudent steps,
when you begin distribution of Opaque copies in quantity, to ensure
that this Transparent copy will remain thus accessible at the stated
location until at least one year after the last time you distribute an
Opaque copy (directly or through your agents or retailers) of that
edition to the public.

It is requested, but not required, that you contact the authors of the
Document well before redistributing any large number of copies, to
give them a chance to provide you with an updated version of the
Document.


4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under
the conditions of sections 2 and 3 above, provided that you release
the Modified Version under precisely this License, with the Modified
Version filling the role of the Document, thus licensing distribution
and modification of the Modified Version to whoever possesses a copy
of it.  In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct
   from that of the Document, and from those of previous versions
   (which should, if there were any, be listed in the History section
   of the Document).  You may use the same title as a previous version
   if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities
   responsible for authorship of the modifications in the Modified
   Version, together with at least five of the principal authors of the
   Document (all of its principal authors, if it has fewer than five),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   unless they release you from this requirement.
C. State on the Title page the name of the publisher of the
   Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications
   adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice
   giving the public permission to use the Modified Version under the
   terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections
   and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add
   to it an item stating at least the title, year, new authors, and
   publisher of the Modified Version as given on the Title Page.  If
   there is no section Entitled "History" in the Document, create one
   stating the title, year, authors, and publisher of the Document as
   given on its Title Page, then add an item describing the Modified
   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on.  These may be placed in the "History" section.
   You may omit a network location for a work that was published at
   least four years before the Document itself, or if the original
   publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications",
   Preserve the Title of the section, and preserve in the section all
   the substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles.  Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements".  Such a section
   may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements"
   or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
copied from the Document, you may at your option designate some or all
of these sections as invariant.  To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains
nothing but endorsements of your Modified Version by various
parties--for example, statements of peer review or that the text has
been approved by an organization as the authoritative definition of a
standard.

You may add a passage of up to five words as a Front-Cover Text, and a
passage of up to 25 words as a Back-Cover Text, to the end of the list
of Cover Texts in the Modified Version.  Only one passage of
Front-Cover Text and one of Back-Cover Text may be added by (or
through arrangements made by) any one entity.  If the Document already
includes a cover text for the same cover, previously added by you or
by arrangement made by the same entity you are acting on behalf of,
you may not add another; but you may replace the old one, on explicit
permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License
give permission to use their names for publicity for or to assert or
imply endorsement of any Modified Version.


5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this
License, under the terms defined in section 4 above for modified
versions, provided that you include in the combination all of the
Invariant Sections of all of the original documents, unmodified, and
list them all as Invariant Sections of your combined work in its
license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and
multiple identical Invariant Sections may be replaced with a single
copy.  If there are multiple Invariant Sections with the same name but
different contents, make the title of each such section unique by
adding at the end of it, in parentheses, the name of the original
author or publisher of that section if known, or else a unique number.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Make the same adjustment to the section titles in the list of
Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History"
in the various original documents, forming one section Entitled
"History"; likewise combine any sections Entitled "Acknowledgements",
and any sections Entitled "Dedications".  You must delete all sections
Entitled "Endorsements".


6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other
documents released under this License, and replace the individual
copies of this License in the various documents with a single copy
that is included in the collection, provided that you follow the rules
of this License for verbatim copying of each of the documents in all
other respects.

You may extract a single document from such a collection, and
distribute it individually under this License, provided you insert a
copy of this License into the extracted document, and follow this
License in all other respects regarding verbatim copying of that
document.


7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate
and independent documents or works, in or on a volume of a storage or
distribution medium, is called an "aggregate" if the copyright
resulting from the compilation is not used to limit the legal rights
of the compilation's users beyond what the individual works permit.
When the Document is included in an aggregate, this License does not
apply to the other works in the aggregate which are not themselves
derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these
copies of the Document, then if the Document is less than one half of
the entire aggregate, the Document's Cover Texts may be placed on
covers that bracket the Document within the aggregate, or the
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
electronic equivalent of covers if the Document is in electronic form.
Otherwise they must appear on printed covers that bracket the whole
aggregate.


8. TRANSLATION

Translation is considered a kind of modification, so you may
distribute translations of the Document under the terms of section 4.
Replacing Invariant Sections with translations requires special
permission from their copyright holders, but you may include
translations of some or all Invariant Sections in addition to the
original versions of these Invariant Sections.  You may include a
translation of this License, and all the license notices in the
Document, and any Warranty Disclaimers, provided that you also include
the original English version of this License and the original versions
of those notices and disclaimers.  In case of a disagreement between
the translation and the original version of this License or a notice
or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements",
"Dedications", or "History", the requirement (section 4) to Preserve
its Title (section 1) will typically require changing the actual
title.


9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense, or distribute it is void, and
will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license
from a particular copyright holder is reinstated (a) provisionally,
unless and until the copyright holder explicitly and finally
terminates your license, and (b) permanently, if the copyright holder
fails to notify you of the violation by some reasonable means prior to
60 days after the cessation.

Moreover, your license from a particular copyright holder is
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, receipt of a copy of some or all of the same material does
not give you any rights to use it.


10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the
GNU Free Documentation License from time to time.  Such new versions
will be similar in spirit to the present version, but may differ in
detail to address new problems or concerns.  See
http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number.
If the Document specifies that a particular numbered version of this
License "or any later version" applies to it, you have the option of
following the terms and conditions either of that specified version or
of any later version that has been published (not as a draft) by the
Free Software Foundation.  If the Document does not specify a version
number of this License, you may choose any version ever published (not
as a draft) by the Free Software Foundation.  If the Document
specifies that a proxy can decide which future versions of this
License can be used, that proxy's public statement of acceptance of a
version permanently authorizes you to choose that version for the
Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any
World Wide Web server that publishes copyrightable works and also
provides prominent facilities for anybody to edit those works.  A
public wiki that anybody can edit is an example of such a server.  A
"Massive Multiauthor Collaboration" (or "MMC") contained in the site
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0
license published by Creative Commons Corporation, a not-for-profit
corporation with a principal place of business in San Francisco,
California, as well as future copyleft versions of that license
published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in
part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this
License, and if all works that were first published under this License
somewhere other than this MMC, and subsequently incorporated in whole or
in part into the MMC, (1) had no cover texts or invariant sections, and
(2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site
under CC-BY-SA on the same site at any time before August 1, 2009,
provided the MMC is eligible for relicensing.


ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of
the License in the document and put the following copyright and
license notices just after the title page:

    Copyright (c)  YEAR  YOUR NAME.
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.3
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```

제 바 부록

GSL 디자인 문서(*)

참고: 본 문서는 Mark Galassi, James Theiler 와 Brian Gough 가 저술한 **GNU Scientific Library – Design document** 의 번역본 입니다.

원문은 https://www.gnu.org/software/gsl/design/gsl-design_toc.html 를 참고할 수 있습니다.

이 문서는 프로젝트의 초창기에 작성되었기 때문에 몇몇 내용들은 현재 라이브러리의 제공 기능들과 맞지 않을 수도 있습니다. 차후 제공 예정으로 서술된 기능이 현재 라이브러리에서 이미 구현되어 있기도 합니다. 일부 내용의 갱신이 이루어지지 않았기 때문에 본 문서는 해당 내용을 일부 수정했습니다(*).

바.1 프로젝트의 시작

과학자와 공학자들이 수치해석 소프트웨어나 모듈을 개발할 때, 다음과 같은 라이브러리의 필요성이 대두되어왔습니다.

- 자유롭게 사용이 가능해 사용자들이 사용, 재배포와 수정이 가능한 라이브러리(자유는 공짜의 의미가 아닙니다).
- 현대적인 코딩, 호출, 스코프 규약을 따라 작성된 C 라이브러리.
- 정갈하고 교육적인 문서를 제공하는 라이브러리; 가급적 TeXinfo로 작성된 문서를 의미합니다. 이는 온라인 베포 및 TeX 변환에 매우 용이합니다.
- 고품질의 최신 알고리즘들을 포함하는 라이브러리.
- `autoconf` 와 `automake` 를 사용해, 손쉬운 이식성과 구성이 가능한 라이브러리.
- GNUlitically correct[1] 한 라이브러리

[1] GNUlitically correct 은 GNU 코딩 규약을 따르고 autoconf 를 사용하는 프로그램을 뜻합니다(*).

기존의 수치 해석 라이브러리들을 사용할 수도 있지만 해당 라이브러리들은 다음과 같은 장단점이 있습니다.

표 6.1: 여러 수치해석 라이브러리

라이브러리	설명
Netlib	Netlib는 AT&T에서 관리하는 라이브러리로, 인터넷 상에서 가장 발전된 수치 해석 알고리즘들을 제공합니다. 불행히도, 대부분의 소프트웨어가 포트란으로 작성되어 있어 대다수의 상황에서 낮선 호출 규약을 사용해야하고, 매우 파편화되어 있어 Netlib의 사용에 큰 노력이 필요합니다.
GAMS	GAMS는 과학 계산 소프트웨어들의 집합체입니다. 하지만, Netlib와 같이, 각각의 기능들마다 구현체의 질과 문서화 수준이 매우 천차만별입니다.
Numerical Recipes	Numerical Recipes는 명확한 방법으로 알고리즘들을 잘 설명하는 훌륭한 책입니다. 하지만, 이 책의 저자는 해당 서적에 있는 코드의 사용은 허용하고 있으나, 재배포에 제약이 있습니다. 따라서 Numerical Recipes는 자유롭지 않습니다. 무엇보다도, 해당 책의 코드 구현체는 포트란스럽다(Fortrannitis)는 평과 다른 한계들이 있습니다. Reviews of Numerical Recipes
SLATEC	SLATEC는 1970년도 Department of Energy program에서 작성된 수치 해석 소프트웨어들의 대규모 집합체입니다. 해당 소프트웨어들은 자유 이용 저작물으로 배포되고 있습니다. 각각의 기능들은 잘 검증되어 있고, 그 시기 한계가 있기는 하지만 정갈하게 잘 조직된 구조를 가지고 있습니다. GSL은 현대적인 SLATEC를 목적으로 하고 있습니다.
NSWC	NSWC는 Naval Surface Warfare Center numerical library의 약자입니다. 자유 이용 저작물으로 배포되는 고수준의 대규모 포트란 라이브러리입니다. 이 라이브러리는 문서를 찾기가 매우 힘듭니다. 출판본의 일부 복사본이 확인되었을 뿐입니다.
NAG와 IMSL	NAG와 IMSL는 모두 상업적으로 판매되는 고수준의 수치 해석 라이브러리입니다. NAG 라이브러리는 IMSL보다 더 많은 기능과 발전된 형태를 가지고 있습니다. IMSL 라이브러리는 편의성에 더 치우쳐져있고, 기본 인자들을 광범위한 가변 인자 배열을 사용해 에뮬레이트합니다.
ESSL와 SCSL	ESSL와 SCSL는 각각 IBM과 SGI에서 상업적으로 판매하는 라이브러리입니다.
Forth Scientific Library	Forth Scientific Library는 Forh 사용자들만을 대상으로 합니다.
Numerical Algorithms with C	Numerical Algorithms with C, G. Engeln-Mullges, F. Uhlig는 서적과 함께 제공되는 ANSI C로 작성된 훌륭한 수치 해석 라이브러리입니다. 코드 사용이 가능하지만, 자유 소프트웨어가 아닙니다.

다음 페이지에 계속

표 6.1 – 이전 페이지에서 계속

라이브러리	설명
NUMAL	NUMAL 라이브러리의 C 버전은 H.T. Lau에 의해 작성되었으며, “A Numerical Library in C for Scientists and Engineers” 제목의 책과 디스크로 출판되었습니다. 코드 사용이 가능하지만, 자유 소프트웨어가 아닙니다.
C Mathematical Function Handbook	C Mathematical Function Handbook by Louis Baker는 “Handbook of Mathematical Functions” by Abramowitz and Stegun의 수학 함수들에 대응되는 근사와 C 구현체 라이브러리입니다. 코드 사용이 가능하지만, 자유 소프트웨어가 아닙니다.
CCMATH	CCMATH by Daniel A. Atkinson 는 GSL과 비슷한 범주를 다루는 C로 작성된 수치해석 라이브러리입니다. 코드가 간결한 장점이 있습니다. 초기 버전은 GPL 라이센스 하에서 배포되었지만, 불행히도 최근 버전은 LGPL로 바뀌었습니다.
CEPHES	CEPHES는 C로 작성된 고품질의 특수 함수 구현체 모음입니다. GPL 라이센스가 아닙니다.
WNLIB	WNLIB는 소규모의 수치 해석 C 구현체의 집합입니다. Will Naylor가 작성했으며, 자유 이용 저작물입니다.
MESHACH	MESHACH는 C로 작성된 포괄적인 행렬-벡터 선행 대수 라이브러리입니다. 자유롭게 사용가능하나 GPL은 아닙니다.
CERNLIB	CERNLIB는 대규모의 고품질 포트란 라이브러리로 CERN에서 개발되어 많은 세월동안 사용되었습니다. 본래 비자유 소프트웨어였으나 최근 GPL 라이센스로 베포되고 있습니다.
COLT	COLT는 자바로 작성된 자유로은 수치 해석 라이브러리로 CERN에서 Wolfgang Hoschek가 작성했습니다. 이 라이브러리는 BSD 라이센스 아래에서 베포됩니다.

GSL은 실제 수치 해석 전문가나 그들의 대학원생이 기여할 환경을 제공하는 것을 장기적인 목표로 삼고 있습니다.

바.2 기여

이 디자인 문서는 1996년도에 작성되었습니다. 2004년에 GSL의 기본 기능들이 완성되었고, 개발자들은 새로운 기능 추가를 주 목표로 하지 않습니다.

프로젝트에서 일반적으로 우선시 하는 작업은 라이브러리 내 함수들의 안전성, 라이브러리의 일관성과 보고된 버그 수정들입니다.

잠재적 기여자들은 GSL CVS 저장소의 `BUGS` 파일에 나열된 버그들을 조사하고 수정해 라이브러리에 먼저 익숙해지는 것을 권장합니다.

개발한 패키지를 한번에 라이브러리에 추가하는 행위는 일반적으로 권장되지 않습니다. 많은 양의 새 코드들은 다른 구현체들과 완성도 면에서 큰 차이를 야기할 수 있기 때문입니다.

라이브러리의 안전성 유지를 위해 이러한 새 기능들은 GSL 프로젝트 최상단에 패키지로 만들어 개발자가 각각 독립적으로 유지보수하는 것을 권장합니다. 이는 Perl CPAN 아카이브나 TEX CTAN 아카이브등과 같은 자유 소프트웨어 프로젝트에서도 사용하는 방법입니다.

바.2.1 패키지

GSL의 설계는 라이브러리 안에 존재하는 기능들을 추가적인 확장 기능들과 간단하게 연결해 사용할 수 있게 합니다. 예로, 별도의 라이브러리로 제공되는 추가적인 난수 생성기 rngextra 와 함께 다음과 같이 사용할 수 있습니다.

```
$tar -xvfz rngextra-0.1.tar.gz
$cd rngextra-0.1
$./configure; make; make check; make install
$...
$gcc -Wall main.c -lrngextra -lgsl -lgslcblas -lm
```

아래 내용은 패키지 디자인 방법에 관한 것입니다. 이 방법은 GSL 스스로 패키지들의 일관성을 보장해 실 사용자들이 사용하기 쉽고, 향후 GSL에 포함될 잘 검증되고 인기 있는 패키지를 해당 패키지만으로 베포할 수 있게하기 위함입니다.

- 이 문서에서 제공하는 GSL과 GNU 코딩 표준 규약을 준수해야합니다.

  이는 표준 GNU 패키징 도구들을 이용하는 것으로, Automake 의 사용, Texinfo 를 사용한 문서화와 test suite 를 제공함을 의미합니다. test suite 는 make check 를 실행해 검증해야하고, GSL에서 제공하는 검증 함수들을 사용해 결과가 PASS:/FAIL: 로 출력되도록 해야합니다. libtool 사용은 필수가 아닙니다. 패키지는 충분히 작게 만들 수 있고, 정적 라이브러리는 손쉽게 만들수 있기 때문입니다.

- 패키지를 위한 새로운 접두사를 만들어야 합니다. gsl_ 은 라이브러리 내부에서 이미 사용되는 접두사이니 사용하면 안됩니다.

  예로, 별도의 난수 생성자는 rngextra 라는 접두사를 사용할 수 있습니다.

  ```
  #include<rngextra.h>
  gsl_rng * r = gsl_rng_alloc (rngextra_lsfr32);
  ```

- 개발단계를 잘 반영하는 버전 숫자를 사용해야 합니다.

  일반적으로 0.x 는 알파 버전으로 기능의 보증성이 없는 버전을 의미합니다. 0.9.x 는 베타 버전으로 필수적인 기능이 완성되었고, 소소한 변화와 버그 수정만 남은 경우를 의미합니다. 첫번째 정식 베포는 1.0 입니다. 1.0 이나 어느 차후 버전이든 간에, 해당 베포판은 잘 정의된 API를 제공해야합니다. API는 갱신과정에서 변경되어서는 안되며, 기존 코드의 수정이 필요 없도록, 작동에 있어 호환성을 가져야합니다(버그 수정은 제외합니다). 패키지에서 API의 변경이 필요한 경우 새로운 정식 베포가 필요합니다(예, 2.0 과 같은).

- GNU 일반 공중 사용 허가서 (GPL)을 사용해야 합니다.

  패키지가 향후 GSL에 포함되기를 원한다면, 저작권 고지를 얻는 일반적인 절차를 따를 수 있습니다.

GSL에 추가할 새로운 기능은 별도의 패키지로 만들어 따로 개발되고 해당 패키지의 개발 과정이 일정 수준 이상이 되면 GSL의 정식 패키지로 포함됩니다. 만드는 패키지의 갱신 소식을 다음에 이메일

```
gsl-discuss@sourceware.org
```

에 올리면 GSL 공식 사이트 <https://www.gnu.org/software/gsl/#extensions> 에 추가 할 수 있습니다(*)[2] .

바.3 GSL의 디자인

바.3.1 언어

GSL 라이브러리는 **C 언어 하나** 만을 사용합니다.

이미 존재하는 컴파일러의 사용이 가능하며, 구조가 간단하고, 손쉽게 범용성을 얻을 수 있는 장점이 있습니다.

바.3.2 다른 언어를 위한 인터페이스

다른 언어를 위한 랩핑은 “별도의” 패키지로 제공됩니다. “핵심” 라이브러리 패키지로는 제공되지 않습니다. 해당 랩핑은 각각의 기여자들이 별도 관리합니다.

랩핑을 위한 표준 도구들을 사용하는 것을 권장합니다. swing이나 g-warp가 있습니다.

[2] 원본 내용은 “만드는 패키지의 갱신 소식 정보를 `sources.redhat.com` 의 `gsl-discuss` 에 올리면, GSL 웹사이트에 추가시킬 수 있습니다. 예시 패키지 `rngextra` 는 두 개의 난수 발생기를 가지고 있습니다. 이들은 http://www.network-theory.co.uk/download/rngextra/ 에서 찾을 수 있습니다.” 였습니다. rngextra 는 BJG가 만든 예시 패키지입니다. 공식 홈페이지에도 확장 기능으로 소개되어 있지만 해당 링크는 출판사 서버에 대한 아카이브 링크로 Network Theory가 폐업됨에 따라 해당 파일은 찾을 수 없습니다(*). Monte Carlo and Quasi-Monte Carlo Wiki(roth.cs.kuleuven.be/wiki/Rngextra)에 따르면, Tiny Encryption Algorithm 을 사용한 별도의 예시 난수 발생자였다고 합니다(*).

바.3.3 구현하는 기능

존재하는 라이브러리들에 있는 모든 기능을 대상으로합니다.

바.3.4 구현하지 않는 기능

- GPL 라이센스 하에 베포되는 고품질의 패키지에 있는 기능
- 너무 광대한 기능 - 하위 기능이 아닌 응용 프로그램 수준을 만드는 것을 의미합니다.

  예를 들어, 편미분 방정식(PDE)의 풀이를 위한 기능은 매우 크고 전문화된 응용 프로그램으로 제공되는 경우가 빈번합니다. 이는 매우 다양한 편미분 방정식과 해, 방법들이 존재하기 때문입니다. 이러한 종류의 기능들은 각각 작은 기능들로 분할해서 남겨야합니다. 이런 경우는 사용자들에게 해당하는 좋은 응용 프로그램들을 추천하는 것이 좋습니다.

- 독립적으로 별도 제공되었을 때 유용한 것들

  날짜와 시간등을 조작하는 기능이나, 재정 관련 함수들은 "과학 계산" 라이브러리에 포함될 수 있습니다. 이는 의심할 여지가 없지만, 이러한 모듈은 다른 프로그램들에서도 독립적으로 사용할 수 있어, 별도의 라이브러리 사용이 더 유용합니다.

바.3.5 수치해석 라이브러리의 디자인

수치해석 라이브러리의 작성을 할 때, 필연적으로 라이브러리의 **완전성** 과 **간결성** 사이에서 갈등하게 됩니다. 완전성은 라이브러리 내부의 객체와 기능들이 서로 서로에게 적용될 때 이러한 연산의 결과들이 라이브러리 내부의 객체들로 표현될 수 있음을 의미합니다. 이러한 성질을 **닫혀** 있다라 표현합니다[4] .

수학 객체들은 무한히 많은 방법으로 결합하거나 표현할 수 있습니다. 예를 들어서, 스칼라 장을 미분해 벡터 장을 표현할 수도 있고, 벡터 장을 이용해 스칼라 장을 얻을 수도 있습니다.

수학 라이브러리를 작성할 때, 무의식적으로 이러한 모든 가능한 객체를 라이브러리로 구현하려는 경향이 있습니다. 이는 기능을 하나씩 추가하는 과정에서 점점 뚜렷하게 나타납니다. 단지 기능 하나만 더 구현하면 되는 일이기에 하지 않을 이유가 없기 때문입니다.

하지만, 큰 그림을 봅시다. 그 누구도 "모든 가능한 수학 구조와 대상을 C언어로 구조를 이용해 나타내고 싶다."라고 말하지 않습니다. 이러한 전략은 종국엔 반드시 실패하게 됩니다. C와 같은 프로그래밍 언어로 나타낼 수 있는 복잡도는 한계가 있습니다. 이러한 언어에서 수학의 복잡한 구조와 객체들을 재현하려는 시도는 결국 유지 보수가 불가능한 코드를 만들어냅니다. 그러나 이러한 경향을 미리 제거하면, 손쉽게 라이브러리의 구현에 도달할 수 있습니다.

[4] 이러한 표현은 수학에서 대수 구조를 정의할 때, 연산에 대해 닫혀 있다라는 정의에서 왔습니다. 수학적으로는 집합 위에 정의된 연산의 모든 결과가 정의된 집합에 있을 때 이를 닫혀 있다라 합니다. 여기서 닫혀 있다는 뜻은 라이브러리에서 제공하는 객체와 기능들이 충분이 방대해 어떠한 연산을 수행하든지 해당 연산의 결과가 표현하는 수학적 구조가 라이브러리 내부의 기능과 객체들에 이미 구현되어 있음을 의미합니다(*).

완전성보다는 간결성이 더 좋은 선택입니다. 라이브러리 내의 새로운 기능을 디자인할 때, 가능한 한 모듈들이 독립적으로 작동할 수 있도록 작성해야 합니다. 만약, 모듈 간의 상호 의존성이 시도된다면, 어디까지 독립성을 위반할지 확실히 정해야 합니다.

바.3.6 코드 재사용

라이브러리 전체를 사용할 필요 없이, 각각의 코드 파일을 사용자가 만드는 프로그램에 포함할 수 있으면 매우 유용합니다. 이와 같은 독립 실행형 파일이 되도록 함이 권장됩니다. 컴파일 과정에서 당연히, 사용자가 `GSL_ERROR` 와 같은 몇몇 매크로들을 정의해야 할 수도 있습니다. 이런 행위까지는 괜찮습니다. 이러한 예시로 라이브러리 내의 단일 난수 생성기(single random number generator)를 볼 수 있습니다.

바.3.7 표준과 규약

이 프로젝트에 참여하는 사람들은 코딩 표준과 규약을 준수해야합니다. 해당 프로젝트에서는 다음의 표준과 규약들을 따릅니다.

- GNU 코딩 표준
- ANSI 표준 C 라이브러리 규약
- GNU C 라이브러리 규약
- glib: GTK 지원 라이브러리 규약

이러한 표준을 위한 참고 문헌들으로 다음을 참고할 수 있습니다.

- GNU Coding Standards
- Harbison and Steels “C, a Reference Manual”, 5th, 2002, Prentice-Hall, ISBN:9780130895929
- GNU, C Library Manual
- Glib Online Documentation

수학 수식은 Abramowitz & Stegun의 Handbook of Mathematical Functions 를 따릅니다. 이 책은 수학계에서 자명한 참고 문헌이며, 자유 이용 저작물으로 사용할 수 있습니다.

본 프로젝트에서 공유하는 정신은 ” **C로 생각하라** ” 입니다. 프로젝트가 C로 이루어지기 때문에, 다른 언어의 특징을 흉내 내기 보다는 C에 집중해, 어떤 점이 C에서 자연스러운가를 생각해야 합니다. C에서 부자연스러워 다른 언어의 형태로 시뮬레이션해야한다면, 해당 사항들은 본 프로젝트에서 포함하지 않을 것입니다. 해당 기능을 없으면 라이브러리에서 특정 기능의 제공이 어렵거나 제한된 버전만 제공한다 하더라도 해당 기능은 제외되어야 합니다. 라이브러리를 지나치게 복잡하게 만드는 일은 가치가 없습니다. 다른 언어들에도 다양한 수치 해석 라이브러리들이 있으며, 해당 언어에서 사용하는 기능이 필요하다면, C 라이브러리를 강제로 사용하는 대신 해당 언어의 라이브러리를 사용하는 것이 현명합니다.

참고: BJG

C 가 매크로 어셈블러라는 사실을 항상 기억하는 것이 좋습니다. 특정 기능이 너무 복잡하다면 스스로 "이 기능을 매크로-어셈블러로 작성할 수 있는가?"를 생각해보길 바랍니다. "아니다"라면 해당 기능은 GSL에 포함하지 말아야 합니다.

다음의 논문을 참고해 볼 수 있습니다.

- Kiem-Phong Vo, "The Discipline and Method Architecture for Reusable Libraries", Software - Practice & Experience, v.30, pp.107-128, 2000. DOI:10.1002/(SICI)1097-024X(200002)30:2<107::AID-SPE289>3.0.CO;2-D

이 논문은 Wiley Online Library 에서 찾아보거나, 더 이전의 기술 보고서를 IEEE Xplore 에서 다음의 내용으로

- Kiem-Phong Vo, "An architecture for reusable libraries," Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203), 1998, pp. 184-194, DOI: 10.1109/ICSR.1998.685743.

찾아볼 수 있습니다.

이식성 있는 C 라이브러리 디자인에 관련한 다음의 논문들이 있습니다.

- Kiem-Phong Vo, "Vmallo A General and Efficient Memory Allocator". Software Practice & Experience, 26:1–18, 1996, DOI: 10.1002/(SICI)1097-024X(199603)26:3<357::AID-SPE15>3.0.CO;2-%23
- iem-Phong Vo. "Cdt: A Container Data Type Library". Soft. Prac. & Exp., 27:1177–1197, 1997, DOI: 10.1002/(SICI)1097-024X(199710)27:10<1177::AID-SPE125>3.0.CO;2-7
- David G. Korn and Kiem-Phong Vo, "Sfio: Safe/Fast String/File IO", Proceedings of the Summer '91 Usenix Conference, pp. 235-256, 1991, DOI: 10.1.1.51.6574

소스 코드들은 GNU Coding Standards에 맞추어 탭이 아닌 스페이스만 사용해야 합니다. 탭으로 작성했을 시 이를 스페이스로 바꾸어 주어야 하는데 여러방법이 있습니다. 예로 `indent` 프로그램을 사용하면:

```
indent -gnu -nut *.c *.h
```

`-nut` 옵션은 탭을 스페이스들로 바꾸어줍니다.

바.3.8 작업전 확인 사항들

기능을 구현하기 전에 관련 내용들에 관한 철저한 조사가 필요합니다. 이는 장기적으로는 많은 시간을 절약해 줍니다. 가장 중요한 두 가지 단계는 다음과 같습니다.

1. 해당 기능이 이미 자유 라이브러리(GPL이나 GPL-호환)에서 제공하는 기능인지 판별하기. 만약, 이미 존재한다면 재구현할 필요 없습니다. Netlib, GAMs, na-net, sci.math.num-analysis, 그리고 일반적인 인터넷에서 조사를 해보아야 합니다. 이러한 과정은 관련성이 있는 기존의 독점 라이브러리 목록도 조사할 수 있습니다. 다음 단계에서 참고할 수 있도록 해당 목록을 기록하는 것을 권장합니다.

2. 기존의 상업/자유 라이브러리들의 구현체들에 대한 비교 조사를 수행합니다. 일반적인 API, 프로그램과 하위 기능들간의 통신 방법을 검사하고, 해당 구현체들이 가지거나 가지지 않는 기능들을 조사하니다. 그리고 이들의 관련 핵심 개념과 기능들에 익숙해지도록 분류합니다. 이미 존재하는 라이러리들의 문서 리뷰는 좋은 레퍼런스가 되어주는 것을 잊지 말아야 합니다.

3. 해당 주제들을 살펴보고 최신 기술이 무엇인지 파악합니다. 가장 최신의 리뷰 논문들을 찾아보고, 다음의 저널들을 검색해 봅시다.

   - ACM Transactions on Mathematical Software
   - Numerische Mathematik
   - Journal of Computation and Applied Mathematics
   - Computer Physics Communications
   - SIAM Journal of Numerical Analysis
   - SIAM Journal of Scientific Computing

GSL이 연구 프로젝트가 아님을 명심합시다. 좋은 구현체를 만드는 일은, 새로운 알고리즘을 만들지 않더라도 충분히 어려운 작업입니다. 본 프로젝트는 구현 가능하고 존재 가능한 알고리즘의 구현체를 목적으로 합니다. 소소한 개선에 시간을 조금 써도 나쁘지는 않지만, 거기에 몰두하지는 말아야합니다.

바.3.9 알고리즘의 선택

가능하면 잘 확장되는 알고리즘을 선택하고 점근적으로 처리를 해야함을 기억해야합니다. 특히 정수 인자가 있는 함수들에서 주의해야 합니다. Abramowitz & Stegun에서는 재귀적 관계와 같이 함수를 정의하는 데 $O(n)$ 의 시간 복잡도를 가지는 간단한 알고리즘을 많이 사용하고 있습니다. 해당 알고리즘을 그대로 구현하는 데 사용하고 싶을 수 있습니다. 그러나, 이러한 알고리즘은 $n = O(10 - 100)$ 에서는 잘 작동할지 몰라도, $n = 1000000$ 인 경우, 원하는 대로 작동하지 않을 것입니다.

비슷하게, 다변량 자료들이 동일한 크기로 조정된 원소들이나 $O(1)$ 의 복잡도를 가지고 있다고 가정하지 말아야합니다. 알고리즘들은 반드시 내부적으로 필요한 스케일 조정과 균형을 처리해야 하고, 이를 위해 적절한 노름들을 사용해야합니다. (예를 들어, $\|x\|$ 보다는 $\|Dx\|$ 를 사용하는 것이 좋습니다. D 는 스케일 조정을 위한 대각 행렬입니다.)

바.3.10 문서화

문서화: 프로젝트 관리자는 문서화 방법에 대한 예제를 제공해야합니다. 고품질의 문서화는 반드시 필요한 작업입니다. 각 문서는 주제를 소개하고 제공하는 함수들에 대해 세심한 참고를 제공해야합니다. 우선 순위는 함수에 대한 좋은 참고 문헌을 제공하는 것이라 튜토리얼을 반드시 문서에 포함시킬 필요는 없습니다.

사용 설명서에 사용될 그래프를 그릴 때, GNU Plotutils와 같은 자유 소프트웨어를 사용해야 합니다.

어떤 그래프들은 gnuplot과 같이 완전히 자유(아니면 GNU) 소프트웨어가 아닌 프로그램으로 만들어질 수도 있고, 선호하는 프로그램으로 만들 수도 있습니다. 이런 그래프들은 GNU plitutils를 사용한 결과물로 교체되어야합니다.

문헌을 참고할 때는 그 분야의 가장 자명하고, 표준적이며 좋은 문헌을 참고해야합니다. 많이 일어나는 일이지만 덜 알려진 교재나 입문서(예를 들어 학부에서 사용되기 위한)의 참고는 지양해야 합니다. 각 분야의 자명한 참고 문헌의 예로 알고리즘은 Knuth[5] , 통계학은 Kendall & Stuart[6] , 특수 함수들은 Abramowitz & Stegun (Handbook of Mathematical Functions AMS-55) 등이 있습니다.

표준 참고 문헌들은 라이브러리 사용자들에게 더 좋은 접근성을 제공해 줍니다. 만약 이러한 문헌을 사용할 수 없다면 사용자가 문헌을 참고하기 위해 서적을 구입해야 하는 상황을 위해 가능한 한 고품질의 서적을 사용해야 합니다. 고품질의 기준은 GSL 사용 설명서에서 다루는 다른 참고 문헌들을 최대한 많이 다루는 서적을 의미합니다. 서로 다른 책들이 너무나 많이 인용되어 있다면 알고리즘의 세부 사항들을 보기 위해 문헌을 참고해야 하는 사용자들에게 매우 비효율적이고 비싼 희생을 강요하게 됩니다. 참고 문헌들은 일반적인 대학 교재들 보다 판본이 더 오래 유지되어야 합니다. 대학 교재들은 몇년만에 판본이 바뀌는 경우가 흔합니다.

비슷하게 될 수 있으면 원 논문을 인용해야합니다. 그리고 해당 문서들의 복사본은 나중에 사용할 수 있도록 잘 보관하는 것이 좋습니다. 예를 들어 버그 보고나 앞으로 유지 보수에 필요할 수도 있기 때문입니다.

문헌을 찾아보기 위해 도움이 필요하다면 `gsl-discuss` 메일링 리스트에 도움을 청할 수 있습니다. GSL 개발자들이 논문의 복사본을 얻는 것을 돕기 위한 봉사자 집단이 있고 그들은 좋은 고품질 자료들(도서관)에 접근할 수 있습니다.

참고: James Theiler 왈

그리고, 소프트웨어 문서화에 열과 성을 다할 것을 약속합니다. 이러한 문서화에는 왜 소프트웨어를 사용해야하는지, 정확히 어떤 기능을 하는지, 어떻게 정확한 호출을 할 수 있을지, 대략적으로 어떻게 알고리즘이 작동하는지, 어디서 알고리즘을 얻었는지, 그리고 우리가 작성하지 않은 부분들은 어디서 코드를 얻었느지를 포함할 것입니다. 우리는 모든 패키지를 계산 알고리즘으로 부터 새로 구축하는 것을 추구하지 않습니다. 이러한 재구축 보다는 이미 존재하는 자유롭게 사용가능한 수학 소프트웨어들의 집합체로써 사용되길 원합니다. 또, 우리가 작성하는 이 소프트웨어도 동일하게 사용될 수 있길 바랍니다.

---

[5] The Art of Computer Programming (TAOCP) (*)
[6] THE ADVANCED THEORY OF STATISTICS (*)

바.3.11 네임 스페이스

모든 외부 호출용 함수와 변수들은 `gsl_` 접두사를 가집니다.

모든 외부 호출용 메크로들은 `GSL_` 접두사를 가집니다.

모든 외부 호출용 헤더 파일들은 접두사 `gsl_` 로 시작하는 이름을 가져야 합니다.

설치되는 모든 라이브러리는 `libgslhistogram.a` 와 같은 이름을 가져야 합니다.

실행 가능한 모든 설치 프로그램(예를 들어 유틸리티 프로그램들)들은 접두사 `gsl-` 을 가져야합니다. (-하이폰(hypen)입니다. _ (underscore)가 아닙니다.)

모든 함수, 변수 이름등은 소문자로, 매크로와 전처리 변수들은 대문자로 써야합니다.

일부 변수와 함수 이름에 대해 추가로:

`1p`

: $+1$ 을 의미합니다. 예: 함수 `log1p(x)` 나 `k1p` 와 같은 변수. 이 변수는 $= k + 1$ 을 의미합니다.

`m1`

: -1 을 의미합니다. 예: 함수 `expm1(x)` 나 `km1` 와 같은 변수. 이 변수는 $= k - 1$ 을 의미합니다.

바.3.12 헤더 파일

헤더 파일들은 반드시 한 번만 포함되어야 합니다. 이를 idempotent 하다라 부릅니다. 예를 들어, 헤더 파일의 내용을 전처리 문구로 감싸서 이를 가능하게 할 수 있습니다.

```
#ifndef __GSL_HISTOGRAM_H__
#define __GSL_HISTOGRAM_H__
...
#endif /* __GSL_HISTOGRAM_H__ */
```

바.3.13 대상 시스템

목표로 하는 대상 시스템은 IEEE 대수를 사용하고, 표준 C 라이브러리를 모두 사용가능한 ANSI C 시스템입니다.

바.3.14 함수 이름

각각의 모듈 이름들은 그 모듈 안의 함수들 이름에 접두사로 작용합니다. 예를 들어서 gsl_fft 모듈에는 gsl_fft_init 함수가 있습니다. 모듈들은 라이브러리 소스 트리의 하위 디렉토리들과 대응됩니다.

바.3.15 객체 지향성

알고리즘들은 ANSI C에서 허용하는 한, 객체 지향적이어야 합니다. 캐스팅의 사용이나 상속을 구현하려는 편법은 권장하지 않고 이들과 비슷한 기능들도 작성하지 않도록 주의해야 합니다. 이는 많은 코딩 패턴들을 금지하지만, 해당 패턴들이 라이브러리에 사용하기에는 너무나 복잡하기 때문에 고려하지 않을 것입니다.

참고: C에서 함수 포인터를 사용해 추상화된 기초적인 클래스를 정의할 수 있습니다. rng 디렉토리를 보면 예시를 볼 수 있습니다.

자유롭게 이용가능한 포트란 코드를 재구현 할 때는 해당 코드를 그대로 배열로 옮기기 보다는 구조체 형태의 적절한 객체를 선언해주시길 바랍니다. 구조체는 파일 내부에서 사용할 때만 유용할 수도 있어 반드시 사용자들에게 제공하지는 않아도 됩니다.

예를 들어서 어느 포트란 프로그램이 다음과 같이 반복작업을 하는 부분이 있다면,

```
SUBROUTINE RESIZE (X, K, ND, K1)
```

$X(K, D)$ 는 $X(K1, D)$ 로 조정될 격자를 의미합니다. 이러한 형태는 구조체를 도입해 좀 더 읽기 편한 형태로 만들 수 있습니다.

```
struct grid {
    int nd;    /* number of dimensions */
    int k;    /* number of bins */
    double * x;     /* partition of axes, array of size x[k][nd] */
}

void resize_grid (struct grid * g, int k_new)
{
    ...
}
```

비슷하게, 단일 파일 내에서 반복적으로 사용되는 코드가 있을 경우 정적 함수나 정적 인라인 함수를 정의해서 사용할 수 있습니다. 이는 코드를 typesafe하게 하고, 해당 코드를 사용하는 모든 곳에서 동일한 기능을 하도록 보장해 줍니다.

바.3.16 주석

GNU 표준 코딩 규약을 따릅니다. 인용구는 다음과 같이 쓸 수 있습니다.

"완전한 문장을 쓰고 첫 단어는 대문자를 써야합니다. 문장의 시작을 소문자인 식별자로 해야한다면 대문자로 바꾸면 안됩니다. 철자를 변경하면 다른 식별자를 의미합니다. 소문자로 문장이 시작되길 원치 않는다면 문장을 다르게 써야합니다(예: "The identifier lower-case is …")."

바.3.17 최소화 된 구조

구조를 최소화하길 바랍니다. 예를 들어 여러 개의 알고리즘들로 풀 수 있는 문제가 있다면 각 경우를 다룰 수 있는 분리된 구조체를 만드는 것이 더 좋습니다. 다시 말해, 런타임 식별자 사용은 권장하지 않습니다. 해당 상황의 예시로 미분값 정보가 있거나 없는 경우를 모두 사용하는 상황이 있습니다.

바.3.18 알고리즘 분해

반복 알고리즘들은 INITIALIZE(초기화), ITERATE(반복), 그리고 TEST(검증) 단계로 분해해 사용자가 반복 과정을 제어가능하게 하고 중간 단계에서 값을 확인 할 수 있게 해야합니다. 이러한 방식은 call-back을 사용하거나 flag를 이용해 중간 값을 출력하도록 제어하는 것보다 더 좋습니다. 사실 call-back의 사용은 권장하지 않습니다. 만일 call-back의 사용이 필요하다면, 이는 알고리즘을 더 세분화해 사용자가 완전히 제어 가능하도록 만들어야한다는 뜻입니다.

예를 들어서 미분방정식을 풀 때 사용자가 개별적인 단계의 해를 실시간으로 확인하며 진행해야 할 경우가 있습니다. 이러한 상황은 알고리즘이 각 단계별로 분해된 형태일 때만 사용 가능합니다. 높은 수준으로 추상화된 분해 알고리즘은 이러한 유연성 측면에서 적절하지 않습니다.

바.3.19 메모리 할당과 소유권

heap 영역에 할당되어야 하는 함수들은 `_alloc` 으로 끝나야 합니다(예: `gsl_foo_alloc`). 그리고 `_free` 가 붙은 대응 함수로 해제되어야 합니다(`gsl_foo_free`).

부분적으로 초기화된 객체에서 오류를 반환해야 하는 경우 함수에 의해 할당된 메모리를 반드시 해제해야 함을 명심해야 합니다.

> **위험:** 절대로, 함수 내부에서 임시로(temporarily) 메모리를 할당하고 반환 전에 해제하면 안됩니다. 이는 사용자의 메모리 할당 관리를 방해합니다.

모든 메모리는 할당과 해제가 각각 분리된 함수로 구현되어야 하고, **작업 공간** 인자를 전달받아야 합니다. 이 방법을 이용하면 메모리 할당을 세부적인 반복 과정에서 고려하지 않아도 됩니다.

소유권의 혼동을 방지하기 위해, 작업 공간은 다른 작업공간을 소유하면 안됩니다. 각기 다른 상황에서 정확하고 손쉬운 사용을 위해, 작업 공간은 다른 객체로부터 파생되기 보다는 정수 인자를 이용해 생성되어야 합니다.

바.3.20 메모리 레이아웃

라이브러리에서 행렬과 벡터들을 저장하는 데 C 스타일의 포인터-포인터 배열이 아니라 메모리 블럭을 이용합니다. 행렬은 행 순서로 저장되며, 열은 메모리를 따라 연속적으로 저장됩니다.

바.3.21 선형대수 단계

선형 대수학에서 쓰이는 함수는 두가지 단계로 나뉘어져있습니다.

1차원 함수들은 C 형식 인자들 (`double *, stride, size`) 을 사용해, 일반적인 C 프로그램에서 `gsl_vector` 함수들을 호출할 필요 없이 간단하게 사용할 수 있습니다.

이 라이브러리의 구현체는 학습 곡선의 최소화를 목표로 합니다. 만약, 어느 사용자가 어느 함수(예를 들어 `fft` 등의)를 사용한다고 했을 때, `gsl_vector` 의 기능을 배우는 데 시간을 쏟지 않아도 되는 상황을 목적으로 합니다.

여기서 왜 행렬에 대해서는 같은 방식을 사용하지 않는지 궁금할 수 있습니다. 행렬의 경우 인자 리스트가 (`size1, size2, tda`) 로 너무 길고 복잡하며, 행과 열의 순서에서 잠재적인 모호성을 피할 수 없기 때문입니다. 이러한 경우에는 `gsl_vector` 와 `gsl_matrix` 를 사용하는 것이 사용자에게 더 편리합니다.

때문에, 라이브러리에서 사용하는 두 단계 구분은 C 타입들에 기반한 저수준 1차원 연산들과 `gsl_matrix` 와 `gsl_vector` 에 기반한 고차원 선형 대수 연산들로 나뉘어져 있습니다.

물론, 벡터로 정의된 저수준 함수들을 정의할 수도 있습니다. 필수적인 기능이 아니라, 아직 구현이 되지않았습니다. 하지만, C 인자들에 `v->data` , `v->stride` , `v->size` 를 대신 입력해 간편하게 사용할 수 있습니다. 저수준의 `gsl_vector` 함수는 많은 편의성을 제공해 줄 수 있습니다.

효율성을 위해 라이브러리 내에서는 BLAS 기능들을 주로 사용하길 바랍니다.

바.3.22 오차 추정

특수 함수들에서 오차의 크기는 “가우스” 오차의 2배로 정해져 있습니다. 이는 2σ 값으로 계산 결과는 98%의 정확도를 가집니다. 함수의 참 값이 계산된 값의 +/- 오차 범주 내에 있음을 기대할 수 있습니다. (1σ 의 경우 32% 비율을 차지해 기대할 수 없습니다.) 물론 실제 계산된 결과의 오차가 가우스 오차는 아니지만 이 두 값들은 실용적으로 잘 쓰입니다.

바.3.23 예외와 오류 관리

기본적인 오류 관리 절차는 오류 값의 반환입니다(gsl_errno.h 에서 가능한 값들을 참고할 수 있습니다). GSL_ERROR 메크로를 사용해 오류를 표시할 수 있습니다. 현재 이 매크로의 정의는 완전하진 않지만, 컴파일 시간에 변경될 수 있습니다.

오류를 나타낼 때, 오류 값을 반환하기 보다 항상 GSL_ERROR 매크로를 사용해야 합니다. 이 매크로는 사용자가 해당 오류들을 디버거를 이용해 잡을 수 있게 해줍니다(gsl_error 함수의 중단점을 정의해서 사용할 수 있습니다).

GSL_ERROR 매크로를 사용하지 말아야 할 상황은 반환 값이 오류를 나타내기보다는 특정한 표기를 위한 경우입니다. 예를 들어서 반복 작업등에서 반환 값은 각 반복 단계의 성공, 실패등을 나타낼 수 있습니다. 일반적으로 반복 알고리즘의 "실패"(GSL_CONTINUE 를 반환합니다.)는 빈번히 일어나는 일이고 이런 경우에 GSL_ERROR 를 사용할 필요는 없습니다.

특정 초기화 객체를 이용한 작업에서 발생한 오류와 같이, 사전에 할당된 메모리에서 오류가 발생했다면, 해당 메모리를 해제하는 것을 잊으면 안됩니다.

바.3.24 영속성

라이브러리를 개발할 때 메모리 블럭을 사용하는 객체(예: vector , matrix , histogram) foo 를 만든다 칩시다. 이 경우 이러한 블럭들을 읽고 쓸 수 있는 함수들을 제공해야 합니다.

```
int gsl_foo_fread (FILE * stream, gsl_foo * v);
int gsl_foo_fwrite (FILE * stream, const gsl_foo * v);
int gsl_foo_fscanf (FILE * stream, gsl_foo * v);
int gsl_foo_fprintf (FILE * stream, const gsl_foo * v, const char *format);
```

이 함수들은 오직 메모리 블럭들만을 인자로 가져야 합니다. 블럭의 길이와 같은 연관된 인자는 가지면 안됩니다. 이는 사용자들이 라이브러리에서 제공하는 함수들을 이용해 고수준의 입/출력 기능들을 작성할 수 있도록 하기 위함입니다. fprintf/fscanf 버전의 함수들은 아키텍처 사이에서 이식 가능하도록 작성되어야 하며, 바이너리 버전은 raw 형태의 데이터를 사용해야 합니다. 다음과 같이 실제로 읽고 쓰는 함수들을 구현하면 됩니다.

```
int gsl_block_fread (FILE * stream, gsl_block * b);
int gsl_block_fwrite (FILE * stream, const gsl_block * b);
int gsl_block_fscanf (FILE * stream, gsl_block * b);
int gsl_block_fprintf (FILE * stream, const gsl_block * b, const char *format);
```

```
int gsl_block_raw_fread (FILE * stream, double * b, size_t n, size_t stride);
int gsl_block_raw_fwrite (FILE * stream, const double * b, size_t n, size_t stride);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
int gsl_block_raw_fscanf (FILE * stream, double * b, size_t n, size_t stride);
int gsl_block_raw_fprintf (FILE * stream, const double * b, size_t n, size_t stride, const char␣
↪*format);
```

바.3.25 반환값 사용

반환값들을 사용하기 전에 항상 변수에 할당을 하고 사용해야합니다. 이 방법은 함수의 디버깅과 반환값의 검사 수정을 용이하게 해줍니다. 만약, 변수가 중요치 않고 임시로 사용된다면, 적절한 범주 내에 포함시켜 사용해야 합니다.

예를 들어서 다음과 같이 쓰기보다는,

```
a  = f(g(h(x,y)))
```

중간값을 저장하는 임시 변수들을 사용해 다음과 같이 작성해야 합니다.

```
{
    double u = h(x,y);
    double v = g(u);
    a = f(v);
}
```

이러한 작성법은 디버거에서 좀 더 쉽게 검사를 수행할 수 있게하며, 정지점(breakpoint)을 더 정확하게 잡을 수 있게해줍니다. 프로그램의 최적화를 수행하는 컴파일러에서는 이러한 임시 변수들이 자동으로 제거됩니다.

바.3.26 변수 이름

변수 이름에 다음의 이름 규약들을 사용해야 합니다.

`dim` : 차원의 수.

`w` : 작업 공간을 가르키는 포인터.

`state` : 상태 변수를 가르키는 포인터. (문자를 저장해야 한다면, `s` 를 사용합시다.)

`result` : 결과(반환 값) 포인터.

`abserr` : 절대 오차.

`relerr` : 상대 오차.

`epsabs` : 절대 허용 오차

`epsrel` : 상대 허용 오차

size : 배열이나, 백터의 크기. 예: double array[size]

stride : 벡터의 stride

size1 : 행렬의 행 갯수.

size2 : 행렬의 열 갯수.

n : 일반적인 정수. 예: 배열의 원소 숫자, fft 등등.

r : 난수 발생자 (gsl_rng).

바.3.27 자료형 크기

ANSI C가 제공하는 int 자료형은 16bit 크기를 보장함을 명심해야합니다[7] . 시스템에 따라 더 큰 크기를 제공할 수도 있지만 해당 자료형의 크기는 C에서 보장하지 않습니다. 따라서, 32bit 크기의 자료형이 필요하다면 long int 를 사용해야 합니다. 이 데이터형은 최소 32bit의 크기를 보장합니다. 물론 많은 플랫폼에서 int 자료형의 크기가 32bit인 경우가 많습니다. 하지만 이 라이브러리의 코드들은 특정 플랫폼보다는 ANSI 표준을 준수할 것입니다.

바.3.28 size_t

모든 객체(예: 메모리 블럭)들은 size_t 로 크기가 측정되어야 합니다. 따라서, 모든 반복 과정(예: for(i=0; i<N; i++))은 size_t 의 형태를 가지는 인덱스를 사용해야 합니다.

int 와 size_t 를 혼용하면 안됩니다. 이 둘은 교환 **불가능** 합니다.

감소하는 반복문을 사용하고 싶다면 주의해야 하는 데, size_t 자료형은 부호가 없는 자료형이기 때문입니다. 일반적인 감소 반복문보다는,

```
for (i = N - 1; i >= 0; i--) { ... } /* DOESN'T WORK */
```

다음과 같이 쓰는 것을 권장합니다. 이는 i=0 근처에서 발생하는 문제를 해결해줍니다.

```
for (i = N; i > 0 && i--;) { ... }
```

혼동을 피하고 싶다면 독립적인 변수를 반복문 안에 삽입해 반복 순서를 반대로 바꾸는 것이 좋습니다.

```
for (i = 0; i < N; i++) { j = N - i; ... }
```

[7] int는 플랫폼에 따라서 다양한 크기를 가질 수 있습니다. 어떤 플랫폼에서는 32bit, 64bit 크기를 가지고 어떤 플랫폼에서는 16bit의 크기를 가질 수도 있습니다. 대표적으로 아두이노와 같은 AVR 시스템에서 16bit 크기를 가진 경우가 흔합니다. 시스템에 따른 이러한 자료형 크기의 차이는 ISO C 표준 문서의 규약이 int 자료형의 최소 크기 16bit와 자료형에 따른 상대적 크기만을 정해 놓았기 때문입니다. 이로 인해 시스템마다 자료형의 실제 크기는 최소 크기보다 같거나 크기만 하면 다양하게 나올 수 있습니다. int 자료형은 일반적으로 구동 플랫폼의 기본 데이터 처리 타입을 따릅니다. 이는 실행 환경에서 가장 빠른 동작을 보장하기 위함입니다(*).

참고: BJG

원래 제시한 방법은

Note that the post-decrement ensures that the loop variable is tested before it reaches zero. Beware that `i` will wraparound on exit from the loop. (This could also be written as `for (i = N; i--;)` since the test for `i>0` is equivalent to `i!=0`` for an unsigned integer)

If you really want to avoid confusion use a separate variable to invert the loop order,

---

참고: BJG

Originally, I suggested using

```
for (i = N; i > 0 && i--;) { ... }
```

which makes the test for `i>0` explicit and leaves `i=0` on exit from the loop. However, it is slower as there is an additional branch which prevents unrolling. Thanks to J. Seward for pointing this out.

참고: As a matter of style, please use post-increment (`i++`) and post-decrement (`i--`) operators by default and only use pre-increment (`++i`) and pre-decrement (`--i`) operators where specifically needed.

바.3.29 배열 vs 포인터

함수의 선언과정에서 포인터 인자나 배열 인자들을 모두 사용할 수 있습니다. 표준 C에서는 이 둘이 동일하다고 간주합니다. 그러나, 실용적으로 이 둘을 구분지어서 사용하는 것이 매우 유용합니다. 포인터는 수정할 단일 객체를 나타내고, 배열은 구분 단위를 가지는 객체의 집합으로 간주합니다. 배열의 수정 여부는 `const` 의 유무에 따릅니다. 벡터의 경우 구분 단위가 별도로 필요하지 않고 포인터 형식이 선호됩니다.

```
/* real value, set on output */
int foo (double * x);

/* real vector, modified */
int foo (double * x, size_t stride, size_t n);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
/* constant real vector */
int foo (const double * x, size_t stride, size_t n);

/* real array, modified */
int bar (double x[], size_t n);

/* real array, not modified */
int baz (const double x[], size_t n);
```

바.3.30 포인터

가능한 한 수식의 우변에 포인터의 역참고를 사용하지 말아야합니다. 이러한 코드가 필요한 경우 임시 변수의 활용이 더 적절합니다. 이는 컴파일러가 최적화를 더 쉽게 할 수 있게 해주며 가독성이 좋은 코드를 만듭니다. 이에 더해 곱셈이나 역참고에 모두 쓰이는 * 기호의 혼동을 줄여줍니다.

다시 말해,

```
while (fabs (f) < 0.5)
{
  *e = *e - 1;
  f  *= 2;
}
```

보다는 다음과 같이 작성하는 것이 좋습니다.

```
{
    int p = *e;

    while (fabs(f) < 0.5)
    {
      p--;
      f *= 2;
    }

    *e = p;
}
```

바.3.31 상수화

함수의 선언에서 const 는 포인터에 의해 가리켜지고 있는 객체가 상수일 때 사용합니다. 함수나 특정 범주 내에서 의미있는 변수들도 const 를 사용할 수 있습니다. 이는 상수인 값들을 실수로 수정하는 행위들을 막아줍니다. 이러한 예시로 배열의 길이 등이 있습니다. 이러한 작성방법은 컴파일러의 최적화에도 도움을 줍니다. const 키워드는 함수로 전달되는 인자가 중요한 의미를 가질 때도 사용할 수 있습니다.

바.3.32 의사 템플릿

몇몇 의사 템플릿 매크로들이 templates_on.h 와 templates_off.h 에 있습니다. block 디렉토리에서 이 기능들의 자세한 사용을 참고해볼 수 있습니다. 가능한 한 사용을 자제해야 하는 것이 좋습니다. 이 기능들을 악몽과 같지만, 사용을 피할 수는 없었습니다.

특히, 주의할 규약은 다음과 같습니다. 템플릿들은 “data”에 작용하는 연산에만 사용되어야 됩니다. 이러한 대상으로 벡터, 행렬, 통계 그리고 정렬이 있습니다. 이 기능은 프로그램이 정해진 형태의 데이터 타입을 생성하는 외부 자료원과 함께 사용해야하는 경우를 다루기 위함입니다. 예로 8 비트로 couter로 생성되는 큰 규모의 문자 배열이 있습니다.

다른 함수들은 부동 소수점에 대해 double 을 사용하거나 정수들에 대해 적절한 정수형을 사용할 수 있습니다. 정수형의 예로 난수에 대해 unsinged long int 가 있습니다. 템플릿의 사용은 라이브러리의 전체 기능들을 제공하기 위함이 아닙니다. 이는 나무 위에서 물고기를 찾는 일과 같이 불가능한 일입니다[8] . 요약하자면, 대부분의 모든 기능들은 일반적인 용도에 적합한 **자연적인 자료형** 으로 구성되어야 한다는 것입니다. 템플릿은 다른 데이터형이 발생하는 것을 발생하는 몇몇 경우를 처리하기 위해 존재할 뿐입니다.

부동 소수점 작업에서 double 이 ‘자연적인 자료형’으로 간주됩니다. 이는 C 언어의 기본 정신이기도 합니다.

바.3.33 임의의 상수

임의의 상수 사용은 피해야 합니다.

예를 들어서, 1e-30 , 1e-100 이나 10*GSL_DBL_EPSILON 과 같은 “작은” 값들을 구현체 안에 하드 코딩하는 행위를 해서는 안됩니다. 이런 작성법은 일반적인 라이브러리에는 적합하지 않습니다[3] .

변수들의 계산은 IEEE 대수를 따라 정확히 계산해야 합니다. 만약, 계산에서 잠재적으로 오차가 중요해질 수도 있다면, 오차항을 상대적으로 계산한 후 사용자에게 제공해야합니다. 이 과정은 수식의 오차 전파를 해석적으로 분석해 제공해야합니다. 어림짐작으로 제공해서는 안됩니다.

[8] 원문은 “putting a quart into a pint pot”로 실현 불가능한 일을 일컫는 표현입니다. quart 는 약 946.353ml이고 pint 는 약 473.176ml입니다(*).

[3] 임베디드 프로그래밍에서는 시스템의 한계로 인해 이렇게 프로그래밍하기도 합니다. 하지만 GSL은 과학 계산 라이브러리입니다. 그러한 시스템은 고려 대상이 아닙니다(*).

주의 깊게 잘 설계된 알고리즘은 일반적으로 임의의 상수가 불필요하고 사용자가 중요한 계수들에 접근할 수 있어야 합니다.

예를 들어서 다음의 코드를 생각해 봅시다.

```
if (residual < 1e-30){
    return 0.0; /* residual is zero within round-off error */
}
```

이 코드는 다음과 같이 수정해 residual 값을 반환하게 해야합니다.

```
return residual;
```

이를 이용해 사용자가 residual 값이 계산에 큰 영향을 끼치는 지, 아닌 지 판단할 수 있게 할 수 있습니다.

GSL_DBL_EPSILON 과 같은 상수들을 사용하는 것이 허용되는 경우는 함수를 근사하는 경우입니다. 테일러 급수나 점근적 확장(asymptotic expansions)등을 사용할 수 있습니다. 이러한 경우 _EPSILON 접미사가 붙은 상수들은 임의의 상수가 아닌 알고리즘의 한 구성요소입니다.

바.3.34 검증

각 모듈의 구현체들은 각 기능들에 대한 적절한 검증 절차를 함께 제공해야합니다.

이러한 검증 절차는 라이브러리를 사용해 알려진 값과 일치하는 지 확인하거나, 여러번의 호출을 통해 나온 결과를 통계적으로 분석하는 프로그램들을 의미합니다. 후자의 예로 난수 생성자가 있습니다.

가장 이상적인 상황은 각 디렉토리마다 있는 검증 프로그램이 작성된 코드의 100% 를 모두 범주에 두고 있어야합니다. 자명하게도 많은 노력이 필요한 작업입니다. 따라서 가장 중요한 부분을 먼저 검증하고 나머지를 검사하는 방법이 효율적입니다. 검사 과정은 발생할 수 있는 모든 오류 조건들을 명시적으로 유발시켜 검증해야합니다. 함수가 잘못된 인자에 대해 오류를 반환하지 않는 상황은 매우 심각한 결점이기 때문입니다.

참고: Null 포인터를 검증하려하지 말아야 합니다. 사용자가 잘못된 포인터를 전달했을 경우 라이브러리에서 세그멘테이션 오류를 발생시키는 것으로 충분합니다.

검증 과정은 결정적(deterministic)으로 이루어져야합니다. gsl_test 함수를 사용해 각 기능들에 대해 독립적으로 검증을 수행할 수도 있습니다. gsl_test 함수는 주어진 기능들의 검증 결과를 독립적으로 각 줄에 PASS/FAIL 을 내보냅니다. 이를 통해 검증 실패 부분을 명확하게 판정할 수 있습니다.

1 나 0 과 같은 간단한 값들은 검증 과정에서 버그를 밝혀내지 못할 수도 있습니다. 예를 들어서, $x = 1$ 변수를 사용하는 경우 x 와 같이 잠재적 검증 실패를 피할 수 있는 값들을 검증 과정에서 사용해야 합니다.

여러 변수들을 사용해 검증을 하는 경우, 변수들 사이에 관계성이 없는지 확인해야합니다. 변수들 사이에 관계성이 있는 경우 몇몇 버그들이 자동으로 보완되어버릴 수도 있습니다.

검증 프로그램에 난수를 넣어야 할 경우 `od -f /dev/random` 을 난수의 발생원으로 사용할 수 있습니다.

검증 프로그램에서 `sprintf` 함수를 사용해서는 안됩니다. `sprintf` 함수는 검증 프로그램이 자체적으로 가지고 있는 버그를 찾기 힘들게 합니다. `gsl_test_...` 함수들은 문자열 인자들의 포멧팅을 지원합니다. 이들을 대신 사용해야 합니다.

바.3.35 컴파일

모든 컴파일 과정은 명료하게 이루어져야합니다. 컴파일 과정에서 엄격한 제약들을 넣어 추가로 검사를 수행해야 합니다.

```
make CFLAGS="-ansi -pedantic -Werror -W -Wall -Wtraditional -Wconversion
-Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings
-Wstrict-prototypes -fshort-enums -fno-common -Wmissing-prototypes
-Wnested-externs -Dinline= -g -O4"
```

그리고 `checkergcc` 를 사용해 스택(stack)과 힙(heap)에서 발생할 수 있는 메모리 문제를 검증해야합니다. `checkergcc` 는 최고의 메모리 검사 도구입니다. `checkergcc` 를 사용할 수 없다면, Electric Fence를 사용해 힙 영역을 검사해야 합니다. 아무런 검사가 없는 것보다는 좋습니다.

메모리 접근을 검사하는 데 `valgrind` 라는 새로운 도구를 사용할 수도 있습니다.

참고: `checkergcc` 의 정식 명칭은 GNU Checker 입니다. 해당 프로그램은 개발이 중단 되었고 Valgrind 를 사용할 수 있습니다. 공식 소개 페이지에서도 Valgrind를 권장하고 GNU Checker 페이지는 교육용으로 남겨두었습니다.

Electric Fence는 Bruce Perens 가 작성한 프로그램입니다. 매우 오래된 프로그램이고(manpage는 1993년도 작성되었습니다.) 리눅스 베포판 저장소 등에서 컴파일 된 패키지를 찾을 수 있습니다. github에 소스코드가 공개되어 있습니다. efence

---

라이브러리가 C++ 컴파일러(g++)로도 컴파일이 이루어지는 지 검사해야합니다. ANSI C로 작성했다면 많은 문제가 발생하지는 않을 것입니다.

바.3.36 스레드 안전성

이 라이브러리는 스레드-안전성을 가지는 프로그램이어야합니다. 모든 함수가 스레드-안전해야하며 정적 변수를 사용하지 않아야합니다.

모든 부분이 스레드-안전해야할 필요는 없지만, 안전하지 않은 부분은 명확히 해야합니다. 예를 들어서 몇몇 전역 변수들이 라이브러리의 전체 행동을 제어하기 위해 사용되기도 합니다. 이러한 예로 범위 확인 기능의 존재 유무나 치명적인 오류 호출 기능 등이 있습니다. 이 값들은 사용자에 의해 직접적으로 접근되고 통제되기 때문에 다중-스레드 프로그램에서 각각의 스레드들에 의해 수정되지 않습니다.

다중 스레드 프로그램에서 GSL 기능들을 호출할 수 없는 경우를 방지하기 위해 명시적으로 스레드 기능을 지원할 필요는 없습니다. 예로 잠금 메커니즘(locking mechanisms) 등이 있습니다.

바.3.37 법적 문제들

- 모든 기여자들은 작성한 코드들이 GNU 일반 공중 사용 허가서 (GPL) 아래에 베포됨을 명심해야합니다. 이는 당신의 고용인으로 부터 면책 특권을 가짐을 의미합니다.
- 존재하는 코드와 알고리즘들의 소유권을 명확히 이해해야합니다.
- 각 기여자들은 선호에 따라 작성한 코드들의 소유권을 유지하거나 FSF로 베포되는 것에 서명할 수도 있습니다. GPL에는 표준적인 면책 특권이 있습니다(확인해 보십시오). 면책 특권을 더 구체적으로 작성수록 고용주가 받아들일 가능성이 커집니다. 예를 들어 다음과 같이 작성할 수 있습니다.

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the software
`GNU Scientific Library - Legendre Functions' (routines for computing
legendre functions numerically in C) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

- **자명하게도, 비-자유 코드들을 사용하거나 가져오면 안됩니다.** 특히, Numerical Recipes 나 ACM TOMS 에서 코드를 가져오거나 번역해오면 안됩니다. Numerical Recipes는 제약 있는 허가서 아래에 있고 자유 소프트웨어가 아닙니다. 출판사인 Cambridge University Press는 책과 그 안의 모든 코드들에 대해 저작권을 행사할 권리가 있고 이는 함수, 변수들의 이름 그리고 수학적으로 정의된 하위식 순서도 포함합니다. GSL에 있는 기능들은 어떠한 방식으로든, Numerical Recipes을 참고하거나 기반해 있으면 안됩니다. TOMS(Transactions on Mathematical Software)에서 출판한 ACM 알고리즘은 자유 이용 저작물이 아닙니다. 물론, 인터넷에 공개되어 있기는 하나, ACM 사용자들은 특수한 비-상업적 허가서 아래에 사용가능하고 GPL과 호환되지 않습니다. 해당 허가서의 자세한 내용은 ACM Transactions on Mathematical Software의 표지나, ACM 웹사이트에서 확인가능합니다. 확실하게 자유로운 허가서 GPL이나 자유 이용 저작물 아래에서 사용가능한 코드만을 사용해야 합니다.

허가서가 없다고 해당 코드들이 자유 이용 저작물인 것이 아닙니다. 명백한 허가서 조항이 필요하고, 저자에게 재확인 해야합니다.

참고: 사견으로, 수치 해석에 관한 고전적인 책의 알고리즘들은 참고할 수 있다고 생각합니다.

BIJ 왈: 코드가 독립적으로 구현되고, 기존 소프트웨어에서 복사된 경우가 아니라면 가능합니다.

바.3.38 비 유닉스 이식성

비 유닉스 시스템에서도 이 라이브러리를 사용할 이유는 충분합니다. DOS는 무시하고, Windows95/Windows 등에서의 사용만을 고려하는 것이 현명합니다.

참고: 사견으로, 파일 이름이 길어질 수 있을 것 같습니다.

반면에 개발에 있어 비-유닉스 시스템 사용을 강요받아서는 안됩니다.

가장 좋은 방법은 "꼭 필요하지 않으면 XYZ를 사용하지 마십시오."와 같은 이식성 관련 지침을 내리는 것입니다. 그러면, Windows 유저들은 필요시 스스로 포팅을 할게 할 수 있을 것입니다.

바.3.39 다른 라이브러리와의 호환성

이 프로젝트는 다른 라이브러리들과의 호환성을 우선 순위로 두지 않습니다.

그러나 Numerical Recipes와 같이 광범위하게 쓰이는 라이브러리와 같은 경우, 해당 라이브러리의 사용을 그대로 대체 가능하다면 사용자들에게 유용할 수 있습니다. 이러한 작업이 완성된다면 해당 구현은 프로젝트와 독립적으로 관리될 것입니다.

몇몇 시스템 라이브러리들에 관한 독립적인 문제들이 있습니다. 예로 BSD 수학 함수와 `expm1`, `log1p`, `hypot` 과 같은 함수들이 있습니다. 라이브러리에 포함된 이 함수들은 가까운 시일 내에 거의 모든 플랫폼에서 사용가능해 질 것입니다.

이러한 네이티브 함수들을 작성에서 가장 좋은 방법은 시스템 공급 업체가 제공하는 라이브러리의 장점을 취할 수 있도록 작성하는 것입니다. 예를 들어서 `log1p` 는 인텔 x86 시스템에서 기계 명령어를 사용할 수 있습니다. 라이브러리에서는 `gsl_hypot` 과 같이, 필요시 자동으로 이식성있는 구현체들을 자동으로 교체하는 기능들을 `autoconf` 를 통해 제공합니다. `gsl/complex/math.c` 에서 `hypot` 가 어떻게 사용되고 있는지 참고해볼 수 있습니다. `gsl_hypot` 의 구현체와 대응되는 파일들인 `configure.in` 과 `config.h.in` 을 예시로 볼 수 있습니다.

바.3.40 병렬 처리 지원

라이브러리의 설계에서 병렬 처리 지원은 고려하지 않습니다. 병렬 처리 라이브러리는 완전히 다른 설계가 필요하고, 다른 응용 프로그램에서 필요로 하지않는 사항들을 요구합니다.

바.3.41 정밀도

알고리즘에서 분지 절단이나 다른 정밀도에 관련된 항들이 있다면 이 항들을 `GSL_DBL_EPSILON` 과 `GSL_DBL_MIN` 를 이용해 이들의 거듭 제곱과 조합으로 작성하길 바랍니다. 이러한 방법은은 각 구현체들을 다른 정밀도로 손쉽게 이식할 수 있게 합니다.

바.3.42 잡다한 사항

변수 이름에 `l` 는 사용하지 말아야 합니다. 숫자 `1` 과 구분하기 힘듭니다. 오래된 포트란 프로그램에서 매우 흔한 일이었습니다.

마지막 첨언

하나의 완벽한 구현체가 오류 있는 많은 구현체보다 낫습니다.

바.4 참고 문헌

바.4.1 수치 해석

- Numerical Computation (2 Volumes) by C.W. Ueberhuber, Springer 1997, ISBN 3540620583 (Vol 1) and ISBN 3540620575 (Vol 2).
- Accuracy and Stability of Numerical Algorithms by N.J. Higham, SIAM, ISBN 0898715210.
- Sources and Development of Mathematical Software edited by W.R. Cowell, Prentice Hall, ISBN 0138235015.
- A Survey of Numerical Mathematics (2 vols) by D.M. Young and R.T. Gregory, ISBN 0486656918, ISBN 0486656926.
- Methods and Programs for Mathematical Functions by Stephen L. Moshier, Hard to find (ISBN 13578980X or 0135789982, possibly others).
- Numerical Methods That Work by Forman S. Acton, ISBN 0883854503.

- Real Computing Made Real: Preventing Errors in Scientific and Engineering Calculations by Forman S. Acton, ISBN 0486442217.

바.4.2 표준 문헌

- Handbook of Mathematical Functions edited by Abramowitz & Stegun, Dover, ISBN 0486612724.
- The Art of Computer Programming (3rd Edition, 3 Volumes) by D. Knuth, Addison Wesley, ISBN 0201485419.

바.4.3 특정 주제

- Matrix Computations (3rd Ed) by G.H. Golub, C.F. Van Loan, Johns Hopkins University Press 1996, ISBN 0801854148.
- LAPACK Users' Guide (3rd Edition), SIAM 1999, ISBN 0898714478.
- Treatise on the Theory of Bessel Functions 2ND Edition by G N Watson, ISBN 0521483913.
- Higher Transcendental Functions satisfying nonhomogeneous linear differential equations by A W Babister, ISBN 1114401773.

바.5 이용

GNU 과학 계산 라이브러리의 소스코드와 부속 기능들은 "자유"롭게 사용가능합니다. 이 의미는 자유롭게 이들을 사용하고 재 베포할 수 있다는 뜻입니다. 그러나 이 라이브러리가 자유 이용 저작물을 의미하지는 않습니다. 저작권이 존재하며 베포 절차에 제약이 있습니다. 하지만 이 제약들은 협력을 원하는 모든 시민들이 원하는 행위를 보장하기 위해 설계되었습니다. 이 절차는 라이브러리를 이용하고 다시 베포할 때, 또다른 사람이 이 프로그램의 어느 판본도 얻지 못하게 하는 행위를 금지하고 있습니다.

분명히 하고 싶은 점은 이 프로그램의 사용자는 GNU 과학 계산 라이브러리와 관련된 프로그램의 복사본을 제공할 수 있으며, 소스코드를 받거나 원하는 경우 습득할수 있고, 이러한 프로그램을 변간하거나 새로운 무료 프로그램에서 사용할 수 있다는 점입니다. 그리고 이러한 사항들을 모두 알고 있어야 합니다.

모두가 이 권리를 가지고 있고, 어느 개인이 타인으로 부터 이 권리를 박탈하는 행위를 금지합니다. 예를 들어서 GNU 과학 계산 라이브러리와 관련된 코드를 베포할 경우, 라이브러리의 이용으로 부여된 모든 권리를 다른 사람들에게도 똑같이 허용해야합니다. 다른 사람들도 소스 코드를 제공 받거나 가져올 수 있는지 확인하고 이러한 권리를 그들에게 알려야 합니다.

또 스스로를 보호하기 위해 덧붙이면, GNU 과학 계산 라이브러리와 관련된 모든 프로그램은 특정한 보증이 없음을 모두가 알고 있어야합니다. 프로그램들이 누군가에 의해 수정되고 전달되었다면, 해당 판본의

수령자들이 받은 파일은 공식적으로 베포한 파일이 아님을 알아야합니다. 따라서 다른 이들에 의해 알려진 어느 문제들도 신용에 영향을 끼치지 말아야 합니다.

GNU 과학 계산 라이브러리와 관련 소프트웨어 이용에 관한 정확한 조건과 허가서는 라이브러리와 함께 제공되는 GNU 일반 공중 사용 허가서에서 확인할 수 있습니다.

제 사 부록

이력

GSL 프로젝트 이력

- gsl-2.7 was released in June 2021.
- gsl-2.6 was released in August 2019.
- gsl-2.5 was released in June 2018.
- gsl-2.4 was released in June 2017.
- gsl-2.3 was released in December 2016.
- gsl-2.2.1 was released in August 2016.
- gsl-2.2 was released in August 2016.
- gsl-2.1 was released in November 2015.
- gsl-2.0 was released in October 2015.
- gsl-1.16 was released in July 2013.
- gsl-1.15 was released in May 2011.
- gsl-1.14 was released in March 2010.
- gsl-1.13 was released in September 2009.
- gsl-1.12 was released in December 2008.
- gsl-1.11 was released in March 2008.
- gsl-1.10 was released in September 2007.
- gsl-1.9 was released in February 2007.

- gsl-1.8 was released in April 2006.
- gsl-1.7 was released in September 2005.
- gsl-1.6 was released in December 2004.
- gsl-1.5 was released in June 2004.
- gsl-1.4 was released in August 2003.
- gsl-1.3 was released in December 2002.
- gsl-1.2 was released in July 2002.
- gsl-1.1.1 was released in March 2002.
- gsl-1.1 was released in February 2002.
- gsl-1.0 was released in November 2001.
- gsl-0.9.4 was released in October 2001 (fifth beta-test release).
- gsl-0.9.3 was released in September 2001 (fourth beta-test release).
- gsl-0.9.2 was released in September 2001 (third beta-test release).
- gsl-0.9.1 was released in August 2001 (second beta-test release).
- gsl-0.9 was released in July 2001 (first beta-test release).
- gsl-0.8 was released in May 2001.
- gsl-0.7 was released in October 2000.
- gsl-0.6 was released in June 2000.
- gsl-0.5 was released in December 1999.
- gsl-0.4.1 was released in February 1999.
- gsl-0.4 was released in August 1998.
- gsl-0.3f was released in May 1998.
- gsl-0.3b was released in February 1998.
- gsl-0.2 was released in October 1996.
- gsl-0.1 was released in sometime in 1996.
- gsl-0.0 was released in sometime in 1996.
- The gsl project was started in May 1996 (earliest recorded changelog entry).

사용 설명서의 출판본

- GNU Scientific Library Reference Manual 3rd
  - Author(s): Mark Galassi et al
  - Publication Date: Jan 2009
  - ISBN: 9780954612078
- GNU Scientific Library Reference Manual 2nd
  - Author(s): Mark Galassi et al
  - Publication Date: Feb 2003
  - ISBN: 9780954161736
- GNU Scientific Library Reference Manual 1st
  - Author(s): Mark Galassi et al
  - Publication Date: Dec 2001
  - ISBN: 9780954161705

Network Theory Ltd

라이브러리 내부 문서에 자주 등장하는 출판사가 Network Theory Ltd 입니다. Brian James Gough 외 몇몇 사람이 운영한 회사로 자유 문서 허가서 (GFDL 외 다른 허가서들) 로 된 사용 설명서와 참고 문헌들을 출판을 주로하던 출판사입니다. 현재 이 출판사는 폐업된 상태입니다.

아래의 내용은 영국 정부의 Companies House 서비스에 기반한 내용입니다.

https://find-and-update.company-information.service.gov.uk/company/03732949

출판사 정보

- 회사명: Network Tehory Limited
- 회사 번호: 03732949
- Nature of Business (SIC): 62012 - Business and domestic software development
- 유형: 사기업
- 상태: 폐업
- 대표: Brian James Gough, 75% 이상 지분 소유
- 설립-폐업: 1999.04.15 - 2019.7.23

표 7.1: Publication History of NTL

Name	Author	Month/Year	ISBN
The Guile 2.0 Reference Manual	The Guile Developers	Nov 2011	9781906966157
R Reference Manual - Volume 4 - Methods and Tools - for R Version 2.13	R Development Core Team	Aug 2011	9781906966126
R Reference Manual - Volume 3 - Statistics - for R Version 2.13	R Development Core Team	Aug 2011	9781906966119
R Reference Manual - Volume 2 - Graphics - for R Version 2.13	R Development Core Team	Aug 2011	9781906966102

다음 페이지에 계속

표 7.1 – 이전 페이지에서 계속

Name	Author	Month/Year	ISBN
R Reference Manual - Volume 1 - Base Package - for R Version 2.13	R Development Core Team	Aug 2011	9781906966096
An Introduction to Python	Guido. Van. Rossum, and Fred. L. Drake Jr	Mar 2011	9781906966133
The Python Language Reference Manual	Guido. Van. Rossum, and Fred. L. Drake Jr	Mar 2011	9781906966140
Apache HTTP Server Reference Manual - for Apache Version 2.2.17	Apache Software Foundation	Dec 2010	9781906966034
The Org Mode 7 Reference Manual (for Org Version 7.3)	Dominik Carsten	Dec 2010	9781906966089
PostgreSQL 9.0 Reference Manual: Server Administration Guide v. 3	PostgreSQL Development Group	Nov 2010	9781906966072
PostgreSQL 9.0 Reference Manual: Programming Guide v. 2	PostgreSQL Development Group	Nov 2010	9781906966065
PostgreSQL 9.0 Reference Manual: SQL Command Reference 1B	PostgreSQL Development Group	Nov 2010	9781906966058
PostgreSQL 9.0 Reference Manual: The SQL Language 1A	PostgreSQL Development Group	Nov 2010	9781906966041

다음 페이지에 계속

표 7.1 – 이전 페이지에서 계속

Name	Author	Month/Year	ISBN
Perl Language Reference Manual - for Perl Version 5.12.1	Larry Wall et al	Jul 2010	9781906966027
XML Path Language (XPath) 2.0 Standard	XML Query and Xsl Working W3c XML Query and Xsl Working Groups	Jul 2010	9781906966010
The XML 1.0 Standard : With XML Namespaces and Information Set	W3C XML Working Group	Mar 2010	9780954612092
An Introduction to R	William N. Venables and David M. Smith	May 2009	9780954612085
GNU Scientific Library Reference Manual 3rd	Mark Galassi et al	Jan 2009	9780954612078
GNU Octave Manual Version 3	John W. Eaton, David Bateman and Soren Hauberg	Oct 2008	9780954612061
Valgrind 3.3 - Advanced Debugging and Profiling for GNU/ Linux Applications	하. Nethercote, J. Weidendorfer and Julian Seward	Mar 2008	9780954612054
The PostgreSQL Reference Manual: Server Administration Guide v. 3	Postgresql Development Group	Jun 2007	9780954612047
The PostgreSQL Reference Manual: Programming Guide v. 2	Postgresql Global Development Group	Jun 2007	9780954612030

다음 페이지에 계속

표 7.1 – 이전 페이지에서 계속

Name	Author	Month/Year	ISBN
The PostgreSQL Reference Manual: SQL Language Reference v. 1	Postgresql Global Development Group	Jun 2007	9780954612023
An Introduction to GCC	Brian J. Gough, Foreword by Richard M. Stallman	Mar 2004	9780954161798
R Reference Manual: vol.2 : Base Package	The R Development Core Team	Dec 2003	9780954612016
R Reference Manual: vol.1 : Base Package	The R Development Core Team	Dec 2003	9780954612009
The Python Language Reference Manual	Guido Van Rossum , and Fred Drake	Sep 2003	9780954161781
An Introduction to Python	Guido Van Rossum , and Fred Drake	Apr 2003	9780954161767
GNU Scientific Library Reference Manual 2nd	Mark Galassi et al	Feb 2003	9780954161736
Comparing and Merging Files with GNU Diff and Patch	Paul Eggert, David MacKenzie, and Richard Stallman	Jan 2003	9780954161750
GNU Bash Reference Manual	Brian Fox, and Chet Ramey	Jan 2003	9780954161774
Version Management with CVS	Per Cederqvist	Dec 2002	9780954161712
An Introduction to R	William N. Venables, and David M. Smith	May 2002	9780954161743
GNU Octave Manual	John W. Eaton	01 Mar 2002	9780954161729
GNU Scientific Library Reference Manual	Mark Galassi et al	Dec 2001	9780954161705

제 아 부록

GSL 설치(*)

이 단원에서는 GSL 라이브러리의 설치에 관해 다룹니다. 라이브러리를 설치한다는 것은 정적이나 공유/동적 라이브러리 파일을 다른 프로그램이나 컴파일 과정에서 사용할 수 있는 환경을 구축하는 행위를 말합니다. 단순히 베포하고 있는 `.c` `.h` 파일들을 복사해서 프로젝트 디렉토리에 포함시켜도 됩니다. 하지만 매우 번거롭고, 프로그램이 공유/동적 라이브러리를 사용할 수 없어 실행 파일의 크기가 커집니다.

이러한 라이브러리의 설치는 소스 코드를 개발 환경에서 컴파일해 구성하는 방안과 사전 컴파일된 라이브러리를 설치하는 두가지 방법입니다. GSL 라이브러리의 경우는 공식적으로 소스코드를 이용한 베포를 사용하고 있습니다.

비공식적으로 Windows, Linux, Mac에서 사전 컴파일 된 라이브러리가 존재합니다. Windows에서는 Cygwin 의 일부분으로 존재하고 Linux, Mac 등에서는 각 베포판의 패키지 저장소에 있습니다.

참고: 이러한 사전 컴파일 라이브러리는 공식 베포가 아니라는 점에 유의해야 합니다.

해당 파일을 사용할 수도 있고 소스코드를 실제 컴파일해서 설치할 수도 있습니다.

아.1 패키지 설치

아.1.1 Linux&Mac

각각 베포판의 저장소에 있는 패키지를 pacakage manager를 이용해 설치하면 됩니다.

Ubuntu/Min/Debian 계열: APT package manager

```
$sudo apt install libgsl-dev
```

Fedora/Cent/RHEL 계열: DNF package manager

```
$sudo dnf install gsl-devel
```

OSX : Homwbrew

```
$brew install gsl
```

아.1.2 Windows

Windows에서는 OS 자체적으로 저장소를 활용한 프로그램 설치를 적극적으로 활용하지 않고 사용자가 개별 파일을 알아서 설치하는 방식이 주가 되어 Linux&Mac과 같은 방법을 사용하려면 별도의 관리 프로그램을 설치하고 그 안에서 설치해야합니다.

Cygwin 의 사전 컴파일된 라이브러리를 설치하는 형식으로 사용할 수 있습니다. 아니면 Microsoft 사에서 베포한 Nuget 이라는 패키지 관리 프로그램에서 컴파일된 라이브러리를 설치할 수도 있습니다. Nuget은 Visual Studio에서 개발하고자 할 때 사용할 수 있는데, 특정 버전에서만 호환될 수도 있습니다.

- Cygwin gsl pacakage: https://cygwin.com/packages/summary/gsl.html
- Nuget gsl pacakage: https://www.nuget.org/packages/gsl-msvc-x64/

MSYS2를 설치하면 Bash, MinGW, Make 등의 환경을 윈도우에서 이용할 수 있습니다. 해당 MSYS2 쉘에서

```
$pacman -S mingw-w64-x86_64-gsl
```

를 설치하면 gsl 라이브러리를 MinGW에서 사용할 수 있습니다. MSVC 등에서 이용하려면 설치후 라이브러리 파일들을 조금 수정해 주어야합니다. 소스 코드 설치에서 dlltool 이용 단원을 참고하길 바랍니다.

아.2 소스 코드 설치

패키지를 사용한 방법은 빠르고 쉽게 설치할 수 있고 버전 관리 측면에서도 많은 유용성이 있지만 한가지 단점은 최신 공식 베포를 빠르게 반영하지는 않는다는 점입니다.

예로 Ubuntu 환경를 들어봅시다.

`$apt search libgsl` 을 사용해 저장소에서 해당 라이브러리를 검색할 경우, 다음과 같은 결과를 볼 수 있습니다. (06-29-2021 확인)

```
libgsl-dbg/focal 2.5+dfsg-6build1 amd64
GNU Scientific Library (GSL) -- debug symbols package
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
libgsl-dev/focal 2.5+dfsg-6build1 amd64
GNU Scientific Library (GSL) -- development package

libgsl23/focal 2.5+dfsg-6build1 amd64
GNU Scientific Library (GSL) -- library package

libgslcblas0/focal 2.5+dfsg-6build1 amd64
GNU Scientific Library (GSL) -- blas library package

libocamlgsl-ocaml/focal 1.19.1-2build3 amd64
GNU scientific library for OCaml

libocamlgsl-ocaml-dev/focal 1.19.1-2build3 amd64
GNU scientific library for OCaml
```

현 시점에서 최신 베포 버전은 2.7 입니다. 이렇듯 사전 컴파일된 라이브러리는 최신 버전을 충실히 반영하지 못합니다. 때문에 최신 버전의 기능들을 활용하고 싶다면 소스코드를 이용한 설치 방법을 알아두어야 할 필요가 있습니다.

상기한 설치파일을 제공하지 않는 프로젝트들도 존재하는 만큼 이러한 설치 방법을 알아두면, 나중에 다른 GNU 프로젝트들을 사용할 때 유용하리라 생각합니다.

소스 코드를 컴파일해 환경을 구성하는 과정은 크게 2가지로 나뉩니다.

1. 소스 코드를 컴파일해 라이브러리 파일 생성

   Linux/Mac : `.a` , `.la` , `.so`

   Windows: `.lib` , `.dll`

2. 컴파일러와 링크 프로그램의 검색 경로에 해당 파일들의 경로 등록

   Linux, Mac, Windows 모두 컴파일러와 링커에 라이브러리가 있는 폴더의 정보를 주어야 합니다. 공유/동적 라이브러리를 사용하는 프로그램은 시스템 PATH 등에 라이브러리 검색 경로를 입력해 주어야 합니다. 여기서는 Linux의 경우를 주로 설명합니다. Windows의 경우 Microsoft Tech Document-동적 연결 라이브러리 검색 순서 를 참고해 볼 수 있습니다.

아.2.1 GSL 설치(Linux & Mac)

GSL의 설치는 다운로드 받은 압축 파일 내의 `INSTALL` 파일에 잘 기술 되어 있습니다.

본 라이브러리는 표준 GNU 설치 절차(GNU installation procedure)를 따릅니다. 표준 GNU 설치 절차는 다음의 6가지 단계로 이루어져있습니다.

1. Prerequisites
2. Downloading the source
3. Configuration
4. Building
5. Testing(optional)
6. Final install

Prerequisites

소스코드를 이용한 설치에서는 Linux 계열의 구분(Debian, Fedora 등)이나 Mac과 차이가 없습니다. 해당 OS들 모두 GNU 프로젝트의 빌드 과정에서 필요한 사전 패키지들을 모두 활용할 수 있기 때문입니다.

소스코드 형태의 프로젝트를 설치하기 위해서는 소스코드들을 컴파일 할 수 있는 컴파일러가 필요하고 컴파일 된 파일들을 이용해 실제 사용가능한 형태로 구성하는 빌드 시스템이 필요합니다. 대부분의 GNU 프로젝트는 Makefile 을 이용해 프로젝트를 구성합니다.

따라서 소스 코드 설치전 다음 두 가지를 사용할수 있는지 확인해야 합니다.

1. 컴파일러
2. make

일반적으로 GNU 프로젝트를 설치할 때에는 GCC(Gnu Compiler Collection)을 기본으로 사용합니다. 굳이 GCC를 쓸 필요는 없습니다. 이 라이브러리는 ANSI C를 지원하는 모든 시스템과 컴파일러에서 사용가능하기 때문입니다. Clang이나 ICC, AOCC[4] 등의 다른 C 컴파일러를 사용할 수도 있습니다.

여기서는 GNU/Linux 중 Ubuntu 환경에서 Bash를 기준으로 설명을 진행합니다. Fedora 계열과 Mac은 별도로 같이 기술합니다.

[4] 각각 Intel C/C++ Compiler, AMD Optimized C/C++ Compiler를 의미합니다. GNU/Linux, Mac, Windows 모두 지원합니다. AOCC의 Windows 지원은 베타 버전에 있습니다. AOCC는 LLVM/Clang의 포크로 만들어졌습니다.

컴파일러 설치

다음을 터미널 창에 입력하면 gcc 의 설치 유무를 알 수 있습니다.

```
$gcc
```

만약, gcc 가 설치되어있다면,

```
gcc fatal error: no input files
compliation terminated
```

의 메세지가 뜰 것입니다.

```
$gcc -v
```

를 입력하면 설치된 gcc 의 버전을 확인 할 수 있습니다. 일반적으로 최신 버전의 프로그램 사용이 권장되므로 다음을 입력해 gcc 의 업그레이드 버전이 있는지 확인하고 이를 업데이트 합니다.

Ubuntu/Debian

```
$sudo apt update
$sudo apt upgrade
```

Fedora/RHEL

```
$sudo dnf check-update
$sudo dnf upgrade
```

OSX

```
$brew update
$brew upgrade
```

만약 설치되어있지 않다면, gcc 를 설치해 주어야합니다. 후술할 build-essential 을 이용해 다른 개발 도구들과 함께 한꺼번에 설치해도 됩니다.

Ubuntu와 같은 Debian 계열의 기본 저장소에서 이는 build-essential 패키지 내에 포함되어 있습니다. 이 패키지는 메타 패키지의 일종으로 다른 여러 패키지의 묶어서 한번에 설치하기 위한 패키지입니다.

다음을 입력해 build-essential 를 설치하면 프로그램 개발에 필요한 gcc , g++ , make 등의 여러 컴파일러와 유틸리티를 설치 할 수 있습니다.

다음을 입력해 build-essential 패키지를 설치합니다.

```
$sudo apt update
$sudo apt install build-essential
```

Fedora/RHEL 계열에서 비슷한 역할을 하는 패키지 묶음으로는 “Development Tools”와 “Development Libraries”가 있습니다.

```
$sudo dnf groupinstall "Development Tools" "Development Libraries"
```

OSX에서는 기본으로 clang을 비롯한 make 프로그램들이 설치 되어 있습니다.

GSL 다운로드

GSL은 Main GNU FTP site 나 가까운 GNU mirror site 에서 소스코드를 내려받을 수 있습니다.

현재 가장 최신버전은 2021년 6월 1일에 배포된 gsl-2.7 버전 입니다. 가장 최신 버전의 라이브러리를 다운로드 하고 싶다면, 다음과 같이 위의 ftp 링크에서 가장 최신 버전의 파일을 내려받거나.

```
gsl-X.Y.tar.gz
gsl-X.Y.tar.gz.sig
```

자동으로 최신 버전으로 업데이트 되는 파일을 내려받을 수도 있습니다.

```
gsl-latest.tar.gz
```

웹 브라우저를 통해 파일을 내려 받을 수 있고 터미널을 이용해 받고 싶다면 wget 나 curl 명령어를 사용하면 됩니다.

다음은 gsl-lastes.tar.gz 을 다운 받을 수 있는 명령어입니다.

```
$wget https://ftp.gnu.org/gnu/gsl/gsl-latest.tar.gz
$curl curl https://ftp.gnu.org/gnu/gsl/gsl-latest.tar.gz --output gsl_latest.tar.gz
```

Windows 에서는 wget 을 Unix 계열의 wget 을 쓰지 않고 자체 기능인 Invoke-WebRequest 의 별칭으로 정의했기 때문에 curl 과 같이 저장할 파일의 이름을 지정해 주어야 파일을 저장합니다.

```
>wget https://ftp.gnu.org/gnu/gsl/gsl-latest.tar.gz -O gsl_latest.tar.gz
```

위에서 설명한 FTP 사이트에서 .tar.gz 파일을 다운로드하고, 이를 다음의 명령어를 통해 압축을 해제합니다.

```
$tar -xvzf gsl-latest.tar.gz
```

이제 압축을 해제한 디렉토리로 들어갑시다.

```
$cd ./gsl-latest
```

Configuration

```
$./configure
```

를 입력하면 자동으로 시스템 설치 환경을 위한 Makefile 을 만들어 냅니다. 이 과정은 시간이 조금 걸립니다. 주어진 시스템과 컴파일러의 기능 지원 여부를 확인해 환경에 맞춘 Makefile을 구성하기 때문입니다.

상황에 따라 사용자 환경에 의존하는 변수들을 담은 .h 확장자의 헤더 파일을 추가로 생성할 수도 있습니다.

모든 작업이 끝나면 config.status 파일을 생성합니다. 이 파일은 shell 스크립트로 차후에 현재 빌드 환경과 같은 설정으로 프로젝트를 빌드할 수 있습니다.

기본 컴파일러는 gcc로 되어있습니다.

실행 할때 컴파일러를 별도로 지정해줄 수 있습니다. 이때, 컴파일러마다 주어진 설정 이름이 다를 수 있습니다. 해당 설정을 별도로 정해주어야 합니다.

clang과 icc등과 같이 다른 컴파일러를 사용한다면 별도로 이를 configure 스크립트에 변수로 넣어주어야 합니다.

예로 clang을 이용하면 다음과 같이 넣어줄 수 있습니다.

```
$./configure CC=clang CPP="clang -E" CFLAGS="-O3" LD="llvm-ld" OTOOL=llvm-ld AR=llvm-ar RANLIB=llvm-
↪ranlib NM=llvm-nm MC=llvmc PROF=llvm-prof AS=llvm-as
```

CC 는 명령줄 인터페이스에서 호출가능한 컴파일러의 이름을 CPP 는 전처리기 단계를 의미합니다.

더 자세한 정보는 라이브러리 베포 파일내의 configure 파일 설명서를 읽어보기 바랍니다.

Windows를 Linuex/Mac과 별개로 서술하는 이유는 이 단계 때문입니다. 해당 파일은 Shell-script를 사용하기 때문에 Windows CMD나 PowerShell에서 사용할 수 없습니다.

Building & Test

config 작업이 끝나면 만들어진 Makefile 을 이용해 소스코드를 컴파일 합니다. build-essential 에 포함된 make 유틸리티가 이 작업을 해줍니다. 다음을 입력합시다.

```
$make
```

선택사항으로 make check 라는 명령어로 패키지에 제공된 자가 검증을 진행할 수도 있습니다. (일반적으로 방금 컴파일 과정을 거쳐 생성된, 미설치된 이진 코드를 사용합니다.)

Final install

make 작업이 끝났으면 다음을 입력해 이를 설치합니다.

```
$sudo make install
```

Configre - Final Install 단계를 한번에 진행하도록 할 수도 있습니다.

```
$./configure && make && make install
```

프로그래밍 환경 구성

6 단계까지 마무리하면 GSL의 설치는 끝납니다. 기본으로 설치된 위치는 /usr/local/lib 입니다. 이 폴더 안에는 다음과 같이 .a 와 .so 확장자로 정적/동적 라이브러리가 담겨있습니다. 컴퓨터 환경에 따라 해당 디렉토리에 담겨있는 라이브러리는 다양할 수 있습니다.

```
User@COMPUTERNNAME:~$ls -l /usr/local/lib
total 47072
-rw-r--r-- 1 root root  28142836 Jul  5 22:43 libgsl.a
-rwxr-xr-x 1 root root       917 Jul  5 22:43 libgsl.la
lrwxrwxrwx 1 root root        16 Jul  5 22:43 libgsl.so -> libgsl.so.25.1.0
lrwxrwxrwx 1 root root        16 Jul  5 22:43 libgsl.so.25 -> libgsl.so.25.1.0
-rwxr-xr-x 1 root root  16451032 Jul  5 22:43 libgsl.so.25.1.0
-rw-r--r-- 1 root root   2255578 Jul  5 22:43 libgslcblas.a
-rwxr-xr-x 1 root root       948 Jul  5 22:43 libgslcblas.la
lrwxrwxrwx 1 root root        20 Jul  5 22:43 libgslcblas.so -> libgslcblas.so.0.0.0
lrwxrwxrwx 1 root root        20 Jul  5 22:43 libgslcblas.so.0 -> libgslcblas.so.0.0.0
-rwxr-xr-x 1 root root   1330608 Jul  5 22:43 libgslcblas.so.0.0.0
drwxr-xr-x 2 root root      4096 Jul  5 22:43 pkgconfig
```

이 라이브러리를 이용해 프로그램을 작성하기 위해서는 링커가 해당 라이브러리에 접근할 수 있어야 합니다. 때문에 이러한 위치를 링커에게 알려주어야 합니다.

먼저, sudo ldconfig -v 를 입력해 /usr/local/lib 가 있는지 확인합니다. 해당 파일이 없다면, 별도의 설정이 필요합니다. 다양한 방법이 존재합니다.

- 실행 중, 환경 변수 LD_LIBRARY_PATH 에 LIBDIR 추가하기

  bash 창에 다음을 입력하면 환경 변수 LD_LIBRARY_PATH 에 위치를 추가할 수 있습니다.

  ```
  LD_LIBRARY_PATH=${LD_LIBRARY_PARH}:/usr/local/lib
  export LD_LIBRARY_PATH
  ```

그러나 이 방법은 새로운 bash 창을 열 때마다 별도로 입력해 주어야합니다. 때문에, 계정의 홈 디렉토리에 있는 .bashrc 파일의 끝에 다음의 문구를 추가해줍니다[3] .

```
LD_LIBRARY_PATH=${LD_LIBRARY_PARH}:/usr/local/lib
export LD_LIBRARY_PATH
```

재부팅 후나 $source ~/.bashrc 를 입력하면 정상적으로 사용이 가능합니다.

- 링크 과정에서 환경 변수 LD_RUN_PATH 에 LIBDIR 추가하기
- -Wl, -rpath -Wl, LIBDIR 옵션을 링크에 넣기
- 관리자 권한을 얻은 후 /etc/ld.so.conf/ 디렉토리에 LIBDIR 이 있는 파일 추가하기

  libc.conf 을 만들어 주면 됩니다. 파일이름은 중요하지 않습니다. .conf 파일은 1 줄에 각각 .so 동적 라이브러리 파일들이 있는 디렉토리 경로를 작성하면 됩니다. 일반적으로 이 방법이 권장됩니다. 최신 Ubuntu에서는 기본으로 libc.conf 파일이 /etc/ld.so.conf/ 디렉토리에 있어 별도의 설정없이 설치 후 라이브러리를 바로 활용 가능합니다. libc.conf 파일의 내용은 다음과 같습니다.

```
# libc default configuration
/usr/local/lib
```

아.3 Windows

Windows 에서의 설치는 복잡합니다. 사실 1가지로 제약하면 의외로 쉽게 해결할 수 있는데 (VS studio 에서만 사용, Mingw에서만 사용 등과 같이) 설치된 모든 컴파일러에서 사용가능하게 구현하려면 결국은 소스코드를 컴파일해서 Windows의 정적/동적 라이브러리 파일을 만들어야합니다. gcc 자체가 크로스 컴파일을 지원하므로 Linux에서 Windows 라이브러리 파일을 만드는 것도 가능합니다. 하지만, Windows 환경에서 개발을 한다고 가정합시다.

근래에 나온 Windows Subsystem for Linux(WSL)를 사용하면 굳이 Windows에서 사용할 목적으로 GSL을 설치하지 않고 Windows 내의 Linux 환경에서 개발을 할 수도 있습니다.

하지만 Windows에서 사용할 어플리케이션에 과학계산이 필요할 때 GSL을 사용해서 직접 프로그램을 짜고 싶을 수도 있습니다. 이 단원은 그러한 사용자들을 위한 단원으로 Windows에서 직접 Source 파일을 컴파일해 정적/동적 라이브러리 파일을 만드는 예제를 보일 것입니다. GNU 공식 웹사이트에서는 Visual Studio 나 Cmake를 이용한 방법에 대해 소개하고 있습니다.

Building GSL on Windows Using Native Tools

Prerequisites에서 필요한 컴파일러, make 도구는 Windows에서도 설치가 가능합니다. 가장 큰 문제는 시스템을 검사해 실제 설치에 사용할 Makefile을 만드는 configure 파일이 Shell-script이기 때문에

[3] 해당 파일은 bash가 시작할 때 실행되는 명령어를 기술하고 있습니다.

Windows의 CMD나 Powershell에서 사용할 수 없다는 점입니다. 이 경우 2가지 방법이 있습니다.

1. Bash 환경을 설치해서 사용하기
2. Make 파일을 만드는 다른 방법을 사용하기

Bash 환경 설치는 다양한 프로그램에서 제공합니다. 이 문서에서는 MSYS2 를 사용할 것입니다. Make 파일을 만들 수 있는 build 도구로 Cmake가 있습니다. 이 프로그램은 GNU/Linux, OSX, Windows 모두 사용가능합니다.

아.3.1 MSYS2

MSYS2는 Windows Native 프로그램을 개발할 수 있게 해주는 도구 모음입니다. 홈페이지에서 설치 파일을 내려받아 MSYS2를 설치합니다.

https://www.msys2.org/

> **경고:** MSYS2를 사용할 때 사용자 이름에 ASCII 코드내 문자열만을 가지고 공백이 없어야합니다. 만약, 사용자 이름이 이 제약을 따르지 않는다면 컴파일, 빌드 과정에서 오류가 발생할 수 있습니다.

MSYS2는 총 6개의 하위 시스템을 가집니다. 기본 설치 위치는 `C:\mysy64` 로 이 안에 다음 6개의 하위 시스템이 있습니다. 크게 2개의 Tool-chain을 제공합니다. GCC와 LLVM/Clang입니다.

표 8.1: MSYS2 Subsystems

Subsystem	Architecture	Description
MSYS	x86_64	Main
MINGW64	x86_64	Main
MINGW32	i686	Main
UCRT64	x86_64	Main
CLANG64	x86_64	Main
CLANG32	i686	Main

첫 실행시 먼저 패키지 데이터 베이스와 시스템을 업데이트 해야합니다. 다음을 입력합시다.

참고: WSYS2에서는 pacman 이라는 패키지 관리자를 사용합니다. 이 관리자는 Arch Linux 의 패키지 관리자이기도 합니다.

```
$pacman -Syu
```

갱신을 위해서는 MYSY2의 재실행이 필요합니다. 재실행 후 다음을 입력해 패키지와 시스템 갱신을 완료합니다.

```
$pacman -Su
```

base-devel에 make가 포함되어 있습니다. GCC 나 Clang tool-chain을 설치하고 싶다면 각각 base-devel과 함께 다음의 명령어로 한꺼번에 설치할 수 있습니다.

```
$pacman -S --needed base-devel mingw-w64-x86_64-toolchain #GCC
$pacman -S --needed base-devel mingw-w64-clang-x86_64-toolchain #LLVM/Clang
```

이제 라이브러리를 빌드하기 위한 준비과정은 끝났습니다. GCC를 선택했다면, MSYS2 MinGW x64를 Clang을 설치했다면 MSYS2 MinGW Clang x64를 열고 리눅스, Mac에서의 빌드 과정을 그대로 따라하면 됩니다. 이때, 각각의 tool-chain은 `C:\mysy64` 아래의 독립된 디렉토리 `mingw64` 와 `clang64` 에서 각각 관리됩니다.

컴파일된 파일들이 각각 빌드 환경; `mingw64` , `clang64` 내의 `bin`, `lib`, `include` 에 존재합니다.

```
\mysy
    └Build Environment
        └bin
        └lib
        └include
```

이 단계에서 Windows IDE에 MYSY의 MinGW, LLVM/Clang을 컴파일러로 사용해 바로 GSL 라이브러리를 사용하는 환경으로 컴파일할 수 있습니다. 해당 컴파일러의 `bin`, `lib`, `include` 디렉토리에 컴파일된 GSL 정적/동적 라이브러리들과 헤더 파일들이 들어있기 때문입니다.

이 과정을 통해 나온 라이브러리 파일들은 `.dll` , `.dll.a` , `.a` , `.la` 파일들입니다. 해당 컴파일러들이 아닌 Windows 내의 다른 컴파일러 예를 들어 MSVS 등에서 사용하려면 `.dll` 파일외에 `.lib` 파일들이 필요합니다.

`.lib` 는 Windows에서 사용하는 정적 라이브러리 파일 포멧입니다. `.a` 는 Unix 계열에서 사용하는 정적 라이브러리 파일 포멧으로 디버그 관련 정보에 차이가 있기 때문에 단순히 확장자를 바꾸는 형식으로 사용할 수는 없습니다.

가능한 방안은 라이브러리 관리툴을 사용해 `.dll` 에서 정적 라이브러리 파일을 새로 생성하는 것입니다. 이 과정은 다음 두가지 과정을 거칩니다.

1. `.dll` 파일에서 `.def` 파일 생성
2. `.def` 파일에서 `.lib` 파일 생성

MSVS[1] 를 사용해 개발하고자 한다면 MSVS 도구를 사용하는 게 간편합니다. MSVS를 사용하지 않아도, 빌드를 위해 설치한 Tool-chain에서 관련 도구들을 제공합니다.

[1] Microsoft Visual Studio

경고: 이 라이브러리는 `autoconf` 를 사용해 라이브러리의 컴파일 과정에서 시스템과 컴파일러에 의존하는 몇몇 최적화를 수행하기도 합니다. MinGW와 Clang을 그대로 사용하면 상관 없겠지만, 이렇게 MSVC와 같은 다른 컴파일러 환경으로 라이브러리를 옮긴다면 해당 사항을 인지하고 있어야 합니다.

def 파일 생성

GCC: gendef, dlltool

LLVM/Clang: llvm-dlltool

MSVC[2] :

lib 파일 생성

디렉토리 내부에 MinGW:

- gendef 유틸로 dll -> def 생성
- dlltool or llvm-dlltool 로 def -> lib 파일 생성

Clang과 GCC를 IDE에서 컴파일러로 설치하고 링크 설정을 완료해 사용하면 됩니다. 만약, MSVC를 사용하고자 한다면 추가 작업이 필요합니다.

def -> lib

몇가지 선택 사항이 있습니다.

- dlltool: GNU binary 도구에 포함된 dll 관리 도구 입니다.
- llvm-dlltool: LLVM/Clang 도구 모음에 포함된 dll 관리 도구 입니다.
- LIB: Visual Studio의 라이브러리 관리 도구입니다. 이를 사용하려면 Visual Studio의 개발자 터미널 내에서 사용해야 합니다. 일반 CMD에서도 사용이 불가능하지는 않지만 몇가지 설정을 변경해야 합니다.

[2] Microsoft Visual C++: Microsofot 사의 MSVC는 C++ 컴파일러로 지원하는 C 표준은 Microsoft C/C++ 언어 규칙 | Microsoft Docs 를 참고할 수 있습니다.

아.4 참고 문헌

라이브러리에 관한 자세한 내용은 다음 문헌을 추천합니다. 정적(static), 공유(shared), 그리고 동적(Dynamic) 라이브러리에 관한 내용을 참고할 수 있습니다.

- David A. Wheeler, Program Library HOWTO, version 1.20, 11 April 2003, URL:https://tldp.org/HOWTO/Program-Library-HOWTO/index.html, Checked: 3.Janurary.2022.

GSL 설치 과정에서 configure 스크립트의 여러 설정 사항들은 다음을 참고할 수 있습니다.

- 베포 라이브러리 파일 내의 `INSTALL` 파일
- configure 설명 `./configure -h` 로 볼 수 있습니다.

이 문서에서 설명한 도구들의 공식 사용 설명서들을 첨부합니다.

Checked: 3.Janurary.2022

- **Bash** https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html
- **GNU/Make** https://www.gnu.org/software/make/manual/make.html
- **GNU/GCC** https://gcc.gnu.org/online/docs/
- **LLVM/Clang** https://clang.llvm.org//docs/index.html
- **Visual Studio and MSVC** https://docs.microsoft.com/ko-kr/visualstudio/windows/?view=vs-2022

Windows 에서의 설치에 사용한 도구들과 관련 내용은 다음을 참고할 수 있습니다.

Checked: 3.Janurary.2022

- **Build GSL on Windows Using Native Tools: MSVC** https://www.gnu.org/software/gsl/extras/native_win_builds.html
- **How to compile GSL for Windows** https://titanwolf.org/Network/Articles/Article?AID=02d574bd-a867-4ebf-acab-34baf0146445
- **GNU Binary Utils Manual- dlltool** https://sourceware.org/binutils/docs/binutils/dlltool.html
- **Microsoft technical documentation, Additional MSVC build tools - LIB Reference** https://docs.microsoft.com/en-us/cpp/build/reference/lib-reference?view=msvc-170

제 자 부록

참고 자료(*)

참고할 만한 서적과 자료 그리고 몇가지 팁들을 제공합니다.

자.1 C 프로그래밍

자.1.1 C 표준

ANSI C 는 미국 국립 표준 협회[3] 에서 지정한 C 표준을 말합니다. 이는 C89와 동치입니다.

C89:

C99:

C 11:

C 표준 수학 라이브러리

위에 서술한 ISO C 표준에 따른 C 라이브러리를 서술합니다. 기초적인 수학 라이브러리와 연관 기능들을 소개합니다.

[3] Americal National Standards Institute, ANSI

자.1.2 Text Encoding

ASCII 코드표[1]

ASCII는 정보 교환을 위한 미국 표준 문자표를 의미합니다[2]. 이 코드표는 7-bit 크기의 숫자로 이루어져 있으며, 0-127까지의 숫자에 표기와 제어를 위한 문자가 할당되어 있습니다.

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 = \text{print or control character}$$

0-31, 127에 제어 문자가 할당되어 있고 나머지는 출력 문자가 할당되어 있습니다.

ASCII 표는 1963년도 처음 표준안이 발표된 이후로 몇몇 개정이 있어왔습니다. 하지만, 1967년도 개정판이 가장 일반적으로 많이 쓰입니다. 다음은 1963, 1965, 1967년도 발표된 ASCII 표입니다.

DE 10진수, HEX: 16진수, OCT: 8진수, CHAR: 문자

1965년도 개정에서는 알파벳 소문자들이 추가되었고, 일부 문자들의 위치와 제어 문자들의 이름이 바뀌었습니다.

0-31, 127에 할당된 제어 문자들은 프린터 같은 기기를 제어하거나 자기 테이프와 같은 저장 장치에서 값을 읽어올 때 구분을 위한 메타 정보 제공을 위함입니다. 1963년도 버전과 이후 개정의 차이는 이러한 제어 문자들의 이름들이 통신과 파일 입출력, 교환등을 위한 이름으로 좀 더 직관적이게 바뀌었다는 점입니다.

표 9.1: ASCII 제어 문자

Control CHAR	설명
NUM	NUM은 NUM입니다.

확장된 ASCII는 1bit가 추가 된 8bit 기반의 코드로 128-255 숫자 범위를 추가로 가지며, 라틴 계열 문자에 대한 지원과 추가적인 선 기호, 수학 기호 등을 포함합니다.

UTF

UTF는 Unicode Transformation Format의 약자로 Unicode의 매핑 방식중 하나입니다.

[1] Gorn, S., Bemer, R. W., & Green, J. (1963). American standard code for information interchange. Communications of the ACM, 6(8), 422-426.

[2] American Standard Code for Information Interchange, ASCII

UCS

UCS는 Universal Character Set의 약자로 Unicode의 도다른 매핑 방식입니다.

자.1.3 서적과 문헌

시중에 다양한 C 입문, 학습서들이 나와있지만 표준적으로 쓰이는 C 서적은 두 가지가 있습니다.

- **Kernighan, B.W. & Ritchie, D.M., The C Programming Language, 2nd ANSI, Pearson Educación, 1988, isbn:9789688802052.**

  Denis Ritchie와 Brian Kernighan이 작성한 책으로 ANSI C 표준을 기반으로한 책입니다. 이 라이브러리 내에서 쓰이는 표준이 ANSI C를 기반하므로 표준을 참고하기에 충분합니다. Denis Ritchie는 C 언어를 설계하고 개발한 사람이기도 합니다[4] .

  몇가지 주의할 점은 이 책의 예제 프로그램 코드 스타일 중 일부는 현재 프로그래밍에서 사용하기에 적절치 않습니다. C에 대해 참고하되 코드의 스타일까지 따라할 필요는 없습니다. 현대적인 프로그래밍 규약과 스타일은 다른 서적들이 더 좋은 자료들을 제공해 줄 수 있습니다.

- **King, K.N., C Programming: A Modern Approach. 2nd, W.W. Norton, 2008, isbn:9780393979503.**

  K.N.King 의 저서로 C99 표준을 따릅니다. 실제 우리가 배우는 많은 편리한 C의 기능들은 ANSI C 이후 C99에서 도입된 기능들이 많습니다. TCPL이 좋은 책이고 C를 공부할 때 한번은 참고해 봐야할 책이지만 현대적인 코딩 규약들과 프로그래밍 방법들에 관한 내용은 이 책이 더 좋은 정보를 제공해 줄 것입니다.

- GNU 표준 코딩 규약

  Richard Stallman과 몇몇 GNU 프로젝트의 참여자들은 관리, 참여하는 GNU 프로젝트의 일관성, 유지 보수성 그리고 설치 용의성의 확보를 위해 GNU 프로젝트에 관한 코딩 규약을 만들었습니다. GNU 프로젝트가 C를 통해 개발되는 대규모 오픈소스 프로젝트인 만큼 C의 개발 과정에서 사용가능한 여러 유용한 정보들을 담고 있습니다.

[4] K&R이나 TCPL(The C Programming Language)로 축약해 부르기도 합니다. 간혹 K&R이 2nd 판이 아닌 1st 판본을 의미하는 경우도 있으니 조심해야합니다.

자.2 수학, 과학

참고 문헌으로 사용 가능한 몇몇 수학과 과학 문헌을 소개합니다. 자유 이용 저작물이거나 GDPL 등과 같은 자유 문서 허가서에 속하는 문서들은 공식 홈페이지거나 고품질의 디지털 문서 베포 사이트를 함께 기술합니다.

경고: 별도의 서술이 없는 이상 디지털 문서가 자유 이용 저작물인 경우는 많지 않습니다. 아래 문헌의 디지털 판본을 복사, 베포, 수정할 때는 항상 공식 홈페이지의 저작권 문항을 주의 깊게 읽어야합니다. 인용하는 경우라도 표절과 적절한 형식을 따라야합니다. 일반 출판본은 자명하므로 언급하지 않겠습니다.

- **Abramwotiz & Stegun**

  - Abramowitz, M., & Stegun, I.A.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. US Government Printing Office, Washington, 10th printing, with corrections (December 1972)

  수학 참고 문헌으로 본 라이브러리에서 Abramwotiz & Stegun을 표준으로 사용하고 있습니다. 해당 문헌은 수학계에서 자명한 표준 문헌이고 자유 이용 저작물로 사용가능한 서적이지만, 새로운 갱신이 중단되었습니다. 1964년도에 NBS[5] 에서 처음 출판된 이레로 1972년도에 마지막 "10th printing, December 1972, with corrections" 판이 출판되었고 더 이상 갱신되지 않고 있습니다.

  자유 이용 저작물이므로 자유롭게 재출판, 변형등이 가능하고 접근도 가능합니다. 일반적으로 대학 도서관에서 찾을 수 있습니다. 온라인에서 html 버전으로 서적의 출판본을 볼 수도 있습니다. Colin B. Macdonald 박사의 홈페이지 에서 제공합니다.

  온라인에 풀려있는 대부분의 PDF는 뒤틀리거나 잘린 스캔본이 대다수입니다. 실제 국내에서 구입가능한 재 출간서도 이러한 스캔본에 기반해 만들어진 경우가 있습니다. 2012년도 버밍엄(Birmingham) 대학의 Alan P. Sexton 박사는 A Resource for Scientific Document Analysis[6] 란 프로젝트로 Abramwotiz & Stegun의 고품질 스캔본을 만들어 베포했습니다. 해당 프로젝트에서는 600dpi 이상의 고품질 스캔과 몇가지 이미지 처리를 사용해 참고에 유용한 품질의 디지털 문서로 사용할 수 있습니다.

  - Abramwotiz & Stegun html 문서 : https://personal.math.ubc.ca/~cbm/aands/
  - A Resource for Scientific Document Analysis: https://www.cs.bham.ac.uk/~aps/research/projects/as/project.php

- **DLMF**

[5] National Bureau of Standards

[6] Sexton, A. P. (2012). Abramowitz and Stegun - A Resource for Mathematical Document Analysis. Intelligent Computer Mathematics, 159-168. doi:10.1007/978-3-642-31374-5_11

Abramwotiz & Stegun를 대체하기 위해 NIST[7] 에서 DLMF 프로젝트를 시작했습니다. 이 프로젝트는 Dgitial Library of Mathematical Functions 의 약자로 수학 함수들에 대한 참고 문헌을 핸드북 형태의 디지털 문서로 제공함을 목적으로 합니다. 해당 문헌은 A&S의 마지막 출간 이후 출판된 여러 새로운 함수와, 기존 함수들의 새 성질들을 포함하고 있으며 지속적으로 갱신되고 있습니다. 각 단원은 해당 함수들의 구현 알고리즘에 관한 논문을 포함하고 있으며, Software 단원에서 문서내 함수들의 구현체들에 대한 정보도 제공합니다. GSL 또한 이 책에 올라가 있습니다. Cambridge University Press 에서 2010년도에 출판된 서적 또한 판매하고 있습니다. 출판본은 NIST Handbook of Mathematical Functions 으로 명명되어 있습니다.

  - https://dlmf.nist.gov/

- **Encyclopedia of Mathematics**

  Encyclopedia of Mathematics 는 유럽 수학회 에서 관리하는 온라인 수학 백과입니다. Springer 출판사에서 Book series Encyclopaedia of Mathematics 로 출판본을 판매하고 있습니다.

  - https://encyclopediaofmath.org/wiki/Main_Page

- **Wolfram Math World**

  Wolfram Math World 는 Wolfram Research, Inc. 에서 Eric Weisstein의 프로젝트를 후원해 만들어진 온라인 수학 참고 문헌입니다. The CRC Encyclopedia of Mathematics 란 이름으로 CRC 에서 2009년도에 출판한 서적이 있습니다.

  - https://mathworld.wolfram.com

- **NIST Standard Reference Data**

  NIST에서 운영하는 사이트로 과학 분야의 인용, 참고로 쓰일 수 있는 표준 데이터들을 연구, 생성, 베포합니다. 최신 물리 상수나 재료의 특성값이 필요한 경우 유용하게 사용할 수 있습니다.

  - https://www.nist.gov/srd

자.3 과학 계산 프로그램과 라이브러리

GSL은 수치 해석을 위한 C 라이브러리입니다. 이 라이브러리를 사용해 다양한 수학 기능들을 C에서 사용할 수 있습니다. 하지만 GSL 외에도 과학 계산을 위한 많은 라이브러리와 소프트웨어들이 존재합니다. 이 단락에서는 과학 계산 분야에서 사용할 수 있는 C 라이브러리들과 관련 소프트웨어들을 간단하게 서술하고자 합니다.

GSL의 확장 기능인 라이브러리들은 별도로 **GSL Extension** 으로 표기합니다.

[7] National Institute of Standards and Technology

자.3.1 라이브러리

- GLPK

  GNU Linear Programming Kit의 약자입니다. 선형 프로그래밍을 위한 여러 기능들을 제공합니다.

  - https://www.gnu.org/software/glpk/

- NLopt

  Nonlinear optimization을 위한 라이브러리입니다. 여러가지 최적화 방법들을 적은 인터페이스로 사용할 수 있는 기능을 제공하며, C, C++, Fortran, Matlab, GNU Octave, Python, GNU Cuile, Juliaa, R, Lia, OCaml 그리고 Rust에 대한 api를 지원합니다.

  - https://nlopt.readthedocs.io/en/latest/

- FFTW

  Fastest Fourier Transform in the West의 약자로 Fast Fourier Transform 기능을 구현한 라이브러리입니다. GSL 내부 구현체와의 차이점은 GSL 구현체에서 제공하는 함수가 처리할 수 있는 크기 이상의 대규모 데이터들에 대해 고속 푸리에 변환을 수행할 수 있다는 점입니다.

  - https://www.fftw.org/

- GMP

  GNU Multiple Precision의 약자입니다. 전체 이름은 The GNU Multiple Precision Arithmetic Library 정밀한 수학 계산을 할 때 많은 경우 double, float 자료형에서 유효 숫자의 한계가 생깁니다. GMPAL는 이러한 고정밀 대수 연산을 위한 라이브러리로 메모리의 제한을 무시한다면 정밀도의 한계 없이 계산을 할 수 있습니다. GMPAL 라이브러리의 주된 활용 분야는 암호, 인터넷 보안, 대수학과 계산 대수학 등이 있습니다

  - https://gmplib.org/

- MPFR

  MPFR은 GMP에 기반한 수학 계산 라이브러리로 다중 정밀도의 부동 소수점 계산 기능과 몇몇 특수 함수들의 구현체를 제공하고 있습니다. 전체 이름은 The GNU MPFR Library 입니다.

  - https://www.mpfr.org/

- SAM

  다중 정밀도의 이산 확률 대수 연산을 구현한 C/C++ 라이브러리입니다. MPFR 라이브러리에 기반해 있습니다. 관련 이론은 다음 논문에 기반해 있습니다.

  - . Graillat, F. Jézéquel, S. Wang, Y. Zhu, Stochastic Arithmetic in Multiprecision, Mathematics in Computer Science, 5(4), pages 359-375, 2011.

  - https://www-pequan.lip6.fr/~jezequel/SAM/

- MPC

  MPC는 MPFR과 같은 목적으로 만들어진 라이브러리입니다. MPFR에서 다중 정밀도의 실수 구현체를 제공한다면 MPC에서는 다중 정밀도의 복소수 구현체와 관련 함수들을 제공합니다. 전체 이름은 GNU MPC 입니다.

  - https://www.multiprecision.org/mpc/

- Libmatheval

  심볼릭 연산을 위한 C, Fortran 라이브러리 입니다.

  - https://www.gnu.org/software/libmatheval/

- Arb

  Ball 대수를 이용한 고 정밀도를 지원하는 수학 함수 구현체들의 모음입니다. GSL 만큼이나 다양한 수학 구현체들을 지원하면 복소수 구현체들도 대부분 포함하고 있습니다. Ball 대수는 다음의 논문을 참고 할 수 있습니다.

  - Joris van der Hoeven. Ball arithmetic. 2009. ☒hal-00432152v3☒

  - https://arblib.org/

- FLINT

  FLINT는 Fast Library for Number Theory의 약자로 정수론 분야의 여러 구현체들을 제공합니다. 다중 정밀도의 정수와 유리수, 정수의 n 모듈러 연산 유한 체, p-진수 등의 기능들을 제공하고 있습니다.

  - http://flintlib.org/

CPU 의존 라이브러리

GSL 설치 단원에서 ICC와 AOCC를 언급했습니다. Intel과 AMD에서는 각자 CPU 플랫폼에서 더 높은 성능과 정확도를 가지는 수학 라이브러리를 제공합니다.

- Intel@oneMKL Mathe Kernel Library: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
- AMD Optimizing CPU Library: https://developer.amd.com/amd-aocl/

GSL Extension

이 단락의 프로그램들은 GNU 공식 홈페이지의 Extensions 목록에 기재되어 있습니다. 그중 상당수의 프로그램들이 폐업한 Network Theory Ltd 출판사의 서버에 보관되어 있어 현재 아카이브 서버에 페이지만

남아있습니다. 아래 목록은 현재 사용가능한 소프트웨어 목록입니다.

- Tensor

  다차원 배열 표현과 처리를 위한 텐서(Tensor) 라이브러리입니다.

  - https://github.com/zhtvk/tensor

기타

- GTK+

  Gimp의 그래픽 인터페이스를 위해 시작된 그래픽 라이브러리로 본격적인 소프트웨어의 GUI를 만드는 데 유용하며, 이를 이용해 사용자 정의 플롯 라이브러리를 짤 수도 있습니다.

  - https://www.gtk.org/

자.4 소프트웨어

- GNU Units

  각기 다른 측정계 (ISO MK, Internationa, Imperial yard&pound ..)로 표현된 값들을 다른 측정계로 변환해주는 프로그램입니다.

  - https://www.gnu.org/software/units/

- Gnuplot

  오픈소스 그래픽 소프트웨어입니다. C api를 제공해 C에서 여러 그래프와 3D 플롯을 그리는 데 사용할 수 있습니다.

  - http://www.gnuplot.info/

- GNU Octave

  수치 해석을 위한 고수준의 과학 계산 언어이자 소프트웨어입니다. 자체 언어가 있지만, C, C++, FOTRAN, Python 등으로 쓰인 모듈을 불러오거나 C++ 등에서 Octave 함수를 사용할 수도 있습니다. C는 C++ 함수를 호출하는 형태로 사용 가능합니다.

  - https://www.gnu.org/software/octave/index

- HDF5

  HDF5는 대용량 데이터 처리를 위한 계층적 파일 형식입니다. 개발 집단인 HDF5 Group 에서 공식적으로 C, FOTRANm C++, Java, Python 에 대한 api를 제공합니다. 막대한 데이터를 다루고자 할때, 이러한 전문 파일 형식의 사용은 크게 유용합니다.

  - https://www.hdfgroup.org/solutions/hdf5/

- Mathematica

기호 계산을 위한 프로그램 중 가장 광범위 하게 쓰이는 소프트웨어입니다. 많은 자연과학, 공학자들의 사용으로 몇몇 전공서들은 신규 개정판에서 Wolfram Language를 이용하는 문제들을 추가하거나 서적의 수학 표기를 Wolfram Math World 와 Mathmatica에서 사용가능한 형태로 바뀌어 가고 있기도 합니다[8] .

근래, Wolfram Inc는 개인 연구가와 프로그래머들을 위해 Wolfram Engine 을 무료로 공개했습니다. 해당 엔진은 Mathematica 및 관련 제품들의 핵심 엔진으로 다양한 Wolfram 사의 제공 서비스를 이용할 수 있습니다. Wolfram 사의 핵심 기능은 기호 계산 소프트웨어이나 수치적 해석 분야의 기능 또한 풍부하게 제공하고 있습니다. C를 위한 api를 제공하기도 하므로 다른 C 라이브러리에 없는 특정 함수의 기능을 C로 구현하기 전에, 검증용으로 사용해 볼 수도 있습니다. C-api는 Wolfram Language & System Document Center의 C/C++ Language Interface 문서를 참고할 수 있습니다. Wolfram 엔진을 서버에서 설치해 사용하는 방법은 Wolfram Language on Research Server 를 참고할 수 있습니다.

  - https://www.wolfram.com/mathematica/
  - https://www.wolfram.com/engine

- PHOEBE

  천체물리에서 식 현상 계산을 위한 모델링 소프트웨어 패키지 입니다.

  NASA의 지원을 받고 있습니다.

자.5 GSL 지원 HPC 서비스

HPC Service on University

- Sheffield 대학 : https://docs.hpc.shef.ac.uk/en/latest/sharc/software/libs/gsl.html?highlight=GSL
- Honkong 대학 : https://hpc.hku.hk/hpc/software/gsl/
- Queen Merry 대학(QMUL) : https://docs.hpc.qmul.ac.uk/apps/dev/numerical/gsl/
- Maryland 대학 : https://www.glue.umd.edu/hpcc/help/software/gsl.html
- Cambridge 대학: https://www.maths.cam.ac.uk/computing/software/gsl
- Case Western Reserve 대학: https://sites.google.com/a/case.edu/hpcc/home
- Louisiana 주립대 : http://www.hpc.lsu.edu/docs/guides/software.php?software=gsl
- Siegen 대학 : https://cluster.uni-siegen.de/omni/application-software/gnu-scientific-library/?lang=en

[8] 대표적인 예시가 Griffith, Introduction to Quantum Mechanics 3rd edition 입니다. 해당 서적은 2nd 판본의 수식들이 대거 개편되었고 Wolfram language 를 사용한 문제들이 추가되었습니다.

상용

- Livermore Computing Center : https://hpc.llnl.gov/software/mathematical-software/gnu-scientific-library
- : http://hpc.iucaa.in/?q=pleiadesBeginnersGuide

제 차 부록

병렬화(*)

디자인 문서에서 밝혔다시피 이 라이브러리는 병렬 처리를 고려하지 않습니다. 이 단원은 기초적인 병렬 컴퓨팅의 개요를 서술합니다. 따라서 간결한 라이브러리 구현에서 병렬화를 왜 고려하지 않는지, 어떠한 사항을 추가로 고려해야 하는지에 대해 알아봅시다.

차.1 용어

병렬 컴퓨팅, 병행 컴퓨팅, 분산 컴퓨팅 등 많은 용어가 존재합니다. 세세한 개념의 차이 자체는 있지만 이들을 완벽하게 구분할 수 있는 기준은 없습니다. 이 부록에서는 병렬 컴퓨팅이라는 용어를 사용합니다. 이 용어에서 시작해 살펴볼 것이며 나머지 개념들은 추가로 해당 내용이 나올 때 서술하겠습니다.

차.1.1 병렬 처리

1. 초고성능의 슈퍼 컴퓨터의 제작
2. 낮은 비용으로 일정 수준 이상의 계산 성능을 가진 계산 시스템의 제작

병렬 처리 구조는 이렇듯 초고성능의 구현과 재정적 효율성 측면 각각에서 모두 고려될 수 있습니다. 전자는 일반적으로 대규모 전산 유체 역학이나 천체 에너지 계산등이 필요한 시뮬레이션 등에서 사용됩니다. 일상적으로는 기상 예측 시스템에서 사용합니다. 후자는 그리드 컴퓨팅으로 잘 구현되어 있습니다. SETI@home 이나 Boinc 시스템이 그 예시입니다.

이러한 분야는 컴퓨터 아키텍처, OS의 구성과 관리 시스템 등과도 연관되어 있습니다. 세세하게 살펴볼 경우, 아키텍처의 역사, 발전, 분류, 그 위에서 돌아가는 OS의 구성, 구현 방법 관리, 병렬화를 지원하는 언어와 그 확장까지 모두 살펴볼 수 있습니다. 이들을 짧은 부록안에서 전부 서술하는 것은 불가능합니다. 간단한 개요 수준으로만 살펴보도록할 것입니다.

차.1.2 컴퓨터 아키텍처의 분류

차.2 스레드와 프로세스

차.2.1 스레드

스레드 안전과 병렬화

스레드 안전한 프로그램은 손쉽게 멀티-스레드 프로그램으로 사용할 수 있습니다. GSL의 대부분의 기능들은 디자인 특성상 스레드-안전합니다.

C에서 스레드-안전에 영향을 미치는 요인으로 다음이 있습니다.

- 전역 변수
- 정적 변수

차.2.2 프로세스

프로세스는

근래의 개인용 컴퓨터들도 멀티-코어 시스템이 매우 흔한만큼 일반 개인들도 병렬 프로그래밍을 공부할 때 하드웨어의 제약이 그리크지는 않습니다. 게임 시장의 활발한 성장에 감사합시다.

차.3 구현 라이브러리와 소프트웨어

Message-Passing Interface에 관한 표준 규약이 존재합니다. 해당 내용은 MPI-Forum <https://www.mpi-forum.org/> 에서 관리하며 표준 문서는 무료로 온라인에 pdf로 공개되어 있습니다.

- POSIX thread routines

  POSIX는 IEEE가 제정한 이식성 있는 운영체제를 위한 어플리케이션 인터페이스로 Portable Operating System Interface uniX의 약자입니다. 약자에서 보이다시피 Unix 및 Unix-like 운영체제에서 사용가능하며, 대표적으로 Linux 와 Mac의 OSX가 있습니다. 이 기능들은 Multi-thread processing을 위한 처리입니다.

- Open MPI

  Open source Message Passing Interface의 약자입니다. MPI 표준을 구현하기 위한

- **OpenMP** Open Multi-Processing의 약자로 shared

차.4 참고 문헌과 추가 자료

병렬 컴퓨팅의 개요로 다음의 서적들을 참고할 수 있습니다.

- Parallel Programming in C with MPI and Open MP
- Parallel Numerical Algorithms

라이브러리들에 관한 자세한 내용과 추가 자료들은 다음의 공식 문헌들을 참고하길 바랍니다.

-

병렬 컴퓨팅을 다루는 학술지로 다음이 있습니다.

- The International Journal of Parallel Programming

참고 문헌

- Blaise Barney, Livermore Computing (retired), Donald Frederick, LLNL, Introduction to Parallel Computing Tutorial, URL: https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial#Whatis, CHECKED: Jan 2022
- Ian Foster, Designing and Building Parallel Programs,
- Victor Eijkhout with Robert van de Geijn and Edmond Chow, Introduction to High Performance Scientific Computing, lulu, 2011, ISBN: 978-1-257-99254-6.
- Selim G. Aki, The Design and Analysis of Parallel Algortihms, Prentice Hall Publication, 1989, ISBM: 0-13-200056-3.
- Alhubail, Maitham Makki, A thread-based parallel programming library for numerical algorithms, MIT PhD Thesis, 2014, URI: http://hdl.handle.net/1721.1/90080

  C++ 기반으로 짜인 스레드 기반 병렬 처리 라이브러리에 관한 논문입니다.

- **GSL Parallel 구현 논문들**
  - Park et al, Training and Research on Computational Science and Cyber-Infrastructure, Government White paper, KISTI, Dec. 2011. ReNum: TRKO201200010737, URL: https://repository.kisti.re.kr/handle/10580/11076.
  - Aliaga J. et al. (2004) Parallelization of GSL: Architecture, Interfaces, and Programming Models. In: Kranzlmüller D., Kacsuk P., Dongarra J. (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2004. Lecture Notes in Computer Science, vol 3241. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30218-6_31
  - Aliaga, J.I., Almeida, F., Badía, J.M. et al. Toward the parallelization of GSL. J Supercomput 48, 88-114 (2009). https://doi.org/10.1007/s11227-008-0207-z

– Yustina Sri Suharini, & Melani, Parallel Programming Implementation of Williamson Array to Calculate skew-Hadamard Matrices , Vol. 9 No. 1 (2014): Jurnal IPTEK, https://doi.org/10.31543/jii.v9i1.48

제 카 부록

영문 용어(*)

본 단원은 사용 설명서 내 용어들의 영문 용어 사전입니다. 번역 과정에서 **번역 용어(원문 용어)** 와 같이 표기할 수도 있지만 내용의 길이가 길어지고 사용 설명서를 읽기 불편해지는 단점이 있습니다. 이 단원은 색인의 역할도 합니다. 번역한 용어들의 한글 용어는 색인 과정에서 오류가 발생해 해당 영문의 색인을 추가하고 이곳에 기록합니다. 찾고 싶은 한글 용어의 위치는 이곳에서 영문 용어를 찾고 해당 용어의 위치를 뒤의 색인 단원에서 찾을 수 있습니다.

번역 과정은 보편적으로 쓰이는 용어를 최대한 활용했습니다.

수학 용어의 경우 대한 수학회에서 제공하는 **수학 용어집** 을 참고했으며,

https://www.kms.or.kr/mathdict/list.html

물리 용어는 대한 물리학회에서 제공하는 **물리학 용어집** 을 참고했습니다.

https://www.kps.or.kr/content/voca/search.php

그럼에도 불구하고 상기 용어집에 없거나 직관적이지 않은 용어들의 경우 번역가가 자의로 번역하기도 했습니다. 또한, 원문 용어에 관한 정보가 번역 용어에 관한 정보보다 더 방대한 자료들이 있는 만큼, 이러한 번역 대상들에 대해 원문 용어에 관한 사전을 제공하면 사용 설명서를 읽고 관련 정보를 찾는 데 큰 도움이 됩니다.

이 용어집은 한글 자모의 배열 순서를 따르지만 받침에 따른 구분은 하지 않습니다.

ㄱ

- 가우스 난수: Gaussian Random variate
- 가우스 분포: Gauss Distribution
- 가중 잔차 제곱합: weighted sum of squared residual
- 계산 복잡도: computational complexity
- 고속 푸리에 변환: Fast Fourier Transform
- 공분산 행렬: covariance matrix
- 광자: Photon

ㄴ

- 노트: Knot
- 노이먼 함수: Neumann function

- 냉각 절차: Cooling schedule

ㄷ

- 다항식: Polynomial
- 담금질 기법: Simulated annealing method
- 단 정밀도: single precision

ㄹ

- 리드버그 상수: Rydberg inverse wavelength
- 리드버그 수: Rydberg constant
- 르장드르 다항식: Legendre polynomial
- 란덴 변환: Landen transform
- 르윈: Lewin
- 람베르트 W 함수: Lambert W Functions

ㅁ

- 뮤온: Muon
- 미세 구조 상수: The fine structure constant
- 마그네톤: Magneton
- 마일: Mile
- 밀: Mil

ㅂ

- 복소수: Complex Number
- 반 복소: half-complex
- 볼츠만 분포: Boltzmann distribution
- 분할 차분 표현: Divided Difference Representation
- 보간 다항식: Interpolating polynomial
- 베셀 함수: Bessel function
- 배 정밀도: double precision
- 벡터: vector
- 별칭: Alias
- 보어 반지름: Bohr radius
- 바른: Barn
- 분산-공분산 행렬: variance-covariance matrix

ㅃ

ㅅ

- 스레드: thread
- 실수: Real number
- 슈테판 볼츠만 방사 상수: Stefan-Boltzmann Radiation Constant

ㅆ

- 쌍극자 모멘트: Dipole moment

ㅇ

- 에어리 함수: Airy function
- 역 변환: Inverse transformation
- 알고리즘: Algorithm
- 외판원 문제: Salesman problem, Traveling Salesman Problem(TSP)
- 에르미트 보간 다항식: Hermit interpolating polynomial
- 인자: Argument
- 유전율: Permittivity
- 아보가드로 수: Avogadro's number
- 옴스트롱: Angstrom
- 인치: Inch
- 야드: Yard
- 원통형 베셀 함수: regular Bessel function
- 의사 템플릿: Pesudo template
- 이산 푸리에 변환: Discrete Fourier Transform

ㅈ

- 지오이드: Geoid
- 정적 변수: static variable
- 중성자: Neutron
- 전자 볼트: Electron Voltage
- 적분 주요값: Principal Value of Integral
- 정규화 된 수소의 구속 상태: Normalized Hydrogenic Bound States
- 자유 이용 저작물: Public domain
- 조정 계수 없는 역 변환:
- 조합적 최적화: Combinatorial optimization
- 조정된 에어리 함수: Scaled version of Airy function
- 점진적 언더플로우: Gradual Underflow

ㅉ

ㅊ

ㅋ

- 컬레 복소: conjugage-complex
- 코시 주요값: Cauchy principal value
- 쿨롱 파동 함수: Coulomb Wave Functions
- 쿨롱 파동함수의 정규화 계수: Coulomb Wave Function Normalization Constant

ㅌ

- 톰슨 단면적: Thomson cross section
- 투자율: Permeability

ㅍ

- 파동 자료표: wavetable
- 푸리에 변환: Fourier transformation
- 푸리에 계수: Fourier coefficient
- 포트란: Fortan
- 포인터: Pointer
- 표준 기체 상수: The molar gas constant
- 플랑크 상수: Plank's constant
- 파섹: Parsec
- 패덤: Fathom
- 포흐하머 기호: Pochhammer symbol

ㅎ

- 헤더 파일: Header file
- 허수: Imaginary number
- 확률적 탐색 기법: Stochastic search techiques
- 행렬: Matrix
- 해리: Nautical mile
- 항등 순열: identity permutation
- 하자드 함수: Hazard function
- 합류 초기하 함수: Confluent hypergeometric

[2]

제 타 부록

역자(*)

타.1 역자 정보

- 김현성
  - 물리 광학과 물리학부 과정중
  - 광주과학기술원(GIST)
  - 이메일: qwqwhsnote@gm.gist.ac.kr
  - 홈페이지: https://hyunseong-kim.github.io/

색인