GNU Scientific Library
Reference Manual

GNU 科学技術計算ライブラリ
リファレンス・マニュアル

第 1.11 版

マーク・ガラッシ
Mark Galassi, Los Alamos National Laboratory

ジム・デイヴィーズ
Jim Davies, Department of Computer Science, Georgia Institute of Technology

ジェイムズ・ティラー
James Theiler, Astrophysics and Radiation Measurements Group, Los Alamos National Laboratory

ブライアン・ガウ
Brian Gough, Network Theory Limited

ジェラルド・ヤングマン
Gerard Jungman, Theoretical Astrophysics Group, Los Alamos National Laboratory

マイケル・ブース
Michael Booth, Department of Physics and Astronomy, The Johns Hopkins University

ファブリス・ロッシ
Fabrice Rossi, University of Paris-Dauphine

とみながだいすけ 訳

富永大介 独立行政法人産業技術総合研究所生命情報工学研究センター

目 次

第1章　はじめに

GNU 科学技術計算ライブラリ (The GNU Scientific Library、GSL) は、数値計算のためのサブルーチン集である。使われているソースコードはすべて C 言語で GSL のために新たに書かれていて、その API (Applications Programming Interface) を利用して、別の高級言語で利用するためのラッパー・ルーチンを書くこともできる。ソースコードは GNU General Public License (GNU 一般公衆利用許諾契約) の元で公開されている。

1.1　GSL で利用できるルーチン

GSL は、以下に挙げる数値計算の幅広い分野をカバーしている。

複素数	多項式の求根法	特殊関数	ベクトルと行列
置換	組み合わせ	ソート	BLAS の利用
線形代数	CBLAS ライブラリ	高速フーリエ変換	固有値問題
乱数	数値積分	確率分布	準乱数系列
ヒストグラム	統計	モンテカルロ積分	N 項組
微分方程式	焼きなまし法	数値微分	補間
級数の加速収束	チェビシェフ近似	求根法	離散ハンケル変換
最小二乗法	最適化	IEEE 浮動小数点	物理定数
B スプライン	ウェーブレット		

このマニュアルではこれらのルーチンの利用方法を解説している。各章では関数定義およびプログラム例と、内部で使っているアルゴリズムについての参考文献があげてある。

GSL の中には、FFTPACK や QUADPACK のような信頼性の高いパブリック・ドメインのソフトウェアを GSL 開発チームが C 言語で書き直したものも含まれている。

1.2　GSL はフリー・ソフトウェアである

GNU 科学技術計算ライブラリで提供されているルーチンは「フリー・ソフトウェア」である。誰もが自由に利用でき、ほかのフリーのプログラムに組み込んで公開、再配布して構わない。しかし GSL はパブリック・ドメインではない。GSL には著作権があり、公開、配布には条件が付いている。この条件は、協力して作業を行っている善意の人たちの行おうとしていることが、すべてできるように考えて作られている。GSL の利用者がしてはならないことは、作ったソフトウェアを他の人が入手、共有しようとすることを妨げることである。

特に、GSL を使って作られたプログラムを共有する権利が誰にでもあり、希望する者にはそのソースコードを入手する権利があること、そのプログラムを改造すること、入手したプログラムの一部を利用して別のフリーのプログラムを作る権利があること、またこの条件を各利用者が知っておくことが重要である、というのが GSL 開発チームの考えである。

こういった権利が誰にもあることを保証するために、この権利は誰も他の人から奪うことができないものとする。たとえば GSL を使ったプログラムを公開、配布するときは、GSL がプログラムの製作者に与えた権利を、そのプログラムを利用しようとする人にも与えなければならない。またソースコードも、ライブラリとプログラムの両方について受け取れるようにしなければならない。そしてこの権利があることを利用者に伝えなければならない。つまりこれは、GSL はプロプライエタリな (訳注: 所有権が販売会社によって占有されている) 商用のプログラムには利用できないことを意味する。

また、GSL は無保証であるとする。これは GSL の開発者をトラブルから守るためである。誰かが GSL を改変して配布した場合、それを入手した人には、それは元の GSL とは違うものであり、その改変によって生じた不都合が GSL のオリジナルの製作者とは関係ないことがわかるようにしてほしい。

GSL および関連するソフトウェアの正式な配布条件は GNU General Public License (GNU 一般公衆利用許諾契約、付録 F 参照) である。このライセンスについては GNU プロジェクトの web サイトの、Frequently Asked Questions about the GNU GPL に詳しい記述がある。

```
http://www.gnu.org/copyleft/gpl-faq.html
```

フリー・ソフトウェア財団 (Free Software Foundataion) は、商業利用したい場合のライセンスについても相談、問い合わせを受け付けている (詳細は `http://www.fsf.org/` 参照)。

1.3　GSL の入手方法

GSL のソースコードはいくつかの方法で入手することができる。知り合いからコピーしてもらう、CD-ROM を購入する、インターネットでダウンロードする、などである。GSL を置いている公開 FTP サーバーのリストが GNU の web サイトにある。

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

GSL を利用するプラットフォームとしては、GNU C コンパイラや GNU C ライブラリの拡張機能が利用できるため、GNU システム環境が好ましい。しかし GSL は移植性に最大限に配慮して実装してあり、C コンパイラがあれば、多くの他のシステムでもコンパイルできる。上記の web サイトでは、コンパイル済みライブラリを販売する企業も紹介されている。

GSL の新しい版のリリースやアップデート、その他のアナウンスはメイリング･リスト `info-gsl@gnu.org` で行われる。以下の e-mail を送ることで、それほど投稿の多くないこのメイリング・リストに登録できる。

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

こう送ると、登録を確認するための e-mail が送られてくる。

1.4 保証について

このマニュアルで解説されているソフトウェアは、無保証であり、配布されたままの状態で改変されずに提供される。ソースコードを参照してルーチンの動作と精度を検証するか、あるいはサポートと保証を提供する企業と契約するかは、利用者の責任において判断しなければならない。ほかのより詳しいことについては GNU General Public License (GNU 一般公衆利用許諾契約、付録 F) を参照のこと。

1.5 バグレポート

現時点ですでに判明しているバグは、GSL の配布パッケージに含まれている 'BUGS' というファイルにリストアップしてある。コンパイル時に生じやすい問題については 'INSTALL' にというファイルに挙げてある。ファイルにないバグを見つけたら、bug-gsl@gnu.org に連絡してほしい。

その際、バグ・レポートには以下の情報を含めてほしい。

- GSL のバージョン
- ハードウェアと OS
- 使ったコンパイラと、そのバージョンおよびコンパイル・オプション
- バグにより、どういった現象が生じるのか
- バグを再現する短いプログラム

また、ライブラリをコンパイルするときに、最適化オプションを指定するかどうかでそのバグがどうなるかを確認してもらえれば、非常に有用な情報になる。ぜひお願いしたい。

このマニュアルの、オリジナル英語版については、間違いや欠落なども、同じところに報告してほしい。この日本語版については訳者まで、下記 e-mail にて直接ご連絡いただければ幸いである。

富永大介: tominaga@cbrc.jp

1.6 その他、問い合わせ先など

マニュアルのオンライン版やほかの関連プロジェクト、メイリング・リストのアーカイブなどは、前出の web サイトにある。

このライブラリの利用やインストールに関する質問はメイリング・リスト help-gsl@gnu.org で受け付けている。以下の e-mail を送れば、このメイリング・リストに参加できる。

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

このメイリング・リストでは、このマニュアルで触れられていないことについての質問や、GSL の開発者に対する連絡なども受け付けている。

もし研究論文などでの参考文献として GSL を示したいときは、たとえば M. Galassi et. al., GNU Scientific Library Reference Manual (2nd Ed.) ISBN 0954161734 としてもらいたい。

URL は http://www.gnu.org/software/gsl/ である。

1.7　このマニュアルの記述スタイル

このマニュアルには、キーボード入力の例が多く載せられている。端末から入力されるコマンドは以下のように書かれている。

```
$ command
```

行頭の最初の文字 `$` は端末上で表示されているプロンプトで、これは入力しなくてよい。プロンプトはシステムや利用者によって様々に異なるが、このマニュアルでは bash を想定し、基本的にドルマーク `$` とする。csh/tcsh や gnuplot など、他のインタラクティブなソフトウェアによるプロンプトには適宜、他の文字を用いる。

例示されているプログラムは、GNU の OS 上での動作を想定している。ほかのシステムでは、出力が若干異なることがある。環境変数を設定するコマンドは GNU システムで標準の Bourne シェル (bash) での例を挙げてある。

1.8　訳について

このマニュアル (の原著) を読んでいると、数値計算プログラムはアルゴリズムを理解してから使うものである、という著者たちの意図を感じることが何度もある。つまり想定されている読者は各分野の研究者であり、アルゴリズムや基本的な考え方を知っている人である。マニュアル中で利用方法についての実例が充実していることと比べると、各アルゴリズムの説明は豊富なようでいて、門外漢にはさほど懇切丁寧な記述にはなっていないと思う。それは、門外漢が「便利だから」といって安易に GSL を利用するプログラムを組み、その結果をろくに検証や考察をせずに鵜呑みにすることに対する警戒心の現れ、と言えるかもしれない。また、どのソースコードを見ても非常に分かりやすく書かれていることも、それを示すと言える。

しかしそれが GSL の日本での普及に与える影響は、必ずしもポジティブとは限らないかもしれないとの余計な心配から、訳者が勝手に説明を補ったり、図やプログラム例を追加したところがある。原文の内容は少しも失われていない、ということでどうかお許し願いたい。

人名の音訳には苦労したが、その人物の母語が何かが分かる場合にはその言語で、そうでない場合には、参考文献などに示されるその人の所属組織のある国の言語での読みを示すよう努力した。古くから知られている人物や手法などで、日本語での慣用表現のある場合にはそれにしたがった。多くの場合はウェブ上の記述を当たれば見当がついたが、どうにも困って、Mac OS X の「スピーチ/テキスト読み上げ」機能で聴いてみた発音を勘案に入れた読みもある。いずれにせよこの文書での音訳は可読性を上げるための便法のつもりであり、可能な限り元の綴りを示した。

原文に上げてある参考文献には、可能な限り当たった。直接挙げてある文献からさらにたどれるものも、可能な限りたどった。そうしたところ、アルゴリズムの原典がドイツ語やフランス語だったものもある (Akima spline の Wodicka による改良実装、Polak と Ribiére の共役勾配法)。ソ連の雑誌が原典になってるものもあった (Sobol 列生成法の論文。これ自体は幸い英語であったが、雑誌全体がロシア語から翻訳されているようであった)。しかし挙げてある参考文献はすべて英語であり、一読に価する優れたものばかりである

また、FORTRAN ライブラリを C 言語で実装し直して GSL に取り込んでいるもの (LAPACK、QUADPACK、FFTPACK、MINPACK、PPPACK) があるからか、参考文献に上がってない背景知識として、FORTRAN の仕様やそのプログラミング上の慣習があるように感じられることもあった。

原文では、多くの図が GNU plotutils を使って描かれているが、この日本語訳中の図はすべて GNUPLOT (ver. 4.2) で作成した。翻訳文中で plotutil の `graph` コマンドの例がたびたび出てくるが、それに続いて載っている図は GNUPLOT で描いたものである。文書の組版には ASCII 社が公開している pLaTeX を使った (Version 3.141592-p3.1.8)。テキスト入力には Vim (ver. 7.2) を使った。MacBook/Mac OS X 上で動作するこれらソフトウェアの作成、維持、配布にかかわる関係諸氏に深く感謝する。興味のある方は、それぞれ以下を参照されたい。

GNUPLOT
: `http://www.gnuplot.info/`
`http://takeno.iee.niit.ac.jp/%7Efoo/gp-jman/gp-jman.html` 新潟工科大学竹野研究室 (GNUPLOT 日本語マニュアルを公開している)

pLaTeX
: `http://ascii.asciimw.jp/pb/ptex/base/sources.html`

Vim
: `http://www.vim.org/`

なお、プログラム例中にコメントも日本語に訳しているが、コンパイラによってはそれを処理できずにエラーになることもあり得る。その際はコメントは削除するか英語になおせばよい。

第2章　GSL の使い方

この章では GSL を使ったプログラムのコンパイルの方法と、GSL 内での関数や変数の命名法などについて説明する。

2.1　プログラム例

GSL の使用例として、ベッセル関数 $J_0(x)$ の $x = 5$ での値を計算する短いプログラムを示す。

```
#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;
}
```

プログラムの出力を以下に示す。計算結果は倍精度である[1]。

```
J0(5) = -1.77596771314338292e-01
```

このプログラムをコンパイルするまでの様々な手順を、以降の節で説明する。

2.2　コンパイルとリンク

GSL のヘッダファイルは、'gsl' ディレクトリに置かれる。したがってプリプロセッサ命令には、インクルード・ディレクトリとして 'gsl/' をヘッダファイル名の前につける必要がある。

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

この 'gsl' ディレクトリのあるディレクトリが C プリプロセッサの`#include` 命令のデフォルトのパスに含まれてない場合は、その場所をコマンドラインで指定しなければならない。デフォルトの 'gsl' ディレクトリの場所は '/usr/local/include/gsl' である。ソースファイル 'example.c' を GNU C コンパイラの gcc でコンパイルする場合のコマンドは、一般的には以下のようになる。

[1]最後の数桁は、コンパイラやプラットフォームによって変わることがある。それは一般的なことであり、バグではない。

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

これでオブジェクト・ファイル ‘example.o’ が作られる。gcc はデフォルトではヘッダファイルを ‘/usr/local/include’ に探しに行くため、GSL がデフォルトの場所にインストールされている場合は、-I オプションは必要ない。

2.2.1 GSL をリンクするには

GSL のライブラリ本体は ‘libgsl.a’ という一つのファイルとしてインストールされる。共有ライブラリ (shared library) をサポートするシステムでは、共有ライブラリ用の ‘libgsl.so’ もインストールされる (訳注: Mac OS X では libgsl.dylib といった名前になる)。デフォルトではこれらのファイルは ‘/usr/local/lib’ に置かれる。リンカーがこれらの場所を探しに行かない場合は、コマンドラインでその場所を指定しなければならない。

ライブラリをリンクするためには、GSL 本体と、GSL でサポートする標準的な線形代数ライブラリの CBLAS を指定しなければならない。CBLAS が独自に用意されていない場合は、GSL のインストールと同時に ‘libgslcblas.a’ という名前で CBLAS がインストールされる。これは、以下のようにするとリンクできる。

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

gcc はデフォルトで ‘/usr/local/lib’ を自動的に探しに行くため、GSL がデフォルトの位置にインストールされていれば-L オプションは必要ない。

(訳注: GSL 中の関数すべてが -lgsl -lgslcblas -lm を要求するわけではなく、使う関数によっては、必ずしも -lgslcblas や -lm が必要でないこともある。)

2.2.2 別にインストールした BLAS とリンクするには

プログラムを他の CBLAS ライブラリ、たとえば ‘libcblas’ とリンクするには以下のようにする。

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

利用者のプラットフォームに最適化された CBLAS が使えるなら、それを-lcblas でリンクする方が GSL で用意しているものよりも高いパフォーマンスを期待できる場合がある。そのライブラリは標準の CBLAS と互換でなければならない。移植性が高く高性能な BLAS ライブラリに CBLAS インターフェイスをかぶせた ATLAS パッケージを利用してもよい。これもフリー・ソフトウェアであり、ベクトルや行列を扱う場合はインストールするとよい。ATLAS ライブラリとその CBLAS インターフェイスをリンクするには、以下のようにする。

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

詳細は第 12 章「BLAS の利用」を参照のこと。

2.3 共有ライブラリ

GSL の共有ライブラリ版をリンクした場合、そのプログラムの実行時に OS が必要な '.so' (訳注: または .dylib) ファイルの場所を知っておく必要がある。ライブラリが見つからなければ、以下のようなエラーが出て実行できない。

```
$ ./a.out
./a.out: error while loading shared libraries:
libgsl.so.0: cannot open shared object file: No such file or directory
```

こういった場合は、環境変数 LD_LIBRARY_PATH の定義に、ライブラリがインストールされているディレクトリを加える。

たとえば Bourne シェル (/bin/sh または /bin/bash) では以下のようにする。

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

C シェル (/bin/csh または /bin/tcsh) では、以下のようにする。

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

上の例では、C シェルの標準的なプロンプトをパーセント '%' で示している。実際の操作では、この文字を入力する必要はない。

各ユーザーまたはシステム全体のユーザーに有効なログインファイルに上のコマンドを書いておけば、シェルのセッションを立ち上げるたびに入力する手間が省ける。

実行ファイルに GSL を静的 (static) にリンクするには、gcc の -static オプションを使えばよい。

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

2.4 ANSI C 準拠

GSL は ANSI C で書かれていて、ANSI C 標準（C89）にしたがうように作られている。ANSI C コンパイラが動作するシステムなら、どこでも GSL が使えるはずである。

GSL の API も ANSI にしたがっており、非 ANSI な様々な拡張には対応していない。プログラムが ANSI 準拠になっていれば、そのプログラム中で GSL を使っても ANSI 準拠である。純粋な ANSI C と互換性のある拡張は利用可能だが、条件付きコンパイルが必要である。これにより、プラットフォーム依存の拡張と GSL を同時に利用することができる。

システムによっては ANSI C が完全に実装されていないこともある。GSL 中の関数のうちその不具合に影響されるものは、GSL のインストール時にライブラリから除外される。それらの関数を使ったプログラムは正しくない値を返す可能性があり、インストールされる GSL から問題のある関数が除外されることで、そういったプログラムはリンクできなくなる。

大域的な名前空間で他のオブジェクトと衝突するのを避けるため、ライブラリ外から参照できる関数名と変数名はすべて先頭に gsl_ が、マクロには GSL_ がつけられている。

2.5　インライン関数

キーワード inline は 標準 ANSI C（C89）では導入されていないため、GSL のデフォルトではインライン関数はライブラリの外からは見えなくなっている。しかし実行性能が非常に重要な一部の関数については、条件付きコンパイルを行うことでインライン関数として使えるようにしてある。関数のインライン版は、利用者が作ったプログラムをコンパイルする時に以下のように HAVE_INLINE マクロを定義することで利用できる。

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

autoconf が利用できる環境では、このマクロは自動的に定義される。HAVE_INLINE を定義しなければ、実行速度の遅い、インラインではない関数が使われる。

実際にはキーワードのインラインは extern inline の形で使われていて、不要な関数定義は gcc が省くようになっている。他のコンパイラで extern inline が不都合を生じる場合は、もっと厳密な autoconf を使うことができる。詳しくは付録 C「Autoconf のマクロ」を参照のこと。

2.6　long double

GSL で実装しているアルゴリズムは基本的に、倍精度実数型 double だけを使っている。倍々精度型の long double は使われない。

一つには、long double の精度はプラットフォームに依存するためである。IEEE 規格では、double の精度はどのプラットフォームでも同じだが、規格からの拡張による他の型については、その最低限の精度を規定しているだけである。

しかし long double 型のデータを外部から読み込む必要がある場合を考えて、ベクトル型と行列型には long double 版を用意している。

システムによっては、stdio.h の書式付き入出力関数 printf および scanf の long double の取り扱いが正しく実装されていないことがある。そこで GSL のインストール時には、configure がそのテストを行い、必要に応じて、問題の生じる GSL 関数をインストールしないようにすることで、不定あるいは正しくない結果が生じることを避けている。configure は、その場合に以下のように出力する。

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

これにより、インストールしようとするシステムで long double の入出力が正しく行われないと判断された場合、その入出力を利用する GSL の関数はリンクできなくなる。

long double の入出力が正しく行われないシステムで入出力をする必要に迫られた場合は、入出力にはバイナリ形式を使うか、入出力の時に変数を double に変換するかなどで対応できる。

2.7　移植性確保のための関数

移植性の高いプログラミングを可能にするため、GSL では、たとえば BSD の math ライブラリのような他のライブラリの関数をいくつか実装している。そういった関数のネイティブ版を使って

プログラムを書いても、それらが使えないシステム上では条件付きコンパイルを行うことで、その関数の GSL 版を利用することができる。

たとえば BSD の hypot 関数が使えるとき、以下のマクロ定義を ‘config.h’ に書いてインクルードすることで、どちらも利用することができる。

```
/* hypot がないシステムでは gsl_hypot を使う */
#ifndef HAVE_HYPOT
#   define hypot gsl_hypot
#endif
```

利用者が書くプログラムでは #include <config.h> を使って、hypot が使えないときはソースプログラム中の hypot を gsl_hypot に置き換えることができる。autoconf が使える場合は、この置き換えを自動化することができる。詳細は付録 C「Autoconf のマクロ」参照。

これらの関数は多くの場合、ネイティブ版ではプラットフォーム依存の最適化が行われていて有利であるため、使える場合はネイティブ版を使い、使えない場合に限って GSL 版を使うようにするのがもっともよい。GSL 自体の内部でもこういった考えで実装が行われている。

2.8 最適化された関数による置き換え

GSL の関数の一部は、プラットフォームによって、その実装が必ずしも最適とは言えないことがある。たとえば正規分布乱数の生成にはいくつかの方法があり、各方法での生成速度はプラットフォームによってまちまちである。そのため GSL ではいくつかの関数について、同じインターフェイスを持つ代替関数を実装している。GSL 標準の関数を使ったプログラムを書いておき、プリプロセッサ命令を使って代替関数に切り替えることができる。また互換性が確保されていれば、独自に実装、最適化された別の関数に切り替えることもできる。以下に正規分布乱数の生成法を切り替えて、プラットフォーム依存の関数を使う例を示す。

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

このプリプロセッサ命令は、該当する関数を使うすべてのソース・プログラムからインクルードするヘッダファイル ‘config.h’ の中に書いておく。しかしプラットフォーム依存の関数は、互換性があるとは言っても標準の関数とビット単位で全く同じ結果を返すとは限らず、特に乱数の場合はまったく異なる系列を発生することがある。

2.9　様々な数値型のサポート

GSL の多くの関数は、様々な型に対してそれぞれの関数が用意されている。それぞれの関数名は、型名による修飾子 (C++ テンプレートのひな形でもある) によって決められる。例として、仮のモジュール名が gsl_foo、関数名が fn の場合につけられる、型の異なるすべての関数名を以下に挙げる。

```
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
```

通常の精度 double はデフォルトであり、型名は付けない。たとえば関数 gsl_stats_mean は複数の倍精度実数値の平均を計算するが、関数 gsl_stats_int_mean は引数で渡された整数の平均値を (倍精度実数型で) 返す。

GSL で定義されている型、gsl_vector や gsl_matrix についても同様である。この場合も型名はその名前の後につけられる。たとえば、あるモジュールで、新しい構造体か typedef によって gsl_foo が定義されている場合は以下のようになる (訳注: foo が vector や matrix に対応する)。

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

モジュールの定義自体が型に依存している場合、GSL では各モジュール独自のヘッダファイルを各型について用意している。各ファイル名は以下に示すような付け方になっているが、手間を省くため、デフォルトのヘッダファイルがすべての型のファイルをインクルードするようになってい

る。したがって、たとえば double 型のヘッダファイルなど特定の型だけをインクルードしたい場合は、そのファイル名を指定する。

```
#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++との互換性

GSL のヘッダファイルは、C++ プログラムからインクルードされるときには自動的に関数定義に `extern "C"` をつけるようになっている。そのため、GSL の関数を C++ のプログラムから直接呼ぶことができる。

自分で書いた C++ の例外処理ルーチンを引数として GSL のルーチンに渡したいなら、GSL をインストールするときに、そのコンパイル・オプション CFLAGS に '`-fexceptions`' を付けておく。

2.11 配列引数の渡し方

ライブラリ関数側で値を変更できる引数として GSL の関数に渡される配列、ベクトル、行列は、参照渡しではないこと、およびメモリ上で各インスタンスが互いに重なってはいないことが前提になっている。この前提によりライブラリ側ではメモリ領域が重なるようなケースを特に判断する必要がなくなり、高度な最適化を利用できるようになる。書き換えられる引数としてメモリ領域が重複したオブジェクトを渡すと、結果は不定である。ライブラリ側で引数の書き換えをしないようにしている場合（たとえば関数のプロトタイプ宣言で引数に const がつけられているような場合）は、メモリ領域がエイリアスでも、また重複していても問題はない。

2.12 スレッド・セーフ

GSL の関数はマルチ・スレッドのプログラムで利用することもできる。どの関数も、静的変数を持たないという意味でスレッド・セーフである。メモリは関数ごとではなく、すべてオブジェクトごとに確保される。一時的な作業領域オブジェクトを使う関数では、それはスレッドごとに確保し

て利用するべきである。値を参照するためだけの、読み出し専用メモリとしてテーブル（表）オブジェクトを使う関数では、テーブル・オブジェクトは複数のスレッドで実行されている同じ関数で同時に使うことができる。引数として渡されるテーブル・オブジェクトは、関数のプロトタイプ宣言ではすべて `const` として定義されており、他のスレッドでも安全に使うことができる。

GSL 全体の動作を制御するための静的変数が、いくつか用意されている (範囲確認を行うか否かのフラグや、致命的エラー (fatal error) の時に呼び出す関数など)。これらの変数の値はプログラム中で直接設定できるが、プログラムの起動時に初期化として一度だけ値を設定するべきであり、他のスレッドで操作すべきではない。

2.13　廃止予定の関数

時が経つにつれ、GSL のいくつかの関数の定義を変更、あるいは削除しなければならなくなることがある。そういった場合、まずその関数は「廃止予定 (deprecated)」であるとされ、その後、次のバージョンから削除される。現在のバージョンに含まれる廃止予定の関数は、`GSL_DISABLE_DEPRECATED` を定義してコンパイルすることで使用不能にすることができる。これを使えば、すでに作ってある GSL を利用するプログラムの、将来の GSL のバージョンアップに対する互換性を確認できる。

2.14　コードの再利用

GSL のコードは、可能な限りファイル間あるいはモジュール間の依存性が生じないように書かれている。そのため、ライブラリ全体をインストールしなくても、個別の関数を抜き出して使うことができるはずである。そうして書いた自分のプログラムを単独でコンパイルするためには、`GSL_ERROR` などのマクロを定義したり、いくつか `#include` 文を削除したりする必要があるだろう。こういった再利用は、GSL のライセンスとして GNU General Public License を採用した理由の一つでもあり、GSL 開発チームとしては歓迎である。

第3章　エラー処理

この章ではプログラムの実行時に発生するエラーを検知、管理するために GSL で用意している機構について説明する。各関数が返すステータスを確認することで、関数の処理が成功したか否かを判断し、失敗している場合にはその原因を正確に知ることができる。また独自のエラー・ハンドラー関数を定義して、GSL のデフォルトの動作に代えることもできる。

この章の関数はヘッダイファイル 'gsl_errno.h' で宣言されている。

3.1　発生したエラーを知るには

GSL でのエラー検知は POSIX スレッド・ライブラリに準拠しており、スレッド・セーフである。エラーが生じたときには非零を、処理がうまくいったときには 0 を返す。

```
int status = gsl_function (...)
if (status) { /* エラーが発生した場合 */
    .....
    /* status の値が生じたエラーの種類を示している。 */
}
```

GSL のルーチンは、要求された処理ができなかった場合にエラーを返す。たとえば求根を行う関数は、要求された精度まで収束できなかったときや、繰り返し計算の回数が上限に達したときに、0 以外の値をを返す。こういったことは数値計算ではよくあることなので、関数の返り値は常に確認するべきである。

ルーチンがエラーを返したときは、返り値がエラーの種類を示している。その値と意味は C ライブラリでの errno 変数と同じ値である。ルーチンを呼び出した方では返り値を確認することで、エラー処理の動作をする、またはたいしたエラーでなければ無視する、などの対応を取ることができる。

エラーを示す返り値に加え、GSL ではエラー・ハンドラー関数 gsl_error も用意されている。この関数は、GSL 内の他の関数内でエラーが発生したときに、呼び出し元に返る直前にその関数から呼ばれる。エラー・ハンドラーのデフォルトの動作は、以下のようなエラーメッセージを表示してプログラムの実行をその場で終了させることである。

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

gsl_error エラー・ハンドラーを使えば、デバッガーでその中にブレイク・ポイントをその中に設定して、実行中にライブラリ内で生じるエラーを捕らえることができる。gsl_error は完成したプログラム中で使われることを想定してはいない。デバッグが終わって完成したプログラムでは、エラーは返り値で判定、処理されるべきである。

3.2　エラー・コード

GSL の関数が返すエラー・コードはファイル 'gsl_errno.h' で定義されている。 エラー・コードはすべて先頭に GSL_ がつけられ、0 以外の整数値に展開される。1024 よりも大きな値はアプリケーション側で利用することを想定し、GSL では使っていない。エラー・コードの名前は多くの場合、標準の C ライブラリにある対応するエラー・コードの名前から作られている。以下によく使われるエラー・コードを示す。

`int` (GSL_EDOM) [Macro]

領域エラー (domain error)。数学関数に渡された引数の値が、定義されている領域内にない場合に返す (C ライブラリの EDOM に対応)。

`int` (GSL_ERANGE) [Macro]

範囲エラー (range error)。数学関数の返す計算値が、オーバーフローやアンダーフローで適切に表現できる範囲にない場合に返す (C ライブラリの ERANGE に対応)。

`int` (GSL_ENOMEM) [Macro]

メモリ不足 (No memory available)。システムが要求されただけの仮想メモリ領域を確保できない時に返す (C ライブラリの ENOMEME に対応)。GSL ルーチンが malloc でのメモリ確保に失敗したときに返す。

`int` (GSL_EINVAL) [Macro]

不正引数 (invalid argument)。ライブラリの関数に渡す引数が正しくないような、様々な状況で返される (C ライブラリの EINVAL に対応)。

エラー・コードは、関数 gsl_strerror を使うことで、対応するエラーメッセージに変換できる。

`const char * gsl_strerror` (*const int gsl_errno*) [Function]

引数 *gsl_errno* で指定されたエラー・コードに対応するエラーメッセージ文字列へのポインタを返す。以下のコードでは、

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

status がたとえば GSL_ERANGE だった場合には、error: output range error のようなメッセージを表示する。

3.3 エラー・ハンドラー

GSL のエラー・ハンドラーのデフォルトの動作は、簡潔なエラーメッセージを表示して `abort()` を呼ぶことである。GSL のルーチンでエラーが発生してこの動作が行われた場合、これを実行中のプログラムは終了してコア・ダンプを生成する。これは、GSL ルーチンの返り値を確認しないようなプログラムを想定した、フェイル・セーフ (fail safe) な仕様である (GSL 開発チームとしては、そういったプログラムは書かないように勧める)。

デフォルトのエラー・ハンドラーの動作を無効にした場合は、ルーチンを呼ぶプログラム側で GSL ルーチンの返り値を確認してそれに対応した動作をせねばならない。独自に作成したハンドラーを使うことで、デフォルトの動作を変えることができる。たとえば発生したエラーをすべてログに記録したり、アンダーフローのようなエラーを無視したり、デバッガーを起動してエラーが発生した実行中のプロセスにアタッチする、といったエラー・ハンドラーを書くことができる。

GSL のエラー・ハンドラーの型はすべて `gsl_error_handler_t` であり、'`gsl_errno.h`' で宣言されている。

`gsl_error_handler_t` [Data Type]

これが GSL のエラー・ハンドラー関連の機能のために用意されている型である。エラー・ハンドラーに渡される引数は 4 つで、エラーの原因 (文字列)、エラーが起きた場所のソースファイル名 (文字列)、ファイル中での行番号 (整数)、エラー・コード (整数) である。ソースファイル名と行番号はコンパイル時にプリプロセッサの `__FILE__` と `__LINE__` ディレクティブ (directive、事前定義マクロ) で決まる。エラー・ハンドラーの返り値は `void` である。したがってエラー・ハンドラー関数は以下のように定義される。

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

独自に作成したエラー・ハンドラーを使うには、`gsl_set_error_handler` 関数を使う。これは '`gsl_errno.h`' で宣言されている。

`gsl_error_handler_t * gsl_set_error_handler` (*gsl_error_handler_t * new_handler*) [Function]

この関数で GSL ライブラリのエラー・ハンドラーを新たに `new_handler` に設定する。この前に設定されていたハンドラーが関数の返り値となる (したがって後で元に戻すことができる)。独自に作成したエラー・ハンドラーへのポインタは大域的な静的変数に格納されているため、一つのプログラム内で同時に使えるエラー・ハンドラーは一つだけである。したがってプログラムがマルチ・スレッドの場合、どのスレッドでも使う汎用のエラー・ハンドラーをマスタースレッドで設定する時以外は、この関数を呼ぶべきではない。以下にエラーハンドラーを設定、復元する例を示す。

```
/* 元のエラーハンドラーを保存して、新しいハンドラーを設定する */
```

```
old_handler = gsl_set_error_handler(&my_handler);

/* 新しいハンドラーを使う処理 */
.....

/* 元のハンドラーに戻す */
gsl_set_error_handler(old_handler);
```

デフォルトの動作 (エラー発生時に abort を呼ぶ) にするには、エラー・ハンドラーとして NULL を設定する。

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

gsl_error_handler_t * gsl_set_error_handler_off () [Function]

何もしないエラー・ハンドラーを設定することで、エラーのハンドリングを無効にする。これにより、エラーが発生してもプログラムはそのまま実行を続けるようになるので、GSL ルーチンの返り値をすべて呼んだ側で確認するべきであり、最終的に完成したプログラムでは、そうするべきである。返り値は以前に設定されていたハンドラーである (したがって後で元に戻すことができる)。

GSL を使うなにか特定のアプリケーションにおいて、アプリケーションのソースコードに手を加えずにエラー・ハンドラーの挙動を変えたいときは、ファイル 'gsl_errno.h' 中にある GSL_ERROR マクロの定義を変更して GSL を再コンパイルすることもできる (次節参照)。

3.4 独自の関数で GSL のエラー・レポートを利用するには

GSL の関数を利用するプログラム中で、別に書いた数値計算を行う関数を呼んでいる場合でも GSL と同じエラー検知機構ができると、便利なこともあるだろう。

これを利用するには、そのエラーについて説明する文字列を引数として関数 gsl_error を呼び、続けて gsl_errno.h で定義されているエラー・コードのどれか、またはたとえば NaN のような特殊な値を返す。これを行うためのマクロが二つ、'gsl_errno.h' 内に定義されている。

GSL_ERROR (*reason*, *gsl_errno*) [Macro]

このマクロは、GSL の様式に従って gsl_errno にある値を返す。これは以下のように展開される。

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

このマクロ定義は 'gsl_errno.h' にあり、実際には構文エラーが出ないように do ... while (0) ブロックで上のコードが囲まれている。

以下にこのマクロの使用例を、要求精度が得られなかった場合にルーチンが返すエラーで示す。このエラーを示すためにはルーチンは GSL_ETOL を返す必要がある。

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

GSL_ERROR_VAL (*reason*, *gsl_errno*, *value*) [Macro]

このマクロは、エラー・コードの代わりに利用者が定義した値 value を返すこと以外は、GSL_ERROR と同じである。実数値を返すような数値計算関数で使うことができる。

以下の例では、GSL_ERROR_VAL マクロを使って、特異点で発生した NaN を返す方法を示す。

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

3.5 プログラム例

以下に、エラーが発生しうる関数の返り値を確認するプログラムの例を示す。

```
#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, 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);
    }
    ...
```

関数 gsl_fft_complex_radix2 は、長さとして整数でかつ 2 のべき乗の値しか受け付けない。変数 n が 2 のべき乗でなかった場合、呼ばれたライブラリのルーチンは GSL_EINVAL を返し、引

数が不正であったことを示す。関数 gsl_set_error_handler_off を呼び出すことでデフォルトのエラー・ハンドラーがプログラムを停止させてしまうことを防ぐ。else 節は他の種類のエラーを処理するためのものである。

第4章　数学の関数

この章では GSL で実装している数学の基本的な関数について説明する。システムによっては、この中のいくつかがすでに用意されているかもしれないが、ない場合には GSL をインストールすることで利用できるようになる。

この章で説明する関数はヘッダイファイル 'gsl_math.h' で宣言されている。

4.1　定数

BSD 標準の数学ライブラリ (BSD 系の OS の libm.a と math.h) に用意されている定数を GSL でも定義している。以下がそのリストである。

M_E	自然対数の底、e
M_LOG2E	e の底が 2 の対数、$\log_2(e)$
M_LOG10E	e の底が 10 の対数、$\log_{10}(e)$
M_SQRT2	2 の平方根、$\sqrt{2}$
M_SQRT1_2	二分の一の平方根、$\sqrt{1/2}$
M_SQRT3	3 の平方根、$\sqrt{3}$
M_PI	円周率、π
M_PI_2	円周率の二分の一、$\pi/2$
M_PI_4	円周率の四分の一、$\pi/4$
M_SQRTPI	円周率の平方根、$\sqrt{\pi}$
M_2_SQRTPI	2 を円周率の平方根で除した値、$2/\sqrt{\pi}$
M_1_PI	円周率の逆数、$1/\pi$
M_2_PI	円周率の逆数の二倍、$2/\pi$
M_LN10	10 の自然対数、$\ln(10)$
M_LN2	2 の自然対数、$\ln(2)$
M_LNPI	円周率の自然対数、$\ln(\pi)$
M_EULER	オイラー定数、γ

4.2　無限大と非数値

GSL_POSINF　[Macro]

正の無限大 $+\infty$ の IEEE 表現。この値は +1.0/0.0 という式を評価することで得られる。

GSL_NEGINF [Macro]

負の無限大 $-\infty$ の IEEE 表現。この値は `-1.0/0.0` という式を評価することで得られる。

GSL_NAN [Macro]

非数値 (Not-a-number) NaN の IEEE 表現。この値は `0.0/0.0` という式を評価することで得られる。

int gsl_isnan (*const double* x) [Function]

x が非数値 (NaN) であれば 1 を返す。そうでなければ 0 を返す。

int gsl_isinf (*const double* x) [Function]

x が正の無限大であれば +1、負の無限大であれば −1、どちらでもなければ 0 を返す。

int gsl_finite (*const double* x) [Function]

x が実数であれば 1、無限大か非数値であれば 0 を返す。

4.3　初等関数

以下に説明するルーチンは、BSD の数学ライブラリ (BSD 系 OS の `lib.a` と `math.h`) 互換として実装したものである。ネイティブ版の関数が使えないときは、その代わりにこれらの関数を使うことができる。`autoconf` を使える環境では、プログラムをコンパイルするときに自動的にどちらの関数を使うか決めることができる (2.7 節「移植性確保のための関数」参照)。

double gsl_log1p (*const double* x) [Function]

x が小さな値の時でも精度の高い方法で $\log(1+x)$ の値を計算する。BSD 数学ライブラリの `log1p(x)` と互換性がある。

double gsl_expm1 (*const double* x) [Function]

x が小さな値の時でも精度の高い方法で $\exp(x)-1$ の値を計算する。BSD 数学ライブラリの `expm1(x)` と同等である。

double gsl_hypot (*const double* x, *const double* y) [Function]

オーバーフローが発生しないように $\sqrt{x^2+y^2}$ の値を計算する。BSD 数学ライブラリの `hypot(x,y)` と同等である。

double gsl_hypot3 (*const double* x, *const double* y, *const double* z) [Function]

オーバーフローできるだけ避けて $\sqrt{x^2+y^2+z^2}$ の値を計算する。

double gsl_acosh (*const double* x) [Function]

arccosh(x) の値を計算する。BSD 数学ライブラリの `acosh(x)` と同等である。

`double gsl_asinh (const double x)` [Function]

arcsinh(x) の値を計算する。BSD 数学ライブラリの `asinh(x)` と同等である。

`double gsl_atanh (const double x)` [Function]

arctanh(x) の値を計算する。BSD 数学ライブラリの `atanh(x)` と同等である。

`double gsl_ldexp (double x, int e)` [Function]

$x * 2^e$ の値を計算する。標準数学ライブラリ (ANSI C の standard math library、libm) の `ldexp(x)` と同等である。

`double gsl_frexp (double x, int * e)` [Function]

x を $x = f * 2^e$ となる仮数部 f と指数部 e に分ける。仮数部は $0.5 \leq f < 1$ となるように正規化される。返り値は f で、e は引数に入れて返される。x が 0 のときは f も e も 0 になる。標準数学ライブラリ (libm) の `frexp(x, e)` と同等である。

4.4 小さな整数でのべき乗

標準 C ライブラリには、(小さな) 整数でのべき乗を計算する関数がないという欠点がある。GSL はその欠点を埋める簡単な関数を用意している。計算速度を向上するため、これらの関数ではオーバーフローやアンダーフローの確認をしていない。

`double gsl_pow_int (double x, int n)` [Function]

整数 n によるべき乗 x^n を計算する。計算は効率よく、たとえば x^8 は $((x^2)^2)^2$ の形で、3 回の乗算だけですむように行われる。関数 `gsl_sf_pow_int_e` は同じ処理を行うが、内部で計算誤差を見積もって、それを返す。

`double gsl_pow_2 (const double x)` [Function]
`double gsl_pow_3 (const double x)` [Function]
`double gsl_pow_4 (const double x)` [Function]
`double gsl_pow_5 (const double x)` [Function]
`double gsl_pow_6 (const double x)` [Function]
`double gsl_pow_7 (const double x)` [Function]
`double gsl_pow_8 (const double x)` [Function]
`double gsl_pow_9 (const double x)` [Function]

x^2 や x^3 などの、実数の小さな整数乗を高速に計算する。これらの関数は `HAVE_INLINE` が定義されているときはインライン展開されるため、同じ演算を明示的に乗算の形で書いた場合と同等の速度を出すことができる。

```
#include <gsl/gsl_math.h>
double y = gsl_pow_4 (3.141) /* 3.141**4 を計算する*/
```

4.5　符号の確認

GSL_SIGN (*x*) [Macro]

これは x の符号を返すマクロで、`((x) >= 0 ? 1 : -1)` と定義されている。この定義では、x の IEEE 符号ビットがどうであっても、0 の符号は正になる。

4.6　偶数、奇数の確認

GSL_IS_ODD (*n*) [Macro]

n が奇数の時 1、偶数の時 0 を返す。引数 n は整数型でなければならない。

GSL_IS_EVEN (*n*) [Macro]

このマクロは `GSL_IS_ODD(n)` と逆で、n が偶数の時 1、奇数の時 0 を返す。引数 n は整数型でなければならない。

4.7　最大値、最小値関数

GSL_MAX (a, *b*) [Macro]

a と b のうち大きな方を返す。このマクロは `((a) > (b) ? (a) : (b))` と定義されている。

GSL_MIN (a, *b*) [Macro]

a と b のうち小さな方を返す。このマクロは `((a) < (b) ? (a) : (b))` と定義されている。

extern inline double GSL_MAX_DBL (*double* a, *double* *b*) [Function]

インライン関数を使って倍精度実数 a と b のうち大きな方を返す。関数なので、マクロと違ってコンパイル時の型チェックが有効であり、安全である。インライン関数が使えないプラットフォームでは、この関数は自動的に `GSL_MAX` に置き換わる。

extern inline double GSL_MIN_DBL (*double* a, *double* *b*) [Function]

インライン関数を使って倍精度実数 a と b のうち小さな方を返す。関数なので、マクロと違ってコンパイル時の型チェックが有効であり、安全である。インライン関数が使えないプラットフォームでは、この関数は自動的に `GSL_MIN` に置き換わる。

extern inline double GSL_MAX_INT (*int* a, *int* *b*) [Function]
extern inline double GSL_MIN_INT (*int* a, *int* *b*) [Function]

整数 a と b のうち大きな方または小さな方を返すインライン関数。インライン関数が使えないプラットフォームでは、この関数は自動的に GSL_MAX または GSL_MIN に置き換わる。

extern inline long double GSL_MAX_LDBL (*long double* a, *long double* b)[Function]
extern inline long double GSL_MIN_LDBL (*long double* a, *long double* b)[Function]

インライン関数を使って四倍精度実数 a と b のうち大きな方または小さな方を返す。インライン関数が使えないプラットフォームでは、この関数は自動的に GSL_MAX または GSL_MIN に置き換わる。

4.8 実数値の近似的な比較

二つの実数値を比較するときに、丸め誤差や切り捨て誤差による影響を避けるために、厳密にではなく、近似的に比較するとよいことがある。このライブラリでは、ドナルド・クヌース (Donald Ervin Knuth) の本 *Seminumerical Algorithms* (3rd edition) の 4.2.2 節にある近似的な比較法を実装した関数を用意している。

int gsl_fcmp (*double* x, *double* y, *double* epsilon) [Function]

x と y が、相対精度 *epsilon* で比較したときに等しいと見なせるかどうかを判定する。判断基準は二数の差が 2δ より大きいかどうかである。ここで $\delta = 2^k\epsilon$ であり、k は frexp() で計算される x と y の (二進数の) 指数部のうち大きな方である。

x と y の差がこの値よりも小さな時、この二つの数値は実質的には同じ値であると判定し、0 を返す。そうでなく $x < y$ の場合には -1、$x > y$ の場合には $+1$ を返す。

x と y の差は相対精度で比較されるため、そのどちらかの値が 0 に近い場合にはこの関数は適さない。

GSL での実装はベルディング (Theodore C. Belding) による fcmp パッケージを元にしている。fcmp については http://sourceforge.net/projects/fcmp/ を参照のこと。

第5章　複素数

この章は複素数 (complex number) を扱う関数について説明する。複素平面 (ガウス平面、Gaussian plane) 上でこれらの関数を評価するとき、関数内部ではできるだけアンダーフローやオーバーフローが発生しないように工夫して実装されている。

多価関数 (multiple-valued function) については、アブラモウィッツ＆ステグン (Abramowitz & Stegun, *Handbook of Mathematical Functions*, 1964) にしたがって分岐 (branch cut) を考え、GNU Calc と同じ主値 (principal value) を返す。つまり *Common Lisp, The Language* (Second Edition) [1]および HP-28/48 シリーズの計算機とも同じである。

複素数そのものの型宣言はヘッダファイル 'gsl_complex.h' に、複素数をあつかう初等関数や代数演算を行う関数は 'gsl_complex_math.h' で宣言されている。

5.1　複素数

複素数は gsl_complex 型で表現される。この型の内部表現はプラットフォームによって異なるため、内部を直接参照、操作するべきではない。以下に説明する関数やマクロを使うことで、移植性の高い操作を行うことができる。

参考までに示すと、gsl_complex はデフォルトでは以下の構造体で定義される。

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

実部と虚部は二つの要素を持つ配列中で連続する要素に保持される。それにより実部と虚部、つまり dat[0] および dat[1] の間のパディングが行われないため、(構造体のメモリ上のサイズをバイト境界に合わせるためのダミー領域の挿入が、dat[0] と dat[1] の間には行われない)、複素数型の配列を packed にした場合も正しく配置される。

gsl_complex gsl_complex_rect (*double* x, *double* y)　[Function]

引数を直交座標系での座標値 (*x*, *y*) として、複素数 $z = x + iy$ を返す。HAVE_INLINE が定義されているときは、インライン展開される。

gsl_complex gsl_complex_polar (*double* r, *double* theta)　[Function]

引数を極座標での座標値 (*r*, *theta*) として、複素数 $z = r\exp(i\theta) = r(\cos(\theta) + i\sin(\theta))$ を返す。

[1] First edition では異なる定義になっている。

GSL_REAL (*z*) [Macro]
GSL_IMAG (*z*) [Macro]

複素数 z の実部あるいは虚部を返すマクロである。

GSL_SET_COMPLEX (*zp*, *x*, *y*) [Macro]

ポインタ *zp* が指す複素数インスタンスに直交座標値 (x, *y*) から得られる複素数を代入するマクロである。たとえば、

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

とすると z は $3+4i$ になる。

GSL_SET_REAL (*zp*, *x*) [Macro]
GSL_SET_IMAG (*zp*, *y*) [Macro]

これらはマクロであり、ポインタ *zp* が指す複素数インスタンスの実部あるいは虚部に値を代入することができる。

5.2　複素数の属性

double gsl_complex_arg (*gsl_complex* *z*) [Function]

複素数 z の偏角 $\arg(z)$ を返す。ただし $-\pi < \arg(z) \leq \pi$ である。

double gsl_complex_abs (*gsl_complex* *z*) [Function]

複素数 z の大きさ（絶対値）$|z|$ を返す。

double gsl_complex_abs2 (*gsl_complex* *z*) [Function]

複素数 z の大きさの二乗 $|z|^2$ を返す。

double gsl_complex_logabs (*gsl_complex* *z*) [Function]

複素数 z の大きさ $|z|$ の自然対数 $\log|z|$ を返す。$|z|$ が 1 に近い値の時にこれをそのまま `log(gsl_complex_abs(z))` で計算すると精度が悪くなるが、この関数ではそれよりも精密な値を返すことができる。

5.3　複素代数的演算子

gsl_complex gsl_complex_add (*gsl_complex* a, *gsl_complex* *b*) [Function]

複素数 a と b の和、$z = a + b$ を返す。

gsl_complex gsl_complex_sub (*gsl_complex* a, *gsl_complex* *b*) [Function]

複素数 a と b の差、$z = a - b$ を返す。

`gsl_complex gsl_complex_mul` (*gsl_complex* `a`, *gsl_complex* `b`) [Function]

複素数 a と b の積、$z = ab$ を返す。

`gsl_complex gsl_complex_div` (*gsl_complex* `a`, *gsl_complex* `b`) [Function]

複素数 a を b で割った商、$z = a/b$ を返す。

`gsl_complex gsl_complex_add_real` (*gsl_complex* `a`, *double* `x`) [Function]

複素数 a と実数 x の和、$z = a + x$ を返す。

`gsl_complex gsl_complex_sub_real` (*gsl_complex* `a`, *double* `x`) [Function]

複素数 a と実数 x の差、$z = a - x$ を返す。

`gsl_complex gsl_complex_mul_real` (*gsl_complex* `a`, *double* `x`) [Function]

複素数 a と実数 x の積、$z = ax$ を返す。

`gsl_complex gsl_complex_div_real` (*gsl_complex* `a`, *double* `x`) [Function]

複素数 a を実数 x で割った商、$z = a/x$ を返す。

`gsl_complex gsl_complex_add_imag` (*gsl_complex* `a`, *double* `y`) [Function]

複素数 a と虚数 iy の和、$z = a + iy$ を返す。

`gsl_complex gsl_complex_sub_imag` (*gsl_complex* `a`, *double* `y`) [Function]

複素数 a と虚数 iy の差、$z = a - iy$ を返す。

`gsl_complex gsl_complex_mul_imag` (*gsl_complex* `a`, *double* `y`) [Function]

複素数 a と虚数 iy の積、$z = a * (iy)$ を返す。

`gsl_complex gsl_complex_div_imag` (*gsl_complex* `a`, *double* `y`) [Function]

複素数 a を虚数 iy で割った商、$z = a/(iy)$ を返す。

`gsl_complex gsl_complex_conjugate` (*gsl_complex* `z`) [Function]

複素数 z の共役複素数 $z^* = x - iy$ を返す。

`gsl_complex gsl_complex_inverse` (*gsl_complex* `z`) [Function]

複素数 z の逆数 $1/z = (x - iy)/(x^2 + y^2)$ を返す。

`gsl_complex gsl_complex_negative` (*gsl_complex* `z`) [Function]

複素数 z の符号を反転した複素数 $-z = (-x) + i(-y)$ を返す。

5.4　初等複素関数

`gsl_complex gsl_complex_sqrt` (*gsl_complex* `z`) [Function]

複素数 z の平方根 $\sqrt{z}$ を返す。分岐を考え、実部が負の結果は返さない。結果は必ず複素平面の右半分の領域にある。

`gsl_complex gsl_complex_sqrt_real` (*double* `x`) [Function]

実数 x の平方根を複素数で返す。x は負でもよい

`gsl_complex gsl_complex_pow` (*gsl_complex* `z`, *gsl_complex* `a`) [Function]

複素数 z の複素数 a 乗、z^a を返す。これは複素対数と複素指数を使って $\exp(\log(z) * a)$ として計算される。

`gsl_complex gsl_complex_pow_real` (*gsl_complex* `z`, *double* `x`) [Function]

複素数 z の実数 x 乗、z^x を返す。

`gsl_complex gsl_complex_exp` (*gsl_complex* `z`) [Function]

複素数 z の指数 $\exp(z)$ を返す。

`gsl_complex gsl_complex_log` (*gsl_complex* `z`) [Function]

複素数 z の自然対数 $\log(z)$ を返す。実部が負の解は返さない。

`gsl_complex gsl_complex_log10` (*gsl_complex* `z`) [Function]

複素数 z の常用対数（底が 10）、$\log_{10}(z)$ を返す。

`gsl_complex gsl_complex_log_b` (*gsl_complex* `z`, *double* `a`) [Function]

複素数 z の b を底とする対数 $\log_b(z)$ を返す。この値は $\log(z)/\log(b)$ として計算される (この log の底は e)。

5.5　複素三角関数

`gsl_complex gsl_complex_sin` (*gsl_complex* `z`) [Function]

複素数 z の複素正弦関数値 $\sin(z) = (\exp(iz) - \exp(-iz))/(2i)$ を返す。

`gsl_complex gsl_complex_cos` (*gsl_complex* `z`) [Function]

複素数 z の複素余弦関数値 $\cos(z) = (\exp(iz) + \exp(-iz))/2$ を返す。

`gsl_complex gsl_complex_tan` (*gsl_complex* `z`) [Function]

複素数 z の複素正接関数値 $\tan(z) = \sin(z)/\cos(z)$ を返す。

`gsl_complex gsl_complex_sec` (*gsl_complex* **z**) [Function]

複素数 z の複素正割関数 (セカント) 値 $\sec(z) = 1/\cos(z)$ を返す。

`gsl_complex gsl_complex_csc` (*gsl_complex* **z**) [Function]

複素数 z の複素余割関数 (コセカント) 値 $\csc(z) = 1/\sin(z)$ を返す。

`gsl_complex gsl_complex_cot` (*gsl_complex* **z**) [Function]

複素数 z の複素余接関数 (コタンジェント) 値 $\cot(z) = 1/\tan(z)$ を返す。

5.6 逆複素三角関数

`gsl_complex gsl_complex_arcsin` (*gsl_complex* **z**) [Function]

複素数 z の複素逆正弦関数値 $\arcsin(z)$ を返す。分岐を考え、実部が -1 よりも小さい、または実部が 1 よりも大きな結果は返さない。

`gsl_complex gsl_complex_arcsin_real` (*double* **z**) [Function]

実数 z の複素逆正弦関数値 $\arcsin(z)$ を返す。z が -1 と 1 の間の値の時、$[-\pi/2, \pi/2]$ の範囲の実数を返す。z が -1 よりも小さいときは、返り値の実部は $-\pi/2$ に、虚部は正の値になる。z が 1 よりも大きいときは、返り値の実部は $\pi/2$ に、虚部は負の値になる。

`gsl_complex gsl_complex_arccos` (*gsl_complex* **z**) [Function]

複素数 z の複素逆余弦関数値 $\arccos(z)$ を返す。分岐を考え、実部が -1 よりも小さい、または実部が 1 よりも大きな結果は、返さない。

`gsl_complex gsl_complex_arccos_real` (*double* **z**) [Function]

実数 z の複素逆余弦関数値 $\arccos(z)$ を返す。z が -1 と 1 の間の値の時、$[0, \pi]$ の範囲の実数を返す。z が -1 よりも小さいときは、返り値の実部は π、虚部は負の値になる。z が 1 よりも大きいときは、正の虚数（実部は 0）になる。

`gsl_complex gsl_complex_arctan` (*gsl_complex* **z**) [Function]

複素数 z の複素逆正接関数値 $\arctan(z)$ を返す。分岐を考え、虚部の値が $-i$ より小さい、および i より大きな値は返さない。

`gsl_complex gsl_complex_arcsec` (*gsl_complex* **z**) [Function]

複素数 z の複素逆正割関数値 $\text{arcsec}(z) = \arccos(1/z)$ を返す。

`gsl_complex gsl_complex_arcsec_real` (*double* **z**) [Function]

実数 z の複素逆余割関数値 $\text{arcsec}(z) = \arccos(1/z)$ を返す。

`gsl_complex gsl_complex_arccsc (`*gsl_complex* `z)` [Function]

複素数 z の複素逆余割関数値 $\mathrm{arccsc}(z) = \arcsin(1/z)$ を返す。

`gsl_complex gsl_complex_arccsc_real (`*double* `z)` [Function]

実数 z の複素逆余割関数値 $\mathrm{arccsc}(z) = \arcsin(1/z)$ を返す。

`gsl_complex gsl_complex_arccot (`*gsl_complex* `z)` [Function]

複素数 z の複素逆余接関数値 $\mathrm{arccot}(z) = \arctan(1/z)$ を返す。

5.7　複素双曲線関数

`gsl_complex gsl_complex_sinh (`*gsl_complex* `z)` [Function]

複素数 z の複素双曲線正弦関数値 $\sinh(z) = (\exp(z) - \exp(-z))/2$ を返す。

`gsl_complex gsl_complex_cosh (`*gsl_complex* `z)` [Function]

複素数 z の複素双曲線余弦関数値 $\cosh(z) = (\exp(z) + \exp(-z))/2$ を返す。

`gsl_complex gsl_complex_tanh (`*gsl_complex* `z)` [Function]

複素数 z の複素双曲線正接関数値 $\tanh(z) = \sinh(z)/\cosh(z)$ を返す。

`gsl_complex gsl_complex_sech (`*gsl_complex* `z)` [Function]

複素数 z の複素双曲線正割関数値 $\mathrm{sech}(z) = 1/\cosh(z)$ を返す。

`gsl_complex gsl_complex_csch (`*gsl_complex* `z)` [Function]

複素数 z の複素双曲線余割関数値 $\mathrm{csch}(z) = 1/\sinh(z)$ を返す。

`gsl_complex gsl_complex_coth (`*gsl_complex* `z)` [Function]

複素数 z の複素双曲線余接関数値 $\coth(z) = 1/\tanh(z)$ を返す。

5.8　逆複素双曲線関数

`gsl_complex gsl_complex_arcsinh (`*gsl_complex* `z)` [Function]

複素数 z の逆複素双曲線正弦関数値 $\mathrm{arcsinh}(z)$ を返す。分岐を考え、虚部の値が $-i$ より小さい、および i より大きな値は返さない。

`gsl_complex gsl_complex_arccosh (`*gsl_complex* `z)` [Function]

複素数 z の逆複素双曲線余弦関数値 $\mathrm{arccosh}(z)$ を返す。分岐を考え、実部が 1 より小さな値は返さない。その場合、アブラモウィッツ＆ステグンの 4.6.21 式 $\mathrm{arccosh}(z) = \log(z - \sqrt{z^2 - 1})$ により負の平方根を計算する。

`gsl_complex gsl_complex_arccosh_real` (*double* **z**) [Function]

実数 z の逆複素双曲線余弦関数値 $\arccosh(z)$ を返す。

`gsl_complex gsl_complex_arctanh` (*gsl_complex* **z**) [Function]

複素数 z の逆複素双曲線正接関数値 $\operatorname{arctanh}(z)$ を返す。分岐を考え、実部が -1 より小さい、および 1 より大きな値は返さない。

`gsl_complex gsl_complex_arctanh_real` (*double* **z**) [Function]

実数 z の逆複素双曲線正接関数値 $\operatorname{arctanh}(z)$ を返す。

`gsl_complex gsl_complex_arcsech` (*gsl_complex* **z**) [Function]

複素数 z の逆複素双曲線正割関数値 $\operatorname{arcsech}(z) = \arccosh(1/z)$ を返す。

`gsl_complex gsl_complex_arccsch` (*gsl_complex* **z**) [Function]

複素数 z の逆複素双曲線余割関数値 $\operatorname{arccsch}(z) = \arcsin(1/z)$ を返す。

`gsl_complex gsl_complex_arccoth` (*gsl_complex* **z**) [Function]

複素数 z の逆複素双曲線余接関数値 $\operatorname{arccoth}(z) = \operatorname{arctanh}(1/z)$ を返す。

5.9 参考文献

基本的な関数と三角関数の実装は、以下の論文によっている。

- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, "Implementing Complex Elementary Functions Using Exception Handling", *ACM Transactions on Mathematical Software*, **20**(2), pp 215–244, Corrigenda, p553 (1994).
- 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*, **23**(3), pp 299–335 (1997).

分岐の方法とその説明は以下の本にある。

- Milton Abramowitz and Irene A. 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, National Bureau Of Standards (1964).
- Dave Gillespie, *Calc Manual*, Free Software Foundation, ISBN 1-882114-18-3 (1992).

第6章　多項式

この章では多項式 (polynomial) の評価 (evaluation)、求根 (solving) に関する関数について説明する。二次 (quadratic) および三次 (cubic) の多項式については、解析的 (analytic) に実数あるいは複素数の解 (root) を得るルーチンが用意されている。それ以外の一般の任意の次数については、実数係数の場合に繰り返し計算で解を求めるルーチンが用意されている。それらの関数はヘッダファイル 'gsl_poly.h' で宣言されている。

6.1　多項式の評価

ここに挙げる関数は多項式 $c[0] + c[1]x + c[2]x^2 + ... + c[len - 1]x^{len-1}$ の値を、数値的安定性の向上を図るためホーナー法 (Horner's method) を使って求める。HAVE_INLINE が定義されているときはインライン展開される。

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

実数変数 x の実数係数の多項式の値を計算する。

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

複素変数 z の実数係数の多項式の値を計算する。

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

複素変数 z の複素数係数の多項式の値を計算する。

6.2　多項式の差分商表現

ここではニュートンの差分商表現 (Newton's divided-difference representation) を扱う関数について説明する。差分商については、アブラモウィッツ＆ステグンの 25.1.4 節、25.2.26 節に説明がある。

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

長さ *size* の配列 *xa* と *ya* に保持された点 (*xa*, *ya*) による補間多項式の差分商表現を計算する。得られた (*xa*, *ya*) の差分商表現は、長さ *size* の配列 *dd* に入れて返される。

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

長さ *size* の配列 *dd* と *xa* に差分商表現で保持されている多項式の、点 *x* における値を計算して返す。`HAVE_INLINE` が定義されているときはインライン展開される。

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

多項式の差分商表現をテイラー展開 (Tayler expansion) に変換する。差分商表現は長さ *size* の配列 *dd* と *xa* に入れて渡す。多項式を点 *xp* で展開して得られたテイラー係数は、やはり長さ *size* の配列 *c* に入れて返される。作業領域として長さ *size* の配列 *w* を渡さなければならない。

6.3　二次方程式

`int gsl_poly_solve_quadratic` (*double* `a`, *double* `b`, *double* `c`, *double* * `x0`, *double* * `x1`) [Function]

以下の形式の二次方程式 (quadratic equation) の実根 (real root) を求める。

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

返り値は実根の個数 (0 か 1 か 2) で、根の値は *x0* と *x1* に入れられる。実根がない場合は *x0* と *x1* の値は変更されない。実根が一つだけ得られる（たとえば $a = 0$）の場合、根は *x0* に入れられる。二つの実根が得られた場合は *x0* と *x1* に昇順に入れられ、重根 (coincident roots, multiple roots) の場合も同様に扱われる。たとえば $(x - 1)^2 = 0$ の場合は根は 2 個あるが、その値は全く同じである。

根の個数は判別式 $b^2 - 4ac$ の符号によって決まる。しかし倍精度で計算しても丸め誤差や桁落ちでその符号が変わることがあり、また多項式の係数が厳密でないときも問題になる。これらの誤差によって根の個数が間違って求められることがある。しかし多項式の係数が小さな整数の場合は、判別式の値は厳密に計算される。

`int gsl_poly_complex_solve_quadratic` (*double* `a`, *double* `b`, *double* `c`, *gsl_complex* * `z0`, *gsl_complex* * `z1`) [Function]

以下の形式の二次方程式の複素根 (complex root) を計算する。

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

返り値は複素根の個数（1 または 2）であり、求められた根は *z0* と *z1* に入れて返される。根はまず実部、次に虚部の値で比較された昇順に引数に入れられる。根が一つしかない場合（たとえば $a = 0$ のときなど）はその根は *z0* に入れられる。

6.4 三次方程式

`int gsl_poly_solve_cubic` (*double* `a`, *double* `b`, *double* `c`, *double* * `x0`, *double* * `x1`, *double* * `x2`) [Function]

以下の形式の三次方程式 (cubic equation) の実数解を計算する。

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

三乗の項の係数は 1 とする。返り値は実根の個数（1 か 3）であり、それらは *x0*、*x1*、*x2* に入れて返される。得られた実根の個数が一個の場合は *x0* だけが書き換えられる。実根が三個得られた場合は *x0*、*x1*、*x2* に昇順に入れられ、重根の場合も同じように扱われる。たとえば $(x-1)^3 = 0$ の場合は三つの引数には全く同じ値が入れられる。二次方程式の場合と同様に、数値計算の精度が限られているために、実数軸上の同じ値あるいは近い値の根が複素平面にあるかのように計算されてしまい、実根の個数が間違って計算されることもある。

`int gsl_poly_complex_solve_cubic` (*double* `a`, *double* `b`, *double* `c`, *gsl_complex* * `z0`, *gsl_complex* * `z1`, *gsl_complex* * `z2`) [Function]

以下の形式の三次方程式の複素根を計算する。

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

返り値は求められた複素根の個数（常に 3）であり、根は *z0*、*z1*、*z2* に入れて返される。まず実部の、次に虚部の昇順に並べたときの順番で入れられる。

6.5 一般の多項式の方程式

二次、三次、四次方程式 (quartic equation) のような特殊な場合を除くと、多項式の解は解析的には得られない。ここでは高次の多項式 (higher order polynomial) の解を近似的に求めるために繰り返し計算を使う方法 (iterative method) について説明する。

`gsl_poly_complex_workspace * gsl_poly_complex_workspace_alloc` (*size_t* `n`) [Function]

n 個の係数を持つ多項式を `gsl_poly_complex_solve` を使って解くための作業領域として、`gsl_poly_complex_workspace` 構造体のインスタンスを確保する。返り値は確保した `gsl_poly_complex_workspace` へのポインタである。エラーが生じたときは null ポインタを返す。

`void gsl_poly_complex_workspace_free` (*gsl_poly_complex_workspace* * `w`) [Function]

作業領域 *w* のメモリを解放する。

`int gsl_poly_complex_solve (`*const double* * `a`, *size_t* `n`, *gsl_poly_complex_workspace* * `w`, *gsl_complex_packed_ptr* `z``)` [Function]

多項式 $P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$ の解を、随伴行列 (companion matrix) の平衡化 (balancing) を使った QR 分解 (balanced-QR reduction) で求める。引数 n で係数を格納している配列の長さを指定する。最高次の係数は 0 であってはならない。適切な大きさの作業領域 w を与えなければならない。長さ $2(n-1)$ の密配置 (packed) な複素数配列 z に $n-1$ 個の解を入れて返す。配列内で実部と虚部が交互に入れられることになる。

すべての解が見つかった場合は関数の返り値は `GSL_SUCCESS` である。QR 分解が収束しなかった場合は、エラーコード `GSL_EFAILED` でエラー・ハンドラーが呼ばれる。計算精度が限られているため、重複度 (multiplicity) の高い根がある方程式では、それが値の近い複数の単根 (simple root) が集まっているかのように計算されてしまうことがある。高次の多項式の根を求めるには、根の重複度を考慮したアルゴリズムを使う必要がある (Zhongguan Zeng, Algorithm 835: MultRoot – A MATLAB package for Computing Polynomial Roots and Multiplicities, *ACM Transactions on Mathematical Software*, **30**(2), pp 218–236 (2004) 参照)。

6.6　例

一般的な多項式の求根法の例として $P(x) = x^5 - 1$ の例を示す。根は以下の 5 つである。

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

これらの根を見つけるプログラムの以下に例示する。

```
#include <stdio.h>
#include <gsl/gsl_poly.h>
int main (void)
{
    int i;
    /* P(x) = -1 + x^5 の係数 */
    double a[6] = { -1, 0, 0, 0, 0, 1 };
    double z[10];

    gsl_poly_complex_workspace * w
        = gsl_poly_complex_workspace_alloc(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;
}
```

プログラムの出力はこのようになる。

```
bash$ ./a.out
z0 = -0.809016994374947451 +0.587785252292473137
z1 = -0.809016994374947451 -0.587785252292473137
z2 = +0.309016994374947451 +0.951056516295153642
z3 = +0.309016994374947451 -0.951056516295153642
z4 = +1.000000000000000000 +0.000000000000000000
```

結果は解析的に得られる解 $z_n = \exp(2\pi ni/5)$ とよく一致している。

6.7 参考文献

平衡化を行う QR 分解とその誤差解析が以下の論文にある。

- R.S. Martin, G. Peters and J.H. Wilkinson, “The QR Algorithm for Real Hessenberg Matrices”, *Numerische Mathematik*, **14**, pp. 219–231 (1970).
- B.N. Parlett and C. Reinsch, “Balancing a Matrix for Calculation of Eigenvalues and Eigenvectors”, *Numerische Mathematik*, **13**, pp. 293–304 (1969).
- A. Edelman and H. Murakami, “Polynomial roots from companion matrix eigenvalues”, *Mathematics of Computation*, **64**(210), pp. 763–776 (1995).

差分商の公式化はアブラモウィッツとステグンによる以下の本にある。

- Milton Abramowitz and Irene A. Stegun, *Handbook of Mathematical Functions*, Sections 25.1.4 and 25.2.26, National Bureau Of Standards (1964).

第7章　特殊関数

この章では特殊関数について説明する。GSL では、エアリー関数、ベッセル関数、クラウゼン関数、クーロンの波動関数、ウィグナーの結合係数、ドーソン関数、デバイの関数、二重対数、楕円積分、ヤコビの楕円関数、ガウスの誤差関数、指数積分、フェルミ=ディラックの関数、ガンマ関数、ゲーゲンバウア関数、超幾何関数、ラゲール関数、ルジャンドル関数と球面調和関数、プサイ (二重ガンマ) 関数、シンクロトロン関数、輸送関数、三角関数とゼータ関数が用意されている。各関数について、計算した関数値に含まれる誤差の推定値も計算できる。

各関数には、たとえば 'gsl_sf_airy.h' や 'gsl_sf_bessel.h' のようにそれぞれヘッダファイルが用意されている。しかし 'gsl_sf.h' ですべて一度にインクルードすることができる。

7.1　利用法

特殊関数の呼び出し方には、関数値を返す「通常形 (natural form)」とエラーコードを返す「エラー形 (error handling form)」の二通りの方法がある。呼び出し方は違うが、関数値の計算にはどちらも同じコードを使っている。

通常では、関数の返り値はその特殊関数の値であるため、数式中で直接、自然な形で使うことができる。たとえば以下の関数呼び出しではベッセル関数 $J_0(x)$ の値を計算する。

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

このやり方では、エラー・コードや推定誤差 (error estimate) を知ることができない。そのためには、値を入れて返すことができる引数を与えて、エラー形で関数を呼び出す。

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

エラー形の呼び出しを行うには、関数名の末尾に _e が付いた関数を使う。オーバーフローや桁落ちなどが発生したら、返り値でそれがわかる。エラーがなにも生じなかったときは GSL_SUCCESS を返す。

7.2　gsl_sf_result 構造体

エラー形の特殊関数は、計算された関数値の推定誤差も同時に計算している。これを返すために関数値と推定誤差を要素に持つ構造体が定義されている。この構造体は 'gsl_sf_result.h' に定義されている。

gsl_sf_result 構造体は、関数値と推定誤差を保持するため、以下のように定義されている。

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

val は関数値、*err* は関数値の推定絶対誤差 (estimate of the absolute error) である。

場合によっては関数中でオーバーフローやアンダーフローを検知して処理することがある。そういった場合、double などの組込型の表現範囲を超えた計算結果を保存するために、関数値と推定誤差の他に指数係数が分かると、解が精密にではなくても得られるので便利である。その指数係数は、gsl_sf_result 構造体の関数値と推定誤差に加えて、関数値が result * 10^(e10) で得られるような係数 *e10* として gsl_sf_result_e10 構造体で用意されている。

```
typedef struct {
    double val;
    double err;
    int e10;
} gsl_sf_result_e10;
```

7.3　モード

各関数の値は、可能な限り倍精度で得られるようにしている。しかし特殊関数の種類によっては、倍精度を得るためには高次の項を計算しなければならず、それに非常に時間がかかるものもある。そういった場合には、mode 引数を使って、関数値の精度を落として計算時間を短くすることができる。この引数には以下の値が指定できる。

GSL_PREC_DOUBLE 倍精度。相対誤差はおおよそ 2×10^{-16} である。

GSL_PREC_SINGLE 単精度。相対誤差はおおよそ 1×10^{-7}　である。

GSL_PREC_APPROX 低精度。相対誤差はおおよそ 5×10^{-4} である。

低精度モードは、精度が最も低くなる代わりに、計算は最も速く行われる。

7.4　エアリー関数とその導関数

エアリー関数 (Airy function) $Ai(x)$、$Bi(x)$ は微分方程式 $y'' - xy = 0$ の解であり、以下の積分として定義される。

$$Ai(x) = \frac{1}{\pi}\int_0^\infty \cos(t^3/3 + xt)dt \tag{7.1}$$

$$Bi(x) = \frac{1}{\pi}\int_0^\infty (e^{-t^3/3} + \sin(t^3/3 + xt))dt \tag{7.2}$$

詳細はアブラモウィッツ＆ステグンの第 10.4 節を参照。エアリー関数はヘッダファイル 'gsl_sf_airy.h' で宣言されている。

7.4.1 エアリー関数

`double gsl_sf_airy_Ai` (*double* `x`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_airy_Ai_e` (*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`)[Function]

エアリー関数 $Ai(x)$ の値を指定された精度 *mode* で計算する。

`double gsl_sf_airy_Bi` (*double* `x`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_airy_Bi_e` (*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`)[Function]

エアリー関数 $Bi(x)$ の値を指定された精度 *mode* で計算する。

`double gsl_sf_airy_Ai_scaled` (*double* `x`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_airy_Ai_scaled_e` (*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`) [Function]

スケーリングされたエアリー関数の値 $S_A(x)Ai(x)$ を計算する (漸近公式で近似した際の指数項がスケーリングによりキャンセルされ、$x \to \infty$ での減少が緩やかになる)。係数 $S_A(x)$ は $x > 0$ のとき $\exp(+(2/3)x^{3/2})$、$x < 0$ のとき 1 である。

`double gsl_sf_airy_Bi_scaled` (*double* `x`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_airy_Bi_scaled_e` (*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`) [Function]

スケーリングされたエアリー関数の値 $S_B(x)Bi(x)$ を計算する (漸近公式で近似した際の指数項がスケーリングによりキャンセルされ、$x \to \infty$ で発散しなくなる)。係数 $S_B(x)$ は $x > 0$ のとき $\exp(-(2/3)x^{3/2})$、$x < 0$ のとき 1 である。

7.4.2 エアリー関数の導関数

`double gsl_sf_airy_Ai_deriv` (*double* `x`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_airy_Ai_deriv_e` (*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`) [Function]

エアリー関数の導関数 $Ai'(x)$ の値を指定された精度 *mode* で計算する。

`double gsl_sf_airy_Bi_deriv` (*double* `x`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_airy_Bi_deriv_e` (*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`) [Function]

エアリー関数の導関数 $Bi'(x)$ の値を指定された精度 *mode* で計算する。

`double gsl_sf_airy_Ai_deriv_scaled` (*double* `x`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_airy_Ai_deriv_scaled_e` (*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`) [Function]

スケーリングされたエアリー関数の導関数 $S_A(x)Ai'(x)$ の値を計算する。係数 $S_A(x)$ は $x > 0$ のとき $\exp(+(2/3)x^{3/2})$、$x < 0$ のとき 1 になる。

`double gsl_sf_airy_Bi_deriv_scaled (`*double* `x`, *gsl_mode_t* `mode``)` [Function]
`int gsl_sf_airy_Bi_deriv_scaled_e (`*double* `x`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result``)` [Function]

スケーリングされたエアリー関数の導関数 $S_B(x)Bi'(x)$ の値を計算する。係数 $S_B(x)$ は $x > 0$ のとき $\exp(-(2/3)x^{3/2})$、$x < 0$ のとき 1 になる。

7.4.3　エアリー関数の零点

`double gsl_sf_airy_zero_Ai (`*unsigned int* `s``)` [Function]
`int gsl_sf_airy_zero_Ai_e (`*unsigned int* `s, gsl_sf_result *`result`)` [Function]

エアリー関数 $Ai(x)$ の s 次の零点の座標を計算する。

`double gsl_sf_airy_zero_Bi (`*unsigned int* `s``)` [Function]
`int gsl_sf_airy_zero_Bi_e (`*unsigned int* `s, gsl_sf_result *`result`)` [Function]

エアリー関数 $Bi(x)$ の s 次の零点の座標を計算する。

7.4.4　エアリー導関数の零点

`double gsl_sf_airy_zero_Ai_deriv (`*unsigned int* `s``)` [Function]
`int gsl_sf_airy_zero_Ai_deriv_e (`*unsigned int* `s, gsl_sf_result *`result`)` [Function]

エアリー関数の導関数 $Ai'(x)$ の s 次の零点の座標を計算する。

`double gsl_sf_airy_zero_Bi_deriv (`*unsigned int* `s``)` [Function]
`int gsl_sf_airy_zero_Bi_deriv_e (`*unsigned int* `s, gsl_sf_result *`result`)` [Function]

エアリー関数の導関数 $Bi'(x)$ の s 次の零点の座標を計算する。

7.5　ベッセル関数

ベッセル関数 (Bessel function) とは、ベッセルの微分方程式

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0$$

の解であり、球ベッセル関数 (specical Bessel function) は球ベッセル微分方程式

$$\frac{d^2y}{dx^2} + \frac{2}{x}\frac{dy}{dx} + \left(1 - \frac{\alpha(\alpha+1)}{x^2}\right)y = 0$$

の解である。

この節では、円柱座標系のベッセル関数 (Cylindrical Bessel function、第一種および第二種ベッセル関数) $J_n(x)$、$Y_n(x)$、円柱座標系の変形ベッセル関数 (Modified cylindrical Bessel function、第一種および第二種の変形ベッセル関数) $I_n(x)$、$K_n(x)$、球ベッセル関数 $j_l(x)$、$y_l(x)$、変形球ベッセル関数 (Modified spherical Bessel function) $i_i(x)$、$k_l(x)$ について説明する。詳細はアブラモウィッツ&ステグンの第 9 および 10 章を参照のこと。ベッセル関数はヘッダファイル 'gsl_sf_bessel.h' で宣言されている。

7.5.1 第一種円柱ベッセル関数

`double gsl_sf_bessel_J0 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_J0_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

0 次の第一種円柱ベッセル関数 (regular cylindrical Bessel function) $J_0(x)$ の値を計算する。

`double gsl_sf_bessel_J1 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_J1_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

一次の第一種円柱ベッセル関数 $J_1(x)$ の値を計算する。

`double gsl_sf_bessel_Jn (`*int* **`n`**, *double* **`x`**`)` [Function]
`int gsl_sf_bessel_Jn_e (`*int* **`n`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

n 次の第一種円柱ベッセル関数 $J_n(x)$ の値を計算する。

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

nmin 次以上 *nmax* 次以下の第一種円柱ベッセル関数 $J_n(x)$ の値を計算し、引数で指定する配列 *result_array* に入れて返す。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

7.5.2 第二種円柱ベッセル関数

`double gsl_sf_bessel_Y0 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_Y0_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ について、0 次の第二種円柱ベッセル関数 irregular cylindrical Bessel function) $Y_0(x)$ の値を計算する。

`double gsl_sf_bessel_Y1 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_Y1_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ について、一次の第二種円柱ベッセル関数 $Y_1(x)$ の値を計算する。

`double gsl_sf_bessel_Yn (int n, double x)` [Function]
`int gsl_sf_bessel_Yn_e (int n, double x, gsl_sf_result * result)` [Function]

$x > 0$ について、n 次の第二種円柱ベッセル関数 $Y_n(x)$ の値を計算する。

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

nmin 次以上 *nmax* 次以下の第二種円柱ベッセル関数 $Y_n(x)$ の値を計算し、引数で指定する配列 *result_array* に入れて返す。定義域は $x > 0$ である。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

7.5.3 第一種変形円柱ベッセル関数

`double gsl_sf_bessel_I0 (double x)` [Function]
`int gsl_sf_bessel_I0_e (double x, gsl_sf_result * result)` [Function]

0 次の第一種変形円柱ベッセル関数 (regular modified cylindrical Bessel function) $I_0(x)$ の値を計算する。

`double gsl_sf_bessel_I1 (double x)` [Function]
`int gsl_sf_bessel_I1_e (double x, gsl_sf_result * result)` [Function]

一次の第一種変形円柱ベッセル関数 $I_1(x)$ の値を計算する。

`double gsl_sf_bessel_In (int n, double x)` [Function]
`int gsl_sf_bessel_In_e (int n, double x, gsl_sf_result * result)` [Function]

n 次の第一種変形円柱ベッセル関数 $I_n(x)$ の値を計算する。

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

nmin 次以上 *nmax* 次以下の第一種変形円柱ベッセル関数 $I_n(x)$ の値を計算し、引数で指定する配列 *result_array* に入れて返す。*nmin* は 0 または正の整数でなければならない。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

`double gsl_sf_bessel_I0_scaled (double x)` [Function]
`int gsl_sf_bessel_I0_scaled_e (double x, gsl_sf_result * result)` [Function]

スケーリングされた 0 次の第一種変形円柱ベッセル関数 $\exp(-|x|)I_0(x)$ の値を計算する (スケーリングにより $x \to \infty$ で発散しなくなる)。

`double gsl_sf_bessel_I1_scaled (double x)` [Function]
`int gsl_sf_bessel_I1_scaled_e (double x, gsl_sf_result * result)` [Function]

スケーリングされた一次の第一種変形円柱ベッセル関数 $\exp(-|x|)I_1(x)$ の値を計算する。

`double gsl_sf_bessel_In_scaled (`*int* **`n`**, *double* **`x`**`)` [Function]
`int gsl_sf_bessel_In_scaled_e (`*int* **`n`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

スケーリングされた n 次の第一種変形円柱ベッセル関数 $\exp(-|x|)I_n(x)$ の値を計算する。

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

nmin 次以上 *nmax* 次以下の第一種変形円柱ベッセル関数 $\exp(-|x|)I_n(x)$ の値を計算し、引数で指定する配列 *result_array* に入れて返す。*nmin* は 0 または正の整数でなければならない。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

7.5.4 第二種変形円柱ベッセル関数

`double gsl_sf_bessel_K0 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_K0_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ について、0 次の第二種変形円柱ベッセル関数 (irregular modified cylindrical Bessel function) $K_0(x)$ の値を計算する。

`double gsl_sf_bessel_K1 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_K1_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ について、一次の第二種変形円柱ベッセル関数 $K_1(x)$ の値を計算する。

`double gsl_sf_bessel_Kn (`*int* **`n`**, *double* **`x`**`)` [Function]
`int gsl_sf_bessel_Kn_e (`*int* **`n`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ について、n 次の第二種変形円柱ベッセル関数 $K_n(x)$ の値を計算する。

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

nmin 次以上 *nmax* 次以下の第二種変形円柱ベッセル関数 $K_n(x)$ の値を計算し、引数で指定する配列 *result_array* に入れて返す。*nmin* は 0 または正でなければならない。関数の定義域は $x > 0$ である。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

`double gsl_sf_bessel_K0_scaled (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_K0_scaled_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ についてスケーリングされた 0 次の第二種変形円柱ベッセル関数 $\exp(x)K_0(x)$ の値を計算する (スケーリングしても $x \to \infty$ で 0 に収束するが、収束が遅くなり大きな x に対しても数値計算上の精度を維持しやすくなる)。

`double gsl_sf_bessel_K1_scaled (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_K1_scaled_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ についてスケーリングされた一次の第二種変形円柱ベッセル関数 $\exp(x)K_1(x)$ の値を計算する。

`double gsl_sf_bessel_Kn_scaled (`*int* **`n`**, *double* **`x`**`)` [Function]
`int gsl_sf_bessel_Kn_scaled_e (`*int* **`n`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$x > 0$ についてスケーリングされた n 次の第二種変形円柱ベッセル関数 $\exp(x)K_n(x)$ の値を計算する。

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

nmin 次以上 *nmax* 次以下のスケーリングされた第二種変形円柱ベッセル関数 $\exp(x)K_n(x)$ の値を計算し、引数で指定する配列 *result_array* に入れて返す。*nmin* は 0 または正でなければならない。関数の定義域は $x > 0$ である。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

7.5.5　第一種球ベッセル関数

`double gsl_sf_bessel_j0 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_j0_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

0 次の第一種球ベッセル関数 (regular spherical Bessel function) $j_0(x) = \sin(x)/x$ の値を計算する。

`double gsl_sf_bessel_j1 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_j1_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

一次の第一種球ベッセル関数 $j_1(x) = (\sin(x)/x - \cos(x))/x$ の値を計算する。

`double gsl_sf_bessel_j2 (`*double* **`x`**`)` [Function]
`int gsl_sf_bessel_j2_e (`*double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

二次の第一種球ベッセル関数 $j_2(x) = ((3/x^2 - 1)\sin(x) - 3\cos(x)/x)/x$ の値を計算する。

`double gsl_sf_bessel_jl (`*int* **`l`**, *double* **`x`**`)` [Function]
`int gsl_sf_bessel_jl_e (`*int* **`l`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

$l \geq 0$, $x \geq 0$ について、l 次の第一種球ベッセル関数 $j_l(x)$ の値を計算する。

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

l 次の第一種球ベッセル関数 $j_l(x)$ の値を、0 から *lmax* までの l の各値について計算する。ここで $lmax \geq 0$ かつ $x \geq 0$ であり、関数値は配列 *result_array* に入れられる。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

`int gsl_sf_bessel_jl_steed_array` (*int* **`lmax`**, *double* **`x`**, *double* * **`jl_x_array`**)
[Function]

l 次の第一種球ベッセル関数 $j_l(x)$ の値を、0 から *lmax* までの l の各値についてスティード (J. W. Steed) のアイデアに基づいた方法を使って計算する。ここで $lmax \geq 0$ かつ $x \geq 0$ であり、関数値は配列 *jl_x_array* に入れられる。GSL に実装されているアルゴリズムはバーネット (R. A. Barnett) が定式化したものであり、*Computational Physics Communications* **21**, pp. 297–314 (1981) に解説がある。スティードの方法は、他の関数で使っている漸化式による方法に比べて安定であるが、計算は遅い。

7.5.6 第二種球ベッセル関数

`double gsl_sf_bessel_y0` (*double* **`x`**) [Function]
`int gsl_sf_bessel_y0_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

0 次の第二種球ベッセル関数 (irregular spherical Bessel function) $y_0(x) = -\cos(x)/x$ の値を計算する。

`double gsl_sf_bessel_y1` (*double* **`x`**) [Function]
`int gsl_sf_bessel_y1_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

一次の第二種球ベッセル関数 $y_1(x) = -(\cos(x)/x + \sin(x))/x$ の値を計算する。

`double gsl_sf_bessel_y2` (*double* **`x`**) [Function]
`int gsl_sf_bessel_y2_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

二次の第二種球ベッセル関数 $y_2(x) = (-3/x^3 + 1/x)\cos(x) - (3/x^2)\sin(x)$ の値を計算する。

`double gsl_sf_bessel_yl` (*int* **`l`**, *double* **`x`**) [Function]
`int gsl_sf_bessel_yl_e` (*int* **`l`**, *double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

$l \geq 0$ について、l 次の第二種球ベッセル関数 $y_l(x)$ の値を計算する。

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

l 次の第一種球ベッセル関数 $y_l(x)$ の値を、0 から $lmax$ までの l の各値について計算する。ここで $lmax \geq 0$ であり、関数値は配列 $result\_array$ に入れられる。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

7.5.7　第一種変形球ベッセル関数

第一種変形球ベッセル関数 (regular modified spherical Bessel function) $i_l(x)$ は、非整数次の変形ベッセル関数から $i_l(x) = \sqrt{\pi/(2x)} I_{l+1/2}(x)$ として得られる関数である。

`double gsl_sf_bessel_i0_scaled (`*double* `x``)` [Function] `int gsl_sf_bessel_i0_scaled_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた 0 次の第一種変形球ベッセル関数 $\exp(-|x|) i_0(x)$ の値を計算する。

`double gsl_sf_bessel_i1_scaled (`*double* `x``)` [Function] `int gsl_sf_bessel_i1_scaled_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた一次の第一種変形球ベッセル関数 $\exp(-|x|) i_1(x)$ の値を計算する。

`double gsl_sf_bessel_i2_scaled (`*double* `x``)` [Function] `int gsl_sf_bessel_i2_scaled_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた二次の第一種変形球ベッセル関数 $\exp(-|x|) i_2(x)$ の値を計算する。

`double gsl_sf_bessel_il_scaled (`*int* `l`, *double* `x``)` [Function] `int gsl_sf_bessel_il_scaled_e (`*int* `l`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた l 次の第二種球ベッセル関数 $\exp(-x) i_l(x)$ の値を計算する。

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

スケーリングされた l 次の第一種変形ベッセル関数 $\exp(-|x|) i_l(x)$ の値を、0 から $lmax$ までの l の各値について計算する。ここで $lmax \geq 0$ であり、関数値は配列 $result\_array$ に入れられる。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある。

7.5.8　第二種変形球ベッセル関数

第二種変形球ベッセル関数 (irregular modified spherical Bessel function) $k_l(x)$ は、非整数次の第二種変形ベッセル関数から $k_l(x) = \sqrt{\pi/(2x)} K_{l+1/2}(x)$ として得られる関数である。

`double gsl_sf_bessel_k0_scaled (`*double* `x``)` [Function]
`int gsl_sf_bessel_k0_scaled_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた 0 次の第一種変形球ベッセル関数 $\exp(x)k_0(x)$ の値を $x > 0$ で計算する。

`double gsl_sf_bessel_k1_scaled (`*double* `x``)` [Function]
`int gsl_sf_bessel_k1_scaled_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた一次の第一種変形球ベッセル関数 $\exp(x)k_1(x)$ の値を $x > 0$ で計算する。

`double gsl_sf_bessel_k2_scaled (`*double* `x``)` [Function]
`int gsl_sf_bessel_k2_scaled_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた二次の第一種変形球ベッセル関数 $\exp(x)k_2(x)$ の値を $x > 0$ で計算する。

`double gsl_sf_bessel_kl_scaled (`*int* `l`, *double* `x``)` [Function]
`int gsl_sf_bessel_kl_scaled_e (`*int* `l`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

スケーリングされた l 次の第一種変形球ベッセル関数 $\exp(x)k_l(x)$ の値を $x > 0$ で計算する。

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

スケーリングされた l 次の第一種変形ベッセル関数 $\exp(x)k_l(x)$ の値を、0 から *lmax* までの l の各値について $x > 0$ で計算する。ここで $lmax \geq 0$ であり、関数値は配列 *result_array* に入れられる。関数値は計算時間の短縮のために漸化式で計算されるため、厳密な値と少し異なることがある

7.5.9 非整数次の第一種ベッセル関数

`double gsl_sf_bessel_Jnu (`*double* `nu`, *double* `x``)` [Function]
`int gsl_sf_bessel_Jnu_e (`*double* `nu`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

非整数の ν 次の第一種円柱ベッセル関数 (regular cylindrical Bessel function of fractional order ν) $J_\nu(x)$ の値を計算する。

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

数列をなす x のそれぞれの値について、非整数の ν 次の第一種円柱ベッセル関数 $J_\nu(x)$ の値を計算する。x の値は長さ *size* の配列 v に入れて渡す。x の値はすべて正で、ソートされていなければならない。v の値は $J_\nu(x_i)$ で上書きされる。

7.5.10　非整数次の第二種ベッセル関数

`double gsl_sf_bessel_Ynu (double nu, double x)` [Function]
`int gsl_sf_bessel_Ynu_e (double nu, double x, gsl_sf_result * result)` [Function]

非整数の ν 次の第二種円柱ベッセル関数 (irregular cylindrical Bessel function of fractional order ν) $Y_\nu(x)$ の値を計算する。

7.5.11　非整数次の第一種変形ベッセル関数

`double gsl_sf_bessel_Inu (double nu, double x)` [Function]
`int gsl_sf_bessel_Inu_e (double nu, double x, gsl_sf_result * result)` [Function]

非整数の ν 次の第一種変形円柱ベッセル関数 (regular modified Bessel function of fractional order ν) $I_\nu(x)$ の値を $x > 0$ かつ $\nu > 0$ で計算する。

`double gsl_sf_bessel_Inu_scaled (double nu, double x)` [Function]
`int gsl_sf_bessel_Inu_scaled_e (double nu, double x, gsl_sf_result * result)` [Function]

スケーリングされた非整数の ν 次の第一種変形円柱ベッセル関数 $\exp(-|x|)I_\nu(x)$ の値を $x > 0$ かつ $\nu > 0$ で計算する。

7.5.12　非整数次の第二種変形ベッセル関数

`double gsl_sf_bessel_Knu (double nu, double x)` [Function]
`int gsl_sf_bessel_Knu_e (double nu, double x, gsl_sf_result * result)` [Function]

非整数の ν 次の第二種変形円柱ベッセル関数 (irregular modified Bessel function of fractional order ν) $K_\nu(x)$ の値を $x > 0$ かつ $\nu > 0$ で計算する。

`double gsl_sf_bessel_lnKnu (double nu, double x)` [Function]
`int gsl_sf_bessel_lnKnu_e (double nu, double x, gsl_sf_result * result)` [Function]

非整数の ν 次の第二種変形円柱ベッセル関数 $ln(K_\nu(x))$ の対数値を $x > 0$ かつ $\nu > 0$ で計算する。

`double gsl_sf_bessel_Knu_scaled (double nu, double x)` [Function]
`int gsl_sf_bessel_Knu_scaled_e (double nu, double x, gsl_sf_result * result)` [Function]

スケーリングされた非整数の ν 次の第二種変形円柱ベッセル関数 $\exp(+|x|)K_\nu(x)$ の値を $x > 0$ かつ $\nu > 0$ で計算する。

7.5.13 第一種ベッセル関数の零点

`double gsl_sf_bessel_zero_J0 (`*unsigned int* **`s`**`)` [Function]
`int gsl_sf_bessel_zero_J0_e (`*unsigned int* **`s`**, *gsl_sf_result ** **`result`**`)` [Function]

ベッセル関数 $J_0(x)$ の s 次の正の零点を求める。

`double gsl_sf_bessel_zero_J1 (`*unsigned int* **`s`**`)` [Function] `int`
`gsl_sf_bessel_zero_J1_e (`*unsigned int* **`s`**, *gsl_sf_result ** **`result`**`)` [Function]

ベッセル関数 $J_1(x)$ の s 次の正の零点を求める。

`double gsl_sf_bessel_zero_Jnu (`*doble* **`nu`**, *unsigned int* **`s`**`)` [Function]
`int gsl_sf_bessel_zero_Jnu_e (`*doble* **`nu`**, *unsigned int* **`s`**, *gsl_sf_result ** **`result`**`)`
[Function]

ベッセル関数 $J_\nu(x)$ の s 次の正の零点を求める。nu が負の場合については、現在の実装では計算できない。

7.6 クラウゼン関数

クラウゼン関数 (Clausen function) は以下の積分で定義される。

$$Cl_2(x) = -\int_0^x \log(2\sin(t/2))dt \tag{7.3}$$

クラウゼン関数は、ダイログ (二重対数、dilogarithm) と $Cl_2(\theta) = \mathrm{Im}(Li_2(e^{i\theta}))$ という関係がある。またリーマンのゼータ関数やディリクレの L 関数とも関係がある。クラウゼン関数は '`gsl_sf_clausen.h`' で宣言されている。

`double gsl_sf_clausen (`*double* **`x`**`)` [Function]
`int gsl_sf_clausen_e (`*double* **`x`**, *gsl_sf_result ** **`result`**`)` [Function]

クラウゼン積分 $Cl_2(x)$ を計算する。

7.7 クーロン関数

クーロン関数 (Coulomb function) はヘッダファイル '`gsl_sf_coulomb.h`' で宣言されている。束縛状態 (bound state) と散乱状態 (scattering solution) の両方が計算できる。

7.7.1 水素様原子の規格化された束縛状態

`double gsl_sf_hydrogenicR_1 (`*double* **`Z`**, *double* **`r`**`)` [Function] `int`
`gsl_sf_hydrogenicR_1_e (`*double* **`Z`**, *double* **`r`**, *gsl_sf_result ** **`result`**`)` [Function]

原子番号 Z の水素の最低次 (主量子数が 1) の束縛状態 ($1s$ 軌道) の規格化された動径波動関数 (the lowest-order normalized hydrogenic bound state radial wavefunction) $R_1 := 2Z\sqrt{Z}\exp(-Zr)$ の値を計算する。r はボーア半径に対する相対値である。

`double gsl_sf_hydrogenicR (`*int* `n`, *int* `l`, *double* `Z`, *double* `r``)` [Function]
`int gsl_sf_hydrogenicR_e (`*int* `n`, *int* `l`, *double* `Z`, *double* `r`, *gsl_sf_result* * `result``)` [Function]

以下で表される、n 次の束縛状態の規格化された動径波動関数の値を計算する (n は主量子数、l は方位量子数)。

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

ここで $L^a_b(x)$ は一般化ラゲール多項式 (generalzied Laguerre polynomial) である (ラゲール関数、7.22 節参照)。波動関数 (wavefunction) ψ を $\psi(n,l,r) = R_n Y_{lm}$ とおいて規格化している。

7.7.2　クーロンの波動関数

クーロンの波動関数 $F_L(\eta,x)$ および $G_L(\eta,x)$ はアブラモウィッツ＆ステグンの第 14 章に説明されている。これらの関数中では変数の値が非常に広い範囲をとるため、オーバーフローに注意する必要がある。オーバーフローが生じたときは `GSL_EOVRFLW` を返し、引数 *exp_F* と *exp_G* に計算結果の値の指数部を入れて返す。その場合、波動関数の値は以下のように計算することで得られる。

$$\begin{aligned} F_L(\eta,) &= fc[k_L] * \exp(exp_F) \\ G_L(\eta,) &= gc[k_L] * \exp(exp_G) \\ F'_L(\eta,x) &= fcp[k_L] * \exp(exp_F) \\ G'_L(\eta,x) &= gcp[k_L] * \exp(exp_G) \end{aligned}$$

`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``)` [Function]

クーロンの波動関数 $F_L(\eta,x)$ および $G_{L-k}(\eta,x)$ と、その x に関する導関数値 $F'_L(\eta,x)$ および $G'_{L-k}(\eta,x)$ を計算する。引数の取りうる値は L によって決まり、$L-k > -1/2$、$x > 0$ を満たさなければならない。k は整数でなければならない。しかし L 自体は整数でなくてもよい。計算結果は、関数値がそれぞれ F と G に、導関数値が Fp と Gp に入れて返される。オーバーフローが生じたときは `GSL_EOVRFLW` が返され、おおよその関数値を得るための指数係数が引数 *exp_F* と *exp_G* に入れて返される。

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

$L = Lmin \ldots Lmin + kmax$ について関数値 $F_L(\eta, x)$ を計算し、結果を *fc_array* に入れて返す。オーバーフローが生じたときは指数係数が *F_exponent* に入れて返される。

`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`) [Function]

$L = Lmin \ldots Lmin + kmax$ について関数値 $F_L(\eta, x)$ と $G_L(\eta, x)$ を計算し、結果を *fc_array* と *gc_array* に入れて返す。オーバーフローが生じたときは指数係数が *F_exponent* と *G_exponent* に入れて返される。

`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`) [Function]

$L = Lmin \ldots Lmin + kmax$ について関数値 $F_L(\eta, x)$ と $G_L(\eta, x)$ とその導関数値 $F'_L(\eta, x)$ と $G'_L(\eta, x)$ を計算し、結果を *fc_array*、*gc_array*、*fcp_array*、*gcp_array* に入れて返す。オーバーフローが生じたときは指数係数が *F_exponent* と *G_exponent* に入れて返される。

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

$L = Lmin \ldots Lmin + kmax$ についてクーロンの波動関数を与えられる引数で除した値 $F_L(\eta, x)/x$ を計算し、結果を *fc_array* に入れて返す。オーバーフローが生じたときは指数係数が *F_exponent* に入れて返される。この関数は、極限 $\eta \to 0$ で球ベッセル関数になる。

7.7.3 クーロンの波動関数の規格化係数

クーロンの波動関数の規格化係数はアブラモウィッツ＆ステグンの 14.1.7 節に定義されている。

`int gsl_sf_coulomb_CL_e` (*double* `L`, *double* `eta`, *gsl_sf_result* * `result`) [Function]

クーロンの波動関数の規格化係数 $C_L(\eta)$ を計算する。$L > -1$ である。

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

クーロンの波動関数の規格化係数 $C_L(\eta)$ を $L = Lmin \ldots Lmin + kmax$ について計算する。$Lmin > -1$ である。

7.8　結合係数

ウィグナー (Eugene Paul Wigner) の 3-j、6-j、9-j 記号は、角運動量ベクトルの合成に用いられる係数である。標準的な結合係数関数ではその引数は整数か整数の 1/2 であるため、GSL の実装では慣例にしたがい、スピンの値を引数に与える際に実際の値を二倍して整数として与える。3-j 記号についてはアブラモウィッツ＆ステグン の 27.9 節を参照のこと。この節の関数はヘッダファイル 'gsl_sf_coupling.h' で宣言されている。

7.8.1　3-j 記号

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

以下に示すウィグナーの 3-j 係数を計算する。

$$\left(\begin{array}{ccc} ja & jb & jc \\ ma & mb & mc \end{array} \right)$$

引数は 1/2 を単位として、$ja = two\_ja/2$、$ma = two\_ma/2$ のように与える。

7.8.2　6-j 記号

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

以下に示すウィグナーの 6-j 係数を計算する。

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

引数は 1/2 を単位として、$ja = two\_ja/2$、$ma = two\_ma/2$ のように与える。

7.8.3　9-j 記号

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) [Function]
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) [Function]

以下に示すウィグナーの 9-j 係数を計算する。

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

引数は 1/2 を単位として、$ja = two\_ja/2$、$ma = two\_ma/2$ のように与える。

7.9 ドーソン関数

ドーソンの積分 (Dawson integral) は $\exp(-x^2)\int_0^x \exp(t^2)dt$ として定義される。ガウスの誤差関数 (Gauss error function) と似た形であり、その値がアブラモウィッツ＆ステグンの表 7.5 に挙げられている。この積分を計算する関数はヘッダファイル 'gsl_sf_dawson.h' で宣言されている。

`double gsl_sf_dawson (`*double* `x``)` [Function]
`int gsl_sf_dawson_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

与えられる x についてドーソンの積分を計算する。

7.10 デバイの関数

デバイの関数 (Debye function) は以下の積分として定義される。

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

多重対数 (polylogarithm) やリーマンのゼータ関数と関係があり、アブラモウィッツ＆ステグンの 27.1 節にその定義と値がある。デバイの関数は 'gsl_sf_debye.h' で宣言されている。

`double gsl_sf_debye_1 (`*double* `x``)` [Function]
`int gsl_sf_debye_1_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

1 次のデバイ関数 $D_1(x) = (1/x)\int_0^x (t/(e^t - 1))dt$ の値を計算する。

`double gsl_sf_debye_2 (`*double* `x``)` [Function]
`int gsl_sf_debye_2_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

2 次のデバイ関数 $D_2(x) = (2/x^2)\int_0^x (t^2/(e^t - 1))dt$ の値を計算する。

`double gsl_sf_debye_3 (`*double* `x``)` [Function]
`int gsl_sf_debye_3_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

3 次のデバイ関数 $D_3(x) = (3/x^3)\int_0^x (t^3/(e^t - 1))dt$ の値を計算する。

`double gsl_sf_debye_4 (`*double* `x``)` [Function]
`int gsl_sf_debye_4_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

4 次のデバイ関数 $D_4(x) = (4/x^4)\int_0^x (t^4/(e^t - 1))dt$ の値を計算する。

`double gsl_sf_debye_5` (*double* **`x`**) [Function]
`int gsl_sf_debye_5_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

5 次のデバイ関数 $D_5(x) = (5/x^5)\int_0^x (t^5/(e^t-1))dt$ の値を計算する。

`double gsl_sf_debye_6` (*double* **`x`**) [Function]
`int gsl_sf_debye_6_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

6 次のデバイ関数 $D_6(x) = (6/x^6)\int_0^x (t^6/(e^t-1))dt$ の値を計算する。

7.11　二重対数

二重対数 (dilogarithm、ダイログ) はヘッダファイル '`gsl_sf_dilog.h`' で宣言されている。

7.11.1　実数引数

`double gsl_sf_dilog` (*double* **`x`**) [Function]
`int gsl_sf_dilog_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

二重対数を実数の引数に対して計算する。これはルウィン (Leonard Lewin) の記法では $Li_2(x)$ で表され、実数 x の二重対数の実部である。これは積分 $Li_2(x) = -\mathrm{Re}\int_0^x \log(1-s)/s ds$ として定義される。$x \leq 1$ のとき $\mathrm{Im}(Li_2(x)) = 0$、$x > 1$ のとき $\mathrm{Im}(Li_2(x)) = -\pi\log(x)$ である。

アブラモウィッツ＆ステグン第 27.7 節では、$Li_2(x)$ の代わりにスペンス積分 (Spence's integral) $S(x) = Li_2(1-x)$ として二重積分を記述している。

7.11.2　複素数引数

`int gsl_sf_complex_dilog_e` (*double* **`r`**, *double* **`theta`**, *gsl_sf_result* * **`result_re`**, *gsl_sf_result* * **`result_im`**) [Function]

複素数の引数 $z = r\exp(i\theta)$ に対し二重対数を複素数として計算する。計算結果の実部と虚部はそれぞれ *result_re* と *result_im* に入れて返される。

7.12　基本演算

以下の関数で、誤差を含む二つの実数値の積に、その誤差がどの程度影響するのかを把握することができる。これらはヘッダファイル '`gsl_sf_elementary.h`' で宣言されている。
`int gsl_sf_multiply_e` (*double* **`x`**, *double* **`y`**, *gsl_sf_result* * **`result`**) [Function]

x と y の値の積を計算し、積とその誤差を *result* に入れて返す。

`int gsl_sf_multiply_err_e` (*double* `x`, *double* `dx`, *double* `y`, *double* `dy`, *gsl_sf_result* * `result`) [Function]

x と y がそれぞれ絶対誤差 dx、dy を持つとして積の値を計算する。その積 $xy \pm xy\sqrt{(dx/x)^2 + (dy/y)^2}$ が *result* に入れて返される。

7.13 楕円積分

この節の関数はヘッダファイル '`gsl_sf_ellint.h`' で宣言されている。アブラモウィッツ＆ステグンの第 17 節に詳細が記述されている。

7.13.1 ルジャンドルの標準形の定義

楕円積分 (elliptic integral) には第一種、第二種、第三種があり、それぞれのルジャンドルの標準形 (Legendre form) での表記 $F(\phi,k)$、$E(\phi,k)$、$\Pi(\phi,k,n)$ は以下で定義される。

$$
\begin{aligned}
F(\phi,k) &= \int_0^\phi \frac{1}{\sqrt{1-k^2\sin^2(t)}}\,dt \\
E(\phi,k) &= \int_0^\phi \sqrt{1-k^2\sin^2(t)}\,dt \\
\Pi(\phi,k,h) &= \int_0^\phi \frac{1}{(1+n\sin^2(t))\sqrt{1-k^2\sin^2(t)}}\,dt
\end{aligned}
$$

第一種および第二種において、t について定積分し k の関数としたものが完全楕円積分 (complete elliptic integral) であり、ルジャンドルの標準形による第一種および第二種の完全楕円積分 (complete Legendre form) が $K(k) = F(\pi/2,k)$ および $E(k) = E(\pi/2,k)$ で表される。この定積分を行わないものを不完全楕円積分 (incomplete Legendre form) と呼ぶ。

なお上の記述は、カールソン (Bille C. Carlson) の論文 "Computing Elliptic Integrals by Duplication", *Numerische Mathematik*, **33**, pp. 1–16 (1979) にしたがっており、アブラモウィッツ＆ステグンでの m を k^2 に、n を $-n$ に置き換えた記述になっている。

7.13.2　カールソン形式の定義

楕円積分はまた、カールソンの対称形式 (Carlson symmetric form) $RC(x,y)$、$RD(x,y,z)$、$RF(x,y,z)$、$RJ(x,y,z,p)$ によってそれぞれ、以下で定義される。

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

7.13.3　ルジャンドルの標準形による完全楕円積分

`double gsl_sf_ellint_Kcomp (`*double* **`k`**, *gsl_mode_t* **`mode`**`)` [Function]
`int gsl_sf_ellint_Kcomp_e (`*double* **`k`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * **`result`**`)` [Function]

完全楕円積分 (complete elliptic integral) $K(k)$ を *mode* で指定された精度で計算する。アブラモウィッツ＆ステグンではパラメータ $m\ (= k^2)$ を使って定義されている。

`double gsl_sf_ellint_Ecomp (`*double* **`k`**, *gsl_mode_t* **`mode`**`)` [Function]
`int gsl_sf_ellint_Ecomp_e (`*double* **`k`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * **`result`**`)` [Function]

完全楕円積分 $E(k)$ を *mode* で指定された精度で計算する。アブラモウィッツ＆ステグンではパラメータ $m\ (= k^2)$ を使って定義されている。

`double gsl_sf_ellint_Pcomp (`*double* **`k`**, *gsl_mode_t* **`mode`**`)` [Function]
`int gsl_sf_ellint_Pcomp_e (`*double* **`k`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * **`result`**`)` [Function]

完全楕円積分 $\Pi(k,n)$ を *mode* で指定された精度で計算する。アブラモウィッツ＆ステグンでは n の符号が $n = -n$ と反転しており、$m = k^2$ および $\sin^2(\alpha) = k^2$ として定義されている。

7.13.4　ルジャンドルの標準形による不完全楕円積分

`double gsl_sf_ellint_F (`*double* **`phi`**, *double* **`k`**, *gsl_mode_t* **`mode`**`)` [Function]
`int gsl_sf_ellint_F_e (`*double* **`phi`**, *double* **`k`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * **`result`**`)` [Function]

不完全楕円積分 (incomplete elliptic integral) $F(\phi, k)$ を *mode* で指定された精度で計算する。アブラモウィッツ＆ステグンではパラメータ $m\ (= k^2)$ を使って定義されている。

`double gsl_sf_ellint_E` (*double* **`phi`**, *double* **`k`**, *gsl_mode_t* **`mode`**) [Function]
`int gsl_sf_ellint_E_e` (*double* **`phi`**, *double* **`k`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * ***`result`***) [Function]

不完全楕円積分 $E(\phi, k)$ を *mode* で指定された精度で計算する。アブラモウィッツ＆ステグンではパラメータ $m\ (= k^2)$ を使って定義されている。

`double gsl_sf_ellint_P` (*double* **`phi`**, *double* **`k`**, *double* **`n`**, *gsl_mode_t* **`mode`**) [Function]
`int gsl_sf_ellint_P_e` (*double* **`phi`**, *double* **`k`**, *double* **`n`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * ***`result`***) [Function]

不完全楕円積分 $\Pi(\phi, k, n)$ を *mode* で指定された精度で計算する。アブラモウィッツ＆ステグンでは n の符号が $n = -n$ と反転しており、$m = k^2$ および $\sin^2(\alpha) = k^2$ として定義されている。

`double gsl_sf_ellint_D` (*double* **`phi`**, *double* **`k`**, *double* **`n`**, *gsl_mode_t* **`mode`**) [Function]
`int gsl_sf_ellint_D_e` (*double* **`phi`**, *double* **`k`**, *double* **`n`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * ***`result`***) [Function]

カールソン形式 $RD(x, y, z)$ を使って以下で定義される不完全楕円積分 $D(\phi, k)$ を計算する。

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

引数の n は計算には使われない。今後廃止される予定である。

7.13.5 カールソン形式

`double gsl_sf_ellint_RC` (*double* **`x`**, *double* **`y`**, *gsl_mode_t* **`mode`**) [Function]
`int gsl_sf_ellint_RC_e` (*double* **`x`**, *double* **`y`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * ***`result`***) [Function]

不完全楕円積分 $RC(x, y)$ を *mode* で指定された精度で計算する。

`double gsl_sf_ellint_RD` (*double* **`x`**, *double* **`y`**, *double* **`z`**, *gsl_mode_t* **`mode`**) [Function]
`int gsl_sf_ellint_RD_e` (*double* **`x`**, *double* **`y`**, *double* **`z`**, *gsl_mode_t* **`mode`**, *gsl_sf_result* * ***`result`***) [Function]

不完全楕円積分 $RD(x, y, z)$ を *mode* で指定された精度で計算する。

`double gsl_sf_ellint_RF` (*double* `x`, *double* `y`, *double* `z`, *gsl_mode_t* `mode`)[Function]
`int gsl_sf_ellint_RF_e` (*double* `x`, *double* `y`, *double* `z`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`) [Function]

不完全楕円積分 $RF(x, y, z)$ を *mode* で指定された精度で計算する。

`double gsl_sf_ellint_RJ` (*double* `x`, *double* `y`, *double* `z`, *double* `p`, *gsl_mode_t* `mode`) [Function]
`int gsl_sf_ellint_RJ_e` (*double* `x`, *double* `y`, *double* `z`, *double* `p`, *gsl_mode_t* `mode`, *gsl_sf_result* * `result`) [Function]

不完全楕円積分 $RJ(x, y, z, p)$ を *mode* で指定された精度で計算する。

7.14　楕円積分 (ヤコビの標準形)

楕円積分の定義はまたヤコビの標準形 (Jacobian elliptic function) によっても表記される。その定義はアブラモウィッツ＆ステグンの第 16 章にある。以下の関数はヘッダファイル '`gsl_sf_elljac.h`' で宣言されている。
`int gsl_sf_elljac_e` (*double* `u`, *double* `m`, *double* * `sn`, *double* * `cn`, *double* * `dn`) [Function]

ヤコビの標準形による楕円関数 $\mathrm{sn}(u|m), \mathrm{cn}(u|m), \mathrm{dn}(u|m)$ を下降ランデン変換 (descending Landen transform) を使って計算する。

7.15　ガウスの誤差関数

誤差関数 (error function) は確率論、統計、物質科学などの幅広い分野で使われているモデルであり、アブラモウィッツ＆ステグンの第 7 章に説明がある。この節の関数はヘッダファイル '`gsl_sf_erf.h`' で宣言されている。

7.15.1　誤差関数

`double gsl_sf_erf` (*double* `x`) [Function]
`int gsl_sf_erf_e` (*double* `x`, *gsl_sf_result* * `result`) [Function]

ガウスの誤差関数 $\mathrm{erf}(x) = (2/\sqrt{\pi}) \int_0^x \exp(-t^2) dt$ を計算する。

7.15.2 誤差補関数

`double gsl_sf_erfc (`*double* `x``)` [Function]
`int gsl_sf_erfc_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

誤差関数の補関数 (complementary error function) $\mathrm{erfc}(x) = 1-\mathrm{erf}(x) = (2/\sqrt{\pi})\int_x^\infty \exp(-t^2)dt$ を計算する。

7.15.3 対数誤差補関数

`double gsl_sf_log_erfc (`*double* `x``)` [Function]
`int gsl_sf_log_erfc_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

誤差補関数の対数値 $\log(\mathrm{erfc}(x))$ を計算する。

7.15.4 確率関数

正規分布 (ガウス分布) の確率関数はアブラモウィッツ＆ステグンの第 26.2 節に説明されている。
`double gsl_sf_erf_Z (`*double* `x``)` [Function]
`int gsl_sf_erf_Z_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

ガウス分布の確率密度関数 $Z(x) = (1/\sqrt{2\pi})\exp(-x^2/2)$ の値を計算する。

`double gsl_sf_erf_Q (`*double* `x``)` [Function]
`int gsl_sf_erf_Q_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

ガウス分布の確率密度関数の上側確率 (upper tail) $Q(x) = (1/\sqrt{2\pi})\int_x^\infty \exp(-t^2/2)dt$ を計算する。

正規分布の危険関数 (hazard function) は、ミルズの逆比 (Inverse Mills ratio) で与えられ、以下のように表される。

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

x が $-\infty$ に向かうとき急激に減少し、x が $+\infty$ に 向かうとき $h(x) \sim x$ に漸近する。
`double gsl_sf_hazard (`*double* `x``)` [Function]
`int gsl_sf_hazard_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

正規分布の場合の危険関数を計算する。

7.16 指数関数

この節の関数はヘッダファイル '`gsl_sf_exp.h`' で宣言されている。

7.16.1　指数関数

`double gsl_sf_exp (`*double* `x``)` [Function]
`int gsl_sf_exp_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

GSL で標準的な二つの関数呼び出し方、つまり数学関数風の呼び出し方およびエラーチェックを行う呼び出し方で、指数関数 (exponential function) $\exp(x)$ の値を計算する。

`int gsl_sf_exp_e10_e (`*double* `x`, *gsl_sf_result_e10* * `result``)` [Function]

`gsl_sf_result_e10` 型を使ってより広い範囲の指数関数値 $\exp(x)$ を計算して返す。関数値 $\exp(x)$ が非常に大きく `double` の表現範囲を超えてオーバーフローするような場合に有用である。

`double gsl_sf_exp_mult (`*double* `x`, *double* `y`)` [Function]
`int gsl_sf_exp_mult_e (`*double* `x`, *double* `y`, *gsl_sf_result* * `result`)` [Function]

x の指数と y との積、$y\exp(x)$ を返す。

`int gsl_sf_exp_mult_e10_e (`*const double* `x`, *const double* `y`, *gsl_sf_result_e10* * `result`)` [Function]

`gsl_sf_result_e10` 型で値を返すことにより、より広い範囲で積 $y\exp(x)$ の値を計算できる。

7.16.2　相対指数関数

`double gsl_sf_expm1 (`*double* `x`)` [Function]
`int gsl_sf_expm1_e (`*double* `x`, *gsl_sf_result* * `result`)` [Function]

x の値が小さいときに精度がよい計算法で $\exp(x) - 1$ の値を計算する。

`double gsl_sf_exprel (`*double* `x`)` [Function]
`int gsl_sf_exprel_e (`*double* `x`, *gsl_sf_result* * `result`)` [Function]

x の値が小さいときに精度がよい計算法で $(\exp(x) - 1)/x$ の値を計算する。$(\exp(x) - 1)/x = 1 + x/2 + x^2/(2*3) + x^3/(2*3*4) + \ldots$ と展開できることを利用する。

`double gsl_sf_exprel_2 (`*double* `x`)` [Function]
`int gsl_sf_exprel_2_e (`*double* `x`, *gsl_sf_result* * `result`)` [Function]

x の値が小さいときに精度がよい計算法で $2(\exp(x) - 1 - x)/x^2$ の値を計算する。$2(\exp(x) - 1 - x)/x^2 = 1 + x/3 + x^2/(3*4) + x^3/(3*4*5) + \ldots$ と展開できることを利用する。

`double gsl_sf_exprel_n (`*double* `x`)` [Function]
`int gsl_sf_exprel_n_e (`*double* `x`, *gsl_sf_result* * `result`)` [Function]

gsl_sf_exprel や gsl_sf_exprel_2 を n 次に拡張し、N 次の相対指数関数 (relative exponential function) の値を計算する。N 次の相対指数関数は以下で与えられる。

$$\begin{aligned}\mathrm{exprel}_N(x) &= N!/x^N \left(\exp(x) - \sum_{k=0}^{N-1} x^k/k!\right) \\ &= 1 + x/(N+1) + x^2/((N+1)(N+2)) + \ldots \\ &= {}_1F_1(1, 1+N, x)\end{aligned}$$

7.16.3 誤差推定を行う指数関数の計算

int gsl_sf_exp_err_e (*double* x, *double* dx, *gsl_sf_result* * result) [Function]

絶対誤差 dx を含む x の指数を計算する。

int gsl_sf_exp_err_e10_e (*double* x, *double* dx, *gsl_sf_result_e10* * result) [Function]

gsl_sf_result_e10 型で値を返すことにより、より広い範囲で、絶対誤差 dx を含む x の指数を計算する。

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

絶対誤差 dx、dy をそれぞれ含む x と y から、積 $y\exp(x)$ を計算する。

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

gsl_sf_result_e10 型で値を返すことにより、より広い範囲で、絶対誤差 dx、dy をそれぞれ含む x と y から、積 $y\exp(x)$ を計算する。

7.17 指数積分

指数積分 (exponential integral) についての詳細はアブラモウィッツ＆ステグンの第 5 章に説明されている。この節の関数はヘッダファイル 'gsl_sf_expint.h' で宣言されている。

7.17.1 指数積分

double gsl_sf_expint_E1 (*double* x) [Function]
int gsl_sf_expint_E1_e (*double* x, *gsl_sf_result* * result) [Function]

以下に示す指数積分 (exponential integral) $E_1(x)$ を計算する。

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

`double gsl_sf_expint_E2 (double x)` [Function]
`int gsl_sf_expint_E2_e (double x, gsl_sf_result * result)` [Function]

以下に示す二次の指数積分 $E_2(x)$ を計算する。

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

`double gsl_sf_expint_En (int n, double x)` [Function]
`int gsl_sf_expint_En_e (int n, double x, gsl_sf_result * result)` [Function]

以下に示す n 次の指数積分 $E_n(x)$ を計算する。

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

7.17.2 $Ei(x)$

`double gsl_sf_expint_Ei (double x)` [Function]
`int gsl_sf_expint_Ei_e (double x, gsl_sf_result * result)` [Function]

以下に示す指数積分 (積分指数関数) $Ei(x)$ を計算する。

$$Ei(x) := -PV\left(\int_{-x}^\infty \exp(-t)/t dt\right)$$

ここで PV はこの積分の主値を表す。

7.17.3 双曲線積分

`double gsl_sf_Shi (double x)` [Function]
`int gsl_sf_Shi_e (double x, gsl_sf_result * result)` [Function]

双曲線正弦積分 $Shi(x) = \int_0^x \sinh(t)/t dt$ を計算する。

`double gsl_sf_Chi (double x)` [Function]
`int gsl_sf_Chi_e (double x, gsl_sf_result * result)` [Function]

双曲線余弦積分 $Chi(x) := Re[\gamma_E + \log(x) + \int_0^x (\cosh(t) - 1)/t dt]$ を計算する。γ_E は
オイラー定数である (マクロ `M_EULER` で参照できる)。

7.17.4 $Ei_3(x)$

`double gsl_sf_expint_3 (double x)` [Function]
`int gsl_sf_expint_3_e (double x, gsl_sf_result * result)` [Function]

$x \geq 0$ について三次の指数積分 $Ei_3(x) = \int_0^x \exp(-t^3) dt$ を計算する。

7.17.5 三角積分

`double gsl_sf_Si (`*double* `x``)` [Function]
`int gsl_sf_Si_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

正弦積分 (sine integral) $Si(x) = \int_0^x \sin(t)/t dt$ を計算する。

`double gsl_sf_Ci (`*double* `x``)` [Function]
`int gsl_sf_Ci_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

$x > 0$ について余弦積分 (cosine integral) $Ci(x) = -\int_x^\infty \cos(t)/t dt$ を計算する。

7.17.6 逆接弦関数の積分

`double gsl_sf_atanint (`*double* `x``)` [Function]
`int gsl_sf_atanint_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

$AtanInt(x) = \int_0^x \arctan(t)/t dt$ で定義される逆接弦関数積分 (arctangent integral) を計算する。

7.18 フェルミ=ディラック積分

この節の関数はヘッダファイル '`gsl_sf_fermi_dirac.h`' で宣言されている。

7.18.1 完全フェルミ=ディラック積分

完全フェルミ=ディラック積分 (complete Fermi-Dirac integral) $F_j(x)$ は以下で与えられる。

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

文献によっては、規格化の係数を除いて定義されていることもある。

`double gsl_sf_fermi_dirac_m1 (`*double* `x``)` [Function]
`int gsl_sf_fermi_dirac_m1_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

$F_{-1}(x) = e^x/(1+e^x)$ で与えられる、-1 次の完全フェルミ=ディラック積分を計算する。

`double gsl_sf_fermi_dirac_0 (`*double* `x``)` [Function]
`int gsl_sf_fermi_dirac_0_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

$F_0(x) = \ln(1+e^x)$ で与えられる、0 次の完全フェルミ=ディラック積分を計算する。

`double gsl_sf_fermi_dirac_1 (`*double* `x``)` [Function]
`int gsl_sf_fermi_dirac_1_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

$F_1(x) = \int_0^\infty (t/(\exp(t-x)+1)dt$ で与えられる、1 次の完全フェルミ=ディラック積分を計算する。

`double gsl_sf_fermi_dirac_2 (double x)` [Function]
`int gsl_sf_fermi_dirac_2_e (double x, gsl_sf_result * result)` [Function]

$F_2(x) = (1/2)\int_0^\infty (t^2/(\exp(t-x)+1)dt$ で与えられる、2 次の完全フェルミ=ディラック積分を計算する。

`double gsl_sf_fermi_dirac_int (int j, double x)` [Function]
`int gsl_sf_fermi_dirac_int_e (int j, double x, gsl_sf_result * result)` [Function]

$F_j(x) = (1/\Gamma(j+1))\int_0^\infty (t^j/(\exp(t-x)+1)dt$ で与えられる、j 次の完全フェルミ=ディラック積分を計算する。

`double gsl_sf_fermi_dirac_mhalf (double x)` [Function]
`int gsl_sf_fermi_dirac_mhalf_e (double x, gsl_sf_result * result)` [Function]

完全フェルミ=ディラック積分 $F_{-1/2}(x)$ を計算する。

`double gsl_sf_fermi_dirac_half (double x)` [Function]
`int gsl_sf_fermi_dirac_half_e (double x, gsl_sf_result * result)` [Function]

完全フェルミ=ディラック積分 $F_{1/2}(x)$ を計算する。

`double gsl_sf_fermi_dirac_3half (double x)` [Function]
`int gsl_sf_fermi_dirac_3half_e (double x, gsl_sf_result * result)` [Function]

完全フェルミ=ディラック積分 $F_{3/2}(x)$ を計算する。

7.18.2　不完全フェルミ=ディラック積分

不完全フェルミ-ディラック積分 (incomplete Fermi-Dirac integral) $F_j(x,b)$ は以下で与えられる。

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

`double gsl_sf_fermi_dirac_inc_0 (double x, double b)` [Function]
`int gsl_sf_fermi_dirac_inc_0_e (double x, double b, gsl_sf_result * result)` [Function]

0 次の不完全フェルミ=ディラック積分 $F_0(x,b) = \ln(1+e^{b-x}) - (b-x)$ を計算する。

7.19　ガンマ関数とベータ関数

この節の関数はヘッダファイル ‘gsl_gamma.h’ で宣言されている。

7.19.1 ガンマ関数

ガンマ関数 (gamma function) は以下の積分で定義される。

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

n が正の整数のとき、Γ 関数と n の階乗には $\Gamma(n) = (n-1)!$ という関係がある。ガンマ関数についての詳細はアブラモウィッツ＆ステグンの第 6 章を参照のこと。

`double gsl_sf_gamma (`*double* `x``)` [Function]
`int gsl_sf_gamma_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

負または 0 でない整数 x についてガンマ関数値 $\Gamma(x)$ を計算する。計算には実数ランチョス法 (real Lanczos method) を使う。マクロ `GSL_SF_GAMMA_XMAX` で、$\Gamma(x)$ がオーバーフローせずにすむ x の最大値を参照できる。その値は 171.0 である。

`double gsl_sf_lngamma (`*double* `x``)` [Function]
`int gsl_sf_lngamma_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

負または 0 でない整数 x についてガンマ関数の対数値 $\log(\Gamma(x))$ を計算する。$x < 0$ の場合は $\log(\Gamma(x))$ の実部を返すため、$\log(|\Gamma(x)|)$ と同じである。実数ランチョス法を使う。

`int gsl_sf_lngamma_sgn_e (`*double* `x`, *gsl_sf_result* * `result_lg`, *double* * `sgn``)` [Function]

負または 0 でない整数 x についてガンマ関数の符号とその対数値を実数ランチョス法で計算する。返り値から $\Gamma(x) = sgn * \exp(resultlg\text{->})$ として Γ 関数の値が得られる。

`double gsl_sf_gammastar (`*double* `x``)` [Function]
`int gsl_sf_gammastar_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

正規化されたガンマ関数 (regulated gamma function) の値 $\Gamma^*(x)$ を $x > 0$ で返す。正規化されたガンマ関数は以下で与えられる。

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

テム (Nico M. Temme) の文献に有用な情報がある。

`double gsl_sf_gammainv (`*double* `x``)` [Function]
`int gsl_sf_gammainv_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

ガンマ関数の逆数 $1/\Gamma(x)$ を実数ランチョス法を使って計算する。

`int gsl_sf_lngamma_complex_e (`*double* `zr`, *double* `zi`, *gsl_sf_result* * `lnr`, *gsl_sf_result* * `arg``)` [Function]

負の整数または 0 でない複素数 z に対して、複素数 $z = z_r + iz_i$ のガンマ関数値の対数 $\log(\Gamma(z))$ を複素数ランチョス法で計算する。返される計算結果は $lnr = \log|\Gamma(z)|$ および $(-\pi, \pi]$ の範囲の $arg = \arg(\Gamma(z))$ である。$|z|$ が非常に大きな場合は、範囲を $(-\pi, \pi]$ に制限しているために大きな丸め誤差が生じ、位相 (arg) が計算できないことがある。その場合は GSL_ELOSS が返される。この場合、絶対値 (lnr) の精度は失われない。

7.19.2　階乗

階乗 (factorial) の値は、非負整数 n に対してはガンマ関数を使って $n! = \Gamma(n+1)$ で計算できるが、小さな n に対しては GSL であらかじめ表を用意しており、それを使う以下の関数が高速である。

`double gsl_sf_fact (`*unsigned int* **`n`**`)` [Function]
`int gsl_sf_fact_e (`*unsigned int* **`n`**, *gsl_sf_result* * **`result`**`)` [Function]

n の階乗 $n!$ を計算する。その値は $n! = \Gamma(n+1)$ である。$n!$ がオーバーフローを起こさないような n の最大値は 170 であり、その値はマクロ GSL_SF_FACT_NMAX に定義されている。

`double gsl_sf_doublefact (`*unsigned int* **`n`**`)` [Function]
`int gsl_sf_doublefact_e (`*unsigned int* **`n`**, *gsl_sf_result* * **`result`**`)` [Function]

n の二重階乗 (double factorial) $n!! = n(n-2)(n-4)\ldots$ の値を計算する。$n!!$ がオーバーフローを起こさないような n の最大値は 297 であり、その値はマクロ GSL_SF_DOUBLEFACT_NMAX に定義されている。

`double gsl_sf_lnfact (`*unsigned int* **`n`**`)` [Function]
`int gsl_sf_lnfact_e (`*unsigned int* **`n`**, *gsl_sf_result* * **`result`**`)` [Function]

n の階乗の対数 $\log(n!)$ を計算する。$n < 170$ のときは gsl_sf_lngamma を使って $\ln(\Gamma(n+1))$ を計算するよりも高速だが、n が大きくなると遅くなる。

`double gsl_sf_lndoublefact (`*unsigned int* **`n`**`)` [Function]
`int gsl_sf_lndoublefact_e (`*unsigned int* **`n`**, *gsl_sf_result* * **`result`**`)` [Function]

$n!!$ の対数 $\log(n!!)$ を計算する。

`double gsl_sf_choose (`*unsigned int* **`n`**, *unsigned int* **`m`**`)` [Function]
`int gsl_sf_choose_e (`*unsigned int* **`n`**, *unsigned int* **`m`**, *gsl_sf_result* * **`result`**`)` [Function]

二項係数 (binomial coefficient) ${}_n\mathrm{C}_r = n!/(m!(n-m)!)$ を計算する。

`double gsl_sf_lnchoose (`*unsigned int* **`n`**, *unsigned int* **`m`**`)` [Function]
`int gsl_sf_lnchoose_e (`*unsigned int* **`n`**, *unsigned int* **`m`**, *gsl_sf_result* * **`result`**`)` [Function]

二項係数の対数を計算する。これは $\log(n!) - \log(m!) - \log((n-m)!)$ と同じである。

`double gsl_sf_taylorcoeff (int n, double x)` [Function]
`int gsl_sf_taylorcoeff_e (int n, double x, gsl_sf_result * result)` [Function]

テイラー係数 (Taylor coefficient) $x^n/n!$ を $x \geq 0, n \geq 0$ について計算する。

7.19.3 ポッホハンマー記号

`double gsl_sf_poch (double a, double x)` [Function]
`int gsl_sf_poch_e (double a, double x, gsl_sf_result * result)` [Function]

負の整数または 0 でない a と $a+x$ に対して、ポッホハンマー記号 (Pochhammer symbol) $(a)_x = \Gamma(a+x)/\Gamma(a)$ の値を計算する。ポッホハンマー記号はアペル記号 (Apell symbol) のとも呼ばれ (a, x) と表記されることもある。

`double gsl_sf_lnpoch (double a, double x)` [Function]
`int gsl_sf_lnpoch_e (double a, double x, gsl_sf_result * result)` [Function]

ポッホハンマー記号の対数 $\log((a)_x) = \log(\Gamma(a+x)/\Gamma(a))$ を計算する。$a > 0$ かつ $a + x > 0$ である。

`int gsl_sf_lnpoch_sgn_e (double a, double x, gsl_sf_result * result, double * sgn)` [Function]

ポッホハンマー記号の値の対数値と、その符号を計算する。計算されるのは *result*->= $\log(|(a)_x|)$ と *sgn* = $\mathrm{sgn}((a)_x)$ である。ここで $(a)_x = \Gamma(a+x)/\Gamma(a)$ であり、a と $a+x$ はともに 0 または負の整数ではない。

`double gsl_sf_pochrel (double a, double x)` [Function]
`int gsl_sf_pochrel_e (double a, double x, gsl_sf_result * result)` [Function]

相対ポッホハンマー記号 $((a)_x - 1)/x$ を計算する。ここで $(a)_x = \Gamma(a+x)/\Gamma(a)$ である。

7.19.4 不完全ガンマ関数

`double gsl_sf_gamma_inc (double a, double x)` [Function]
`int gsl_sf_gamma_inc_e (double a, double x, gsl_sf_result * result)` [Function]

規格化されていない不完全ガンマ関数 (unnormalized incomplete gamma function) $\Gamma(a,x) = \int_x^\infty t^{a-1}\exp(-t)dt$ を計算する。a は実数 で $x \geq 0$ である。

`double gsl_sf_gamma_inc_Q (double a, double x)` [Function]
`int gsl_sf_gamma_inc_Q_e (double a, double x, gsl_sf_result * result)` [Function]

正規化された不完全ガンマ関数 (normalized incomplete gamma function) $Q(a,x) = 1/\Gamma(a)\int_x^\infty t^{a-1}\exp(-t)dt$ を計算する。$a > 0$ で $x \geq 0$ である。

`double gsl_sf_gamma_inc_P (`*double* `a`, *double* `x``)` [Function]
`int gsl_sf_gamma_inc_P_e (`*double* `a`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

正規化された不完全補ガンマ関数 (complementary normalized incomplete gamma function) $P(a,x) = 1 - Q(a,x) = 1/\Gamma(a)\int_0^x t^{a-1}\exp(-t)dt$ を計算する。$a > 0$ で $x \geq 0$ である。アブラモウィッツ＆ステグンでは $P(a,x)$ を不完全ガンマ関数と呼んでいる (6.5 節)。

7.19.5　ベータ関数

`double gsl_sf_beta (`*double* `a`, *double* `b``)` [Function]
`int gsl_sf_beta_e (`*double* `a`, *double* `b`, *gsl_sf_result* * `result``)` [Function]

a と b が負の整数でないとき、完全ベータ関数 (complete beta function) $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$ を計算する。

`double gsl_sf_lnbeta (`*double* `a`, *double* `b``)` [Function]
`int gsl_sf_lnbeta_e (`*double* `a`, *double* `b`, *gsl_sf_result* * `result``)` [Function]

a と b が負の整数でないとき、ベータ関数の対数値 $\log(B(a,b))$ を計算する。

7.19.6　不完全ベータ関数

`double gsl_sf_beta_inc (`*double* `a`, *double* `b`, *double* `x``)` [Function]
`int gsl_sf_beta_e_inc (`*double* `a`, *double* `b`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

規格化された不完全ベータ関数 (normalized incomplete beta function) $I_x(a,b) = B_x(a,b)/B(a,b)$ の値を計算する。ここで、$B_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1}dt$ で、$0 \leq x \leq 1$ である。$a > 0$ かつ $b > 0$ のときは連分数に展開して計算される。そうでないときは、$I_x(a,b,x) = (1/a)x^a{}_2F_1(a,1-b,a+1,x)/B(a,b)$ という関係を使って計算される ($_2F_1$ はガウスの超幾何関数、第 7.21 節参照)。

7.20　ゲーゲンバウア関数

ゲーゲンバウア多項式 (Gegenbauer polynomial) はアブラモウィッツ＆ステグンの第 22 章に定義されている。これは超球多項式 (Ultraspherical polynomial) とも呼ばれる。この節の関数はヘッダファイル 'gsl_sf_gegenbauer.h' で宣言されている。

`double gsl_sf_gegenpoly_1 (`*double* `lambda`, *double* `x``)` [Function]
`double gsl_sf_gegenpoly_2 (`*double* `lambda`, *double* `x``)` [Function]

`double gsl_sf_gegenpoly_3 (double lambda, double x)` [Function]
`int gsl_sf_gegenpoly_1_e (double lambda, double x, gsl_sf_result * result)` [Function]
`int gsl_sf_gegenpoly_2_e (double lambda, double x, gsl_sf_result * result)` [Function]
`int gsl_sf_gegenpoly_3_e (double lambda, double x, gsl_sf_result * result)` [Function]

ゲーゲンバウア多項式 $C_n^{(\lambda)}(x)$ の値を、$n = 1, 2, 3$ に関して陽関数形式を使って計算する。陽に記述される計算法で (漸化式ではなく直接にその値を) 計算する。

`double gsl_sf_gegenpoly_n (int n, double lambda, double x)` [Function]
`int gsl_sf_gegenpoly_n_e (int n, double lambda, double x, gsl_sf_result * result)` [Function]

指定された n、$lambda$、x に対してゲーゲンバウア多項式 $C_n^{(\lambda)}(x)$ の値を (条件によっては漸化式で) 計算する。$\lambda > -1/2$、$n \geq 0$ である。

`int gsl_sf_gegenpoly_array (int nmax, double lambda, double x, double result_array[])` [Function]

ゲーゲンバウア多項式 $C_n^{(\lambda)}(x)$ の値を、$n = 0, 1, 2, \ldots, nmax$ に関して計算し、配列に入れて返す。$\lambda > -1/2$、$n \geq 0$ である。

7.21 超幾何関数

超幾何関数 (hypergeometric function) はアブラモウィッツ&ステグンの第 13 章および第 15 章に定義されている。この節の関数は 'gsl_sf_hyperg.h' で宣言されている。
`double gsl_sf_hyperg_0F1 (double c, double x)` [Function]
`int gsl_sf_hyperg_0F1_e (double c, double x, gsl_sf_result * result)` [Function]

超幾何関数 ${}_0F_1(c, x)$ の値を計算する。

`double gsl_sf_hyperg_1F1_int (int n, int m, double x)` [Function]
`int gsl_sf_hyperg_1F1_int_e (int n, int m, double x, gsl_sf_result * result)` [Function]

合流型超幾何関数 (confluent hypergeometric function) ${}_1F_1(m, n, x) = M(m, n, x)$ の値を、整数 m、n に対して計算する。

`double gsl_sf_hyperg_1F1 (double a, double b, double x)` [Function]
`int gsl_sf_hyperg_1F1_e (double a, double b, double x, gsl_sf_result * result)` [Function]

合流型超幾何関数 ${}_1F_1(a,b,x) = M(a,b,x)$ の値を、一般の実数 a、b に対して計算する。

`double gsl_sf_hyperg_U_int (`*int* **`n`**, *int* **`m`**, *double* **`x`**`)` [Function]
`int gsl_sf_hyperg_U_int_e (`*int* **`n`**, *int* **`m`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

合流型超幾何関数 $U(m,n,x)$ の値を、整数 m、n に対して計算する。

`int gsl_sf_hyperg_U_int_e10_e (`*int* **`n`**, *int* **`m`**, *double* **`x`**, *gsl_sf_result_e10* * **`result`**`)` [Function]

合流型超幾何関数 $U(m,n,x)$ の値を、整数 m、n に対して計算する。`gsl_sf_result_e10` 型を使ってより広い範囲で返すことができる。

`double gsl_sf_hyperg_U (`*double* **`a`**, *double* **`b`**, *double* **`x`**`)` [Function]
`int gsl_sf_hyperg_U_e (`*double* **`a`**, *double* **`b`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

合流型超幾何関数 $U(a,b,x)$ の値を一般の実数 a、b に対して計算する。

`int gsl_sf_hyperg_U_e10_e (`*double* **`a`**, *double* **`b`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

合流型超幾何関数 $U(a,b,x)$ の値を計算する。`gsl_sf_result_e10` 型を使ってより広い範囲で返すことができる。

`double gsl_sf_hyperg_2F1 (`*double* **`a`**, *double* **`b`**, *double* **`c`**, *double* **`x`**`)` [Function]
`int gsl_sf_hyperg_2F1_e (`*double* **`a`**, *double* **`b`**, *double* **`c`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

ガウスの超幾何関数 ${}_2F_1(a,b,c,x)$ の値を $|x| < 1$ のとき計算する。

引数 (a,b,c,x) が特異点を生じそうな値のときは、超幾何級数の収束が非常に遅くなることがあり、その場合には `GSL_EMAXITER` を返すことがある。整数 m に対して $x = 1$、$c - a - b = m$ となる領域の近くがそういった値になる。

`double gsl_sf_hyperg_2F1_conj (`*double* **`aR`**, *double* **`aI`**, *double* **`c`**, *double* **`x`**`)` [Function]
`int gsl_sf_hyperg_2F1_conj_e (`*double* **`aR`**, *double* **`aI`**, *double* **`c`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

複素数パラメータを持つガウスの超幾何関数 ${}_2F_1(a_R + ia_I, a_R - ia_I, c, x)$ の値を計算する。$|x| < 1$ である。

`double gsl_sf_hyperg_2F1_renorm (`*double* **`a`**, *double* **`b`**, *double* **`c`**, *double* **`x`**`)` [Function]
`int gsl_sf_hyperg_2F1_renorm_e (`*double* **`a`**, *double* **`b`**, *double* **`c`**, *double* **`x`**, *gsl_sf_result* * **`result`**`)` [Function]

くりこまれた (renormalized) ガウスの超幾何関数 ${}_2F_1(a,b,c,x)/\Gamma(c)$ の値を計算する。$|x|<1$ である。

`double gsl_sf_hyperg_2F1_conj_renorm (`*double* `aR`, *double* `aI`, *double* `c`, *double* `x``)` [Function]
`int gsl_sf_hyperg_2F1_conj_renorm_e (`*double* `aR`, *double* `aI`, *double* `c`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

複素数パラメータを持つ、くりこまれたガウスの超幾何関数 ${}_2F_1(a_R+ia_I, a_R-ia_I, c, x)/\Gamma(c)$ の値を計算する。$|x|<1$ である。

`double gsl_sf_hyperg_2F0 (`*double* `a`, *double* `b`, *double* `x``)` [Function]
`int gsl_sf_hyperg_2F0_e (`*double* `a`, *double* `b`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

超幾何関数 ${}_2F_0(a,b,x)$ の値を計算する。この級数は一般には発散するが、$x<0$ では ${}_2F_0(a,b,x) = (-1/x)^a U(a, 1+a-b, -1/x)$ となる。

7.22 ラゲール多項式

一般化ラゲール多項式 (generalized Laguerre polynomial) は合流型超幾何関数 (confluent hypergeometric function) の一種で、$L_n^a(x) = ((a+1)_n/n!){}_1F_1(-n, a+1, x)$ として定義されラゲールの随伴多項式 (associated Laguerre polynomial) とも呼ばれる。これらは元のラゲール多項式 $L_n(x)$ ($L_n^0(x) = L_n(x)$ である) および $L_n^k(x) = (-1)^k (d^k/dx^k) L_{(n+k)}(x)$ から得られる。詳細はアブラモウィッツ&ステグンの第 22 章を参照のこと。

この節の関数はヘッダファイル '`gsl_sf_laguerre.h`' で宣言されている。

`double gsl_sf_laguerre_1 (`*double* `a`, *double* `x``)` [Function]
`double gsl_sf_laguerre_2 (`*double* `a`, *double* `x``)` [Function]
`double gsl_sf_laguerre_3 (`*double* `a`, *double* `x``)` [Function]
`int gsl_sf_laguerre_1_e (`*double* `a`, *double* `x`, *gsl_sf_result* * `result``)` [Function]
`int gsl_sf_laguerre_2_e (`*double* `a`, *double* `x`, *gsl_sf_result* * `result``)` [Function]
`int gsl_sf_laguerre_3_e (`*double* `a`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

ラゲール多項式 $L_1^a(x)$、$L_2^a(x)$、$L_3^a(x)$ の値を陽に記述される計算法で (漸近公式ではなく直接にその値を) 計算する。

`double gsl_sf_laguerre_n (`*int* `n`, *double* `a`, *double* `x``)` [Function]
`int gsl_sf_laguerre_n_e (`*int* `n`, *double* `a`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

一般化ラゲール多項式 $L_n^a(x)$ を $a > -1$、$n \geq 0$ について (漸近公式で) 計算する。

7.23　ランベルトの W 関数

ランベルトの W 関数 (Lambert's W function) $W(x)$ は、$W(x)\exp(W(x)) = x$ の解として定義される。この関数には $x < 0$ で複数の分枝 (multiple branch) があるが、実数の分枝は二つだけである。ここでは、$x < 0$ で $W > -1$ となる $W_0(x)$ を主枝 (principle branch)、$x < 0$ で $W < -1$ となるもう一方を $W_{-1}(x)$ とする。

ランベルトの関数は 'gsl_sf_lambert.h' で宣言されている。

`double gsl_sf_lambert_W0 (`*double* `x``)` [Function]
`int gsl_sf_lambert_W0_e (`*double* `x, gsl_sf_result *`result`)` [Function]

ランベルトの W 関数の主枝 $W_0(x)$ の値を計算する。

`double gsl_sf_lambert_Wm1 (`*double* `x``)` [Function]
`int gsl_sf_lambert_Wm1_e (`*double* `x, gsl_sf_result *`result`)` [Function]

ランベルトの W 関数の主枝でない実数の分枝 $W_{-1}(x)$ の値を計算する。

7.24　ルジャンドル関数と球面調和関数

ルジャンドル関数 (Legendre function) とルジャンドル多項式 (Legendre polynomial) についてはアブラモウィッツ＆ステグンの第 8 章に解説されている。

この節の関数はヘッダファイル 'gsl_sf_legendre.h' で宣言されている。

7.24.1　ルジャンドル多項式

`double gsl_sf_legendre_P1 (`*double* `x``)` [Function]
`double gsl_sf_legendre_P2 (`*double* `x``)` [Function]
`double gsl_sf_legendre_P3 (`*double* `x``)` [Function]
`int gsl_sf_legendre_P1_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]
`int gsl_sf_legendre_P2_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]
`int gsl_sf_legendre_P3_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

ルジャンドル多項式 $P_l(x)$ を陽に書き下された関数の形式 (explicit representation) でそれぞれ $l = 1, 2, 3$ について計算する。

`double gsl_sf_legendre_Pl (`*int* `l`, *double* `x``)` [Function]
`int gsl_sf_legendre_Pl_e (`*int* `l`, *double* `x`, *gsl_sf_result* * `result``)` [Function]

指定された値 l、x ($l \geq 0$ かつ $|x| \leq 1$) について、ルジャンドル多項式 $P_l(x)$ を (条件により漸化式または漸近展開で) 計算する。

`int gsl_sf_legendre_Pl_array (`*int* `lmax`, *double* `x`, *double* `result_array []`, *double* `result_deriv_array []``)` [Function]

$l = 0, \ldots, lmax$, $|x| \leq 1$ についてルジャンドル多項式 $P_l(x)$ を計算すると同時に、導関数 $dP_l(x)/dx$ の値も計算する。

`double gsl_sf_legendre_Q0 (double x)` [Function]
`int gsl_sf_legendre_Q0_e (double x, gsl_sf_result * result)` [Function]

$x > -1$、$x \neq 1$ についてルジャンドル関数 $Q_0(x)$ を計算する。

`double gsl_sf_legendre_Q1 (double x)` [Function]
`int gsl_sf_legendre_Q1_e (double x, gsl_sf_result * result)` [Function]

$x > -1$、$x \neq 1$ についてルジャンドル関数 $\mathrm{Q}_1(x)$ を計算する。

`double gsl_sf_legendre_Ql (int l, double x)` [Function]
`int gsl_sf_legendre_Ql_e (int l, double x, gsl_sf_result * result)` [Function]

$x > -1$、$x \neq 1$、$l \geq 0$ についてルジャンドル関数 $Q_l(x)$ を計算する。

7.24.2 ルジャンドル陪関数と球面調和関数

以下の関数はルジャンドル陪関数 (associated Legendre polynomial) $P_l^m(x)$ を計算する。この関数の値は l に関して組み合わせ爆発的に増加するため、l が約 150 程度よりも大きくなるとオーバーフロを起こす。m が小さければ問題はないが、l と m の両方が大きな値になるとオーバーフローが生じる。以下の関数ではオーバーフローの発生を防ぐため、l と m の値が大きすぎると判断される場合には関数値 $P_l^m(x)$ の計算を中止し、`GSL_EOVRFLW` を返す。

球面調和関数 (spherical harmonic) を計算したいときにルジャンドル陪関数をそのまま使うのは避けるべきである。その代わりに、同じような再帰アルゴリズムではあるが、正規化されている関数 `gsl_sf_legendre_sphPlm` を使うべきである。

`double gsl_sf_legendre_Plm (int l, int m, double x)` [Function]
`int gsl_sf_legendre_Plm_e (int l, int m, double x, gsl_sf_result * result)` [Function]

$m \geq 0$、$l \geq m$、$|x| \leq 1$ についてルジャンドル陪関数 $P_l^m(x)$ を計算する。

`int gsl_sf_legendre_Plm_array (int lmax, int m, double x, double result_array [])` [Function]
`int gsl_sf_legendre_Plm_deriv_array (int lmax, int m, double x, double result_array [], double result_deriv_array [])` [Function]

$m \geq 0$、$l = |m|, \ldots, lmax$、$|x| \leq 1$ について、ルジャンドル多項式の値 $P_l^m(x)$ を、またはあわせてその導関数値 $dP_l^m(x)/dx$ を計算 する。

`double gsl_sf_legendre_sphPlm (int l, int m, double x)` [Function]
`int gsl_sf_legendre_sphPlm_e (int l, int m, double x, gsl_sf_result * result)` [Function]

球面調和関数の計算に適した、正規化されルジャンドル陪関数 $\sqrt{(2l+1)/(4\pi)}\sqrt{(l-m)!/(l+m)!}P_l^m(x)$ を計算する。引数は $m \geq 0$、$l \geq m$、$|x| \leq 1$ でなければならない。$P_l^m(x)$ を普通に正規化すると避けられないオーバーフローを、この関数では避けることができる。

`int gsl_sf_legendre_sphPlm_array` (*int* **`lmax`**, *int* **`m`**, *double* **`x`**, *double* **`result_array []`**) [Function]

`int gsl_sf_legendre_sphPlm_deriv_array` (*int* **`lmax`**, *int* **`m`**, *double* **`x`**, *double* **`result_array []`**, *double* **`result_deriv_array []`**) [Function]

$m \geq 0$、$l = |m|, \ldots, lmax$、$|x| \leq 1$ について、正規化されたルジャンドル陪関数 $\sqrt{(2l+1)/(4\pi)}\sqrt{(l-m)!/(l+m)!}P_l^m(x)$ の値、またはそれに加えて導関数値を計算する。

`int gsl_sf_legendre_array_size` (*const int* **`lmax`**, *const int* **`m`**) [Function]

P_l$^m(x)$ の値を配列で計算するときに必要になる配列 *result array []* の大きさ $lmax-m+1$ を返す。`HAVE_INLINE` が定義されているときは、この関数のインライン版が使われる。

7.24.3　円錐関数

円錐関数 (conical function、独 Kugelfunktion) $P^{\mu}_{-(1/2)+i\lambda}$、$Q^{\mu}_{-(1/2)+i\lambda}$ についてはアブラモウィッツ＆ステグンの第 8.12 節に説明されている。

`double gsl_sf_conicalP_half` (*double* **`lambda`**, *double* **`x`**) [Function]

`int gsl_sf_conicalP_half_e` (*double* **`lambda`**, *double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

$x > -1$ について正規化されていない球面円錐関数 (irregular spherical conical function) $P^{1/2}_{-1/2+i\lambda}(x)$ を 計算する。

`double gsl_sf_conicalP_mhalf` (*double* **`lambda`**, *double* **`x`**) [Function]

`int gsl_sf_conicalP_mhalf_e` (*double* **`lambda`**, *double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

$x > -1$ について正規化された球面円錐関数 (regular spherical conical function) $P^{-1/2}_{-1/2+i\lambda}(x)$ を計算する。

`double gsl_sf_conicalP_0` (*double* **`lambda`**, *double* **`x`**) [Function]

`int gsl_sf_conicalP_0_e` (*double* **`lambda`**, *double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

$x > -1$ について円錐関数 $P^{0}_{-1/2+i\lambda}(x)$ を計算する。

`double gsl_sf_conicalP_1` (*double* **`lambda`**, *double* **`x`**) [Function]

`int gsl_sf_conicalP_1_e` (*double* **`lambda`**, *double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

$x > -1$ について円錐関数 $P^1_{-1/2+i\lambda}(x)$ を計算する。

`double gsl_sf_conicalP_sph_reg (int l, double lambda, double x)` [Function]
`int gsl_sf_conicalP_sph_reg_e (int l, double lambda, double x, gsl_sf_result * result)` [Function]

$x > -1$、$l \geq -1$ について正規化された球面円錐関数 $P^{-1/2-l}_{-1/2+i\lambda}(x)$ を計算する。

`double gsl_sf_conicalP_cyl_reg (int m, double lambda, double x)` [Function]
`int gsl_sf_conicalP_cyl_reg_e (int m, double lambda, double x, gsl_sf_result * result)` [Function]

$x > -1$、$l \geq -1$ について正規化された円柱円錐関数 (regular cylindrical conical function) $P^{-m}_{-1/2+i\lambda}(x)$ を計算する。

7.24.4 双曲面上の円形関数

以下の球面関数 (spherical function) はルジャンドル関数の特殊形であり、三次元の双曲面 (3-dimentional hyperbolic space) *H3d* 上のラプラシアン (Laplacian) の正則な固有関数 (regular eigenfunction) である。$\lambda\eta$ の値を固定した $\lambda \to \infty$、$\eta \to 0$ の極限 (flat limit) で球ベッセル関数になることが知られている。
`double gsl_sf_legendre_H3d_0 (double lambda, double eta)` [Function]
`int gsl_sf_legendre_H3d_0_e (double lambda, double eta, gsl_sf_result * result)` [Function]

三次元双曲面上のラプラシアンの 0 次の円形固有関数、

$$L^{H3d}_0(\lambda, \eta) := \frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)}$$

を $\eta \geq 0$ で計算する。極限 (flat limit) では関数値は $L^{H3d}_0(\lambda, \eta) = j_0(\lambda\eta)$ になる。

`double gsl_sf_legendre_H3d_1 (double lambda, double eta)` [Function]
`int gsl_sf_legendre_H3d_1_e (double lambda, double eta, gsl_sf_result * result)` [Function]

三次元双曲面上のラプラシアンの、一次の円形固有関数、

$$L^{H3d}_1(\lambda, \eta) := \frac{1}{\sqrt{\lambda^2 + 1}} \left(\frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)} \right) (\coth(\eta) - \lambda \cot(\lambda\eta))$$

を $\eta \geq 0$ で計算する。極限では関数値は $L^{H3d}_1(\lambda, \eta) = j_1(\lambda\eta)$ になる。

`double gsl_sf_legendre_H3d (int l, double lambda, double eta)` [Function]
`int gsl_sf_legendre_H3d_e (int l, double lambda, double eta, gsl_sf_result * result)` [Function]

三次元の双曲面上のラプラシアンの、l 次の円形固有関数を $\eta \geq 0$、$l \geq 0$ で計算する。
極限では関数値は $L_l^{H3d}(\lambda, \eta) = j_l(\lambda\eta)$ になる。

`int gsl_sf_legendre_H3d_array (`*int* **`lmax`**, *double* **`lambda`**, *double* **`eta`**, *double* ***`result_array []`***`)` [Function]

$0 \leq l \leq lmax$ での円形固有関数 $L_l^{H3d}(\lambda, \eta)$ の値を計算する。

7.25　対数関連の関数

対数関数 (logarithm function) の性質などについてはアブラモウィッツ＆ステグンの第 4 章にある。この節の関数はヘッダファイル '`gsl_sf_log.h`' で宣言されている。

`double gsl_sf_log (`*double* **`x`**`)` [Function]
`int gsl_sf_log_e (`*double* **`x`**, *gsl_sf_result* * ***`result`***`)` [Function]

x の対数値 $\log(x)$ を $x > 0$ で計算する。

`double gsl_sf_log_abs (`*double* **`x`**`)` [Function]
`int gsl_sf_log_abs_e (`*double* **`x`**, *gsl_sf_result* * ***`result`***`)` [Function]

$x \neq 0$ である x の絶対値の対数値 $\log(|x|)$ を計算する。

`int gsl_sf_complex_log_e (`*double* ***`zr`***, *double* ***`zi`***, *gsl_sf_result* * ***`lnr`***, *gsl_sf_result* * ***`theta`***`)` [Function]

複素数 $z = z_r + iz_i$ の対数値を計算する。計算結果は $\exp(lnr + i\theta) = z_r + iz_i$ となる
lnr と *theta* であり、引数に入れて返される。θ の範囲は $[-\pi, \pi]$ である。

`double gsl_sf_log_1plusx (`*double* **`x`**`)` [Function]
`int gsl_sf_log_1plusx_e (`*double* **`x`**, *gsl_sf_result* * ***`result`***`)` [Function]

$x > -1$ である x に対して、その値が小さいときに精度のよいアルゴリズムを使って、
$\log(1 + x)$ を計算する。

`double gsl_sf_log_1plusx_mx (`*double* **`x`**`)` [Function]
`int gsl_sf_log_1plusx_mx_e (`*double* **`x`**, *gsl_sf_result* * ***`result`***`)` [Function]

$x > -1$ である x に対して、その値が小さいときに精度のよいアルゴリズムを使って、
$\log(1 + x) - x$ を計算する。

7.26　マチウ関数

この節の関数は第一種マチウ関数 (angular Mathieu function) および変形マチウ関数 (radial Mathieu function) とその特徴量 (characteristic values) を計算する。マチウ関数は、以下の微分方程式

の解として得られる。

$$\begin{aligned} \frac{d^2y}{dx^2} + (a - 2q\cos 2v)y &= 0 \\ \frac{d^2f}{du^2} - (a - 2q\cosh 2v)f &= 0 \end{aligned}$$

一つ目の方程式はマチウ方程式と呼ばれ、その周期解 (periodic solution) $ce_r(x,q)$、$se_r(x,q)$ (それぞれ偶周期と奇周期) が第一種マチウ関数と呼ばれる。それぞれの解は、特徴量の数列 $a = a_r(q)$ (偶周期) および $a = b_r(q)$ (奇周期) が特定の離散値をとるときにのみ解が存在する。

変形マチウ関数 $Mc_r^{(j)}(z,q)$ および $Ms_r^{(j)}(z,q)$ は、変形マチウ方程式 (Mathieu's modified equation) と呼ばれる二つ目の方程式の解として定義される。変形マチウ関数には第 1 種から第 4 種まである (1 から 4 までの値をとる j で示される)。

詳細はアブラモウィッツ＆ステグン第 20 章 Mathieu Function を参照のこと。この節の関数はヘッダファイル 'gsl_sf_mathieu.h' で宣言されている。

7.26.1 マチウ関数を計算するための作業領域

マチウ関数は配列操作を基本としたアルゴリズムにより逐次、あるいは同時に計算できる。そのために、あからじめ作業領域となる配列を確保しておかねばならない。

gsl_sf_mathieu_workspace * gsl_sf_mathieu_alloc (*size_t* **n**, *double* **qmax**)[Function]

マチウ関数の配列版で使う作業領域を確保して返す。引数の n は何次まで計算するか、*qmax* は q 値をそれぞれ指定する。

void gsl_sf_mathieu_free (*gsl_sf_mathieu_workspace* * **work**) [Function]

確保されていた作業領域 *work* を解放する。

7.26.2 マチウ関数の特徴量

int gsl_sf_mathieu_a (*int* **n**, *double* **q**, *gsl_sf_result* * **result**) [Function]
int gsl_sf_mathieu_b (*int* **n**, *double* **q**, *gsl_sf_result* * **result**) [Function]

マチウ関数 $ce_n(x,q)$ および $se_n(x,q)$ の特徴量 $a_n(q)$、$b_n(q)$ をそれぞれ計算する。

int gsl_sf_mathieu_a_array (*int* **order_min**, *int* **order_max**, *double* **q**, *gsl_sf_mathieu_workspace* * **work**, *double* **result_array[]**) [Function]
int gsl_sf_mathieu_b_array (*int* **order_min**, *int* **order_max**, *double* **q**, *gsl_sf_mathieu_workspace* * **work**, *double* **result_array[]**) [Function]

order_min から *order_max* までの n について、マチウ関数 $ce_n(x,q)$ および $se_n(x,q)$ の特徴量 $a_n(q)$、$b_n(q)$ をそれぞれ計算し、配列 *result_array* に入れて返す。

7.26.3 第一種マチウ関数

int gsl_sf_mathieu_ce (*int* **n**, *double* **q**, *double* **x**, *gsl_sf_result* * **result**) [Function]
int gsl_sf_mathieu_se (*int* **n**, *double* **q**, *double* **x**, *gsl_sf_result* * **result**) [Function]

第一種マチウ関数 $ce_n(x,q)$ および $se_n(x,q)$ をそれぞれ計算する。

int gsl_sf_mathieu_ce_array (*int* **nmin**, *int* **nmax**, *double* **q**, *double* **x**, *gsl_sf_mathieu_workspace* * **work**, *double* **result_array** **[]**) [Function]
int gsl_sf_mathieu_se_array (*int* **nmin**, *int* **nmax**, *double* **q**, *double* **x**, *gsl_sf_mathieu_workspace* * **work**, *double* **result_array** **[]**) [Function]

nmin 以上 *nmax* 以下の n 次について、第一種マチウ関数 $ce_n(x,q)$ および $se_n(x,q)$ をそれぞれ計算し、配列 *result_array* に入れて返す。

7.26.4 変形マチウ関数

int gsl_sf_mathieu_Mc (*int* **j**, *int* **n**, *double* **q**, *double* **x**, *gsl_sf_result* * **result**) [Function]
int gsl_sf_mathieu_Ms (*int* **j**, *int* **n**, *double* **q**, *double* **x**, *gsl_sf_result* * **result**) [Function]

n 次の変形マチウ関数 $Mc_n^{(j)}(q,x)$ および $Ms_n^{(j)}(q,x)$ をそれぞれ計算する。

j の値は 1 か 2 を指定する。$j = 3,4$ の場合の関数値は、$M_n^{(3)} = M_n^{(1)} + iM_n^{(2)}$、$M_n^{(4)} = M_n^{(1)} - iM_n^{(2)}$ として計算できる。ここで $M_n^{(j)} = Mc_n^{(j)}$ または $Ms_n^{(j)}$ である。

int gsl_sf_mathieu_Mc_array (*int* **j**, *int* **nmin**, *int* **nmax**, *double* **q**, *double* **x**, *gsl_sf_mathieu_workspace* * **work**, *double* **result_array** **[]**) [Function]
int gsl_sf_mathieu_Ms_array (*int* **j**, *int* **nmin**, *int* **nmax**, *double* **q**, *double* **x**, *gsl_sf_mathieu_workspace* * **work**, *double* **result_array** **[]**) [Function]

nmin 以上 *nmax* 以下の n 次について、第 j 種の変形マチウ関数の値を計算し、配列 *result_array* に入れて返す。

7.27 べき乗関数

以下の関数は、誤差推定を行うこと以外は gsl_pow_int (4.4 節「小さな整数でのべき乗」参照) と同じである。ヘッダファイル 'gsl_sf_pow_int.h' で宣言されている。

double gsl_sf_pow_int (*double* **x**, *int* **n**) [Function]
int gsl_sf_pow_int_e (*double* **x**, *int* **n**, *gsl_sf_result* * **result**) [Function]

整数 n についてべき乗 x^n を計算する。べき乗は、積の演算が最も少なくなる形に分解して行われる。たとえば x^8 を $((x^2)^2)^2$ の形に分解すると積算が三回ですむ。計算速度を向上するため、オーバーフローやアンダーフローの検知は行わない。

```
#include <gsl/gsl_sf_pow_int.h>
    /* 3.0**12 を計算する */
    double y = gsl_sf_pow_int(3.0, 12);
```

7.28 プサイ (二重ガンマ) 関数

n 次の多重ガンマ関数 (polygamma function) は以下の式で定義される。

$$\psi^{(n)} = \left(\frac{d}{dx}\right)^n \psi(x) = \left(\frac{d}{dx}\right)^{n+1} \log(\Gamma(x))$$

ここで $\psi(x) = \Gamma'(x)/\Gamma(x)$ は二重ガンマ関数 (digamma function) と呼ばれている。この節の関数はヘッダファイル 'gsl_sf_psi.h' で宣言されている。

7.28.1 二重ガンマ関数

`double gsl_sf_psi_int (int n)` [Function]
`int gsl_sf_psi_int_e (int n, gsl_sf_result * result)` [Function]

二重ガンマ関数 $\psi(n)$ を正の整数 n に対して計算する。二重ガンマ関数はプサイ関数 (psi function) とも呼ばれる。

`double gsl_sf_psi (double x)` [Function]
`int gsl_sf_psi_e (double x, gsl_sf_result * result)` [Function]

二重ガンマ関数 $\psi(x)$ を実数 x (ただし $x \neq 0$) に対して計算する。

`double gsl_sf_psi_1piy (double y)` [Function]
`int gsl_sf_psi_1piy_e (double y, gsl_sf_result * result)` [Function]

直線 $1 + iy$ 上の二重ガンマ関数の実部 $\mathrm{Re}[\psi(1 + iy)]$ を計算する。

7.28.2 三重ガンマ関数

`double gsl_sf_psi_1_int (int n)` [Function]
`int gsl_sf_psi_1_int_e (int n, gsl_sf_result * result)` [Function]

三重ガンマ関数 (trigamma function) $\psi'(n)$ を正の整数 n に対して計算する。

`double gsl_sf_psi_1 (double x)` [Function]
`int gsl_sf_psi_1_e (double x, gsl_sf_result * result)` [Function]

三重ガンマ関数 $\psi'(x)$ を実数 x に対して計算する。

7.28.3　多重ガンマ関数

`double gsl_sf_psi_n` (*int* **`n`**, *double* **`x`**) [Function]
`int gsl_sf_psi_n_e` (*int* **`n`**, *double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

多重ガンマ関数 $\psi^{(n)}(x)$ を $n \geq 0$ および $x > 0$ に対して計算する。

7.29　シンクロトロン関数

シンクロトロン関数 (synchrotron function) は、シンクロトロン放射 (synchrotron radiation) のスペクトルを計算するときに用いられ、第二種変形ベッセル関数 K_j を使って定義される。この節の関数はヘッダファイル '`gsl_sf_synchrotron.h`' で宣言されている。

`double gsl_sf_synchrotron_1` (*double* **`x`**) [Function]
`int gsl_sf_synchrotron_1_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

第一シンクロトロン関数 (first synchrotron function) $x \int_x^\infty K_{5/3}(t)dt$ を $x \geq 0$ に対して計算する。

`double gsl_sf_synchrotron_2` (*double* **`x`**) [Function]
`int gsl_sf_synchrotron_2_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

第二シンクロトロン関数 (second synchrotron function) $xK_{2/3}(x)dt$ を $x \geq 0$ に対して計算する。

7.30　輸送関数

輸送関数 (transport function) $J(n,x)$ は $J(n,x) := \int_0^x t^n e^t/(e^t-1)^2 dt$ という積分で定義され、デバイの比熱式 (Debye model, $n = 4$) や低温における金属の抵抗率のブロッホ-グリュナイゼンの式 (Bloch-Grüneisen relation of electrical resistivity of metals, $n = 5$) など、物理現象の記述にしばしば現れる (Allan J. MacLeod, "The numerical computation of transport integrals", *Computer Physics Communications*,**69**, pp. 229–234, 1992)。関数はヘッダファイル '`gsl_sf_transport.h`' で宣言されている。

`double gsl_sf_transport_2` (*double* **`x`**) [Function]
`int gsl_sf_transport_2_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

輸送関数 $J(2,x)$ を計算する。

`double gsl_sf_transport_3` (*double* **`x`**) [Function]
`int gsl_sf_transport_3_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

輸送関数 $J(3,x)$ を計算する。

`double gsl_sf_transport_4` (*double* **`x`**) [Function]
`int gsl_sf_transport_4_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

輸送関数 $J(4,x)$ を計算する。

`double gsl_sf_transport_5 (`*double* `x``)` [Function]
`int gsl_sf_transport_5_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

輸送関数 $J(5,x)$ を計算する。

7.31 三角関数

異なるプラットフォーム間での整合性や誤差推定の信頼性を確保するため、GSL では独自に三角関数 (trigonometric function) を実装している。関数はヘッダファイル '`gsl_sf_trig.h`' で宣言されている。

7.31.1 円の三角関数

`double gsl_sf_sin (`*double* `x``)` [Function]
`int gsl_sf_sin_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

正弦関数 (sine function) $\sin(x)$ の値を計算する。

`double gsl_sf_cos (`*double* `x``)` [Function]
`int gsl_sf_cos_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

余弦関数 (cosine function) $\cos(x)$ の値を計算する。

`double gsl_sf_hypot (`*double* `x``)` [Function]
`int gsl_sf_hypot_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

オーバーフローやアンダーフローをできるだけ避けて斜辺関数 (hypotenuse function) $\sqrt{x^2+y^2}$ の値を計算する。

`double gsl_sf_sinc (`*double* `x``)` [Function]
`int gsl_sf_sinc_e (`*double* `x`, *gsl_sf_result* * `result``)` [Function]

いかなる x の値についても、正規化されたシンク関数 (sinc function, $\mathrm{sinc}(x) = \sin(\pi x)/(\pi x)$) の値を計算する。$x$ の値に制限はない。

7.31.2 複素数引数の三角関数

`int gsl_sf_complex_sin_e (`*double* `zr`, *double* `zi`, *gsl_sf_result* * `szr`, *gsl_sf_result* * `szi``)` [Function]

複素正弦関数 $\sin(z_r + iz_i)$ の値を計算し、実部と虚部をそれぞれ *szr* と *szi* に入れて返す。

`int gsl_sf_complex_cos_e` (*double* **`zr`**, *double* **`zi`**, *gsl_sf_result* * **`czr`**, *gsl_sf_result* * **`czi`**) [Function]

複素余弦関数 $\cos(z_r + iz_i)$ の値を計算し、実部と虚部をそれぞれ *czr* と *czi* に入れて返す。

`int gsl_sf_complex_logsin_e` (*double* **`zr`**, *double* **`zi`**, *gsl_sf_result* * **`lszr`**, *gsl_sf_result* * **`lszi`**) [Function]

複素正弦関数の対数 $\log(\sin(z_r + iz_i))$ の値を計算し、実部と虚部をそれぞれ *lszr* と *lszi* に入れて返す。

7.31.3　双曲線関数の対数

`double gsl_sf_lnsinh` (*double* **`x`**) [Function]
`int gsl_sf_lnsinh_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

$\log(\sinh(x))$ を $x > 0$ に対して計算する。

`double gsl_sf_lncosh` (*double* **`x`**) [Function]
`int gsl_sf_lncosh_e` (*double* **`x`**, *gsl_sf_result* * **`result`**) [Function]

$\log(\cosh(x))$ を計算する。x の値に制限はない。

7.31.4　座標変換のための関数

`int gsl_sf_polar_to_rect` (*double* **`r`**, *double* **`theta`**, *gsl_sf_result* * **`x`**, *gsl_sf_result* * **`y`**) [Function]

極座標値 (polar cooridnate, r, *theta*) を直交座標値 (rectilinear cooridinate, x, y) に、$x = r\cos(\theta), y = r\sin(\theta)$ として変換する。

`int gsl_sf_rect_to_polar` (*double* **`x`**, *double* **`y`**, *gsl_sf_result* * **`r`**, *gsl_sf_result* * **`theta`**) [Function]

直交座標値 (x, y) を極座標値 (r, *theta*) に、$x = r\cos(\theta), y = r\sin(\theta)$ から計算する。*theta* の値は $[-\pi, \pi]$ の範囲内になる。

7.31.5　制限範囲内の値に変換する関数

`double gsl_sf_angle_restrict_symm` (*double* **`theta`**) [Function]
`int gsl_sf_angle_restrict_symm_e` (*double* * **`theta`**) [Function]

角度 *theta* を $(-\pi, \pi]$ の範囲内に換算する。

GSL 中で定義している円周率の近似値 `M_PI` は、厳密な円周率の値 π よりもわずかに小さいため、`M_PI` とその符号を反転した値 `-M_PI` は、この関数があつかう範囲の境界上ではなく、その内側になる。

`double gsl_sf_angle_restrict_pos (`*double* **`theta`**`)` [Function]
`int gsl_sf_angle_restrict_pos_e (`*double* * **`theta`**`)` [Function]

角度 *theta* を $[0, 2\pi)$ の範囲内に換算する。GSL 中で定義している円周率の近似値 `M_PI` は、厳密な円周率の値 π よりもわずかに小さいため、`2*M_PI` の値は、この関数があつかう範囲の境界上ではなく、その内側になる。

7.31.6 誤差推定を行う三角関数

`int gsl_sf_sin_err_e (`*double* **`x`**, *double* **`dx`**, *gsl_sf_result* * **`result`**`)` [Function]

角度 x が絶対誤差 dx を含むとしたときのその正弦関数値 $\sin(x \pm dx)$ を計算する。引数の誤差が関数値にどの程度影響するかを知るために使われる。

`int gsl_sf_cos_err_e (`*double* **`x`**, *double* **`dx`**, *gsl_sf_result* * **`result`**`)` [Function]

角度 x が絶対誤差 dx を含むとしたときのその余弦関数値 $\cos(x \pm dx)$ を計算する。引数の誤差が関数値にどの程度影響するかを知るために使われる。

7.32 ゼータ関数

リーマンのゼータ関数 (Riemann zeta function) の定義はアブラモウィッツ＆ステグンの第 23.2 節にある。関数はヘッダファイル '`gsl_sf_zeta.h`' で宣言されている。

7.32.1 リーマンのゼータ関数

リーマンのゼータ関数は、無限級数 $\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$ で定義される。

`double gsl_sf_zeta_int (`*int* **`n`**`)` [Function]
`int gsl_sf_zeta_int_e (`*int* **`n`**, *gsl_sf_result* * **`result`**`)` [Function]

$n \neq 1$ の 整数 n に対してリーマンのゼータ関数 $\zeta(n)$ を計算する。

`double gsl_sf_zeta (`*double* **`s`**`)` [Function]
`int gsl_sf_zeta_e (`*double* **`s`**, *gsl_sf_result* * **`result`**`)` [Function]

$s \neq 1$ の任意の実数 s に対してリーマンのゼータ関数 $\zeta(s)$ を計算する。

7.32.2　リーマンのゼータ関数から 1 を引いた値

引数が大きな正の数の場合、リーマンのゼータ関数の値は 1 に近づくが、そういった場合は関数値の小数部分が重要である。そのため、小数部だけを計算する方法を用意している。

`double gsl_sf_zetam1_int (int n)` [Function]
`int gsl_sf_zetam1_int_e (int n, gsl_sf_result * result)` [Function]

$n \neq 1$ の 整数 n に対して $\zeta(n) - 1$ を計算する。

`double gsl_sf_zetam1 (double s)` [Function]
`int gsl_sf_zetam1_e (double s, gsl_sf_result * result)` [Function]

$s \neq 1$ の任意の実数 s に対して $\zeta(s) - 1$ を計算する。

7.32.3　フルヴィッツのゼータ関数

フルヴィッツ (Hurwitz) のゼータ関数は $\zeta(s,q) = \sum_{k=0}^{\infty}(k+q)^{-s}$ で定義され、数論やフラクタル理論、統計学などの幅広い分野で見られる。

`double gsl_sf_hzeta (double s, double q)` [Function]
`int gsl_sf_hzeta_e (double s, double q, gsl_sf_result * result)` [Function]

$s > 1$、$q > 0$ に対してフルヴィッツのゼータ関数 $\zeta(s,q)$ の値を計算する。

7.32.4　イータ関数 (η 関数)

イータ関数 (ディリクレのイータ関数、Dirichlet eta function) はリーマンのゼータ関数を使って $\eta(s) = (1 - 2^{1-s})\zeta(s)$ で定義され、数論などで用いられる。

`double gsl_sf_eta_int (int n)` [Function]
`int gsl_sf_eta_int_e (int n, gsl_sf_result * result)` [Function]

整数 n に対してイータ関数 $\eta(n)$ の値を計算する。

`double gsl_sf_eta (double s)` [Function]
`int gsl_sf_eta_e (double s, gsl_sf_result * result)` [Function]

任意の実数 s に対してイータ関数 $\eta(s)$ の値を計算する。

7.33　例

以下の例では、ベッセル関数 $J_0(5.0)$ の計算でエラー形の呼び出しを行う。

```
#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;
}
```

プログラムの実行結果を以下に示す。

```
$ ./a.out
status  = success
J0(5.0) = -0.177596771314338292
      +/-  0.000000000000000193
exact   = -0.177596771314338292
```

次のプログラムでは、同じ計算を一般形の呼び出しで行う。この場合は誤差の項 *result.err* やエラーを示す返り値は利用できない。

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

int main (void)
{
    double x = 5.0;
    double expected = -0.17759677131433830434739701;

    double y = gsl_sf_bessel_J0 (x);

    printf ("J0(5.0) = %.18f\n", y);
    printf ("exact   = %.18f\n", expected);
```

```
    return 0;
}
```

関数値の計算結果は同じである。

```
$ ./a.out
J0(5.0) = -0.177596771314338292
exact   = -0.177596771314338292
```

7.34　参考文献

このライブラリではできる限り、アブラモウィッツ＆ステグンの例にならっている。

- Abramowitz, Stegun (eds.), *Handbook of Mathematical Functions*, National Bureau of Standards, U.S. (1964).

特殊関数の計算法については、以下の論文に解説がある。

- Allan J. MacLeod, “MISCFUN: A software package to compute uncommon special functions”, *ACM Transactions on Mathematical Software*, **22**, pp. 288–301 (1996).
- Allan J. MacLeod, ”The numerical computation of transport integrals”, *Computer Physics Communications*, **69**, pp. 229–234 (1992).
- 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)
- 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).
- Nico M. Temme, *Special Functions: An Introduction to the Classical Functions of Mathematical Physics*, WileyBlackwell, ISBN 978-0471113133 (1996).

第8章　ベクトルと行列

この章では、一般的な C 言語の配列を使ったベクトルと行列の実装について説明する。どちらの実装も実体は配列だが、GSL の内部では「ブロック (block)」という一つのデータ型で実装されている。これらのベクトルや行列を使ったプログラムを書く時、GSL で定義している要素と次元数の両方を保持する構造体を使うことで、一つの引数として GSL の様々な関数に渡すことができる。この構造体は BLAS で実装されているベクトルと行列の形式と互換である。

8.1　データの型

GSL で用意している関数は、標準的なデータ型それぞれについて用意されている。double 型のための関数は、その名前の先頭に gsl_block、gsl_vector、gsl_matrix のいずれかがついている。これが単精度の float 型の場合は gsl_block_float、gsl_vector_float、gsl_matrix_float がついている。実装されているものの、block の場合のリストを以下にあげる。

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

gsl_vector と gsl_matrix についてもそれぞれ、対応する関数がある。

8.2　ブロック

メモリ操作に統一性と一貫性を持たせるため、メモリ確保はすべて gsl_block 構造体を使って行われる。この構造体にはメモリ領域の大きさ、メモリ領域へのポインタの二つの要素がある。

gsl_block 構造体は以下のように定義されている。

```
typedef struct {
    size_t size;
    double * data;
} gsl_block;
```

ベクトルと行列は、確保されたブロックを「スライス (slice)」することで作られる。スライスとは、オフセットブロック内でのスライスの開始位置、そこからそのスライスのデータが配置される) と刻み幅 (step-size) で指定される、ブロック中に一定間隔で置かれる一連のメモリ要素である。行列の場合、列の添え字に対する刻み幅はすなわち行の長さである (つまり、列の添字を 1 増やすと、メモリ上では行の長さだけ先の要素を指すことになる)。ベクトルの場合は、刻み幅をストライド (stride) とも呼ぶ。

ブロックの確保と解放を行う関数は 'gsl_block.h' で宣言されている。

8.2.1　ブロックの確保

ブロックのメモリを確保する関数は malloc と free と同様の使い方ができる。それに加えて独自のエラーチェックを行う。ブロックに割り当てる十分なメモリが確保できない場合は、GSL のエラー・ハンドラーをエラー番号 GSL_ENOMEM で呼び出して、NULL ポインタを返す。したがって、GSL のエラー・ハンドラーを使ってプログラムを終了させるなら、alloc などを呼ぶたびにエラーをチェックする必要はない (デフォルトのエラー・ハンドラーが abort を呼ぶ)。

gsl_block * gsl_block_alloc (*size_t* **n**) [Function]

倍精度実数の要素が n 個からなるブロックのメモリを確保し、ブロック構造体へのポインタを返す。ブロックは初期化されないため、確保されたブロック内の各要素の値は不定である。値を 0 で初期化したい場合は gsl_block_calloc を使う。

十分な大きさのメモリが確保できなかった場合は NULL ポインタを返す。

gsl_block * gsl_block_calloc (*size_t* **n**) [Function]

ブロックに割り当てるメモリを確保し、すべての要素の値を 0 にする。

void gsl_block_free (*gsl_block* * **b**) [Function]

gsl_block_alloc または gsl_block_calloc でブロック b にすでに割り当てられているメモリを解放する。b は正常に確保されたブロックでなければならない (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

8.2.2　ブロックのファイル入出力

GSL では、ブロックの内容をバイナリあるいは書式付きテキスト形式でファイルに読み書きする関数が実装されている。

int gsl_block_fwrite (*FILE* * **stream**, *const gsl_block* * **b**) [Function]

ブロック *b* の要素をファイル *stream* にバイナリ形式で書き込む。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。バイナリ形式は実行中のアーキテクチャに依存した形式なので、移植性は保証されない。

int gsl_block_fread (*FILE * **stream**, const gsl_block * **b***) [Function]

ブロック *b* の要素をファイル *stream* を開いてバイナリ形式で読み込む。そのブロックの大きさにしたがって読み込みが行われるため、ブロック *b* はあらかじめ正しい大きさで確保しておかねばならない。読み込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。データは以前に同じアーキテクチャのバイナリ形式で書き込まれたものとして読み込む。

int gsl_block_fprintf (*FILE * **stream**, const gsl_block * **b**, const char * **format***) [Function]

ブロック *b* の要素を 1 行ずつ format で指定される書式でファイル *stream* にテキスト形式で書き込む。書式指定は浮動小数点に対しては %g、%e、%f、整数に対しては %d を用いる。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。

int gsl_block_fscanf (*FILE * **stream**, const gsl_block * **b***) [Function]

ブロック *b* の要素をファイル *stream* からテキスト形式で読み込む。そのブロックの大きさにしたがって読み込みが行われるため、ブロック *b* はあらかじめ正しい大きさで確保しておかねばならない。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。

8.2.3 ブロックのプログラム例

以下にブロックを確保する例を示す。

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

int main (void) {
    gsl_block * b = gsl_block_alloc(100);

    printf("length of block = %u\n", b->size);
    printf("block data address = %#x\n", b->data);

    gsl_block_free(b);
    return 0;
}
```

上のプログラムの出力はこのようになる (block data address はほとんどの場合に下の例とは違う値になるが、異常ではない)。

```
length of block = 100
block data address = 0x804b0d8
```

8.3　ベクトル

ベクトルは、ブロックのスライスとして gsl_vector 構造体で定義される。そのため、一つの同じブロック上に複数のベクトルを保持することができる。一つのベクトル・スライスは、一つのメモリ領域中に等間隔に並ぶ複数の要素である。

gsl_vector 構造体は以下のような定義であり、五つの要素からなる。それぞれ、大きさ *size*、刻み幅 *stride*、ブロック中でのベクトルの先頭要素へのポインタ *data*、要素が保持されているブロックへのポインタ *block*、そのブロックの管理権を表す *owner* である。

```
typedef struct {
    size_t size;
    size_t stride;
    double * data;
    gsl_block * block;
    int owner;
} gsl_vector;
```

size は単にベクトルの要素の個数である。したがってベクトル要素の添え字の範囲は 0 から size-1 である。*stride* は物理的なメモリ上でのある要素から次の要素までの間隔で、その型のオブジェクトでの個数単位である (たとえば double なら、*stride* が 4 のとき、実メモリ上では 4 * sizeof(double) バイト単位になる)。ポインタ *data* はベクトルの先頭要素のメモリ上での位置を指す。ポインタ *block* はベクトルの要素を保持するブロックへのポインタである (そういうブロックがある場合)。ベクトルがそのブロックを所持している、とされる場合には *owner* フラグに 1 が代入され、ベクトルが解放されるときにそのブロックも解放される。そのブロックが他のベクトルに所持されている場合、*owner* に 0 が代入される。その場合、ベクトルの解放時にはそのブロックは解放されない (後述のベクトルおよび行列の「像 (view)」を、ベクトルあるいは行列の一部から作ることができる。その場合に、新しく作った像 = 部分ベクトルは独自にメモリ領域を持つわけではなく、元のベクトルの要素を指すポインタと刻み幅から作られる。したがって元のデータを保持するブロックの *owner* にはならない。この「像」がまだあるうちに元のベクトルを gsl_vector_free で解放してしまうと、その「像」は実体を失うことになる)。

ベクトルを生成、操作する関数は 'gsl_vector.h' で定義されている。

8.3.1　ベクトルの確保

ベクトルのメモリを確保する関数は malloc と free と同様の使い方ができる。それに加えて独自のエラーチェックを行う。ベクトルに割り当てる十分なメモリが確保できない場合は、GSL のエラー・ハンドラーをエラー番号 GSL_ENOMEM で呼び出して、NULL ポインタを返す。したがって、

GSL のエラー・ハンドラーを使ってプログラムを終了させるなら、alloc などを呼ぶたびにエラーをチェックする必要はない。

gsl_vector * gsl_vector_alloc (*size_t* n) [Function]

長さ n のベクトルを生成し、生成したベクトル構造体へのポインタを返す。ベクトルの要素にはブロックが割り当てられ、構造体のメンバー *block* に保持される。このブロックはこのベクトルが所持することになり、このベクトルが解放されるときにブロックも解放される。

gsl_vector * gsl_vector_calloc (*size_t* n) [Function]

長さ n のベクトルを生成し、ベクトルの要素を 0 にする。

void gsl_vector_free (*gsl_vector* * v) [Function]

すでに確保されているベクトル v を解放する。そのベクトルが gsl_vector_alloc で確保されたものなら、v が所持するブロックも解放される。v がほかのオブジェクトから生成されたベクトルである場合は、ブロックはそのオブジェクトが所持した状態に保たれ、解放されない。ベクトル v は正常に確保されたものでなければならない (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

8.3.2 ベクトル要素の操作

FORTRAN のコンパイラと違って、C コンパイラは普通、ベクトルや行列の添え字の範囲の確認を行わない。範囲確認は GNU C コンパイラの拡張機能の bounds-checking (境界確認) 機能で可能になるが、GCC のデフォルト・インストールでは無効になっている。GSL の関数 gsl_vector_get と gsl_vector_set を使えば添字の範囲確認ができ、また移植性もある。範囲外となる要素を操作しようとしたときにエラーを出すこともできる。

ベクトルや行列の要素を操作する関数は 'gsl_vector.h' で、関数呼び出しのオーバーヘッドを減らすため extern_inline として宣言されている。これらの関数を使うプログラムでインライン展開による速度向上を図りたいときは、マクロ HAVE_INLINE を定義してコンパイルしすればよい。

もし必要があれば、GSL_RANGE_CHECK_OFF を定義して GSL を利用するプログラムを再コンパイルすることで、ソース・コードを変更せずに範囲確認を完全に無効にすることができる。コンパイラがインライン関数をサポートしている場合は、範囲確認を無効にすることは、gsl_vector_get(v,i) という関数呼び出しを v->data[i*v->stride] に、gsl_vector_set(v,i,x) を v->data[i*v->stride]=x に置き換えることと同じである。これらの範囲確認を行う関数をプログラム中で使っていても、範囲確認を無効にすれば、範囲確認による実行速度の低下はないはずである。

double gsl_vector_get (*const gsl_vector* * v, *size_t* i) [Function]

ベクトル v の i 番目の要素を返す。i が 0 から $n-1$ の範囲になければ、エラーハンドラーを呼び出し、0 を返す。HAVE_INLINE が定義されているときは、インライン展開される。

`double gsl_vector_set` (*const gsl_vector* * `v`, *size_t* `i`, *double* `x`) [Function]

ベクトル v の i 番目の要素に x の値を代入する。i が 0 から $n-1$ の範囲になければ、エラーハンドラーを呼び出す。`HAVE_INLINE` が定義されているときは、インライン展開される。

`double * gsl_vector_ptr` (*gsl_vector* * `v`, *size_t* `i`) [Function]
`const double * gsl_vector_const_ptr` (*const gsl_vector* * `v`, *size_t* `i`) [Function]

ベクトル v の i 番目の要素へのポインタを返す。i が 0 から $n-1$ の範囲になければ、エラーハンドラーを呼び出し、NULL ポインタを返す。`HAVE_INLINE` が定義されているときは、インライン展開される。

8.3.3 ベクトル要素の初期化

`void gsl_vector_set_all` (*gsl_vector* * `v`, *double* `x`) [Function]

ベクトル v のすべての要素の値を x の値にする。

`void gsl_vector_set_zero` (*gsl_vector* * `v`) [Function]

ベクトル v のすべての要素の値を 0 にする。

`void gsl_vector_set_basis` (*gsl_vector* * `v`, *size_t* `i`) [Function]

ベクトル v の i 番目の要素の値を 1 に、ほかの要素の値を 0 にすることで、基底ベクトルを作る。

8.3.4 ベクトルのファイル入出力

GSL では、ベクトルをバイナリあるいは書式付きテキスト形式でファイルに読み書きする関数が実装されている。

`int gsl_vector_fwrite` (*FILE* * `stream`, *const gsl_vector* * `v`) [Function]

ベクトル v の要素をファイル *steam* にバイナリ形式で書き込む。書き込みが成功すれば 0 を、失敗すれば `GSL_EFAILED` を返す。バイナリ形式は実行中のアーキテクチャに依存した形式なので、移植性は保証されない。

`int gsl_vector_fread` (*FILE* * `stream`, *const gsl_vector* * `v`) [Function]

ベクトル v の要素をファイル *steam* を開いてバイナリ形式で読み込む。読み込むバイト数はベクトルの大きさから決められるため、ベクトル v はあらかじめ正しい大きさで確保しておかねばならない。読み込みが成功すれば 0 を、失敗すれば `GSL_EFAILED` を返す。データは以前に同じアーキテクチャのバイナリ形式で書き込まれたものという前提で読み込む。

int gsl_vector_fprintf (*FILE * stream*, *const gsl_vector * v*, *const char * format*) [Function]

ベクトル *v* の要素を 1 行ずつ *format* で指定される形式でファイル *steam* にテキスト形式で書き込む。形式指定は浮動小数点に対しては %g、%e、%f 、整数に対しては %d を用いる。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。

int gsl_vector_fscanf (*FILE * stream*, *gsl_vector * v*) [Function]

ベクトル *v* の要素をファイル *stream* からテキスト形式で読み込む。読み込む数値の個数はブロックの大きさから決められるため、ベクトル *v* はあらかじめ正しい大きさで確保しておかねばならない。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。

8.3.5 ベクトルの像

ブロックのスライスからベクトルを作るのと同様に、ベクトルのスライスからベクトルの像 (vector view) を作ることもできる。たとえば、あるベクトルの部分ベクトルを像として表現することができ、またベクトルの偶数番目の要素すべてと、奇数番目の要素すべてから、それぞれ一つの像を作ることができる。

ベクトルの像は、メモリスタック上に作られる一時的なオブジェクトであり、ベクトルの要素の部分集合に対する演算に使うことができる。ベクトルの像は、const および const でないベクトルにそれぞれ対応した型を使って定義できる。const でないベクトルから作る像の型は gsl_vector_view、const なベクトルから作る像は gsl_vector_const_view である。どちらの場合も、像の各要素の値は、像のオブジェクトの vector 要素を gsl_vector として扱うことで参照できる。gsl_vector * 型または const gsl_vector * 型といった、ベクトル型へのポインタはこれらの要素に & 演算子をつけることで得られる。

ポインタを使う際には、ベクトルの像のスコープに気をつけねばならない。そのためには、ベクトルの像へのポインタを常に &*view*.vector の形で使うのがもっとも簡潔である。この値を他の変数に代入することは避けるべきである (像は関数中の局所変数と同様に、それを宣言した関数の中でしか有効ではない。したがって像のポインタを他の大域変数などに代入したとしても、像を造った関数から一度 return してしまうと、もうその値は無意味である)。

gsl_vector_view gsl_vector_subvector (*gsl_vector * v*, *size_t offset*, *size_t n*) [Function]

gsl_vector_const_view gsl_vector_const_subvector (*const gsl_vector * v*, *size_t offset*, *size_t n*) [Function]

ベクトル *v* の部分ベクトルを、ベクトルの像として返す。新しいベクトルの先頭は、元のベクトルの先頭から *offset* だけずれた要素である。新しいベクトルの要素数は *n* である。数学的には、新しいベクトル *v'* の *i* 番目の要素は以下で表される。

```
v'(i) = v->data[(offset + i)*v->stride]
```

ここで添え字 i の範囲は 0 から $n-1$ である。

返されたベクトル構造体の `data` ポインタは、引数 ($offset$, n) が元のベクトルの大きさに収まらない場合には NULL となる。

新しく作られるベクトルは元のベクトル v の持つブロックの「像」でしかない。v の要素が置かれているブロックは、像として返される新しいベクトルが所有するわけではない。そのとき有効であるスコープの外に出ても、ベクトル v とそのブロックはそのまま残る。元のベクトルのメモリを解放するには、元のベクトルを解放せねばならない。また像を操作、参照している間は、元のベクトルを解放してはいけない。

関数 `gsl_vector_const_subvector` は、`const` と宣言されたベクトルに使えること以外は、`gsl_vector_subvector` と同じである。

`gsl_vector_view gsl_vector_subvector_with_stride` (*gsl_vector* * **v**, *size_t* ***offset***, *size_t* ***stride***, *size_t* **n**) [Function]

`gsl_vector_const_view gsl_vector_const_subvector_with_stride` (*const gsl_vector* * **v**, *size_t* ***offset***, *size_t* ***stride***, *size_t* **n**) [Function]

ベクトル v の、引数 *stride* で指定された刻み幅を持つ部分ベクトルを表す像を返す。部分ベクトルは `gsl_vector_subvector` と同様に作られるが、新しいベクトルは元のベクトルにおいて、ある要素から次の要素までの刻み幅が *stride* の n 個の要素を持つ。数学的には、新しいベクトル v' の i 番目の要素は以下で表される。

```
v'(i) = v->data[(offset + i*stride)*v->stride]
```

ここで添え字 i の範囲は 0 から $n-1$ である。

部分ベクトルによる像を使うと、元のベクトルの要素を直接に参照、操作できる。たとえば以下のプログラムでは、長さ n のベクトル v の偶数番目の要素の値を 0 にし、奇数番目の要素の値は変更しない。

```
gsl_vector_view v_even
    = gsl_vector_subvector_with_stride(v, 0, 2, n/2);
gsl_vector_set_zero(&v_even.vector);
```

ベクトルの像は、&*view*`.vector` を使うことで、ベクトルを引数に指定できる関数に、直接生成されたベクトルと同様に渡すことができる。たとえば以下のプログラムは、BLAS の関数 DNRM2 を使って、v の偶数番目の要素のノルムを計算する。

```
gsl_vector_view v_odd
    = gsl_vector_subvector_with_stride(v, 1, 2, n/2);
double r = gsl_blas_dnrm2(&v_odd.vector);
```

関数 `gsl_vector_const_subvector_with_stride` は、`const` と宣言されたベクトルに使えること以外は、`gsl_vector_subvector_with_stride` と同じである。

gsl_vector_view gsl_vector_complex_real (*gsl_vector_complex* * v) [Function]
gsl_vector_const_view gsl_vector_complex_const_real (*const gsl_vector_complex* * v) [Function]

複素数ベクトル *v* の実部からなる像を返す。

関数 gsl_vector_complex_const_real は、const と宣言されたベクトルに使えること以外は、gsl_vector_complex_real と同じである。

gsl_vector_view gsl_vector_complex_imag (*gsl_vector_complex* * v) [Function]
gsl_vector_const_view gsl_vector_complex_const_ imag (*const gsl_vector_complex* * v) [Function]

複素数ベクトル *v* の虚部からなる像を返す。

関数 gsl_vector_complex_const_imag は、const と宣言されたベクトルに使えること以外は、gsl_vector_complex_imag と同じである。

gsl_vector_view gsl_vector_view_array (*double* * **base**, *size_t* **n**) [Function]
gsl_vector_const_view gsl_vector_const_view_array (*const double* * **base**, *size_t* **n**) [Function]

指定された配列から、ベクトルの像を返す。新しいベクトルの先頭は *base* で、要素数は *n* で指定する。数学的には、新しいベクトル *v'* の *i* 番目の要素は以下で表される。

```
v'(i) = base[i]
```

ここで添え字 *i* の範囲は 0 から n-1 である。

v の要素を保持する配列は、新しいベクトルが所有するわけではない。像がスコープから出た場合も元の配列はそのままである。元の配列のメモリは、元のポインタ *base* を使ってのみ解放できる。像を参照、操作している間は元の配列を解放してはいけない。

関数 gsl_vector_const_view_array は、const と宣言された配列に使えること以外は、gsl_vector_view_array と同じである。

gsl_vector_view gsl_vector_view_array_with_stride (*double* * **base**, *size_t* **stride**, *size_t* **n**) [Function]
gsl_vector_const_view gsl_vector_const_view_array_with_stride (*const double* * **base**, *size_t* **stride**, *size_t* **n**) [Function]

指定された配列 *base* から、引数で指定された刻み幅を持つベクトルの像を返す。部分ベクトルは gsl_vector_view_array と同様に作られるが、新しいベクトルは元のベクトルにおいて、ある要素から次の要素までの刻み幅が *stride* の *n* 個の要素を持つ。数学的には新しいベクトル *v'* の *i* 番目の要素は以下で表される。

```
v'(i) = base[i*stride]
```

ここで添え字 i の範囲は 0 から $n-1$ である。

ベクトルの像を使うことで、元の配列の要素を直接参照、操作することができる。ベクトルの像は `&`*view*`.vector` を使うことで、ベクトルを引数に指定できる関数に、直接生成されたベクトルと同様に渡すことができる。

関数 `gsl_vector_const_view_array_with_stride` は、`const` と宣言された配列に使えること以外は、`gsl_vector_view_array_with_stride` と同じである。

8.3.6　ベクトルの複製

加算や乗算などの、よく行われるベクトル演算は、このライブラリの BLAS 関連の機能として実装されている (第 12 章 [BLAS Support] 参照)。しかしいくつかの簡単な演算には、BLAS のコード全体を使わずにすむように、以下の関数が用意されている。

`int gsl_vector_memcpy (`*gsl_vector* * ***dest***, *const gsl_vector* * ***src***`)` [Function]

ベクトル *src* の要素をベクトル *dest* にコピーする。二つのベクトルは同じ大きさでなければならない。

`int gsl_vector_swap (`*gsl_vector* * `v`, *const gsl_vector* * `w``)` [Function]

ベクトル v とベクトル w を、コピーによって交換する（一方がもう一方を上書きすることがない)。二つのベクトルは同じ大きさでなければならない。

8.3.7　要素の入れ換え

以下の関数でベクトルの要素を交換 (exchange)、あるいは置換 (permute) することができる。

`int gsl_vector_swap_elements (`*gsl_vector* * `v`, *size_t* `i`, *size_t* `j``)` [Function]

ベクトル v の i 番目と j 番目の要素を入れ替える。

`int gsl_vector_reverse (`*gsl_vector* * `v``)` [Function]

ベクトル v の要素を、逆の順番に並べ替える。

8.3.8　ベクトルの演算

以下の演算は実数ベクトルについてのみ定義されている。

`int gsl_vector_add (`*gsl_vector* * `a`, *const gsl_vector* * `b``)` [Function]

ベクトル b の要素の値をベクトル a の要素に、$a'_i = a_i + b_i$ のように加える。二つのベクトルは同じ大きさでなければならない。

`int gsl_vector_sub (`*gsl_vector* * `a`, *const gsl_vector* * `b``)` [Function]

ベクトル b の要素の値をベクトル a の要素から、$a'_i = a_i - b_i$ のように減ずる。二つのベクトルは同じ大きさでなければならない。

`int gsl_vector_mul` (*gsl_vector* * `a`, *const gsl_vector* * `b`) [Function]

ベクトル b の要素の値をベクトル a の要素に、$a'_i = a_i * b_i$ のように乗ずる。二つのベクトルは同じ大きさでなければならない。

`int gsl_vector_div` (*gsl_vector* * `a`, *const gsl_vector* * `b`) [Function]

ベクトル a の要素をベクトル b の要素で、$a'_i = a_i/b_i$ のように除する。二つのベクトルは同じ大きさでなければならない。

`int gsl_vector_scale` (*gsl_vector* * `a`, *const double* * `x`) [Function]

ベクトル a の要素に定数係数 x を $a'_i = xa_i$ のように乗ずる。

`int gsl_vector_add_constant` (*gsl_vector* * `a`, *const double* * `x`) [Function]

ベクトル a の要素に定数値 x を $a'_i = a_i + x$ のようにして加える。

8.3.9 ベクトル中の最大、最小要素の検索

`double gsl_vector_max` (*const gsl_vector* * `v`) [Function]

ベクトル v の要素の中で最大のものの値を返す。

`double gsl_vector_min` (*const gsl_vector* * `v`) [Function]

ベクトル v の要素の中で最小のものの値を返す。

`void gsl_vector_minmax` (*const gsl_vector* * `v`, *double* * `min_out`, *double* * `max_out`) [Function]

ベクトル v の要素の中で最小および最大のものの値を *min_out* および *max_out* に入れて返す。

`size_t gsl_vector_max_index` (*const gsl_vector* * `v`) [Function]

ベクトル v の要素の中で最大のものの添え字を返す。複数の要素が該当するときはもっとも小さな添え字を返す。

`size_t gsl_vector_min_index` (*const gsl_vector* * `v`) [Function]

ベクトル v の要素の中で最小のものの添え字を返す。複数の要素が該当するときはもっとも小さな添え字を返す。

`void gsl_vector_minmax_index` (*const gsl_vector* * `v`, *size_t* * `imin`, *size_t* * `imax`) [Function]

ベクトル v の要素の中で最小および最大のものの添え字を *imin* および *imax* に入れて返す。複数の要素が該当するときは、それぞれでもっとも小さな添え字を返す。

8.3.10　ベクトルの属性

`int gsl_vector_isnull (const gsl_vector * v)` [Function]
`int gsl_vector_ispos (const gsl_vector * v)` [Function]
`int gsl_vector_isneg (const gsl_vector * v)` [Function]
`int gsl_vector_isnonneg (const gsl_vector * v)` [Function]

それぞれ、ベクトル v の要素がすべて 0、すべて正、すべて負、すべて非負 のとき 1、そうでないとき 0 を返す。

8.3.11　ベクトルのプログラム例

以下に、関数 gsl_vector_alloc、gsl_vector_set、gsl_vector_get を使ってベクトルを確保、初期化して読み込むプログラムを示す。

```
#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++) /* わざと範囲外にアクセス */
        printf("v_%d = %g\n", i, gsl_vector_get(v, i));

    gsl_vector_free(v);
    return 0;
}
```

以下にプログラムの出力を示す。プログラムの最後のループは、範囲外の要素にアクセスしてエラーを発生させ、gsl_vector_get 内の範囲確認ルーチンでトラップさせるためのものである。

```
./a.out
v_0 = 1.23
v_1 = 2.23
v_2 = 3.23
gsl: vector_source.c:12: ERROR: index out of range
Default GSL error handler invoked.
Aborted (core dumped)
```

次のプログラムではベクトルをファイルに書き込む。

```
#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;
}
```

このプログラムを実行すると、書式 %.5g でベクトル v の要素の値がファイル 'test.dat' に書き込まれる。書き込まれたベクトルは以下のようにして、関数 gsl_vector_fscanf(f, v) を使って読みなおすことができる。

```
#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);

    for (i = 0; i < 10; i++)
        printf("%g\n", gsl_vector_get(v, i));

    gsl_vector_free (v);
    return 0;
}
```

8.4　行列

行列は、gsl_matrix 構造体によって、ブロックのスライスとして定義される。ベクトルと同様にメモリ領域内にある要素の集合として扱われるが、添え字は一つではなく二つである。

gsl_matrix 構造体には 6 個の要素がある。それぞれ、行列の二つの次数、物理的な次元数 *tda*、メモリ上で行列の先頭要素を保持している要素へのポインタ *data*、行列が「所持」しているブロックへのポインタ *block*、所持しているかどうかを表すフラグ *owner* である。物理次元数 *tda* はメモリ上での配置を決めるために使われ、部分行列を扱うときには行列の次元数とは違った値にすることができる。gsl_matrix 構造体は非常に単純で、たとえば以下のように定義される。

```
typedef struct {
    size_t size1;
    size_t size2;
    size_t tda;
    double * data;
    gsl_block * block;
    int owner;
} gsl_matrix;
```

行列は行指向、つまりメモリ領域内では行内の要素が連続して並ぶように保持される。これは C 言語での二次元配列の並び方と同じであり、列指向の FORTRAN とは逆である。この構造体のメンバの *size1* が行列の行の数であり、行の添え字の有効範囲は 0 から $size1 - 1$ である。列の数が *size2* である。列の添え字の範囲は 0 から $size2 - 1$ である。物理的な次元数 *tda* はここでは展開次元 (trailing dimension) とも呼び、メモリ上に展開されている行列の一行の長さ、つまり列の数を表す。

たとえば以下の行列では、*size1* が 3、*size2* が 4、*tda* が 8 である。物理的なメモリ配置は左上の隅から、左から右に行にそって進み、次の行に続く。

```
00 01 02 03 XX XX XX XX
10 11 12 13 XX XX XX XX
20 21 22 23 XX XX XX XX
```

メモリ上の使われていない場所を “XX” で示している。ポインタ *data* はメモリ上の行列の先頭の要素を指す。ポインタ *block* はメモリ上の行列の要素があるブロックの場所を指す（ブロックを所有している場合)。行列がこのブロックを所持していれば *onwer* フラグが 1 になっており、この行列が解放されるときにブロックも解放される。この行列が、ほかの行列やベクトルが「所持」しているブロックのスライスでしかないときは *onwer* は 0 で、その行列を開放してもブロックは解放されない。

行列の確保と参照、操作を行う関数は ‘gsl_matrix.h’ で定義されている。

8.4.1 行列の確保

行列のメモリを確保する関数は malloc と free と同様の使い方ができる。それに加えて独自のエラーチェックを行う。行列に割り当てる十分なメモリが確保できない場合は、GSL のエラー・ハンドラーをエラー番号 GSL_ENOMEM で呼び出して、NULL ポインタを返す。したがって、GSL のエラー・ハンドラーを使ってプログラムを終了させるなら、alloc などを呼ぶたびにエラーをチェックする必要はない。

gsl_matrix * gsl_matrix_alloc (*size_t n1*, *size_t n2*) [Function]

大きさが *n1* 行 *n2* 列の行列を生成し、新しい初期化された行列構造体へのポインタを返す。行列要素のためにブロックが確保され、行列構造体の *block* 要素に保持される。ブロックはこの行列構造体に「所持」され、行列が解放されるときにこの所持しているブロックも解放される。tda は *n2* になる。

gsl_matrix * gsl_matrix_calloc (*size_t n1*, *size_t n2*) [Function]

大きさが *n1* 行 *n2* の行列を生成し、行列のすべての要素を 0 に初期化する。

void gsl_matrix_free (*gsl_matrix * m*) [Function]

すでに確保されている行列 *m* を解放する。その行列が gsl_matrix_alloc を使って生成されたものの場合は、その行列が所持するブロックも解放する。ほかのオブジェクトから生成された行列の場合はブロックは元のオブジェクトが所持したままにされ、解放されない。行列 *m* は正常に確保されたものでなければならない (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

8.4.2 行列の要素の操作

行列の要素を参照、操作する関数はベクトルの場合と同様に添え字の範囲を確認するシステムを備えている。プリプロセッサで GSL_RANGE_CHECK_OFF を define してプログラムを再コンパイルすれば、範囲確認を無効にすることができる (第 節「ベクトル要素の操作」参照)。

行列の要素は、C 言語での順序、つまり二番目の添え字がメモリ上で連続した要素を表す。つまり、関数 gsl_matrix_get(m,i,j) と gsl_matrix_set(m,i,j,x) で参照、操作される要素は以下のように表現できる。

```
m->data[i * m->tda + j]
```

ここで *tda* は行列の物理次元数である。

double gsl_matrix_get (*const gsl_matrix * m*, *size_t i*, *size_t j*) [Function]

行列 *m* の (i, j) 成分を返す。*i* や *j* が 0 から $n1-1$ または 0 から $n2-1$ の範囲内になければ、エラー・ハンドラーを呼び出し、0 を返す。HAVE_INLINE が定義されているときは、インライン展開される。

`void gsl_matrix_set` (*gsl_matrix* * `m`, *size_t* `i`, *size_t* `j`, *double* `x`) [Function]

行列 m の (i,j) 成分に x の値を代入する。i や j が 0 から $n1-1$ または 0 から $n2-1$ の範囲内になければ、エラー・ハンドラーを呼び出す。`HAVE_INLINE` が定義されているときは、インライン展開される。

`double * gsl_matrix_ptr` (*gsl_matrix* * `m`, *size_t* `i`, *size_t* `j`) [Function]
`const double * gsl_matrix_const_ptr` (*const gsl_matrix* * `m`, *size_t* `i`, *size_t* `j`) [Function]

行列 m の (i,j) 成分へのポインタを返す。i や j が 0 から $n1-1$ または 0 から $n2-1$ の範囲内になければ、エラー・ハンドラーを呼び出し、NULL ポインタを返す。`HAVE_INLINE` が定義されているときは、インライン展開される。

8.4.3 行列要素の初期化

`void gsl_matrix_set_all` (*gsl_matrix* * `m`, *double* `x`) [Function]

行列 m のすべての要素の値を x にする。

`void gsl_matrix_set_zero` (*gsl_matrix* * `m`) [Function]

行列 m のすべての要素の値を 0 にする。

`void gsl_matrix_set_identity` (*gsl_matrix* * `m`) [Function]

行列 m の素の値を単位行列 $m(i,j) = \delta(i,j)$、つまり対角成分が 1 で非対角成分が 0 の行列の対応する要素の値にする。この関数は正方行列にもそれ以外にも使うことができる。

8.4.4 行列のファイル入出力

GSL では、行列をバイナリあるいは書式付きテキスト形式でファイルに読み書きする関数が実装されている。

`int gsl_matrix_fwrite` (*FILE* * `stream`, *const gsl_matrix* * `m`) [Function]

行列 m の要素をファイル *stream* にバイナリ形式で書き込む。書き込みが成功すれば 0 を、失敗すれば `GSL_EFAILED` を返す。バイナリ形式は実行中のアーキテクチャに依存した形式なので、移植性は保証されない。

`int gsl_matrix_fread` (*FILE* * `stream`, *gsl_matrix* * `m`) [Function]

行列 m の要素をファイル *stream* を開いてバイナリ形式で読み込む。読み込むバイト数は行列の大きさから決められるため、行列 m はあらかじめ正しい大きさで確保しておかねばならない。読み込みが成功すれば 0 を、失敗すれば `GSL_EFAILED` を返す。データは以前に同じアーキテクチャのバイナリ形式で書き込まれたものとして読み込む。

int gsl_matrix_fprintf (*FILE * stream*, *const gsl_matrix * m*, *const char * format*) [Function]

行列 m の要素を 1 行ずつ *format* で指定される書式でファイル *stream* にテキスト形式で書き込む。書式指定は浮動小数点に対しては%g、%e、%f、整数に対しては %d を用いる。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。

int gsl_matrix_fscanf (*FILE * stream*, *gsl_matrix * m*) [Function]

行列 m の要素をファイル *stream* からテキスト形式で読み込む。読み込む数値の個数はブロックの大きさから決められるため、行列 m はあらかじめ正しい大きさで確保しておかねばならない。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。

8.4.5 行列の像

行列の像 (view) は一時的なオブジェクトとしてメモリスタック上に保持され、行列の要素の部分集合に対する演算で使うことができる。行列の像は const および const でない行列に対してそれぞれ違った型を使って定義できるため、たとえば const な行列の像を作るために const でないコピーを作るといった手間は必要ない。const でない行列と const な行列の像の型はそれぞれ gsl_matrix_view および gsl_matrix_const_view である。どちらの場合も像の要素は、像のオブジェクトの matrix 要素を使って参照、操作することができる。gsl_matrix * 型または const gsl_matrix * 型の行列へのポインタは、matrix 要素に & 演算子をつけることで得られる。また、行または列の像のように、行列からベクトルの像を造ることができる。

gsl_matrix_view gsl_matrix_submatrix (*gsl_matrix * m*, *size_t k1*, *size_t k2*, *size_t n1*, *size_t n2*) [Function]

gsl_matrix_const_view gsl_matrix_const_submatrix (*const gsl_matrix * m*, *size_t k1*, *size_t k2*, *size_t n1*, *size_t n2*) [Function]

行列 m の部分行列の像を返す。返される部分行列は、元の行列 m の $(k1, k2)$ 成分を左上端とし、返す像の大きさは $n1$ 行 $n2$ 列である。物理的なメモリ上での列数は元の行列と同じで、tda で与えられる。数学的には、新しい行列の (i, j) 成分は以下のようになる。

```
m'(i,j) = m->data[(k1*m->tda + k2) + i*m->tda + j]
```

ここで i の範囲は 0 から $n1 - 1$、j の範囲は 0 から $n2 - 1$ である。

生成される行列構造体のメンバーであるポインタ *data* は、行列のほかのパラメータ $(i, j, n1, n2, tda)$ が元の行列のの範囲に収まらない場合、NULL となる。

新しい行列は、元の行列 m の持つブロックの像にすぎない。元の行列 m の要素が置かれているブロックは、新しい行列が所有するわけではない。新しい行列の像が、そのとき有効であるスコープの外に出た場合も、行列 m とそのブロックはそのまま残る。元の行列のメモリは、元の行列を解放するまで保持されている。したがって像を操作、参照している間は、元の行列を解放してはいけない。

関数 gsl_matrix_const_submatrix は、const と宣言された行列に使えること以外は、gsl_matrix_submatrix と同じである。

gsl_matrix_view gsl_matrix_view_array (*double * base*, *size_t n1*, *size_t n2*) [Function]

gsl_matrix_const_view gsl_matrix_const_view_array (*const double * base*, *size_t n1*, *size_t n2*) [Function]

配列 *base* の像を行列の像として返す。返される行列は *n1* 行 *n2* 列である。メモリ中の物理的な列数も *n2* になる。数学的には、新しい行列の (i, j) 成分は以下で表される。

```
m'(i,j) = base[i*n2 + j]
```

ここで i の範囲は 0 から $n1 - 1$、j の範囲は 0 から $n2 - 1$ である。

新しく作られる行列は配列 *base* の像にすぎない。その像がそのとき有効であるスコープの外に出た場合も、配列 *base* はそのまま残る。元のメモリは元の配列を解放するまで確保されている。したがって像を操作、参照している間は、元の行列を解放してはいけない。

関数 gsl_matrix_const_view_array は、const と宣言された行列に使えること以外は、gsl_matrix_view_array と同じである。

gsl_matrix_view gsl_matrix_view_array_with_tda (*double * base*, *size_t n1*, *size_t n2*, *size_t tda*) [Function]

gsl_matrix_const_view gsl_matrix_const_view_array_with_tda (*const double * base*, *size_t n1*, *size_t n2*, *size_t tda*) [Function]

tda で指定される物理的な列数（行列の次元が示す列の数と異なっていてもよい）を持つ、配列 *base* の行列の像を返す。返される行列は *n1* 行 *n2* 列で、メモリ中の列の物理的な列数は与えられた値 *tda* になる。数学的には、新しい行列の (i, j) 成分は以下で表される。

```
m'(i,j) = base[i*tda + j]
```

ここで i の範囲は 0 から $n1 - 1$、j の範囲は 0 から $n2 - 1$ である。

新しく作られる行列は配列 *base* の像にすぎない。その像がそのとき有効であるスコープの外に出た場合も、配列 *base* はそのまま残る。元のメモリは元の配列を解放するまで確保されている。したがって像を操作、参照している間は、元の行列を解放してはいけない。

関数 gsl_matrix_const_view_array_with_tda は、const と宣言された行列に使えること以外は、gsl_matrix_view_array_with_tda と同じである。

gsl_matrix_view gsl_matrix_view_vector (*gsl_vector * v*, *size_t n1*, *size_t n2*) [Function]

gsl_matrix_const_view gsl_matrix_const_view_vector (*const gsl_vector * v*, *size_t n1*, *size_t n2*) [Function]

ベクトル v から行列の形で像を作って返す。返される行列は $n1$ 行 $n2$ 列である。ベクトルの刻み幅 $stride$ は 1 でなければならない。メモリ中の物理的な列数も $n2$ になる。数学的には、新しい行列の (i, j) 成分は以下で表される。

```
m'(i,j) = v->data[i*n2 + j]
```

ここで i の範囲は 0 から $n1 - 1$、j の範囲は 0 から $n2 - 1$ である。

新しく作られる行列はベクトル v の像にすぎない。その像がそのとき有効であるスコープの外に出た場合も、ベクトル v はそのまま残る。元のメモリは元の配列を解放するまで確保されている。したがって像を操作、参照している間は、元のベクトルを解放してはいけない。

関数 `gsl_matrix_const_view_vector` は、`const` と宣言された行列に使えること以外は、`gsl_matrix_view_vector` と同じである。

`gsl_matrix_view gsl_matrix_view_vector_with_tda` (*gsl_vector* * `v`, *size_t* `n1`, *size_t* `n2`, *size_t* `tda`) [Function]

`gsl_matrix_const_view gsl_matrix_const_view_vector_with_tda` (*const gsl_vector* * `v`, *size_t* `n1`, *size_t* `n2`, *size_t* `tda`) [Function]

tda で指定される物理的な列数（行列の次元が示す列の数と異なっていてもよい）で、ベクトル v から行列の形で像を作って返す。ベクトルの刻み幅 $stride$ は 1 でなければならない。返される行列は $n1$ 行 $n2$ 列で、メモリ中の列の物理的な個数は与えられた値 tda になる。数学的には、新しい行列の (i, j) 成分は以下で表される。

```
m'(i,j) = v->data[i*tda + j]
```

ここで i の範囲は 0 から $n1 - 1$、j の範囲は 0 から $n2 - 1$ である。

新しく作られる行列はベクトル v の像にすぎない。その像がそのとき有効であるスコープの外に出た場合も、ベクトル v はそのまま残る。元のメモリは元の配列を解放するまで確保されている。したがって像を操作、参照している間は、元のベクトルを解放してはいけない。

関数 `gsl_matrix_const_view_vector_with_tda` は、`const` と宣言された行列に使えること以外は、`gsl_matrix_view_vector_with_tda` と同じである。

8.4.6 行または列の像の生成

一般にメモリ・オブジェクトにアクセスするには、参照と複製の二つの方法がある。この節では、行列の行または列の像を、参照によりベクトル像として生成する関数について説明する。生成されたベクトル像と元の行列のデータはどちらも実体は同じメモリブロックであるため、像の要素に変更を加えると、元の行列の要素の値も変更される。

`gsl_vector_view gsl_matrix_row` (*gsl_matrix* * `m`, *size_t* `i`) [Function]

`gsl_vector_const_view gsl_matrix_const_row` (*const gsl_matrix* * `m`, *size_t* `i`) [Function]

行列 m の i 番目の行のベクトル像を返す。i が範囲外の場合、新しく生成されたベクトルの data ポインタは NULL にされる。

関数 gsl_vector_const_row は、const と宣言された行列に使えること以外は、gsl_matrix_row と同じである。

gsl_vector_view gsl_matrix_column (*gsl_matrix * m*, *size_t j*) [Function]
gsl_vector_const_view gsl_matrix_const_column (*const gsl_matrix * m*, *size_t j*) [Function]

行列 m の j 番目の列のベクトル像を返す。j が範囲外の場合、新しく生成されたベクトルの data ポインタは NULL にされる。

関数 gsl_vector_const_column は、const と宣言された行列に使えること以外は、gsl_matrix_column と同じである。

gsl_vector_view gsl_matrix_subrow (*gsl_matrix * m*, *size_t i*, *size_t offset*, *size_t n*) [Function]
gsl_vector_const_view gsl_matrix_const_subrow (*const gsl_matrix * m*, *size_t i*, *size_t offset*, *size_t n*) [Function]

行列 m の i 番目の行の先頭から *offset* 番目の要素を先頭とし、n 個の要素からなるベクトル像を返す。i、*offset*、n のいずれかが範囲外の場合、新しく生成されたベクトルの data ポインタは NULL にされる。

関数 gsl_vector_const_subrow は、const と宣言された行列に使えること以外は、gsl_matrix_subrow と同じである。

gsl_vector_view gsl_matrix_subcolumn (*gsl_matrix * m*, *size_t j*, *size_t offset*, *size_t n*) [Function]
gsl_vector_const_view gsl_matrix_const_subcolumn (*const gsl_matrix * m*, *size_t j*, *size_t offset*, *size_t n*) [Function]

行列 m の j 番目の列の先頭から下に *offset* 番目の要素を先頭とし、n 個の要素からなるベクトル像を返す。j、*offset*、n のいずれかが範囲外の場合、新しく生成されたベクトルの data ポインタは NULL にされる。

関数 gsl_vector_const_subcolumn は、const と宣言された行列に使えること以外は、gsl_matrix_subcolumn と同じである。

gsl_vector_view gsl_matrix_diagonal (*gsl_matrix * m*) [Function]
gsl_vector_const_view gsl_matrix_const_diagonal (*const gsl_matrix * m*)[Function]

行列 m の対角成分からなるベクトル像を返す。行列 m は正方行列でなくてもよい。その場合、ベクトルの長さは行列の次元の小さい方になる。

関数 gsl_matrix_const_diagonal は、const と宣言された行列に使えること以外は、gsl_matrix_diagonal と同じである。

gsl_vector_view gsl_matrix_subdiagonal (*gsl_matrix * m*, *size_t k*) [Function]
gsl_vector_const_view gsl_matrix_const_subdiagonal (*const gsl_matrix * m*, *size_t k*) [Function]

行列 m の k 次の下対角成分からなるベクトル像を返す。行列 m は正方行列でなくてもよい。$k = 0$ で元の行列のすべての対角成分からなるベクトルが得られる。

関数 gsl_matrix_const_subdiagonal は、const と宣言された行列に使えること以外は、gsl_matrix_subdiagonal と同じである。

gsl_vector_view gsl_matrix_superdiagonal (*gsl_matrix * m*, *size_t k*) [Function]
gsl_vector_const_view gsl_matrix_const_superdiagonal (*const gsl_matrix * m*, *size_t k*) [Function]

行列 m の k 次の上対角成分からなるベクトル像を返す。行列 m は正方行列でなくてもよい。$k = 0$ で元の行列のすべての対角成分からなるベクトルが得られる。

関数 gsl_matrix_const_superdiagonal は、const と宣言された行列に使えること以外は、gsl_matrix_superdiagonal と同じである。

8.4.7 行列の複製

int gsl_matrix_memcpy (*gsl_matrix * dest*, *const gsl_matrix * src*) [Function]

行列 *src* の要素を行列 *dest* にコピーする。二つの行列の次元は等しくなければならない。

int gsl_matrix_swap (*gsl_matrix * m1*, *gsl_matrix * m2*) [Function]

行列 *m1* と *m2* の要素を、内部で (データの上書きや消失がないように) 複製を行いながら交換する。二つの行列の次元は等しくなければならない。

8.4.8 行または列の複製

この節では行列の行または列を複製してベクトルとする関数について説明する。これは像を使う場合と違って、作られたベクトルと行列の要素を別々に操作することができる。行列とベクトルがそれぞれ指すメモリ領域が重なっている場合は、操作の結果は不定である。以下に説明する関数は、行列の行や列のベクトル像に対して汎用の関数 gsl_vector_memcpy を使うことで、同じことができる。

int gsl_matrix_get_row (*gsl_vector * v*, *const gsl_matrix * m*, *size_t * i*) [Function]

行列 m の i 番目の行をベクトル v にコピーする。ベクトルの長さは行列の行の長さと同じでなければならない。

`int gsl_matrix_get_col` (*gsl_vector * v*, *const gsl_matrix * m*, *size_t * j*) [Function]

行列 m の j 番目の列をベクトル v にコピーする。ベクトルの長さは行列の列の長さと同じでなければならない。

`int gsl_matrix_set_row` (*gsl_matrix * m*, *size_t * i*, *const gsl_vector * v*) [Function]

ベクトル v を行列 m の i 番目の行にコピーする。ベクトルの長さは行列の行の長さと同じでなければならない。

`int gsl_matrix_set_col` (*gsl_matrix * m*, *size_t * j*, *const gsl_vector * v*) [Function]

ベクトル v を行列 m の j 番目の列にコピーする。ベクトルの長さは行列の列の長さと同じでなければならない。

8.4.9　行または列の入れ換え

行列の行や列の入れ換えは、以下の関数を使って行うことができる。

`int gsl_matrix_swap_rows` (*gsl_matrix * m*, *size_t * i*, *size_t * j*) [Function]

行列 m の i 番目と j 番目の行を入れ替える。

`int gsl_matrix_swap_columns` (*gsl_matrix * m*, *size_t * i*, *size_t * j*) [Function]

行列 m の i 番目と j 番目の列を入れ替える。

`int gsl_matrix_swap_rowcol` (*gsl_matrix * m*, *size_t * i*, *size_t * j*) [Function]

行列 m の i 番目の行と j 番目の列を入れ替える。行列 m は正方行列でなければならない。

`int gsl_matrix_transpose_memcpy` (*gsl_matrix * dest*, *const gsl_matrix * src*) [Function]

行列 *src* を転置して行列 *dest* にコピーする。*src* の転置行列の次元と *dest* の次元が一致していなければならない。

`int gsl_matrix_transpose` (*gsl_matrix * m*) [Function]

行列 m を、その転置行列で置き換える。行列 m は正方行列でなければならない。

8.4.10 行列の演算

実数および複素数の行列に対して、以下の演算が定義されている。

`int gsl_matrix_add (`*gsl_matrix * a, const gsl_matrix * b*`)` [Function]

行列 b の要素の値を行列 a の要素に $a'(i,j) = a(i,j) + b(i,j)$ のようにして加える。二つの行列の次元は同じでなければならない。

`int gsl_matrix_sub (`*gsl_matrix * a, const gsl_matrix * b*`)` [Function]

行列 b の要素の値を行列 a の要素から $a'(i,j) = a(i,j) - b(i,j)$ のようにして減ずる。二つの行列の次元は同じでなければならない。

`int gsl_matrix_mul_elements (`*gsl_matrix * a, const gsl_matrix * b*`)` [Function]

行列 b の要素の値を行列 a の要素に $a'(i,j) = a(i,j) * b(i,j)$ のようにして乗じる。二つの行列の次元は同じでなければならない。

`int gsl_matrix_div_elements (`*gsl_matrix * a, const gsl_matrix * b*`)` [Function]

行列 b の要素の値で行列 a の要素を $a'(i,j) = a(i,j)/b(i,j)$ のようにして除する。二つの行列の次元は同じでなければならない。

`int gsl_matrix_scale (`*gsl_matrix * a, const double x*`)` [Function]

定数値 x を行列 a のすべての要素に $a'(i,j) = xa(i,j)$ のようにして乗じる。

`int gsl_matrix_add_constant (`*gsl_matrix * a, const double x*`)` [Function]

定数値 x を行列 a のすべての要素に $a'(i,j) = a(i,j) + x$ のようにして加える。

8.4.11 行列中の最大、最小要素の探索

以下の演算は実数行列に対してのみ定義されている。

`double gsl_matrix_max (`*const gsl_matrix * m*`)` [Function]

行列 m 中の要素で最大のものの値を返す。

`double gsl_matrix_min (`*const gsl_matrix * m*`)` [Function]

行列 m 中の要素で最小のものの値を返す。

`void gsl_matrix_minmax (`*const gsl_matrix * m, double * min_out, double * max_out*`)` [Function]

行列 m 中の要素で最小および最大のものの値を、*min_out* および *max_out* に入れて返す。

`void gsl_matrix_max_index (const gsl_matrix * m, size_t * imax, size_t * jmax)` [Function]

行列 m 中の要素で最大のもの添え字の値を、引数 *imax* および *jmax* に入れて返す。同じ値のものが複数あるときは、行優先で探索して最初に見つかったものを返す。

`void gsl_matrix_min_index (const gsl_matrix * m, size_t * imin, size_t * jmin)` [Function]

行列 m 中の要素で最小のもの添え字の値を、引数 *imin* および *jmin* に入れて返す。同じ値のものが複数あるときは、行優先で探索して最初に見つかったものを返す。

`void gsl_matrix_minmax_index (const gsl_matrix * m, size_t * imin, size_t * jmin, size_t * imax, size_t * jmax)` [Function]

行列 m 中の要素で最小および最大のもの添え字の値を、それぞれ引数 (*imin*, *jmin*)、(*imax*, *jmax*) に入れて返す。同じ値のものが複数あるときは、行優先で探索して最初に見つかったものを返す。

8.4.12　行列の属性

`int gsl_matrix_isnull (const gsl_matrix * m)` [Function]
`int gsl_matrix_ispos (const gsl_matrix * m)` [Function]
`int gsl_matrix_isneg (const gsl_matrix * m)` [Function]
`int gsl_matrix_isnonneg (const gsl_matrix * m)` [Function]

それぞれ、行列 m 中のすべての要素の値が 0 の時、0 でない正のとき、0 でない負の時、非負のときに 1 を、そうでないときは 0 を返す。行列が正定値かどうかは、コレスキー分解（13.5 節を参照）で調べることができる。

8.4.13　行列のプログラム例

以下に示すプログラム例では、関数 `gsl_matrix_alloc`、`gsl_matrix_set`、`gsl_matrix_get` を使って行列を確保、初期化、読み込みを行う。

```
#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++) /* わざと範囲外にアクセス */
        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;
}
```

以下に例示したプログラムの出力を示す。プログラムの最後のループは、gsl_matrix_get での行列 m の範囲確認でエラーを出してトラップするためのものである。

```
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.c:13: ERROR: first index out of range
Default GSL error handler invoked.
Aborted (core dumped)
```

次のプログラムでは行列をファイルに書き出す。

```
#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);
}
```

このプログラムを実行すると、‘test.dat’ というファイルに m の要素の値がバイナリ形式で書き込まれる。それを関数 gsl_matrix_fread で読み込むと、元の行列と完全に同じものが得られる。

以下のプログラムではベクトル像の使用例として、行列の列ノルムの計算を示す。

```
#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 %d, norm = %g\n", j, d);
        }

        gsl_matrix_free(m);
        return 0;
    }
```

以下にプログラムの出力を示す。

```
$ ./a.out
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
```

結果の正しさは GNU OCTAVE を使って確認することができる。

```
$ 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 参考文献

GSL でのブロック、ベクトル、行列オブジェクトは C++ の valarray にならっている。これは以下の参考文献で解説されている。

- Bjarne Stroustrup, *The C++ Programming Language* (3rd Ed), Section 22.4 Vector Arithmetic, Addison-Wesley, ISBN 0-201-88954-4 (1997).

上記の文献の和訳が以下の書籍として出版されており、またその正誤表が WWW で見られる。原著の正誤表もそこからたどれる。

- Bjarne Stroustrup (長尾高弘訳), プログラミング言語 C++ (アスキーアジソンウェスレイシリーズ – Ascii Addison Wesley programming series), アジソンウェスレイパブリッシャーズジャパン, ISBN 978-4756118950 (1998).
  正誤表: `http://www.longtail.co.jp/errata/`

第9章　置換

この章では、置換 (permutation) を生成、操作する関数について説明する。置換 p は 0 から $n-1$ までの n 個の整数の要素を持つ配列で表現され、配列の各要素の値 p_i は配列中で一つだけ必ず含まれる。「置換 p をベクトル v に適用する」と、ベクトル v から $v'_i = v_{p_i}$ として新しいベクトル v' が作られる。たとえば要素数 4 のベクトルの末尾の二つの要素を入れ替える操作は、配列 $(0, 1, 3, 2)$ で表される。同様に、恒等置換 (identity permutation、適用しても元のベクトルと同じものが得られる置換) は $(0, 1, 2, 3)$ で表される。

線形代数関連のルーチンで扱われる置換は、行列の列を入れ替える操作に相当する。したがってベクトルに置換を適用する時、そのベクトルは列ベクトルではなく、置換は行ベクトルに対して $v' = vP$ のように表される操作であると考えなければならない。

この章の関数はヘッダファイル 'gsl_permutation.h' で宣言されている。

9.1　置換構造体

置換オブジェクトは、その大きさと置換を表す配列へのポインタを持つ構造体で保持される。配列の各要素の型はすべて size_t である。gsl_permutation は以下のような定義である。

```
typedef struct {
    size_t size;
    size_t * data;
} gsl_permutation;
```

9.2　置換の確保

gsl_permutation * gsl_permutation_alloc (*size_t* **n**)　[Function]

大きさ n の置換を新たに生成する。初期化は行われないので、置換の内容は生成時には不定である。一方、gsl_permutation_calloc は恒等置換を生成する。指定された大きさの置換を保持するだけのメモリが確保できなかったときは、NULL ポインタを返す。

gsl_permutation * gsl_permutation_calloc (*size_t* **n**)　[Function]

大きさ n の恒等置換を新たに生成する。指定された大きさの置換を保持するだけのメモリが確保できなかったときは、NULL ポインタを返す。

`void gsl_permutation_init (`*gsl_permutation * `p`*`)` [Function]

渡された置換 p を恒等置換 $(0, 1, 2, \ldots, n-1)$ にする。

`void gsl_permutation_free (`*gsl_permutation * `p`*`)` [Function]

置換 p のメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_permutation_memcpy (`*gsl_permutation * `dest`, const gsl_permutation * `src`*`)` [Function]

置換 *src* の要素を置換 *dest* にコピーする。二つの置換の大きさは同じでなければならない。

9.3　置換の要素の参照と操作

置換を操作する以下の関数が用意されている。

`size_t gsl_permutation_get (`*const gsl_permutation * `p`, const size_t `i`*`)` [Function]

置換 p の i 番目の要素を返す。i が 0 から $n-1$ の範囲からはずれている場合はエラー・ハンドラーが呼ばれ、0 を返す。`HAVE_INLINE` が定義されているときは、インライン展開される。

`int gsl_permutation_swap (`*gsl_permutation * `p`, const size_t `i`, const size_t `j`*`)` [Function]

置換 p の i 番目の要素と j 番目の要素を入れ替える。

9.4　置換の属性

`size_t gsl_permutation_size (`*const gsl_permutation * `p`*`)` [Function]

置換 p の大きさを返す。

`size_t * gsl_permutation_data (`*const gsl_permutation * `p`*`)` [Function]

置換 p の要素を保持する配列へのポインタを返す。

`int gsl_permutation_valid (`*const gsl_permutation * `p`*`)` [Function]

置換 p が意味を持つものかどうかを判定する。n 個の要素に 0 から $n-1$ までの整数が一つずつ含まれていればよい。

9.5 置換を扱う関数

`void gsl_permutation_reverse (gsl_permutation * p)` [Function]

置換 p の要素の並びを逆にする。

`int gsl_permutation_inverse (gsl_permutation * inv, const gsl_permutation * p)` [Function]

置換 p の逆置換を計算し、inv に入れて返す。

`int gsl_permutation_next (gsl_permutation * p)` [Function]

置換 p を辞書順で次の置換に置き換えて GSL_SUCCESS を返す。辞書順で次になる置換が作れない場合は GSL_FAILURE を返し、p は変化しない。恒等置換からはじめてこの関数を繰り返し適用していくことで、すべてのあり得る置換を得ることができる。

`int gsl_permutation_prev (gsl_permutation * p)` [Function]

置換 p を辞書順で一つ前の置換に置き換えて GSL_SUCCESS を返す。辞書順で前の置換がない場合には GSL_FAILURE を返し、p は変化しない。

9.6 置換の適用

`int gsl_permute (const size_t * p, double * data, size_t stride, size_t n)`[Function]

置換 p を刻み幅 $stride$ で大きさ n の配列 $data$ に適用する。

`int gsl_permute_inverse (const size_t * p, double * data, size_t stride, size_t n)` [Function]

置換 p の逆置換を、刻み幅 $stride$ で大きさ n の配列 $data$ に適用する。

`int gsl_permute_vector (const gsl_permutation * p, gsl_vector * v)` [Function]

置換 p をベクトル v に適用する。ベクトルは行ベクトルと想定され、置換の適用は置換行列 P を右から $v' = vP$ のようにして適用することになる。この置換行列 P の j 番目の列は単位行列の p_j 番目の列である。置換 p とベクトル v は同じ大きさでなければならない。

`int gsl_permute_vector_inverse (const gsl_permutation * p, gsl_vector * v)` [Function]

置換 p の逆置換を行ベクトル v に、右から $v' = vP^T$ のようにして適用する。逆置換は、転置行列による置換と同じことである。この置換行列 P の j 番目の列は単位行列の p_j 番目の列である。置換 p とベクトル v は同じ大きさでなければならない。

int gsl_permutation_mul (*gsl_permutation * p*, *const gsl_permutation * pa*, *const gsl_permutation * pb*) [Function]

二つの置換 *pa* と *pb* を一つの置換 $p = pa.pb$ にまとめる。得られる置換 p を適用することは、最初に *pb* を適用した後に *pa* を適用するのと同じことである。

9.7 置換のファイル入出力

GSL では、置換をバイナリあるいは書式付きテキストとしてファイルに対して読み書きする関数を用意している。

int gsl_permutation_fwrite (*FILE * stream*, *const gsl_permutation * p*)[Function]

置換 p の要素をバイナリ・フォーマットでファイル *stream* に書き込む。書き込みに際してエラーが発生したときは GSL_EFAILED を返す。実行中のアーキテクチャーに依存した形式なので、移植性は低い。

int gsl_permutation_fread (*FILE * stream*, *gsl_permutation * p*) [Function]

置換 p の要素をバイナリ・フォーマットとしてファイル *stream* から読み込む。この関数は置換の大きさから読み込むバイト数を決定するため、置換 p はあらかじめ、正しい大きさで確保されていなければならない。読み込みに際してエラーが発生したときは GSL_EFAILED を返す。データ形式は、実行中のアーキテクチャの形式で保存されたものと仮定してている。

int gsl_permutation_fprintf (*FILE * stream*, *const gsl_permutation * p*, *const char * format*) [Function]

置換 p の要素を一行ずつ、size_t の変数に適した指定の書式 *format* にしたがってファイル *stream* に書き出す。ISO C99 規格では記述子 z が size_t を表すので、"%zu\n" と書くのがよい[1]。書き込みに際してエラーが発生したときは GSL_EFAILED を返す。

int gsl_permutation_fscanf (*FILE * stream*, *gsl_permutation * p*) [Function]

ファイル *stream* から置換 p に要素を読み込む。この関数は置換の大きさから読み込む数値の個数を決定するため、置換 p はあらかじめ、正しい大きさで確保されていなければならない。読み込むときにエラーが発生した場合は GSL_EFAILED を返す。

9.8 巡回置換

置換は一般に、線形置換 (linear form) または巡回置換 (cyclic form) の二つの形式で表すことができる。ここで説明する関数はこの二つの形式の間の変換を行うものである。線形表現とは上述して

[1] ISO C99 より前の GNU C library では "Z" であった。

きたように、添え字の置き換え方を表したベクトルである。巡回表現とは、置換中に出てきた要素を、次に出てくる要素の場所に置くという表現であり、単に巡回 (cycle) とも呼ぶ。

たとえば巡回 (1 2 3) では、先頭の要素を 2 番目に、2 番目の要素を 3 番目に、3 番目の要素を先頭に移動する、という巡回的な置き換えを表す。要素を複数の集合に分けた巡回は、それぞれ独立に適用することができる。たとえば (1 2 3) (4 5) は巡回 (1 2 3) と、要素 4 と 5 を入れ替える巡回 (4 5) に分けられる。一つの要素からなる巡回は適用しても何も変化させない。これはシングルトン (singleton、単集合) と呼ばれる。

すべての置換は複数の巡回に分解することができる。その分解は一般に一意に定まらないが、巡回の要素を特定の順序で並べることで正規形 (canonical form) として一意に定めることができる。GSL での正規形の実装は、クヌース (*The Art of Computer Programming* 3rd Ed., Vol 1, 1997, p.178) による定義によっている。

クヌースの正規型を得るには、以下のようにする。

1. すべての単集合巡回を列挙する。
2. 各巡回について、もっとも小さなものを先頭に移す。
3. 先頭の要素の降順に巡回を並べる。

たとえば線形表現 (2 4 3 0 1) は正規型では (1 4) (0 2 3) となる。これは要素 1 と 4 を入れ替え、0 と 2 と 3 を一つずつずらす置換である。

正規型で表された置換は、各巡回からカッコを取った形に変形することができる。またカッコを取ることで、異なる置換の線形表現ととらえることもできる。上の例では置換 (2 4 3 0 1) は (1 4 0 2 3) になる。こういった変換は、置換の理論分野では広く応用されている。

`int gsl_permutation_linear_to_canonical` (*gsl_permutation * q*, *const gsl_permutation * p*) [Function]

置換 p の正規型を計算し引数 q に入れて返す。

`int gsl_permutation_canonical_to_linear` (*gsl_permutation * p*, *const gsl_permutation * q*) [Function]

正規型の置換 q を線形表現に戻して引数 q に入れて返す。

`size_t gsl_permutation_inversions` (*const gsl_permutation * p*) [Function]

置換 p に含まれる、二要素の「逆向き (inversion)」の個数を数える。「逆向き」とは、順序関係が逆転して並んでいる二つの数の組である。たとえば置換 2031 は (2, 0)、(2, 1)、(3, 1) に相当する 3 つの「逆向き」を含んでいる。恒等置換には「逆向き」は含まれていない。

`size_t gsl_permutation_linear_cycles` (*const gsl_permutation * p*) [Function]

線形表現で与えられる置換 p に含まれる巡回の個数を数える。

`size_t gsl_permutation_canonical_cycles` (*const gsl_permutation * q*) [Function]

正規形で与えられる置換 q に含まれる巡回の個数を数える。

9.9　例

以下のプログラムでは、ランダムな置換を生成し、その逆置換を表示する。

```
#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;
    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;
}
```

以下にプログラムの出力を示す (乱数によって、initial permutation 以外は異なった値になることもある)。

```
bash$ ./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
```

ランダムに生成された置換 `p[i]` とその逆置換 `q[i]` は恒等置換をなす、`p[q[i]] = i` という関係があり、これを使って逆置換の検証ができる。

次のプログラムは、恒等置換から初めて、三次の置換をすべて列挙する。

```
#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;
}
```

以下にプログラムの出力を示す。

```
bash$ ./a.out
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

置換はすべてで 6 つあり、辞書順に生成されている。順序を逆にするには最後の置換 (恒等置換を逆にしたもの) からはじめて、gsl_permutation_next の代わりに gsl_permutation_prev を使えばよい。

9.10　参考文献

置換についてはクヌースの文献に幅広く述べられている。

- Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching* 3rd Ed., Vol 3, Addison-Wesley, ISBN 978-0201896855 (1997).

正規型の定義については、以下を参照のこと。

- Donald E. Knuth, *The Art of Computer Programming: Fundamental Algorithms* 3rd Ed, Vol. 1, Section 1.3.3 “An Unusual Correspondence”, p.178–179, Addison-Wesley, ISBN 978-0201896831 (1997).

第10章　組み合わせ

この章では組み合わせ (combination) を生成、操作する関数について説明する。組み合わせ c は k 個の整数による配列で表現され、各要素 c_i は 0 から $n-1$ の値を取り、重複することはない (が含まれない値はありえる)。組み合わせ c は、要素数 n のベクトルから k 個の要素を選ぶときの、選ぶ要素を表す添え字である。ある集合から k 個の要素を選ぶことで生成される部分集合すべてについて、なにかの操作を行いたいときに使うことができる。

この章で説明する関数はヘッダファイル 'gsl_combination.h' で宣言されている。

10.1　組み合わせ構造体

組み合わせは、n、k、組み合わせ配列へのポインタの三つの要素を持つ構造体に保持される。組み合わせ配列の要素の型は size_t であり、昇順に格納される。gsl_combination 構造体は以下のように定義されている。

```
typedef struct {
    size_t n;
    size_t k;
    size_t *data;
} gsl_combination;
```

10.2　組み合わせの確保

gsl_combination * gsl_combination_alloc (*size_t* **n**, *size_t* **k**)　[Function]

引数 n、k で指定される組み合わせのためのメモリを確保する。組み合わせは初期化されないため、確保時の組み合わせの内容は不定である。関数 gsl_combination_calloc を使うと、メモリを確保すると同時に、あり得る組み合わせのうち辞書順で最初のものになるよう初期化される。十分なメモリが確保できないときは、null ポインタを返す。

gsl_combination * gsl_combination_calloc (*size_t* **n**, *size_t* **k**)　[Function]

引数 n、k で指定される組み合わせのためのメモリを確保し、辞書順で最初になる組み合わせになるよう初期化する。十分なメモリが確保できないときは、null ポインタを返す。

void gsl_combination_init_first (*gsl_combination* * **c**)　[Function]

組み合わせ c を初期化し、辞書順で最初になる組み合わせ、たとえば $(0, 1, 2, \ldots, k-1)$ にする。

`void gsl_combination_init_last (gsl_combination * c)` [Function]

組み合わせ c を初期化し、辞書順で最後になる組み合わせ、たとえば $(n-k, n-k+1, \ldots, n-1)$ にする。

`void gsl_combination_free (gsl_combination * c)` [Function]

組み合わせ c のメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_combination_memcpy (gsl_combination * dest, const gsl_combination * src)` [Function]

組み合わせ *src* の要素を組み合わせ *dest* にコピーする。二つの組み合わせは同じ大きさでなければならない。

10.3　組み合わせの要素の参照と操作

以下の関数を使って、組み合わせの要素の参照と操作ができる。

`size_t gsl_combination_get (const gsl_combination * c, const size_t i)` [Function]

組み合わせ c の i 番目の要素の値を返す。i が 0 から $k-1$ の範囲内でなければ、エラー・ハンドラーが呼ばれ、0 を返す。`HAVE_INLINE` が定義されているときは、インライン展開される。

10.4　組み合わせの属性

`size_t gsl_combination_n (const gsl_combination * c)` [Function]

組み合わせ c のパラメータ n の値を返す。

`size_t gsl_combination_k (const gsl_combination * c)` [Function]

組み合わせ c のパラメータ k の値を返す。

`size_t * gsl_combination_data (const gsl_combination * c)` [Function]

組み合わせ c の要素を保持する配列へのポインタを返す。

`int gsl_combination_valid (gsl_combination * c)` [Function]

組み合わせ c が意味を持つかどうかを判定する。k 個の要素すべてが 0 から $n-1$ の範囲で、その範囲内の値は含まれないか、1 回だけ含まれていればよい。値は昇順に並べられていなければならない。

10.5 組み合わせを扱う関数

`int gsl_combination_next` (*gsl_combination* * **c**) [Function]

組み合わせ c を辞書順で次の組み合わせに書き換え、GSL_SUCCESS を返す。辞書順で次の組み合わせがないときは GSL_FAILURE を返し、c は書き換えられない。最初の組み合わせからはじめて、この関数を次々と適用することで、すべての組み合わせを生成することができる。

`int gsl_combination_prev` (*gsl_combination* * **c**) [Function]

組み合わせ c を辞書順で一つ前の組み合わせに書き換え、GSL_SUCCESS を返す。辞書順で次の組み合わせがないときは GSL_FAILURE を返し、c は書き換えられない。

10.6 組み合わせのファイル入出力

GSL では、組み合わせをバイナリ・データまたは書式付きテキストとしてファイルから読み込み、あるいはファイルに書き込む関数が実装されている。

`int gsl_combination_fwrite` (*FILE* * **stream**, *const gsl_combination* * **c**) [Function]

組み合わせ c の要素をファイル *stream* にバイナリ形式で書き込む。書き込みに際してエラーが発生したときは、GSL_FAILURE を返す。データ形式は実行中のアーキテクチャに依存するので、移植性は低い。

`int gsl_combination_fread` (*FILE* * **stream**, *gsl_combination* * **c**) [Function]

組み合わせ c の要素をファイル *stream* からバイナリ形式で読み込む。読み込むバイト数は c の大きさから決定されるため、組み合わせ c はあらかじめ、正しい値の n と k で確保されていなければならない。読み込みに際してエラーが発生したときは、GSL_FAILURE を返す。データは実行中のアーキテクチャによる形式で書き込まれたもの、と仮定される。

`int gsl_combination_fprintf` (*FILE* * **stream**, *const gsl_combination* * **c**, *const char* * **format**) [Function]

組み合わせ c の要素をファイル *stream* に一行ずつ、size_t の変数に対して適切に指定された *format* にしたがった書式で書き込む。ISO C99 規格では z が size_t を表現するため、"%zu\n" などとするのがよい[1]。書き込みに際してエラーが発生したときは、GSL_FAILURE を返す。

`int gsl_combination_fscanf` (*FILE* * **stream**, *gsl_combination* * **c**) [Function]

[1] ISO C99 より前の GNU C library では "Z" であった。

組み合わせ c の要素をファイル *stream* から読み込む。読み込む数値の個数は c の大きさから決定されるため、組み合わせ c はあらかじめ、正しい値の n と k で確保されていなければならない。読み込みに際してエラーが発生したときは、GSL_FAILURE を返す。

10.7　例

以下のプログラムは集合 $\{0,1,2,3\}$ のすべての部分集合を、その大きさの順に並べて出力する。大きさが同じ時は辞書順に並べる。

```
#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 {
            printf("{");
            gsl_combination_fprintf(stdout, c, " %u");
            printf(" }\n");
        } while (gsl_combination_next(c) == GSL_SUCCESS);
        gsl_combination_free(c);
    }

    return 0;
}
```

以下にプログラムの出力を示す。

```
bash$ ./a.out
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 }
```

部分集合は計 16 個あり、それらはその大きさと辞書順で整列されている。

10.8 参考文献

組み合わせに関する解説は、以下の文献にある。

- Donald L. Kreher, Douglas R. Stinson, *Combinatorial Algorithms: Generation, Enumeration and Search*, CRC Press LLC, ISBN 084933988X (1998).

第11章　ソート

この章ではソーティング (sorting/sort、整列) を直接、および (配列の添字を並べ替えることで) 間接的に行う関数について説明する。どの関数もアルゴリズムはヒープソート (heapsort) である。ヒープソートの計算量は $O(N \log N)$ であり、作業領域を別途必要とすることはなく、いつでもそれなりによい性能を発揮する。もっとも速度が遅くなる場合（すでにソートされているデータに対するソーティング）でも、平均的あるいはもっとも速い場合と、大きくは変わらない。しかしヒープソートは、同じ値を持つ要素の順序が保たれない「非安定 (unstable)」なアルゴリズムである。ここで実装している関数では、同じ値の要素の順序がどのように変わってしまうかはデータによるが、異なるプラットフォーム上で実行しても同じ結果となる。

11.1　ソートのためのオブジェクト

以下の関数は C 言語の標準ライブラリ関数の `qsort` と同等の機能を持つが、`qsort` を持たないシステムのために用意しているもので、`qsort` があるときにそれを置き換えることが目的ではない。`qsort` は、同じ値の要素の順序を保持する安定 (stable) な整列法であり、また整列されていないデータに対してはヒープソートより平均的に速いため、使える場合は `qsort` を使うべきである。GNU C Library Reference Manual で `qsort` の解説を読むことができる。

この章で説明する関数はヘッダファイル '`gsl_heapsort.h`' で宣言されている。

`void gsl_heapsort (`*void * **array**, size_t **count**, size_t **size**, gsl_comparison_fn_t **compare***`)` [Function]

要素の大小を比較する関数 *compare* を使って、*count* 個の大きさ *size* の要素を持つ配列 *array* を昇順に整列する。比較関数の型は以下のように定義される。

```
int (*gsl_comparison_fn_t) (const void * a, const void * b)
```

比較関数は、一番目の引数 a が二番目の引数 b よりも小さいときには負の整数を、引数の値が二つとも同じ時には 0 を、一番目の引数が二番目の引数よりも大きいときには正の整数を返すものを指定する。

例えば、実数をその値の昇順に整列するためには、以下のような比較関数を使えばよい。

```
int compare_doubles (const double * a, const double * b)
{
    if      (*a > *b) return  1;
    else if (*a < *b) return -1;
```

```
    else              return  0;
}
```

整列を行うためには、以下のようにしてヒープソート関数を呼び出す。

```
gsl_heapsort (array, count, sizeof(double), compare_doubles);
```

qsort と違って、ヒープソートのポインタによる演算では安定な整列を行うことはできない。比較関数の中で同じ値を持つ要素のポインタをどのように工夫して比較しても、ヒープソートの内部でデータの並べ替えを行うため、元データの並び順は変わってしまう。

int gsl_heapsort_index (*size_t* * ***p***, *const void* * ***array***, *size_t* ***count***, *size_t* ***size***, *gsl_comparison_fn_t* ***compare***) [Function]

要素の大小を比較する関数 *compare* を使って、*count* 個の大きさ *size* の要素を持つ配列 *array* を昇順に、間接的に整列する。並べ替えを表す置換が大きさ n の配列 p に入れて返される。p の要素 p[i] は、配列 *array* を並べ替えて上書きするときに、i 番目になるのは元の配列のどの要素か、を表す。つまり、p の最初の要素は *array* 中でもっとも小さな要素を示し、p の最後の要素は *array* 中で最も大きな要素を示す。配列 *array* そのものは変化しない。

11.2 ベクトルのソート

以下の関数は、直接、および (配列の添字を並べ替えることで) 間接的にベクトルや配列の要素を整列する。ベクトルの成分は実数でも整数でもよく、GSL の通常の規則にしたがった、それぞれの型に対応した名前の関数で整列される。例えば、float の配列をソートする関数は gsl_sort_float と gsl_sort_float_index である。それに対応するベクトルをソートする関数は gsl_sort_vector_float と gsl_sort_vector_float_index である。そのプロトタイプ宣言はそれぞれ、ヘッダファイル 'gsl_sort_float.h' と 'gsl_sort_vector_float.h' にある。配列、ベクトルそれぞれに対するすべてのソート関数のプロトタイプ宣言は、ヘッダファイル 'gsl_sort.h' と 'gsl_sort_vector.h' にある。

複素数の配列やベクトルに対しては、複素数の順序づけが定義されないため、用意されていない。複素数のベクトルを整列するには、まずその絶対値を要素とする実数ベクトルを計算し、その実数ベクトルを間接的に (添え字で) 整列するとよい。返された添え字はそのまま、元の複素数配列を整列する順序を表す。

void gsl_sort (*double* * ***data***, *size_t* ***stride***, *size_t* ***n***) [Function]

要素数 n の配列 *data* を、刻み幅 *stride* で数値の昇順に整列する。

void gsl_sort_vector (*gsl_vector* * ***v***) [Function]

ベクトル v の要素を数値の昇順に整列する。

`int gsl_sort_index` (*size_t* * **p**, *const double* * **data**, *size_t* **stride**, *size_t* **n**) [Function]

要素数 n の配列 *data* を、刻み幅 *stride* で数値の昇順に間接的に整列する。結果は添字の置換として p に入れて返す。配列 p は、要素数 n の置換を保持するのに十分な大きさであらかじめ確保しておく。p の要素 `p[i]` は、配列 *data* を並べ替えて上書きするときに、`i` 番目になるのは元の配列のどの要素か、を表す。元の配列 *data* は変化しない。

`int gsl_sort_vector_index` (*gsl_permutation* * **p**, *const gsl_vector* * **v**) [Function]

ベクトル v の要素を昇順に間接的に整列するための置換を、p に入れて返す。p の要素は、ベクトルの要素を整列して元のベクトルに上書きした場合に、新しいベクトル中の要素の元のベクトル中での位置を表す添え字である。置換 p は、ベクトルの要素を並べ替えて上書きするときに、どのように置き換えるかを表す。p の一番目の要素は v の要素のうち最小のもの、p の最後の要素は v 中の最大の要素が、それぞれ整列後にどの順位になるかを表す。元のベクトル v は変化しない。

11.3 最小または最大の複数の要素の取り出し

この節の関数は、N 個の要素を含むデータの集合から k 個の最大または最小の要素を取り出すものである。これは、取り出される要素数 k の部分集合がデータ集合全体に比べてずっと小さい場合に適した、計算量のオーダーが $O(kN)$ の挿入ソートアルゴリズム (direct insertion algorithm) を使っている。例えば、点数が 1,000,000 個のデータから上位 10 個のデータを選び出すというような目的に適している。同じデータから 100,000 個を取り出すようなことには、あまり適していない。取り出す個数がある程度多い場合は、元のデータ集合を $O(N \log N)$ のアルゴリズムで直接整列してから、上または下から順にほしい個数だけ数えて最大または最小の要素を得る方が速いだろう。

`void gsl_sort_smallest` (*double* * **dest**, *size_t* **k**, *const double* * **src**, *size_t* **stride**, *size_t* **n**) [Function]

大きさ n、刻み幅 *stride* の配列 *src* の要素のうち、小さいもの k 個を数値の昇順に並べて配列 *dest* にコピーする。部分集合の大きさ k は n 以下でなければならない。元のデータ *src* は変化しない。

`void gsl_sort_largest` (*double* * **dest**, *size_t* **k**, *const double* * **src**, *size_t* **stride**, *size_t* **n**) [Function]

大きさ n、刻み幅 *stride* の配列 *src* の要素のうち、大きいもの k 個を数値の降順に並べて配列 *dest* にコピーする。部分集合の大きさ k は n 以下でなければならない。元のデータ *src* は変化しない。

void gsl_sort_vector_smallest (*double * dest*, *size_t k*, *const gsl_vector * v*) [Function]
void gsl_sort_vector_largest (*double * dest*, *size_t k*, *const gsl_vector * v*) [Function]

ベクトル v の要素のうち最大または最小の k 個を *dest* にコピーする。k はベクトルの大きさ以下でなければならない。

以下の関数は、データ集合中の最大または最小の k 個の要素を示す添え字を返す。

void gsl_sort_smallest_index (*size_t * p*, *size_t k*, *const double * src*, *size_t stride*, *size_t n*) [Function]

大きさ n、刻み幅 *stride* の配列 *src* の要素のうち、小さいもの k 個の添え字を配列 p に入れて返す。この添え字は要素の持つ値の昇順に並べられる。k は n 以下でなければならない。元のデータ *src* は変化しない。

void gsl_sort_largest_index (*size_t * p*, *size_t k*, *const double * src*, *size_t stride*, *size_t n*) [Function]

大きさ n、刻み幅 *stride* の配列 *src* の要素のうち、大きいもの k 個の添え字を配列 p に入れて返す。この添え字は要素の持つ値の降順に並べられる。k は n 以下でなければならない。元のデータ *src* は変化しない。

void gsl_sort_vector_smallest_index (*size_t p*, *size_t k*, *const gsl_vector * v*) [Function]
void gsl_sort_vector_largest_index (*size_t p*, *size_t k*, *const gsl_vector * v*) [Function]

ベクトル v の要素のうち、小さいものまたは大きいもの k 個の添え字を配列 p に入れて返す。この添え字は要素の持つ値の降順に並べられる。k はベクトル v の大きさ以下でなければならない。

11.4 順位の計算

要素の順位 (rank) とは、整列されたデータの中での順番 (order) のことである。順位は添え字に対する置換 p の逆写像でもあり、以下のアルゴリズムで得ることができる。

```
for (i = 0; i < p->size; i++) {
    size_t pi = p->data[i];
    rank->data[pi] = i;
}
```

順位は、gsl_permutation_inverse(rank,p) 関数を使って直接得ることができる。

以下の関数では、ベクトル v の各要素の順位を表示する。

```
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]);
    }
    gsl_permutation_free(perm);
    gsl_permutation_free(rank);
}
```

11.5 例

以下の例では、ベクトル v の要素を、置換 p を使って昇順に表示する。

```
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);
}
```

次の例では、関数 gsl_sort_smallest を使って、配列に保持されている 100000 個の乱数値から最小の 5 個を取り出す。

```
#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);

    for (i = 0; i < N; i++) x[i] = gsl_rng_uniform(r);

    gsl_sort_smallest(small, k, x, 1, N);
    printf("%d smallest values from %d\n", k, N);
    for (i = 0; i < k; i++) printf ("%d: %.18f\n", i, small[i]);

    free(x);
    free(small);
    gsl_rng_free(r);
    return 0;
}
```

このプログラムを実行すると、5 個の最小値が昇順に整列されて出力される。

```
$ ./a.out
5 smallest values from 100000
0: 0.000003489200025797
1: 0.000008199829608202
2: 0.000008953968062997
3: 0.000010712770745158
4: 0.000033531803637743
```

11.6　参考文献

整列法については、クヌースの文献により広い説明がある。

- Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching* 3rd Ed., Vol. 3, Addison-Wesley, ISBN 978-0201896855 (1997).

ヒープソートは以下の本に説明がある。

- Robert Sedgewick, *Algorithms in C*, Addison-Wesley, ISBN 0201514257.

また GNU C Library Reference Manual は、以下のサイトで様々な形式のファイルで提供されている。残念ながら、全体の和訳はないようである。

- `http://www.gnu.org/software/libc/manual/`

第12章　BLAS の利用

基本線形代数ルーチン集 BLAS (The Basic Linear Algebra Subprograms) は、ベクトルと行列を扱う基礎的な演算のサブルーチン集であり、これを使うことでより高次の、最適化された線形代数の計算機能を実装することができる。

このライブラリでは、“CBLAS” (C 言語で書かれた BLAS の標準ライブラリ) を直接操作するための低レベルの関数 (low level function、アルゴリズムとしての抽象化レベルが低いという意味) と、GSL で定義するベクトルや行列を操作するための高レベルの関数の両方を用意している。GSL のベクトルと行列のオブジェクトに対する比較的単純な演算は、‘gsl_blas.h’ で宣言されている高レベルの関数を使うとよい。ほとんどの場合はこれを使うのが安全である。GSL の行列は、疎行列 (sparse matrix) でも帯行列 (band matrix) でもない一般の密な行列 (dense-storage) としてだけ実装されているため、高レベル関数も BLAS の密行列に対応した関数に対応するもののみである。帯行列に特化した形式 (帯形式、band-format) と疎行列を圧縮して格納する形式 (パック形式、packed-format) のそれぞれに対応した BLAS の機能は、低レベルの CBLAS の関数ですべて使うことができる。

低レベルの gsl_cblas の関数の宣言は ‘gsl_cblas.h’ にある。これは古い実装の BLAS に対する C 言語による API の仕様を定めた BLAS 技術フォーラムの暫定標準に対応している。他の CBLAS に準拠する実装が使える場合は、GSL で用意している CBLAS の代わりにそれが使える。FORTRAN の BLAS ライブラリしかない場合でも、CBLAS 変換ラッパーを使ってその FORTRAN ライブラリを CBLAS として使うことができる。そのラッパーは CBLAS の暫定標準に含まれており、Netlib から入手できる。CBLAS に含まれるすべての関数のリストを付録に挙げる (付録 D「GSL CBLAS ライブラリ」、503 ページ参照)。

BLAS に用意されている演算は、3 つのレベルに分かれている。以下では、x, y がベクトルで、A, B, C が行列、α, β が定数である。

Level 1 ベクトル同士の演算。$y = \alpha x + y$ など。

Level 2 行列とベクトルの演算。$y = \alpha A x + \beta y$ など。

Level 3 行列同士の演算。$C = \alpha AB + C$ など。

各ルーチンの名前には、演算の種類、行列の種類、演算精度を表す文字ががつけられている。その中で、以下のものが代表的である。

DOT スカラー積。$x^T y$ など。

AXPY ベクトルの和。$\alpha x + y$ など。

MV 行列とベクトルの積。Ax など。

SV 行列とベクトルの積の解。$inv(A)x$ など。

MM 行列同士の積。AB など。

SM 行列同士の積の解。$inv(A)B$ など。

行列の種類には以下のものがある。

GE 一般的な行列 (General)

GB 一般的な帯行列 (General band)

SY 対称行列 (symmetirc)

SB 対称帯行列 (symmetric band)

SP 対称パック行列 (symmetric packed)

HE エルミート行列 (Hermitian)

HB エルミート帯行列 (Hermitian band)

HP エルミートパック行列 (Hermitian packed)

TR 三角行列 (triangular)

TB 三角帯行列 (triangular band)

TP 三角パック行列 (triangular packed)

各演算は、四種類の精度で用意されている。

S 単精度実数

D 倍精度実数

C 単精度複素数

Z 倍精度複素数

したがって例えば、SGEMM という名前は「単精度で一般の行列同士の積」、ZGEMM は「倍精度複素数での一般の行列同士の積」という演算を表す。

12.1 GSL から blas を利用する関数

GSL では、一般の密な (疎でない) ベクトルと行列のオブジェクトを C 言語の各組込み型に対して用意している。GSL で用意している各種演算を行う関数は、これら GSL のオブジェクトに BLAS で定義されている演算を適用する。これらの API は 'gsl_blas.h' ファイルにある。

12.1.1 Level 1

`int gsl_blas_sdsdot` (*float* **alpha**, *const gsl_vector_float* * **x**, *const gsl_vector_float* * **y**, *float* * ***result***) [Function]

二つの実数ベクトル x と y に関して和 $\alpha + x^T y$ を計算し、引数 *result* に入れて返す。

`int gsl_blas_sdot` (*const gsl_vector_float* * **x**, *const gsl_vector_float* * **y**, *float* * ***result***) [Function]
`int gsl_blas_dsdot` (*const gsl_vector_float* * **x**, *const gsl_vector_float* * **y**, *double* * ***result***) [Function]
`int gsl_blas_ddot` (*const gsl_vector* * **x**, *const gsl_vector* * **y**, *double* * ***result***) [Function]

二つの実数ベクトル x と y に関してスカラー積 $x^T y$ を計算し、引数 *result* に入れて返す。

`int gsl_blas_cdotu` (*const gsl_vector_complex_float* * **x**, *const gsl_vector_complex_float* * **y**, *gsl_complex_float* * ***dotu***) [Function]
`int gsl_blas_zdotu` (*const gsl_vector_complex* * **x**, *const gsl_vector_complex* * **y**, *gsl_complex* * ***dotu***) [Function]

二つの複素数ベクトル x と y に関してスカラー積 $x^T y$ を計算し、引数 *dotu* に入れて返す。

`int gsl_blas_cdotc` (*const gsl_vector_complex_float* * **x**, *const gsl_vector_complex_float* * **y**, *gsl_complex_float* * ***dotc***) [Function]
`int gsl_blas_zdotc` (*const gsl_vector_complex* * **x**, *const gsl_vector_complex* * **y**, *gsl_complex* * ***dotc***) [Function]

二つの複素数ベクトル x と y に関して複素共役スカラー積 $x^H y$ を計算し、引数 *dotc* に入れて返す。

`float gsl_blas_snrm2` (*const gsl_vector_float* * **x**) [Function]
`double gsl_blas_dnrm2` (*const gsl_vector* * **x**) [Function]

ベクトル x のユークリッド・ノルム $||x||_2 = \sqrt{\sum x_i^2}$ を計算する。

`float gsl_blas_scnrm2` (*const gsl_vector_complex_float* * **x**) [Function]
`double gsl_blas_dznrm2` (*const gsl_vector_complex* * **x**) [Function]

以下のような複素ベクトル x のユークリッド・ノルムを計算する。

$$||x||_2 = \sqrt{\sum (\mathrm{Re}(x_i)^2 + \mathrm{Im}(x_i)^2)}$$

`float gsl_blas_sasum` (*const gsl_vector_float* * **x**) [Function]
`double gsl_blas_dasum` (*const gsl_vector* * **x**) [Function]

ベクトル x の要素の絶対値の和 $\sum |x_i|$ を計算する。

`float gsl_blas_scasum` (*const gsl_vector_complex_float * x*) [Function]
`double gsl_blas_dzasum` (*const gsl_vector_complex * x*) [Function]

ベクトル x の要素の実部と虚部の絶対値の和 $\sum(|\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)|)$ を計算する。

`CBLAS_INDEX_t gsl_blas_isamax` (*const gsl_vector_float * x*) [Function]
`CBLAS_INDEX_t gsl_blas_idamax` (*const gsl_vector * x*) [Function]
`CBLAS_INDEX_t gsl_blas_icamax` (*const gsl_vector_complex_float * x*) [Function]
`CBLAS_INDEX_t gsl_blas_izamax` (*const gsl_vector_complex * x*) [Function]

ベクトル x の要素で最大のものの添え字を返す。実数ベクトルでは要素の絶対値で、複素数ベクトルでは各要素の実部と虚部の絶対値の和 $\sum(|\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)|)$ で最大値を決める。最大値を持つ要素が複数ある場合は、添え字がもっとも小さなものを返す。

`int gsl_blas_sswap` (*gsl_vector_float * x*, *gsl_vector_float * y*) [Function]
`int gsl_blas_dswap` (*gsl_vector * x*, *gsl_vector * y*) [Function]
`int gsl_blas_cswap` (*gsl_vector_complex_float * x*, *gsl_vector_complex_float * y*) [Function]
`int gsl_blas_zswap` (*gsl_vector_complex * x*, *gsl_vector_complex * y*) [Function]

ベクトル x と y のすべての要素を交換する。

`int gsl_blas_scopy` (*const gsl_vector_float * x*, *gsl_vector_float * y*) [Function]
`int gsl_blas_dcopy` (*const gsl_vector * x*, *gsl_vector * y*) [Function]
`int gsl_blas_ccopy` (*const gsl_vector_complex_float * x*, *gsl_vector_complex_float * y*) [Function]
`int gsl_blas_zcopy` (*const gsl_vector_complex * x*, *gsl_vector_complex * y*) [Function]

ベクトル x のすべての要素を y にコピーする。

`int gsl_blas_saxpy` (*float* `alpha`, *const gsl_vector_float * x*, *gsl_vector_float * y*) [Function]
`int gsl_blas_daxpy` (*double* `alpha`, *const gsl_vector * x*, *gsl_vector * y*) [Function]
`int gsl_blas_caxpy` (*const gsl_complex_float* `alpha`, *const gsl_vector_complex_float * x*, *gsl_vector_complex_float * y*) [Function]
`int gsl_blas_zaxpy` (*const gsl_complex* `alpha`, *const gsl_vector_complex * x*, *gsl_vector_complex * y*) [Function]

ベクトル x の α 倍と y の和 $y = \alpha x + y$ を計算する。

`void gsl_blas_sscal` (*float* `alpha`, *gsl_vector_float * x*) [Function]
`void gsl_blas_dscal` (*double* `alpha`, *gsl_vector * x*) [Function]

`void gsl_blas_cscal` (*const gsl_complex_float* `alpha`, *gsl_vector_complex_float* * `x`) [Function]
`void gsl_blas_zscal` (*const gsl_complex* `alpha`, *gsl_vector_complex* * `x`) [Function]
`void gsl_blas_csscal` (*float* `alpha`, *gsl_vector_complex_float* * `x`) [Function]
`void gsl_blas_zdscal` (*double* `alpha`, *gsl_vector_complex* * `x`) [Function]

ベクトル x を係数 *alpha* 倍する。

`int gsl_blas_srotg` (*float* `a[]`, *float* `b[]`, *float* `c[]`, *float* `s[]`) [Function]
`int gsl_blas_bdrotg` (*double* `a[]`, *double* `b[]`, *double* `c[]`, *double* `s[]`) [Function]

ベクトル (a, b) を 0 にするギブンズ変換 (Givens rotation) (c, s) を求める。

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r' \\ 0 \end{pmatrix}$$

引数のベクトル a と b はそれぞれ r' と 0 で上書きされる。

`int gsl_blas_srot` (*gsl_vector_float* * `x`, *gsl_vector_float* * `y`, *float* `c`, *float* `s`) [Function]
`int gsl_blas_drot` (*gsl_vector_float* * `x`, *gsl_vector_float* * `y`, *const double* `c`, *const double* `s`) [Function]

ベクトル x と y にギブンズ変換 $(x', y') = (cx + sy, -sx + cy)$ を適用する。

`int gsl_blas_srotmg` (*float* `d1[]`, *float* `d2[]`, *float* `b1[]`, *float* `b2[]`, *float* `p`) [Function]
`int gsl_blas_drotmg` (*double* `d1[]`, *double* `d2[]`, *double* `b1[]`, *double* `b2[]`, *double* `p[]`) [Function]

修正ギブンズ変換 (modified Givens transformation) を求める。参考文献に挙げてあるオリジナルの Level-1 BLAS の仕様にしたがっている。

`int gsl_blas_srotm` (*gsl_vector_float* * `x`, *gsl_vector_float* * `y`, *const float* `p[]`) [Function]
`int gsl_blas_drotm` (*gsl_vector* * `x`, *gsl_vector* * `y`, *const double* `p[]`) [Function]

修正ギブンズ変換を適用する。

12.1.2 Level 2

`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`) [Function]

int gsl_blas_dgemv (*CBLAS_TRANSPOSE_t* **TransA**, *double* **alpha**, *const gsl_matrix* * **A**, *const gsl_vector* * **x**, *double* **beta**, *gsl_vector* * **y**) [Function]
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**) [Function]
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**) [Function]

行列とベクトルの積と和 $y = \alpha op(A)x + \beta y$ を計算する。*TransA* が CblasNoTrans、CblasTrans、 CblasConjTrans のときそれぞれ $op(A) = A, A^T, A^H$ である。

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**) [Function]
int gsl_blas_dtrmv (*CBLAS_UPLO_t* **Uplo**, *CBLAS_TRANSPOSE_t* **TransA**, *CBLAS_DIAG_t* **Diag**, *const gsl_matrix* * **A**, *gsl_vector* * **x**) [Function]
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**) [Function]
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**) [Function]

三角行列 A とベクトル x の積と和 $x = op(A)x$ を計算する。*TransA* が CblasNoTrans、CblasTrans、 CblasConjTrans のときそれぞれ $op(A) = A, A^T, A^H$ である。*Uplo* が CblasUpper のとき A の上三角成分が使われ、*Uplo* が CblasLower のとき A の下三角成分が使われる。*Diag* が CblasNonUnit のとき行列の対角成分が使われ、*Diag* が CblasUnit のとき行列の対角成分は 1 であると見なされ A に入っている対角成分の値は無視される。

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**) [Function]
int gsl_blas_dtrsv (*CBLAS_UPLO_t* **Uplo**, *CBLAS_TRANSPOSE_t* **TransA**, *CBLAS_DIAG_t* **Diag**, *const gsl_matrix* * **A**, *gsl_vector* * **x**) [Function]
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**) [Function]
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**) [Function]

x に対して $inv(op(A))x$ を計算する。*TransA* が CblasNoTrans、CblasTrans、CblasConjTrans のときそれぞれ $op(A) = A, A^T, A^H$ である。*Uplo* が CblasUpper のとき A の上三

角成分が使われ、*Uplo* が CblasLower のとき A の下三角成分が使われる。*Diag* が CblasNonUnit のとき行列の対角成分が使われ、*Diag* が CblasUnit のとき行列の対角成分は 1 であると見なされ A に入っている対角成分の値は無視される。

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**) [Function]
int gsl_blas_dsymv (*CBLAS_UPLO_t* **Uplo**, *double* **alpha**, *const gsl_matrix* * **A**, *const gsl_vector* * **x**, *double* **beta**, *gsl_vector* * **y**) [Function]

対称行列 A に対して行列とベクトルの積と和 $y = \alpha Ax + \beta y$ を計算する。行列 A は対称行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき A の上三角成分が使われ、*Uplo* が CblasLower のとき A の下三角成分が使われる。

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**) [Function]
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**) [Function]

エルミート行列 A に対して行列とベクトルの積と和 $y = \alpha Ax + \beta y$ を計算する。行列 A はエルミート行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき A の上三角成分と対角成分が、*Uplo* が CblasLower のとき A の下三角成分と対角成分が使われる。対角成分の虚部は 0 であると見なされ、A に入っている対角成分の虚部の値は無視される。

int gsl_blas_sger (*float* **alpha**, *const gsl_vector_float* * **x**, *const gsl_vector_float* * **y**, *gsl_matrix_float* * **A**) [Function]
int gsl_blas_dger (*double* **alpha**, *const gsl_vector* * **x**, *const gsl_vector* * **y**, *gsl_matrix* * **A**) [Function]
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**) [Function]
int gsl_blas_zgeru (*const gsl_complex* **alpha**, *const gsl_vector_complex* * **x**, *const gsl_vector_complex* * **y**, *gsl_matrix_complex* * **A**) [Function]

行列 A のランク 1 の更新演算 (rank-1 update) $A = \alpha xy^T + A$ を計算する。

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**) [Function]
int gsl_blas_zgerc (*const gsl_complex* **alpha**, *const gsl_vector_complex* * **x**, *const gsl_vector_complex* * **y**, *gsl_matrix_complex* * **A**) [Function]

行列 A のランク 1 の共役更新 (conjugate rank-1 update) $A = \alpha xy^H + A$ を計算する。

int gsl_blas_ssyr (*CBLAS_UPLO_t* **Uplo**, *float* **alpha**, *const gsl_vector_float* * **x**, *gsl_matrix_float* * **A**) [Function]
int gsl_blas_dsyr (*CBLAS_UPLO_t* **Uplo**, *double* **alpha**, *const gsl_vector* * **x**, *gsl_matrix* * **A**) [Function]

対称行列 A のランク 1 の対称更新 (symmetric rank-1 update) $A = \alpha x x^T + A$ を計算する。行列 A は対称行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき A の上三角成分と対角成分が、*Uplo* が CblasLower のとき A の下三角成分と対角成分が使われる。

int gsl_blas_cher (*CBLAS_UPLO_t* **Uplo**, *float* **alpha**, *const gsl_vector_complex_float* * **x**, *gsl_matrix_complex_float* * **A**) [Function]
int gsl_blas_zher (*CBLAS_UPLO_t* **Uplo**, *double* **alpha**, *const gsl_vector_complex* * **x**, *gsl_matrix_complex* * **A**) [Function]

エルミート行列 A のランク 1 のエルミート更新 (hermitian rank-1 update) $A = \alpha x x^H + A$ を計算する。行列 A はエルミート行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき A の上三角成分と対角成分が、*Uplo* が CblasLower のとき A の下三角成分と対角成分が使われる。対角成分の虚部は 0 であると見なされ、A に入っている対角成分の虚部値は無視される。

int gsl_blas_ssyr2 (*CBLAS_UPLO_t* **Uplo**, *float* **alpha**, *const gsl_vector_float* * **x**, *const gsl_vector_float* * **y**, *gsl_matrix_float* * **A**) [Function]
int gsl_blas_dsyr2 (*CBLAS_UPLO_t* **Uplo**, *double* **alpha**, *const gsl_vector* * **x**, *const gsl_vector* * **y**, *gsl_matrix* * **A**) [Function]

対称行列 A のランク 2 の対称更新 (rank-2 update) $A = \alpha x y^T + \alpha y x^T + A$ を計算する。行列 A は対称行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき A の上三角成分と対角成分が、*Uplo* が CblasLower のとき A の下三角成分と対角成分が使われる。

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**) [Function]
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**) [Function]

エルミート行列 A のランク 2 のエルミート更新 $A = \alpha x y^H + \alpha^* y x^H + A$ を計算する。行列 A はエルミート行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき A の上三角成分と対角成分が、*Uplo* が CblasLower のとき A の下三角成分と対角成分が使われる。対角成分の虚部は 0 であると見なされ、A に入っている対角成分の虚部の値は無視される。

12.1.3 Level 3

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***) [Function]
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***) [Function]
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***) [Function]
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***) [Function]

行列同士の積と和 $C = \alpha op(A)op(B) + \beta C$ を計算する。*TransA* CblasNoTrans、CblasTrans、 CblasConjTrans のときそれぞれ $op(A) = A, A^T, A^H$ であり、引数 *TransB* に対しても同じである。

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***) [Function]
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***)[Function]
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***) [Function]
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***) [Function]

行列同士の積と和を計算する。A は対称行列で、*Side* が CblasLeft のとき $C = \alpha AB + \beta C$ を、*Side* が CblasRight のとき $C = \alpha BA + \beta C$ を計算する。*Uplo* が CblasUpper のとき A の上三角成分と対角成分が、*Uplo* が CblasLower のとき A の下三角成分と対角成分が使われる。

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***) [Function]
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***) [Function]

行列同士の積と和を計算する。A はエルミート行列で、*Side* が `CblasLeft` のとき $C = \alpha AB + \beta C$ を、*Side* が `CblasRight` のとき $C = \alpha BA + \beta C$ を計算する。*Uplo* が `CblasUpper` のとき A の上三角成分と対角成分が、*Uplo* が `CblasLower` のとき A の下三角成分と対角成分が使われる。対角成分の虚部は 0 であると見なされ、無視される。

`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`**) [Function]
`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`**) [Function]
`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`**) [Function]
`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`**) [Function]

行列同士の積を計算する。*Side* が `CblasLeft` のとき $B = \alpha op(A)B$ を、*Side* が `CblasRight` のとき $B = \alpha Bop(A)$ を計算する。行列 A は三角行列で、*TransA* が `CblasNoTrans`、`CblasTrans`、`CblasConjTrans` のときそれぞれ $op(A) = A, A^T, A^H$ である。*Uplo* が `CblasUpper` のとき A の上三角成分が、*Uplo* が `CblasLower` のとき A の下三角成分が使われる。*Diag* が `CblasNonUnit` のとき行列 A の対角成分が使われ、*Diag* が `CblasUnit` のとき行列 A の対角成分は 1 であると見なされ無視される。

`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`**) [Function]
`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`**) [Function]
`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`**) [Function]
`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`**) [Function]

行列同士の積を計算する。*Side* が `CblasLeft` のとき $B = \alpha op(inv(A))B$ を、*Side* が `CblasRight` のとき $B = \alpha Bop(inv(A))$ を計算する。行列 A は三角行列で、*TransA* が `CblasNoTrans`、`CblasTrans`、`CblasConjTrans` のときそれぞれ $op(A) = A, A^T, A^H$

である。*Uplo* が CblasUpper のとき A の上三角成分が、*Uplo* が CblasLower のとき A の下三角成分が使われる。*Diag* が CblasNonUnit のとき行列 A の対角成分が使われ、*Diag* が CblasUnit のとき行列 A の対角成分は 1 であると見なされ無視される。

int gsl_blas_ssyrk (*CBLAS_UPLO_t* **Uplo**, *CBLAS_TRANSPOSE_t* **TransA**, *float* **alpha**, *const gsl_matrix_float* * **A**, *float* **beta**, *gsl_matrix_float* * **C**)[Function]
int gsl_blas_dsyrk (*CBLAS_UPLO_t* **Uplo**, *CBLAS_TRANSPOSE_t* **TransA**, *double* **alpha**, *const gsl_matrix* * **A**, *double* **beta**, *gsl_matrix* * **C**) [Function]
int gsl_blas_csyrk (*CBLAS_UPLO_t* **Uplo**, *CBLAS_TRANSPOSE_t* **TransA**, *const gsl_complex_float* **alpha**, *const gsl_matrix_complex_float* * **A**, *const gsl_complex_float* **beta**, *gsl_matrix_complex_float* * **C**) [Function]
int gsl_blas_zsyrk (*CBLAS_UPLO_t* **Uplo**, *CBLAS_TRANSPOSE_t* **TransA**, *const gsl_complex* **alpha**, *const gsl_matrix_complex* * **A**, *const gsl_complex* **beta**, *gsl_matrix_complex* * **C**) [Function]

対称行列 C のランク k の更新演算を行う。*Trans* が CblasNoTrans、 CblasTrans のときそれぞれ $C = \alpha AA^T + \beta C$、$C = \alpha A^T A + \beta C$ である。行列 C は対称行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき C の上三角成分と対角成分が、*Uplo* が CblasLower のとき C の下三角成分と対角成分が使われる。

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**) [Function]
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**) [Function]

エルミート行列 C のランク k の更新演算を行う。*Trans* が CblasNoTrans、CblasTrans のときそれぞれ $C = \alpha AA^H + \beta C$、$C = \alpha A^H A + \beta C$ である。行列 C はエルミート行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき C の上三角成分と対角成分が、*Uplo* が CblasLower のとき C の下三角成分と対角成分が使われる。対角成分の虚部は自動的に 0 に設定される。

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**) [Function]
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**) [Function]
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**) [Function]

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**) [Function]

対称行列 C のランク $2k$ の更新演算を行う。*Trans* が CblasNoTrans、 CblasTrans のときそれぞれ $C = \alpha AB^T + \alpha BA^T + \beta C$、$C = \alpha A^T B + \alpha B^T A + \beta C$ である。行列 C は対称行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき C の上三角成分と対角成分が、*Uplo* が CblasLower のとき C の下三角成分と対角成分が使われる。

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**, *const gsl_complex_float* **beta**, *gsl_matrix_complex_float* * **C**) [Function]

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**, *const gsl_complex* **beta**, *gsl_matrix_complex* * **C**) [Function]

エルミート行列 C のランク $2k$ の更新演算を行う。*Trans* が CblasNoTrans、CblasConjTrans のときそれぞれ $C = \alpha AB^H + \alpha^* BA^H + \beta C$、$C = \alpha A^H B + \alpha^* B^H A + \beta C$ である。行列 C はエルミート行列として扱われるので、上または下三角成分だけが入っていればよい。*Uplo* が CblasUpper のとき C の上三角成分と対角成分が、*Uplo* が CblasLower のとき C の下三角成分と対角成分が使われる。対角成分の虚部は自動的に 0 に設定される。

12.2 例

以下に Level-3 BLAS の関数 DGEMM を使って、二つの行列の積を計算する例を示す。

$$\begin{pmatrix} 0.11 & 0.12 & 0.13 \\ 0.21 & 0.22 & 0.23 \end{pmatrix} \begin{pmatrix} 1011 & 1012 \\ 1021 & 1022 \\ 1031 & 1032 \end{pmatrix} = \begin{pmatrix} 367.76 & 368.12 \\ 674.06 & 674.72 \end{pmatrix}$$

行列は、C 言語での配列の格納方法にならってメモリ中では行優先で格納される。

```
#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;
}
```

このプログラムを実行したときの出力はこのようになる。

```
$ ./a.out
[ 367.76, 368.12
  674.06, 674.72 ]
```

12.3 参考文献

BLAS の標準化に関する情報は、旧いものや暫定標準と合わせて、BLAS のホームページと BLAS 技術フォーラム (BLAS Technical Forum) の web サイトから得られる。

- BLAS Homepage
  `http://www.netlib.org/blas/`
- BLAS Technical Forum
  `http://www.netlib.org/cgi-bin/checkout/blast/blast.pl`

BLAS の Level 1、2、3 の各仕様は、以下の論文に述べられている。

- C. Lawson, R. Hanson, D. Kincaid, F. Krogh, “Basic Linear Algebra Subprograms for Fortran Usage”, *ACM Transactions on Mathematical Software*, **5**(3), pp. 308-323, 324-325 (1979).
- J. J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson, “An Extended Set of Fortran Basic Linear Algebra Subprograms”, *ACM Transactions on Mathematical Software*, **14**(2), pp. 1-32 (1988).
- J. J. Dongarra, I. Duff, J. DuCroz, S. Hammarling, “A Set of Level 3 Basic Linear Algebra Subprograms”, *ACM Transactions on Mathematical Software*, **16**(1), pp. 1-28 (1990).

上の Dongarra による論文は、`http://www.netlib.org/blas/` で PostScript 版が公開されている。FORTRAN *blas* ライブラリを使うための *cblas* ラッパーもそこにある。

第13章　線形代数

この章では線形問題 (linear system) を解くための関数について説明する。GSL では `gsl_vector` および `gsl_matrix` オブジェクトに対して直接演算を行う関数を用意している。各関数は Golub & Van Loan (1996) によるアルゴリズムと BLAS の Level-1 および Level-2 のルーチンを使うことで、効率のよい計算を行う。

この章で説明する関数はヘッダファイル '`gsl_linalg.h`' で宣言されている。

13.1　LU 分解

一般の正方行列 (square matrix) A は、LU 分解 (LU decomposition) で上三角 (upper triangular) および下三角 (upper triangular) の二つの行列に、以下のようにわけることができる。

$$PA = LU$$

ここで P は置換行列 (permutation matrix)、L は単位下三角行列 (unit lower triangular matrix)、U は上三角行列 (upper triangular matrix) である (LU 分解は LUP 分解、PLU 分解と呼ばれることもある)。正方行列では、この分解を使うことで線形問題 $Ax = b$ を二つの三角問題 (triangular system) ($Ly = Pb, Ux = y$) に変換でき、それぞれ前進消去 (forward substitution) および後退代入 (backward s.) で解けるようになる。LU 分解は、特異行列 (singular matrix) に対しても使うことができる。

`int gsl_linalg_LU_decomp (`*gsl_matrix * **A***, *gsl_permutation * **p***, *int * **signum***`)` [Function]

`int gsl_linalg_complex_LU_decomp (`*gsl_matrix_complex * **A***, *gsl_permutation * **p***, *int * **signum***`)` [Function]

> 正方行列 A を $PA = LU$ の形となるように LU 分解する。行列 A の上三角成分と対角成分を行列 U で上書きし、行列 A の下三角成分を行列 L で上書きする。L の対角成分は 1 であることは既知なので、L には入れられない。
>
> 置換行列 P は置換 p に入れられる。置換ベクトルの第 j 要素が $k = p_j$ のとき、行列 P の第 j 列が単位行列の第 k 列になる。置換の符号は *signum* に入れられる。これは、n を置換による交換の回数とするとき、$(-1)^n$ で与えられる。ここで用いられるアルゴリズムは部分ピボッティングを導入したガウスの消去法である (Golub & Van Loan (1996), Algorithm 3.4.1 参照)。

`int gsl_linalg_LU_solve (`*const gsl_matrix * **LU***, *const gsl_permutation * **p***, *const gsl_vector * **b***, *gsl_vector * **x***`)` [Function]

int gsl_linalg_complex_LU_solve (*const gsl_matrix_complex * LU*, *const gsl_permutation * p*, *const gsl_vector_complex * b*, *gsl_vector_complex * x*) [Function]

gsl_linalg_LU_decomp または gsl_linalg_complex_LU_decomp を使って得られる正方行列 A の LU 分解 (LU, p) を使って、$Ax = b$ を解く。

int gsl_linalg_LU_svx (*const gsl_matrix * LU*, *const gsl_permutation * p*, *gsl_vector * x*) [Function]
int gsl_linalg_complex_LU_svx (*const gsl_matrix_complex * LU*, *const gsl_permutation * p*, *gsl_vector_complex * x*) [Function]

行列 A を LU 分解した結果 (LU, p) を使って、$Ax = b$ を解き、引数を結果で上書きする。引数 x には、関数の呼び出し時に右辺の b を保持しておく。これが解で上書きされる。

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 * residual*) [Function]
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 * residual*) [Function]

$Ax = b$ の解を、行列 A の LU 分解 (LU, p) を使った繰り返し計算で求めたいときに、その繰り返し部分を 1 回だけ行う。それにより改善される与えられた引数における残差 $r = Ax - b$ が計算され、*residual* に入れられる。

int gsl_linalg_LU_invert (*const gsl_matrix * LU*, *const gsl_permutation * p*, *gsl_matrix * inverse*) [Function]
int gsl_linalg_complex_LU_invert (*const gsl_matrix_complex * LU*, *const gsl_permutation * p*, *gsl_matrix_complex * inverse*) [Function]

行列 A の LU 分解 (LU, p) から、その逆行列を計算して行列 *inverse* に入れて返す。逆行列は単位行列の各列について $Ax = b$ を解くことで計算できる。この関数が使える場合は、直接に逆行列を計算するよりも、この関数の方が速度と精度の両面において有利である（詳しくは線形代数の数値計算に関する教科書などを参照のこと）。

double gsl_linalg_LU_det (*gsl_matrix * LU*, *int signum*) [Function]
gsl_complex gsl_linalg_complex_LU_det (*gsl_matrix_complex * LU*, *int signum*) [Function]

行列 A の LU 分解 LU から、その行列式を計算する。行列式は U の対角成分の積と行の置換 *signum* の符号から計算される。

double gsl_linalg_LU_lndet (*gsl_matrix * LU*) [Function]
double gsl_linalg_complex_LU_lndet (*gsl_matrix_complex * LU*) [Function]

行列 A の LU 分解 LU から、その行列式の絶対値の対数 $\ln|\det(A)|$ を計算する。行列式を直接計算するとオーバーフローやアンダーフローが起こる場合に有用である。

`int gsl_linalg_LU_sgndet (`*gsl_matrix * **LU***, int **signum***`)` [Function]
`gsl_complex gsl_linalg_complex_LU_sgndet (`*gsl_matrix_complex * **LU***, int **signum***`)` [Function]

行列 A の LU 分解 LU から、その行列式の符号または位相因子 (phase factor) $\det(A)/|\det(A)|$ を計算する。

13.2 QR 分解

正方でない一般の $M \times N$ 行列 A は以下のように、一般に QR 分解 (QR decomposition) を使って直交 M 次正方行列 Q (orthogonal M-by-M square matrix、$Q^T Q = \mathrm{I}$ という性質を持つ) と $M \times N$ の右三角行列 (right-triangular matrix) R に分けられる。

$$A = QR$$

QR 分解によって線形問題 $Ax = b$ は三角問題 $Rx = Q^T b$ に変換でき、これにより後退代入で解けるようになる。またベクトルの集合に対して、その直交基底 (orthogonal basis) を得るのにも使うことができる。A のランクが列指向で最大 (full column rank) のとき、Q の最初の N 列は、行列 A の値域 (range) $ran(A)$ で直交基底をなす。

`int gsl_linalg_QR_decomp (`*gsl_matrix * **A***, gsl_vector * **tau***`)` [Function]

$M \times N$ 行列 A を $A = QR$ の形に QR 分解する。A の対角成分と上三角成分に R が上書きされる。ベクトル *tau* と A の下三角成分の列にそれぞれハウスホルダー係数とハウスホルダー・ベクトルが入れられる。ベクトル *tau* の長さ k は $k = \min(M, N)$ でなければならない。この τ と A から行列 Q が、次のようにして得られる。v_i をハウスホルダー・ベクトル $v_i = (0, \ldots, 1, A(i+1, i), A(i+2, i), \ldots, A(m, i))$ として $Q_i = I - \tau_i v_i v_i^T$ とするとき、行列 Q は $Q = Q_k \ldots Q_2 Q_1$ と表される。これは LAPACK と同じであ、Q は GSL 関数の `gsl_linalg_QR_unpack` を使っても得られる。

ここで使っているアルゴリズムはハウスホルダー QR 法である。(Golub & Van Loan (1996), Algorithm 5.2.1 参照)。

`int gsl_linalg_QR_solve (`*const gsl_matrix * **QR***, const gsl_vector * **tau***, const gsl_vector * **b***, gsl_vector * **x***`)` [Function]

`gsl_linalg_QR_decomp` によって得られる正方行列 A の QR 分解 (QR, *tau*) を使って $Ax = b$ を解く。A が正方行列でない場合は、`gsl_linalg_RQ_lssolve` によって最小二乗法で解を得ることができる。

`int gsl_linalg_QR_svx (`*const gsl_matrix * **QR***, const gsl_vector * **tau***, gsl_vector * **x***`)` [Function]

gsl_linalg_QR_decomp によって得られる正方行列 A の QR 分解 (QR, tau) を使って $Ax = b$ を解く。引数 x は関数の呼び出し時に右辺の b を保持しておく。これが解で上書きされる。

int gsl_linalg_QR_lssolve (*const gsl_matrix * QR*, *const gsl_vector * tau*, *const gsl_vector * b*, *gsl_vector * x*, *gsl_vector * residual*) [Function]

行列 A の行が列よりも多い場合に、最小二乗法で $Ax = b$ の解を計算する。残差 $||Ax - b||$ のユークリッド・ノルムを最小にする解を得る。この関数の内部では gsl_linalg_QR_decomp を使って A を (QR, tau) に QR 分解する。得られた解は x に入れて返される。残差は二乗で計算され、*residual* に入れて返される。

int gsl_linalg_QR_QTvec (*const gsl_matrix * QR*, *const gsl_vector * tau*, *gsl_vector * v*) [Function]

分解された (QR, tau) から得られる Q^T とベクトル v の積を計算し、結果 $Q^T v$ を v に上書きして返す。行列の積の演算は、行列 Q^T の全体の計算を必要としないよう、ハウスホルダー・ベクトルを直接使って行われる。

int gsl_linalg_QR_Qvec (*const gsl_matrix * QR*, *const gsl_vector * tau*, *gsl_vector * v*) [Function]

分解された (QR, tau) から得られる Q とベクトル v の積を計算し、結果 Qv を v に上書きして返す。行列の積の演算は、行列 Q の全体の計算を必要としないよう、ハウスホルダー・ベクトルを直接使って行われる。

int gsl_linalg_QR_QTmat (*const gsl_matrix * QR*, *const gsl_vector * tau*, *gsl_matrix * A*) [Function]

分解された (QR, tau) から得られる Q^T と行列 A の積を計算し、結果 $Q^T A$ を A に上書きして返す。行列の積の演算は、行列 Q^T の全体の計算を必要としないよう、ハウスホルダー・ベクトルを直接使って行われる。

int gsl_linalg_QR_Rsolve (*const gsl_matrix * QR*, *const gsl_vector * b*, *gsl_vector * x*) [Function]

三角問題 $Rx = b$ を x について解く。$b' = Q^T b$ がすでに gsl_linalg_QR_QTvec を使って得られている場合に有用である。

int gsl_linalg_QR_Rsvx (*const gsl_matrix * QR*, *gsl_vector * x*) [Function]

三角問題 $Rx = b$ を x について解く。引数 x には関数の呼び出し時に右辺の b を保持しておく。これが解で上書きされる。$b' = Q^T b$ がすでに gsl_linalg_QR_QTvec を使って得られている場合に有用である。

int gsl_linalg_QR_unpack (*const gsl_matrix * QR*, *const gsl_vector * tau*, *gsl_matrix * Q*, *gsl_matrix * R*) [Function]

QR 分解された (QR, tau) を展開して Q と R に入れて返す。Q が $M \times M$ の正方行列で R が $M \times N$ である。

`int gsl_linalg_QR_QRsolve (`*gsl_matrix * `Q`, gsl_matrix * `R`, const gsl_vector * `b`, gsl_vector * `x`*`)` [Function]

$Rx = Q^T b$ を x について解く。行列の QR 分解がすでに得られていて、(Q, R) として展開されている場合に有用である。

`int gsl_linalg_QR_update (`*gsl_matrix * `Q`, gsl_matrix * `R`, gsl_vector * `w`, const gsl_vector * `v`*`)` [Function]

行列の QR 分解 (Q, R) の、ランク 1 の更新演算 (rank-1 update) wv^T を行う。更新は $Q'R' = QR + wv^T$ で与えられ、得られる Q' と R' はどちらも直交右三角になる。w の値は保存されない。

`int gsl_linalg_R_solve (`*const gsl_matrix * `R`, const gsl_vector * `b`, gsl_vector * `x`*`)` [Function]

R が N 次の正方行列のとき、三角問題 $Rx = b$ を解く。

`int gsl_linalg_R_svx (`*const gsl_matrix * `R`, gsl_vector * `x`*`)` [Function]

R が N 次の正方行列のとき、三角問題 $Rx = b$ を解く。引数 x は関数の呼び出し時に右辺の b を保持しておく。これが解で上書きされる。

13.3 列ピボット交換を行う QR 分解

列に対する置換 P を導入すると、以下のように、ランク (rank, 階数) が低い場合でも QR 分解ができる。

$$AP = QR$$

この Q の最初の r 列は、列のランクが r の行列に対して A の値域 (range) で直交基底をなす。この分解で線形問題 $Ax = b$ を三角問題 $Ry = Q^T b, x = Py$ に変換して、後退代入と置換で解くことができるようになる。$A = QRP^T$ であることから、以下では列ピボッティングを行う QR 分解のことを QRP^T と表す。

`int gsl_linalg_QRPT_decomp (`*gsl_matrix * `A`, gsl_vector * `tau`, gsl_permutation * `p`, int * `signum`, gsl_vector * `norm`*`)` [Function]

$M \times N$ 行列 A を $A = QRP^T$ の形の積に分解する。引数で与えられる行列 A の対角成分と上三角成分が R になる。置換行列 P は引数の置換 p に入れられる。置換の符号は *signum* に入れられる。その置換中での交換の回数を n とするとき、置換の符号は $(-1)^n$ になる。ベクトル tau にハウスホルダー係数が、行列 A の下三角成分中の列にハウスホルダー・ベクトルが入れられ、これらが直交行列 Q を表す。ベクトル tau の長さ k は $k = \min(M, N)$ でなければならない。v_i をハウスホルダー・ベクト

ル $v_i = (0, \ldots, 1, A(i+1,i), A(i+2,i), \ldots, A(m,i))$ として $Q_i = I - \tau_i v_i v_i^T$ とするとき、行列 Q は $Q = Q_k \ldots Q_2 Q_1$ と表される。これは LAPACK と同じである。ベクトル *norm* は長さ N で、列ピボッティング (column pivoting) を行うための作業領域である。

ここでは、列ピボッティングを行うハウスホルダー法 (Householder QR decomposition) を採用している (Golub & Van Loan (1996), Algorithm 5.4.1 参照)。

`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)` [Function]

引数で与えられる A を変更することなく、A を $A = QRP^T$ に分解する。分解結果は別に与える行列 q と r に入れて返される。

`int gsl_linalg_QRPT_solve (const gsl_matrix * QR, const gsl_vector * tau, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)` [Function]

`gsl_linalg_QRPT_decomp` によって得られる QRP^T 分解、(QR, tau, p) を使って $Ax = b$ を解く。

`int gsl_linalg_QRPT_svx (const gsl_matrix * QR, const gsl_vector * tau, const gsl_permutation * p, gsl_vector * x)` [Function]

QRP^T 分解 (QR, tau, p) を使って $Ax = b$ を解く。引数 x は関数の呼び出し時に右辺の b を保持しておく。これが解で上書きされる。

`int gsl_linalg_QRPT_QRsolve (const gsl_matrix * Q, const gsl_matrix * R, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)` [Function]

$RP^T x = Q^T b$ を x に関して解く。あらかじめ行列の QR 分解が (Q, R) の形に分けて得られているときに有用である。

`int gsl_linalg_QRPT_update (const gsl_matrix * Q, const gsl_matrix * R, const gsl_permutation * p, gsl_vector * w, const gsl_vector * v)` [Function]

QRP^T 分解 (Q, R, p) のランク 1 の更新演算を行う。更新演算は $Q'R' = QR + wv^T$ の形で表される。ここで出力として得られる Q' と R' はどちらも直交右三角行列である。引数 w の値は保存されない。置換 p は変化しない。

`int gsl_linalg_QRPT_Rsolve (const gsl_matrix * QR, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)` [Function]

引数で与えられる QR 中の $N \times N$ 行列 R で定義される三角問題 $RP^T x = b$ を解く。

`int gsl_linalg_QRPT_Rsvx (const gsl_matrix * QR, const gsl_permutation * p, gsl_vector * x)` [Function]

引数で与えられる QR 中の $N \times N$ 行列 R で定義される三角問題 $RP^T x = b$ を解く。引数 x は関数の呼び出し時に右辺の b を保持しておく。これが解で上書きされる。

13.4 特異値分解

一般の正方でない $M \times N$ の行列 A は、$M \times N$ の直交行列 U と、特異値を表す $N \times N$ の対角行列 S と、$N \times N$ の正方直交行列 V の転置行列の積として、以下のように分解できる。

$$A = USV^T$$

これを特異値分解 (singular value decomposition, svd) と呼ぶ。特異値 $\sigma_i = S_{ii}$ はすべて非負で、$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_N \geq 0$ となるように、降順に整列することが多い。

特異値分解には多くの応用例がある。特異値のうち最大のものと最小のものの比として、行列の条件数 (condition number) が与えられる。特異値に 0 が含まれていれば、その行列は正則ではない。非零の特異値の個数は行列のランクである。ランクが低い行列を特異値分解する場合、数値計算の精度の限界から、0 になるべき特異値が厳密には 0 にならないことがある。適切な誤差推定を行って、非常に小さな値の特異値を 0 とみなすべきかどうか判断せねばならない。

ランクが低い行列の場合、0 である特異値に対応する行列 V の各列は A の零空間 (null space) を与える。同様に、0 でない特異値に対応する行列 U の各列は A の値域 (range of A) を与える。

`int gsl_linalg_SV_decomp (`*gsl_matrix * **A***, *gsl_matrix * **V***, *gsl_vector * **S***, *gsl_vector * **work***)` [Function]

> $M \geq N$ である $M \times N$ 行列 A を $A = USV^T$ の形に特異値分解する。引数で与えられる A は計算で得られる U で置き換えられる。特異値行列 S の対角成分はベクトル S に入れられる。特異値は非負であり、S_1 から S_N に降順に並べられる。引数の行列 V には行列 V が転置されずに入れられる。USV^T の形の積を得るには、これを転置する必要がある。作業領域として長さ N のベクトル *work* が必要である。
>
> ここではゴラブ-ラインシュ (Gene Howard Golub and Christian H. Reinsch) の特異値分解法を採用している。

`int gsl_linalg_SV_decomp_mod (`*gsl_matrix * **A***, *gsl_matrix * **X***, *gsl_matrix * **V***, *gsl_vector * **S***, *gsl_vector * **work***)` [Function]

> 修正ゴラブ・ラインシュ法 (modified Golub-Reinsch algorithm) を使って特異値分解を行う。この方法は $M \gg N$ のときに高速である。作業領域としてベクトル *work* の他に $N \times N$ 行列 X が必要である。

`int gsl_linalg_SV_decomp_jacobi (`*gsl_matrix * **A***, *gsl_matrix * **V***, *gsl_vector * **S***)` [Function]

> 片側ヤコビ法による直交化 (one-sided Jacobi orthogolization) を使って特異値分解を行う。ヤコビ法はゴラブ・ラインシュ法に比較すると精度がよい。

`int gsl_linalg_SV_solve (`*gsl_matrix * **U***, *gsl_matrix * **V***, *gsl_vector * **S***, *const gsl_vector * **b***, *gsl_vector * **x***)` [Function]

> `gsl_linalg_SV_decomp` を使って行列 A を (U, S, V) に特異値分解することで $Ax = b$ を解く。

解を得るのには、非零の特異値のみが使われる。0 の特異値に対応する解は無視される。他の特異値も、この関数を呼ぶ前に 0 にすることで無視することができる。

行列 A の列よりも行が多いような優決定 (over-determined) な系の場合、解は最小二乗法的に計算され、ノルム $||Ax - b||_2$ を最小にする x が返される。

13.5　コレスキー分解

正定値の対称正方行列 A は、コレスキー分解 (Cholesky decomposition) を行うことで下三角行列 L とその転置行列 L^T の積として表すことができる。

$$A = LL^T$$

これを行列の平方根 (square root) として扱うこともある。コレスキー分解は、行列の固有値がすべて正の場合にのみ行うことができ、またこれにより、線形問題 $Ax = b$ を二つの三角問題 $(Ly = b, L^T x = y)$ に分解し、前進消去および後退代入で解くことができるようになる。

`int gsl_linalg_cholesky_decomp (`*gsl_matrix * **A***`)` [Function] `int gsl_linalg_complex_cholesky_decomp (`*gsl_matrix_complex * **A***`)` [Function]

正定値の正方行列 A を $A = LL^T$ の形（複素数の場合は $A = LL^\dagger$）にコレスキー分解する。計算の結果返されるのは L の対角成分と下三角成分である。入力として引数で与える A の上三角部分は無視され、A の対角成分と下三角成分が L で、上三角成分が L^T で置き換えられる (対角成分は L と L^T で同じである)。与えられる行列が正定値 (positive-definite) でない場合は、コレスキー分解は途中で失敗し、エラーコード `GSL_EDOM` を返す。

この関数を使って行列が正定値かどうかを調べることもできる。その時はエラー・ハンドラーを無効にして、そこに制御が移るのを避ける。

`int gsl_linalg_cholesky_solve (`*const gsl_matrix * **cholesky***, *const gsl_vector * **b***, *gsl_vector * **x***`)` [Function] `int gsl_linalg_complex_cholesky_solve (`*const gsl_matrix_complex * **cholesky***, *const gsl_vector_complex * **b***, *gsl_vector_complex * **x***`)` [Function]

`gsl_linalg_cholesky_decomp` または `gsl_linalg_comples_cholesky_decomp` によって A を コレスキー分解して得られる行列 *cholesky* を使って、線形問題 $Ax = b$ を解く。

`int gsl_linalg_cholesky_svx (`*const gsl_matrix * **cholesky***, *gsl_vector * **x***`)` [Function] `int gsl_linalg_complex_cholesky_svx (`*const gsl_matrix_complex * **cholesky***, *gsl_vector_complex * **x***`)` [Function]

`gsl_linalg_cholesky_decomp` または `gsl_linalg_comples_cholesky_decomp` によって A をコレスキー分解して得られる行列 *cholesky* を使って、線形問題 $Ax = b$ を解く。引数 x は関数の呼び出し時に右辺の b を保持しておく。これが解で上書きされる。

13.6 実対称行列の三重対角分解

対称行列 A は相似変換 (similarity transformation) で以下のような積の形に表すことができる。

$$A = QTQ^T$$

ここで Q は直交行列、T は対称三重対角行列 (symmetric tridiagonal matrix) である。

`int gsl_linalg_symmtd_decomp (`*gsl_matrix * **A***, *gsl_vector * **tau***`)` [Function]

対称正方行列 A を対称三重対角行列の積 QTQ^T に分解する。三重対角行列 T は、引数で与えられる行列 A の対角および下副対角成分に入れて返される。A の下三角部分の残りの部分には直交行列 Q を変換したハウスホルダー・ベクトルが入れられ、ハウスホルダー係数が *tau* に入れられる。A のどこにベクトルのどの要素が入れられるかは、LAPACK と同じ形式である。A の上三角成分に入れられる値には意味はない。

`int gsl_linalg_symmtd_unpack (`*const gsl_matrix * **A***, *const gsl_vector * **tau***, *gsl_matrix * **Q***, *gsl_vector * **diag***, *gsl_vector * **subdiag***`)` [Function]

関数 `gsl_linalg_symmtd_decomp` によって得られた対称三重対角分解の計算結果 (*A*, *tau*) から直交行列を Q に、対角成分を *diag* に、副対角成分を *subdiag* に取り出す。

`int gsl_linalg_symmtd_unpack_T (`*const gsl_matrix * **A***, *gsl_vector * **diag***, *gsl_vector * **subdiag***`)` [Function]

関数 `gsl_linalg_symmtd_decomp` によって得られた対称三重対角分解の計算結果 (*A*, tau) から対角成分を *diag* に、副対角成分を *subdiag* に取り出す。

13.7 エルミート行列の三重対角分解

エルミート行列 A も相似変換で以下のような積の形に表すことができる。

$$A = UTU^T$$

ここで U はユニタリー行列 (unitary matrix)、T は実対称三重対角行列である。

`int gsl_linalg_hermtd_decomp (`*gsl_matrix_complex * **A***, *gsl_vector_complex * **tau***`)` [Function]

エルミート行列 A を対称三重対角行列の積 UTU^T に分解する。三重対角行列 T は、引数で与えられる行列 A の対角および下副対角成分の実部に入れて返される。A の下三角成分の残りの部分には直交行列 Q を変換したハウスホルダー・ベクトルが入れられ、ハウスホルダー係数が *tau* に入れられる。A のどこにベクトルのどの要素が入れられるかは、LAPACK と同じ形式である。A の上三角成分と対角成分の虚部に入れられる値には意味はない。

`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***`)`[Function]

関数 gsl_linalg_hermtd_decomp によって得られた対称三重対角分解の結果 (A, tau) からユニタリー行列を U に、対角成分を実数ベクトル *diag* に、副対角成分を実数ベクトル *subdiag* に取り出す。

int gsl_linalg_hermtd_unpack_T (*const gsl_matrix_complex * **A***, *gsl_vector * **diag***, *gsl_vector * **subdiag***) [Function]

関数 gsl_linalg_hermtd_decomp によって得られた対称三重対角分解の結果 (A, tau) から対角成分を実数ベクトル *diag* に、副対角成分を実数ベクトル *subdiag* に取り出す。

13.8 実数行列のヘッセンベルク分解

一般に実数の行列 A は相似変換で以下のような積の形に表すことができる。

$$A = UHU^T$$

ここで U は直交行列、H は上ヘッセンベルク行列 (Hessenberg matrix、副対角成分より左下がすべて 0) である。ヘッセンベルク分解 (Hessenberg reduction) は、非対称固有値問題 (nonsymmetric eigenvalue problem) を解くためのシューア分解 (Schur decomposition) の最初のステップとして行われるが、他にも利用例は多い。

int gsl_linalg_hessenberg_decomp (*gsl matrix * **A***, *gsl vector * **tau***) [Function]

相似変換 $H = U^T AU$ を計算することによって行列 A に対してヘッセンベルク分解を行う。分解の結果得られる行列 H は A の右上部に入れられる。U を得るために必要な情報は A の下三角部に入れられる。U は $N-2$ 個のハウスホルダー行列の積である。ハウスホルダー・ベクトルが A の下三角部 (副対角成分よりも下の部分) に入れられ、ハウスホルダー係数がベクトル *tau* に入れられる。*tau* の長さは N でなければならない。

int gsl_linalg_hessenberg_unpack (*gsl matrix * **H***, *gsl vector * **tau***, *gsl matrix * **U***) [Function]

ヘッセンベルク行列 H とベクトル *tau* から直行行列 U を計算する。H と *tau* はそれぞれ、gsl_linalg_hessenberg_decomp が返す結果である。

int gsl_linalg_hessenberg_unpack_accum (*gsl matrix * **H***, *gsl vector * **tau***, *gsl matrix * **V***) [Function]

この関数は gsl_linalg_hessenberg_unpack とほぼ同じだが、計算して得られる U を引数で与えられる V に適用し、$V' = UV$ として V に入れて返す。したがってこの関数を呼ぶときに、V には値を入れておかねばならない。V を単位行列にしておくと、gsl_linalg_hessenberg_unpack と同じ結果になる。H が N 次の正方行列のとき、V の列の数は N でなければならないが、行の数はいくつでもよい。

int gsl_linalg_hessenberg_set_zero (*gsl_matrix * **H***) [Function]

行列 H の副対角成分よりも下の要素を 0 にする。gsl_linalg_hessenberg_decomp の後に、ハウスホルダー・ベクトルを 0 にリセットするのに使う。

13.9 実数行列のヘッセンベルク三角分解

二つの実数行列 (A, B) は直行対称変換によって以下のように分解できる。

$$\begin{aligned} A &= UHV^T \\ B &= URV^T \end{aligned}$$

ここで U と V は直行行列、H は上ヘッセンベルク行列、R は 上三角行列である。ヘッセンベルク三角分解 (Hessenberg-triangular reduction) は、一般的な固有値問題を解くための一般シューア分解の最初のステップとして行われる。

`int gsl_linglg_hesstri_decomp` (*gsl_matrix* * **A**, *gsl_matrix* * **B**, *gsl_matrix* * **U**, *gsl_matrix* * **V**, *gsl_vector* * **work**) [Function]

二つの行列 (A, B) から、そのヘッセンベルク三角分解を計算する。計算結果として、H が A に、R が B に入れて返される。U と V が与えられた場合は（与えない場合は NULL にする)、それらに相似変換を入れて返す。長さ N の作業領域 *work* を与えなければならない。

13.10 二重対角化

一般に行列 A は相似変換で以下のような積の形に表すことができる。

$$A = UBV^T$$

ここで U と V は直交行列、B は $N \times N$ の二重対角行列 (bidiagonal matrix) で、対角成分と上副対角成分以外の要素は 0 である。U の大きさは $M \times N$ で、V は $N \times N$ である。

`int gsl_linalg_bidiag_decomp` (*gsl_matrix* * **A**, *gsl_vector* * **tau_U**, *gsl_vector* * **tau_V**) [Function]

$M \times N$ の行列 A を UBV^T の形に分解する。A の対角成分および上対角成分に行列 B の対角成分および上対角成分を入れて返す。直交行列 U と V は、A の空いているところにまとめて入れられる。ハウスホルダー係数はベクトル *tau_U* と *tau_V* に入れて返される。*tau_U* の長さは行列 A の対角成分の個数と同じ、*tau_V* の長さはそれよりも 1 だけ短くなければならない。

`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**) [Function]

関数 `gsl_linalg_bidiag_decomp` によって得られた行列 A の二重対角分解 (*A*, *tau_U*, *tau_V*) から二つの直交行列 U と V、対角ベクトル *diag* と上対角ベクトル *superdiag* を取り出す。メモリを効率よく使うため、U は $U^TU = I$ を満たす $M \times N$ の直交行列に入れられる。

int gsl_linalg_bidiag_unpack2 (*gsl_matrix * A*, *gsl_vector * tau_U*, *gsl_vector * tau_V*, *gsl_matrix * V*) [Function]

関数 gsl_linalg_bidiag_decomp によって得られた行列 A の二重対角分解 (A, tau_U, tau_V) から二つの直交行列 U と V、対角ベクトル *diag* と上対角ベクトル *superdiag* を取り出す。A の内容を U で上書きする。

int gsl_linalg_bidiag_unpack_B (*const gsl_matrix * A*, *gsl_vector * diag*, *gsl_vector * superdiag*) [Function]

関数 gsl_linalg_bidiag_decomp によって得られた行列 A の対角成分と二重対角分解を、対角ベクトル *diag* と上対角ベクトル *superdiag* に取り出す。

13.11 ハウスホルダー変換

ハウスホルダー変換 (Householder transformation) とは単位行列に対するランク 1 の更新演算 (rank-1 modification) であり、これによりベクトルの任意の要素を選んで 0 にすることができる。ハウスホルダー行列 P は以下の形式を取る。

$$P = I - \tau v v^T$$

ここで v はハウスホルダー・ベクトルと呼ばれ、$\tau = 2/(v^T v)$ を満たす。この節の関数はハウスホルダー行列のランク 1 構造 (rank-1 structure) を使って、効率よくハウスホルダー変換を行う。

double gsl_linalg_householder_transform (*gsl_vector * v*) [Function]
gsl_complex gsl_linalg_complex_hauseholder_transform (*gsl_vector_complex * v*) [Function]

引数で与えられるベクトルの要素を先頭をのぞいてすべて 0 にするハウスホルダー変換 $P = I - \tau v v^T$ を用意する。変換ベクトルを v に上書きして入れ、スカラー値 τ を返す。

int gsl_linalg_householder_hm (*double tau*, *const gsl_vector * v*, *gsl_matrix * A*) [Function]
int gsl_linalg_complex_householder_hm (*gsl_complex tau*, *const gsl_vector_complex * v*, *gsl_matrix_complex * A*) [Function]

スカラー値 tau とベクトル v で定義されるハウスホルダー行列 P を行列 A に左から作用させて変換を行う。変換の結果 PA で引数 A を上書きする。

int gsl_linalg_householder_mh (*double tau*, *const gsl_vector * v*, *gsl_matrix * A*) [Function]
int gsl_linalg_complex_householder_mh (*gsl_complex tau*, *const gsl_vector_complex * v*, *gsl_matrix_complex * A*) [Function]

スカラー値 tau とベクトル v で定義されるハウスホルダー行列 P を行列 A に右から作用させて変換を行う。変換の結果 PA で引数 A を上書きする。

`int gsl_linalg_householder_hv (`*double* `tau`, *const gsl_vector ** `v`, *gsl_vector ** `w``)` [Function]

`int gsl_linalg_complex_householder_hv (`*gsl_complex* `tau`, *const gsl_vector_complex ** `v`, *gsl_vector_complex ** `w``)` [Function]

スカラー値 tau とベクトル v で定義されるハウスホルダー行列 P をベクトル w に適用する。変換の結果 Pw で引数 w を上書きする。

13.12 ハウスホルダー変換による線形問題の解法

`int gsl_linalg_HH_solve (`*gsl_matrix ** `A`, *const gsl_vector ** `b`, *gsl_vector ** `x``)` [Function]

ハウスホルダー変換を使って線形問題 $Ax = b$ を解く。解は x に入れられ、b の値は変化しない。行列 A の要素はハウスホルダー変換により書き換えられる。

`int gsl_linalg_HH_svx (`*gsl_matrix ** `A`, *gsl_vector ** `x``)` [Function]

ハウスホルダー変換を使って線形問題 $Ax = b$ を解く。関数を呼び出すときに、x には b の値を入れておく。x は解で上書きされる。行列 A の要素はハウスホルダー変換により書き換えられる。

13.13 三重対角問題

この節の関数は、対称、非対称および巡回三重対角問題 (symmeric, nonsymmetric, cyclic tridiagonal system) を最小のメモリ使用量で効率よく解くものである。現在の実装で使っているアルゴリズムはコレスキー分解の一種であり、そのため三重対角行列は正定値でなければならない。与えられた行列が正定値でない場合は、エラーコードとして `GSL_ESING` を返す。

`int gsl_linalg_solve_tridiag (`*const gsl_vector ** `diag`, *const gsl_vector ** `e`, *const gsl_vector ** `f`, *const gsl_vector ** `b`, *gsl_vector ** `x``)` [Function]

A が $N \times N$ の三重対角行列 ($N \geq 2$) のとき、三重対角問題 $Ax = b$ を解く。上対角および下対角成分のベクトル e と f の長さはどちらも、対角成分ベクトル $diag$ よりも 1 だけ短くなければならない。三重対角行列は、A が 4×4 の場合は以下のような形である。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ 0 & 0 & f_2 & d_3 \end{pmatrix}$$

`int gsl_linalg_solve_symm_tridiag` (*const gsl_vector * **diag***, *const gsl_vector * **e***, *const gsl_vector * **b***, *gsl_vector * **x***) [Function]

A が $N \times N$ の対称三重対角行列 ($N \geq 2$) のとき、線形問題 $Ax = b$ を解く。非対角成分のベクトル e の長さは、対角成分ベクトル *diag* よりも 1 だけ短くなければならない。対称三重対角行列は、A が 4×4 の場合は以下のようになる。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ 0 & 0 & e_2 & d_3 \end{pmatrix}$$

`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***) [Function]

A が $N \times N$ の巡回三重対角行列 ($N \geq 3$) のとき、線形問題 $Ax = b$ を解く。巡回上および下対角成分のベクトル e および f の長さは、対角成分ベクトル *diag* と同じでなければならない。巡回三重対角行列は、A が 4×4 の場合は以下のようになる。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & f_3 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ e_3 & 0 & f_2 & d_3 \end{pmatrix}$$

`int gsl_linalg_solve_symm_cyc_tridiag` (*const gsl_vector * **diag***, *const gsl_vector * **e***, *const gsl_vector * **b***, *gsl_vector * **x***) [Function]

A が $N \times N$ の対称巡回三重対角行列 ($N \geq 3$) のとき、線形問題 $Ax = b$ を解く。巡回非対角成分のベクトル e の長さは、対角成分ベクトル *diag* と同じでなければならない。対称巡回三重対角行列は、A が 4×4 の場合は以下のようになる。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & e_3 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ e_3 & 0 & e_2 & d_3 \end{pmatrix}$$

13.14　平衡化

平衡化 (balancing) とは、相似変換によって行列の各列と各行のそれぞれのノルムの値を近づけることである。これを行うことで例えば、固有値問題を解くときの丸め誤差を減らすことができる。行列 A の平衡化は、以下のように A を相似な行列で置き換えることである。

$$A' = D^{-1} A D$$

ここで D は対角行列で、その対角要素は浮動小数点の基数のベキ乗である。

`int gsl_linalg_balance_matrix` (*gsl_matrix * **A***, *gsl_vector * **D***) [Function]

行列 A を平衡化された行列 (balanced counterpart) で置き換え、平衡化に使われた対称変換の対角要素をベクトル D に入れて返す。

13.15 例

以下のプログラムでは、線形問題 $Ax = b$ を解く例を示す。係数は以下のようになっている。

$$\begin{pmatrix} 0.18 & 0.60 & 0.57 & 0.96 \\ 0.41 & 0.24 & 0.99 & 0.58 \\ 0.14 & 0.30 & 0.97 & 0.66 \\ 0.51 & 0.13 & 0.19 & 0.85 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}$$

この方程式の解は行列 A の LU 分解を使って得られる。

```
#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  = 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;
}
```

以下にプログラムの出力を示す。

```
x = -4.05205
-12.6056
1.66091
8.69377
```

GNU OCTAVE を使って解 x と元の行列 A の積を計算すると下のようになり、解の正しさを確認することができる。

```
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
ans =
  1.0000
  2.0000
  3.0000
  4.0000
```

元の式 $Ax = b$ に従って、これにより右辺の b が得られる。

13.16　参考文献

この節で説明した関数で使っているアルゴリズムに関しては、以下の本に解説がある。

- G. H. Golub, C. F. Van Loan, *Matrix Computations* (3rd Ed), Johns Hopkins University Press, ISBN 0-8018-5414-8 (1996).

LAPACK については以下のマニュアルに説明されている。

- *LAPACK Users' Guide* (3rd Ed.), Society for Industrial and Applied Mathematics, ISBN 0-89871-447-8 (1999).
  `http://www.netlib.org/lapack`

LAPACK のソースコードも上記の web サイトから、利用案内とともに入手できる。

修正ゴラブ-ラインシュ法については以下の論文に述べられている。

- Tony F. Chan, "An Improved Algorithm for Computing the Singular Value Decomposition", *ACM Transactions on Mathematical Software*, **8**(1), pp. 72–83 (1982).

特異値分解を行うヤコビ法については、下の論文に述べられている。

- J.C.Nash, “A one-sided transformation method for the singular value decomposition and algebraic eigenproblem”, *Computer Journal*, **18**(1), pp. 74–76 (1973).
- James Demmel, Kresimir Veselic, “Jacobi’s Method is more accurate than QR”, Lapack Working Note 15 (LAWN-15), (1989.10).

後者は netlib `http://www.netlib.org/lapack/` の `lawns` または `lawnspdf` ディレクトリにある。

第14章　固有値問題

この章では行列の固有値 (eigenvalue) と固有ベクトル (eigenvector) を計算する関数について説明する。GSL では実数対称行列 (real symmetric)、実数非対称行列 (real nonsymmetric)、複素エルミート行列 (complex Hermitian)、実数の一般対称定値行列 (real generalized symmetirc-definite)、複素数の定値エルミート行列 (complex generalized Hermitian-definite)、実数の一般非対称行列 (real nonsymmetric) のそれぞれに対する固有値計算関数を実装している。固有ベクトルがある時はそれを使えるが、なくても計算できる。エルミート行列と実数の対称行列については、対称二重対角化 (symmetric bidiagonalization) に続けて QR 分解を行うアルゴリズムを使う。非対称行列にはフランシス (John. G. F. Francis) の二重シフト QR 法 (double shift QR) を使う。一般の非対称行列には、モラー (Cleve Barry Moler) とスチュワート (G. W. (Pete) Stewart) による QZ 法を使う。

この章で説明する関数はヘッダファイル 'gsl_eigen.h' で宣言されている。

14.1　実数対称行列

GSL では、実数の対称行列の固有値問題には対称二重対角化に続けて QR 分解を行うアルゴリズムを使う (Golub & van Loan (1996), 第 8.3 節)。計算で得られる固有値の精度は、ϵ を計算精度 (機械イプシロン、machine epsilon) とするとき $\epsilon||A||_2$ である。

`gsl_eigen_symm_workspace * gsl_eigen_symm_alloc (const size_t n)` [Function]

> $n \times n$ の実数対称行列の固有値を計算するための作業領域を確保する。作業領域の大きさのオーダーは $O(2n)$ である。

`void gsl_eigen_symm_free (gsl_eigen_symm_workspace * w)` [Function]

> 作業領域 w が確保しているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_eigen_symm (gsl_matrix * A, gsl_vector * eval, gsl_eigen_symm_workspace * w)` [Function]

> 実数対称行列 A の固有値を計算する。適切な大きさの作業領域をあらかじめ確保して、w として指定する必要がある。計算では A の対角成分および下三角成分が参照され、その値は変えられてしまうが、上三角成分は無視される。計算された固有値はベクトル *eval* に、整列されずに入れて返される。

gsl_eigen_symmv_workspace * gsl_eigen_symmv_alloc (*const size_t* **n**) [Function]

$n \times n$ の実数対称行列の固有値と固有ベクトルを計算するための作業領域を確保する。作業領域の大きさのオーダーは $O(4n)$ である。

void gsl_eigen_symmv_free (*gsl_eigen_symmv_workspace* * **w**) [Function]

作業領域 *w* が確保しているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

int gsl_eigen_symmv (*gsl_matrix* * **A**, *gsl_vector* * **eval**, *gsl_matrix* * **evec**, *gsl_eigen_symmv_workspace* * **w**) [Function]

実数対称行列 *A* の固有値と固有ベクトルを計算する。適切な大きさの作業領域をあらかじめ確保して、*w* として指定する必要がある。計算では *A* の対角成分および下三角成分が参照され、その値は変えられてしまうが、上三角成分は無視される。計算された固有値はベクトル *eval* に、整列されずに入れて返される。対応する固有ベクトルは、行列 *evec* に列として入れて返される。たとえば最初の列に入っている固有ベクトルは、最初に入っている固有値に対応する。固有ベクトルはお互いに直交し、それぞれの長さは 1 になるように正規化される。

14.2 複素エルミート行列

gsl_eigen_herm_workspace * gsl_eigen_herm_alloc (*const size_t* **n**) [Function]

$n \times n$ の複素エルミート行列の固有値を計算するための作業領域を確保する。作業領域の大きさのオーダーは $O(3n)$ である。

void gsl_eigen_herm_free (*gsl_eigen_herm_workspace* * **w**) [Function]

作業領域 *w* が確保しているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

int gsl_eigen_herm (*gsl_matrix_complex* * **A**, *gsl_vector* * **eval**, *gsl_eigen_herm_workspace* * **w**) [Function]

複素エルミート行列 *A* の固有値を計算する。適切な大きさの作業領域をあらかじめ確保して、*w* として指定する必要がある。計算では *A* の対角成分および下三角成分が参照され、その値は変えられてしまうが、上三角成分は無視される。対角成分の虚部は 0 であると見なされ、値が入っていても無視される。計算された固有値はベクトル *eval* に、整列されずに入れて返される。

gsl_eigen_hermv_workspace * gsl_eigen_hermv_alloc (*const size_t* **n**) [Function]

$n \times n$ の複素エルミート行列の固有値と固有ベクトルを計算するための作業領域を確保する。作業領域の大きさのオーダーは $O(5n)$ である。

`void gsl_eigen_hermv_free (gsl_eigen_hermv_workspace * w)` [Function]

作業領域 w が確保しているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_eigen_hermv (gsl_matrix_complex * A, gsl_vector * eval, gsl_matrix_complex * evec, gsl_eigen_hermv_workspace * w)` [Function]

複素エルミート行列 A の固有値と固有ベクトルを計算する。適切な大きさの作業領域をあらかじめ確保して、w として指定する必要がある。計算では A の対角成分および下三角成分が参照され、その値は変えられてしまうが、上三角成分は無視される。対角成分の虚部は 0 であると見なされ、値が入っていても無視される。計算された固有値はベクトル *eval* に、整列されずに入れて返される。対応する複素固有ベクトルは、行列 *evec* に列として入れて返される。たとえば最初の列に入っている固有ベクトルは、最初に入っている固有値に対応する。固有ベクトルはお互いに直交し、それぞれの長さは 1 になるように正規化される。

14.3 実数非対称行列

実数非対称行列 A の固有値問題は、以下のシューア分解 (Schur decomposition) を用いて解かれる。

$$A = ZTZ^T$$

ここで Z はシューア・ベクトルからなる直交行列であり、T は上三角行列に似た形 (ブロック上三角行列) で、シューア形式 (Schur form) と呼ばれる。T の対角線上の 1×1 あるいは 2×2 のブロックが対角成分に相当し、1×1 のブロック (普通の意味での対角要素) は実数の固有値、2×2 のブロックは複素数の固有値の複素共役である。GSL ではフランシスの二重シフト法を用いて実装している。

`gsl_eigen_nonsymm_workspace * gsl_eigen_nonsymm_alloc (const size_t n)` [Function]

$n \times n$ の実数非対称行列の固有値を計算するための作業領域を確保する。作業領域の大きさは $O(2n)$ のオーダーである。

`void gsl_eigen_nonsymm_free (gsl_eigen_nonsymm_workspace * w)` [Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`void gsl_eigen_nonsymm_params (const int compute_t, const int balance, gsl_eigen_nonsymm_worksp * w)` [Function]

固有値を求める関数 gsl_eigen_nonsymm にどのように計算を行うかを指示するための設定を行う。

gsl_eigen_nonsymm で完全なシューア形式 T を計算するためには、*compute_t* を 1 にしてこの関数を呼ぶ。これを 0 にすると T は計算されない (これがデフォルトの動作である)。完全なシューア形式を計算すると、計算時間が 1.5 から 2 倍になる。

balance を 1 にすると固有値の計算をする前に平衡化 (balancing) を行う設定になる。平衡化により行列の各行、各列のノルムの値ができるだけ近い値になるように変換され、行列の各要素の値の桁が大きく異なるような場合でも、固有値計算の精度を上げることができる。(13.14 節「平衡化」、166 ページ参照)。平衡化によってシューア・ベクトルの直交性が失われるため、gsl_eigen_nonsysmm_Z 関数を使って得られるシューア・ベクトルは、元の行列の代わりに平衡化された行列のものになる。その関係は、以下で表される。

$$T = Q^t D^{-1} A D Q$$

ここで Q は平衡化された行列のシューア・ベクトルによる行列、D は平衡化を行う変換行列である。gsl_eigen_nonsysmm_Z は以下の関係を満たす行列 Z を求める。

$$T = Z^{-1} A Z$$

ここで $Z = DQ$ である。Z は直交行列ではない。上記の理由により、平衡化はデフォルトでは行わない設定になっている。

`int gsl_eigen_nonsymm (`*gsl matrix* * `A`, *gsl_vector_complex* * `eval`, *gsl_eigen_nonsymm_workspa* * `w``)` [Function]

実数非対称行列 A の固有値を計算し、ベクトル *eval* に入れて返す。シューア形式 T の計算を行う設定になっているときは T が求められ A の上部に入れられる。そうでないときは、A の各対角線上の 1×1 あるいは 2×2 のブロックに固有値が入れられる。1×1 のブロックには実数の、2×2 のブロックには複素数の固有値の複素共役が入れられる。A の他の成分は計算により書き変わる。場合によってはすべての固有値が計算できないことがあるが、そのときはエラーコードを返し、収束できた固有値の個数が w->n_evals に入れられる。収束した固有値そのものは *eval* の先頭から入れられる。

`int gsl_eigen_nonsymm_Z (`*gsl_matrix* * `A`, *gsl_vector_complex* * `eval`, *gsl_matrix* * `Z`, *gsl_eigen_nonsymm_workspace* * `w``)` [Function]

この関数は、シューア・ベクトルを計算して Z に入れて返すこと以外は、gsl_eigen_nonsymm と同じである。

`gsl_eigen_nonsymmv_workspace * gsl_eigen_nonsymmv_alloc (`*const size_t* `n``)` [Function]

$n \times n$ の非対称実数行列の固有値と固有ベクトルを計算するための作業領域を確保する。作業領域の大きさは $O(5n)$ のオーダーである。

`void gsl_eigen_nonsymmv_free (`*gsl_eigen_nonsymmv_workspace* * `w`)[Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_eigen_nonsymmv (`*gsl_matrix* * `A`, *gsl_vector_complex* * `eval`, *gsl_matrix_complex* * `evec`, *gsl_eigen_nonsymmv_workspace* * `w`) [Function]

$n \times n$ の非対称実数行列 A の固有値と直交固有ベクトルを計算する。この関数の内部では最初に、固有値、シューア形式 T、シューア・ベクトルを計算するために `gsl_eigen_nonsymm` が呼ばれ、それから T の固有値を求め、シューア・ベクトルを使って逆変換する。シューア・ベクトルは計算の過程で破棄されるが、`gsl_eigen_nonsymmv_Z` を使うことでその値を得ることができる。計算された各固有ベクトルは長さが 1 になるよう正規化される。A の上部にはシューア形式 T が入れて返される。`gsl_eigen_nonsymm` の計算が失敗した場合は固有ベクトルは計算されず、エラーコードが返される。

`int gsl_eigen_nonsymmv_Z (`*gsl_matrix* * `A`, *gsl_vector_complex* * `eval`, *gsl_matrix_complex* * `evec`, *gsl_matrix* * `Z`, *gsl_eigen_nonsymmv_workspace* * `w`) [Function]

この関数はシューア・ベクトルを計算して Z に入れて返すこと以外は、`gsl_eigen_nonsymmv` と同じである。

14.4 実数の正定値対称行列の固有値問題

実数の正定値対称行列の固有値問題とは、以下のような固有値 λ と固有ベクトル x を求めることである。

$$Ax = \lambda Bx$$

ここで A と B は対称行列で、B は正定値である。B をコレスキー分解することで、この問題は以下のように、標準的な対称行列の固有値問題になる。

$$\begin{aligned} Ax &= \lambda Bx \\ Ax &= \lambda LL^t x \\ (L^{-1}AL^{-t})L^t x &= \lambda L^t x \end{aligned}$$

ここで $C = L^{-1}AL^t, y = L^t x$ とすると C は対称行列であるため、3 番目の式は $Cy = \lambda y$ という形式になり、ごく普通の、対称行列の固有値問題を解く手法がそのまま使えることになる。一般の正定値対称行列の固有値と固有ベクトルは常に実数になる。

`gsl_eigen_gensymm_workspace * gsl_eigen_gensymm_alloc (`*const size_t* `n`)[Function]

$n \times n$ の正定値対称固有値問題を解くための作業領域を確保する。作業領域の大きさは $O(2n)$ のオーダーである。

`void gsl_eigen_gensymm_free (`*gsl_eigen_gensymm_workspace* * `w`) [Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_eigen_gensymm (`*gsl_matrix* * `A`, *gsl_matrix* * `B`, *gsl_vector* * `eval`, *gsl_eigen_gensymm_w* * `w`) [Function]

上で概説した方法で二つの正定値対称行列 (A, B) から固有値を計算し、*eval* に入れて返す。B には、与えられた B のコレスキー分解が入れて返される。A の内容は破壊される。

`gsl_eigen_gensymmv_workspace * gsl_eigen_gensymmv_alloc (`*const size_t* `n`) [Function]

$n \times n$ の正定値対称問題で固有値と固有ベクトルを求めるための作業領域を確保する。作業領域の大きさは $O(4n)$ のオーダーである。

`void gsl_eigen_gensymmv_free (`*gsl_eigen_gensymmv_workspace* * `w`) [Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_eigen_gensymmv (`*gsl_matrix* * `A`, *gsl_matrix* * `B`, *gsl_vector* * `eval`, *gsl_matrix* * `evec`, *gsl_eigen_gensymmv_workspace* * `w`) [Function]

二つの正定値対称行列 (A, B) から固有値と固有ベクトルを計算し、それぞれ *eval* と *evec* に入れて返す。各固有ベクトルは長さが 1 になるよう正規化される。B には、与えられた B のコレスキー分解が入れて返される。A の内容は破壊される。

14.5 複素数の正定値エルミート行列の固有値問題

複素数の正定値エルミート行列の固有値問題とは、以下のような固有値 λ と固有ベクトル x を求めることである。

$$Ax = \lambda Bx$$

ここで A と B はエルミート行列で、B は正定値である。実数の場合と同様、$C = L^{-1}AL^{-\dagger}, y = L^{\dagger}x$ とすると C はエルミート行列であるため、$Cy = \lambda y$ という問題になり、ごく普通のエルミート行列の固有値問題を解く手法がそのまま使えることになる。この正定値エルミート行列の固有値問題で得られる固有値と固有ベクトルは常に実数になる。

`gsl_eigen_genherm_workspace * gsl_eigen_genherm_alloc (`*const size_t* `n`) [Function]

複素数の正定値エルミート固有値問題を解くための作業領域を確保する。作業領域の大きさは $O(3n)$ のオーダーである。

`void gsl_eigen_genherm_free (`*gsl_eigen_genherm_workspace* `* w)` [Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_eigen_genherm (`*gsl_matrix_complex* `* A`, *gsl_matrix_complex* `* B`, *gsl_vector* `* eval`, *gsl_eigen_genherm_workspace* `* w)` [Function]

上で概説した方法で二つの正定値エルミート行列 (A, B) から固有値を計算し、*eval* に入れて返す。B には、与えられた B のコレスキー分解が入れて返される。A の内容は破壊される。

`gsl_eigen_genhermv_workspace * gsl_eigen_genhermv_alloc (`*const size_t* `n)` [Function]

$n \times n$ の正定値エルミート問題で固有値と固有ベクトルを求めるための作業領域を確保する。作業領域の大きさは $O(5n)$ のオーダーである。

`void gsl_eigen_genhermv_free (`*gsl_eigen_genhermv_workspace* `* w)` [Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`int gsl_eigen_genhermv (`*gsl_matrix_complex* `* A`, *gsl_matrix_complex* `* B`, *gsl_vector* `* eval`, *gsl_matrix_complex* `* evec`, *gsl_eigen_genhermv_workspace* `* w)` [Function]

二つの正定値エルミート行列 (A, B) から固有値と固有ベクトルを計算し、それぞれ *eval* と *evec* に入れて返す。各固有ベクトルは長さが 1 になるよう正規化される。B には、与えられた B のコレスキー分解が入れて返される。A の内容は破壊される。

14.6 実数の一般非対称行列の固有値問題

二つの正方行列 (A, B) が与えられた時、実数の一般非対称行列の固有値問題とは以下のような固有値 λ と固有ベクトル x を求めることである

$$Ax = \lambda Bx$$

ここで同様に、固有値 μ と固有ベクトル y を求める問題を定義する。

$$\mu Ay = By$$

λ も μ も 0 でない場合、$\lambda = 1/\mu$ とおけばこの二つの問題は等価である。しかしそうすると、$\lambda = 0$ のときに一方の問題は性質のよい (well-defined な) 固有値問題になるが μ を含む方はそうはいかない。固有値に 0 (あるいは無限大) が含まれてもいいような問題は以下のようになる。

$$\beta Ax = \alpha Bx$$

以下で説明するルーチンは α と β の値を返し、その値から $\lambda = \alpha/\beta$、$\mu = \beta/\alpha$ と計算すればそれぞれの値が得られるようにしている。

λ のすべての値について行列束 $A - \lambda B$ の行列式が 0 になる場合、この問題は「正則ではない (singular)」。そうでない場合は正則 (regular) である。正則でない、あるいは特異である場合は普通 $\alpha = \beta = 0$ となり、これは固有値問題として条件がよくないか、計算により求められる固有値がない、と解釈される。この節の関数は正則な問題を想定して作られており、正則でない問題に対してはその計算結果は予測できない。

二つの行列 (A, B) についての、実数の一般非対称行列の固有値問題の解を得るには、以下の一般シューア分解を行う。

$$\begin{aligned} A &= QSZ^T \\ B &= QTZ^T \end{aligned}$$

ここで Q と Z はそれぞれ左、および右のシューア・ベクトルからなる直交行列であり、(S, T) は一般シューア形式で、その対角成分は α および β である。実装されているのはモラー (Cleve Barry Moler) とスチュワート (G. W. (Pete) Stewart) による QZ 法である。

`gsl_eigen_gen_workspace * gsl_eigen_gen_alloc` (*const size_t* `n`) [Function]

$n \times n$ の実数の一般非対称行列の固有値と固有ベクトルを求めるための作業領域を確保する。作業領域の大きさは $O(n)$ のオーダーである。

`void gsl_eigen_gen_free` (*gsl_eigen_gen_workspace* * `w`) [Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`void gsl_eigen_gen_params` (*const int* `compute_s`, *const int* `compute_t`, *const int* `balance`, *gsl eigen gen workspace* * `w`) [Function]

固有値を求める関数 `gsl_eigen_gen` にどのように計算を行うかを指示するための設定を行う。

compute_s を 1 にしてこの関数を呼ぶと、その後に呼ばれる `gsl_eigen_gen` では完全なシューア形式 S が計算される。0 にすると S は計算されない (これがデフォルトの設定である)。S は上三角行列に似た形 (ブロック上三角行列) で、その各対角線上の 1×1 あるいは 2×2 のブロックが対角成分に相当する。1×1 の要素は実数、2×2 の要素は複素数の固有値に対応する。

compute_t を 1 にしてこの関数を呼ぶと、その後に呼ばれる `gsl_eigen_gen` では完全なシューア形式 T が計算される。0 にすると T は計算されない (これがデフォルトの設定である)。T はすべての対角要素が非負のブロック上三角行列である。S で 2×2 のブロックの要素は T の 2×2 のブロック対角要素に対応する。

現在の実装では、引数 *balancing* は無視される。一般的な平衡化法が未だ実装されていないためである。

`int gsl_eigen_gen` (*gsl_matrix* * `A`, *gsl_matrix* * `B`, *gsl_vector_complex* * `alpha`, *gsl_vector* * `beta`, *gsl_eigen_gen_workspace* * `w`) [Function]

二つの実数一般非対称行列 (A, B) から固有値を計算し、(*alpha*, *beta*) に入れて返す。*alpha* は複素数、*beta* は実数になる。β_i が 0 でなければ、$\lambda = \alpha_i/\beta_i$ が固有値である。同様に α_i が 0 でなければ、$\mu = \beta_i/\alpha_i$ がもう一方の固有値問題 $\mu Ay = By$ の 解である。*beta* の要素は非負になるように正規化される。

設定により S の計算が要求されているときは、S が計算され A に上書きして返される。T が要求されているときは B に上書きされる。(*alpha*, *beta*) の中での固有値の並ぶ順番は、シューア形式 S と T での対角要素の順番と同じである。場合によってはすべての固有値を計算できないことがあるが、その場合にはエラーコードを返す。

`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`) [Function]

この関数は、左および右のシューア・ベクトルを計算してそれぞれ Q および Z に入れて返すこと以外は、`gsl_eigen_gen` と同じである。

`gsl_eigen_genv_workspace * gsl_eigen_genv_alloc` (*const size_t* `n`) [Function]

$n \times n$ の実数の一般非対称行列の固有値と固有ベクトルを求めるための作業領域を確保する。作業領域の大きさは $O(7n)$ のオーダーである。

`void gsl_eigen_genv_free` (*gsl_eigen_genv_workspace* * `w`) [Function]

確保されている作業領域 w を解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`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`) [Function]

二つの $n \times n$ の実数一般非対称行列 (A, B) から固有値と直交固有ベクトルを計算する。固有値は (*alpha*, *beta*) に、固有ベクトルは *evec* に入れて返される。この関数は内部で最初に `gsl_eigen_gen` を呼んで固有値、シューア形式、シューア・ベクトルを計算する。そしてシューア形式の固有値を求め、シューア・ベクトルを使って逆変換する。シューア・ベクトルは計算の過程で破棄されるが、`gsl_eigen_genv_QZ` を使うとその値を得ることができる。各固有ベクトルは長さが 1 になるよう正規化される。(A, B) には一般シューア形式 (S, T) が上書きされて返される。`gsl_eigen_gen` が固有値の計算に失敗した場合、固有ベクトルは計算されず、エラーコードを返す。

`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`) [Function]

この関数は、左および右のシューア・ベクトルを計算して Q および Z に入れて返すこと以外は、`gsl_eigen_genv` と同じである。

14.7　固有値と固有ベクトルの整列

int gsl_eigen_symmv_sort (*gsl_vector* * **eval**, *gsl_matrix* * **evec**, *gsl_eigen_sort_t* ***sort_type***) [Function]

ベクトル *eval* に入っている固有値と、それに対応する行列 *evec* の各列に入っている固有ベクトルを、引数 *sort_type* での指定にしたがって昇順あるいは降順に整列する。*sort_type* には以下の値が指定できる。

GSL_EIGEN_SORT_VAL_ASC 数値の昇順に整列

GSL_EIGEN_SORT_VAL_DESC 数値の降順に整列

GSL_EIGEN_SORT_ABS_ASC 絶対値の昇順に整列

GSL_EIGEN_SORT_ABS_DESC 絶対値の降順に整列

int gsl_eigen_hermv_sort (*gsl_vector* * **eval**, *gsl_matrix_complex* * **evec**, *gsl_eigen_sort_t* ***sort_type***) [Function]

ベクトル *eval* に入っている固有値と、行列 *evec* に入っている、固有値に対応する複素固有ベクトルを、引数 *sort_type* での指定に従って昇順あるいは降順に整列する。*sort_type* には上述の値が指定できる。

int gsl_eigen_nonsymmv_sort (*gsl_vector_complex* * **eval**, *gsl_matrix_complex* * **evec**, *gsl_eigen_sort_t* ***sort_type***) [Function]

ベクトル *eval* に保存されている固有値、およびそれに対応した順序で行列 *evec* の各列に保存されているいる固有ベクトルを同時に、上述の *sort_type* の値にしたがって昇順あるいは降順に整列する。固有値は一般に複素数なので、ここでは GSL_EIGEN_SORT_ABS_ASC または GSL_EIGEN_SORT_ABS_DESC だけが指定できる。

int gsl_eigen_gensymmv_sort (*gsl_vector* * **eval**, *gsl_matrix* * **evec**, *gsl_eigen_sort_t* ***sort_type***) [Function]

ベクトル *eval* に保存されている固有値、およびそれに対応した順序で行列 *evec* の各列に保存されているいる実数の固有ベクトルを同時に、上述の *sort_type* の値にしたがって昇順あるいは降順に整列する。

int gsl_eigen_genhermv_sort (*gsl vector* * **eval**, *gsl_matrix_complex* * **evec**, *gsl_eigen_sort_t* ***sort_type***) [Function]

ベクトル *eval* に保存されている固有値、およびそれに対応した順序で行列 *evec* の各列に保存されているいる複素数の固有ベクトルを同時に、上述の *sort_type* の値にしたがって昇順あるいは降順に整列する。

int gsl_eigen_genv_sort (*gsl_vector_complex* * **alpha**, *gsl_vector* * **beta**, *gsl_matrix_complex* * **evec**, *gsl_eigen_sort_t* ***sort_type***) [Function]

ベクトル *alpha* および *beta* に保存されている固有値、およびそれに対応した順序で行列 *evec* の各列に保存されているいる固有ベクトルを同時に、上述の *sort_type* の値にしたがって昇順あるいは降順に整列する。固有値は一般に複素数なので、ここでは GSL_EIGEN_SORT_ABS_ASC または GSL_EIGEN_SORT_ABS_DESC だけが指定できる。

14.8 例

以下に四次のヒルベルト行列 $H(i,j) = 1/(i+j+1)$ の固有値と固有ベクトルを計算するプログラムを示す。

```
#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,
                      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;
}
```

プログラムの出力は、最初の方は以下のようになる。

```
$ ./a.out
eigenvalue = 9.67023e-05
eigenvector =
-0.0291933
0.328712
-0.791411
0.514553
...
```

GNU OCTAVE では以下のようにすると、同じ計算を行うことができる。

```
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
```

固有ベクトルの符号は任意なので、符号が異なる結果が得られることがある。

次に非対称行列の例を示す。$x = (-1, -2, 3, 4)$ について、ヴァンデルモンド行列 (Vandermonde matrix) $V(x; i, j) = x_i^{n-j}$ (n は列数、形式の異なる定義もある) の固有値と固有ベクトルを計算する。

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

int main (void)
```

```
{
    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));
            }
        }
    }

    gsl_vector_complex_free(eval);
    gsl_matrix_complex_free(evec);
    return 0;
}
```

プログラムの出力は、最初の方は以下のようになる。

```
$ ./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 で以下のようにすると、同じ計算を行うことができる。

```
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:

  -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
```

固有値 $5.54555 + 3.08545i$ に対応する固有ベクトルの値が、若干異なっている。これは、大きさが 1 の定数 $0.9999984 + 0.0017674i$ を乗じることによる。

14.9　参考文献

この章で触れたアルゴリズムについては、以下の文献に解説がある。

- G. H. Golub, C. F. Van Loan, *Matrix Computations* (3rd Ed), Johns Hopkins University Press, ISBN 0-8018-5414-8 (1996).

QZ 法についての詳しい解説は、以下の論文にある。

- C. B. Moler, G. W. Stewart, “An Algorithm for Generalized Matrix Eigenvalue Problems”, *SIAM Journal on Numerical Analysis*, **10**(2) pp. 214-256 (1973).

巨大な行列の固有値問題を解くルーチンは、FORTRAN ライブラリの LAPACK にある。LAPACK については以下に述べられている。

- *LAPACK Users' Guide* (Third Edition), Society for Industrial and Applied Mathematics, ISBN 0-89871-447-8 (1999). `http://www.netlib.org/lapack`

LAPACK のソースプログラム、ユーザーマニュアルのオンライン版とともに上記の web サイトから入手できる。

第15章　高速フーリエ変換 (FFT)

この章では高速フーリエ変換 (Fast Fourier Transforms, FFT) を行う関数について説明する。GSL では等間隔データに対する基数 (radix) が 2 の FFT (データ長が 2 のべき乗) と混合基数 FFT (任意のデータ長) の両方が実装されている。実行速度の向上をはかるため、各関数には実数版と複素数版を用意している。混合基数の関数はパウル・シュヴァルツラウバー (Paul Swarztrauber) の FFTPACK ライブラリを実装し直したものである。FFTPACK の FORTRAN コードは Netlib に含まれている (FFTPACK には sin および cos 変換のプログラムも含まれているが、それらは現在 GSL では実装していない)。アルゴリズムの詳細や導出については、GSL の付属文書 *GSL FFT Algorithms* を参照のこと。

15.1　数学的定義

高速フーリエ変換とは、以下の離散フーリエ変換 (Discrete Fourier transform, DFT) を効率よく行う計算法である。

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi i j k/N)$$

DFT は一般的に、関数値が時間軸上または空間内の離散的な点で得られるときに、その連続フーリエ変換 (continuous Fourier transform) の近似として用いられる。そのまま計算すると離散フーリエ変換は行列とベクトルの積 $W\vec{z}$ の計算である。行列とベクトルの積の計算量は、データ数が N のとき $O(N^2)$ である。高速フーリエ変換は分割統治法 (divide-and-conquer strategy) を使って行列 W をデータ長 N の約数に対応する複数の小行列に分解する。N が整数の積 $f_1 f_2 \dots f_n$ で表されるとき、 DFT の計算量は $O(N \sum f_i)$ である。基数 2 の FFT では $O(N \log_2 N)$ になる。

GSL の FFT 関数はすべて三種類の演算を行うことができる。順方向 (forward)、逆変換 (inverse)、逆方向 (backward) である。どれも基本となる FFT の数学的な定義は同じである。順方向フーリエ変換 $x = \mathrm{FFT}(z)$ の定義は以下である。

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi i j k/N)$$

また逆フーリエ変換 $z = \mathrm{IFFT}(x)$ の定義は以下である。

$$z_j = \frac{1}{N} \sum_{k=0}^{N-1} x_k \exp(2\pi i j k/N)$$

上式の係数 $1/N$ により、正しい逆変換になる。`gsl_fft_complex_forward` に続いて `gsl_fft_complex_inverse` を呼び出すと、(数値計算上の誤差を除いて) 元のデータと同じ数値が得られる。

順変換、逆変換を組み合わせるとき、指数関数 (に渡す引数) の符号の取り方には二通りある。GSL では FFTPACK と同じで、順変換で負の符号である。これにより、逆変換では単純にフーリエ級数を計算することで元のデータが得られる。「ニュメリカル・レシピ (Press, et. al., 1986)」ではこれとは逆で、順方向で正の符号になっている。

三つ目の変換、逆方向 FFT は、以下で定義される正規化を行わない逆変換である。

$$z_j^{backwards} = \sum_{k=0}^{N-1} x_k \exp(2\pi i j k/N)$$

変換後の全体的な数値の大きさがあまり重要ではないような場合は、除算を行わない分だけ、逆変換よりも逆方向の方が簡便である。

15.2　複素数データに対する FFT

複素数 FFT に対するデータの受け渡しは浮動小数点実数の密配置の配列 (packed array、実メモリ上でアライメント行わず、数値の型の大きさで各変数が確保されている) である。密配列中では各複素数の実部と虚部が交互に並べられる。たとえば長さ 6 の密配列を以下のように定義すると、

```
double x[3*2];
gsl_complex_packed_array data = x;
```

三つの複素数を保持する配列 `z[3]` を以下のように使うことができる。

```
data[0] = Re(z[0])
data[1] = Im(z[0])
data[2] = Re(z[1])
data[3] = Im(z[1])
data[4] = Re(z[2])
data[5] = Im(z[2])
```

配列の添え字の順序は DFT の定義と同じである。データの順序に関する変換や置換は行われない。

FFT 関数の引数の *stride* を使うことで、`z[i]` ではなく `z[stride*i]` のデータだけを使った変換を行うことができる。刻み幅 *stride* を 2 以上にすると、行列の列に対する FFT で、結果をその行列の中に書き込むことができる。刻み幅が 1 の場合は要素間に隙間を空けることなくデータにアクセスすることになる。

`gsl_vector_complex * v` 型などのベクトルを引数として FFT を行いたい場合、以下の (もしくは以下と同等な) 置き換えを行ってから、引数に渡す。

```
gsl_complex_packed_array data = v->data;
size_t stride = v->stride;
size_t n = v->size;
```

現実的な問題に応用する場合、DFT での添え字が物理的な周波数と直接に対応しているわけではないことに留意せねばならない。DFT の時間刻み幅が Δ のとき、周波数領域では 0 をはさん

で $-1/(2\Delta)$ から $+1/(2\Delta)$ までの、正および負の値が現れる。正の値が配列の先頭から中央までに入れられ、負の値は配列の終端から逆向きに中央までに入れられる。

時系列データまたは FFT の結果を保持する配列 *data* の添字 `index` と、時間領域での値 z、周波数領域での値 x がどのように対応しているかを以下の表に示す。

```
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 が偶数の時、$N/2$ の位置には最大の周波数での値が入れられる (正の値 $+1/(2\Delta)$ または負の値 $-1/(2\Delta)$ で、FFT の結果は同じ値になる)。n が奇数の時も上の表の通りだが、$n/2$ が整数でないので $n/2$ の要素はない。

15.3 複素数に対する基数 2 の FFT

ここで説明する基数 2 の FFT (radix-2 FFT) アルゴリズムは、効率はよくないが単純かつ簡潔である。クーリーとテューキー (James William Cooley and John Wilder Tukey) のアルゴリズムを使って 2 のべき乗個のデータに対し複素数置換 FFT を行う。基数 2 の FFT は与えられるデータを書き換えながら計算を行う「置換法 (in-place computation)」で行われるため、別途の作業領域を必要としない。これに対して自己整列混合基数法 (self-sorting mixed-radix FFT) は別に与えられる作業領域を使うことで高速に計算を行う。

以下に上げる関数はヘッダファイル '`gsl_fft_complex.h`' で宣言されている。

`int gsl_fft_complex_radix2_forward` (*gsl_complex_packed_array* **`data`**, *size_t* **`stride`**, *size_t* **`n`**) [Function]

`int gsl_fft_complex_radix2_transform` (*gsl_complex_packed_array* **`data`**, *size_t* **`stride`**, *size_t* **`n`**, *gsl_fft_direction* **`sign`**) [Function]

`int gsl_fft_complex_radix2_backward` (*gsl_complex_packed_array* **`data`**, *size_t* **`stride`**, *size_t* **`n`**) [Function]

`int gsl_fft_complex_radix2_inverse` (*gsl_complex_packed_array* **`data`**, *size_t* **`stride`**, *size_t* **`n`**) [Function]

長さ n で刻み幅が *stride* の複素数配列 *data* に対して、時間間引き (decimation-in-time) で基数 2 の置換アルゴリズムで、順方向、逆方向、逆変換の FFT を行う。変換

長 n は 2 のべき乗でなければならない。関数名に `transform` がついているものでは、*sign* 引数に `forward` (-1) か `backward` (+1) のいずれかを指定する。

関数の処理中に何もエラーが出なければ、これらの関数は `GSL_SUCCESS` を返す。データ長 n が 2 のべき乗でないときには `GSL_EDOM` を返す。

`int gsl_fft_complex_radix2_dif_forward` (*gsl_complex_packed_array* **`data`**, *size_t* ***`stride`***, *size_t* **`n`**) [Function]
`int gsl_fft_complex_radix2_dif_transform` (*gsl_complex_packed_array* **`data`**, *size_t* ***`stride`***, *size_t* **`n`**, *gsl_fft_direction* ***`sign`***) [Function]
`int gsl_fft_complex_radix2_dif_backward` (*gsl_complex_packed_array* **`data`**, *size_t* ***`stride`***, *size_t* **`n`**) [Function]
`int gsl_fft_complex_radix2_dif_inverse` (*gsl_complex_packed_array* **`data`**, *size_t* ***`stride`***, *size_t* **`n`**) [Function]

これらは周波数間引き (decimation-in-frequency) の基数 2 の FFT である。

以下に、データ長 128 の短いパルス波の FFT を計算するプログラムを示す。変換結果が実数になるためには、信号は時間が正の領域と負の領域で同じ波形に定義されていなければならない ($-10 \ldots 10$)。負の時刻のデータは配列の後半に保持されていなければならない。

```
#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];

    for (i = 0; i < 128; i++) REAL(data,i) = 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");

    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;
}
```

ここでは、プログラム中ではデフォルトのエラー・ハンドラーを設定していると仮定している (エラー発生時には abort 関数が呼び出される)。あまり安全でないエラー・ハンドラーを使う場合は、関数 gsl_fft_complex_radix2_forward の返り値をチェックするべきである。

変換されたデータは $1/\sqrt{N}$ でスケーリングされ、入力データと同じグラフにプロットできるようになっている。入力データでは虚数部は 0 なので、実部だけを示す。時間が負の領域は $t = 128$ で折り返しており、時間の最小単位は k/N なので、DFT は以下のように sin 関数で連続フーリエ変換を近似していることになる。

$$\int_{-a}^{+a} e^{-2\pi ikx} dx = \frac{\sin(2\pi ka)}{\pi k}$$

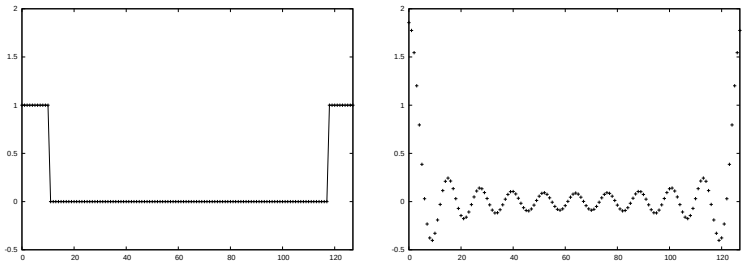


図 15.1: 元のパルス波（左）と、例示したプログラムによるその離散フーリエ変換（右）。

15.4 複素数に対する混合基数 FFT

この節では複素数に対する混合基数 (mixed-radix) FFT アルゴリズムについて説明する。混合基数の関数は任意のデータ長に対して変換を行うことができる。GSL の FFT 関数はパウル・シュヴァルツラウバー (Paul Swarztrauber) による FORTRAN の FFTPACK ライブラリを実装し直したものである。理論的な背景はテンパートン (Clive Temperton) のレビュー記事に述べられている。GSL での配列の添え字の順序や基本的なアルゴリズムは FFTPACK と同じである。

混合基数法は複数の変換法の組み合わせである。短いデータに対して高度に最適化された FFT をつなぎ合わせることで、長いデータに対する FFT を行う。基数 2、3、4、5、6、7 に対する効率の高い FFT が用意されており、合成数である 4 と 6 での演算はそれぞれ 2×2 や 2×3 の組み合わせによる FFT よりも高速である。

ここで実装されていない基数での演算は、DFT を効率よく行うシングルトン (Richard C. Singleton) の方法を用いた一般のデータ長 n に対する FFT になる。この方法でのデータ長 n に対する計算量は $O(n^2)$ で、特定の基数に特化した方法よりも遅い。一般のデータ長 n に対する演算は、因数分解されてから行われる。たとえばデータ長が 143 の場合は 11×13 に分解される。したがって、たとえばデータ長を因数分解すると $n = 2 \times 3 \times 99991$ のような大きな素数が出現する場合には効率が上がらない。この場合にはその素数に対する計算量 $O(n^2)$ が全体の計算量に対して支配的になる (このような問題に直面したときには、GSL の配布パッケージに同梱されている *GSL FFT Algorithms* を参考にするとよい)。

混合基数の場合の初期化関数 `gsl_fft_complex_wavetable_alloc` は、与えられるデータのサイズ N から GSL ルーチンが自動的に決定する基数のリストを返す。これをチェックすることで演算に要する計算時間を見積もることができる。ごく大雑把には、実行時間は $N \sum f_i$ に比例すると考えてよい（f_i は N の約数)。これにより、データサイズの値の因数分解があまり都合良くない場合には警告を出すなどして、プログラムを実行する際にその動作を選べるようにすることもできる。用意されている基数に分解できないような事が頻繁に起こる場合など、他の基数による FFT ルーチンを作成したい場合は前述の付属文書、*GSL FFT Algorithms* を参照するとよい。

以下の関数の宣言は全てヘッダファイル 'gsl_fft_complex.h' にある。

`gsl_fft_complex_wavetable * gsl_fft_complex_wavetable_alloc` (*size_t* **`n`**) [Function]

> データ長 n の複素数 FFT で使う三角関数の値をあらかじめ計算して表を作る。エラーが発生しなければ `gsl_fft_complex_wavetable` 型のインスタンスを生成してポインタを返し、エラーの時には NULL を返す。データ長 n は、小さなデータサイズに分割して FFT を行うために因数分解され、基数とその三角関数の係数が表に入れられる。三角関数の係数は精度を落とさないよう、`sin` および `cos` 関数を使って直接計算される。表を速く計算するために漸化式を用いるが、プログラム中で同じデータ長に対して複数回の FFT を行うときには、表の作成は最初の一回だけでよい。それによって 2 回目以降の FFT の結果が影響を受けることはない。
>
> 表を保持する構造体のインスタンスは、同じデータ長であればそのまま何度でも再利用できる。また他の FFT 関数の呼び出しによって表の値が変わることもない。データ長が同じであれば順方向および逆方向 (および逆変換) のいずれにも同じ表が使える。

`void gsl_fft_complex_wavetable_free` (*gsl_fft_complex_wavetable* * **`wavetable`**) [Function]

> 三角関数の表 *wavetable* のインスタンスを消去し、メモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。同じデータ長での FFT をこれ以上行わない場合は、表を破棄してもよい。

以下の関数は `gsl_fft_complex_wavetable` 構造体のインスタンスが保持する三角関数表を使って演算を行う。関数内部のパラメータを直接設定する必要はないが、それらの値をチェックするとよいこともある。たとえばデータ長の因数分解は自動で行われるが、それをチェックすることで演

算に要する時間や演算誤差を見積もることができる。三角関数表を保持する構造体を下に示すが、これはヘッダファイル 'gsl_fft_complex.h' で定義されている。

gsl_fft_complex_wavetable [Data Type]

この構造体は混合基数 FFT での基数リストと三角関数表を保持し、以下の要素を持つ。

size_t n
: 複素数データの点数

size_t nf
: データ長 n を因数分解した後の基数の個数

size_t factor[64]
: 基数を保持する配列。最初の nf 個のみが使われる。

gsl_complex * trig
: 初期化関数によって確保される n 個の複素数からなる三角関数表へのポインタ

gsl_complex * twiddle[64]
: trig 中の、各基数での係数がある場所へのポインタ

混合基数法では演算の途中経過を保持するための作業領域も必要である。

gsl_fft_complex_workspace * gsl_fft_complex_workspace_alloc (*size_t* **n**) [Function]

データ長 n の複素数 FFT で使う作業領域を確保する。

void gsl_fft_complex_workspace_free (*gsl_fft_complex_workspace* * **workspace**) [Function]

作業領域 *workspace* に割り当てられているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。同じデータ長での FFT をこれ以上行わない場合は、表を破棄してもよい。

以下の関数で変換を行う。

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**) [Function]

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**) [Function]

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**) [Function]

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**) [Function]

密配置 (Pakced) な複素数配列 *data* で与えられるデータに対して、データ長 n で刻み幅 *stride* の順方向、逆方向、逆変換の混合基数 FFT を行う。データ長 n に関する制限はない。データ長 2、3、4、5、6、7 に対する高速な副変換法を内部で使う。他の基数については n に対して計算量 $O(n^2)$ の汎用の低速な変換が用いられる。関数を呼び出すときは三角関数表 *wavetable* と作業領域 *work* を指定せねばならない。関数名に `transform` が付いているものでは引数 *sign* に `forward` (-1) または `backward` (+1) を指定できる。

エラーが発生せずに変換が終了したときには 0 を返す。エラー発生時の返り値として、以下の `gsl_errno` が定義されている。

`GSL_EDOM`
: データ長 n が正の整数でない（たとえば n が 0 など)。

`GSL_EINVA`
: データ長 n と計算に用いる三角関数表 *wavetable* の大きさが一致しない。

以下にデータ長 630（$= 2 \times 3 \times 3 \times 5 \times 7$）の短いパルス波の FFT を混合基数法で計算するプログラムを示す。

```
#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) = IMAG(data,i)   = 0.0;

    data[0] = 1.0;

    for (i = 1; i <= 10; i++) REAL(data,i) = REAL(data,n-i) = 1.0;

    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 < wavetable->nf; i++)
            printf("# factor %d: %d\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;
    }
```

ここでは、プログラム中ではデフォルトのエラー・ハンドラーを設定していると仮定している (エラー発生時には abort 関数が呼び出される)。あまり安全でないエラー・ハンドラーを使う場合は、すべての GSL 関数内で返り値をチェックするべきである。

15.5 実数データに対する FFT の概要

実数データに対する関数は複素数に対する関数とほぼ同じであるが、順方向と逆変換の間に大きな違いがある。実数列に対するフーリエ変換は一般に、実数にはならない。変換結果は、以下のような対称性を持つ複素数列になる。

$$z_k = z^*_{N-k}$$

このような対称性を持つ数列を複素共役 (conjugate-complex) または半複素数 (half-complex) と呼ぶ。このため順方向 (実数から半複素数) と逆変換 (半複素数から実数) で違ったデータ配置が必要になる。したがってルーチンは二種類に分けられている。一方は実数列を変換する gsl_fft_real、もう一方は半複素数列を変換する gsl_fft_halfcomplex である。

gsl_fft_real の関数は実数の時系列データから周波数係数を計算する。実数列 x から得られる半複素数係数 c は以下のフーリエ変換で与えられる。

$$c_k = \sum_{j=0}^{N-1} x_j \exp(-2\pi i j k/N)$$

gsl_fft_halfcomplex の関数は逆変換あるいは逆方向変換を行う。以下のようにして半複素数係

数 c からフーリエ級数で実数列を再現する。

$$x_j = \frac{1}{N} \sum_{k=0}^{N-1} c_k \exp(2\pi i j k / N)$$

半複素数列は対称性を持つので、計算結果として返される数列は半分でよいことになる。返されない残りの半分は、半複素対称性から得ることができる (これはデータ長が奇数でも偶数でも同じである。偶数の場合は中央 $k = N/2$ の値は実数になる。奇数でも偶数でも、$k = 0$ の値は実数である)。この半複素数列を保持するためには N 個の実数があればよく、実数列を変換した結果は同じ大きさの配列で保持できる。

配列中にどのようにデータを置くかはアルゴリズムに依存し、基数 2 と混合基数とで異なっている。基数 2 の方法は、変換結果を元データと置き換えるため、各要素を置く場所が決まっている。混合基数の場合にはそういった場所の決まりがないため、各項の複素数の実部と虚部を隣り合わせにして置くことにしている (メモリアクセスの効率を考えるとこの配置がよい)。

15.6　実数データに対する基数 2 の FFT

この節では実数データの対する基数 2 の FFT について説明する。データ長が 2 のべき乗の場合に、元データを変換結果で置き換える、クーリーとテューキー Cooley-Tukey のアルゴリズムを使っている。

実数に基数 2 の FFT を行う関数はヘッダファイル 'gsl_fft_real.h' で宣言されている。

`int gsl_fft_real_radix2_transform` (*double* ***data[]***, *size_t* ***stride***, *size_t* ***n***) [Function]

> 与えられるデータ *data* に対してデータ長 n で刻み幅 *stride* で基数 2 の置換 FFT を行う。変換結果は半複素数列で、与えられるデータを結果で置き換える。半複素数列は以下のように配列中に配置される。$k < N/2$ に対して k 番目の要素の実部が配列の k 番目に、対応する虚部が配列の $N - k$ 番目に入れられる。$k > N/2$ の係数は対称性 $z_k = z^*_{N-k}$ から得られる。$k = 0$ および $k = N/2$ の係数はどちらも虚部のない実数になる。これらはそれぞれ配列の 0 番目と $N/2$ 番目に入れられ、虚部は 0 なのでどこにも置かれない。
>
> 実数の FFT の結果 *data* と、虚部をすべて 0 にした複素数データの FFT の結果の対応は、以下のようになる (刻み幅は 1 とする)。

```
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].real  =    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]
```

変換結果は、gsl_fft_halfcomplex_unpack を使って GSL の複素数型配列に入れ直すことができる。

半複素数データに対して基数 2 の FFT を行う関数はヘッダファイル 'gsl_fft_halfcomplex.h' で宣言されている。

`int gsl_fft_halfcomplex_radix2_inverse (`*double data []*, *size_t* ***stride***, *size_t* ***n***`)` [Function]

`int gsl_fft_halfcomplex_radix2_backward (`*double data []*, *size_t* ***stride***, *size_t* ***n***`)` [Function]

与える半複素数列 *data* に対してデータ長 *n*、刻み幅 *stride* で gsl_fft_real_radix2 を使って基数 2 の置換 FFT を行う。変換結果の実数列は、時系列として自然な順序で入れられる。

15.7 実数データに対する混合基数 FFT

この節では実数データに対する混合基数 FFT について説明する。混合基数法は任意のデータ長に対して適用できる。このライブラリで用意している関数はパウル・シュヴァルツラウバー (Paul Swarztrauber) による FORTRAN の FFTPACK ライブラリを実装し直したものである。理論的な背景はテンパートン (Clive Temperton) の記事に述べられている。このライブラリでの配列の添え字の順序や基本的なアルゴリズムは FFTPACK と同じである。

この関数は FFTPACK と同様に半複素数列を保持する。したがって実数列を変換した半複素数列は周波数 0 から昇順に、各周波数成分の実部と虚部を隣り合わせにして並べられる。虚部が 0 になることが分かっている要素の虚部は省かれる。周波数 0 に対応する成分の虚部は 0 になることが分かっているので (それは単に入力データ (全て実数である) の和になる) 省かれることになる。データ長が偶数の場合、周波数 $N/2$ に対応する成分の虚部も省かれる。これは変換結果には対称性 $z_k = z^*_{N-k}$ があり、虚部が 0 の単なる実数になるからである。

変換結果の配置は例を見るのがもっとも理解しやすい。以下の表はデータ長が $N = 5$、奇数の例である。二つの列で、gsl_fft_real_transform が返す 5 個の半複素数からなる配列 *halfcomplex []* と、同じ実数列を虚部が 0 の複素数として gsl_fft_complex_backward に与えたときに得られる複素数の配列 *complex[]* との対応を示す。

```
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]
```

配列 *complex[]* で後の方の要素 `complex[3]` と `complex[4]` の値は対称性を使って得られる。周波数 0 にあたる項 `complex[0].imag` の虚部はその対称性から 0 であることが知られている。

次の表はデータ長が偶数、$n = 6$ の例である。偶数の場合、二つの項で虚部が 0 になる。

```
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]
```

配列 *complex* で後の方の要素 `complex[4]` と `complex[5]` の値は対称性を使って得られる。`complex[0].imag` と `complex[3].imag` の値は 0 にになることが分かっている。

以下の関数の宣言はヘッダファイル '`gsl_fft_real.h`' および '`gsl_fft_halfcomplex.h`' にある。

`gsl_fft_real_wavetable * gsl_fft_real_wavetable_alloc` (*size_t* ***n***)　[Function]

`gsl_fft_halfcomplex_wavetable * gsl_fft_halfcomplex_wavetable_alloc` (*size_t* ***n***)　[Function]

データ長 n の実数列に対する FFT で使用する三角関数表を生成する。特にエラーが生じなければ新しく生成した構造体のインスタンスへのポインタを返し、エラーが発生したときは NULL を返す。n は用意されている副変換に対応する因数に分解され、その因数（=基数）と基数に対応する三角関数表が、返される構造体に入っている。三角関数の係数は精度を落とさないよう、`sin` および `cos` 関数を使って直接計算される。表を速く計算するために漸化式を用いるが、プログラム中で同じデータ長に対して複

数回の FFT を行うときには、表の作成は最初の一回だけでよい。それによって FFT の結果が影響を受けることはない。

表を保持する構造体のインスタンスは、同じデータ長であればそのまま何度でも再利用できる。また他の FFT 関数の呼び出しによって表の値が変わることもない。順方向の実数に対する変換、または半複素数列に対する逆変換にはそれぞれについて三角関数表を用意せねばならない。

`void gsl_fft_real_wavetable_free (gsl_fft_real_wavetable * wavetable)` [Function]
`void gsl_fft_halfcomplex_wavetable_free (gsl_fft_halfcomplex_wavetable * wavetable)` [Function]

三角関数表 *wavetable* に割り当てられているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。同じデータ長での FFT をこれ以上行わない場合は、表を破棄してもよい。

混合基数法では演算の途中経過を保持するための作業領域も必要である。

`gsl_fft_real_workspace * gsl_fft_real_workspace_alloc (size_t n)` [Function]

データ長 n の実数に対する FFT の作業領域を確保する。実数に対する順方向変換と半複素数列に対する逆変換の両方に同じ作業領域を使うことができる。

`void gsl_fft_real_workspace_free (gsl_fft_real_workspace * workspace)` [Function]

作業領域 *workspace* に割り当てられたメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。同じデータ長での FFT をこれ以上行わない場合は、表を破棄してもよい。

以下の関数は実数および半複素数に対する変換を行う。

`int gsl_fft_real_transform (double data[], size_t stride, size_t n, const gsl_fft_real_wavetable * wavetable, gsl_fft_real_workspace * work)` [Function]
`int gsl_fft_halfcomplex_transform (double data[], size_t stride, size_t n, const gsl_fft_halfcomplex_wavetable * wavetable, gsl_fft_real_workspace * work)` [Function]

データ長 n の実数または半複素数列 *data* を、周波数混合基数法で変換する。`gsl_fft_real_transform` では *data* は実数の時系列データである。`gsl_fft_halfcomplex_transform` では *data* は前述の順序による半複素数のフーリエ係数である。n には特に制限はない。高効率な副変換が基数 2、3、4、5 に対して用意されている。他の基数での演算は計算量が $O(n^2)$ の汎用の n 基数による遅い方法で行われる。関数呼び出しの際には三角関数表 *wavetable* と作業領域 *work* を指定する必要がある。

int gsl_fft_real_unpack (*const double* ***real_coefficient[]***, *gsl_complex_packed_array* ***complex_coefficient[]***, *size_t* ***stride***, *size_t* ***n***) [Function]

gsl_fft_complex ルーチンで使用するために、一つの実数配列 *real_coefficient* を、それと等価な複素数 (虚部が 0 の複素数) の配列 *complex_coefficient* に変換する。変換は単純で、以下のように行われる。

```
for (i = 0; i < n; i++) {
    complex_coefficient[i].real = real_coefficient[i];
    complex_coefficient[i].imag = 0.0;
}
```

int gsl_fft_halfcomplex_unpack (*const double* ***halfcomplex_coefficient[]***, *gsl_complex_packed_array* ***complex_coefficient***, *size_t* ***stride***, *size_t* ***n***) [Function]

gsl_fft_real_transform により計算される半複素数係数の配列 *halfcomplex_coefficient* を一般的な複素数配列 *complex_coefficient* に変換する。これは以下のようにして、対称性 $zk = z^*_{N-k}$ を使って冗長的な要素を計算して埋める。

```
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];
    double hc_imag = halfcomplex_coefficient[2 * i];
    complex_coefficient[i].real = hc_real;
    complex_coefficient[i].imag = hc_imag;
    complex_coefficient[n - i].real = hc_real;
    complex_coefficient[n - i].imag = -hc_imag;
}
if (i == n - i) {
    complex_coefficient[i].real = halfcomplex_coefficient[n - 1];
    complex_coefficient[i].imag = 0.0;
}
```

以下に gsl_fft_real_transform と gsl_fft_halfcomplex_inverse を使ったプログラムを示す。プログラムでは方形パルスの実数信号を生成する。この信号は周波数領域にフーリエ変換され、gsl_fft_real_transform が返すフーリエ係数のうち低周波数の要素 10 個を残して、それ以外は消去する。

残ったフーリエ係数を時間領域に逆変換することで、方形パルスにローパスフィルタをかけた信号をシミュレートする。フーリエ係数は半複素対称で保持されているので、周波数が正の領域と負の領域の両方で係数が消去され、逆変換で得られる時系列信号は実数列になる。

```
#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++) 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;
}
```

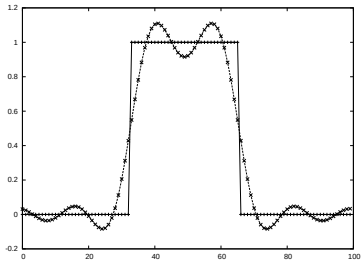


図 15.2: 例示したプログラムによる、ローパスフィルタをかけられた実数のパルス。

15.8　参考文献

FFT について更に理解を深めたいときは、デュアメル (Pierre Duhamel) とヴェターリ (Martin Vetterli) による以下のレビュー記事から見るとよい。

- P. Duhamel, M. Vetterli, "Fast fourier transforms: A tutorial review and a state of the art", *Signal Processing*, **19**(4), pp. 259–299 (1990).

GSL で使われているアルゴリズムは、GSL のパッケージに付属している *GSL FFT Algorithms* に説明されている (ファイルは '`doc/fftalgorithms.tex`' である)。この文書に FFT についての一般的な説明と各関数の実装に関する具体的な式の導出がある。他に、いくつか重要な文献を以下に列挙する。

サンプルプログラム付きの FFT の入門書がいくつかある。以下に二つ紹介する。

- E. Oran Brigham, *The Fast Fourier Transform*, Prentice Hall (1974).
- C. S. Burrus and T. W. Parks, *DFT/FFT and Convolution Algorithms*, Wiley (1984).

上の二つの本では基数 2 の FFT が詳しく説明されている。

FFTPACK の真髄でもある混合基数法は以下の論文に説明されている。

- Clive Temperton, "Self-sorting mixed-radix fast fourier transforms", *Journal of Computational Physics*, **52**(1), pp. 1–23 (1983).

実数データに対する FFT の導出は以下の二つの記事にある。

- 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), pp. 849–863 (1987).
- Clive Temperton, "Fast mixed-radix real fourier transforms", *Journal of Computational Physics*, **52**, pp. 340–350 (1983).

1979 年に IEEE は、FORTRAN で書かれた FFT 関連のプログラムを慎重に審査して選び出し、その大綱を発表した。各種 FFT アルゴリズムを実装する際に参考になる。

- *Programs for Digital Signal Processing*, Digital Signal Processing Committee and IEEE Acoustics, Speech, and Signal Processing Committee (ed.), IEEE Press (1979).

重要なプログラムで FFT を使いたいときには、フリゴ (Matteo Frigo) とジョンソン (Steven G. Johnson) による FFTW ライブラリがよい。このライブラリは使用するハードウェア・プラットフォームに合わせて実行速度の最適化を行う。GNU GPL の元で使うことができる。

- FFTW Website, `http://www.fftw.org/`

FFTPACK のソースコードは Netlib から得ることができる。

- FFTPACK, `http://www.netlib.org/fftpack/`

本文中で DFT の定義の記述法で比較されている「ニュメリカル・レシピ」は、以下の書籍である。C 言語でアルゴリズムの実装が収録されている邦訳版 (初版の翻訳) がある。

- William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes*, Cambridge University Press, ISBN 978-0521308113 (1986).
- William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery (丹慶勝市, 佐藤俊郎, 奥村晴彦, 小林誠訳), ニューメリカルレシピ・イン・シー 日本語版 - C 言語による数値計算のレシピ, 技術評論社, ISBN 978-4874085608 (1993)

第16章　数値積分

この章では一次元の関数に対する数値積分 (求積法、quadrature) を実行するルーチンについて説明する。このライブラリでは汎用のものとして適応型 (adaptive) と非適応型 (non-adaptive) のルーチンを用意しており、他にいくつかの特殊なケースに特化したルーチンもある。それには無限および半無限 (semi-infinite) の領域での積分、特異積分 (singular integral)、対数特異点 (logarithmic singularity) を含む積分、コーシーの主値 (Cauchy principal value)、振動する関数の積分などがある。GSL で実装しているのはピセンズ (Robert Piessens)、ドンカー=カペンガ (Elise de Doncker-Kapenga)、ユーバーフーバー (Christoph W. Uberhuber)、カハナー (David K. Kahaner) による数値積分ライブラリの QUADPACK で使われているアルゴリズムを実装し直したものである。QUADPACK の FORTRAN の ソースコードは Netlib から入手できる。

この章で説明する関数はヘッダイファイル '`gsl_integration.h`' で宣言されている。

16.1　はじめに

各アルゴリズムでは以下の形式の有限の積分値の近似値を計算する。

$$I = \int_a^b f(x)w(x)dx$$

ここで $w(x)$ は重み関数 (weight function、一般的な被積分関数では $w(x) = 1$) である。この積分値 I を推定する時の計算精度は、絶対許容誤差 ($epsabs$) と相対許容誤差 ($epsrel$) によって以下のように指定される。

$$|RESULT - I| \leq \max(epsabs, epsrel|I|)$$

ここで $RESULT$ は各アルゴリズムによって得られる積分の近似値である。解析的な真の積分値 I は実際には不明であり、各アルゴリズムでは以下の不等式を満たす絶対誤差 $ABSERR = |RESULT - I|$ を推定する。

$$|RESULT - I| \leq ABSERR \leq \max(epsabs, epsrel|I|)$$

$epsabs$ と $epsrel$ は必ずしも両方とも指定する必要はない。絶対誤差の範囲だけを指定したいときには $epsrel$ を 0 にする。相対誤差だけを指定したいときには $epsabs$ を 0 にする。要求精度が厳しすぎると積分ルーチンは収束できないことがあるが、常にその時点での最良の近似解を返すようになっている。

QUADPACK で使われているアルゴリズムは以下のような命名規則にしたがっている。

`Q` - 求積法ルーチン (quadratur routine)

N - 非適応型積分法 (non-adaptive integrator)

A - 適応型積分法 (adaptive integrator)

G - 汎用 (general integrand, 被積分関数を別途定義、指定する)

W - 指定された被積分関数に重み関数をかける (weight function with integrand)

S - 特異点を持つ関数を高速に積分する (singlarities)

P - 積分が困難な点を指定できる (points of special difficulty)

I - 無限区間での積分 (inifinite range)

O - 周期振動する重み関数を使う (oscillatory weight function, sin または cos)

F - フーリエ積分 (Fourier integral)

C - コーシーの主値 (Cauchy principal value)

実装しているアルゴリズムでは、内部で高次と低次の二種類の求積法を使っている。高次の方法では限られた小さな積分範囲で積分値を精密に求める。これを低次での積分値と比べることで、積分値の近似計算の誤差を推定する。

16.1.1　重み関数のない被積分関数の場合

GSL で実装している汎用の求積法 (重み関数を使わない) は、ガウス・クロンロッド (Gauss-Kronrod, Johann Carl Friedrich Gauss, Aleksandr Semenovich Kronrod) 法である。

ガウス・クロンロッド法は最初に、古典的な m 次のガウスの求積法を行う。これに対して横軸上の各点から $2m+1$ 次の高次のクロンロッド法を行う。クロンロッド法では、ガウス法で求めた関数値を再利用して無駄を省いている。

高次のクロンロッド法で求められた値をルーチンの計算結果 (積分値の近似値) とし、高次の方法と低次の方法の積分値の違いを使って近似誤差を推定する。

16.1.2　重み関数のある積分の場合

被積分関数に重み関数が含まれている場合は、クレンショーとカーティス (Clenshaw-Curtis, Charles William Clenshaw, A. R. Curtis) の求積法を使う。

クレンショーとカーティスの方法はまず n 次のチェビシェフ多項式 (Chebyshev polynomial) で被積分関数を近似する。この近似多項式の積分値は解析的に求められ、被積分関数の積分の近似値とすることができる。チェビシェフ展開は次数を上げることで近似精度を上げ、計算誤差を推定することができる。

16.1.3 特異点を持つ重み関数のある積分の場合

チェビシェフ近似では、被積分関数に特異点などがあると収束が遅くなる。QUADPACK で実装されている修正クレンショー・カーティス法では、よく使われる重み関数のうち収束が遅くなるものについて、そのためのルーチンを別に用意している。

まずチェビシェフ多項式を解析的に積分して重み関数の修正チェビシェフ・モーメントをあらかじめ計算しておく。それと、関数を近似するチェビシェフ多項式から、目的の積分が得られる。特異点の周囲を解析的に積分することで、どの計算値がキャンセルするか正確に知ることができ、結果的に積分全体の収束性を向上することができる。

16.2 QNG 法 - 非適応型ガウス・クロンロッド積分

QNG アルゴリズムは横軸上に固定幅で最大 87 点まで被積分関数の値を計算するガウス・クロンロッド・パターソン (Gauss-Kronrod-Patterson, Thomas N. L. Patterson) 法である。滑らかな関数を高速に積分するのに適している。

`int gsl_integration_qng` (*const gsl_function * f*, *double a*, *double b*, *double epsabs*, *double epsrel*, *double * result*, *double * abserr*, *size_t * neval*) [Function]

> 関数 f の区間 (a, b) での積分値の推定誤差が引数で指定された絶対および相対許容誤差 *epsabs*、*epsrel* 内に収束するまで、10 点、21 点、43 点、87 点のガウス・クロンロッド法を順番に適用していく。積分の近似値が *result* に、推定絶対誤差が *abserr* に、被積分関数の評価回数が *neval* にそれぞれ入れて返される。被積分関数の評価回数を減らすため評価した関数値を全て保持、利用するように実装されている。

16.3 QAG 法 - 適応型積分

QAG 法は簡単な適応型積分計算を行う。積分範囲を分割し、分割された各区間のうち推定誤差が最大の区間を二等分する。そしてまた各区間での誤差を推定し、最大の区間を二等分する。積分の難しい場所に計算量が集中することで、全体での誤差を大幅に減少することができる。`gsl_integration_workspace` 構造体で分割した各積分区間の範囲、積分結果、推定誤差を保持する。

`gsl_integration_workspace * gsl_integration_workspace_alloc` (*size_t n*)[Function]

> n 個の区間での積分結果と推定誤差を倍精度で保持するための作業領域を確保する。

`void gsl_integration_workspace_free` (*gsl_integration_workspace * w*) [Function]

> 作業領域 w のメモリを解放する。

`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*) [Function]

関数 f に対して適応型積分計算を、区間 (a, b) での積分値の推定誤差が、与えられる絶対および相対許容誤差 *epsabs*、*epsrel* 内に収まるように収束するまで適用する。積分の近似値が *result* に、推定絶対誤差が *abserr* にそれぞれ入れて返される。適用する積分法は、以下に示す *key* の値で指定される。

```
GSL_INTEG_GAUSS15 (key = 1)
GSL_INTEG_GAUSS21 (key = 2)
GSL_INTEG_GAUSS31 (key = 3)
GSL_INTEG_GAUSS41 (key = 4)
GSL_INTEG_GAUSS51 (key = 5)
GSL_INTEG_GAUSS61 (key = 6)
```

上から順に 15 点、21 点、31 点、41 点、51 点、61 点のガウス・クロンロッド法に対応する。高次の積分法は、滑らかな関数を精度よく積分することができる。低次の方法は、被積分関数に不連続な点があるなど、積分が難しい点を含む場合に、計算時間を短縮できる。

適応型積分計算の繰り返し計算の各回では、推定誤差が最大の積分区間を二等分する。各区間とその区間の推定積分値は *workspace* が指すメモリに保持される。区間の個数の上限を *limit* で与える。これは確保した作業領域で保持できそうな個数にしておく。

16.4 QAGS 法 - 特異点に対応した適応型積分

積分範囲内に特異点がある場合、適応型積分計算では特異点の周辺に小区間が集中していくが、作られる小区間の幅はだんだんと小さくなっていくため、積分がうまく近似されていれば、極限では積分値が収束する。この収束は補外 (extrapolation) を使って加速することができる。QAGS 法では、適応型の二等分法にウィン (Peter Wynn) のイプシロン・アルゴリズム (epsilon, ϵ algorithm) を組み合わせることで様々な形式の特異点に対して高速に積分を行うことができる。

`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*) [Function]

関数 f の区間 (a, b) での積分を、21 点のガウス・クロンロッドを使って推定絶対誤差と推定相対誤差が指定された値 *epsabs* と *epsrel* に収束するまで計算する。推定積分値はイプシロン法を使って補外されたものであり、不連続な点や積分における特異点がある被積分関数に対して収束を加速することができる。積分の近似値が *result* に、推定絶対誤差が *abserr* にそれぞれ入れて返される。各区間とその区間での推定積分値は *workspace* が指すメモリに保持される。確保した作業領域で保持できる程度で、区間の個数の上限を *limit* で与える。

16.5 QAGP 法 - 特異点が分かっている関数に対する適応型積分

`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***) [Function]

指定された特異点を考慮して、適応型積分 QAGS 法を行う。大きさ $npts$ の配列 pts に、積分の区分点として積分範囲の両境界の座標と特異点の座標を入れる。例えば積分範囲が (a, b) で特異点が x_1, x_2, x_3 $(a < x_1 < x_2 < x_3 < b)$ にあるとき、pts には以下のように値を入れ、$npts = 5$ とする。

```
pts[0] = a
pts[1] = x_1
pts[2] = x_2
pts[3] = x_3
pts[4] = b
```

特異点の座標が分かっている場合には QAGS 法を使うよりもこの方が計算が速い。

16.6 QAGI 法 - 無限区間に対する適応型積分計算

`int gsl_integration_qagi` (*gsl_function * **f***, *double **epsabs***, *double **epsrel***, *size_t **limit***, *gsl_integration_workspace * **workspace***, *double * **result***, *double * **abserr***) [Function]

関数 f の区間 $(-\infty, +\infty)$ での積分値を計算する。$x = (1-t)/t$ として積分範囲を半開区間 $(0, 1]$ に移して、QAGS 法で以下の数値積分を行う。

$$\int_{-\infty}^{+\infty} f(x)\mathrm{dx} = \int_0^1 (f((1-t)/t) + f(-(1-t)/t))/t^2\mathrm{dt}$$

QAGS 法では 21 点のガウス・クロンロッド法を使うが、区間を移すことにより原点に特異点が生じるため、ここでは 15 点で積分を行う。この場合は低次の計算法の方が効率がよい。

`int gsl_integration_qagiu` (*gsl_function * **f***, *double* `a`, *double **epsabs***, *double **epsrel***, *size_t **limit***, *gsl_integration_workspace * **workspace***, *double * **result***, *double * **abserr***) [Function]

関数 f の半無限区間 $(a, +\infty)$ での積分値を計算する。$x = a + (1-t)/t$ として積分範囲を半開区間 $(0, 1]$ に移して、QAGS 法で以下の積分を行う。

$$\int_a^{+\infty} f(x)\mathrm{dx} = \int_0^1 f(a + (1-t)/t)/t^2\mathrm{dt}$$

`int gsl_integration_qagil` (*gsl_function* * **`f`**, *double* **`b`**, *double* ***`epsabs`***, *double* ***`epsrel`***, *size_t* ***`limit`***, *gsl_integration_workspace* * ***`workspace`***, *double* * ***`result`***, *double* * ***`abserr`***) [Function]

関数 f の半無限区間 $(-\infty, b)$ での積分値を計算する。$x = b - (1-t)/t$ として積分範囲を半開区間 $(0, 1]$ に移して、QAGS 法で以下の積分を行う。

$$\int_{-\infty}^{b} f(x)\mathrm{dx} = \int_{0}^{1} f(b-(1-t)/t)/t^2\mathrm{dt}$$

16.7　QAWC 法 - コーシーの主値の適応型積分

`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`***) [Function]

区間 (a, b) で、c が特異点の関数 f について、以下で与えられるコーシーの主値を計算する。

$$I = \int_a^b \frac{f(x)}{x-c}\mathrm{dx} = \lim_{\epsilon\to 0}\left(\int_a^{c-\epsilon}\frac{f(x)}{x-c}\mathrm{dx} + \int_{c+\epsilon}^{b}\frac{f(x)}{x-c}\mathrm{dx}\right)$$

QAG (適応型二分法) を使うが、特異点 $x = c$ 上で区間分割が行われないように修正して積分する。小区間が点 $x = c$ を含むか、その点に近い場合、25 点のクレンショー・カーティス法が使われる。特異点から遠い区間では 15 点のガウス・クロンロッド法を使う。

16.8　QAWS 法 - 特異点を持つ関数のための適応型積分

QAWS 法では、積分範囲の境界上に代数的対数による特異点 (algebraic-logarithmic singular point) を持つ関数の積分値が計算できる。計算を高速に行うため、あらかじめチェビシェフ・モーメントを計算しておく必要がある。

`gsl_integration_qaws_table * gsl_integration_qaws_table_alloc` (*double* **`alpha`**, *double* **`beta`**, *int* **`mu`**, *int* **`nu`**) [Function]

以下で定義される特異点のある重み関数 $W(x)$ とそのためのパラメータ $(\alpha, \beta, \mu, \nu)$ を保持するための作業領域を `gsl_integration_qaws_table` として確保する。

$$W(x) = (x-a)^{\alpha}(b-x)^{\beta}\log^{\mu}(x-a)\log^{\nu}(b-x)$$

ここで $\alpha > -1,\ \beta > -1,\ \mu = 0, 1,\ \nu = 0, 1$ である。重み関数は μ と ν の値によっ

て、以下の四つの形式のうちのどれかを取る。

$$W(x) = (x-a)^\alpha (b-x)^\beta \qquad (\mu = 0, \nu = 0)$$
$$W(x) = (x-a)^\alpha (b-x)^\beta \log(x-a) \qquad (\mu = 1, \nu = 0)$$
$$W(x) = (x-a)^\alpha (b-x)^\beta \log(b-x) \qquad (\mu = 0, \nu = 1)$$
$$W(x) = (x-a)^\alpha (b-x)^\beta \log(x-a) \log(b-x) \qquad (\mu = 1, \nu = 1)$$

特異点 (a, b) は境界上の点なので、数値積分を行うときに指定すればよく、この表を作る段階では必要ない。

計算中にエラーが発生しなかった場合には `gsl_integration_qaws_table` 構造体へのポインタが返され、エラーが発生した場合は 0 が返される。

`int gsl_integration_qaws_table_set` (*gsl_integration_qaws_table * **t***, *double* ***alpha***, *double* ***beta***, *int* ***mu***, *int* ***nu***) [Function]

すでに確保されている `gsl_integration_qaws_table` のインスタンス t に設定されているパラメータ $(\alpha, \beta, \mu, \nu)$ の値を引数で指定する値で設定し直す。

`void gsl_integration_qaws_table_free` (*gsl_integration_qaws_table * **t***) [Function]

すでに確保されている `gsl_integration_qaws_table` のインスタンス t のメモリを解放する。

`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***) [Function]

区間 (a, b) で特異点のある重み関数が $(x-a)^\alpha (b-x)^\beta \log^\mu (x-a) \log^\nu (b-x)$ のときの関数 f の積分値を計算する。重み関数のパラメータ $(\alpha, \beta, \mu, \nu)$ は t に設定しておく。以下の積分が行われる。

$$I = \int_a^b f(x)(x-a)^\alpha (b-x)^\beta \log^\mu (x-a) \log^\nu (b-x) \mathrm{dx}$$

積分法には QAG (適応型二分法) が使われる。小区間が境界点を含む場合、25 点のクレンショー・カーティス法が使われる。そうでない場合は 15 点のガウス・クロンロッド法を使う。

16.9 QAWO 法 - 振動する関数のための適応型積分

QAWO 法は $\sin(\omega x)$ や $\cos(\omega x)$ のような振動する要素を持つ重み関数を使って積分値を計算するための方法である。計算を高速に行うため、あらかじめ以下の関数を呼んでチェビシェフ・モーメントを計算しておく必要がある。

gsl_integration_qawo_table * gsl_integration_qawo_table_alloc (*double* **omega**, *double* **L**, *enum gsl_integration_qawo_enum* **sine**, *size_t* **n**) [Function]

以下の振動重み関数 $W(x)$ とそのパラメータ (ω, L) を保持するための作業領域として gsl_integration_qawo_table のインスタンスを生成する。

$$W(x) = \left\{ \begin{array}{l} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

引数 L は積分範囲全体の幅 $L = b - a$ を指定する。引数 *sine* に以下の二つの値のどちらかを使うことで、重み関数に使う関数を正弦関数と余弦関数のどちらにするかを指定する。

```
GSL_INTEG_COSINE
GSL_INTEG_SINE
```

gsl_integration_qawo_table は積分計算で必要になる三角形数の表である。パラメータ n は計算される係数のレベル数を指定する。各レベルは積分範囲 L を一回二等分することに相当し、したがって n 個のレベルがあれば区間の幅を $L/2^n$ まで小さくすることができる。積分ルーチン gsl_integration_qawo はレベル数が足りなくて要求される精度で計算ができないとき、エラーとして GSL_ETABLE を返す。

int gsl_integration_qawo_table_set (*gsl_integration_qawo_table * **t**, *double* **omega**, *double* **L**, *enum gsl_integration_qawo_enum* **sine**) [Function]

すでに確保されている作業領域 t に設定されているパラメータ *omega*、*L*、*sine* の値を変更する。

int gsl_integration_qawo_table_set_length (*gsl_integration_qawo_table * **t**, *double* **L**) [Function]

作業領域 t に設定されているパラメータ L の値を設定し直す。

void gsl_integration_qawo table free (*gsl_integration_qawo_table * **t**)[Function]

作業領域 t のメモリを解放する。

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**) [Function]

適応型積分で関数 f の区間 (a, b) での積分値を、*wf* で定義されている重み関数 $\sin(\omega x)$ または $\cos(\omega x)$ を使って計算する。

$$I = \int_a^b f(x) \left\{ \begin{array}{l} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\} \mathrm{dx}$$

積分値の収束はイプシロン法を使って加速され、返される積分値はそれによって補外された値になる。補外による値が近似積分値として *result* に、推定絶対誤差が *abserr* に入れて返さる。分割された区間と各区間での積分値を保持する作業領域を *workspace* に指定する。確保した作業領域で保持できる程度で、区間の個数の上限を *limit* で与える。

$d\omega > 4$ となる「大きな」区間に対しては振動の数も多いことを想定して 25 点のクレンショー・カーティス法が使われる。$d\omega < 4$ となる「小さな」(せまい) 区間に対しては 15 点のガウス・クロンロッド法を使う。

16.10 QAWF 法 - フーリエ積分のための適応型積分

`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`**) [Function]

以下のように表される、半無限区間 $[a, +\infty)$ での関数 f のフーリエ積分を計算する。

$$I = \int_a^{+\infty} f(x) \left\{ \begin{array}{c} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\} \mathrm{dx}$$

パラメータ ω と、sin と cos のどちらを使うかは、表 *wf* に指定しておく (長さ L はフーリエ積分に適した値にこの関数の内部で変更されるので、どんな値になっていてもよい)。積分は以下のように、各小区間について QAWO 法で行われる。

$$C_1 = [a, a + c] \tag{16.1}$$
$$C_2 = [a + c, a + 2c] \tag{16.2}$$
$$\ldots = \ldots \tag{16.3}$$
$$C_k = [a + (k - 1)c, a + kc] \tag{16.4}$$

ここで $c = (2\mathrm{floor}(|\omega|) + 1)\pi/|\omega|$ である。幅 c は小区間の数が奇数になるようにとられる。すると関数 f が正で単調減少の時に、各小区間の寄与は符号を交互に変えながら単調減少する。この各項の寄与の数列の和の計算は、イプシロン法を使って収束加速により計算される。

積分値の計算は絶対誤差を *abserr* 以下にするように行われる。このアルゴリズムでは各小区間 C_k で誤差を以下に示す TOL_k 以下に抑えるように計算する。

$$TOL_k = u_k \, abserr$$

ここで $u_k = (1 - p)p^{k-1}$ および $p = 9/10$ である。各項の寄与は等比数列であり、その和は全体での最大誤差 *abserr* である。

積分が困難な小区間があるときは、その区間での要求精度を以下のように下げる。

$$TOL_k = u_k \max(abserr, \max_{i<k}\{E_i\})$$

ここで E_k は区間 C_k での推定誤差である。

小区間とそこでの積分値は *workspace* に保持される。確保した作業領域で保持できる程度で、区間の個数の上限を *limit* で与える。各小区間での積分計算は引数で与えられる QAWO 法のための作業領域 *cycle_workspace* を使用する。

16.11　エラーコード

この章で説明した数値積分の関数は、標準の適切でない引数を示すエラーコードに加え、以下のエラーコードも返す。

`GSL_EMAXITER`
: 積分中に小区間の個数が最大個数を超えたことを示す。

`GSL_EROUND`
: 丸め誤差のために許容誤差に到達できなかったか、補外に使う表で丸め誤差が生じたことを示す。

`GSL_ESING`
: 指定された区間内で、特異点や被積分関数の挙動が積分不可能であることを示す。

`GSL_EDIVERGE`
: 数値積分として積分値が発散するか、収束が非常に遅いことを示す。

16.12　例

積分法 QAGS は多種の積分法を扱うことができる。たとえば原点が代数的対数の特異点である以下の積分を考えてみる。

$$\int_0^1 x^{-1/2}\log(x)dx = -4$$

以下のプログラムはこの積分を、相対誤差 `1e-7` で計算する。

```
#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;
    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 = %d\n", w->size);

    gsl_integration_workspace_free(w);
    return 0;
}
```

プログラムの出力を以下に示す。積分区間の数が 8 のときに、要求精度を満たす結果を得ている。

```
$ ./a.out
result          = -3.999999999999973799
exact result    = -4.000000000000000000
estimated error =  0.000000000000246025
actual error    =  0.000000000000026201
intervals =  8
```

この結果は、QAGS 法による補外で、精度の桁数が約 2 倍になったことによる。収束加速のための補外法によって得られる推定誤差は、実際の誤差よりも余裕を持って見積もられており、ここでは約 1 桁大きな値になっている (実行環境により、`exact result` 以外の数値は異なることがある)。

16.13　参考文献

以下の書籍は QUADPACK 開発者によって書かれた決定版である。アルゴリズムの解説、プログラムのリスト、テストプログラムと例が載っている。数値積分を行う上での注意点や QUADPACK 開発で使われた参考文献も載っている。

- R. Piessens, E. de Doncker-Kapenga, C. W. Uberhuber, D. K. Kahaner, QUADPACK *A subroutine package for automatic integration*, Springer Verlag (1983).

第17章　乱数の生成

GSL には様々な乱数の生成法が用意されており、すべて同じ API (Application Programming Interface) で利用することができる。プログラムの実行時に設定されている環境変数によって、乱数発生器 (random number generator) の種類や乱数の種 (seed) を選ぶことができるため、プログラムの再コンパイルをしなくても実行時にそれらを切り替えて使うことができる。乱数発生器のインスタンスはそれぞれ、その時の状態や設定 (state) を個別に保持しているため、マルチスレッドで実行されるプログラムでも問題なく使うことができる。発生した一様乱数 (uniform random number) から、正規分布 (Gaussian distribution)、対数正規分布 (log-normal d.)、ポアソン分布 (Poisson d.) など、各種の連続および離散分布に変換する関数も用意されている。

関数はヘッダファイル '`gsl_rng.h`' で宣言されている。

17.1　乱数に関する注意

1988 年のパーク (Stephen K. Park) とミラー (Keith W. Miller) による論文 ("Random number generators: good ones are hard to find.", *Communications of the ACM*, **31**(10), pp. 1192–1201) によると、優れた乱数発生器がすでにいくつもあるにもかかわらず、よくないものが未だ広く使われている。計算機システムに付属の乱数発生器でいいこともあるかもしれないが、一般に計算機の速度向上にしたがって、乱数発生器に要求されることも多くなってきている。今日では、乱数を数百万個も生成するようなシミュレーションも、コーヒーを片手にほんの少し休憩している間に終わってしまう。

ピエール・レキュエル (Pierre L'Ecuyer) が書いた *Handbook on Simulation*, Jerry Banks, ed. Wiley (1997) の第 4 章が非常によい参考になる。この文章は彼のウェブサイト (章末参照) から PostScript 形式で入手することができる。クヌース (Donald E. Knuth) の「準数値計算法」(原著は 1968 年刊、章末参照) も乱数生成法に関して 170 ページを割いており、最近改訂 3 版が出ている (1997 年)。これは非常に優れた定番の本である。もしまだ持っていないのなら、この GSL リファレンス・マニュアルを今すぐ横に置いて、書店に走っていってクヌースの本を買い、そっちを先に読むべきである。

優れた乱数発生器は理論的性質と統計的性質の両方の面で優れていなければならない。理論的な優秀さを確保するのは難しいことである (高度な数学を要求するため)。しかし一般に、周期が長く、線形従属性が低く、「平面に乗ってしまう」ことのないような性質が望まれる。乱数をシミュレーションに用いるときには、その前に統計的な検定を行ってその乱数の性質を調べるべきである。一般的には、確率論によって厳密な解が得られるような何らかの問題に対して、定量的な推定を行うために乱数発生器が使われる。その厳密な解と乱数を比較して「無作為性 (randomness)」を (定量的に) 評価する。

17.2　乱数発生器の使い方

乱数発生器関数は、例えば正弦関数や余弦関数のようないわゆる「本物」の関数ではない (「関数」とは、独立変数がある値を取るとき、従属変数の値が一意に決まる写像である)。そういった関数と違って乱数発生器関数が返す値は、うまく計算できたときには毎回違う値を返す。乱数発生器はまさそのための関数だが、これを実現するためには関数内部に発生器の「状態 (state)」を示す変数を保持しておく必要がある。状態は一つの整数で表されることもあるが (単に直前に発生した乱数の値のこともある)、生成する乱数全体を保持する配列のことである。その場合、配列の添え字を指定することで乱数を返す。GSL で用意している乱数発生器は、そういった状態の管理法や、アルゴリズムによって異なる処理の詳細などは知らなくても利用できる。

GSL の乱数発生器は二種類の構造体を使っている。`gsl_rng_type` では各種の乱数発生器ついて、その情報を静的 (static) に保持する。`gsl_rng` では `gsl_rng_type` で示される型 (type) の乱数発生器のインスタンスについての情報を保持する。

この節で説明する関数はヘッダファイル '`gsl_rng.h`' で宣言されている。

17.3　乱数発生器の初期化

`gsl_rng * gsl_rng_alloc` (*const gsl_rng_type * T*) [Function]

型 T の乱数発生器のインスタンスを生成して、そのポインタを返す。たとえば以下のコードではトーズワース (Robert C. Tausworthe) の乱数発生器のインスタンスを生成する。

```
gsl_rng * r = gsl_rng_alloc(gsl_rng_taus);
```

十分なメモリが確保できないときは、null ポインタを返しエラー・コード `GSL_ENOMEM` でエラー・ハンドラーを呼ぶ。生成された乱数発生器は、特に指定がなければ `gsl_rng_default_seed` で初期化される。その値はデフォルトでは 0 だが、直接に、あるいは環境変数 `GSL_RNG_SEED` を使って変更することができる (17.6 節「乱数発生器が参照する環境変数」、221 ページ参照)。

利用できる乱数発生器の種類については後述する (第 17.9 節「乱数発生アルゴリズム」、223 ページ参照)。

`void gsl_rng_set` (*const gsl_rng * r*, *unsigned long int s*) [Function]

乱数発生器に「種 (seed) を与える」ことで、乱数発生器を初期化する (seeding、シーディング)。乱数発生器の型が同じであれば、同じ値の種 (s、ただし $s \geq 1$) によって初期化されればいつも同じ乱数系列を生成する (したがって、種の値を記録しておくことで再現性のあるシミュレーションを行うことができる)。s に異なる値を与えて呼び出した場合は、全く異なる乱数系列が生成される。s に 0 を与えると、乱数発生器の各型ごとに実装されているデフォルト値が種として使われる。たとえば乱数発生器 `ranlux` の元となった FORTRAN 版では種の規定値は 314159265 であり、その GSL

版 gsl_rng_ranlux を使うとき、s に 0 を与えると 0 の代わりにこの値が種として使われる。

`void gsl_rng_free (`*gsl_rng* * `r``)` [Function]

乱数発生器のインスタンス r に割り当てられているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

17.4 乱数発生器を使った乱数の生成

以下の関数は、整数または倍精度浮動小数点実数で一様乱数を生成して返す。一様でない分布の乱数の生成については、19 章「確率分布と乱数」、241 ページを参照のこと。

`unsigned long int gsl_rng_get (`*const gsl_rng* * `r``)` [Function]

乱数発生器 r を使って乱数を整数で返す。返せる値の下限と上限は使うアルゴリズムによって異なるが、返される値は [*min*,*max*] の範囲内で一様の確率で生成される整数である。*min* と *max* の値は別の関数 gsl_rng_max(r) と gsl_rng_min(r) を使って知ることができる。

`double gsl_rng_uniform (`*const gsl_rng* * `r``)` [Function]

範囲 $[0,1)$ 内に一様に分布する乱数を一つ生成して、倍精度浮動小数点実数で返す。範囲内に 0.0 は含まれるが 1.0 は含まれない。 これは普通、関数 gsl_rng_get(r) が返す値を gsl_rng_max(r) + 1.0 で (倍精度で) 除した値である。乱数発生器の種類によっては除算を関数内部で独自に行い、32 ビット以上の無作為性を得られるものもある (発生する乱数の無作為性の最大ビット数は一つの unsigned long int の整数で表現されているので、移植性は高い)。

`double gsl_rng_uniform_pos (`*const gsl_rng* * `r``)` [Function]

範囲 $(0,1)$ 内に一様に分布する乱数を一つ生成し、正の倍精度浮動小数点実数で返す。範囲に 0.0 と 1.0 は含まれない。乱数は、gsl_rng_uniform アルゴリズムを 0.0 でない値を生成するまで呼び出すことで発生する。0.0 が特異点となるような計算に使える。

`unsigned long int gsl_rng_uniform_int (`*const gsl_rng* * `r`, *unsigned long int* `n`)` [Function]

乱数発生器 r が生成する乱数をスケーリングし、場合によっては捨てることで、0 以上 $n-1$ 以下の乱数を生成して返す。返される整数は、使われるアルゴリズムの種類によらず $[0,n-1]$ の範囲内で一様分布である。使われるアルゴリズムによって発生する乱数の最小値が違うため、0 の発生確率を正しくするための操作が内部で行われる。

この乱数発生器は、内部で使う乱数発生器の生成範囲よりも狭い範囲で乱数を生成するように設計されている。したがって n は乱数発生器 r の範囲の大きさ以内でなけれ

ばならない。n が乱数発生器の発生する乱数の最大値よりも大きい場合は、エラー・ハンドラーをエラー・コード GSL_EINVAL で呼び出し、0 を返す。

特に、この関数は符号なし整数の取りうる範囲 $[0, 2^{32} - 1]$ 全体を想定してはいない。そうしたい場合は、gsl_rng_ranlxd1、gsl_rng_mt19937、gsl_rng_taus などのアルゴリズムを gsl_rng_get() で直接使うべきである。各乱数発生器の生成範囲は、次の補助関数の節で説明する。

17.5　乱数発生器の補助関数

生成した乱数発生器のインスタンスに関する情報を参照、操作するための補助関数について以下に説明する。乱数発生のパラメータはプログラム中にハード・コーディング (直接値を書き込んでおくこと) してしまわずに、これらの関数を使うようにするのが望ましい。

`const char * gsl_rng_name` (*const gsl_rng * r*) [Function]

乱数発生器の名前の文字列へのポインタを返す。たとえば以下のようにすると、

```
printf ("r is a '%s' generator\n", gsl_rng_name (r));
```

`r is a 'taus' generator` というような出力が得られる。

`unsigned long int gsl_rng_max` (*const gsl_rng * r*) [Function]

gsl_rng_get が返す値の最大値を返す。

`unsigned long int gsl_rng_min` (*const gsl_rng * r*) [Function]

gsl_rng_get が返す値の最小値を返す。普通はこの値は 0 になるが、0 を返せないアルゴリズムもあり、そういったものに対しては 1 を返す。

`void * gsl_rng_state` (*const gsl_rng * r*) [Function]
`size_t gsl_rng_size` (*const gsl_rng * r*) [Function]

乱数発生器 r の状態変数へのポインタとその大きさを返す。この関数で、状態変数を直接参照、操作することができる。たとえば以下のコードでは、乱数発生器の状態をファイルに出力する。

```
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*) [Function]

利用できる乱数発生器のすべての型の名前を文字列として持つ配列へのポインタを返す。配列の最後の要素は NULL ポインタである。プログラムの実行時、必要に応じて最初に一度だけ呼ぶのが望ましい。以下のコードは、乱数発生器の型を保持する配列を使って、利用できるアルゴリズムの種類を表示する。

```
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);
```

17.6 乱数発生器が参照する環境変数

デフォルトで使用される乱数発生アルゴリズムと種は、環境変数 GSL_RNG_TYPE と GSL_RNG_SEED および関数 gsl_rng_env_setup で指定することができる。これを利用することで、プログラムを再コンパイルすることなく、様々なアルゴリズムや種を容易に切り替えて試すことができる。

const gsl_rng_type * gsl_rng_env_setup (*void*) [Function]

環境変数 GSL_RNG_TYPE および GSL_RNG_SEED の値を取得し、GSL で用意している変数 gsl_rng_default と gsl_rng_default_seed に対応する値を設定する。これらは大域変数 (global variable) として以下のように定義されている。

```
extern const gsl_rng_type *gsl_rng_default
extern unsigned long int gsl_rng_default_seed
```

環境変数 GSL_RNG_TYPE の値には taus や mt19937 など、乱数発生器の名前を指定する。環境変数 GSL_RNG_SEED の値は、使いたい種の値にする。その値は、C 言語の標準ライブラリ関数 strtoul によって unsigned long int 型に変換されてから種として使われる。

GSL_RNG_TYPE で乱数発生器を指定しない場合は、gsl_rng_mt19937 が規定値として使われる。gsl_rng_default_seed の初期値は 0 である。

以下に環境変数 GSL_RNG_TYPE と GSL_RNG_SEED を使って大域的なスコープで利用できる乱数発生器インスタンスを生成する短いプログラムを示す。

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

gsl_rng * r; /* 使う乱数発生器を大域変数として宣言 */

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;
}
```

環境変数を設定せずにこのプログラムを実行すると、乱数発生器には mt19937 が、種には 0 が使われる。

```
bash$ ./a.out
generator type: mt19937
seed = 0
first value = 4293858116
```

コマンドラインで二つの環境変数を設定すると、これらの規定値を変更できる。

```
bash$ 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
```

17.7　乱数発生器の状態の複製

上述した方法では、乱数発生器を呼び出すごとに変化していく、その「状態」については考慮されない。しかしこれを保存、読み出したいような状況 (たとえばシミュレーションを中断、再開したい場合など) を想定し、そのための関数を用意している。

`int gsl_rng_memcpy (`*gsl_rng* * **`dest`**, *const gsl_rng* * **`src`**`)` [Function]

乱数発生器 *src* をすでに生成している乱数発生器のインスタンス *dest* に、全く同じ (状態を含めて) になるようにコピーする。*src* と *dest* は同じ型でなければならない。

`gsl_rng * gsl_rng_clone (`*const gsl_rng* * **`r`**`)` [Function]

乱数発生器 *r* の全く同じコピーを生成し、そのインスタンスへのポインタを返す。

17.8　乱数発生器の状態の読み込みと保存

このライブラリでは、ファイルにたいして乱数発生器の状態を、バイナリ形式で読み書きできる関数を用意している。

int gsl_rng_fwrite (*FILE * stream*, *const gsl_rng * r*) [Function]

乱数発生器 *r* の状態をファイル *stream* にバイナリ形式で書き込む。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。データはプラットフォーム依存のバイナリ形式で書き込まれるため、異なるハードウェア間での移植性はない。

int gsl_rng_fread (*FILE * stream*, *gsl_rng * r*) [Function]

乱数発生器 *r* に乱数の状態を、開いているファイル *stream* からバイナリ形式で読み込む。乱数発生器の型の情報は保存されないため、*r* は正しく初期化されていなければならない。書き込みが成功すれば 0 を、失敗すれば GSL_EFAILED を返す。読み込まれるデータは、同じプラットフォーム上で書かれたバイナリ形式であると想定される。

17.9 乱数発生アルゴリズム

GSL には、シミュレーションに利用できるもの、他のライブラリとの互換性を保つためのもの、昔からある古典的なものなど様々な乱数発生器があるが、これまでに説明した関数では、実際に使われるアルゴリズムを参照、操作することはできない。これはプログラムのソースコードを変更することなく乱数発生アルゴリズムを切り替えられるようにするためである。

以下の乱数発生器は、シミュレーションに利用できる高品質なものである。周期が長く、発生した乱数間の相関が低く、多くの統計検定にパスできる。相関を持たない数値の発生源としては、改良版 (2nd generation) の RANLUX がもっとも無作為性ががあり、信頼性の高いアルゴリズムである。

gsl_rng_mt19937 [Generator]

松本眞と西村拓士による MT19937 は「メルセンヌ・ツイスター」という名前で知られており、一般化ひねりフィードバック・シフト・レジスタ (twisted generalized feedback shift-register) 型アルゴリズムである。この乱数の周期は、メルセンヌ素数 (Mersenne prime) $2^{19937}-1$ (約 10^{6000}) であり、623 次元空間で均等に分布する。統計的検定 DIEHARD でも合格である。一つの乱数発生器について 624 ワードで状態を保持し、速度も他の乱数発生器に比べて遜色はない。メルセンヌ・ツイスターの種の値は元々 4357 であり、*s* を 0 にして呼び出すと gsl_rng_set はこの値を使う。改良版のメルセンヌ・ツイスターでは 5489 を使うようになったが、GSL ルーチンで種をその値にしたいときは gsl_rng_set でその値を明示的に指定する。詳細は以下を参照のこと。

> Makoto Matsumoto, Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", *ACM Transactions on Modeling and Computer Simulation*, **8**(1), pp. 3–30 (1998).

gsl_rng_mt19937 は、同じ著者らが発表した改良版 (2002 年) の方法で初期化を行う。元の方法では、種の値によってはおかしな挙動を示すことがあるが、その元のアルゴリズムも gsl_rng_mt19937_1999 と gsl_rng_mt19937_1998 で利用できる。

gsl_rng_ranlxs0 [Generator]
gsl_rng_ranlxs1 [Generator]
gsl_rng_ranlxs2 [Generator]

乱数発生器 ranlxs0 は、リュシャー (Martin Lüscher) による RANLUX (“luxury random numbers”) の改良版アルゴリズムである。この乱数発生器は三段階の「贅沢さ (luxury)」レベル ranlxs0、 ranlxs1、ranlxs2 を持ち (後者ほど贅沢レベルが高い)、単精度 (24 ビット) で乱数を出力する。内部では倍精度実数で演算を行っており、特に 64 ビット CPU 上では整数版の ranlux に比べ非常に速い。乱数の周期はおおよそ 10^{171} である。このアルゴリズムの性質の良さは数学的に証明されており、どの程度の無作為性が保証されるかを示すことができる。贅沢レベルを上げると発生する数値間の相関が減り、また保証された無作為性レベルに対して安全マージンを取ることができる。

gsl_rng_ranlxd1 [Generator]
gsl_rng_ranlxd2 [Generator]

乱数発生器 ranlxs を使って倍精度 (48 ビット) の乱数を発生する。二段階の贅沢レベルについて、ranlxd1 および ranlxd2 の二つの関数を用意している。

gsl_rng_ranlux [Generator]
gsl_rng_ranlux389 [Generator]

ranlux はリュシャーの元々のアルゴリズムによる乱数発生器である。これは「贅沢な乱数」(“luxury random numbers”) を発生するために、間欠ずれフィボナッチ・アルゴリズム (lagged-fibonacci-with-skipping) を使っており、元々 IEEE の単精度実数のために作られた 24 ビットの乱数である。内部では整数演算を行っているため、上述した浮動小数点を使う第二世代の ranlxs と ranlxd の方が高速であることが多い (プラットフォームによって異なる)。乱数の周期は約 10^{171} である。このアルゴリズムも数学的に各種の性質が証明されており、どのくらいの無作為性レベルで発生する乱数の間には相関がないかがわかっている。リュシャーによる無相関レベルがデフォルト値になっているアルゴリズムが gsl_rng_ranlux で、無相関レベルが最高のアルゴリズムが gsl_rng_ranlux389 で使える。最高レベルでは 24 ビットで相関をなくすことができる。どちらの乱数発生器も 24 ワードの状態変数を使う。詳細については、以下を参照のこと。

> M. Lüscher, “A portable high-quality random number generator for lattice field theory calculations”, *Computer Physics Communications*, **79**(1), pp. 100–110 (1994).
> F. James, “RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Lüscher”, *Computer Physics Communications*, **79** pp. 111–114 (1994).

gsl_rng_cmrg [Generator]

レキュエル (Pierre L'Ecuyer) の結合多重再帰 (combined multiple recursive generator) による乱数発生器である。以下の漸化式で乱数を生成する。

$$z_n = (x_n - y_n) \bmod m_1$$

ここで x_n と y_n は はそれぞれ、アルゴリズムの内部で使われる乱数発生器であり、以下で与えられる。

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \bmod m_1$$
$$y_n = (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \bmod m_2$$

係数は $a_1 = 0, a_2 = 63308, a_3 = -183326, b_1 = 86098, b_2 = 0, b_3 = -539608$ であり、モジュラは $m_1 = 2^{31} - 1 = 2147483647$ および $m_2 = 2145483479$ である。

乱数の周期は $m_1^3 - 1$ と $m_2^3 - 1$ の最小公倍数であり、約 2^{185} (約 10^{56}) である。この乱数発生器は 6 ワードの状態変数を使う。詳細については以下を参照のこと。

> P. L'Ecuyer, "Combined Multiple Recursive Random Number Generators", *Operations Research*, **44**(5), pp. 816–822 (1996).

`gsl_rng_mrg` [Generator]

レキュエル (Pierre L'Ecuyer)、ブローイン (François Blouin)、クチュール (Raymond Couture) による 5 次の結合多重再起による乱数発生器である。以下の漸化式で乱数を生成する。

$$x_n = (a_1 x_{n-1} + a_5 x_{n-5}) \bmod m$$

ここで $a_1 = 107374182, (a_2 = a_3 = a_4 = 0,)a_5 = 104480$ および $m = 2^{31} - 1$ である。

乱数の周期は約 10^{46} である。この乱数発生器は 5 ワードの状態変数を使う。詳細については、以下を参照のこと。

> P. L'Ecuyer, F. Blouin, R. Couture, "A search for good multiple recursive random number generators", *ACM Transactions on Modeling and Computer Simulation*, **3**(2), pp. 87–98 (1993).

`gsl_rng_taus` [Generator]
`gsl_rng_taus2` [Generator]

レキュエル (Pierre L'Ecuyer) による、等分散性を最大にしたトーズワース (Tausworthe) 型乱数発生器である。乱数は以下の漸化式で発生される。

$$x_n = (s_n^1 \oplus s_n^2 \oplus s_n^3)$$

ここで

$$s_{n+1}^1 = (((s_n^1 \& 4294967294) \ll 12) \oplus (((s_n^1 \ll 13) \oplus s_n^1) \gg 19))$$
$$s_{n+1}^2 = (((s_n^2 \& 4294967288) \ll 4) \oplus (((s_n^2 \ll 2) \oplus s_n^2) \gg 25))$$
$$s_{n+1}^3 = (((s_n^3 \& 4294967280) \ll 17) \oplus (((s_n^3 \ll 3) \oplus s_n^3) \gg 11))$$

を 2^{32} を法として計算する。上の式で $\oplus$ は「排他的論理和 (exclusive or)」を表す。このアルゴリズムは 32 ビット整数での演算を 64 ビット計算機上でも実行できるように、`0xFFFFFFFF` をビットマスクとして使うように実装されている。

乱数の周期は 2^{88}(約 10^{26}) で、状態変数として 3 ワードを使う。詳細は以下を参照のこと。

> P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators", *Mathematics of Computation*, **65**(213), pp. 203–213 (1996).

乱数発生器 `gsl_rng_taus2` は、`gsl_rng_taus` と同じアルゴリズムだが、種を生成する方法を以下の論文による方法に変更している。

> P. L'Ecuyer, "Tables of Maximally Equidistributed Combined LFSR Generators", *Mathematics of Computation*, **68**(225), pp. 261–269 (1999).

`gsl_rng_taus` よりも `gsl_rng_taus2` を用いる方が好ましい。

gsl_rng_gfsr4 [Generator]

乱数発生器 `gfsr4` は、ずれフィボナッチ (lagged-fibonacci) 法に似ており、直前までの 4 回で発生した乱数の排他的論理和 `xor` を次の乱数値とする。

$$r_n = r_{n-A} \oplus r_{n-B} \oplus r_{n-C} \oplus r_{n-D}$$

後述のジフ (Robert M. Ziff) の文献によると、「よく知られている」二点法 (two-tap registers, 後述の R250 など) は、その定義から三点相関 (three-point correlation) が生じてしまうなど、深刻な欠点がある。一般フィードバック・シフト・レジスタ (GFSR, Generalized Feedback Shift Register) では数学的によい性質を保つことができ、特に四点式 (Four-tap GFSR) では、うまくシフト量を選べば、数値的にも優れていることが著者による検定で示されている。

このライブラリでは、ジフの文献の 392 ページの値を使っている。そこでは、$A = 471, B = 1586, C = 6988, D = 9689$ である。

シフト量を適切に選ぶことで、乱数の周期を最大にすることができる (GSL ではそうしている)。最大のシフト量を D とすると、最大周期は 2^{D-1} になる (最大周期は 2^D よりも 1 だけ小さいのは、配列 `ra[]` ではすべての要素が 0 になってはならないように実装されているためである)。この実装では $D = 9689$ であり、周期は約 10^{2917} である。

このライブラリの実装では、32 ビット整数の乱数発生器が、1 ビットの乱数発生器を 32 個、並列に実行するのと同じであることを利用している。つまり、32 ビット乱数発生器の周期と 1 ビット 乱数発生器の周期は同じである。各 1 ビット乱数発生器はそれぞれ独立であると見なせるため、32 ビットのビットパターンはすべて等確率で発生しうる。したがって、0 も乱数値として発生しうる (GFSR 型乱数発生器特有のこの性質は、ヘイコ・バウケ (Heiko Bauke) が GSL 開発チームに知らせてくれた。感謝する)。詳細は以下を参照のこと。

Robert M. Ziff, “Four-tap shift-register-sequence random-number generators”, *Computers in Physics*, **12**(4), pp 385–392 (1998).

17.10 Unix の乱数発生器

Unix の標準ライブラリ関数に含まれている乱数発生器 `rand`、`random`、`rand48` も、GSL で用意している。これらの関数は非常に多くのプラットフォーム上で利用できるが、これらすべてを利用できるプラットフォームは、そう多くない。したがって、これらの関数を使っても移植性の高いプログラムを書くことは難しいため、GSL でまとめて用意している。しかしこれらの乱数発生器はあまり高品質ではなく、統計的な精密さ、正確さを要求するような用途には適さない。そういった統計的な利用法でなく、プログラム動作に何か変化を出したいときなどには便利である。

`gsl_rng_rand` [Generator]

BSD の乱数発生器 `rand`。乱数を以下の漸化式で生成する。

$$x_{n+1} = (a x_n + c) \bmod m$$

ここで $a = 1103515245, c = 12345, m = 2^{31}$ である。種の値は直接、最初に生成される乱数の値 x_1 となる。この乱数発生器での乱数の周期は 2^{31} で、状態変数として 1 ワードを使う。

`gsl_rng_random_bsd` [Generator]
`gsl_rng_random_libc5` [Generator]
`gsl_rng_random_glibc2` [Generator]

元々 BSD で使われていた線形フィードバック・シフト・レジスタ (linear feedback shift register) 型の乱数発生器 (`random` 関連の関数群) を実装したものである。現在、`random` にはいくつかの版があるが、GSL では元の BSD 版 (たとえば SunOS4 など)、libc5 版 (GNU/Linux のものなど)、glibc2 版が利用できる。各版では種の使い方が異なるため、異なる乱数系列を発生する。

BSD 版は乱数発生器の状態変数として可変長の変数を利用することができ、長い変数を使えば、乱数の品質、無作為性を上げることができる。`random` では変数の長さとして 8、32、64、128、256 バイトを利用するアルゴリズムがそれぞれ実装されており、利用者が指定した変数長以下で最大のものが実際に使われる。このアルゴリズムを利用する関数は、それぞれ以下に示す名前が付けられている。

```
gsl_rng_random8_bsd
gsl_rng_random32_bsd
gsl_rng_random64_bsd
gsl_rng_random128_bsd
gsl_rng_random256_bsd
```

数字は変数長を表す。元々の BSD 版では random 関数は 128 バイトを使っており、gsl_rng_random_bsd は gsl_rng_random128_bsd と同じである。libc5 および glibc2 に対応した版もそれぞれ用意されており、gsl_rng_random8_libc5 や gsl_rng_random8_glibc2 といった名前で利用できる。

gsl_rng_rand48 [Generator]

Unix の rand48 乱数発生器である。以下の漸化式で乱数を生成する。

$$x_{n+1} = (ax_n + c) \bmod m$$

ここで $a = 25214903917, c = 11, m = 2^{48}$ であり、これらは 48 ビットの符号なし整数である。与えられた種は、最初の乱数値 x_1 の上位 32 ビットに使われる。下位 16 ビットは 0x330E に決められている。関数 gsl_rng_get は、漸化式から得られる乱数の上位 32 ビットを返す。元々の rand48 の関数群は、そのままの形では GSL では用意していないが、この関数の返り値の型を long int に変換することで mrand48 と同じ出力を得られる。また関数 gsl_rng_uniform は、48 ビットの内部状態全体を使って倍精度実数値 x_n/m を返し、drand48 と同じ動作を行う。以前の GNU C ライブラリでは mrand48 にバグがあって、発生する乱数が異なるものがあった (返り値の下位 16 ビットだけが使われる)。

17.11　その他の乱数発生器

以下に既存のライブラリとの互換性を保つために用意されている乱数発生器について説明する。すでにあるプログラムを GSL を利用するように修正する場合、ここに上げる関数を使って、元のプログラムと変更後のものの違いを確認し、同じ動作であることを確認してから、より性能の良い乱数発生器に切り替えればよい。

ここに上げる乱数発生器の多くは線形合同法 (linear congruence relation) のルーチンであり、これはもっとも単純な乱数生成方式の一つである。線形合同法は、特に非素数を法としたときに性能が悪くなるが、そういった関数も以下に含まれている (たとえば 2 のベキ乗 2^{31} や 2^{32} を法とするものがある)。この場合、発生する乱数の下位ビットが周期性を持つことになり、無作為性は上位ビットに偏ることになる。こういった関数を使う場合は、発生した乱数の上位ビットだけを使うようにすべきである。

gsl_rng_ranf [Generator]

CRAY の乱数発生器 RANF である。以下の漸化式で乱数を発生する。

$$x_{n+1} = (ax_n) \bmod m$$

変数は 48 ビット符号なし整数であり、ここで $a = 44485709377909, m = 2^{48}$ である。与える種は最初に発生する乱数 x_1 値の下位 32 ビットに使われるが、種が偶数になるのを避けるため、最下位ビットは 1 にセットされる。x_1 の上位 16 ビットは 0 にセッ

トされる。したがってこの漸化式では、種が 2 のときと 3 のとき、4 のときと 5 のときなど、同じ乱数系列を与える種の組み合わせがある。

この関数は CRAY の MATHLIB に含まれる RANF と互換である。GSL で用意しているものは CRAY のものと同様に、倍精度浮動小数点実数を返す。

GSL ではこのアルゴリズムの種の処理の実装に、少し工夫をしている。初期状態は、逆剰余 (modular inverse) $a \bmod m$ を乗ずることで 1 ステップ逆戻りするようになっている。こうすることで、CRAY のオリジナル版と同じ動作をするようになっている。

種に指定できる最大値は 2^{32} である。元の CRAY 版では移植性のないワイド整数型を使うことで、最大 2^{48} の乱数の状態を表現できるようになっている。

関数 `gsl_rng_get` は漸化式による値の上位 32 ビットを返す。関数 `gsl_rng_uniform` は 48 ビットすべてを使って、乱数を倍精度実数 x_n/m で返す。

この乱数の周期は 2^{46} である。

`gsl_rng_ranmar` [Generator]

RANMAR はマルサグリア (George Marsaglia)、ザマン (Arif Zaman)、ツァン (Wai Wan Tsang) による遅れフィボナッチ (lagged-fibonacci) 型の乱数発生器である。これは元々 IEEE の単精度浮動小数点実数として 24 ビットの乱数を発生するものである。高エネルギー物理学用のライブラリ CERNLIB の一部である。

`gsl_rng_r250` [Generator]

カークパトリック (Scott Kirkpatrick) とストール (Erich P. Stoll) によるシフト・レジスタ型の乱数発生器である。以下の漸化式で乱数を発生する。

$$x_n = x_{n-103} \oplus x_{n-250}$$

ここで $\oplus$ は 32 ビットのワードに定義された「排他的論理和 (exclusive-or)」である。乱数の周期は約 2^{250} で、状態変数として 250 ワードを使う。

詳細については以下を参照のこと。

> S. Kirkpatrick and E. Stoll, "A very fast shift-register sequence random number generator", *Journal of Computational Physics*, **40**, pp. 517–526 (1981).

`gsl_rng_tt800` [Generator]

初期のひねり一般化フィードバック・シフト・レジスタ (twisted generalized feedback shift-register) 型の乱数発生器で、後に MT19937 に発展するものの原型である。しかし現在でも通用する性能を持っている。乱数の周期は 2^{800} で、状態変数は 33 ワードである。

詳細については、以下を参照のこと。

Makoto Matsumoto and Yoshiharu Kurita, "Twisted GFSR Generators II", *ACM Transactions on Modelling and Computer Simulation*, **4**(3), pp. 254–266 (1994).

`gsl_rng_vax` [Generator]

VAX の乱数発生器 `MTH$RANDOM` である。以下の漸化式で乱数を発生する。

$$x_{n+1} = (ax_n + c) \bmod m$$

ここで、$a = 69069, c = 1, m = 2^{32}$ である。種の値は最初に発生する乱数の値 x_1 に使われる。乱数の周期は 2^{32} で状態変数は 1 ワードである。

`gsl_rng_transputer` [Generator]

INMOS Transputer Development system による乱数発生器である。以下の漸化式で乱数を発生する。

$$x_{n+1} = (ax_n) \bmod m$$

ここで $a = 1664525$、$m = 2^{32}$ である。種の値は最初に発生する乱数の値 x_1 に使われる。

`gsl_rng_randu` [Generator]

IBM の `RANDU` 乱数発生器である。以下の漸化式で乱数を発生する。

$$x_{n+1} = (ax_n) \bmod m$$

ここで $a = 65539, m = 2^{31}$ である。種の値は最初に発生する乱数の値 x_1 に使われる。乱数の周期は 2^{29} にすぎない。現在では、低品質の乱数の見本のようなものである。

`gsl_rng_minstd` [Generator]

パーク (Stephen K. Park) とミラー (Keith W. Miller) による「最小標準 ("minimal standard")」乱数発生器 MINSTD である。単純な線形合同法だが、線形合同法アルゴリズムの持つ大きな落とし穴を避けるように実装されている。以下の漸化式で乱数を発生する。

$$x_{n+1} = (ax_n) \bmod m$$

ここで $a = 16807$ で $m = 2^{31} - 1 = 2147483647$ である。種の値は最初に発生する乱数の値 x_1 に使われる。乱数の周期は約 2^{31} である。

この乱数発生器は IMSL ライブラリ (サブルーチン RNUN) と MATLAB (関数 RAND) に実装されており、"GGL" とも呼ばれている (なんの略かはわからないが)。

詳細については、以下を参照のこと。

S. K. Park, K. W. Miller, "Random Number Generators: Good ones are hard to find", *Communications of the ACM*, **31**(10), pp. 1192–1201 (1988).

gsl_rng_uni [Generator]
gsl_rng_uni32 [Generator]

16 ビット SLATEC の乱数発生器 RUNIF である。これを 32 ビットに拡張したものが gsl_rng_uni32 である。元の版のソースコードは NETLIB にある。

gsl_rng_slatec [Generator]

SLATEC の乱数発生器 RAND である。これは非常に古い。元の版のソースコードは NETLIB にある。

gsl_rng_zuf [Generator]

ペーターゼン (Wesley P. Petersen) による遅れフィボナッチ (lagged-fibonaci) 型の乱数発生器 ZUFALL である。以下の漸化式で乱数を発生する。

$$t = u_{n-273} + u_{n-607}$$
$$u_n = t - \mathrm{floor}(t)$$

元の版のソースコードは NETLIB にある。詳細については、以下を参照のこと。

> W. P. Petersen, "Lagged Fibonacci Random Number Generators for the NEC SX-3", *International Journal of High Speed Computing*, **6**(3), pp. 387–398 (1994).

gsl_rng_knuthran2 [Generator]

これはクヌースの *Seminumerical Algorithms*, 3rd Ed., p. 108 にある、二次の多重再帰による乱数発生器である。以下の漸化式で乱数を発生する。

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2}) \bmod m$$

ここで $a_1 = 271828183, a_2 = 314159269, m = 2^{31} - 1$ である。

gsl_rng_knuthran2002 [Generator]
gsl_rng_knuthran [Generator]

これはクヌースの *Seminumerical Algorithms*, 3rd Ed., Section 3.6 にある、二次多重再帰による乱数発生器である。クヌースによるC言語のプログラムがある。gsl_rng_knuthran2002 は、同書の第 9 刷で gsl_rng_knuthran にあったいくつかの欠点を修正したものである。

gsl_rng_borosh13 [Generator]
gsl_rng_fishman18 [Generator]
gsl_rng_fishman20 [Generator]
gsl_rng_lecuyer21 [Generator]
gsl_rng_waterman14 [Generator]

これはクヌースの *Seminumerical Algorithms*, 3rd Ed., pp. 106–108 にある、二次多重再帰による乱数発生器である。以下の漸化式で乱数を発生する。

$$x_{n+1} = (ax_n) \bmod m$$

種の値は最初に発生する乱数の値 x_1 になる。パラメータ a と m の値は、それぞれ以下である。

algorithm	a	m
Borosh-Niederreiter	1812433253	2^{32}
Fishman18	62089911	$2^{31} - 1$
Fishman20	48271	$2^{31} - 1$
L'Ecuyer	40692	$2^{31} - 249$
Waterman	1566083941	2^{32}

`gsl_rng_fishman2x` [Generator]

レキュエル (Pierre L'Ecuyer) とフィシュマン (George S. Fishman) による乱数発生器である。ここでの実装はクヌースの *Seminumerical Algorithms*, 3rd Ed.、p. 108 による。以下の漸化式で乱数を発生する。

$$z_{n+1} = (x_n - y_n) \bmod m$$

ここで $m = 2^{31} - 1$ である。x_n と y_n の値は、`fishman20` と `fishman21` アルゴリズムにより生成する。種の値は最初に発生する乱数の値 x_1 になる。

`gsl_rng_coveyou` [Generator]

カビュー (Robert R. Coveyou) よる乱数発生器である。ここでの実装はクヌースの *Seminumerical Algorithms*, 3rd Ed., 第 3.2.2 節による。以下の漸化式で乱数を発生する。

$$x_{n+1} = (x_n(x_n + 1)) \bmod m$$

ここで $m = 2^{32}$ である。種の値は最初に発生する乱数の値 x_1 に使われる。

17.12　性能、品質

GSL で用意している乱数発生器の性能比較を以下の表に示す。シミュレーションに適用できる品質を持つもので最も速いのは `taus`、`gfsr4`、`mt19937` である。数学的に最も優れた品質を持つものは RANLUX を使っているものである。

```
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

1852 k ints/sec,935 k doubles/sec, ran3
 813 k ints/sec,575 k doubles/sec, ran0
 787 k ints/sec,476 k doubles/sec, ran1
 379 k ints/sec,292 k doubles/sec, ran2
```

17.13 例

範囲 $[0.0, 1.0)$ の一様乱数を発生するための、乱数発生器の使用例を以下のプログラムで示す。

```
#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;
    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;
}
```

このプログラムの出力を以下に示す。

```
$ ./a.out
0.99974
0.16291
0.28262
0.94720
0.23166
0.48497
0.95748
0.74431
0.54004
0.73995
```

表示される乱数の値は、乱数発生器に与える種の値によって異なる。発生する乱数系列を変えるには、種の値の規定値を環境変数 GSL_RNG_SEED で変更すればよい。使用する乱数発生器の種類も環境変数 GSL_RNG_TYPE で切り替えることもできる。以下に、種の値に 123、乱数発生器として重回帰乱数発生器 mrg を使う例を示す。

```
$ GSL_RNG_SEED=123 GSL_RNG_TYPE=mrg ./a.out
GSL_RNG_TYPE=mrg
GSL_RNG_SEED=123
0.33050
0.86631
0.32982
0.67620
0.53391
0.06457
0.16847
0.70229
0.04371
0.86374
```

17.14　参考文献

乱数発生器とその検定については、クヌースの *Seminumerical Algorithms* に幅広く解説されている。

- Donald E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms* Vol. 2, (3rd Ed), Addison-Wesley, ISBN 0201896842 (1997).

さらに詳細については、ピエール・レキュエルの論文に述べられている。

- P. L'Ecuyer, "Random Number Generation", Chapter 4 of *the Handbook on Simulation*, Jerry Banks Ed., Wiley, pp. 93–137 (1998).

http://www.iro.umontreal.ca/~lecuyer/papers.html にある 'handsim.ps' というファイル。

乱数発生器の検定を行うプログラム DIEHARD のソースコードもオンラインで入手できる。

- G. Marsaglia, "The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness", http://stat.fsu.edu/pub/diehard/

乱数の検定法は、NIST が網羅的にまとめている。

- "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications", NIST Special Publication 800-22.
  http://csrc.nist.gov/rng/

17.15 備考

乱数発生器のソースコードのライセンスを GPL にしてくれた松本眞、西村拓士、栗田良春の各氏に感謝する (MT19937, MM&TN; TT800, MM&YK)。マーティン・リュシャー Martin Lüscher は RANLXS と RANLXD のソースコードその他を提供してくれた。感謝する。

第18章　準乱数系列

この章では、任意次元の準乱数系列 (quasi-random sequence) を発生する関数について説明する。準乱数系列は、d 次元空間に一様に分布する点を次々に発生していく。準乱数系列は、重複の少ない数値の列とも捉えられる。準乱数系列発生器の使い方は、乱数発生器と同じであるが、種を設定する必要がない。そのため、いつも同じ順番で同じ数値を出力する。

この節で説明する関数の宣言はヘッダファイル 'gsl_qrng.h' にある。

18.1　準乱数発生器の初期化

gsl_qrng * gsl_qrng_alloc (*const gsl_qrng_type * T*, *unsigned int d*) [Function]

指定された型 T を使う d 次元の準乱数発生器のインスタンスを生成し、そのインスタンスへのポインタを返す。生成のための十分なメモリが確保できないときは、NULL ポインタを返し、エラーコード GSL_ENOMEN でエラーハンドラーを呼び出す。

void gsl_qrng_free (*gsl_qrng * q*) [Function]

準乱数発生器のインスタンス q が持つメモリを解放する (引数に NULL ポインタを与えたときは何もしない)。

void gsl_qrng_init (*gsl_qrng * q*) [Function]

すでに生成されている準乱数発生器のインスタンス q を、再初期化する。準乱数系列では、種を設定する必要がなく、いつも同じ順番で同じ数値を出力する。

18.2　準乱数系列の発生

int gsl_qrng_get (*const gsl_qrng * q*, *double x[]*) [Function]

準乱数発生器のインスタンス q で、それまでに発生した準乱数の次の準乱数を発生して配列 x に返す。x の次元と準乱数発生器の次元は一致していなければならない。発生する点 x の各要素 x_i について、$0 < x_i < 1$ となる。HAVE_INLINE が定義されているときは、インライン展開される。

18.3　準乱数系列に関連する関数

`const char * gsl_qrng_name` (*const gsl_qrng * q*) [Function]

準乱数発生器の型の名前を格納している文字列へのポインタを返す。

`size_t gsl_qrng_size` (*const gsl_qrng * q*) [Function]
`void * gsl_qrng_state` (*const gsl_qrng * q*) [Function]

準乱数発生器のインスタンス r の状態を表すデータの構造体の大きさと、その構造体へのポインタを返す。準乱数発生器の状態を直接参照、操作するのに使う。たとえば以下のコードは、準乱数発生器のそのときの状態を出力する。

```
void * state = gsl_qrng_state(q);
size_t n = gsl_qrng_size(q);
fwrite(state, n, 1, stream);
```

18.4　準乱数発生器の状態の保存と読み出し

`int gsl_qrng_memcpy` (*gsl_qrng * dest*, *const gsl_qrng * src*) [Function]

準乱数発生器 *src* を、すでに確保した準乱数発生器のインスタンス *dest* に複製する。二つのインスタンスは同じ型でなければならない。

`gsl_qrng * gsl_qrng_clone` (*const gsl_qrng * q*) [Function]

準乱数発生器 q の複製を新たに生成し、そのインスタンスへのポインタを返す。

18.5　準乱数発生アルゴリズム

以下のアルゴリズムが用意されている。

`gsl_qrng_niederreiter2` [Generator]

ブラトリー (Paul Bratley)、フォックス (Bennett L. Fox)、ニーダーライター (Harald Niederreiter) によるアルゴリズム (*ACM Transactions on Modeling and Computer Simulation*, **2**(3), pp. 195–213, 1992)。最大 12 次元までの点を発生することができる。

`gsl_qrng_sobol` [Generator]

アントノフ (I. A. Antonov)、サレーブ (V. M. Saleev) によるソボル列 (Sobol sequence) を使う方法 ("An economic method of computing LP_τ - Sequences", *USSR Computational Mathematics and Mathematical Physics*, **19**, pp. 252–256, 1980)。これは 40 次元まで発生できる。

`gsl_qrng_halton` [Generator]
`gsl_qrng_reverse_halton` [Generator]

ハルトン系列および逆ハルトン系列を生成する方法 (J. H. Halton, "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals", *Numerische Mathematik*, **2**, pp. 84–90, 1960 および B. Vandewoestyne, R. Cools, "Good permutations for deterministic scrambled Halton sequences in terms of L_2-discrepancy", *Computational and Applied Mathematics*, **189**(1&2), pp. 341–361, 2006)。どちらも 1229 次元まで利用可能である。

18.6 例

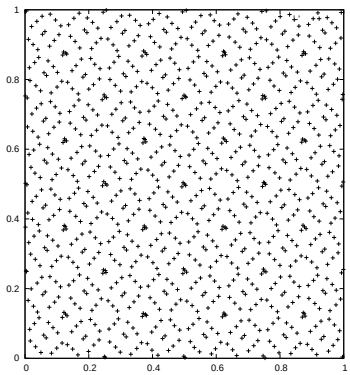
以下に示すプログラムでは、2 次元空間内の 1024 個の点をソボル列を使う方法で生成する。

```
#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++) {
        double v[2];
        gsl_qrng_get(q, v);
        printf("%.5f %.5f\n", v[0], v[1]);
    }

    gsl_qrng_free(q);
    return 0;
}
```

プログラムの出力を以下に示す。

```
$ ./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
....
```

進行に伴い、これまでに現れた点の間に次々と新しい点が現れ、だんだんと空間を覆っていく様子を見ることができる。以下にソボル列の方法で生成した最初の 1024 個の点を xy 平面にプロットしたものを示す。



準乱数のソボル列による最初の 1024 点の分布。

18.7　参考文献

準乱数系列関数の実装は、以下の論文のアルゴリズムによる。

- P. Bratley, B.L. Fox, H. Niederreiter, “Algorithm 738: Programs to Generate Niederreiter’s Low-discrepancy Sequences”, *ACM Transactions on Mathematical Software*, **20**(4), pp. 494–495 (1994).

第19章　乱数を使った確率分布

この章では、乱数を発生する関数およびその確率分布 (probability distribution) を計算する関数について説明する。GSL で実装している確率分布にしたがう乱数は、GSL 乱数発生器ならどれを使っても発生させることができる。

一様でない乱数を発生するもっとも簡潔な方法は、得たい分布になるような変換を一様乱数に加えることである。そうすると 1 個の乱数を発生するのに 1 回だけ乱数発生器を使う。また、得たい分布と似てることがあらかじめ分かっている分布から発生した乱数を、得たい分布と比較して、それを使うかやめるかを決める試行錯誤的な方法もある。この場合は一般に、目的の分布にしたがう乱数を 1 個生成するのに、乱数発生器を複数回呼ぶことになる。

GSL では累積分布関数 (cumulative distribution function, CDF) とその逆関数 (inverse cumulative distribution function, 分位関数 quantile function とも) を用意している。累積分布関数とその逆関数は、分布関数の上側 (上の裾、upper tail) と下側 (下の裾、lower tail) をそれぞれ別に計算するため、確率が小さくなる場合でも精度を保つことができる。

この章の乱数分布と確率密度分布 (probability density function, PDF) の関数は '`gsl_randist.h`' で宣言されている。それらに対応する累積分布関数は '`gsl_cdf.h`' で宣言されている。

離散値をとる乱数関数は常に `unsigned int` 型で値を返す。多くのプラットフォームでは、この型の取りうる値の上限は $2^{32} - 1 \approx 4.29 \times 10^9$ である。乱数を発生させるときには確率分布のパラメータを引数として関数を呼ぶが、パラメータの値によっては大きな数値 (で表される事象) の発生確率が高くなり、`unsigned int` 型を超える値の発生確率が無視でなくなるような場合もあり得る。そうした事態は避けなければならず、引数の値が安全な範囲に収まっているかどうかに気をつけなければならない。

19.1　はじめに

連続値をとる乱数の分布は確率密度関数 $p(x)$ で定義され、これは事象 x が x から $x + dx$ までの微小区間に生じる確率が $p\ dx$ であることを表す。累積分布関数 $P(x)$ の下側確率は以下の積分で表され、発生する事象が x より小さくなる確率を与える。

$$P(x) = \int_{-\infty}^{x} p(x')dx'$$

累積分布関数の上側確率 $Q(x)$ は以下の積分で表され、発生する事象が x より大きくなる確率を与える。

$$Q(x) = \int_{x}^{+\infty} p(x')dx'$$

この二つの関数には $P(x)+Q(x)=1$ という関係があり、また $0 \leq P(x) \leq 1, 0 \leq Q(x) \leq 1$ である。

逆累積分布関数 $x=P^{-1}(P)$ または $x=Q^{-1}(Q)$ は P か Q が特定の値を取るような x の値を表す。これらは確率の値からその信頼区間 (confidence limit) を計算するのに使われる。

離散値をとる分布では、整数の値 k が生じる確率は $p(k)$ と表され、$\sum_k p(k)=1$ である。この場合の累積分布関数の下側確率 $P(k)$ は以下の式で表され、発生する事象が k 以下になる確率を与える。

$$P(k)=\sum_{i \leq k} p(i)$$

また累積分布関数の上側確率 $Q(k)$ は以下の式で表される。これは発生する事象が k よりも大きくなる確率を与える。

$$Q(k)=\sum_{i>k} p(i)$$

これら二つの関数の間にも $P(x)+Q(x)=1$ という関係がある。

また、分布関数の範囲が 1 以上 n 以下の時は $P(n)=1,\ Q(n)=0$ で、$P(1)=p(1),\ Q(1)=1-p(1)$ である。

19.2 正規分布

正規分布 (normal distribution、ガウス分布 Gaussian distribution) は自然科学の非常に幅広い分野で非常に多く用いられる統計分布のモデルである。平均が μ、分散が σ^2 の正規分布は $N(\mu, \sigma^2)$ とも表され、平均が 0、分散が 1 (したがって標準偏差も 1) の正規分布 ($N(0,1)$) を特に標準正規分布 (standard normal distribution) と呼ぶ。

`double gsl_ran_gaussian (const gsl_rng * r, const double sigma)` [Function]

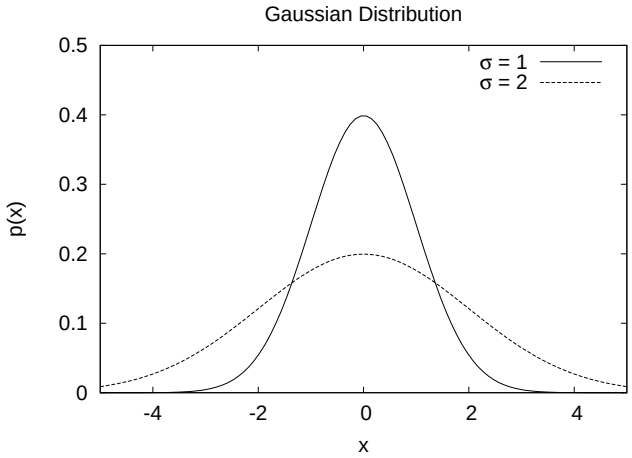
期待値 0、標準偏差 *sigma* の正規分布にしたがう乱数を返す。この分布の確率密度関数は以下の式で表される。

$$p(x)dx = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$$

x の範囲は $-\infty$ から $+\infty$ である。`gsl_ran_gaussian` が返す値 x に μ を加えて $z = \mu + x$ とすることで期待値 μ の正規分布乱数 z が得られる。この関数はボックス＝ミューラー (Box-Muller) 法を使い、乱数発生器 r を内部で二回呼び出す。

`double gsl_ran_gaussian_pdf (const double x, const double sigma)` [Function]

標準偏差 *sigma* の正規分布乱数の確率密度関数 $p(x)$ の値を返す。



`double gsl_ran_gaussian_ziggurat (const gsl_rng * r, const double sigma)` [Function]

`double gsl_ran_gaussian_ratio_method (const gsl_rng * r, const double sigma)` [Function]

正規分布乱数をそれぞれ、マルサグリア (George Marsaglia, マーセイリア) とツァン (Wai Wan Tsang) によるジッグラト、およびレバ (Joseph L. Leva) が改良したキンダーマン (A. J. Kinderman) とモナハン (John F. Monahan) の比による方法で生成する。ほとんどの場合はジッグラトの方法がもっとも高速である。

`double gsl_ran_ugaussian (`*const gsl_rng* `* r)` [Function]
`double gsl_ran_ugaussian_pdf (`*const double* `x)` [Function]
`double gsl_ran_ugaussian_ratio_method (`*const gsl_rng* `* r)` [Function]

標準正規分布 (unit Gaussian distribution) にしたがう値を返す。上記の関数で *sigma* を 1 にしたものと同じである。

`double gsl_cdf_gaussian_P (`*const double* `x`, *const double* `sigma)` [Function]
`double gsl_cdf_gaussian_Q (`*const double* `x`, *const double* `sigma)` [Function]
`double gsl_cdf_gaussian_Pinv (`*const double* `P`, *const double* `sigma)` [Function]
`double gsl_cdf_gaussian_Qinv (`*const double* `Q`, *const double* `sigma)` [Function]

標準偏差 *sigma* の累積分布関数 $P(x), Q(x)$ とその逆関数を返す。

`double gsl_cdf_ugaussian_P (`*const double* `x)` [Function]
`double gsl_cdf_ugaussian_Q (`*const double* `x)` [Function]
`double gsl_cdf_ugaussian_Pinv (`*const double* `P)` [Function]
`double gsl_cdf_ugaussian_Qinv (`*const double* `Q)` [Function]

標準正規分布の累積分布関数 $P(x), Q(x)$ とその逆関数を返す。

19.3 正規分布の裾

正規分布の PDF は指数関数であるため、確率変数 x が大きくなるにつれ PDF の値は非常に小さくなる。そういった領域に特化したアルゴリズムを使うことで数値計算の精度を向上することができる。

`double gsl_ran_gaussian_tail (const gsl_rng * r, double a, double sigma)` [Function]

標準偏差 *sigma* の正規分布の上側の裾での分布 (Gaussian tail distribution) にしたがう乱数を返す。返される値は正の値である下限 a よりも大きくなる。これはクヌースの第二巻第三版の p.139 および p. 586 (exercize 11) に載っている有名なマルサグリアの方形楔形裾 (rectangle-wedge-tail) アルゴリズムを使っている (G. Marsaglia, "Expressing a random variate in terms of uniform random variables", *Annals of Mathematical Statistics*, **32**, pp. 894–899, 1961)。

正規分布の裾分布は以下で与えられる。

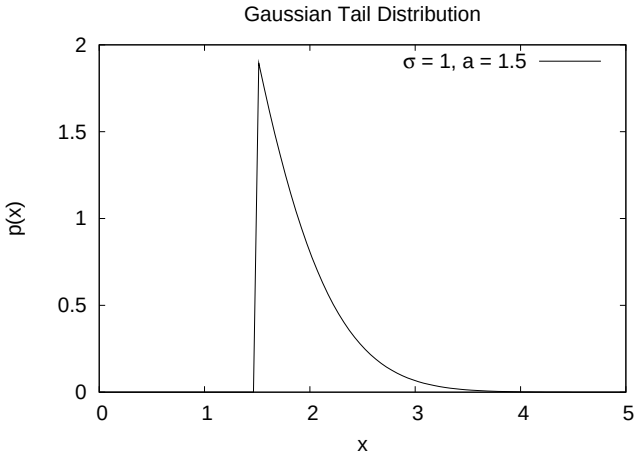
$$p(x)dx = \frac{1}{N(a;\sigma)\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$$

ここで $x > a$ で $N(a;\sigma)$ は正規化係数であり、以下で表される。$\mathrm{erfc}(x)$ はガウスの誤差関数の補関数 (63 ページ) である。

$$N(a;\sigma) = \frac{1}{2}\mathrm{erfc}\left(\frac{1}{\sqrt{2\sigma^2}}\right)$$

`double gsl_ran_gaussian_tail_pdf (const double x, const double a, const double sigma)` [Function]

上の式にしたがい、標準偏差 *sigma* の正規分布の裾分布で x における確率密度関数値 $p(x)$ を返す。下限値は a である。



`double gsl_ran_ugaussian_tail` (*const gsl_rng* * `r`, *const double* `a`) [Function]
`double gsl_ran_ugaussian_tail_pdf` (*const double* `x`, *const double* `a`) [Function]

sigma = 1、つまり標準正規分布にしたがう乱数および確率密度関数値を返す。

19.4 二変数の正規分布

`void gsl_ran_bivariate_gaussian` (*const gsl_rng * **r***, *double **sigma_x***, *double **sigma_y***, *double **rho***, *double * **x***, *double * **y***) [Function]

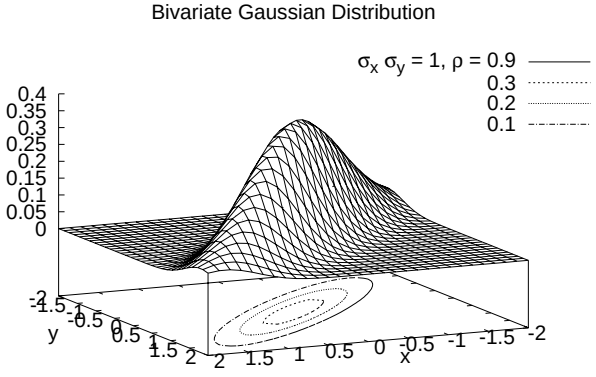
期待値 0、相関係数 *rho* で、相関を持つ二つの正規分布乱数 x と y を生成する (bivariate Gaussian distribution)。x と y の各方向での標準偏差をそれぞれ *sigma_x*, *sigma_y* で指定する。この乱数 x と y は $-\infty$ から $+\infty$ の範囲をとり、その確率分布は以下で与えられる。

$$p(x,y)dxdy = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho}}\exp\left(-\frac{x^2/\sigma_x^2 + y^2/\sigma_y^2 - 2\rho xy/(\sigma_x\sigma_y))}{2(1-\rho^2)}\right)dxdy$$

相関係数 *rho* の値は -1 以上 1 以下でなければならない。

`double gsl_ran_bivariate_gaussian_pdf` (*const double **x***, *const double **y***, *const double **sigma_x***, *const double **sigma_y***, *const double **rho***) [Function]

x および y 方向の標準偏差がそれぞれ *sigma_x*, *sigma_y* で、相関係数 *rho* の、上の式にしたがう二変数正規分布の確率密度関数 $p(x,y)$ の値を返す。



19.5　指数分布

ある事象について、平均発生間隔が分かっているとき、事象の発生する間隔が x になる確率のモデルである。たとえば、ある流星群を観測すると 1 時間に 100 個観測されることがわかっているとき (平均間隔 0.6 分)、流星を一つ観測してから次を観測するまでの時間間隔が x 分になる確率などのモデルとなる。

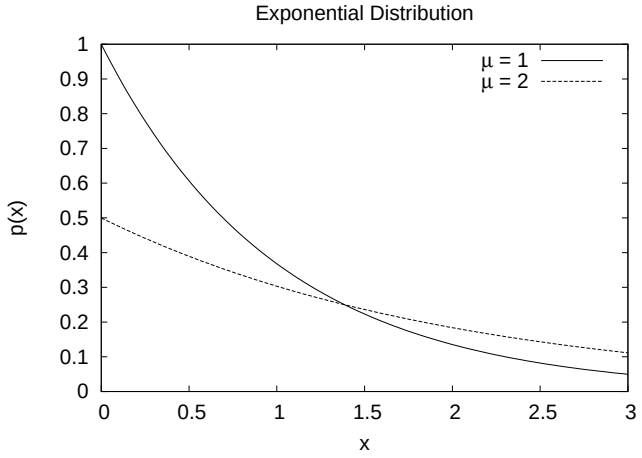
`double gsl_ran_exponential (`*const gsl_rng* `* r`, *const double* `mu)` [Function]

期待値 *mu* の指数分布 (exponential distribution) にしたがう乱数を返す。指数分布は $x \geq 0$ で以下で与えられる。

$$p(x)dx = \frac{1}{\mu}\exp(-x/\mu)dx$$

`double gsl_ran_exponential_pdf (`*const double* `x`, *const double* `mu)` [Function]

上の式にしたがって、x における期待値 *mu* の指数分布の確率密度関数の値を返す。



`double gsl_cdf_exponential_P (`*const double* `x`, *const double* `mu)` [Function]
`double gsl_cdf_exponential_Q (`*const double* `x`, *const double* `mu)` [Function]
`double gsl_cdf_exponential_Pinv (`*const double* `P`, *const double* `mu)` [Function]
`double gsl_cdf_exponential_Qinv (`*const double* `Q`, *const double* `mu)` [Function]

期待値 *mu* の指数分布について、累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.6 ラプラス分布

`double gsl_ran_laplace (const gsl_rng * r, const double a)` [Function]

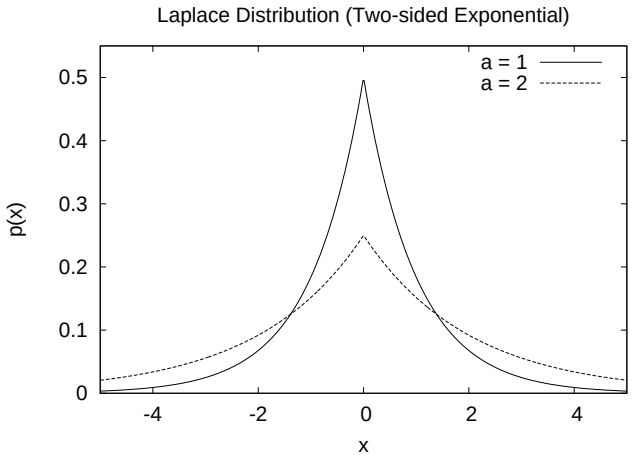
幅 a のラプラス分布 (Laplace distribution) にしたがう乱数を返す。ラプラス分布は以下で与えられる。

$$p(x)dx = \frac{1}{2a}\exp(-|x/a|)dx$$

x の範囲は $-\infty < x < \infty$ である。

`double gsl_ran_laplace_pdf (const double x, const double a)` [Function]

上の式にしたがって、x における幅 a のラプラス分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_laplace_P (const double x, const double a)` [Function]
`double gsl_cdf_laplace_Q (const double x, const double a)` [Function]
`double gsl_cdf_laplace_Pinv (const double P, const double a)` [Function]
`double gsl_cdf_laplace_Qinv (const double Q, const double a)` [Function]

幅 a のラプラス分布について、累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.7　指数べき分布

`double gsl_ran_exppow (const gsl_rng * r, const double a, const double b)`[Function]

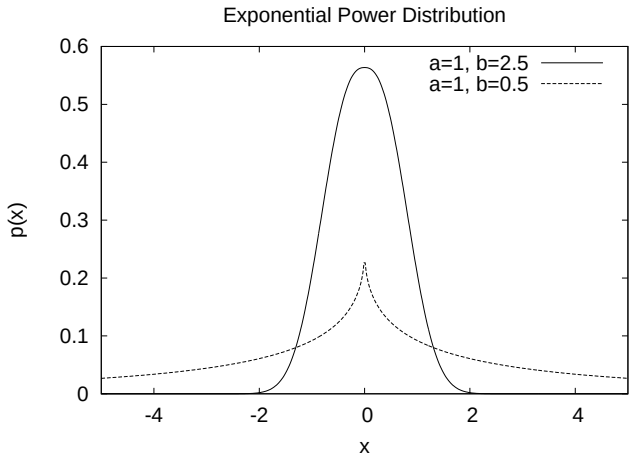
パラメータが a、b のべき指数分布 (exponential power distribution) にしたがう乱数を返す。べき指数分布は $x \geq 0$ について以下で与えられる。

$$p(x)dx = \frac{1}{2a\Gamma(1+1/b)} \exp(-|x/a|^b)dx$$

$b = 1$ のときはラプラス分布である。$b = 2$ のときは正規分布と同じだが、$a = \sqrt{2}\sigma$ である。

`double gsl_ran_exppow_pdf (const double x, const double a, const double b)`[Function]

パラメータが a、b のべき指数分布の確率密度関数 $p(x)$ の x における値を返す。



`double gsl_cdf_exppow_P (const double x, const double a, const double b)`[Function]
`double gsl_cdf_exppow_Q (const double x, const double a, const double b)`[Function]

パラメータ a、b の指数べき分布について、累積分布関数 $P(x), Q(x)$ の値を返す。

19.8 コーシー分布

互いに独立な二つの確率変数 X、Y がそれぞれ標準正規分布 $N(0,1)$ にしたがうとき、比 X/Y はコーシー分布 (Cauchy distribution) にしたがう。コーシー分布はローレンツ分布 (Lorentz distribution) とも呼ばれる。

`double gsl_ran_cauchy (`*const gsl_rng * `r`, const double* `a`)` [Function]

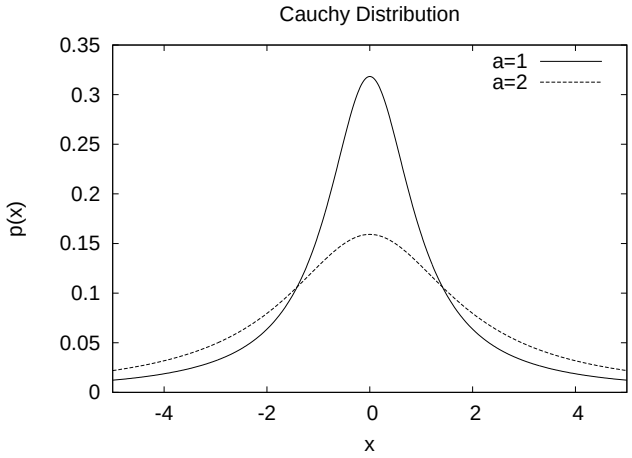
パラメータ a のコーシー分布 (Cauchy distribution) にしたがう乱数を返す。コーシー分布は以下で与えられる。

$$p(x)dx = \frac{1}{a\pi(1+(x/a)^2)}dx$$

x の範囲は $-\infty < x < \infty$ である。

`double gsl_ran_cauchy_pdf (`*const double* `x`, *const double* `a`)` [Function]

x におけるパラメータ a のコーシー分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_cauchy_P (`*const double* `x`, *const double* `a`)` [Function]
`double gsl_cdf_cauchy_Q (`*const double* `x`, *const double* `a`)` [Function]
`double gsl_cdf_cauchy_Pinv (`*const double* `P`, *const double* `a`)` [Function]
`double gsl_cdf_cauchy_Qinv (`*const double* `Q`, *const double* `a`)` [Function]

パラメータが a のコーシー分布について、累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.9　レイリー分布

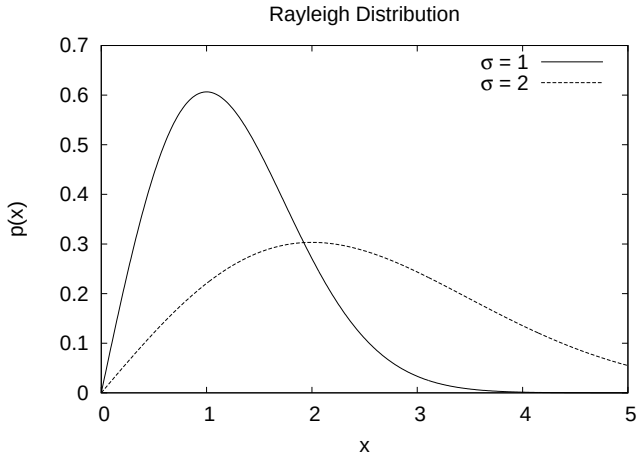
`double gsl_ran_rayleigh` (*const gsl_rng* * **`r`**, *const double* ***`sigma`***) [Function]

パラメータ *sigma* のレイリー分布 (Rayleigh distribution) にしたがう乱数を返す。レイリー分布は $x \geq 0$ について以下で与えられる。

$$p(x)dx = \frac{x}{\sigma^2} \exp(-x^2/(2\sigma^2))dx$$

`double gsl_ran_rayleigh_pdf` (*const double* **`x`**, *const double* ***`sigma`***) [Function]

上の式にしたがって、x におけるパラメータ *sigma* のレイリー分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_rayleigh_P` (*const double* **`x`**, *const double* ***`sigma`***) [Function]
`double gsl_cdf_rayleigh_Q` (*const double* **`x`**, *const double* ***`sigma`***) [Function]
`double gsl_cdf_rayleigh_Pinv` (*const double* **`P`**, *const double* ***`sigma`***) [Function]
`double gsl_cdf_rayleigh_Qinv` (*const double* **`Q`**, *const double* ***`sigma`***) [Function]

パラメータが a のレイリー分布について、累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.10 レイリーの裾分布

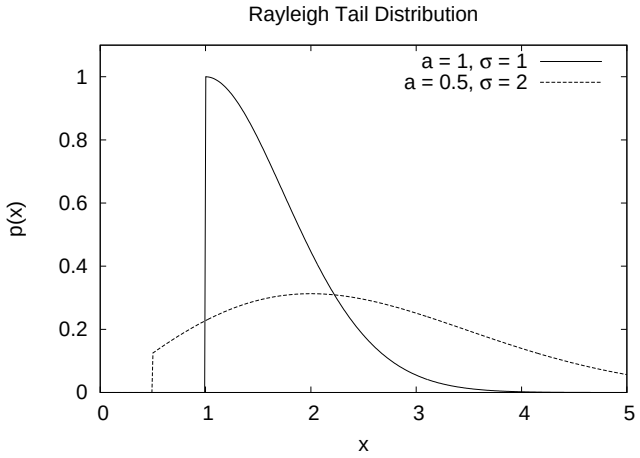
`double gsl_ran_rayleigh_tail` (*const gsl_rng* * `r`, *const double* `a`, *const double* `sigma`) [Function]

パラメータ *sigma* のレイリー分布の裾での分布 (Rayleigh tail distribution) にしたがう乱数を下限 a で返す。レイリーの裾分布は $x > a$ について以下で与えられる。

$$p(x)dx = \frac{x}{\sigma^2}\exp((a^2 - x^2)/(2\sigma^2))dx$$

`double gsl_ran_rayleigh_tail_pdf` (*const double* `x`, *const double* `a`, *const double* `sigma`) [Function]

上の式にしたがって、x におけるパラメータ *sigma* のレイリー分布の裾での分布の確率密度関数 $p(x)$ の値を下限 a で返す。



19.11　ランダウ分布

`double gsl_ran_landau (const gsl_rng * r)` [Function]

ランダウ分布 (Landau distribution) にしたがう乱数を返す。ランダウ分布は以下の複素積分 (complex integral) で定義される。

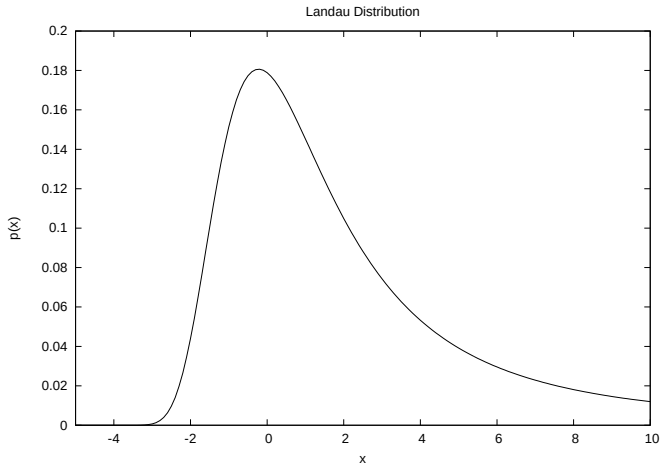
$$p(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} \exp(s\log(s) + xs)\, ds$$

数値計算においては、上式と同値な以下の式を使うほうが便利である。

$$p(x) = (1/\pi) \int_0^\infty \exp(-t\log(t) - xt)\sin(\pi t)\, dt$$

`double gsl_ran_landau_pdf (const double x)` [Function]

上の式にしたがって、x におけるランダウ分布の確率密度関数 $p(x)$ の値を返す。



19.12 レヴィの α 安定分布

`double gsl_ran_levy (const gsl_rng * r, const double c, const double alpha)` [Function]

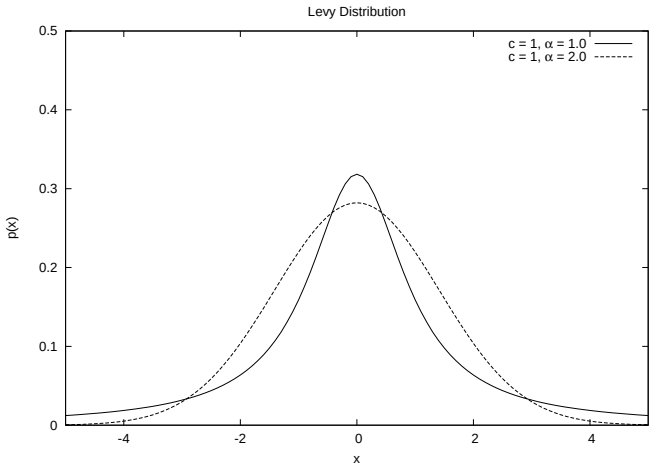
パラメータ c、*alpha* のレヴィの α 安定分布 (Levy alpha-stable distribution, 対称安定分布 Levy symmetric stable distribution とも) にしたがう乱数を返す。α 安定分布は以下のフーリエ変換で定義される。

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \exp(-itx - |ct|^\alpha) dt$$

$p(x)$ の解析的な解はないため、GSL ではこの分布に関しては確率密度関数 `pdf` を用意していない。の分布は $\alpha = 1, c = 1$ の時はコーシー分布 ($a = 1$, 251 ページ) になる。$\alpha = 2$ の時は $\sigma = \sqrt{2}c$ の正規分布である。$\alpha < 1$ ではこの分布の裾は非常に広くなる。

GSL で実装しているアルゴリズムでは、$0 < \alpha \leq 2$ でなければならない。

ここに示す確率分布関数のプロットは、PDF の定義式に含まれる被積分関数から簡易な計算で描いたものである。



19.13　レヴィの非対称 α 安定分布

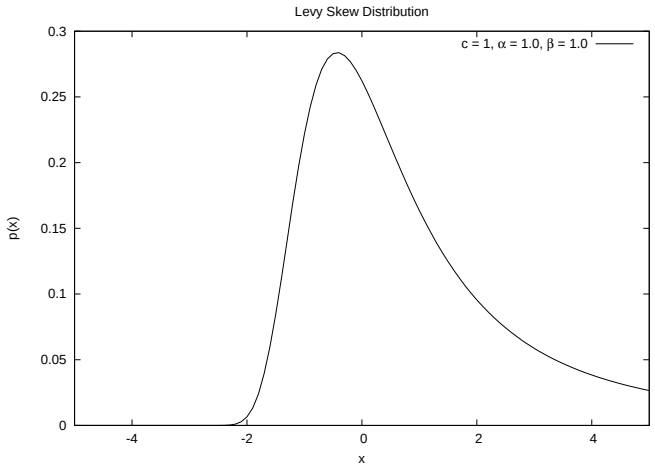
`double gsl_ran_levy_skew` (*const gsl_rng* * **`r`**, *const double* **`c`**, *const double* **`alpha`**, *const double* **`beta`**) [Function]

パラメータ c、指数 *alpha*、歪度係数 *beta* のレヴィの非対称 α 安定分布 (Levy skew alpha-stable distribution) にしたがう乱数を返す。歪度係数は $[-1, 1]$ の範囲内でなければならない。レヴィの非対称安定分布は以下のフーリエ変換で定義される。

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \exp(-itx - |ct|^\alpha (1 - i\beta \mathrm{sign}(t) \tan(\pi\alpha/2)))dt$$

$\alpha = 1$ のとき、$\tan(\pi\alpha/2)$ の項は $-(2/\pi)\log|t|$ で置き換えることができる。$p(x)$ の解析的な解はないため、GSL ではこの分布に関しては確率密度関数 `pdf` を用意していない。$\alpha = 2$ の時この分布は $\sigma = \sqrt{2}c$ の正規分布になり、非対称パラメータは意味を持たない。$\alpha < 1$ ではこの分布の裾は非常に広くなる。$\beta = 0$ のとき、前項の対称分布と同じである。

ここに示す確率分布関数のプロットは、PDF の定義式に含まれる被積分関数から簡易な計算で描いたものである。



レヴィの α 安定分布は、N 個の α 安定な値が分布 $p(c, \alpha, \beta)$ から与えられたとき、その和 $Y = X_1 + X_2 + \ldots + X_N$ も α 安定な分布 $p(N^{1/\alpha}c, \alpha, \beta)$ にしたがう、という性質がある。

19.14 ガンマ分布

`double gsl_ran_gamma (const gsl_rng * r, const double a, const double b)` [Function]

ガンマ分布 (gamma distribution) にしたがう乱数を返す。ガンマ分布は $x > 0$ について以下で定義される。

$$p(x)dx = \frac{1}{\Gamma(a)b^a} x^{a-1} e^{-x/b} dx$$

引数 a が整数の時はアーラン分布 (Erlang distribution) とも呼ばれる。

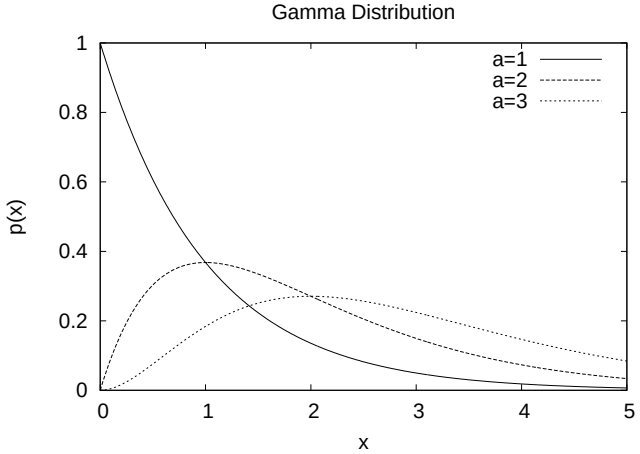
この分布の分散は、マルサグリア (George Marsaglia) とツァン (Wai Wan Tsang) の高速ガンマ法 (fast gamma method) で計算できる。高速ガンマ法を行うガンマ分布関数は以前の GSL では `gsl_ran_gamma_mt` として実装されていた。今のところ、それもそのまま使える。

`double gsl_ran_gamma_knuth (const double x, const double a, const double b)` [Function]

クヌース (1997) の方法を使ってガンマ分布にしたがう乱数を返す。

`double gsl_ran_gamma_pdf (const double x, const double a, const double b)` [Function]

与えられたパラメータ a、b および独立変数 x において、上述の確率密度関数 $p(x)$ の値を計算して返す。



`double gsl_cdf_gamma_P (const double x, const double a, const double b)` [Function]

`double gsl_cdf_gamma_Q (const double x, const double a, const double b)` [Function]

`double gsl_cdf_gamma_Pinv` (*const double* `P`, *const double* `a`, *const double* `b`)
[Function]
`double gsl_cdf_gamma_Qinv` (*const double* `Q`, *const double* `a`, *const double* `b`)
[Function]

引数が a と b のときのガンマ分布について、累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.15 一様分布

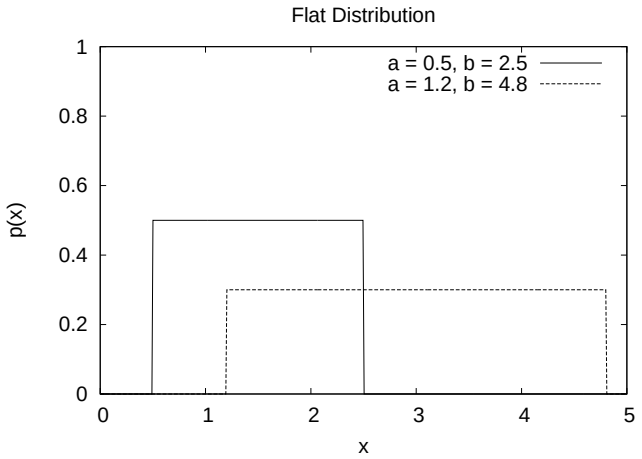
`double gsl_ran_flat` (*const gsl_rng* * `r`, *const double* `a`, *const double* `b`) [Function]

a から b の間に一様に分布する乱数を返す (flat distribution, uniform distribution) 。
この分布は $a \le x < b$ について以下で定義される。この範囲外では 0 である。

$$p(x)dx = \frac{1}{b-a}dx$$

`double gsl_ran_flat_pdf` (*double* `x`, *const double* `a`, *const double* `b`) [Function]

上の式にしたがう、a から b の間の一様分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_flat_P` (*const double* `x`, *const double* `a`, *const double* `b`) [Function]
`double gsl_cdf_flat_Q` (*const double* `x`, *const double* `a`, *const double* `b`) [Function]
`double gsl_cdf_flat_Pinv` (*const double* `P`, *const double* `a`, *const double* `b`) [Function]
`double gsl_cdf_flat_Qinv` (*const double* `Q`, *const double* `a`, *const double* `b`) [Function]

a から b の間の一様分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.16　対数正規分布

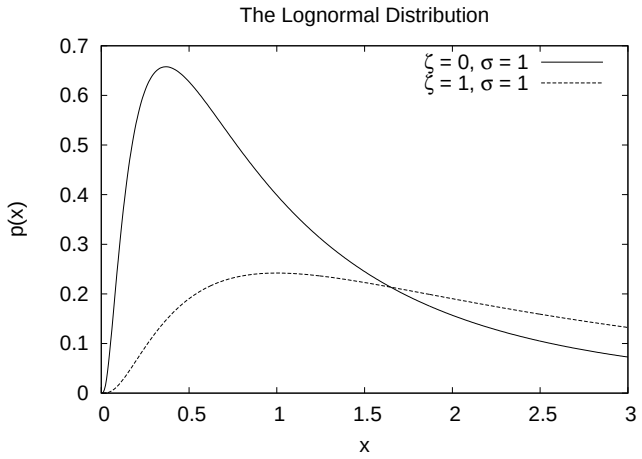
`double gsl_ran_lognormal` (*const gsl_rng* * **`r`**, *const double* **`zeta`**, *const double* ***`sigma`***) [Function]

対数正規分布 (lognormal distribution) にしたがう乱数を返す。この分布は $x > 0$ について以下で定義される。

$$p(x)dx = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp(-(\ln(x) - \zeta)^2/2\sigma^2)dx$$

`double gsl_ran_lognormal_pdf` (*const double* **`x`**, *const double* **`zeta`**, *const double* ***`sigma`***) [Function]

パラメータ値が *zeta* と *sigma* の対数正規分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_lognormal_P` (*const double* **`x`**, *const double* **`zeta`**, *const double* ***`sigma`***) [Function]
`double gsl_cdf_lognormal_Q` (*const double* **`x`**, *const double* **`zeta`**, *const double* ***`sigma`***) [Function]
`double gsl_cdf_lognormal_Pinv` (*const double* **`P`**, *const double* **`zeta`**, *const double* ***`sigma`***) [Function]
`double gsl_cdf_lognormal_Qinv` (*const double* **`Q`**, *const double* **`zeta`**, *const double* ***`sigma`***) [Function]

パラメータ値が *zeta* と *sigma* の対数正規分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.17 カイ二乗分布

カイ二乗分布 (chi-squared distribution) は統計学でよく見られる分布である。Y_i がそれぞれ独立な n 個の標準正規分布乱数であるとき、以下の和 X_i

$$X_i = \sum_i Y_i^2$$

は自由度 n のカイ二乗分布にしたがう。

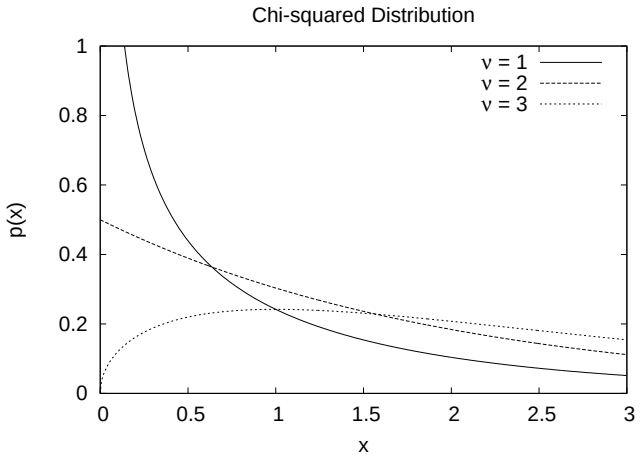
`double gsl_ran_chisq (const gsl_rng * r, const double nu)` [Function]

自由度 nu のカイ二乗分布にしたがう乱数を返す。この分布は $x \geq 0$ について以下で定義される。

$$p(x)dx = \frac{1}{2\Gamma(\nu/2)}(x/2)^{\nu/2-1}\exp(-x/2)dx$$

`double gsl_ran_chisq_pdf (const double x, const double nu)` [Function]

上の式にしたがう自由度 nu のカイ二乗分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_chisq_P (const double x, const double nu)` [Function]
`double gsl_cdf_chisq_Q (const double x, const double nu)` [Function]
`double gsl_cdf_chisq_Pinv (const double P, const double nu)` [Function]
`double gsl_cdf_chisq_Qinv (const double Q, const double nu)` [Function]

上の式にしたがう自由度 n のカイ二乗分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.18　F 分布

F 分布 (F-distribution) は統計学でよく見られる分布である。Y_1 と Y_2 がそれぞれ自由度 ν_1, ν_2 のカイ二乗分布にしたがう乱数であるとき、以下の比

$$X = \frac{Y_1/\nu_1}{Y_2/\nu_2}$$

は F 分布 $F(x;\nu_1,\nu_2)$ にしたがう。

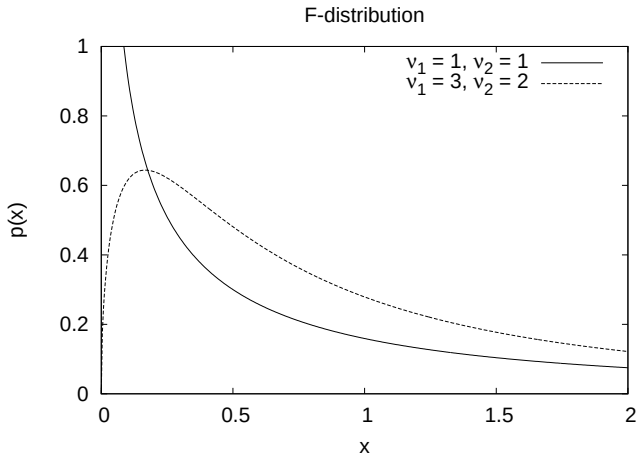
`double gsl_ran_fdist (`*const gsl_rng * `r`, const double `nu`*`)` [Function]

自由度 *nu1*、*nu2* の F 分布にしたがう乱数を返す。この分布は $x \geq 0$ について以下で定義される。

$$p(x)dx = \frac{\Gamma((\nu_1+\nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)}\nu_1^{\nu_1/2}\nu_2^{\nu_2/2}x^{\nu_1/2-1}(\nu_2+\nu_1 x)^{-\nu_1/2-\nu_2/2}$$

`double gsl_ran_fdist_pdf (`*const double `x`, const double `nu`*`)` [Function]

自由度 *nu1*、*nu2* の F 分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_fdist_P (`*const double `x`, const double `nu1`,const double `nu2`*`)` [Function]
`double gsl_cdf_fdist_Q (`*const double `x`, const double `nu1`,const double `nu2`*`)` [Function]
`double gsl_cdf_fdist_Pinv (`*const double `P`, const double `nu1`,const double `nu2`*`)` [Function]
`double gsl_cdf_fdist_Qinv (`*const double `Q`, const double `nu1`,const double `nu2`*`)` [Function]

自由度 *nu1*、*nu2* の F 分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.19 t 分布

t 分布 (t-distribution) は統計学でよく見られる分布である。Y_1 が正規分布に、Y_2 が自由度 ν のカイ二乗分布にしたがうとき、以下の比

$$X = \frac{Y_1}{\sqrt{Y_2/\nu}}$$

は自由度 ν の t 分布 $t(x;\nu)$ にしたがう。

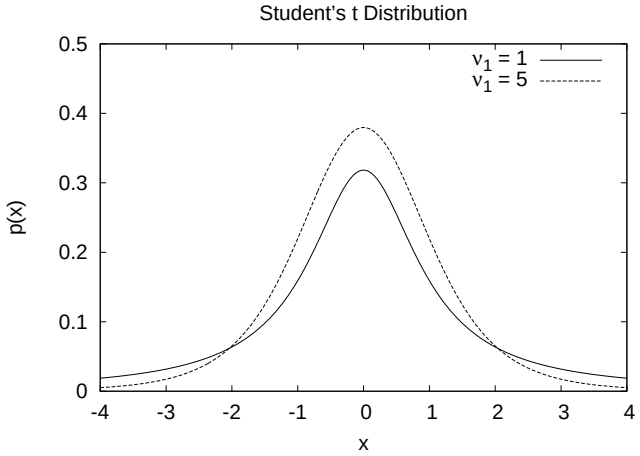
`double gsl_ran_tdist (const gsl_rng * r, const double nu)` [Function]

以下の t 分布にしたがう乱数を返す。$-\infty < x < \infty$ である。

$$p(x)dx = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}(1+x^2/\nu)^{-(\nu+1)/2}dx$$

`double gsl_ran_tdist_pdf (const double x, const double nu)` [Function]

上の式にしたがう、自由度 nu の t 分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_tdist_P (const double x, const double nu)` [Function]
`double gsl_cdf_tdist_Q (const double x, const double nu)` [Function]
`double gsl_cdf_tdist_Pinv (const double P, const double nu)` [Function]
`double gsl_cdf_tdist_Qinv (const double Q, const double nu)` [Function]

自由度 nu の t 分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.20　ベータ分布

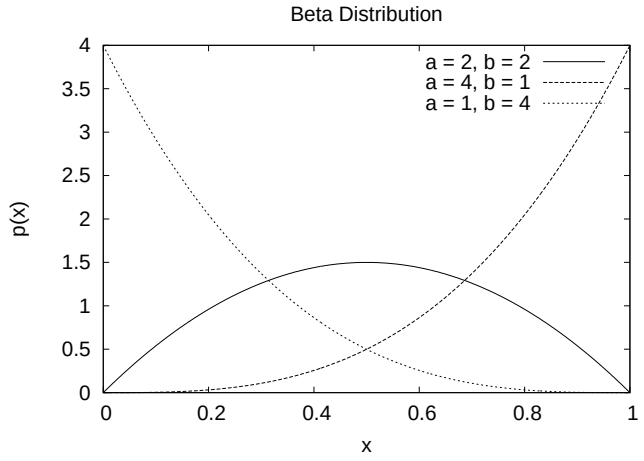
`double gsl_ran_beta` (*const gsl_rng* * `r`, *const double* `a`, *const double* `b`) [Function]

ベータ分布 (beta distribution) にしたがう乱数を返す。この分布は $0 \leq x \leq 1$ について以下で定義される。

$$p(x)dx = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}dx$$

`double gsl_ran_beta_pdf` (*const double* `x`, *const double* `a`, *const double* `b`)[Function]

パラメータ値が a と b のベータ分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_beta_P` (*const double* `x`, *const double* `a`, *const double* `b`) [Function]
`double gsl_cdf_beta_Q` (*const double* `x`, *const double* `a`, *const double* `b`) [Function]
`double gsl_cdf_beta_Pinv` (*const double* `P`, *const double* `a`, *const double* `b`) [Function]
`double gsl_cdf_beta_Qinv` (*const double* `Q`, *const double* `a`, *const double* `b`) [Function]

パラメータ値が a と b のベータ分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.21 ロジスティック分布

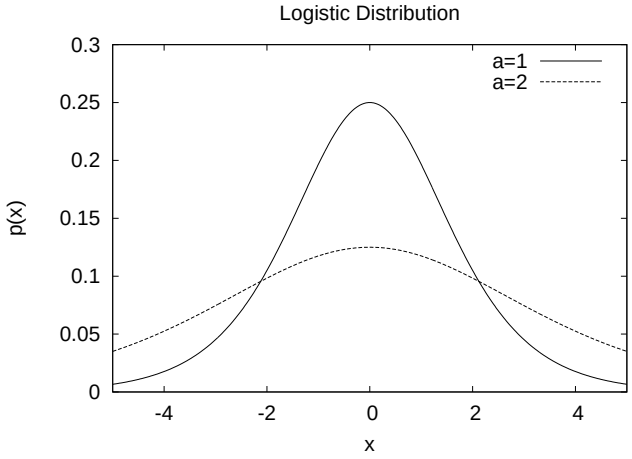
`double gsl_ran_logistic (const gsl_rng * r, const double a)` [Function]

パラメータが a のロジスティック分布 (logistic distribution) にしたがう乱数を返す。
この分布は $-\infty < x < +\infty$ について以下で定義される。

$$p(x)dx = \frac{\exp(-x/a)}{a(1+\exp(-x/a))^2}dx$$

`double gsl_ran_logistic_pdf (const double x, const double a)` [Function]

パラメータが a のロジスティック分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_logistic_P (const double x, const double a)` [Function]
`double gsl_cdf_logistic_Q (const double x, const double a)` [Function]
`double gsl_cdf_logistic_Pinv (const double P, const double a)` [Function]
`double gsl_cdf_logistic_Qinv (const double Q, const double a)` [Function]

パラメータが a のロジスティック分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.22　パレート分布

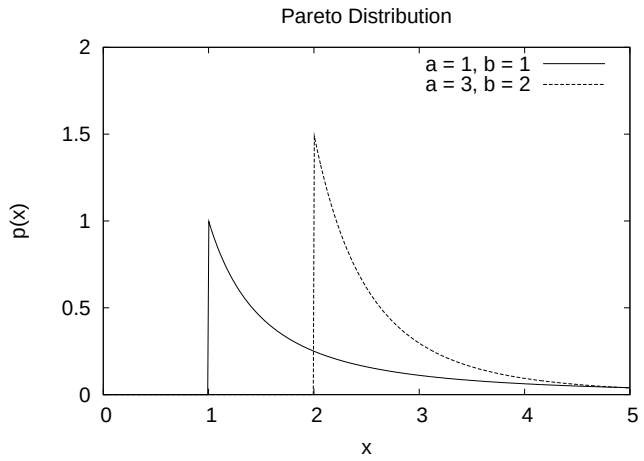
`double gsl_ran_pareto` (*const gsl_rng* * `r`, *double* `a`, *const double* `b`) [Function]

a 次のパレート分布にしたがう乱数を返す。この分布は $x \geq b$ について以下で定義される。

$$p(x)dx = (a/b)/(x/b)^{a+1}dx$$

`double gsl_ran_pareto_pdf` (*const double* `x`, *const double* `a`, *const double* `b`) [Function]

パラメータが a、b のパレート分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_pareto_P` (*const double* `x`, *const double* `a`, *const double* `b`)[Function]
`double gsl_cdf_pareto_Q` (*const double* `x`, *const double* `a`, *const double* `b`)[Function]
`double gsl_cdf_pareto_Pinv` (*const double* `P`, *const double* `a`, *const double* `b`) [Function]
`double gsl_cdf_pareto_Qinv` (*const double* `Q`, *const double* `a`, *const double* `b`) [Function]

パラメータが a、b のパレート分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.23 球面分布

球面分布 (spherical vector distribution) は球面上にランダムに分布するベクトルを生成する。これはシミュレーションでランダム・ウォークを行うようなときにランダムな方向として使うことができる。

`void gsl_ran_dir_2d (const gsl_rng * r, double * x, double * y)` [Function]
`void gsl_ran_dir_2d_trig_method (const gsl_rng * r, double * x, double * y)` [Function]

二次元のランダムなベクトル $v = (x, y)$ を返す。ベクトルは $|v|^2 = x^2 + y^2$ となるように正規化される。この乱数は 0 から 2π の間の一様乱数を発生して、x はその値の正弦値、y はその値の余弦値をとることで生成される。昔は三角関数を二回呼び出すのは計算コストが大きかったが、最近の計算機アーキテクチャでは、直接三角関数を計算するのがもっとも速い方法であることもある。Pentium はその一つである (しかし Sun の SparcStation はそうではない)。三角関数の呼び出しを避けるには、x と y を単位円の内側にとり (一様乱数の発生器で $0 \le x < 1, 0 \le y < 1$ の正方形の内側の点を生成し、それが単位円の外側だったら捨てて取り直せばよい)、$\sqrt{x^2 + y^2}$ で割ればよい。ノイマン (John von Neumann) によるもっと優れた方法 (Knuth 1997, p. 140, Excersize 23) を使えば、三角関数も平方根も計算しなくてすむ。この方法では単位円内に u, v の二つの乱数を生成し、$x = (u^2 - v^2)/(u^2 + v^2), y = 2uv/(u^2 + v^2)$ とする。

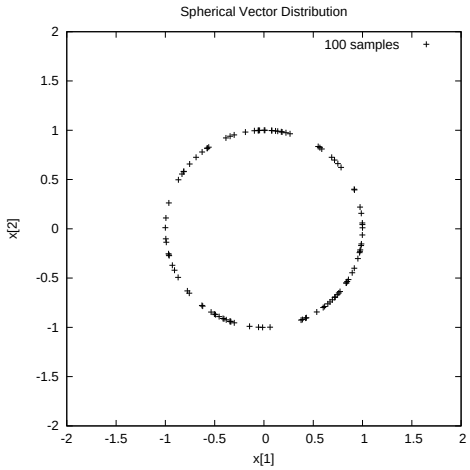
`void gsl_ran_dir_3d (const gsl_rng * r, double * x, double * y, double * z)` [Function]

三次元の球面にランダムに分布するベクトル $v = (x, y, z)$ を返す。返されるベクトルは $|v|^2 = x^2 + y^2 + z^2 = 1$ となるように正規化される。GSL で実装されているアルゴリズムはノップ (Robert E. Knop, "Algorithm 381 Random vectors uniform in solid angle [G5]", *Communications of the ACM*, **13**(5), p. 326, 1970) によるもので、これはクヌース第二巻第三版 p. 136 でも説明されている。この方法は、この分布はどの座標軸に添って投影しても一様分布になるということを利用している (これは三次元でのみ成り立つ)。

`void gsl_ran_dir_nd (const gsl_rng * r, size_t n, double * x)` [Function]

n 次元の球面にランダムに分布するベクトル $v = (x_1, x_2, \ldots, x_n)$ を返す。返されるベクトルは $|v|^2 = x_1^2 + x_2^2 + \ldots + x_n^2 = 1$ となるように正規化される。この関数では、多変量正規分布は球面対称であることを利用している。返されるベクトルの各要素は正規分布にしたがって生成され、それから正規化される。この方法はブラウン (G. W. Brown, "Monte Carlo Methods", in *Modern Mathematics for the Engineer*, ed. Beckenbach, McGraw-Hill, 1956) によるもので、クヌース第二巻第三版 pp. 135-136 で説明されている。

以下の図は、`gsl_ran_dir_nd` で n= 2 として、球面分布にしたがう乱数を発生させ、プロットしたものである。


Spherical Vector Distribution
100 samples
x[2]
x[1]
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

19.24 ワイブル分布

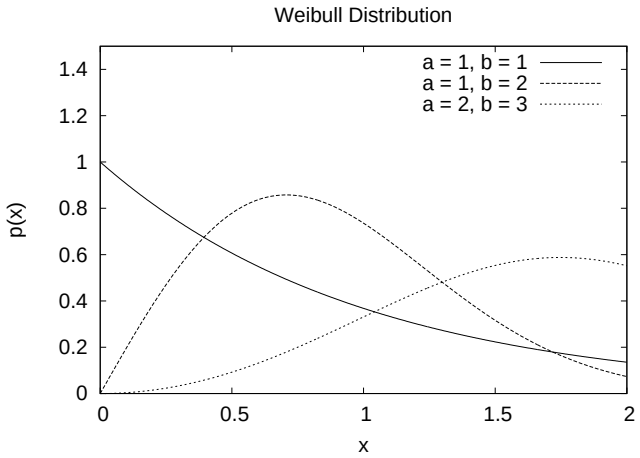
`double gsl_ran_weibull (const gsl_rng * r, const double a, const double b)` [Function]

パラメータが a と b のワイブル分布 (Weibull distribution) にしたがう乱数を返す。この分布は $x \geq 0$ について以下で定義される。

$$p(x)dx = \frac{b}{a^b} x^{b-1} \exp(-(x/a)^b) dx$$

`double gsl_ran_weibull_pdf (const double x, const double a, const double b)` [Function]

パラメータが a と b のワイブル分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_weibull_P (const double x, const double a, const double b)` [Function]
`double gsl_cdf_weibull_Q (const double x, const double a, const double b)` [Function]
`double gsl_cdf_weibull_Pinv (const double P, const double a, const double b)` [Function]
`double gsl_cdf_weibull_Qinv (const double Q, const double a, const double b)` [Function]

パラメータが a と b のワイブル分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.25　第一種極値分布 (グンベル分布)

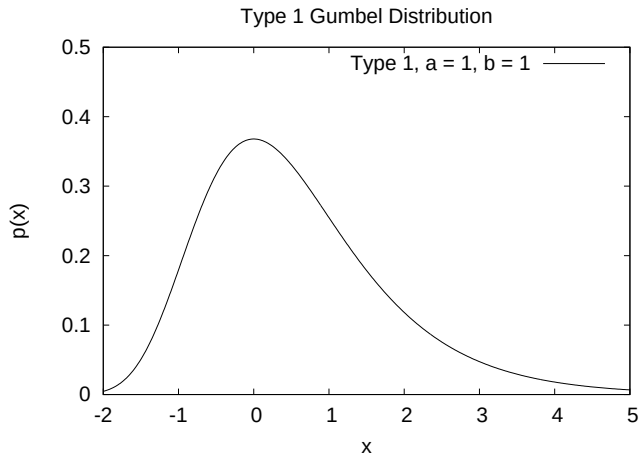
`double gsl_ran_gumbel1 (const gsl_rng * r, const double a, const double b)` [Function]

パラメータが a と b の第一種極値分布 (Type-1 extreme value distribution, グンベル分布 Gumbel distribution とも) にしたがう乱数を返す。この分布は $-\infty < x < +\infty$ について以下で定義される。

$$p(x)dx = ab\exp(-(b\exp(-ax) + ax))dx$$

`double gsl_ran_gumbel1_pdf (const double x, const double a, double b)` [Function]

パラメータが a と b の第一種極値分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_gumbel1_P (const double x, const double a, const double b)` [Function]
`double gsl_cdf_gumbel1_Q (const double x, const double a, const double b)` [Function]
`double gsl_cdf_gumbel1_Pinv (const double P, const double a, const double b)` [Function]
`double gsl_cdf_gumbel1_Qinv (const double Q, const double a, const double b)` [Function]

パラメータが a と b の第一種極値分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.26 第二種極値分布 (フレシェ分布)

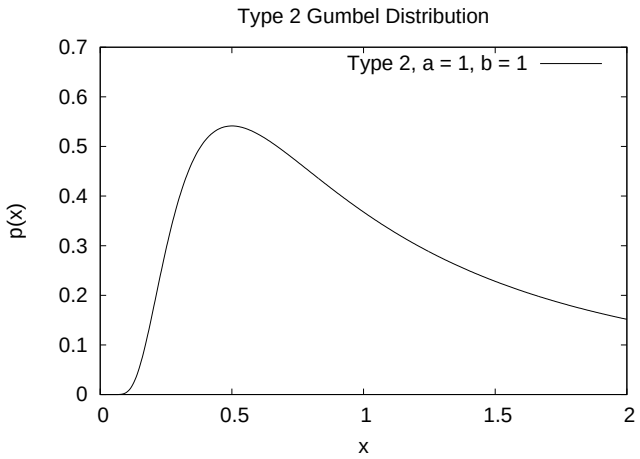
`double gsl_ran_gumbel2 (const gsl_rng * r, const double a, const double b)` [Function]

パラメータが a と b の第二種極値分布 (Type 2 extreme value distribution, フレシェ分布 Fréchet distribution とも。原著では type 2 Gumbel distribution となっている) にしたがう乱数を返す。この分布は $-\infty < x < +\infty$ について以下で定義される。

$$p(x)dx = abx^{-a-1}\exp(-bx^{-a})dx$$

`double gsl_ran_gumbel2_pdf (const double x, const double a, const double b)` [Function]

パラメータが a と b の第二種極値分布の確率密度関数 $p(x)$ の値を返す。



`double gsl_cdf_gumbel2_P (const double x, const double a, const double b)` [Function]
`double gsl_cdf_gumbel2_Q (const double x, const double a, const double b)` [Function]
`double gsl_cdf_gumbel2_Pinv (const double P, const double a, const double b)` [Function]
`double gsl_cdf_gumbel2_Qinv (const double Q, const double a, const double b)` [Function]

パラメータが a と b の第二種極値分布の累積分布関数 $P(x), Q(x)$ とその逆関数の値を返す。

19.27　ディリクレ分布

`void gsl_ran_dirichlet` (*const gsl_rng* * `r`, *const size_t* `K`, *const double* `alpha[]`, *double* `theta[]`) [Function]

$K-1$ 次のディリクレ分布 (Dirichlet distribution、多変量ベータ分布 multivariate beta distribution とも) にしたがう K 個の 乱数を生成し *theta[]* に入れて返す。この分布は $\theta_i \geq 0$ および $\alpha_i \geq 0$ で以下の式で定義される。

$$p(\theta_1, \ldots, \theta_K) d\theta_1 \cdots \theta_K = \frac{1}{Z} \prod_{i=1}^{K} \theta_i^{\alpha_i - 1} \delta(1 - \sum_{i=1}^{K} \theta_i) d\theta_1 \cdots \theta_K$$

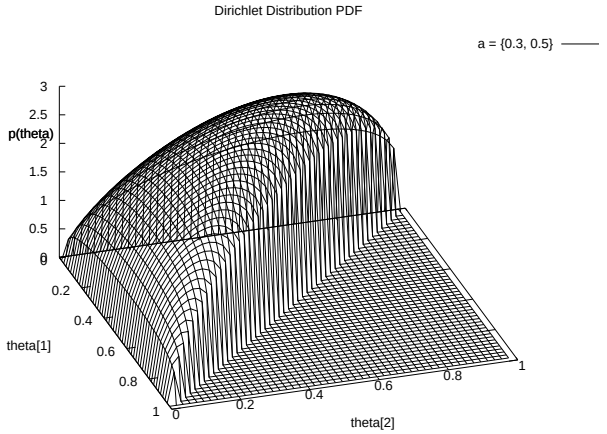
$\sum \theta_i = 1$ である。正規化係数 Z は以下で表される。

$$Z = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{K} \alpha_i)}$$

ディリクレ分布にしたがう乱数は、パラメータ $a = \alpha_i,\ b = 1$ のガンマ分布 (257 ページ) にしたがう K 個の乱数から生成される。詳細はロウとケルトン (Averill M. Law, W. David Kelton, *Simulation Modeling and Analysis*, 3rd ed., ISBN 0070592926, 2000) を参照のこと。

`double gsl_ran_dirichlet_pdf` (*const size_t* `K`, *const double* `alpha[]`, *const double* `theta[]`) [Function]

パラメータが *alpha[K]* の *theta[K]* でのディリクレ分布の確率密度関数 $p(\theta_1, \ldots, \theta_K)$ の値を返す。



`double gsl_ran_dirichlet_lnpdf` (*const size_t* `K`, *const double* `alpha[]`, *const double* `theta[]`) [Function]

パラメータが *alpha[K]* のディリクレ分布の確率密度関数 $p(\theta_1, \ldots, \theta_K)$ の対数値を返す。

19.28 離散分布について

K 個の離散事象がそれぞれ異なった確率 $P[k]$ で生じるとするとき、その分布にしたがう乱数値 k を生成することを考える。

もっとも分かりやすい方法は、あらかじめ $K+1$ 個の要素を持つ累積分布 (cumulative probability) を計算し、リストにしておくことである。

$$\begin{aligned} C[0] &= 0 \\ C[k+1] &= C[k] + P[k] \end{aligned}$$

この場合 $C[K] = 1$ になる。ここで 0 から 1 の間で偏りなくある値 (uniform deviate) u をとり、$C[k] \leq u < C[k+1]$ となる k を探す。一般にはこの探索には $\log K$ の オーダーのステップ数が必要になるが、探索の初期値を $\lfloor uK \rfloor$ にとると、多くの場合、より速く探索できる。

しかしもっと速い方法がある。あらかじめ確率のリストを作っておくが、それを簡単にひける表の形にしておく方法である。これにより乱数発生器を呼び出してランダムな離散事象を得るのがより高速に行える。マルサグリアが考案したその方法 (G. Marsaglia, "Generating discrete random variables in a computer", *Communications of the ACM*, **6**(1), pp. 37–38, 1963) は非常に優秀である。この論文は短いがうまくまとまっており、アルゴリズム設計のよい手本として一読をお勧めする。しかし K が大きくなると、マルサグリアによる方法で生成される表は非常に大きくなってしまう。

ウォーカー (Alastair J. Walker) により、さらに優れた方法 ("An efficient method for generating discrete random variables with general distributions", *ACM Transactions on Mathematical Software*, **3**(3), pp. 253–256 (1977), Knuth 1997, pp. 120-121, 139 も参照) が考案されている。この方法は浮動小数点実数と整数の二つの表を使うが、どちらの表も大きさは K である。元の確率分布から一度表を作ってしまえば、K が大きくなっても乱数生成にかかる計算時間のオーダーは $O(1)$ に抑えられる。ウォーカーによる表生成の計算量は $O(K^2)$ だが、その計算量を要することは実際にはなく、GSL での実装では $O(K)$ である。一般的には、大きな計算量で表を作れば乱数の生成も速くなるが、あまり手間をかけすぎても、そこまで効果は上がらなくなってくる。クヌースによると、最適な表はどうなるかという問題は、K が大きくなると組み合わせ爆発的に難しくなる。

上に述べた方法は、二項分布などの、この節のいくつかの乱数生成法の高速化に効果があるが、たとえば、ポアソン分布の場合などはとりうる値が K 個しかない有限集合であるため、修正が必要なこともある。

`gsl_ran_discrete_t * gsl_ran_discrete_preproc` (*size_t K*, *const double * P*) [Function]

> 離散乱数の生成器で使うための表を保持する構造体へのポインタを返す。配列 *P[]* が離散事象を保持する。配列要素は正でなければならないが、その合計が 1 である必要はなく (したがってこれを、一般的な「重み」としてとらえることもできる)、この関数内で正規化される。この関数の返り値は以下の関数 `gsl_ran_discrete` への引数に使う。

`size_t gsl_ran_discrete` (*const gsl_rng* * **r**, *const gsl_ran_discrete_t* * **g**)[Function]

上述の表作成関数を呼んだ後、離散値をとる乱数を得るのにこの関数を使う。

`double gsl_ran_discrete_pdf` (*size_t* **k**, *const gsl_ran_discrete_t* * **g**) [Function]

観測された事象 k が与えられたとき、その生じる確率 $P[k]$ を返す。$P[k]$ は表に保持されてはいないため、表から再計算される。この計算にかかる計算量のオーダーは $O(K)$ なので、K が大きいときは表を作るのに使った $P[k]$ を保持しておくようにするとよい。

`void gsl_ran_discrete_free` (*gsl_ran_discrete_t* * **g**) [Function]

g が指す表を保持するメモリーを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

19.29 ポアソン分布

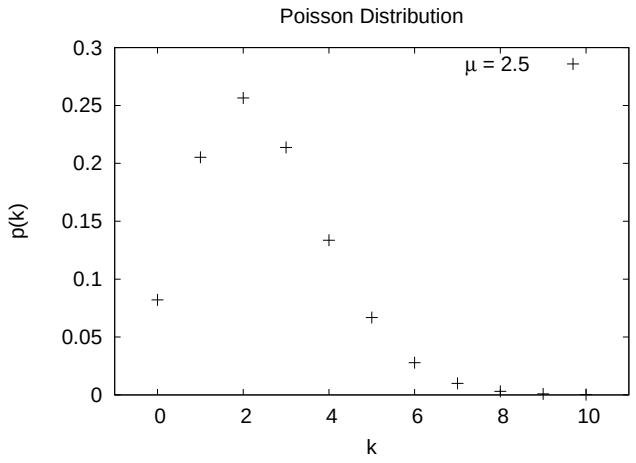
`unsigned int gsl_ran_poisson` (*const gsl_rng * r*, *double mu*) [Function]

期待値 *mu* のポアソン分布 (Poisson distribution) にしたがう整数の乱数を生成する。
この分布は $k \geq 0$ で以下の式で定義される。

$$p(k) = \frac{\mu^k}{k!} \exp(-\mu)$$

`double gsl_ran_poisson_pdf` (*const unsigned int k*, *const double mu*) [Function]

上の式の期待値 *mu* のポアソン分布にしたがう事象 k が生じる確率 $p(k)$ の値を返す。



`double gsl_cdf_poisson_P` (*const unsigned int k*, *const double mu*) [Function]
`double gsl_cdf_poisson_Q` (*const unsigned int k*, *const double mu*) [Function]

上の式にしたがう期待値 *mu* のポアソン分布の累積分布関数 $P(k), Q(k)$ の値を返す。

19.30　ベルヌーイ分布

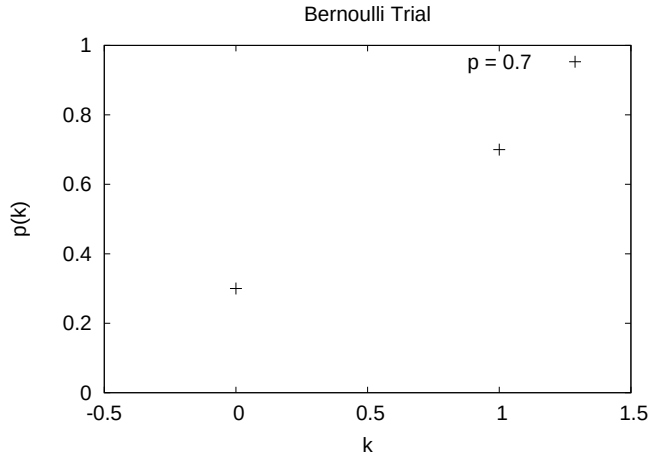
`unsigned int gsl_ran_bernoulli` (*const gsl_rng* * **r**, *double* **p**)　[Function]

確率 p のベルヌーイ試行 (Bernoulli trial) の結果にしたがって 0 または 1 を返す。この分布は以下で定義される。

$$\begin{aligned} p(0) &= 1 - p \\ p(1) &= p \end{aligned}$$

`double gsl_ran_bernoulli_pdf` (*const unsigned int* **k**, *double* **p**)　[Function]

上の式の確率 p のベルヌーイ分布にしたがう事象 k（0 または 1）が生じる確率 $p(k)$ の値を返す。



19.31 二項分布

`unsigned int gsl_ran_binomial` (*const gsl_rng * r*, *double p*, *unsigned int n*) [Function]

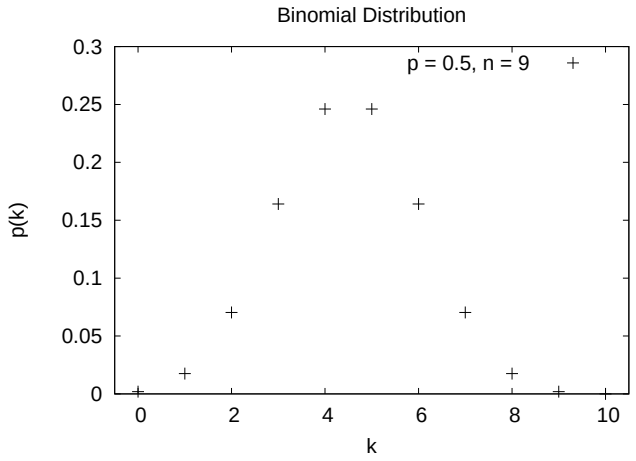
二項分布 (binomial distribution)、つまり n 回の独立した試行が確率 p で成功するときのその回数の分布にしたがう整数の乱数を生成する。この分布は $0 \le k \le n$ で以下の式で定義される。

$$p(k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

要するに、表と裏の出る確率が違う (それぞれ p と $1-p$) 出来の悪いコインでコイントスを n 回やったら何回表が出るか、を乱数でシミュレートして返す。

`double gsl_ran_binomial_pdf` (*const unsigned int k*, *const double p*, *const unsigned int n*) [Function]

パラメータが p および n の二項分布で事象 k が生じる確率 $p(k)$ の値を返す。



`double gsl_cdf_binomial_P` (*const unsigned int k*, *const double p*, *const unsigned int n*) [Function]
`double gsl_cdf_binomial_Q` (*const unsigned int k*, *const double p*, *const unsigned int n*) [Function]

パラメータが p および n で与えられる二項分布の累積分布関数 $P(k), Q(k)$ の値を返す。

19.32　多項分布

`void gsl_ran_multinomial` (*const gsl_rng * `r`*, *const size_t `K`*, *const unsigned int `N`*, *const double `p[]`*, *unsigned int `n[]`*) [Function]

多項分布 (multinomial distribution)、つまり確率分布 *p[]* にしたがった N 回の試行による多項分布からランダムな標本 *n[]* を得る (*p[]*、*n[]* とも大きさ K の配列)。*n[]* の分布関数は以下のようになる。

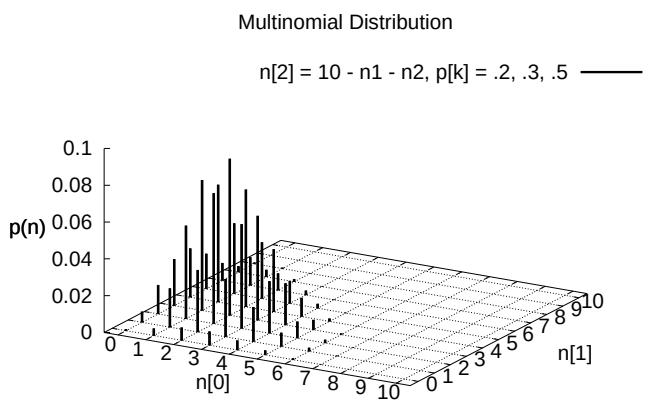
$$P(n_1, n_2, \ldots, n_K) = \frac{N!}{n_1! n_2! \cdots n_K!} p_1^{n_1} p_2^{n_2} \cdots p_K^{n_K}$$

ここで $(n_1, n2, \ldots, n_K)$ は非負の整数で $\sum_{k=1}^{K} n_k = N$ であり、$(p_1, p_2, \ldots p_K)$ は確率分布で $\sum p_i = 1$ である。*p[]* が、和が 1 になるように正規化されずに渡されたときは、それは重みとして扱われ、適切に正規化される。乱数は条件付き二項分布により発生する (詳しくは Charles S. Davis, "The computer generation of multinomial random variates", *Computational Statistics & Data Analysis*, **16**(2), pp. 205–217, 1993 参照)。

要するに、二項分布の場合の出来の悪いコインを、やはり出来の悪い多面体のサイコロに置き換えたシミュレーションを行う (K が面の数になる)。出来の悪いサイコロの各目の出る確率は等しいとは限らず、それぞれいくらなのかを *p[]* で指定する。これに基づいてサイコロを N 回振り、各目が N 回のうち何回出たかを *n[]* に入れて返す。

`double gsl_ran_multinomial_pdf` (*const size_t `K`*, *const double `p[]`*, *const unsigned int `n[]`*) [Function]

確率 *p[]* を持つ多項分布にしたがって事象 *n[]* が生じる確率 $P(n_1, n_2, \ldots, n_K)$ を計算する。$K = 3$, $N = \sum n_i = 10$、$p_i = 0.2, 0.3, 0.5$ の場合のプロットを以下に示す (三面体サイコロは現実にはあり得ないが)。n_1 と n_2 が決まれば n_3 は $n_3 = N - n_1 - n_2$ で決まるため、n_1 と n_2 に対して $p(n_1, n_2, n_3)$ をプロットしている。



`double gsl_ran_multinomial_lnpdf` (*const unsigned int* `K`, *const double* `p[]`, *const unsigned int* `n[]`) [Function]

確率 *p[K]* を持つ多項分布にしたがって事象 *n[K]* が生じる確率 $P(n_1, n_2, \ldots, n_K)$ の対数値を計算する。

19.33　負の二項分布

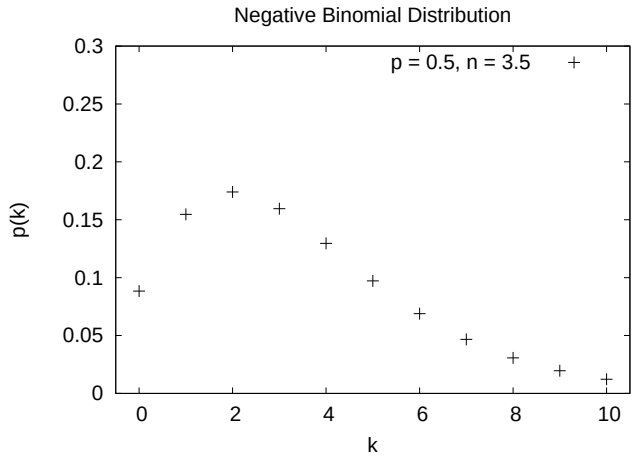
`unsigned int gsl_ran_negative_binomial` (*const gsl_rng* * **`r`**, *double* **`p`**, *double* **`n`**) [Function]

負の二項分布 (netagive binomial distribution)、つまり試行が成功する確率が p であるときに、試行が n 回成功する前に生じる失敗の回数の分布にしたがう整数の乱数を返す。この分布は以下で定義される。n は整数でなくてもよい。

$$p(k) = \frac{\Gamma(n+k)}{\Gamma(k+1)\Gamma(n)} p^n (1-p)^k$$

`double gsl_ran_negative_binomial_pdf` (*const unsigned int* **`k`**, *const double* **`p`**, *double* **`n`**) [Function]

パラメータが p と n で与えられる負の二項分布にしたがう事象 k が生じる確率 $p(k)$ の値を返す。



`double gsl_cdf_negative_binomial_P` (*const unsigned int* **`k`**, *const double* **`p`**, *const double* **`n`**) [Function]
`double gsl_cdf_negative_binomial_Q` (*const unsigned int* **`k`**, *const double* **`p`**, *const double* **`n`**) [Function]

パラメータが p と n で与えられる負の二項分布の累積分布関数 $P(k), Q(k)$ の値を返す。

19.34 パスカル分布

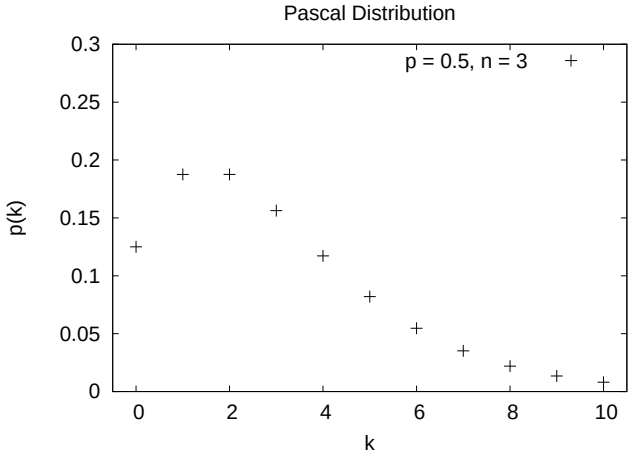
`unsigned int gsl_ran_pascal` (*const gsl_rng * r*, *double p*, *unsigned int n*) [Function]

パスカル分布 (Pascal distribution) にしたがう整数の乱数を返す。パスカル分布は n が整数である負の二項分布と同じであり、$k \geq 0$ について以下で定義される。

$$p(k) = \frac{(n+k-1)!}{k!(n-1)!} p^n (1-p)^k$$

`double gsl_ran_pascal_pdf` (*const unsigned int k*, *const double p*, *unsigned int n*) [Function]

パラメータが p と n で与えられるパスカル分布にしたがう事象 k が生じる確率 $p(k)$ の値を返す。



`double gsl_cdf_pascal_P` (*const unsigned int k*, *const double p*, *const unsigned int n*) [Function]
`double gsl_cdf_pascal_Q` (*const unsigned int k*, *const double p*, *const unsigned int n*) [Function]

パラメータが p と n で与えられるパスカル分布の累積分布関数 $P(k), Q(k)$ の値を返す。

19.35　幾何分布

`unsigned int gsl_ran_geometric` (*const gsl_rng * r*, *const double p*)[Function]

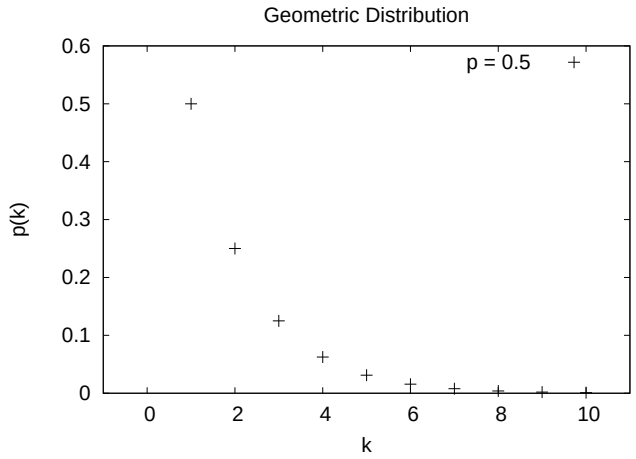
幾何分布 (geometric distribution)、つまり試行が成功する確率が p であるときに、最初に成功するまでに要する試行の回数の分布にしたがう整数の乱数を返す。この分布は $k \geq 1$ で以下で定義される。

$$p(k) = p(1-p)^{k-1}$$

この定義では、$k = 1$ から始まるが、指数を $k-1$ ではなく k としている場合もある。

`double gsl_ran_geometric_pdf` (*const unsigned int k*, *const double p*)[Function]

パラメータが p で与えられる幾何分布にしたがう事象 k が生じる確率 $p(k)$ の値を返す。



`double gsl_cdf_geometric_P` (*const unsigned int k*, *const double p*)　[Function]
`double gsl_cdf_geometric_Q` (*const unsigned int k*, *const double p*)　[Function]

パラメータが p で与えられる幾何分布の累積分布関数 $P(k), Q(k)$ の値を返す。

19.36 超幾何分布

`unsigned int gsl_ran_hypergeometric` (*const gsl_rng * **r***, *unsigned int **n1***, *unsigned int **n2***, *unsigned int **t***) [Function]

超幾何分布 (hypergeometric distribution) にしたがう整数の乱数を返す。この分布は以下で定義される。

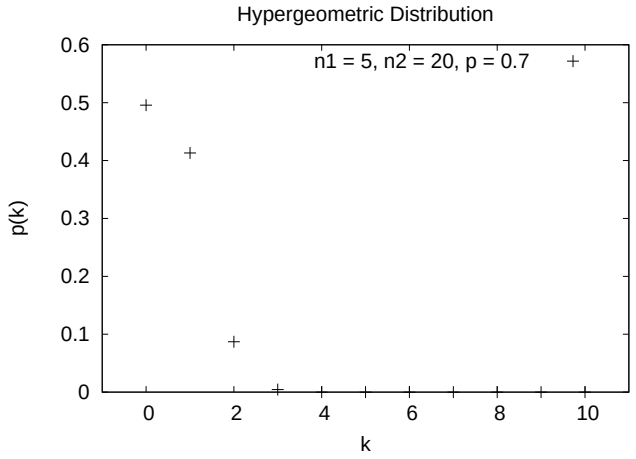
$$p(k) = C(n_1, k)C(n_2, t-k)/C(n_1+n_2, t)$$

ここで $C(a,b) = a!/(b!(a-b)!)$ で $t \le n_1 + n_2$ である。k の範囲は $\max(0, t-n_2), \ldots, \min(t, n_1)$ である。

もし、種類 1 のものが n_1 個、種類 2 のものが n_2 個あるとき、それらを合わせたものから一度に t 個をとり出した中に種類 1 のものが k 個含まれている確率が超幾何分布で表される。

`double gsl_ran_hypergeometric_pdf` (*double unsigned int **k***, *double unsigned int **n1***, *double unsigned int **n2***, *unsigned int **t***) [Function]

パラメータが *n1*、*n2*、t で与えられる超幾何分布にしたがう事象 k が生じる確率 $p(k)$ の値を返す。



`double gsl_cdf_hypergeometric_P` (*const unsigned int **k***, *const unsigned int **n1***, *const unsigned int **n2***, *const unsigned int **t***) [Function]
`double gsl_cdf_hypergeometric_Q` (*const unsigned int **k***, *const unsigned int **n1***, *const unsigned int **n2***, *const unsigned int **t***) [Function]

パラメータが *n1*、*n2*、t で与えられる超幾何分布の累積分布関数 $P(k), Q(k)$ の値を返す。

19.37　対数分布

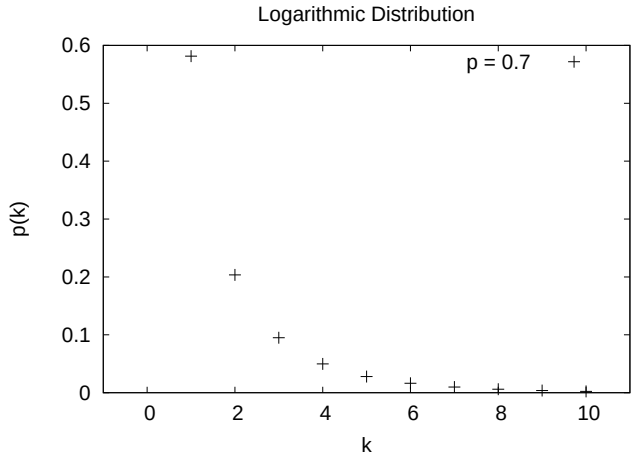
`unsigned int gsl_ran_logarithmic` (*const gsl_rng* * `r`, *const double* `p`) [Function]

対数分布にしたがう整数の乱数を返す。この分布は $k \geq 1$ で以下で定義される。

$$p(k) = \frac{-1}{\log(1-p)} \left(\frac{p^k}{k} \right)$$

`double gsl_ran_logarithmic_pdf` (*const unsigned int* `k`, *const double* `p`) [Function]

パラメータが p で与えられる対数分布にしたがう事象 k が生じる確率 $p(k)$ の値を返す。



19.38 かき混ぜと観測

以下の関数は、事象をかき混ぜ (shuffle)、取り出す (sample) というシミュレーションを行うためのものである。そのアルゴリズム中では乱数発生器を用いているため、品質の悪い乱数発生器を使うと、得られる結果の中になんらかの相関が反映されてしまうことがある。特に、周期の短い乱数発生器を使ってはいけない。詳しくはクヌース (1997) の 3.4.2 節、“Random Sampling and Shuffiling” を参照のこと。

`void gsl_ran_shuffle` (*const gsl_rng * **r***, *void * **base***, *size_t **n***, *size_t **size***) [Function]

配列 *base[0..n - 1]* に入れて渡される n 個のサイズが *size* のオブジェクトの順番をかき混ぜる。乱数発生器 r を使って置換を生成する。乱数発生器が理想的な乱数を返すことを前提に、$n!$ 個のあり得るすべての置換が同じ確率で生じるようにしている。

以下に 0 から 51 までをランダムにかき混ぜるコードを示す。

```
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***) [Function]

src[0 .. n - 1] に入れて渡される n 個のオブジェクトからランダムに選んだ k 個のオブジェクトを配列 *dest[k]* に入れる。両配列に保持される各オブジェクトの大きさは *size* で、同じでなければならない。乱数発生器 r がオブジェクトの選択に使われる。乱数発生器が理想的な乱数を返すことを前提に、すべてのオブジェクトが同じ確率で選ばれるようになっている。

同じオブジェクトが複数回選ばれることはなく、*dest[k]* の中で同じものが重複することはない。k は n 以下でなければならない。選ばれたオブジェクトの配列 *src* の中での順序は *dest* の中でも保たれる。この順序をランダムにしたいときは `gsl_ran_shuffle(r, dest, k, size)` を使えばよい。

以下に、0 から 99 までの数値から互いに異なる 3 つの値を取り出すコードを示す。

```
double a[3], b[100];
for (i = 0; i < 100; i++) 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***) [Function]

`gsl_ran_choose` と同じだが、これは同じものが複数回選ばれる可能性がある (選ばれたものを元の配列に戻す)。したがって *dest[k]* 中に同じオブジェクトが複数入れられることがある。k が n よりも大きくても構わない。

19.39　例

以下に、乱数発生器を使って乱数を発生するプログラムを示す。ここでは期待値が 3 のポアソン分布にしたがう乱数を 10 個生成する。

```
#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;

    /* 環境変数 GSL_RNG_TYPE を参照して使う乱数発生器を選ぶ */
    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc(T);

    /* 期待値 mu のポアソン分布にしたがう乱数を n 個表示する */
    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;
}
```

GSL のライブラリとヘッダファイルが '/usr/local' (デフォルトの場所) 以下にインストールされている場合は、コマンドラインで以下のようにすればこのプログラムがコンパイルできる。

```
$ gcc -Wall demo.c -lgsl -lgslcblas -lm
```

プログラムを実行したときの出力を以下に示す。

```
$ ./a.out
 2 5 5 2 1 0 3 4 1 1
```

発生する乱数の値は、乱数発生器に与える種によって変化する。使用する乱数生成器のデフォルトは変数 gsl_rng_default の値に指定されており、以下のようにすれば環境変数 GSL_RNG_SEED でその乱数発生器によって生成される乱数系列の種を変更できる。

```
$ GSL_RNG_SEED=123 ./a.out
GSL_RNG_SEED=123
 4 5 6 3 3 1 4 2 5 5
```

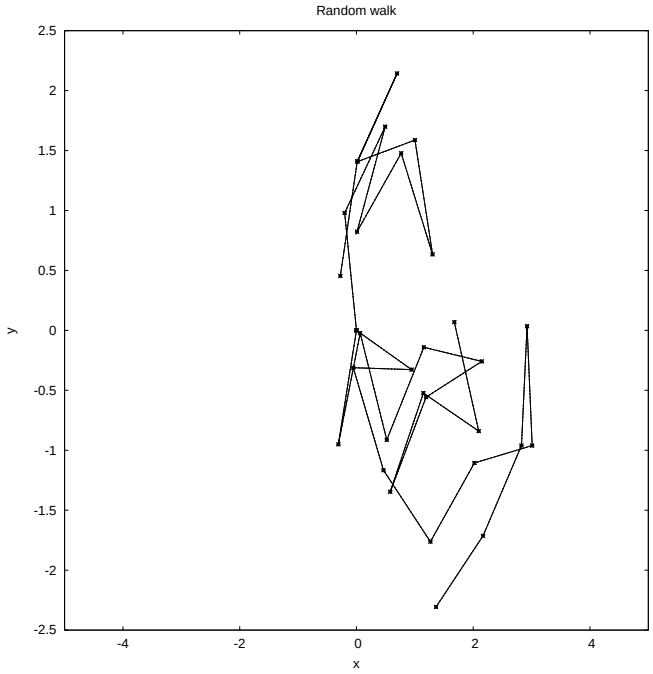
以下に、二次元空間でのランダム・ウォークを行うプログラムを示す。

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

int main (void)
{
    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;
}
```

図に、原点から 10 ステップのランダム・ウォークを 3 回行った結果を示す (種を変えずにプログラムを実行すると、まったく同じ系列を発生する。図は、種の値を 1970, 1971, 1972 と指定して行ったときのものである)。

以下のプログラムでは、$x = 2$ での標準正規分布の上および下の累積分布関数の値を計算する。

```
#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);
    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;
}
```

このプログラムを実行したときの結果を以下に示す。

```
prob(x < 2.000000) = 0.977250
prob(x > 2.000000) = 0.022750
Pinv(0.977250) = 2.000000
Qinv(0.022750) = 2.000000
```

19.40 参考文献

とにかくカバー範囲の広い、リュック・デブロイの *Non-Uniform Random Variate Generation* が参考書として推薦できる。この本には想像できる限りの膨大な種類の分布とそのアルゴリズムが載っている (オンラインでも入手できる。正誤表も同じサイトにある。著者は日本で過ごした経験があり、また冗談好きでもあるようで、本書の前書きに sukebe friends や Burger King mates への謝辞がある)。

- Luc Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, ISBN 0-387-96305-7.
  `http://cg.scs.carleton.ca/~luc/rnbookindex.html`

乱数の発生法はクヌースの本にもあり、よく知られた分布のアルゴリズムが載っている。またこの本には邦訳もある。

- Donald E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, 3rd Ed, Addison-Wesley, ISBN 0201896842 (1997).
- Donald E.Knuth (著), 有沢 誠, 和田 英一 (監訳), 斎藤 博昭, 長尾 高弘, 松井 祥悟, 松井 孝雄, 山内 斉 (訳), The Art of Computer Programming (2) 日本語版 Seminumerical algorithms (Ascii Addison Wesley programming series), アスキー, ISBN 978-4756145437 (2004).

米ローレンス・バークレー国立研究所の素粒子グループでは、以下のレビューの “Monte Coarlo” の章で乱数の分布を生成する手法について述べており、そのレビューの PS および PDF 形式のファイルをオンラインで自由に見ることができる。

- R. M. Barnett et al., “Review of Particle Properties”, *Physical Review*, **D54**(1) (1996).
  `http://pdg.lbl.gov/`

累積分布関数の計算法については、ケネディーとジェントルの *Statistical Computing* に概観されている。ほかにシステッドの *Elements of Statistical Computing* にもレビューがある。

- William E. Kennedy, James E. Gentle, *Statistical Computing*, Marcel Dekker, ISBN: 0-8247-6898-1 (1980).
- Ronald A. Thisted, *Elements of Statistical Computing*, Chapman & Hall, ISBN 0-412-01371-1 (1988).

正規分布の累積分布関数は、以下の論文を元にしている。

- W. J. Cody, W. Fraser, J. F. Hart, “Rational Chebyshev Approximations Using Linear Equations”, *Numerische Mathematik*, **12**(4), pp. 242–251 (1968).
- W. J. Cody, “Rational Chebyshev Approximations for the Error Function”, *Mathematics of Computation*, **23**(n107), pp. 631–637 (1969).

第20章　確率統計

この章では、GSL で用意している統計値を計算する関数について説明する。基本的な平均値 (mean)、分散 (variance)、標準偏差 (standard deviation) の計算に加え、絶対偏差 (absolute deviation)、歪度 (わいど、skewness)、尖度 (せんど、kurtosis)、中央値 (median)、任意の百分位数 (percentile) を計算できる。GSL で使っているアルゴリズムでは、平均値の計算中に巨大な値が生じてオーバーフローする事態を避けるために、再帰的に計算する。

GSL で用意している関数は標準的な浮動小数点や整数のデータ型を扱うことができる。倍精度実数を扱う関数は関数名に `gsl_stats` が付けられていて、'`gsl_statistics_double.h`' で宣言されている。整数を扱う関数には `gsl_stats_int` が付けられていて、'`gsl_statistics_int.h`' で宣言されている。

20.1　平均値、標準偏差、分散

`double gsl_stats_mean (`*const double* `data[]`, *const size_t* `stride`, *const size_t* `n``)` [Function]

データ長 n で刻み幅 *stride* のデータセット *data* の算術平均 (arithmetic mean) を返す。算術平均は標本平均 (sample mean) とも呼ばれ、$\hat{\mu}$ で表され、以下で定義される。

$$\hat{\mu} = \frac{1}{N} \sum x_i$$

ここで x_i はデータセット *data* の要素である。サンプルが正規分布に従うときは、$\hat{\mu}$ の分散は σ^2/N になる。

`double gsl_stats_variance (`*const double* `data[]`, *const size_t* `stride`, *const size_t* `n``)` [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の推定分散 (estimated variance)、または標本分散 (sample variance) を返す。標本分散は以下の $\hat{\sigma}^2$ で定義される。

$$\hat{\sigma}^2 = \frac{1}{N-1} \sum (x_i - \hat{\mu})^2$$

ここで x_i はデータセット *data* の要素である。正規化係数が ($1/N$ ではなく) $1/(N-1)$ であることで、標本分散は母分散 (population variance) $\hat{\sigma}^2$ に対する不偏分散 (unbiased variance) になる。サンプルが正規分布に従うとき、$\hat{\sigma}^2$ そのものの分散は $2\sigma^4/N$ になる。

この関数は内部で gsl_stats_mean を呼び出して平均値を計算する。もしすでに平均値を計算している場合には、その値を gsl_stats_variance_m に渡せば再計算せずにすむ。

double gsl_stats_variance_m (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **mean**) [Function]

与えられる平均値 *mean* に対する標本分散を返す。この関数では以下のように、$\hat{\mu}$ を引数で与えられる平均値で置き換えて、分散を計算する。

$$\hat{\sigma}^2 = \frac{1}{N-1}\sum(x_i - mean)^2$$

double gsl_stats_sd (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]
double gsl_stats_sd_m (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **mean**) [Function]

分散の平方根を計算して標準偏差として返す。それぞれ、上述した分散を計算する関数に対応している。

double gsl_stats_tss (*const double* **data[]**, *size_t* **stride**, *size_t* **n**) [Function]
double gsl_stats_tss_m (*const double* **data[]**, *size_t* **stride**, *size_t* **n**, *double* **mean**) [Function]

data[] で与えられる各データから平均値を引いて、二乗和 (total sum of squares, TSS) を求めて返す。gsl_stats_tss_m では引数で与えた平均値 *mean* を、gsl_stats_tss では内部で gsl_stats_mean を呼んで計算した平均値を使って、以下で与えられる値を計算する。

$$\mathrm{TSS} = \sum(x_i - mean)^2$$

double gsl_stats_variance_with_fixed_mean (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **mean**) [Function]

母集団の分布とその期待値があらかじめ分かっている場合に、*data* の不偏分散 (unbiased variance) を求めて返す。この関数では、以下のように、正規化係数として $1/N$ を使い、標本平均 $\hat{\mu}$ の代わりに与えられる母集団の既知の期待値 μ を使う。

$$\hat{\sigma}^2 = \frac{1}{N}\sum(x_i - \mu)^2$$

double gsl_stats_sd_with_fixed_mean (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **mean**) [Function]

固定値である母集団の平均値 *mean* に対して *data* の標準偏差を計算する。返される値は対応する分散の平方根である。

20.2 絶対偏差

`double gsl_stats_absdev` (*const double* **`data[]`**, *const size_t* **`stride`**, *const size_t* **`n`**) [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の、平均からの絶対偏差 (absolute deviation) を計算して返す。平均値に対する絶対偏差は以下の式で定義される。

$$absdev = \frac{1}{N}\sum |x_i - \hat{\mu}|$$

ここで x_i はデータセット *data* の要素である。データの散らばり具合を表す指標としては、分散よりも、平均値からの絶対偏差のほうがロバストである。内部で `gsl_stats_mean` を呼び出して *data* の平均値を計算する。

`double gsl_stats_absdev_m` (*const double* **`data[]`**, *const size_t* **`stride`**, *const size_t* **`n`**, *const double* **`mean`**) [Function]

以下で与えられる平均値 *mean* からの、データ長 n、刻み幅 *stride* のデータセット *data* の絶対偏差を計算して返す。

$$absdev = \frac{1}{N}\sum |x_i - mean|$$

既にデータセット *data* の平均値を計算しているような場合 (そして再計算をしなくてもよい場合) には、この関数を使った方がより早く処理を行うことができる。また、たとえば 0 や中央値などに対する絶対偏差を計算したい場合にも使うことができる。

20.3 高次モーメント (歪度と尖度)

`double gsl_stats_skew` (*const double* **`data[]`**, *const size_t* **`stride`**, *const size_t* **`n`**) [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の歪度 (わいど、skewness) を返す。期待値 μ、分散 σ^2 の確率変数 X の規準変数 $(X-\mu)/\sigma$ の、三次モーメントが歪度であり、たとえば正規分布のように左右対称な分布では 0、カイ二乗分布 (第 19.17 節) では正、$a > b$ のベータ分布 (第 19.20 節) では負になる。GSL での歪度の計算は以下で定義される。

$$skew = \frac{1}{N}\sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}}\right)^3$$

ここで x_i はデータセット *data* の要素である。

この関数は内部で `gsl_stats_mean` と `gsl_stats_sd` を呼び出して、それぞれ *data* の平均値と標準偏差をを計算する。

`double gsl_stats_skew_m_sd` (*const double* **`data[]`**, *const size_t* **`stride`**, *const size_t* **`n`**, *const double* **`mean`**, *const double* **`sd`**) [Function]

データ長 n、刻み幅 $stride$ のデータセット $data$ の歪度を、与えられるデータ平均値 $mean$ と標準偏差 sd を使って計算する。

既にデータセット $data$ の平均値と標準偏差を計算しているような場合には、この関数を使った方がより早く処理を行うことができる

`double gsl_stats_kurtosis (const double data[], const size_t stride, const size_t n)` [Function]

データ長 n、刻み幅 $stride$ のデータセット $data$ の尖度 (せんど、kurtosis) を返す。期待値 μ、分散 σ^2 の確率変数 X の規準変数 $(X-\mu)/\sigma$ の、四次モーメントが尖度であり、X が大きく (小さく) なっていくときにどれだけ速く確率密度が小さくなっていくか、あるいは「裾の重さ」を表す。GSL での尖度の計算は以下で定義され、正規分布の場合 0 になる。

$$kurtosis = \left(\frac{1}{N} \sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^4 \right) - 3$$

この定義では尖度は正規分布で 0 になるように正規化されているが、3 を引かずに、正規分布の場合に値が 3 になる定義が使われることもある。

`double gsl_stats_kurtosis_m_sd (const double data[], const size_t stride, const size_t n, const double mean, const double sd)` [Function]

以下で与えられるデータ長 n、刻み幅 $stride$ のデータセット $data$ の尖度を、与えられるデータ平均値 $mean$ と標準偏差 sd を使って計算する。

$$kurtosis = \left(\frac{1}{N} \sum \left(\frac{x_i - mean}{\hat{\sigma}} \right)^4 \right) - 3$$

既にデータセット $data$ の平均値と標準偏差を計算しているような場合には、この関数を使った方がより早く処理を行うことができる。

20.4 自己相関

`double gsl_stats_lag1_autocorrelation (const double data[], const size_t stride, const size_t n)` [Function]

以下で与えられる、データセット $data$ のずれ (時間遅れ、ラグ、lag) が 1 の自己相関係数 (lag-1 autocorrelation coefficient) を計算する。

$$a_1 = \frac{\sum_{i=1}^{n} (x_i - \hat{\mu})(x_{i-1} - \hat{\mu})}{\sum_{i=1}^{n} (x_i - \hat{\mu})(x_i - \hat{\mu})}$$

`double gsl_stats_lag1_autocorrelation_m (const double data[], const size_t stride, const size_t n, const double mean)` [Function]

与えられる平均値 $mean$ を使って、データセット $data$ のずれが 1 の自己相関係数を計算する。

20.5 共分散

`double gsl_stats_covariance (const double data1[], const size_t stride1, const double data2[], const size_t stride2, const size_t n)` [Function]

同じデータ長 n を持つ二つのデータセット *data1* と *data2* の共分散 (covariance) を以下で計算する。

$$cov = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{x})(y_i - \hat{y})$$

`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)` [Function]

同じデータ長 n を持つ二つのデータセット *data1* と *data2* に対して、与えられる二つの平均値 *mean1* と *mean2* を使って共分散を計算する。既にデータセット *data1* と *mean2* の平均値を計算しているような場合には、この関数を使った方がより早く処理を行うことができる。

20.6 相関係数

`double gsl_stats_correlation (const double data1[], const size_t stride1, const double data2[], const size_t stride2, const size_t n)` [Function]

与えられる二つのデータセット *data1* と *data2* から、以下で与えられるピアソン (Karl Pearson FRS) の相関係数 (correlation coefficient) を効率よく計算して返す。二つのデータセットのサイズは同じでなければならず、引数 n で与える。

$$r = \frac{cov(x,y)}{\hat{\sigma_x}\hat{\sigma_y}} = \frac{\frac{1}{n-1}\sum(x_i - \hat{x})(y_i - \hat{y})}{\sqrt{\frac{1}{n-1}\sum(x_i - \hat{x})^2}\sqrt{\frac{1}{n-1}\sum(y_i - \hat{y})^2}}$$

20.7 重み付きデータ

この節の関数は重み付き標本 (weighted sampling) の処理を行う。関数には標本を入れた配列 x_i と各標本に対応する重みの配列 w_i を渡す。各標本 x_i は分散 σ_i^2 の正規分布から得られたものとして扱われ、与えられる重み w_i はその分散の逆数 $w_i = 1/\sigma_i^2$ と定義される。重みが 0 とされた標本はデータセットから除去されたのと同じ事になる。

`double gsl_stats_wmean (const double w[], const size_t wstride, const double data[], const size_t stride, const size_t n)` [Function]

データ長 *n*、刻み幅 *stride* のデータセット *data* の重み付き平均値を、刻み幅 *wstride* で配列に格納されている重み w、データ長 *n* を使って計算する。重み付き平均値は以下の定義で計算される。

$$\hat{\mu} = \frac{\sum w_i x_i}{\sum w_i}$$

double gsl_stats_wvariance (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の標本分散を、刻み幅 *wstride* で配列に格納されている重み w、データ長 n を使って計算して返す。重み付きの標本分散は以下の定義で計算される。

$$\hat{\sigma}^2 = \frac{\sum w_i}{(\sum w_i)^2 - \sum (w_i^2)} \sum w_i (x_i - \hat{\mu})^2$$

N 個の重みがすべて等しい時、この式による重み付き分散は、重みのない係数が $1/(N-1)$ の分散 (不偏分散) と同じになる。

double gsl_stats_wvariance_m (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **wmean**)[Function]

データ長 n、刻み幅 *stride* のデータセット *data* の標本分散を、与えられた重み付き平均値 *wmean* を使って計算して返す。

double gsl_stats_wsd (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

標準偏差は分散の平方根として定義される。この関数は標本分散を計算する上述の関数 gsl_stats_wvariance の値の平方根を返す。

double gsl_stats_wsd_m (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **wmean**) [Function]

標本分散を計算する上述の関数 gsl_stats_wvariance_m の値の平方根を返す。

double gsl_stats_wvariance_with_fixed_mean (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **mean**) [Function]

母集団の分布の期待値 *mean* があらかじめ知られているときに、それを使って重み付きデータセット *data* の分散の不偏推定量 (すなわち不偏分散、unbiased estimate of the variance) の値を計算する。この場合は以下のように、標本平均 $\hat{\mu}$ を与えられる既知の期待値 μ に置き換えた値の計算を行う。

$$\hat{\sigma}^2 = \frac{\sum w_i (x_i - \mu)^2}{\sum w_i}$$

double gsl_stats_wsd_with_fixed_mean (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **mean**)[Function]

標準偏差は分散の平方根として定義される。上の関数gsl_stats_wvariance_with_fixed_mean の値の平方根を返す。

double gsl_stats_wtss (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**,*const size_t* **n**) [Function]
double gsl_stats_wtss_m (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *double* **wmean**) [Function]

data[] で与えられる各データから重み付き平均値を引いて、重み付きの二乗和 (total sum of squares, TSS) を求めて返す。gsl_stats_wtss_m では引数で与えた期待値 *wmean* を、gsl_stats_wtss では内部で gsl_stats_wmean を呼んで計算した平均値を *wmean* として、以下で与えられる値を計算する。

$$\mathrm{TSS} = \sum w_i (x_i - wmean)^2$$

double gsl_stats_wabsdev (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

重み付きデータセット *data* の絶対偏差 (absolute deviation) を計算する。平均値から計算される絶対偏差は以下の式で定義される。

$$absdev = \frac{\sum w_i |x_i - \hat{\mu}|}{\sum w_i}$$

double gsl_stats_wabsdev_m (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **wmean**) [Function]

重み付きデータセット *data* の絶対偏差を、与えられる重み付き期待値 *wmean* を使って計算する。

double gsl_stats_wskew (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

以下で与えられる、データセット *data* の重み付き歪度を計算する。

$$skew = \frac{\sum w_i ((x_i - \bar{x})/\sigma)^3}{\sum w_i}$$

double gsl_stats_wskew_m_sd (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **wmean**, *const double* **wsd**) [Function]

データセット *data* の重み付き歪度を、与えられる重み付き期待値 *wmean* と重み付き標準偏差 *wsd* を使って計算する。

double gsl_stats_wkurtosis (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

以下で与えられる、データセット *data* の重み付き尖度を計算する。

$$kurtosis = \frac{\sum w_i ((x_i - \bar{x})/\sigma)^4}{\sum w_i} - 3$$

double gsl_stats_wkurtosis_m_sd (*const double* **w[]**, *const size_t* **wstride**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**, *const double* **wmean**, *const double* **wsd**) [Function]

データセット *data* の重み付き尖度を、与えられる重み付き期待値 *wmean* と重み付き標準偏差 *wsd* を使って計算する。

20.8　最大値、最小値

以下の関数は、与えられるデータセットの中から値が最大のものや最小のもの (またはその添え字) を探して返す。データ中に NaN があったときは、最大および最小は定義されないので、NaN を返す。それが添え字を返す関数であれば、与えられた配列中の最初に NaN がある場所を返す。

double gsl_stats_max (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の最大値を返す。最大の要素 x_i は、全ての j に関して $x_i \geq x_j$ を満たす要素として定義される。

絶対値が最大の要素を探したいときは、この関数にデータを渡す前に fabs または abs でデータを絶対値に変換しておけばよい。

double gsl_stats_min (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の最小値を返す。最小の要素 x_i は、全ての j に関して $x_i \leq x_j$ を満たす要素として定義される。

絶対値が最小の要素を探したいときは、この関数にデータを渡す前に fabs または abs でデータを絶対値に変換しておけばよい。

void gsl_stats_minmax (*double* * **min**, *double* * **max**, *const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

データセット *data* から最小値 *min* と最大値 *max* を一度に探して、*min* と *max* に入れて返す。

size_t gsl_stats_max_index (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の最大値を持つ要素の添え字を返す。最大の要素 x_i は、全ての j に関して $x_i \geq x_j$ を満たす要素として定義される。同じ最大値を持つ要素が複数あるときには、最初のものが返される。

size_t gsl_stats_min_index (*const double* **data[]**, *const size_t* **stride**, *const size_t* **n**) [Function]

データ長 n、刻み幅 *stride* のデータセット *data* の最小値を持つ要素の添え字を返す。最小の要素 x_i は、全ての j に関して $x_i \leq x_j$ を満たす要素として定義される。同じ最小値を持つ要素が複数あるときには、最初のものが返される。

`void gsl_stats_minmax_index` (*size_t* * ***min_index***, *size_t* * ***max_index***, *const double* ***data[]***, *const size_t* ***stride***, *const size_t* ***n***) [Function]

データセット *data* から最小要素の添え字 *min_index* と最大要素の添え字 *max_index* を一度に探して、*min_index* と *max_index* に入れて返す。同じ最大値、最小値を持つ要素が複数あるときには、最初のものが返される。

20.9 中央値と百分位数

以下の関数は、あらかじめ整列されたデータに対する中央値 (median) と分位数 (quantile) を計算する。GSL では利用上の便宜を図るため、百分位数 (percentile、0 から 100 までの範囲) ではなく分位数 (0 から 1 まで) を用いている。

`double gsl_stats_median_from_sorted_data` (*const double* ***sorted_data[]***, *const size_t* ***stride***, *const size_t* ***n***) [Function]

データ長 n、刻み幅 *stride* のデータ *sorted_data* の中央値を返す。配列中のデータは数値の昇順に整列しておかねばならない。整列されているかどうかのチェックは関数の内部では行われないので、必要に応じて先に `gsl_sort` 関数などで整列しておく。

データ長が奇数の場合、中央値は添え字が $(n-1)/2$ の要素の値になる。データ長が偶数なら二つの要素 $(n-1)/2$ と $n/2$ の平均値になる。つまりデータ長が偶数の場合は補間が行われるということであり、そのため、データが整数の場合でも返り値は実数である。

`double gsl_stats_quantile_from_sorted_data` (*const double* ***sorted_data[]***, *const size_t* ***stride***, *const size_t* ***n***, *const double* ***f***) [Function]

データ長 n、刻み幅 *stride* の倍精度実数のデータ *sorted_data* の分位数 (quantile) を返す。データセットの要素は昇順に整列されていなければならない。分位数は 0 から 1 の間の小数 f で指定される。たとえば 75 番目の百分位数を計算するには f の値を 0.75 にする。整列されているかどうかのチェックは関数の内部では行われないので、先に `gsl_sort` 関数などで整列しておかねばならない。

分位数は以下のように補間して計算される。

$$\text{quantile} = (1-\delta)x_i + \delta x_{i+1}$$

ここで i は `floor`$((n-1)f)$ で、δ は $(n-1)f - i$ である。

データの最小値 `data[0*stride]` は f を 0 に、最大値 `data[(n-1)*stride]` は f を 1 に、中央値は f を 0.5 にすることで得られる。分位数を求めるのに補間を行うため、データが整数の場合でも返り値は実数である。

20.10　例

統計値計算の簡単な例を以下に示す。

```
#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;
}
```

このプログラムは以下のように出力する。

```
The dataset is 17.2, 18.1, 16.5, 18.3, 12.6
The sample mean is 16.54
The estimated variance is 4.2984
The largest value is 18.3
The smallest value is 12.6
```

次に、整列済みのデータを使う簡単な例を以下に示す。

```
#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);
    printf("The upper quartile is %g\n", upperq);
    printf("The lower quartile is %g\n", lowerq);

    return 0;
}
```

このプログラムは以下のように出力する。

```
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
```

20.11 参考文献

以下のシリーズが、統計に関するほとんど全ての分野を網羅している。

- Maurice Kendall, Alan Stuart, J. Keith Ord, *The Advanced Theory of Statistics* (multiple volumes) reprinted as *Kendall's Advanced Theory of Statistics*, John Wiley & Sons, ISBN 047023380X (1994).

統計学での考え方は、ベイズ理論的に考えると理解しやすくなることも多い。以下の本がそういう意味で参考になる。

- Andrew Gelman, John B. Carlin, Hal S. Stern, Donald B. Rubin, *Bayesian Data Analysis*, Chapman & Hall, ISBN 0412039915 (1995).

物理学に親しんでいる人向けには、ローレンス・バークレー国立研究所の素粒子観測グループが雑誌に書いた確率統計の記事がよいだろう。このレビューは、`http://pdg.lbl.gov/` で自由に読むことができる。

- R. M. Barnett et al., “Review of Particle Properties”, *Physical Review*, **D54**(1) (1996).

第21章　ヒストグラム

この章ではヒストグラム (histogram) を生成する関数について説明する。ヒストグラムはデータの分布を概観するのに便利な手法である。一つのヒストグラムは複数の階級 (bin) からなり、データの発生する区間を分割した小区間の一つ一つが各階級に割り当てられ、変数 x がその区間内の値をとった回数が、その階級の値 (度数) になる。GSL では階級値は実数で実装してあり、したがって整数と非整数の両方の確率分布モデルに使うことができる。各階級には任意の幅の区間を割り当てることができる (デフォルトは等間隔である)。また一次元および二次元のヒストグラムを扱うことができる。

生成したヒストグラムは確率分布関数 (probability distribution function) に変換することができる。その確率分布にしたがう乱数を生成するルーチンが用意されており、現実のデータを用いたシミュレーションを容易に行うことができる。

この章の関数はすべて ‘`gsl_histogram.h`’ および ‘`gsl_histogram2d.h`’ で宣言されている。

21.1　ヒストグラム構造体

一つのヒストグラムは以下の構造体で定義される。

`gsl_histogram` [Data Type]

`size_t n`
: そのヒストグラムが持つ階級の数

`double * range`
: 各階級に割り当てられた小区間。ポインタ *range* が指す要素数が n+1 の配列に保持される。

`double * bin`
: 各階級の度数。度数はポインタ *bin* が指す要素数 n の配列に保持される。階級の度数は実数なので、必要によって非整数で数えることもできる。

階級が n 個あれば配列 *range* の要素数は n+1 であり、階級 $bin[i]$ の小区間の範囲は $range[i]$ 以上 $range[i+1]$ 未満である。数式で表現すると以下のようになる。

$$bin[i] \;\; corresponds\ to \;\; range[i] \leq x < range[i+1]$$

各階級と小区間の関係を変数 x の数直線上に図示すると以下のようになる。

```
[ bin[0] )[ bin[1] )[ bin[2] )[ bin[3] )[ bin[4] )
```

```
---|---------|---------|---------|---------|---------|--- x
 r[0]      r[1]      r[2]      r[3]      r[4]      r[5]
```

この図で、配列 *range* の要素が持つ値は r で表されている。各階級で左側の角括弧 '[' は「以上」($r \leq x$) を表し、右側の丸括弧 ')' は未満 ($x < r$) を表す。したがってヒストグラム全体の範囲のちょうど最大値になる標本は数える対象にならない。もしこれを対象に含めたい場合には、そのための階級を用意する必要がある。

gsl_histogram 構造体とヒストグラムに関する関数はヘッダファイル 'gsl_histogram.h' で定義されている。

21.2 ヒストグラム・インスタンスの生成

以下の関数は malloc や free と同じ要領でヒストグラムのインスタンスの生成および解放を行い、また独自のエラー・チェックも行う。インスタンスを生成するための十分なメモリが確保できない場合、エラー・ハンドラーをエラー・コード GSL_ENOMEM で呼び出し、NULL を返す。もし GSL で用意しているエラー・ハンドラーを使ってプログラムを強制終了させるのであれば、alloc のたびにエラー・チェックを行う必要はない。

gsl_histogram * gsl_histogram_alloc (*size_t* **n**) [Function]

階級数が n のヒストグラム構造体 gsl_histogram のインスタンスを生成し、インスタンスへのポインタを返す。メモリが確保できなかったときはエラー・コード GSL_ENOMEM でエラー・ハンドラーを呼び出し、NULL ポインタを返す。階級の度数とその区間は初期化されないので、構造体をヒストグラムとして使うためには次の関数を使って初期化する必要がある。

int gsl_histogram_set_ranges (*gsl_histogram* * **h**, *const double* ***range[]***, *size_t* ***size***) [Function]

既に確保されているヒストグラムのインスタンス h に、与えられる階級の幅 *range* と階級の数 *size* を設定する。各階級の度数には 0 が入れられる。配列 *range* には階級の幅を入れておく。それは任意の値でよいが、添え字に対して単調増加でなければならない。

以下に階級の幅に対数、$[1, 10)$、$[10, 100)$、$[100, 1000)$ を使うヒストグラムの例を示す。

```
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);
```

配列 *range* の要素数は階級の数よりも一つ多いことに注意しなければならない。上限値（上の例では 1000.0）に対する度数を数えたいときは、そのための階級を加える。

`int gsl_histogram_set_ranges_uniform (`*gsl_histogram * **h***, *double **xmin***, *double **xmax***`)` [Function]

下限 *xmin* から上限 *xmax* までを等分する階級を既に確保されているヒストグラムのインスタンス *h* に設定する。各階級の度数には 0 が入れられる。各階級の区間は以下のようになる。

$$\begin{array}{lll} \textit{bin[0]} & \text{corresponds to} & \textit{xmin} \le x < \textit{xmin} + d \\ \textit{bin[1]} & \text{corresponds to} & \textit{xmin} + d \le x < \textit{xmin} + 2d \\ & \cdots\cdots\cdots & \\ \textit{bin[n-1]} & \text{corresponds to} & \textit{xmin} + (n-1)d \le x < \textit{xmax} \end{array}$$

ここで d は階級の幅 $d = (\textit{xmax} - \textit{xmin})/n$ である。

`void gsl_histogram_free (`*gsl_histogram * **h***`)` [Function]

ヒストグラム *h* が確保している全てのメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

21.3 ヒストグラムの複製

`int gsl_histogram_memcpy (`*gsl_histogram * **dest***, *const gsl_histogram * **src***`)` [Function]

ヒストグラム *src* を既に確保されているヒストグラムのインスタンス *dest* にコピーし、両者を全く同じものにする。ヒストグラムのサイズは両方で同じでなければならない。

`gsl_histogram * gsl_histogram_clone (`*const gsl_histogram * **src***`)` [Function]

ヒストグラムのインスタンスを新たに生成し、そこに *src* をコピーして、新しく生成したインスタンスへのポインタを返す。

21.4 ヒストグラム中の要素の参照と操作

ヒストグラムの持つ階級を参照、あるいは操作するには、対象とする階級を、x の値で指定する方法と、階級を直接指定する方法の二通りがある。以下の関数は指定される x の値がどの階級に属するかを二分探索を行って決定する。

`int gsl_histogram_increment (`*gsl_histogram * **h***, *double **x***`)` [Function]

ヒストグラム h で、指定される値 x が属する階級に 1.0 を加える。x が属する階級が存在すれば、返り値は 0 になり、エラーがなかったことを示す。x が下限値よりも小さいときは GSL_EDOM を返し、ヒストグラム中の階級値はどこも変わらない。x が上限値以上のときも同じであり、単にそういった値は無視される。これは、巨大なデータセットで各階級の幅が小さいような場合は探索に時間がかかることがあるためであり、どちらの場合もエラー・ハンドラーは呼ばれない。

`int gsl_histogram_accumulate (`*gsl_histogram * h, double x, double weight*`)` [Function]

階級に 1 ではなく実数 *weight* を加えること以外は gsl_histogram_increment と同じである。

`double gsl_histogram_get (`*const gsl_histogram * h, size_t i*`)` [Function]

ヒストグラム h の i 番目の階級の値を返す。i が h の持たない階級を指すような場合はエラー コード GSL_EDOM でエラー・ハンドラーが呼ばれ、関数は 0 を返す。

`int gsl_histogram_get_range (`*const gsl_histogram * h, size_t i, double * lower, double * upper*`)` [Function]

ヒストグラム h の i 番目の階級の、下限値と上限値を返す。指定される i がそのヒストグラムの範囲に入っていれば、下限値が *lower* に、上限値が *upper* にそれぞれ入れられる。下限値はその階級の範囲に含まれるが (ちょうどその下限値を持つ標本はその階級に数えられる)、上限値は含まれない (ちょうどその上限値を持つ標本は、一つ上の階級があればそこに含まれる)。i が有効であれば 0 を返す。そうでなければエラー・ハンドラーを呼びエラー・コード GSL_EDOM を返す。

`double gsl_histogram_max (`*const gsl_histogram * h*`)` [Function]
`double gsl_histogram_min (`*const gsl_histogram * h*`)` [Function]
`size_t gsl_histogram_bins (`*const gsl_histogram * h*`)` [Function]

それぞれ、ヒストグラム h の最上階級の上限値、最下階級の下限値、階級の数を返す。これらを使えば gsl_histogram の要素を直接見なくてもすむ。

`void gsl_histogram_reset (`*gsl_histogram * h*`)` [Function]

ヒストグラム h の全ての階級値を 0 にする。

21.5 ヒストグラムの範囲の検索

以下の関数で、x のある値を範囲に含む階級の値を見たり、変更したりできる。

`int gsl_histogram_find (`*const gsl_histogram * h, double x, size_t * i*`)` [Function]

与えられる x の値をその範囲に含む階級の番号を探し、i に入れる。探索は、階級の幅が等間隔の場合について最適化された二分探索で行われる。x がヒストグラムの範囲に含まれていれば i を見つけ GSL_SUCCESS を返す。そうでなければエラー・ハンドラーを呼び出し、GSL_EDOM を返す。

21.6 ヒストグラムの統計値

`double gsl_histogram_max_val (const gsl_histogram * h)` [Function]

ヒストグラム h のすべての階級中で最大の度数を返す。

`size_t gsl_histogram_max_bin (const gsl_histogram * h)` [Function]

ヒストグラム h のすべての階級中で最大の度数を持つ階級を示す添え字を返す。複数の階級が同じ度数の時は、範囲がもっとも下の階級を返す。

`double gsl_histogram_min_val (const gsl_histogram * h)` [Function]

ヒストグラム h のすべての階級中で最小の度数を返す。

`size_t gsl_histogram_min_bin (const gsl_histogram * h)` [Function]

ヒストグラム h のすべての階級中で最小の度数を持つ階級を示す添え字を返す。複数の階級が同じ度数の時は、範囲がもっとも下の階級を返す。

`double gsl_histogram_mean (const gsl_histogram * h)` [Function]

ヒストグラムに数えられている変数が確率分布にしたがっているとして、その平均値を返す。階級の度数が負の場合、その階級は無視される。得られる値の精度は、階級の幅よりも精密にはならない。

`double gsl_histogram_sigma (const gsl_histogram * h)` [Function]

ヒストグラムに数えられている変数が確率分布にしたがっているとして、その標準偏差を返す。階級の度数が負の場合、その階級は無視される。得られる値の精度は、階級の幅よりも精密にはならない。

`double gsl_histogram_sum (const gsl_histogram * h)` [Function]

全ての階級の度数の和を返す。度数が負の場合も無視はされない (無視する場合よりも和が小さくなる)。

21.7 ヒストグラムの操作

`int gsl_histogram_equal_bins_p (const gsl_histogram * h1, const gsl_histogram * h2)` [Function]

二つのヒストグラムを比較し、全ての階級の範囲が一致するとき 1 を、そうでなければ 0 を返す。

`int gsl_histogram_add (gsl_histogram * h1, const gsl_histogram * h2)` [Function]

ヒストグラム *h1* の各階級の度数にヒストグラム *h2* の対応する階級の度数を $h_1'(i) = h_1(i) + h_2(i)$ のようにして加える。二つのヒストグラムの階級はすべて範囲が一致している必要がある。

`int gsl_histogram_sub` (*gsl_histogram* * **`h1`**, *const gsl_histogram* * **`h2`**) [Function]

ヒストグラム *h1* の各階級の度数からヒストグラム *h2* の対応する階級の度数を $h_1'(i) = h_1(i) - h_2(i)$ のようにして減ずる。二つのヒストグラムの階級はすべて範囲が一致している必要がある。

`int gsl_histogram_mul` (*gsl_histogram* * **`h1`**, *const gsl_histogram* * **`h2`**) [Function]

ヒストグラム *h1* の各階級の度数に、ヒストグラム *h2* の対応する階級の度数を $h_1'(i) = h_1(i) \times h_2(i)$ のようにして乗じる。二つのヒストグラムの階級はすべて範囲が一致している必要がある。

`int gsl_histogram_div` (*gsl_histogram* * **`h1`**, *const gsl_histogram* * **`h2`**) [Function]

ヒストグラム *h1* の各階級の度数を、ヒストグラム *h2* の対応する階級の度数で $h_1'(i) = h_1(i)/h_2(i)$ のようにして除する。二つのヒストグラムの階級はすべて範囲が一致している必要がある。

`int gsl_histogram_scale` (*gsl_histogram* * **`h`**, *double* **`scale`**) [Function]

ヒストグラム *h* の各階級の度数に、$h_1'(i) = h_1(i) \times scale$ のようにして *scale* を乗ずる。

`int gsl_histogram_shift` (*gsl_histogram* * **`h`**, *double* **`offset`**) [Function]

ヒストグラム h の各階級の度数を、$h_1'(i) = h_1(i) + offset$ として増減する。

21.8 ヒストグラムのファイル入出力

GSL では、ヒストグラムをバイナリ・データまたは整形済みテキストとしてファイルに保存、またはファイルから読み込む関数を用意している。

`int gsl_histogram_fwrite` (*FILE* * **`stream`**, *const gsl_histogram* * **`h`**) [Function]

ヒストグラム *h* の各階級の範囲と度数をファイル *stream* に出力する。出力できれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。出力形式は計算機アーキテクチャに依存したバイナリ形式なので、移植性は低い。

`int gsl_histogram_fread` (*FILE* * **`stream`**, *gsl_histogram* * **`h`**) [Function]

ヒストグラム *h* の各階級の範囲と度数をファイル *stream* から読み込む。ヒストグラムのインスタンス *h* はあらかじめ、読み込もうとするデータにあわせた階級幅、階級数で生成しておかねばならない。この階級数を使って読み込むバイト数が決定される。読み込みが完了すれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。入力データは計算機アーキテクチャに依存したバイナリ形式でなければならない。

`int gsl_histogram_fprintf (`*FILE * **stream***, const gsl_histogram * **h***, const char * **range_format***, const char * **bin_format***`)` [Function]

ヒストグラム *h* の各階級の範囲と度数をファイル *stream* に、一行ずつ指定された形式 *range_format* および *bin_format* で出力する。形式は実数に対する `%g`、`%e`、`%f` のいずれかでなければならない。出力できれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。出力形式は以下のように、空白文字で区切られた三列からなる。

```
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]
```

範囲を示す値は二つとも *range_format* で指定される形式で、階級の度数を表す数値は *bin_format* で指定される形式で出力される。各行は一つの階級の下限値、上限値、度数からなる。ある階級の上限値は、一つ上の階級の下限値でもあるため、出力される行は違うがこれらの値は重複している。しかし行指向のツール (awk, perl, ruby など) で操作するときの利便性を考えて、こういった形式になっている。

`int gsl_histogram_fscanf (`*FILE * **stream***, gsl_histogram * **h***`)` [Function]

ヒストグラム *h* の各階級の範囲と度数をファイル *stream* から読み込む。データの形式は `gsl_histogram_fprintf` で出力される三列からなる形式である。ヒストグラムのインスタンス *h* はあらかじめ、読み込もうとするデータにあわせた階級数で生成しておかねばならない。この階級数を使って読み込む数値の数が決定される。読み込みが完了すれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。

21.9 ヒストグラムからの確率分布事象の発生

ヒストグラムは、確率分布にしたがって生じる事象を数えることで作られる。統計上の誤差はあるが、各階級の度数は変数 x の値がその階級の持つ範囲に入る確率を表していると言える。その確率分布関数を一変数関数 $p(x)dx$ として表すと、

$$p(x) = n_i/(N w_i)$$

となる。ここで n_i は階級 i の小区間内に発生した x の回数、w_i は階級の幅、N は事象の全回数である。各階級内での事象の分布は一様であると仮定される。

21.10 ヒストグラム確率分布構造体

ヒストグラムに当てはめる確率分布関数 (probability distribution function, PDF) は階級の集合であり、各階級の度数は連続値の変数 x がその階級の範囲内に生じる確率を表す。確率分布関数は、

実際には以下の構造体に保持される累積分布関数 (cumulative probability distribution function) で定義される。累積分布関数のとる値は範囲 $[0,1]$ で単調増加であり、x の値と一対一対応がつくため、逆関数で標本を発生させるときには、$[0,1]$ の範囲で一様乱数を発生させ、累積分布関数の値がそれになる x の値を見ることで目的とする確率分布にしたがう標本を得ることができる。

`gsl_histogram_pdf` [Data Type]

`size_t n`
確率分布関数を近似するための階級の数。

`double * range`
各階級の幅。ポインタ *range* が指す n+1 個の要素を持つ配列に保持される。

`double * sum`
ポインタ sum が差す n 個の要素を持つ配列に保持される累積度数。

この `gsl_histogram_pdf` 構造体は、確率分布を表し、それにしたがった乱数の生成に利用できる。以下の関数で生成できる。

`gsl_histogram_pdf * gsl_histogram_pdf_alloc` (*size_t* **`n`**) [Function]

階級数 n の確率分布のインスタンスを生成し、その `gsl_histogram_pdf` 構造体インスタンスへのポインタを返す。十分なメモリが確保できなければエラー・コード `GSL_ENOMEM` でエラー・ハンドラーを呼び出し、NULL を返す。

`int gsl_histogram_pdf_init` (*gsl_histogram_pdf* * **`p`**, *const gsl_histogram* * **`h`**) [Function]

すでにデータの入っているヒストグラム h に基づいて確率分布のインスタンス p を初期化する。h に度数が負の階級があるときは、確率分布関数は負の値を取れないため、エラー・コード `GSL_EDOM` でエラー・ハンドラーを呼び出す。

`void gsl_histogram_pdf_free` (*gsl_histogram_pdf* * **`p`**) [Function]

確率分布のインスタンス p に関するメモリをすべて解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

`double gsl_histogram_pdf_sample` (*const gsl_histogram_pdf* * **`p`**, *double* **`r`**) [Function]

0 から 1 の範囲の一様乱数 r を使って、確率分布 p にしたがう標本 s を一つ発生する。標本 s を発生するアルゴリズムは、i が $sum[i] \le r < sum[i+1]$ を満たす添え字で、δ が比 $(r - sum[i])/(sum[i+1] - sum[i])$ であるとき、以下で与えられる。

$$s = \mathrm{range}[i] + \delta \times (\mathrm{range}[i+1] - \mathrm{range}[i])$$

21.11 ヒストグラムのプログラム例

以下に、標準入力から一列の数値データを読み込んでヒストグラムを生成する簡単なプログラムを示す。実行時にコマンドライン引数でヒストグラム全体の上限値、下限値、階級数を指定する。

標準入力からは一度に一行ずつ数値を読み込んで、その都度ヒストグラムに加えていく。入力が終了したとき gsl_histogram_fprintf を使ってヒストグラムを表示する。

```
#include <stdio.h>
#include <stdlib.h>
#include <gsl/gsl_histogram.h>

int main (int argc, char **argv)
{
    double a, b, x;
    size_t n;
    gsl_histogram * h;

    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]);

    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);

    return 0;
}
```

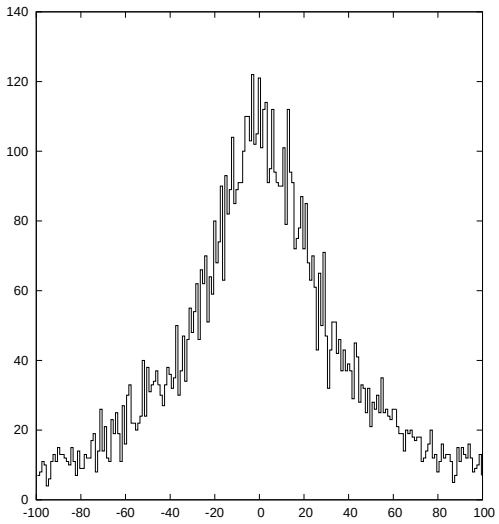
以下にプログラムの実行例を示す。上のプログラムを gsl-histogram という名前の実行ファイルにコンパイルし、パラメータが $a = 30$ のコーシー分布 (第 19.8 節) にしたがう乱数を 10000 個生成

し、−100 から 100 の範囲で階級数 200 のヒストグラムを生成する (gsl-randist というコマンドは、GSL のライブラリ本体と同時にインストールされる。デフォルトでは '/usr/local/bin/gsl-randist')。

```
$ gsl-randist 0 10000 cauchy 30 | gsl-histogram -100 100 200 > histogram.dat
```

得られるヒストグラムはコーシー分布に近い形をしているが、標本数が有限であるために理想的な形とはならない。

```
$ awk '{print $1, $3 ; print $2, $3}' histogram.dat | graph -T X
```



21.12 二次元ヒストグラム

二次元ヒストグラムは、(x, y) 平面上の各範囲に発生する事象を数えるための階級の集合からなる。最も簡単な二次元ヒストグラムの使い方は、二次元座標での位置情報 $n(x, y)$ を数えることである。または相関のある変数を数えることで同時分布 (joint distribution) を推定することなどがある。たとえばある実験において、ある事象の位置 (x) とエネルギーの大きさ E が同時に検知器で得られたとすると、これらは同時分布 $n(x, E)$ としてヒストグラムを作れる。

21.13 二次元ヒストグラム構造体

二次元ヒストグラムは以下の構造体で定義される。

gsl_histogram2d [Data Type]

`size_t nx, ny`
: x 方向と y 方向でのそれぞれのヒストグラムの階級数。

`double * xrange`
: x 方向での階級の幅。ポインタ *xrange* が指す要素数 $nx+1$ の配列に保持される。

`double * yrange`
: y 方向での階級の幅。ポインタ *yrange* が指す要素数 $ny+1$ の配列に保持される。

`double * bin`
: 各階級の度数。ポインタ *bin* が指す配列に保持される。階級の度数は浮動小数点実数なので、非整数値でも数えることができる。配列 *bin* には階級の二次元配列が一つのメモリブロック `bin(i, j) = bin[i * ny + j]` のようにして格納される。

階級 `bin(i, j)` の範囲は、x 方向では `xrange[i]` 以上 `xrange[i+1]` 未満、y 方向では `yrange[j]` 以上 `yrange[j+1]` 未満である。不等式で表現すると以下のようになる。

$$\begin{array}{lcl} \text{bin(i, j)} & \text{corresponds to} & xrange[i] \le x < xrange[i+1] \\ & \text{and} & yrange[j] \le y < yrange[j+1] \end{array}$$

ヒストグラム全体での上限の値は勘定に入らないことに注意する必要がある。もしこれを対象に含めたい場合には、そのための階級（行または列）を加える必要がある。

`gsl_histogram2d` 構造体と二次元ヒストグラムの関数はヘッダファイル '`gsl_histogram2d.h`' で宣言されている。

21.14 二次元ヒストグラム構造体のインスタンスの生成

二次元ヒストグラムのメモリは `malloc` や `free` と同じ要領で確保される。また独自のエラー・チェックも行う。インスタンスを生成するための十分なメモリが確保できない場合、エラー・ハンドラーをエラー・コード `GSL_ENOMEM` で呼び出し、NULL を返す。もしライブラリで用意されているエラー・ハンドラーを使ってプログラムを強制終了させるのであれば、`alloc` のたびにエラー・チェックを行う必要はない。

`gsl_histogram2d * gsl_histogram2d_alloc` (*size_t* **`nx`**, *size_t* **`ny`**) [Function]

x 方向の階級数が nx、y 方向の階級数が ny の二次元ヒストグラムのインスタンスを生成し、`gsl_histogram2d` 構造体へのポインタを返す。メモリが確保できなかったときはエラー・コード `GSL_ENOMEM` でエラー・ハンドラーを呼び出し、NULL ポインタを返す。ヒストグラムとして使うためには、階級の度数と幅を以下の関数を使って初期化する必要がある。

`int gsl_histogram2d_set_ranges` (*gsl_histogram2d* * **`h`**, *const double* **`xrange[]`**, *size_t* **`xsize`**, *const double* **`yrange[]`**, *size_t* **`ysize`**) [Function]

既に確保されているヒストグラムのインスタンス h に、与えられる各階級の小区間の幅 *xrange* と *yrange* 、階級の数 *xsize* と *ysize* を設定する。各階級の度数には 0 が入れられる。

`int gsl_histogram2d_set_ranges_uniform` (*gsl_histogram2d* * **`h`**, *double* **`xmin`**, *double* **`xmax`**, *double* **`ymin`**, *double* **`ymax`**) [Function]

下限 *xmin* から上限 *xmax* まで、および下限 *ymin* から上限 *ymax* までを等分する階級を先に確保したヒストグラムのインスタンス *h* に設定する。各階級の度数には 0 が入れられる。

`void gsl_histogram2d_free` (*gsl_histogram2d * **h***) [Function]

ヒストグラム *h* が確保している全てのメモリを解放する。

21.15　二次元ヒストグラムの複製

`int gsl_histogram2d_memcpy` (*gsl_histogram2d * **dest***, *const gsl_histogram2d * **src***) [Function]

ヒストグラム *src* を既に確保されているヒストグラムのインスタンスを *dest* にコピーし、両者を全く同じものにする。ヒストグラムのサイズは両方で同じでなければならない。

`gsl_histogram2d * gsl_histogram2d_clone` (*const gsl_histogram2d * **src***) [Function]

ヒストグラムのインスタンスを新たに生成し、そこに *src* をコピーして、新しく生成したインスタンスへのポインタを返す。

21.16　二次元ヒストグラム中の要素の参照と操作

二次元ヒストグラムの持つ階級を参照、操作するには、(x, y) の値を組で指定する方法と、階級 (i, j) を直接指定する方法の二通りがある。(x, y) 座標を使う方法は、x 方向と y 方向の両方で二分探索を行って、その座標を含む階級を探し出す。

`int gsl_histogram2d_increment` (*gsl_histogram2d * **h***, *double **x***, *double **y***) [Function]

ヒストグラム *h* で、指定される値 x、y が属する階級に 1.0 を加える。

(x, y) が属する階級が存在すれば、返り値は 0 になり、エラーがなかったことを示す。(x, y) がヒストグラムの範囲外であれば `GSL_EDOM` を返し、ヒストグラム中の階級値はどこも変わらず、単にそういった値は無視される。これは、巨大なデータセットで各階級の幅が小さいような場合は探索に時間がかかることがあるためであり、どちらの場合もエラー・ハンドラーは呼ばれない。

`int gsl_histogram2d_accumulate` (*gsl_histogram2d * **h***, *double **x***, *double **y***, *double **weight***) [Function]

階級に 1 ではなく *weight* を加えること以外は `gsl_histogram2d_increment` と同じである。

double gsl_histogram2d_get (*const gsl_histogram2d * h*, *size_t i*, *size_t j*)[Function]

ヒストグラム h の (i,j) 番目の階級の値を返す。(i,j) が h の持たない階級を指すような場合はエラー・コード GSL_EDOM でエラー・ハンドラーが呼ばれ、関数は 0 を返す。

int gsl_histogram2d_get_xrange (*const gsl_histogram2d * h*, *size_t i*, *double * xlower*, *double * xupper*) [Function]
int gsl_histogram2d_get_yrange (*const gsl_histogram2d * h*, *size_t j*, *double * ylower*, *double * yupper*) [Function]

ヒストグラム h の x、y 方向でそれぞれ i、j 番目の階級の下限値と上限値を返す。下限値が *xlower* と *ylower* に、上限値が *xupper* と *yupper* にそれぞれ入れられる。下限値はその階級の範囲に含まれるが（ちょうどその下限値を持つ標本はその階級に数えられる）、上限値は含まれない（ちょうどその上限値を持つ標本は、一つ上の階級があればそこに含まれる）。i と j がヒストグラム内で有効であれば 0 を返す。そうでなければエラー・コード GSL_EDOM でエラー・ハンドラーを呼び出す。

double gsl_histogram2d_xmax (*const gsl_histogram2d * h*) [Function]
double gsl_histogram2d_xmin (*const gsl_histogram2d * h*) [Function]
size_t gsl_histogram2d_nx (*const gsl_histogram2d * h*) [Function]
double gsl_histogram2d_ymax (*const gsl_histogram2d * h*) [Function]
double gsl_histogram2d_ymin (*const gsl_histogram2d * h*) [Function]
size_t gsl_histogram2d_ny (*const gsl_histogram2d * h*) [Function]

それぞれ、ヒストグラム h の x 方向および y 方向それぞれでの最上階級の上限値、最下階級の下限値、階級の数を返す。これらを使えば gsl_histogram2d の要素を直接見なくてもすむ。

void gsl_histogram2d_reset (*gsl_histogram2d * h*) [Function]

ヒストグラム h の全ての階級値を 0 にする。

21.17 二次元ヒストグラム中での範囲の検索

ある (x,y) の値をその範囲に含む階級の値を見たり、値を変更するのには以下の関数がある。

int gsl_histogram2d_find (*const gsl_histogram2d * h*, *double x*, *double y*, *size_t * i*, *size_t * j*) [Function]

与えられる座標 (x,y) の値をその範囲に含む階級の番号を探し、i と j に入れる。探索は、階級の幅が等間隔の場合について最適化された二分探索で行われる。(x,y) がヒストグラムの範囲に含まれていればその階級を (i,j) に入れ、GSL_SUCCESS を返す。そうでなければエラー・ハンドラーを 呼び出し、GSL_EDOM を返す。

21.18 二次元ヒストグラムの統計値

double gsl_histogram2d_max_val (*const gsl_histogram2d * h*) [Function]

ヒストグラム h のすべての階級中で最大の度数を返す。

void gsl_histogram2d_max_bin (*const gsl_histogram2d * h*, *size_t * i*, *size_t * j*) [Function]

ヒストグラム h のすべての階級中で最大の度数を持つ階級の添え字 (i, j) を返す。複数の階級が同じ度数の時は、最初に見つかったものを返す (まず $i = 0$ について j を 0 から探索し、次に i を 1 だけ増やして j を探索し . . . を繰り返す順序で探索を行う)。

double gsl_histogram2d_min_val (*const gsl_histogram2d * h*) [Function]

ヒストグラム h のすべての階級中で最小の度数を返す。

void gsl_histogram2d_min_bin (*const gsl_histogram2d * h*, *size_t * i*, *size_t * j*) [Function]

ヒストグラム h のすべての階級中で最小の度数を持つ階級の添え字 (i, j) を返す。複数の階級が同じ度数の時は、最初に見つかったものを返す (まず $i = 0$ について j を 0 から探索し、次に i を 1 だけ増やして j を探索し . . . を繰り返す順序で探索を行う)。

double gsl_histogram2d_xmean (*const gsl_histogram2d * h*) [Function]

ヒストグラムが確率分布にしたがうとして、ヒストグラムにカウントされている x 変数の平均値を返す。度数が負の階級は無視される。

double gsl_histogram2d_ymean (*const gsl_histogram2d * h*) [Function]

ヒストグラムが確率分布にしたがうとして、ヒストグラムにカウントされている y 変数の平均値を返す。度数が負の階級は無視される。

double gsl_histogram2d_xsigma (*const gsl_histogram2d * h*) [Function]

ヒストグラムが確率分布にしたがうとして、ヒストグラムにカウントされている x 変数の標準偏差を返す。度数が負の階級は無視される。

double gsl_histogram2d_ysigma (*const gsl_histogram2d * h*) [Function]

ヒストグラムが確率分布にしたがうとして、ヒストグラムにカウントされている y 変数の標準偏差を返す。度数が負の階級は無視される。

double gsl_histogram2d_cov (*const gsl_histogram2d * h*) [Function]

ヒストグラムが確率分布にしたがうとして、ヒストグラムにカウントされている x と y 変数の共分散を返す。度数が負の階級は無視される。

double gsl_histogram2d_sum (*const gsl_histogram2d * h*) [Function]

全ての階級の度数の和を返す。度数が負の場合も無視されない。

21.19 二次元ヒストグラムの操作

`int gsl_histogram2d_equal_bins_p` (*const gsl_histogram2d * **h1***, *const gsl_histogram2d * **h2***) [Function]

二つのヒストグラムで、全ての階級の範囲が一致するとき 1 、そうでなければ 0 を返す。

`int gsl_histogram2d_add` (*gsl_histogram2d * **h1***, *const gsl_histogram2d * **h2***) [Function]

ヒストグラム *h1* の各階級にヒストグラム *h2* の対応する階級の度数を $h_1'(i,j) = h_1(i,j) + h_2(i,j)$ のようにして加える。二つのヒストグラムの階級の範囲はすべて一致してなければならない。

`int gsl_histogram2d_sub` (*gsl_histogram2d * **h1***, *const gsl_histogram2d * **h2***) [Function]

ヒストグラム *h1* の各階級からヒストグラム *h2* の対応する階級の度数を $h_1'(i,j) = h_1(i,j) - h_2(i,j)$ のようにして引く。二つのヒストグラムの階級の範囲はすべて一致してなければならない。

`int gsl_histogram2d_mul` (*gsl_histogram2d * **h1***, *const gsl_histogram2d * **h2***) [Function]

ヒストグラム *h1* の各階級に、ヒストグラム *h2* の対応する階級の度数を $h_1'(i,j) = h_1(i,j) \times h_2(i,j)$ のようにして掛ける。二つのヒストグラムの階級はすべて範囲が一致してなければならない。

`int gsl_histogram2d_div` (*gsl_histogram2d * **h1***, *const gsl_histogram2d * **h2***) [Function]

ヒストグラム *h1* の各階級を、ヒストグラム *h2* の対応する階級の度数で $h_1'(i,j) = h_1(i,j)/h_2(i,j)$ のようにして割る。二つのヒストグラムの階級はすべて範囲が一致してなければならない。

`int gsl_histogram2d_scale` (*gsl_histogram2d * **h***, *double **scale***) [Function]

ヒストグラム *h* の各階級の度数に $h_1'(i,j) = h_1(i,j) \times scale$ のようにして *scale* を乗ずる。

`int gsl_histogram2d_shift` (*gsl_histogram2d * **h***, *double **offset***) [Function]

ヒストグラム *h* の各階級の度数に $h_1'(i,j) = h_1(i,j) + offset$ のようにして offset を加える。

21.20　二次元ヒストグラムのファイル入出力

GSL では、二次元ヒストグラムをバイナリ・データまたは整形済みテキストとしてファイルに保存、またはファイルから読み込む関数が用意している。

`int gsl_histogram2d_fwrite` (*FILE * **stream***, *const gsl_histogram2d * **h***) [Function]

ヒストグラム *h* の各階級の範囲と度数をバイナリ形式でファイル *stream* に出力する。出力できれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。出力形式は計算機アーキテクチャに依存したバイナリ形式なので、移植性は低い。

`int gsl_histogram2d_fread` (*FILE * **stream***, *gsl_histogram2d * **h***) [Function]

ヒストグラム *h* の各階級の範囲と度数をバイナリ形式でファイル *stream* から読み込む。ヒストグラムのインスタンス *h* はあらかじめ、読み込もうとするデータにあわせた大きさで生成しておかねばならない。その階級数を使って読み込むバイト数が決定される。読み込みが完了すれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。入力データは計算機アーキテクチャに依存したバイナリ形式でなければならない。

`int gsl_histogram2d_fprintf` (*FILE * **stream***, *const gsl_histogram2d * **h***, *const char * **range_format***, *const char * **bin_format***) [Function]

ヒストグラム *h* の各階級の範囲と度数をファイル *stream* に、一行ずつ指定された形式 *range_format* および *bin_format* で出力する。形式は実数に対する `%g`、`%e`、`%f` のいずれかでなければならない。出力できれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。出力形式は以下のように、空白文字で区切られた五列からなる。

```
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)
```

各行は階級の幅と値からなる。ある階級の上限値は、一つ上の階級の下限値でもあるため、出力される行は違うがこれらの値は重複している。しかし行指向のツール (awk, perl, ruby など) での操作の利便性を考えて、こういった形式になっている。

`int gsl_histogram2d_fscanf (`*FILE * **stream***, *gsl_histogram2d * **h**`)` [Function]

ヒストグラム *h* の各階級の範囲と度数をファイル *stream* から読み込む。データの形式は *gsl_histogram_fprintf* で出力される五列からなる形式である。ヒストグラムのインスタンス *h* はあらかじめ、読み込もうとするデータにあわせた階級数で生成しておかねばならない。この階級数を使って読み込む数値の数が決定される。読み込みが完了すれば 0 を、エラーが発生したら `GSL_EFAILED` を返す。

21.21 二次元ヒストグラムからの確率分布事象の発生

一次元ヒストグラムと同様に二次元ヒストグラムでも、事象を数えることは確率分布にしたがう事象を観察することであると考えられる。統計上の誤差はあるが、一つの階級の大きさは事象 (x, y) がその階級の範囲に発生する確率を表していると言える。二次元ヒストグラムでは確率分布を $p(x, y)dxdy$ として表すと、以下のようになる。

$$p(x, y) = n_{ij}/(NA_{ij})$$

n_{ij} は (x, y) を含む階級の事象の数、A_{ij} は階級の面積、N は事象の総合計である。各階級内での事象の発生確率は一様であるとしている。

`gsl_histogram2d_pdf` [Data Type]

`size_t nx, ny`
: x および y 方向での、確率分布関数を近似するための階級の数。

`double * xrange`
: x 方向での各階級の幅。ポインタ *xrange* が指す、要素数 $nx+1$ 個の配列に保持される。

`double * yrange`
: y 方向での各階級の幅。ポインタ *yrange* が指す、要素数 $ny+1$ 個の配列に保持される。

`double * sum`
: ポインタ *sum* が差す $nx \times ny$ 個の要素を持つ配列に保持される累積度数。

この確率分布にしたがった標本を乱数で発生させるための `gsl_histogram2d_pdf` 構造体のインスタンスを以下の関数で生成できる。

`gsl_histogram2d_pdf * gsl_histogram2d_pdf_alloc (`*size_t* **nx**, *size_t* **ny**`)`[Function]

階級数 $nx \times ny$ の二次元の確率分布のインスタンスを生成し、gsl_histogram2d_pdf 構造体インスタンスへのポインタを返す。十分なメモリが確保できなかったときはエラー・コード GSL_ENOMEM でエラー・ハンドラーを呼び出し、NULL を返す。

int gsl_histogram2d_pdf_init (*gsl_histogram2d_pdf * p*, *const gsl_histogram2d * h*) [Function]

ヒストグラム h に基づいて確率分布のインスタンス p を初期化する。h に度数が負の階級があるときは、確率分布関数は負の値を取れないため、エラー・コード GSL_EDOM でエラー・ハンドラーを呼び出す。

void gsl_histogram2d_pdf_free (*gsl_histogram2d_pdf * p*) [Function]

p が指す二次元確率分布関数のインスタンスのメモリをすべて解放する。

int gsl_histogram2d_pdf_sample (*const gsl_histogram2_ pdf * p*, *double r1*, *double r2*, *double * x*, *double * y*) [Function]

0 から 1 の範囲の二つの一様乱数 *r1* と *r2* を使って、二次元の確率分布 p にしたがう標本 s を一つ発生する。

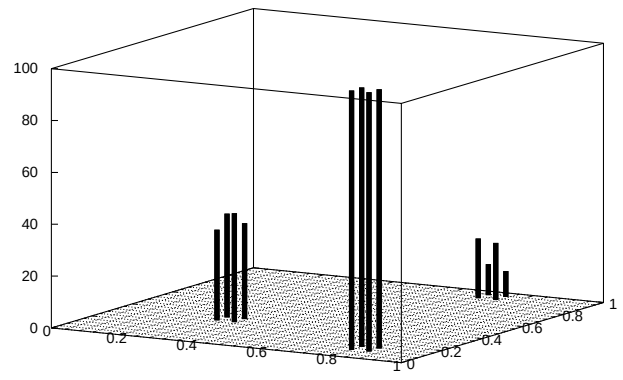
21.22　二次元ヒストグラムのプログラム例

以下のプログラムで二次元ヒストグラムの二つの使い方を示す。まず x と y の範囲がどちらも 0 から 1 の、10×10 の二次元ヒストグラムを生成する。そして点 (0.3,0.3) に高さ 1、(0.8,0.1) に高さ 5、(0.7,0.9) に高さ 0.5 の標本を発生する。3 つの標本を加えたこのヒストグラムから、1000 個のランダムな標本を生成して出力する。

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_histogram2d.h>

int main (void)
{
    int i;
    double x, y;
    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, 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);

    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 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;
}
```

出力されたランダムな標本 (をカウントしたヒストグラム) のプロットを以下に示す。解像度を上げる (ヒストグラムの階級幅を細かくする) ことで、標本生成に使ったヒストグラムの形が見える。そのヒストグラムの階級内では一様の確率で事象を発生しているが、実際に発生した標本の分布がばらついているのも見える。



第22章　N項組

この章では N 項組 (N-tuple、N 組、タプルとも)、つまり「事象に関連づけられ、順序づけられた値の集合」を操作する関数について説明する。GSL の N 項組は常にファイルと関連づけられ、ファイル入出力を基本として操作される。その値はあらゆる組合せで取り出すことができ、選択関数 (selection function) を使ってヒストグラムにカウントすること (booking) ができる。

N 項組のデータの形式は、構造体として定義する必要がある。その構造体を単位としてデータのファイル入出力が行われる。データの入出力は、メモリの実体を持つその構造体のインスタンスを一つだけ使って、逐次的に行われる。ファイルから読み込む場合であれば、i) 読ファイルからみ込んで構造体インスタンスに入れる、ii) そのデータを利用するか他に退避する、そしてまた i) 構造体インスタンスに次のデータが読み込まれる、. . . というループを N 項組のデータの数だけ繰り返すことになる。

N 項組と、それに対する選択関数および数値化関数 (value function) を与えることで、N 項組からヒストグラムを生成できる。選択関数は、一組の N 項組 (これが一つの事象 (event) になる) について、その各要素 (associated value) から、その N 項組データが解析対象かどうかを判定する。数値化関数は、選択関数が解析対象と判定する一組の N 項組データをヒストグラムにカウントするときに、その一つのデータに対して、度数をいくつ増やすかを計算する。

N 項組の関数はすべてヘッダファイル 'gsl_ntuple.h' に宣言されている。

22.1　N 項組構造体

N 項組は gsl_ntuple 構造体として保持、操作される。この構造体には N 項組を保存するファイルへのポインタ、操作したいある一つの N 項組データ (N-tuple row) へのポインタ、および利用者が定義する N 項組データ構造体の大きさを保持する。

```
typedef struct {
    FILE * file;
    void * ntuple_data;
    size_t size;
} gsl_ntuple;
```

22.2　N 項組ファイルの作成

gsl_ntuple * gs_ntuple_create (*char * filename*, *void * ntuple_data*, *size_t size*)　[Function]

大きさ *size* の N 項組を保存するファイル *filename* を書き込み専用モードで生成し、新たに生成した N 項組構造体へのポインタを返す。そのファイル名のファイルが既に存在していた場合は、そのファイルは大きさが 0 になり、上書きされる。引数に書き込みたい N 項組データを保持する変数へのポインタ *ntuple_data* を指定する必要がある。このポインタは N 項組データのファイルへの出力に使われる。

22.3 すでにある N 項組ファイルのオープン

`gsl_ntuple * gsl_ntuple_open` (*char * **filename***, *void * **ntuple_data***, *size_t **size***) [Function]

すでに存在する N 項組のファイル *filename* を読み込みモードでオープンし、N 項組構造体へのポインタを返す。ファイル中の N 項組の大きさは *size* でなければならない。読み込んだデータを格納する N 項組変数へのポインタ *ntuple_data* を指定しなければならない。このポインタは N 項組データのファイルからの読み込みに使われる。

22.4 N 項組の書き込み

`int gsl_ntuple_write` (*gsl_ntuple * **ntuple***) [Function]

大きさが *ntuple*`->size` の一組の N 項組のデータ *ntuple*`->ntuple_data` を、ファイル *ntuple*`->file` に書き込む。

`int gsl_ntuple_bookdata` (*gsl_ntuple * **ntuple***) [Function]

関数名以外は `gsl_ntuple_write` と同じである。

22.5 N 項組の読み込み

`int gsl_ntuple_read` (*gsl_ntuple * **ntuple***) [Function]

N 項組のファイルの現在の行から一組の N 項組データを読み込み、*ntuple*`->data` にその一組のデータを格納する。

22.6 N 項組ファイルのクローズ

`int gsl_ntuple_close` (*gsl_ntuple * **ntuple***) [Function]

N 項組 *ntuple* のファイルをクローズし、確保していたメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

22.7 N 項組からのヒストグラムの生成

N 項組のデータセットから、関数 gsl_ntuple_project を使ってヒストグラムを作る方法がいくつかあるが、いずれにせよ、ある N 項組データに対して、それをヒストグラムにカウントするかどうかを決める関数 (選択関数) と、選び出されたデータからスカラー値を計算する関数 (数値化関数) の二つを与える必要がある。あらかじめ想定する階級を持つヒストグラムのインスタンスを生成しておき、N 項組データを数値化した値がどの階級に入るかをカウントすることでヒストグラムができる。選択関数、数値化関数のどちらも、第一引数で一組の N 項組データを、第二引数でその他のパラメータを受け取るよう定義する。

選択関数は、与えられた一つの N 項組データを無視するか、ヒストグラムにカウントするかを決める関数として、以下の構造体で定義する。

```
typedef struct {
    int (* function) (void * ntuple_data, void * params);
    void * params;
} gsl_ntuple_select_fn;
```

構造体のメンバ *function* は、与えられた N 項組データがヒストグラムにカウントする対象なら非零の値を返すように定義する。

スカラー値を計算する関数は、選択関数で選び出された N 項組データから値を計算して返す関数として、以下の構造体で定義する。

```
typedef struct {
    double (* function) (void * ntuple_data, void * params);
    void * params;
} gsl_ntuple_value_fn;
```

ここでは構造体のメンバ *function* は、対象としている N 項組の行から、ヒストグラムに加えるべき値を返すように定義する。

`int gsl_ntuple_project (`*gsl_histogram* * **`h`**, *gsl_ntuple* * **`ntuple`**, *gsl_ntuple_value_fn* * **`value_func`**, *gsl_ntuple_select_fn* * **`select_func`**`)` [Function]

関数 *value_func* および *select_func* を使って、与えられた N 項組データ *ntuple* をヒストグラム *h* にカウントする。関数 *select_func* が非零を返す N 項組データに対して、関数 *value_func* を使って値が計算され、それがヒストグラムに加えられる。*select_func* が 0 を返すデータは無視される。最初は新しい階級がヒストグラムに加えられ、階級がすでにあればそこに加える。

22.8 例

以下に、N 項組で大規模なデータを扱う二つのプログラムを例示する。まず最初のプログラムでは、三つの属性値 (x, y, z) を持つ 10,000 個の事象をシミュレーションで発生する。これらは分散

1 の正規分布乱数で生成され、N 項組ファイル ‘test.dat’ に出力される (生成されるのはバイナリ・ファイルである)。

```
#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;
    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;
}
```

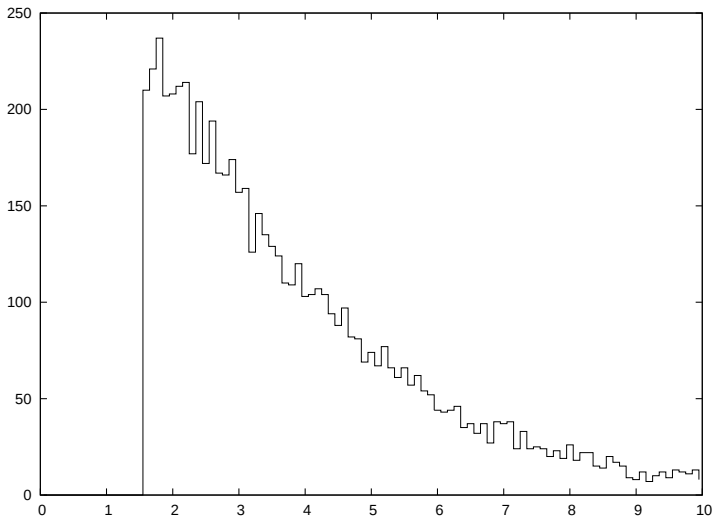
次のプログラムではファイル ‘test.dat’ の N 項組データを解析する。解析は、数値化関数 val_func で各事象の平方 $E^2 = x^2 + y^2 + z^2$ を計算し、選択関数 sel_func で値が 1.5 以上のものだけを選

び出すことで行う。選び出された事象を、E^2 値を使ってヒストグラムにする。

```
#include <math.h>
#include <gsl/gsl_ntuple.h>
#include <gsl/gsl_histogram.h>

struct data {
    double x;
    double y;
    double z;
};

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;
    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)
{
    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;
}
```

以下に、選択された事象のヒストグラムを示す。下限値以上のみがカウントされているのがわかる。



22.9 参考文献

CERNLIB (CERN Program Library) の PAW および HBOOK パッケージに参考になる記述が同梱されている (オンラインで自由に入手できる)。

- CERNLIB: `http://cernlib.web.cern.ch/cernlib/`
- PAW: `http://paw.web.cern.ch/paw/`
- HBOOK: `http://wwwasdoc.web.cern.ch/wwwasdoc/Welcome.html`

第23章　モンテカルロ積分

この章では多次元モンテカルロ積分 (multidimensional Monte Carlo integration) を行うルーチンについて説明する。これには初期の素朴なモンテカルロ法に加え、VEGAS と MISER というそれぞれ加重サンプリング (importance sampling) と層化抽出法 (stratified sampling) を行う適応的手法 (adaptive method) が含まれている。以下で表される多次元定積分 (multidimensional difinite integral) の値を近似的に求める。

$$I = \int_{x_l}^{x_u} dx \int_{y_l}^{y_u} dy \ldots f(f, y, \ldots)$$

超立方体 (hypercube) $((x_l, x_u), (y_l, y_u), \ldots)$ 領域内で、関数が有限回数だけ計算される。GSL が提供するルーチンでは近似誤差の統計的推定も行われるが、誤差の上限値を厳密に示すものではなく、単なる目安である。ランダム・サンプリングでは、超立方体内のすべての点を網羅し関数の特徴を把握できるわけではなく、誤差は過小評価されることがあるためである。

関数は 'gsl_monte_plain.h'、'gsl_monte_miser.h' および 'gsl_monte_vegas.h' の三つのヘッダファイルに分けて宣言されている。

23.1　利用法

モンテカルロ積分のルーチンはアルゴリズムによって三種類が実装されているが、どれも同じ形式で使うことができるようになっている。制御変数 (control variable) と作業領域を確保するルーチン、制御変数を初期化するルーチン、積分を行うルーチン、その後にメモリを解放するルーチンである。

積分を行う各関数には、使用する乱数生成ルーチンを指定しなければならない。そして各積分関数は積分の推定値とその標準偏差を返す。得られる推定値の精度は、被積分関数の評価回数を指定することで決まる。ある程度高い精度の結果を得たい場合は、要求精度が得られるまで、数値積分を複数回行ってその結果を平均すればよい。

モンテカルロ積分ルーチン中で、乱数で生成される点は常に積分範囲の内側である。したがって境界上に特異点があっても、それは避けられる。

モンテカルロ積分とそのための型はヘッダファイル 'gsl_monte.h' で宣言されている。

`gsl_monte_function` [Data Type]

> この型は、被積分関数を定義するための汎用の型であり、被積分関数のパラメータの配列へのポインタを内部に保持する。

`double (* f) (double * x, size_t dim, void * params)`
: 引数が x、パラメータが $params$ の時の関数値 $f(x, params)$ を返す。x は大きさ dim の配列で、関数を評価する座標を与える。

`size_t dim`
: x の次元数。

`void * params`
: 関数を定義するパラメータへのポインタ。

以下に二次元の二次形式の関数の例を示す。

$$f(x, y) = ax^2 + bxy + cy^2$$

以下のコードで、数値積分関数に渡す被積分関数 `gsl_monte_function F` を定義できる。ここで $a = 3,\ b = 2,\ c = 1$ とする。

```
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) { 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;
```

関数 $f(x)$ は以下のマクロで評価される。

```
#define GSL_MONTE_FN_EVAL(F,x)
(*((F)->f))(x,(F)->dim,(F)->params)
```

23.2　シンプルなモンテカルロ積分

シンプルなモンテカルロ積分 (plain Monte Carlo integration) アルゴリズムでは、積分範囲内でランダムに点を発生して積分値を評価し誤差を見積もる。このアルゴリズムでランダムに分布する

N 個の点 x_i から計算される積分の推定値 $E(f;N)$ は、V を積分範囲の体積とするとき、以下のように表される ($\langle f\rangle$ は f の算術平均)。

$$E(f;N) = V\langle f\rangle = \frac{V}{N}\sum_i^N f(x_i)$$

この推定積分値の誤差 $\sigma(E;N)$ は平均値の標本分散から以下で計算される。

$$\sigma^2(E;N) = \frac{V}{N}\sum_i^N (f(x_i) - \langle f\rangle)^2$$

$\mathrm{Var}(f)$ を積分範囲内ので関数値の真の分散とすると、N が大きくなると σ^2 は $\mathrm{Var}(f)/N$ に近づき、小さくなっていく。推定誤差自体は $\sigma(f)/\sqrt{N}$ の形で小さくなる。つまり誤差は $1/\sqrt{N}$ にしたがって小さくなるため、誤差を 1/10 にするためにはサンプル点を 100 倍に増やさなければならないことになる。この節の関数はヘッダファイル 'gsl_monte_plain.h' で宣言されている。

`gsl_monte_plain_state * gsl_monte_plain_alloc` (*size_t* ***dim***) [Function]

次元数 *dim* のモンテカルロ積分のための作業領域を確保し、初期化する。

`int gsl_monte_plain_init` (*gsl_monte_plain state* * ***s***) [Function]

すでに確保されている作業領域を初期化する。別の積分で使われた作業領域を再利用するときに使う。

`int gsl_monte_plain_integrate` (*gsl_monte_function* * ***f***, *double* * ***xl***, *double* * ***xu***, *size_t* ***dim***, *size_t* ***calls***, *gsl_rng* * ***r***, *gsl_monte_plain state* * ***s***, *double* * ***result***, *double* * ***abserr***) [Function]

シンプルなモンテカルロ積分アルゴリズムを使って、*dim* 次元の超立体内を範囲として積分値を計算する。大きさ *dim* の各配列 *xl* と *xu* の要素でそれぞれ積分範囲の下界 (lower limit)、上界 (upper limit) を指定する。積分値は乱数発生器 *r* で生成する点を使った *calls* 回の関数評価で計算される。あらかじめ作業領域 *s* を確保して指定する。推定積分値は *result* に、推定絶対誤差は *abserr* に入れられる。

`void gsl_monte_plain_free` (*gsl_monte_plain state* * ***s***) [Function]

積分のための作業領域 *s* に割り当てられているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

23.3 MISER

プレス (William. H. Press) とファラー (Glennys R. Farrar) による MISER アルゴリズムは再帰的層化抽出法 (recursive stratified sampling) を使った方法である。これは、積分範囲内で関数値の分散がもっとも大きな領域に集中して点を取ることで、積分の誤差を抑えようとする方法である。

層化抽出法ではまず積分範囲を二等分し、その二つの領域 a と b についてモンテカルロ積分 $E_a(f)$ と $E_b(f)$、分散 $\sigma_a^2(f)$、$\sigma_b^2(f)$ を計算し、推定積分値の平均 $E(f) = \frac{1}{2}(E_a(f) + E_b(f))$ の分散 $\mathrm{Var}(f)$ を以下の式で計算する。

$$\mathrm{Var}(f) = \frac{\sigma_a^2(f)}{4N_a} + \frac{\sigma_b^2(f)}{4N_b}$$

この分散は、各領域でのサンプリング点数 N_a、N_b を以下のようにすることで最小化できる。

$$\frac{N_a}{N_a + N_b} = \frac{\sigma_a}{\sigma_a + \sigma_b}$$

つまり各領域での関数値の標準偏差に比例するようにサンプリング点の数をとると、推定誤差を最小化できる。

MISER 法は繰り返し計算 (iteration) である。その各ステップで、被積分関数の定義域空間のどれか一つの座標軸に直交する平面で積分領域を二等分する。定義域の d 本の各座標軸について、分割した場合の二つの小領域の分散を計算し、その分散が最小になるものをそのステップで選ぶ。各ステップにおける被積分関数の計算回数は、全ステップでのトータルの回数から一定の割合で決められ、その回数による被積分関数の計算で各小領域内での被積分関数の分散が計算される。もっともよい分割による二つの各領域について、これを再帰的に繰り返す。指定された被積分関数の評価回数に達するまでの残りの関数評価点は、上の N_a と N_b の式に基づいて各領域に割り当てていく。再帰は指定された深さまで続けられ、最も深いレベルでは、各領域内でシンプルなモンテカルロ積分アルゴリズムが行われる。そのときの各領域での推定積分値と推定誤差はさかのぼってまとめられ、全体の結果として推定積分値と推定誤差が得られる。

この節で説明する関数はヘッダファイル 'gsl_monte_miser.h' で宣言されている。

`gsl_monte_miser_state * gsl_monte_miser_alloc` (*size_t* ***dim***) [Function]

次元数 *dim* のモンテカルロ積分のための作業領域を確保し、初期化する。作業領域は積分の進行を管理するために使われる。

`int gsl_monte_miser_init` (*gsl_monte_miser state* * ***s***) [Function]

すでに確保されている作業領域を初期化する。別の積分で使われた作業領域を再利用するときに使う。

`int gsl_monte_miser_integrate` (*gsl_monte_function* * ***f***, *double* * ***xl***, *double* * ***xu***, *size_t* ***dim***, *size_t* ***calls***, *gsl_rng* * ***r***, *gsl_monte_miser state* * ***s***, *double* * ***result***, *double* * ***abser***) [Function]

大きさ *dim* の各配列 *xl* と *xu* の要素でそれぞれ下界 (lower limit)、上界 (upper limit) を指定される *dim* 次元の超立体内を範囲として、関数 f の積分値を MISER 法で計算する。関数評価は *calls* 回行われ、サンプリング点は乱数発生器 r によって決定される。あらかじめ作業領域を確保して s として指定する。積分結果は *result* に、絶対誤差の推定値は *abserr* に入れられる。

`void gsl_monte_miser_free` (*gsl_monte_miser state* * ***s***) [Function]

作業領域 s で確保されているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

MISER 法にはアルゴリズムの動作を調整するためのパラメータがいくつか用意されている。`gsl_monte_miser_state` 構造体の以下のパラメータを操作できる

`double estimate_frac` [Variable]

分散を推定するために使われる、再帰の各ステップでの関数評価の回数の、積分全体の総評価回数に対する割合を指定する。デフォルト値は 0.1 である。

`size_t min_calls` [Variable]

一つの領域の分散を推定するために必要とする関数評価の最小回数を指定する。分散の推定のための *estimate_frac* による関数評価の回数が *min_calls* 未満のときは、*min_calls* 回の評価が行われる。これにより、分散の推定は毎回、ある程度の精度を保つことができる。デフォルト値は `16 * dim` である。

`size_t min_calls_per_bisection` [Variable]

最良の分割を決める 1 回のステップで行われる関数評価の最小回数を指定する。ある深さまで進んだステップで、総評価回数からの残り評価回数が `min_calls_per_bisection` より少ない場合は、そのステップでの積分領域に対してシンプルなモンテカルロ積分アルゴリズムを適用し、再帰はそこで終了する。デフォルト値は `32 * min_calls` である。

`double alpha` [Variable]

分割された二つの各探索領域の標本分散から、そのステップでの全体の分散をどのように計算するのかを指定する。分割の方法は、二つの小領域の分散が最小となるように選ばれるため、再帰を繰り返す過程で、分散は $1/N$ よりも早く小さくなる。MISER 法ではこれを取り入れるため、全体の分散を以下のように係数 α で調整できるようにしてある。

$$\mathrm{Var}(f) = \frac{\sigma_a}{N_a^\alpha} + \frac{\sigma_b}{N_b^\alpha}$$

MISER 法の原著論文では数値実験の結果として $\alpha = 2$ を推奨しており、GSL でもそれをデフォルト値として採用している。

`double dither` [Variable]

領域の二分割が被積分関数の積分範囲の中央部に集中するのを避けるため、大きさ *dither* のランダムな変化を領域分割の際に与える。デフォルト値は 0 であり、領域分割に乱数は使われない。必要な場合は、0.1 程度の値が一般的である (二等分 = 0.5 倍とするときの 0.5 に対する変更量として指定する)。

23.4 VEGAS

リページ (G. Peter Lepage) の VEGAS 法は加重サンプリング (importance sampling) が基本であり、被積分関数 $|f|$ の値に比例した確率でサンプリング点を発生させていく。そのため、積分値にもっとも寄与の大きな領域に集中してサンプリング点を発生することになる。

一般的に、関数 f のモンテカルロ積分を関数 g で記述される確率分布にしたがって生成したサンプリング点で行うとして、以下の推定積分値 $E_g(f;N)$ と、その分散 V_g を考える。

$$\begin{aligned} E_g(f;N) &= E(f/g;N) \\ V_g(f;N) &= V(f/g;N) \end{aligned}$$

確率分布を $g = |f|/I(|f|)$ となるようにとれれば ($I(|f|)$ は関数 $|f|$ の真の積分値)、分散 $V_g(f;N)$ は 0 になり、推定積分値における誤差も 0 になる。現実的には確率分布関数 g をそうはとれない ($I(|f|)$ が不明だから数値積分を行っているはずである) が、加重サンプリングは真の分布関数をできるだけうまく近似するように行う。

VEGAS 法では、関数 f のヒストグラムを作りながら積分領域全体をカバーするというステップを何度も繰り返して、分布関数を近似していく。あるステップで作られたヒストグラムを、次のステップでサンプリングするための分布関数として利用する。これを繰り返すことで分布関数を次第に正確にしていく。また、定義域空間の次元数 d が大きくなるときにヒストグラムの階級数が K^d のオーダーで爆発的に増加するのを避けるため、確率分布関数 g は $g(x_1, x_2, \ldots) = g_1(x_1)g_2(x_2)\ldots$ として関数の積に分割できる形式で定義される。これにより階級数を $k \times d$ に抑えることができるが、これは被積分関数を座標軸に投影した形から確率密度関数の形が決まることを意味する。VEGAS 法の効率はこの仮定がどの程度正しいかに依存し、投影したときに被積分関数のピークが重なって一方が隠れてしまうようなことがないように、うまく局在化しているのが理想である。被積分関数が複数の関数の積の形で近似できる場合は VEGAS 法で効率よく積分できる。

VEGAS 法には他にもいくつか工夫点があり、層化抽出法と加重サンプリングを組み合わせて用いている。積分範囲はいくつかの「直方体 (box)」に分けられ、各直方体には決まった数のサンプリング点が割り当てられる (最終的には 2 点になる)。各直方体ではヒストグラムの階級数に端数がでることもある。直方体一つあたりの階級数が 2 よりも小さくなったとき、VEGAS では加重サンプリングから分散低減法 (variance reduction) に切り替わる。

`gsl_monte_vegas_state * gsl_monte_vegas_alloc` (*size_t* ***dim***) [Function]

次元数 *dim* のモンテカルロ積分のための作業領域を確保し、初期化する。作業領域は積分の進行を管理するために使われる。

`int gsl_monte_vegas_init` (*gsl_monte_vegas_state* * ***s***) [Function]

すでに確保されている作業領域を初期化する。別の積分で使われた作業領域を再利用するときに使う。

`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***) [Function]

大きさ *dim* の各配列 *xl* と *xu* の要素でそれぞれ下界 (lower limit)、上界 (upper limit) を指定される *dim* 次元の超立体内を範囲として、関数 *f* の積分値を VEGAS モンテカルロ法で計算する。関数評価は *calls* 回行われ、サンプリング点は乱数発生器 *r* によって決定される。あらかじめ作業領域を確保して *s* として指定する。積分結果は *result* に、絶対誤差の推定値は *abserr* に入れられる。積分値とその推定誤差は独立した各サンプリングの重み付き平均 (weighted average) から計算される。重み付き平均の一自由度当たりのカイ二乗値が構造体の要素 *s*->chisq に入れられ、重み付き平均の信頼性が高ければその値は 1 に近い値になる。

`void gsl_monte_vegas_free (`*gsl_monte_vegas_state* `* s)` [Function]

作業領域 *s* で確保されているメモリを解放する (ポインタが指す構造体のメンバーにアクセスしようとするため、NULL ポインタを引数として渡してはならない)。

VEGAS 法では、後述の `iterations` パラメータにしたがって、積分値の計算を複数回 (GSL のデフォルトでは 5 回)、内部でそれぞれ独立して行い、その重み付き平均を最終的な積分の推定値として返す。その各回の積分値を得る計算の中では、積分範囲内でランダム・サンプリングが行われ、各点で被積分関数値が計算されるが、サンプリングの行われる場所によっては、たとえば関数値が定数値となるような領域などで被積分関数値の分散が 0 になることがある。そういった場合には重み付き平均の計算が破綻するため、VEGAS の元の FORTRAN による実装 (Lepage, 1980) では、誤差が 0 の場合にそれを非常に小さな値 (`1e-30` など) で置き換えている。しかし GSL ではこういった恣意的な定数を使わないようにするため、以下のようにしてそれまでの推定積分値による重みの平均を使うか、または無視するかを決める。

推定積分値の分散が **0** になったが、その前までは計算できていた場合
: 前のステップまでの重みの平均を、現在の推定積分値の重みにする。

推定積分値の分散が非零になったが、それまではずっと **0** だった場合
: それまでの推定積分値は破棄され、非零になった計算を 1 回目として、また最初からやりなおす (デフォルトでは 5 回の計算の 1 回目終了時点に戻る)。

推定積分値の分散が **0** になったが、前回のステップでも **0** だった場合
: 推定積分値は (重みなしの) 算術平均として計算され、分散は計算されない。

VEGAS 法では `gsl_monte_vegas_state` 構造体の以下のメンバー変数で詳細な調整ができる。

`double result` [Variable]
`double sigma` [Variable]

積分計算の、その時点での最後の繰り返し計算での結果 *result* とその誤差 *sigma*。

`double chisq` [Variable]

積分値の重み付き推定値の、一自由度あたりのカイ二乗値。*chisq* の値は 1 に近い値となる。*chisq* が 1 と大きく異なっている場合は、繰り返し計算の各ステップで整合性の取れない値が得られていることを示す。そういった場合は重み付き誤差の値は過小に評価されているため、積分値の信頼性を上げるために積分計算の繰り返しを、さらに続ける必要がある。

`double alpha` [Variable]

ヒストグラムの階級の再設定のされにくさ。一般に 1 から 2 の間で、既定値は 1.5 である。0 にすると階級は固定になる。

`size_t iterations` [Variable]

ルーチンが 1 回呼ばれたときに内部で行われる積分の繰り返し計算の回数。デフォルトでは 5 回。

`int stage` [Variable]

これを設定することで計算のレベルを決めることができる。通常は `stage = 0` として、領域を等分割し重み付き平均を 0 から始める設定とする。`stage = 1` として VEGAS を呼び出すと、重み付き平均は 0 から始めるが前回の呼び出しの時の領域の分割の仕方を使う。これにより、比較的小さな個数のサンプリング点で領域分割を行っておき、その後 `stage = 1` として呼び出すことで、最適な領域分割で多数の点を使う計算を行うことができる。`stage = 2` とすると、領域分割に加えて重み付き平均の初期値として前回のものを使うが、関数評価の回数に応じて分割領域中のヒストグラムの階級数を増やす (または減らす) ことはできる。`stage = 3` ではなにも設定せずに繰り返し計算を始める。つまり前回の呼び出しの続きとして計算を行うことができる。

`int mode` [Variable]

設定できる値は、

- `GSL_VEGAS_MODE_IMPORTANCE`
- `GSL_VEGAS_MODE_STRATIFIED`
- `GSL_VEGAS_MODE_IMPORTANCE_ONLY`

のいずれかである。VEGAS のサンプリングの方法を、加重サンプリングか層化抽出法のどちらを使うか、またはアルゴリズムの実行中に自動で選べるようにしておくかを指定する。次元数が小さいときは VEGAS では層化抽出法を使う (正確には各分割領域中での階級数が 2 よりも 小さくなったときに層化抽出法を使う)。

`int verbose` [Variable]
`FILE * ostream` [Variable]

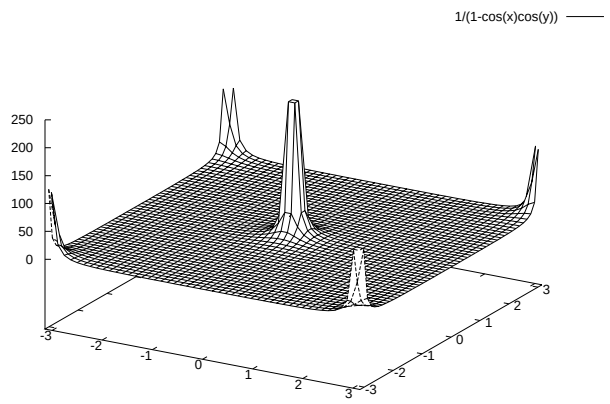
VEGAS が出力する情報のレベルを設定する。出力は全て *ostream* に行われる (デフォルトでは `stdout`)。*verbose* のデフォルト値は `-1` で、何も出力されない。*verbose* を `0` にすると重み付き平均と積分結果が、`1` にするとそれに加えて領域分割の座標が表示される。`2` では繰り返し計算の各回での階級の再設定の様子が表示される。

23.5　例

三次元の以下の積分を計算するモンテカルロ法のプログラムを例示する。

$$I = \int_{-\pi}^{+\pi} \frac{dk_x}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_y}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_z}{2\pi} \frac{1}{1 - \cos(k_x)\cos(k_y)\cos(k_z)}$$

図に、変数を一つ減らした場合の被積分関数をプロットしている。図には正確には表現されていないが、原点と $k_{x,y,z} = \pm\pi$ が divid by zero な特異点である。



この積分の解析的な値は $I = \Gamma(1/4)^4/(4\pi^3) = 1.393203929685676859\ldots$ である。この積分では、三次元の立方体中でのランダム・ウォークが原点で費やした時間の平均値が表示される。

プログラムを簡単にするため、積分範囲を $(0,0,0)$ から (π,π,π) にし、得られた積分値を 8 倍して最終的な積分結果としている。積分範囲の中央部では積分値はあまり変化せず、頂点 $(0,0,0)$、$(0,\pi,\pi)$、$(\pi,0,\pi)$、$(\pi,\pi,0)$ では特異点となっている。モンテカルロ積分ルーチンでは積分範囲の内側のみにサンプリング点を取るため、特異点を避けるための工夫をする必要はない。

```
#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>

/*  以下の積分を、(-pi,-pi,-pi) to (+pi, +pi, +pi) で計算する。
          I = int (dx dy dz)/(2pi)^3 1/(1-cos(x)cos(y)cos(z))
    解析的に解くと Gamma(1/4)^4/(4 pi^3) になる。これは以下の文献による。
    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) */

/*  計算を単純にするために、範囲 (0,0,0) -> (pi,pi,pi) で数値積分を行い、
    その結果を 8 倍して解とする。 */

double exact = 1.393203929685676859184246260325;
```

```
double g (double *k, size_t dim, 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 = %.1g 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;
    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);
            printf("result = % .6f sigma = % .6f chisq/dof = %.1f\n",
                   res, err, s->chisq);
        } while (fabs (s->chisq - 1.0) > 0.5);

        display_results ("vegas final", res, err);
        gsl_monte_vegas_free (s);
    }

    gsl_rng_free(r);
    return 0;
}
```

被積分関数の評価回数を 500,000 回としたとき、シンプルなモンテカルロ積分では相対誤差は約 0.6% である。推定誤差 sigma は実際の誤差とおおよそ同じくらいで、積分値と真の値との差は、標準偏差と同程度である。

```
plain ==================
result =  1.385867
sigma  =  0.007938
exact  =  1.393204
error  = -0.007337 = 0.9 sigma
```

MISER 法では誤差を約 1/2 に減らすことができ、誤差の値もより正確に見積もることができる。

```
miser ==================
result =  1.390656
sigma  =  0.003743
exact  =  1.393204
error  = -0.002548 = 0.7 sigma
```

VEGAS 法を使う場合、プログラム内ではまず領域分割の「準備」として、100,000 回の被積分関数評価による積分を行う。これに続いて 100,000 回の被積分関数評価を行う繰り返し計算を 5 回行う。5 回の繰り返し計算による一自由度あたりのカイ二乗値が 1 に近いかどうか毎回確認され、収束していない場合に計算が続けられる。その場合推定値は最初の実行で得られた値とされる。

```
vegas warm-up ==================
result =  1.386925
sigma  =  0.002651
exact  =  1.393204
error  = -0.006278 = 2 sigma
converging...
result = 1.392957 sigma =  0.000452 chisq/dof = 1.1
vegas final ================
result =  1.392957
sigma  =  0.000452
exact  =  1.393204
error  = -0.000247 = 0.5 sigma
```

chisq の値が 1 と大きく異なる場合、積分結果が悪くて誤差が過小評価されていることを示す。VEGAS による積分結果 (被積分関数の評価回数が同程度) は他の二種類の方法に比べて非常に精度が高い。

23.6　参考文献

MISER 法は以下の論文に解説されている。

- William H. Press, Glennys R. Farrar, "Recursive Stratified Sampling for Multidimensional Monte Carlo Integration", *Computers in Physics*, **4**, pp. 190–195 (1990).

VEGAS は以下の論文に説明されている。

- G. P. Lepage, "A New Algorithm for Adaptive Multidimensional Integration", *Journal of Computational Physics*, **27**, pp. 192–203 (1978).
- G. P. Lepage, "VEGAS: An Adaptive Multi-dimensional Integration Program", *CNSL-80/447* (Cornell University preprint, 1980).
  http://www.slac.stanford.edu/spires/find/hep/www?r=CLNS-80/447

William H. Press、Saul A. Teukolsky、William T. Vetterling、Brian P. Flannery による *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, (Cambridge University Press, ISBN 978-0521880688, 2007, 初版の邦訳: *Numerical Recipes in C* 日本語版, 技術評論社, ISBN 978-4-87408-560-1, 1993) の著者の一人が MIZER の原著者の一人である。この本の第 7.9 節 "Adaptive & Recursive Monte Carlo Methods" には MIZER と VEGAS の両方が詳しく説明されている。邦訳は初版に基づいているため、残念ながらその節はない。

第24章　シミュレーティド・アニーリング

最適化問題 (関数の最小化/最大化問題) において、関数の形 (プロファイル、探索空間の構造) がよくわかってないときや連続でない時などには、ヤコビアン行列 (Jacobian derivative matrix) を必要とするニュートン法 (Newton's method) のような方法は使えないが、確率的探索法 (stochastic search、あるいは発見的探索法 heuristic search) が利用できることがある。特に、巡回セールスマン問題のような組み合わせ最適化 (combinatorial optimization) によく用いられる。

確率的探索では、実数関数であるエネルギー関数 (またはコスト関数) が最小になる探索空間中の点を探すことである。GSL で実装しているシミュレーティド・アニーリング (simulated annealing、焼き鈍し法) は、局所解に捕まることを避けながら優良な解を探すことができる最適化手法であり、基本的には温度を次第に下げながら (だんだんと探索を局所的にしながら) 乱数探索 (random walk) で関数値が小さくなる点を探す。その過程で、乱数で決めた探索点を採用するかどうかはボルツマン分布 (Boltzmann distribution) が与える確率で決める。

この章の関数はヘッダファイル '`gsl_siman.h`' で宣言されている。

24.1　シミュレーティド・アニーリング

シミュレーティド・アニーリングは、関数が定義されている探索空間内で乱数探索を行い、エネルギー E (目的関数の値) が小さくなる点を探す。乱数探索において、乱数で決められた次の探索点を採用するかどうかは、$E_{i+1} > E_i$ であれば以下のボルツマン分布にしたがった確率で決定される。

$$p = e^{-(E_{i+1}-E_i)/(kT)}$$

ここで $E_{i+1} \leq E_i$ であれば $p = 1$ である。

次の探索点でのエネルギーが小さくなる場合には、そこに探索を進める。しかしエネルギーが大きくなる場合にもその探索点に進む可能性も残しておく。その可能性は温度 T に比例し、現在の点と進もうとする次の点のエネルギーの差 $E_{i+1} - E_i$ に反比例した値によって決めるようにする。

探索を始めるときには温度 T を高い値に設定し、乱数探索をその温度で行う。その後温度を、たとえば $T \to T/\mu_T$ のような冷却スケジュールにしたがって、ゆっくりと下げていく。このとき μ_T は 1 よりも少し大きな値とする。

エネルギーが高い点に進む確率は小さいが、それにより局所解から抜け出すことができる。

24.2　シミュレーティド・アニーリング関数

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*)　[Function]

この関数は、与えられた空間内でシミュレーティド・アニーリングによる探索を行う。目的関数は二つの関数 *Ef* と *distance* を渡すことで指定される。乱数発生器 *r* と *take_step* を使って探索点を生成していく。

探索開始点を *x0_p* で与える。このルーチンでは探索点に固定長または可変長の二種類のモードがある。固定長モードでは、探索点は大きさ *element_size* の一つのメモリブロックに保持される。内部では標準ライブラリ関数の `malloc`、`memcpy`、`free` を使って探索点の生成や複製、破棄が行われる。関数へのポインタ *copyfunc*、*copy_constructor*、*destructor* は固定長モードでは NULL ポインタにしておく。可変長モードでは探索点の生成、複製、破棄を行う関数 *copyfunc*、*copy_constructor*、*destructor* を指定する。可変長モードでは *element_size* は 0 にしておく。

params 構造体 (後述) に、シミュレーティド・アニーリングを制御するための温度スケジュールその他のパラメータを指定する。この関数は終了時に、探索中に見つかった最も良い解を **x0_p* に入れて返す。アニーリングが成功しているときには、この値が探索空間中にある最適解をよく近似する点になっているはずである。

関数へのポインタ *print_position* が NULL でない場合は、以下の形式で探索の進行を示す情報と指定する関数 *print_position* 自身の出力が `stdout` に出力される。

```
#-iter  #-evals temperature  position  energy  best_energy
```

print_position が NULL の場合はなにも出力されない。

シミュレーティド・アニーリングのルーチンを使うためには、探索条件、探索空間、エネルギー関数を定義するために、ユーザーがいくつかの関数を指定する必要がある。それらの関数は以下の型で定義されなければならない。

gsl_siman_Efunc_t　[Data Type]

点 *xp* でのエネルギー関数値を返す関数。

```
double (*gsl_siman_Efunc_t) (void *xp)
```

gsl_siman_step_t　[Data Type]

乱数発生器 *r* を使って探索点を更新する関数。更新による探索点の移動する距離が *step_size* 以下になるように行われる。

```
void (*gsl_siman_step_t) (const gsl_rng *r, void *xp, double step_size)
```

gsl_siman_metric_t　[Data Type]

二点 *xp* および *yp* の間の距離を返す関数。

```
double (*gsl_siman_metric_t) (void *xp, void *yp)
```

gsl_siman_print_t [Data Type]

探索点 *xp* を表示する関数。

```
void (*gsl_siman_print_t) (void *xp)
```

gsl_siman_copy_t [Data Type]

探索点 *source* を *dest* にコピーする関数。

```
void (*gsl_siman_copy_t) (void *source, void *dest)
```

gsl_siman_copy_construct_t [Data Type]

新しい探索点を生成し、探索点 *xp* をそこに複製する関数。

```
void * (*gsl_siman_copy_construct_t) (void *xp)
```

gsl_siman_destroy_t [Data Type]

探索点 *xp* を破棄してメモリを解放する関数。

```
void (*gsl_siman_destroy_t) (void *xp)
```

gsl_siman_params_t [Data Type]

これは *gsl_siman_solve* の動作を制御するためのパラメータである。この構造体のメンバー変数が、エネルギー関数、探索点を更新する関数、探索開始点以外の、探索を制御するために必要となる情報のすべてである。

`int n_tries`
: 1 回の探索点の更新のために探す点の数。

`int iters_fixed_T`
: 温度を変えずに行う繰り返しの回数。

`double step_size`
: 乱数探索での探索の最大幅。

`double k, t_initial, mu_t, t_min`
: ボルツマン分布と冷却スケジュールのパラメータ。

24.3 例

GSL で用意しているシミュレーティド・アニーリング・パッケージは、多態性 (polymorphism) を C 言語で実装しようとしているため、未だ完成度はあまり高くない。以下に若干の変更で実際の問題にそのまま使うことのできるサンプル・プログラムを示す。

24.3.1　簡単な例

最初のサンプルは、一次元デカルト座標系の空間内の減衰 sin 関数をエネルギー関数とする例である。これには多数の局所解があるが最適解は 1.0 と 1.5 の間にある一つだけである。探索開始点は 15.5 とした。最適解から離れていて、その間にいくつかの局所解がある。

```
#include <math.h>
#include <stdlib.h>
#include <gsl/gsl_siman.h>

/* シミュレーティド・アニーリングを実行するためのパラメータの設定 */
#define N_TRIES 200             /* 次の探索点を決めるための周辺探索の回数 */
#define ITERS_FIXED_T 1000      /* 各温度で何回まで繰り返すか */
#define STEP_SIZE 1.0           /* 乱数探索の最大探索距離 */
#define K 1.0                   /* ボルツマン定数 */
#define T_INITIAL 0.008         /* 温度の初期値 */
#define MU_T 1.003              /* 温度低下係数 */
#define T_MIN 2.0e-6

gsl_siman_params_t params
    = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,K, T_INITIAL, MU_T, T_MIN};

/* 一次元でのテストのための関数 */
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(int argc, char *argv[])
{
    const gsl_rng_type * T;
    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;
}
```

プログラム siman_test を以下のように実行して得られる二つのプロットを示す。

```
$ ./siman_test | awk '!/^#/{print $1, $4}'
    | graph -y 1.34 1.4 -W0 -X generation -Y position
    | plot -Tps > siman-test.eps
$ ./siman_test | awk '!/^#/{print $1, $5}'
    | graph -y -0.88 -0.83 -W0 -X generation -Y energy
    | plot -Tps > siman-energy.eps
```

24.3.2 巡回セールスマン問題

TSP (Traveling Salesman Problem、巡回セールスマン問題) は古典的な組み合わせ最適化問題である。ここでは非常に単純な、アメリカ南部の 12 の都市の例を挙げる。人によってはこの例は、

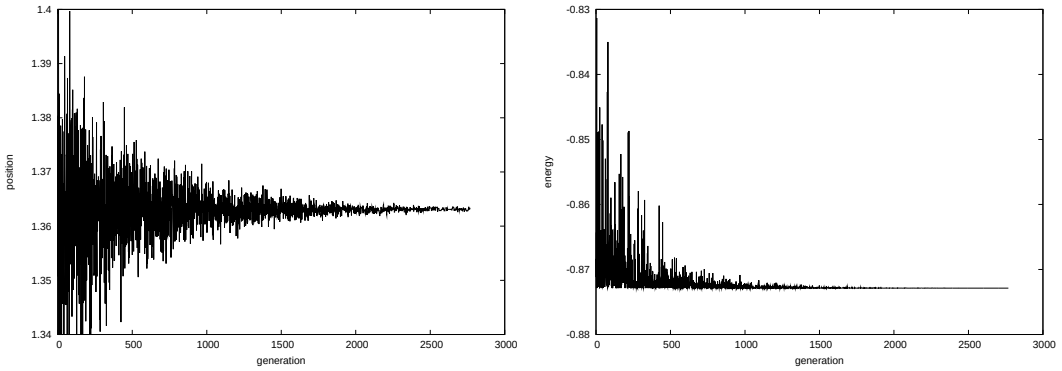


図 24.1: シミュレーティド・アニーリングの実行例。温度が高い時 (左の方) では変動が大きく、温度が低くなるにつれ収束していく様子がわかる。

都市間の距離が自動車で行ける距離ではないような気がするかもしれない。そういう人には空飛ぶセールスマン問題 (Flying Salesman Problem) とでもいった方がいいかもしれない。地球の表面は球面であると仮定し、ジオイド面上の距離は使わない。

`gsl_siman_solve` ルーチンにより、距離が 3490.62km の巡回経路を解として得る。得られた経路でのスタート地点をスタート地点とする場合の、全てのあり得る経路を全数探索するとその解を確認できる。GSL のソースコードに同梱されている '`siman/siman_tsp.c`' が完全なソースコードであるが、ここには、それを `siman_tsp` という名前にコンパイルし、以下のようにして得られるプロットを示す。

```
$ ./siman_tsp > tsp.output
$ grep -v "^#" tsp.output
 | awk '{print $1, $NF}'
 | graph -y 3300 6500 -W0 -X generation -Y distance
         -L "TSP - 12 southwest cities"
 | plot -Tps > 12-cities.eps
$ grep inital_city_coord tsp.output
 | awk '{print $2, $3}'
 | graph -X "longitude (- mean west)" -Y latitude
         -L "TSP - initial-order" -f0.03 -S 1 0.1
 | plot -Tps > initial-route.eps
$ grep final_city_coord tsp.output
 | awk '{print $2, $3}'
 | graph -X "longitude (- mean west)" -Y latitude
         -L "TSP - final-order" -f0.03 -S 1 0.1
 | plot -Tps > final-route.eps
```

世代 (新しく温度の値が設定されてからの計算した点の数) に対するコスト関数 (エネルギー関

数) のプロットは、この問題では以下のようになる。

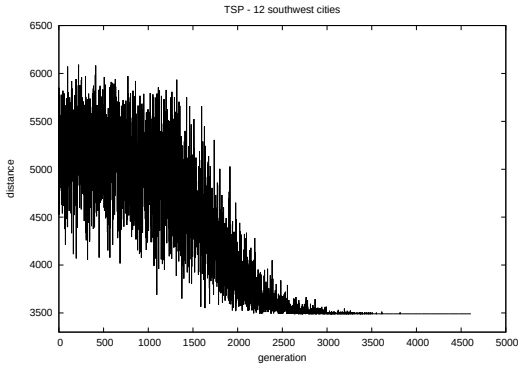


図 24.2: 南西部の 12 都市に関する空飛ぶセールスマン問題にシミュレーティド・アニーリングを適用したときの、最適化の進行に伴うエネルギー関数値の変化。

以下に示す出力は、探索初期の段階で得られる経路である。経度が負の値になっているのは、都市が西半球にあるためである。

```
# 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
```

最適解は以下のようになる。

```
# final coordinates of cities (longitude and latitude)
###final_city_coord: -105.95 35.68 Santa Fe
###final_city_coord: -106.28 35.89 Los Alamos
###final_city_coord: -106.62 35.12 Albuquerque
###final_city_coord: -107.84 35.15 Grants
###final_city_coord: -107.87 37.29 Durango
###final_city_coord: -108.58 37.35 Cortez
```

```
###final_city_coord: -108.74 35.52 Gallup
###final_city_coord: -112.07 33.54 Phoenix
###final_city_coord: -106.76 32.34 Las Cruces
###final_city_coord: -96.77 32.79 Dallas
###final_city_coord: -103.2 34.41 Clovis
###final_city_coord: -105.92 35.77 Tesuque
###final_city_coord: -105.95 35.68 Santa Fe
```

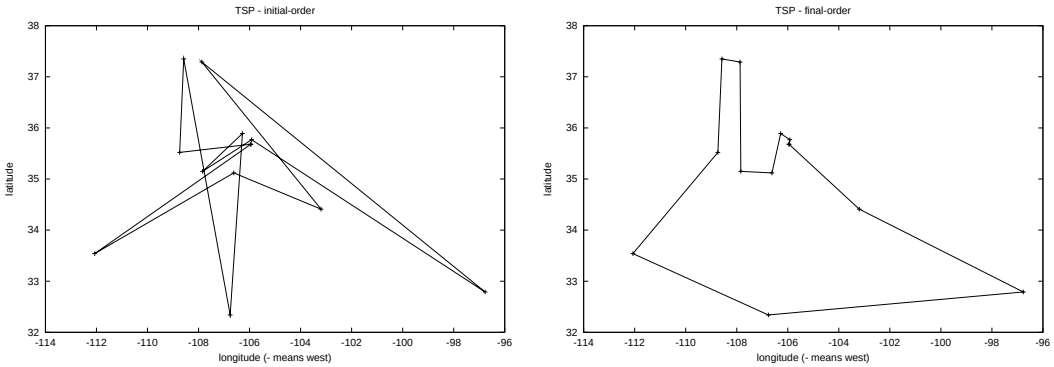


図 24.3: 南西部の 12 都市に関する空飛ぶセールスマン問題の、初期の解と最終的に得られる解 (最適解)。

24.4　参考文献

より詳しくは、以下の文献が参考になる。

- Colin R. Reeves (ed.), *Modern Heuristic Techniques for Combinatorial Problems*, McGraw-Hill, ISBN 0-07-709239-2 (1995).

第25章　常微分方程式

この章では常微分方程式 (Ordinary Differential Equations, ODE) の初期値問題 (initial value problem) を解く関数について説明する。GSL には、ルンゲ・クッタ法 (Runge-Kutta method) やブリアシュ・シュテア法 (Bulirsch-Stoer method) などの様々な低レベルのルーチン (low level function、アルゴリズムとして抽象化レベルが低いという意味) と、ステップ幅を自動で計算、調整する高レベルのルーチンを用意している。計算の内部の過程をチェックしながら、これらを適宜組み合わせてプログラムを作ることができる。

この節の関数は 'gsl_odeiv.h' で宣言されている。

25.1　微分方程式系の定義

ここで説明するルーチンは、一般的な n 次の連立一階微分方程式 (n-dimentional first-order system)

$$\frac{dy_i(t)}{dt} = f_i(t, y_1(t), \ldots, y_n(t)) \qquad (i = 1, \ldots, n)$$

を t について解くためのものである。ステップを進める関数は導関数のベクトル f_i とヤコビアン行列 $J_{ij} = \partial f_i(t, y(t))/\partial y_j$ に基づいて計算を行う。微分方程式系は gsl_odeiv_system 構造体で定義する。

gsl_odeiv_system　[Data Type]

解こうとする常微分方程式系を、この構造体を使って定義する。導関数とヤコビアンの定義には、任意の個数のパラメータが使える。

`int (* function) (double t, const double y[], double dydt[], void * params)`
引数が (t, y) で同関数定義に含まれるパラメータが *params* のとき、各導関数の値 $f_i(t, y, params)$ を計算し、配列 *dydt* に入れる。計算がすべてうまく行ったときには GSL_SUCCESS を返し、そうでなければ、エラーを示す値を返すようにする。

`int (* jacobian) (double t, const double y[], double * dfdy, double dfdt[], void * params);`
連立方程式の次数が *dimension* で与えられるとき、偏微分係数ベクトルの要素 $\partial f_i(t, y, params)/\partial t$ を配列 *dfdt* に入れ、ヤコビアン行列 J_{ij} を `J(i,j) = dfdy[i * dimension + j]` となるように行ごとに並べて *dfdy* に入れる。計算がすべてうまく行ったときには GSL_SUCCESS を返し、そうでなければ、エラーを示す値を返すようにする。

簡単なアルゴリズムではヤコビアン行列を使わないものもあるので、必ずヤコビアン行列を指定する必要があるわけではない (そういったアルゴリズムを使うときは、この構造体の jacobiana 要素に NULL ポインタを代入する)。しかし、精度が高く効率のよいアルゴリズムはヤコビアン行列を使うので、場合によってはアルゴリズムを切り替えて使うことを考えると、できるならヤコビアン行列を与えておいた方がよい。

`size_t dimension;` 微分方程式の次数。

`void * params` 微分方程式の定義に使うパラメータ。

25.2　ステップを進める関数

もっとも低レベルな関数はステップ関数 (stepping function) で、時刻 t から $t+h$ へ固定幅のステップ h だけ進んだ点での解を計算し、その点での局所的な誤差を計算する。

`gsl_odeiv_step * gsl_odeiv_step_alloc` (*const gsl_odeiv_step_type * T*, *size_t dim*) [Function]

アルゴリズム T を使って次数 *dim* の方程式を解くステップ関数のインスタンスを生成して、そのインスタンスへのポインタを返す。

`int gsl_odeiv_step_reset` (*gsl_odeiv_step * s*) [Function]

ステップ関数のインスタンスを再初期化する。s を使い続けても解が進められないような場合に使う。

`void gsl_odeiv_step_free` (*gsl_odeiv_step * s*) [Function]

ステップ関数のインスタンス s に割り当てられたメモリを解放する。

`const char * gsl_odeiv_step_name` (*const gsl_odeiv_step * s*) [Function]

ステップ関数の名前文字列へのポインタを返す。たとえば以下の文

```
printf ("step method is '%s'\n",gsl_odeiv_step_name (s));
```

は `step method is 'rkf45'` のように出力する。

`unsigned int gsl_odeiv_step_order` (*const gsl_odeiv_step * s*) [Function]

直前のステップでのステップ関数の次数 (order) を返す。ステップが可変の場合には次数が変化しうる。

`int gsl_odeiv_step_apply` (*gsl_odeiv_step * s*, *double t*, *double h*, *double y[]*, *double yerr[]*, *const double dydt_in[]*, *double dydt_out[]*, *const gsl_odeiv_system * dydt*) [Function]

ステップ関数のインスタンスが持っているステップ関数 s を $dydt$ で定義される微分方程式に適用し、時刻 t、その時点での解 y から $t + h$ までステップ幅 h だけ解を進める。y は新しい解で上書きされ、各要素についての推定絶対誤差が $yerr$ に入れられる。引数 $dydt\_in$ には時刻 t での方程式の導関数値を入力として入れておくか、NULL ポインタにしておく。NULL にした場合には導関数値はルーチンの内部で計算されるが、すでに導関数値が計算されている場合には、それを再利用する方がよい。$dydt\_out$ が NULL でなければ、時刻 $t+h$ における導関数値が計算され、そこに入れられる。

$dydt$ に設定されている、微分係数やヤコビアンを計算する関数がこの関数の内部で呼ばれるが、それらが `GSL_SUCCESS` 以外の、なにかエラーを示す値を返したときは、ステップは進められない。y の値はステップを進めようとする前の値に戻され (`gsl_odeiv_step_apply` を呼んだときの値のままになる)、返ってきたエラー・コードをそのまま返す。ステップ幅 h は、エラーが生じたときの値になる。エラーの後、たとえばステップ幅を $h/10$ などのより小さな値にしてこの関数を呼んだ場合、エラーの原因となった点 (特異点などであろう) に、より近いところまで進めるかもしれない。導関数やヤコビアンが返すエラー・コードは、$gsl\_odeiv\_step\_apply$ 自体からのものと見分けられるように、GSL 標準のものと異なるように定義しておくことが必要である。

ステップを進めるアルゴリズムには、以下のものが利用できる。

`gsl_odeiv_step_rk2` [Step Type]

埋め込み型 RK23 公式 (三次のルンゲ・クッタ法に二次の公式が埋め込まれている)。

`gsl_odeiv_step_rk4` [Step Type]

普通の四次のルンゲ・クッタ法。ステップ幅二等分して誤差を見積もる。次のルンゲ・クッタ・フェールベルク法 (Runge-Kutta-Fehlberg method) では、もっと効率のよい見積もりを行う。

`gsl_odeiv_step_rkf45` [Step Type]

埋め込み型 RKF45 公式 (ルンゲ・クッタ・フェールベルク法、Embedded Runge-Kutta-Fehlberg(4, 5))。汎用できる、よい方法である。

`gsl_odeiv_step_rkck` [Step Type]

埋め込み型 RKCK45 公式 (キャシュ (Jeff R. Cash) とカープ (Alan H. Karp) のルンゲ・クッタ法、Embedded Runge-Kutta Cash-Karp (4, 5))。

`gsl_odeiv_step_rk8pd` [Step Type]

埋め込み型 RKPD89 公式 (プリンス (Peter J. Prince) とドルマンド (John R. Dormand) のルンゲ・クッタ法 Embedded Runge-Kutta Prince-Dormand (8, 9))。

`gsl_odeiv_step_rk2imp` [Step Type]

ガウス型公式による二次の陰的ルンゲ・クッタ法 (Implicit 2nd order Runge-Kutta at Gaussian points)。

`gsl_odeiv_step_rk4imp` [Step Type]

ガウス型公式による四次の陰的ルンゲ・クッタ法 (Implicit 4th order Runge-Kutta at Gaussian points)。

`gsl_odeiv_step_bsimp` [Step Type]

バダーとドイフルハルト (G. Bader and Peter Deuflhard) の陰的ブリアシュ・シュテア法 (Implicit Bulirsh-Stoer method)。この方法はヤコビアン行列を使う。

`gsl_odeiv_step_gear1` [Step Type]

$M = 1$ の陰的ギア法。

`gsl_odeiv_step_gear2` [Step Type]

$M = 2$ の陰的ギア法。

25.3 ステップ幅の適応制御

ステップ幅を調整する関数は、ステップ関数が計算したステップ幅を取ったときに解の値の変化と推定誤差がどうなるかを確認し、指定された誤差範囲に収まるような最適なステップ幅を決定する。

`gsl_odeiv_control * gsl_odeiv_control_standard_new` (*double* **`eps_abs`**, *double* **`eps_rel`**, *double* `a_y`, *double* `a_dydt`) [Function]

標準ステップ調整法のインスタンスを生成する。相対誤差 *eps_abs* と相対誤差 *eps_rel*、微分方程式のその時点での解 $y(t)$ と導関数 $y'(t)$ に対する係数 *a_y* と *a_dydt* の 4 つのパラメータを使って、最適ステップ幅の探索を行う。

ステップ幅の調整ではまず、各従属変数について要求される精度 (誤差レベル) D_i を計算する。

$$D_i = \epsilon_{abs} + \epsilon_{rel}(a_y|y_i| + a_{dydt}h|y'_i|)$$

そしてこれを実際に生じた誤差 $E_i = |yerr_i|$ と比較する。実際の誤差 E が要求精度 D の 1.1 倍を上回るような要素があった場合は、以下で与えられる係数を乗じてステップ幅を縮小する。

$$h_{new} = h_{old} \times S \times (E/D)^{-1/q}$$

ここで q は求解アルゴリズムの次数 (たとえば埋め込み型 RK45 公式では $q = 4$)、S は余裕を見るための係数で 0.9 である。比 E/D は E_i/D_i のうち最大のものを取る。

もし E_i/D_i の最大値に対して実際の誤差 E が要求精度 D の半分よりも小さければ、誤差を要求精度に納める範囲でステップ幅の拡大を図る。

$$h_{new} = h_{old} \times S \times (E/D)^{-1/(1+q)}$$

下のステップ調整法はどれもこの方法に基づいている。ステップ幅の拡大、縮小の係数は、1/5 から 5 の範囲に制限され、値があばれることのないようになっている。

`gsl_odeiv_control * gsl_odeiv_control_y_new` (*double* **`eps_abs`**, *double* **`eps_rel`**) [Function]

各ステップにおいて解の値 $y_i(t)$ に対する誤差を絶対誤差 *eps_abs* および相対誤差 *eps_rel* の範囲内に抑えるようなステップ調整法のインスタンスを生成する。これは標準ステップ調整法のインスタンスを a_y=1、a_dydt=0 で生成するのと同じである。

`gsl_odeiv_control * gsl_odeiv_control_yp_new` (*double* **`eps_abs`**, *double* **`eps_rel`**) [Function]

各ステップにおいて解の微分値 $y_i'(t)$ に対する誤差を絶対誤差 *eps_abs* および相対誤差 *eps_rel* の範囲内に抑えるようなステップ調整法のインスタンスを生成する。これは標準ステップ調整法のインスタンスを a_y=0、a_dydt=1 で生成するのと同じである。

`gsl_odeiv_control * gsl_odeiv_control_scaled_new` (*double* **`eps_abs`**, *double* **`eps_rel`**, *double* **`a_y`**, *double* **`a_dydt`**, *const double* **`scale_abs[]`**, *size_t* **`dim`**) [Function]

`gsl_odeiv_control_standard_new` と同じステップ調整法のインスタンスを生成する。各要素に対する許容絶対誤差を *scale_abs* でスケーリングする。ここでは D_i を以下の式で計算する。

$$D_i = \epsilon_{abs} s_i + \epsilon_{rel} * (a_y |y_i| + a_{dydt} h |y_i'|)$$

ここで s_i は配列 *scale_abs* の i 番目の要素である。MATLAB の ODE 機能でも同じ方法で誤差を制御している。

`gsl_odeiv_control * gsl_odeiv_control_alloc` (*const gsl_odeiv_control_type * T*) [Function]

方法 T を使うステップ調整法のインスタンスを生成し、そのインスタンスへのポインタを返す。この関数はステップ調整法を新しく定義した場合にのみ使う。ほとんどの場合は上の標準の方法で十分である。

`int gsl_odeiv_control_init` (*gsl_odeiv_control * c*, *double* **`eps_abs`**, *double* **`eps_rel`**, *double* **`a_y`**, *double* **`a_dydt`**) [Function]

ステップ調整法のインスタンス c をパラメータ *eps_abs* (許容絶対誤差)、*eps_rel* (許容相対誤差)、a_y (y のスケーリング係数)、a_dydt (微分値のスケーリング係数) を使うよう初期化する。

`void gsl_odeiv_control_free` (*gsl_odeiv_control * c*) [Function]

ステップ調整法のインスタンス c に割り当てられているメモリを解放する。

`int gsl_odeiv_control_hadjust` (*gsl_odeiv_control * c*, *gsl_odeiv_step * s*, *const double y0[]*, *const double yerr[]*, *const double dydt[]*, *double * h*) [Function]

ステップ調整法のインスタンス c および現時点での y、$yerr$、$dydt$ の値を使ってステップ幅 h を調整する。この関数の内部で求解アルゴリズムの次数 (q) を得るためにステップ関数 $step$ を与える。y の値の誤差 $yerr$ が大きすぎるときは h を縮小し GSL_ODEIV_HADJ_DEC を返す。誤差が十分に小さい場合は h を拡大し GSL_ODEIV_HADJ_INC を返す。ステップ幅を変えなかったときは GSL_ODEIV_HADJ_NIL を返す。利用者が指定した要求精度を現時点で保つようなステップ幅の最大値を求めたいときに、この関数を使う。

const char * gsl_odeiv_control_name (*const gsl_odeiv_control * c*) [Function]

ステップ調整法のインスタンスが保持している調整法の名前文字列へのポインタを返す。例えば以下の文は、control method is 'standard' と出力する。

```
printf("control method is '%s'\n", gsl_odeiv_control_name (c));
```

25.4 求解

問題を解く上でもっとも高レベルな関数は、ステップ関数とステップ調整関数を組合せ、区間 (t_0, t_1) 全体にわたって求解を進めていく、求解 (evolution) を行う関数である。ステップ調整関数からステップ幅を縮小するようにシグナルを送ってきた場合、求解を行う関数は現在のステップから戻って、計算で求められたより短いステップで求解を試みる。これを適切なステップ幅が見つかるまで繰り返す。

gsl_odeiv_evolve * gsl_odeiv_evolve_alloc (*size_t dim*) [Function]

dim 次の方程式のための発展関数のインスタンスを生成し、そのインスタンスへのポインタを返す。

int gsl_odeiv_evolve_apply (*gsl_odeiv_evolve * e*, *gsl_odeiv_control * con*, *gsl_odeiv_step * step*, *const gsl_odeiv_system * dydt*, *double * t*, *double t1*, *double * h*, *double y[]*) [Function]

微分方程式系 $(e, dydt)$ の求解を、時刻 t、解 y からステップ関数 $step$ を使って進める。進んだあとの時刻と解は、引数 t と y に上書きされる。ステップ幅の初期値は h をとるが、ステップ調整関数 c により、必要があれば誤差が十分に小さくなるような値に調整される。最適なステップ幅を得るために $step$ が複数回呼ばれることもある。各ステップにおける (局所的な) 誤差の推定値が配列 e->yerr[] に入れられる。ステップ幅が変更された場合は、その値が引数 h に入れられる。ステップ幅は最大でも、ステップを適用したあとの時刻が $t1$ を超えない範囲に調整される。最後のステップでは、適用後の時刻がちょうど $t1$ になるようにステップ幅が決められる。

$dydt$ に設定されている、微分係数やヤコビアンを計算する関数がこの関数の内部で呼ばれるが、それらが GSL_SUCCESS 以外の、なにかエラーを示す値を返したときは、ステップは進められない。y の値はステップを進めようとする前の値に戻され

(gsl_odeiv_step_apply を呼んだときの値のままになる)、返ってきたエラー・コードをそのまま返す。導関数やヤコビアンが返すエラー・コードは、*gsl_odeiv_step_apply* 自体からのものと見分けられるように、GSL 標準のものと異なるように定義しておくことが必要である。

`int gsl_odeiv_evolve_reset (`*gsl_odeiv_evolve* `* e)` [Function]

発展関数のインスタンス e を再設定する。そのまま e を使っても求解が進まないような場合に使う。

`void gsl_odeiv_evolve_free (`*gsl_odeiv_evolve* `* e)` [Function]

発展関数のインスタンス e に割り当てられたメモリを解放する。

微分係数が非連続的に変化するような点があらかじめ分かっているときは、計算したい区間をその点の両側の小区間に分けて、それぞれで時間発展を行うようにするとよい。たとえば力学系のモデルで、外力の大きさが時刻 $t_a, t_b, t_c, \ldots$ で切り替わるようなとき、区間全体 (t_0, t_1) を設定して一度に全体を解くよりも、(t_0, t_a), $(t_a, t_b), \ldots,$ (t_c, t_1) にわけた方が効率がいい。

微分係数が非連続になる点を含む区間で高精度を要求すると、その非連続になる点の近くでステップ幅が非常に小さくなる (そしてほとんど機械イプシロン (machine epsilon、または機械精度 machine precision) にまでなってしまう)。こういった問題では、以下のようにしてステップ幅に下限値 hmin を設定した方がよい。

```
while (t < t1) {
    gsl_odeiv_evolve_apply(e, c, s, &sys, &t, t1, &h, y);
    if (h < hmin) { h = hmin; }
}
```

関数 gsl_odeiv_evolve_apply が返す h の大きさはあくまで参考であり、必要に応じて確認、変更するようにすべきである。

25.5 例

以下に二次の非線形系であるファン・デル・ポルの方程式 (Van der Pol oscillator equation) を解くプログラムを例示する。

$$x''(t) + \mu x'(t)(x(t)^2 - 1) + x(t) = 0$$

この系は速度を $y = x'(t)$ と置くことで一次の連立常微分方程式系に変換することができ、これによりこの章で説明したルーチンで解ける形式にすることができる。

$$\begin{aligned} x' &= y \\ y' &= -x + \mu y(1 - x^2) \end{aligned}$$

プログラムではまず導関数とヤコビアンを定義する。

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv.h>

int func (double t, const double y[], double f[], void *params)
{
    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)
{
    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)
{
    const gsl_odeiv_step_type * T = gsl_odeiv_step_rk8pd;
    gsl_odeiv_step    * s = gsl_odeiv_step_alloc(T, 2);
    gsl_odeiv_control * c = gsl_odeiv_control_y_new(1e-6, 0.0);
    gsl_odeiv_evolve  * e = gsl_odeiv_evolve_alloc(2);
    double mu = 10;
    gsl_odeiv_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) {
```

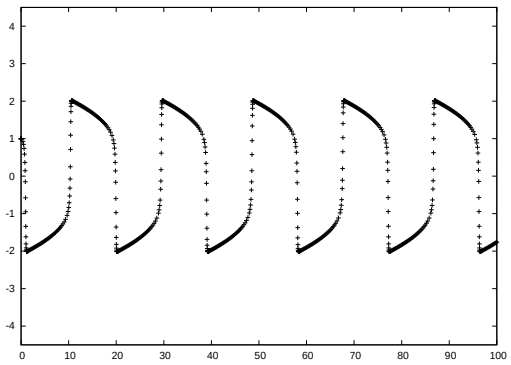


図 25.1: プリンス・ドルマンドによる 8 次のルンゲ・クッタ法を使って計算したファン・デル・ポル方程式の数値解。

```
            int status = gsl_odeiv_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_odeiv_evolve_free(e);
        gsl_odeiv_control_free(c);
        gsl_odeiv_step_free(s);
        return 0;
    }
```

パラメータが複数あるような微分方程式系を計算したい場合は、構造体へのポインタを、*params* に代入して引数に渡すとよい。

時間発展を追う繰り返し部では、時刻 $t = 0$、初期値 $(y, y') = (1, 0)$ からスタートして $t = 100$ まで解を計算していく。ステップ幅 h はステップ調整関数により、y に対する許容絶対誤差が 10^{-6} に収まるように、自動的に調整される。

ステップ調整関数による可変時刻ではなく、一定間隔での原関数値を得るには、繰り返し部を逐次的に点を進めるように変更すればよい。時刻 $t = 0, 1, 2, \ldots, 100$ で解の値を得るには、プログラムの `while` ループのところを以下の `for` ループに変更する。

```
    for (i = 1; i <= 100; i++) {
        double ti = i * t1 / 100.0;
        while (t < ti) gsl_odeiv_evolve_apply(e, c, s, &sys, &t, ti, &h, y);
        printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }
```

`ti` の値の選び方によって、積分範囲を変えることができる。ここでは前述のプログラムと同じになるようにしている。

特に工夫をしない、単にステップ関数のみを用いる数値積分を行うこともできる。四次のルンゲ・クッタ法 rk4 をステップ幅を 0.01 に固定して行うには以下のようにする。

```
int main (void)
{
    const gsl_odeiv_step_type * T = gsl_odeiv_step_rk4;
    gsl_odeiv_step * s = gsl_odeiv_step_alloc (T, 2);
    double mu = 10;
    gsl_odeiv_system sys = {func, jac, 2, &mu}
    double t = 0.0, t1 = 100.0;
    double h = 1e-2;
    double y[2] = { 1.0, 0.0 }, y_err[2];
    double dydt_in[2], dydt_out[2];

    /* 系を定義するパラメータで dydt_in を初期化する */
    GSL_ODEIV_FN_EVAL(&sys, t, y, dydt_in);
    while (t < t1) {
        int status = gsl_odeiv_step_apply(s, t, h, y, y_err,
                                          dydt_in, dydt_out, &sys);
        if (status != GSL_SUCCESS) break;
        dydt_in[0] = dydt_out[0];
        dydt_in[1] = dydt_out[1];
        t += h;
        printf("%.5e %.5e %.5e\n", t, y[0], y[1]);
    }
    gsl_odeiv_step_free(s);
    return 0;
}
```

導関数について、最初のステップを計算する前に時刻 t を $t = 0$ で初期化する必要がある。前のステップの出力である計算された微分係数 *dydt_out* を、次のステップでは入力 *dydt_in* として扱い、ステップの計算に使う。

25.6　参考文献

基本的なルンゲ・クッタ法の公式が、以下の本にいろいろと紹介されている。

- M. Abramowitz, I. Stegun (eds.), *Handbook of Mathematical Functions*, Section 25.5, National Bureau of Standards, U.S. (1964).

陰的ブリアシュ・シュテア法 bsimp は以下の論文にある

- G. Bader, P. Deuflhard, “A Semi-Implicit Mid-Point Rule for Stiff Systems of Ordinary Differential Equations.”, *Numerische Mathematik*, **41**(3), pp. 373–398 (1983).

第26章　補間

この章では、補間 (interpolation) を行う関数について説明する。GSL には三次スプライン (cubic spline) や秋間スプライン (Akima splsine) などのいくつかの補間法が用意されている。これらの補間法は、再コンパイルしなくても実行時に切り替えることができる。境界条件には、通常の条件と両端で周期的になる条件の両方を使うことができる。また補間関数の微分値と積分を計算する関数も用意されている。

この章の関数はヘッダファイル 'gsl_interp.h' および 'gsl_spline.h' で宣言されている。

26.1　はじめに

データ点 $(x_1, y_1) \ldots (x_n, y_n)$ が与えられるとき、GSL で用意しているルーチンは、$y(x_i) = y_i$ となる連続な補間関数 $y(x)$ を計算する。補間関数は各区間内においては連続だが、その両端では用いる補間の種類により異なる。

26.2　補間を行う関数

与えられるデータセットを補間する関数は gsl_interp インスタンスに格納される。このインスタンスは以下の関数で生成、初期化される。

gsl_interp * gsl_interp_alloc (*const gsl_interp_type * T*, *size_t size*) [Function]

点数が *size* のデータに対して補間法 T を用いる補間インスタンスを生成し、そのインスタンスへのポインタを返す。

int gsl_interp_init (*gsl_interp * interp*, *const double xa[]*, *const double ya[]*, *size_t size*) [Function]

補間インスタンス *interp* を、データ *xa*、*ya* (*xa* と *ya* はそれぞれ要素数 *size* の配列) を用いるように初期化する。補間インスタンス (*gsl_interp*) 内にはデータ配列 *xa* および *ya* は保持されず、データから計算された結果のみが保持される。データ配列 *xa* は x の値の昇順に整列しておく必要がある。*ya* にはそういった条件はない。

この関数の中で補間関数を求める計算が行われる。

void gsl_interp_free (*gsl_interp * interp*) [Function]

補間インスタンス *interp* に割り当てられているメモリを解放する。

26.3 補間法

このライブラリでは 6 種類の補間法を用意している。

`gsl_interp_linear` [Interpolation Type]

線形補間 (linear interpolation)。この方法は、与えるデータ以上のメモリを必要としない。

`gsl_interp_polynomial` [Interpolation Type]

多項式補間 (polynomial interpolation)。この方法で得られる補間関数は振動しやすく、データの性質がいいのに振動することもあるため、データ点数が少ないときに適用するのがよい。多項式の項数はデータ点数と同じになる。

`gsl_interp_cspline` [Interpolation Type]

自然な境界の三次スプライン (自然スプライン、cubic spline with natural boundary conditions)。得られる補間関数は各区間ごとに定義される三次関数であり、各区間の境界 (与えられるデータ点) において導関数値と二次導関数値が一致し、最初及び最後のデータ点で二次導関数値が 0 になるように作られる。

`gsl_interp_cspline_periodic` [Interpolation Type]

周期的境界の三次スプライン (周期的スプライン、cubic spline with periodic boundary conditions)。得られる補間関数は各区間ごとに定義される三次関数であり、最初及び最後の点を含む、与えられるすべてのデータ点において導関数値と二次導関数値が一致するように作られる。最後のデータ点の値 (y の値) は、最初の点と同じでなければならない。そうでない場合は、求められる周期的スプラインは両境界において連続ではなくなってしまう。

`gsl_interp_akima` [Interpolation Type]

自然な境界条件での「角張った (non-rounded)」秋間スプライン。ヴォディカ (Reinhard Wodicka) による改良アルゴリズムを使う。(Akima spline は補間区間の両側とその隣の計 4 点から三次の補間曲線を求める方法である。データ点における二次導関数の連続性は保証されない。Wodicka による改良では、その不連続な点で二次導関数の値の変化する量が、元の Akima の方法よりも小さくなるよう工夫している)。

`gsl_interp_akima_periodic` [Interpolation Type]

周期的境界条件での「角張った」秋間スプライン。ヴォディカによる改良アルゴリズムを使う。

以下の関数も利用できる。

`const char * gsl_interp_name (`*const gsl_interp * **interp***`)` [Function]

インスタンス *interp* が使っている補間法の名前文字列へのポインタを返す。例えば以下の文は `interp uses 'cspline' interpolation.` のように出力する。

```
printf("interp uses '%s' interpolation.\n", gsl_interp_name (interp));
```

`unsigned int gsl_interp_min_size` (*const gsl_interp * **interp***) [Function]

インスタンス *interp* に設定されている補間法が、最低で何個のデータ点を必要とするかを返す。例えば秋間スプラインによる補間では、5 点以上の点を必要とする。

26.4 添え字検索とその高速化

x のある任意の値が、与えられるデータ点配列の中のどの位置にあるか (どの二点の間にあるか) を検索するという作業は、一種の繰り返し計算である。とびとびのデータ点を補間する関数を滑らかにプロットしようとするとき、データ点よりもずっと多い点数についてこの検索を行って、その点を含む区間における補間関数を使って関数値を計算することになるが、その結果は `gsl_interp_accel` インスタンスに保存できる。これを使って x の値を検索しておくことで、その x の値で補間関数の評価が失敗した時、それがデータ点配列中のどこだったかを直ちに知ることができる。

`size_t gsl_interp_bsearch` (*const double **x_array[]***, *double* `x`, *size_t **index_lo***, *size_t **index_hi***) [Function]

`x_array[i] <= x < x_array[i+1]` となるような *x_array* の添え字 i を返す。添え字は [*index_lo*, *index_hi*] の範囲で検索される。`HAVE_INLINE` が定義されているときは、インライン展開される。

`gsl_interp_accel * gsl_interp_accel_alloc` (*void*) [Function]

データ配列中の検索の状況を保持するインスタンス (加速検索インスタンス、accelerator object) へのポインタを返す。このインスタンスで検索の状況を追跡することで、補間関数にいろいろな高速化の工夫をすることができる。

`size_t gsl_interp_accel_find` (*gsl_interp_accel * a*, *const double **x_array[]***, *size_t **size***, *double **x***) [Function]

加速検索インスタンス a を使って、(直前の検索結果を初期値として) データ配列中の x の含まれる区間を検索する。補間関数の評価を行う関数の中では、この関数が使われている。`x_array[i] <= x < x_array[i+1]` となる添え字 i を返す。`HAVE_INLINE` が定義されているときは、インライン展開される。

`void gsl_interp_accel_free` (*gsl_interp_accel * acc*) [Function]

高速検索インスタンス *acc* のメモリを解放する。

26.5 補間関数の関数値の計算

`double gsl_interp_eval` (*const gsl_interp * **interp***, *const double **xa[]***, *const double* `ya[]`, *double* `x`, *gsl_interp_accel * acc*) [Function]

`int gsl_interp_eval_e` (*const gsl_interp * **interp***, *const double **xa[]***, *const double **ya[]***, *double **x***, *gsl_interp_accel * **acc***, *double * **y***) [Function]

与えられる点 x での補間関数値 y を、補間インスタンス *interp*、データ配列 *xa*、*ya*、加速検索インスタンス *acc* を使って計算し、返す。

`double gsl_interp_eval_deriv` (*const gsl_interp * **interp***, *const double **xa[]***, *const double **ya[]***, *double **x***, *gsl_interp_accel * **acc***) [Function]
`int gsl_interp_eval_deriv_e` (*const gsl_interp * **interp***, *const double **xa[]***, *const double **ya[]***, *double **x***, *gsl_interp_accel * **acc***, *double * **d***) [Function]

与えられる点 x での補間関数値の導関数値 d を、補間インスタンス *interp*、データ配列 *xa*、*ya*、加速検索インスタンス *acc* を使って計算し、返す。

`double gsl_interp_eval_deriv2` (*const gsl_interp * **interp***, *const double **xa[]***, *const double **ya[]***, *double **x***, *gsl_interp_accel * **acc***) [Function]
`int gsl_interp_eval_deriv2_e` (*const gsl_interp * **interp***, *const double **xa[]***, *const double **ya[]***, *double **x***, *gsl_interp_accel * **acc***, *double * **d2***) [Function]

与えられる点 x での補間関数値の二階導関数値 *d2* を、補間インスタンス *interp*、データ配列 *xa*、*ya*、加速検索インスタンス *acc* を使って計算し、返す。

`double gsl_interp_eval_integ` (*const gsl_interp * **interp***, *const double **xa[]***, *const double **ya[]***, *double **a***, *double **b***, *gsl_interp_accel * **acc***) [Function]
`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***) [Function]

与えられる区間 $[a, b]$ での補間関数の定積分値 *result* を、補間インスタンス *interp*、データ配列 *xa*、*ya*、加速検索インスタンス *acc* を使って計算し、返す。

26.6 高レベル関数

前節までの関数では、呼び出しごとに利用者が配列 x や y の配列へのポインタを指定する必要がある。以下の関数はそれぞれ対応する `gsl_interp` の関数と同じであるが、与えられたデータ配列の要素を `gsl_spline` オブジェクトのインスタンス内にコピーし、保持する。これにより呼び出しごとに *xa* と *ya* を引数として指定する必要がなくなる。これらの関数はヘッダファイル '`gsl_spline.h`' で宣言されている。

`gsl_spline * gsl_spline_alloc` (*const gsl_interp_type * **T***, *size_t **size***) [Function]
`int gsl_spline_init` (*gsl_spline * **spline***, *const double **xa[]***, *const double **ya[]***, *size_t **size***) [Function]
`void gsl_spline_free` (*gsl_spline * **spline***) [Function]

const char * gsl_spline_name (*const gsl_spline * **spline***) [Function]
unsigned int gsl_spline_min_size (*const gsl_spline * **spline***) [Function]
double gsl_spline_eval (*const gsl_spline * **spline***, *double **x***, *gsl_interp_accel * **acc***) [Function]
int gsl_spline_eval_e (*const gsl_spline * **spline***, *double **x***, *gsl_interp_accel * **acc***, *double * **y***) [Function]
double gsl_spline_eval_deriv (*const gsl_spline * **spline***, *double **x***, *gsl_interp_accel * **acc***) [Function]
int gsl_spline_eval_deriv_e (*const gsl_spline * **spline***, *double **x***, *gsl_interp_accel * **acc***, *double * **d***) [Function]
double gsl_spline_eval_deriv2 (*const gsl_spline * **spline***, *double **x***, *gsl_interp_accel * **acc***) [Function]
int gsl_spline_eval_deriv2_e (*const gsl_spline * **spline***, *double **x***, *gsl_interp_accel * **acc***, *double * **d2***) [Function]
double gsl_spline_eval_integ (*const gsl_spline * **spline***, *double **a***, *double **b***, *gsl_interp_accel * **acc***) [Function]
int gsl_spline_eval_integ_e (*const gsl_spline * **spline***, *double **a***, *double **b***, *gsl_interp_accel * **acc***, *double * **result***) [Function]

26.7 例

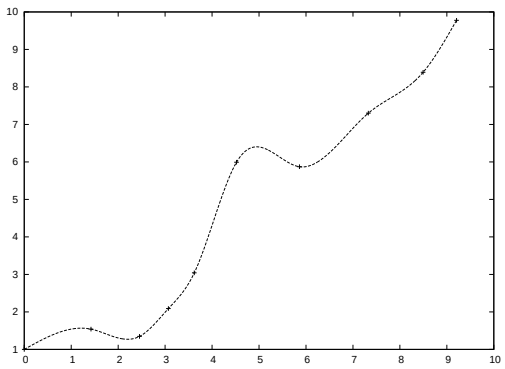
以下に補間関数とスプライン関数を使ったプログラムを例示する。$x_i = i + \sin(i)/2$、$y_i = i + \cos(i^2)$ で与えられる点を $i = 0 \cdots 9$ で計算した 10 点のデータ (x_i, y_i) について三次スプライン補間を行う。

```
#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];
    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    gsl_spline *spline    = gsl_spline_alloc(gsl_interp_cspline, 10);

    printf ("#m=0,S=2\n"); /* plotutils の graph のコマンド */
```

```
    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"); /* plotutils の graph のコマンド */

    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);
    }
    gsl_spline_free(spline);
    gsl_interp_accel_free(acc);

    return 0;
}
```

gnu plotutils の graph プログラムを使って出力をプロットする例を示す。

```
$ ./a.out > interp.dat
$ graph -T ps < interp.dat > interp.ps
```



データ点を滑らかに補間する結果が得られている。補間法は gsl_spline_alloc の最初の引数を変えることで容易に切り替えることができる。

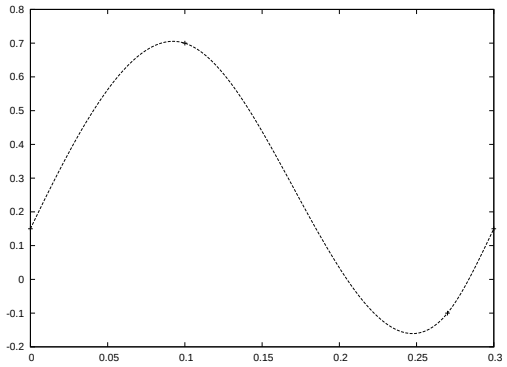
次に、データ点が 4 点の場合の周期的三次スプラインによる補間の例を示す。周期的スプラインでは、データの最初の点と最後の点の y の値は同じでなければならない。

```
#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, i;
    double xi, yi;
    double x[4] = {0.00, 0.10, 0.27, 0.30};
    double y[4] = {0.15, 0.70, -0.10, 0.15}; /* 最初と最後の値は同じ */
    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);

    printf ("#m=0,S=5\n"); /* plotutils の graph のコマンド */

    for (i = 0; i < N; i++) printf ("%g %g\n", x[i], y[i]);

    printf ("#m=1,S=0\n"); /* plotutils の graph のコマンド */
    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;
}
```

GNU graph で出力をプロットすることができる。

```
$ ./a.out > interp.dat
$ graph -T ps < interp.dat > interp.ps
```



ここでは与えられたデータ点に対して、周期的補間を行っている。最初及び最後のデータ点における補間曲線の傾きは、同じになっている。二次微分係数も同様である。

最後に、自然な境界の秋間スプラインの例を示す。このプログラムの出力は、そのまま GNUPLOT への入力になっている。

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

#define RES 300

int main (void)
{
    int N = 6;
    double x[6] = {0., 1., 2., 3., 4., 5. };
    double y[6] = {1., 2., 1., 2., 0., 0. };
    gsl_interp_accel *acc = gsl_interp_accel_alloc ();
    const gsl_interp_type *t = gsl_interp_akima;
    gsl_spline *spline = gsl_spline_alloc (t, N);

    int i; double xi, yi;

    /* for gnuplot */
    printf("plot [:][:] '-' with points," /* Given data x[], y[] */
                      "'-' with dots,"     /* spline curve */
                      "'-' with dots,"     /* derivative */
                      "'-' with dots\n");  /* 2nd derivative */
```

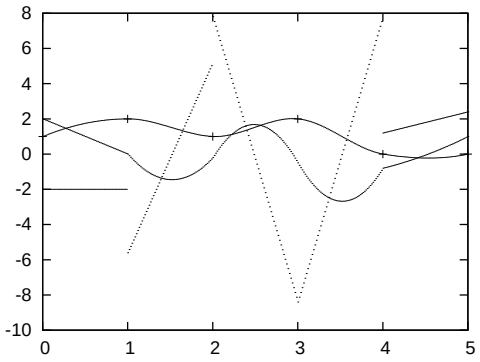
```
    /* plot given data */
    for (i = 0; i < N; i++) printf ("%g %g\n", x[i], y[i]);
    printf("e\n");

    /* plot spline curve */
    gsl_spline_init (spline, x, y, N);
    for (i = 0; i <= RES; i++) {
        xi = (1 - i / (double)RES) * x[0] + (i / (double)RES) * x[N-1];
        yi = gsl_spline_eval (spline, xi, acc);
        printf ("%g %g\n", xi, yi);
    }
    printf("e\n");

    /* plot the first derivative */
    for (i = 0; i <= RES; i++) {
        xi = (1 - i / (double)RES) * x[0] + (i / (double)RES) * x[N-1];
        yi = gsl_spline_eval_deriv (spline, xi, acc);
        printf ("%g %g\n", xi, yi);
    }
    printf("e\n");

    /* plot the second derivative */
    for (i = 0; i <= RES; i++) {
        xi = (1 - i / (double)RES) * x[0] + (i / (double)RES) * x[N-1];
        yi = gsl_spline_eval_deriv2 (spline, xi, acc);
        printf ("%g %g\n", xi, yi);
    }
    printf("e\n");

    gsl_spline_free (spline);
    gsl_interp_accel_free (acc);

    return 0;
}
```

このプログラムを akima という名前でコンパイルすると、

```
$ ./akima | gnuplot
```

で画面にプロットが表示される。もし一瞬だけ表示されてすぐに消えてしまう場合は、上のプログラムの return の前に printf("pause -1\n"); を入れるとよい。出力されるプロットを以下に示す。図中で、データ点を通っているのがスプライン曲線、滑らかではないが連続なのがスプライン

関数の導関数、データ点で不連続な直線が二次導関数である。



26.8 参考文献

補間のアルゴリズムと参考文献については、以下の本に示されている。

- Christoph W. Ueberhuber, *Numerical Computation* Volume 1, Chapter 9 “Interpolation”, Springer, ISBN 3-540-62058-3 (1997).
- David M. Young, Robert Todd Gregory, *A Survey of Numerical Mathematics* Volume 1, Chapter 6.8, Dover, ISBN 0-486-65691-8 (1988).

ヴォディカが改良した Akima スプラインのアルゴリズムについては、以下を参照のこと (ドイツ語)。

- Reinhard Wodicka, “Engänzungen zu Akima’s Steigungsformel”, *Mitteilungen aus dem Mathematischen Seminar Giessen* (Selbstverlag des Mathematischen Instituts, Giessen), Heft 203, pp. 39–47 (1991).

第27章　数値微分

この章では有限差分 (finite differencing) により数値微分 (numerical derivative) を計算する関数について説明する。推定誤差を計算し、それを最小化するようにステップ幅を決める適応型の計算法 (adaptive algorithm) を使っている。ここで説明する関数はヘッダファイル 'gsl_deriv.h' で宣言されている。

27.1　関数

`int gsl_deriv_central (const gsl_function * f, double x, double h, double * result, double * abserr)` [Function]

点 x における関数 f の微分係数をステップ幅 h の中心差分法 (central difference algorithm) で計算して引数 *result* に、推定絶対誤差を *abserr* に入れて返す。

引数で指定される h の値は、微分係数の計算の際の打ち切り誤差 (truncation error) および丸め誤差 (round-off error) について最適なステップ幅を求めるための初期値として使われる。微分係数は横軸上で等間隔に取られる五個の点 $x-h$, $x-h/2$, x, $x+h/2$, $x+h$ から五点則 (5-point rule) を使って計算され、推定誤差はその五点の間の差から、$x-h$, x, $x+h$ での三点則を使って計算される。点 x における関数値は微分係数の計算には寄与せず、実質的には四点が使われるのみである。

`int gsl_deriv_forward (const gsl_function * f, double x, double h, double * result, double * abserr)` [Function]

点 x における関数 f の微分係数をステップ幅 h の前進差分法 (forward difference algorithm) で計算して引数 *result* に、推定絶対誤差を *abserr* に入れて返す。関数値は x よりも大きな点でのみ計算され、x での値は計算されない。この関数は点 x で関数 $f(x)$ が連続でない場合や、x よりも小さな範囲では未定義であるような場合に使うことができる。

引数で指定される h の値は、微分係数の計算の際の打ち切り誤差 (truncation error) および丸め誤差 (round-off error) について最適なステップ幅を求めるための初期値として使われる。横軸上で等間隔に取られる点 $x+h/4$, $x+h/2$, $x+3h/4$, $x+h$ を使った「開いた (open)」四点則で点 x での微分係数が、その四点での差から、$x+h/2$, $x+h$ での二点則を使って推定誤差が計算される。

`int gsl_deriv_backward (const gsl_function * f, double x, double h, double * result, double * abserr)` [Function]

点 x における関数 f の微分係数をステップ幅 h の後退差分法 (backward difference algorithm) で計算して引数 *result* に、推定絶対誤差を *abserr* に入れて返す。関数値は x よりも小さな点でのみ計算され、x での値は計算されない。この関数は点 x で関数 $f(x)$ が連続でない場合や、x よりも大きな範囲では未定義であるような場合に使うことができる。

これは gsl_deriv_forward をステップ幅を負にして呼び出すのと同じである。

27.2　例

以下のプログラムでは関数 $f(x) = x^{3/2}$ の微分係数を点 $x = 2$ と $x = 0$ で計算する。関数 $f(x)$ は $x < 0$ では未定義なので、$x = 0$ での微分係数は gsl_deriv_forward を使って計算する。

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_deriv.h>
double f (double x, void * params)
{
    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);
    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;
}
```

このプログラムの出力例を以下に示す。

```
$ ./a.out
f(x) = x^(3/2)
x = 2.0
f'(x) = 2.1213203120 +/- 0.0000004064
exact = 2.1213203436

x = 0.0
f'(x) = 0.0000000160 +/- 0.0000000339
exact = 0.0000000000
```

27.3 参考文献

ここで説明した関数で使われているアルゴリズムは、以下の文献にある。

- Abramowitz, Stegun (eds.), *Handbook of Mathematical Functions*, Section 25.3.4, and Table 25.5 (Coefficients for Differentiation), National Bureau of Standards, U.S. (1964).
- Samuel Daniel Conte, Carl de Boor, *Elementary Numerical Analysis: An Algorithmic Approach*, McGraw-Hill (1972).
  3rd ed. が 1980 年に出ている (ISBN 978-0070124479)。

第28章　チェビシェフ近似

この章では一変数関数 (univariate function) のチェビシェフ近似 (Chebyshev approximation) を計算する関数について説明する。区間 $[-1, 1]$、重み関数 $1/\sqrt{(1-x^2)}$ のもとで直交基底 (orthogonal basis) となるチェビシェフ多項式 (Chebyshev polynomial) $T_n(x) = \cos(n \arccos(x))$ の有限級数で任意の関数を $f(x) = \sum c_n T_n(x)$ と近似するのがチェビシェフ近似である。低次のチェビシェフ多項式は $T_0(x) = 1,\ T_1(x) = x,\ T_2(x) = 2x^2 - 1, \ldots$ である。詳しくはアブラモウィッツ＆ステグン (Abramowitz and Stegun) の第 22 章を参照のこと。

この章で説明する関数はヘッダファイル '`gsl_chebyshev.h`' で宣言されている。

28.1　gsl_cheb_series 構造体

チェビシェフ近似は以下の構造体に保持される。

```
typedef struct {
    double * c; /* 係数 c[0] .. c[order] */
    int order;  /* 展開する項数 */
    double a;   /* 区間の下界 */
    double b;   /* 区間の上界 */
    ...
} gsl_cheb_struct
```

c[0] を含む *order*+1 個の項によって区間 [a, *b*] での近似が計算される。級数は以下で計算される。

$$f(x) = \frac{c_o}{2} + \sum_{n=1} c_n T_n(x)$$

係数の値を直接参照する場合には、この式によって解釈する。

28.2　チェビシェフ近似のインスタンスと計算

`gsl_cheb_series * gsl_cheb_alloc` (*const size_t* `n`)　[Function]

n 次のチェビシェフ近似のためのメモリを確保し、生成した `gsl_cheb_series` 構造体へのポインタを返す。

`void gsl_cheb_free` (*gsl_cheb_series* * `cs`)　[Function]

チェビシェフ近似のインスタンス *cs* のメモリを解放する。

`int gsl_cheb_init` (*gsl_cheb_series * **cs***, *const gsl_function * **f***, *const double* `a`, *const double* `b`) [Function]

関数 f の区間 (a,b) でのチェビシェフ近似を、前もって指定されていた次数で計算する。チェビシェフ近似の計算量のオーダーは $O(n^2)$ で、関数値の計算が n 回必要である。

28.3　チェビシェフ近似による近似値の計算

`double gsl_cheb_eval` (*const gsl_cheb_series * **cs***, *double* `x`) [Function]

与えられる点 x でのチェビシェフ近似の値 *cs* を計算する。

`int gsl_cheb_eval_err` (*const gsl_cheb_series * **cs***, *const double* `x`, *double * **result***, *double * **abserr***) [Function]

与えられる点 x でのチェビシェフ近似の値 *cs* を計算し、級数の値を *result* に、推定絶対誤差を *abserr* に入れて返す。推定誤差は級数計算での打ち切り誤差から計算する。

`double gsl_cheb_eval_n` (*const gsl_cheb_series * **cs***, *size_t **order***, *double* `x`) [Function]

与えられる点 x でのチェビシェフ近似の値 *cs* を、指定される次数 *order* (と *cs* に設定されている次数のどちらか小さい方) で計算する。

`int gsl_cheb_eval_n_err` (*const gsl_cheb_series * **cs***, *const size_t **order***, *const double* `x`, *double * **result***, *double * **abserr***) [Function]

与えられる点 x でのチェビシェフ級数 *cs* を指定される次数 *order* (と *cs* に設定されている次数のどちらか小さい方) で計算し、級数の値を *result* に、推定絶対誤差を *abserr* に入れて返す。推定誤差は級数計算での打ち切り誤差から計算する。

28.4　微分と積分

以下の関数でチェビシェフ級数の微分、積分を行って新しいチェビシェフ級数を作ることができる。微分による数列での誤差は、高次の項の切り捨てにより小さく見積もられることがある。

`int gsl_cheb_calc_deriv` (*gsl_cheb_series * **deriv***, *const gsl_cheb_series * **cs***) [Function]

cs のチェビシェフ級数の導関数を計算する。チェビシェフ級数の導関数は、同じ次数のチェビシェフ級数の係数を変えるだけで表現できるため、求めた導関数はチェビシェフ近似オブジェクト *deriv* として返される。*cs* と *deriv* は同じ次数で確保されていなければならない。

`int gsl_cheb_calc_integ` (*gsl_cheb_series * **integ***, *const gsl_cheb_series * **cs***)
[Function]

cs のチェビシェフ級数の不定積分を計算する。求めた原関数はチェビシェフ近似オブジェクト *integ* として返される。*cs* と *integ* は同じ次数で確保されていなければならない。積分範囲の下界 (積分の境界条件の場所) は、*cs* で設定されている区間の下界 `a` である。

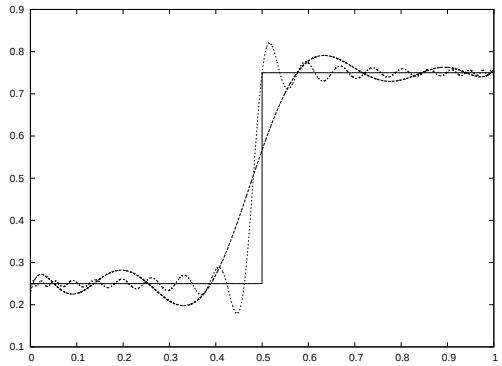
28.5 例

以下のプログラムでは、ステップ関数のチェビシェフ近似を計算する。ステップ関数は不連続であるため近似は非常に難しく、誤差がはっきり見えるよい例である。連続関数に対してはチェビシェフ近似は非常に速く収束し、誤差はほとんど見えない。

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

double f (double x, void *p)
{
    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);
        printf ("%g %g %g %g\n", x, GSL_FN_EVAL (&F, x), r10, r40);
    }
    gsl_cheb_free (cs);
```

```
        return 0;
    }
```

プログラムは元の関数の値と 10 次および 40 次のチェビシェフ近似を x が 0.001 刻みで出力する。



28.6　参考文献

チェビシェフ近似に関するいくつかの FORTRAN ルーチンが利用法が以下の文献にある。

- R. Broucke, “Ten Subroutines for the Manipulation of Chebyshev Series [C1] (Algorithm 446)”. *Communications of the ACM*, **16**(4), pp. 254–256 (1973).

また、このマニュアルの至る所で参照されている聖典、「アブラモウィッツ＆ステグン」は、以下である。

- Abramowitz, Stegun (eds.), *Handbook of Mathematical Functions*, National Bureau of Standards, U.S. (1964).

第 29 章　級数の収束加速

この章では、級数の収束を加速するレヴィンの u 変換 (Levin's u-transform) を行う関数について説明する。この方法では級数の最初のいくつかの項の値から、補外 (extrapolation) によってその先の項の値の近似値と近似誤差を計算する。u 変換は漸近するものを含め、収束する級数と発散する級数の両方に使うことができる。

この章で説明する関数はヘッダファイル 'gsl_sum.h' で宣言されている。

29.1　収束を加速する関数

以下の関数は級数のレヴィンの u 変換と推定誤差を計算する。誤差の推定量は補外する最後の項まで各項の丸め誤差を伝播させていくことで計算する。

この節の関数は、級数の各項が高精度で計算できれば得られる和が解析的な極限値に近づくこと、また、丸め誤差 (rounding error) は計算精度がある有限の値であることから生じることを想定している。その計算精度は、各項における相対誤差が GSL_DBL_EPSILON のオーダーに収まるようになっている。

補外した項における誤差の推定量を得るための計算量は $O(N^2)$ のオーダーであり、計算時間とメモリの両方を大きく消費する。補外した値の収束の様子から誤差を見積もると信頼性は低いが計算は速い。これは次の節で説明する。この節の関数は、推定誤差の信頼性を確保するため $O(N)$ までのすべての関数値とその導関数値を計算し、保持する。

`gsl_sum_levin_u_workspace * gsl_sum_levin_u_alloc (`*size_t* **`n`**`)` [Function]

> 項数 n のレヴィンの u 変換の作業領域のためのメモリを確保する。確保するメモリの大きさは $O(2n^2 + 3n)$ のオーダーである。

`int gsl_sum_levin_u_free (`*gsl_sum_levin_u_workspace* `*` **`w`**`)` [Function]

> 作業領域 w に割り当てられているメモリを解放する。

`int gsl_sum_levin_u_accel (`*const double* `*` **`array`**, *size_t* **`array_size`**, *gsl_sum_levin_u_workspace* `*` **`w`**, *double* `*` **`sum_accel`**, *double* `*` **`abserr`**`)` [Function]

> 大きさ *array_size* の配列 *array* で渡される級数の最初の *array_size* 個の項から、レヴィンの u 変換を使って無限級数の極限値を計算する。別途に確保した作業領域 w が必要になる。級数が収束するであろう極限値が計算されて *sum_accel* に、その推定絶対誤差が *abserr* に入れられる。また各項までの和が *w*->sum_plain に入れられる。この方法では打ち切り誤差 (補外により得られる二つの連続した項の値の差) と丸め誤差

(各項で発生し次の項に伝播していく) を計算し、それに基づいて最適な補外項数を決定する。配列 *array* で渡す級数の各項は、すべて非零でなければならない。

29.2　誤差の推定を行わない加速関数

この節で説明する関数は級数をレヴィンの u 変換で計算し、打ち切り誤差を最後の二つの近似項の差とする。この方法では誤差の推定は補外した項の変化していく様子から直接計算されるので、微分値を計算、保存しておく必要がなく、計算量もメモリ使用量も $O(N)$ のオーダーですむ。もし級数の収束が十分に速いときは、この方法を使うとよい。また高速で同じような収束を示す多数の級数を計算する必要があるときも有用である。たとえば、複数の積分が、互いに似通った値のパラメータで定義される級数になっているような場合である。前節の関数を使って誤差の推定量を最初に一度計算しておくと、この方法でもある程度は信頼性のある近似誤差の値が得られるだろう。

`gsl_sum_levin_utrunc_workspace * gsl_sum_levin_utrunc_alloc` (*size_t* **`n`**) [Function]

項数 n で誤差推定を行わないレヴィンの u 変換の作業領域のためのメモリを確保する。確保するメモリの大きさは $O(3n)$ のオーダーである。

`int gsl_sum_levin_utrunc_free` (*gsl_sum_levin_utrunc_workspace* * **`w`**) [Function]

作業領域 w に割り当てられているメモリを解放する。

`int gsl_sum_levin_utrunc_accel` (*const double* * **`array`**, *size_t* **`array_size`**, *gsl_sum_levin_utru* * **`w`**, *double* * **`sum_accel`**, *double* * **`abserr_trunc`**) [Function]

大きさ *array_size* の配列 *array* で渡される級数の最初の *array_size* 個の項から、レヴィンの u 変換を使って無限級数の極限値を計算する。別途に確保した作業領域 w が必要になる。級数が収束するであろう極限値が計算されて *sum_accel* に、また各項までの和が `w->sum_plain` に入れられる。

補外により得られる連続した二つの項の差が最小値を取るか、十分に小さくなったときに補外を打ち切る。この差が誤差の推定量として *abserr_trunc* に入れられる。打ち切り誤差を計算するときに、補外で得られる項の値の代わりに、値の動きがさほど不安定でない移動平均値 (moving average) を使うと、信頼性をあげることができる。

29.3　例

以下のプログラムでは、

$$\zeta(2) = 1 + 1/2^2 + 1/3^2 + 1/4^2 + \ldots$$

を使って $\zeta(2) = \pi^2/6$ を計算する。項数 N のとき級数に含まれる誤差のオーダーは $O(1/N)$ になり、項の値を直接加えていく方法では収束が遅くなっていく。

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

#define N 20

int main (void)
{
    double t[N], sum_accel, err, 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;

    /* zeta(2) = \sum_{n=1}^{\infty} 1/n^2 の項の計算 */
    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 %d terms\n",
           w->sum_plain, w->terms_used);
    printf("exact value      = % .16f\n", zeta_2);
    printf("accelerated sum  = % .16f using %d 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;
}
```

プログラムの出力を以下に示す。レヴィンの u 変換を使うことで、最初の 11 項で 10^{10} 分の 1 の精度で級数の値が求められている。関数が返した推定誤差も高精度で、厳密解とよく一致している。

```
$ ./a.out
term-by-term sum =  1.5961632439130233 using 20 terms
term-by-term sum =  1.5759958390005426 using 13 terms
```

```
exact value      =  1.6449340668482264
accelerated sum  =  1.6449340668166479 using 13 terms
estimated error  =  0.0000000000508580
actual error     = -0.0000000000315785
```

項を直接加えていく方法で同じ精度を得ようとすると、10^{10} 個の項が必要だが、加速法を使うと 13 項で済んでいる。

29.4　参考文献

ここで説明した関数で使っているアルゴリズムは、以下の論文に説明されている。

- T. Fessler, W.F. Ford, D.A. Smith, "HURRY: *An acceleration algorithm for scalar sequences and series*", *ACM Transactions on Mathematical Software (TOMS)*, **9**(3), pp. 346–354 (1983), and "Algorithm 602", *TOMS*, **9**(3), pp. 355–357 (1983).

レヴィンによる u 変換の理論は以下の論文にある。

- D. Levin, "Development of Non-Linear Transformations for Improving Convergence of Sequences", *International Journal of Computer Mathematics*, **3**(1), pp. 371–388 (1973).

各種のレヴィン変換の総説を web でも見ることができる。

- Herbert H. H. Homeier, *Scalar Levin-Type Sequence Transformations* (2008). `http://arxiv.org/abs/math/0005209`

第30章　ウェーブレット変換

この章では離散ウェーブレット変換 (Discrete Wavelet Transform, DWT) を行う関数について説明する。GSL のウェーブレット関数では、一次元および二次元の実数空間でのウェーブレット変換を行うことができる。関数はヘッダファイル 'gsl_wavelet.h' および 'gsl_wavelet2d.h' で宣言されている。

30.1　DWT の定義

連続ウェーブレット変換 (continuous wavelet transform) は以下の式で定義される。

$$w(s,\tau) = \int_{-\infty}^{\infty} f(t)\psi^*_{s,\tau}(t)\,dt$$

また逆変換は以下の式である。

$$f(t) = \int_0^{\infty} ds \int_{-\infty}^{\infty} w(s,\tau)\psi_{s,\tau}(t)\,d\tau$$

ここで $\psi_{s,\tau}$ は基底関数 (basis function) であり、マザー・ウェーブレット (mother wavelet) と呼ばれるひとつの関数から、スケーリング (scaling、拡大縮小) および平行移動 (translation) によって得られる。

離散ウェーブレット変換は等間隔データに対して適用される。そのときに使われるスケーリング値および平行移動量 (s,τ) はある固定された間隔の離散値である。周波数と時間のサンプリング点は、ぞれぞれの座標軸上で 2^j で表される離散値を取る係数 (j をレベル・パラメータ level parameter とよぶ) でディアディック (diadic、二つのベクトルを並べたもの) として決まる。それによって得られる一連の関数 $\{\psi_{j,n}\}$ は二乗可積分 (square-integrable) な直交基底 (orthogonal basis) をなす。

離散ウェーブレット変換の計算量は $O(N)$ のオーダーである。そのため高速ウェーブレット変換 (fast wavelet transform) とも呼ばれる。

30.2　DWT 関数の初期化

gsl_wavelet 構造体はウェーブレットを定義する係数と、それに関わるオフセット・パラメータを保持する。

`gsl_wavelet * gsl_wavelet_alloc` (*const gsl_wavelet_type * T*, *size_t k*) [Function]

T で示される種類のウェーブレットのインスタンスを生成する。引数 k で、その中でどのウェーブレットを使うかを指定する。無効な種類を指定したり十分なメモリが確保できなかったときには NULL ポインタを返す。

ウェーブレットの種類には以下のようなものが用意されている。

`gsl_wavelet_daubechies` [Wavelet]
`gsl_wavelet_daubechies_centered` [Wavelet]

消失モーメント (vanishing moment) が $k/2$ の、最大位相のドブシー・ウェーブレット (Daubechies wavelet)。ここで実装されているウェーブレットは k が偶数に対してであり、$k = 4, 6, \ldots, 20$ である。

`gsl_wavelet_haar` [Wavelet]
`gsl_wavelet_haar_centered` [Wavelet]

ハール・ウェーブレット (Haar wavelet)。$k = 2$ でなければならない。

`gsl_wavelet_bspline` [Wavelet]
`gsl_wavelet_bspline_centered` [Wavelet]

(i, j) 次の双直行 B-スプライン・ウェーブレット (biorthogonal B-spline wavelet)。$k = 100 * i + j$ が 103、105、202、204、206、208、301、303、305、307、309 について実装されている。

中心化されたウェーブレット (centered form) では、各基底関数 (が表す波形) の端が揃うように係数が決められる。そのため、そのウェーブレット変換の係数は、位相空間で見るのが理解しやすい。

`const char * gsl_wavelet_name (`*const gsl_wavelet * `w*`)` [Function]

ウェーブレット w の名前文字列へのポインタを返す。

`void gsl_wavelet_free (`*gsl_wavelet * `w*`)` [Function]

ウェーブレットのインスタンス w のメモリを解放する。

`gsl_wavelet_workspace` 構造体には、変換中の途中結果を保持するための、入力データと同じ大きさの作業用メモリ領域が確保されている。

`gsl_wavelet_workspace * gsl_wavelet_workspace_alloc (`*size_t `n`*`)` [Function]

離散ウェーブレット変換のための作業領域を確保する。n 個の要素での一次元変換を行うための大きさ n の領域が確保される。二次元の $n \times n$ 行列の場合でも、各行と各列についてそれぞれ独立に変換が行われるため、大きさ n の領域があればよい。

`void gsl_wavelet_workspace_free (`*gsl_wavelet_workspace * `work`*`)` [Function]

workspace のメモリを解放する。

30.3 変換関数

この節では、実際に変換を行う関数について説明する。その変換では周期的境界 (periodic boundary condition) が条件であることに留意せねばならない。信号がサンプル全体の長さで周期的でない場合は、変換の各段階 (各サブバンドにおける変換) の最初と最後で、係数の値が間違って計算される。

30.3.1 一次元のウェーブレット変換

`int gsl_wavelet_transform` (*const gsl_wavelet* * `w`, *double* * `data`, *size_t* `stride`, *size_t* `n`, *gsl_wavelet_direction* `dir`, *gsl_wavelet_workspace* * `work`) [Function]
`int gsl_wavelet_transform_forward` (*const gsl_wavelet* * `w`, *double* * `data`, *size_t* `stride`, *size_t* `n`, *gsl_wavelet_workspace* * `work`) [Function]
`int gsl_wavelet_transform_inverse` (*const gsl_wavelet* * `w`, *double* * `data`, *size_t* `stride`, *size_t* `n`, *gsl_wavelet_workspace* * `work`) [Function]

長さ n で刻み幅 *strie* のデータ *data* の、順または逆離散ウェーブレット変換を計算する。変換長 n は 2 のべき乗でなければならない。一番上の `transform` の関数は、引数 *dir* に `forward` (+1) または `backward` (-1) を指定することができる。またどの関数も長さ n の作業領域 *work* を確保して指定しなければならない。

順方向の変換では、配列が保持している元データは、離散ウェーブレット変換 $f_i \to w_{j,ki}$ によって三角形式で圧縮されて置き換えられる (ワード境界へのアライメント／パディングは行われない)。ここで j は $j = 0 \ldots J-1$ でレベルの添え字、k は各レベルでの係数の添え字で $k = 0 \ldots 2^j - 1$ である。レベルの総数は $J = \log_2(n)$ である。出力されるデータは以下の形式である。

$$(s_{-1,0},\ d_{0,0},\ d_{1,0},\ d_{1,1},\ d_{2,0},\ \ldots,\ d_{j,k},\ldots,\ d_{J-1,2^{J-1}})$$

最初の要素は平滑化係数 $s_{-1,0}$ で、各レベル j の係数 $d_{j,k}$ が後に続く。逆変換はこれらの係数から元データを得る。

関数の返り値は、変換がうまく終了したときは `GSL_SUCCESS`、n が 2 のべき乗ではないとき、または作業領域の大きさがたりないときには `GSL_EINVAL` である。

30.3.2 二次元のウェーブレット変換

GSL では正方行列に対する二次元のウェーブレット変換を行う関数を用意している。行列の各次元は 2 のべき乗でなければならない。二次元ウェーブレット変換での行と列の並べ方には「標準 (standard)」と「非標準 (non-standard)」の二通りがある。

標準変換では、まず各行に対してそれぞれ離散ウェーブレット変換が行われ、続いて変換された値に対して、列ごとに個別の離散ウェーブレット変換が行われる。これは二次元のフーリエ変換と同じ順序である。

非標準変換では、各レベルごとに行の、つづいて列の変換が行われる。最初のレベルでは、まず各行について変換が行われ、次に各列が変換される。同様に次のレベルでの変換をデータの各行と列について適用することを繰り返し、全てのレベルの離散ウェーブレット変換が行われた時点で終了する。非標準変換は主に画像解析で利用されている。

この節の関数はヘッダファイル 'gsl_wavelet2d.h' で宣言されている。

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*) [Function]

int gsl_wavelet2d_transform_forward (*const gsl_wavelet * w*, *double * data*, *size_t tda*, *size_t size1*, *size_t size2*, *gsl_wavelet_workspace * work*) [Function]

int gsl_wavelet2d_transform_inverse (*const gsl_wavelet * w*, *double * data*, *size_t tda*, *size_t size1*, *size_t size2*, *gsl_wavelet_workspace * work*) [Function]

行指向形式で次数が *size1*、*size2*、行の長さが *tda* のデータ *data* に対して、標準形式で順および逆離散ウェーブレット変換を行う。行と列の次元は等しくなければならず (正方行列でなければならない)、2 のべき乗でなければならない。transform の関数は引数 *dir* に forward (+1) または backward (-1) を指定する。またどの関数も作業領域をあらかじめ確保して、*work* として指定する。関数が終了するときに *data* の内容は計算された離散ウェーブレット変換で置き換えられる。

変換が正常に終了したときは返り値は GSL_SUCCESS になる。*size1* と *size2* の値が等しくないときや 2 のべき乗でないときや作業領域が十分でないときは GSL_EINVAL を返す。

int gsl_wavelet2d_transform_matrix (*const gsl_wavelet * w*, *gsl_matrix * m*, *gsl_wavelet_direction dir*, *gsl_wavelet_workspace * work*) [Function]

int gsl_wavelet2d_transform_matrix_forward (*const gsl_wavelet * w*, *gsl_matrix * m*, *gsl_wavelet_workspace * work*) [Function]

int gsl_wavelet2d_transform_matrix_inverse (*const gsl_wavelet * w*, *gsl_matrix * m*, *gsl_wavelet_workspace * work*) [Function]

行列 a で与えられるデータについて二次元ウェーブレット変換を計算し、元のデータを変換結果で上書きする。

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*) [Function]

int gsl_wavelet2d_nstransform_forward (*const gsl_wavelet * w*, *double * data*, *size_t tda*, *size_t size1*, *size_t size2*, *gsl_wavelet_workspace * work*) [Function]

int gsl_wavelet2d_nstransform_inverse (*const gsl_wavelet * w*, *double * data*, *size_t tda*, *size_t size1*, *size_t size2*, *gsl_wavelet_workspace * work*) [Function]

非標準二次元ウェーブレット変換を計算する。

int gsl_wavelet2d_nstransform_matrix (*const gsl_wavelet * w*, *gsl_matrix * m*, *gsl_wavelet_direction dir*, *gsl_wavelet_workspace * work*) [Function]
int gsl_wavelet2d_nstransform_matrix_forward (*const gsl_wavelet * w*, *gsl_matrix * m*, *gsl_wavelet_workspace * work*) [Function]
int gsl_wavelet2d_nstransform_matrix_inverse (*const gsl_wavelet * w*, *gsl_matrix * m*, *gsl_wavelet_workspace * work*) [Function]

行列 m のデータの非標準二次元ウェーブレット変換を計算し、元のデータを変換結果で上書きする。

30.4 例

以下のプログラムは一次元のウェーブレット変換を行う。長さ 256 の入力信号に対して、ウェーブレット変換で得られる要素のうち上位 20 個を使い、それ以外は 0 と置いて近似を行う。

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_wavelet.h>

int main (int argc, char **argv)
{
    int i, n = 256, nc = 20;
    double *data = malloc(n * sizeof (double));
    double *abscoeff = malloc(n * sizeof (double));
    size_t *p = malloc(n * sizeof (size_t));
    gsl_wavelet *w;
    gsl_wavelet_workspace *work;

    w = gsl_wavelet_alloc(gsl_wavelet_daubechies, 4);
    work = gsl_wavelet_workspace_alloc(n);
    FILE *f = fopen(argv[1], "r");

    for (i = 0; i < n; i++) fscanf(f, "%lg", &data[i]);
    fclose(f);

    gsl_wavelet_transform_forward(w, data, 1, n, work);
    for (i = 0; i < n; i++) abscoeff[i] = fabs(data[i]);

    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\n", data[i]);

    gsl_wavelet_free(w);
    gsl_wavelet_workspace_free(work);

    free(data);
    free(abscoeff);
    free(p);
    return 0;
}
```

このプログラムの出力はそのまま GNU の plotuilts に含まれる graph コマンドの入力として渡すことができる。

```
$ ./a.out ecg.dat > dwt.dat
$ graph -T ps -x 0 256 32 -h 0.3 -a dwt.dat > dwt.ps
```

以下に示すグラフには MIT-BIH の不整脈データベースに登録されている ECG 記録をサンプルとして、原信号とウェーブレットによる近似信号がプロットされている。このデータベースは PhysioNet の公開医療データベースの一部である。

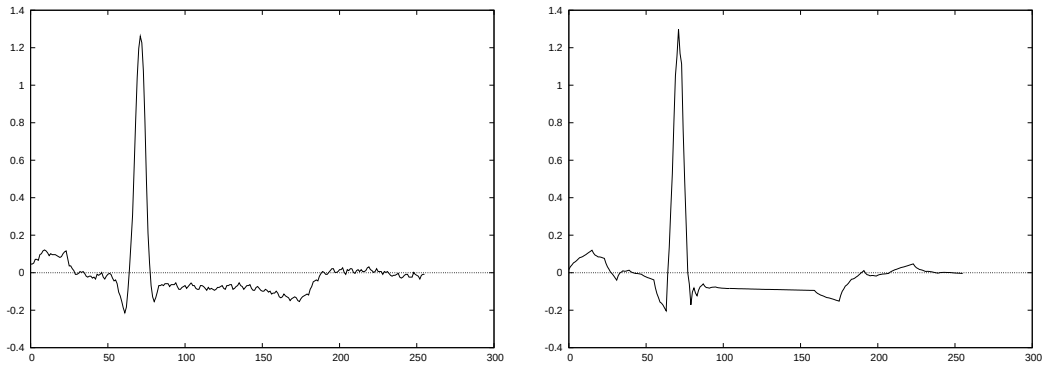


図 30.1: *
ECG 信号の原信号 (左) と、ドブシー (4) の離散ウェーブレット変換による上位 20 要素を使ったウェーブレット変換による近似信号 (右)。

30.5　参考文献

ウェーブレット変換の数学的な記述はドブシー (Ingrid Daubechies) の以下の原著 (講演録) にある。

- Ingrid Daubechies, *Ten Lectures on Wavelets*, CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, ISBN 0898712742 (1992).

様々な分野での応用を概観したいときには、以下の本がよい。

- Paul S. Addison, *The Illustrated Wavelet Transform Handbook*, Institute of Physics Publishing, ISBN 0750306920 (2002).

ウェーブレット、ウェーブレット・パケット、局所コサイン基底による信号処理についての記述が以下 の本にある。

- S. G. Mallat, *A wavelet tour of signal processing* (Second edition), Academic Press, ISBN 012466606X (1999).

ウェーブレット解析の背景にある多重解像度解析 (multi-resolution analysis) の考え方は、以下が参考になる。

- S. G. Mallat, "Multiresolution Approximations and Wavelet Orthonormal Bases of L2 (R)", *Transactions of the American Mathematical Society*, **315**(1), pp. 69–87 (1989).

- S. G. Mallat, "A Theory for Multiresolution Signal Decomposition – The Wavelet Representationi", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **11**(7), pp. 674–693 (1989).

GSL での実装で使っている各ウェーブレットでの係数の値は、以下の論文にある。

- I. Daubechies, "Orthonormal Bases of Compactly Supported Wavelets", *Communications on Pure and Applied Mathematics*, **41**(7), pp. 909–996 (1988).

- A. Cohen, I. Daubechies, and J.-C. Feauveau. "Biorthogonal Bases of Compactly Supported Wavelets", *Communications on Pure and Applied Mathematics*, **45**(5), pp. 485–560 (1992).

生理学データセットのデータベース PhysioNet にはオンライン `http://www.physionet.org/` でアクセスできる。下記の論文を参照。

- Goldberger, et al, "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals", *Circulation*, **101**(23), pp. e215–e220 (2000).

第31章　離散ハンケル変換

この章では離散ハンケル変換 (discrete Hankel transforms, DHT) を行う関数について説明する。関数はヘッダファイル ‘gsl_dht.h’ で宣言されている。

31.1　定義

フーリエ変換を、三角関数の零点においてサンプリングされたデータに適用される変換と考えると、離散ハンケル変換は、次数固定のベッセル関数 (Bessel function) の零点におけるデータ列を変換するものだと考えることができる。

$f(t)$ を長さ 1 の区間内で定義される関数であり、g_m が以下を満たす数値の集合であるとする。

$$g_m = \int_0^1 t dt J_\nu(j_{\nu,m} t) f(t)$$

このとき、$f(t)$ の有限 ν-ハンケル変換は

$$f(t) = \sum_{m=1}^{\infty} \frac{2 J_\nu(j_{\nu,m} x)}{J_{\nu+1}(j_{\nu,m})^2} g_m$$

と表される。ここで $m > M$ に対して $g_m = 0$ であるとすると (有限個のベッセル関数で $f(t)$ が表現できるとする、あるいは高次の項を無視する)、以下の標本化定理 (sampling theorem、サンプリング定理) を得る。

$$g_m = \frac{2}{j_{\nu,M}^2} \sum_{k=1}^{M-1} f\left(\frac{j_{\nu,k}}{j_{\nu,M}}\right) \frac{J_\nu(j_{\nu,m} j_{\nu,k} / j_{\nu,M})}{J_{\nu+1}(j_{\nu,k})^2}$$

これが、離散値で表現された離散ハンケル変換の定義である。和記号中の核 (kernel) が大きさ $M-1$ の ν-ハンケル変換の行列を定義する。行列にかかる係数の値は ν と M に依存するが、GSL では、`gsl_dht` のインスタンスの初期化の際にこれをあらかじめ計算してインスタンス内に保持しておく。メモリ確保関数 `gsl_dht_alloc` で生成した `gsl_dht` 型のインスタンスは関数 `gsl_dht_init` を使って初期化する必要があり、その後に `gsl_dht_apply` を使うことで、与えるデータに対して、初期化の際に指定した値の ν と M についてハンケル変換を行うことができる。GSL では、利便性をはかるためにデータのサンプリングされる区間をスケーリングできるようになっているため、変換対象のデータのサンプリング区間の長さは必ずしも 1 でなくてもよく、$[0, X]$ の形で書ける区間になっていればよい。

サンプリング区間の長さが 1 でない場合、逆変換と上述の g_m の式から、f の値はサンプリング区間の両端で 0 になることになる。したがってこの変換は、ベッセル関数の微分方程式につい

てのディリクレ問題 (Dirichlet problem) の固有関数 (eigenfunction) を、直交するように拡張したものと考えることができる。

31.2　関数

`gsl_dht * gsl_dht_alloc` (*size_t* **`size`**) [Function]

大きさ *size* の離散ハンケル変換のインスタンスを生成する。

`int gsl_dht_init` (*gsl_dht* * **`t`**, *double* **`nu`**, *double* **`xmax`**) [Function]

与えられる *nu* と *xmax* を使ってインスタンス *t* を初期化する。

`gsl_dht * gsl_dht_new` (*size_t* **`size`**, *double* **`nu`**, *double* **`xmax`**) [Function]

与えられる *nu* と *xmax* を使ってインスタンス *t* を初期化する。

`void gsl_dht_free` (*gsl_dht* * **`t`**) [Function]

インスタンス *t* のメモリを解放する。

`int gsl_dht_apply` (*const gsl_dht* * **`t`**, *double* * **`f_in`**, *double* * **`f_out`**) [Function]

インスタンス *t* に設定されているのと同じサイズの配列 *f_in* に対して変換を行う。変換の結果はやはり同じ大きさの配列 *f_out* に入れられる。

`double gsl_dht_x_sample` (*const gsl_dht* * **`t`**, *int* **`n`**) [Function]

長さ 1 のサンプリング区間 $[0,1]$ 内での n 番目のサンプリング・ポイントでの値 $j_{\nu,n+1}/j_{\nu,M}X$ を返す。これが $f(t)$ のサンプリング・ポイントであると考えられる。

`double gsl_dht_k_sample` (*const gsl_dht* * **`t`**, *int* **`n`**) [Function]

「k-空間 (k-space)」内での n 番目のサンプリング・ポイントでの値 $j_{\nu,n+1}/X$ を返す。

31.3　参考文献

変換アルゴリズムは以下の論文に記述されている。

- H. Fisk Johnson, “An improved method for computing a discrete Hankel transform”, *Computer Physics Communications*, **43**(2), pp. 181–202 (1987).
- Didier Lemoine, ”The discrete Bessel transform algorithm”, *Journal of Chemical Physics*, **101**(5), pp. 3936–3944 (1994).

第32章　一次元関数の求根法

この章では、任意の一次元関数 (one-dimentional function) に対する求根法 (root finding method) のルーチンに関して説明する。GSL には根を求めるための繰り返し計算法とその収束の判定法について、低レベルのルーチン (low level component、アルゴリズムとして抽象化レベルが低いという意味) をそれぞれ複数用意している。これらを適切に組み合わせてプログラムを作成することで、計算の各ステップの状況を常に確認しながら目的の解を得ることができる。それぞれ同じフレームワークを使用しており、実行時にアルゴリズムを切り替えることができる。その際プログラムの再コンパイルは不要である。求根法の各インスタンスは探索点を各々で常に保持しており、マルチスレッド対応のプログラミングができる。

求根法の関数などのプロトタイプ宣言はヘッダファイル 'gsl_roots.h' にある。

32.1　概要

一次元求根法のアルゴリズムは囲い込み法 (root bracketing) と漸近法 (root polishing) の二種類に大別される。囲い込み法は収束を保証する。囲い込み法ではまず、根を含むことがすでに分かっている有限の区間を指定する。繰り返し計算でその区間をだんだんと狭く絞っていき、十分に絞り込んだところで終了する。したがって根のある場所について、誤差の大きさを正確に評価できる。

漸近法は根の推定値を、初期値 (区間ではなく、ある一つの値) から改善していく方法である。この方法は初期値が解に「十分に近い」場合にのみ収束する。誤差評価は精密にはできないが収束は速く、根の近傍で目的関数の形を近似することで、初期値から根に高次収束する (誤差の減少のしかたが、繰り返し計算の回数の 1 乗よりも大きい) 。関数の形が想定される性質を持ち、かつよい初期値が与えられれば、漸近法は非常に高速に収束する。

繰り返し計算は主に以下の三段階からなる。

- 求根法 T を指定して、求根インスタンス s を初期化する。
- T による繰り返し計算を一回行って s を更新する。
- s の収束を判定し、必要なら繰り返し計算を続ける。

GSL ではこの各段階について、それぞれ独立した関数を用意しており、これらを使って高レベル (抽象化レベルが高い) の求根ルーチンを書くことができる。GSL では囲い込み法と漸近法の両方を同じフレームワークで使える。

囲い込み法での求根法ルーチンの囲い込み区間は gsl_root_fsolver 構造体に保持されている。区間の更新には関数評価のみを用いる (導関数は使わない)。漸近法による求根法ルーチンの探索

点は gsl_root_fdfsolver 構造体に保持されている。更新には、別途定義する関数とその導関数 (関数名 fdf) を用いる。

32.2　注意点

どの求根法も一度に一つの根しか求められない。探索範囲に複数の根がある場合、最初に見つかる根が解として返されるが、どの根が最初に見つかるかを前もって予測するのは困難である。見つかった根以外に根があったとしても、ほとんどの場合、何のエラーも出ない。

重根を持つ関数の場合も注意を要する。例えば $f(x) = (x - x_0)^2$ または $f(x) = (x - x_0)^3$ のような場合である。囲い込み法は偶重根 (even-multiplicity root) には使えない。囲い込み法では、最初の囲い込み区間には関数と x 軸との交点があり、区間の一端では関数値が負、もう一方では正であることが必要だが、根が偶重根の場合は関数は x 軸と交わらず、接するだけである。奇重根 (三次、五次、...) の場合なら囲い込み法を使うことができる。漸近法は一般の高次の重根の場合にも使うことができるが、収束は遅くなる。その場合にはステフェンソンの方法 (Steffenson algorithm) を使えば収束を加速することができる。

探索範囲内に f の根が絶対に存在しなければならない、というわけではない (条件によっては、根がなくても求根法ルーチンがエラーを返さないこともあり得る)。したがって、根が存在するか否かを知るために数値的な求根法を使うべきではない。それにはよい方法が他にある。数値解法が予想もしない結果に終わることはよくあることであり、特性があまり理解できないような問題に対して求根法をとりあえず適用してみる、といったことは避けるべきである。一般的に、根を探す前にまず関数をプロットしてみて、画像で見てみるとよい。

32.3　求根法インスタンスの初期化

gsl_root_fsolver * gsl_root_fsolver_alloc (*const gsl_root_fsolver_type * T*) [Function]

求根法 T のインスタンスを生成して、そのポインタを返す。たとえば以下のコードでは二分法 (bisection method) のインスタンスが作られる。

```
const gsl_root_fsolver_type * T = gsl_root_fsolver_bisection;
gsl_root_fsolver * s = gsl_root_fsolver_alloc (T);
```

インスタンスを作るのに十分な大きさのメモリが確保できない場合は NULL ポインタが返され、エラーコード GSL_ENOMEM でエラーハンドラーが呼ばれる。

gsl_root_fdfsolver * gsl_root_fdfsolver_alloc (*const gsl_root_fdfsolver_type * T*) [Function]

勾配法 (gradient method) T のインスタンスを生成し、そのポインタを返す。以下のコードではニュートン・ラフソン法 (Newton-Raphson method) のインスタンスを生成する。

```
const gsl_root_fdfsolver_type * T = gsl_root_fdfsolver_newton;
gsl_root_fdfsolver * s = gsl_root_fdfsolver_alloc (T);
```

インスタンスを作るのに十分な大きさのメモリが確保できない場合は NULL ポインタが返され、エラーコード GSL_ENOMEM でエラーハンドラーが呼ばれる。

int gsl_root_fsolver_set (*gsl_root_fsolver* * **s**, *gsl_function* * **f**, *double* **x_lower**, *double* **x_upper**) [Function]

既に生成されているインスタンス s を関数 f に適用するために初期化し (再初期化もできる)、探索区間の初期値を [*x_lower*, *x_upper*] に設定する。

int gsl_root_fdfsolver_set (*gsl_root_fdfsolver* * **s**, *gsl_function fdf* * **fdf**, *double* **root**) [Function]

既に生成されているインスタンス s を関数および導関数 *fdf* に適用するために初期化し (再初期化もできる)、探索点の初期値を *root* に設定する。

void gsl_root_fsolver_free (*gsl_root_fsolver* * **s**) [Function]
void gsl_root_fdfsolver_free (*gsl_root_fdfsolver* * **s**) [Function]

インスタンス s に割り当てられているメモリを解放する。

const char * gsl_root_fsolver_name (*const gsl_root_fsolver* * **s**) [Function]
const char * gsl_root_fdfsolver_name (*const gsl_root_fdfsolver* * **s**) [Function]

与えられたインスタンスが使っている求根法の名前文字列へのポインタを返す。例えば

```
printf ("s is a '%s' solver\n", gsl_root_fsolver_name (s));
```

では `s is a 'bisection' solver` のように出力する。

32.4 目的関数の設定

求根法のインスタンスに対して、根を求めたい目的関数として一変数の連続関数を、求根法によってはさらに一階導関数を与えなければならない。関数は以下の型で定義する必要がある。

gsl_function [Data Type]

以下のメンバーを持つ、数学関数を定義するための汎用の型 (構造体) で、パラメータを使って関数を定義できる。

```
double (* function) (double x, void * params)
```
この関数は、引数 x、パラメータ *params* のときの関数値 $f(x, params)$ を返す。

```
void * params
```
関数のパラメータへのポインタ。

パラメータのある一般的な二次関数の例を以下に示す。

$$f(x) = ax^2 + bx + c$$

パラメータは $a = 3,\ b = 2,\ c = 1$ とする。これを求根法インスタンスに渡す場合、関数 gsl_function F は以下のように定義しておく。

```
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;
```

関数値 $f(x)$ は以下のマクロで評価することができる。

```
#define GSL_FN_EVAL(F,x) (*((F)->function))(x,(F)->params)
```

gsl_function_fdf [Data Type]

この型はパラメータで記述される一般的な関数と、その一階導関数を定義するのに使う。以下のメンバーを持つ構造体である。

`double (* f) (double x, void * params)`
: 引数 x、パラメータ $params$ での関数値 $f(x, params)$ を返す。

`double (* df) (double x, void * params)`
: 引数 x、パラメータ $params$ での f の x に関する導関数の値 $f'(x, params)$ を返す。

`void (* fdf) (double x, void * params, double * f, double * df)`
: この関数は引数が x、パラメータが $params$ の時の関数 $f(x, params)$ の値を f に、その導関数 $f'(x, params)$ の値を df に代入する。$f(x)$ および $f'(x)$ を同じルーチン内で計算することで重複する計算を省くことができる場合は、それぞれ別の関数を同時に呼び出すよりも、この関数を使った方が速く実行できる。

`void * params`
: 関数のパラメータへのポインタ。

以下に、$f(x) = \exp(2x)$ の例を示す。

```
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; /* 計算してある値を再利用 */
}
FDF.f = &my_f;
FDF.df = &my_df;
FDF.fdf = &my_fdf;
FDF.params = 0;
```

関数 $f(x)$ の値を計算するために、以下のマクロが定義されている。

```
#define GSL_FN_FDF_EVAL_F(FDF,x) (*((FDF)->f))(x,(FDF)->params)
```

導関数 $f'(x)$ の値は以下のマクロ使って計算できる。

```
#define GSL_FN_FDF_EVAL_DF(FDF,x) (*((FDF)->df))(x,(FDF)->params)
```

また以下のマクロで、関数 $y = f(x)$ とその導関数 $dy = f'(x)$ の値を同時に計算することができる。

```
#define GSL_FN_FDF_EVAL_F_DF(FDF,x,y,dy)
    (*((FDF)->fdf))(x,(FDF)->params,(y),(dy))
```

このマクロは $f(x)$ を引数 y に、$f'(x)$ を dy に保存する。これらは `double` 型へのポインタでなければならない。

32.5 探索範囲と初期推定

探索をはじめるときに、囲い込み法には探索範囲 (search bound) を、漸近法には初期推定値 (initial guess, 探索開始点) を指定する。初期推定値は単なる x の値で、要求される精度の根の値になるまで、値が繰り返し更新される。これは `double` 型である。

探索範囲は解を含む区間の両端であり、区間の幅が要求される精度 (幅) よりも小さな値になるまで繰り返し計算される。区間は上端と下端の二つの値で定義される。両端の値が区間に含まれるかどうかは、場合によって異なる。

32.6　繰り返し計算

以下の関数が繰り返し計算を実行する。関数はそれぞれ、インスタンスが持つ探索点または探索区間の更新を、求根法による繰り返し計算で一回行う。同じ関数が全ての求根法に使え、プログラムを書き換えることなく、実行時にアルゴリズムを切り替えることができる。

`int gsl_root_fsolver_iterate (`*gsl_root_fsolver * s*`)` [Function]
`int gsl_root_fdfsolver_iterate (`*gsl_root_fdfsolver * s*`)` [Function]

これらの関数は求根法のインスタンス s の繰り返し計算を 1 回行う。計算でなにか予期しない問題が生じた場合は、以下のエラーコードを返す。

`GSL_EBADFUNC`
関数値や導関数値が `Inf` や `NaN` になるような特異点が発生した事を示す。

`GSL_EZERODIV`
探索点で導関数値が 0 になり、零除算により計算を続けられなくなった事を示す。

求根法のインスタンスは探索中の各時点での最良な根の推定値を常に保持している。囲い込み法のインスタンスは、根を含む最良区間も保持している。これらは以下の補助的な関数で参照することができる。

`double gsl_root_fsolver_root (`*const gsl_root_fsolver * s*`)` [Function]
`double gsl_root_fdfsolver_root (`*const gsl_root_fdfsolver * s*`)` [Function]

これらの関数は求根法のインスタンス s が持つ現時点での根の推定値を返す。

`double gsl_root_fsolver_x_lower (`*const gsl_root_fsolver * s*`)` [Function]
`double gsl_root_fsolver_x_upper (`*const gsl_root_fsolver * s*`)` [Function]

これらの関数は求根法のインスタンス s が現時点までに囲い込んだ区間を返す。

32.7　停止条件

求根法は、以下の条件のいずれかが真になった時に停止する。

- 設定された精度で根が見つかったとき。
- 設定された最大回数に繰り返し計算が達したとき。
- エラーが発生したとき。

これらの条件は任意に設定することができる。以下の関数でその時の最良探索点とその精度を調べることができる。

`int gsl_root_test_interval` (*double* **`x_lower`**, *double* **`x_upper`**, *double* **`epsabs`**, *double* **`epsrel`**) [Function]

この関数は指定される絶対誤差 *epsabs* と相対誤差 *epsrel* を使って区間 [*x_lower*, *x_upper*] の収束を判定し、以下の条件が満たされている時 `GSL_SUCCESS` を返す。

$$|a - b| < epsabs + epsrel\ \min(|a|, |b|)$$

ここで $x = [a, b]$ は原点を含まないものとする。区間内に原点が含まれる場合は $\min(|a|, |b|)$ は 0（その区間上での $|x|$ の最小値）で置き換えられる。これにより、原点に近い根の相対誤差を正確に得ることができる。

探索区間内でこの条件が成り立つことは、真の根 r^* が探索区間内にあるとき、根の推定値 r は真の根 r^* に対して以下の条件を満たすということである。

$$|r - r^*| < epsabs + epsrel\ r^*$$

`int gsl_root_test_delta` (*double* **`x1`**, *double* **`x0`**, *double* **`epsabs`**, *double* **`epsrel`**) [Function]

この関数は絶対誤差が *epsabs* で相対誤差が *epsrel* のときの、更新されてきた根の推定値がなす数列の、最新の 2 個 x_0, x_1 から収束を判定する。この関数は以下の条件が真になっているとき `GSL_SUCCESS` を返す。そうでないときには `GSL_CONTINUE` を返す。

$$|x_1 - x_0| < epsabs + epsrel\ |x_1|$$

`int gsl_root_test_residual` (*double* **`f`**, *double* **`epsabs`**) [Function]

この関数は許容絶対誤差 *epsabs* に対する残差 f を判定する。この関数は以下の条件が真になっているときに `GSL_SUCCESS` を返す。

$$|f| < epsabs$$

条件が満たされていないときには `GSL_CONTINUE` を返す。この判定基準は、残差 $|f(x)|$ が十分に小さくなればよく、根 x の正確な位置はあまり重要ではないような場合に使うとよい。

32.8 囲い込み法

この節で述べる囲い込み法では、最初に指定する探索区間内に必ず根があることが必要である。さらに、a と b を区間の両端とするとき、$f(a)$ と $f(b)$ の符号が異なっていなければならない。つまり関数が少なくとも一回は 0 になるということである。この条件が成立していれば、関数の挙動が特殊 (特異点があるなど) でない限り囲い込み法は成功する。

なお、囲い込み法は偶数次の重根を見つけることはできない。関数が x 軸と接するだけで交わらないからである。

`gsl_root_fsolver_bisection` [Solver]

二分法 (bisection algorithm) は、囲い込み法のうちで最も単純な方法である。GSL で提供する手法の中ではもっとも遅く、収束は線形 (linear convergence) である。

繰り返し計算の各回では、探索区間が二等分され、区間の中点での関数値が計算される。この値の符号により根を含んでないのは二等分された区間のどちらであるかを決める。含んでいない方は捨てられ、根を含んでいる方が新たな、より狭い探索区間となる。この操作が区間が十分に狭くなるまで続けられる。

各時点での根の推定値は、その時の探索区間の中点である。

`gsl_root_fsolver_falsepos` [Solver]

はさみうち法 (false position algorithm, 羅 regula falsi) は線形補間 (linear interpolation) を使った求根法である。収束は線形だが、二分法よりも速いことが多い。

繰り返し計算の各回では、区間の両端の点 $(a, f(a))$ と $(b, f(b))$ を線分で結び、その線分と x 軸との交点での関数値を計算し、その符号から交点のどちら側に根があるかを決める。根を含まない方の区間は捨てられ、残りが新しい、より狭い探索区間となる。この操作が区間の幅が十分に狭くなるまで続けられる。

各時点での根の推定値は、その時の探索区間の両端を線形に補間して決められる。

`gsl_root_fsolver_brent` [Solver]

ブレントの方法 (Brent-Dekker method, 単にブレントの方法 Brent's method とも呼ばれる) もやはり二分法と補間を組み合わせたものである。収束が早く、かつロバストである。

ブレントの方法では、繰り返し計算の各回で関数を補間曲線で近似する。最初の一回は、探索区間の両端を線形補間する。続く計算では、最新の 3 点を使って二次式 (quadratic curve) で補間するため、より精度が高い。この二次曲線と x 軸との交点を根の近似値とする。もしその点が探索区間内であればそれを採用し、その点を端点とする小さな区間を新たな探索区間とする。採用しない場合には、普通の二分法による区間の更新を行う。

各時点での根の推定値は、直前の補間、あるいは二分法による値である。

32.9　導関数を使う方法

この節の方法は漸近法であり、どれも根の初期推定値を必要とする。必ず根に収束するという保証はなく、対象となる関数の形がこの方法に適していてかつ根の初期推定値が十分に真の根に近くないと、この方法はうまくいかない。しかしこれらの条件が満たされていれば、収束は早い (二次収束、quadratic convergence である)。

これらの方法では、関数とその導関数の両方を使う。

`gsl_root_fdfsolver_newton` [Solver]

ニュートン法は、代表的な漸近法の一つである。この方法は根の初期推定値からはじめ、繰り返し計算の各回でそのときの探索点で関数 f に接線を引く。この接線が x 軸と交わる点を新たな推定値とする。繰り返し計算は以下のように定義される。

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

ニュートン法は、単根の場合は二次収束、重根に対しては線形収束 (一次収束) である。

`gsl_root_fdfsolver_secant` [Solver]

割線法 (secant method)。微分係数の代わりに差分を使うことでニュートン法を簡略化したものであり、したがって繰り返し計算の各回での導関数の計算が必要ない。

探索の最初の回では、ニュートン法と同様に導関数値を用いて以下のように計算する。

$$x_1 = x_0 \frac{f(x_i)}{f'(x_i)}$$

これに続く繰り返し計算では、導関数値の代わりに直前の二点間を結ぶ直線の傾きを使うことで、導関数の計算を避ける。

$$x_{i+1} = x_i - \frac{f(x_i)}{f'_{est}} \quad \text{where} \quad \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

根に近いところでは導関数値が大きく変動しないことが多く、割線法による時間の節約の効果は大きい。割線法がニュートン法よりも速くなるのは、おおよそ、導関数値の計算にかかる時間が関数値にかかる時間の 0.44 倍よりも大きなときである。一般的に、他の数値微分の計算と同様、二点間の距離が小さくなりすぎると、桁落ちの影響を受ける。

重根でない場合、収束の速さ (オーダー) は $(1 + \sqrt{5})/2$ (約 1.62) である。重根の場合には線形収束である。

`gsl_root_fdfsolver_steffenson` [Solver]

ステフェンセン (Johan Frederik Steffensen) の方法。ここに挙げるルーチンの中では最も速い。これは基本的なニュートン法とエイトケン (Alexander Craig Aitken) の 「デルタ二乗 (delta-squared)」加速法を使う。ニュートン法の繰り返し計算の各回を x_i とするとき、エイトケンの加速法ではそれとは別に数列 R_i を計算する。

$$R_i = x_i - \frac{(x_{i+1} - x_i)^2}{x_{i+2} - 2x_{i+1} + x_i}$$

条件がよければ、R_i はもとの x_i よりも速く収束する。R_i を計算するのにはまず項が三つ必要なので、加速されるのは二回目以降の繰り返し計算である。一回目はニュートン法と同じである。加速項の分母が 0 になる場合はニュートン法と同じ値を返す。

他の全ての加速法と同様、この方法も関数の形によっては安定でないことがある。

32.10　例

どの求根法にも対象となる関数を用意せねばならないが、ここでは前述の、ごく一般的な二次関数を例として用いる。以下の定義はヘッダファイル (‘demo_fn.h’) に用意されている。

```
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);
```

求根法で使うための関数定義は別のファイル (‘demo_fn.c’) に記述してある。

```
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;
    double c = p->c;
    return 2.0 * a * x + b;
}

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;
}
```

最初のプログラムは以下の方程式を解くために、ブレントの方法 gsl_root_fsolver_brent と上述の二次関数を適用したものである。

$$x^2 - 5 = 0$$

この式の根は $x = \sqrt{5} = 2.236068\ldots$ である。

```
#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;
    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;
    }
```

実行すると、以下のように出力される。

```
$ ./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
```

gsl_root_fsolver_brent を gsl_root_fsolver_bisection に書き換えて、ブレントの方法の代わりに二分法を使うようにして比べると、二分法の収束がどの程度遅いかが分かる。

```
$ ./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
```

次のプログラムは、同じ関数の求根に、導関数も使っている例である。

```
#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_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);
        } while (status == GSL_CONTINUE && iter < max_iter);

        gsl_root_fsolver_free(s);
        return status;
    }
```

ニュートン法による結果は以下のようになる。

```
$ ./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
```

近似誤差は現時点および前回での値から計算されているが、これを現時点と次の計算から求めると、より正確になる。また、gsl_root_fdfsolver_newton を gsl_root_fdfsolver_secant や gsl_root_fdfsolver_steffenson に代えることで、他の勾配法も試すことができる。

32.11　参考文献

ブレントの方法については、以下の文献を参照のこと。

- Richard P. Brent, “An algorithm with guaranteed convergence for finding a zero of a function”, *Computer Journal*, **14**(4), pp. 422–425 (1971).
- 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*, **1**(4), pp. 330–345 (1975).

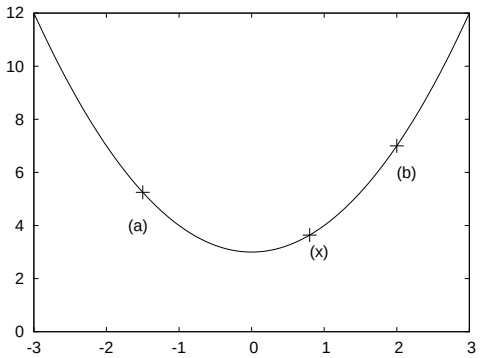
第 33 章　一次元関数の最適化

この章では、任意の一次元関数 (univariate/one-dimentional function) に対して最小化 (minimization、関数値が最小となる独立変数値を探索すること) を行うルーチンに関して説明する。GSL には最適化を行う繰り返し計算法とその収束の判定法について、低レベルのルーチン (low level component、アルゴリズムとして抽象化レベルが低いという意味) をそれぞれ複数用意している。これらを適切に組み合わせてプログラムを作成することで、計算の各ステップの状況を常に確認しながら目的の解を得ることができる。それぞれ同じフレームワークを使用しており、実行時にアルゴリズムを切り替えることができる。その際プログラムの再コンパイルは不要である。最適化の各インスタンスは探索点を各々で常に保持しており、マルチスレッド対応のプログラミングができる。

最小化関数と関連する各種宣言はヘッダファイル '`gsl_min.h`' にある。関数の最大値を求めたいときは、目的関数の符号を反転して最小化を行えばよい。

33.1　概要

最小化法ではまず最初に、すでに最小値を含むことが分かっている区間を探索領域として指定しなければならない。区間は下端 a と上端 b で表され、関数の最小値を与える場所 (最小点) の推定値を x で表す。



x での関数値は以下のように、探索区間の両端での関数値よりも小さくなければならない。

$$f(a) > f(x) < f(b)$$

この条件を満たしていれば、探索区間の中のどこかに最小値があることが保証される。GSL で用意している複数の囲い込み法はいずれも、繰り返し計算を行うたびにそれぞれの方法で新しい探索点 x' を決定していく。新しい探索点で元の点よりも関数値が小さくなる、つまり $f(x') < f(x)$ と

なる場合に最小点の推定値 x を x' で置き換える。また新しい x を $f(a) > f(x) < f(b)$ となるように選ぶときに、その区間 $[a, b]$ 区間の幅をより狭くすることができれば、探索区間をそれだけ絞り込むことができる。探索区間の幅は、真の最小値を囲む幅が十分に狭くなるまで縮小される。最小点の推定値はその区間の中にあり、推定誤差はその区間の幅そのものである。したがって最小点の推定値に対する誤差が精密に把握できる。

繰り返し計算は主に以下の三段階からなる。

- 最小化法 T の探索点 s を初期化する。
- T の繰り返し計算を使って s を更新する。
- s の収束を判定し、必要なら繰り返し計算を続ける。

GSL ではこの各段階について、それぞれ独立した関数を用意しており、これらを使って高レベルの (抽象化レベルが高い) 最小化ルーチンを書くことができる。

探索点などの情報は `gsl_min_fminimizer` 構造体に保持される。この更新には関数値のみが用いられる (導関数値は使われない)。

33.2　注意点

ここにある最小化関数は、一度に一つの最小値しか探すことはできない。探索区間内に複数の極小点がある場合、最初に見つかる極小点が解として返されるが、どの極小点が最初に見つかるかを前もって予測するのは困難である。見つかった極小点以外に極小点があったとしても、ほとんどの場合、何のエラーも出ない。

どの最小化法を使っても、数値計算として可能な最高の精度で最小点を得ることは困難である。たとえば最小点 x^* の近傍での関数の挙動は、テイラー展開

$$y = f(x^*) + \frac{1}{2} f''(x^*)(x - x^*)^2$$

を使って近似できるが、数値の精度には限りがあるため、第一項に第二項を加えても第一項の値からほとんど変化しないような場合がある。これにより x^* を探索する際に $\sqrt{\epsilon}$ に比例して誤差が拡大する (ϵ は浮動小数点の相対的精度) 。たとえば x^4 のような高次の関数の最小値を探索するときには、誤差はさらに拡大する。こういった場合は最小点の座標ではなく、関数値を収束させるとよい。

33.3　最小化インスタンスの初期化

`gsl_min_fminimizer * gsl_min_fminimizer_alloc` (*const gsl_min_fminimizer_type * T*) [Function]

最小化法 T のインスタンスを生成し、そのインスタンスへのポインタを返す。以下の例では黄金分割法のインスタンスを生成する。

```
const gsl_min_fminimizer_type * T = gsl_min_fminimizer_goldensection;
gsl_min_fminimizer * s = gsl_min_fminimizer_alloc (T);
```

インスタンスを生成するためのメモリが足りない場合は NULL ポインタを返し、エラーコード GSL_ENOMEM でエラーハンドラーを呼び出す。

int gsl_min_fminimizer_set (*gsl_min_fminimizer* * **s**, *gsl_function* * **f**, *double* **x_minimum**, *double* **x_lower**, *double* **x_upper**) [Function]

すでに生成されている最小化のインスタンス *s* に、関数 *f*、探索区間 [*x_lower*, *x_upper*]、最小点の初期推定 (探索開始点) *x_minimum* を設定、あるいは再設定する。

この関数の内部では *x_minimum*、*x_lower*、*x_upper* における目的関数の値を計算する。そのとき、*x_minimum* における目的関数値が区間の両端での関数値よりも小さくなかった場合、エラーコード GSL_INVAL を返す。

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**) [Function]

gsl_min_fminimizer_set とほぼ同じだが、f(x_minimum)、f(x_lower)、f(x_upper) を計算する代わりに引数で与える *f_minimum*、*f_lower*、*f_upper* を用いる。

void gsl_min_fminimizer_free (*gsl_min_fminimizer* * **s**) [Function]

この関数は最小化法のインスタンス *s* に割り当てられたメモリを解放する。

const char * gsl_min_fminimizer_name (*const gsl_min_fminimizer* * **s**) [Function]

この関数は、与えられたインスタンスが使っている最小化法の名前文字列へのポインタを返す。たとえば以下の文は、s is a 'brent' minimizer のように出力する。

```
printf ("s is a '%s' minimizer\n", gsl_min_fminimizer_name (s));
```

33.4 目的関数の設定

最小化される目的関数として、一変数の連続関数を設定しなければならない。パラメータで一般的に目的関数を定義できるようにするため、目的関数は gsl_function 型として定義する必要がある (32.4 節「目的関数の設定」参照)。

33.5 繰り返し計算

以下の関数が繰り返し計算を実行する。関数はそれぞれ、インスタンスが持つアルゴリズムによる繰り返し計算による探索点の更新を一回行う。同じ関数が全てのアルゴリズムに使え、プログラムを書き換えることなく、実行時にアルゴリズムを切り替えることができる。

int gsl_min_fminimizer_iterate (*gsl_min_fminimizer* * **s**) [Function]

この関数は最小化法のインスタンス s の繰り返し計算を一回行う。計算で何か予期しない問題が生じたときは、以下のエラーコードを返す。

`GSL_EBADFUNC` 関数値が `Inf` や `NaN` になるような特異点が生じた事を示す。

`GSL_FAILURE` 現在の探索点よりも良い点が見つからなかった事を示す。

最小化インスタンスは常に、現時点での最小点を含む探索区間と最良解の近似を保持している。これらは以下の関数を使って参照できる。

`double gsl_min_fminimizer_x_minimum` (*const gsl_min_fminimizer* * **`s`**) [Function]

この関数は最小化法インスタンス s の現時点の最小点の近似推定値を返す。

`double gsl_min_fminimizer_x_upper` (*const gsl_min_fminimizer* * **`s`**) [Function]
`double gsl_min_fminimizer_x_lower` (*const gsl_min_fminimizer* * **`s`**) [Function]

これらの関数は最小化法インスタンス s の現時点での探索区間の上端、下端を返す。

`double gsl_min_fminimizer_f_minimum` (*const gsl_min_fminimizer* * **`s`**) [Function]
`double gsl_min_fminimizer_f_upper` (*const gsl_min_fminimizer* * **`s`**) [Function]
`double gsl_min_fminimizer_f_lower` (*const gsl_min_fminimizer* * **`s`**) [Function]

これらの関数は最小化法インスタンス s の現時点での近似最小点、探索区間の上端、下端での関数値を返す。

33.6　停止条件

最小化アルゴリズムは、以下の条件のいずれかが真になったときに停止する。

- 設定された精度で最小点が見つかったとき。
- 設定された最大回数に繰り返し計算が達したとき。
- 何らかのエラーが発生したとき。

これらの条件は任意に設定することができる。以下の関数でその時の最良探索点とその精度を調べることができる。

`int gsl_min_test_interval` (*double* **`x_lower`**, *double* **`x_upper`**, *double* **`epsabs`**, *double* **`epsrel`**) [Function]

この関数は指定される絶対誤差 *epsabs* および相対誤差 *epsrel* を使って区間 [*x_lower*, *x_upper*] の収束を判定し、以下の条件が満たされている時 `GSL_SUCCESS` を返す。

$$|a - b| < epsabs + epsrel\ \min(|a|, |b|)$$

ここで区間 $x = [a, b]$ は原点を含まないものとする。区間中に原点が含まれる場合、$\min(|a|, |b|)$ が 0 (その区間上での $|x|$ の最小値) で置き換えられる。これにより、原点に近い場所での相対誤差を正確に得ることができる。

探索区間内でこの条件が成り立つことは、真の最小点 x_m^* が探索区間内にあるとき、近似最小点 x_m は x_m^* に対して以下の条件を満たすということである。

$$|x_m - x_m *| < epsabs + epsrel\ x_m*$$

33.7 最小化アルゴリズム

GSL で用意している最小化法では、最初に与える探索区間内に最小値が含まれていることを示す必要がある。つまり探索区間の両端を a, b、最小点の初期推定値を x とするとき、$f(a) > f(x) < f(b)$ である。これにより、探索区間内に最小点が存在することが保証される。初期探索区間が上記を満たせば、対象となる関数が特異な形でない限りは、ここにある方法で最小点が得られる。

`gsl_min_fminimizer_goldensection` [Minimizer]

黄金分割法 (golden section algorithm) は関数の最小値を囲い込むもっとも単純な方法である。収束は線形で、GSL にある方法の中では最も遅い。

この方法は繰り返し計算の各回で、区間 $[a, x]$ と $[x, b]$ を比較する。二つのうち広い方の区間を黄金比で分割し (よく知られている比 $(3 - \sqrt{5})/2 = 0.3819660\ldots$ である)、新しく決めた点 x' での関数値を計算する。そして区間両端の点 a、b のどちらかを捨て、残った三点を新しい区間と最小点の推定値とする。その際、残った三点で $f(a') > f(x') < f(b')$ という条件が満たされるように捨てる点を選ぶ。これを探索区間が十分に小さくなるまで繰り返す。区間を二分割する比に黄金比を使うことで、この種の方法としては最も速い収束を示す。

`gsl_min_fminimizer_brent` [Minimizer]

ブレント (Richard Peirce Brent) の最小化法は放物線による補間と黄金分割法を組み合わせたものである。高速でかつ、ロバストである。

おおまかまとめると以下のような方法である。繰り返し計算の各回で、与えられる三点から放物線で関数を近似する。放物線の最小点が関数の最小点の近似となる。近似最小点が探索区間内にあればそれを使って区間を縮小する。そうでなければ黄金分割法を行う。ブレントの方法ではこれに加え、収束を改善するためにいくつかのチェックが行われている。

33.8 例

以下のプログラムでは、ブレントの方法で関数 $f(x) = \cos(x) + 1$ の最小点 $x = \pi$ を求める。初期探索区間は $(0, 6)$、最小点の初期推定は 2 である。

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_min.h>

double fn1 (double x, void * params)
{
    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;
    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 %.7f\n",
               iter, a, b, m, m_expected, m-m_expected, b-a);
```

```
    } while (status == GSL_CONTINUE && iter < max_iter);

    gsl_min_fminimizer_free(s);
    return status;
}
```

以下に最小化の様子を示す。

```
$ ./a.out
    0 [0.0000000, 6.0000000] 2.0000000 -1.1415927 6.0000000
    1 [2.0000000, 6.0000000] 3.2758640 +0.1342713 4.0000000
    2 [2.0000000, 3.2831929] 3.2758640 +0.1342713 1.2831929
    3 [2.8689068, 3.2831929] 3.2758640 +0.1342713 0.4142862
    4 [2.8689068, 3.2831929] 3.2758640 +0.1342713 0.4142862
    5 [2.8689068, 3.2758640] 3.1460585 +0.0044658 0.4069572
    6 [3.1346075, 3.2758640] 3.1460585 +0.0044658 0.1412565
    7 [3.1346075, 3.1874620] 3.1460585 +0.0044658 0.0528545
    8 [3.1346075, 3.1460585] 3.1460585 +0.0044658 0.0114510
    9 [3.1346075, 3.1460585] 3.1424060 +0.0008133 0.0114510
   10 [3.1346075, 3.1424060] 3.1415885 -0.0000041 0.0077985
Converged:
   11 [3.1415885, 3.1424060] 3.1415927 -0.0000000 0.0008175
```

33.9 参考文献

ブレントの方法については、以下の本が参考になる。

- Richard P. Brent, *Algorithms for minimization without derivatives*, Prentice-Hall, Englewood Cliffs, NJ (1973).
  Dover Publications, Mineola, New York, ISBN 0-486-41998-3 (2002, 再版).

第34章　多次元関数の求根法

この章では、多次元関数の求根法 (multi-dimentional root finding、n 個の未知変数を含む n 個の方程式からなる非線形系の解法) について説明する。GSL には根を求めるための繰り返し計算法とその収束の判定法について、低レベルのルーチン (low level component、アルゴリズムとして抽象化レベルが低いという意味) をそれぞれ複数用意している。これらを適切に組み合わせてプログラムを作成することで、計算の各ステップの状況を常に確認しながら目的の解を得ることができる。それぞれ同じフレームワークを使用しており、実行時にこれらのメソッドを切り替えることができる。その際プログラムの再コンパイルは不要である。求根法の各インスタンスは探索点を常に各々で保持しており、マルチスレッドに対応している。ここに用意されている求根法は FORTRAN で書かれたライブラリ MINPACK が元になっている。

多次元求根法の関数などのプロトタイプ宣言はヘッダファイル '`gsl_multiroots.h`' にある。

34.1　概要

多次元関数の求根においては、以下のような n 個の変数 x_i についての n 個の方程式 f_i を同時に解く必要がある。

$$f_i(x_1, \ldots, x_n) = 0 \quad \text{for } i = 1 \ldots n$$

一般的に、n 次元空間では囲い込み法 (bracketing method) は使えず、解が存在するかどうかを確認する方法もない。どの手法でも根の初期推定値 (initial guess) から、下のようにニュートン法 (Newton's method) のような繰り返し計算で探索を進める。

$$x \to x' = x - J^{-1} f(x)$$

ここで x と f はベクトル、J はヤコビアン行列 (Jacobian matrix) $J_{ij} = \partial f_i / \partial x_j$ である。各種のアルゴリズムでは、これを元に根に収束できる探索点の範囲を広げる工夫を行う。改良にはニュートン法の各ステップでノルム $|f|$ の縮小を図ることや、$|f|$ の勾配がもっとも急 (steepest descent) な負の値になるように探索方向をとろうとする方法がある。

求根を行う繰り返し計算は主に以下の三段階からなる。

- 求根法 T の探索点 s を初期化する。
- T の繰り返し計算を一回行って s を更新する。
- s の収束を判定し、必要なら更新を繰り返す。

GSL ではこの各段階について、それぞれ独立した関数を用意しており、利用者はこれらを使って高レベルの (抽象化レベルが高い) 求根ルーチンを書くことができる。GSL では複数のアルゴリズムを同じフレームワークで使える。

ヤコビアン行列の計算は、微分係数の計算が事実上不可能であったり、または行列の n^2 個の項を計算することに非現実的な時間がかかったりすることから、よく問題になる。このためこのライブラリでは求根法を、導関数を使うか使わないかで二種類に分けている。

導関数 (ヤコビアン行列) を使うアルゴリズムの場合、解析的に計算されたヤコビアン行列と現時点の探索点は gsl_multiroot_fdfsolver 構造体に保持される。探索点の更新には関数と導関数が必要であり、それは別途定義せねばならない。

導関数を解析的に計算しなくてもよい求根法の探索点は gsl_multiroot_fsolver 構造体に保持される。これには関数値のみがあればよい。これらの方法ではヤコビアン行列 J またはその逆行列 J^{-1} は近似法により計算される。

34.2　求根法インスタンスの初期化

以下の関数で、導関数を使う求根法と使わない求根法の両方を初期化できる。求根法のインスタンス自体は探索空間の次元数と、どの求根法を用いるかだけに依存し、生成後にそのまま他の問題に適用し直すことができる。

gsl_multiroot_fsolver * gsl_multiroot_fsolver_alloc (*const gsl_multiroot_fsolver_type * **T***, *size_t* **n**) [Function]

探索空間の次元が n の求根法 T のインスタンスを生成して、そのインスタンスへのポインタを返す。以下のコードは未知変数が 3 個の連立方程式をパウエル (Michael James David Powell) の修正ハイブリッド法 (modified version of Powell's hybrid method) で解くインスタンスを生成する。

```
const gsl_multiroot_fsolver_type * T = gsl_multiroot_fsolver_hybrid;
gsl_multiroot_fsolver * s = gsl_multiroot_fsolver_alloc (T, 3);
```

インスタンスを生成するだけのメモリが確保できない場合、この関数は NULL ポインタを返し、エラーコード GSL_ENOMEM でエラーハンドラーを呼び出す。

gsl_multiroot_fdfsolver * gsl_multiroot_fdfsolver_alloc (*const gsl_multiroot_fdfsolver_t * **T***, *size_t* **n**) [Function]

探索空間の次元が n で、導関数を用いる求根法 T のインスタンスを生成して、そのインスタンスへのポインタを返す。以下のコードは二次の連立方程式を解くニュートン・ラフソン法 (Newton-Raphson method) のインスタンスを生成する。

```
const gsl_multiroot_fdfsolver_type * T = gsl_multiroot_fdfsolver_newton;
gsl_multiroot_fdfsolver * s = gsl_multiroot_fdfsolver_alloc (T, 2);
```

インスタンスを生成するだけのメモリが確保できない場合、この関数は NULL ポインタを返し、エラーコード GSL_ENOMEM でエラーハンドラーを呼び出す。

`int gsl_multiroot_fsolver_set` (*gsl_multiroot_fsolver * **s***, *gsl_multiroot_function * **f***, *gsl_vector * **x***) [Function]
`int gsl_multiroot_fdfsolver_set` (*gsl_multiroot_fdfsolver * **s***, *gsl_multiroot_function fdf * **fdf***, *gsl_vector * **x***) [Function]

既に生成されているインスタンス s を、関数 f または関数と導関数の組 *fdf* の求根に、探索開始点を x として適用するために初期化、あるいは再設定する。x の値は変更されず、この後に求根ルーチンを呼んでもその値は変わらない。

`void gsl_multiroot_fsolver_free` (*gsl_multiroot_fsolver * **s***) [Function]
`void gsl_multiroot_fdfsolver_free` (*gsl_multiroot_fdfsolver * **s***) [Function]

求根法のインスタンス s に割り当てられているメモリを解放する。

`const char * gsl_multiroot_fsolver_name` (*const gsl_multiroot_fsolver * **s***)[Function]
`const char * gsl_multiroot_fdfsolver_name` (*const gsl_multiroot_fdfsolver * **s***) [Function]

指定された求根法インスタンスが使っている求根法の名前文字列へのポインタを返す。以下の例では、`s is a 'newton' solver` のように出力する。

```
printf ("s is a '%s' solver\n", gsl_multiroot_fdfsolver_name (s));
```

34.3 目的関数の設定

求根法のインスタンスに対して、根を求めたい目的関数として n 変数の関数を n 個指定しなければならない。一般的な目的関数をパラメータを用いて定義できるようにするため、目的関数は以下の構造体で定義する必要がある。

`gsl_multiroot_function` [Data Type]

以下のメンバーを持つ、目的関数を定義するための汎用の型 (構造体) で、関数の定義にパラメータを使うことができる。

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`
関数へのポインタで、その関数 $f(x, params)$ は、引数 x とパラメータ *params* を与えて評価したときの結果を引数ベクトル f に入れ、関数値が計算できないときには場合に応じてエラーコードを返すように定義しておく。

`size_t n`
系の次元数。ベクトル x や f の要素の個数。

`void * params`
関数定義に使うパラメータへのポインタ。

以下にパウエルの関数 (Powell's test function) をテストに使った例を示す。パウエルの関数は以下で定義される。

$$f_1(x) = Ax_0x_1 - 1,\ f_2(x) = \exp(-x_0) + \exp(-x_1) - (1 + 1/A)$$

この例では $A = 10^4$ とする。以下のコードは gsl_multiroot_function で系 F を定義している。これはそのまま求根法のインスタンスに渡すことができる。

```
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
}

gsl_multiroot_function F;
struct powell_params params = { 10000.0 };
F.f = &powell;
F.n = 2;
F.params = &params;
```

gsl_multiroot_function_fdf [Data Type]

以下のメンバーを持つ、目的関数を定義するための汎用の型 (構造体) で、パラメータを使って関数系とヤコビアン行列を定義できる。

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`
関数へのポインタで、その関数 $f(x, params)$ は、引数 x とパラメータ $params$ を与えて評価したときの結果を引数ベクトル f に入れ、関数値が計算できないときには場合に応じてエラーコードを返すように定義しておく。

`int (* df) (const gsl_vector * x, void * params, gsl_matrix * J)`
導関数を定義する関数へのポインタである。引数 x、パラメータ $params$ のときに得られる $n \times n$ 行列 $J_{ij} = \partial f_i(x, params)/\partial x_j$ を J に入れ、計算できないときは場合に応じたエラーコードを返すように定義しておく。

```
int (* fdf) (const gsl_vector * x, void * params, gsl_vector * f, gsl_matrix * J)
```

このポインタがさす関数は、引数 x、パラメータ $params$ のときの f と J の値をこの関数の一回の呼び出しで計算し、前二者の関数と同様に代入するように定義しておく。これにより、個別の関数である $f(x)$ と $J(x)$ の両方の値を計算するときに重複する計算を省いて計算時間を短縮することができる。

`size_t n`
系の次元数、ベクトル x や f の要素の個数である。

`void * params`
関数のパラメータへのポインタ。

前の例で定義されたパウエルの関数を、解析的に導関数が計算できるものとして改良したときのコードを以下に示す。

```
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;
FDF.fdf = &powell_fdf;
FDF.n = 2;
FDF.params = 0;
```

powell_fdf はヤコビアンを計算するとき、既に計算されている項の値を再利用することで計算時間を短縮している。

34.4　繰り返し計算

以下の関数は、各求根法の繰り返し計算を実行する。各関数は、インスタンスが持つその時点での探索点をインスタンスが指定されている求根法で更新するための繰り返し計算を一回行う。これらの関数はどの求根法についても使えるため、プログラムを変更しなくても、実行時に求根法を切り換えて使うことができる。

`int gsl_multiroot_fsolver_iterate (`*gsl_multiroot_fsolver * s*`)` [Function]
`int gsl_multiroot_fdfsolver_iterate (`*gsl_multiroot_fdfsolver * s*`)` [Function]

これらの関数は求根法のインスタンス s の繰り返し計算を一回実行する。計算が予期しない問題を生じたときは、以下のエラーコードを返す。

GSL_EBADFUNC
: 計算中に関数値や導関数値が Inf や NaN になる特異点を生じた事を示す。

GSL_ENOPROG
: 計算を行っても解が改善されずアルゴリズムの実行を一時停止している事を示す。

求根法のインスタンスは探索中の各時点での最良な解の推定値を常に s->x に、それに対応する関数値を s->f に保持している。これらは以下の補助的な関数で参照することができる。

`gsl_vector * gsl_multiroot_fsolver_root (`*const gsl_multiroot_fsolver * s*`)`[Function]
`gsl_vector * gsl_multiroot_fdfsolver_root (`*const gsl_multiroot_fdfsolver * s*`)` [Function]

求根法のインスタンス s のその時点での解を返す。その値は s->x でも参照できる。

gsl_vector * gsl_multiroot_fsolver_f (*const gsl_multiroot_fsolver * **s***)[Function]
gsl_vector * gsl_multiroot_fdfsolver_f (*const gsl_multiroot_fdfsolver * **s***)[Function]

求根法のインスタンス s が持つその時点での解における関数値 $f(x)$ を返す。その値は s->f でも参照できる。

gsl_vector * gsl_multiroot_fsolver_dx (*const gsl_multiroot_fsolver * **s***)[Function]
gsl_vector * gsl_multiroot_fdfsolver_dx (*const gsl_multiroot_fdfsolver * **s***)[Function]

インスタンス s が直前にとったステップ幅 dx を返す。その値は s->dx でも参照できる。

34.5 停止条件

求根法は以下の条件のいずれかが成立したときに終了する。

- 設定された精度で根が見つかったとき。
- 設定された最大回数に繰り返し計算が達したとき。
- エラーが発生したとき。

これらの条件は任意に設定することができる。以下の関数でその時の最良探索点とその精度を調べることができる。

int gsl_multiroot_test_delta (*const gsl_vector * **dx***, *const gsl_vector * **x***, *double **epsabs***, *double **epsrel***) [Function]

この関数は、ステップ幅 dx と探索点 x を、指定される絶対誤差 *epsabs* および相対誤差 *epsrel* を使って比較し、収束を判定する。以下の条件が成立していれば GSL_SUCCESS を、そうでなければ GSL_CONTINUE を返す。

$$|dx_i| < epsabs + epsrel\ |x_i|$$

int gsl_multiroot_test_residual (*const gsl_vector * **f***, *double **epsabs***)[Function]

この関数は f と指定される絶対誤差 *epsabs* に対する残差を判定する。以下の条件が成立していれば、この関数は GSL_SUCCESS を返す。

$$\sum_i |f_i| < epsabs$$

成立していなければ GSL_CONTINUE を返す。この判定基準では、根の正確な位置 x はあまり重要でなく、それでも残差が十分に小さな値が見つかるときに使うことができる。

34.6　導関数を使う方法

ここにまず、目的関数とその導関数の両方を使う方法を挙げる。いずれも根の初期推定値が必要である。そして必ずしも根に収束するとは限らない。根が求まるためには、関数が特殊な振る舞いをしないこと、初期値が十分に真の根に近いことが必要である。これらの条件が満たされれば、二次収束 (quadratic convergence) を示す。

`gsl_multiroot_fdfsolver_hybridsj` [Derivative Solver]

MINPACK に `hybrj` として実装されているパウエルの修正ハイブリッド法 (modified version of Powell's Hybrid method)。MINPACK は モレ (Jorge J. Moré)、ガーバウ (Burton S. Garbow)、ヒルストロム (Kenneth E. Hillstrom) の三人による FORTRAN ライブラリである。修正ハイブリッド法はニュートン法の収束の速さを保ちながら、ニュートン法がうまく行かないような場合でも、関数値をより小さくすることができる方法である。

この方法では探索ステップを常に管理下におくために、一般化信頼領域 (generalized trust region、次の `hybridj` アルゴリズムでの球形の信頼領域と比べて general という意味) を設定する。新しい探索点候補 x を採用するかどうかを、D を対角係数行列 (diagonal scaling matrix、x の変化量を各座標軸についてスケーリングする) とし、δ が信頼領域の大きさのとき、$|D(x' - x)| < \delta$ という条件を満たすかどうかで決定する。D の各要素はヤコビアン行列の列ノルムを使って内部で計算され、x の各成分が関数値の残差 (目的の値 $= 0$ との差) にどの程度の影響を持つか (感度、sensitivity) が推定される。これにより、関数値が大きく変動するような特殊な挙動を示す関数での探索能力を向上する。

繰り返し計算の各回ではまず、$Jdx = -f$ を解いてニュートン法でステップを決める。そのステップが信頼領域内に収まっていたら、それを次のステップとする。信頼領域の外だったら、ニュートン法による探索方向と勾配法による方向の線形結合を、信頼領域内で関数のノルムが最小化となるように縮小して次のステップに使う。

$$dx = -\alpha J^{-1} f(x) - \beta \nabla |f(x)|^2$$

この線形結合によるステップを「くの字ステップ (dogleg step)」 と呼ぶ。

次に、こうして決めたステップにより決まる点 $x' = x + dx$ での関数値を計算する。これにより関数のノルムが十分に小さいならその点を採用し、信頼領域を拡張する。このステップで解が改善されない場合は信頼領域を狭くし、ステップを計算しなおす。この方法では、ヤコビアンの変化量をランク 1 公式 (rank-1 update) で近似計算することで計算時間を短縮する。ステップの生成が残差の減少に二回続けて失敗したときに、ヤコビアン全体を計算し直す。また探索の進行を常に監視し、数回のステップで連続して解の改善に失敗すると以下のエラーを出す。

`GSL_ENOPROG`
: 繰り返し計算で解が改善されず、探索が一時停止されている。

GSL_ENOPROGJ
ヤコビアン行列の再計算結果から繰り返し計算による改善が見込めないことが分かり、探索が一時停止されている。

gsl_multiroot_fdfsolver_hybridj [Derivative Solver]

hybridsj と同じだが信頼領域のスケーリングを行わない方法である。ステップは球形の信頼領域 $|x' - x| < \delta$ にとどまるように制御される。この方法は hybridsj が設定する一般形の信頼領域ではうまくいかないときに有効な場合がある。

gsl_multiroot_fdfsolver_newton [Derivative Solver]

ニュートン法 (Newton's method)。代表的な改善法の一つである。あらかじめ与える解の初期推定値から探索を始める。繰り返し計算の各回では、関数 F の線形近似を使ってステップを決め、残差の各要素が 0 になるまで繰り返す。繰り返し計算は以下のように定義される。

$$x \to x' = x - J^{-1} f(x)$$

ここで J は f で与えられる導関数から計算されるヤコビアン行列である。ステップ dx は以下の線形方程式を LU 分解で解くことで得られる。

$$J dx = -f(x)$$

gsl_multiroot_fdfsolver_gnewton [Derivative Solver]

これは修正ニュートン法 (modifid Newton's method) で、各ステップで残差のユークリッド・ノルム $|f(x)|$ を縮小するようにすることで大域的な収束の改善を期待する方法である。ニュートン法によるステップをとるとノルムが大きくなる場合に、以下の式でステップ幅を計算する。

$$t = (\sqrt{1 + 6r} - 1)/(3r)$$

ここで r はノルムの比 $|f(x')|^2/|f(x)|^2$ である。この計算をステップ幅が適切な値になるまで繰り返す。

34.7 導関数を使わない方法

この節のアルゴリズムは、導関数を使わず目的関数値のみで根を探索する。必要となる微分値はすべて有限差分で近似的に計算される。以下のルーチンで自動的に決定される刻み幅がおかしな、またはあまりよくない値になる場合は、別に導関数値を計算すれば、前節のアルゴリズムをその場で適用できる。

gsl_multiroot_fsolver_hybrids [Solver]

パウエルの修正ハイブリッド法。前節のと違って、ヤコビアン行列の計算をする代わりに有限差分近似 (finite difference approximation) を使う。`GSL_SQRT_DBL_EPSILON` に対する相対的なステップ幅の大きさと `gsl_multiroots_fdjac` を使って有限差分を計算する。

`gsl_multiroot_fsolver_hybrid` [Solver]

スケーリングを使わない (信頼領域が球形の) 修正ハイブリッド法。導関数の代わりに有限差分を使う。

`gsl_multiroot_fsolver_dnewton` [Solver]

離散ニュートン法 (discrete Newton algorithm)。もっとも単純な多次元求根法である。これはニュートン法の繰り返し計算を以下で行う。

$$x \to x - J^{-1} f(x)$$

ここで、ヤコビアン行列 J を関数 f の有限差分で近似する。GSL の実装では以下の式で近似を行う。

$$J_{ij} = (f_i(x + \delta_j) - f_i(x))/\delta_j$$

ここで δ_j は、ϵ を機械イプシロン ($\epsilon \approx 2.22 \times 10^{-16}$) とするときのステップ幅 $\sqrt{\epsilon}|x_j|$ である。ニュートン法は二次収束するが、繰り返し計算の各回で有限差分の計算に n^2 回の関数値の計算が必要である。有限差分が真の微分値をあまりよく近似しない場合は、この方法は不安定である。

`gsl_multiroot_fsolver_broyde` [Solver]

ブロイデン法 (Broyden algorithm)。離散ニュートン法の一つであり、繰り返し計算の各回でのヤコビアン行列の計算をできるだけ節約しようとするものである。またヤコビアンの変化量を、ランク 1 公式で以下のように近似する。

$$J^{-1} \to -J^{-1} - (J^{-1} df - dx) dx^T J^{-1} / dx^T J^{-1} df$$

ここでベクトル dx と df はそれぞれ x と f の変化量である。繰り返し計算の最初の一回では離散ニュートン法と同様、有限差分を使ってヤコビアン行列の逆行列を計算する。

有限差分を使うことでランク 1 公式の計算は速くなるが、ヤコビアン近似値の変化量が小さくない、あるいは何度も繰り返し計算を行ううちにヤコビアン行列の逆行列の近似が悪くなる場合にはいい方法とは言えない。この方法は探索開始点が真の解に近くないときには不安定になりがちである。そこで、不安定になってきた場合にはヤコビアン行列を再計算する (詳しくはソースコード参照)。

この方法は、高い制度が要求される用途には推奨できない。GSL では単に手法のデモンストレーションとして用意している。

34.8 例

多次元求根法も、一次元求根法と同様の方法で利用することができる。最初に信頼領域のスケーリングを行い、かつ導関数を必要としない hybrids の例を示す。この例ではローゼンブロックの方程式 (Rosenbrock system of equations)、

$$f_1(x,y) = a(1-x),\ f_2(x,y) = b(y-x^2)$$

を $a = 1, b = 10$ の場合について解く。この方程式の解は $(x,y) = (1,1)$ で、狭い谷間にある。

プログラムではまず方程式の定義を行う。

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_multiroots.h>

struct rparams {
    double a;
    double b;
};

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;
}
```

プログラムではまず関数のオブジェクト f を引数 (x,y)、パラメータ (a,b) で生成する。求根法のインスタンス s はこの関数を使って hybrids で初期化される。

```
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]);
    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) /* 求根法がつまづいていないかのチェック */
            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;
}
```

探索が局所解にはまりこんでしまう場合もあるため、各ステップで返り値の確認が重要である。探索が進められないような、なにかのエラーが発生した場合、利用者はそのエラーを調べて新しい探索点を与えなおしたり、他の求根法に切り替えたりすることができる。

探索の途中経過は以下の関数で調べることができる。求根法のインスタンスが保持している探索状況は、現在の探索点を示すベクトル `s->x` と、`x` その点に対応する関数値のベクトル `s->f` である。

```
int print_state (size_t iter, gsl_multiroot_fsolver * s) {
    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));
}
```

プログラムを実行した結果を以下に示す。探索は、根から離れた点 $(-10, -5)$ から開始している。根は谷間に隠れており、初期のステップは大きな残差を縮小すべく関数値を下る方向にとられる。繰り返し計算の 8 回目で解の近くまで進むとニュートン法になり、非常に速く収束する。

```
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
```

このアルゴリズム中の繰り返し計算は、毎回探索点を更新するとは限らない。繰り返し計算では、ステップが発散しそうなときに信頼領域を調整したり、収束が悪くなってきたときにヤコビアン行列の再計算をしたりする。

次の例では探索を加速するために導関数を使う。`rosenbrock_df` と `rosenbrock_fdf` の二つの関数で導関数が定義されるが、後者は関数値と導関数値の両方を同時に計算するため、関数と導関数の両方に共通している項の計算を無駄を省いて最適化することができ、また `f` と `df` をそれぞれ別に呼び出ださずにすむ。以下に示すコードでは単にコードをシンプルにするために使っている。

```
int rosenbrock_df(const gsl_vector * x, void *params, 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;
}
```

そして main 関数では、fdfsolver の導関数版を呼び出す。

```
int main (void)
{
    const gsl_multiroot_fdfsolver_type *T;
    gsl_multiroot_fdfsolver *s;
    int status;
    size_t i, iter = 0;
    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;
    }
```

ヤコビアン行列の数値計算による近似はここでは十分な精度を持っているので、hybrids インスタンスに渡す導関数値によって特に探索に違いが出るわけではない。ここでは gnewton に切り替えて他の求根法との違いを示す。この方法はニュートン法だが、ステップをとると関数値が大きくなる場合に、ステップを縮小する方法である。以下に gnewton 法による出力示す。

```
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
```

収束は更に速くなるが、かなり離れた点 $(-4.23, -65.3)$ にも探索が及んでいる。現実の問題にこの方法を適用すると、探索が迷い込んでしまうこともあり得る。解に向かって坂を下る修正ハイブリッド法の方が、より信頼性が高い。

34.9 参考文献

パウエルによるオリジナルのハイブリッド法は以下の文献に示されている。

- 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).

この節に示した方法については、以下の文献がよい。

- Jorge J. Moré, Michel Y. Cosnard, "Numerical Solution of Nonlinear Equations", *ACM Transactions on Mathematical Software*, **5**(1), pp. 64–85 (1979).
- C. G. Broyden, "A Class of Methods for Solving Nonlinear Simultaneous Equations", *Mathematics of Computation*, **19** pp. 577–593 (1965)

- Jorge J. Moré, Burton S. Garbow, Kenneth E. Hillstrom, “Testing Unconstrained Optimization Software”, *ACM Transactions on Mathematical Software*, **7**(1), pp. 17–41 (1981)

以下の和書の書籍に解説されているパウエルの方法や準/修正ニュートン法などは最適化法であるが、求根法の参考にもなる。

- 嘉納秀明, システムの最適理論と最適化 (コンピュータ制御機械システムシリーズ 3), コロナ社, ISBN 4-339-04123-8 (1987)

第35章　多次元関数の最適化

この章では、任意の多次元関数 (multidimentional function) に対して最小化 (minimization、関数値が最小となる独立変数値を探索すること) を行うルーチンに関して説明する。GSL には最適化を行う繰り返し計算法とその収束の判定法について、低レベルのルーチン (low level component、アルゴリズムとして抽象化レベルが低いという意味) をそれぞれ複数用意している。これらを適切に組み合わせてプログラムを作成することで、計算の各ステップの状況を常に確認しながら目的の解を得ることができる。それぞれ同じフレームワークを使用しており、実行時にこれらのメソッドを切り替えることができる。その際プログラムの再コンパイルは不要である。最適化の各インスタンスは探索点を常に各々で保持しており、マルチスレッドに対応している。関数の最大値を求めたいときは、目的関数の符号を反転して最小化を行えばよい。

ヘッダファイル 'gsl_multimin.h' に最小化関数のプロトタイプその他の宣言が書かれている。

35.1　概要

多次元最小化問題とは、スカラー関数

$$f(x_1, \ldots, x_n)$$

が近傍のどの点よりも小さい値を取る点 x を探索することである。連続関数ではその点での勾配 $g = \nabla f$ は 0 になる。一般に、n 変数関数を最小化できる囲い込み法はなく、初期推定値から勾配を下る方向に進もうとする方法しかない。

関数の勾配を使う方法は全て、最小値が十分な精度で見つかるまでその勾配の方向で直線探索による最小化を行う。そして関数値と導関数値を使って探索方向を変えながら、n 次元空間での最小値が見つかるまで探索が続けられる。

ネルダーとミード (John Ashworth Nelder, Roger Mead) によるシンプレックス法 (simplex method、または滑降シンプレックス法、downhill simplex method) はそういった方法とは異なり、n 次元の単体 (simplex、n 次元空間内で $n+1$ 個の頂点からなるポリトープ) の頂点ベクトルを $n+1$ 個の探索パラメータ・ベクトル とし、繰り返し計算の各回で最も悪い頂点ベクトルを単純移動させることで改善し、単体の大きさが指定された大きさよりも小さくなるまで改善を繰り返す。

繰り返し計算は主に以下の三段階からなる。

- 最小化法 T の探索点 s を初期化する。
- T の繰り返し計算を使って s を更新する。
- s の収束を判定し、必要なら繰り返し計算を続ける。

GSL ではこの各段階について、それぞれ独立した関数を用意しており、これらを使って高レベル (抽象化レベルが高い) の最小化ルーチンを書くことができる。

繰り返し計算の各回では、現在の探索方向に沿った一次元探索と探索方向の更新が行われる。最小化の状況は gsl_multimin_fdfminimizer または gsl_multimin_fminimizer 構造体に保持される。

35.2　注意点

ここにある最小化法は、一度に一つの最小値しか探せない。探索領域に複数の極小点がある場合、最初に見つかった点を結果として返すが、どの極小点が最初に見つかるかを前もって予測することは困難である。その領域に他にも最小点があったとしても、ほとんどの場合なんのエラーも出さずに解を一つだけ返す。

また、これらの最小化法は局所解 (local minimum) を探すことしかできない。見つかった極小点が大域的な最小点 (global minimum) かどうかを判定することはできない。

35.3　多次元最小化法インスタンスの初期化

以下の関数は、多次元最小化法のインスタンスを初期化する。インスタンス自体は探索空間の次元と用いる最小化法の種類にのみ依存し、他の問題に適用し直すことができる。

gsl_multimin_fdfminimizer * gsl_multimin_fdfminimizer_alloc (*const gsl_multimin_fdfmi*
* **T**, *size_t* **n**) [Function]

gsl_multimin_fminimizer * gsl_multimin_fminimizer_alloc (*const gsl_multimin_fminimizer*
* **T**, *size_t* **n**) [Function]

新しく生成した型 T の最小化法のインスタンスへのポインタを返す。そのためのメモリが確保できなかった場合には NULL ポインタを返し、エラーコード GSL_ENOMEM でエラーハンドラーを呼び出す。

int gsl_multimin_fdfminimizer_set (*gsl_multimin_fdfminimizer* * **s**, *gsl_multimin_function_fd*
* **fdf**, *const gsl_vector* * **x**, *double* **step_size**, *double* **tol**) [Function]

この関数は最小化法のインスタンス s を、関数 *fdf* を探索開始点 x から探索するように初期化する。また直線探索の精度を *tol* で指定するが、このパラメータの意味するところは、厳密には最小化法の種類によって異なる。直線探索の典型的な終了条件は、関数の勾配 g と探索方向 p が直交していると精度 *tol* でみなされる、つまり $p \cdot g < tol\ |p||g|$ が成立するときである。直線探索は粗い探索としてしか使われないため、ほとんどの場合は *tol* の値は 0.1 程度で構わない (デフォルト値は不定なので、値を明示的に指定しなければならない)。この値を 0 にしてしまうと、「厳密な」直線探索が要求されることになり、膨大な計算時間を要することになる。

int gsl_multimin_fminimizer_set (*gsl_multimin_fminimizer* * **s**, *gsl_multimin_function*
* **f**, *const gsl_vector* * **x**, *const gsl_vector* * **step_size**) [Function]

この関数は最小化法のインスタンス s を、関数 f を探索開始点 x から最小化するように初期化する。最初の探索のステップ幅を *step_size* で指定する。このパラメータも、正確な意味は使う最小化法によって異なる。

`void gsl_multimin_fdfminimizer_free (`*gsl_multimin_fdfminimizer* `* s)` [Function]
`void gsl_multimin_fminimizer_free (`*gsl_multimin_fminimizer* `* s)` [Function]

この関数は最小化法のインスタンス s に割り当てられたメモリを解放する。

`const char * gsl_multimin_fdfminimizer_name (`*const gsl_multimin_fdfminimizer* `* s)` [Function]
`const char * gsl_multimin_fminimizer_name (`*const gsl_multimin_fminimizer* `* s)` [Function]

この関数は、与えられた最小化法の名前文字列へのポインタを返す。たとえば以下のコードは `s is a 'conjugate_pr' minimizer` のように出力する。

```
printf ("s is a '%s' minimizer\n", gsl_multimin_fdfminimizer_name(s));
```

35.4 目的関数の設定

最小化したい目的関数は、パラメータを含む n 変数の関数として定義しておく必要がある。また関数の勾配を計算するルーチン、関数値と勾配の両方を同時に計算するルーチンも用意する必要がある。一般的な目的関数をパラメータを用いて定義できるようにするため、目的関数は以下の構造体で定義する必要がある。

`gsl_multimin_function_fdf` [Data Type]

この型は、以下のパラメータと導関数が与えられるときの n 変数の一般的な関数の型を定義する。

`double (* f) (const gsl_vector * x, void * params)`
この関数は引数 x 、パラメータ *params* のときの関数値 $f(x, params)$ を返す。

`void (* df) (const gsl_vector * x, void * params, gsl_vector * g)`
この関数は、引数 x、パラメータ *params* のときの n 次元の勾配ベクトル $gi = \partial f(x, params)/\partial x_i$ をベクトル g に入れ、計算ができなかった場合には対応するエラーコードを返す。

`void (* fdf) (const gsl_vector * x, void * params, double * f, gsl_vector * g)`

この関数は、引数 x、パラメータ *params* のときに上述の f と g を設定する。この関数は分かれている関数 $f(x)$ と $g(x)$ について、関数値と導関数値を同時に計算することで計算時間を短縮する。

`size_t n`
　探索空間の次元数、ベクトル x の要素の数である。

`void * params`
　関数のパラメータへのポインタ。

`gsl_multimin_function` [Data Type]

これは以下のパラメータを持つ n 変数の一般的な関数を定義する型である。

`double (* f) (const gsl_vector * x, void * params )`
　この関数は引数 x 、パラメータ $params$ のときの関数値 $f(x, params)$ を返す。

`size_t n`
　探索空間の次元数、ベクトル x の要素の数である。

`void * params`
　関数のパラメータへのポインタ。

以下に例示する関数は、パラメータを二個持つ単純な放物面である。

```
/* 中心が (p[0], p[1])、係数が (p[2], p[3])、最小値が p[4] の放物面 */

double my_f (const gsl_vector *v, void *params)
{
    double x, y;
    double *dp = (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];
}

/* f の勾配 df = (df/dx, df/dy) */
void my_df (const gsl_vector *v, void *params, gsl_vector *df)
{
    double x, y;
    double *dp = (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]));
}

/* f と df の両方を計算 */
void my_fdf (const gsl_vector *x, void *params, double *f, gsl_vector *df)
{
    *f = my_f(x, params);
    my_df(x, params, df);
}
```

この関数は、以下のようなコードで初期化できる。

```
gsl_multimin_function_fdf my_func;
/* 中心は (1,2)、係数が (10,20)、最小値が 30 */
double p[5] = { 1.0, 2.0, 10.0, 20.0, 30.0 };
my_func.n = 2; /* 関数の定義域の次元数 */
my_func.f = &my_f;
my_func.df = &my_df;
my_func.fdf = &my_fdf;
my_func.params = (void *)p;
```

35.5 繰り返し計算

以下の関数が繰り返し計算を実行する。関数はそれぞれ、インスタンスが持つアルゴリズムによる繰り返し計算による探索点の更新を一回行う。同じ関数が全てのアルゴリズムに使え、プログラムを書き換えることなく、実行時にアルゴリズムを切り替えることができる。

`int gsl_multimin_fdfminimizer_iterate (gsl_multimin_fdfminimizer * s)` [Function]

`int gsl_multimin_fminimizer_iterate (gsl_multimin_fminimizer * s)` [Function]

最小化法のインスタンス s の繰り返し計算を一回行う。予期しない問題が発生した場合はエラーコードを返す。

最小化法のインスタンスは現在の最良近似解を常に保持しており、以下の補助関数を使って参照することができる。

`gsl_vector * gsl_multimin_fdfminimizer_x (const gsl_multimin_fdfminimizer * s)` [Function]

`gsl_vector * gsl_multimin_fminimizer_x (const gsl_multimin_fminimizer * s)` [Function]

`double gsl_multimin_fdfminimizer_minimum (const gsl_multimin_fdfminimizer * s)` [Function]

double gsl_multimin_fminimizer_minimum (*const gsl_multimin_fminimizer * **s***) [Function]

gsl_vector * gsl_multimin_fdfminimizer_gradient (*const gsl_multimin_fdfminimizer * **s***) [Function]

double gsl_multimin_fminimizer_size (*const gsl_multimin_fminimizer * **s***) [Function]

これらの関数は渡された最小化インスタンス s について、その時点での最小点の最良推定値、その点での関数値および導関数値、およびアルゴリズムごとに定義される特徴量を返す (現時点では fminimizer はシンプレックス法しか実装されていないため、gsl_multimin_fminimizer_size が使えるのはシンプレックス法だけである)。

int gsl_multimin_fdfminimizer_restart (*gsl_multimin_fdfminimizer * **s***) [Function]

この関数は最小化法のインスタンス s を、現在の探索点から探索をやり直すように再初期化する。

35.6　停止条件

最小化アルゴリズムは、以下の条件のいずれかが真になったときに停止する。

- 設定された精度で最小点が見つかったとき。
- 設定された最大回数に繰り返し計算が達したとき。
- エラーが発生したとき。

これらの条件は任意に設定することができる。以下の関数でその時の最良探索点とその精度を調べることができる。

int gsl_multimin_test_gradient (*const gsl_vector * **g**, double **epsabs***) [Function]

この関数は与えられる許容絶対値に対する勾配 g のノルムを判定する。多次元関数の勾配は、最小点で 0 になるはずである。この判定関数は以下の条件が成立しているときに GSL_SUCCESS を返し、成立していなければ GSL_CONTINUE を返す。

$$|g| < epsabs$$

epsabs の値は、x の微小な変動に対する関数値の変動から、どれだけ関数値の変動を許すかを考えて決めなければならない。それは $\delta f = g\delta x$ で見積もることができる。

int gsl_multimin_test_size (*const double **size**, double **epsabs***) [Function]

この関数は最小化法の特徴量（その最小化法で使える場合のみ）を、指定される許容絶対値に対して判定する。この関数は、特徴量が許容量より小さければ GSL_SUCCESS を、そうでなければ GSL_CONTINUE を返す。

35.7 最小化アルゴリズム

このライブラリではいくつかの最小化法を用意している。問題により適しているアルゴリズムは異なる。シンプレックス法は関数値のみを、それ以外の方法はどれも関数値と導関数値の両方を使う。

`gsl_multimin_fdfminimizer_conjugate_fr` [Minimizer]

フレッチャー-リーブズ法 (Roger Fletcher と C. M. Reeves の共役勾配法 conjugate gradient method、FR 法)。FR 法は直線探索を繰り返すことで進んでいく。探索方向をどう変えていくかで最小値の近傍での関数の形を近似する。

探索方向の初期値 p は勾配を使って決められ、その方向に沿って直線探索で極小値を探索する。その決定の精度をパラメータ *tol* で与える。つまり $p \cdot g < tol\ |p||g|$ が成り立つ点で関数の勾配 g と探索方向 p が直交していると見なし、そこをその直線上での関数の極小点とする。探索方向はフレッチャーとリーブスの公式 $p = g - \beta g$ (ただし $\beta = -|g'|^2/|g|^2$) を使って決定され、次々と決める方向に従って直線探索が繰り返される。

`gsl_multimin_fdfminimizer_conjugate_pr` [Minimizer]

ポラックとリビエール (E. Polak、G. Ribiére) の共役勾配法 (PRCG 法)。この方法は FR 法と似ているが、係数 β の取り方が異なる。どちらの方法も、目的関数の超平面が二次関数で近似でき、探索点が最小点に十分に近い場合にはうまく探索できる。

`gsl_multimin_fdfminimizer_vector_bfgs2` [Minimizer]
`gsl_multimin_fdfminimizer_vector_bfgs` [Minimizer]

ブロイデン・フレッチャー・ゴールドファーブ・シャノ法 (Broyden-Fletcher-Golodfarb-Shanno の共役勾配法、BFGS 法)。勾配ベクトル間の差を使って関数 f の二階導関数を近似する準ニュートン法である。一階および二階導関数を組み合わせ、関数を二次関数的であると仮定することで、最小点に向かってニュートン法的なステップをとることができる。

`bsgf2` では、直線探索を効率化するためにフレッチャーによる *Practical Methods of Optimization* にある Algorithm 2.6.2 および 2.6.4 を忠実に実装して使っている。元の `bfgs` アルゴリズムよりも関数や導関数の評価回数が少なくて済むため、`bfgs` の代わりにこちらを使うべきである。フレッチャーの言うパラメータ σ が *tol* に対応する。多くの場合は 0.1 程度にしておいてよい (値を大きくすると、直線探索の精度が落ちる)。

`gsl_multimin_fdfminimizer_steepest_descent` [Minimizer]

最急降下法 (steepest descent) は、各ステップで勾配が下がる方向へ探索していく方法である。降下ステップが成功した場合は、ステップ幅を二倍する。降下ステップにより関数値が大きくなる場合は探索点を逆戻りし、パラメータ *tol* でステップ幅を縮小する。この *tol* の値は多くの場合 0.1 程度でよい。最級降下法はあまり性能が良くなく、単なるデモンストレーション用である。

`gsl_multimin_fminimizer_nmsimplex` [Minimizer]

ネルダー-ミード法 (Nelder-Mead method、シンプレックス法 simplex method、滑降シンプレックス法 downhill simplex method) である。この方法では初期ベクトル x とベクトル `step_size` から n 本のベクトル p_i を以下のようにして生成する。

$$p_0 = (x_0, x_1, \cdots, x_n) \tag{35.1}$$
$$p_1 = (x_0 + step\_size_0, x_1, \cdots, x_n) \tag{35.2}$$
$$p_0 = (x_0, x_1 + step\_size_1, \cdots, x_n) \tag{35.3}$$
$$\ldots = \ldots \tag{35.4}$$
$$p_n = (x_0, x_1, \cdots, x_n + step\_size_1) \tag{35.5}$$

これらのベクトルは n 次元空間内の単体の $n+1$ 個の頂点をなす。繰り返し計算の各ステップで、関数値が最も大きな点に対応するパラメータベクトル p_i に対し単純な位置転換 (simple geometrical transformation) による改善を試みる。この変換は鏡映 (reflecetion、転換される頂点を除いた単体の重心に対して、対称移動する) であり、続いて展開 (expansion)、縮約 (contraction)、多重縮約 (multiple contraction) が行われる。これにより単体は探索空間内を最小点に向かって移動、縮約していく。

繰り返し計算の各回で、最も良い頂点が返される。この方法では繰り返し計算で必ずしも毎回、パラメータベクトルが改善されるとは限らず、複数回の繰り返し計算が必要であることが多い。

この最小化法特有の特徴量として、単体の重心から各頂点への平均距離が計算される。この量は単体が最小点に縮約しているかどうかを示し、停止条件の判定に使うことができる。この量は `gsl_multimin_fminimizer_size` の返り値として得られる。

35.8　例

以下の例は、前に定義された放物面関数の最小点を探索するものである。最小点は原典から少し離れており、そこでの関数値は 0 ではない。メイン・プログラムを以下に示すが、この他にこの章で先に示した関数の定義が必要である。

```
int main (void)
{
    size_t iter = 0;
    int status;
    const gsl_multimin_fdfminimizer_type *T;
    gsl_multimin_fdfminimizer *s;

    /* 中心は (1,2)、係数が (10,20)、最小値が 30 */
    double par[5] = { 1.0, 2.0, 10.0, 20.0, 30.0 };
```

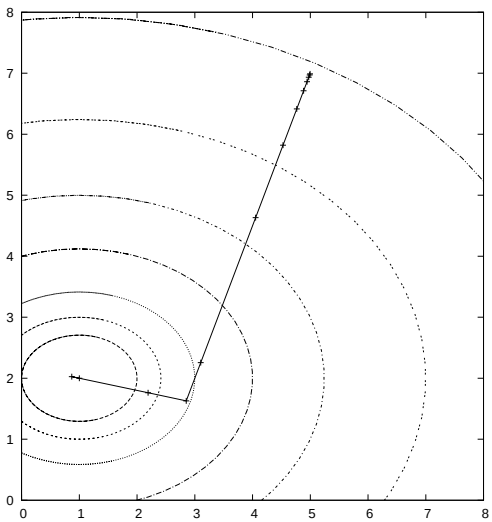
```
    gsl_vector *x;
    gsl_multimin_function_fdf my_func;

    my_func.n = 2;
    my_func.f = &my_f;
    my_func.df = &my_df;
    my_func.fdf = &my_fdf;
    my_func.params = &par;

    /* 探索開始点は 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 {
        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;
}
```

ステップ幅の初期値は、この例では適当であろうと思われる 0.01 とし、直線探索の終了条件のパラメータは 0.0001 とする。探索は勾配ベクトルのノルムが 0.001 以下になったときに終了する。

このプログラムの出力を以下に示す。

```
      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
12 0.86892 2.02622  30.18555
Minimum found at:
13 1.00000 2.00000  30.00000
```

この方法は、探索点をプロットすると分かるように、降下ステップが成功するとステップ幅を次第に大きくしていく。



共役勾配法は、関数が二次形式であれば二つめの探索方向で最小値を見つけ出す。関数の形が複雑になれば、繰り返し計算も増える。

以下に、ネルダーとミードのシンプレックス法を使って同じ関数を最小化する例を示す。

```
int main(void)
{
    size_t np = 2;
```

```
double par[5] = {1.0, 2.0, 10.0, 20.0, 30.0};

const gsl_multimin_fminimizer_type *T =
    gsl_multimin_fminimizer_nmsimplex;
gsl_multimin_fminimizer *s = NULL;
gsl_vector *ss, *x;
gsl_multimin_function minex_func;

size_t iter = 0, i;
int status;
double size;

/* 探索開始点 */
x = gsl_vector_alloc(2);
gsl_vector_set(x, 0, 5.0);
gsl_vector_set(x, 1, 7.0);

/* ステップ幅の初期値は全て 1 */
ss = gsl_vector_alloc(2);
gsl_vector_set_all (ss, 1.0);

/* インスタンスの初期化 */
minex_func.n = 2;
minex_func.f = &my_f;
minex_func.params = (void *)&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 %.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_vector_free(x);
        gsl_vector_free(ss);
        gsl_multimin_fminimizer_free(s);

        return status;
    }
```

探索は単体の大きさが 0.01 以下になったときに終了する。プログラムの出力を以下に示す。

```
        1 6.500e+00  5.000e+00 f() = 512.500 size = 1.082
        2 5.250e+00  4.000e+00 f() = 290.625 size = 1.372
        3 5.250e+00  4.000e+00 f() = 290.625 size = 1.372
        4 5.500e+00  1.000e+00 f() = 252.500 size = 1.372
        5 2.625e+00  3.500e+00 f() = 101.406 size = 1.823
        6 3.469e+00  1.375e+00 f() = 98.760  size = 1.526
        7 1.820e+00  3.156e+00 f() = 63.467  size = 1.105
        8 1.820e+00  3.156e+00 f() = 63.467  size = 1.105
        9 1.016e+00  2.812e+00 f() = 43.206  size = 1.105
       10 2.041e+00  2.008e+00 f() = 40.838  size = 0.645
       11 1.236e+00  1.664e+00 f() = 32.816  size = 0.645
       12 1.236e+00  1.664e+00 f() = 32.816  size = 0.447
       13 5.225e-01  1.980e+00 f() = 32.288  size = 0.447
       14 1.103e+00  2.073e+00 f() = 30.214  size = 0.345
       15 1.103e+00  2.073e+00 f() = 30.214  size = 0.264
       16 1.103e+00  2.073e+00 f() = 30.214  size = 0.160
       17 9.864e-01  1.934e+00 f() = 30.090  size = 0.132
       18 9.190e-01  1.987e+00 f() = 30.069  size = 0.092
       19 1.028e+00  2.017e+00 f() = 30.013  size = 0.056
       20 1.028e+00  2.017e+00 f() = 30.013  size = 0.046
       21 1.028e+00  2.017e+00 f() = 30.013  size = 0.033
       22 9.874e-01  1.985e+00 f() = 30.006  size = 0.028
       23 9.846e-01  1.995e+00 f() = 30.003  size = 0.023
       24 1.007e+00  2.003e+00 f() = 30.001  size = 0.012
    converged to minimum at
       25 1.007e+00  2.003e+00 f() = 30.001  size = 0.010
```

探索初期には単体は、最小点に向かって移動しながら大きくなっている。しばらくしてから大きさは縮小に転じ、単体は最小点のまわりで縮約する。

35.9 参考文献

共役勾配法と BFGS 法は以下の本に詳しく説明されている。

- R. Fletcher, *Practical Methods of Optimization (Second Edition)*, Wiley, ISBN 0471915475 (1987).

多次元最小化法の概要と参考情報は以下の書籍にある。

- C. W. Ueberhuber, *Numerical Computation* (Volume 2), Chapter 14, Section 4.4 “Minimization Methods”, pp. 325–335, Springer, ISBN 3-540-62057-5 (1997).

シンプレックス法については、以下の論文を参照のこと。

- J. A. Nelder and R. Mead, “A simplex method for function minimization”, *Computer Journal*, **7**(4), pp. 308–313 (1965).

第36章　最小二乗近似

この章では、実験観測データに対して関数の線形結合で最小二乗近似 (least-squares fitting) を行うルーチンについて説明する。データには重み (各点でのデータの分散の逆数) があるもの (weighted) とないもの (unweighted) を想定する。重み付きデータに対しては最良近似を与えるパラメータと共分散行列 (covariance matrix) を計算する。重みなしデータに対しては、与えられた分散共分散行列 (variance-covariance matrix) と点の散らばりから共分散行列を得る。

当てはめに使われる関数は、パラメータが 1 個あるいは 2 個の単純なもの (線形回帰) と、それ以上のものに分けてある。関数はヘッダファイル 'gsl_fit.h' で宣言されている。

36.1　概要

最小二乗法で決定された最適パラメータ値を持つモデル $Y(c,x)$ とは、n 個の重み付き観測データの対 (x_i, y_i) とモデルの値との残差の二乗和である以下の χ^2 を最小化するようにしたものである。

$$\chi^2 = \sum_i w_i (y_i - Y(c, x_i))^2$$

ベクトル $c = \{c_0, c_1, \ldots\}$ はモデルを記述する p 個のパラメータである。重み w_i は $w_i = 1/\sigma_i^2$ で表される。σ_i はデータ点 y_i に含まれる観測値のばらつきであり、ばらつきの値は各データ点の間で独立で、各データは正規分布にしたがって生じていると仮定している。重み付きでないデータでは $w_i = 1$ として計算を行う。

最小二乗法のルーチンは p 個のパラメータの最適値と、大きさが $p \times p$ の共分散行列を計算する。共分散行列は、最適パラメータ値の統計的ばらつき (statistical error) を表し、これはデータのばらつき σ_i から計算される。$\langle\ \rangle$ がデータに想定される誤差正規分布 (Gaussian error distribution) の平均値を表すとしたとき、共分散行列は $C_{ab} = \langle \delta c_a \delta c_b \rangle$ で定義される。

共分散行列はデータ点のばらつき σ_i がどのように最適パラメータ値に伝わるかから計算される。データ点の値が小さく δy_i だけ (ばらつきにより) 変化したときに、それによる最適パラメータ値の変化量 δc_a は以下で与えられる。

$$\delta c_a = \sum_i \frac{\partial c_a}{\partial y_i} \delta y_i$$

これにより、共分散行列をデータ点のばらつきで記述できる。

$$C_{ab} = \sum_{i,j} \frac{\partial c_a}{\partial y_i} \frac{\partial c_b}{\partial y_j} \langle \delta y_i \delta y_j \rangle$$

各点のデータが互いに独立で相関がない (uncorrelated) 場合、データ点の変動は $\langle \delta y_i \delta y_j \rangle = \sigma_i^2 \delta_{ij}$ を満たす。したがって、上のパラメータの共分散行列は以下のようになる。

$$C_{ab} = \sum_{i,j} \frac{1}{w_i} \frac{\partial c_a}{\partial y_i} \frac{\partial c_b}{\partial y_j}$$

たとえばばらつきが見積もられていないなど、重みのついていないデータについては、もっとも精度のよい (最適パラメータ値を持つ) モデルとデータとの残差の二乗和 $\sigma^2 = \sum (y_i - Y(c, x_i))^2/(n-p)$ からすべての w_i を $w = 1/\sigma^2$ として、上の式で共分散行列を計算する。

最適パラメータの標準偏差は、共分散行列の対角成分の平方根 $\sigma_{c_a} = \sqrt{C_{aa}}$ で得られる。また c_a と c_b の間の相関係数は $\rho_{ab} = C_{ab}\sqrt{C_{aa}C_{bb}}$ で得られる。

36.2 線形回帰

この節では直線回帰 (linear regression) モデル $Y(c, x) = c_0 + c_1 x$ による近似を行う関数について説明する。

`int gsl_fit_linear` (*const double * **x***, *const size_t **xstride***, *const double * **y***, *const size_t **ystride***, *size_t **n***, *double * **c0***, *double * **c1***, *double * **cov00***, *double * **cov01***, *double * **cov11***, *double * **sumsq***) [Function]

> 長さ *n*、刻み幅がそれぞれ *xstride* と *ystride* の二つのベクトル (x, y) で与えられるデータセットに対する最良近似として、直線回帰モデル $Y = c_0 + c_1 X$ のパラメータ (c_0, c_1) を計算する。y の値のばらつきは不明であるとし、したがって二つのパラメータ (c_0, c_1) の分散共分散行列は最良近似直線に対して y の値がどの程度散らばっているかから計算され、引数 (*cov00*, *cov01*, *cov11*) に入れて返される。最良近似直線からの残差の二乗和は *sumsq* に返される。注: データの相関係数は `gsl_stats_correlation` (第 20.6 節) で計算できる。データ自体の相関係数はフィッティングの結果とは無関係である。

`int gsl_fit_wlinear` (*const double * **x***, *const size_t **xstride***, *const double * **w***, *const size_t **wstride***, *const double * **y***, *const size_t **ystride***, *size_t **n***, *double * **c0***, *double * **c1***, *double * **cov00***, *double * **cov01***, *double * **cov11***, *double * **chisq***) [Function]

> 長さ *n*、刻み幅がそれぞれ *xstride* と *ystride* の二つのベクトル (x, y) で与えられるデータセットに対する最良近似として、直線回帰モデル $Y = c_0 + c_1 X$ のパラメータ (c_0, c_1) を計算する。長さ *n*、刻み幅 *wstride* のベクトル *w* でデータの各点の重みを指定する。重みはデータ y の各点の分散の逆数である。
>
> パラメータ (c_0, c_1) の共分散行列は重み付きデータから計算され、引数 (*cov00*, *cov01*, *cov11*) に入れて返される。最良近似直線からの残差の重み付き二乗和 χ^2 は *chisq* に返される。

`int gsl_fit_linear_est` (*double* `x`, *double* `c0`, *double* `c1`, *double* `c00`, *double* `c01`, *double* `c11`, *double* * `y`, *double* * `y_err`) [Function]

この関数は最良近似直線のパラメータ c_0 と c_1 および計算された共分散 (*cov00*, *cov01*, *cov11* から、近似関数値 y と x における $Y = c_0 + c_1 X$ の標準偏差 *y_err* を計算する。

36.3 定数項のない線形近似

この節の関数は、定数項のない直線モデル $Y = c_1 X$ での最小二乗近似を行う。

`int gsl_fit_mul` (*const double* * `x`, *const size_t* `xstride`, *const double* * `y`, *const size_t* `ystride`, *size_t* `n`, *double* * `c1`, *double* * `cov11`, *double* * `sumsq`)[Function]

長さ n、刻み幅がそれぞれ *xstride* と *ystride* の二つのベクトル (x, y) で与えられるデータセットに対する最良近似として、直線回帰モデル $Y = c_1 X$ のパラメータ *c1* を計算する。パラメータ *c1* の分散は最良近似直線に対するデータ点の分布から計算され、引数 *cov11* に入れて返される。最良近似直線からの残差の二乗和は *sumsq* に返される。

`int gsl_fit_wmul` (*const double* * `x`, *const size_t* `xstride`, *const double* * `w`, *const size_t* `wstride`, *const double* * `y`, *const size_t* `ystride`, *size_t* `n`, *double* * `c1`, *double* * `cov11`, *double* * `sumsq`) [Function]

長さ n、刻み幅がそれぞれ *xstride* と *ystride* の二つのベクトル (x, y) で与えられるデータセットに対する最良近似として、直線回帰モデル $Y = c_1 X$ のパラメータ *c1* を計算する。長さ n、刻み幅 *wstride* のベクトル w でデータの各点の重みを指定する。重みはデータ y の各点の分散の逆数である。

パラメータ *c1* の分散は重み付きデータから計算され、パラメータ *cov11* に入れて返される。最良近似直線からの残差の重み付き二乗和 χ_2 は *chisq* に返される。

`int gsl_fit_mul_est` (*double* `x`, *double* `c1`, *double* `c11`, *double* * `y`, *double* * `y_err`) [Function]

この関数は最良近似直線のパラメータ *c1* および計算された共分散 *cov11* から、近似関数値 y と x における $Y = c_1 X$ の標準偏差 *y_err* を計算する。

36.4 重回帰

ここでは、n 個の観測値からなるベクトル y、$n \times p$ 次元の予測変数行列 X に対して、p 個のパラメータ c の最適な値を最小二乗法で求める重回帰 (multi-parameter fitting) を行う関数について説明する。カイ二乗値は $\chi^2 = \sum_i w_i (y_i - \sum_j X_{ij} c_j)^2$ で与えられる。

行列 X をうまく作ることで、さまざまな関数や変数をいくつでも使って最小二乗近似ができる。たとえば p 次の x の多項式を使うには、以下の行列を使う。

$$X_{ij} = x_i^j$$

ここで i は観測値に対する添え字、j は 0 から $p-1$ までである。

周波数を $w_1, w_2, \ldots, w_p$ に固定した p 個の正弦関数 (sinusoidal function) で近似を行うには、

$$X_{ij} = \sin(w_j x_i)$$

とすればよい。また p 個の独立変数 $x_1, x_2, \ldots, x_p$ を使うには、

$$X_{ij} = x_j(i)$$

でよい。ここで $x_j(i)$ は、予測変数 x_j の i 番目の値である。

この節の関数はヘッダファイル ‘gsl_multifit.h’ で宣言されている。

この節の汎用最小二乗法ルーチンは、行列 X の特異値分解などで途中の計算結果を保持するための作業領域が別途必要である。

`gsl_multifit_linear_workspace * gsl_multifit_linear_alloc` (*size_t* **n**, *size_t* **p**) [Function]

この関数は n 変数で p 個のパラメータを持つモデルで近似するための作業領域を確保する。

`void gsl_multifit_linear_free` (*gsl_multifit_linear_workspace* * **work**) [Function]

この関数は作業領域 w に割り当てられたメモリを解放する。

`int gsl_multifit_linear` (*const gsl_matrix* * **X**, *const gsl_vector* * **y**, *gsl_vector* * **c**, *gsl_matrix* * **cov**, *double* * **chisq**, *gsl_multifit_linear_workspace* * **work**)[Function]

`int gsl_multifit_linear_svd` (*const gsl_matrix* * **X**, *const gsl_vector* * **y**, *double* **tol**, *size_t* * **rank**, *gsl_vector* * **c**, *gsl_matrix* * **cov**, *double* * **chisq**, *gsl_multifit_linear_workspace* * **work**) [Function]

この関数は観測データ y と予測変数の行列 X に対する、モデル $y = Xc$ の最良近似パラメータ c を計算する。パラメータの分散共分散行列 *cov* は、最適近似に対するデータのばらつきから計算される。近似モデルからの残差二乗和 χ^2 は *chisq* に入れて返される。決定係数 (coefficient of determination) の値が必要な場合は、`gsl_stats_tss` 関数を使ってデータ y の残差二乗の総和 (total sum of squares, TSS) を計算すれば、$R^2 = 1 - \chi^2/TSS$ によってその値を得ることができる。

行列 X の特異値分解で得られる最適パラメータ値は、あらかじめ確保しておいた作業領域 *work* に書き込まれる。特異値分解 (singlar value decomposition) には修正ゴルブ・ラインシュ法 (modified Golub-Reinsch algorithm) を用い、列ごとにスケーリング

することで特異値の精度を向上する。特異値が 0 (かつ機械精度以下) になるパラメータは結果から除外される。後者の関数では、特異値の比 s_i/s_0 が *tol* で指定される値以下になるパラメータも除外され、残った要素による実質的なランクが *rank* に返される。

`int gsl_multifit_wlinear` (*const gsl_matrix * **X***, *const gsl_vector * **w***, *const gsl_vector * **y***, *gsl_vector * **c***, *gsl_matrix * **cov***, *double * **chisq***, *gsl_multifit_linear_workspace * **work***) [Function]

`int gsl_multifit_wlinear_svd` (*const gsl_matrix * **X***, *const gsl_vector * **w***, *const gsl_vector * **y***, *double **tol***, *size_t * **rank***, *gsl_vector * **c***, *gsl_matrix * **cov***, *double * **chisq***, *gsl_multifit_linear_workspace * **work***) [Function]

この関数は観測データ y と予測変数の行列 X に対する、モデル $y = Xc$ の最良近似パラメータ c を計算する。パラメータの共分散行列 *cov* は、与えられるデータの重みから計算される。近似モデルからの重み付き残差二乗和 χ^2 は *chisq* に入れて返される。決定係数 (coefficient of determination) の値が必要な場合は、`gsl_stats_wtss` 関数を使ってデータ y の重み付き残差二乗の総和 (weighted total sum of squares, WTSS) を計算すれば、$R^2 = 1 - \chi^2/WTSS$ によってその値を得ることができる。

行列 X の特異値分解で得られる最適パラメータ値は、あらかじめ確保しておいた作業領域 *work* に書き込まれる。特異値が 0 (かつ機械精度以下) になるパラメータは結果から除外される。後者の関数では、特異値の比 s_i/s_0 が *tol* で指定される値以下になるパラメータも除外され、残った要素による実質的なランクが *rank* に返される。

`int gsl_multifit_linear_est` (*const gsl_vector * **x***, *const gsl_vector * **c***, *const gsl_matrix * **cov***, *double * **y***, *double * **y_err***) [Function]

重回帰係数 c と共分散行列 *cov* を使って、モデル $y = x.c$ の与えられた点 x での値 y と標準偏差 `y_err` を計算する。

`int gsl_multifit_linear_residuals` (*const gsl_matrix * **X***, *const gsl_vector * **y***, *const gsl_vector * **c***, *gsl_vector * **r***) [Function]

与えられるデータ y、パラメータ c、予測変数 X から各データ点での残差のベクトル $r = y - Xc$ を計算する。

36.5 例

以下のプログラムは単純な (架空の) データセットに対して最小二乗線形近似を行い、近似直線と、各点での標準偏差のエラーバーを出力する。

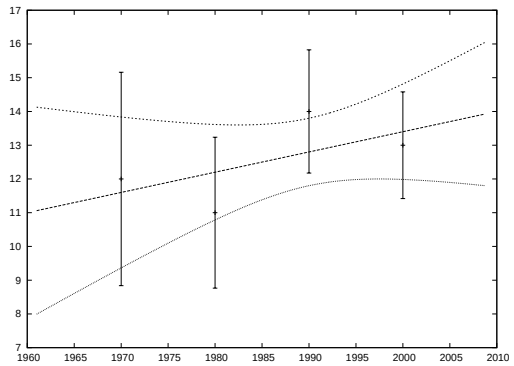
```
#include <stdio.h>
#include <gsl/gsl_fit.h>
```

```
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);
    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;
}
```

以下のコマンドは、上記プログラムの出力からデータを読み取り、GNU plotutils の graph コマンドを使ってグラフ画像を出力する。

```
$ ./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 "^$n" tmp | cut -d: -f2 > $n ;
done
$ graph -T X -X x -Y y -y 0 20 -m 0 -S 2 -Ie data
        -S 0 -I a -m 1 fit -m 2 hi -m 2 lo
```



次のプログラムは重み付きデータに対して最小二乗関数 gsl_multifit_wlinear を使って二次式 $y = c_0 + c_1 x + c_2 x^2$ による近似を行う。二次式モデルを定義する行列 X は以下のようになる。

$$X = \begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \cdots & \cdots & \cdots \end{pmatrix}$$

この行列の第一列は定数項 c_0 に、残りの二列はそれぞれ $c_1 x$ と $c_2 x^2$ に対応する。

プログラムではまず各行に (x, y, err) の順に並んだデータを n 行読み込む。err は y のばらつき (標準偏差) である。

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

main (int argc, char **argv)
{
    int i, n;
    double xi, yi, ei, chisq;
    gsl_matrix *X, *cov;
    gsl_vector *y, *w, *c;
    gsl_multifit_linear_workspace * work = gsl_multifit_linear_alloc (n, 3);
```

```
    if (argc != 2) {
        fprintf(stderr,"usage: fit n < data\n");
        exit(-1);
    }

    n = atoi(argv[1]);
    X = gsl_matrix_alloc(n, 3);
    y = gsl_vector_alloc(n);
    w = gsl_vector_alloc(n);
    c = gsl_vector_alloc(3);
    cov = gsl_matrix_alloc(3, 3);

    for (i = 0; i < n; i++) {
        int count = fscanf(stdin, "%lg %lg %lg", &xi, &yi, &ei);
        if (count != 3) {
            fprintf (stderr, "error reading file\n");
            exit (-1);
        }
        printf ("%g %g +/- %g\n", xi, yi, ei);
        gsl_matrix_set (X, i, 0, 1.0);
        gsl_matrix_set (X, i, 1, xi);
        gsl_matrix_set (X, i, 2, xi*xi);
        gsl_vector_set (y, i, yi);
        gsl_vector_set (w, i, 1.0/(ei*ei));
    }

    gsl_multifit_wlinear (X, w, y, c, cov, &chisq, work);
    gsl_multifit_linear_free (work);

#define C(i) (gsl_vector_get(c,(i)))
#define COV(i,j) (gsl_matrix_get(cov,(i),(j)))

    printf("# best fit: Y = %g + %g X + %g X^2\n", C(0), C(1), C(2));
    printf("# covariance matrix:\n");
    printf("[ %+.5e, %+.5e, %+.5e  \n", COV(0,0), COV(0,1), COV(0,2));
    printf("  %+.5e, %+.5e, %+.5e  \n", COV(1,0), COV(1,1), COV(1,2));
    printf("  %+.5e, %+.5e, %+.5e ]\n", COV(2,0), COV(2,1), COV(2,2));
    printf("# chisq = %g\n", chisq);
```

```
    gsl_matrix_free(X);
    gsl_vector_free(y);
    gsl_vector_free(w);
    gsl_vector_free(c);
    gsl_matrix_free(cov);
    return 0;
}
```

近似に適したデータが以下のプログラムで生成できる。このプログラムは曲線 $y = e^x$ に正規分布でばらつきを加えた点を $0 < x < 2$ の範囲で出力する。

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_randist.h>

int main (void)
{
    double x;
    const gsl_rng_type * T;
    gsl_rng * r;

    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc(T);

    for (x = 0.1; x < 2; x+= 0.1) {
        double y0 = exp (x);
        double sigma = 0.1 * y0;
        double dy = gsl_ran_gaussian (r, sigma);
        printf ("%g %g %g\n", x, y0 + dy, sigma);
    }

    gsl_rng_free(r);
    return 0;
}
```

コンパイルしてできる実行ファイルは、以下のように出力する。

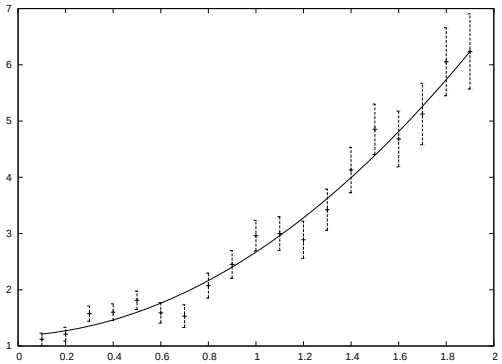
```
$ ./generate > exp.dat
$ more exp.dat
0.1 0.97935 0.110517
0.2 1.3359 0.12214
```

```
0.3 1.52573 0.134986
0.4 1.60318 0.149182
0.5 1.81731 0.164872
0.6 1.92475 0.182212
....
```

前述のプログラムを、実行時のコマンドライン引数にデータ点数 (ここでは 19 点とする) を与えて実行しこのデータを近似すると、以下のような出力を得る。

```
$ ./fit 19 < exp.dat
0.1 0.97935 +/- 0.110517
0.2 1.3359 +/- 0.12214
...
# best fit: Y = 1.02318 + 0.956201 X + 0.876796 X^2
# covariance matrix:
[ +1.25612e-02, -3.64387e-02, +1.94389e-02
  -3.64387e-02, +1.42339e-01, -8.48761e-02
  +1.94389e-02, -8.48761e-02, +5.60243e-02 ]
# chisq = 23.0987
```

二次の近似式のパラメータは e^x を展開したときの係数と一致する。x の値が大きくなると指数関数と二次式の値の差は $O(x^3)$ で大きくなっていくため、この近似式の誤差によく注意しなければならない。近似式の各パラメータによる誤差は、共分散行列中で対応する対角成分の二乗根として得られる。1 自由度当たりのカイ二乗値は 1.4 で、近似は妥当であることを示している。



36.6　参考文献

最小二乗法の数式や様々な手法は、the Particle Data Group による Annual Review of Particle Physics の “Statistics” の章にまとめられており、下記のウェブサイトで見ることができる。

- R.M. Barnett et al., “Review of Particle Properties”, *Physical Review*, **D54**(1) (1996). `http://pdg.lbl.gov/`

このライブラリにあるルーチンの検証には、NIST Statistical Reference Datasets を用いた。そのデータセットと解説が NIST のウェブサイトにある。

- `http://www.nist.gov/itl/div898/strd/index.html`

第37章　非線形最小二乗近似

この章では多次元非線形最小二乗法による近似 (multidimentional nonlinear least-squares fitting) について説明する。GSL には最適化を行う繰り返し計算法とその収束の判定法について、低レベルのルーチン (low level component、アルゴリズムとして抽象化レベルが低いという意味) をそれぞれ複数用意している。これらを適切に組み合わせてプログラムを作成することで、計算の各ステップの状況を常に確認しながら目的の解を得ることができる。それぞれ同じフレームワークを使用しており、実行時にメソッドを切り替えることができる。その際プログラムの再コンパイルは不要である。最適化の各インスタンスは探索点を各々で常に保持しており、マルチスレッド対応のプログラミングができる。

多次元非線形最小二乗法についての関数はヘッダファイル 'gsl_multifit_nlin.h' に宣言されている。

37.1　概要

多次元非線形最小二乗法では、以下のような n 個の関数 f_i の残差 (residual) の二乗和が最小になるように関数の p 個のパラメータ x_i を最適化することが必要である。

$$\Phi(x) = \frac{1}{2}||F(x)||^2 = \frac{1}{2}\sum_{i=1}^{n} f_i(x_1, \ldots, x_p)^2$$

どの方法も探索開始点から以下のような線形近似 (linearization) を使って進んでいく。

$$\phi = ||F(x+p)|| \approx ||F(x) + Jp||$$

ここで x は探索開始点、p は与えられるステップ幅、J はヤコビアン行列 $J_{ij} = \partial f_i/\partial x_j$ である。また他に収束できる範囲を広げる工夫を行う。これらの手法には各ステップでノルム $||F||$ を小さくすることを要求するものや、線形で近似できるような範囲から探索が逸脱しないように信頼領域 (trust region) を設定するものがある。

互いに独立な、正規分布のばらつき σ_i を持つデータ (t_i, y_i) を非線形モデル $Y(x,t)$ で近似するには、以下の形式の関数を使う。

$$f_i = \frac{Y(x,t_i) - y_i}{\sigma_i}$$

この節では、複数のパラメータで定義されるモデル関数をデータにフィットさせるようなケースを想定し、モデルパラメータを x で表す。与えられるデータがなんであれ、その独立変数は t で表す。

したがって、$Y_i = Y(x,t_i)$ とするときヤコビアンは $J_{ij} = (1/\sigma_i)\partial Y_i/\partial x_j$ で表される。

37.2　多次元非線形最小二乗法インスタンスの初期化

gsl_multifit_fsolver * gsl_multifit_fsolver_alloc (*const gsl_multifit_fsolver type * T*, *size_t n*, *size_t p*) [Function]

データ点数 n、パラメータ数 p、アルゴリズム T の最小二乗法インスタンスを生成し、そのインスタンスへのポインタを返す。インスタンスを生成するためのメモリが足りない場合は、NULL ポインタを返し、エラー・コード GSL_ENOMEM でエラー・ハンドラーを呼び出す。

gsl_multifit_fdfsolver * gsl_multifit_fdfsolver_alloc (*const gsl_multifit_fdfsolver_type * T*, *size_t n*, *size_t p*) [Function]

データ点数 n、パラメータ数 p、導関数を使うアルゴリズム T の最小二乗法インスタンスを生成し、そのインスタンスへのポインタを返す。たとえば以下のコードではレベンバーグ・マルカルト (Levenberg-Marquardt) 法のインスタンスをデータ数 100 点、パラメータ数 3 点で生成する。

```
const gsl_multifit_fdfsolver_type * T = gsl_multifit_fdfsolver_lmder;
gsl_multifit_fdfsolver * s = gsl_multifit_fdfsolver_alloc (T, 100, 3);
```

インスタンスを生成するためのメモリが足りない場合は、NULL ポインタを返し、エラー・コード GSL_ENOMEM でエラー・ハンドラーを呼び出す。

int gsl_multifit_fsolver_set (*gsl_multifit_fsolver * s*, *gsl_multifit_function * f*, *gsl_vector * x*) [Function]

生成されている最小二乗法のインスタンス s に、目的関数 f、探索開始点 x を使うように設定、あるいは再設定する。

int gsl_multifit_fdfsolver_set (*gsl_multifit_fdfsolver * s*, *gsl_function fdf * fdf*, *gsl_vector * x*) [Function]

生成されている最小二乗法のインスタンス s に、目的関数とその導関数を一度に計算する関数 *fdf*、探索開始点 x を使うように設定、あるいは再設定する。

void gsl_multifit_fsolver_free (*gsl_multifit_fsolver * s*) [Function]
void gsl_multifit_fdfsolver_free (*gsl_multifit_fdfsolver * s*) [Function]

この関数は最小二乗法のインスタンス s に割り当てられたメモリを解放する。

const char * gsl_multifit_fsolver_name (*const gsl_multifit_fdfsolver * s*) [Function]
const char * gsl_multifit_fdfsolver_name (*const gsl_multifit_fdfsolver * s*) [Function]

この関数は与えられたインスタンスが使っている最小二乗法の名前文字列へのポインタを返す。たとえば以下の文は、s is a 'lmder' solver のように出力する。

```
printf("s is a '%s' solver\n", gsl_multifit_fdfsolver_name (s));
```

37.3 目的関数の設定

利用者は最小化の対象となる、p 変数の関数を n 個指定しなければならない。一般的な目的関数をパラメータを用いて定義できるようにするため、目的関数は以下の構造体で定義する必要がある。

gsl_multifit_function [Data Type]

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`
: 引数 x、パラメータ $params$ のときの関数値ベクトル $f(x, params)$ を f に入れ、関数値が計算できないときには対応するエラーコードを返すように定義した関数へのポインタ。

`size_t n`
: 関数の個数。ベクトル f の要素の個数である。

`size_t p`
: 独立変数の個数。ベクトル x の要素の個数である。

`void * params`
: 関数のパラメータへのポインタ。

gsl_multifit_function_fdf [Data Type]

この型はパラメータを持つ一般的な関数と、その導関数であるヤコビアン行列を定義するための型である。

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`
: この関数は引数 x 、パラメータ $params$ のときの関数値ベクトル $f(x, params)$ を f に入れ、関数値が計算できないときには対応するエラーコードを返す。

`int (* df) (const gsl_vector * x, void * params, gsl_matrix * J)`
: この関数は引数 x、パラメータ $params$ のとき $n \times p$ 行列 $J_{ij} = \partial f_i(x, params)/\partial x_j$ を計算して J に入れ、関数値が計算できないときは対応するエラーコードを返す。

`int (* fdf) (const gsl_vector * x, void * params, gsl_vector * f, gsl_matrix * J)`
: この関数は引数 x 、パラメータ $params$ のときの関数値ベクトル f と J を計算する。$f(x)$ と $J(x)$ を別々の関数で計算するよりも、この関数を使ったほうが速い。

`size_t n`
: これは関数の個数、たとえばベクトル f の要素の個数である。

`size_t p`
: これは独立変数の個数で、たとえばベクトル x の要素の個数である。

`void * params`
: これは関数のパラメータへのポインタである。

37.4　繰り返し計算

以下の関数で繰り返し計算を実行する。関数はそれぞれ、インスタンスが持つアルゴリズムによる繰り返し計算による探索点の更新を一回行う。同じ関数が全てのアルゴリズムに使え、プログラムを書き換えることなく、実行時にアルゴリズムを切り替えることができる。

`int gsl_multifit_fsolver_iterate (`*gsl_multifit_fsolver * s*`)` [Function]

`int gsl_multifit_fdfsolver_iterate (`*gsl_multifit_fdfsolver * s*`)` [Function]

これらの関数は最小化法のインスタンス s の繰り返し計算を一回行う。予期しない問題が発生した場合はエラーコードを返す。

最適化構造体 (`gsl_multifit_fsolver` および `gsl_multifit_fdfsolver`) のインスタンス s は以下の情報を保持しており、これにより探索の進行状況を知ることができる。

`gsl_vector * x`
: 現在の探索点。

`gsl_vector * f`
: 現在の探索点での関数値。

`gsl_vector * dx`
: 現在の探索点と一つ前の探索点の位置の差。たとえば最後のステップベクトルの大きさ。

`gsl_matrix * J`
: 現在の探索点でのヤコビアン行列 (`gsl_multifit_fdfsolver` のみ)。

最小化法のインスタンスは現在の最良近似解を常に保持しており、以下の補助関数を使って参照することができる。

`gsl_vector * gsl_multifit_fsolver_position (`*const gsl_multifit_fsolver * s*`)` [Function]

`gsl_vector * gsl_multifit_fdfsolver_position (`*const gsl_multifit_fdfsolver * s*`)` [Function]

これらの関数は、インスタンス s のメンバー `s->x` で保持される現在の探索点 (最良近似パラメータ) を返す。

37.5　停止条件

最小化法は以下の条件のうちどれか一つが成立したとき、停止する。

- 設定された精度で最小点が見つかったとき。
- 設定された最大回数に繰り返し計算が達したとき。
- エラーが発生したとき。

これらの条件は任意に設定することができる。以下の関数でその時の最良探索点とその精度を調べることができる。

`int gsl_multifit_test_delta` (*const gsl_vector * **dx***, *const gsl_vector * **x***, *double **epsabs***, *double **epsrel***) [Function]

許容絶対誤差が *epsabs*、許容相対誤差が *epsrel* で与えられるときのステップ幅 *dx* の収束を判定する。*x* の各要素について以下の条件が成立しているときは `GSL_SUCCESS` を返し、成立していなければ `GSL_CONTINUE` を返す。

$$|dx_i| < epsabs + epsrel|x_i|$$

`int gsl_multifit_test_gradient` (*const gsl_vector * **g***, *double **epsabs***) [Function]

与えられる許容絶対誤差 *epsabs* で残差の勾配 g の収束を判定する。理論上、最小点では勾配は 0 になるはずである。以下の条件が成立しているときは `GSL_SUCCESS` を返し、成立していなければ `GSL_CONTINUE` を返す。

$$\sum_i |g_i| < epsabs$$

この判定基準は、最小点の正確な位置 x はあまり重要でなく、それでも勾配が十分に小さくなるような値が見つかるときに使う。

`int gsl_multifit_gradient` (*const gsl_matrix * **J***, *const gsl_vector * **f***, *gsl_vector * **g***) [Function]

この関数はヤコビアン行列 J と関数値 f から、$g = J^T f$ という関係を使って $\Phi(x) = (1/2)||F(x)||^2$ の勾配 g を計算する。

37.6 導関数を使う最小化法

この節の最小二乗法は、目的関数とその導関数の両方を用いる。最小点の推定値として探索開始点が指定されなければならない。また収束は保証されない。関数が以下の手法にうまくあうような形であり、探索開始点が最小点に十分に近いことが必要である。

`gsl_multifit_fdfsolver_lmsder` [Derivative Solver]

レベンバーグ・マルカルト法 (Levenberg-Marquardt algorithm) を一般化信頼領域 (generalize trust region) を用いるよう改良した MINPACK の `lmder` ルーチン (を C で実装し直したもの)。MINPACK は Jorge J. Moré、Burton S. Garbow、Kenneth E. Hillstrom によって書かれた FORTRAN ライブラリである。

このアルゴリズムはステップ幅を制御するために信頼領域を設定する。新しい探索点候補 x を採用するかどうかは、D を対角係数行列 (diagonal scaling matrix、x の変化量を各座標軸についてスケーリングする) とし、δ が信頼領域の大きさのとき、$|D(x' - x)| < \delta$

という条件を満たすかどうかで決定する。D の要素はヤコビアン行列の列ノルムを使って内部で計算され、これにより x の各成分が関数値の残差 (目的の値 $= 0$ との差) にどの程度の影響を持つか (感度、sensitivity) を推定できる。これにより、関数値が大きく変動するような特殊な挙動を示す関数での探索能力を向上する。

この方法では繰り返し計算の各回で、$|Dp| < \Delta$ という条件下で $|F + Jp|$ を最小化する。この制約付き線形問題の解をレベンバーグ・マルカルト法により得ることができる。

新しく決めたステップ・ベクトルは、そのステップをとったときの点 x で関数値がどうなるかで判定される。そのステップにより関数のノルムが十分に小さくなり、信頼領域内での関数の形が想定されているとおりであれば、そのステップを採用し、信頼領域を拡大する。そのステップでは関数値が改善されないか、そのステップによる信頼領域内の関数の形が想定と大きく異なるときは、信頼領域を縮小し、ステップを計算し直す。

このアルゴリズムでは解の改善の様子を見ながら、改善が計算機の精度よりも小さくなったときには、状況に応じて以下のようなエラーを返す。

`GSL_ETOLF`
: 関数値の減少が計算機の精度よりも小さくなることを示す。

`GSL_ETOLX`
: 探索点の変化が計算機の精度よりも小さくなることを示す。

`GSL_ETOLG`
: 関数のノルムに対する勾配のノルムの大きさが計算機の精度よりも小さくなることを示す。

これらのエラーは、それ以上繰り返し計算を続けても、その時点での解よりもよい解は得られないであろうことを示す。

`gsl_multifit_fdfsolver_lmder` [Derivative Solver]

これは係数行列を使わない `lmder` 法である。係数行列 D の対角成分は 1 である。この方法は、係数行列を使う `lmder` 法の収束が遅いときや、関数がすでに適切にスケーリングされているときに有効である。

37.7　導関数を使わない最小化法

現時点ではこの種の手法は実装されていない。

37.8　最良近似パラメータの共分散行列の計算

`int gsl_multifit_covar` (*const gsl_matrix * J*, *double* ***epsrel***, *gsl_matrix * covar*) [Function]

この関数はヤコビアン行列 J を使って最良近似パラメータの共分散行列 (covariance matrix) *covar* を計算する。J のランクが低いときには、*epsrel* を使って線形従属 (linear-dependent) な列を削除する。

以下で与えられる共分散行列は、ヤコビアン行列を列に対してピボッティングする QR 分解で計算される。

$$C = (J^T J)^{-1}$$

QR 分解で得られる R の各列が以下の関係を満たすとき、線形従属であるとみなされ、共分散行列から取り除かれる（共分散行列の対応する行と列の要素を 0 にする)。

$$|R_{kk}| \leq epsrel|R_{11}|$$

最適化される関数が $f_i = (Y(x, t_i) - y_i)/\sigma_i$ のように重み付きの場合、上記の共分散行列は最適パラメータ値におけるばらつきを与え、データ y_i のばらつきに正規分布モデルを仮定した場合の標準偏差 (Gaussian error) が σ_i となる。これは、$\delta f = J\delta c$ およびデータ y_i からの f の変位を σ_i で正規化すると $\langle \delta f \delta f^T \rangle = I$ となることから示される。

重み付きでない最適化関数 $f_i = (Y(x, t_i) - y_i)$ の場合、最適パラメータ値における残差に分散を乗じた $\sum (y_i - Y(x, t_i))^2/(n - p)$ は分散共分散行列 (variance-covariance matrix) $\sigma^2 C$ を与える。これによって、データのばらつきによる、最適パラメータ値におけるばらつきを統計的に見ることができる。

共分散行列については「最小二乗近似」の第 1 節 (447 ページ) を参照のこと。

37.9 例

以下のプログラムではバックグラウンド・ノイズのある重み付き指数モデル $Y = A\exp(-\lambda t) + b$ で実験データを近似する。プログラムの最初の部分ではモデルとそのヤコビアン行列を計算する関数 `expb_f` と `expb_df` を定義している。最良近似を与えるモデルは以下の式で表される。

$$f_i = ((A\exp(-\lambda t_i) + b) - y_i)/\sigma_i$$

ここでは $t_i = i$ とした。ヤコビアン行列 J は 3 つのパラメータ A, λ, b で上記の関数を微分したものであり、以下の式で表される。

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

ここでは $x_0 = A,\ x_1 = \lambda,\ x_2 = b$ である。

```
/* expfit.c -- 指数モデルと背景ノイズの和をモデリングした関数 */
struct data {
    size_t n;
    double * y;
    double * sigma;
```

```
};

int expb_f (const gsl_vector * x, void *data, gsl_vector * f)
{
    size_t n = ((struct data *)data)->n;
    double *y = ((struct data *)data)->y;
    double *sigma = ((struct data *) data)->sigma;
    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++) {
        /* モデル: Yi = A * exp(-lambda * i) + b */
        double t = i;
        double Yi = A * exp(-lambda * t) + b;
        gsl_vector_set(f, i, (Yi - y[i])/sigma[i]);
    }

    return GSL_SUCCESS;
}

int expb_df (const gsl_vector * x, void *data, gsl_matrix * J)
{
    size_t n = ((struct data *)data)->n;
    double *sigma = ((struct data *) data)->sigma;
    double A = gsl_vector_get(x, 0);
    double lambda = gsl_vector_get(x, 1);
    size_t i;

    for (i = 0; i < n; i++) {
        /* ヤコビアン行列: J(i,j) = dfi / dxj, */
        /* ただし fi = (Yi - yi)/sigma[i], */
        /*        Yi = A * exp(-lambda * i) + b */
        /*        xj は パラメータ (A,lambda,b) を表す */
        double t = i;
        double s = sigma[i];
        double e = exp(-lambda * t);
        gsl_matrix_set(J, i, 0, e/s);
        gsl_matrix_set(J, i, 1, -t * A * e/s);
```

```
            gsl_matrix_set(J, i, 2, 1/s);
        }
        return GSL_SUCCESS;
    }

    int expb_fdf (const gsl_vector * x, void *data, gsl_vector * f,
                  gsl_matrix * J)
    {
        expb_f(x, data, f);
        expb_df(x, data, J);
        return GSL_SUCCESS;
    }
```

プログラムの main 関数では、レベンバーグ・マルカルト法のインスタンスを初期化し、乱数によるデータを生成する。データは既知のパラメータ値 (1.0, 5.0, 0.1) を持つモデルに正規分布乱数 (標準偏差 0.1) を加えることで 40 点生成したものである。パラメータの初期推定値は (0.0, 1.0, 0.0) としている。

```
    #include <stdlib.h>
    #include <stdio.h>
    #include <gsl/gsl_rng.h>
    #include <gsl/gsl_randist.h>
    #include <gsl/gsl_vector.h>
    #include <gsl/gsl_blas.h>
    #include <gsl/gsl_multifit_nlin.h>
    #include "expfit.c"
    #define N 40

    void print_state (size_t iter, gsl_multifit_fdfsolver * s);

    int main (void)
    {
        const gsl_multifit_fdfsolver_type *T;
        gsl_multifit_fdfsolver *s;
        int status;
        size_t i, iter = 0;
        const size_t n = N;
        const size_t p = 3;
        gsl_matrix *covar = gsl_matrix_alloc (p, p);
        double y[N], sigma[N];
        struct data d = { n, y, sigma};
```

```
    gsl_multifit_function_fdf f;
    double x_init[3] = { 1.0, 0.0, 0.0 };
    gsl_vector_view x = gsl_vector_view_array (x_init, p);
    const gsl_rng_type * type;
    gsl_rng * r;

    gsl_rng_env_setup();
    type = gsl_rng_default;
    r = gsl_rng_alloc (type);
    f.f = &expb_f;
    f.df = &expb_df;
    f.fdf = &expb_fdf;
    f.n = n;
    f.p = p;
    f.params = &d;

    /* 近似対象となる観測データを生成 */
    for (i = 0; i < n; i++) {
        double t = i;
        y[i] = 1.0 + 5*exp(-0.1*t) + gsl_ran_gaussian(r, 0.1);
        sigma[i] = 0.1;
        printf("data: %d %g %g\n", i, y[i], sigma[i]);
    };
    T = gsl_multifit_fdfsolver_lmsder;
    s = gsl_multifit_fdfsolver_alloc(T, n, p);
    gsl_multifit_fdfsolver_set(s, &f, &x.vector);
    print_state(iter, s);

    do {
        iter++;
        status = gsl_multifit_fdfsolver_iterate(s);
        printf("status = %s\n", gsl_strerror (status));
        print_state(iter, s);
        if (status) break;
        status = gsl_multifit_test_delta(s->dx, s->x, 1e-4, 1e-4);
    } while (status == GSL_CONTINUE && iter < 500);

    gsl_multifit_covar(s->J, 0.0, covar);

#define FIT(i) gsl_vector_get(s->x, i)
```

```
#define ERR(i) sqrt(gsl_matrix_get(covar,i,i))

    double chi = gsl_blas_dnrm2(s->f);
    double dof = n - p;
    double c = GSL_MAX_DBL(1, chi / sqrt(dof));

    printf("chisq/dof = %g\n", pow(chi, 2.0) / dof);
    printf("A = %.5f +/- %.5f\n", FIT(0), c*ERR(0));
    printf("lambda = %.5f +/- %.5f\n", FIT(1), c*ERR(1));
    printf("b = %.5f +/- %.5f\n", FIT(2), c*ERR(2));

    printf("status = %s\n", gsl_strerror (status));
    gsl_multifit_fdfsolver_free(s);

    return 0;
}

void print_state (size_t iter, gsl_multifit_fdfsolver * s)
{
    printf("iter: %3u x = % 15.8f % 15.8f % 15.8f |f(x)| = %g\n",
           iter,
           gsl_vector_get (s->x, 0), gsl_vector_get (s->x, 1),
           gsl_vector_get (s->x, 2), gsl_blas_dnrm2 (s->f));
}
```

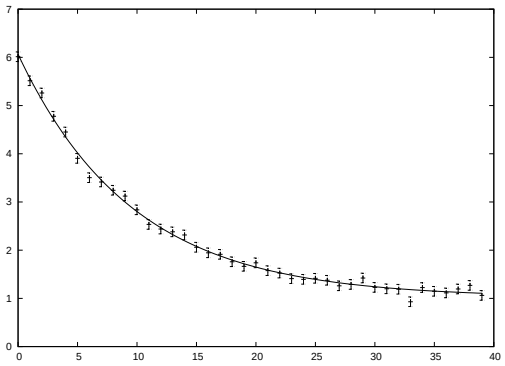
繰り返し計算は x の変化の絶対誤差と相対誤差の両方が 0.0001 よりも小さくなったときに停止する。プログラムを実行した結果を以下に示す。

```
iter: 0 x=1.00000000 0.00000000 0.00000000 |f(x)|=117.349
status=success
iter: 1 x=1.64659312 0.01814772 0.64659312 |f(x)|=76.4578
status=success
iter: 2 x=2.85876037 0.08092095 1.44796363 |f(x)|=37.6838
status=success
iter: 3 x=4.94899512 0.11942928 1.09457665 |f(x)|=9.58079
status=success
iter: 4 x=5.02175572 0.10287787 1.03388354 |f(x)|=5.63049
status=success
iter: 5 x=5.04520433 0.10405523 1.01941607 |f(x)|=5.44398
status=success
iter: 6 x=5.04535782 0.10404906 1.01924871 |f(x)|=5.44397
```

```
chisq/dof = 0.800996
A      = 5.04536 +/- 0.06028
lambda = 0.10405 +/- 0.00316
b      = 1.01925 +/- 0.03782
status = success
```

パラメータの近似値が正しく得られ、カイ二乗の値はこれがよい近似であることを示している (1 自由度当たりのカイ二乗の値は約 1 である)。求められたパラメータ値を持つモデルと与えられたデータとの誤差は、共分散行列中で対応する対角成分の平方根として得られる。

カイ二乗の値が、近似がよくないことを示しているとき (たとえば $\chi^2/(n-p) \gg 1$ のようなとき) は共分散行列から得られる誤差は非常に小さな値になる。近似がよくない場合、一般的には上のプログラムのように推定誤差の値に $\sqrt{\chi^2/(n-p)}$ をかけることでその値を大きくする。当てはめようとするモデルが悪ければ近似はうまくいかないこと、また値を操作した推定誤差は正規分布の適用範囲からははずれていることに注意せねばならない。



37.10　参考文献

MINPACK で使われているアルゴリズムは、以下の記事に述べられている。

- J. J. Moré, "The Levenberg-Marquardt Algorithm: Implementation and Theory", *Lecture Notes in Mathematics*, v630, ed G. Watson (1978).

以下の論文も、ここで使っているアルゴリズムに関連が深い。

- Jorge J. Moré, Burton S. Garbow, Kenneth E. Hillstrom, "Testing Unconstrained Optimization Software", *ACM Transactions on Mathematical Software*, **7**(1), pp. 17–41 (1981)

第38章　B スプライン

この章ではデータの平滑化 (smoothing) に用いられる B スプライン (basis spline, B-spline) について説明する。B スプラインに関する宣言や定義はヘッダファイル 'gsl_bspline.h' にある。

38.1　概要

B スプラインは、大規模なデータに滑らかな曲線を当てはめるときに、その基底関数 (basis function) として広く用いられている。B スプラインで平滑化を行うためにはまず、横軸 (x 軸や t 軸など) を複数の小区間に分割する。各小区間の両端を「区切り点 (breakpoint)」と呼ぶ。各区切り点から、連続性と滑らかさに関する条件を満たす「節点 (knot)」が決まる。節点を昇順に並べたベクトル $t = \{t_0, t_1, \ldots, t_{n+k-1}\}$ が与えられた時、k 次の B スプラインは以下で定義される。

$$B_{i,1}(x) = \begin{cases} 1, & t_i \le x < t_{i+1} \\ 0, & else \end{cases}$$

$$B_{i,k}(x) = [(x - t_i)/(t_{i+k-1} - t_i)]B_{i,k-1}(x) + [(t_{i+k} - x)/(t_{i+k} - t_{t+1})]B_{i+1,k-1}(x)$$

ここで $i = 0, \ldots, n-1$ である。よく使われる 3 次の B スプラインは $k = 4$ である。この漸化式はド・ブーア (Carl(-Wilhelm Reinhold) de Boor) による、数値的に安定なアルゴリズムである。

閉区間 $[a, b]$ 内に適切に節点をとることによって、B スプラインによる基底関数はその区間内で完備集合 (complete set) になる。したがって、十分な数のデータ点 $(x_j, f(x_j))$ を与えることにより、データを平滑化して表現する関数 (smoothing function) は以下のように展開して表される。

$$f(x) = \sum_{i=0}^{n-1} c_i B_{i,k}(x)$$

c_i は最小二乗近似 (least-squares fitting) により容易に得られる。

38.2　B スプラインを得る関数の初期化

`gsl_bspline_workspace * gsl_bspline_alloc` (*const size_t* **`k`**, *const size_t* **`nbreak`**)
[Function]

k 次の B スプラインのための作業領域を確保する。区切り点の個数を *nbreak* で指定する。基底関数の数は $n = nbreak + k - 2$ となる。3 次の B スプライン (cubic B-spline) は $k = 4$ である。作業領域の大きさは $O(5k + nbreak)$ のオーダーである。

void gsl_bspline_free (*gsl_bspline_workspace* * **w**) [Function]

B スプラインのための作業領域 w を解放する。

38.3 節点ベクトルの計算

int gsl_bspline_knots (*const gsl_vector* * **breakpts**, *gsl_bspline_workspace* * **w**) [Function]

与えられた区切り点から節点を求め、w->knots に入れて返す。

int gsl_bspline_knots_uniform (*const double* **a**, *conts double* **b**, *gsl_bspline_workspace* * **w**) [Function]

区切り点の数が *nbreak* で区間 $[a, b]$ 内で等間隔であるとして、対応する節点を求め、節点ベクトルとして w->knots に入れて返す。

38.4 B スプラインの計算

int gsl_bspline_eval (*const double* **x**, *gsl_vector* **B**, *gsl_bspline_workspace* * **w**) [Function]

点 x における B スプラインを求め、B に入れて返す。B の i 番目の要素が $B_i(x)$ になる。B の大きさ n は $n = nbreak + k - 2$ でなければならない。必要となる n の値は gsl_bspline_ncoeffs で調べることができる。基底関数は漸化式で計算できるため、各基底関数を個別に求めるよりは、すべて一度に計算する方がはるかに効率がよい。

size_t gsl_bspline_ncoeffs (*gsl_bspline_workspace* * **w**) [Function]

$n = nbreak + k - 2$ で与えられる B スプラインの係数の個数を返す.

38.5 例

以下のプログラムでは、区切り点が等間隔の 3 次の B スプラインを使って線形の最小二乗近似を行う。データは区間 $[0, 15]$ において $y(x) = \cos(x)\exp(-x/10)$ にガウシアン・ノイズを加えて生成する。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_rng.h>
```

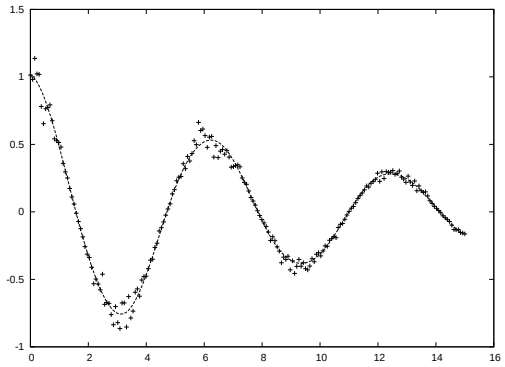
```
#include <gsl/gsl_randist.h>
#include <gsl/gsl_statistics.h>

/* フィットするデータ点数 */
#define N 200

/* 係数の個数 */
#define NCOEFFS 12

/* 区切り点の数: k = 4 なので、nbreak = ncoeffs + 2 - k = ncoeffs - 2 */
#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;
    double Rsq;
    double dof;

    gsl_rng_env_setup();
    r = gsl_rng_alloc(gsl_rng_default);

    /* 3 次の B スプラインのための作業領域確保 (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);

printf("#m=0,S=0\n"); /* plotutil の graph コマンドのための出力 */
/* データ生成 */
for (i = 0; i < n; ++i) {
    double sigma;
    double xi = (15.0 / (N - 1)) * i;
    double yi = cos(xi) * exp(-0.1 * xi);

    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);
}

/* フィッティングする区間 [0, 15] を等間隔に分割する */
gsl_bspline_knots_uniform(0.0, 15.0, bw);

/* 各点における基底関数の係数による行列 X を求める*/
for (i = 0; i < n; ++i) {
    double xi = gsl_vector_get(x, i);

    /* 各 j について B_j(xi) を計算する */
    gsl_bspline_eval(xi, B, bw);

    /* X の第 i 行を計算 */
    for (j = 0; j < ncoeffs; ++j) {
        double Bj = gsl_vector_get(B, j);
        gsl_matrix_set(X, i, j, Bj);
    }
}
```

```
    /* フィットを行う */
    gsl_multifit_wlinear(X, w, y, c, cov, &chisq, mw);

    dof = n - ncoeffs;
    Rsq = 1.0 - chisq / gsl_stats_wtss(w->data, 1, y->data, 1, y->size);

    fprintf(stderr, "chisq/dof = %e, Rsq = %f\n", chisq / dof, Rsq);

    /* 滑らかになった曲線を出力 */
    double xi, yi, yerr;
    printf("#m=1,S=0\n"); /* plotutil の graph コマンドのための出力 */
    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() */
```

GNU graph を使って、結果をプロットできる。

```
$ ./a.out > bspline.dat
chisq/dof = 1.118217e+00, Rsq = 0.989771
$ graph -T ps -X x -Y y -x 0 15 -y -1 1.3 < bspline.dat > bspline.ps
```



38.6　参考文献

より詳しいことは、以下の書籍を参照されたい。

- Carl de Boor, *A Practical Guide to Splines* (Applied Mathematical Sciences **27**), Springer-Verlag, ISBN 0-387-90356-9 (1978).

上の書籍のアルゴリズムを FORTRAN で実装し、多数の B スプライン関連の関数を収録しているライブラリ PPPACK (Piece-wise Polynomial Package) が Netlib `http://www.netlib.org/pppack` から入手できる。場合によっては `http://people.sc.fsu.edu/ burkardt/f_src/pppack/pppack.html` も参考になるかもしれない。そこには FORTRAN 90、77 の両バージョンの PPPACK がある。

第39章　物理定数

この章では、光速 (speed of light) c や重力定数 (gravitational constant)G などの物理定数の値を定義するマクロについて説明する。標準的な MKSA (メートル、キログラム、秒、アンペア) と天文学などで用いられている CGSM (センチメートル、グラム、秒、ガウス) で利用できる。

MKSA での定数の定義はヘッダファイル 'gsl_const_mksa.h' にある。CGSM での定義は 'gsl_const_cgsm.h' にある。微細構造定数 (fine structure constant) のような無次元の定数は 'gsl_const_num.h' に 単なる数値として宣言されている。

以下に定義されている全ての定数を簡単な説明を付けて列挙する。各定数の値とその単位については、各ヘッダファイルを参照のこと。

39.1　基本的な定数

GSL_CONST_MKSA_SPEED_OF_LIGHT 真空中での光速 c。

GSL_CONST_MKSA_VACUUM_PERMEABILITY 真空中の透磁率 (permeability) μ_0。MKSA でのみ定義されている。

GSL_CONST_MKSA_VACUUM_PERMITTIVITY 真空中の誘電率 (permittivity) ϵ_0。MKSA でのみ定義されている。

GSL_CONST_MKSA_PLANCKS_CONSTANT_H プランク定数 (Planck's constant) h。

GSL_CONST_MKSA_PLANCKS_CONSTANT_HBAR プランク定数を 2π で除した値 $\hbar$。

GSL_CONST_NUM_AVOGADRO アボガドロ定数 (Avogadro's number) N_a。

GSL_CONST_MKSA_FARADAY 1 ファラデー (Faraday) の電荷 (アボガドロ数個の電子の総電荷)。

GSL_CONST_MKSA_BOLTZMANN ボルツマン定数 (Boltzmann constant) k。

GSL_CONST_MKSA_MOLAR_GAS 気体定数 (molar gas constant) R_0 (1 モルの理想気体における PV/T)。

GSL_CONST_MKSA_STANDARD_GAS_VOLUME 標準状態の気体の体積 (standard gas voloume) V_0 。

GSL_CONST_MKSA_STEFAN_BOLTZMANN_CONSTANT ステファン・ボルツマン (Stefan-Boltzmann) の放射定数 (radiation constant) σ。

GSL_CONST_MKSA_GAUSS 1 ガウス (Gauss) の磁場。

39.2　天文学と天文物理学

GSL_CONST_MKSA_ASTRONOMICAL_UNIT 1 天文単位 (astronomical unit、地球と太陽の平均距離) au。

GSL_CONST_MKSA_GRAVITATIONAL_CONSTANT 重力定数 G。

GSL_CONST_MKSA_LIGHT_YEAR 1 光年 (light-year) の距離 ly。

GSL_CONST_MKSA_PARSEC 1 パーセク (parsec) の距離 pc。

GSL_CONST_MKSA_GRAV_ACCEL 地球上での重力加速度の標準値 (standard gravitational acceleration) g。

GSL_CONST_MKSA_SOLAR_MASS 太陽の質量。

39.3　原子物理学、核物理学

GSL_CONST_MKSA_ELECTRON_CHARGE 電子 1 個の電荷 e。

GSL_CONST_MKSA_ELECTRON_VOLT 1 電子ボルト (electron volt) のエネルギー eV。

GSL_CONST_MKSA_UNIFIED_ATOMIC_MASS 原子量 (atomic mass) の単位 amu。

GSL_CONST_MKSA_MASS_ELECTRON 電子の質量 m_e。

GSL_CONST_MKSA_MASS_MUON ミューオン (muon) の質量 m_μ。

GSL_CONST_MKSA_MASS_PROTON 陽子の質量 m_p。

GSL_CONST_MKSA_MASS_NEUTRON 中性子の質量 m_n。

GSL_CONST_NUM_FINE_STRUCTURE 電磁気的な微細構造定数 α。

GSL_CONST_MKSA_RYDBERG エネルギーの単位 (次元) を持つリュードベリ定数 (Rydberg constant) Ry。この値はリュードベリ波長の逆数 (いわゆるリュードベリ定数、inverse wavelength、wavenumber) R_∞ から $Ry = hcR_\infty$ として得られる。

GSL_CONST_MKSA_BOHR_RADIUS ボーア半径 (Bohr radius) a_0。

GSL_CONST_MKSA_ANGSTROM 1 オングストローム (angstrom) の長さ。

GSL_CONST_MKSA_BARN 1 バーン (barn) の面積。

GSL_CONST_MKSA_BOHR_MAGNETON ボーア磁子 (Bohr magneton) μ_B。

GSL_CONST_MKSA_NUCLEAR_MAGNETON 核磁子 (Nuclear magneton) μ_N。

GSL_CONST_MKSA_ELECTRON_MAGNETIC_MOMENT 電子の磁気モーメント (magnetic moment) の絶対値 μ_e。電子の磁気モーメントは、実際には負の値である。

GSL_CONST_MKSA_PROTON_MAGNETIC_MOMENT 陽子の磁気モーメント μ_p。

GSL_CONST_MKSA_THOMSON_CROSS_SECTION トムソン断面積 (Thomson cross section) σ_T。

GSL_CONST_MKSA_DEBYE 1 デバイ (Debye) の双極子モーメント (dipole moment) D。

39.4 時間の単位

GSL_CONST_MKSA_MINUTE 1 分の秒数。

GSL_CONST_MKSA_HOUR 1 時間の秒数。

GSL_CONST_MKSA_DAY 1 日の秒数。

GSL_CONST_MKSA_WEEK 1 週間の秒数。

39.5 ヤード・ポンド法

GSL_CONST_MKSA_INCH 1 インチの長さ。

GSL_CONST_MKSA_FOOT 1 フィートの長さ。

GSL_CONST_MKSA_YARD 1 ヤードの長さ。

GSL_CONST_MKSA_MILE 1 マイルの長さ。

GSL_CONST_MKSA_MIL 1 ミル (1 インチの 1/1000) の長さ。

39.6 速度および海事で用いる単位

GSL_CONST_MKSA_KILOMETERS_PER_HOUR 時速 1 キロメートルの速度。

GSL_CONST_MKSA_MILES_PER_HOUR 時速 1 マイルの速度。

GSL_CONST_MKSA_NAUTICAL_MILE 1 海里 (nautical mile) の長さ。

GSL_CONST_MKSA_FATHOM 1 尋 (fathom) の長さ。

GSL_CONST_MKSA_KNOT 1 ノット (knot) の速度。

39.7 印刷、組版で用いる単位

GSL_CONST_MKSA_POINT 1 ポイントの長さ (1/72 インチ)。

GSL_CONST_MKSA_TEXPOINT TEX での 1 ポイントの長さ (1/72.27 インチ)。

39.8　長さ、面積、容積

GSL_CONST_MKSA_MICRON 1 ミクロンの長さ。

GSL_CONST_MKSA_HECTARE 1 ヘクタール (hectare) の面積。

GSL_CONST_MKSA_ACRE 1 エーカー (acre) の面積。

GSL_CONST_MKSA_LITER 1 リットルの容積。

GSL_CONST_MKSA_US_GALLON 1 US ガロン (US gallon) の容積。

GSL_CONST_MKSA_CANADIAN_GALLON 1 カナダガロン (Canadian gallon) の容積。

GSL_CONST_MKSA_UK_GALLON 1 UK ガロン (UK gallon) の容積。

GSL_CONST_MKSA_QUART 1 クォート (quart) の容積。

GSL_CONST_MKSA_PINT 1 パイント (pint) の容積。

39.9　質量と重さ

GSL_CONST_MKSA_POUND_MASS 1 ポンド (pound) の質量。

GSL_CONST_MKSA_OUNCE_MASS 1 オンス (ounce) の質量。

GSL_CONST_MKSA_TON 1 トン ton の質量（訳注:米トン/小トン)。

GSL_CONST_MKSA_METRIC_TON 1 メートルトン (metric ton) の質量 (1000kg)。

GSL_CONST_MKSA_UK_TON 1 英トン (UK ton) の質量。

GSL_CONST_MKSA_TROY_OUNCE 1 トロイオンス (troy ounce、金衡オンス) の質量。

GSL_CONST_MKSA_CARAT 1 カラット (carat) の質量。

GSL_CONST_MKSA_GRAM_FORCE 1 グラム重 (gram weight) の力。

GSL_CONST_MKSA_POUND_FORCE 1 ポンド重 (pound weight) の力。

GSL_CONST_MKSA_KILOPOUND_FORCE 1 キロポンド重 (kilopound weight) の力。

GSL_CONST_MKSA_POUNDAL 1 ポンダル (poundal、パウンダル) の力。

39.10 熱エネルギーと仕事率

GSL_CONST_MKSA_CALORIE 1 カロリー (calorie) のエネルギー。

GSL_CONST_MKSA_BTU 1 英国熱量単位 (British Thermal Unit) のエネルギー *btu*。

GSL_CONST_MKSA_THERM 1 サーム (therm, 100000 btu) のエネルギー。

GSL_CONST_MKSA_HORSEPOWER 1 馬力 (hoursepower) の仕事率 (power)。

39.11 圧力

GSL_CONST_MKSA_BAR 1 バール (bar) の圧力。

GSL_CONST_MKSA_STD_ATMOSPHERE 1 標準気圧 (standard atmosphere) の圧力。

GSL_CONST_MKSA_TORR 1 トル (torr) の圧力。

GSL_CONST_MKSA_METER_OF_MERCURY 1 水銀柱メートル (meter of mercury) の圧力。

GSL_CONST_MKSA_INCH_OF_MERCURY 1 水銀柱インチ (inch of mercury) の圧力。

GSL_CONST_MKSA_INCH_OF_WATER 1 水柱インチ (inch of water) の圧力。

GSL_CONST_MKSA_PSI 1 ポンド毎平方インチ (pound per square inch) の圧力。

39.12 粘性

GSL_CONST_MKSA_POISE 1 ポアズ (poise) の粘度 (粘性率、dynamic viscosity)。

GSL_CONST_MKSA_STOKES 1 ストークス (stokes) の動粘度 (動粘性率、kinematic viscosity)。

39.13 光と明かるさ

GSL_CONST_MKSA_STILB 1 スチルブ (stilb) の輝度 (luminance)。

GSL_CONST_MKSA_LUMEN 1 ルーメン (lumen) の光束 (luminous flux)。

GSL_CONST_MKSA_LUX 1 ルクス (lux) の照度 (illuminance)。

GSL_CONST_MKSA_PHOT 1 フォト (phot) の照度 (illuminance)。

GSL_CONST_MKSA_FOOTCANDLE 1 フート燭 (footcandle) の照度 (illuminance)。

GSL_CONST_MKSA_LAMBERT 1 ランバート (lambert) の輝度 (luminance)。

GSL_CONST_MKSA_FOOTLAMBERT 1 フート・ランバート (footlambert) の輝度 (luminance)。

39.14　放射性

GSL_CONST_MKSA_CURIE 1 キュリー (curie) の放射能 (radioactivity)。

GSL_CONST_MKSA_ROENTGEN 1 レントゲン (roentgen) の照射線量 (exposure)。

GSL_CONST_MKSA_RAD 1 ラド (rad) の吸収線量 (absorbed dose)。

39.15　力とエネルギー

GSL_CONST_MKSA_NEWTON SI 系での力の単位、1 ニュートン (Newton)。

GSL_CONST_MKSA_DYNE 1 ダイン (dyne) の力。10^{-5} ニュートン。

GSL_CONST_MKSA_JOULE SI 系でのエネルギーの単位、1 ジュール (Joule)。

GSL_CONST_MKSA_ERG 1 エルグ (erg) のエネルギー。10^{-7} ジュール。

39.16　接頭辞

これらは無次元の係数である。

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^{3}

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}

39.17 例

以下に、上に挙げた物理的な定数を計算に使うプログラム例を示す。地球から火星まで光速でどのくらいの時間がかかるかを計算する。

必要となるデータは、両惑星の太陽からの平均距離 (天文単位。ここでは軌道の離心率は考慮しない) と光速である。火星の公転軌道の平均半径は 1.52 天文単位であり、地球のは定義上 1 である。到達に要する最大時間と最短時間を秒数で求めるため、光速と天文単位にあうように、これらの値を MKS 単位系での値に換算して計算を行う。プログラムの出力では、表示される直前に秒から分に換算する。

```
#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;

    double r_earth = 1.00 * au; /* 距離はメートル単位になっている */
    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;
}
```

プログラムの出力を以下に示す。

```
light travel time from Earth to Mars:
minimum = 4.3 minutes
maximum = 21.0 minutes
```

39.18　参考文献

以下に示す “CODATA recommended values” に信頼できる物理定数の値が載っている。また、NIST のウェブサイトで詳細を見ることができる。

- Peter J. Mohr, Barry N. Taylor, David B. Newell, “CODATA Recommended Values of the Fundamental Physical Constants: 2006”, *Reviews of Modern Physics*, **80**(2), pp. 633–730 (2008).
- Peter J. Mohr, Barry N. Taylor, “CODATA recommended values of the fundamental physical constants: 2002”, *Reviews of Modern Physics*, **77**(1), pp. 1–107 (2005).
- Peter J. Mohr, Barry N. Taylor, “CODATA recommended values of the fundamental physical constants: 1998”, *Reviews of Modern Physics*, **72**(2), pp. 351–495 (2000).
- Peter J. Mohr, Barry N. Taylor, “CODATA Recommended Values of the Fundamental Physical Constants: 1998”, *Journal of Physical and Chemical Reference Data*, **28**(6), pp. 1713–1852 (1999).
- `http://www.physics.nist.gov/cuu/Constants/index.html`
- `http://physics.nist.gov/Pubs/SP811/appenB9.html`

第40章　IEEE浮動小数点演算

この章では浮動小数点数 (floating point number) の内部表現を調べ、プログラム内でそれを操作するための関数について説明する。この章で説明する関数はヘッダファイル 'gsl_ieee_utils.h' で宣言されている。

40.1　浮動小数点の内部表現

IEEE が定める標準の二進数浮動小数点演算 (Binary Floating-Point Arithmetic) では、単精度 (single precision) 及び倍精度実数 (double precision) の二進数表現 (binary format) を定義している。一つの実数は符号ビット (sign bit, s)、指数部 (exponent, E)、仮数部 (fraction, f) の三つの部分から成り立っている。以下のように、この 3 つの二進数の値によって、一つの実数が表現される。

$$(-1)^s(1 \cdot fffff\ldots)2^E$$

符号ビットは 0 か 1 のどちらかである。指数部は精度によって異なり、最小値 E_{min} から最大値 E_{max} の範囲内の値である。指数部は実際にはバイアス付き指数部 (biased exponent) と呼ばれる符号なしの値 e で表現されており、それが実際に意味する指数値は、バイアス・パラメータ $bias$ を使って $E = e - bias$ として変換した値である。上の式の $fffff\ldots$ は二進数の仮数部 f で、指数部を調整して仮数部の先頭ビットが 1 になるようにする正規形式 (normalized form) になっている。正規形式では仮数部の先頭ビットは常に 1 なので、表現上は省かれる。$2^{E_{min}}$ よりも小さな数は、先頭に 0 をつけた以下の非正規形式 (denormalized form) で表現される。

$$(-1)^s(0 \cdot fffff\ldots)2^{E_{min}}$$

これにより精度が p ビットのときのアンダーフローを $2^{E_{min}-p}$ で済ませることができる。0 は指数部を $2^{E_{min}-1}$ に、無限大は指数部を $2^{E_{max}+1}$ にすることで表現する。

単精度実数は 32 ビットで以下のように表現される。

```
seeeeeeeefffffffffffffffffffffff

s = 符号ビット、1 ビット
e = 指数部、8 ビット (E_min=-126, E_max=127, バイアス=127)
f = 仮数部、23 ビット
```

倍精度実数は 64 ビットで以下のように表現される。

```
seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffffff
s = 符号ビット、1 ビット
e = 指数部、11 ビット (E_min=-1022, E_max=1023, バイアス=1023)
f = 仮数部、52 ビット
```

このようなビット単位で計算の様子を調べることが便利なことも場合によってはある。GSL では人に読める形式で IEEE 表現を出力する関数を用意している。

`void gsl_ieee_fprintf_float (`*FILE * **stream***, *const float * **x***`)` [Function]
`void gsl_ieee_fprintf_double (`*FILE * **stream***, *const double * **x***`)` [Function]

与えられる変数 *x* が指す IEEE 浮動小数点数の内部表現を *stream* に出力する。ポインタを使うのは、引数で渡すときに `float` から `double` に自動的に型変換 (キャスティング) されるのを避けるためである。出力は、以下のいずれかの形式である。

`NaN`
: Not-a-Number、どの数値でもないことを表す。

`Inf, -Inf`
: 正または負の無限大。

`1.fffff...*2^E, -1.fffff...*2^E`
: 正規形式の浮動小数点の内部表現。

`0.fffff...*2^E, -0.fffff...*2^E`
: 非正規形式の浮動小数点の内部表現。

`0, -0`
: 正または負の 0 であることを表す。

これらの出力は `2#` を前に付けて二進数であることを明示すれば、GNU Emacs の Calc モードでそのまま利用することができる。

`void gsl_ieee_printf_float (`*const float * **x***`)` [Function]
`void gsl_ieee_printf_double (`*const double * **x***`)` [Function]

与えられる変数 *x* が指す IEEE 浮動小数点数の内部表現を `stdout` に出力する。

以下のプログラムは仮数部が 1/3 のときの単精度及び倍精度実数の内部表現を表示する。比較を容易にするため、単精度の出力には倍精度に変換したときの内部表現も同時に表示する。

```
#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; /* 単精度から倍精度にキャスティング */

    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 を二進数で表現すると 0.01010101 . . . である。以下の出力例では、正規形式に加えて仮数部の先頭ビットの 1 が表示されている。

```
 f= 1.01010101010101010101011*2^-2
fd= 1.01010101010101010101011000000000000000000000000000000*2^-2
 d= 1.01010101010101010101010101010101010101010101010101010*2^-2
```

単精度から倍精度への変換は、二進数表現に 0 を付け加えることで行われることが見て取れる。

40.2 IEEE 演算環境の設定

IEEE 規格では浮動小数点演算の実行の制御モード (mode) がいくつか定義されている。各モードでは重要な演算特性、たとえば丸めの方向 (最も近い数値に切り上げるのか、切り捨てるのか) や丸め精度、零除算などの例外の処理の方法などが定義される。

これらの特性は、fpsetround のような標準関数が使えるときはそれで制御できる。しかし昔はこういった処理などを制御するための汎用の API がなく、処理系ごとに独自の低レベルなアクセス法を用意していた。移植性の高いプログラムを書けるようにするため、GSL では環境変数 GSL_IEEE_MODE を使ってプラットフォームに依存しない方法でモードを指定できるようにしている。また gsl_ieee_env_setup 関数を呼ぶことで、実行環境によって個別に必要となる初期化処理ができる。

`void gsl_ieee_env_setup ()` [Function]

環境変数 GSL_IEEE_MODE の値を参照し、対応する IEEE モードを設定する。環境変数の値は以下のような、コンマで区切られたキーワードのリストとされる。

```
GSL_IEEE_MODE = "keyword,keyword,..."
```

指定できる *keyword* は以下のモード名のどれかである。

```
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 の内容が空、または環境変数が定義されていない場合はプラットフォーム上のその時の IEEE モードのまま、なにも変更されない。GSL_IEEE_MODE に指定されるキーワードで対応する動作が ON になった場合、それから後のプログラムの動作にそれが反映されることを示すため、短くその旨が表示される。

利用しているプラットフォームでは無効なキーワードが指定されていた場合はエラー・ハンドラーが呼び出され、エラー・コード GSL_EUNSUP を返す。

デフォルトでは、指定されていないモードに関しては精度は可能な限り高く (倍精度か、プラットフォームによってはさらに高精度な型)、丸めモードは round-to-nearest に設定され、INEXACT 例外以外のすべての例外が有効になる。INEXACT 例外は数値の丸めが起こるたびに発生してしまうため、特に必要とされる場合以外は無効にしておいた方がよい。その他の浮動小数点例外は、アンダーフローや非正規化数 (denormalized number) を含め、安全を期すためデフォルトではすべて有効である。個別の例外を無効にするには mask- を、すべて無効にするには mask-all を使う。

多くの場合、以下のようなモードの組合せにしておくと便利である。

```
GSL_IEEE_MODE="double-precision,"\
              "mask-underflow,"\
              "mask-denormalized"
```

この組合せでは、小さな数に関するエラー (非正規化形式や 0 になるアンダーフロー) が無視されるが、オーバーフロー、零除算、無効な演算は検知される。

x86 シリーズのプロセッサでは、これらの関数は x87 モードと、SSE 浮動小数点演算を制御する MXCSR モードの両方を設定する。SSE 浮動小数点ユニットは精度を設定するフラグを持たないため、常に倍精度で動作する。この場合は単精度、あるいは拡張精度を指定しても無視される。

丸めのモードを切り替えるとどうなるかを、非常に速く収束する級数

$$e = 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \ldots = 2.71828182846\ldots$$

を使って自然対数の底 e を計算するプログラムで示す。

```
#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;

    gsl_ieee_env_setup(); /* GSL_IEEE_MODE の読み込み*/

    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;
}
```

まず round-to-nearest モードで実行した結果を以下に示す。これは IEEE でのデフォルトのモードなので、特に指定しなければこのモードになる。

```
$ 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
```

第 19 項で級数は正しい値に、誤差 4×10^{-16} で収束している。次にモードを round-down にして実行した場合の結果を以下に示す。精度が落ちているのが分かる。

```
$ GSL_IEEE_MODE="round-down" ./a.out
i= 1 sum=1.000000000000000000 error=-1.71828
....
i=19 sum=2.718281828459041094 error=-3.9968e-15
```

収束した値は正解より約 4×10^{-15} だけ小さく、round-to-nearest モードで計算した場合よりも正しく求められた桁数が少ない。

モードを round-up にすると収束しなくなる (各項を加えるたびに級数の和が切り上げられ、必ず値が増加していくため)。これを避けるには適切な epsilon の値を設定して、while (fabs(sum - oldsum) > epsilon) のような安全な収束条件にする必要がある。

最後にデフォルトの round-to-nearest モードで単精度で丸めを行った場合の級数計算のの例を示す。プログラム内では倍精度を使っていることになっているが、CPU は浮動小数点演算のたびに単精度で丸めを行う。この例で double 型の代わりに単精度の float を使うとどうなるかがわかる。繰り返し計算は約半分の回数で終了し、結果の精度は悪くなる。

```
GSL_IEEE_MODE="single-precision" ./a.out
....
i=12 sum=2.718281984329223633 error=1.5587e-07
```

収束したときの誤差は $O(10^{-7})$ であり、単精度実数の精度 (約 10^7 分の 1) と同じである。これ以上繰り返し計算を続けても、その後の結果はすべて切り捨てられて同じ値になるため、誤差が小さくなることはない。

40.3 参考文献

IEEE 規格の定義は、以下にある。

- *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.

もう少し教科書的な解説が以下の論文にある。

- David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys*, **23**(1), pp. 5–48 (1991).
- Corrigendum, *ACM Computing Surveys*, **23**(3), p. 413 (1991).
- B. A. Wichmann, Charles B. Dunham in Surveyor's Forum, "What Every Computer Scientist Should Know About Floating-Point Arithmetic". *ACM Computing Surveys*, **24**(3), p. 319 (1992).

SIAM (SIAM Press) から、IEEE 演算と実例について、詳しい教科書が出ている。

- Michael L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM Press, ISBN 0898715717 (2001).

付 録A　数値計算プログラムのデバッグ

この章では GSL を使った数値計算プログラムをデバッグするときのノウハウ (tips and tricks) について説明する。

A.1　gdb を使う場合

ライブラリの関数から報告されるエラーはすべて、関数 gsl_error に渡される。gdb 上でプログラムを実行するときにブレイク・ポイントをこの関数の中に設定することで、ライブラリ内で生じるエラーを自動的に捕まえることができる。

```
break gsl_error
```

上の行をプログラムを実行するディレクトリの '.gdbinit' ファイルに書いておくと、自動的に gsl_error にブレイク・ポイントを設定できる。

ブレイク・ポイントでエラーを捕まえたら、バックトレース・コマンド (bt) で関数呼び出しのレベルと、その各関数呼び出しでの引数 (エラーの原因かもしれない) を確認することができる。また、呼び出し側の関数に戻っていくことで、呼び出し時点での各変数の値を確認することができる。たとえば、プログラム fft/test_trap には以下の行が含まれているとする。

```
status = gsl_fft_complex_wavetable_alloc (0, &complex_wavetable);
```

関数 gsl_fft_complex_wavetable_alloc は、1 番目の引数として FFT の長さを指定される。この行が実行されると、FFT では長さ 0 の指定を受け付けないことになっているので、エラーが発生する。

これをデバッグするには、まずファイル '.gdbinit' 中でブレイク・ポイントを gsl_error に設定して gdb を起動する。

```
$ 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
```

プログラムを gdb 上で実行すると、発生したエラーがブレイク・ポイントで捕まえられ、エラーの原因が表示される。

```
(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)
```

gsl_error の 1 番目の引数は、どのようなエラーかを説明する文字列である。どんな問題が起こったのかを見るため、バックトレースしてみると、以下のようになる。

```
(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__ ()
```

これを見ると、エラーが発生したのは関数 gsl_fft_complex_wavetable_alloc が呼び出されたときで、そのときの 1 番目の引数が n=0 であることがわかる。またその引数で呼び出しているのはファイル 'test_trap.c' の 94 行目であることもわかる。

間違った引数で呼び出しているレベルまで上がることで、その行を見ることができる。

```
(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);
```

このレベルまで上がると、complex_wavetable などの他の変数の値を表示することもできる。

A.2 浮動小数点レジスタの確認

浮動小数点レジスタ (floating point register) の内容は info float で (これをサポートしているシステムなら) 確認することができる。

```
(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
```

reg という名前のレジスタの内容はを *$reg* で見ることができる。

```
(gdb) p $st1
$1 = 0.028594644542612103477 19
```

A.3 浮動小数点例外の処理

浮動小数点例外 (floating point exception) SIGFPE は、設定によってその発生時にプログラムの実行を終了させたり、あるいは無視したりすることができる。想定外の無限大や NaN の発生を見つけたいときに便利である。設定がどうなっているかはコマンド info signal SIGFPE で確認できる。

```
(gdb) info signal SIGFPE
Signal  Stop  Print  Pass to program Description
SIGFPE  Yes   Yes    Yes             Arithmetic exception
```

デフォルトの設定を変更して、プログラム中でシグナル・ハンドラーを使わないようにすると、SIGFPE が発生してもそれはプログラム側には渡されず、そのプログラムの実行が終了させられる。この設定を変えるには handle SIGFPE stop nopass とするとよい。

```
(gdb) handle SIGFPE stop nopass
Signal  Stop  Print  Pass to program Description
SIGFPE  Yes   Yes    No              Arithmetic exception
```

プラットフォームによっては、カーネルにあらかじめ命令しておかないと浮動小数点例外が発生しないことがある。そんなときは環境変数 GSL_IEEE_MODE を設定して関数 gsl_ieee_env_setup() を呼べばよい (第 40 章「IEEE 浮動小数点演算」参照)。

```
(gdb) set env GSL_IEEE_MODE=double-precision
```

A.4 数値計算プログラムで有用な gcc の警告オプション

C 言語では数値計算プログラムの信頼性を確保するためには、非常に注意深くプログラミングを行わねばならない。GCC の警告オプションのうち、以下のものがその役に立つだろう。

```
gcc -ansi -pedantic -Werror -Wall -W
  -Wmissing-prototypes -Wstrict-prototypes
  -Wtraditional -Wconversion -Wshadow
  -Wpointer-arith -Wcast-qual -Wcast-align
  -Wwrite-strings -Wnested-externs
  -fshort-enums -fno-common -Dinline= -g -O2
```

各オプションの詳細については、GCC のマニュアル “Using and Porting GCC” を参照のこと。以下に、これらのオプションにより表示される警告を簡単に説明する。

`-ansi -pedantic` ANSI C を使う。プログラム中で ANSI C に準拠していない拡張機能を使っていれば警告する。他のシステムでも使われるような移植性の高いプログラムを書くときに有用である。

`-Werror` 警告をエラーと同様に扱い、警告の発生時にコンパイルを中断する。警告が多いときに、その表示で画面がスクロールして見えなくなってしまうのを避けられる。警告が完全になくなるまで、コンパイルは完了できない。

`-Wall` 一般的なプログラミング上の問題についての警告を出す。`-Wall` は必要である。そしてこれだけでは十分ではない。

`-O2` 最適化を行う。`-Wall` も指定されていれば最適化ルーチンによるコード解析を利用して、初期化されていない変数があれば警告する。最適化をしないときにはこの警告は出ない。

`-W` 返り値がない、符号付きと符号なし整数の比較など、`-Wall` では出されない警告をいくつか出す。

`-Wmissing-prototypes -Wstrict-prototypes` プロトタイプ宣言がないか、正しくないときに警告する。プロトタイプ宣言がない場合は引数が正しいかどうかを判定するのは困難である。

`-Wtraditional` 旧い C と ANSI C とで振る舞いが違うような構文があれば警告する。他のコンパイラでは、プログラムからそれが旧い C 言語と ANSI C のどちらで書かれているかを判定するのは困難である。

`-Wconversion` たとえば `unsigned int x = -1` のような、符号付き整数から符号なし整数への変換が行われているときに警告する。このような変換をプログラム中で行う必要がある場合は、明示的に型のキャスティングを行うべきである。

`-Wshadow` 局所変数がほかの局所変数と同じ名前を持っているときに警告する。複数の変数に同じ名前が付いているときは、名前が衝突する原因になることがある。

`-Wpointer-arith -Wcast-qual -Wcast-align` `void` のような大きさを持たない型へのポインタの値を増減しようとしたり、ポインタからキャスティングで `const` を取り除こうとしたり、異なる大きさへの型へポインタをキャストしようとしたりしてメモリ上のアライメントが破壊される可能性があるような演算があるときに警告する。

`-Wwrite-strings` 文字列を `const` と見なして、それを上書きする処理があればコンパイル時に警告する。

`-fshort-enums` 可能なら `enum` を `short` 型として扱う。一般的にはこれにより `enum` は `int` と異なることになる。したがってポインタを整数として扱ったり `enum` として扱ったりしているような処理では、メモリのアライメントでエラーを出すことになる。

`-fno-common` 異なるオブジェクトファイル中で同じ名前の大域変数を定義しているときに (リンク時にエラーが出る) エラーを出す。そういった変数は、一つのファイル中でのみ定義し、他のファイルからは `extern` 宣言を使って参照されるべきである。

`-Wnested-externs` 関数内で `extern` 宣言が行われているときに警告する。

`-Dinline=` キーワード `inline` は ANSI C では定められていない。インライン関数を使うプログラムのコンパイル時に `-ansi` を使いたいときに、このプリプロセッサ定義を使うことで、プログラム中の `inline` キーワードを無視させることができる。

`-g` 生成される実行ファイルにデバッグ・シンボルを付加し、gdb でデバッグできるようにする。デバッグ・シンボルの付加による影響は実行ファイルのサイズの増加のみであり、必要に応じて `strip` コマンドでこれを取り除くことができる。

A.5 参考文献

数値計算プログラムを GCC や GDB で開発するに際して重要な本を挙げる。

- 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

後者は日本語に翻訳されて出版されている。

- リチャード・ストールマン, ローランド・ペシュ, GDB デバッギング入門, アスキー, ISBN 978-4756130167 (1999).

GNU C コンパイラとその他のプログラムについての入門には以下の本が参考になる。

- B. J. Gough, *An Introduction to GCC*, Network Theory Ltd, ISBN 0954161793

付 録B　GSLの開発にかかわった人々

(最新の情報については、配布パッケージ中の 'AUTHORS' ファイルを参照のこと)

Mark Galassi GSL の発案 (James Theiler と) とその設計の文書化。シミュレーティド・アニーリング・パッケージとマニュアルの相当部分。

James Theiler GSL の発案 (Mark Galassi と)。乱数生成とマニュアルの相当部分。

Jim Davies 統計ルーチンとマニュアルの相当部分。

Brian Gough FFT、数値積分、乱数発生器と確率分布、求根法、最適化とフィッティング、多項式の求根法、複素数、物理定数、置換、ベクトルと行列、ヒストグラム、確率統計、IEEE 関連、改訂版 CBLAS の Lebel 2 と 3、行列の分解、固有値問題、累積分布関数、検定、文書化、リリース管理。

Reid Priedhorsky ロス・アラモス国立研究所数理モデル解析グループでの、初期の求根法ルーチンの作成とその文書化。

Gerard Jungman 特殊関数、級数の収束の加速、ODE、BLAS、線形代数、固有値問題、ハンケル変換。

Mike Booth モンテカルロ積分。

Jorma Olavi Tähtinen 初期の複素数の演算関数。

Thomas Walter 初期のヒープソートとコレスキー分解。

Fabrice Rossi 多次元最小化。

Carlo Perassi クヌースの *Seminumerical Algorithms*, 3rd Ed. に基づいた乱数生成法の実装。

Szymon Jaroszewicz 組み合わせの生成ルーチン。

Nicolas Darnis 巡回置換と初期の正規置換のルーチン。

Jason H. Stover 主な累積分布関数。

Ivo Alxneit ウェーブレット変換ルーチン。

Tuomo Keskitalo ODE 実装の改良。

Lowell Johnson マチウ関数の実装。

Patrick Alken 非対称固有値問題の解法と B スプラインの実装。

このマニュアルの校訂は Nigel Lowry がしてくれた。感謝する。

非対称行列の固有値問題関連の関数は、線形代数ライブラリ LAPACK のコードを元にしている。LAPACK は以下のライセンスの元に公開されている。

付 録C　Autoconf のマクロ

(この章の説明はすべて、make の使い方の知識があることを前提としている。)
autoconf を使えば、AC_CHECK_LIB マクロを使って configure スクリプトで GSL を自動的にリンクするようにできる。GSL は CBLAS や他の数値計算ライブラリに依存するので、libgsl をリンクする前にそれらがインストールされていなければならないが、'configure.in' ファイルに以下の命令を書いておけば、それが確認できる。

```
AC_CHECK_LIB(m,main)
AC_CHECK_LIB(gslcblas,main)
AC_CHECK_LIB(gsl,main)
```

libgsl よりも前に libm と libgslcblas の確認をすることが重要で、順序を間違えると、正しい環境になっていてもエラーになる。これらのライブラリが確認されると、configure は以下のようなメッセージを出力していく。

```
checking for main in -lm... yes
checking for main in -lgslcblas... yes
checking for main in -lgsl... yes
```

ライブラリが見つかった場合、マクロ HAVE_LIBGSL、HAVE_LIBGSLCBLAS、HAVE_LIBM が定義され、変数 LIBS に-lgsl -lgslcblas -lm が追加される。
　以上の確認では、GSL のバージョンがなんであっても、インストールされていれば yes になる。バージョンがあまり重要ではない場合はこれでよいが、特定のバージョンを確認するためのマクロもファイル 'gsl.m4' に用意されている。これを使うには、上述の確認の行の代わりに以下の行を 'configure.in' に加えればよい。

```
AM_PATH_GSL(GSL_VERSION,
            [action-if-found],
            [action-if-not-found])
```

引数 GSL_VERSION は二つあるいは三つの数からなる、MAJOR.MINOR または MAJOR.MINOR.MICRO の形式のバージョン番号で、これで確認したいバージョンを指定する。action-if-not-found は以下のように指定するとよい。

```
AC_MSG_ERROR(could not find required version of GSL)
```

それから、正しいコンパイル・オプションを決めるために、Makefile.am 中で GSL_LIBS 変数および GSL_CFLAGS 変数を定義する。GSL_LIBS はコマンド gsl-config --libs の出力と同じで、

GSL_CFLAGS はコマンド gsl-config --cflags の出力と同じである。たとえば以下のような具合である。

```
libfoo_la_LDFLAGS = -lfoo $(GSL_LIBS) -lgslcblas
```

マクロ AM_PATH_GSL は C コンパイラを必要とするため、このマクロは‘configure.in’中で、C++ を使う AC_LANG_CPLUSPLUS マクロよりも前に置く必要がある。

inline が使えるかどうかを確認するには、‘configure.in’ ファイル中に以下のコードを書いておく。

```
AC_C_INLINE

if test "$ac_cv_c_inline" != no ; then
  AC_DEFINE(HAVE_INLINE,1)
  AC_SUBST(HAVE_INLINE)
fi
```

このマクロはコンパイル・フラグ内か、‘config.h’ 内で定義される。‘config.h’ は他のすべてのライブラリ・ヘッダファイルよりも前にインクルードされる。

autoconf では以下で extern inline が使えるかどうかを確認できる。

```
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"])
])

if test "$ac_cv_c_extern_inline" != no ; then
  AC_DEFINE(HAVE_INLINE,1)
  AC_SUBST(HAVE_INLINE)
fi
```

autoconf を使えば、ある関数を自動的に、別の移植性のある関数で置き換えることもできる。たとえば BSD 関数 hypot があるかどうかを確認するためには以下の行を ‘configure.in’ に書いておく。

```
AC_CHECK_FUNCS(hypot)
```

そして以下のマクロ定義を ‘config.h.in’ ファイルに書いておく

```
/* hypot がないシステムでは gsl_hypot を使う */

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

これにより hypot を使うプログラムのソースファイルでは、hypot がない場合に自動的に、インクルード文#include <config.h> を使って hypot を gsl_hypot に置き換えることになる。

付 録D　GSL CBLAS ライブラリ

低レベルの CBLAS 関数のプロトタイプ宣言が 'gsl_cblas.h' にある。これらの関数の定義は Netlib (12.3 節「BLAS の利用: 参考文献」参照) にある解説書を参照のこと。

D.1 Level 1

`float cblas_sdsdot` (*const int* `N`, *const float* `alpha`, *const float ** `x`, *const int* `incx`, *const float ** `y`, *const int* `incy`) [Function]
`double cblas_dsdot` (*const int* `N`, *const float ** `x`, *const int* `incx`, *const float ** `y`, *const int* `incy`) [Function]
`float cblas_sdot` (*const int* `N`, *const float ** `x`, *const int* `incx`, *const float ** `y`, *const int* `incy`) [Function]
`double cblas_ddot` (*const int* `N`, *const double ** `x`, *const int* `incx`, *const double ** `y`, *const int* `incy`) [Function]
`void cblas_cdotu_sub` (*const int* `N`, *const void ** `x`, *const int* `incx`, *const void ** `y`, *const int* `incy`, *void ** `dotu`) [Function]
`void cblas_cdotc_sub` (*const int* `N`, *const void ** `x`, *const int* `incx`, *const void ** `y`, *const int* `incy`, *void ** `dotc`) [Function]
`void cblas_zdotu_sub` (*const int* `N`, *const void ** `x`, *const int* `incx`, *const void ** `y`, *const int* `incy`, *void ** `dotu`) [Function]
`void cblas_zdotc_sub` (*const int* `N`, *const void ** `x`, *const int* `incx`, *const void ** `y`, *const int* `incy`, *void ** `dotc`) [Function]
`float cblas_snrm2` (*const int* `N`, *const float ** `x`, *const int* `incx`) [Function]
`float cblas_sasum` (*const int* `N`, *const float ** `x`, *const int* `incx`) [Function]
`double cblas_dnrm2` (*const int* `N`, *const double ** `x`, *const int* `incx`) [Function]
`double cblas_dasum` (*const int* `N`, *const double ** `x`, *const int* `incx`) [Function]
`float cblas_scnrm2` (*const int* `N`, *const void ** `x`, *const int* `incx`) [Function]
`float cblas_scasum` (*const int* `N`, *const void ** `x`, *const int* `incx`) [Function]
`double cblas_dznrm2` (*const int* `N`, *const void ** `x`, *const int* `incx`) [Function]
`double cblas_dzasum` (*const int* `N`, *const void ** `x`, *const int* `incx`) [Function]
`CBLAS_INDEX cblas_isamax` (*const int* `N`, *const float ** `x`, *const int* `incx`) [Function]
`CBLAS_INDEX cblas_idamax` (*const int* `N`, *const double ** `x`, *const int* `incx`)[Function]

CBLAS_INDEX cblas_icamax (*const int N*, *const void * x*, *const int incx*) [Function]
CBLAS_INDEX cblas_izamax (*const int N*, *const void * x*, *const int incx*) [Function]
void cblas_sswap (*const int N*, *float * x*, *const int incx*, *float * y*, *const int incy*) [Function]
void cblas_scopy (*const int N*, *const float * x*, *const int incx*, *float * y*, *const int incy*) [Function]
void cblas_saxpy (*const int N*, *const float alpha*, *const float * x*, *const int incx*, *float * y*, *const int incy*) [Function]
void cblas_dswap (*const int N*, *double * x*, *const int incx*, *double * y*, *const int incy*) [Function]
void cblas_dcopy (*const int N*, *const double * x*, *const int incx*, *double * y*, *const int incy*) [Function]
void cblas_daxpy (*const int N*, *const double alpha*, *const double * x*, *const int incx*, *double * y*, *const int incy*) [Function]
void cblas_cswap (*const int N*, *void * x*, *const int incx*, *void * y*, *const int incy*) [Function]
void cblas_ccopy (*const int N*, *const void * x*, *const int incx*, *void * y*, *const int incy*) [Function]
void cblas_caxpy (*const int N*, *const void * alpha*, *const void * x*, *const int incx*, *void * y*, *const int incy*) [Function]
void cblas_zswap (*const int N*, *void * x*, *const int incx*, *void * y*, *const int incy*) [Function]
void cblas_zcopy (*const int N*, *const void * x*, *const int incx*, *void * y*, *const int incy*) [Function]
void cblas_zaxpy (*const int N*, *const void * alpha*, *const void * x*, *const int incx*, *void * y*, *const int incy*) [Function]
void cblas_srotg (*float * a*, *float * b*, *float * c*, *float * s*) [Function]
void cblas_srotmg (*float * d1*, *float * d2*, *float * b1*, *const float b2*, *float * P*) [Function]
void cblas_srot (*const int N*, *float * x*, *const int incx*, *float * y*, *const int incy*, *const float c*, *const float s*) [Function]
void cblas_srotm (*const int N*, *float * x*, *const int incx*, *float * y*, *const int incy*, *const float * P*) [Function]
void cblas_drotg (*double * a*, *double * b*, *double * c*, *double * s*) [Function]
void cblas_drotmg (*double * d1*, *double * d2*, *double * b1*, *const double b2*, *double * P*) [Function]
void cblas_drot (*const int N*, *double * x*, *const int incx*, *double * y*, *const int incy*, *const double c*, *const double s*) [Function]

`void cblas_drotm` (*const int* `N`, *double ** `x`, *const int* `incx`, *double ** `y`, *const int* `incy`, *const double ** `P`) [Function]
`void cblas_sscal` (*const int* `N`, *const float* `alpha`, *float ** `x`, *const int* `incx`) [Function]
`void cblas_dscal` (*const int* `N`, *const double* `alpha`, *double ** `x`, *const int* `incx`) [Function]
`void cblas_cscal` (*const int* `N`, *const void ** `alpha`, *void ** `x`, *const int* `incx`) [Function]
`void cblas_zscal` (*const int* `N`, *const void ** `alpha`, *void ** `x`, *const int* `incx`) [Function]
`void cblas_csscal` (*const int* `N`, *const float* `alpha`, *void ** `x`, *const int* `incx`) [Function]
`void cblas_zdscal` (*const int* `N`, *const double* `alpha`, *void ** `x`, *const int* `incx`) [Function]

D.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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`)
[Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`)[Function]
`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`)
[Function]
`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`) [Function]
`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`)[Function]
`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`)
[Function]
`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`) [Function]
`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`)[Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`)[Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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*) [Function]
`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`) [Function]
`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*) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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*) [Function]
`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`) [Function]
`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*) [Function]
`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`) [Function]
`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`)[Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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*)

[Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`)[Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]
`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`) [Function]

D.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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`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`*) [Function]

`void cblas_xerbla` (*int `p`*, *const char * `rout`*, *const char * `form`*, *... *) [Function]

D.4 例

以下に Level-3 BLAS 関数の SGEMM を使う、二つの行列の積を計算するプログラムを示す。

$$\begin{pmatrix} 0.11 & 0.12 & 0.13 \\ 0.21 & 0.22 & 0.23 \end{pmatrix} \begin{pmatrix} 1011 & 1012 \\ 1021 & 1022 \\ 1031 & 1032 \end{pmatrix} = \begin{pmatrix} 367.76 & 368.12 \\ 647.06 & 647.72 \end{pmatrix}$$

二つの行列は行優先の順で格納されるが、cblas_sgemm 呼び出しの際に最初の引数を CblasColMajor にすることで、列優先にすることもできる。

```
#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,
                  1021, 1022,
                  1031, 1032 };
    int ldc = 2;
    float C[] = { 0.00, 0.00,
                  0.00, 0.00 };

    /* C = A B の計算 */
    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;
}
```

以下でこのプログラムをコンパイルできる。

```
$ gcc -Wall demo.c -lgslcblas
```

GSL の CBLAS は GSL 本体のライブラリとは独立しているため、この場合は -lgsl で GSL ライブラリ本体をリンクする必要はない。以下にこのプログラムの出力を示す。

```
$ ./a.out
```

```
[ 367.76, 368.12
  674.06, 674.72 ]
```

付 録E Free Software Needs Free Documentation

The following article was written by Richard Stallman, founder of the GNU Project.

The biggest deficiency in the free software community today is not in the software–it is the lack of good free documentation that we can include with the free software. Many of our most important programs do not come with free reference manuals and free introductory texts. Documentation is an essential part of any software package; when an important free software package does not come with a free manual and a free tutorial, that is a ma jor gap. We have many such gaps today.

Consider Perl, for instance. The tutorial manuals that people normally use are non-free. How did this come about? Because the authors of those manuals published them with restrictive terms–no copying, no modification, source files not available—which exclude them from the free software world.

That wasn't the first time this sort of thing happened, and it was far from the last. Many times we have heard a GNU user eagerly describe a manual that he is writing, his intended contribution to the community, only to learn that he had ruined everything by signing a publication contract to make it non-free.

Free documentation, like free software, is a matter of freedom, not price. The problem with the non-free manual is not that publishers charge a price for printed copies—that in itself is fine. (The Free Software Foundation sells printed copies of manuals, too.) The problem is the restrictions on the use of the manual. Free manuals are available in source code form, and give you permission to copy and modify. Non-free manuals do not allow this.

The criteria of freedom for a free manual are roughly the same as for free software. Redistribution (including the normal kinds of commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, both on-line and on paper. Permission for modification of the technical content is crucial too. When people modify the software, adding or changing features, if they are conscientious they will change the manual too–so they can provide accurate and clear documentation for the modified program. A manual that leaves you no choice but to write a new manual to document a changed version of the program is not really available to our community.

Some kinds of limits on the way modification is handled are acceptable. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they

were modified. Even entire sections that may not be deleted or changed are acceptable, as long as they deal with nontechnical topics (like this one). These kinds of restrictions are acceptable because they don't obstruct the community's normal use of the manual.

However, it must be possible to modify all the technical content of the manual, and then distribute the result in all the usual media, through all the usual channels. Otherwise, the restrictions obstruct the use of the manual, it is not free, and we need another manual to replace it.

Please spread the word about this issue. Our community continues to lose manuals to proprietary publishing. If we spread the word that free software needs free reference manuals and free tutorials, perhaps the next person who wants to contribute by writing documentation will realize, before it is too late, that only free manuals contribute to the free software community.

If you are writing documentation, please insist on publishing it under the GNU Free Documentation License or another free documentation license. Remember that this decision requires your approval–you don't have to let the publisher decide. Some commercial publishers will use a free license if you insist, but they will not propose the option; it is up to you to raise the issue and say firmly that this is what you want. If the publisher you are dealing with refuses, please try other publishers. If you're not sure whether a proposed license is free, write to `licensing@gnu.org`.

You can encourage commercial publishers to sell more free, copylefted manuals and tutorials by buying them, and particularly by buying copies from the publishers that paid for their writing or for ma jor improvements. Meanwhile, try to avoid buying non-free documentation at all. Check the distribution terms of a manual before you buy it, and insist that whoever seeks your business must respect your freedom. Check the history of the book, and try reward the publishers that have paid or pay the authors to work on it. The Free Software Foundation maintains a list of free documentation published by other publishers:

`http://www.fsf.org/doc/other-free-books.html`.

付 録F　GNU一般公衆利用許諾契約書

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

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.

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.

"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 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 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 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 "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 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 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 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.

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 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 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 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 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 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/.

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.

F.1 GPL を適用するには

あなたが何かプログラムを作成し、そのプログラムを最大限にいろんな人たちに使ってもらいたいと考えるなら、このライセンスの元であなたのプログラムの再配布、改変を許可し、フリーソフトウェアとするとよい。

そうするには、以下の文面をあなたのプログラムに付け加えればよい。あなたのプログラムには保証はないことをしっかりと伝えるには、各ソースファイルの先頭に付け加えるとよい。各ファイルには少なくとも 1 行の著作権表示 (Copyright) の行があり、正式な文面がどこにあるかを示すとよい。たとえば、以下のようにするといいだろう。

```
one line to give the program's name and a brief idea
of what it does.
Copyright (C) yyyy 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 2 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., 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA.
```

またあなたの郵便または E-mail の連絡先を示しておく。あなたのプログラムが対話的に動作するものなら、プログラムを起動したときに以下のような文面を表示するようにしておくのもよい。

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision 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.
```

もちろん、この表示の中の `show w` や `show c` は架空のものなので、ちゃんと該当の表示を行う適切なコマンドに書き換えなければならない。またコマンドではなく、メニューをマウスクリックするなどでもよい。プログラムの作成時に好きな方法を実装すればよい。

もし会社（あなたが会社員でプログラマーとして働いている場合）や学校などでは、著作権放棄の文面が必要なときもあるだろう。この他、GNU GPL を適用するにあたっての詳細は、`http://www.gnu.org/licenses` を参照されたい。

この GNU 一般公衆利用許諾契約 (General Public License) というライセンスは、あなたのプログラムを市販のプログラムの中で利用することを禁止している。しかしあなたがサブルーチン・ライブラリを作成した場合などは、市販のソフトウェアにリンクしたいと思うこともあるだろう。そういった場合はこのライセンスよりも GNU Lesser General Public License を使う方がよいだろう。いずれにしても最初に、`http://www.gnu.org/philosophy/why-not-lgpl.html` を読むべきである。

付 録G GNU Free Documentation License

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.

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.

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.

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 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 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), 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 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. 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 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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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.

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 © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 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.

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.

索 引