GNU Scientific Library

GNU 科学技術計算ライブラリ
リファレンス・マニュアル
第 1.8 版（GSL Version 1.8 対応版）
平成 18 年 11 月 14 日

原著

Mark Galassi — Los Alamos National Laboratory
Jim Davies — Department of Computer Science, Georgia Institute of Technology
James Theiler — Astrophysics and Radiation Measurements Group, Los Alamos National Laboratory
Brian Gough — Network Theory Limited
Gerard Jungman — Theoretical Astrophysics Group, Los Alamos National Laboratory
Michael Booth — Department of Physics and Astronomy, The Johns Hopkins University
Fabrice Rossi — University of Paris-Dauphine

日本語訳 Japanese Translation

とみながだいすけ TOMINAGA Daisuke

独立行政法人 産業技術総合研究所 生命情報科学研究センター
Computational Biology Research Center, National Institute of Advance Industrial Science and Technology

14 November, 2006

Copyright of Original English version:

日本語版のための著作権表示 Copyright of this Japanese Translation

訳者連絡先：`tominaga@cbrc.jp`
訳者紹介頁：`http://www.cbrc.jp/~tominaga/`

第 1 章 はじめに

GNU 科学技術計算ライブラリ (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 のために書かれており、新たに API (Applications Programming Interface) を提供し、高級言語でのプログラミングでラッパーを書くことができる。ソースコードは GNU 一般公衆利用許諾契約 (General Public License) の元で公開されている。

1.1 GSL で提供されているルーチン

このライブラリは、以下のような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	整列	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	乱数分布	疑似乱数列
ヒストグラム	統計	モンテカルロ積分	N 項組
微分方程式	シミュレーティド・アニーリング		数値微分
補間	級数展開	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	ウェーブレット
IEEE 浮動小数点表現	物理定数		

このマニュアルには、これらのルーチンの利用方法を記述している。各章では関数定義およびプログラム例に加え、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。

このライブラリの中には、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。これは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、再配布してもよいということである。このライブラリはパブリック・ドメインではない。このライブラリには著作権があり、公開、配布には条件が付いている。この条件は、共同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。このライブラリの利用者がしてはならないことは、他の人がその利用者からこのライブラリを入手し利用しようとすることを妨げることである。

特に、入手した GSL を使って作られたプログラムを共有する権利が誰にもあり、希望するもの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し、またはそ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。

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、そのプログラムを利用しようとする人に、プログラムの制作者が GSL から得た権利を与える必要がある。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、このライブラリが（proprietary な）商用のプログラムからは利用できないことを意味する。

またライブラリの制作者を守るため、GSL は無保証であるとする。このライブラリが誰かに改変されて配布された場合、それを入手した人には、それは元の GSL とは違うものであることを知らせ、

その改造によって生じた不都合がこのライブラリのオリジナルの制作者側とは関係ないことがわかるようにしたいと考えている。

GSL および関連するソフトウェアの正式な配布条件は GNU 一般公衆利用許諾契約 (付録 F を参照) 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、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/ まで)。

1.3 GSL の入手方法

このライブラリ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、cdrom 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。このライブラリがある公開 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/

このライブラリは GNU C コンパイラや GNU C ライブラリがある GNU システム環境ではコンパイラやライブラリの拡張機能が利用できるが、移植性を重視して作ってあるため、その他の多くのシステムでコンパイルできる。上記の web サイトでは、商用のコンパイル済みライブラリとサポート契約企業も紹介されている。

新しい版やアップデート、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。以下に e-mail を送ることでこのそれほど投稿の多くないメイリング・リストに登録することができる。

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 無保証

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。ルーチンの動作と精度は、利用者の責任においてソースコードを参照し、検証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 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

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’ ファイルにリストアップしてある。コンパイル時に生じる問題は ‘INSTALL’ に挙げてある。

ファイルにリストアップされていないバグを見つけたら、bug-gsl@gnu.org に連絡されたい。

バグ・レポートには以下の情報を含めてほしい。

- GSL の版（version number）
- ハードウェアと OS
- 使ったコンパイラの名前、版、コンパイル・オプション
- どのようなバグか、の説明

・バグを再現するための短いプログラム

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したかどうかで同じバグが発生するかどうかを確認、報告してもらえれば、非常に有用である。

このマニュアル中の間違いや欠落も、同じところに報告してもらいたい。

1.6 詳細について

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・リストの記録などについては、上述の web サイトを参照のこと。

このライブラリの利用、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。このメイリング・リストに参加するには、以下に e-mail を送る。

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

このメイリング・リストでは、このマニュアルについての質問や、ライブラリの開発者に対する連絡なども受け付けている。

もし論文誌などで記事を書くときにこの GSL についての参考文献を示すときは、たとえば M. Galassi et. al., GNU Scientific Library Reference Manual (2nd Ed.) ISBN 0954161734 を引用してもらいたい。

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

1.7 このマニュアルでの記述法

このマニュアルには、キーボード入力の例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一貫してドルマーク '$'とする。

例示されているプログラムは、GNU システム環境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。

第 2 章 このライブラリの使い方

この章では GSL を使ったプログラムのコンパイルの仕方と 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

以下に示す短いプログラムでは、ベッセル関数 $J_0(x)$ の x = 5 での値を計算することで、このライブラリの使用例を示す。

```
#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。

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

このプログラムをコンパイルするための手順を、以下の説に説明していく。

2.2 コンパイルとリンク

このライブラリ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>
```

このディレクトリが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 gcc でコンパイルする場合の普通のコンパイル・コマンドは、以下である。

```
$ 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' に探しにいくため、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 このライブラリをプログラムにリンクするには

このライブラリは、一つのファイル 'libgsl.a' としてインストールされる。共有ライブラリをサポートするシステムでは、共有ライブラリ版の 'libgsl.so' もインストールされる。デフォルトではこれらのファイルは '/usr/local/lib' に置かれる。利用者の環境で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、メインになるライブラリと、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。利用者のシステムで CBLAS が独自に用意されていない場合は、'libgslcblas.a' という名前でインストールされているだろう。以下に、このライブラリを使ったリンクの方法を例示をする。

```
$ 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。

2.2.2 BLAS とリンクするには

同じプログラムを、‘libcblas’ という他のライブラリとリンクするには、以下のようにする。

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

実行速度を上げるには、利用者のプラットフォームに最適化された CBLAS を -lcblas で使うようにするべきである。そのライブラリは標準の CBLAS と同じ内容でなければならない。移植性の高いの高性能 BLAS ライブラリに CBLAS インターフェイスをかぶせた ATLAS パッケージも利用できる。これもフリー・ソフトウェアで、ベクトルや行列を扱うところで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

このライブラリの共有ライブラリ版をリンクしたプログラムを実行するためには、実行時にその ‘.so’ ファイルの場所を指示しなければならない。シェルがそのライブラリを見つけられなければ、以下のようなエラーが出て実行されない。

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

シェルに実行させるには、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、以下のようにして設定することができる。

```
$ 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) 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。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、セッションを立ち上げるたびに入力する必要はなくなる。

静的にリンクされているプログラムとしてコンパイルする場合は、以下のように gcc に -static オプションをつける。

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

2.4 ANSI C への準拠

このライブラリは 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、どこでもこのライブラリを利用することができ

る。

利用者が実際に使用する GSL の API も ANSI にしたがっており、GSL を使っていても ANSI 準拠のプログラムを作成できる。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をライブラリと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り、その場合はこのライブラリでは、その不具合に相当する関数が使えないことになる。そういった関数は正しくない値を返すことがあるためで、そのようなプログラムはリンクできない方がよい。

大域的に利用できる関数名や変数名の名前空間での衝突をさけるため、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、このライブラリでもデフォルトではインライン関数は定義されていない。しかし実効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が使えるものがある。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 `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版の関数が使われる。

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、gcc により不要な関数定義が省かれるようになっている。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

拡張数値型 `long double` は ANSI C に導入されていて、新しいコンパイラはすべてこれに対応していると思われるが、`long double` の精度はプラットフォーム依存であることを使うときに意識しておかねばならない。IEEE 規格では、最低でも `double` 以上の精度を持つように規定されているだけである。

`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システムがいくつかある。このライブラリをコンパイルする時に、`configure` がそのテストを行って不定あるいは正しくない結果をさけ、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している。`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せねばならない。

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きでる。

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。

多くの環境では、これらの関数のネイティブ版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

このライブラリ中の一部の関数は、すべてのプラットフォームに対して最適化されているわけではない。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、その生成速度はプラットフォーム依存である。いくつかのこういった場合について、このライブラリでは同じインターフェイスを持つ代替関数を実装している。このライブラリの標準の関数を使ったプログラムを書いたとき、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、利用者が独自に実装、最適化した関数を使うこともできる。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
```

この命令は利用者のすべてのソース・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。プラットフォーム依存の関数は、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、乱数の場合はまったく異なる乱数系列を発生することがある。

2.9 異なる数値型のサポート

このライブラリの多くの関数は、様々な型に対してそれぞれの関数が用意されている。C++ テンプレートの基本形の型修飾を関数名につけることで、対応を実装している。修飾子は関数名の末尾に、モジュール名の後に続けてつけられている。仮のモジュール名 gsl_foo に関数名 fn の場合につけられる、型修飾子によ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 がデフォルトとされ、修飾子は付かない。たとえば関数 gsl_stats_mean は実数値の平均を計算するが、関数 gsl_stats_int_mean は整数の平均値を返す。

ライブラリで定義されている型、gsl_vector や gsl_matrix に対しても同様に名前が付けられている。この場合も修飾子は型の名前の後につけられる。たとえばモジュールで新しい構造体か typedef 演算子で gsl_foo が定義されている場合、修飾子は以下のようにつけられる。

```
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
```

モジュールが型に依存する定義を含んでいる場合、このライブラリでは、その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。ファイル名は以下に示すような付け方になっている。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。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

このライブラリ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。

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、このライブラリをコンパイルするときに CFLAGS に ‘-fexceptions’ を付けておく。

2.11 配列のエイリアス

このライブラリでは、配列、ベクトル、行列はエイリアスではなく、値をライブラリ側で変更して返せる引数として渡され、互いに重なることがないと想定している。こう前提をおくことで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、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

このライブラリはマルチ・スレッドのプログラムでも利用することができる。関数はすべてスレッド・セーフで、その意味ではどの関数も静的変数を持たない。メモリはすべてオブジェクトごとに確保され、関数ごとではない。一時的に利用するために workspace 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必要がある。読み出し専用メモリとして table オブジェクトを使う関数では、table は複数のスレッドで同時に使うことができる。table を引数として渡す場合は、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。

このライブラリ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の時に呼び出す関数など)。これらの変数の値は利用者が直接設定できるが、プログラムの起動時に一度だけ初期化するべきであり、他のスレッドで操作すべきではない。

2.13 古い deprecated な関数

時が経つにつれ、このライブラリの中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 deprecated であると決められ、このライブラリの次の版から削除される。現在の版に含まれる deprecated な関数は、GSL_DISABLE_DEPRECATED を定義してコンパイルすることで使えなくできる。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で、deprecated な関数が使われているかどうかを調べることができる。

第 3 章 エラー処理

この章では GSL の関数でのエラーを検知、管理するための方法について説明する。各関数が返すステータスを確認することで、利用者は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どうなったかを正確に知ることができる。利用者が独自のエラーハンドラー関数を定義して、このライブラリのデフォルトの動作に代えることもできる。

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

3.1 エラー報告

このライブラリは、posix スレッド・ライブラリの、スレッド・セーフなエラー報告法に準拠している。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) { /* エラー発生 */
    .....
    /* status の値が生じたエラーの種類を示している。 */
}
```

このライブラリのルーチンは、要求された処理ができなかった場合にもエラーを返す。たとえば求根法の関数は、要求精度で収束できなかったときや、繰り返し計算の回数が規定値に達したとき非零を返す。こういった状況は数値計算ライブラリではよくあることなので、利用者は関数を呼んだらその返り値を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。返り値は 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して、エラー処理の動作をする、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。

返り値によるエラー報告に加え、このライブラリ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、ライブラリ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、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 ハンドラーの目的は、デバッガーでの実行中にライブラリ内で生じるエラーを捕らえるためのブレイク・ポイントをその中に設定できるようにすることである。利用者か作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り、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作られているわけではない。

3.2 エラー・コード

ライブラリの関数が返すエラー・コードはファイル 'gsl_errno.h' で宣言されている。エラー・コードはすべて先頭に GSL_ がつけられ、非零の整数値に展開される。エラー・コードの多くは C ライブラリにある、対応するエラー・コードの名前を含んでいる。以下によく使われるエラー・コードを示す。

[Macro] int GSL_EDOM

領域エラー。数学関数に渡された引数が、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)。

[Macro] int GSL_ERANGE

範囲エラー。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な範囲にない場合に返す (C ライブラリの ERANGE に対応)。

[Macro] int GSL_ENOMEM

メモリ不足。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 では、ライブラリのルーチンが `malloc` でのメモリ確保に失敗したときに返す。

[Macro] int GSL_EINVAL

不正引数。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。

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

引数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。ライブラリ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、ライブラリの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・セーフな仕様である (GSL 開発者は、そういったプログラムは書かないように勧めている)。

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、利用者のプログラム側でルーチンの返りを確認してそれに対応した動作をせねばならない。利用者は、独自に作成したエラーハンドラーを使うこともできる。独自のハンドラーを使うことで、たとえばエラーのログをファイルに記録したり、アンダーフローのようなエラーを無視したり、デバッガーを使って、エラーを発生した実行中のプロセスにアタッチする、といったことができる。

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`' 内に宣言されている。

[Data Type] gsl_error_handler_t

これが GSL のエラーハンドラーの型である。エラーハンドラーに渡される引数は四つで、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 `__FILE__` と `__LINE__` ディレクティブを使ってプリプロセッサによって決められる。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)
```

利用者が独自に作成したエラーハンドラーを使いたい場合は、gsl_set_error_handler 関数を呼ぶ必要がある。これは ‘gsl_errno.h’ 内で宣言されている。

[Function] gsl_error_handler_t * gsl_set_error_handler (gsl_error_handler_t new_handler)

この関数で新しいエラーハンドラー new_handler を GSL ライブラリのルーチンに対して設定できる。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)。利用者が独自に作成したエラーハンドラーへのポインタは静的変数として格納されているため、そのプログラム内でのみ有効である。つまりその関数をマルチ・スレッドのプログラムで使う場合は、プログラム全体で利用できるようにマスター・スレッドで設定しなければならない。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) とするには、エラーハンドラーとして NULL を設定する。

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

[Function] gsl_error_handler_t * gsl_set_error_handler_off ()

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解除する。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、ライブラリのルーチンの返り値をすべて確認するべきである。最終的に完成したプログラムでは、この設定がよい。返り血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)。

ファイル ‘gsl_errno.h’ 中にある GSL_ERROR マクロを定義を変更してライブラリを再コンパイルすることで、プログラム中でのエラーハンドラーの挙動を変えることができる。

3.4 独自の関数で GSL のエラー報告を利用するには

プログラム中で GSL のコードを使って数値計算を行う関数を書いた場合、ライブラリ中で使っているエラー報告と同じことができると便利である。

エラーを報告するには、エラーを説明する文字列を引数として関数 gsl_error を呼び、それに続いて gsl_errno.h 中にある対応するエラー・コードを返すか、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。

[Macro] GSL_ERROR (reason, gsl errno)

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。

```
gsl_error (reason, __FILE__, __LINE__, gsl_errno);
return 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) ブロックなどでコードを囲む。

以下に、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の場合で、このマクロの使用例を示す。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);
}
```

[Macro] GSL_ERROR_VAL (reason, gsl errno, value)

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 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 の階乗の値しか受け付けない。変数 n が 2 の階乗でなかった場合、呼び出し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

この章では数学の基本的な関数について説明する。いくつかの関数はシステムのライブラリにもあるかもしれないが、そういったものがない場合にはこのライブラリで用意している関数が使える。

この章で説明する関数とマクロはヘッダファイル 'gsl_math.h' で宣言されている。

4.1 定数

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

[Macro] GSL_POSINF

IEEE での正の無限大 +∞。この値は +1.0/0.0 という式を評価することで得られる。

[Macro] GSL_NEGINF

IEEE での負の無限大 −∞。この値は -1.0/0.0 という式を評価することで得られる。

[Macro] GSL_NAN

IEEE での非数値を表す記号 NaN 。この値は 0.0/0.0 という式を評価することで得られる。

[Function] int gsl_isnan (const double x)

x が非数値 (NaN) であれば 1 を返す。

[Function] int gsl_isinf (const double x)

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。

[Function] int gsl_finite (const double x)

x が有効な実数であれば 1、無限大か非数値であれば 0 を返す。

4.3 基本的な関数

以下に説明するルーチンは、BSD の数学ライブラリにある関数に移植性を持たせた実装である。ネイティブ版の関数が使えないときは、その代わりにここに上げる関数を使うことができる。`autoconf` を使える環境では、利用者が書いた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)。

[Function] double gsl_log1p (const double x)

x が小さな値の時に、できるだけ正確に log(1 + x) の値を計算する。BSD の数学ライブラリにある `log1p(x)` と同等である。

[Function] double gsl_expm1 (const double x)

x が小さな値の時に、できるだけ正確に exp(x) − 1 の値を計算する。BSD の数学ライブラリにある `expm1(x)` と同等である。

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

オーバーフローが発生しないように √(x2 + y2) の値を計算する。BSD の数学ライブラリにある `hypot(x,y)` と同等である。

[Function] double gsl_acosh (const double x)

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

[Function] double gsl_asinh (const double x)

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

[Function] double gsl_atanh (const double x)

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

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

$x \times 2^e$の値を計算する。BSD の数学ライブラリにある `ldexp(x)` と同等である。

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

x を、$x = f \times 2^e$ かつ $0.5 \le f < 1$ となるような正規化された仮数部 f と指数部 e に分ける。返り値は f で、e は引数に入れて返される。x が零のときは f も e も零になる。BSD の数学ライブラリにある `frexp(x, e)` と同等である。

4.4 小さな整数でのべき乗

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用意されていない。GSL にはそのための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。

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

整数 n に対してべき乗 x^n を計算する。計算は効率よく行われる。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、計算誤差を見積もって返り値とする。

[Function] 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)

小さな整数に対する x^2 や x^3 などを高速に計算する。これらの関数は可能な場合はインライン展開されるため、同じ演算を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

[Macro] GSL_SIGN (x)

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、零の符号は正になる (IEEE の符号ビットを見ない)。

4.6 偶数、奇数の確認

[Macro] GSL_IS_ODD (n)

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。

[Macro] GSL_IS_EVEN (n)

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

[Macro] GSL_MAX (a, b)

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。

[Macro] GSL_MIN (a, b)

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。

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

倍精度実数 a と b のうち大きな方を返すインライン関数。この関数を使うと、引数の型を確認することができる。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。

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

倍精度実数 a と b のうち小さな方を返すインライン関数。この関数を使うと、引数の型を確認することができる。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。

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

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

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。この関数を使うと、引数の型を確認することができる。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 または GSL_MIN に置き換わる。

[Function] 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)

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。この関数を使うと、引数の型を確認することができる。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` 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 D.E. Knuth の本 Seminumerical Algorithms (3rd edition) の 4.2.2 節にある近似的比較法を実装した関数を用意している。

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

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。相対精度は 2δ とし、ここで $\delta = 2k$、k は `frexp()` で計算される x と y の底が 2 の指数部のうち大きな方とする。

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。

このライブラリでの実装はベルディング T.C. Belding による `fcmp` パッケージによる。

第 5 章 複素数

この章で説明する関数は複素数を扱うためのものである。複素数平面上でこれらの関数を評価するとき、できるだけアンダーフローやオーバーフローが発生する領域が小さくなるように実装されている。

多値の関数についてアブラモビッツ Abramowitz とステグン Stegun での慣例 (Handbook of Mathematical Functions にある)にしたがうため、分枝限定法を導入している。それらの関数は、GNU Calc と同じ主値を返す。つまり Common Lisp, The Language (Second Edition) および HP-28/48 シリーズの計算機とも同じである。

複素数に関する型宣言はヘッダファイル ‘gsl_complex.h’ にあり、複素関数と代数的演算子は ‘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、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。

参考として、gsl_complex 構造体の定義を示す。

```
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。実部と虚部、dat[0] および dat[1] の間のパディングは行われないため、複素数型の配列を packed にした場合も正しく配置される。

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

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。

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

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

[Macro] GSL_REAL (z)

[Macro] GSL_IMAG (z)

これらはマクロであり、複素数 z の実部あるいは虚部を返す。

[Macro] GSL SET_COMPLEX (zp, x, y)

これはマクロであり、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。たとえば、

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

とすると z は 3 + 4i になる。

[Macro] GSL_SET_REAL (zp, x)

[Macro] GSL_SET_IMAG (zp, y)

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

[Function] double gsl_complex_arg (gsl_complex z)

複素数 z の偏角 distinct(z) を返す。ただし $-\pi$ < distinct(z) ≤ π である。

[Function] double gsl_complex_abs (gsl_complex z)

複素数 z の大きさ（絶対値）|z| を返す。

[Function] double gsl_complex_abs2 (gsl_complex z)

複素数 z の大きさの二乗 $|z|^2$を返す。

[Function] double gsl_complex_logabs (gsl_complex z)

複素数 z の大きさ |z| の自然対数 log |z| を返す。|z| が 1 に近い値の時にも、精密な値を返すことができる。そういった場合、直接 `log(gsl_complex_abs(z))`とすると、精度が悪くなる。

5.3 複素代数的演算子

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

複素数 a と b の和、z = a + b を返す。

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

複素数 a と b の差、z = a − b を返す。

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

複素数 a と b の積、z = ab を返す。

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

複素数 a を b で割った商、z = a/b を返す。

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

複素数 a と実数 x の和、z = a + x を返す。

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

複素数 a と実数 x の差、z = a − x を返す。

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

複素数 a と実数 x の積、z = ax を返す。

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

複素数 a を実数 x で割った商、z = a/x を返す。

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

複素数 a と虚数 iy の和、z = a + iy を返す。

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

複素数 a と虚数 iy の差、z = a − iy を返す。

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

複素数 a と虚数 iy の積、z = a(iy) を返す。

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

複素数 a を虚数 y で割った商、z = a/(iy) を返す。

[Function] gsl_complex gsl_complex_conjugate (gsl_complex z)

複素数 z の共役複素数 z^* = x − iy を返す。

[Function] gsl_complex gsl_complex_inverse (gsl_complex z)

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

[Function] gsl_complex gsl_complex_negative (gsl_complex z)

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

5.4 基本的な複素関数

[Function] gsl_complex gsl_complex_sqrt (gsl_complex z)

複素数 z の平方根 √z を返す。分枝限定法で実部が負の領域は探索しない。結果は必ず複素平面の右側半分の領域にある。

[Function] gsl_complex gsl_complex_sqrt_real (double x)

実数 x の平方根を複素数で返す。x は負でもよい

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

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。

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

複素数 z の実数 x 乗 z^x を返す。

[Function] gsl_complex gsl_complex_exp (gsl_complex z)

複素数 z の指数 exp(z) を返す。

[Function] gsl_complex gsl_complex_log (gsl_complex z)

複素数 z の自然対数 log(z) を返す。実部が負の領域は分枝限定法により探索しない。

[Function] gsl_complex gsl_complex_log10 (gsl_complex z)

複素数 z の常用対数 (10 を底とする)log10(z) を返す。

[Function] gsl_complex gsl_complex_log_b (gsl_complex z, gsl_complex b)

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

5.5 複素三角関数

[Function] gsl_complex gsl_complex_sin (gsl_complex z)

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

[Function] gsl_complex gsl_complex_cos (gsl_complex z)

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

[Function] gsl_complex gsl_complex_tan (gsl_complex z)

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

[Function] gsl_complex gsl_complex_sec (gsl_complex z)

複素数 z の複素正割関数値 sec(z) = 1/cos(z) を返す。

[Function] gsl_complex gsl_complex_csc (gsl_complex z)

複素数 z の複素余割関数値 csc(z) = 1/sin(z) を返す。

[Function] gsl_complex gsl_complex_cot (gsl_complex z)

複素数 z の複素余接関数値 cot(z) = 1/tan(z) を返す。

5.6 複素逆三角関数

[Function] gsl_complex gsl_complex_arcsin (gsl_complex z)

複素数 z の複素逆正弦関数値 arcsin(z) を返す。実部が−1 よりも小さい、または実部が 1 よりも大きな領域は、分枝限定法により探索しない。

[Function] gsl_complex gsl_complex_arcsin_real (double z)

実数 z の複素逆正弦関数値 arcsin(z) を返す。z が−1 と 1 の間の時、$(-\pi, \pi]$ の範囲の値を返す。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。

[Function] gsl_complex gsl_complex_arccos (gsl_complex z)

複素数 z の複素逆余弦関数値 arccos(z) を返す。実部が−1 よりも小さい、または実部が 1 よりも大きな領域は、分枝限定法により探索しない。

[Function] gsl_complex gsl_complex_arccos_real (double z)

実数 z の複素逆余弦関数値 arccos(z) を返す。z が−1 と 1 の間の時、$[0, \pi]$ の範囲の値を返す。z が−1 よりも小さいときは、返り値の実部は $-\pi/2$ に、虚部は負の値になる。z が 1 よりも大きいときは、正の虚数になる。

[Function] gsl_complex gsl_complex_arctan (gsl_complex z)

複素数 z の複素逆正接関数値 arctan(z) を返す。分枝限定法により、虚数軸での値が−i 以下、および i 以上の領域は探索しない。

[Function] gsl_complex gsl_complex_arcsec (gsl_complex z)

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

[Function] gsl_complex gsl_complex_arcsec_real (double z)

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

[Function] gsl_complex gsl_complex_arccsc (gsl_complex z)

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

[Function] gsl_complex gsl_complex_arccsc_real (double z)

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

[Function] gsl_complex gsl_complex_arccot (gsl_complex z)

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

5.7 複素双曲線関数

[Function] gsl_complex gsl_complex_sinh (gsl_complex z)

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。

[Function] gsl_complex gsl_complex_cosh (gsl_complex z)

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

[Function] gsl_complex gsl_complex_tanh (gsl_complex z)

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

[Function] gsl_complex gsl_complex_sech (gsl_complex z)

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

[Function] gsl_complex gsl_complex_csch (gsl_complex z)

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

[Function] gsl_complex gsl_complex_coth (gsl_complex z)

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 逆双曲線関数

[Function] gsl_complex gsl_complex_arcsinh (gsl_complex z)

複素数 z の複素双曲線逆正弦関数値 arcsinh(z) を返す。分枝限定法により虚部が −i 以下と i 以上の領域は探索しない。

[Function] gsl_complex gsl_complex_arccosh (gsl_complex z)

複素数 z の複素双曲線逆余弦関数値 arccosh(z) を返す。分枝限定法により実部が 1 以下の領域は探索しない。

[Function] gsl_complex gsl_complex_arccosh_real (double z)

実数 z の複素双曲線逆余弦関数値 arccosh(z) を返す。

[Function] gsl_complex gsl_complex_arctanh (gsl_complex z)

複素数 z の複素双曲線逆正接関数値 arctanh(z) を返す。分枝限定法により実部が −1 以下および 1 以上の領域は探索しない。

[Function] gsl_complex gsl_complex_arctanh_real (double z)

実数 z の複素双曲線逆正接関数値 arctanh(z) を返す。

[Function] gsl_complex gsl_complex_arcsech (gsl_complex z)

複素数 z の複素双曲線逆正割関数値 arcsech(z) = arccosh(1/z) を返す。

[Function] gsl_complex gsl_complex_arccsch (gsl_complex z)

複素数 z の複素双曲線逆余割関数値 arccsch(z) = arcsin(1/z) を返す。

[Function] gsl_complex gsl_complex_arccoth (gsl_complex z)

複素数 z の複素双曲線逆余接関数値 arccoth(z) = 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, Volume 20 (1994), pp 215-244, Corrigenda, p553.

- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, "Implementing the complex arcsin and arccosine functions using exception handling", ACM Transactions on Mathematical Software, Volume 23 (1997) pp 299-335.

分枝限定法の一般的な説明は以下の本にある。

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

第 6 章 多項式

この章では多項式の評価、求根に関する関数について説明する。このライブラリでは、二次および三次の多項式を解析的に解いて実数あるいは複素数の解を得るルーチンが用意されている。実数係数の (任意次数の) 多項式についても、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

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

多項式 $c[0] + c[1]x + c[2]x^2 + \ldots + c[len - 1]x^{len-1}$ の値をホーナー法を使って求める。可能な場合はこの関数はインライン展開される。

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を扱う関数について説明する。差分商はアブラモヴィッツとステグンによる説明がある (Abramowitz & Stegun sections 25.1.4 節、25.2.26 節参照)。

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

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。

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

長さ size の配列 dd と xa に保持されている差分商表現の多項式の、点 x における値を計算して返す。

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

多項式の差分商表現をテイラー展開に変換する。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

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

以下の形式の二次方程式を解く。

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

返り値は実数解の個数 (零か 2 のどちらか) で、解の値は x0 と x1 に入れられる。実数解がない場合は x0 と x1 の値は変更されない。二つの実数解が得られた場合は x0 と x1 に昇順に入れられる。重根の場合も同様に扱われる。たとえば $(x - 1)^2 = 0$ の場合は解は 2 個あるが、その値は全く同じである。

解の個数は判別式 $b^2 - 4ac$ の符号によって決まる。倍精度で計算しても丸め誤差や桁落ちが問題になるが、多項式の係数が厳密でないときの表現誤差も問題になる。これらの誤差によって解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求められる。

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

gsl_complex *z1)

以下の形式の二次方程式の複素根を計算する。

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

返り値は複素根の個数（1 または 2）であり、求められた解は z0 と z1 に入れて返される。解はまず実部、次に虚部の値で比較され、昇順に引数に入れられる。重根の場合（たとえば a = 0 のときなど）はその解は z0 に入れられる。

6.4 三次方程式

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

以下の形式の三次方程式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。返り値は実数解の個数（1 か 3）であり、それらは x0、x1、x2 に入れて返される。得られた実数解の個数が一個の場合は x0 だけが書き換えられる。実数解が三個得られた場合は x0、x1、x2 に昇順に入れられる。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。

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

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 一般の多項式方程式

二次、三次方程式のような特殊な場合を除くと、一般的に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の解を近似的に求めるために繰り返し計算を使う方法について説明する。

[Function] gsl_poly_complex_workspace * gsl_poly_complex_workspace_alloc (size_t n)

`gsl_poly_complex_workspace` 構造体のインスタンスを、`gsl_poly_complex_solve` を使って n 個の係数を持つ多項式を解くための作業領域として確保する。

返り値は確保した `gsl_poly_complex_workspace` へのポインタであり、エラーが生じたときは null ポインタを返す。

[Function] void gsl_poly_complex_workspace_free (gsl_poly_complex_workspace * w)

作業領域 w のメモリを解放する。

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

随伴行列を使った調整型 QR 分解で多項式 $P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ の解を求める。引数 n で係数配列の長さを指定する。最高次の係数は零であってはならない。適切な大きさの作業領域 w をあらかじめ確保して、指定しなければならない。長さ 2(n − 1) の packed な複素数配列 z に、実部と虚部を交互にして得られた n − 1 個の解を入れて返す。

すべての解が見つかった場合は GSL_SUCCESS を、QR 分解が収束しなかった場合は GSL_EFAILED を返す。計算精度は限られているため、多重根の場合に、互いに値がわずかに異なった多数の低精度な根の集まりが得られることがある。高次の重根を持つ多項式の根を求めるには、それを考慮したアルゴリズムを使う必要がある（Z. Zeng, Algorithm 835, ACM Transactions on Mathematical Software, Volume 30, Issue 2 (2004), pp 218-236 参照)。

6.6 例

一般的な多項式の求根法の例として、以下に示す解を持つ方程式 $P(x) = x^5 - 1$ の例を示す。

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

以下のプログラムで、これらの根が見つかる。

```
#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

調整型 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 (1970), 219–231.
- B.N. Parlett and C. Reinsch, “Balancing a Matrix for Calculation of Eigenvalues and Eigenvectors”, Numerische Mathematik, 13 (1969), 293–304.
- A. Edelman and H. Murakami, “Polynomial roots from companion matrix eigenvalues”, Mathematics of Computation, Vol. 64 No. 210 (1995), 763–776.

差分商の公式化はアブラモヴィッツとステグンの以下の本にある。

- Abramowitz and Stegun, Handbook of Mathematical Functions, Sections 25.1.4 and 25.2.26.

第 7 章 特殊関数

この章では、GSL で用意している特殊関数について説明する。GSL では、エアリー Airy 関数、ベッセル関数、クラウゼン関数、クーロンの波動関数、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、エラー関数、指数積分、フェルミーディラックの関数、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、転位関数、三角関数とゼータ関数が用意されている。各ルーチンでは関数値計算の誤差の推定値も計算する。

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' をインクルードすることで、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、関数値を返す一般型とエラーコードを返すエラー型の二通りの方法がある。呼び出し方は違うが、関数値の計算法にはどちらも同じコードを使っている。

一般型では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、エラー型で関数を呼び出す。

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

エラー型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

エラー型の特殊関数は、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。

```
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計数を返すことがある。gsl_sf_result_e10 構造体には関数値と推定誤差の他に、関数値が result * 10^(e10) で得られるような指数係数も用意されている。

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

```
} gsl_sf_result_e10;
```

7.3 モード

このライブラリでは、可能な限り倍精度で関数値が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、関数値の精度を最低限必要な程度にまで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 $A_i(x)$、$B_i(x)$ は以下の積分として定義される。

$$
\begin{aligned}
Ai(x) &= \frac{1}{\pi}\int_0^\infty \cos\left(\frac{1}{3}t^3 + xt\right) dt \\
Bi(x) &= \frac{1}{\pi}\int_0^\infty e\ -t^3/3 + \sin\left(\frac{1}{3}t^3 + xt\right) dt
\end{aligned}
$$

詳細は Abramowitz & Stegun, 第 10.4 節を参照のこと。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

[Function] 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)

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

[Function] 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)

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

[Function] 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)

エアリー関数 $S_A(x)$Ai(x) の値を係数を分けて計算する。係数 $S_A(x)$ は x > 0 のとき exp(+(2/3)x3/2)、x < 0 のとき 1 になる。

[Function] 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)

エアリー関数 S_B (x)Bi(x) の値を係数を分けて計算する。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。

7.4.2 エアリー関数の導関数

[Function] 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)

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

[Function] 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)

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

[Function] 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)

係数を分けて計算したエアリー関数 $S_A(x)Ai'(x)$ の導関数の値を計算する。x > 0 のときは係数 $S_A(x)$ は $\exp(+(2/3)x^{3/2})$、x < 0 のときは 1 である。

[Function] 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)

係数を分けて計算したエアリー関数 $S_B(x)Bi'(x)$ の導関数の値を計算する。x > 0 のときは係数 $S_A(x)$ は $\exp(-(2/3)x^{3/2})$、x < 0 のときは 1 である。

7.4.3 エアリー関数の零点

[Function] double gsl_sf_airy_zero_Ai (unsigned int s)

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

エアリー関数 $A_i(x)$ の s 次の零点の座標を計算する。

[Function] double gsl_sf_airy_zero_Bi (unsigned int s)

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

エアリー関数 $B_i(x)$ の s 次の零点の座標を計算する。

7.4.4 エアリー関数の導関数の零点

[Function] 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)

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。

[Function] 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)

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

この節では、円柱ベッセル関数（第一種および第二種ベッセル関数）$J_n(x)$、$Y_n(x)$、修正円柱ベッセル関数（第一種および第二種の変形ベッセル関数） $I_n(x)$、$K_n(x)$、球面ベッセル関数 $i_l(x)$、$k_l(x)$ について説明する。詳細は Abramowitz & Stegun 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 正規円柱ベッセル関数

[Function] double gsl_sf_bessel_J0 (double x)

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

零次の正規円柱ベッセル関数 $J_0(x)$ の値を計算する。

[Function] double gsl_sf_bessel_J1 (double x)

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

一次の正規円柱ベッセル関数 $J_1(x)$ の値を計算する。

[Function] double gsl_sf_bessel_Jn (int n, double x)

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

n 次の正規円柱ベッセル関数 $J_n(x)$ の値を計算する。

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

nmin 次以上 nmax 次以下の正規円柱ベッセル関数 $J_n(x)$ 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 非正規円柱ベッセル関数

[Function] double gsl_sf_bessel_Y0 (double x)

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

x > 0 について、零次の非正規円柱ベッセル関数 $Y_0(x)$ の値を計算する。

[Function] double gsl_sf_bessel_Y1 (double x)

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

x > 0 について、一次の非正規円柱ベッセル関数 $Y_1(x)$ の値を計算する。

[Function] double gsl_sf_bessel_Yn (int n,double x)

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

x > 0 について、n 次の非正規円柱ベッセル関数 $Y_n(x)$ の値を計算する。

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

nmin 次以上 nmax 次以下の非正規円柱ベッセル関数 $Y_n(x)$ の値を計算し、引数で指定する配列 result_array に入れて返す。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 正規修正円柱ベッセル関数

[Function] double gsl_sf_bessel_I0 (double x)

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

零次の正規修正円柱ベッセル関数 $I_0(x)$ の値を計算する。

[Function] double gsl_sf_bessel_I1 (double x)

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

一次の正規修正円柱ベッセル関数 $I_1(x)$ の値を計算する。

[Function] double gsl_sf_bessel_In (int n, double x)

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

n 次の正規修正円柱ベッセル関数 $I_n(x)$ の値を計算する。

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

nmin 次以上 nmax 次以下の正規修正円柱ベッセル関数 $I_n(x)$ の値を計算し、引数で指定する配列 result array に入れて返す。範囲の下限 nmin は零か正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。

[Function] double gsl_sf_bessel_I0_scaled (double x)

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

零次の正規円柱ベッセル関数の値を、係数を分けて $\exp(-|x|)I_0(x)$ として計算する。

[Function] double gsl_sf_bessel_I1_scaled (double x)

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

一次の正規円柱ベッセル関数の値を、係数を分けて $\exp(-|x|)I_1(x)$ として計算する。

[Function] 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)

n 次の正規円柱ベッセル関数の値を、係数を分けて $\exp(-|x|)I_n(x)$ として計算する。

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

nmin 次以上 nmax 次以下の正規円柱ベッセル関数の値を係数を分けて $\exp(-|x|)I_n(x)$ として計算し、引数で指定する配列 result_array に入れて返す。範囲の下限 nmin は零か正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 非正規修正円柱ベッセル関数

[Function] double gsl_sf_bessel_K0 (double x)

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

$x > 0$ について、零次の非正規修正円柱ベッセル関数 $K_0(x)$ の値を計算する。

[Function] double gsl_sf_bessel_K1 (double x)

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

$x > 0$ について、一次の非正規修正円柱ベッセル関数 $K_1(x)$ の値を計算する。

[Function] double gsl_sf_bessel_Kn (int n, double x)

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

$x > 0$ について、n 次の非正規修正円柱ベッセル関数 $K_n(x)$ の値を計算する。

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

nmin 次以上 nmax 次以下の非正規修正円柱ベッセル関数 $K_n(x)$ の値を計算し、引数で指定する配列 result_array に入れて返す。範囲の下限 nmin は零か正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。

[Function] double gsl_sf_bessel_K0_scaled (double x)

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

零次の非正規修正円柱ベッセル関数の値を、係数を分けて $\exp(-|x|)K_0(x)$ として計算する。

[Function] double gsl_sf_bessel_K1_scaled (double x)

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

一次の非正規修正円柱ベッセル関数の値を、係数を分けて $\exp(-|x|)K_1(x)$ として計算する。

[Function] 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)

$x > 0$ について、n 次の非正規修正円柱ベッセル関数 $K_n(x)$ の値を係数を分けて $\exp(x)K_n(x)$ として計算する。

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

nmin 次以上 nmax 次以下の非正規修正円柱ベッセル関数 $K_n(x)$ の値を、係数を分けて $\exp(x)K_n(x)$ として計算し、引数で指定する配列 result_array に入れて返す。範囲の下限 nmin は零か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 正規球面ベッセル関数

[Function] double gsl_sf_bessel_j0 (double x)

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

零次の正規球面ベッセル関数 $j_0(x) = \sin(x)/x$ の値を計算する。

[Function] double gsl_sf_bessel_j1 (double x)

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

一次の正規球面ベッセル関数 $j_1(x) = (\sin(x)/x - \cos(x))/x$ の値を計算する。

[Function] double gsl_sf_bessel_j2 (double x)

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

二次の正規球面ベッセル関数 $j_2(x) = ((3/x^2 - 1)\sin(x) - 3\cos(x)/x)/x$ の値を計算する。

[Function] double gsl_sf_bessel_jl (int l, double x)

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

l 次の正規球面ベッセル関数 $j_l(x)$ の値を計算する。ここで $l \geq 0$ かつ $x \geq 0$ である。

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

l 次の正規球面ベッセル関数 $j_l(x)$ の値を、0 から lmax までの l の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。

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

スティード Steed の方法を使って l 次の正規球面ベッセル関数 $j_l(x)$ の値を、0 から lmax までの l の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。スティードとバーネット Barnett のアルゴリズムは Comp. Phys. Comm. 21, 297 (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、計算に時間がかかる。

7.5.6 非正規球面ベッセル関数

[Function] double gsl_sf_bessel_y0 (double x)

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

零次の非正規球面ベッセル関数 $y_0(x) = -\cos(x)/x$ の値を計算する。

[Function] double gsl_sf_bessel_y1 (double x)

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

一次の非正規球面ベッセル関数 $y_1(x) = -(\cos(x)/x + \sin(x))/x$ の値を計算する。

[Function] double gsl_sf_bessel_y2 (double x)

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

二次の非正規球面ベッセル関数 $y_2(x) = (-3/x^3 + 1/x)\cos(x) - (3/x^2)\sin(x)$ の値を計算する。

[Function] double gsl_sf_bessel_yl (int l, double x)

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

l 次の非正規球面ベッセル関数 $y_l(x)$ の値を計算する。ここで $l \geq 0$ である。

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

l 次の非正規球面ベッセル関数 $y_l(x)$ の値を、0 から lmax までの l の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 正規修正球面ベッセル関数

非正規修正球面ベッセル関数 $i_l(x)$は、非整数次の非正規修正ベッセル関数 $i_l(x) = \sqrt{(\pi/(2x))}\, i_{l+1/2}(x)$に関するものである。

[Function] double gsl_sf_bessel_i0_scaled (double x)

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

零次の正規修正球面ベッセル関数 $\exp(-|x|)i_0(x)$ の値を、係数を分けて計算する。

[Function] double gsl_sf_bessel_i1_scaled (double x)

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

一次の正規修正球面ベッセル関数 $\exp(-|x|)i_1(x)$ の値を、係数を分けて計算する。

[Function] double gsl_sf_bessel_i2_scaled (double x)

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

二次の正規修正球面ベッセル関数 $\exp(-|x|)i_2(x)$ の値を、係数を分けて計算する。

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

l 次の正規修正球面ベッセル関数 $\exp(-|x|)i_l(x)$ の値を、0 から lmax までの l の値について、係数を分けて計算する。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 非正規修正球面ベッセル関数

非正規修正球面ベッセル関数 $k_l(x)$は、非整数次の非正規修正ベッセル関数 $k_l(x) = \sqrt{(\pi/(2x))} K_{l+1/2}(x)$に関するものである。

[Function] double gsl_sf_bessel_k0_scaled (double x)

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

零次の非正規修正球面ベッセル関数 $\exp(x)k_0(x)$ の値を、係数を分けて計算する。x > 0 である。

[Function] double gsl_sf_bessel_k1_scaled (double x)

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

一次の非正規修正球面ベッセル関数 $\exp(x)k_1(x)$ の値を、係数を分けて計算する。x > 0 である。

[Function] double gsl_sf_bessel_k2_scaled (double x)

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

二次の非正規修正球面ベッセル関数 $\exp(x)k_2(x)$ の値を、係数を分けて計算する。x > 0 である。

[Function] 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)

l 次の非正規修正球面ベッセル関数 $\exp(x)k_l(x)$ の値を、係数を分けて計算する。x > 0 である。

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

l 次の非正規修正球面ベッセル関数 $\exp(x)k_l(x)$ の値を、0 から lmax までの l の値について係数を分けて x > 0 で計算する。ここで lmax ≥ 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 正規ベッセル関数 - 非整数

[Function] double gsl_sf_bessel_Jnu (double nu, double x)

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

非整数 nu 次の正規円柱ベッセル関数 $J_\nu(x)$ の値を計算する。

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

数列をなす x のそれぞれの値について、非整数 nu 次の正規円柱ベッセル関数 $J_\nu(x)$ の値を計算する。長さ size の配列 v に x の値を入れて渡す。x の値はすべて正で整列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 非正規ベッセル関数 - 非整数

[Function] double gsl_sf_bessel_Ynu (double nu, double x)

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

非整数 nu 次の非正規円柱ベッセル関数 $Y_\nu(x)$ の値を計算する。

7.5.11 正規修正ベッセル関数 - 非整数

[Function] double gsl_sf_bessel_Inu (double nu, double x)

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

非整数 nu 次の正規修正ベッセル関数 $I_\nu(x)$ の値を計算する。x > 0 かつ $\nu > 0$ である。

[Function] 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)

非整数 nu 次の正規修正ベッセル関数の値を、係数を分けて $\exp(-|x|)I_\nu(x)$ として計算する。x > 0 かつ $\nu > 0$ である。

7.5.12 非正規修正ベッセル関数 - 非整数

[Function] double gsl_sf_bessel_Knu (double nu, double x)

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

非整数 nu 次の非正規修正ベッセル関数 $K_\nu(x)$ の値を計算する。x > 0 かつ $\nu > 0$ である。

[Function] double gsl_sf_bessel_lnKnu (double nu, double x)

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

非整数 nu 次の非正規修正ベッセル関数の対数の値 $\ln(K_\nu(x))$ を計算する。x > 0 かつ $\nu > 0$ である。

[Function] 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)

非整数 nu 次の非正規修正ベッセル関数の値を、係数を分けて $\exp(+|x|)K_\nu(x)$ として計算する。x > 0 かつ $\nu > 0$ である。

7.5.13 正規ベッセル関数の零点

[Function] double gsl_sf_bessel_zero_J0 (unsigned int s)

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

ベッセル関数 $J_0(x)$ の s 番目の正の零点を求める。

[Function] double gsl_sf_bessel_zero_J1 (unsigned int s)

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

ベッセル関数 $J_1(x)$ の s 番目の正の零点を求める。

[Function] double gsl_sf_bessel_zero_Jnu (double nu, unsigned int s)

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

ベッセル関数 $J_\nu(x)$ の s 番目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

クラウゼン関数は、以下の積分として定義される。

$$Cl_2(x) \quad = \quad -\int_0^x dt \log\left(2\sin(t/2)\right)$$

これは $\mathrm{Cl}_2(\theta) = \mathrm{Im}(\mathrm{Li}_2(e^{i\theta}))$ に関するものである。クラウゼン関数は 'gsl_sf_clausen.h'

で宣言されている。

[Function] double gsl_sf_clausen (double x)

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

クラウゼン積分 $Cl_2(x)$ を計算する。

7.7 クーロン関数

クーロン関数はヘッダイファイル 'gsl_sf_coulomb.h' で宣言されている。境界状態 bound state と分散解 scattering solution の両方が利用できる。

7.7.1 水素の正規化境界状態

[Function] double gsl_sf_hydrogenicR_1 (double Z, double r)

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

水素ラジカルのもっとも次数の低い正規化境界状態波動関数 $R_1 := 2Z\sqrt{(Z)}\exp(-Zr)$ を計算する。

[Function] 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)

以下で表される、ラジカルの n 次の正規化境界状態波動関数を計算する。

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

ここで $L_b^a(x)$は一般化ラゲール多項式である（ラゲール関数の節参照)。正規化は、たとえば波動関数 ψ を $\psi(n, l, r) = R_n Y_{lm}$ とすることで行われる。

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 Abramowitz & Stegun 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、オーバーフローを丁寧に検知する必要がある。オーバーフローが生じたときは、シグナル GSL_EOVRFLW を通知して、引数 exp_F と exp_G に計算結果の値の指数部を入れて返す。計算された関数値は、以下のようにして得られる。

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

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

[Function] 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)

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、係数として分けられた指数部が引数 exp_F と exp_G に入れて返される。

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

L = Lmin . . . Lmin + kmax に対して関数値 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部が F_exponent に入れて返される。

[Function] 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)

L = Lmin . . . Lmin + kmax に対して関数値 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部が F_exponent と G_exponent に入れて返される。

[Function] 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)

L = Lmin . . . Lmin +kmax に対して関数値 F_L(eta, x) と G_L(eta, x)、その導関数値 F_L'(eta, x) と G_L'(eta, x) を計算し、関数値を fc_array と gc_array に、導関数値を fcp_array と gcp_array に入れて返す。オーバーフローが生じたときは指数部が F_exponent と G_exponent に入れて返される。

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

L = Lmin . . . Lmin + kmax に対してクーロン関数を引数で除した値 F_L(eta, x)/x を計算し、fc_array に入れて返す。オーバーフローが生じたときは指数部が F_exponent に入れて返される。この関数は、極限 $\eta \to 0$ で球面ベッセル関数になる。

7.7.3 クーロンの波動関数の正規化定数

クーロンの波動関数の正規化定数は Abramowitz 14.1.7 に定義されている。

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

クーロンの波動関数の正規化定数 C_L(eta) を計算する。L > −1 である。

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

クーロンの波動関数の正規化定数 C_L(eta) を L = Lmin . . . Lmin + kmax について計算する。Lmin > −1 である。

7.8 結合係数

ウィグナーの 3-j、6-j、9-j 記号は、角運動量ベクトルの組み合わせを表現する結合係数として用いられる。標準的な結合係数ではその引数は整数か整数の 1/2 であるため、以下に説明する関数では慣例に従い、引数の型を整数として実際のスピンの二倍の値を与えるものとする。3-j 記号については Abramowitz & Stegun の 27.9 節を参照のこと。この節で説明する関数はヘッダイファイル ‘gsl_sf_coupling.h’ で宣言されている。

7.8.1 3-j 記号

[Function] 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)

以下に示すウィグナーの 3-j 係数を計算する。

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

引数は整数の 1/2 を単位として、ja = two_ja/2、ma = two_ma/2 のように与えられる。

7.8.2 6-j 記号

[Function] 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)

以下に示すウィグナーの 6-j 係数を計算する。

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

引数は整数の 1/2 を単位として、ja = two_ja/2、ma = two_ma/2 のように与えられる。

7.8.3 9-j 記号

[Function] 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)

以下に示すウィグナーの 9-j 係数を計算する。

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

引数は整数の 1/2 を単位として、ja = two_ja/2、ma = two_ma/2 のように与えられる。

7.9 ドーソン関数

ドーソンの積分は $\exp(-x^2)\int_0^x dt \exp(t^2)$ として定義される。その値が Abramowitz & Stegun の表 7.5 に挙げられている。ドーソン関数はヘッダイファイル ‘`gsl_sf_dawson.h`’ で宣言されている。

[Function] double gsl_sf_dawson (double x)

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

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

デバイの関数は積分は以下として定義される。

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

詳細は Abramowitz & Stegun の 27.1 節を参照のこと。デバイの関数は ‘`gsl_sf_debye.h`’

で宣言されている。

[Function] double gsl_sf_debye_1 (double x)

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

一次のデバイ関数 $D_1(x) = (1/x)\int_0^x dt(t/(e^t-1))$ を計算する。

[Function] double gsl_sf_debye_2 (double x)

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

二次のデバイ関数 $D_2(x) = (2/x^2)\int_0^x dt(t^2/(e^t-1))$ を計算する。

[Function] double gsl_sf_debye_3 (double x)

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

三次のデバイ関数 $D_3(x) = (2/x^3)\int_0^x dt(t^3/(e^t-1))$ を計算する。

[Function] double gsl_sf_debye_4 (double x)

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

四次のデバイ関数 $D_4(x) = (2/x^4)\int_0^x dt(t^4/(e^t-1))$を計算する。

[Function] double gsl_sf_debye_5 (double x)

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

五次のデバイ関数 $D_5(x) = (2/x^5)\int_0^x dt(t^5/(e^t-1))$を計算する。

[Function] double gsl_sf_debye_6 (double x)

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

六次のデバイ関数 $D_6(x) = (2/x^6)\int_0^x dt(t^6/(e^t-1))$を計算する。

7.11 二重対数

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

7.11.1 実数引数

[Function] double gsl_sf_dilog (double x)

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

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 Lewin の記法では $Li_2(x)$ と表され、x の二重対数のの実部である。これは積分 $Li_2(x) = -Re\int_0^x ds\ \log(1-s)/s$ として定義される。$x \le 1$ のとき $Im(Li_2(x)) = 0$、$x > 1$ のとき $-\pi \log(x)$ である。

7.11.2 複素数引数

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

複素数の引数に対し二重対数 $z = r \exp(i\theta)$ 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 基礎的な演算

以下の関数を使うことで、値を掛け合わせることで誤差がどう大きくなる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。

[Function] int gsl_sf_multiply_e (double x, double y, gsl_sf_result * result)

x と y の値の積を計算し、積と誤差を result に入れて返す。

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

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。積 xy ± xy √((dx/x)² + (dy/y)²) は result に入れて返される。

7.13 楕円積分

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

7.13.1 ルジャンドル形式の定義

楕円積分のルジャンドル形式 F(φ, k)、E(φ, k)、P(φ, k, n) は以下で定義される。

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

完全なルジャンドル形式は K(k) = F (π/2, k) および E(k) = E(π/2, k) と表される。楕円積分のルジャンドル形式についての詳細は Abramowitz & Stegun の第 17 章を参照のこと。ここで用いる記述法は Carlson の Numerische Mathematik 、第 33 巻 (1979) に従っており Abramowitz & Stegun とは異なる部分もある。

7.13.2 カールソン形式の定義

楕円積分のカールソンの対称形式 RC (x, y)、RD(x, y, z)、RF (x, y, z) 、RJ (x, y, z , p) は以下で定義される。

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

7.13.3 完全楕円積分のルジャンドル形式

[Function] 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)

完全楕円積分 K(k) を mode で指定された精度で計算する。

[Function] 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)

完全楕円積分 E(k) を mode で指定された精度で計算する。

7.13.4 不完全楕円積分のルジャンドル形式

[Function] 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)

不完全楕円積分 F(ϕ, k) を mode で指定された精度で計算する。

[Function] 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)

不完全楕円積分 E(ϕ, k) を mode で指定された精度で計算する。

[Function] 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)

不完全楕円積分 P(ϕ, k, n) を mode で指定された精度で計算する。

[Function] 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)

カールソン形式 RD(x, y, z) を使って以下の関係で定義される不完全楕円積分 D(ϕ, k, n) を計算する。

$$D(\phi, k, n) \quad = \quad RD(1 - \sin^2(\phi), 1 - k^2 \sin^2(\phi), 1)$$

7.13.5 カールソン形式

[Function] double gsl_sf_ellin_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)

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。

[Function] 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)

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。

[Function] 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)

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。

[Function] 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)

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 (ヤコビ法)

ヤコビの楕円関数は Abramowitz & Stegun の第 16 章で定義されている。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。

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

ヤコビの楕円関数 sn(u|m)、cn(u|m)、dn(u|m) を下降ランデン変換を使って計算する。

7.15 誤差関数

誤差関数は Abramowitz & Stegun の第 7 章に説明がある。この節で説明する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

[Function] double gsl_sf_erf (double x)

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

誤差関数 erf(x) = $(2/\sqrt{\pi})\int_x^\infty \exp(-t^2)$ を計算する。

7.15.2 相補数誤差関数

[Function] double gsl_sf_erfc (double x)

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

相補誤差関数 efrc(x) = 1 - erf(x) = $(2/\sqrt{\pi})\int_x^\infty \exp(-t^2)$ を計算する。

7.15.3 対数相補誤差関数

[Function] double gsl_sf_log_erfc (double x)

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

相補誤差関数の対数値 log(erfc(x)) を計算する。

7.15.4 確率関数

正規あるいはガウス分布の確率関数は Abramowitz & Stegun の第 26.2 節に説明されている。

[Function] double gsl_sf_erf_Z (double x)

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

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

[Function] double gsl_sf_erf_Q (double x)

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

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

正規分布の危険関数 hazard function はミル比の逆数であり、以下で定義される。

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

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) ~ x に漸近する。

[Function] double gsl_sf_hazard (double x)

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

正規分布の危険関数を計算する。

7.16 指数関数

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

7.16.1 指数関数

[Function] double gsl_sf_exp (double x)

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

一般的な普通の方法および誤差見積もりを行う方法で指数関数 exp(x) の値を計算する。

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

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) が非常に大きくなり double の表現範囲を超えてオーバーフローする場合に有用である。

[Function] double gsl_sf_exp_mult (double x, double y)

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

x の指数と y との積を計算し、y exp(x) を返す。

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

gsl_sf_result_e10 型を使ってより広い範囲で積 y exp(x) を計算して返す。

7.16.2 相対指数関数

[Function] double gsl_sf_expm1 (double x)

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

x の値が小さいときに精度がよくなる計算法を使って exp(x) − 1 の値を計算する。

[Function] double gsl_sf_exprel (double x)

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

x の値が小さいときに精度がよくなる計算法を使って (exp(x) − 1)/x の値を計算する。これは $(\exp(x) - 1)/x = 1 + x/2 + x^2/(2\times3) + x^3/(2\times3\times4) + \ldots$ という展開を利用している。

[Function] double gsl_sf_exprel_2 (double x)

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

x の値が小さいときに精度がよくなる計算法を使って $2(\exp(x) - 1 - x)/x^2$ の値を計算する。これは $2(\exp(x) - 1 - x)/x^2 = 1 + x/3 + x^2/(3\times4) + x^3/(3\times4\times5) + \ldots$ という展開を利用している。

[Function] double gsl_sf_exprel_n (int n, double x)

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

gsl_sf_exprel や gsl_sf_exprel2 を一般の n 次に拡張し、N 次の相対指数関数値を計算する。N 次の相対指数関数は以下で与えられる。

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

7.16.3 誤差推定を行う指数計算

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

x の指数と推定絶対誤差 dx を計算する。

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

x とその推定絶対誤差 dx から `gsl_sf_result_e10` 型を使ってより広い範囲で x の指数を計算して返す。

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

x 、y とその推定絶対誤差 dx 、dy から x の指数と y の積 y exp(x) を計算する。

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

x 、y とその推定絶対誤差 dx 、dy から `gsl_sf_result_e10` 型を使ってより広い範囲で x の指数と y の積 y exp(x) を計算する。

7.17 指数積分

指数積分についての詳細は Abramowitz & Stegun の第 5 章に説明されている。この節で説明する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

[Function] double gsl_sf_expint_E1 (double x)

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

以下に示す指数積分 $E_1(x)$ を計算する。

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

[Function] double gsl_sf_expint_E2 (double x)

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

以下に示す二次の指数積分 $E_2(x)$ を計算する。

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

7.17.2 Ei(x)

[Function] double gsl_sf_expint_Ei (double x)

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

以下に示す指数積分 $E_{i(}(x)$ を計算する。

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

ここで PV はこの積分の主値である。

7.17.3 双曲線積分

[Function] double gsl_sf_Shi (double x)

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

積分 Shi(x) = $\int_0^x$dt sinh(t)/t を計算する。

[Function] double gsl_sf_Chi (double x)

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

積分 Chi(x) :=Re[γ_E+log(x)+$\int_0^x$dt(cosh[t]-1)/t] 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 $Ei_3(x)$

[Function] double gsl_sf_expint_3 (double x)

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

指数積分 $Ei_3(x) = \int_0^x dt\, \exp(-t^3)$ を計算する。$x \geq 0$ である。

7.17.5 三角関数の積分

[Function] double gsl_sf_Si (const double x)

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

正弦関数の積分 Si(x) = $\int_0^x$dt sin(t)/t を計算する。

[Function] double gsl_sf_Ci (const double x)

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

余弦関数の積分 Ci(x) = -$\int_x^\infty$dt cos(t)/t を計算する。$x > 0$ である。

7.17.6 逆接弦関数の積分

[Function] double gsl_sf_atanint (double x)

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

逆接弦関数の積分 AtanInt(x) = $\int_0^x$dt arctan(t)/t を計算する。

7.18 フェルミ - ディラックの関数

この節で説明する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 $F_j(x)$ は以下で与えられる。

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

[Function] double gsl_sf_fermi_dirac_m1 (double x)

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

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

[Function] double gsl_sf_fermi_dirac_0 (double x)

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

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

[Function] double gsl_sf_fermi_dirac_1 (double x)

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

添え字が 1 の完全フェルミ - ディラック積分 $F_1(x) = \int_0^\infty dt(t/(\exp(t-x)+1))$ を計算する。

[Function] double gsl_sf_fermi_dirac_2 (double x)

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

添え字が 2 の完全フェルミ - ディラック積分 $F_2(x) = (1/2)\int_0^\infty dt(t^2/(\exp(t-x)+1))$ を計 算する。

[Function] 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)

添え字が j の完全フェルミ - ディラック積分 $F_J(x) = (1/\Gamma(J+1))\int_0^\infty dt(t^J/(\exp(t-x)+1))$ を計算する。

[Function] double gsl_sf_fermi_dirac_mhalf (double x)

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

完全フェルミ - ディラック積分 $F_{-1/2}(x)$ を計算する。

[Function] double gsl_sf_fermi_dirac_half (double x)

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

完全フェルミ - ディラック積分 $F_{1/2}(x)$ を計算する。

[Function] double gsl_sf_fermi_dirac_3half (double x)

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

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 $F_j(x, b)$ は以下で与えられる。

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

[Function] 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)

零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

ガンマ関数は以下の積分で定義される。

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

7.19.1 ガンマ関数

n が正の整数のとき、Γ 関数と n の階乗には Γ(n) = n! という関係がある。ガンマ関数についての詳細は Abramowitz & Stegun の第 6 章を参照のこと。この節の関数は 'gsl_sf_gamma.h' で宣言されている。

[Function] double gsl_sf_gamma (double x)

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

負の整数でない x についてガンマ関数値 Γ(x) を計算する。この関数では実数ランチョス Lanczos 法を使う。Γ(x) をオーバーフローさせずにすむ x の最大値はマクロ GSL_SF_GAMMA_XMAX で参照でき、171.0 である。

[Function] double gsl_sf_lngamma (double x)

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

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

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

負の整数でない x についてガンマ関数の符号とその対数を計算する。実数ランチョス法を使う。返り値から Γ(x) = sgn * exp(resultlg) として Γ 関数の値が得られる。

[Function] double gsl_sf_gammastar (double x)

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

正則なガンマ関数の値 $\Gamma^*(x)$ を x > 0 で返す。正則なガンマ関数は以下で与えられる。

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

またテム (Nico Temme) の文献に有用な情報がある。

[Function] double gsl_sf_gammainv (double x)

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

ガンマ関数の逆数 1/Γ(x) をランチョス法を使って計算する。

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

負の整数でない複素数 z に対して、複素数 $z = z_r + iz_i$ のガンマ関数値の対数 log(Γ(z)) をランチョス法で計算する。計算結果 lnr = log|Γ(z)|、arg = arg(Γ(z)) の取りうる値は (–π, π] で、引数に入れて返される。|z| が非常に大きな場合は、範囲を (–π, π] 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。しかし絶対値 (lnr) の精度が失われることはない。

7.19.2 階乗

階乗の値は、非負整数 n に対しては n! = Γ(n+1)ガンマ関数を使って計算できるが、あらかじめ表をライブラリ中に持っておき、特に n が小さな値の時に高速な関数について以下に説明する。

[Function] double gsl_sf_fact (unsigned int n)

[Function] int gsl_sf_fact_e (unsigned int n, gsl sf result * result)

階乗 n! を計算する。その値は n! = Γ(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。

[Function] double gsl_sf_doublefact (unsigned int n)

[Function] int gsl_sf_doublefact_e (unsigned int n, gsl sf result * result)

n!! = n(n - 2)(n - 4)... 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。

[Function] double gsl_sf_lnfact (unsigned int n)

[Function] int gsl_sf_lnfact_e (unsigned int n, gsl sf result * result)

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(Γ(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。

[Function] double gsl_sf_lndoublefact (unsigned int n)

[Function] int gsl_sf_lndoublefact_e (unsigned int n, gsl sf result * result)

n!! の対数を計算する。

[Function] 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)

組み合わせの数 n choose m = n!(m!(n - m)!) を計算する。

[Function] 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)

組み合わせの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。

[Function] double gsl_sf_taylorcoeff (int n, double x)

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

テイラー係数 xn /n! を x ≥ 0, n ≥ 0 について計算する。

7.19.3 ポッホハンマーの記号

[Function] double gsl_sf_poch (double a, double x)

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

負の整数でない a と a + x に対して、ポッホハンマー Pochhammer の記号 (a)x := Γ(a + x)/Γ(a) の値を計算する。ポッホハンマーの記号はアペル Apell の記号とも呼ばれている。

[Function] double gsl_sf_lnpoch (double a, double x)

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

ポッホハンマーの記号の対数 log((a)x) = log(Γ(a + x)/Γ(a)) を計算する。a > 0 かつ a + x > 0 である。

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

ポッホハンマーの記号の符号と対数の大きさを計算する。e 計算されるのは result = log(| (a) x |) と sgn = sgn((a)x) である。ここで (a)x := Γ(a + x)/Γ(a) であり、a と a + x は負の整数ではない。

[Function] double gsl_sf_pochrel (double a, double x)

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

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) = (a) x := Γ(a + x)/Γ(a) である。

7.19.4 不完全ガンマ関数

[Function] double gsl_sf_gamma_inc (double a, double x)

[Function] int gsl_sf_gamma_inc_e (double a, double x, gsl sf result * result)

非正規不完全ガンマ関数 Γ(a, x) = $\int_x^\infty dt\ t^{a-1}\exp(-t)$ を計算する。a > 0 で x ≥ 0 である。

[Function] 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)

正規化不完全ガンマ関数 Q(a, x) = 1/Γ(a)$\int_x^\infty dt\ t^{a-1}\exp(-t)$ を計算する。a > 0 で x ≥ 0 である。

[Function] 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)

補数的正規化不完全ガンマ関数 P(a, x) = 1 - Q(a, x) = 1/Γ(a)$\int_0^x dt\ t^{a-1}\exp(-t)$ を計算する。a > 0 で x ≥ 0 である。

Abramowitz & Stegun では P (a, x) を不完全ガンマ関数と呼んでいる (6.5 節参照)。

7.19.5 ベータ関数

[Function] double gsl_sf_beta (double a, double b)

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

a > 0、b > 0 のとき完全ベータ関数 B(a, b) = Γ(a)Γ(b)/Γ(a + b) を計算する。

[Function] double gsl_sf_lnbeta (double a, double b)

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

a > 0、b > 0 のときベータ関数値の対数 log(B(a, b)) を計算する。

7.19.6 不完全ベータ関数

[Function] double gsl_sf_beta_inc (double a, double b, double x)

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

正規化不完全ベータ関数 $B_x(a, b)/B(a, b)$ の値を計算する。ここで $B_x(x) = \int_0^x t^{a-1}(1-t)^{b-1}dt$ で、

a > 0 かつ b > 0 で 0 ≤ x ≤ 1 である。

7.20 ゲーゲンバウア関数

ゲーゲンバウア多項式は Abramowitz & Stegun の第 22 章に定義されている。これは超球面

多項式とも呼ばれる。この節で説明する関数はヘッダファイル 'gsl_sf_gegenbauer.h' で宣言されている。

[Function] 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)

陽に表されたゲーゲンバウア多項式 $C_n^{(\lambda)}(x)$ の値を、n = 1, 2, 3 に関して計算する。

[Function] 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)

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

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

ゲーゲンバウア多項式 $C_n^{(\lambda)}(x)$ の配列の値を、n = 0, 1, 2, . . . , nmax に関して、$\lambda > -1/2$, nmax ≥ 0 のとき計算する。

7.21 超幾何関数

超幾何関数は Abramowitz & Stegun の第 13 および 15 章に定義されている。この節で説明する関数は 'gsl_sf_hyperg.h' で宣言されている。

[Function] double gsl_sf_hyperg_0F1 (double c, double x)

[Function] int gsl_sf_hyperg_0F1_e (double c, double x, gsl_sf_result * result)

超幾何関数 ${}_0F_1(c, x)$ の値を計算する。

[Function] double gsl_sf_hyperg_1F1_int (int m, int n, double x)

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

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

[Function] 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)

合流型超幾何関数 ${}_1F_1(a, b, x) = M(a, b, x)$ の値を一般の a、b に対して計算する。

[Function] double gsl_sf_hyperg_U_int (int m, int n, double x)

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

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。

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

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を返すことができる。

[Function] double gsl_sf_hyperg_U (double a, double b, double x)

[Function] int gsl_sf_hyperg_U_e (double a, double b, double x)

合流型超幾何関数 U (a, b, x) の値を計算する。

[Function] int gsl_sf_hyperg_U_e10_e (double a, double b, double x, gsl_sf_result_e10 * result)

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を返すことができる。

[Function] 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)

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) が特異点に非常に近い値で、関数計算の収束が非常に遅い場合は、 `GSL_EMAXITER` を返すことがある。これは x = 1 で整数 m に対して c − a − b = m となるようなときに生じる。

[Function] 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)

ガウスの超幾何関数 $_2F_1$ ($a_R + ia_I$, $a_R - ia_I$, c, x) の値を複素数に対して計算する。|x| < 1 である。

[Function] 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)

再正規化ガウス超幾何関数 $_2F_1$ (a, b, c, x)/Γ (c) の値を計算する。|x| < 1 である。

[Function] 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)

再正規化ガウス超幾何関数 $_2F_1$ ($a_R + ia_I$, $a_R - ia_I$, c, x)/Γ (c) の値を計算する。|x| < 1 である。

[Function] 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)

超幾何関数 $_2F_0$ (a, b, x) の値を計算する。これは数列で表現すると発散するが、x < 0 では $_2F_0$ (a, b, x) = $(-1/x)^a$ U (a, 1 + a − b, −1/x) となる。

7.22 ラゲール関数

ラゲール多項式は合流型超幾何関数の一種で、L_n^a (x) = $((a + 1)_n / n!)_1F_1$ (−n, a + 1, x) として定義される。この節で説明する関数はヘッダファイル ‘`gsl_sf_laguerre.h`’ で宣言されている。

[Function] 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)

一般ラゲール多項式 $L_a^1(x)$、$L_a^2(x)$、$L_a^3(x)$ の値を陽な記述に従って計算する。

[Function] double gsl_sf_laguerre_n (const int n, const double a, const double x)

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

一般ラゲール多項式 $L_a^n(x)$ を $a > -1$、$n \geq 0$ について計算する。

7.23 ランバートの W 関数

ランバートの W 関数 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 $x < 0$ で複数の枝があるが、実数の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、$x < 0$ で $W < -1$ となるもう一方を $W_{-1}(x)$ とする。ランバートの関数は 'gsl_sf_lambert.h' で宣言されている。

[Function] double gsl_sf_lambert_W0 (double x)

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

ランバートの W 関数の主枝 $W_0(x)$ の値を計算する。

[Function] double gsl_sf_lambert_Wm1 (double x)

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

ランバートの W 関数の主枝でない枝 $W_{-1}(x)$ の値を計算する。

7.24 ルジャンドル関数と球面調和関数

ルジャンドル関数とルジャンドル多項式については Abramowitz & Stegun の第 8 章に解説されている。この節で説明する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

[Function] 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)

それぞれ $l = 1, 2, 3$ についてルジャンドル多項式 $P_l(x)$ を計算する。

[Function] double gsl_sf_legendre_Pl (int l, double x)

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

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 $P_l(x)$ を計算する。

[Function] int gsl_sf_legendre_Pl_array (int lmax, double x, double result_array [])

$l = 0, \ldots, lmax$, $|x| \leq 1$ についてルジャンドル多項式 $P_l(x)$ を計算する。

[Function] double gsl_sf_legendre_Q0 (double x)

[Function] int gsl_sf_legendre_Q0_e (double x, gsl_sf_result * result)

$x > -1$, x についてルジャンドル多項式 $Q_0(x)$ を計算する。

[Function] double gsl_sf_legendre_Q1 (double x)

[Function] int gsl_sf_legendre_Q1_e (double x, gsl_sf_result * result)

x > −1, x ≠ 1 についてルジャンドル多項式 $Q_1(x)$ を計算する。

[Function] double gsl_sf_legendre_Ql (int l, double x)

[Function] int gsl_sf_legendre_Ql_e (int l, double x, gsl_sf_result * result)

x > −1, x ≠ 1 かつ l ≥ 0 についてルジャンドル多項式 $Q_l(x)$ を計算する。

7.24.2 ルジャンドル陪関数と球面調和関数

以下に説明する関数ではルジャンドル陪関数 $P_l^m(x)$ 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 程度になるとオーバーフロを起こすことがある。m が小さければ何の問題もないが、m と l がどちらも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 の値がどちらも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。

球面調和関数を計算したい場合には、これらの関数を使ってはいけない。その代わりに、同じような再起計算を行うが正規化されている gsl_sf_legendre_sphPlm() を使うべきである。

[Function] 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)

m ≥ 0, l ≥ m, |x| ≤ 1 についてルジャンドル陪関数 $P_l^m(x)$ を計算する。

[Function] 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[])

m ≥ 0, l = |m|, . . . , lmax, |x| ≤ 1 について、ルジャンドル多項式の配列 $P_l^m(x)$ を、またオプションとしてその導関数値を計算 する。

[Function] 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)

球面調和関数を計算するため、正規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 ≥ 0, l ≥ m, |x| ≤ 1 でなければならない。$P_l^m(x)$ に標準的な正規化を行っても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。

[Function] 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[])

正規ルジャンドル陪関数の配列 $\sqrt{((2l + 1)/(4\pi))}\ \sqrt{((l - m)! /(l + m)!)}\ P_l^m(x)$ を、またオプションとしてその導関数値を計算する。ただし m ≥ 0, l = |m|, . . . , lmax, |x| ≤ 1 である。

[Function] int gsl_sf_legendre_array_size (const int lmax, const int m)

P_l^m (x) の配列を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。

7.24.3 円錐関数

円錐関数 $P_{\mu-(1/2)+i\lambda}(x)$, $Q_{\mu-(1/2)+i\lambda}$ については Abramowitz & Stegun の第 8.12 節に説明されている。

[Function] double gsl_sf_conicalP_half (double lambda, double x)

[Function] int gsl_sf_conicalP_half_e (double lambda, double x, gsl_sf_result * result)

非正規球面円錐関数 $P_{-1/2-1/2+i\lambda}(x)$ を $x > -1$ で計算する

[Function] double gsl_sf_conicalP_mhalf (double lambda, double x)

[Function] int gsl_sf_conicalP_mhalf_e (double lambda, double x, gsl_sf_result * result)

正規球面円錐関数 $P_{-1/2-1/2+i\lambda}(x)$ を $x > -1$ で計算する。

[Function] double gsl_sf_conicalP_0 (double lambda, double x)

[Function] int gsl_sf_conicalP_0_e (double lambda, double x, gsl_sf_result * result)

円錐関数 $P_{0-1/2+i\lambda}(x)$ を $x > -1$ で計算する。

[Function] double gsl_sf_conicalP_1 (double lambda, double x)

[Function] int gsl_sf_conicalP_1_e (double lambda, double x, gsl_sf_result * result)

円錐関数 $P_{1-1/2+i\lambda}(x)$ を $x > -1$ で計算する。

[Function] 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)

正規球面円錐関数 $P_{-1/2-l-1/2+i\lambda}(x)$ を $x > -1, l \geq -1$ で計算する。

[Function] 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)

正規円柱円錐関数 $P_{-m-1/2+i\lambda}(x)$ を $x > -1, m \geq -1$ で計算する。

7.24.4 双曲面上の円形関数

以下で説明する球面関数はルジャンドル関数から得られ、これにより三次元の双曲面 H3d 上のラプラシアンの正則な固有関数が得られる。$\lambda \to \infty$、$\eta \to 0$ の極限でも $\lambda\,\eta$ の値は一定である。

[Function] 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)

三次元の双曲面上のラプラシアンの、零次の円形固有関数 $L_0^{H3d}(\lambda, \eta) := \sin(\lambda\,\eta)/(\lambda \sinh(\eta))$ を $\eta \geq 0$ で計算する。上述の極限で関数値は $L_0^{H3d}(\lambda, \eta) = j_0(\lambda\,\eta)$ になる。

[Function] 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)

三次元の双曲面上のラプラシアンの、一次の円形固有関数 $L_1^{H3d}(\lambda, \eta) := 1/\sqrt{\lambda^2 + 1}\,\sin(\lambda\,\eta)/(\lambda\,\sinh(\eta))(\coth(\eta) - \lambda\,\cot(\lambda\,\eta))$ を $\eta \geq 0$ で計算する。上述の極限で関数値は $L_1^{H3d}(\lambda, \eta) = j_1(\lambda\,\eta)$ になる。

[Function] 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)

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 $\eta \geq 0$ を $l \geq 0$ で計算する。上述の極限で関数値は $L_l^{H3d}(\lambda, \eta) = j_l(\lambda\,\eta)$ になる。

[Function] int gsl_sf_legendre_H3d_array (int lmax, double lambda, double eta, double result_array [])

円形固有関数の配列 $L_l^{H3d}(\lambda, \eta)$ を $0 \le l \le lmax$ で計算する。

7.25 対数関連の関数

対数関数の性質などについては Abramowitz & Stegun の第 4 章にある。この節で説明する関数はヘッダファイル 'gsl_sf_log.h' で宣言されている。

[Function] double gsl_sf_log (double x)

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

x の対数値 log(x) を x > 0 で計算する。

[Function] double gsl_sf_log_abs (double x)

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

x の絶対値の対数値 log(|x|) を x で計算する。

[Function] int gsl_sf_complex_log_e (double zr, double zi, gsl_sf_result * lnr, gsl_sf_result * theta)

複素数 $z = z_r + iz_i$ の対数値を計算する。計算結果は引数 lnr と theta に入れられる。ここで θ は $[-\pi, \pi]$ の範囲内で、$\exp(lnr + i\theta) = z_r + iz_i$ である。

[Function] double gsl_sf_log_1plusx (double x)

[Function] int gsl_sf_log_1plusx_e (double x, gsl_sf_result * result)

x 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 x > −1 で計算する。

[Function] double gsl_sf_log_1plusx_mx (double x)

[Function] int gsl_sf_log_1plusx_mx_e (double x, gsl_sf_result * result)

x 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 x > −1 で計算する。

7.26 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。

[Function] double gsl_sf_pow_int (double x, int n)

[Function] int gsl_sf_pow_int_e (double x, int n, gsl_sf_result * result)

整数 n についてべき乗 x^n を計算する。べき乗は、最も少ない積の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);
```

7.27 プサイ (二重ガンマ) 関数

m 次の多重ガンマ関数は以下の式で定義される。

$$\psi^{(m)}(x) = \left(\frac{d}{dx}\right)^m \psi(x) = \left(\frac{d}{dx}\right)^m \log(\Gamma(x))$$

ここで $\psi(x) = \Gamma'(x)/\Gamma(x)$ は二重ガンマ関数である。これらの関数はヘッダファイル ‘gsl_sf_psi.h’ で宣言されている。

7.27.1 二重ガンマ関数

[Function] double gsl_sf_psi_int (int n)

[Function] int gsl_sf_psi_int_e (int n, gsl_sf_result * result)

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とも呼ばれる。

[Function] double gsl_sf_psi (double x)

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

二重ガンマ関数 $\psi(n)$ を x ($x \neq 0$) に対して計算する。

[Function] double gsl_sf_psi_1piy (double y)

[Function] int gsl_sf_psi_1piy_e (double y, gsl_sf_result * result)

二重ガンマ関数の直線 $1 + iy$ での実数部 $\mathrm{Re}[\psi(1 + iy)]$ を計算する。

7.27.2 三重ガンマ関数

[Function] double gsl_sf_psi_1_int (int n)

[Function] int gsl_sf_psi_1_int_e (int n, gsl_sf_result * result)

三重ガンマ関数 $\psi'(n)$ を正の整数 n に対して計算する。

[Function] double gsl_sf_psi_1 (double x)

[Function] int gsl_sf_psi_1_e (double x, gsl_sf_result * result)

三重ガンマ関数 $\psi'(n)$ を一般の x に対して計算する。

7.27.3 多重ガンマ関数

[Function] double gsl_sf_psi_n (int m, double x)

[Function] int gsl_sf_psi_n_e (int m, double x, gsl_sf_result * result)

多重ガンマ関数 $\psi^{(m)}(x)$ を $m \geq 0$ および $x > 0$ に対して計算する。

7.28 シンクロトロン関数

この節で説明する関数はヘッダファイル ‘gsl_sf_synchrotron.h’ で宣言されている。

[Function] double gsl_sf_synchrotron_1 (double x)

[Function] int gsl_sf_synchrotron_1_e (double x, gsl_sf_result * result)

最初のシンクロトロン関数 $x \int_x^\infty dt K_{5/3}(t)$ を $x \geq 0$ に対して計算する。

[Function] double gsl_sf_synchrotron_2 (double x)

[Function] int gsl_sf_synchrotron_2_e (double x, gsl_sf_result * result)

二番目のシンクロトロン関数 $xK_{2/3}(x)$ を $x \geq 0$ に対して計算する。

7.29 転移関数

転移関数 J(n, x) は積分 J(n, x) := $\int_0^x$ dt t^n $e^t/(e^t - 1)^2$ として定義される。関数はヘッダファイル‘gsl_sf_transport.h’ で宣言されている。

[Function] double gsl_sf_transport_2 (double x)

[Function] int gsl_sf_transport_2_e (double x, gsl_sf_result * result)

転移関数 J(2, x) を計算する。

[Function] double gsl_sf_transport_3 (double x)

[Function] int gsl_sf_transport_3_e (double x, gsl_sf_result * result)

転移関数 J(3, x) を計算する。

[Function] double gsl_sf_transport_4 (double x)

[Function] int gsl_sf_transport_4_e (double x, gsl_sf_result * result)

転移関数 J(4, x) を計算する。

[Function] double gsl_sf_transport_5 (double x)

[Function] int gsl_sf_transport_5_e (double x, gsl_sf_result * result)

転移関数 J(5, x) を計算する。

7.30 三角関数

このライブラリでは、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確保のため、独自の三角関数を用意している。関数はヘッダファイル ‘gsl_sf_trig.h’ で宣言されている。

7.30.1 周期的な三角関数

[Function] double gsl_sf_sin (double x)

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

正弦関数 sin(x) の値を計算する。

[Function] double gsl_sf_cos (double x)

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

余弦関数 cos(x) の値を計算する。

[Function] double gsl_sf_hypot (double x, double y)

[Function] int gsl_sf_hypot_e (double x, double y, gsl_sf_result * result)

オーバーフローやアンダーフローを避けて斜辺関数 $\sqrt{(x^2 + y^2)}$ の値を計算する。

[Function] double gsl_sf_sinc (double x)

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

いかなる x の値についても、sinc(x) = $\sin(\pi x)/(\pi x)$ の値を計算する。

7.30.2 複素引数の三角関数

[Function] int gsl_sf_complex_sin_e (double zr, double zi, gsl_sf_result * szr, gsl_sf_result * szi)

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。

[Function] int gsl_sf_complex_cos_e (double zr, double zi, gsl_sf_result * czr, gsl_sf_result * czi)

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 szr と szi に入れて返す。

[Function] int gsl_sf_complex_logsin_e (double zr, double zi, gsl_sf_result * lszr, gsl_sf_result * lszi)

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 szr と szi に入れて返す。

7.30.3 双曲線三角関数

[Function] double gsl_sf_lnsinh (double x)

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

$\log(\sinh(x))$ を $x > 0$ に対して計算する。

[Function] double gsl_sf_lncosh (double x)

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

$\log(\cosh(x))$ をいかなる x の値に対しても計算する。

7.30.4 変換関数

[Function] int gsl_sf_polar_to_rect (double r, double theta, gsl_sf_result * x, gsl_sf_result * y)

極座標値 (r, theta) を直交座標値 (x,y) に、$x = r\cos(\theta)$, $y = r\sin(\theta)$ として変換する。

[Function] int gsl_sf_rect_to_polar (double x, double y, gsl_sf_result * r, gsl_sf_result * theta)

直交座標値 (x,y) を極座標値 (r,theta) に、$x = r\cos(\theta)$, $y = r\sin(\theta)$ として計算する。引数 theta は $[-\pi, \pi]$ の範囲内になる。

7.30.5 範囲制限関数

[Function] double gsl_sf_angle_restrict_symm (double theta)

[Function] int gsl_sf_angle_restrict_symm_e (double * theta)

角度 theta を $(-\pi, \pi]$ の範囲内に換算する。

[Function] double gsl_sf_angle_restrict_pos (double theta)

[Function] int gsl_sf_angle_restrict_pos_e (double * theta)

角度 theta を $[0, 2\pi)$ の範囲内に換算する。

7.30.6 誤差見積もりを行う三角関数

[Function] double gsl_sf_sin_err (double x, double dx)

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

角度 x が絶対誤差 dx を含むものとして正弦関数値 $\sin(x \pm dx)$ を計算する。誤差がどの程度伝播していくかを知るために使われる関数であり、エラーハンドラーとともに用いる。

[Function] double gsl_sf_cos_err (double x, double dx)

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

角度 x が絶対誤差 dx を含むものとして余弦関数値 $\cos(x \pm dx)$ を計算する。誤差がどの程度伝播していくかを知るために使われる関数であり、エラーハンドラーとともに用いる。

7.31 ゼータ関数

リーマンのゼータ関数の定義は Abramowitz & Stegun 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。

7.31.1 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 $\zeta(s) = \Sigma_{k=1}^{\infty} k^{-s}$ で定義される。

[Function] double gsl_sf_zeta_int (int n)

[Function] int gsl_sf_zeta_int_e (int n, gsl_sf_result * result)

整数 n、ただし $n \neq 1$ に対してリーマンのゼータ関数 $\zeta(n)$ を計算する。

[Function] double gsl_sf_zeta (double s)

[Function] int gsl_sf_zeta_e (double s, gsl_sf_result * result)

任意の s 、ただし $s \neq 1$ に対してリーマンのゼータ関数 $\zeta(n)$ を計算する。

7.31.2 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、その小数部分は興味深い挙動を示す。そのため、小数部を正確に計算する方法を用意している。

[Function] double gsl_sf_zetam1_int (int n)

[Function] int gsl_sf_zetam1_int_e (int n, gsl_sf_result * result)

整数 n ただし $n \neq 1$ に対して $\zeta(n) - 1$ を計算する。

[Function] double gsl_sf_zetam1 (double s)

[Function] int gsl_sf_zetam1_e (double s, gsl_sf_result * result)

任意の s ただし $s \neq 1$ に対して $\zeta(s) - 1$ を計算する。

7.31.3 フルヴィッツのゼータ関数

フルヴィッツのゼータ関数は $\zeta(s, q) = \Sigma_0^{\infty}(k + q)^{-s}$ で定義される。

[Function] double gsl_sf_hzeta (double s, double q)

[Function] int gsl_sf_hzeta_e (double s, double q, gsl_sf_result * result)

$s > 1$、$q > 0$ 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。

7.31.4 エータ関数

エータ関数は $\eta(s) = (1 - 2^{1-s})\zeta(s)$ で定義される。

[Function] double gsl_sf_eta_int (int n)

[Function] int gsl_sf_eta_int_e (int n, gsl_sf_result * result)

整数 n に対してエータ関数 $\eta(n)$ の値を計算する。

[Function] double gsl_sf_eta (double s)

[Function] int gsl_sf_eta_e (double s, gsl_sf_result * result)

任意の s に対してエータ関数 $\eta(n)$ の値を計算する。

7.32 例

以下の例では、ベッセル関数 $J_0(5.0)$ の計算でエラー型の呼び出しを行う。

```
#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、同じ計算を一般型の呼び出しで行う。この場合はエラー項 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
```

7.33 参考文献

このライブラリではできる限り、Abramowitz & Stegun の例にならっている。

- Abramowitz & Stegun (eds.), Handbook of Mathematical Functions

特殊関数の計算法については、以下の論文に解説がある。

- MISCFUN: A software package to compute uncommon special functions. ACM Trans. Math. Soft., vol. 22, 1996, 288-301
- 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).

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簡便なベクトルと行列について説明する。こういった配列のメモリ管理は、すべてブロックとして実装されている。これらのベクトルや行列を使ったプログラム中で関数の引数として要素の値および次元の両方を渡したい時も、一つの構造体を渡すだけでよい。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

標準的なデータ型について、それぞれ対応する関数が用意されている。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。同様に単精度の float 型に対しては 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

整合性を保つため、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からなる。gsl_block 構造体は以下のような内容である。

```
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することで作られる。スライスは先頭の位置と、添え字とその飛び幅の組み合わせからなる要素の集合である。行列の場合、列の添え字の飛び幅はつまり行の長さである。ベクトルの場合は、飛び幅はストライド stride と呼ばれている。

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

メモリを確保してブロックに割り当てる関数は、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 エラーハンドラーをエラー番号 GSL_ENOMEM で呼び出して、さらに null ポインタを返す。このライブラリで実装されているエラーハンドラーを使ってプログラムを終了させる場合、alloc な

どを呼ぶたびにエラーをチェックする必要はない。

[Function] gsl_block * gsl_block_alloc (size_t n)

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、確保された要素の値は不定である。値を零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。

[Function] gsl_block * gsl_block_calloc (size_t n)

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零として初期化する。

[Function] void gsl_block_free (gsl_block * b)

`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。

8.2.2 ブロックの読み書き

GSL では、ブロックの内容をバイナリあるいは整形済み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。

[Function] int gsl_block_fwrite (FILE * stream, const gsl_block * b)

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、移植性はない。

[Function] int gsl_block_fread (FILE * stream, gsl_block * b)

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。読み込むバイト数はブロックの大きさから決められるため、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。

[Function] int gsl_block_fprintf (FILE * stream, const gsl_block * b, const char * format)

ブロック b の要素を 1 行ずつ format で指定される形式でファイル stream 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。

[Function] int gsl_block_fscanf (FILE * stream, gsl_block * b)

ブロック b の要素を format で指定される形式でファイル 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、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;
}
```

以下に上記のプログラムの出力を示す。

```
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記述されている gsl_vector 構造体で定義される。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、大きさ、飛び幅、要素が保持されるメモリ領域へのポインタであるデータ、ベクトルが持つブロックがあればそれへのポインタであるブロック、ブロックを持っていることを表すフラグ owner 、の五つの要素からなる。以下のような定義がされている。

```
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。添え字の範囲は 0 から size - 1 である。stride は一つの要素から次の要素までの物理的なメモリの、適切なデータ型での飛び幅である。ポインタ data はベクトルの最初の要素のメモリ上の位置を指す。ポインタ block はベクトルの要素を保持するブロック (がある場合) へのポインタである。ベクトルがブロックを保持している場合は owner フラグに 1 が代入され、ベクトルが解放されるときにブロックも解放される。ベクトル構造体内のポインタが他のインスタンスが持つブロックを指している場合、その解放時には owner に 0 が代入されるだけで、どのブロックも解放されない。

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 エラーハンドラーをエラー番号 GSL_ENOMEM で呼び出して、さらに null ポインタを返す。このライブラリで実装されているエラーハンドラーを使ってプログラムを終了させる場合、alloc 等を呼ぶたびにエラーをチェックする必要はない。

[Function] gsl_vector * gsl_vector_alloc (size_t n)

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。

[Function] gsl_vector * gsl_vector_calloc (size_t n)

長さ n のベクトルを生成し、ベクトルの要素を零に初期化する。

[Function] void gsl_vector_free (gsl_vector * v)

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、ベクトルが所持するブロックも解放される。ほかのオブジェクトから生成されたベクトルの場合は、メモリはそのオブジェクトに保持されたままにされ、解放されない。

8.3.2 ベクトル要素の操作

FORTRAN のコンパイラと違って、C コンパイラはベクトルや行列の添え字の範囲の確認を行わないことが多い。範囲確認は GNU C コンパイラの 境界確認 bounds-checking 拡張で可能になるが、これは GCC のデフォルトの機能ではない。関数 gsl_vector_get と gsl_vector_set を使えば移植性のある範囲確認ができ、範囲外となる要素を操作しようとしたときにはエラーを出すことができる。

ベクトルや行列の要素を操作する関数は 'gsl_vector.h' に定義され、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。これを有効にするには、マクロ HAVE_INLINE を定義してコンパイルする必要がある。

もし必要があれば、GSL_RANGE_CHECK_OFF を define してプログラムを再コンパイルすることで、ソースプログラム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れば、範囲確認を無効にすることは、関数呼び出し gsl_vector_get (v,i) を v->data[i*v->stride] に、gsl_vector_set(v,i,x) を v->data[i*v->stride]=x に置き換えることである。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、実行速度の低下はまったくない。

[Function] double gsl_vector_get (const gsl_vector * v, size_t i)

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。

[Function] void gsl_vector_set (gsl_vector * v, size_t i, double x)

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。

[Function] double * gsl_vector_ptr (gsl_vector * v, size_t i)

[Function] const double * gsl_vector_const_ptr (const gsl_vector * v, size_t i)

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。

8.3.3 ベクトル要素の初期化

[Function] void gsl_vector_set_all (gsl_vector * v, double x)

ベクトル v のすべての要素の値を x の値にする。

[Function] void gsl_vector_set_zero (gsl_vector * v)

ベクトル v のすべての要素の値を零にする。

[Function] int gsl_vector_set_basis (gsl_vector * v, size_t i)

ベクトル v の i 番目の要素の値を 1 に、ほかの要素の値を零にすることで、基底ベクトルを作る。

8.3.4 ベクトルの読み書き

このライブラリでは、ベクトルをバイナリあるいは整形済み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。

[Function] int gsl_vector_fwrite (FILE * stream, const gsl_vector * v)

ベクトル v 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、移植性はない。

[Function] int gsl_vector_fread (FILE * stream, gsl_vector * v)

ベクトル v 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して読み込む。

[Function] int gsl_vector_fprintf (FILE * stream, const gsl_vector * v, const char * format)

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 stream 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。

[Function] int gsl_vector_fscanf (FILE * stream, gsl_vector * v)

ベクトル v の要素を format で指定される形式でファイル 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に加え、ベクトルのスライスからベクトルの像 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は像で記述でき、またベクトルの偶数番目と奇数番目の要素からそれぞれ二つの像を造ることができる。

ベクトルの像は一時的なオブジェクトとしてスタックに保持され、ベクトルの要素の部分集合に対する演算で使うことができる。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 なベクトルの像を作るために const でないコピーを作るといった手間は必要ない。ベクトルの像の型は gsl_vector_view、const なベクトルに対しては gsl_vector_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 vector 要素を使って gsl_vector として操作することができる。gsl_vector * 型または const gsl_vector * 型のベクトルへのポインタは、これらの要素に & 演算子をつけることで得られる。

このポインタを使うとき、像そのものが属するスコープを意識しておかねばならない。もっとも簡単なのは、ポインタを常に &view.vector として書き、ほかのポインタ変数に入れないことである。

[Function] 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)

ベクトル v の部分ベクトルの像を返す。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、パラメータ (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。

新しいベクトルは元のベクトル v の持つブロックの像である。v の要素を持つブロックは、新しいベクトルが所有するわけではない。新しいベクトルの像が、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 とそのブロックはそのまま残る。元のベクトルのメモリは、元のベクトルを解放するまで保持されている。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。

[Function] 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)

ベクトル v の、引数で指定された飛び幅を持つ部分ベクトルの像を返す。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でのある要素から次の要素までの飛び幅が 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零にし、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。

[Function] 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)

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。

[Function] 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)

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。

[Function] 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)

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 と宣言されたベクトルに使えること以外は、gsl_vector_view_array と同じである。

[Function] 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)

指定された配列 base から、引数で指定された飛び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でのある要素から次の要素までの飛び幅が 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 と宣言されたベクトル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

加算や乗算などのベクトルに共通して使われる演算は、このライブラリの BLAS くぉ使う部分にある (12 章 [BLAS Support]参照)。しかし BLAS のコードの全体を使わずにすむ関数がいくつかあった方が便利であるため、以下の関数が用意されている。

[Function] int gsl_vector_memcpy (gsl_vector * dest, const gsl_vector * src)

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。

[Function] int gsl_vector_swap (gsl_vector * v, gsl_vector * w)

ベクトル v とベクトル w を、コピーを使って交換する。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 要素の交換

以下の関数でベクトルの要素を交換、置換することができる。

[Function] int gsl_vector_swap_elements (gsl_vector * v, size_t i, size_t j)

ベクトル v の i 番目と j 番目の要素を入れ替える。

[Function] int gsl_vector_reverse (gsl_vector * v)

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。

[Function] int gsl_vector_add (gsl_vector * a, const gsl_vector * b)

ベクトル b の要素の値をベクトル a の要素に、$a = a_i + b_i$ のようにして加える。二つのベクトルは同じ大きさでなければならない。

[Function] int gsl_vector_sub (gsl_vector * a, const gsl_vector * b)

ベクトル b の要素の値をベクトル a の要素から、$a = a_i - b_i$ のようにして減ずる。二つのベクトルは同じ大きさでなければならない。

[Function] int gsl_vector_mul (gsl_vector * a, const gsl_vector * b)

ベクトル b の要素の値をベクトル a の要素に、$a = a_i * b_i$ のようにして乗ずる。二つのベクトルは同じ大きさでなければならない。

[Function] int gsl_vector_div (gsl_vector * a, const gsl_vector * b)

ベクトル a の要素をベクトル b の要素で、$a = a_i / b_i$ のようにして除する。二つのベクトルは同じ大きさでなければならない。

[Function] int gsl_vector_scale (gsl_vector * a, const double x)

ベクトル a の要素に定数係数 x を $a_i' = xa_i$ のようにして乗ずる。

[Function] int gsl_vector_add_constant (gsl_vector * a, const double x)

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

[Function] double gsl_vector_max (const gsl_vector * v)

ベクトル v の要素の中で最大のものの値を返す。

[Function] double gsl_vector_min (const gsl_vector * v)

ベクトル v の要素の中で最小のものの値を返す。

[Function] void gsl_vector_minmax (const gsl_vector * v, double * min_out, double * max_out)

ベクトル v の要素の中で最大および最小のものの値を、max out および min out に入れて返す。

[Function] size_t gsl_vector_max_index (const gsl_vector * v)

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。

[Function] size_t gsl_vector_min_index (const gsl_vector * v)

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。

[Function] void gsl_vector_minmax_index (const gsl_vector * v, size_t * imin, size_t * imax)

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もっとも小さな添え字を返す。

8.3.10 ベクトルの属性

[Function] int gsl_vector_isnull (const gsl_vector * v)

ベクトル v の要素がすべて零のとき 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++)
        printf("v_%d = %g\n", i, gsl_vector_get(v, i));

    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。

```
v_0 = 1.23
v_1 = 2.23
v_2 = 3.23
gsl: vector_source.c:8: ERROR: index out of range
Default GSL error handler invoked.
Aborted (core dumped)
```

次のプログラム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);

    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));

    return 0;
}
```

8.4 行列

行列は一般的なブロックのスライスとして、`gsl_matrix` 構造体で定義される。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、行列の二つの次元数、物理的な次元数、行列の要素を保持しているメモリ領域へのポインタ data、行列が保持しているブロックへのポインタ block 、所持フラグ owner の六個のメンバーがある。物理的次元数はメモリの配置を決め、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` 構造体は以下のような内容である。

```
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る。FORTRAN は列指向である。行の数が size1 、行の添え字の意味を持つ範囲は 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 はここでは追随次元（trailing dimension）とも呼び、メモ

リ上に展開されている行列の行の大きさを表す。

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有していれば 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。ほかの行列が持つブロックのスライスでしかない行列の owner 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 エラーハンドラーをエラー番号 GSL_ENOMEM で呼び出して、さらに null ポインタを返す。このライブラリで実装されているエラーハンドラーを使ってプログラムを終了させる場合、alloc 類を呼ぶたびにエラーをチェックする必要はない。

[Function] gsl_matrix * gsl_matrix_alloc (size_t n1, size_t n2)

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「所有」され、行列が解放されるときにこの所有しているブロックも解放される。

[Function] gsl_matrix * gsl_matrix_calloc (size_t n1, size_t n2)

大きさが n1 行 × n2 列の行列を生成し、行列のすべての要素を零に初期化する。

[Function] void gsl_matrix_free (gsl_matrix * m)

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、その行列が所有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有したままにされ、解放されない。

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。

行列の要素は、C 言語での順序、つまり二番目の添え字についてメモリ上で連続して保持される。正確には、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なる。

```
m->data[i * m->tda + j]
```

ここで tda は行列の物理的な行の長さである。

[Function] double gsl_matrix_get (const gsl_matrix * m, size_t i, size_t j)

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。

[Function] void gsl_matrix_set (gsl_matrix * m, size_t i, size_t j, double x)

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。

[Function] 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)

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。

8.4.3 行列要素の初期化

[Function] void gsl_matrix_set_all (gsl_matrix * m, double x)

行列 m のすべての要素の値を x にする。

[Function] void gsl_matrix_set_zero (gsl_matrix * m)

行列 m のすべての要素の値を 0 にする。

[Function] void gsl_matrix_set_identity (gsl_matrix * m)

行列 m の素の値を単位行列 m(i, j) = δ(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 行列の読み書き

このライブラリでは、行列をバイナリあるいは整形済み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。

[Function] int gsl_matrix_fwrite (FILE * stream, const gsl_matrix * m)

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、移植性はない。

[Function] int gsl_matrix_fread (FILE * stream, gsl_matrix * m)

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。

[Function] int gsl_matrix_fprintf (FILE * stream, const gsl_matrix * m, const char * format)

行列 m の要素を 1 行ずつ format で指定される形式でファイル stream 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。

[Function] int gsl_matrix_fscanf (FILE * stream, gsl_matrix * m)

行列 m の要素を format で指定される形式でファイル 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スタックに保持され、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 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。また行列の像では、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。

[Function] 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)

行列 m の部分行列の像を返す。部分行列のもっとも左上の要素は、元の行列の (k1, k2) 成分である。部分行列は n1 行 × n2 列である。メモリ上での列の物理的な個数は元の行列と像で同じである。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 + k1) + i*m->tda + j]
```

ここで i は 0 から n1 - 1 の範囲、j は 0 から n2 - 1 の範囲である。

生成される行列が保持するポインタ 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 ポインタにされる。

新しい行列は元の行列 m の持つブロックの像である。m の要素を持つブロックは、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。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。

[Function] 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)

配列 base の行列の像を返す。返される行列は n1 行 × n2 列である。メモリ中の列の物理的な個数も 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]
```

添え字 i の範囲は 0 から n1 - 1、添え字 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。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。

[Function] 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)

tda で指定される物理的な行の長さ（追随次元、行列の次元が示す列の数と異なっていてもよい）で、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) = base[i*tda + j]
```

添え字 i の範囲は 0 から n1 - 1、添え字 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。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。

[Function] 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)

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。ベクトルの飛び幅は 1 でなければならない。メモリ中の列の物理的な個数も 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]
```

添え字 i の範囲は 0 から n1 - 1、添え字 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。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。

[Function] 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)

tda で指定される物理的な行の長さ（追随次元、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。ベクトルの飛び幅は 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]
```

添え字 i の範囲は 0 から n1 - 1、添え字 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。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、オブジェクトを操作するには、参照と複製の二つの方法がある。この説では、参照を使って、行列の行または列のベクトル像を生成する関数について説明する。ベクトル像と行列は同じメモリブロックを使っているため、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。

[Function] 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)

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。

[Function] 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)

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。

[Function] gsl_vector_view gsl_matrix_diagonal (gsl_matrix * m)

[Function] gsl_vector_const_view gsl_matrix_const_diagonal (const gsl_matrix * m)

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。

[Function] 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)

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。行列の対角成分は k = 0 にあたる。

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。

[Function] 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)

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。行列の対角成分は k = 0 にあたる。

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

[Function] int gsl_matrix_memcpy (gsl_matrix * dest, const gsl_matrix * src)

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。

[Function] int gsl_matrix_swap (gsl_matrix * m1, gsl_matrix * m2)

行列 m1 と m2 の要素を、コピーを使って交換する。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

この説では行列の行または列をベクトルとして複製する関数について説明する。これにより、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 `gsl_vector_memcpy` を使うことで、同じことができる。

[Function] int gsl_matrix_get_row (gsl_vector * v, const gsl_matrix * m, size_t i)

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。

[Function] int gsl_matrix_get_col (gsl_vector * v, const gsl_matrix * m, size_t j)

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。

[Function] int gsl_matrix_set_row (gsl_matrix * m, size_t i, const gsl_vector * v)

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。

[Function] int gsl_matrix_set_col (gsl_matrix * m, size_t j, const gsl_vector * v)

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 行または列の交換

行列の行や列の交換は、以下の関数を使って行うことができる。

[Function] int gsl_matrix_swap_rows (gsl_matrix * m, size_t i, size_t j)

行列 m の i 番目と j 番目の行を入れ替える。

[Function] int gsl_matrix_swap_columns (gsl_matrix * m, size_t i, size_t j)

行列 m の i 番目と j 番目の列を入れ替える。

[Function] int gsl_matrix_swap_rowcol (gsl_matrix * m, size_t i, size_t j)

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。

[Function] int gsl_matrix_transpose_memcpy (gsl_matrix * dest, const gsl_matrix * src)

行列 dest を行列 src の転置行列にする。行列 src の要素の値を dest にコピーする。行列 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。

[Function] int gsl_matrix_transpose (gsl_matrix * m)

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。

[Function] int gsl_matrix_add (gsl_matrix * a, const gsl_matrix * b)

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。

[Function] int gsl_matrix_sub (gsl_matrix * a, const gsl_matrix * b)

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。

[Function] int gsl_matrix_mul_elements (gsl_matrix * a, const gsl_matrix * b)

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。

[Function] int gsl_matrix_div_elements (gsl_matrix * a, const gsl_matrix * b)

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。

[Function] int gsl_matrix_scale (gsl_matrix * a, const double x)

定数値 x を行列 a のすべての要素に a (i, j) = x・a(i, j) のようにして乗じる。

[Function] int gsl_matrix_add_constant (gsl_matrix * a, const double x)

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。

[Function] double gsl_matrix_max (const gsl_matrix * m)

行列 m 中の要素で最大のものの値を返す。

[Function] double gsl_matrix_min (const gsl_matrix * m)

行列 m 中の要素で最小のものの値を返す。

[Function] void gsl_matrix_minmax (const gsl_matrix * m, double * min_out, double * max_out)

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。

[Function] void gsl_matrix_max_index (const gsl_matrix * m, size_t * imax, size_t * jmax)

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見つかったときは、行優先で探索して最初に見つかったものを返す。

[Function] void gsl_matrix_min_index (const gsl_matrix * m, size_t * imin, size_t * jmin)

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見つかったときは、行優先で探索して最初に見つかったものを返す。

[Function] void gsl_matrix_minmax_index (const gsl_matrix * m, size_t * imin, size_t * imax)

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見つかったときは、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

[Function] int gsl_matrix_isnull (const gsl_matrix * m)

行列 m 中のすべての要素の値が零の時 1 を、そうでないときは 0 を返す。

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 < 10; i++)
        for (j = 0; j < 3; j++)
            printf("m(%d,%d) = %g\n", i, j,
                   gsl_matrix_get(m, i, j));
    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");g
    sl_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++;
         }

    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
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
```

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。

・ B. Stroustrup, The C++ Programming Language (3rd Ed), Section 22.4 Vector Arithmetic. Addison-Wesley 1997, ISBN 0-201-88954-4.

第 9 章 置換

この章では、置換を生成、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 に適用するとは、$v_i' = v_{pi}$ のようにして新しいベクトル v を作ることである。たとえば、配列 (0, 1, 3, 2) は要素数 4 のベクトルの末尾の二つの要素を入れ替える操作を意味する。同様に、恒等置換は (0, 1, 2, 3) で表される。

線形代数のルーチンで生成される置換は、行列の列を入れ替えることであり、したがってベクトルの要素に対する置換とは、列ベクトルではなく、行ベクトルに対しての操作 v' = vP であると考えなければならない。

この章で説明する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

[Function] gsl_permutation * gsl_permutation_alloc (size_t n)

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、配列要素の値は生成時には不定である。gsl_permutation_calloc を使えば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。

[Function] gsl_permutation * gsl_permutation_calloc (size_t n)

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。

[Function] void gsl_permutation_init (gsl_permutation * p)

置換 p を初期化して（(0, 1, 2, ..., n– 1) のような）恒等置換にする。

[Function] void gsl_permutation_free (gsl_permutation * p)

置換 p のメモリを解放する。

[Function] int gsl_permutation_memcpy (gsl_permutation * dest, const gsl_permutation * src)

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。

[Function] size_t gsl_permutation_get (const gsl_permutation * p, const size_t i)

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。

[Function] int gsl_permutation_swap (gsl_permutation * p, const size_t i, const size_t j)

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

[Function] size_t gsl_permutation_size (const gsl_permutation * p)

置換 p の大きさを返す。

[Function] size_t * gsl_permutation_data (const gsl_permutation * p)

置換 p の要素を保持する配列へのポインタを返す。

[Function] int gsl_permutation_valid (gsl_permutation * p)

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

[Function] void gsl_permutation_reverse (gsl_permutation * p)

置換 p の要素を逆にする。

[Function] int gsl_permutation_inverse (gsl_permutation * inv, const gsl_permutation * p)

置換 p の逆置換を計算し、inv に入れて返す。

[Function] int gsl_permutation_next (gsl_permutation * p)

置換 p を辞書順で次の置換に置き換えて GSL_SUCCESS を返す。辞書順では次がない場合は 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。

[Function] int gsl_permutation_prev (gsl_permutation * p)

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

[Function] int gsl_permute (const size_t * p, double * data, size_t stride, size_t n)

置換 p を大きさ n、飛び幅 stride の配列 data に適用する。

[Function] int gsl_permute_inverse (const size_t * p, double * data, size_t stride, size_t n)

置換 p の逆置換を、大きさ n、飛び幅 stride の配列 data に適用する。

[Function] int gsl_permute_vector (const gsl_permutation * p, gsl_vector * v)

置換 p を行ベクトル v に、右から v = vP のようにして適用する。置換行列 P の j 番目の列は単位行列の p_j 番目の列とされる。置換 p とベクトル v は同じ大きさでなければならない。

[Function] int gsl_permute_vector_inverse (const gsl_permutation * p, gsl_vector * v)

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。置換行列 P の j 番目の列は単位行列の p_j 番目の列とされる。置換 p とベクトル v は同じ大きさでなければならない。

[Function] int gsl_permutation_mul (gsl_permutation * p, const gsl_permutation * pa, const gsl_permutation * pb)

二つの置換 pa と pb を一つの置換 p (= pa.pb) にまとめる。置換 p 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 置換の読み込みと書き込み

このライブラリでは、置換をバイナリあるいは整形済み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。

[Function] int gsl_permutation_fwrite (FILE * stream, const gsl_permutation * p)

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。

[Function] int gsl_permutation_fread (FILE * stream, gsl_permutation * p)

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。

[Function] int gsl_permutation_fprintf (FILE * stream, const gsl_permutation * p, const char *format)

置換 p の要素を一行ずつ、size_t に対して指定されたフォーマット format にしたがってファイル stream に書き出す。GNU では記述子 z が size_t を表すので、"%zu\n" と書くのがよい。書き込みに際してエラーが発生したときは GSL_EFAILED を返す。

[Function] int gsl_permutation_fscanf (FILE * stream, gsl_permutation * p)

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

ある一つの置換を、線形または巡回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、上述してきた添え字の置き換えである。巡回表現とは、要素を元の順序で次にある要素で置き換える置換であり、巡回とも呼ぶ。

巡回 (1 2 3) では、たとえば、1 は 2 に、2 は 3 に、3 は 1 に、巡回的に置き換えられる。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。これは単集合 singleton と呼ばれる。

すべての置換を複数の巡回に分解することができる。置換の巡回形式は一意には決まらないが、要素を並べ替えることで正規型 canonical form として一意に定まる。ここでの実装では、クヌース著 The Art of Computer Programming (Vol 1, 3rd Ed, 1997) p.178 による定義を用いている。

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、置換の理論ではさ広く応用される。

[Function] int gsl_permutation_linear_to_canonical (gsl_permutation * q, const gsl_permutation * p)

置換 p の正規型を計算し引数 q に入れて返す。

[Function] int gsl_permutation_canonical_to_linear (gsl_permutation * p, const gsl_permutation * q)

正規型の置換 q を線形表現に戻して引数 p に入れて返す。

[Function] size_t gsl_permutation_inversions (const gsl_permutation * p)

置換 p に含まれる、2 要素の入れ替えの個数を数える。

[Function] size_t gsl_permutation_linear_cycles (const gsl_permutation * p)

置換 p に含まれる巡回の個数を数える。

[Function] size_t gsl_permutation_canonical_cycles (const gsl_permutation * q)

置換 q が正規型の時に、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");

    return 0;
}
```

以下にプログラムの出力を示す。

```
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);

    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 Sorting and Searching に幅広く述べられている。

・ Donald E. Knuth, The Art of Computer Programming: Sorting and Searching (Vol 3, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850.

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 (Vol 1, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850. Section 1.3.3, An Unusual Correspondence, p. 178-179.

第 10 章 組み合わせ

この章では組み合わせを生成、操作する関数について説明する。組み合わせ c は 0 以上 n − 1 以下の k 個の整数の配列で表現され、各 c_i は 0 から n − 1 の値を取り、含まれない値はありえるが重複することはない。組み合わせ c は、要素数 n のベクトルから k 個の要素を選んだときのその要素の添え字である。ある集合から k 個の要素を選んだ部分集合すべてについて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

[Function] gsl_combination * gsl_combination_alloc (size_t n, size_t k)

新しい組み合わせのためのメモリを、パラメータ n、k で確保する。組み合わせは初期化されないため、確保時の配列要素の値は不定である。関数 gsl_combination_calloc を使うと、辞書順で最初になる組み合わせをメモリ確保と同時に生成できる。十分なメモリが確保できないときは、null ポインタを返す。

[Function] gsl_combination * gsl_combination_calloc (size_t n, size_t k)

新しい組み合わせのためのメモリを、パラメータ n、k で確保し、辞書順で最初になる組み合わせをメモリ確保と同時に生成する。十分なメモリが確保できないときは、null ポインタを返す。

[Function] void gsl_combination_init_first (gsl_combination * c)

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 (0, 1, 2, ..., k − 1) にする。

[Function] void gsl_combination_init_last (gsl_combination * c)

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 (n − k, n − k + 1, ..., n − 1) にする。

[Function] void gsl_combination_free (gsl_combination * c)

組み合わせ c のメモリを解放する。

[Function] int gsl_combination_memcpy (gsl_combination * dest, const gsl_combination * src)

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。

[Function] size_t gsl_combination_get (const gsl_combination * c, const size_t i)

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。

10.4 組み合わせの属性

[Function] size_t gsl_combination_n (const gsl_combination * c)

組み合わせ c のパラメータ n の値を返す。

[Function] size_t gsl_combination_k (const gsl_combination * c)

組み合わせ c のパラメータ k の値を返す。

[Function] size_t * gsl_combination_data (const gsl_combination * c)

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。

[Function] int gsl_combination_valid (gsl_combination * c)

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 回だけ含まれる。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

[Function] int gsl_combination_next (gsl_combination * c)

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。

[Function] int gsl_combination_prev (gsl_combination * c)

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 組み合わせの読み込みと書き込み

このライブラリでは、組み合わせをバイナリ・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。

[Function] int gsl_combination_fwrite (FILE * stream, const gsl_combination * c)

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_EFAILED 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。

[Function] int gsl_combination_fread (FILE * stream, gsl_combination * c)

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_EFAILED 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。

[Function] int gsl_combination_fprintf (FILE * stream, const gsl_combination * c, const char *format)

組み合わせ c の要素をファイル stream に一行ずつ format に指定された書式で書き込む。format は size_t にあわせて、GNU システムなら Z が size_t を表現するため、"%Zu\n" などとするのがよい。書き込みに際してエラーが発生したときは、GSL_EFAILED を返す。

[Function] int gsl_combination_fscanf (FILE * stream, gsl_combination * c)

組み合わせ c の要素をファイル stream から書式に従って読み込む。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_EFAILED 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 }
```

すべての部分集合は 16 個あり、それらはその大きさと辞書順に整列されている。

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, 1998, CRC Press LLC, ISBN 084933988X
- Donald E. Knuth, The Art of Computer Programming: Combinatorial Algorithms (Vol 4, pre-fascicle 2c) http://www-cs-faculty.stanford.edu/~knuth/fasc2c.ps.gz

第 11 章 整列

この章では整列を直接的および間接的 (インデックスを使う) に行う関数について説明する。どの関数もヒープソート法である。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、安定した性能を発揮する。もっとも速度が遅くなる場合（すでに整列されているデータに対する整列）でも、平均的またはもっとも早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なアルゴリズムである。ここで実装している関数は、異なるプラットフォーム上で実行しても、同じ要素の順序が保たれるかどうかは同じ結果となる。

11.1 整列オブジェクト

以下の関数は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、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な整列法であり、より速いため、使える場合は使うべきである。`qsort` の解説は GNU C Library Reference Manual にある。

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。

[Function] void gsl_heapsort (void * array, size_t count, size_t size, gsl_comparison_fn_t compare)

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しておく。

```
int (*gsl_comparison_fn_t) (const void * a, const void * b)
```

比較関数は 1 番目の引数が二番目の引数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、1 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はポインタによる演算で安定な整列を行うことはできない。比較関数中で同じ値を持つ要素のポインタを比較する工夫は、ヒープソートでは使えない。ヒープソートは内部でデータの並べ替えを行うため最初の並びは変わってしまう。

[Function] int gsl_heapsort_index (size_t * p, const void * array, size_t count, size_t size, gsl_comparison_fn_t compare)

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 の要素は、配列が並べ替えられて同じ配列を上書きしたとするときの順番を表す。p の最初の要素は array 中でもっとも小さな要素の順番 (インデックス) を示し、p の最後の要素は array 中で最も大きな要素の順番を示す。配列そのものは変化しない。

11.2 ベクトルの整列

以下に説明する関数は、ベクトルを要素とする配列を直接に、または間接的に (添え字 index を) 整列する。ベクトルの成分は実数でも整数でもよく、配列は添え字を普通に使って、それぞれに対応した関数で整列される。例えば、`float` 配列の関数は `gsl_sort_float` と `gsl_sort_float_index` である。それぞれ対応するベクトル関数は `gsl_sort_vector_float` と `gsl_sort_vector_float_index` である。そのプロトタイプ宣言はヘッダファイル ‘`gsl_sort_vector_float.h`’にある。すべてのベクトル整列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`’ をインクルードすることで参照できる。

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一意には行えないため、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。

[Function] void gsl_sort (double * data, size_t stride, size_t n)

要素数 n の配列 data を、飛び幅 stride で数値の昇順に整列する。

[Function] void gsl_sort_vector (gsl_vector * v)

ベクトル v の要素を数値の昇順に整列する。

[Function] int gsl_sort_index (size_t * p, const double * data, size_t stride, size_t n)

要素数 n の配列 data を、飛び幅 stride で数値の昇順に間接的に (添え字で) 整列し、置換（添え字）を 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 の要素は、配列が並べ替えられて同じ配列を上書きしたとするときの順番を表す。元の配列 data は変化しない。

[Function] int gsl_sort_vector_index (gsl_permutation * p, const gsl_vector * v)

ベクトル v の要素を、昇順で整列した結果を表す置換を 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。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 中の最大の要素を指す。元のベクトル v は変化しない。

11.3 最小または最大のもの k 個の取り出し

この節で説明する関数は、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り、データ集合全体の比べて要素数が少ない、その部分集合に対して調整された、計算量 O(kN) の直接挿入法を使っている。例えば点数が 1,000,000 万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。元のデータ集合の中から取り出される部分集合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しまってから最大または最小の要素を得る方が速いだろう。

[Function] void gsl_sort_smallest (double * dest, size_t k, const double * src, size_t stride, size_t n)

大きさ n、飛び幅 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。

[Function] void gsl_sort_largest (double * dest, size_t k, const double * src, size_t stride, size_t n)

大きさ n、飛び幅 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。

[Function] 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)

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 個の要素の添え字を得る。

[Function] void gsl_sort_smallest_index (size_t * p, size_t k, const double * src, size_t stride, size_t n)

大きさ n、飛び幅 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。

[Function] void gsl_sort_largest_index (size_t * p, size_t k, const double * src, size_t stride, size_t n)

大きさ n、飛び幅 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。

[Function] 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)

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 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位置のことである。順位は添え字の置換 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;
}
```

これは 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 b 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]);

    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 Sorting and Searching により広い説明がある。

・ Donald E. Knuth, The Art of Computer Programming: Sorting and Searching (Vol 3, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850.

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.

第 12 章 BLAS の利用

基本線形代数ルーチン集 (The Basic Linear Algebra Subprograms, BLAS) ではベクトルと行列に基礎的な演算をいくつか定義しており、最適化された質の高い線形代数機能の実装に使うことができる。

このライブラリでは、“CBLAS” と呼ばれる 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な関数と、GSL でのベクトルと行列を操作するための高レベル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演算を行いたい場合はファイル `gsl_blas.h` で宣言されている高レベルな関数を使うとよい。ほとんどの場合はこれを使う方が安全である。GSL の行列は、メモリ上で密に詰め込まれるように (dense-storage) 実装されているため、高レベル関数もそれに対応するもののみである。バンド形式とパック形式の行列に対応した BLAS の機能はすべて、低レベルの CBLAS の関数で使うことができる。

`gsl_cblas` レベルを利用する関数の仕様は `'gsl_cblas.h'` にある。これらの関数は、旧い BLAS を利用するための C 言語の関数を定めた BLAS 技術フォーラムの暫定標準による。他の CBLAS に準拠する実装が使える場合は、このライブラリが提供する版の代わりにそれを使うこともできる。FORTRAN の BLAS ライブラリしかない場合は CBLAS 変換ラッパーを使って FORTRAN ライブラリを CBLAS に変換することができる。旧い FORTRAN 実装を利用するための CBLAS 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 のすべて揃ったパッケージにある関数を付録に挙げる (付録 D「GSL CBLAS ライブラリ」参照)。

BLAS の関数による演算は、3 レベルに分かれている。

- Level 1 ベクトル演算。$y = \alpha x + y$ など。
- Level 2 行列とベクトルの演算。$y = \alpha Ax + \beta y$ 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、行列の種類とその精度を表す部分が付いている。いくつかの一般的な演算やその名前を以下に挙げる。

DOT スカラー積。$x^T y$

AXPY ベクトルの和。$\alpha x + y$

MV 行列とベクトルの積。Ax

SV 行列とベクトルの積の解。inv(A)x

MM 行列同士の積。AB

SM 行列同士の積の解。inv(A)B

行列の種類には以下のものがある。

GE 一般的な行列

GB 一般的なバンド行列

SY 対称行列

SB 対称バンド行列

SP 対称パック行列

HE エルミート行列

HB エルミートバンド行列

HP エルミートパック行列

TR 三角行列

TB 三角バンド行列

TP 三角パック行列

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」を表し、`zgemm` は「倍精度で複素数の行列同士の積」を表す。

12.1 GSL から BLAS を利用する関数

GSL では、組み込みの型を使った密なベクトルを行列のオブジェクトを用意している。このライブラリでは、これらのオブジェクトを使って BLAS による演算を利用する。これらの機能は `gsl_blas.h` ファイルで利用できるようになる。

12.1.1 Level 1

[Function] int gsl_blas_sdsdot (float alpha, const gsl_vector_float * x, const gsl_vector_float * y, float * result)

二つのベクトル 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。

[Function] 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)

二つのベクトル 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。

[Function] 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)

二つのベクトル x と y に関して複素スカラー積 $x^T y$ を計算し、引数 result に入れて返す。

[Function] 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)

二つのベクトル 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 result に入れて返す。

[Function] float gsl_blas_snrm2 (const gsl_vector_float * x)

[Function] double gsl_blas_dnrm2 (const gsl_vector * x)

ベクトル x のユークリッド・ノルム $||x||_2 = \sqrt{(\Sigma x_i^2)}$ を計算する。

[Function] float gsl_blas_scnrm2 (const gsl_vector_complex_float * x)

[Function] double gsl_blas_dznrm2 (const gsl_vector_complex * x)

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。

$$||x||_2 = \sqrt{(\Sigma(Re(x_i)^2 + Im(x_i)^2))}$$

[Function] float gsl_blas_sasum (const gsl_vector_float * x)

[Function] double gsl_blas_dasum (const gsl_vector * x)

ベクトル x の要素の絶対値の和 $\Sigma|x_i|$ を計算する。

[Function] float gsl_blas_scasum (const gsl_vector_complex_float * x)

[Function] double gsl_blas_dzasum (const gsl_vector_complex * x)

ベクトル x の要素の実部と虚部の大きさの和 $\Sigma(|Re(x_i)| + |Im(x_i)|)$ を計算する。

[Function] 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)

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要素の実部と虚部の絶対値の和 $|Re(x_i)| + |Im(x_i)|$ 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。

[Function] 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)

ベクトル x と y の要素を交換する。

[Function] 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)

ベクトル x の要素を y にコピーする。

[Function] 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)

ベクトル x と y の和を計算する。

[Function] 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)

ベクトル x を係数 alpha 倍する。

[Function] 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 [])

ベクトル (a, b) を零にするギブンズ Givens 変換 (c, s) を計算する。

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r' \\ 0 \end{pmatrix}$$

引数のベクトル a と b は上書きされる。

[Function] int gsl_blas_srot (gsl_vector_float * x, gsl_vector_float * y, float c, float s)

[Function] int gsl_blas_drot (gsl_vector * x, gsl_vector * y, const double c, const double s)

ベクトル x と y にギブンズ変換を (x , y) = (cx + sy, −sx + cy) として適用する。

[Function] 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 [])

修正ギブンズ変換を計算する。その仕様はオリジナルの Level-1blas で定められており、参考文献に挙げてある。

[Function] 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 [])

修正ギブンズ変換を適用する。

12.1.2 Level 2

[Function] 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)

行列とベクトルの積と和 y = αop(A)x + βy を計算する。TransA = `CblasNoTrans`、`CblasTrans`、`CblasConjTrans` のときそれぞれ op(A) = A、A^T、A^H である。

[Function] 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)

三角行列 A とベクトルの積と和 x = αop(A)x 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 であると見なされ無視される。

[Function] 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)

x に対して inv(op(A))x 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 であると見なされ無視される。

[Function] 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)

対称行列 A に対して行列とベクトルの積と和 y = αAx + βy 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。

[Function_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)

エルミート行列 A に対して行列とベクトルの積と和 y = αAx + βy 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。対角成分の虚部は零であると見なされ、無視される。

[Function] 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)

行列 A の階数 1 の更新（rank-1 update） $A = \alpha x y^T + A$ を計算する。

[Function] 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)

行列 A の階数 1 の共役更新（conjugate rank-1 update） $A = \alpha x y^H + A$ を計算する。

[Function] 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)

対称行列 A の階数 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。

[Function] 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)

エルミート行列 A の階数 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。対角成分の虚部は零であると見なされ、無視される。

[Function] 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)

対称行列 A の二階対称更新 $A = \alpha x y^T + \alpha y x^T + A$ 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。

[Function] 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)

エルミート行列 A の二階エルミート更新 A = $\alpha xy^H + \alpha * yx^H$ 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。対角成分の虚部は零であると見なされ、無視される。

12.1.3 Level 3

[Function] 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)

行列同士の積と和 C = αop(A)op(B) + βC 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。

[Function] 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)

行列同士の積と和を計算する。A が対称行列で Side が `CblasLeft` のとき C = αAB + βC を、Side が `CblasRight` のとき C = αBA + βC 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。

[Function] 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)

行列同士の積と和を計算する。A がエルミート行列で Side が `CblasLeft` のとき C = αAB + β C を、Side が `CblasRight` のとき C = αBA + β 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。対角成分の虚部は零であると見なされ、無視される。

[Function] 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)

行列同士の積を計算する。Side が`CblasLef のとき B = αop(A)B を、Side が CblasRight のとき B = αBop(A) を計算する。行列 A は三角行列で、TransA = 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。

[Function] 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)

行列同士の積を計算する。Side が CblasLeft のときを B = αop(inv(A))B を、Side が`CblasRight のとき B = αBop(inv(A)) を計算する。行列 A は三角行列で、TransA = 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。

[Function] int gsl_blas_ssyrk (CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_float * A, float beta, gsl_matrix_float * C)

[Function] int gsl_blas_dsyrk (CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix * A, double beta, gsl matrixgsl_matrix * C)

[Function] int gsl_blas_csyrk (CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl complex float alpha, const gsl_matrix_complex_float * A, const gsl_complex_float beta, gsl_matrix_complex_float * C)

[Function] int gsl_blas_zsyrk (CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_complex beta, gsl_matrix_complex * C)

対称行列 C の k 階対称更新を計算する。Trans が `CblasNoTrans` のとき $C = \alpha AA^T + \beta C$ 、Trans が `CblasTrans` のとき $C = \alpha A^TA + \beta C$ 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。

[Function] 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)

エルミート行列 C の k 階エルミート更新を計算する。Trans が `CblasNoTrans` のとき $C = \alpha AA^H + \beta C$ 、Trans が `CblasTrans` のとき $C = \alpha A^HA + \beta C$ 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零に設定される。

[Function] 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)

対称行列 C の 2k 階対称更新を計算する。Trans が `CblasNoTrans` のとき $C = \alpha AB^T + \alpha BA^T + \beta C$ 、Trans が `CblasTrans` のとき $C = \alpha AB^T + \alpha BA^T + \beta C$ 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。

[Function] int gsl_blas_cher2k (CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_matrix_complex_float * B, float beta, gsl_matrix_complex_float * C)

[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, double beta, gsl_matrix_complex * C)

エルミート行列 C の 2k 階エルミート更新を計算する。Trans が CblasNoTrans のとき $C = \alpha AB^H + \alpha * BA^H + \beta C$ 、Trans が `CblasConjTrans` のとき $C = \alpha A^HB + \alpha * B^HA + \beta C$ 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零

に設定される。

12.2 例

以下に Level-3blas 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

Level 1、2、3blasBLAS の仕様は、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, Vol. 5 (1979), Pages 308-325.
- J.J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms", ACM Transactions on Mathematical Software, Vol. 14, No. 1 (1988), Pages 1-32.
- J.J. Dongarra, I. Duff, J. DuCroz, S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms", ACM Transactions on Mathematical Software, Vol. 16 (1990), Pages 1-28.

後二者の論文の PostScript 版が http://www.netlib.org/blas/ にある。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を解くための関数について説明する。このライブラリでは、gsl_vector および gsl_matrix オブジェクトに対して直接演算を行う、簡便な線形代数の演算を行う関数を用意している。簡潔なアルゴリズムでも支障のでない「小さな」系を想定している。

規模の大きな問題を解こうとするときは、LAPACK にある洗練されたルーチンを使うことができる。線形代数の標準パッケージとして、LAPACK の FORTRAN 版が推奨される。ブロック分割アルゴリズム、特殊なデータ構造、その他の最適化をそれで行うことができる。

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

13.1 LU 分解

一般の正方行列 A は、LU 分解で上三角および下三角の二つの行列にわけることができる。

$$PA = LU$$

ここで P は置換行列、L は下三角単位行列、L は上三角行列である。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 (Ly = Pb, Ux = y) に変換でき、それぞれ前方および後方置換で解けるようになる。LU 分解は、特異行列に対しても使うことができる。

[Function] 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)

正方行列 A を P A = 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、特に返されない。

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, Matrix Computations, Algorithm 3.4.1 参照)。

[Function] 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)

行列 A を gsl_linalg_LU_decomp または gsl_linalg_complex_LU_decomp を使って (LU, p) に LU 分解して、Ax = b を解く。

[Function] 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)

行列 A を (LU, p) に LU 分解して、Ax = b を解いて結果で上書きする。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。

[Function] 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)

行列 A を (LU, p) に繰り返し何度も LU 分解することで、Ax = b のよい解 x を得る。初期残差 r = Ax − b も計算され、residual に入れられる。

[Function] 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)

行列 A のに 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 を解いて得られる。可能な場合は、直接に逆行列を計算するよりもこの方がよい。

[Function] double gsl_linalg_LU_det (gsl_matrix * LU, int signum)

[Function] gsl_complex gsl_linalg_complex_LU_det (gsl_matrix_complex * LU, int signum)

行列 A のに 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置換 signum の符号から計算される。

[Function] double gsl_linalg_LU_lndet (gsl_matrix * LU)

[Function] double gsl_linalg_complex_LU_lndet (gsl_matrix_complex * LU)

行列 A のに 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。

[Function] int gsl_linalg_LU_sgndet (gsl_matrix * LU, int signum)

[Function] gsl_complex gsl_linalg_complex_LU_sgndet (gsl_matrix_complex * LU, int signum)

行列 A のに LU 分解 LU から、その行列式の符号または位相 det(A)/|det(A)| を計算する。

13.2 QR 分解

正方でない一般の M × N 行列 A は、一般に QR 分解を使って直交 M 次正方行列 Q（$Q^T Q = I$ という性質を持つ）と M × N の右三角行列 R に分けられる。

$$A = QR$$

QR 分解は線形問題 Ax = b を三角問題 $Rx = Q^T b$ に変換し、これにより後退置換で解けるようになる。またベクトルの集合に対して、その直交基底の計算にも使うことができる。A がフル・ランクのとき、Q の最初の N 列は、行列 A の範囲 ran(A) の直交基底をなす。

[Function] int gsl_linalg_QR_decomp (gsl_matrix * A, gsl_vector * tau)

M × 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 の下三角成分の列がそれぞれハウスホルダー係数とハウスホルダー・ベクトルで、この二つで行列 Q を表す。ベクトル tau の長さは k = min(M, N) でなければならない。$Q_i = I - \tau_i v_i v_i^T$ で v_i がハウスホルダー・ベクトル v_i = (0, ..., 1, A(i+1, i), A(i+2, i), ..., A(m, i)) とするとき、行列 Q は $Q = Q_k \ldots Q_2 Q_1$ と表される。この格納様式は LAPACK と同じである。

ここではハウスホルダー QR 分解のアルゴリズムを使っている (Golub & Van Loan, Matrix Computations, Algorithm 5.2.1 参照)。

[Function] int gsl_linalg_QR_solve (const gsl_matrix * QR, const gsl_vector * tau, const gsl_vector * b, gsl_vector * x)

gsl_linalg_QR_decomp を使って行列 A を (QR, tau) に QR 分解することで Ax = b を解く。

[Function] int gsl_linalg_QR_svx (const gsl_matrix * QR, const gsl_vector * tau, gsl_vector * x)

gsl_linalg_QR_decomp を使って行列 A を (QR, tau) に QR 分解することで 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。

[Function] int gsl_linalg_QR_lssolve (const gsl_matrix * QR, const gsl_vector * tau, const gsl_vector * b, gsl_vector * x, gsl_vector * residual)

行列 A の行や列が多すぎる場合に、Ax = b の最小二乗誤差の解を計算する。最小二乗誤差の解とは残差のユークリッド・ノルム ||Ax – b|| を最小にする解のことである。ここでは gsl_linalg_QR_decomp を使って A を (QR, tau) に QR 分解し、結果を x に入れて返す。残差の値は残差ベクトルの二乗として計算され、residual に入れて返される。

[Function] int gsl_linalg_QR_QTvec (const gsl_matrix * QR, const gsl_vector * tau, gsl_vector * v)

分解された (QR,tau) 中に含まれる QT をベクトル v に適用し、結果 $Q^T v$ を v に上書きして返す。行列との積はハウスホルダー・ベクトルを直接使って行われるため、行列 QT の全体を計算する必要はない。

[Function] int gsl_linalg_QR_Qvec (const gsl_matrix * QR, const gsl_vector * tau, gsl_vector * v)

分解された (QR, tau) 中に含まれる Q をベクトル v に適用し、結果 Qv を v に上書きして返す。行列との積はハウスホルダー・ベクトルを直接使って行われるため、行列 Q の全体を計算する必要はない。

[Function] int gsl_linalg_QR_Rsolve (const gsl_matrix * QR, const gsl_vector * b, gsl_vector * x)

三角問題 Rx = b を x について解く。b' = Q^Tb 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。

[Function] int gsl_linalg_QR_Rsvx (const gsl_matrix * QR, gsl_vector * x)

三角問題 Rx = b を x について解く。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。b' = Q^Tb 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。

[Function] int gsl_linalg_QR_unpack (const gsl_matrix * QR, const gsl_vector * tau, gsl_matrix * Q, gsl_matrix * R)

Q が M × M の正方行列で R が M × N のとき、QR 分解された (QR, tau) を展開して Q と R に入れて返す。

[Function] int gsl_linalg_QR_QRsolve (gsl_matrix * Q, gsl_matrix * R, const gsl_vector * b, gsl_vector * x)

Rx = Q^Tb 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 と R に展開されている場合に有用である。

[Function] int gsl_linalg_QR_update (gsl_matrix * Q, gsl_matrix * R, gsl_vector * w, const gsl_vector * v)

行列の QR 分解 (Q, R) の、一階の更新 wvT を計算する。更新は Q'R' = QR + wv^T で与え

られ、得られる Q と R はどちらも直交で右三角になる。w の値は上書きされる。

[Function] int gsl_linalg_R_solve (const gsl_matrix * R, const gsl_vector * b, gsl_vector * x)

三角問題 Rx = b を N 次の正方行列 R について解く。

[Function] int gsl_linalg_R_svx (const gsl_matrix * R, gsl_vector * x)

三角問題 Rx = b を N 次の正方行列 R について解く。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

列置換 P を導入すると、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 列は列の階数が r の行列に対して A の範囲で直交基底をなす。この分解で線形問題 Ax = b を三角問題 Ry = Q^Tb, x = Py に変換して、後退置換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、ここでは QR 分解を QRP^T と書く。

[Function] int gsl_linalg_QRPT_decomp (gsl_matrix * A, gsl_vector * tau, gsl_permutation * p, int *signum, gsl_vector * norm)

M × N 行列 A を A = QRP^T と表せる積に分解する。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、と行列 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 の長さは k = min(M, N) でなければならない。$Q_i = I - \tau_i v_i v_i^T$ で v_i = (0, ..., 1, A(i+1, i), A(i+2, i), ..., A(m, i)) とするとき、行列 Q は $Q = Q_k ... Q_2 Q_1$ の形に書き表すことができ、これは LAPACK でのデータ形式と同じである。ベクトル norm は長さ N で、列ピボッティング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を採用している (Golub & Van Loan, Matrix Computations, Algorithm 5.4.1 参照)。

[Function] 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)

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。

[Function] int gsl_linalg_QRPT_solve (const gsl_matrix * QR, const gsl_vector * tau, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)

`gsl_linalg_QRPT_decomp` を使って行列 A を QRP^T 分解し、Ax = b を解く。分解した結果を (QR, tau, p) に入れて返す。

[Function] int gsl_linalg_QRPT_svx (const gsl_matrix * QR, const gsl_vector * tau, const gsl_permutation * p, gsl_vector * x)

行列 A を QRP^T 分解し、Ax = b を解く。分解した結果を (QR, tau, p) に入れて返す。引数で与えるときに x には b の値を持たせ、これが解で置き換えられる。

[Function] int gsl_linalg_QRPT_QRsolve (const gsl_matrix * Q, const gsl_matrix * R, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)

$RP^Tx = Q^Tb$ 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。

[Function] int gsl_linalg_QRPT_update (gsl_matrix * Q, gsl_matrix * R, const gsl_permutation * p, gsl_vector * u, const gsl_vector * v)

QRP^T 分解 (Q, R, p) の一階の更新を計算する。更新は 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 はどちらも直交行列で右三角である。引数 w の値は書き換えられる。置換 p は変化しない。

[Function] int gsl_linalg_QRPT_Rsolve (const gsl_matrix * QR, const gsl_permutation * p, const gsl_vector * b, gsl_vector * x)

引数で与えられる QR 中の N × N 行列 R で定義される三角問題 RP^{T}x = b を解く。

[Function] int gsl_linalg_QRPT_Rsvx (const gsl_matrix * QR, const gsl_permutation * p, gsl_vector * x)

引数で与えられる QR 中の N × N 行列 R で定義される三角問題 RP^{T}x = b を解く。引数で与えるときに x には b の値を持たせ、これが解で置き換えられる。

13.4 特異値分解

一般の正方でない M × N の行列 A は、M × N の直交行列 U と、特異値を表す N × N の対角行列 S と、N × N の正方直交行列 V の転置行列の積の形に、以下のように分解できる。

$$A = USV^T$$

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、降順に整列される。

行列の特異値分解には幅広い応用例がある。行列の条件数は、特異値のうち最大のものと最小のものの比として与えられる。特異値に零が含まれていれば、その行列は正則ではない。非零の特異値の個数は行列の階数である。しかし階数が縮退している行列を特異値分解しても、数値計算の精度の限界から厳密に零になる特異値は得られない。非常に小さな値の特異値は、適切な誤差見積もりの元で零とみなすべきかどうか判断される。

[Function] int gsl_linalg_SV_decomp (gsl_matrix * A, gsl_matrix * V, gsl_vector * S, gsl_vector * work)

M ≥ N である M × 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。

ここではゴルブ- ラインシュ Golub-Reinsch の特異値分解法を採用している。

[Function] int gsl_linalg_SV_decomp_mod (gsl_matrix * A, gsl_matrix * X, gsl_matrix * V, gsl_vector * S, gsl_vector * work)

修正ゴルブ- ラインシュ法を使って特異値分解を行う。この方法は M >> 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 × N 行列 X が必要である。

[Function] int gsl_linalg_SV_decomp_jacobi (gsl_matrix * A, gsl_matrix * V, gsl_vector * S)

単方向ヤコビ直交化（参考文献参照）を使って特異値分解を行う。ヤコビ法はゴルブ- ラインシュ法に比較すると精度がよい。

[Function] int gsl_linalg_SV_solve (gsl_matrix * U, gsl_matrix * V, gsl_vector * S, const gsl_vector * b, gsl_vector * x)

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。零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零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 (over-determined な) 場合、解は両辺の差の二乗 $||Ax - b||^2$ を最小化するように計算される。

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る。これにより、線形問題 Ax = b を二つの三角問題 (Ly = b, L^Tx = y) に分解し、前進および後退置換でとくことができるようになる。

[Function] int gsl_linalg_cholesky_decomp (gsl_matrix * A)

正定値の正方行列 A を $A = LL^T$ の形にコレスキー分解する。引数で与えられる 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。

[Function] int gsl_linalg_cholesky_solve (const gsl_matrix * cholesky, const gsl_vector * b, gsl_vector * x)

gsl_linalg_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。

[Function] int gsl_linalg_cholesky_svx (const gsl_matrix * cholesky, gsl_vector * x)

gsl_linalg_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。x には引数として与えられるときには b の値を入れておく。得られた解はこれを上書きし、x に入れて返される。

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である。

[Function] int gsl_linalg_symmtd_decomp (gsl_matrix * A, gsl_vector * tau)

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 の対角および副対角成分に入れて返される。A の下三角成分には直交行列 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 の上三角成分は無視される。

[Function] int gsl_linalg_symmtd_unpack (const gsl_matrix * A, const gsl_vector * tau, gsl_matrix * Q, gsl_vector * diag, gsl_vector * subdiag)

関数 gsl_linalg_symmtd_decomp によって得られた対称三重対角行列 (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。

[Function] int gsl_linalg_symmtd_unpack_T (const gsl_matrix * A, gsl_vector * diag, gsl_vector * subdiag)

関数 gsl_linalg_symmtd_decomp によって得られた対称三重対角行列 (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 ハミルトン行列の三重対角分解

ハミルトン行列 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、T は実対称三重対角行列である。

[Function] int gsl_linalg_hermtd_decomp (gsl_matrix_complex * A, gsl_vector_complex * tau)

ハミルトン行列 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 の対角および副対角成分の実部に入れて返される。A の下三角成分には直交行列 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は無視される

[Function] int gsl_linalg_hermtd_unpack (const gsl_matrix_complex * A, const gsl_vector_complex * tau, gsl_matrix_complex * Q, gsl_vector * diag, gsl_vector * subdiag)

関数 gsl_linalg_hermtd_decomp によって得られた対称三重対角行列 (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。

[Function] int gsl_linalg_hermtd_unpack_T (const gsl_matrix_complex * A, gsl_vector * diag, gsl_vector * subdiag)

関数 gsl_linalg_hermtd_decomp によって得られた対称三重対角行列 (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 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 × N の二重対角行列で、対角成分と上副対角成分以外の要素は零である。U の大きさは M × N で、V は N × N である。

[Function] int gsl_linalg_bidiag_decomp (gsl_matrix * A, gsl_vector * tau_U, gsl_vector * tau_V)

M × N の行列 varA を U BV^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 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で、tau_V の長さはそれよりも 1 だけ短くなければならない。

[Function] 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)

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 $U^T U = I$ を満たす M × N の直交行列に入れられる。

[Function] int gsl_linalg_bidiag_unpack2 (gsl_matrix * A, gsl_vector * tau_U, gsl_vector * tau_V, gsl_matrix * V)

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。

[Function] int gsl_linalg_bidiag_unpack_B (const gsl_matrix * A, gsl_vector * diag, gsl_vector * superdiag)

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。

13.9 ハウスホルダー変換

ハウスホルダー変換とは単位行列に対する階数 1 の書き換えであり、これによりベクトルの任意の要素を選んで零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。この節で説明する関数はハウスホルダー行列の一階構造を使って効率よくハウスホルダー変換を行う。

[Function] double gsl_linalg_householder_transform (gsl_vector * v)

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零にするハウスホルダー変換 $P = I - \tau v v^T$ を設定する。変換を行うためのベクトルが v に入れられ、スカラー値 τ が返される。

[Function] int gsl_linalg_householder_hm (double tau, const gsl_vector * v, gsl_matrix * A)

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。

[Function] int gsl_linalg_householder_mh (double tau, const gsl_vector * v, gsl_matrix * A)

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。

[Function] int gsl_linalg_householder_hv (double tau, const gsl_vector * v, gsl_vector * w)

スカラー値 tau とベクトル v およびベクトル w で定義されるハウスホルダー行列 P を行う。変換の結果 Pw で引数 w を上書きする。

13.10 ハウスホルダー変換による線形問題の解法

[Function] int gsl_linalg_HH_solve (gsl_matrix * A, const gsl_vector * b, gsl_vector * x)

ハウスホルダー変換を使って c 直接線形問題 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。

[Function] int gsl_linalg_HH_svx (gsl_matrix * A, gsl_vector * x)

ハウスホルダー変換を使って c 直接線形問題 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。

13.11 三重対角問題

[Function] int gsl_linalg_solve_tridiag (const gsl_vector * diag, const gsl_vector * e, const gsl_vector * f, const gsl_vector * b, gsl_vector * x)

A が N × N の三重対角行列 (N ≥ 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、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になる。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ 0 & 0 & f_2 & d_3 \end{pmatrix}$$

[Function] int gsl_linalg_solve_symm_tridiag (const gsl_vector * diag, const gsl_vector * e, const gsl_vector * b, gsl_vector * x)

A が N × N の対称三重対角行列 (N ≥ 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。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ 0 & 0 & e_2 & d_3 \end{pmatrix}$$

[Function] 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)

A が N × N の巡回三重対角行列 (N ≥ 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。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & f_3 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ e_3 & 0 & f_2 & d_3 \end{pmatrix}$$

[Function] int gsl_linalg_solve_symm_cyc_tridiag (const gsl_vector * diag, const gsl_vector * e, const gsl_vector * b, gsl_vector * x)

A が N × N の対称巡回三重対角行列 (N ≥ 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。

$$A = \begin{pmatrix} d_0 & e_0 & 0 & e_3 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ e_3 & 0 & e_2 & d_3 \end{pmatrix}$$

13.12 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);

    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
```

解の正しさは、GNU octave を使って解 x と元の行列 A の積を計算することで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
```

元の式 Ax = b に従って、これにより右辺の b が得られる。

13.13 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, 1996), Johns Hopkins University Press, ISBN 0-8018-5414-8.

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 (Third Edition, 1999), Published by SIAM, ISBN 0-89871-447-8. 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。

修正ゴルブ- ラインシュ法については以下の論文に述べられている。

- T.F. Chan, "An Improved Algorithm for Computing the Singular Value Decomposition", ACM Transactions on Mathematical Software, 8 (1982), pp 72–83.

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, Volume 18, Number 1 (1973), p 74—76
- James Demmel, Kresimir Veselic, "Jacobi's Method is more accurate than QR", Lapack Working Note 15 (LAWN-15), October 1989. Available from netlib, http://www.netlib.org/lapack/ in the lawns or lawnspdf directories.

第 14 章 固有値問題

この章では行列の固有値と固有ベクトルを計算する関数について説明する。GSL には実数対称行列と複素ハミルトン行列に対する固有値計算関数があり、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。ここでは対称二重対角化に続いて QR 分解を行うアルゴリズムを使っている。

このライブラリで用意しているルーチンは、簡単なアルゴリズムでも支障のでない「小さな」系を想定している。大きな行列に対する固有値と固有ベクトルの計算を行いたいなら、LAPACK にある洗練されたルーチンを使うべきである。線形代数の標準パッケージとして LAPACK の FORTRAN 版が推奨される。

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

14.1 実数対称行列

[Function] gsl_eigen_symm_workspace * gsl_eigen_symm_alloc (const size_t n)

n × 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。

[Function] void gsl_eigen_symm_free (gsl_eigen_symm_workspace * w)

作業領域 w が確保しているメモリを解放する。

[Function] int gsl_eigen_symm (gsl_matrix * A, gsl_vector * eval, gsl_eigen_symm_workspace * w)

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。A の対角成分および下三角成分は計算処理で値が変えられてしまうが、上三角成分は参照されない。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。

[Function] gsl_eigen_symmv_workspace * gsl_eigen_symmv_alloc (const size_t n)

n × 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。

[Function] void gsl_eigen_symmv_free (gsl_eigen_symmv_workspace * w)

作業領域 w が確保しているメモリを解放する。

[Function] int gsl_eigen_symmv (gsl_matrix * A, gsl_vector * eval, gsl_matrix * evec, gsl_eigen_symmv_workspace * w)

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。A の対角成分および下三角成分は計算処理で値が変えられてしまうが、上三角成分は参照されない。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 複素ハミルトン行列

[Function] gsl_eigen_herm_workspace * gsl_eigen_herm_alloc (const size_t n)

n × n の複素ハミルトン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。

[Function] void gsl_eigen_herm_free (gsl_eigen_herm_workspace * w)

作業領域 w が確保しているメモリを解放する。

[Function] int gsl_eigen_herm (gsl_matrix_complex * A, gsl_vector * eval, gsl_eigen_herm_workspace * w)

複素ハミルトン行列 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。A の対角成分および下三角成分は計算処理で値が変えられてしまうが、上三角成分は参照されない。対角成分の虚部は零であると見なされ、その値は無視される。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。

[Function] gsl_eigen_hermv_workspace * gsl_eigen_hermv_alloc (const size_t n)

n × n の複素ハミルトン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。

[Function] void gsl_eigen_hermv_free (gsl_eigen_hermv_workspace * w)

作業領域 w が確保しているメモリを解放する。

[Function] int gsl_eigen_hermv (gsl_matrix_complex * A, gsl_vector * eval, gsl_matrix_complex * evec, gsl_eigen_hermv_workspace * w)

複素ハミルトン行列 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。A の対角成分および下三角成分は計算処理で値が変えられてしまうが、上三角成分は参照されない。対角成分の虚部は零であると見なされ、その値は無視される。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 固有値と固有ベクトルの整列

[Function] int gsl_eigen_symmv_sort (gsl_vector * eval, gsl_matrix * evec, gsl_eigen_sort_t sort_type)

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 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`	大きさの昇順に整列
`GSL_EIGEN_SORT_ABS_DES`	大きさの降順に整列

[Function] int gsl_eigen_hermv_sort (gsl_vector * eval, gsl_matrix_complex * evec, gsl_eigen_sort_t sort_type)

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。

14.4 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例示する。

```
#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");
    }

    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 で以下のようにすると、結果を比較することができる。

```
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。

14.5 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, 1996), Johns Hopkins University Press, ISBN 0-8018-5414-8.

LAPACK については以下に述べられている。

・ LAPACK Users' Guide (Third Edition, 1999), Published by SIAM, ISBN 0-89871-447-8. 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。このライブラリでは基数 2 の FFT(データ長が 2 の累乗)と混合基数 FFT (任意のデータ長)の関数が実装されている。実行速度の向上のため各関数には実数版と複素数版がある。混合基数関数はパウル・シュヴァルツラウバーの FFTPACK ライブラリを実装し直したものである。FFTPACK の ORTRAN 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 に付属する文書 GSL FFT Algorithms を参照のこと(15.8 節「参考文献」参照)。

15.1 数学的定義

高速フーリエ変換は、効率よく離散フーリエ変換（DFT）を行う計算法である。

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi i j k / N)$$

DFT は一般的に、時間上または空間内の離散的な点での連続フーリエ変換を近似するものである。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 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 $f_1 f_2 \ldots f_n$ で表されるとき、DFT の計算量は $O(N \Sigma f_i)$ である。基数 2 の FFT ではこれは $O(N \log_2 N)$ になる。

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。どれも数学的な定義は同じである。順方向フーリエ変換 x = FFT(z) の定義は以下である。

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi i j k / N)$$

またフーリエ逆変換 x = IFFT(z) 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 1/N の約数により、ちゃんと逆変換になる。`gsl_fft_complex_forward` に続いて `gsl_fft_complex_inverse` を呼び出すと、元のデータが得られる（計算誤差は含まれるが)。

順変換、逆変換を組み合わせるとき、指数関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」ではこれとは逆では、順方向で正の符号になっている。

ここでは、以下の定義による逆変換 FFT のスケーリングを行わないものを逆方向 FFT と呼ぶことにしている。

$$z_j^{backwards} = \sum_{k=0}^{N-1} x_k \exp(2\pi i j k / N)$$

変換後のスケール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

方が高速である。

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 packed array である。packed array 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 packed array 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。

stride パラメータを使うことで、`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。行列に対する列指向の FFT の場合に飛び幅 stride を 1 以上にすることができる。飛び幅が 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。

`gsl_complex_vector * 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（もしくは以下と同等な）定義を使って、この章に後述する関数を使う。

```
gsl_complex_packed_array data = v->data;
size_t stride = v->stride;
size_t n = v->size;
```

現実例に応用する場合、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/(2Δ) から +1/(2Δ) 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。

配列 data と時間領域の値 z、周波数領域の値 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 の位置には周波数の最大値（正の値 +1/(2Δ) と負の値 −1/(2Δ) は等しい）が入れられる。N が負の時は上の表の通りだが、N/2 に対応する要素はない。

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 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 Cooley-Tukey のアルゴリズムを使って 2 の累乗個のデータに対し複素数置換 FFT を行う。置換法であるため、別途の作業領域を必要としない。これに対応する自己整列混合基数法は別の作業領域を使って高速に計算を行う。

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。

[Function] 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)

長さ n で飛び幅が 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 の累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 の累乗でないときには GSL_EDOM を返す。

[Function] 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)

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例示する。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. . . 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。

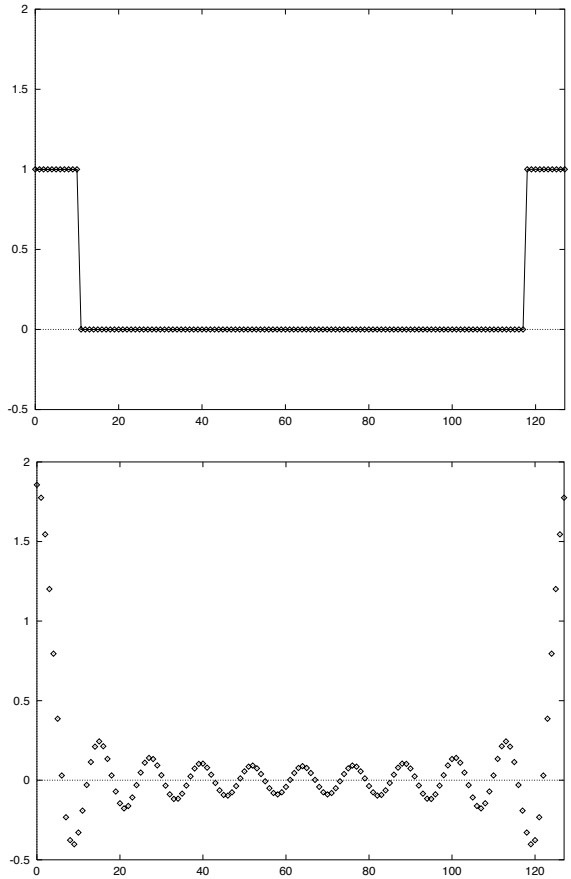
```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

#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) = 0.0; IMAG(data,i) = 0.0;

REAL(data,0) = 1.0;
```

```
        for (i = 1; i <= 10; i++)
             REAL(data,i) = REAL(data,128-i) = 1.0;

        for (i = 0; i < 128; i++)
             printf("%d %e %e\n", i, REAL(data,i), IMAG(data,i));

        printf("\n");
        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、関数の返り値 `gsl_fft_complex_radix2_forward` をチェックするべきである。

変換されたデータは 1/√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零なので、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 は変換した sin 関数で連続フーリエ変換を近似していることになる。

$$\int_{-a}^{+a} e^{-2\pi i k x} dx = \frac{\sin(2\pi k a)}{\pi k}$$



元のパルス波と、例示したプログラムによるその離散フーリエ変換。

15.4 複素数に対する混合基数 FFT

以下では複素数に対する混合基数 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。このライブラリで用意している関数はパウル・シュヴァルツライバー 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。理論的根拠はクライブ・テンパートン Clive Temperton のレビュー記事 Self-sorting Mixed-radix FFTs（自己整列混合基数 FFT）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。

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。因数 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、2×3 と組み合わせた 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 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、特定の因数に対する方法よりも遅い。一般的なデータ長 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 = 2×3×99991 のような大きな素数が出現する場合には効率が上げられない。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（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)。

混合基数法の初期化関数 `gsl_fft_complex_wavetable_alloc` は、与えられるデータ長 N に対してライブラリ側で自動的に決定する因数のリストを返す。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。まず最初に見積もるときは、実行時間は $N\Sigma f_i$ の倍数になると考えるとよい。f_i は N の約数である。プログラムの実行を利用者が操作するためには、因数分解があまりうまくいかないときに警告を出すようにするとよい。実際にこのライブラリを使っていて、用意されている因数では分解できないような事が頻繁に起こる場合、GSL FFT Algorithms を参照して他の因数での変換ルーチンを用意することができる。

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。

[Function] gsl_fft_complex_wavetable * gsl_fft_complex_wavetable_alloc (size_t n)

> 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型 `gsl_fft_complex_wavetable` 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 は副変換のために因数分解され、因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しか行われない。この省略によって 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。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。

[Function] void gsl_fft_complex_wavetable_free (gsl_fft_complex_wavetable * wavetable)

> 三角関数の表 wavetable のインスタンスを消去、メモリを解放する。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。

三角関数表の構造体はヘッダファイル 'gsl_fft_complex.h' で宣言されている。

[Data Type] gsl_fft_complex_wavetable

この構造体は混合基数 FFT での因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　　複素数でのデータ点数

size_t nf　　データ長 n を因数分解した後の因数の個数

size_t factor[64]　　因数を保持する配列。最初の nf 個のみが使われる。

gsl_complex * trig　　あらかじめ確保する 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、各計算で使われる因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が必要である。

[Function] gsl_fft_complex_workspace * gsl_fft_complex_workspace alloc (size_t n)

データ長 n の複素数 FFT で使う作業領域を確保する。

[Function] void gsl_fft_complex_workspace_free (gsl_fft_complex_workspace * workspace)

作業領域 workspace に割り当てられているメモリを解放する。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。

[Function] 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)

複素数配列 data で与えられるデータに対して、データ長 n で飛び幅 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が内部に用意されている。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。これらの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 が零など）。

`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（= 2×3×3×5×7）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) = 0.0;
        IMAG(data,i) = 0.0;
    }

    data[0] = 1.0;

    for (i = 1; i <= 10; i++)
        REAL(data,i) = REAL(data,n-i) = 1.0;

    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実数になるとは限らない。特殊な対称性を持つ以下の

ような複素数列になる。

$$z_k = z_{N-k}^*$$

このような対称性を持つ数列を複素共役または半複素と呼ぶ。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 `gsl_fft_halfcomplex` で半複素数列を変換する。

`gsl_fft_real` の関数は実数列に対する周波数係数を計算する。実数列 x に対する半複素数係数 c は以下のフーリエ解析で与えられる。

$$c_k = \sum_{j=0}^{N-1} x_j \exp(-2\pi i j k/N)$$

`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）。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より、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。

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。

[Function] int gsl_fft_real_radix2_transform (double data [], size_t stride, size_t n)

与えられるデータ data に対してデータ長 n で飛び幅 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、虚部は零なのでどこにも置かれない。

変換結果 data と、実質的には同じことになる虚部がすべて 0 のデータを変換した結果の対応を以下の表に示す。

```
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]
```

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。

[Function] 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)

与える半複素数列 data に対してデータ長 n、飛び幅 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、そのままの順序で入れられる。

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・シュヴァルツライバー 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。理論的根拠はクライブ・テンパートン Clive Temperton の記事 Fast Mixed-Radix Real Fourier Transforms (高速混合基数 FFT)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。虚部が零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零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。これは変換結果の対称性 $z_k = z_{N-k}^*$ から、虚部が零の単なる実数になることが分かるからである。

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 の基数の場合の例である。二つの列はそれぞれ、gsl_fft_real_transform が返す 5 個の要素からなる型 halfcomplex [] の半複素数列と、同じ実数列を gsl_fft_complex_backward に、虚部が零の半複素数列として与えたときに得られる実数列である。

```
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[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埋められている。周波数 0 にあたる項 complex[0].imag の虚部はその対称性から零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、二つの項で虚部が零になることが知られている。

```
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埋められている。`complex[0].imag` と `complex[3].imag` の値は零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。

[Function] 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)

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、その因数と因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しか行われない。この省略によって 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。

[Function] void gsl_fft_real_wavetable_free (gsl_fft_real_wavetable * wavetable)

[Function] void gsl_fft_halfcomplex_wavetable_free (gsl_fft_halfcomplex_wavetable * wavetable)

三角関数表 wavetable に割り当てられているメモリを解放する。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が必要である。

[Function] gsl_fft_real_workspace * gsl_fft_real_workspace_alloc (size_t n)

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。

[Function] void gsl_fft_real_workspace_free (gsl_fft_real_workspace * workspace)

作業領域 workspace に割り当てられたメモリを解放する。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。

[Function] 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)

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 は上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。

[Function] int gsl_fft_real_unpack (const double real_coefficient [], gsl_complex_packed_array complex_coefficient [], size_t stride, size_t n)

`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（虚部が零の複素数）の配列 complex_coefficient に変換する。変換は以下のように単純に行われる。

```
for (i = 0; i < n; i++) {
    complex_coefficient[i].real = real_coefficient[i];
    complex_coefficient[i].imag = 0.0;
}
```

[Function] int gsl_fft_halfcomplex_unpack (const double halfcomplex_coefficient [], gsl_complex_packed_array complex_coefficient, size_t stride, size_t n)

gsl_fft_real_transform 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 $z_k = z_{N-k}^*$ を使って冗長的な要素を計算して埋める。

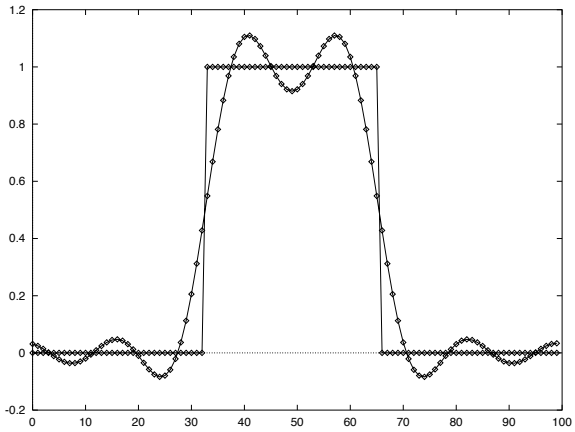
```
complex_coefficient[0].real = halfcomplex_coefficient[0];
complex_coefficient[0].imag = 0.0;
for (i = 1; i < n - i; i++) {
    double hc_real= halfcomplex_coefficient[2 * i - 1];
    double hc_imag = halfcomplex_coefficient[2 * i];
    complex_coefficient[i].real = hc_real;
    complex_coefficient[i].imag = 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;
}
```



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、ドゥハメルとヴェターリによる以下のレビュー記事から見るとよい。

- P. Duhamel and M. Vetterli. Fast fourier transforms: A tutorial review and a state of the art. Signal Processing, 19:259–299, 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。この文書の他にもよい文献はいくつもあるので、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):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):849–863, 1987.
- Clive Temperton. Fast mixed-radix real fourier transforms. Journal of Computational Physics, 52: 340–350, 1983.

1979 年に IEEE は、慎重に書かれた FFT の FORTRAN プログラムの大綱を発表した。いろいろな FFT アルゴリズムを実装する際に参考になる。

- Digital Signal Processing Committee and IEEE Acoustics, Speech, and Signal Processing Committee, editors. Programs for Digital Signal Processing. 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 Frigo とジョンソン 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実行速度に関する自己最適化を行うもので、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/

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）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と非適応型のルーチンを用意しており、別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の領域での積分、特異積分、対数特異点を含む積分、コーシーの主値積分、振動型積分の計算などがある。GSL で実装しているのはピセンズ Piessens、ドンカー - カペンガ Doncker-Kapenga、ユーバーフーバー Uberhuber、カハナー 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 のソースコードが 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（一般的な被積分関数では w(x) = 1）。利用者は以下のような要求精度を許容絶対誤差と許容相対誤差 (epsabs, epsrel) で指定する。

$$|\text{RESULT} - I| \leq \max(\text{epsabs}, \text{epsrel}|I|)$$

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。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。

$$|\text{RESULT} - I| \leq \text{ABSERR} \leq \max(\text{epsabs}, \text{epsrel}|I|)$$

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、常に最新の最良近似値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

`N` - 非適応型積分法
`A` - 適応型積分法

`G` - 汎用 (被積分関数を利用者が指定する)
`W` - 重み関数と被積分関数の積を使う

`S` - 特異点を持つ関数を高速に積分する
`P` - 積分が困難ないくつかのケースに対応している
`I` - 無限区間での積分
`O` - 振動する重み関数 (三角関数 sin または cos)
`F` - フーリエ積分
`C` - コーシーの主値

実装しているアルゴリズムでは高次と低次で異なる求積法を使うようになっている。高次の方法では限られた小さな範囲内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、その近

似誤差を見積もることができる。

16.1.1 重み関数のない被積分関数の場合

GSL での汎用の求積法（重み関数を使わない）は、ガウス・クロンロッド Gauss-Kuronrod の求積法である。

ガウス・クロンロッド法は最初に m 次の古典的なガウスの求積法を行う。これに対して横軸上の複数の点から 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するため、効率がよい。高次のクロンロッド法は積分値の近似をよくするために使われ、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 重み関数のある被積分関数の場合

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 - カーティス Clenshaw-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よくすることができる。

16.1.3 特異重み関数のある被積分関数の場合

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 で使われている修正クレンショー・カーティス法では、収束を遅くするような重み関数のうち共通するものを分離して積分を行う。これらの重み関数は修正チェビシェフ・モーメントをあらかじめ計算するために、そのチェビシェフ多項式を使って解析的に積分される。修正チェビシェフ・モーメントを使ってチェビシェフ多項式と重み関数をまとめることで、目的の積分を得る。特異点を持つ重み関数を解析的に積分することにより正確に相殺され、積分値の収束を改善することができる。

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・クロンロッド法である。滑らかな関数を高速に積分するための方法である。

[Function] int gsl_integration_qng (const gsl_function *f, double a, double b, double epsabs, double epsrel, double * result, double * abserr, size_t * neval)

> 10 点、21 点、43 点、87 点のガウス・クロンロッド法を、関数 f の区間 (a, b) での積分値の推定誤差が利用者が与え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順番に適用する。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。ガウス・クロンロッド法では、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、利用するようになっている。

16.3 QAG 法: 適応型積分

QAG 法では単純な適応型積分計算を行う。積分範囲を分割し、繰り返し計算の各回で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。積分の難しい場所を特定してそこに計算量を集中させることで、全体での誤差を急激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。

[Function] gsl_integration_workspace * gsl_integration_workspace_alloc (size_t n)

> 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。

[Function] void gsl_integration_workspace_free (gsl_integration_workspace * w)

作業領域 w のメモリを解放する。

[Function] 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)

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利用者が与え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、以下の示す 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、滑らかな関数に対しては高精度に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の積分が難しい点を含む場合に、計算時間を短縮するために用いる。

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そうな、区間の個数の上限を limit で与える。

16.4 QAGS 法: 特異値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を含む小区間を作ってそこに計算を集中する。区間の幅は小さくなっていくため、積分値の近似値は極限で収束するように振る舞う。この極限への収束は補外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ワイン Wynn イプシロン・アルゴリズム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。

[Function] 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)

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利用者が指定する値 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そうな、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

[Function] 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)

利用者が指定する特異点を考慮して、適応型積分 QAGS 法を行う。大きさ npts の配列 pts

に積分範囲の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 x1、x2、x3 (a < x1 < x2 < x3 < 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

[Function] int gsl_integration_qagi (gsl_function * f, double epsabs, double epsrel, size_t limit, gsl_integration_workspace * workspace, double *result, double *abserr)

関数 f の区間 (−∞, +∞) での積分値を計算する。積分は、以下のように区間を x = (1 − t)/t で (0, 1] に移した後に、QAGS 法で行われる。

$$\int_{-\infty}^{+\infty} dx f(x) = \int_0^1 dt (f((1-t)/t) + f(-(1-t)/t))/t^2$$

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。

[Function] int gsl_integration_qagiu (gsl_function * f, double a, double epsabs, double epsrel, size_t limit, gsl_integration_workspace * workspace, double *result, double *abserr)

関数 f の半無限区間 (a, +∞) での積分値を計算する。以下のように区間を x = a + (1 − t)/t で (0, 1] に移して、QAGS 法で積分を行う。

$$\int_a^{+\infty} dx f(x) = \int_0^1 dt f(a + (1-t)/t)/t^2$$

[Function] int gsl_integration_qagil (gsl_function * f, double b, double epsabs, double epsrel, size_t limit, gsl_integration_workspace * workspace, double *result, double *abserr)

関数 f の半無限区間 (−∞, b) での積分値を計算する。以下のように区間を x = b − (1 − t)/t で (0, 1] に移して、QAGS 法で積分を行う。

$$\int_{-\infty}^b dx f(x) = \int_0^1 dt f(b - (1-t)/t)/t^2$$

16.7 QAWC 法: コーシーの主値の適応型積分

[Function] 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)

関数 f の区間 (a, b) で特異点が c でのコーシーの主値を計算する。

$$I = \int_a^b dx \frac{f(x)}{x-c} = \lim_{c\to 0} \left\{ \int_a^{c-\epsilon} dx \frac{f(x)}{x-c} + \int_{c+\epsilon}^b dx \frac{f(x)}{x-c} \right\}$$

QAG の適応型二等分法を使うが、分割は特異点 x = c で生じないように行われる。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。

[Function] gsl_integration_qaws_table * gsl_integration_qaws_table_alloc (double alpha, double beta, int mu, int nu)

以下の特異値重み関数 W (x) とそのためのパラメータ (α, β, μ, ν) 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。

$W(x) = (x-a)^{\alpha}(b-x)^{\beta}$ ($\mu = 0, \nu = 0$)

$W(x) = (x-a)^{\alpha}(b-x)^{\beta}\log(x-a)$ ($\mu = 1, \nu = 0$)

$W(x) = (x-a)^{\alpha}(b-x)^{\beta}\log(b-x)$ ($\mu = 0, \nu = 1$)

$W(x) = (x-a)^{\alpha}(b-x)^{\beta}\log(x-a)\mathrm{og}(b-x)$ ($\mu = 1, \nu = 1$)

特異点 (a, b) は境界上の点なので、積分が計算されるまでは必要ない。

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。

[Function] int gsl_integration_qaws_table_set (gsl_integration_qaws_table * t, double alpha, double beta, int mu, int nu)

すでに確保されている `gsl_integration_qaws_table` のインスタンス t が持つパラメータ (α, β, μ, ν) の値を設定する。

[Function] void gsl_integration_qaws_table_free (gsl_integration_qaws_table * t)

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。

[Function] 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)

区間 (a, b) で特異値重み関数が $(x-a)^{\alpha}(b-x)^{\beta}\log^{\mu}(x-a)\log^{\nu}(b-x)$ 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 (α, β, μ, ν) は t に設定しておく。以下の積分が行われる。

$$I = \int_a^b dx f(x)(x-a)^{\alpha}(b-x)^{\beta}\log^{\mu}(x-a)\log^{\nu}(b-x)$$

適応型二等分 QAG 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関数の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。

[Function] gsl_integration_qawo_table * gsl_integration_qawo_table_alloc (double omega, double L, enum gsl_integration_qawo_enum sine, size_t n)

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。

$$W(x) = \left\{ \begin{array}{l} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

パラメータ 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。

[Function] int gsl_integration_qawo_table_set (gsl_integration_qawo_table * t, double omega, double L, enum gsl_integration_qawo_enum sine)

すでに確保されている作業領域が持つパラメータ omega、L、sine の値を設定する。

[Function] int gsl_integration_qawo_table_set_length (gsl_integration_qawo_table * t, double L)

作業領域 t の持つパラメータ L の値を設定し直す。

[Function] void gsl_integration_qawo table free (gsl_integration_qawo_table * t)

作業領域 t のメモリを解放する。

[Function] 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)

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(ωx) または cos(ωx) を使って計算する。

$$I = \int_a^b dx\ f(x) \left\{ \begin{array}{l} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

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そうな、区間の個数の上限を limit で与える。

dω > 4 となる「大きな」区間に対しては 25 点のクレンショー・カーティス法が使われる。dω < 4 となる「小さな」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

[Function] 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)

以下のように表される、半無限区間 [a, + ∞) での関数 f のフーリエ積分を計算する。

$$I = \int_a^{+\infty} dx\ f(x) \left\{ \begin{array}{l} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

パラメータ ω は表 wf に指定しておく（長さ L はフーリエ積分のための値でこの関数が上書きするので、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]

C_2 = [a+c, a+2c]

... = ...

C_k = [a+(k-1)c, a+kc]

ここで c = (2 floor(|ω|) + 1)π/|ω| である。幅 c は小区間の数が奇数になるようにとる。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加速できる。

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。

$$TOL_k = u_k \text{abserr}$$

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。

小区間での積分が困難なときは、その区間での要求精度を以下のように下げる。

$$TOL_k = u_k \max(\text{abserr}, \max_{i<k}\{E_i\})$$

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そうな、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利用者が与える 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

適切でない引数を示す標準で用意されているエラーコードに加え、この章で説明した関数は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 で計算することができる。

```
#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);

    return 0;
}
```

プログラムの出力を以下に示す。積分区間を 8 個で、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
```

実際には、QAGS 法による補外を使うことで、精度の桁数が約 2 倍になる。補外法が返す推定誤差は実際の誤差よりも余裕を持って、1 桁ほど大きい値になっている。

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

このライブラリでは様々な一様乱数の生成法が用意されている。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、乱数発生器の種類や乱数の種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各自のその時の状態をそれぞれ保持しているため、マルチスレッドプログラムでも問題なく使うことができる。一様乱数から、正規分布、対数正規分布、ポアソン分布等の連続あるいは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 年のパク Park とミラー Miller による論文（"Random number generators: good ones are hard to find.", Commun. ACM, 31, 1192–1201）によると、優れた乱数発生器がすでにいくつもあるのにかかわらず、よくないものが未だ広く使われている。計算機に付属している乱数発生器でもいいかもしれないが、計算機の速度向上に従って、乱数発生器に要求されることも多くなってきていることに注意せねばならない。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、ほんの少しコーヒーを片手に休憩している間に終わってしまう。

ピエール・レキエル 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 (参考文献参照) から PostScript 形式で入手することができる。クヌース Knuth の本の準数値計算法 (原著は 1968 年刊) 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、このマニュアルを読んでいるよりも書店に走っていって、買ってから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。シミュレーションに用いるときには統計的な検定を行う。一般的には、なにか厳密な解を与える確率論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」を評価する。

17.2 乱数発生器の呼び出し

乱数発生器関数は、例えば正弦関数や余弦関数のような「実数」関数ではないことに常に留意せねばならない。そういった関数と違って乱数発生器関数が返す値は、処理に成功すればいつも同じ、というわけにはいかない。乱数発生器はまさにそのための関数であるが、これを実現するためには関数内部に発生器の「状態」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か〝(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であったりもする。その場合、添え字を指定することで乱数を返す。このライブラリで用意されている乱数発生器では、状態管理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 が用意している乱数発生器で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、その静的な情報を保持する。gsl_rng では型 gsl_rng_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

[Function] gsl_rng * gsl_rng_alloc (const gsl_rng_type * T)

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タウスワース 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既定の種 gsl_rng_default_seed で初期化される。その既定値は零だが、直接あるいは環境変数 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」参照) 。

利用できる乱数発生器の種類は、この章で後述する。

[Function] void gsl_rng_set (const gsl_rng * r, unsigned long int s)

乱数発生器を初期化、つまり乱数発生器に「種」を与える。この関数を複数回、同じ種 s を与えて呼び出し初期化した場合、それに続く乱数発生器の呼び出しでは同じ乱数系列が生成される。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だろう。s に零を与えると、各種の乱数発生器ごとに実装されている規定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、このライブラリの using gsl_rng_ranlux を使う場合、s に零を与えると零の代わりにこの値が種として使われる。

[Function] void gsl_rng_free (gsl_rng * r)

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。

17.4 乱数生成期を使った乱数の生成

以下に説明する関数は、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」を参照のこと。

[Function] unsigned long int gsl_rng_get (const gsl_rng * r)

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)を使って設定できる。

[Function] double gsl_rng_uniform (const gsl_rng * r)

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。返される値は、関数 gsl_rng_get(r) が返す値を gsl_rng_max(r) + 1.0 で除す計算を倍精度で行った値であることが多い。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整数 unsigned long int で表現されているので、移植性は高い)。

[Function] double gsl_rng_uniform_pos (const gsl_rng * r)

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いたいときに有用である。

[Function] unsigned long int gsl_rng_uniform_int (const gsl_rng * r, unsigned long int n)

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よって乱数の最小値が違うため、零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する乱数の取りうる範囲よりも狭い範囲で乱数を発生するように設計されている。n は乱数発生器 r の範囲内でなければならない。n が乱数発生器が発生する乱数の最大値よりも大きな値の場合は、エラーハンドラーをエラーコード GSL_EINVAL で呼び出し、零を返す。

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。

[Function] const char * gsl_rng_name (const gsl_rng * r)

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 という出力が得られる。

[Function] unsigned long int gsl_rng_max (const gsl_rng * r)

gsl_rng_get が返す値の最大値を返す。

[Function] unsigned long int gsl_rng_min (const gsl_rng * r)

gsl_rng_get が返す値の最小値を返す。普通はこの値は零になるが、零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。

[Function] void * gsl_rng_state (const gsl_rng * r)

[Function] size_t gsl_rng_size (const gsl_rng * r)

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。これを使うことで、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);
```

[Function] const gsl_rng_type ** gsl_rng_types_setup (void)

利用できる乱数発生器のすべての型を持つ配列へのポインタを返す。必要に応じて、プログラムの実行時、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

使用する乱数発生アルゴリズムと種の規定値は、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。

[Function] const gsl_rng_type * gsl_rng_env_setup (void)

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。これらは大域変数で、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。種の値は、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 の初期値は零である。

以下に環境変数 GSL_RNG_TYPE と GSL_RNG_SEED を使って大域的に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; /* global generator */

int main (void)
{
    const gsl_rng_type * T;

    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc(T);

    printf("generator type: %s\n", gsl_rng_name (r));
    printf("seed = %lu\n", gsl_rng_default_seed);
    printf("first value = %lu\n", gsl_rng_get (r));

    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乱数の「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もあるので、そのための関数も用意されている。

[Function] int gsl_rng_memcpy (gsl_rng * dest, const gsl_rng * src)

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。

[Function] gsl_rng * gsl_rng_clone (const gsl_rng * r)

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。

[Function] int gsl_rng_fwrite (FILE * stream, const gsl_rng * r)

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。

[Function] int gsl_rng_fread (FILE * stream, gsl_rng * r)

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、乱数発生器 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仮定される。

17.9 乱数発生アルゴリズム

これまでに説明したでは、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アルゴリズムを切り替えられることを説明するためである。このライブラリには、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に用意されているもの、昔からある古典的なものなど、様々な乱数発生器がある。

以下に説明する乱数発生器は、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、多くの統計的検定で合格できる。

[Generator] gsl_rng_mt19937

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いてる捩れ汎型フィードバックレジスタ移動（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 $2^{19937} - 1$ (約 10^{6000}) であり、623 次元空間の各次元で同様に分布する。統計的検定 DIEHARD に合格できる。一つの乱数発生器について 623 ワードで状態を保持し、他の乱数発生器に比べて高速である。彼らの元々の乱数発生器では種は 4357 になっており、s を零にして呼び出すと gsl_rng_set はこの値を使う。

詳細は以下を参照のこと。

Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1 (Jan. 1998), Pages 3-30

乱数発生器 gsl_rng_19937 は種の生成について、同じ著者が発表した第二版（2002 年）の方法を使う。元の方法では種の生成時に、種として与える値によっては間違った挙動を示すことがあるためである。`gsl_rng_mt19937_1999` と `gsl_rng_mt19937_1998` としてその元の方法も利用できる。

[Generator] gsl_rng_ranlxs0

[Generator] gsl_rng_ranlxs1

[Generator] gsl_rng_ranlxs2

乱数発生器 ranlxs0 は、リュシャー Lüscher による ranlux（"luxury random numbers"）の第二世代のアルゴリズムである。この乱数発生器は三段階の「贅沢さ」レベル ranlxs0、ranlxs1、ranlxs2 を持ち、単精度（24 ビット）で乱数を出力する。内部では倍精度で演算を行っており、特に 64 ビット CPU 上では整数での演算に比べ非常に速い。乱数の周期はおおよそ 10171 である。このアルゴリズムは数学的に各種の性質が証明されており、現在知られているレベルでの無作為性において、真に相関を持たない乱数を発生する。贅沢レベルを上げると、安全マージンを増やすことで相関を小さくする。

[Generator] gsl_rng_ranlxd1

[Generator] gsl_rng_ranlxd2

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。

[Generator] gsl_rn_ranlux

[Generator] gsl_rng_ranlux389

リュシャーの元のアルゴリズムによる乱数発生器が ranlux 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。内部では整数を使った演算を行っており、上述した浮動小数点を使う第二世代の ranlxs と ranlxd の方が多くの場合は高速である。乱数の周期は約 10171 である。このアルゴリズムは数学的に各種の性質が証明されており、現在知られているレベルでの無作為性において全く相関を持たない乱数を発生する。リュシャーによる無相関レベルの既定値が `gsl_rng_ranlux` で、最高レベルが `gsl_rng_ranlux389` で参照でき、最高レベルでは 24 ビットで相関をなくすことができる。どちらの型の乱数発生器も 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 (1994) 100-110.

F. James, "RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of L..uscher", Computer Physics Communications, 79 (1994) 111-114.

[Generator] gsl_rng_cmrg

レキュエルの重再起結合乱数発生器（combined multiple recursive generator）である。以下の漸化式で乱数を生成する。

$$z_n = (x_n - y_n) \bmod m1$$

ここに用いた二つの乱数発生器 x_nと y_nは以下である。

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \bmod m_1$$

$$y_n = (a_1 y_{n-1} + a_2 y_{n-2} + a_3 y_{n-3}) \bmod m_2$$

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 2^{185} (約 10^{61}) 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 (1996), 816–822.*

[Generator] gsl_rng_mrg

レキュエル L'Ecuyer、ブルイン Blouin、クトレ Coutre による 5 次の重再起結合乱数発生器である。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 1046 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, and R. Coutre, "A search for good multiple recursive random number generators", ACM Transactions on Modeling and Computer Simulation 3, 87-98 (1993).*

[Generator] gsl_rng_taus

[Generator] gsl_rng_taus2

レキュエルによる、等分散性を最大にしたタウスワース型乱数発生器である。乱数は以下の漸化式で発生される。

$$xn = (s_n^1 \oplus s_n^2 \oplus s_n^3)$$

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 232 を法として計算する。上の式で⊕ は「排他的論理和」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 (1996), 203–213.

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 (1999), 261–269.

乱数発生器としては `gsl_rng_taus` よりも `gsl_rng_taus2` を用いる方がよい。

[Generator] gsl_rng_gfsr4

乱数発生器 gfsr4 は、ずれフィボナッチ法に似ており、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 Ziff の文献によると、「よく知られている」二点法 (R250 など、文献参照) は、その定義から 3 点相関が見られるという深刻な欠点がある。汎化フィードバック移動法 (GFSR, Four-tap Generalized Feedback Shift Register) では数学的によい性質を保つことができ、その四点式では、うまく移動量を選べば、非常に無作為性が高いことが著者の検定で示された、としている。

このライブラリでは、ジフの文献にある例 p392 の値を使っている。そこでは、A = 471、B = 1586、C = 6988、D = 9689 である。

移動量を適切に選ぶことて゛(このライブラリではそうしている)、乱数の周期を最大にすることができる。最大の移動量を D とすると、最大周期は $2^D - 1$ になる（配列 `ra[]`ではすべての要素が零になることは避けられるため、最大周期は 2^D よりも 1 だけ小さくなる)。この実装では D = 9689 であり、周期は約 102917 である。

このライブラリの実装では、32 ビット整数を使って、1 ビットの乱数発生器を 32 個同時に実行するようになっている。つまり、32 ビット乱数発生器の周期は 1 ビット乱数発生器の周期と同じである。また 32 ビットのパターンはすべて当確率であり、従って 0 も乱数値として発生しうる（GFSR 型乱数発生器についてのこの性質は、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), Jul/Aug 1998, pp 385-392.

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。従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応用はできない。そういった統計的な利用法でなく、プログラム中で何か動作に変化を出したいときなどには便利である。

[Generator] gsl_rng_rand

BSD`rand()`乱数発生器。乱数を以下の漸化式で生成する。

$$x_{n+1} = (ax_n + c) \bmod m$$

ここで a = 1103515245、c = 12345 および 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。

[Generator] gsl_rng_random_bsd

[Generator] gsl_rng_random_libc5

[Generator] gsl_rng_random_glibc2

元々 BSD で使われていた線形フィードバック移動式の乱数発生器を実装した、`random()` 関連の関数群である。現在、`random()`にはいくつかの版、元の BSD 版 (たとえば SunOS4 など)、libc5 版 (GNU/Linux のものなど)、glibc 版が利用できる。各版では種の生成法が異なるため、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。

[Generator] gsl_rng_rand48

Unix の `rand48` 乱数発生器。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 および 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` に相当する関数は用意されていないが、返り値の型を `long int` に変換することで `mrand48` の出力を得られる。関数 `gsl_rng_uniform` は内部で使っている 48 ビットから倍精度実数値 x_n/m を返し、`drand48` と同じ動作を行う。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る（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

ここでは、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することができる。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を使っており、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 の累乗 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しかない。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。

[Generator] gsl_rng_ranf

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 および $m = 2^{48}$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。この漸化式では、種の 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対がある。

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 ではここでの種の処理の実装に細かい工夫をしている。初期状態は、逆剰余 a mod 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 の元の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、乱数を実 x_n/m で返す。

この乱数の周期は 2^{46}である。

[Generator] gsl_rng_ranmar

マルサグリア Marsaglia、ザマン Zaman、ツアン Tsang による遅れフィボナッチ型の乱数発生器 `RANMAR` 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。

[Generator] gsl_rng_r250

カークパトリック Kirkpatrick とストール Stoll による移動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 ビットのワードについての「排他的論理和」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, 517-526 (1981)*

[Generator] gsl_rng_tt800

初期の捩り汎化フィードバック移動型の乱数発生器で、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, Vol. 4, No. 3, 1994, pages 254-266.*

[Generator] gsl_rng_vax

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。

$$x_{n+1} = (ax_n + c) \mod m$$

ここで、a = 69069、c = 1 および $m = 2^{32}$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。

[Generator] gsl_rng_transputer

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。

$$x_{n+1} = (ax_n) \mod m$$

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。

[Generator] gsl_rng_randu

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。

$$x_{n+1} = (ax_n) \mod m$$

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。この乱数発生器は、低品質の乱数の見本である。

[Generator] gsl_rng_minstd

パク Park とミラー Miller による「最小標準」minstd 乱数発生器である。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。

$$x_{n+1} = (ax_n) \mod m$$

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 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`）に実装されており別名 "GGL" とも呼ばれている（なぜそう呼ばれているのかはわからない）。

詳細については、以下を参照のこと。

> *Park and Miller, "Random Number Generators: Good ones are hard to find", Communications of the ACM, October 1988, Volume 31, No 10, pages 1192-1201.*

[Generator] gsl_rng_uni

[Generator] gsl_rng_uni32

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。

[Generator] gsl_rng_slatec

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。

[Generator] gsl_rng_zuf

Peterson による ZUFALL 遅れフィボナッチ型の乱数発生器である。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. Petersen, "Lagged Fibonacci Random Number Generators for the NEC SX-3", International Journal of High Speed Computing (1994).

[Generator] gsl_rng_borosh13

ボロシュ Borosh- ニーダーライター Niederreiter 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.、106-108 ページによる。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 = 1812433253、m = 2^{32}である。種の値は最初に発生する乱数の値 x_1になる。

[Generator] gsl_rng_coveyou

カビュー Coveyou 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.、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。

[Generator] gsl_rng_fishman18

フィシュマン Fishman とムーア III 世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.、106-108 ページによる。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 = 62089911、m = 2^{31}− 1 である。種の値は最初に発生する乱数の値 x_1になる。

[Generator] gsl_rng_fishman20

フィシュマン 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.、108 ページによる。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 = 48271、m = 2^{31}−1 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。

[Generator] gsl_rng_fishman2x

レキュエル L'Ecuyer とフィシュマン 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.、108 ページによる。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の値はそれぞれ `fishman20` と `lecuyer21` のアルゴリズムから得られる。種の値は最初に発生する乱数の値 x_1に使われる。

[Generator] gsl_rng_knuthran2

クヌースの Seminumerical Algorithms, 3rd Ed.、108 ページによる二次の重回帰乱数発生器である。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。

[Generator] gsl_rng_knuthran

クヌースの Seminumerical Algorithms, 3rd Ed.、3.6 節による二次の重回帰乱数発生器である。著者による C 言語ソースプログラムがある。

[Generator] gsl_rng_lecuyer21

レキュエル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.、106-108 ページによる。以下の漸化式で乱数を発生する。

$$x_{n+1} = (a x_n) \bmod m$$

ここで $a = 40692$、$m = 2^{31}-249$ である。種の値は最初に発生する乱数の値 x_1になる。

[Generator] gsl_rng_waterman14

ウォーターマン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.、106-108 ページ による。以下の漸化式で乱数を発生する。

$$x_{n+1} = (a x_n) \bmod m$$

ここで $a = 1566083941$、$m = 2^{32}$である。種の値は最初に発生する乱数の値 x_1になる。

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 で切り替えることができる。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, 1997), Addison-Wesley, ISBN 0201896842.

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, 1998, 93–137.
- http://www.iro.umontreal.ca/~lecuyer/papers.html in the file 'handsim.ps'.

Web 上では pLab ホームページ (http://random.mat.sbg.ac.at/) に、乱数発生器の開発についての記述や、他の多くの乱数のサイトへのリンクがある。

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。

- DIEHARD source code G. Marsaglia, http://stat.fsu.edu/pub/diehard/

乱数の検定法は、NIST に包括的に用意されている。

- NIST Special Publication 800-22, "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications".
- http://csrc.nist.gov/rng/

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、Yoshiharu Kurita 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。

第 18 章 疑似乱数列

この章では、任意次元の疑似乱数系列を発生する関数について説明する。疑似乱数系列は、d 次元空間に一様に分布する点を次々に発生していく。疑似乱数系列発生器の使い方は、乱数発生器と同じである。

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 疑似乱数発生器の初期化

[Function] gsl_qrng * gsl_qrng_alloc (const gsl_qrng_type * T, unsigned int d)

指定された型 T を使う d 次元の疑似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_ENOMEM でエラーハンドラーを呼び出す。

[Function] void gsl_qrng_free (gsl_qrng * q)

疑似乱数発生器のインスタンス q が持つメモリを解放する。

[Function] void gsl_qrng_init (gsl_qrng * q)

すでに生成されている疑似乱数発生器のインスタンス q を、再初期化する。

18.2 疑似乱数系列の発生

[Function] int gsl_qrng_get (const gsl_qrng * q, double x [])

疑似乱数発生器のインスタンス q で、それまでに発生した疑似乱数の次の疑似乱数を発生して配列 x に返す。x の次元と疑似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 x_iについて、$0 < x_i < 1$ となる。

18.3 疑似乱数系列に関連する関数

[Function] const char * gsl_qrng_name (const gsl_qrng * q)

疑似乱数発生器の型の名前文字列へのポインタを返す。

[Function] size_t gsl_qrng_size (const gsl_qrng * q)

[Function] void * gsl_qrng_state (const gsl_qrng * q)

疑似乱数発生器のインスタンス r の状態とその大きさへのポインタを返す。疑似乱数発生器の状態を直接参照、操作するのにこの関数を使う。たとえば以下のコードは、疑似乱数発生器の状態を出力する。

```
void * state = gsl_qrng_state(q);
size_t n = gsl_qrng_size(q);
fwrite(state, n, 1, stream);
```

18.4 疑似乱数発生器の状態の保存と読み出し

[Function] int gsl_qrng_memcpy (gsl_qrng * dest, const gsl_qrng * src)

疑似乱数発生器 src を、すでに確保した疑似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。

[Function] gsl_qrng * gsl_qrng_clone (const gsl_qrng * q)

疑似乱数発生器 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 疑似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。

[Generator] gsl_qrng_niederreiter2

ブラトリー Bratley、フォックス Fox、ニーダーライター Niederreiter の ACM Trans. Model. Comp. Sim. 2, 195 (1992) によるアルゴリズム。最大 12 次元までの点を発生することができる。

[Generator] gsl_qrng_sobol

アントノフ Antonov、サレーブ Saleev による USSR Comput. Maths. Math. Phys. 19, 252 (1980) に述べられている、ソボル Sobol の方法。これは 40 次元まで発生できる。

18.6 例

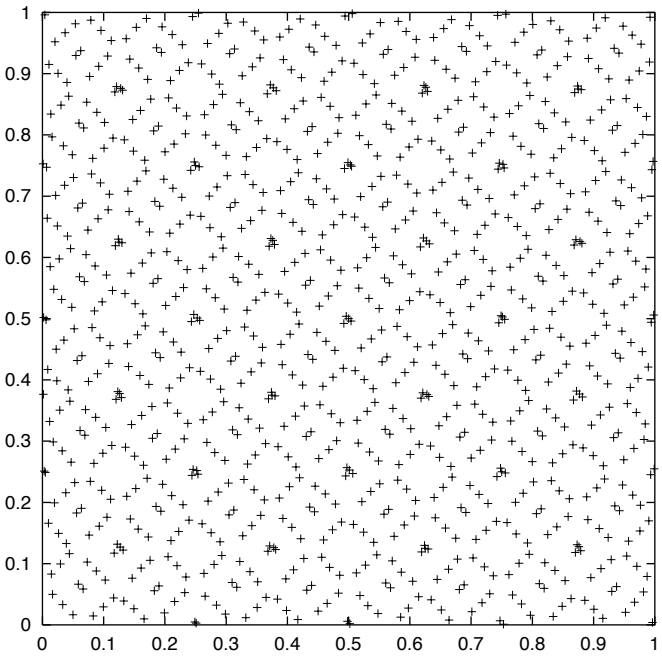
以下に示すプログラムでは、2 次元空間内の 1024 個の点をソボルの方法で生成する。

```
#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、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。



ソボルの疑似乱数による
最初の 1024 点の分布

18.7 参考文献

疑似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 and B.L. Fox and H. Niederreiter, “Algorithm 738: Programs to Generate Niederreiter’s Low-discrepancy Sequences”, ACM Transactions on Mathematical Software, Vol. 20, No. 4, December, 1994, p. 494–495.

第 19 章 確率分布と乱数

この章では、乱数を発生する関数およびその確率分布を計算する関数について説明する。その確率分布にしたがう乱数は、GSL のアルゴリズムの乱数発生器を使って生成することができる。

一様でない乱数を生成するもっとも簡潔な方法は、一様乱数に得たい分布になるような変換を加えることである。この方法では 1 個の乱数を発生するのに 1 回だけ乱数発生器を使う。また複雑な方法として、得たい分布と、あらかじめ似ているとわかっている分布を比較してそれを使うかやめるかを決める受理 - 拒否法もある。この場合は乱数発生器から先に複数の乱数を生成しておく。

GSL では累積分布関数と逆累積分布関数も用意しており、これらは分位関数として使える。累積分布関数とその逆関数は、分布関数の上の裾と下の裾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出来る。

この章で説明する乱数の関数と確率密度分布の関数は '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 ≈ 4.29×10^9である。これらの関数を呼ぶときは、オーバーフローを起こさないように引数の値が安全な範囲（発生した乱数が返せる値の上限を超えてしまう確率が無視できる程度）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

連続値をとる乱数分布は確率密度関数 p(x)として定義され、これは事象 x が x から 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。

下の裾の累積分布関数 P(x)は以下の積分で表され、発生する事象が x 以下となる確率を与える。

$$P(x) = \int_{-\infty}^{x} dx' p(x')$$

上の裾の累積分布関数 Q(x)は以下の積分で表され、発生する事象が x 以上となる確率を与える。

$$P(x) = \int_{x}^{+\infty} dx' p(x')$$

この二つの関数には P(x) + Q(x) = 1 という関係があり 0 ≤ P(x) ≤ 1、0 ≤ Q(x) ≤ 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、Σ_kp(k) = 1 である。この場合の下の裾の累積分布関数 P(k) は以下の式で表され、この値は k 以下になる。

$$P(k) = \sum_{i \le k} p(i)$$

また上の裾の累積分布関数 Q(k)は以下の式で表され、この値は k よりも大きくなる。

$$Q(k) = \sum_{i > k} p(i)$$

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

[Function] double gsl_ran_gaussian (const gsl_rng * r, double sigma)

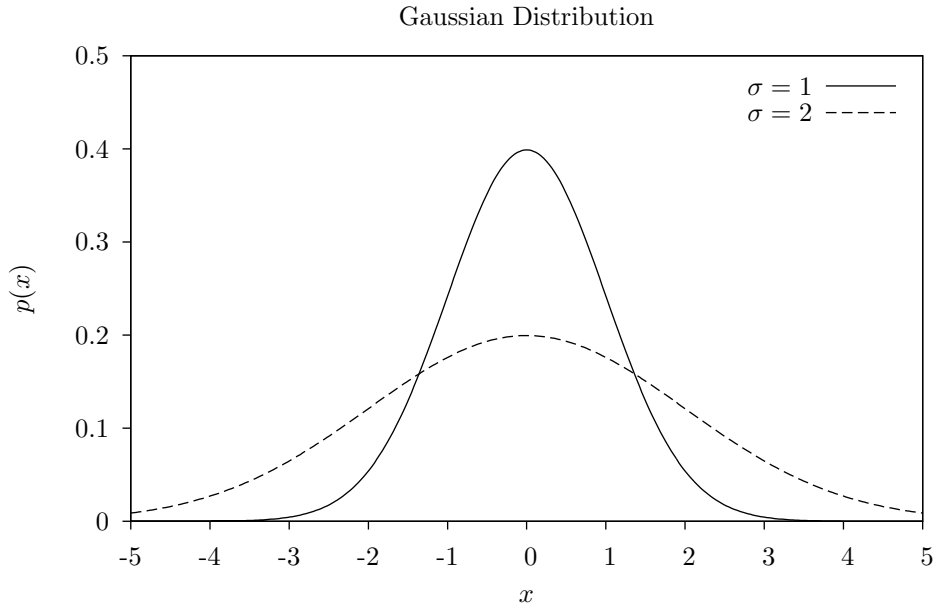
平均 0、標準偏差 sigma の正規分布にしたがう乱数を返す。この分布は以下の式で表される。

$$p(x)dx = \frac{1}{2\pi\sigma^2}\exp(-x^2/2\sigma^2)dx$$

x の範囲は -∞ から +∞ である。gsl_ran_gaussian が返す値に z=μ+x という変換を行うことで平均値 μ の正規分布乱数が得られる。この関数はボックス - ミューラー Box-Mueller の方法を使い、乱数発生器 r を内部で二回呼び出す。

[Function] double gsl_ran_gaussian_pdf (double x, double sigma)

上の式で与えられる標準偏差 μ 正規分布乱数について、確率密度関数 p(x)の値を返す。



[Function] double gsl_ran_gaussian_ziggurat (const gsl_rng * r, double sigma)

[Function] double gsl_ran_gaussian_ratio_method (const gsl_rng * r, double sigma)

正規分布乱数をそれぞれ、マルサグリア - ツァンのジッグラト、キンダーマン - モナハン - レバの比による方法を使って生成する。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。

[Function] double gsl_ran_ugaussian (const gsl_rng * r)

[Function] double gsl_ran_ugaussian_pdf (double x)

[Function] double gsl_ran_ugaussian_ratio_method (const gsl_rng * r)

標準正規分布にしたがう値を返す。上記の関数で標準偏差を 1 にしたものと同じである。

[Function] double gsl_cdf_gaussian_P (double x, double sigma)

[Function] double gsl_cdf_gaussian_Q (double x, double sigma)

[Function] double gsl_cdf_gaussian_Pinv (double P, double sigma)

[Function] double gsl_cdf_gaussian_Qinv (double Q, double sigma)

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。

[Function] double gsl_cdf_ugaussian_P (double x)

[Function] double gsl_cdf_ugaussian_Q (double x)

[Function] double gsl_cdf_ugaussian_Pinv (double P)

[Function] double gsl_cdf_ugaussian_Qinv (double Q)

標準偏差 1 の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

[Function] double gsl_ran_gaussian_tail (const gsl_rng * r, double a, double sigma)

標準偏差 sigma の正規分布の上の裾での分布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。これはクヌース第二巻第三版の p.139,586（練習問題 11）に載っている有名なマルサグリアの方形 - 楔形裾アルゴリズムを使っている（Ann. Math. Stat. 32, 894-899 (1961)）。

正規分布の裾分布は以下で与えられる。

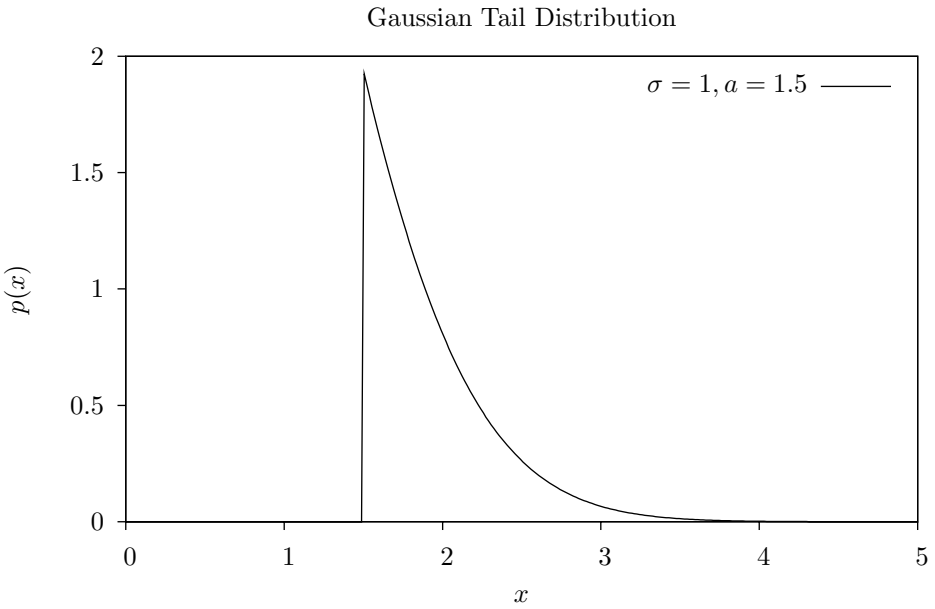
$$p(x)dx = \frac{1}{N(a;\sigma)\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$$

ここで x > a で N(a;σ)は正規化係数であり、以下で表される。

$$N(a;\sigma) = \frac{1}{2}\mathrm{efrc}\left(\frac{a}{\sqrt{2\sigma^2}}\right)$$

[Function] double gsl_ran_gaussian_tail_pdf (double x, double a, double sigma)

上の式に従い、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。



[Function] double gsl_ran_ugaussian_tail (const gsl_rng * r, double a)

[Function] double gsl_ran_ugaussian_tail_pdf (double x, double a)

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

[Function] void gsl_ran_bivariate_gaussian (const gsl_rng * r, double sigma_x, double sigma_y, double rho, double * x, double * y)

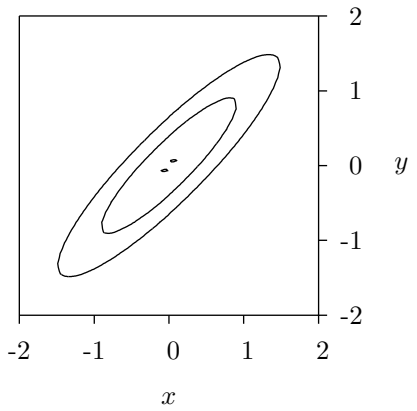
平均値 0、相関係数 rho で、相関を持つ二つの正規分布乱数 x と y を生成する。x と y の標準偏差はそれぞれ sigma_x と sigma_y である。この乱数 x と y は -∞ から +∞ の範囲をとり、その確率分布は以下で与えられる。

$$p(x,y)dxdy = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left(-\frac{x^2/\sigma_x^2 + y^2/\sigma_y^2 - 2\rho xy/(\sigma_x\sigma_y)}{2(1-\rho^2)}\right) dxdy$$

相関係数 rho の値は -1 以上 1 以下でなければならない。

[Function] double gsl_ran_bivariate_gaussian_pdf (double x, double y, double sigma_x, double sigma_y, double rho)

上の式にしたがう、x および y における標準偏差 sigma_x、sigma_y、相関係数 rho の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

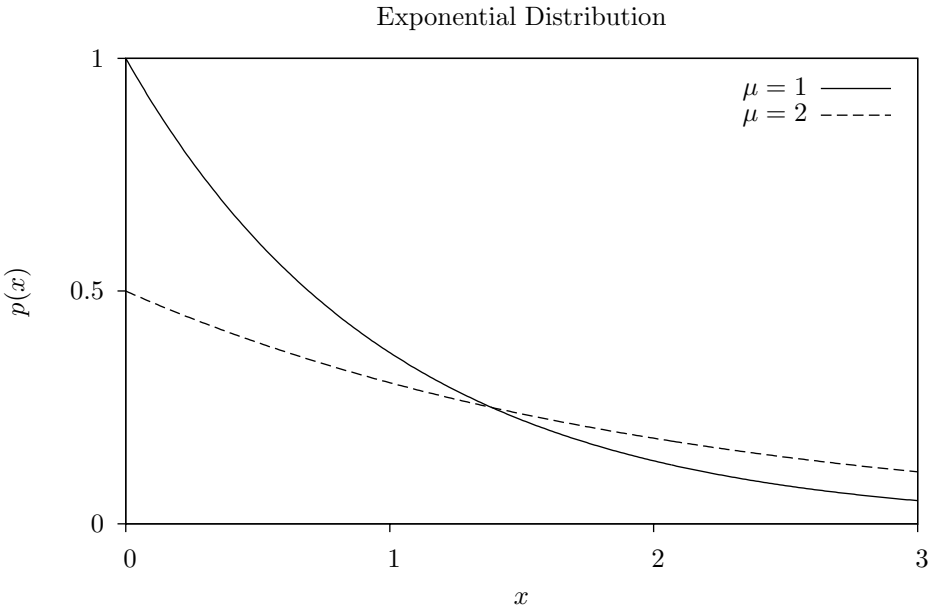
[Function] double gsl_ran_exponential (const gsl_rng * r, double mu)

平均値 mu の指数分布にしたがう乱数を返す。指数分布は x ≥ 0 で以下で与えられる。

$$p(x)dx = \frac{1}{\mu}\exp(-x/\mu)dx$$

[Function] double gsl_ran_exponential_pdf (double x, double mu)

上の式にしたがって、x における平均値 mu の指数分布の確率密度関数の値を返す。



[Function] double gsl_cdf_exponential_P (double x, double mu)

[Function] double gsl_cdf_exponential_Q (double x, double mu)

[Function] double gsl_cdf_exponential_Pinv (double P, double mu)

[Function] double gsl_cdf_exponential_Qinv (double Q, double mu)

平均値 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

[Function] double gsl_ran_laplace (const gsl_rng * r, double a)

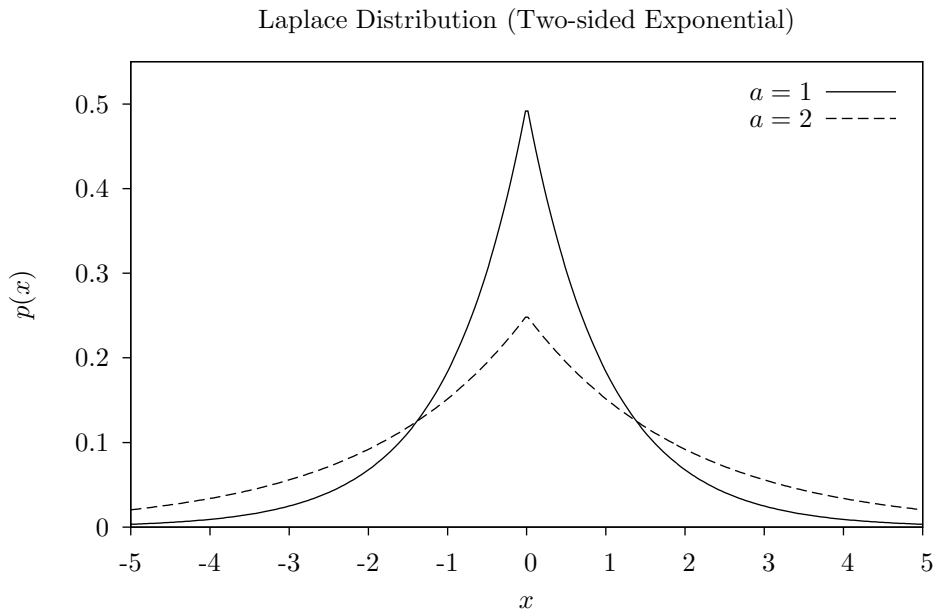
幅 a のラプラス分布にしたがう乱数を返す。ラプラス分布は以下で与えられる。

$$p(x)dx = \frac{1}{2a}\exp(-|x/a|)dx$$

x の範囲は -∞ ≤ x ≤ ∞ である。

[Function] double gsl_ran_laplace_pdf (double x, double a)

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。



[Function] double gsl_cdf_laplace_P (double x, double a)

[Function] double gsl_cdf_laplace_Q (double x, double a)

[Function] double gsl_cdf_laplace_Pinv (double P, double a)

[Function] double gsl_cdf_laplace_Qinv (double Q, double a)

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

[Function] double gsl_ran_exppow (const gsl_rng * r, double a, double b)

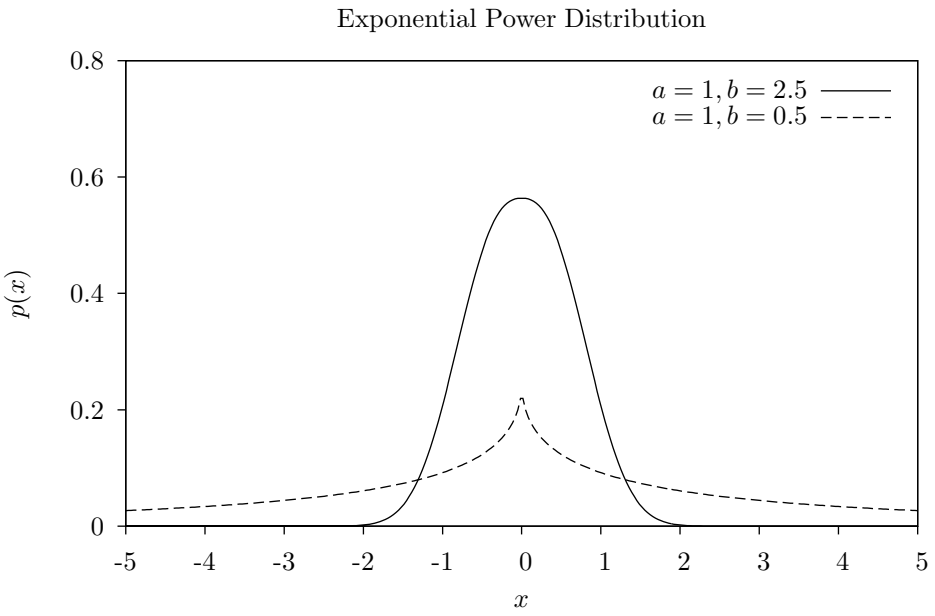
底が a、指数が b のべき指数分布にしたがう乱数を返す。べき指数分布は x ≥ 0 について以下で与えられる。

$$p(x)dx = \frac{1}{2a\Gamma(1+1/b)} \exp(-|x/a|^b)dx$$

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=√(2) σ である。

[Function] double gsl_ran_exppow_pdf (double x, double a, double b)

上の式にしたがって、x における底が a、指数が b のべき指数分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_exppow_P (double x, double a, double b)

[Function] double gsl_cdf_exppow_Q (double x, double a, double b)

底が a、指数が b 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。

19.8 コーシー分布

[Function] double gsl_ran_cauchy (const gsl_rng * r, double a)

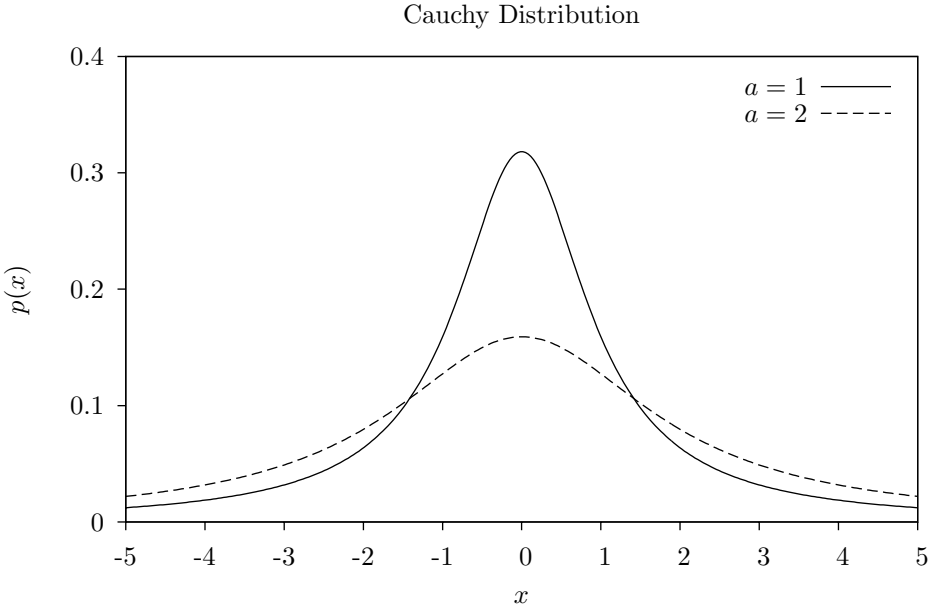
係数 a のコーシー分布にしたがう乱数を返す。コーシー分布は以下で与えられる。

$$p(x)dx = \frac{1}{a\pi(1 + (x/a)^2)}dx$$

x の範囲は -∞ ≤ x ≤ ∞ である。コーシー分布はローレンツ分布としても知られる。

[Function] double gsl_ran_cauchy_pdf (double x, double a)

上の式にしたがって、x における係数 a のコーシー分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_cauchy_P (double x, double a)

[Function] double gsl_cdf_cauchy_Q (double x, double a)

[Function] double gsl_cdf_cauchy_Pinv (double P, double a)

[Function] double gsl_cdf_cauchy_Qinv (double Q, double a)

係数が 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

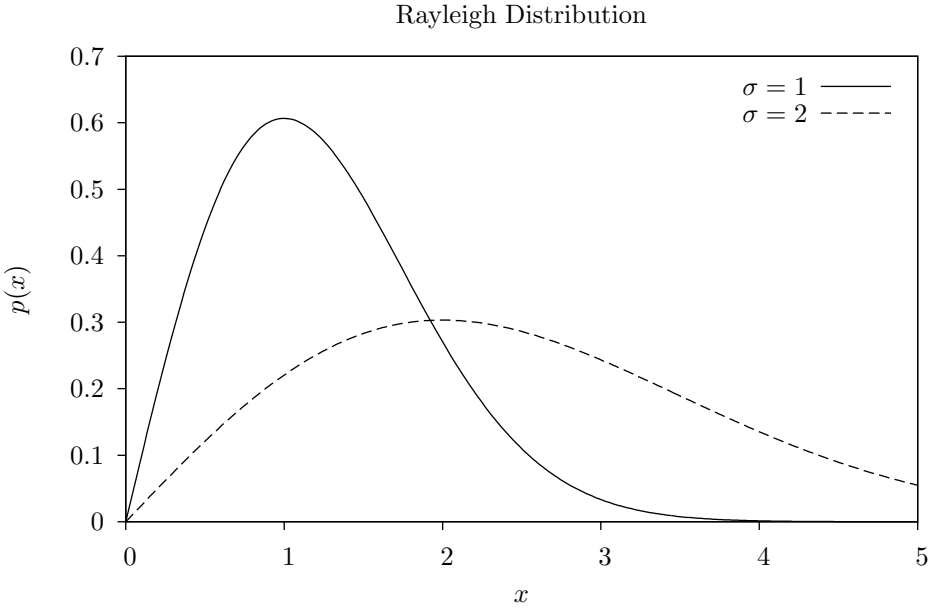
[Function] double gsl_ran_rayleigh (const gsl_rng * r, double sigma)

係数 sigma のレイリー分布にしたがう乱数を返す。レイリー分布は x ≥ 0 について以下で与えられる。

$$p(x)dx = \frac{x}{\sigma^2}\exp(-x^2/(2\sigma^2))dx$$

[Function] double gsl_ran_rayleigh_pdf (double x, double sigma)

上の式にしたがって、x における係数 sigma のレイリー分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_rayleigh_P (double x, double sigma)

[Function] double gsl_cdf_rayleigh_Q (double x, double sigma)

[Function] double gsl_cdf_rayleigh_Pinv (double P, double sigma)

[Function] double gsl_cdf_rayleigh_Qinv (double Q, double sigma)

係数が 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

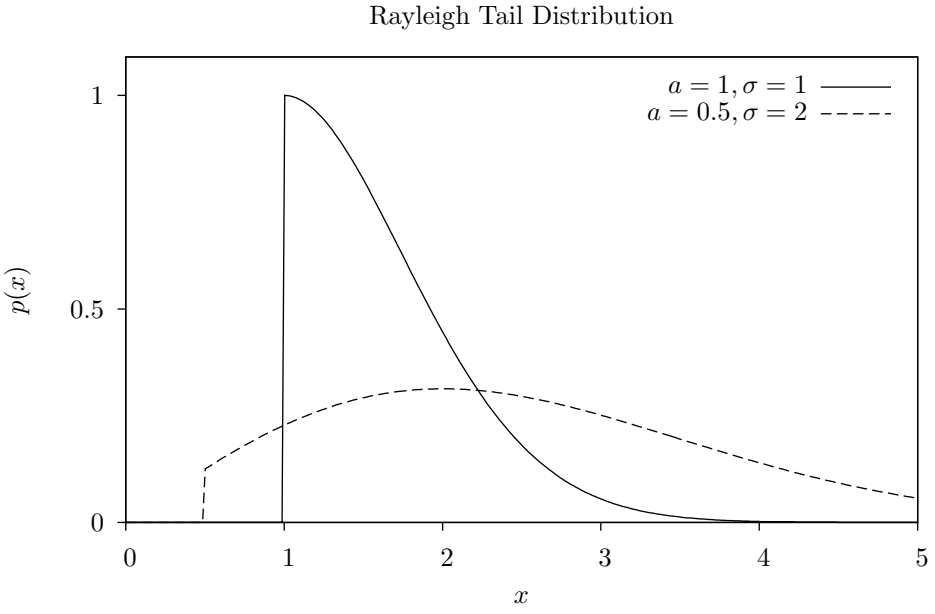
[Function] double gsl_ran_rayleigh_tail (const gsl_rng * r, double a, double sigma)

係数 sigma のレイリー分布の裾での分布にしたがう乱数を下限 a で返す。レイリー分布は x ≥ 0 について以下で与えられる。

$$p(x)dx = \frac{x}{\sigma^2}\exp((a^x - x^2)/(2\sigma^2))dx$$

[Function] double gsl_ran_rayleigh_tail_pdf (double x, double a, double sigma)

上の式にしたがって、x における係数 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

[Function] double gsl_ran_landau (const gsl_rng * r)

ランダウ分布にしたがう乱数を返す。ランダウ分布は以下の複素積分で定義される。

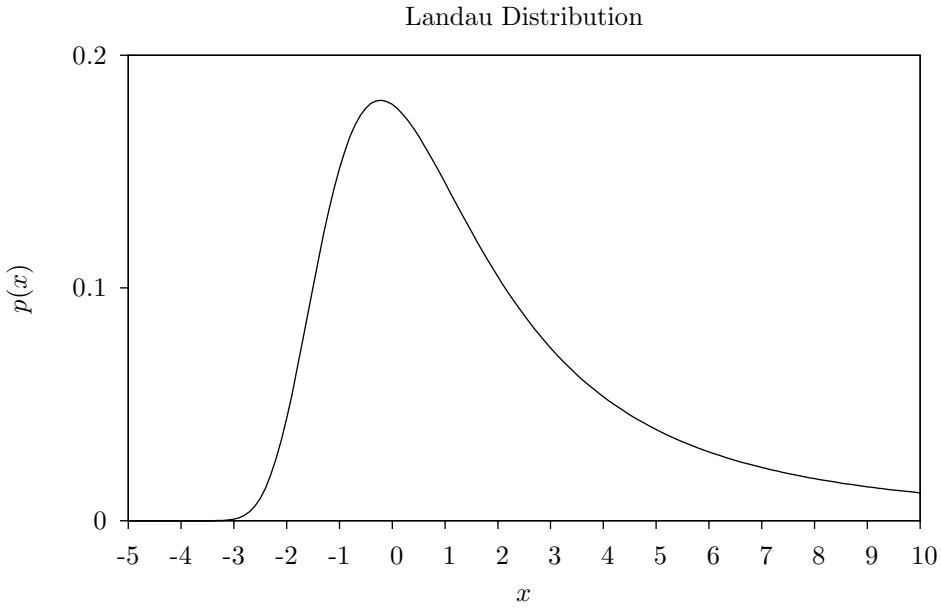
$$p(x)dx = \frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} ds\exp(s\log(s)+xs)$$

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。

$$p(x)dx = \frac{1}{\pi}\int_{0}^{\infty} dt\exp(-t\log(t)-xt)\sin(\pi t)$$

[Function] double gsl_ran_landau_pdf (double x)

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 レビの α 安定分布

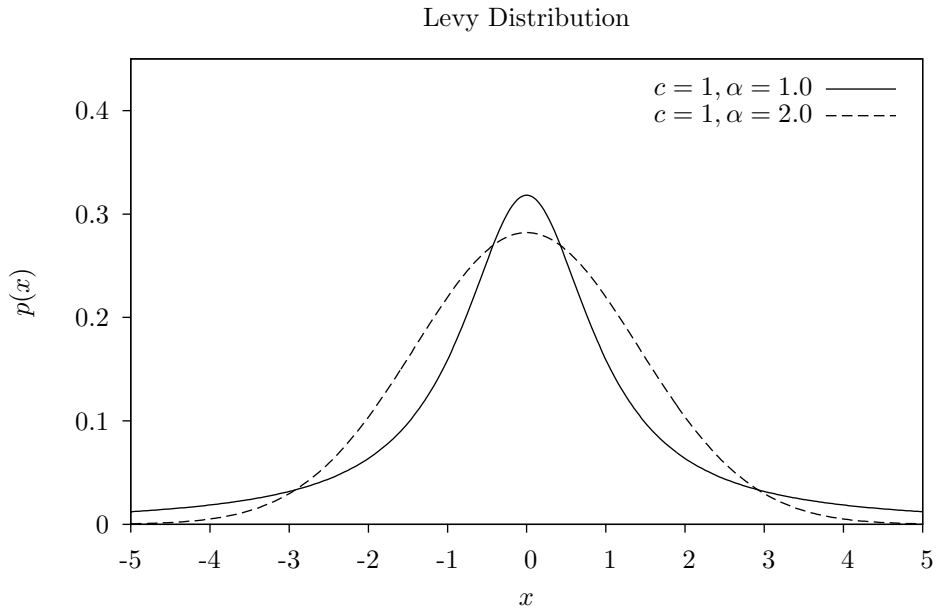
[Function] double gsl_ran_levy (const gsl_rng * r, double c, double alpha)

係数が c、指数が alpha のレビの対称安定分布にしたがう乱数を返す。対称安定分布は以下のフーリエ変換で定義される。

$$p(x)dx = \frac{1}{2\pi}\int_{-\infty}^{\infty} dt \exp(-itx - |ct|^{\alpha})$$

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を用意していない。この分布は α=2 の時は σ=√(2)c の正規分布である。α < 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 < α ≤ 2 でなければならない。



19.13 レビの非対称 α 安定分布

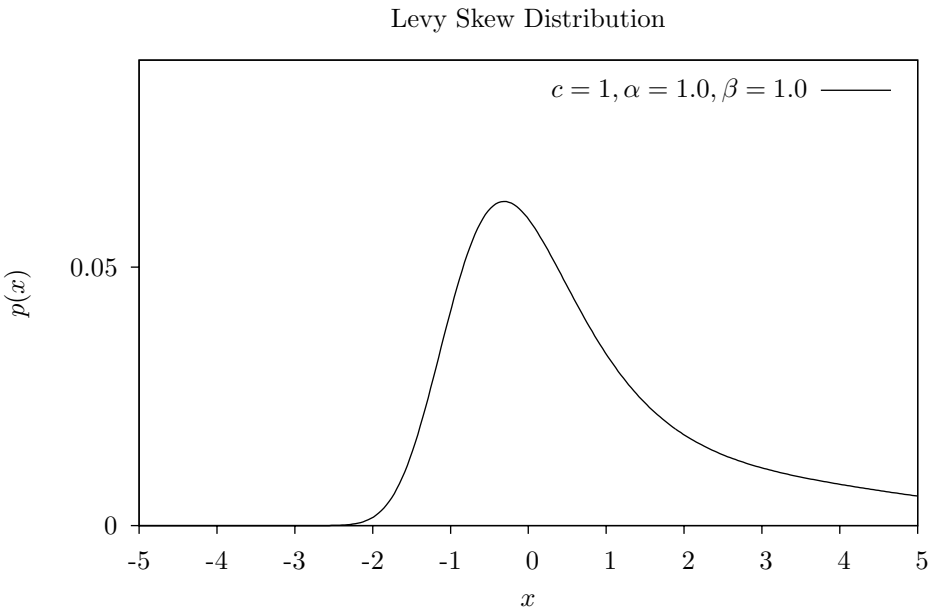
[Function] double gsl_ran_levy_skew (const gsl_rng * r, double c, double alpha, double beta)

係数が c、指数が alpha、非対称係数 beta のレビの非対称安定分布にしたがう乱数を返す。対称安定分布は以下のフーリエ変換で定義される。非対称係数は[-1, 1]の範囲内でなければならない。レビの非対称安定分布は以下のフーリエ変換で定義される。

$$p(x)dx = \frac{1}{2\pi}\int_{-\infty}^{\infty} dt \exp(-itx - |ct|^{\alpha}(1 - i\beta \mathrm{sign}(t)\tan(\pi\alpha/2)))$$

α=1 のとき、tan(π α/2) の項は -(2/π)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を用意していない。α=2 の時この分布は σ=√(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。α < 1 ではこの分布の裾は非常に広くなる。対象分布は β=0 と同値である。

レビの α 安定分布は、N 個の α 安定な値が分布 p(c, α, β)から与えられたとき、その和 $Y = X_1+X_2+...X_N$ 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

[Function] double gsl_ran_gamma (const gsl_rng * r, double a, double b)

ガンマ分布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。

$$p(x)dx = \frac{1}{\Gamma(a)b^a} x^{a-1} e^{-x/b} dx$$

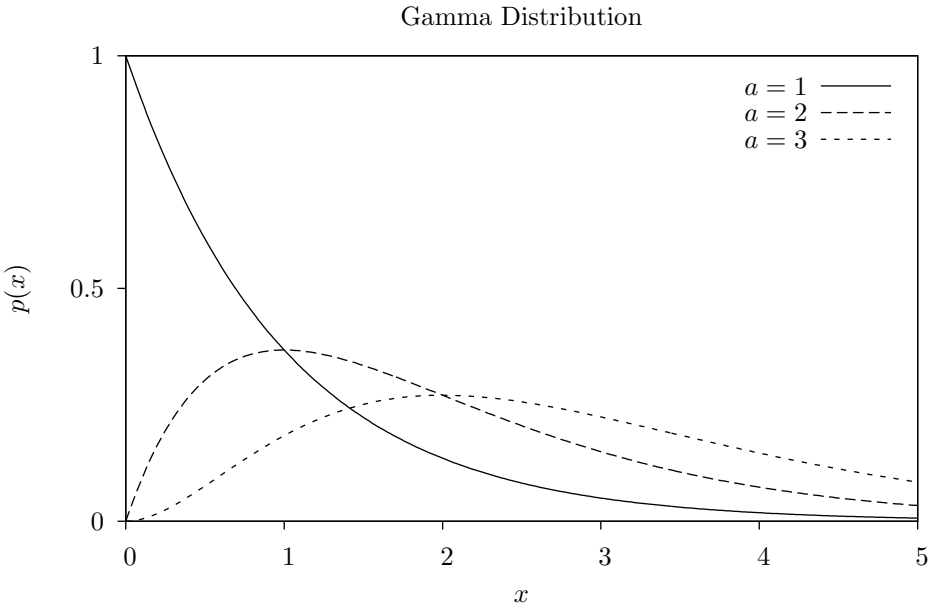
引数が整数の時のガンマ分布はエルラング Erlang の分布としても知られる。この乱数はクヌース第二巻のアルゴリズムで生成される。

[Function] double gsl_ran_gamma_pdf (double x, double a, double b)

マルサグリア - ツァンの高速な方法を使ってガンマ分布にしたがう乱数を返す。

[Function] double gsl_ran_gamma_mt (const gsl_rng * r, double a, double b)

引数が a と b のときの、上の式にしたがった x におけるガンマ分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_gamma_P (double x, double a, double b)

[Function] double gsl_cdf_gamma_Q (double x, double a, double b)

[Function] double gsl_cdf_gamma_Pinv (double P, double a, double b)

[Function] double gsl_cdf_gamma_Qinv (double Q, double a, double b)

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

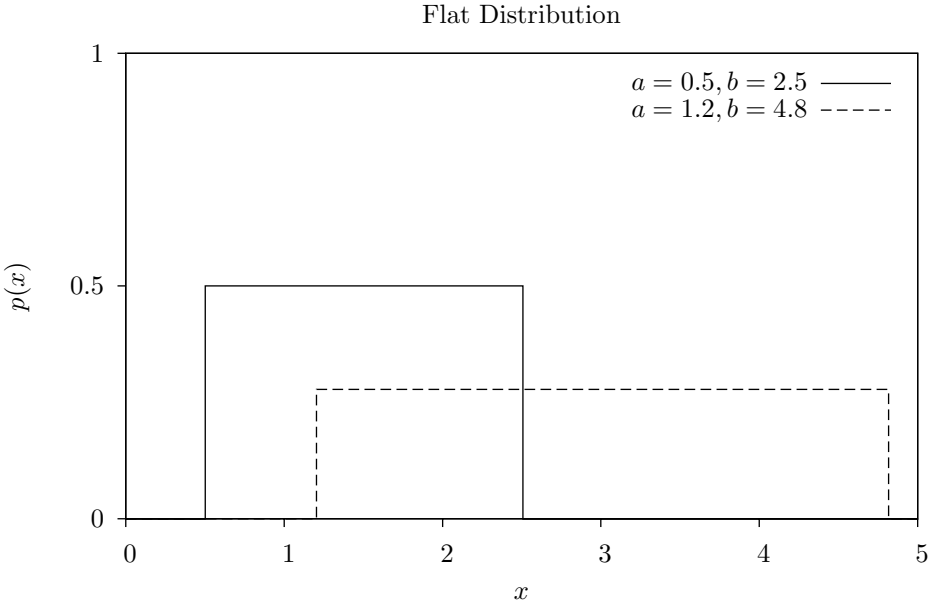
[Function] double gsl_ran_flat (const gsl_rng * r, double a, double b)

a から b の間に一様に分布する乱数を返す。この分布は a ≤ x < b について以下で定義される。

$$p(x)dx = \frac{1}{b-a}dx$$

[Function] double gsl_ran_flat_pdf (double x, double a, double b)

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。



[Function] double gsl_cdf_flat_P (double x, double a, double b)

[Function] double gsl_cdf_flat_Q (double x, double a, double b)

[Function] double gsl_cdf_flat_Pinv (double P, double a, double b)

[Function] double gsl_cdf_flat_Qinv (double Q, double a, double b)

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

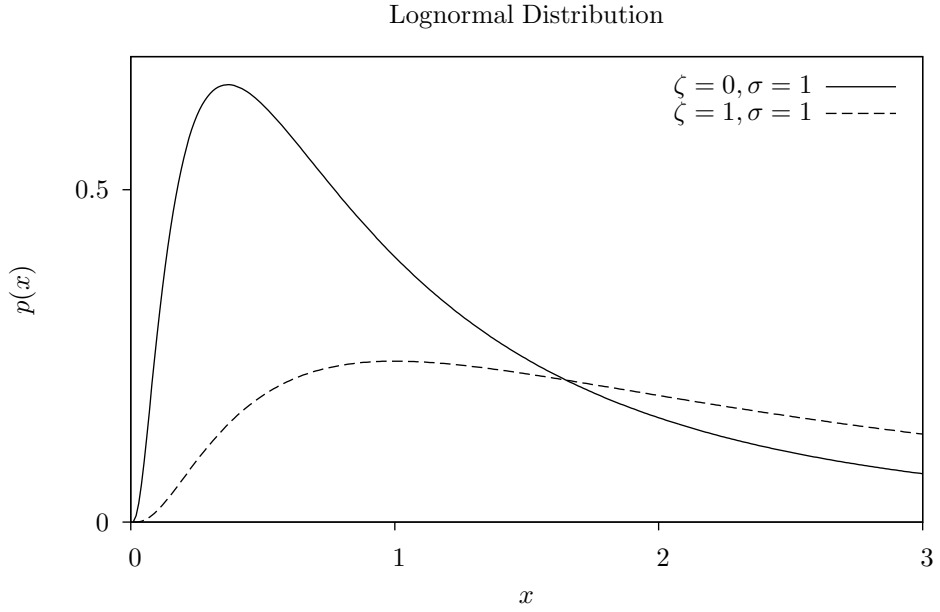
[Function] double gsl_ran_lognormal (const gsl_rng * r, double zeta, double sigma)

対数正規分布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。

$$p(x)dx = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp(-(\ln(x) - \zeta)^2/2\sigma^2)dx$$

[Function] double gsl_ran_lognormal_pdf (double x, double zeta, double sigma)

パラメータ値が zeta と sigma で与えられたとき、上の式にしたがう対数正規分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_lognormal_P (double x, double zeta, double sigma)

[Function] double gsl_cdf_lognormal_Q (double x, double zeta, double sigma)

[Function] double gsl_cdf_lognormal_Pinv (double P, double zeta, double sigma)

[Function] double gsl_cdf_lognormal_Qinv (double Q, double zeta, double sigma)

パラメータ値が zeta と sigma で与えられたとき、上の式にしたがう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

カイ二乗分布は統計学でよく用いられる。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、

$$X_i = \sum_i Y_i^2$$

は自由度 n のカイ二乗分布にしたがう。

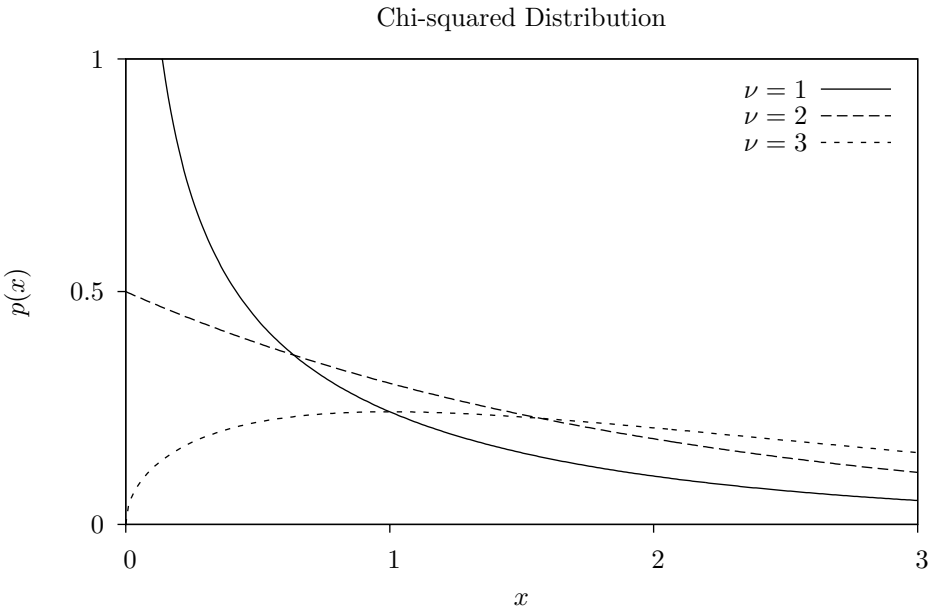
[Function] double gsl_ran_chisq (const gsl_rng * r, double nu)

自由度 n のカイ二乗分布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。

$$p(x)dx = \frac{1}{2\Gamma(\nu/2)}(x/2)^{\nu/2-1}\exp(-x/2)dx$$

[Function] double gsl_ran_chisq_pdf (double x, double nu)

上の式にしたがう自由度 n のカイ二乗分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_chisq_P (double x, double nu)

[Function] double gsl_cdf_chisq_Q (double x, double nu)

[Function] double gsl_cdf_chisq_Pinv (double P, double nu)

[Function] double gsl_cdf_chisq_Qinv (double Q, double nu)

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 分布は統計学でよく用いられる。Y_i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;ν_1, ν_2)にしたがう。

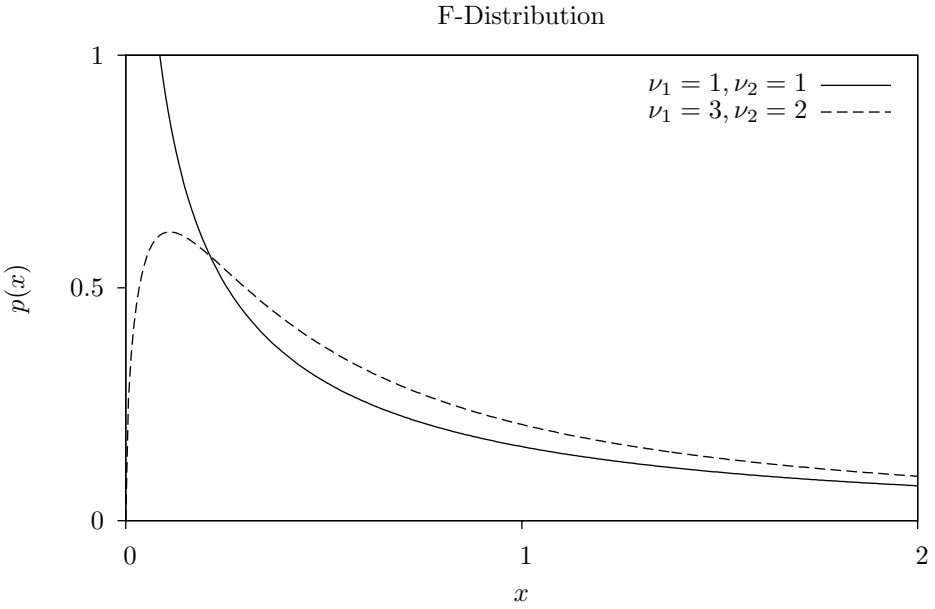
[Function] double gsl_ran_fdist (const gsl_rng * r, double nu)

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 ≧ 0 について以下で定義される。

$$p(x)dx = \frac{\Gamma((\nu_1+\nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)}\nu_1^{\nu_1/2}\nu_2^{\nu_2/2}x^{\nu_1/2-1}(\nu_2+\nu_1 x)^{-\nu_1/2-\nu_2/2}$$

[Function] double gsl_ran_fdist_pdf (double x, double nu)

上の式にしたがう、自由度 nu1、nu2 の F 分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_fdist_P (double x, double nu)

[Function] double gsl_cdf_fdist_Q (double x, double nu)

[Function] double gsl_cdf_fdist_Pinv (double P, double nu)

[Function] double gsl_cdf_fdist_Qinv (double Q, double nu)

上の式にしたがう自由度 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 分布は統計学でよく用いられる。Y_iが正規分布に、Y_2自由度 ν のカイ二乗分布にしたがうとき、以下の比

$$X = \frac{Y_i}{\sqrt{Y_i/\nu}}$$

は自由度 ν の t 分布 t(x; ν)にしたがう。

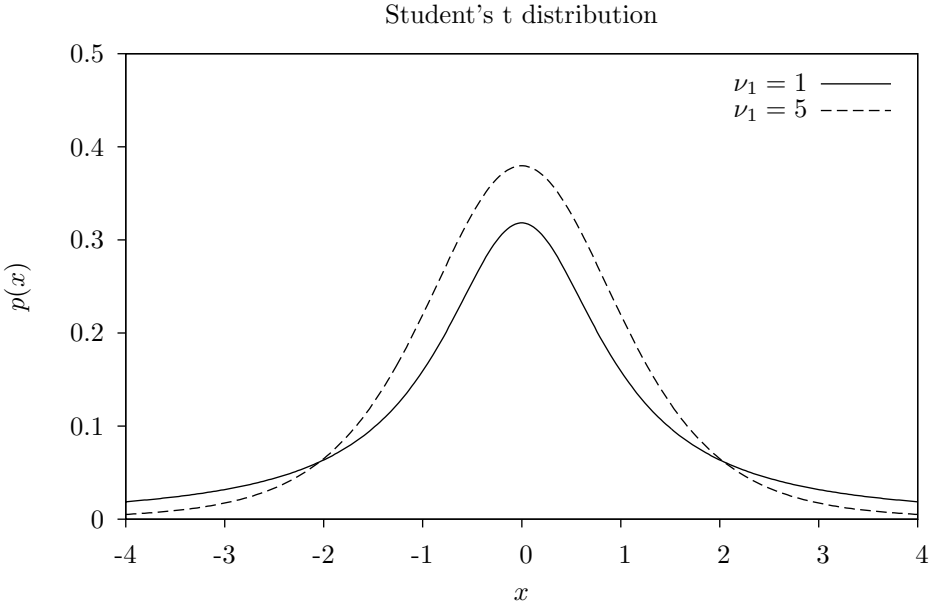
[Function] double gsl_ran_tdist (const gsl_rng * r, double nu)

以下の t 分布にしたがう乱数を返す。-∞ < x < +∞ である。

$$p(x)dx = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}(1+x^2/\nu)^{-(\nu+1)/2}dx$$

[Function] double gsl_ran_tdist_pdf (double x, double nu)

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。



[Function] double gsl_cdf_tdist_P (double x, double nu)

[Function] double gsl_cdf_tdist_Q (double x, double nu)

[Function] double gsl_cdf_tdist_Pinv (double P, double nu)

[Function] double gsl_cdf_tdist_Qinv (double Q, double nu)

上の式にしたがう自由度 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

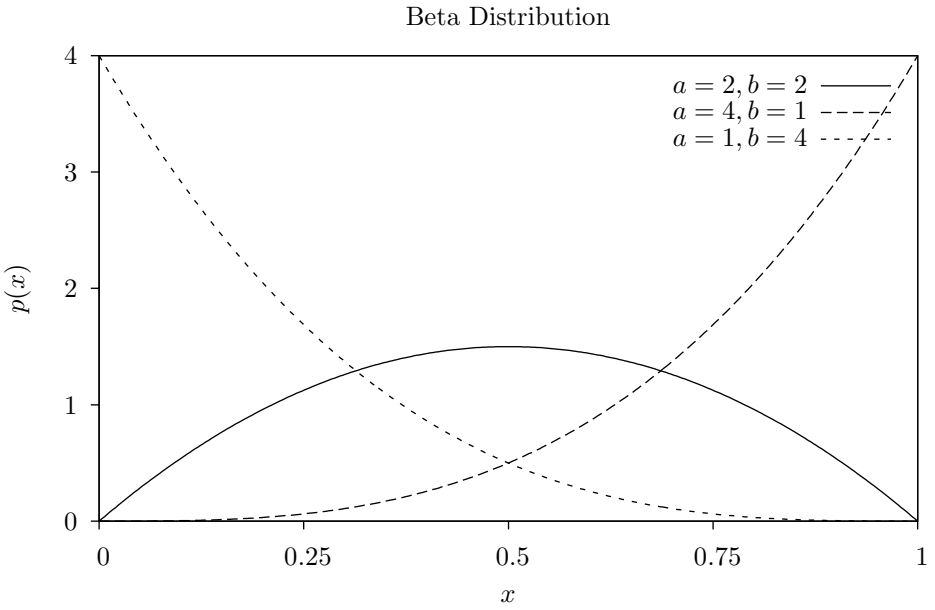
[Function] double gsl_ran_beta (const gsl_rng * r, double a, double b)

ベータ分布にしたがう乱数を返す。この分布は 0 ≦ x < 0 について以下で定義される。

$$p(x)dx = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}dx$$

[Function] double gsl_ran_beta_pdf (double x, double a, double b)

パラメータ値が a と b で与えられたとき、上の式にしたがうベータ分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_beta_P (double x, double a, double b)

[Function] double gsl_cdf_beta_Q (double x, double a, double b)

[Function] double gsl_cdf_beta_Pinv (double P, double a, double b)

[Function] double gsl_cdf_beta_Qinv (double Q, double a, double b)

パラメータ値が a と b で与えられたとき、上の式にしたがう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

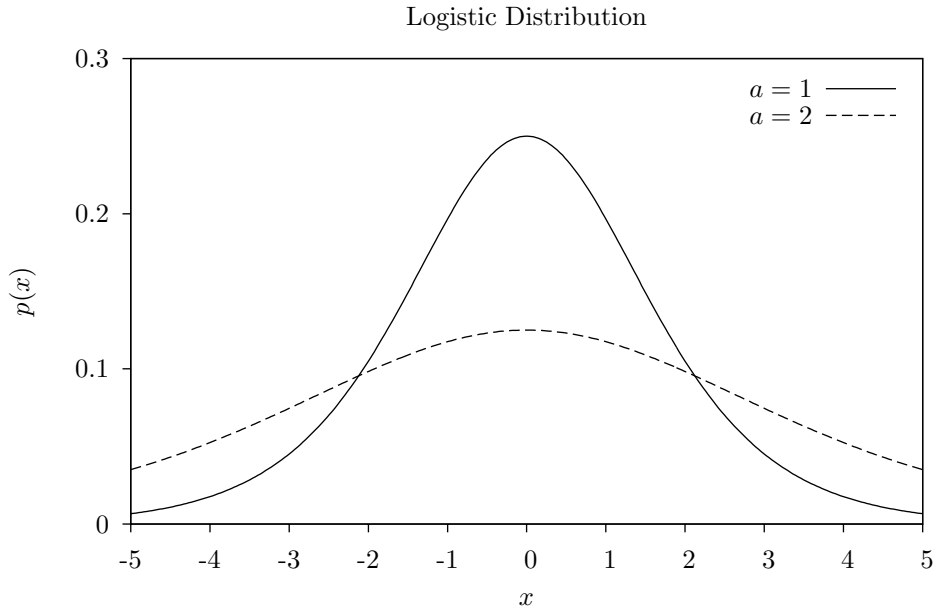
[Function] double gsl_ran_logistic (const gsl_rng * r, double a)

ロジスティック分布にしたがう乱数を返す。この分布は -∞ ≦ x < +∞ について以下で定義される。

$$p(x)dx = \frac{\exp(-x/a)}{a(1+\exp(-x/a))^2}dx$$

[Function] double gsl_ran_logistic_pdf (double x, double a)

係数 a が与えられたとき、上の式にしたがうロジスティック分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_logistic_P (double x, double a)

[Function] double gsl_cdf_logistic_Q (double x, double a)

[Function] double gsl_cdf_logistic_Pinv (double P, double a)

[Function] double gsl_cdf_logistic_Qinv (double Q, double a)

係数 a が与えられたとき、上の式にしたがう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

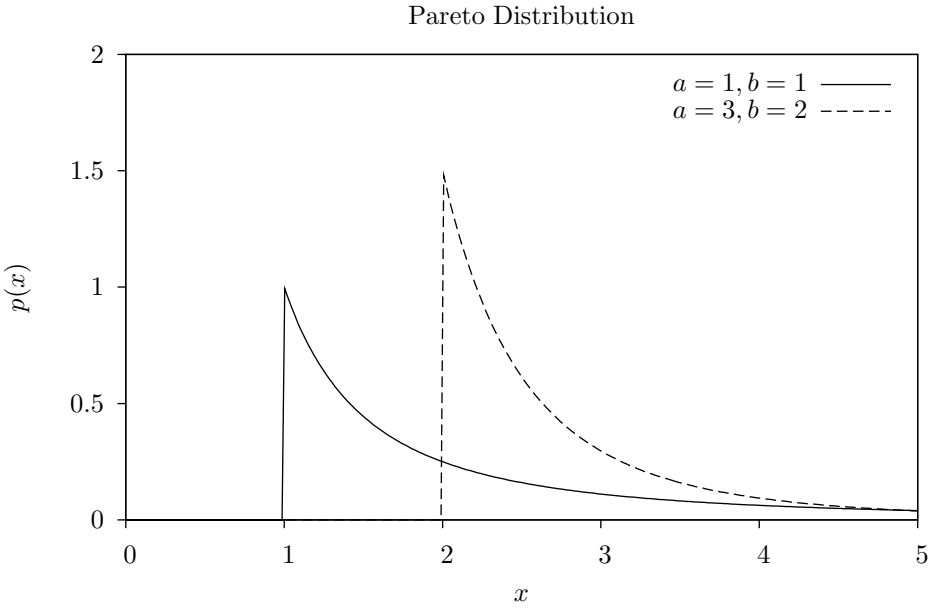
[Function] double gsl_ran_pareto (const gsl_rng * r, double a, double b)

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 ≧ b について以下で定義される。

$$p(x)dx = (a/b)/(x/b)^{a+1}dx$$

[Function] double gsl_ran_pareto_pdf (double x, double a, double b)

次数 a と係数 b が与えられたとき、上の式にしたがうパレート分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_pareto_P (double x, double a, double b)

[Function] double gsl_cdf_pareto_Q (double x, double a, double b)

[Function] double gsl_cdf_pareto_Pinv (double P, double a, double b)

[Function] double gsl_cdf_pareto_Qinv (double Q, double a, double b)

次数 a と係数 b が与えられたとき、上の式にしたがう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

球面分布は旧免状にランダムに分布するベクトルを生成する。これはランダム・ウォークの各ステップで使うような、ランダムな方向として使うことができる。

[Function] 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)

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、最近の計算機にはハードウェアで三角関数を計算するものもあるため、場合によってはこの関数が最も速い。Pentium はその一つである（しかし Sun の Sparc-station はそうではない)。三角関数の呼び出しを避ける、つまりこの関数を使わずに同じ乱数を生成するには、x と y を単位円の内側にとり（一様乱数の発生器で（$0 \leqq x < 1$、$0 \leqq y < 1$ 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)、その点の x、y 座標を $\sqrt{(x^2 + y^2)}$で割ればよい。ノイマン von Neumann によるもっと優れた方法（クヌースの第二巻第三版 140 頁、演習 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。

[Function] void gsl_ran_dir_3d (const gsl_rng * r, double * x, double * y, double * z)

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 $|v|^2 = x^2 + y^2 + z^2$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はクヌース第二巻第三版 136 頁で説明されている Robert E. Knop (CACM 13, 326 (1970)) によるものである。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)。

[Function] void gsl_ran_dir_nd (const gsl_rng * r, size_t n, double * x)

n 次元の球面にランダムに分布するベクトル $v = (x_1, x_2, ..., x_n)$ を返す。返されるベクトルは $|v|^2 = x_1^2 + x_2^2 + ... + x_n^2$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うように生成されてから正規化される。この方法は G. W. Brown, Modern Mathematics for the Engineer (1956)によるもので、クヌース第二巻第三版 135-136 頁で説明されている。

19.24 ワイブル分布

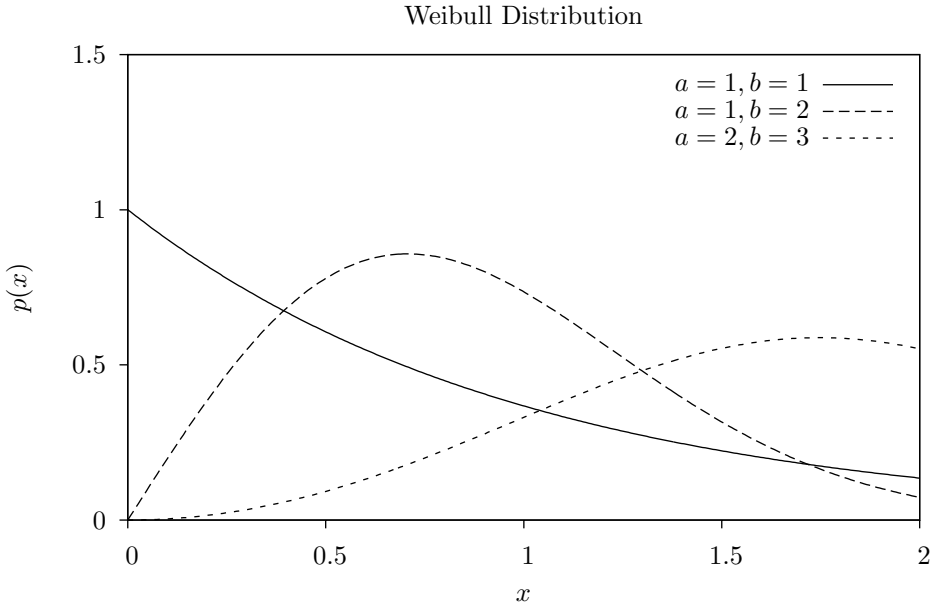
[Function] double gsl_ran_weibull (const gsl_rng * r, double a, double b)

ワイブル分布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。

$$p(x)dx = \frac{b}{a^b} x^{b-1} \exp(-(x/a)^b) dx$$

[Function] double gsl_ran_weibull_pdf (double x, double a, double b)

係数 a と次数 b が与えられたとき、上の式にしたがうワイブル分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_weibull_P (double x, double a, double b)

[Function] double gsl_cdf_weibull_Q (double x, double a, double b)

[Function] double gsl_cdf_weibull_Pinv (double P, double a, double b)

[Function] double gsl_cdf_weibull_Qinv (double Q, double a, double b)

係数 a と次数 b が与えられたとき、上の式にしたがう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 グンベル I 型分布（第一種極値分布）

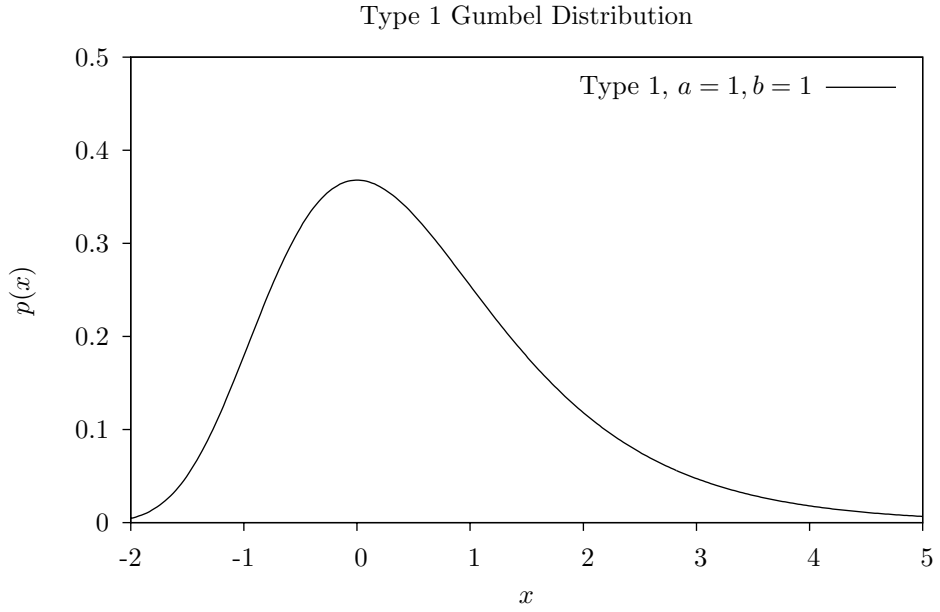
[Function] double gsl_ran_gumbel1 (const gsl_rng * r, double a, double b)

第一種極値分布にしたがう乱数を返す。この分布は -∞ ≦ x ≦ ∞ について以下で定義される。

$$p(x)dx = ab\exp(-(b\exp(-ax) + ax))dx$$

[Function] double gsl_ran_gumbel1_pdf (double x, double a, double b)

パラメータ a、b が与えられたとき、上の式にしたがう第一種極値分布の確率密度関数 p(x) の値を返す。



[Function] double gsl_cdf_gumbel1_P (double x, double a, double b)

[Function] double gsl_cdf_gumbel1_Q (double x, double a, double b)

[Function] double gsl_cdf_gumbel1_Pinv (double P, double a, double b)

[Function] double gsl_cdf_gumbel1_Qinv (double Q, double a, double b)

パラメータ a、b が与えられたとき、上の式にしたがう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 グンベル II 型分布

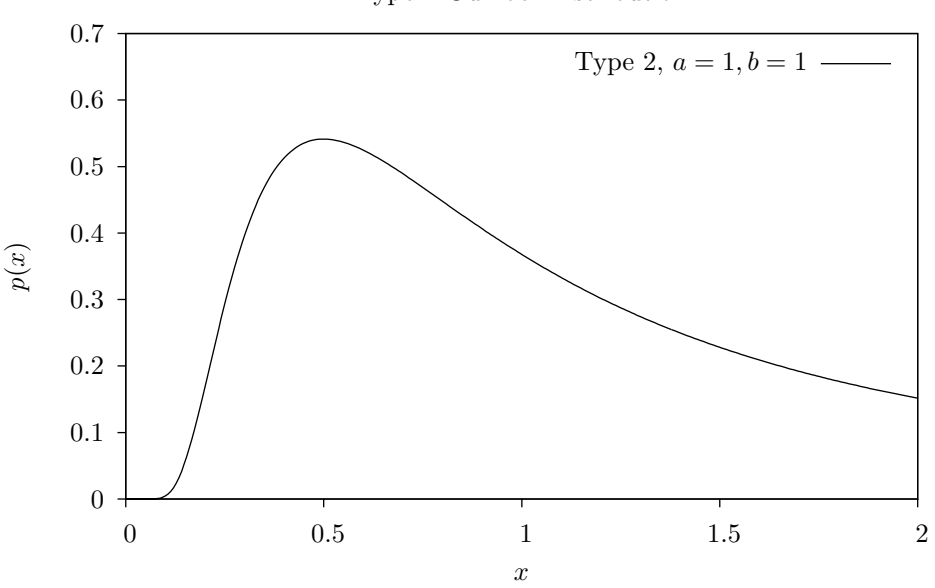
[Function] double gsl_ran_gumbel2 (const gsl_rng * r, double a, double b)

グンベル II 型分布にしたがう乱数を返す。この分布は -∞ ≦ x ≦ ∞ について以下で定義される。

$$p(x)dx = abx^{-a-1}\exp(-bx^{-a})dx$$

[Function] double gsl_ran_gumbel2_pdf (double x, double a, double b)

パラメータ a、b が与えられたとき、上の式にしたがうグンベル II 型分布の確率密度関数 p(x)の値を返す。



[Function] double gsl_cdf_gumbel2_P (double x, double a, double b)

[Function] double gsl_cdf_gumbel2_Q (double x, double a, double b)

[Function] double gsl_cdf_gumbel2_Pinv (double P, double a, double b)

[Function] double gsl_cdf_gumbel2_Qinv (double Q, double a, double b)

パラメータ a、b が与えられたとき、上の式にしたがうグンベル II 型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

[Function] void gsl_ran_dirichlet (const gsl_rng * r, size_t K, const double alpha [], double theta [])

K-1 次のディリクレ分布にしたがう K 個の乱数を生成する。この分布は $\theta_i \geqq 0$ および $\alpha_i \geqq 0$ で以下の式で定義される。

$$p(\theta_1, \ldots, \theta_K) d\theta_1 \cdots d\theta_K = \frac{1}{Z} \prod_{i=1} K \theta_i^{\alpha_i - 1} \delta(1 - \sum_{i=1}^{K} \theta_i) d\theta_1, \ldots, \theta_K$$

ここでデルタ関数は $\Sigma \theta_i = 1$ を満たす。正規化係数 Z は以下で表される。

$$Z = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{K} \alpha_i)}$$

生成される乱数は K 個で、それぞれが $a = \alpha_i$、b = 1 のガンマ分布にしたがう。A.M. Law, W.D. Kelton, Simulation Modeling and Analysis (1991) を参照のこと。

[Function] double gsl_ran_dirichlet_pdf (size_t K, const double alpha [], const double theta [])

パラメータ alpha[K] が与えられたとき、上の式にしたがう 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。

[Function] double gsl_ran_dirichlet_lnpdf (size_t K, const double alpha [], const double theta [])

パラメータ alpha[K] が与えられたとき、上の式にしたがうで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 個の事象がそれぞれ異なった確率 P[k]で生じたとするとき、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を計算し、リストにしておくことである。

$$C[0] = 0$$

$$C[k+1] = C[k] + P[k]$$

こうすると C[K] = 1 になる。ここで 0 から 1 の間の単位偏差 u をとり、C{k] ≦ u < C[k+1] となる k を探すと、一般にはこの探索には log K のオーターのステップ数が必要になるが、こ探索の初期値を ⌊uK」にとるとより速く探索できることがある。

この方法はすでに実装されている。この方法はあらかじめ確率のリストを処理して表引きできるようににしておくためのものである。これによりランダムな離散事象の関数呼び出しがより高速になる。マルサグリアが考案した方法（Generating discrete random numbers in a computer, Comm ACM 6, 37–38 (1963)）は非常に優れた方法で、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。

ウォーカーにより、さらに優れた方法（An efficient method for generating discrete random variables with general distributions, ACM Trans on Math Software 3, 253–256 (1977); クヌースの第二巻第三版 120-121、139 頁も参照）が考案されている。この方法は浮動小数点次数と整数の二種類の表を使うが、どちらの表も大きさは K である。元の確率のリストを処理して表を生成した後は、K が大きくなっても乱数生成にかかる計算量のオーダーは O(1)である。ウォーカーによる表の生成法の計算量は $O(K^2)$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、前処理をやればやるほど乱数生成はより速くなるが、その速度はかける手間に比べるとあまり上がらなくなる。クヌースによると、K が大きくなると最適な前処理はどうしたらよいか、という問題は組み合わせ問題的に難しくなる。

上に述べた方法は、以下に説明するいくつかの乱数生成法でその速度を上げるのに有効だが、たとえば、とりうる値が K 個の有限集合であるポアソン分布の場合など、修正が必要なこともある。

[Function] gsl_ran_discrete_t * gsl_ran_discrete_preproc (size_t K, const double * P)

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、もっと一般的に「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。

[Function] size_t gsl_ran_discrete (const gsl_rng * r, const gsl ran discrete t * g)

上述の前処理関数を呼んだ後、離散値をとる乱数を得るのにこの関数を使う。

[Function] double gsl_ran_discrete_pdf (size_t k, const gsl ran discrete t * g)

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 P[k} を使った後、別途保持しておくようにするとよい。

[Function] void gsl_ran_discrete_free (gsl ran discrete t * g)

g が指す表を保持するメモリーを解放する。

19.29 ポアソン分布

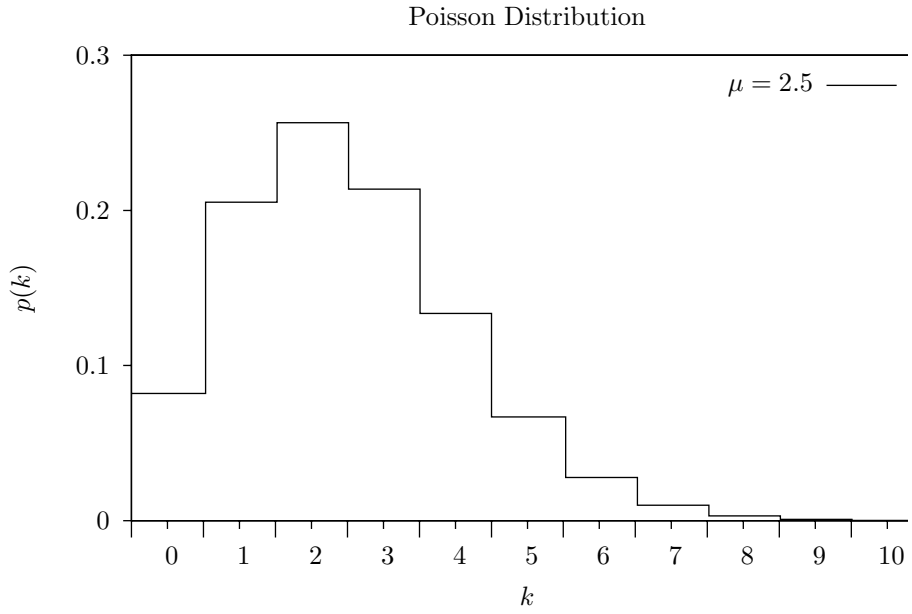
[Function] unsigned int gsl_ran_poisson (const gsl_rng * r, double mu)

平均 mu のポアソン分布にしたがう整数の乱数を生成する。この分布は k ≧ 0 で以下の式で定義される。

$$p(k) = \frac{\mu^k}{k!} \exp(-\mu)$$

[Function] double gsl_ran_poisson_pdf (unsigned int k, double mu)

上の式の平均 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。



[Function] double gsl_cdf_poisson_P (unsigned int k, double mu)

[Function] double gsl_cdf_poisson_Q (unsigned int k, double mu)

上の式にしたがう平均 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

[Function] unsigned int gsl_ran_bernoulli (const gsl_rng * r, double p)

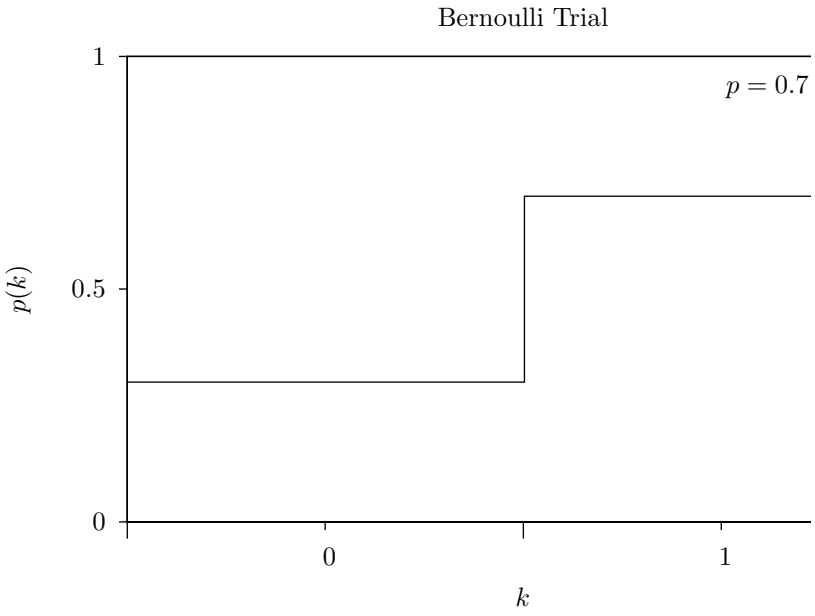
確率 p のベルヌーイ分布にしたがって 0 または 1 を返す。この分布は以下で定義される。

$$p(0) = 1 - p$$

$$p(1) = p$$

[Function] double gsl_ran_bernoulli_pdf (unsigned int k, double p)

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

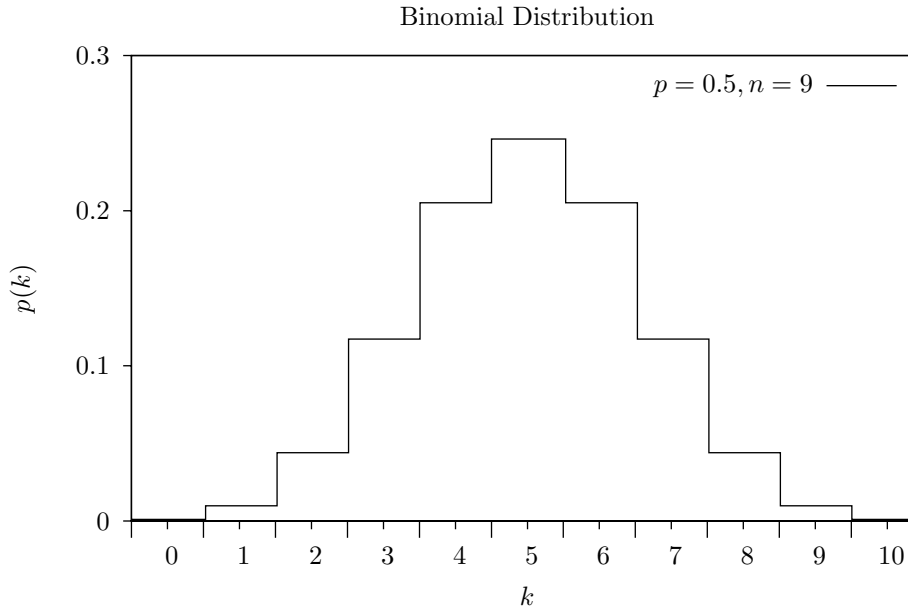
[Function] unsigned int gsl_ran_binomial (const gsl_rng * r, double p, unsigned int n)

二項分布、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 ≦ k ≦ n で以下の式で定義される。

$$p(k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

[Function] double gsl_ran_binomial_pdf (unsigned int k, double p, unsigned int n)

上の式のパラメータが 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。



[Function] double gsl_cdf_binomial_P (unsigned int k, double p, unsigned int n)

[Function] double gsl_cdf_binomial_Q (unsigned int k, double p, unsigned int n)

上の式のパラメータが 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

[Function] void gsl_ran_multinomial (const gsl_rng * r, double p, unsigned int n)

多項分布にしたがう K 個の乱数を配列 K に入れて返す。この分布は以下で定義される。

$$p(n_1, n_2, \ldots, n_K) = \frac{N!}{n_1! n_2! \cdots n_K!} p_1^{n_1} p_2^{n_2} \cdots P_K^{n_K}$$

ここで (n_1, n_2, ... , n_K) は非負の整数で $\Sigma_{k=1}{}^{K} n_k = N$ であり、(p_1, p_2, ... , p_K) は確率分布で $\Sigma p_i = 1$ である。p[K] が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。乱数は条件付きに項分布により生成される（詳しくは C.S. David, The computer generation of multinomial random variates, Comp. Stat. Data Anal. 16 (1993) 205–217 参照)。

[Function] double gsl_ran_multinomial_pdf (unsigned int k, double p, unsigned int n)

上の式の、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, ... , n_K) を計算する。

[Function] double gsl_ran_multinomial_lnpdf (unsigned int k, double p, unsigned int n)

上の式の、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, ... , n_K) の対数値を計算する。

19.33 負の二項分布

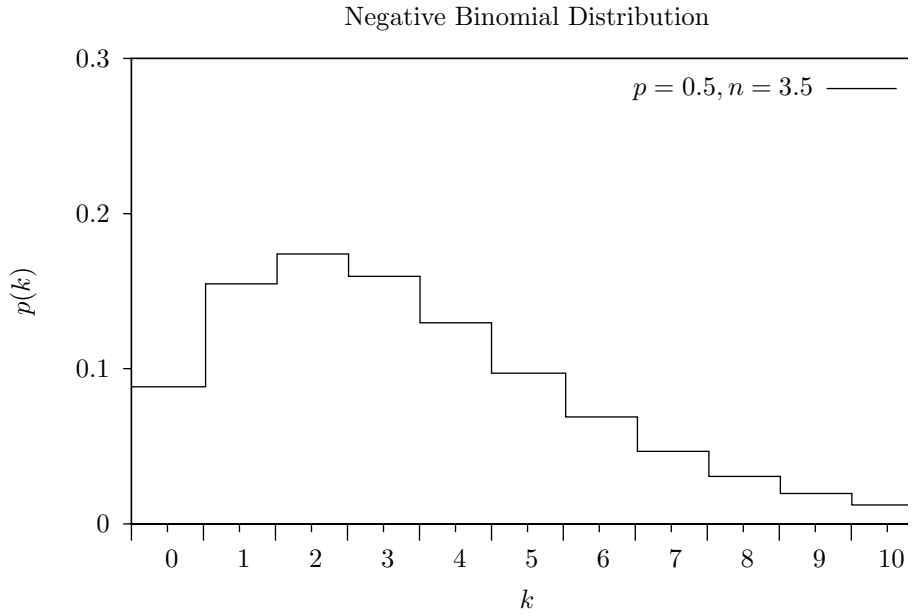
[Function] unsigned int gsl_ran_negative_binomial (const gsl_rng * r, double p, double n)

負の二項分布、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。

$$p(k) = \frac{\Gamma(n+k)}{\Gamma(k+1)\Gamma(n)} p^n (1-p)^k$$

[Function] double gsl_ran_negative_binomial_pdf (unsigned int k, double p, double n)

上の式のパラメータが 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。



[Function] double gsl_cdf_negative_binomial_P (unsigned int k, double p, double n)

[Function] double gsl_cdf_negative_binomial_Q (unsigned int k, double p, double n)

上の式のパラメータが p および 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

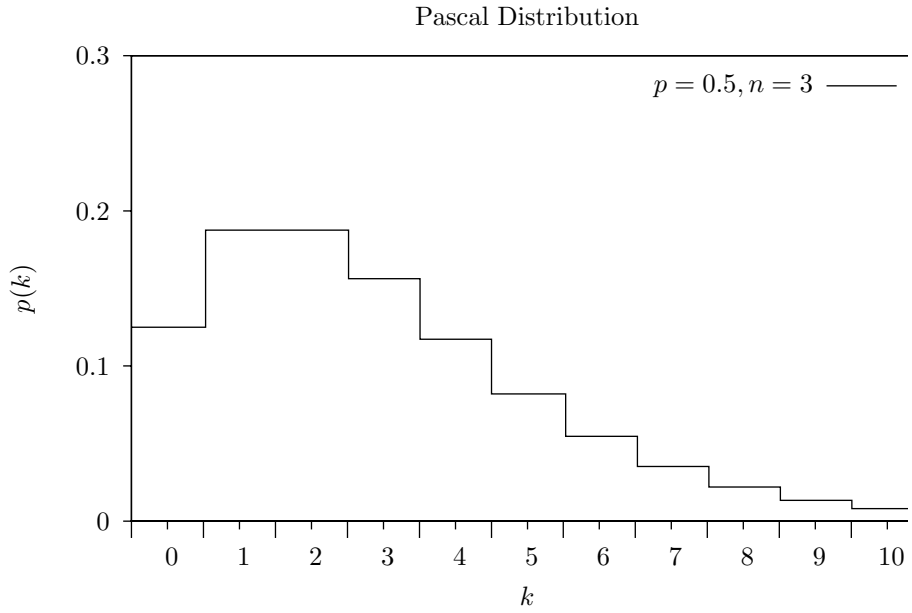
[Function] unsigned int gsl_ran_pascal (const gsl_rng * r, double p, unsigned int n)

パスカル分布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 ≧ 0 について以下で定義される。

$$p(k) = \frac{(n+k-1)!}{k!(n-1)!} p^n (1-p)^k$$

[Function] double gsl_ran_pascal_pdf (unsigned int k, double p, unsigned int n)

上の式のパラメータが 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。



[Function] double gsl_cdf_pascal_P (unsigned int k, double p, unsigned int n)

[Function] double gsl_cdf_pascal_Q (unsigned int k, double p, unsigned int n)

上の式のパラメータが p および 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

[Function] unsigned int gsl_ran_geometric (const gsl_rng * r, double p)

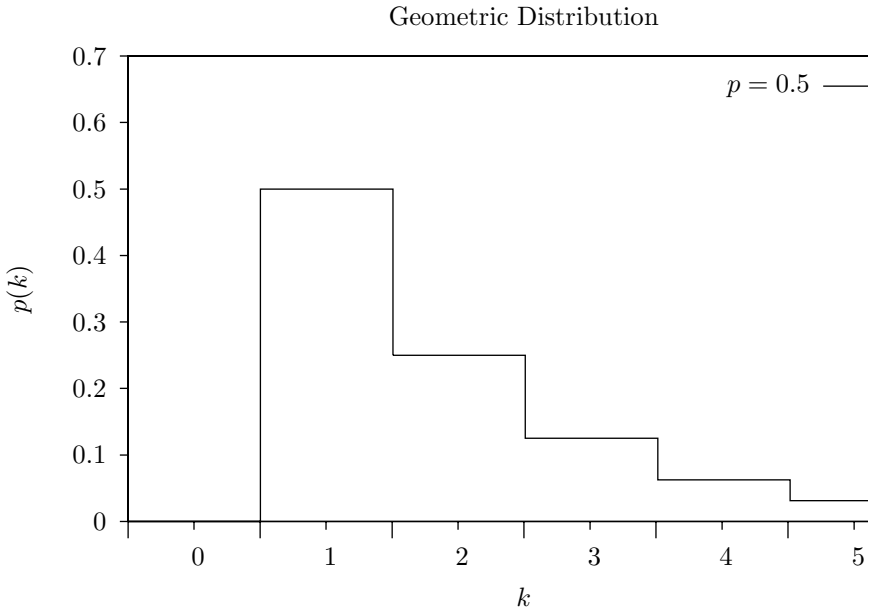
幾何分布、つまり試行が成功する確率が p であるときに、最初に成功するまでに要する試行の回数の分布にしたがう整数の乱数を返す。この分布は k ≧ 1 で以下で定義される。

$$p(k) = p(1-p)^{k-1}$$

個の定義では、k = 1 から始まるが、指数を k - 1 ではなく k とすることもある。

[Function] double gsl_ran_geometric_pdf (unsigned int k, double p)

上の式のパラメータが 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。



[Function] double gsl_cdf_geometric_P (unsigned int k, double p)

[Function] double gsl_cdf_geometric_Q (unsigned int k, double p)

上の式のパラメータが 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

[Function] unsigned int gsl_ran_hypergeometric (const gsl_rng * r, unsigned int n1, unsigned int n2, unsigned int t)

超幾何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。

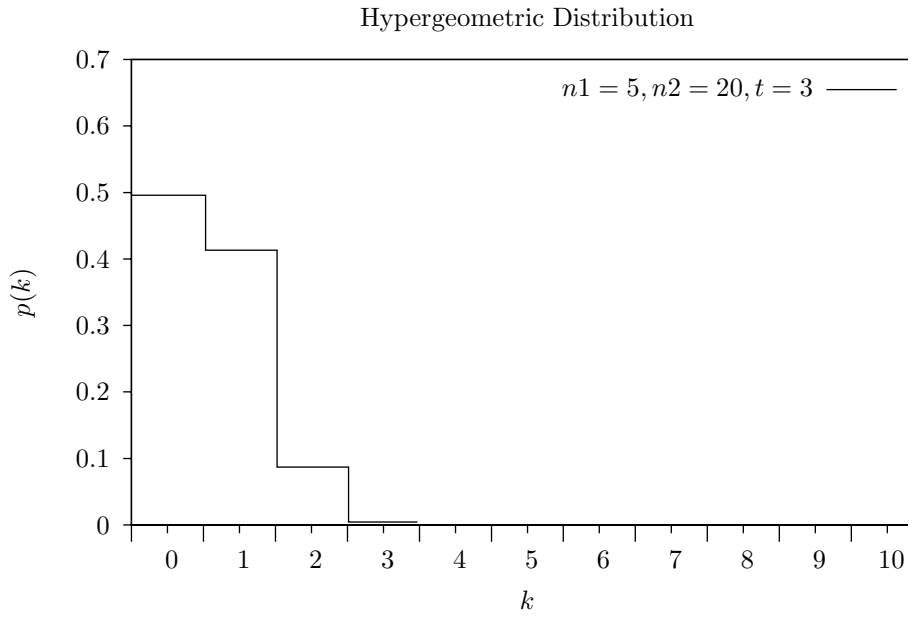
$$p(k) = C(n_1, k)C(n_2, t-k)/C(n_1+n_2, t)$$

ここで C(a, b) = a! /(b!(a - b)!) で t ≦ n_1 + n_2 である。k の範囲は max(0, t - n_2)), ... , 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。

[Function] double gsl_ran_hypergeometric_pdf (unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

上の式のパラメータが n1、n2、k 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。



[Function] double gsl_cdf_hypergeometric_P (unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

[Function] double gsl_cdf_hypergeometric_Q (unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

上の式のパラメータが n1、n2、k 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

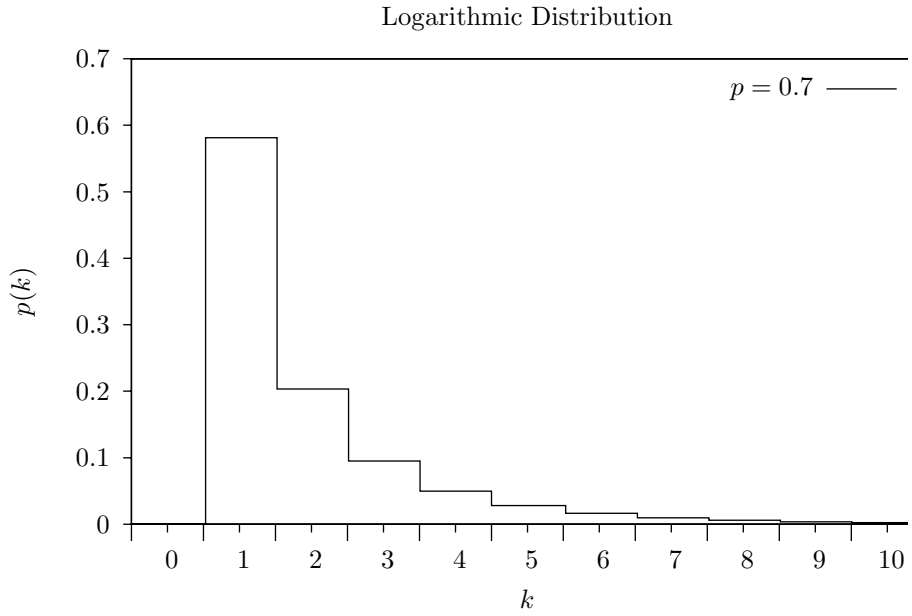
[Function] unsigned int gsl_ran_logarithmic (const gsl_rng * r, double p)

対数分布にしたがう整数の乱数を返す。この分布は k ≧ 1 で以下で定義される。

$$p(k) = \frac{-1}{\log(1-p)} \left(\frac{p^k}{k} \right)$$

[Function] double gsl_ran_logarithmic_pdf (unsigned int k, double p)

上の式のパラメータが p で与えられる対数何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、取り出して観測するためのものである。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生じ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。詳しくはクヌースの第二巻第三版 3.4.2 節、"Random Sampling and Shuffiling" を参照のこと。

[Function] void gsl_ran_shuffle (const gsl_rng * r, void * base, size_t n, size_t size)

それぞれサイズが size で配列 base[0..n - 1] に入れて渡される n 個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。

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));
```

[Function] int gsl_ran_choose (const gsl_rng * r, void * dest, size_t k, void * src, size_t n, size_t size)

src[0 .. n - 1] に入れて渡される k 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。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。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, n, 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値を 3 つ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));
```

[Function] void gsl_ran_sample (const gsl_rng * r, void * dest, size_t k, void * src, size_t n, size_t size)

`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。ここでは平均値が 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;
```

```
    /* create a generator chosen by the environment variable
       GSL_RNG_TYPE */
    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc(T);

    /* print n random variates chosen from the poisson
       distribution with mean parameter mu */
    for (i = 0; i < n; i++) {
         unsigned int k = gsl_ran_poisson(r, mu);
         printf(" %u", k);
    }
    printf("\n");

    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
```

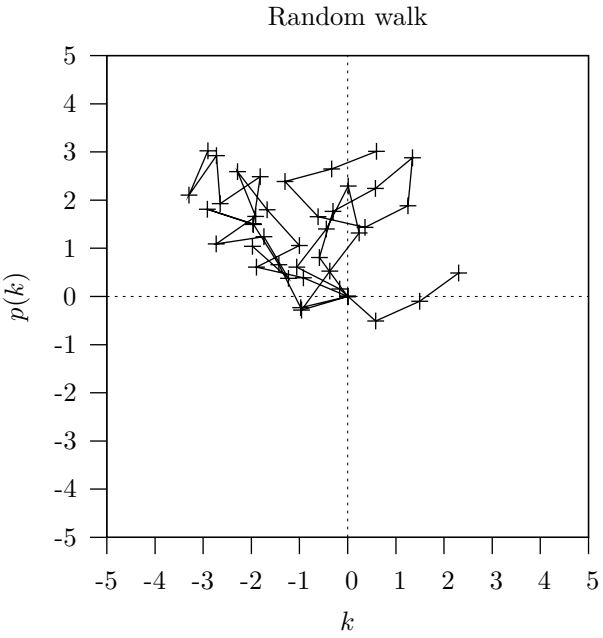
発生する乱数の値は、乱数発生器に与える種によって変化する。使用する乱数生成器のデフォルトは変数 gsl_rng_default の値に指定されており、以下のようにすれば環境変数 GSL_RNG_SEED でその乱数発生器によって生成される乱数系列の種を変更できる。

```
$ 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);
    }

    return 0;
}
```

以下に、原点から 10 ステップのランダム・ウォークを行った結果を示す。



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

幅広い分野をカバーする事典のような本として、リュック・デブロイの Non-Uniform Random Variate Generation をお勧めする。この本には想像できる限りのさまざまな分布と膨大なアルゴリズムが載っている。

・ Luc Devroye, Non-Uniform Random Variate Generation, Springer-Verlag, ISBN 0-387-96305-7.

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。

- Donald E. Knuth, The Art of Computer Programming: Seminumerical Algorithms (Vol 2, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896842.

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いる。

- Review of Particle Properties R.M. Barnett et al., Physical Review D54, 1 (1996)
- http://pdg.lbl.gov/

このレビューの 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。

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による Statistical Computing の記事に概観されている。ほかにシステッドの Elements of Statistical Computing の記事にもレビューがある。

- William E. Kennedy and James E. Gentle, Statistical Computing (1980), Marcel Dekker, ISBN 0-8247-6898-1.
- Ronald A. Thisted, Elements of Statistical Computing (1988), Chapman & Hall, ISBN 0-412-01371-1.

正規分布の累積分布関数は、以下の論文を元にしている。

- Rational Chebyshev Approximations Using Linear Equations, W.J. Cody, W. Fraser, J.F. Hart. Numerische Mathematik 12, 242–251 (1968).
- Rational Chebyshev Approximations for the Error Function, W.J. Cody. Mathematics of Computation 23, n107, 631–637 (July 1969).

第 20 章 統計

この章では、ライブラリに用意している統計に関する関数について説明する。基本的な平均値、分散、標準偏差を計算するようなものに加え、絶対偏差、ひずみ度、尖度、中央値、任意の百分位数を計算するものもある。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なるような事態を避けるために、再帰を使って計算する。

ここ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

[Function] double gsl_stats_mean (const double data [], size_t stride, size_t n)

データ長 n で飛び幅 stride のデータセット data の算術平均を返す。算術平均は標本平均とも呼ばれ、以下の μ として定義される。

$$\mu = \frac{1}{N} \sum x_i$$

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、μ の分散は σ^2/N になる。

[Function] double gsl_stats_variance (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data の推定分散、または標本分散を返す。推定分散は以下の σ^2 で定義される。

$$\sigma^2 = \frac{1}{N-1} \sum (x_i - \mu)^2$$

ここで x_i はデータセット data の要素である。正規化係数の 1/(N − 1) をかけることで推定分散は母分散 σ^2 のバイアスのない推定値になる。サンプルが正規分布に従うとき、σ^2 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、その値を直接 `gsl_stats_variance_m` に渡してもよい。

[Function] double gsl_stats_variance_m (const double data [], size_t stride, size_t n, double mean)

与えられる平均値 mean に対する標本分散を返す。この関数は利用者が指定する平均値で μ を置き換えて分散を計算する。

$$\sigma^2 = \frac{1}{N-1} \sum (x_i - mean)^2$$

[Function] double gsl_stats_sd (const double data [], size_t stride, size_t n)

[Function] double gsl_stats_sd_m (const double data [], size_t stride, size_t n, double mean)

分散の平方根を計算して標準偏差として返す。上述した分散のうち、それぞれの関数が対応するものを返す。

[Function] double gsl_stats_variance_with_fixed_mean (const double data [], size_t stride, size_t

n, double mean)

母集団の分布とその平均があらかじめ分かっている場合に、data の分散のバイアスのない推定値を返す。この関数は正規化係数として 1/N を使い、標本平均 μ として既知の母集団の平均値を使う。

$$\sigma^2 = \frac{1}{N-1}\sum(x_i - \mu_{known})^2$$

[Function] double gsl_stats_sd_with_fixed_mean (const double data [], size_t stride, size_t n, double mean)

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

[Function] double gsl_stats_absdev (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data の平均から、絶対偏差を計算して返す。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。分散よりも平均値からの絶対偏差のほうが、データの散らばり具合を表す指標としては優れている。内部で `gsl_stats_mean` を呼び出して data の平均値を計算する。

[Function] double gsl_stats_absdev_m (const double data [], size_t stride, size_t n, double mean)

与えられる平均値 mean からの、データ長 n、飛び幅 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。また違う平均値（たとえば零や中央値など）を使って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(ひずみ度と尖度)

[Function] double gsl_stats_skew (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data のひずみ度を返す。ひずみ度は以下で定義される。

$$skew = \frac{1}{N}\sum\left(\frac{x_i - \hat{\mu}}{\hat{\sigma}}\right)^2$$

ここで x_i はデータセット data の要素である。ひずみ度はデータの分布の裾の非対称性の大きさを表す。

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。

[Function] double gsl_stats_skew_m_sd (const double data [], size_t stride, size_t n, double mean, double sd)

データ長 n、飛び幅 stride のデータセット data のひずみ度を、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。

$$skew = \frac{1}{N} \sum \left(\frac{x_i - mean}{sd} \right)^2$$

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

[Function] double gsl_stats_kurtosis (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data の尖度を返す。尖度は以下で定義される。

$$kurtosis = \left(\frac{1}{N} \sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^4 \right) - 3$$

尖度は、分布の幅に対してそのピークがどの程度鋭く尖っているかを表す。尖度は正規分布で零になるように正規化されている。

[Function] double gsl_stats_kurtosis_m_sd (const double data [], size_t stride, size_t n, double mean, double sd)

データ長 n、飛び幅 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。

$$kurtosis = \frac{1}{N} \left(\sum \left(\frac{x_i - mean}{sd} \right)^4 \right) - 3$$

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

[Function] double gsl_stats_lag1_autocorrelation (const double data [], const size_t stride, const size_t n)

以下で与えられる、データセット data の、すれが１ １ の自己相関係数を計算する。

$$a_1 = \frac{\sum_{i=1}^{n} (x_i - \hat{\mu})(x_{i-1} - \hat{\mu})}{\sum_{i=1}^{n} (x_i - \hat{\mu})(x_i - \hat{\mu})}$$

[Function] double gsl_stats_lag1_autocorrelation_m (const double data [], const size_t stride, const size_t n, const double mean)

与えられる平均値 mean で、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

[Function] double gsl_stats_covariance (const double data1 [], const size_t stride1, const double data2 [], const size_t stride2, const size_t n)

同じデータ長 n を持つ二つのデータセット data1 と data2 の共分散を計算する。

$$covar = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

[Function] double gsl_stats_covariance_m (const double data1 [], const size_t stride1, const double data2 [], const size_t n, const double mean1, const double mean2)

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 重み付きデータ

この節で説明する関数は重み付きサンプリングに関する処理を行うためのである。関数にはサンプル配列 x_i と対応する重みの配列 w_iを渡す。各サンプル x_iの分布は分散 σ_i^2の正規分布に従うと考え、重み w_i はその分散の逆数 $w_i = 1/\sigma_i^2$として定義される。重みが零とされたサンプルはデータセットから除去されたのと同じ事になる。

[Function] double gsl_stats_wmean (const double w [], size_t wstride, const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data の重み付き平均値を、重み w、重みの飛び幅 wstride 、データ長 n を使って計算する。重み付き平均の値は以下で定義される。

$$\hat{\mu} = \frac{\sum w_i x_i}{\sum w_i}$$

[Function] double gsl_stats_wvariance (const double w [], size_t wstride, const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data の推定分散を、重み w、重みの飛び幅 wstride、データ長 n を使って計算して返す。重み付きの推定分散は以下で定義される。

$$\hat{\sigma}^2 = \frac{\sum w_i}{(\sum w_i)^2 - \sum(w_i^2)}\sum w_i(x_i - \hat{\mu})^2$$

大きさの等しい重みが N 個あるとき、この式による重み付き分散は、1/(N - 1) を重みのない分散にかけたものと等しくなる。

[Function] double gsl_stats_wvariance_m (const double w [], size_t wstride, const double data [], size_t stride, size_t n, double wmean)

データ長 n、飛び幅 stride のデータセット data の推定分散を、与えられた重み付き平均値 mean を使って計算して返す。

[Function] double gsl_stats_wsd (const double w [], size_t wstride, const double data [], size_t stride, size_t n)

標準偏差は分散の平方根として定義される。この関数は対応する推定分散を計算する上述の関数 `gsl_stats_wvariance` の値の平方根を返す。

[Function] double gsl_stats_wsd_m (const double w [], size_t wstride, const double data [], size_t stride, size_t n, double wmean)

この関数は対応する推定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。

[Function] double gsl_stats_wvariance_with_fixed_mean (const double w [], size_t wstride, const double data [], size_t stride, size_t n, const double mean)

母集団の分布の平均値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 の不偏推定分散の値を計算する。この場合は以下のように、標本平均を与えられる既知の平均値 μ に置き換えて計算を行う。

$$\hat{\sigma}^2 = \frac{\sum w_i (x_i - \mu)^2}{\sum w_i}$$

[Function] double gsl_stats_wsd_with_fixed_mean (const double w [], size_t wstride, const double data [], size_t stride, size_t n, const double mean)

標準偏差は分散の平方根として定義される。この関数は対応する分散を計算する上述の関数の値の平方根を返す。

[Function] double gsl_stats_wabsdev (const double w [], size_t wstride, const double data [], size_t stride, size_t n)

重み付きデータセット data の絶対偏差を計算する。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。

$$absdev = \frac{\sum w_i |x_i - \hat{\mu}|}{\sum w_i}$$

[Function] double gsl_stats_wabsdev_m (const double w [], size_t wstride, const double data [], size_t stride, size_t n, double wmean)

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平均値 wmean を使って計算する。

[Function] double gsl_stats_wskew (const double w [], size_t wstride, const double data [], size_t stride, size_t n)

以下で与えられる、データセット data の重み付きひずみ度を計算する。

$$skew = \frac{\sum w_i ((x_i - xbar)/\sigma)^3}{\sum w_i}$$

[Function] double gsl_stats_wskew_m_sd (const double w [], size_t wstride, const double data [], size_t stride, size_t n, double wmean, double wsd)

データセット data の重み付きひずみ度を、与えられる重み付き平均値 wmean と重み付き標準偏差 wsd を使って計算する。

[Function] double gsl_stats_wkurtosis (const double w [], size_t wstride, const double data [], size_t stride, size_t n)

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。

$$kurtosis = \frac{\sum w_i ((x_i - xbar)/\sigma)^4}{\sum w_i} - 3$$

[Function] double gsl_stats_wkurtosis_m_sd (const double w [], size_t wstride, const double data [], size_t stride, size_t n, double wmean, double wsd)

データセット data の重み付き尖度を、与えられる重み付き平均値 wmean と重み付き標準偏差 wsd を使って計算する。

20.7 最大値、最小値

以下に、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）を探して返す関数を説明する。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。

[Function] double gsl_stats_max (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data の最大値を返す。要素 x_i の最大値は、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。

[Function] double gsl_stats_min (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータセット data の最小値を返す。要素 x_i の最小値は、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。

[Function] void gsl_stats_minmax (double * min, double * max, const double data [], size_t stride, size_t n)

データセット data から最小値 min と最大値 max を一度に探す。

[Function] size_t gsl_stats_max_index (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 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。要素 x_i の最大値は、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。

[Function] size_t gsl_stats_min_index (const double data [], size_t stride, size_t n)

データ長 n、飛び幅 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。要素 x_i の最小値は、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。

[Function] void gsl_stats_minmax_index (size_t * min_index, size_t * max_index, const double data [], size_t stride, size_t n)

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 を一度に探す。

20.8 中央値と百分位数

ここでは、あらかじめ整列されたデータに対する中央値と百分位数に関する関数について説明する。ここでは便宜上、百分位数（percentile、0 から 100 までの範囲）ではなく分位数（quantile、0 から 1 まで）を用いている。

[Function] double gsl_stats_median_from_sorted_data (const double sorted_data [], size_t stride, size_t n)

データ長 n、飛び幅 stride のデータ sorted data の中央値（median）を返す。配列中のデータは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、先に gsl_sort 関数などで整列しておくとよい。

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。つまりここで使っている中央値を求めるアルゴリズムでは補間を行うため、データが整数の場合でも返値は浮動小数点実数である。

[Function] double gsl_stats_quantile_from_sorted_data (const double sorted_data [], size_t stride, size_t n, double f)

データ長 n、飛び幅 stride の倍精度実数のデータ sorted_data の分位数を返す。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くとよい。

分位数（quantile）は以下の公式を使って計算される。

$$\text{quantile} = (1 - \delta)x_i + \delta x_{i+1}$$

ここで i は floor((n – 1)f) で、δ は (n – 1)f – i である。

配列の最小値（`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。このアルゴリズムでは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浮動小数点実数である。

20.9 例

統計関数を使う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
```

20.10 参考文献

統計に関するほとんど全ての分野について、以下のシリーズに述べられている。

・ Maurice Kendall, Alan Stuart, and J. Keith Ord. The Advanced Theory of Statistics (multiple volumes) reprinted as Kendall's Advanced Theory of Statistics. Wiley, ISBN 047023380X.

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.

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。

・ Review of Particle Properties R.M. Barnett et al., Physical Review D54, 1 (1996)

このレビューは、 http://pdg.lbl.gov/ で自由に読むことができる。

第 21 章 ヒストグラム

この章ではヒストグラム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からなり、各階級は幅が連続値をとる変数 x である小区間であり、そこに分類される事象の数が各階級の値になる。GSL では階級値は浮動小数点実数であり、したがって整数と非整数の両方の確率分布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に変換することができる。その確率分布に従う乱数を生成するルーチンをこのライブラリでは用意しており、現実のデータを用いたシミュレーションを行うときに有用である。

この章で説明する関数はすべて ‘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。

[Data Type] gsl_histogram

`size_t n`
: 階級数

`double * range`
: 各階級の幅。ポインタ `range` が指す要素数 `n+1` の配列に保持される。

`double * bin`
: 各階級の度数。ポインタ `bin` が指す要素数 `n` の配列に保持される。階級の度数は浮動小数点実数なので、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]` の範囲は `range [i]` 以上 `range [i+1]` 未満である。つまり以下のようになる。

`bin[i]` corresponds to `range[i]` ≤ x < `range[i+1]`

各階級と範囲の関係を変数 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 ≤ 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サンプル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、そのための階級を加える必要がある。

`gsl_histogram` 構造体とそれ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、ヒストグラムの alloc を行うたびにエラーチェックを行う必要はない。

[Function] gsl_histogram * gsl_histogram_alloc (size_t n)

n 階級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。階級の度数とその幅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以下の関数を使って初期化する必要がある。

[Function] int gsl_histogram_set_ranges (gsl_histogram * h, const double range [], size_t size)

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。各階級の度数には零が入れられる。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。

[Function] int gsl_histogram_set_ranges_uniform (gsl_histogram * h, double xmin, double xmax)

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。各階級の度数には零が入れられる。階級の幅は以下のようになる。

```
bin[0]   corresponds to xmin ≤ x < xmin + d
bin[1]   corresponds to xmin + d ≤ x < xmin + 2d
.........
bin[n-1] corresponds to xmin + (n − 1)d ≤ x < xmax
```

ここで d は階級の幅 d = (xmax − xmin)/n である。

[Function] void gsl_histogram_free (gsl_histogram * h)

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。

21.3 ヒストグラムの複製

[Function] int gsl_histogram_memcpy (gsl_histogram * dest, const gsl_histogram * src)

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。

[Function] gsl_histogram * gsl_histogram_clone (const gsl_histogram * src)

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するには、x 軸の値を指定する方法と、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行って決定する。

[Function] int gsl_histogram_increment (gsl_histogram * h, double x)

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、返り値は零になり、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。同様に x が上限値以上のときも GSL_EDOM を返し、どの階級も変化しない。データセットが大きく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ので、どちらの場合もエラーハンドラーは呼ばれず、単にそういった値は無視されるだけである。

[Function] int gsl_histogram_accumulate (gsl_histogram * h, double x, double weight)

階級に 1 ではなく weight を加えること以外は `gsl_histogram_increment` と同じである。

[Function] double gsl_histogram_get (const gsl_histogram * h, size_t i)

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。

[Function] int gsl_histogram_get_range (const gsl_histogram * h, size_t i, double * lower, double * upper)

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 が有効であれば、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。

[Function] 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)

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。

[Function] void gsl_histogram_reset (gsl_histogram * h)

ヒストグラム h の全ての階級値を零にする。

21.5 ヒストグラムでの範囲の検索

ある x の値を範囲に含む階級の値を見たり、値を変更するのに以下の関数を使うことができる。

[Function] int gsl_histogram_find (const gsl_histogram * h, double x, size_t * i)

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。探索は二分探索で行われ、階級の幅が等間隔の場合について最適化されているので、その場合はすぐに答えを返す。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 ヒストグラムに対する統計

[Function] double gsl_histogram_max_val (const gsl_histogram * h)

ヒストグラム h のすべての階級中で最大の度数を返す。

[Function] size_t gsl_histogram_max_bin (const gsl_histogram * h)

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。

[Function] double gsl_histogram_min_val (const gsl_histogram * h)

ヒストグラム h のすべての階級中で最小の度数を返す。

[Function] size_t gsl_histogram_min_bin (const gsl_histogram * h)

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。

[Function] double gsl_histogram_mean (const gsl_histogram * h)

ヒストグラムが確率分布にしたがうとして、ヒストグラムに数えられている変数の平均値を返す。階級の度数が負の場合、その階級は無視される。精度は階級の幅に依存する。

[Function] double gsl_histogram_sigma (const gsl_histogram * h)

ヒストグラムが確率分布にしたがうとして、ヒストグラムに数えられている変数の標準偏差を返す。階級の度数が負の場合、その階級は無視される。精度は階級の幅に依存する。

[Function] double gsl_histogram_sum (const gsl_histogram * h)

全ての階級の度数の和を返す。度数が負の場合も加えられる。

21.7 ヒストグラムの操作

[Function] int gsl_histogram_equal_bins_p (const gsl_histogram * h1, const gsl_histogram * h2)

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 を、そうでなければ 0 を返す。

[Function] int gsl_histogram_add (gsl_histogram * h1, const gsl_histogram * h2)

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(i) = h1(i) + h2(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。

[Function] int gsl_histogram_sub (gsl_histogram * h1, const gsl_histogram * h2)

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(i) = h1(i)-h2(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。

[Function] int gsl_histogram_mul (gsl_histogram * h1, const gsl_histogram * h2)

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(i) = h1(i)×h2(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。

[Function] int gsl_histogram_div (gsl_histogram * h1, const gsl_histogram * h2)

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(i) = h1(i)/h2(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。

[Function] int gsl_histogram_scale (gsl_histogram * h, double scale)

ヒストグラム h の各階級の度数に、h(i) = h1(i)×scale のようにして scale を乗ずる。

[Function] int gsl_histogram_shift (gsl_histogram * h, double offset)

ヒストグラム h の各階級の度数を、h(i) = h1(i) + offset として増減する。

21.8 ヒストグラムの読み込みと書き出し

このライブラリでは、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が用意している。

[Function] int gsl_histogram_fwrite (FILE * stream, const gsl_histogram * h)

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。

[Function] int gsl_histogram_fread (FILE * stream, gsl_histogram * h)

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。

[Function] int gsl_histogram_fprintf (FILE * stream, const gsl_histogram * h, const char * range_format, const char * bin_format)

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で操作しやすくするために、こういった形式になっている。

[Function] int gsl_histogram_fscanf (FILE * stream, gsl_histogram * h)

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、確率分布を持つ事象を数えることで成立する。統計上の誤差を許して考えると、各階級の高さは変数 x の値がその階級の持つ範囲に入るような確率を表すと言える。その確率分布関数は一変数関数 p(x)dx として表され、ここで

$$p(x) = n_i / (N w_i)$$

となる。ここで n_i は x を範囲に含む階級での事象の発生個数、w_i は階級の幅、N は事象の全個数である。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

ヒストグラムの確率分布関数は、連続値の変数 x がどの範囲に発生するかをはかるための小区間の集合である階級からなる。確率分布関数は、以下の構造体に保持される累積分布関数として定義される。累積分布関数は範囲 [0,1] 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自然なやり方が使える。 [0,1] の範囲で一様乱数を発生させ、これに対応する累積分布関数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。

[Data Type] gsl_histogram_pdf

`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確率。

この確率分布にしたがって乱数でサンプルを発生させる `gsl_histogram_pdf` 構造体を以下の関数で生成できる。

[Function] gsl_histogram_pdf * gsl_histogram_pdf_alloc (size_t n)

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。

[Function] int gsl_histogram_pdf_init (gsl_histogram_pdf * p, const gsl_histogram * h)

ヒストグラム h に基づく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ので、エラーコード GSL_EDOM でエラーハンドラーを呼び出す。

[Function] void gsl_histogram_pdf_free (gsl_histogram_pdf * p)

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。

[Function] double gsl_histogram_pdf_sample (const gsl_histogram_pdf * p, double r)

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]` ≦ r < `sum[i+1]` となる添え字で、δ が比 (r − `sum[i]`)/(`sum[i+1]` − `sum[i]`) であるとき、以下の公式で与えられる。

$$s = \texttt{range[i]} + \delta \times (\texttt{range[i + 1]} - \texttt{range[i]})$$

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。

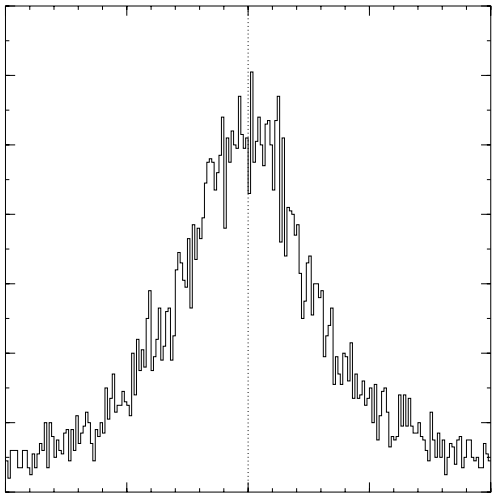
```
#include <stdio.h>
#include <stdlib.h>
#include <gsl/gsl_histogram.h>

int main (int argc, char **argv)
{
    double a, b, x;
    size_t n;

    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]);

    gsl_histogram * h = gsl_histogram_alloc(n);
    gsl_histogram_set_ranges_uniform(h, a, b);

    while (fscanf(stdin, "%lg", &x) == 1)
        gsl_histogram_increment(h, x);

    gsl_histogram_fprintf(stdout, h, "%g", "%g");
    gsl_histogram_free(h);

    return 0;
}
```

以下にプログラムの実行例を示す。幅 30 のコーシー分布に従う乱数を 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 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、サンプル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一組の階級からなる。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なす。

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。

[Data Type] gsl_histogram2d

`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。

bin(i, j) corresponds to `xrange[i]` ≤ x < `xrange[i+1]`
and `yrange[j]` ≤ y < `yrange[j+1]`

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` 構造体とこれに関する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、ヒストグラムの `alloc` を行うたびにエラーチェックを行う必要はない。

[Function] gsl_histogram2d * gsl_histogram2d_alloc (size_t nx, size_t ny)

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。

[Function] int gsl_histogram2d_set_ranges (gsl_histogram2d * h, const double xrange [], size_t xsize, const double yrange [], size_t ysize)

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 xrange と yrange 、階級の数 xsize と ysize を設定する。各階級の度数には零が入れられる。

[Function] int gsl_histogram2d_set_ranges_uniform (gsl_histogram2d * h, double xmin, double xmax, double ymin, double ymax)

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。各階級の度数には零が入れられる。

[Function] void gsl_histogram2d_free (gsl_histogram2d * h)

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

[Function] int gsl_histogram2d_memcpy (gsl_histogram2d * dest, const gsl_histogram2d * src)

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。

[Function] gsl_histogram2d * gsl_histogram2d_clone (const gsl_histogram2d * src)

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。

[Function] int gsl_histogram2d_increment (gsl_histogram2d * h, double x, double y)

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、返り値は零になり、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ない。データセットが大きく各階級の幅が小さいような場合には探索に時間がかかることがあるので、どちらの場合もエラーハンドラーは呼ばれず、単にそういった値は無視されるだけである。

[Function] int gsl_histogram2d_accumulate (gsl_histogram2d * h, double x, double y, double weight)

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。

[Function] double gsl_histogram2d_get (const gsl_histogram2d * h, size_t i, size_t j)

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。

[Function] 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)

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 upper 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。

[Function] 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)

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_histogram の要素を直接見なくてもすむ。

[Function] void gsl_histogram2d_reset (gsl_histogram2d * h)

ヒストグラム h の全ての階級値を零にする。

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。

[Function] int gsl_histogram2d_find (const gsl_histogram2d * h, double x, double y, size_t * i, size_t * j)

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。探索は二分探索で行われ、階級の幅が等間隔の場合について最適化されているので、その場合はすぐに答えを返す。(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での統計

[Function] double gsl_histogram2d_max_val (const gsl_histogram2d * h)

ヒストグラム h のすべての階級中で最大の度数を返す。

[Function] void gsl_histogram2d_max_bin (const gsl_histogram2d * h, size_t * i, size_t * j)

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。

[Function] double gsl_histogram2d_min_val (const gsl_histogram2d * h)

ヒストグラム h のすべての階級中で最小の度数を返す。

[Function] void gsl_histogram2d_min_bin (const gsl_histogram2d * h, size_t * i, size_t * j)

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。

[Function] double gsl_histogram2d_xmean (const gsl_histogram2d * h)

ヒストグラムが確率分布にしたがうとして、ヒストグラムの x 変数の平均値を返す。階級の度数が負の場合、その階級は無視される。

[Function] double gsl_histogram2d_ymean (const gsl_histogram2d * h)

ヒストグラムが確率分布にしたがうとして、ヒストグラムの y 変数の平均値を返す。階級の度数が負の場合、その階級は無視される。

[Function] double gsl_histogram2d_xsigma (const gsl_histogram2d * h)

ヒストグラムが確率分布にしたがうとして、ヒストグラムの 変数の標準偏差を返す。階級の度数が負の場合、その階級は無視される。

[Function] double gsl_histogram2d_ysigma (const gsl_histogram2d * h)

ヒストグラムが確率分布にしたがうとして、ヒストグラムの y 変数の標準偏差を返す。階級の度数が負の場合、その階級は無視される。

[Function] double gsl_histogram2d_cov (const gsl_histogram2d * h)

ヒストグラムが確率分布にしたがうとして、ヒストグラムの x と y 変数の共分散を返す。階級の度数が負の場合、その階級は無視される。

[Function] double gsl_histogram2d_sum (const gsl_histogram2d * h)

全ての階級の度数の和を返す。度数が負の場合も加えられる。

21.19 二次元ヒストグラムの操作

[Function] int gsl_histogram2d_equal_bins_p (const gsl_histogram2d * h1, const gsl_histogram2d * h2)

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。

[Function] int gsl_histogram2d_add (gsl_histogram2d * h1, const gsl_histogram2d * h2)

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(i, j) = h1(i, j) + h2(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。

[Function] int gsl_histogram2d_sub (gsl_histogram2d * h1, const gsl_histogram2d * h2)

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(i, j) = h1(i, j) − h2(i, j) のようにして引く。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。

[Function] int gsl_histogram2d_mul (gsl_histogram2d * h1, const gsl_histogram2d * h2)

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(i, j) = h1(i, j) ∗ h2(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。

[Function] int gsl_histogram2d_div (gsl_histogram2d * h1, const gsl_histogram2d * h2)

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(i, j) = h1(i, j) /h2(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。

[Function] int gsl_histogram2d_scale (gsl_histogram2d * h, double scale)

ヒストグラム h の各階級の度数に h(i, j) = h1(i, j) ∗ scale のようにして scale を乗ずる。

[Function] int gsl_histogram2d_shift (gsl_histogram2d * h, double offset)

ヒストグラム h の各階級の度数に h(i, j) = h1(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読み込みと書き出し

このライブラリでは、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されている。

[Function] int gsl_histogram2d_fwrite (FILE * stream, const gsl_histogram2d * h)

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。

[Function] int gsl_histogram2d_fread (FILE * stream, gsl_histogram2d * h)

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。

[Function] int gsl_histogram2d_fprintf (FILE * stream, const gsl_histogram2d * h, const char * range_format, const char * bin_format)

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で操作しやすくするために、こういった形式になっている。

[Function] int gsl_histogram2d_fscanf (FILE * stream, gsl_histogram2d * h)

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。統計的な誤差を考え、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は以下のように p(x, y)dxdy の形で表される。

$$p(x, y) = n_{ij} /(N A_{ij})$$

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。

[Data Type] gsl_histogram2d_pdf

`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 `nx+1` 要素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 `ny+1` 要素の配列に保持される。

`double * sum`
: ポインタ `sum` が差す `nx * ny` 個の要素を持つ配列に保持される累積確率。

この確率分布にしたがって乱数でサンプルを発生させる `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。

[Function] gsl_histogram2d_pdf * gsl_histogram2d_pdf_alloc (size_t nx, size_t ny)

階級数 nx × 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。

[Function] int gsl_histogram2d_pdf_init (gsl_histogram2d_pdf * p, const gsl_histogram2d * h)

ヒストグラム h に基づく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ので、エラーコード GSL_EDOM でエラーハンドラーを呼び出す。

[Function] void gsl_histogram2d_pdf_free (gsl_histogram2d_pdf * p)

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。

[Function] int gsl_histogram2d_pdf_sample (const gsl_histogram2_ pdf * p, double r1, double r2, double * x, double * y)

零から 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 にしたがうサンプル s を一つ発生する。

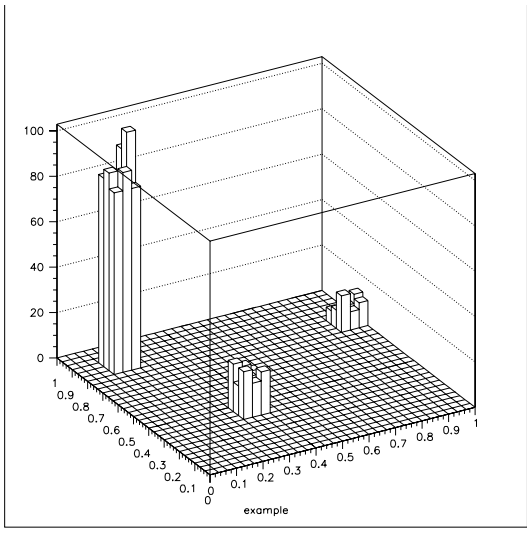
21.22 二次元ヒストグラムのプログラム例

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、10x10 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 のサンプルを発生する。3 つのサンプルを加えたこのヒストグラムから、1000 個のランダムサンプル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)
{
    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);

    int i;
    gsl_histogram2d_pdf * p
        = gsl_histogram2d_pdf_alloc(h->nx, h->ny);
    gsl_histogram2d_pdf_init(p, h);

    for (i = 0; i < 1000; i++) {
        double x, y;
        double u = gsl_rng_uniform(r);
        double v = gsl_rng_uniform(r);
        gsl_histogram2d_pdf_sample(p, u, v, &x, &y);
        printf ("%g %g\n", x, y);
    }

    return 0;
}
```

出力されたランダムサンプルのプロットを以下に示す。解像度を上げることで標本生成に使ったヒストグラムと、その階級の範囲内では一様分布とすることによる分布のゆがみを見ることができる。



第 22 章 N 項組

この章では N 項組（N タプル、N-tuple)、つまり事象に関連づけられた値の集合ををを生成、操作する関数について説明する。N 項組はファイルに保存される。その値はあらゆる組合せで取り出すことができ、選択関数を使ってヒストグラムに書き入れることができる。

ファイルに保存される値のデータ構造は利用者が定義し、そのデータ構造にしたがって N 項組が生成される。値は以下の関数でそのファイルに（普通はループで）保存される。

選択関数と値関数を使うことで、N 項組からヒストグラムを作ることができる。選択関数は、事象が解析対象となる部分集合に含まれるかどうかを指定する。値関数は各事象がヒストグラムのどこに数えられるかを計算する。

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に関する情報、N 項組の参照時点でのデータ行へのポインタ、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 項組の生成

[Function] gsl_ntuple * gs_ntuple_create (char * filename, void * ntuple_data, size_t size)

大きさ size の N 項組のファイル filename を書き込みのみモードで生成し、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零になり、上書きされる。現在の N 項組の行 ntuple data へのポインタを指定する必要がある。このポインタはファイルへ、またはファイルから N 項組をコピーするのに使われる。

22.3 すでにある N 項組ファイルのオープン

[Function] gsl_ntuple * gsl_ntuple_open (char * filename, void * ntuple_data, size_t size)

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。現在の N 項組の行 ntuple data へのポインタを指定しなければならない。このポインタはファイルへ、またはファイルから N 項組をコピーするのに使われる。

22.4 N 項組の書き込み

[Function] int gsl_ntuple_write (gsl_ntuple * ntuple)

大きさ ntuple->size の参照時点での N 項組 ntuple->ntuple_data を、そのファイルに書き込む。

[Function] int gsl_ntuple_bookdata (gsl_ntuple * ntuple)

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

[Function] int gsl_ntuple_read (gsl_ntuple * ntuple)

ntuple のファイルから現在の行を読み込み、ntuple->data にその値を保持する。

22.6 N 項組ファイルのクローズ

[Function] int gsl_ntuple_close (gsl_ntuple * ntuple)

ntuple のファイルをクローズし、保持していたメモリを解放する。

22.7 N 項組からのヒストグラムの生成

N 項組から関数 gsl_ntuple_project を使っていくつかの方法でヒストグラムを作ることができる。事象を選択する関数と、スカラー値を計算する関数の二つを利用者が定義する必要がある。二つの関数は第一引数で N 項組の行を、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、N 項組の行のうちのどれをヒストグラムにするかを選択する関数として、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、N 項組の各行に対して、それが対象としているヒストグラムに含まれるなら非零の値を返すように定義する。

値を計算する関数は、選択関数で得られる N 項組の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。

[Function] int gsl_ntuple_project (gsl_histogram * h, gsl_ntuple * ntuple, gsl_ntuple value fn * value_func, gsl_ntuple select fn * select_func)

関数 value_func および select_func を使って N 項組 ntuple でヒストグラム h を更新する。関数 select_func が非零を返す N 項組の行に対して、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 が零を返す行は無視される。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。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 項ファイル 'test.dat' に出力される。

```
#include <gsl/gsl_ntuple.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

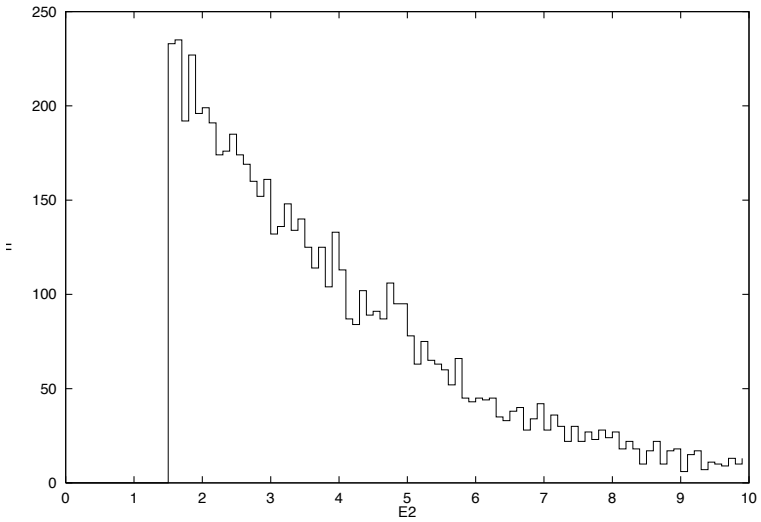
```
struct data {
    double x;
    double y;
    double z;
};

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);

    return 0;
}
```

次のプログラムはファイル 'test.dat' の n 組データを解析する。解析は、各事象の平方値 $E^2 = x^2 + y^2 + z^2$ を計算し、下限値 1.5 以上の値を持つ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、選択された事象のプロットを示す。下限値以上のみをプロットしている。



22.9 参考文献

cern 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(オンラインで入手できる)。

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を行うルーチンについて説明する。これには従来の意味のモンテカルロ法に加え、vegas と miser の重要度サンプリング (importance sampling) と階層サンプリング (stratified sampling) を行う適応的手法が含まれている。各アルゴリズムは、以下で表される多次元有限積分の値を近似的に求める。

$$I = \int_{x_i}^{x_u} dx \int_{y_i}^{y_u} dy \ldots f(x, y, \ldots)$$

超立体 ((x_l , x_u), (y_l , y_u), ...) 領域内で、関数が有限回数だけ計算される。ライブラリ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、これは厳密な誤差限界（上限）を示すものではなく、単なる目安として用いるべきである。乱数的サンプリングでは超立体内のすべての点を網羅し関数の特徴を反映できるわけではなく、誤差は過小評価されるためである。

関数は三つのヘッダファイル 'gsl_monte_plain.h'、'gsl_monte_miser.h' および 'gsl_monte_vegas.h' 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全て、同じ形式で呼ぶことができるようになっている。制御変数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。

各積分関数には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、関数呼び出しの回数を利用者が指定することで決められる。もし特定の精度を要求したい場合は、積分関数を数回呼んでそれぞれの結果を平均することを、要求精度に達するまで行えばよい。

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り、したがって境界上で特異点があっても、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。

[Data Type] gsl_monte_function

この型は、パラメータを内部に保持しモンテカルロ積分を行う一般的な関数の型である。

`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。

$$f(x, y) = ax^2 + bxy + cy^2$$

ここで a = 3、b = 2、c = 1 とする。以下のコードで定義する関数 `gsl_monte_function F` を積分関数に渡す。

```
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 素朴なモンテカルロ積分

素朴なモンテカルロ積分アルゴリズムでは、積分範囲内でランダムに点を発生し、その点で関数を評価し誤差を見積もる。このアルゴリズムを使って、ランダムに分布する N 個の点 x_i から計算される積分 E(f; N) は、V は積分範囲の体積とするとき、以下のように表される。

$$E(f;N) = V(f) = \frac{V}{N}\sum_i^N f(x_i)$$

この推定積分値の誤差 σ(E; N) は平均値の推定分散から計算される。

$$\sigma^2(E;N) = V(f) = \frac{V}{N}\sum_i^N (f(x_i) - < f >)^2$$

var(f) を積分範囲内ので関数値の真の分散とすると、N が大きくなるにしたがってこの分散の値は小さくなり、var(f)/N に近づいていく。推定誤差は σ(f)/√N に従って小さくなる。誤差は 1/√N にしたがって小さくなることを考えると、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。

この節で説明する関数はヘッダファイル ‘gsl_monte_plain.h’ で宣言されている。

[Function] gsl_monte_plain_state * gsl_monte_plain_alloc (size_t dim)

モンテカルロ積分のための作業領域を次元数を dim として確保し、初期化する。

[Function] int gsl_monte_plain_init (gsl_monte_plain state* s)

あらかじめ確保されている作業領域を初期化する。使い終わった作業領域を別の積分に使い回したいときに使うことができる。

[Function] 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)

素朴なモンテカルロ積分アルゴリズムを使って、大きさ dim の配列 x_lと x_uで指定される上下

限で定義される dim 次元の超立体内を範囲として積分値を計算する。積分値は乱数発生器 r で生成する点を使った calls 回の関数呼び出しで計算される。あらかじめ作業領域 s を確保して指定する必要がある。推定積分値は result に、推定絶対誤差は abserr に入れられる。

[Function] void gsl_monte_plain_free (gsl_monte_plain state * s)

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。

23.3 MISER

プレス Press とファラール Farrar による miser アルゴリズムは再帰的層化抽出法を使った方法である。これは、積分範囲内でもっとも分散の大きな領域に集中して点を取ることで、積分誤差を抑えようとする方法である。

層化抽出法では、二つの離れた領域 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 $E(f) = (E_a(f) + E_b(f))/2$ の分散を以下の式で計算することから始める。

$$\mathrm{Var}(f) = \frac{\sigma_a^2(f)}{4N_a} + \frac{\sigma_b^2(f)}{4N_b}$$

この分散は点を以下のように分布させ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すると、誤差を最小化できる。

miser 法は、一つの座標軸に沿って積分領域を二分することを 1 ステップとして計算を進めていく。全てのあり得る d 通りの二分の仕方を試し、二つの領域の分散が最小になる分け方を選ぶ。各領域内での分散は、その時点でのステップで有効なすべてのサンプル点の一部から推定する。もっともよい分割による二つの各領域について、以上の方法を再帰的に繰り返す。残りのサンプル点は N_a と N_b の式に基づいて各領域に割り当てていく。再帰は利用者が指定したの深さまで続けられ、そこでは各領域内が素朴なモンテカルロ積分アルゴリズムと同じになる。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。

[Function] gsl_monte_miser_state * gsl_monte_miser_alloc (size_t dim)

次元数 dim で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。

[Function] int gsl_monte_miser_init (gsl_monte_miser state* s)

すでに確保されている積分のための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。

[Function] 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)

関数 f の、上限と下限がそれぞれ大きさ dim の配列 xl と xu で指定される dim 次元の超立体内での積分値を miser モンテカルロ法で計算する。関数呼び出しは 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。

[Function] void gsl_monte_miser_free (gsl_monte_miser state * s)

作業領域 s で確保されているメモリを解放する。

miser 法には調整用のパラメータがいくつか用意されている。 `gsl_monte_miser_state` 構造体を使って以下の変数を操作できる。

[Variable] double estimate_frac

再帰の各ステップでの分散を推定するために使われる、現時点までの関数呼び出しの回数の割合を指定する。規定値は 0.1。

[Variable] size_t min_calls

分散を推定するために必要とする関数呼び出しの回数の最小の割合を指定する。分散の推定のための `estimate_frac` を使った関数呼び出しの回数が `min_calls` 未満のときは、代わりに `min_calls` が使われる。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。`min_calls` の規定値は 16 * dim である。

[Variable] size_t min_calls_per_bisection

二分探索を進めるための関数呼び出しの最小回数を指定する。再帰処理による呼び出しが `min_calls_per_bisection` より少ない場合は、その時点での探索領域に対して素朴なモンテカルロ積分アルゴリズムを適用し、その再帰はそこで終了する。このパラメータの規定値は 32 * `min_calls` である。

[Variable] double alpha

二つに分割された各探索領域での推定分散から、点を割り当てるときにどのように一つの分散値を推定するかを指定する。分割された各領域でのサンプル点は分散を最小にするように明示的に選ばれるため、再帰的なサンプル点決定により分散の傾きは 1/N よりもよくなる。miser 法ではこれを取り入れるため、全体の分散が係数 α で調整できるようにしてある。

$$Var(f) = \frac{\sigma_a}{N_a^\alpha} + \frac{\sigma_b}{N_b^\alpha}$$

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、このライブラリでもそれを既定値として採用している。

[Variable] double dither

二分探索での領域分割が非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。既定値は零であり、領域分割に乱数は使われない。現実的には dither の値は 0.1 程度がよいと考えられる。

23.4 VEGAS

ルパージュ Lepage の vegas 法はサンプリングの重み付けが基本であり、関数 |f| で記述される確率分布にしたがって点を選んでいく。そのため、積分値にもっとも寄与の大きな領域に点が集中して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が関数 g で記述される確率分布にしたがったサンプリング点で行われるとすると、積分の推定値 Eg (f ; N) は以下のようになる。

$$E_g(f; N) = E(f/g; N)$$

このとき、分散は以下のようになる。

$$Var_g(f;N) = Var(f/g;N)$$

確率分布を g = |f| / I(|f|) となるようにとると、分散 V_g(f; N) は零になり、推定積分値と真の積分値の誤差も零になる。現実的には確率分布関数 g を厳密にそうなるようにはとれないが、重み付きサンプリングは、理想とする分布関数を効率よく近似することを目指して行われる。

vegas 法では厳密な分布関数を、関数 f のヒストグラムを作りながら積分する領域全体に渡って経路をいくつも作っていくことで近似する。一つのヒストグラムから、サンプリングのための分布関数を定義し、次の経路を作る。これを繰り返すことで厳密な分布関数に漸近していく。ヒストグラムの階級数が K^d のような爆発的に増加してしまうのを避けるため、確率分布関数は $g(x_1, x_2, \ldots) = g_1(x_1)g_2(x_2)\ldots$のように分割して定義される。これにより階級数を K×d に抑えることができる。これにより関数のピークが被積分関数の射影から座標軸上に来ることになる。vegas 法の効率はこの仮定がどの程度正しいかに依存する。もっともよいのは被積分関数のピークが局在化するときである。被積分関数が複数の関数の積の形でよく近似できる場合は、vegas 法を使うことで効率を上げることができる。

vegas 法には他にもいくつか工夫点があり、層化抽出法と重み付き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」に分けられ、各直方体では同じ数の点がサンプリングされる（最終的には 2 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 では重み付きサンプリングから分散低減法に切り替わる。

[Function] gsl_monte_vegas_state * gsl_monte_vegas_alloc (size_t dim)

次元数 dim で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。この作業領域を使って積分の進行を管理する。

[Function] int gsl_monte_vegas_init (gsl_monte_vegas state* s)

すでに確保されている積分のための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。

[Function] 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)

関数 f の、上限と下限がそれぞれ大きさ dim の配列 xl と xu で指定される dim 次元の超立体内での積分値を vegas モンテカルロ法で計算する。関数呼び出しは 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サンプルについての重み付き平均から計算される。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 になる。

[Function] void gsl_monte_vegas_free (gsl_monte_vegas state * s)

作業領域 s で確保されているメモリを解放する。

vegas 法では、以下に述べる iterations パラメータにしたがって、積分値の計算を複数回、内部でそれぞれ独立して行い、その重み付き平均を返す。被積分関数のランダム・サンプリングは、関数値が定数値となるような領域などで、誤差が零になることがあるが、この場合には重み付き平均の計算が破綻するため、そういった場合は別に扱う。vegas の元の FORTRAN による実装では、誤差が零の場合にそれを非常に小さな値（1e-30 など）で置き換える。GSL での実装ではこれと異なり、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値に対する重みの平均を使うか、または無視するかを決める。

現在の推定値の誤差が零で、重み付き平均値の誤差が有限の値の場合
: 現在の推定値に対する重みは、それまでの推定値に対する重みの平均とする。

現在の推定値の誤差が有限値で、重み付き平均値の誤差が零の場合
: それまでの推定値は破棄され重み付き平均の値は現在の推定値から計算を始める。

現在の推定値の誤差が零で、重み付き平均値の誤差も零の場合
: 推定値は算術平均として計算され、誤差は計算されない。

vegas 法では `gsl_monte_vegas_state` 構造体の要素を使って詳細な調整ができる。

[Variable] double result

[Variable] double sigma

積分計算の最後の繰り返し計算での結果 result とその誤差 sigma。

[Variable] double chisq

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、繰り返し計算の各回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、積分値の信頼性を上げるために繰り返し計算を続ける必要がある。

[Variable] double alpha

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。零にすると階級は固定される。

[Variable] size_t iterations

ルーチンが 1 回呼ばれたときに内部で行われる繰り返し計算の回数。既定値は 5 回。

[Variable] int stage

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、領域を等分割し重み付き平均を零から始める設定として stage = 0 とする。stage = 1 として vegas を呼び出すと、重み付き平均は零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多数の点を使う計算でも最適化された領域分割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、関数呼び出しの回数に応じた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。

[Variable] int mode

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重み付きサンプリング、層化抽出法、関数値そのもののいずれかに設定する。次元数が小さいときは vegas では厳密な層化抽出法を使う（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）。

[Variable] int verbose

[Variable] FILE * ostream

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。verbose の規定値は -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。

$$I = \int_{-\pi}^{+\pi} \frac{dk_x}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_y}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_z}{2\pi} \frac{1}{1-\cos(k_x)\cos(k_y)\cos(k_z)}$$

この積分の解析的な値は I = Γ(1/4)/(4π) = 1.393203929685676859... 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, π, π)、(π, 0, π)、(π, π, 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))
  解析的に解くとは Gamma(1/4)^4/(4 pi^3) にある。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、以下の範囲でのみ積分を行う。
  (0,0,0) -> (pi,pi,pi) and multiply by 8 */

double exact = 1.393203929685676859184246260325 5;

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);

    return 0;
}
```

被積分関数の呼び出し回数を 500,000 回としたとき、素朴なモンテカルロ積分では相対誤差は 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 法では誤差を約半分にすることができ、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 回の被積分関数呼び出しによる積分を行う。これに続いて 100,000 回の被積分関数呼び出し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（被積分関数の呼び出し回数が同程度）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。

- W.H. Press, G.R. Farrar, Recursive Stratified Sampling for Multidimensional Monte Carlo Integration, Computers in Physics, v4 (1990), pp190–195.

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, 192–203, (1978)
- G.P. Lepage, VEGAS: An Adaptive Multi-dimensional Integration Program, Cornell preprint CLNS 80-447, March 1980

第 24 章 シミュレーティド・アニーリング

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、ヤコビアン行列を必要とするニュートン法のような方法は使えないが、確率的探索法（発見的探索法）が利用できることがある。こういった方法は特に、巡回セールスマン問題のような組み合わせ最適化によく用いられる。

ここでの目標は、実数関数であるエネルギー関数（またはコスト関数）を探索空間中で最小にする点を探すことである。シミュレーティド・アニーリングは、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、乱数で決めた次の探索点を採用するかどうかはボルツマン分布が与える確率で決める。

この章で述べる関数は、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（関数値）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従い決定される。

$$p = e^{-(E_{i+1}-E_i)/(kT)}$$

また $E_{i+1} \le E_i$ であれば p = 1 である。

つまり次の探索点でのエネルギーが小さくなる場合にのみ、そこに探索を進めるということである。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ことはあり、その場合その可能性は温度 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 $E_{i+1} - E_i$ に反比例するようにする。

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従って、ゆっくりと下げていく。ここで μ_T は 1 よりもわずかに大きな値とする。

エネルギーが高い点に進む確率は小さいが、こ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

[Function] 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)

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。探索空間は複数の関数 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。焼き鈍し（アニーリング）が成功したときには、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。print position が null の場合はなにも出力されない。

```
number_of_iterations temperature x x-(*x0_p) Ef(x)
```

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。

[Data Type] gsl_siman_Efunc_t

点 xp でのエネルギー関数値を返す関数。

```
double (*gsl_siman_Efunc_t) (void *xp)
```

[Data Type] gsl_siman_step_t

乱数発生器 r を使って乱数探索で探索点を更新する関数。更新量は step_size 以下になる。

```
void (*gsl_siman_step_t) (const gsl_rng *r, void *xp,
      double step_size)
```

[Data Type] gsl_siman_metric_t

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)
```

[Data Type] gsl_siman_print_t

探索点 xp を表示する関数。

```
void (*gsl_siman_print_t) (void *xp)
```

[Data Type] gsl_siman_copy_t

探索点 source を dest にコピーする関数。

```
void (*gsl_siman_copy_t) (void *source, void *dest)
```

[Data Type] gsl_siman_copy_construct_t

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)
```

[Data Type] gsl_siman_destroy_t

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)
```

[Data Type] gsl_siman_params_t

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。この構造体は探索を制御するための全ての情報、つまりエネルギー関数、探索点を更新する関数、探索開始点その他を保持する。

`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

このライブラリが用意するシミュレーティド・アニーリング・パッケージは、C 言語で書かれていながらさまざまな振る舞いをすることを目指してしているため、未だ完成度はあまり高くないが、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。

24.4 簡単な例

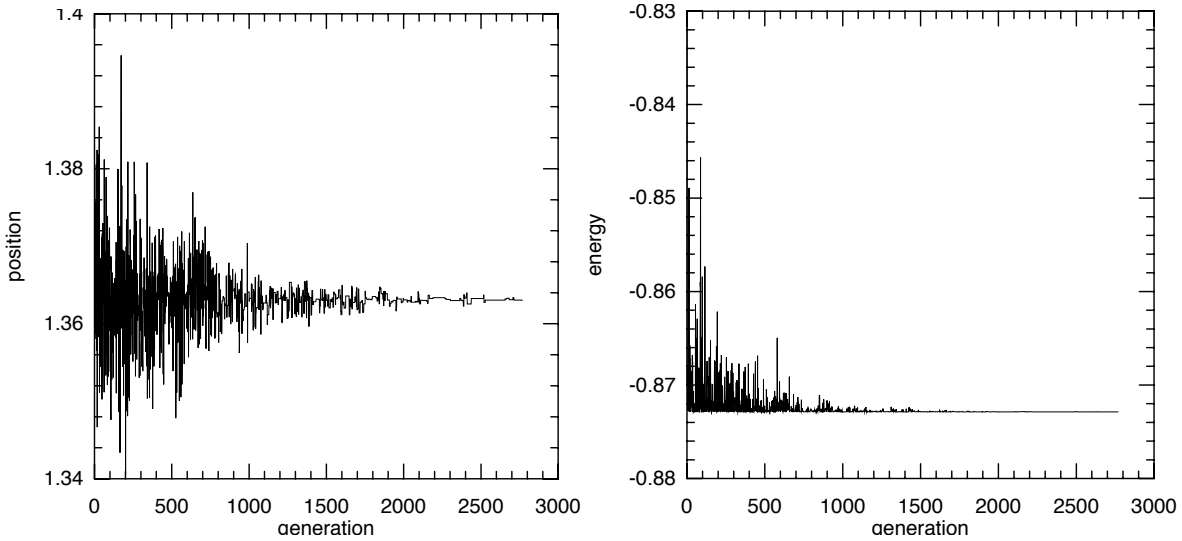
最初のサンプルは、一次元デカルト座標系の空間内の減衰 sin 関数をエネルギー関数とする例である。これには多数の局所解があるが最適解は 1.0 と 1.5 の間にある一つだけである。探索開始点は 15.5 とした。最適解から離れていて、間にいくつかの局所解がある。

```
#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 */
/* 次の探索点を決めるための周辺探索の回数 */
#define N_TRIES 200

/* 各温度で何回まで繰り返すか */
#define ITERS_FIXED_T 10

/* 乱数探索の最大探索距離 */
#define STEP_SIZE 10

/* ボルツマン定数 */
#define K 1.0

/* 温度の初期値 */
#define T_INITIAL 0.002

/* 温度低下係数 */
#define MU_T 1.005
#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);
    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 | grep -v "^#"
    | xyplot -xyil -y -0.88 -0.83 -d "x...y"
    | xyps -d > siman-test.eps
$ ./siman_test | grep -v "^#"
    | xyplot -xyil -xl "generation" -yl "energy" -d "x..y"
    | xyps -d > siman-energy.eps
```



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。

24.5 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。この例では、都市間の距離が自動車で行ける距離ではないような気もするので、空飛ぶセールスマン問題 Flying Salesman Problem とでも呼ぶべきかもしれない。また地球の表面は球面であると仮定し、ジオイドでの距離は使わない。

`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。

完全なソースコードは ‘`siman/siman_tsp.c`’ であるが、以下のようにして得られるプロットをここに示す。

```
./siman_tsp > tsp.output
grep -v "^#" tsp.output
    | xyplot -xyil -d "x..............y"
        -lx "generation" -ly "distance"
        -lt "TSP -- 12 southwest cities"
    | xyps -d > 12-cities.eps
grep initial_city_coord tsp.output
    | awk '{print $2, $3, $4, $5}'
    | xyplot -xyil -lb0 -cs 0.8
        -lx "longitude (- means west)"
        -ly "latitude"
        -lt "TSP -- initial-order"
    | xyps -d > initial-route.eps
grep final_city_coord tsp.output
    | awk '{print $2, $3, $4, $5}'
    | xyplot -xyil -lb0 -cs 0.8
        -lx "longitude (- means west)"
        -ly "latitude"
        -lt "TSP -- final-order"
    | xyps -d > final-route.eps
```

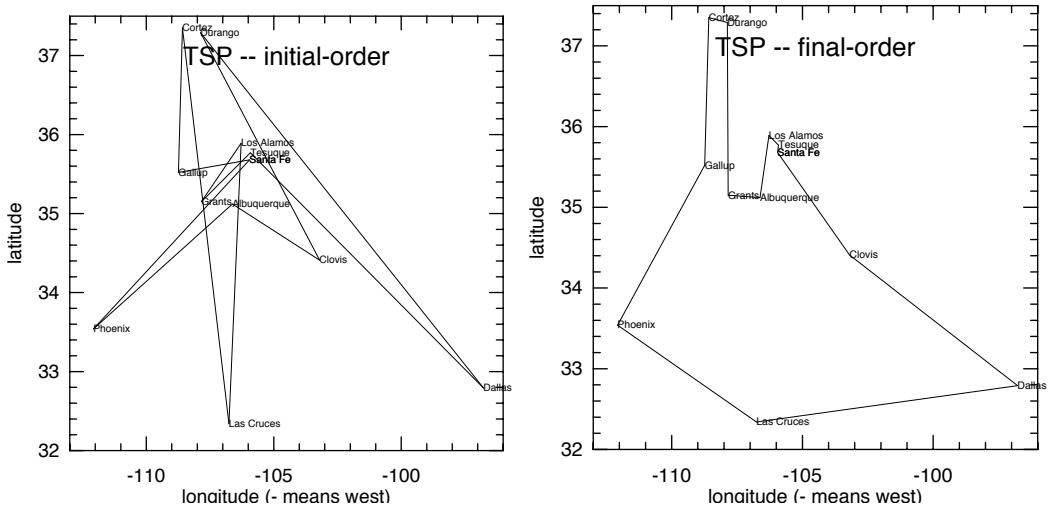
以下に示す出力は、探索初期の段階で得られる経路である。経度が負の値になっているのは、今回のような、都市が西半球にある場合に図のようなプロットを得るためである。

```
# 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。

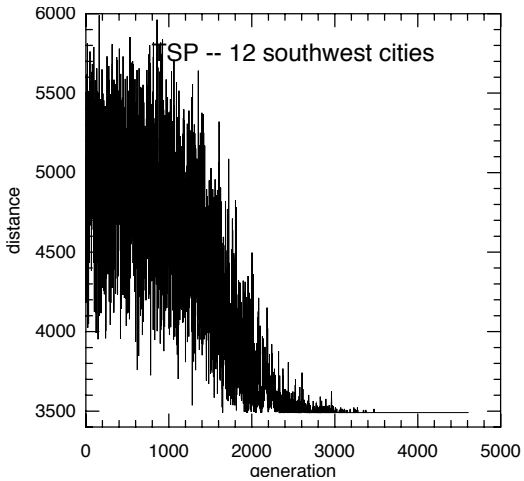
```
# final coordinates of cities (longitude and latitude)
###final_city_coord: -105.95 35.68 Santa Fe
###final_city_coord: -106.28 35.89 Los Alamos
```

```
###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
```



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)。

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。



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例。

24.6 参考文献

より詳しくは、以下の文献が参考になる。

・ Modern Heuristic Techniques for Combinatorial Problems, Colin R. Reeves (ed.), McGraw-Hill, 1995 (ISBN 0-07-709239-2).

第 25 章 常微分方程式

この章では常微分方程式（ODE）の初期値問題の解法について説明する。このライブラリには、ルンゲ・クッタ法やブリルシュ・ストア法などの様々な低レベルルーチンと、ステップ幅を計算、調整する高レベルルーチンをを用意している。利用者は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必要がある。

ここで触れる関数は ‘gsl_odeiv.h’ で宣言されている。

25.1 解こうとする微分方程式の定義

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

$$\frac{dy_i(t)}{dt} = f_i(t, y_1(t), \ldots, y_n(t))$$

を解く（ただし i = 1, . . . , n）ためのものである。ステップを進める関数は導関数 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。微分方程式は型 gsl_odeiv_system で定義する。

[Data Type] gsl_odeiv_system

この型は汎用のパラメータで定義される常微分方程式（ODE）を定義する。

```
int (* function) (double t, const double y[], double dydt[], void *
params)
```

引数が t, y でパラメータが params のとき、ベクトル要素 f_i(t, y, params) を配列 dydt に入れる。

```
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、ベクトル要素 ∂f_i(t, y, params)/∂ 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] として行ごとに並べて dfdy に入れる。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 jacobian 要素の代わりに null ポインタを指定する）。しかしもっとも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、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
```

微分方程式の次数。

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。

[Function] gsl_odeiv_step * gsl_odeiv_step_alloc (const gsl_odeiv_step_type * T, size_t dim)

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。

[Function] int gsl_odeiv_step_reset (gsl_odeiv_step * s)

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。

[Function] void gsl_odeiv_step_free (gsl_odeiv_step * s)

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。

[Function] const char * gsl_odeiv_step_name (const gsl_odeiv_step * s)

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 'rk4'`のように出力する。

[Function] unsigned int gsl_odeiv_step_order (const gsl_odeiv_step * s)

直前のステップでのステップ関数の精度（オーダー）を返す。ステップが可変の場合には精度が変化する。

[Function] 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)

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。新しい解が y に、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。

このライブラリでは、以下のアルゴリズムが利用できる。

[Step Type] gsl_odeiv_step_rk2

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）。

[Step Type] gsl_odeiv_step_rk4

普通の四次のルンゲ・クッタ法。

[Step Type] gsl_odeiv_step_rkf45

埋め込み型 RKF45 公式（ルンゲ・クッタ・フェルバーク法 Embedded Runge-Kutta-Fehlberg(4, 5)）。幅広い目的に使える手法である。

[Step Type] gsl_odeiv_step_rkck

埋め込み型 RKCK45 公式（ルンゲ・クッタ・キャシュ・カープ法 Embedded Runge-Kutta Cash-Karp (4, 5)）。

[Step Type] gsl_odeiv_step_rk8pd

埋め込み型 RKPD89 公式（ルンゲ・クッタ・プリンス・ドルマンド法 Embedded Runge-Kutta Prince-Dormand (8, 9)）。

[Step Type] gsl_odeiv_step_rk2imp

Gaussian point での二次の陰的ルンゲ・クッタ法。

[Step Type] gsl_odeiv_step_rk4imp

Gaussian point での四次の陰的ルンゲ・クッタ法。

[Step Type] gsl_odeiv_step_bsimp

バダーとドイフルハルトの陰的ブリルシュ・ストア法。この方法はヤコビアン行列を使う。

[Step Type] gsl_odeiv_step_gear1

M=1 の陰的ギア法。

[Step Type] gsl_odeiv_step_gear2

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の解の値の変化と推定誤差を判定し、利用者が指定する誤差範囲に収まるような最適なステップ幅を決定する。

[Function] gsl_odeiv_control * gsl_odeiv_control_standard_new (double eps_abs, double eps_rel, double a_y, double a_dydt)

標準の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) に対するスケーリング係数 a_y と a_dydt の 4 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、各要素について要求される精度（誤差レベル）D を計算する。

$$D_i = \mathrm{abs} + \mathrm{rel} * (a_y|y_i| + a_{dydt}\, h|y_i'|)$$

そしてこれを実際の誤差 $E_i = |\mathrm{yerr}_i|$ と比較する。実際の誤差 E が要求精度 D の 1.1 倍を上回る場合は、以下で与えられる係数を乗じることでステップ幅を縮小する。

$$h_{new} = h_{old} * S * (E/D)^{-1/q}$$

ここで q は解法の次数（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。

$$h_{new} = h_{old} * S * (E/D)^{-1+q}$$

誤差の縮小は全てこの方法で行われる。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、極端な値にならないようになっている。

[Function] gsl_odeiv_control * gsl_odeiv_control_y_new (double eps_abs, double eps_rel)

各ステップにおける誤差を絶対誤差 eps_abs およびその時点での解の値 $y_i(t)$ に対する相対誤差 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。

[Function] gsl_odeiv_control * gsl_odeiv_control_yp_new (double eps_abs, double eps_rel)

各ステップにおける誤差を絶対誤差 eps abs およびその時点での解の微分値 $y_i'(t)$ に対する相対誤差 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。

[Function] 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)

ステップ調整法のインスタンスを生成する。gsl_odeiv_control_standard_new と同じ計算法を使うが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。

$$D_i = \varepsilon_{abs} s_i + \varepsilon_{rel} * (a_y |y_i| + a_{dydt} h |y_i|)$$

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。

[Function] gsl_odeiv_control * gsl_odeiv_control_alloc (const gsl_odeiv_control_type * T)

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述の標準的な方法で十分である。

[Function] int gsl_odeiv_control_init (gsl odeiv control * c, double eps_abs, double eps_rel, double a_y, double a_dydt)

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。

[Function] void gsl_odeiv_control_free (gsl_odeiv_control * c)

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。

[Function] int gsl_odeiv_control_hadjust (gsl_odeiv_control * c, gsl_odeiv_step * s, const double y0 [], const double yerr [], const double dydt [], double * h)

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。ステップを進めるステップ関数 step でも、この手法の次数を決めなければならない。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。

[Function] const char * gsl_odeiv_control_name (const gsl_odeiv_control * c)

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 時間発展

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、時間発展を行う関数である。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、発展関数は現在のステップから戻って、計算され縮小されたステップを使う。これが適切なステップ幅が見つかるまで繰り返される。

[Function] gsl_odeiv_evolve * gsl_odeiv_evolve_alloc (size_t dim)

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。

[Function] 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 [])

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。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。

[Function] int gsl_odeiv_evolve_reset (gsl_odeiv_evolve * e)

発展関数のインスタンス e を再設定する。それまでのステップでは求解が進まないような場合に使われる。

[Function] void gsl_odeiv_evolve_free (gsl_odeiv_evolve * e)

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。

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。

$$x' = y$$

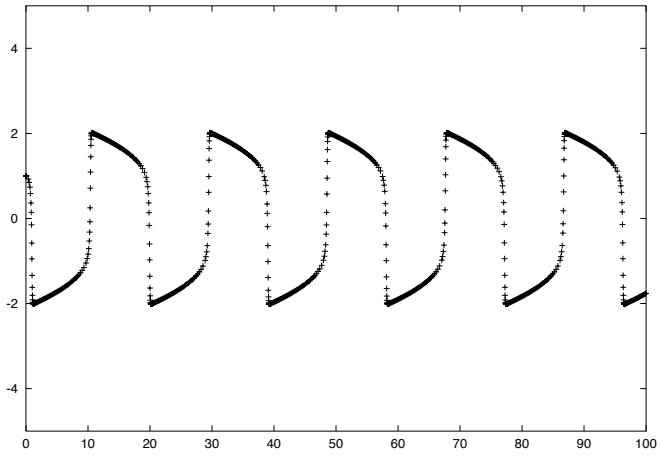
$$y' = -x + \mu y(1 - x^2)$$

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) {
          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;
}
```

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。



プリンス・ドルマンドの 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。

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, . . . , 100 で解の値を得るには、プログラムを以下のように変更する。

```
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]);
}
```

特に工夫しない、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 = 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。

- Abramowitz & Stegun (eds.), Handbook of Mathematical Functions, Section 25.5.

陰的ブリルシュ・ストア法 bsimp は以下の論文にある。

- G. Bader and P. Deuflhard, “A Semi-Implicit Mid-Point Rule for Stiff Systems of Ordinary Differential Equations.”, Numer. Math. 41, 373–398, 1983.

第 26 章 補間

この章では、補間を行う関数について説明する。このライブラリには三次スプラインや秋間スプライン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中に切り替えることができる。境界条件は、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。

この章で説明する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、このライブラリ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。補間関数は区間内では連続だが、境界上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

補間を行う関数は gsl_interp インスタンスに格納されたデータを用いる。このインスタンスは以下のような関数で生成される。

[Function] gsl_interp * gsl_interp_alloc (const gsl_interp_type * T, size_t size)

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。

[Function] int gsl_interp_init (gsl_interp * interp, const double xa [], const double ya [], size_t size)

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）はデータ配列 xa および ya を別のどこかに保存したりはせず、インスタンスの内部で静的 static に保持しておくだけである。データ配列 xa は整列しておく必要がある。他の引数にはそういった条件はない。

[Function] void gsl_interp_free (gsl_interp * interp)

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五種類の補間法を用意している。

[Interpolation Type] gsl_interp_linear

線形補間。この方法は使用メモリが最も少なくて済む。

[Interpolation Type] gsl_interp_polynomial

多項式補間。この方法は一般に振動しやすく、データの性質があまり悪くなくても振動することがあるため、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。

[Interpolation Type] gsl_interp_cspline

自然な境界の三次スプライン（自然スプライン）。

[Interpolation Type] gsl_interp_cspline_periodic

周期的境界の三次スプライン（周期的スプライン）。

[Interpolation Type] gsl_interp_akima

自然な境界条件での non-rounded な秋間スプライン。ウォディカ Wodicka による non-rounded corner 法を使う。

[Interpolation Type] gsl_interp_akima_periodic

周期的境界条件での non-rounded な秋間スプライン。ウォディカによる non-rounded corner 法を使う。

以下のような関数も利用できる。

[Function] const char * gsl_interp_name (const gsl_interp * interp)

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.' と出力する。

```
printf("interp uses '%s' interpolation.\n",
       gsl_interp_name (interp));
```

[Function] unsigned int gsl_interp_min_size (const gsl_interp * interp)

インスタンス interp が使っている補間法で要求するデータ点数を返す。例えば秋間スプライン補間では、5 点以上の点を必要とする。

26.4 添え字検索と加速法

補間関数の探索状況は、補間区間の添え字検索の繰り返しの状況として gsl_interp_accel インスタンスに保存され、現在から見た添え字検索の直前の値が保持される。同じ補間区間幅を取って次の補間点で補間が失敗した場合は、その区間の添え字がその場で返される。

[Function] size_t gsl_interp_bsearch (const double x_array [], double x, size_t index_lo, size_t index_hi)

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。

[Function] gsl_interp_accel * gsl_interp_accel_alloc (void)

補完関数を探索する繰り返し計算（加速法とも言う）を行うインスタンスへのポインタを返す。このオブジェクトは検索状況をリアルタイムで保持し、これを利用して様々な加速法を適用できる。

[Function] size_t gsl_interp_accel_find (gsl_interp_accel * a, const double x_array [], size_t size, double x)

大きさ size のデータ配列 x_array から、加速法 a を使って添え字検索を行う。補間関数の評価における添え字検索は、この関数を使って行われる。x_array[i] ≦ x < x_array[i+1]の範囲内にある添え字 i を返す。

[Function] void gsl_interp_accel_free (gsl_interp_accel * acc)

補完関数を探索する繰り返し計算を行うインスタンス acc のメモリを解放する。

26.5 補間関数の関数値の計算

[Function] 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)

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、加速法 acc を使って計算し、返す。

[Function] 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)

与えられる点 x での補間関数の導関数値 d を、補間インスタンス interp、データ配列 xa、ya、加速法 acc を使って計算し、返す。

[Function] 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)

与えられる点 x での補間関数の二階導関数の値 d2 を、補間インスタンス interp、データ配列 xa、ya、加速法 acc を使って計算し、返す。

[Function] 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)

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、加速法 acc を使って計算し、返す。

26.6 高レベルの利用関数

ここまでに説明した関数では、呼び出しごとに利用者が配列 x や y へのポインタを指定する必要がある。以下に説明する関数はそれぞれ対応する `gsl_interp` の関数と同じであるが、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。

[Function] 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)

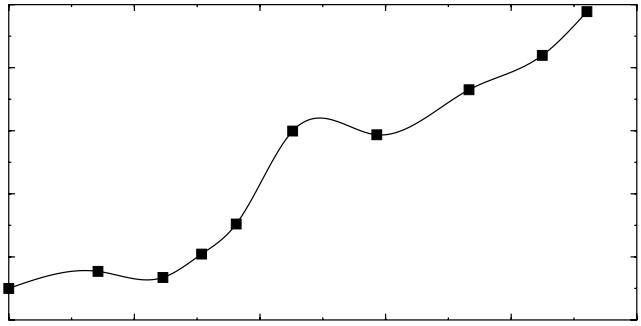
26.7 例

以下のプログラムは補間関数とスプラインを使った例である。x_i = i + sin(i)/2、y_i = i + cos(i2) で与えられる点を i = 0 … 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];

    printf ("#m=0,S=2\n");
    for (i = 0; i < 10; i++) {
        x[i] = i + 0.5 * sin(i);
        y[i] = i + cos(i * i);
        printf("%g %g\n", x[i], y[i]);
    }
    printf("#m=1,S=0\n");

    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    gsl_spline *spline = gsl_spline_alloc(gsl_interp_cspline,
                                          10);
    gsl_spline_init(spline, x, y, 10);
    for (xi = x[0]; xi < x[9]; xi += 0.01) {
        yi = gsl_spline_eval(spline, xi, acc);
        printf("%g %g\n", xi, yi);
    }
    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
```



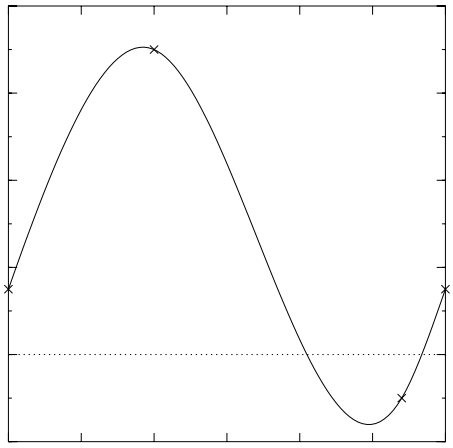
データ点を滑らかに補間する結果が得られている。補間法は gsl_spline_alloc の最初の引数を変えることで容易に切り替えることができる。

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;
    double x[4] = {0.00, 0.10, 0.27, 0.30};

    /* 最初と最後の値は同じでなければならない */
    double y[4] = {0.15, 0.70, -0.10, 0.15};
    gsl_interp_accel *acc = gsl_interp_accel_alloc ();
    const gsl_interp_type *t = gsl_interp_cspline_periodic;
    gsl_spline *spline = gsl_spline_alloc (t, N);

    int i; double xi, yi;

    printf ("#m=0,S=5\n");

    for (i = 0; i < N; i++) printf ("%g %g\n", x[i], y[i]);

    printf ("#m=1,S=0\n"); gsl_spline_init (spline, x, y, N);
    for (i = 0; i <= 100; i++) {
        xi = (1 - i / 100.0) * x[0] + (i / 100.0) * x[N-1];
        yi = gsl_spline_eval (spline, xi, acc);
        printf ("%g %g\n", xi, yi);
    }

    gsl_spline_free (spline);
    gsl_interp_accel_free (acc);

    return 0;
}
```

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
```



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。

- C.W. Ueberhuber, Numerical Computation (Volume 1), Chapter 9 “Interpolation”, Springer (1997), ISBN 3-540-62058-3.
- D.M. Young, R.T. Gregory A Survey of Numerical Mathematics (Volume 1), Chapter 6.8, Dover (1988), ISBN 0-486-65691-8.

第 27 章 数値微分

この章では有限差分により数値微分を行う関数について説明する。最良の有限差分を選び、微分係数との誤差見積もりを計算するためにもっともよい方法を用いている。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

[Function] int gsl_deriv_central (const gsl_function * f, double x, double h, double * result, double * abserr)

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、微分係数を計算する際の切り捨て及び丸め誤差のスケーリングによるステップ幅の最適化を行う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 から五点則を使って計算され、誤差見積もり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、実際には四点が使われるのみである。

[Function] int gsl_deriv_forward (const gsl_function * f, double x, double h, double * result, double * abserr)

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、微分係数を計算する際の切り捨て及び丸め誤差のスケーリングによるステップ幅の最適化を行う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」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誤差見積もりが計算される。

[Function] int gsl_deriv_backward (const gsl_function * f, double x, double h, double * result, double * abserr)

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 and Stegun, Handbook of Mathematical Functions, Section 25.3.4, and Table 25.5 (Coefficients for Differentiation).
- S.D. Conte and Carl de Boor, Elementary Numerical Analysis: An Algorithmic Ap- proach, McGraw-Hill, 1972.

第 28 章 チェビシェフ近似

この章では一変数関数のチェビシェフ近似を計算する関数について説明する。区間 [−1, 1] で重み関 数 $1/\sqrt{(1 - x^2)}$ を持つ直交基底多項式としてのチェビシェフ多項式を $T_n(x) = \cos(n \arccos(x))$ とするとき、級数 $f(x) = \Sigma c_n T_n(x)$ を有限項数で打ち切ったものがチェビシェフ近似である。最初の方のチェビシェフ多項式は $T_0(x) = 1$、$T_1(x) = x$、$T_2(x) = 2x^2 - 1$ である。詳しくは第 22 章の参考文献 Abramowitz & Stegun 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;   /* 区間の下限 */
    double b;   /* 区間の上限 */
    ...
} 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。級数は以下の保存則を使って計算される。

$$f(x) = \frac{c_0}{2} + \sum_{n=1} c_n T_n(x)$$

係数を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 チェビシェフ級数の生成と計算

[Function] gsl_cheb_series * gsl_cheb_alloc (const size_t n)

n 次のチェビシェフ級数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。

[Function] void gsl_cheb_free (gsl_cheb_series * cs)

チェビシェフ級数のインスタンス cs のメモリを解放する。

[Function] int gsl_cheb_init (gsl_cheb_series * cs, const gsl_function * f, const double a, const double b)

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 チェビシェフ近似の計算

[Function] double gsl_cheb_eval (const gsl_cheb_series * cs, double x)

与えられる点 x でのチェビシェフ級数 cs を計算する。

[Function] int gsl_cheb_eval_err (const gsl_cheb_series * cs, const double x, double * result, double * abserr)

与えられる点 x でのチェビシェフ級数 cs を計算し、級数の値を result に、絶対誤差見積もりを abserr に入れて返す。誤差見積もりは切り捨てられた項から計算する。

[Function] double gsl_cheb_eval_n (const gsl_cheb_series * cs, size_t order, double x)

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 以下で計算する。

[Function] int gsl_cheb_eval_n_err (const gsl_cheb_series * cs, const size_t order, const double x, double * result, double * abserr)

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 以下で計算し、級数の値を result に、絶対誤差見積もりを abserr に入れて返す。誤差見積もりは切り捨てられた項から計算する。

28.4 微分と積分

以下の関数でチェビシェフ数列の微分、積分を行って新しいチェビシェフ数列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。

[Function] int gsl_cheb_calc_deriv (gsl_cheb_series * deriv, const gsl_cheb_series * cs)

数列 cs の微分を計算し、微分係数をあらかじめメモリを確保されている deriv に入れる。cs と deriv の二つの数列は、同じサイズの配列でなければならない。

[Function] int gsl_cheb_calc_integ (gsl_cheb_series * integ, const gsl_cheb_series * cs)

数列 cs の積分を計算し、微分係数をあらかじめメモリを確保されている integ に入れる。cs と integ の二つの数列は、同じサイズの配列でなければならない。積分範囲の下限は、区間 a の左側の値を用いる。

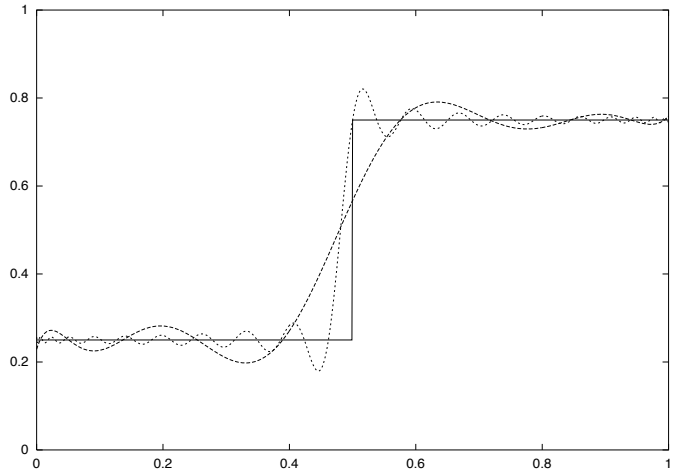
28.5 例

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

チェビシェフ近似の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), 254–256 (1973)

第 29 章 級数の収束の加速

この章では、級数の収束を加速するレヴィンの u 変換を行う関数について説明する。この方法は級数の最初のいくつかの項から体系的な近似を使って補外していき、誤差を見積もる。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 変換と誤差見積もりとともに計算する。誤差見積もりは、級数の最後の項まで各項の丸め誤差を伝達していくことで計算する。

GSL で用意している関数では、数列の各項が高精度で計算でき、計算を始めるときの丸め誤差が有限であると仮定できる場合に、級数の解析的な和に対する誤差が得られるように作られている。各項での誤差は GSL_DBL_EPSILON に対する相対誤差として得られる。。

補外した項での誤差見積もりの計算量は $O(N^2)$ 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いう速いが信頼性の低い方法もあり、それは次の節で説明する。ここで説明する方法は誤差見積もり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、保持する。

[Function] gsl_sum_levin_u_workspace * gsl_sum_levin_u_alloc (size_t n)

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。

[Function] int gsl_sum_levin_u_free (gsl_sum_levin_u_workspace * w)

作業領域 w に割り当てられているメモリを解放する。

[Function] int gsl_sum_levin_u_accel (const double * array, size_t array_size, gsl_sum_levin_u_workspace * w, double * sum_accel, double * abserr)

大きさ array_size の配列 array が持つ項での外挿の限界をレヴィンの u 変換を使って計算する。別途に確保した作業領域 w が必要になる。外挿による級数が sum_accel に、推定絶対誤差が abserr に入れられる。また各項ごとに加えていった和が w->sum_plain に入れられる。この方法では切り捨て誤差（二つの外挿した項の差）と丸め誤差（それぞれ独立した項に伝播していくもの）を計算し、それに基づいて外挿する項数を決定する。

29.2 誤差見積もり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、外挿による切り捨て誤差を最後の二つの近似項の差として誤差見積もりを行うものである。この方法では誤差見積もりは外挿値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い。そのためこの方法は計算量も使用メモリ量も O(N) のオーダーですむ。もし級数の収束が十分に速ければ、この方法を使うとよい。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に有用である。たとえば、似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積分を計算するときなどである。この方法の信頼性を検証するには、前節で説明した方法で誤差見積もりを最初に一度やっておくとよい。

[Function] gsl_sum_levin_utrunc_workspace * gsl_sum_levin_utrunc_alloc (size_t n)

項数 n で誤差見積もり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。

[Function] int gsl_sum_levin_utrunc_free (gsl_sum_levin_utrunc_workspace * w)

作業領域 w に割り当てられているメモリを解放する。

[Function] int gsl_sum_levin_utrunc_accel (const double * array, size_t array_size, gsl_sum_levin_utrunc_workspace * w, double * sum_accel, double * abserr_trunc)

大きさ array_size の配列 array が持つ項での外挿の限界をレヴィンの u 変換を使って計算する。別途に確保した作業領域 w が必要になる。外挿による級数が sum_accel に、各項ごとに加えていった和が w->sum_plain に入れられる。外挿した項の最後の二項の差が最小値を取るか、十分に小さくなったときに外挿を打ち切る。この差が誤差見積もりに使われ、abserr_trunc に入れられる。丸め誤差を計算するときに、外挿値の代わりに移動平均値を使って変化量を小さくすると、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 まで

級数の値が求められている。関数が返した誤差見積もりでも高精度で、厳密階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、1010 個の項が必要となる。

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, 9(3):346–354, 1983. and Algorithm 602 9(3):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, Intern. J. Computer Math. B3:371–388, 1973.

レヴィン変換の解説を web で見ることができる。

- Herbert H. H. Homeier, Scalar Levin-Type Sequence Transformations,
- 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（DWT）を行う関数について説明する。このライブラリで要している関数では、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’ に宣言されている。

30.1 DWT の定義

連続ウェーブレット変換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。

$$f(t) = \int_0^{\infty} ds \int_{-\infty}^{\infty} w(s,\tau) * \psi_{s,\tau}(t) dt$$

ここで基底関数 $\psi_{s,\tau}$は、マザー・ウェーブレット mother wavelet と呼ばれる単一の関数をスケーリング、変換することで得られる。

離散ウェーブレット変換は偶数個のデータと、固定値でのスケーリング及び変換ステップ (s, τ) により行われる。周波数軸と時間軸では、レベルパラメータ j を使って 2^jのスケールでディアディック dyadic（二つのベクトルを並べて書いたような）点で計算される。ウェーブレット変換 ψ はスケーリング関数 ϕ で表すと以下のようになる。

$$\psi(2^{j-1}, t) = \sum_{k=0}^{2^j-1} g_j(k) * \bar{\varphi}(2^j t - k)$$

また

$$\varphi(2^{j-1}, t) = \sum_{k=0}^{2^j-1} h_j(k) * \bar{\varphi}(2^j t - k)$$

関数 ψ と ϕ については、L を係数の個数とするとき、n = 0…L − 1 で係数 $g_n = (-1)^n h_{L-1-n}$ という関係がある。二つの係数集合 h_jと g_iでスケーリング関数とウェーブレットが定義される得られる関数群 $\{\psi_{j,n}\}$ は平方可積分 square-integrable な信号を表現する直行基底を構成する。

離散ウェーブレット変換の計算量は O(N) のオーダーであり、これは高速ウェーブレット変換とも呼ばれる。

30.2 DWT 関数の初期化

gsl_wavelet 構造体はウェーブレットとそのオフセット・パラメータ(センター・サポートのウェーブレット)を定義する係数を持つ。

[Function] gsl_wavelet * gsl_wavelet_alloc (const gsl_wavelet_type * T, size_t k)

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 でウェーブレット・ファミリーのメンバーを指定する。無効なメンバー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。

[Wavelet] gsl_wavelet_daubechies

[Wavelet] gsl_wavelet_daubechies_centered

消失モーメント k/2 の最大位相のドブシ・ウェーブレットである。ここで実装されているウェーブレットは偶数の k に対して k = 4, 6, . . . , 20 である。

[Wavelet] gsl_wavelet_haar

[Wavelet] gsl_wavelet_haar_centered

ハール・ウェーブレット。ここでは k = 2 でなければならない。

[Wavelet] gsl_wavelet_bspline

[Wavelet] gsl_wavelet_bspline_centered

(i, j) 次の双直行 B スプライン・ウェーブレット。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では副バンドの係数が縁に合わせられている。したがってウェーブレットの係数を位相空間で見ると理解しやすい。

[Function] const char * gsl_wavelet_name (const gsl_wavelet * w)

ウェーブレット w の名前文字列へのポインタを返す。

[Function] void gsl_wavelet_free (gsl_wavelet * w)

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。

[Function] gsl_wavelet_workspace * gsl_wavelet_workspace_alloc (size_t n)

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 nxn 行列の場合でも、行と列で独立に変換が行われるため、大きさ n の領域が確保されればよい。

[Function] void gsl_wavelet_workspace_free (gsl_wavelet_workspace * work)

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が条件であることに留意せねばならない。サンプル全体で信号が周期的でない場合は、変換の各段階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

[Function] 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)

配列 data の、長さ n で進み幅 stride の順および逆離散ウェーブレット変換を計算する。

変換長 n は 2 の累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 $f_i \rightarrow w_j$, k で、三角形式で圧縮されて置き換えられる。ここで j は j = 0 … 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 … 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。

$$(s_{-1,0}, d_{0,0}, d_{1,0}, d_{2,0}, \ldots, d_{j,k}, \ldots, d_{J-1,2^{j-1}-1})$$

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 の累乗じゃないとき、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。

30.4 二次元のウェーブレット変換

このライブラリでは、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。行列の次元は 2 の累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」と「非標準」の二通りがある。

標準変換では、まず全ての行に対して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、行列の行と列に対して各レベルの変換が飛び飛びに行われる。最初のレベルでの変換は、まず行について行われ、続いて一部が変換された値で列について変換される。データの各行と列について、次のレベルでの変換を繰り返し、全て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。

この節で説明する関数はヘッダファイル ‘gsl_wavelet2d.h’ で宣言されている。

[Function] 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)

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、標準および非標準形式の順および逆離散ウェーブレット変換を行う。次元は等しくなければならず(また正方行列でなければならない)、2 の累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 の累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。

[Function] 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)

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する。

[Function] 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)

非標準二次元ウェーブレット変換を計算する。

[Function] 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)

非標準二次元ウェーブレット変換を計算する。

30.5 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、それ以外は零とおいた近似を行う。

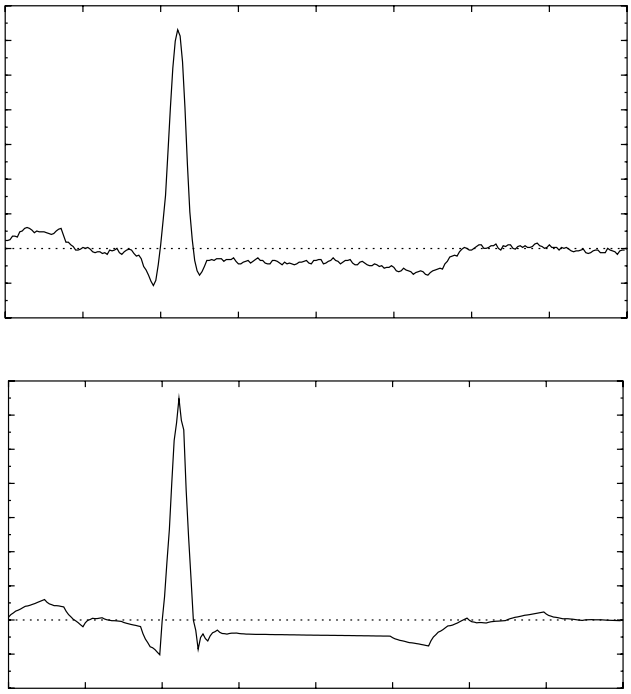
```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_wavelet.h>

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]);
}
```

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た場合の、原信号と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。



ECG 信号の原信号(上)と、ドベシ(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(下)。

30.6 参考文献

ウェーブレット変換の数学的な記述はドベシの以下の原著(講演録)にある。

- Ingrid Daubechies. Ten Lectures on Wavelets. CBMS-NSF Regional Conference Series in Applied Mathematics (1992), SIAM, ISBN 0898712742.

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 (2002), ISBN 0750306920.

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 (1999), ISBN 012466606X.

ウェーブレット解析の背景にある多重解像度解析の考え方は、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), 1989, 69–87.
・ S. G. Mallat. A Theory for Multiresolution Signal Decomposition—The Wavelet Representation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11, 1989, 674–693.

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 (1988) 909–996.
・ A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal Bases of Compactly Supported Wavelets. Communications on Pure and Applied Mathematics, 45 (1992) 485–560.

生理学データセットのデータベース PhysioNet http://www.physionet.org/ にはオンラインでアクセスできる。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):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

離散ハンケル変換は、次数が固定のベッセル関数の値が零になる点に対応する点での値を持つベクトルに対する演算である。これに対してフーリエ変換は、三角関数（正弦または余弦関数）の零点に対応する点での演算である。

特に f(t) が単位長を間隔とする点上の関数の場合、f(t) の有限 ν ハンケル変換は、数値の集合 g_mとして以下のように定義される。

$$g_m = \int_0^1 t dt J_\nu(j_{\nu,m}, t) f(t)$$

したがって

$$f(t) = \sum_{m=1}^{M-1} \frac{2 J_\nu(j_{\nu,m} x)}{J_{\nu+1}(j_{\nu,m})^2} g_m$$

f の帯域が m > M に対して g_m = 0 という制限を考えると、以下の基本サンプリング定理を得る。

$$g_m = \frac{2}{j_{\nu,M}^2} \sum_{k=1}^{M-1} f\left(\frac{j_{\nu,k}}{j_{\nu,M}}\right) \frac{J_\nu(j_{\nu,m} j_{\nu,k} / j_{\nu,M})}{J_{\nu+1}(j_{\nu,k})^2}$$

これが離散の場合に拡張した、すなわち離散ハンケル変換である。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。行列中の係数の値は ν と M に依存し、あらかじめ計算して保持しておかねばならない。これは gsl_dht のインスタンスを通じて行う。メモリ確保関数 gsl_dht_alloc で生成した gsl_dht のインスタンスは初期化関数 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、固定値の ν と M 関してデータベクトルに対して演算を行うことができる。このライブラリでは、基本間隔をスケーリングすることができるため、変換対象のデータのサンプリング間隔が単位長ではない場合、つまり[0, X] という間隔で定義されているときも変換を行うことができる。

上述のサンプリング公式について、その逆変換は、定義間隔の端点で f(t) の値が零になるときに成り立つ。従ってこの変換は、ベッセル微分方程式についてのディリクレ問題を直交固有関数について拡張したものと考えることができる。

31.2 関数

[Function] gsl_dht * gsl_dht_alloc (size_t size)

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。

[Function] int gsl_dht_init (gsl_dht * t, double nu, double xmax)

与えられる nu と x を使ってインスタンス t を初期化する。

[Function] gsl_dht * gsl_dht_new (size_t size, double nu, double xmax)

与えられる nu と x を使ってインスタンス t を初期化する。

[Function] void gsl_dht_free (gsl_dht * t)

インスタンス t のメモリを解放する。

[Function] int gsl_dht_apply (const gsl_dht * t, double * f_in, double * f_out)

インスタンス t 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。

[Function] double gsl_dht_x_sample (const gsl_dht * t, int n)

単位長間隔で 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。

[Function] double gsl_dht_k_sample (const gsl_dht * t, int n)

「k 空間」での単位長間隔で 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, Comp. Phys. Comm. 43, 181 (1987).
- D. Lemoine, J. Chem. Phys. 101, 3936 (1994).

第 32 章 一次元関数の求根法

この章では、任意の一次元関数に対する求根法のルーチンに関して説明する。GSL には繰り返し計算による求根法と収束判定を行うための低レベルルーチンをいくつか用意している。利用者は繰り返し計算の内部の進行を確認しながらこれらを適宜組み合わせて求根プログラムを作る必要がある。これらのメソッドの各クラスは同じフレームワークを使用しており、利用者は実行時にこれらのメソッドを切り替えて使うことができる。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 と改善法 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、根を含むことが知られている有限の領域を指定する。この領域が繰り返し計算でだんだんと小さく狭められ、十分に絞り込まれたところで終了する。根のある場所について、精密に誤差を評価できる。

改善法は根の初期推定値を改善していこうとする方法である。この方法は初期値が解に「十分に近い」場合にのみ収束し、収束は速いが誤差評価は精密にはできない。根の近傍で関数の形を近似することで、初期値を高次の速さで改善することを図る。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、この方法は高速に収束する。

GSL では両方の求根法を同じフレームワークで用意している。求根の各ステップで必要になる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最小化ルーチンを書く。繰り返し計算は主に以下の三段階からなる。

- 求根法のインスタンス T に対して、探索点 s を初期化する。
- T による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。

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）。改善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。更新には関数とその導関数（関数名 fdf）を用いる。導関数は利用者が指定する。

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予測するのは困難である。複数の根を含む領域で一つの根だけを探してしまっても、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。根が偶重根であるような場合には囲い込み法は使えない。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である。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。囲い込み法は奇重根（三次、五次、…）の場合には使うことができる。改善法は高次の重根の場合にも使うことができるが、収束が遅くなる。その場合にはステフェンセンの方法 Steffenson 法を使えば重根に対する収束を加速することができる。

f が探索領域内に根を持つことは、必ずしも要求されるわけではない。したがって数値的求根法

を、根が存在するか否かを知るために使うべきではない。それには他によい方法がある。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、特性をあまり理解してい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

[Function] gsl_root_fsolver * gsl_root_fsolver_alloc (const gsl_root_fsolver_type * T)

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。

[Function] gsl_root_fdfsolver * gsl_root_fdfsolver_alloc (const gsl_root_fdfsolver_type * T)

勾配法 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・ラプソン法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。

[Function] int gsl_root_fsolver_set (gsl_root_fsolver * s, gsl_function * f, double x_lower, double x_upper)

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（再初期化）し、探索幅の初期値を [x_lower, x_upper] に設定する。

[Function] int gsl_root_fdfsolver_set (gsl_root_fdfsolver * s, gsl_function fdf * fdf, double root)

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（再初期化）し、探索点の初期値を root に設定する。

[Function] void gsl_root_fsolver_free (gsl_root_fsolver * s)

[Function] void gsl_root_fdfsolver_free (gsl_root_fdfsolver * s)

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。

[Function] const char * gsl_root_fsolver_name (const gsl_root_fsolver * s)

[Function] const char * gsl_root_fdfsolver_name (const gsl_root_fdfsolver * s)

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` と出力する。

32.4 対象とする関数の設定

求根法のインスタンスに対しては、一変数の連続関数、および求根法によっては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。

[Data Type] gsl_function

この型はパラメータで記述される関数の一般的な型である

`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。

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。利用者が求根法インスタンスに渡す関数 `gsl_function F` は以下のように定義する。

```
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)
```

[Data Type] gsl_function_fdf

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。

`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 * d_f)`
この関数は引数が x、パラメータが params の時の関数 f の値を f(x, params)に、その導関数 df の値を d_f (x, params) に代入する。f(x)および d_f(x) の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) の値は以下のマクロ使って計算できる。

```
#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

利用者は探索範囲または推定値（探索開始点）のどちらかを指定するが、この節ではそれらにどのような役割があり、関数の引数でそれらがどのように扱われるかを説明する。

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繰り返し推定が続けられる。これは `double` 型である。

探索範囲とはある区間の両端のことであり、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、そのときの条件による。

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求根法による繰り返し計算による探索点の更新を 1 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、実行時に切り替えることができる。

[Function] int gsl_root_fsolver_iterate (gsl_root_fsolver * s)

[Function] int gsl_root_fdfsolver_iterate (gsl_root_fdfsolver * s)

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。これらを以下の補助的な関数で参照することができる。

[Function] double gsl_root_fsolver_root (const gsl_root_fsolver * s)

[Function] double gsl_root_fdfsolver_root (const gsl_root_fdfsolver * s)

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。

[Function] double gsl_root_fsolver_x_lower (const gsl_root_fsolver * s)

[Function] double gsl_root_fsolver_x_upper (const gsl_root_fsolver * s)

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 探索終了条件

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。

- 利用者が指定した精度で根が得られた時。
- 繰り返し計算の回数が、利用者が指定した回数に達した時。
- エラーが発生した時。

これらの条件は、利用者が設定するとこができる。以下の関数で、現時点での最良探索点に対して精度の標準的な検証ができる。

[Function] int gsl_root_test_interval (double x_lower, double x_upper, double epsabs, double epsrel)

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た時 GSL_SUCCESS を返す。

$$|a - b| < \text{epsabs} + \text{epsrel}\ \min(|a|, |b|)$$

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。

$$|r - r^*| < \text{epsabs} + \text{epsrel}\ r^*$$

[Function] int gsl_root_test_delta (double x1, double x0, double epsabs, double epsrel)

この関数は絶対誤差が epsabs で相対誤差が epsrel のときの列 . . . 、x0、x1 の収束を判定する。この関数は以下の条件が真になったとき GSL_SUCCESS を返す。そうでないときには GSL_CONTINUE を返す。

$$|x_1 - x_0| < \text{epsabs} + \text{epsrel}\ |x_1|$$

[Function] int gsl_root_test_residual (double f, double epsabs)

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たときに GSL_SUCCESS を返す。

$$|f| < \text{epsabs}$$

条件が満たされないときには 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、初期探索区間が必ず根を含むことが必要である。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。こうした区間が初期区間として与えられれば、関数が特殊な形式でない限り囲い込み法は成功する。

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。

[Solver] gsl_root_fsolver_bisection

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。この操作が制限なく、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、各回の中点の値である。

[Solver] gsl_root_fsolver_falsepos

失敗点法 false position algorithm は線形な補間を使った求根法である。収束も線形であるが、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。

各時点での根の近似解は、そのときの探索区間の両端を線形に補間して決められる。

[Solver] gsl_root_fsolver_brent

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）は二分法と線形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、それを使ってより小さな区間を作る。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。

各時点での根の最良近似は、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

この節で述べる改善法は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（二次収束する）。

これらの方法では、関数とその導関数の両方を使う。

[Derivative Solver] gsl_root_fdfsolver_newton

代表的な改善法としてニュートン法がある。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の関数 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。

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

ニュートン法は一つの根に対して二次収束し、重根に対しては線形に収束する。

[Derivative Solver] gsl_root_fdfsolver_secant

割線法 secant method はニュートン法を簡略化したものであり、繰り返し計算の各回での導関数の計算を必要としない。

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。

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

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。

$$x_{i+1} = x_i - \frac{f(x_i)}{f'_{est}}, \quad \text{where} \quad f'_{est} = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

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 + √5)/2（約 1.62）である。重根の場合には線形収束である。

[Derivative Solver] gsl_root_fdfsolver_steffenson

ステフェンソンの方法 Steffenson Method はここに挙げるルーチンの中では最も速い方法である。これは基本的なニュートン法とエイトケンの「デルタ二乗」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別に 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。加速項の分母が零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。関数のパラメータを定義するためにはヘッダファイル（‘demo_fn.h’）をインクルードする。

```
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);
```

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;
}

doublequadratic_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 $x = \sqrt{5} = 2.236068...$ 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);

    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
```

sl_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、二分法の収束が遅いが分かる。

```
$ ./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);

    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。

- R. P. Brent, “An algorithm with guaranteed convergence for finding a zero of a function”, Computer Journal, 14 (1971) 422–425
- J. C. P. Bus and T. J. Dekker, “Two Efficient Algorithms with Guaranteed Convergence for Finding a Zero of a Function”, ACM Transactions of Mathematical Software, Vol. 1 No. 4 (1975) 330–345

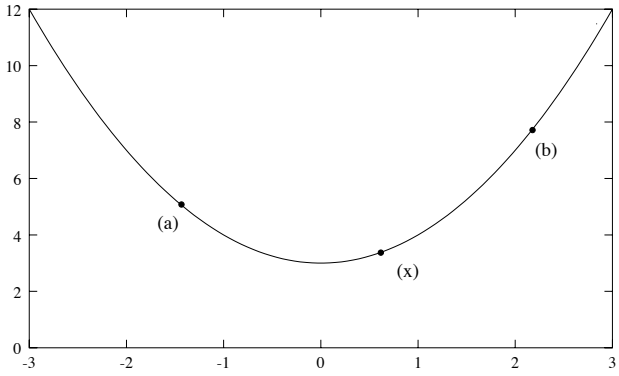
第 33 章 一次元関数の最適化

この章では、任意の一次元関数に対する最小化のルーチンに関して説明する。このライブラリには、繰り返し計算による最小化法と収束判定を行うための低レベルルーチンをいくつか用意している。利用者は繰り返し計算の内部の進行を確認しながらこれらを適宜組み合わせて最小化プログラムを作る必要がある。これらのメソッドの各クラスは同じフレームワークを使用しており、利用者は実行時にこれらのメソッドを切り替えて使うことができる。その際、プログラムの再コンパイルは不要である。最小化の各インスタンスは探索点をそれぞれ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。

最小化関数とそれに関わる各種宣言はヘッダファイル ‘gsl_min.h’ に記述されている。関数の最大値を求める問題に最小化プログラムを使いたいときは、単に関数の符号を反転すればよい。

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領域を探索領域として指定しなければならない。領域は下端 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 では繰り返し計算の各回で、用意されている方法のいずれかを使って新しい探索点 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 を更新する。また同時に、f (a) > f (x) < f (b) という条件を満たすような点のもっとも小さな集合を選ぶことにより、探索区間の幅を縮小することができる。探索区間の幅は、真の最小値を囲む幅が要求される幅になるまで縮小される。これにより最小点の位置を推定し、そのときの誤差を精密に決めることができる。

GSL では、数種類の囲い込み法を同じフレームワークで使えるようにしている。また各ステップで必要になるそれぞれ独立した関数が用意しており、利用者はこれらを使って高レベルの最小化ルーチンを書く必要がある。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。

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

最小化関数は一度に一つの最小値しか探索しない。探索領域内に複数の極小点がある場合、最初に見つけた点を探索結果として返すが、どの極小点が最初に見つかるかを予想するのは困難である。極小値が複数ある領域で一つの最小値だけを見つけても、ほとんどの場合なんのエラーもでない。

どの最小化法を使っても、数値的な精度の最高限度で最小点を探索することは困難である。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それが失われることがある。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。もっともよい対応策は、最小点の座標ではなく関数値を収束させることである。

33.3 最小化インスタンスの初期化

[Function] gsl_min_fminimizer * gsl_min_fminimizer_alloc (const gsl_min_fminimizer_type * T)

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。

[Function] int gsl_min_fminimizer_set (gsl_min_fminimizer * s, gsl_function * f, double x_minimum, double x_lower, double x_upper)

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。

探索区間に最小値が含まれていない場合、この関数はエラーコード GSL_FAILURE を返す。

[Function] int gsl_min_fminimizer_set_with_values (gsl_min_fminimizer * s, gsl_function * f, double x_minimum, double f_minimum, double x_lower, double f_lower, double x_upper, double f_upper)

gsl_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利用者が与える f_minimum、f_lower、f_upper を用いる。

[Function] void gsl_min_fminimizer_free (gsl_min_fminimizer * s)

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。

[Function] const char * gsl_min_fminimizer_name (const gsl_min_fminimizer * s)

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 最小化される関数の設定

最小化される関数として利用者は、一変数の連続関数を設定しなければならない。関数で汎用的なパラメータを使えるようにするため、関数は gsl_function 型として定義する必要がある（32.4 節「対象とする関数の設定」参照）。

33.5 繰り返し計算

以下の関数は各最小化法での繰り返し計算を行う。各関数は、繰り返し計算の一回を行い探索点を更新する。各関数はすべての最小化法に対して使うことができ、実行時にそれぞれ、プログラムを書き換えることなく切り替えて使うことができる。

[Function] int gsl_min_fminimizer_iterate (gsl_min_fminimizer * s)

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。

[Function] double gsl_min_fminimizer_x_minimum (const gsl_min_fminimizer * s)

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。

[Function] double gsl_min_fminimizer_x_upper (const gsl_min_fminimizer * s)

[Function] double gsl_min_fminimizer_x_lower (const gsl_min_fminimizer * s)

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。

[Function] 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)

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

最小化は以下の条件のいずれかが成立したときに停止する。

- 利用者が指定する精度で最小点が見つかったとき。
- 繰り返し計算の回数が利用者の指定回数に達したとき。
- 何らかのエラーが発生したとき。

これらの条件は利用者が設定することができる。以下の関数を使って現時点での最良探索点の精度を判定することができる。

[Function] int gsl_min_test_interval (double x_lower, double x_upper, double epsabs, double epsrel)

この関数は指定される絶対誤差 epsabs および相対誤差が 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た時 GSL_SUCCESS を返す。

$$|a - b| < \text{epsabs} + \text{epsrel} \min(|a|, |b|)$$

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|) が零（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。

$$|x_m - x_m^*| < \text{epsabs} + \text{epsrel}\, x_m^*$$

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、初期探索区間内に最小値が含まれていることが必要である。つまり探索区間の両端を a、b、予想される最小点を 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。

[Minimizer] gsl_min_fminimizer_goldensection

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、両端点それぞれから現時点での最小点までの二つの小区間を比較する。大きな方の小区間を黄金比で分割し（よく知られている比 $(3-\sqrt{5})/2 = 0.3189660\ldots$ である）、新しく決めた点での関数値を計算する。新しい値は、次に採用する区間に最小値を含ませるために f(a') > f(x') < f(b')という条件を満たすときに採用され、関数値のもっとも大きな点を捨てる。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として黄金比を使うことで、この種の方法では最も速い収束を示す。

[Minimizer] gsl_min_fminimizer_brent

ブレントの最小化法 Brent minimization algorithm 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 = π を求めるる。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);

    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 Brent, Algorithms for minimization without derivatives, Prentice-Hall (1973), republished by Dover in paperback (2002), ISBN 0-486-41998-3.

第 34 章 多次元関数の求根法

この章では、多次元空間での求根法（n が未知のときの n 個の非線形方程式系の解法）について説明する。GSL には繰り返し計算による求根法と収束判定を行うための低レベルルーチンをいくつか用意している。利用者は繰り返し計算の内部の進行を確認しながらこれらを適宜組み合わせて求根法プログラムを作る必要がある。これらのメソッドの各クラスは同じフレームワークを使用しており、利用者は実行時にこれらのメソッドを切り替えて使うことができる。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それぞれで常に保持しており、プログラムをマルチスレッドに対応させることができる。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

多次元空間での求根法では、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は使えず、解の存在を確認する方法もない。どの手法でも根の初期推定値から、下のようにニュートン法のようなステップで探索を進める。

$$x \to x' = x - J^{-1} f(x)$$

ここで x と f はベクトル、J はヤコビアン行列 $J_{ij} = \partial f_i / \partial x_j$ である。これに根に収束できる探索点の範囲を広げるための改良を加える。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勾配の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。

いくつかの求根法が一つのフレームワークで使える。利用者は高レベル探索ルーチンを、ライブラリが持つ各ステップでそれぞれ独立した関数を使って書く必要がある。繰り返し計算には主に、以下の三段階がある。

- 求根法 T の探索点 s を初期化する。
- T の繰り返し計算を使って 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。

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。

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、それは利用者が定義せねばならない。ヤコビアン行列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（導関数は必要とされない）。これらの方法では行列 J または 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、導関数を必要とする求根法および必要とし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。

[Function] gsl_multiroot_fsolver * gsl_multiroot_fsolver_alloc (const gsl_multiroot_fsolver_type

*** T, size_t n)**

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。以下のコードは三次の連立方程式を解く組み合わせ法の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。

[Function] gsl_multiroot_fdfsolver * gsl_multiroot_fdfsolver_alloc (const gsl_multiroot_fdfsolver_type * T, size_t n)

探索空間の次元が n の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・ラプソン法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。

[Function] int gsl_multiroot_fsolver_set (gsl_multiroot_fsolver * s, gsl_multiroot_function * f, gsl_vector * x)

この関数は、既に生成されているインスタンス s を関数 f に探索開始点 x で適用するために初期化（再初期化）する。

[Function] int gsl_multiroot_fdfsolver_set (gsl_multiroot_fdfsolver * s, gsl_multiroot_function fdf * fdf, gsl_vector * x)

この関数はすでにある求根法のインスタンス s に、関数と導関数 fdf 、根の初期推定値 x を使うように設定あるいは再設定する。

[Function] void gsl_multiroot_fsolver_free (gsl_multiroot_fsolver * s)

[Function] void gsl_multiroot_fdfsolver_free (gsl_multiroot_fdfsolver * s)

この関数は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。

[Function] const char * gsl_multiroot_fsolver_name (const gsl_multiroot_fsolver * s)

[Function] const char * gsl_multiroot_fdfsolver_name (const gsl_multiroot_fdfsolver * s)

この関数は、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 対象とする関数の設定

求根法ルーチンを利用するためには、n 変数の関数を n 個指定しなければならない。関数で汎用的にパラメータを利用するためには、それらの関数は以下の型で定義されなければならない。

[Data Type] gsl_multiroot_function

この型はパラメータを含む一般的な関数系を定義する。

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`
この関数は、関数 f(x, params) に引数 x とパラメータ params を与えて評価したときの結果をベクトル 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。

`size_t n`
これは系の次元数、例えばベクトル x や f の要素の個数である。

`void * params`
これは関数に渡すパラメータへのポインタである。

以下にパウエルの関数をテストに使った例を示す。

$$f_1(x) = Ax_0x_1 - 1,\ f_2(x) = \exp(-x_0) + \exp(-x_1) - (1 + 1/4)$$

ここで A = 10^4 である。以下のコードは `gsl_multiroot_function` の系 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;
```

[Data Type] gsl_multiroot_function_fdf

この型はパラメータを持つ一般的な関数系でヤコビアン行列を持つものを定義する。

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`
この関数は、引数が x でパラメータが params のときの関数値 f(x, params) を f に入れ、関数値が計算できないときは場合に応じた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 のときに得られる nxn 行列 $J_{ij} = \partial f_i$ (x, params)/∂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。

`int (* fdf) (const gsl_vector * x, void * params, gsl_vector * f, gsl_matrix * J)`
この関数は数 x、パラメータ params のときの f と J の値を一つ前の関数と同様に代入する。これにより、別の関数である f(x) と J(x) を同時に計算したいときにも、計算時間を短縮することができる。

`size_t n`

これは系の次元数、たとえばベクトル x や f 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。

前述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。

[Function] int gsl_multiroot_fsolver_iterate (gsl_multiroot_fsolver * s)

[Function] int gsl_multiroot_fdfsolver_iterate (gsl_multiroot_fdfsolver * s)

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常にその時点での解の近似を保持している。これらは以下の補助的な関数で参照することができる。

[Function] gsl_vector * gsl_multiroot_fsolver_root (const gsl_multiroot_fsolver * s)

[Function] gsl_vector * gsl_multiroot_fdfsolver_root (const gsl_multiroot_fdfsolver * s)

求根法のインスタンス s のその時点での解を返す。

[Function] gsl_vector * gsl_multiroot_fsolver_f (const gsl_multiroot_fsolver * s)

[Function] gsl_vector * gsl_multiroot_fdfsolver_f (const gsl_multiroot_fdfsolver * s)

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。

[Function] gsl_vector * gsl_multiroot_fsolver_dx (const gsl_multiroot_fsolver * s)

[Function] gsl_vector * gsl_multiroot_fdfsolver_dx (const gsl_multiroot_fdfsolver * s)

インスタンス s が直前にとったステップ幅 dx を返す。

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。

- 解が利用者の指定する精度で得られたとき。
- 繰り返し計算の回数が利用者の指定する回数に達したとき。
- エラーが発生したとき。

これらの条件は利用者が設定することができる。また以下の関数を使ってその時点での解の精度を、標準的な方法で判定できる。

[Function] int gsl_multiroot_test_delta (const gsl_vector * dx, const gsl_vector * x, double epsabs, double epsrel)

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。

$$|dx_i| < \text{epsabs} + \text{epsrel}\,|x_i|$$

[Function] int gsl_multiroot_test_residual (const gsl_vector * f, double epsabs)

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

この節で述べる求根法は関数とその導関数の両方を使う方法である。これらの方法は根の初期推定値を必要とするが、必ずしも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、二次収束を示す。

[Derivative Solver] gsl_multiroot_fdfsolver_hybridsj

これは MINPACK に hybrj として実装されているパウエルの修正組み合わせ法である。MINPACK は Jorge J. Moŕe、Burton S. Garbow、Kenneth E. Hillstrom の三人によって書かれた。修正組み合わせ法はニュートン法の収束の速さを保ちながら、その残差が小さくなりにくいという欠点を改善する方法である。

この方法では探索ステップを制御するために信頼区域を設定する。新しい探索点候補 x を採用するかどうかは、D が係数対角行列、δ が信頼区域の大きさのとき、|D(x' − x)| < δ という条件を満たすかどうかで決定する。D の要素は、x の各要素に対する残差の感度を計算するため、ヤコビアンの列ノルムを使って内部で計算される。これにより、スケールが大きく変動する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、J dx = −f を解いてニュートン法でステップを決める。ステップが信頼区域外に出てしまったときは、これを次の段階で最初のステップとして使う。信頼区域内だったら、領域内で関数のノルムが最小化となるような、勾配の方向とニュートン法によるステップの線形結合を使う。

$$dx = -\alpha J^{-1} f(x) - \beta \nabla |f(x)|^2$$

このニュートン法と勾配方向の線形結合はドッグレッ゛dogleg step と呼ばれる。

次に、こうして決めたステップにより決まる点 x 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くならその点を採用し、信頼区域を拡張する。このステップで解が改善されない場合は信頼区域を狭くし、ステップを計算しなおす。

この方法では、ヤコビアンの近似の変化量を階数 1 の更新（rank-1 update）で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。

[Derivative Solver] gsl_multiroot_fdfsolver_hybridj

hybridsj と同じだがスケーリングを行わない方法である。ステップは一般的な形ではなく、球状の信頼区域 |x' − x| < δ にとどまるように制御される。この方法は hybridsj に

よる一般的な形の信頼区域が適切でないときに有用である

[Derivative Solver] gsl_multiroot_fdfsolver_newton

ニュートン法はもっともよく使われる、標準的な求根法である。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、残差の各要素が零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。

$$J dx = -f(x)$$

[Derivative Solver] gsl_multiroot_fdfsolver_gnewton

これは修正ニュートン法で、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

この節で述べる方法は、利用者が導関数を定義しなくてもいい方法である。必要となる微分値はすべて有限差分から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不適当な値になる場合は、前節の関数を使ってユーザーが導関数値を計算しておいて与えることができる。

[Solver] gsl_multiroot_fsolver_hybrids

これは修正組み合わせ法だが、ヤコビアン行列の計算をする代わりに有限差分近似を使う。`GSL_SQRT_DBL_EPSILON` に対する相対的なステップ幅と `gsl_multiroots_fdjac` を使って有限差分を計算する。

[Solver] gsl_multiroot_fsolver_hybrid

これはスケーリングを使わない修正組み合わせ法と同じだが、有限差分を使う。

[Solver] gsl_multiroot_fsolver_dnewton

離散ニュートン法 discrete Newton algorithm はもっとも単純な多次元求根法である。これは以下のニュートン法の繰り返し計算を行う。

$$x \to x - J^{-1} f(x)$$

ここでのヤコビアン行列 J を関数 f の有限差分で近似する。GSL での近似法を以下に示す。

$$J_{ij} = (f_i(x + \delta_j) - f_i(x))/\delta_j$$

ここで δ_j は、ε を機械イプシロン（$\varepsilon \approx 2.22 \times 10^{-16}$）とするときのステップ幅 $\sqrt{(\varepsilon)}|x_j|$ である。ニュートン法は二次収束するが、有限差分の計算には n^2 回の関数値の計算が繰り返し計算の各回で必要である。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。

[Solver] gsl_multiroot_fsolver_broyden

ブロイデン法 Broyden algorithm は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 で以下のように近似する。

$$J^{-1} \rightarrow J^{-1} - (J^{-1}df - dx)dx^T J^{-1}/dx^T J^{-1}df$$

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で、有限差分を使ってヤコビアン行列の逆行列を計算する。この近似で探索点の更新を速くできるが、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)。

この方法はあまり推奨しない。単に手法のデモンストレーションとして用意してある。

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。最初にスケーリングを行い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

$$f_1(x, y) = a(1 - x), \quad f_2(x, y) = b(y - x^2)$$

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、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の信頼領域を表すパラメータの調整や、収束が悪くなってきたときのヤコビアン行列の再計算なども行われる。

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最適化することができる。また f と df をそれぞれ別に呼び出ださずにすむため、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。現実にある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。根に向かって坂を下る修正組み合わせ法がより信頼性が高い。

34.9 参考文献

オリジナルの修正組み合わせ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。

- J.J. More, M.Y. Cosnard, “Numerical Solution of Nonlinear Equations”, ACM Trans- actions on Mathematical Software, Vol 5, No 1, (1979), p 64–85
- C.G. Broyden, “A Class of Methods for Solving Nonlinear Simultaneous Equations”, Mathematics of Computation, Vol 19 (1965), p 577–593
- J.J. Mor̀e, B.S. Garbow, K.E. Hillstrom, “Testing Unconstrained Optimization Software”, ACM Transactions on Mathematical Software, Vol 7, No 1 (1981), p 17–41

第 35 章 多次元関数の最適化

この章では、任意の多次元関数に対する最小化のルーチンに関して説明する。このライブラリには、いくつかの繰り返し計算による最小化法と収束判定を行うための低レベルルーチンを用意している。利用者は繰り返し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最小化プログラムを作る必要がある。これらのメソッドの各クラスは同じフレームワークを使用しており、利用者は実行時にこれらのメソッドを切り替えて使うことができる。その際、プログラムの再コンパイルは不要である。最小化の各インスタンスは探索点をそれぞれで常に保持しており、プログラムをマルチスレッドに対応させることができる。最小化法は、関数の符号を反転することで最大化にも使うことができる。

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 = ∇f は零になる。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。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。

ネルダーとミードのシンプレックス法は、そういった方法とは異なり n 次元の単体（シンプレックス）の頂点ベクトルとして n+1 個の探索パラメータ・ベクトルを使う。繰り返し計算の各回で最も悪い頂点を単純な位置転換で改善し、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。

いくつかの最小化法が同じフレームワークで使える。このライブラリには、各ステップで必要になるそれぞれ独立した関数が用意されており、利用者はこれらを使って高レベルの最小化ルーチンを書く必要がある。繰り返し計算には以下のように、主に三段階ある。

- 最小化インスタンス T の探索点 s を初期化する。
- 繰り返し計算 T で探索点 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。

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、または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すことはできない。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が結果として返されるが、どの極小点が最初に見つかるかを予測することは困難である。ほとんどの場合、複数の極小点を含む領域で一つの最小点を探そうとしても、何のエラーも出ない。

また、これらの最小化法は局所解を探すことしかできない。見つかった極小点が、問題となって

いる関数の大域的な最小点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。

[Function] gsl_multimin_fdfminimizer * gsl_multimin_fdfminimizer_alloc (const gsl_multimin_fdfminimizer_type * T, size_t n)

[Function] gsl_multimin_fminimizer * gsl_multimin_fminimizer_alloc (const gsl_multimin_fminimizer_type * T, size_t n)

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。

[Function] int gsl_multimin_fdfminimizer_set (gsl_multimin_fdfminimizer * s, gsl_multimin_function_fdf * fdf, const gsl_vector * x, double step_size, double tol)

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、厳密にはこのパラメータの意味は使う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・g < tol |p||g| が成立するときである。

[Function] int gsl_multimin_fminimizer_set (gsl_multimin_fminimizer * s, gsl_multimin_function * f, const gsl_vector * x, const gsl_vector * step_size)

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。

[Function] void gsl_multimin_fdfminimizer_free (gsl_multimin_fdfminimizer * s)

[Function] void gsl_multimin_fminimizer_free (gsl_multimin_fminimizer * s)

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。

[Function] const char * gsl_multimin_fdfminimizer_name (const gsl_multimin_fdfminimizer * s)

[Function] const char * gsl_multimin_fminimizer_name (const gsl_multimin_fminimizer * s)

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` と出力する。

```
printf ("s is a '%s' minimizer\n",
    gsl_multimin_fdfminimizer_name(s));
```

35.4 最小化される関数の設定

最小化したい関数は、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。最小化対象の関数定義に汎用のパラメータを使えるようにするため、利用者が定義する関数は以下の型で定義しなければならない。

[Data Type] gsl_multimin_function_fdf

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 $g_i = \partial f(x, params)/\partial x_i$ 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、たとえばベクトル 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。

[Data Type] gsl_multimin_function

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、たとえばベクトル 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 (dp[0],dp[1]) 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 10.0 * (x - dp[0]) * (x - dp[0]) +
           20.0 * (y - dp[1]) * (y - dp[1]) + 30.0;
}

/* 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, 20.0 * (x - dp[0]));
```

```
    gsl_vector_set(df, 1, 40.0 * (y - dp[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;
double p[2] = { 1.0, 2.0 }; /* 中心は (1,2) */
my_func.f = &my_f;
my_func.df = &my_df;
my_func.fdf = &my_fdf;
my_func.n = 2;
my_func.params = (void *)p;
```

35.5 繰り返し計算

以下の関数で、各最小化法の繰り返し計算を実際に行う。この関数は繰り返し計算の一回を行い、最小化法のインスタンスが持つ探索点を更新する。この関数はすべての最小化法に使うことができ、プログラムを変更することなく実行時に最小化法を切り替えることができる。

[Function] int gsl_multimin_fdfminimizer_iterate (gsl_multimin_fdfminimizer * s)

[Function] int gsl_multimin_fminimizer_iterate (gsl_multimin_fminimizer * s)

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。

[Function] 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)

これらの関数は、最小点の位置の、現時点での最良推定値、その点での関数値および導関数値、および各最小化法に特有の特徴量を返す。

[Function] int gsl_multimin_fdfminimizer_restart (gsl_multimin_fdfminimizer * s)

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。

・利用者が設定する精度で最小点が見つかったとき。

・利用者が設定する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、利用者が設定するとこができる。以下の関数で、現時点での結果の精度の標準的な検証ができる。

[Function] int gsl_multimin_test_gradient (const gsl_vector * g, double epsabs)

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、最小点で零になるはずである。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。

$$|g| < \text{epsabs}$$

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る。これらの量には $\delta f = g \delta x$ という関係がある。

[Function] int gsl_multimin_test_size (const double size, double epsabs)

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。問題によって、どの方法が適しているかは変わる。どの方法も、探索点における関数値を使い、一部の方法は導関数値も使う。

[Minimizer] gsl_multimin_fdfminimizer_conjugate_fr

フレッチャー - リーブズ法（Fletcher-Reeves の共役勾配法、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。この直線上での極小点では、関数の勾配 g と探索方向 p は直交しており、直線探索は $p \cdot g < tol|p||g|$ が成立したときに終了する。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。

[Minimizer] gsl_multimin_fdfminimizer_conjugate_pr

ポラックとリヴィエル Polak-Riviere の共役勾配法。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。

[Minimizer] gsl_multimin_fdfminimizer_vector_bfgs

ブロイデン - フレッチャー - ゴールドファーブ - シャノ 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。

[Minimizer] gsl_multimin_fdfminimizer_steepest_descen

最急降下法は、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。降下ステップをとった結果関数値が大きくなった

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 である。最級降下法はあまり性能が良くなく、デモンストレーションのためだけにこのライブラリに用意されている。

[Minimizer] gsl_multimin_fminimizer_nmsimplex

これはネルダー - ミード法（Nelder と Mead のシンプレックス法）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。

$$
\begin{aligned}
p_0 &= (x_0, x_1, \cdots, x_n) \\
p_1 &= (x_0 + stepsize_0, x_1, \cdots, x_n) \\
p_2 &= (x_0, x_1 + stepsize_1, \cdots, x_n) \\
\cdots &= \cdots \\
p_n &= (x_0, x_1, \cdots, x_n + stepsize_n)
\end{aligned}
$$

これらのベクトルは n 次元空間で単体の n + 1 個の頂点をなす。繰り返し計算の各ステップで、関数値が最も高い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。この変換は鏡映であり、続いて展開、縮約、多重縮約が行われる。これらの変換により単体はパラメータ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。この方法は、必ずしも最良パラメータベクトルを改善するとは限らないのが特徴である。それには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 x、y 座標でオフセットがあり、その点での関数値は零ではない。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。

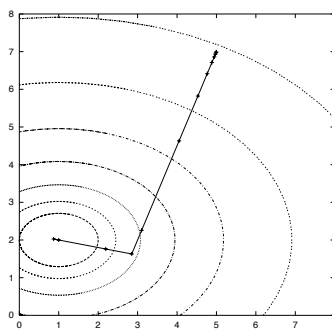
```
int main (void)
{
    size_t iter = 0;
    int status;
    const gsl_multimin_fdfminimizer_type *T;
    gsl_multimin_fdfminimizer *s;
    /* 最小点の位置は (1,2) */
    double par[2] = { 1.0, 2.0 };
    gsl_vector *x;
    gsl_multimin_function_fdf my_func;

    my_func.f = &my_f;
    my_func.df = &my_df;
    my_func.fdf = &my_fdf;
    my_func.n = 2;
    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[2] = {1.0, 2.0};
    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;

    /* 初期の頂点の大きさのベクトル */
    ss = gsl_vector_alloc (np);

    /* ステップ幅の初期値は全て 1 */
    gsl_vector_set_all (ss, 1.0);

    /* 探索開始点 */
    x = gsl_vector_alloc (np);
    gsl_vector_set(x, 0, 5.0);
    gsl_vector_set(x, 1, 7.0);

    /* インスタンスの初期化 */
    minex_func.f = &my_f;
    minex_func.n = np;
    minex_func.params = (void *)&par;
    s = gsl_multimin_fminimizer_alloc(T, np);

    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 ", iter);
         for (i = 0; i < np; i++)
              printf ("%10.3e ", gsl_vector_get (s->x, i));
         printf ("f() = %7.3f size = %.3f\n", s->fval, size);
    } 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

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”, p. 325–335, Springer (1997), ISBN 3-540-62057-5.

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 vol. 7 (1965), 308–315.

第 36 章 最小二乗法

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法近似を行うルーチンについて説明する。データには重み付けがあるものとないものを想定する。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と点の散らばりから共分散行列を得る。関数は、パラメータが 1 個あるいは 2 個のためのものと、それ以上のため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

最小二乗法による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 χ^2を最小化するようにしたものである。

$$\chi^2 = \sum_i w_i (y_i - Y(c_i x_i))^2$$

ベクトル c = {c_0, c_1, ...} 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に含まれる観測誤差であり、誤差の値は各データ点の間で独立で、正規分布することを仮定している。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 個のパラメータ最適値とその p×p 次の共分散行列を計算する。共分散行列は、データの誤差 σ_iから計算される最適パラメータ値のもつ統計的誤差を表し、<…> がデータ点に仮定した誤差正規分布の平均値を表すとしたとき、$C_{ab} = <\delta c_a \delta c_b>$ で定義される。

共分散行列はデータ点の誤差 σ_iから誤差から計算される。データ点の変化 δy_iによる最適パラメータの変化 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誤差で記述できる。

$$C_{ab} = \sum_{i,j} \frac{\partial c_a}{\partial y_i} \frac{\partial c_b}{\partial y_j} \langle \delta y_i \delta y_j \rangle$$

データが互いに独立で相関がない場合、データ点の変動は $<\delta y_i \delta y_j> = \sigma_i^2 \delta_{ij}$となる。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。

$$C_{ab} = \sum_i \frac{1}{w_i} \frac{\partial c_a}{\partial y_i} \frac{\partial c_b}{\partial y_j}$$

たとえば誤差が見積もられていないデータなどの重みのついていないデータに対して共分散行列を計算するときは、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 $\sigma^2 = \Sigma(y_i - Y(c_i, x_i))^2 / (n - p)$ からすべての w_iを $w=1/\sigma^2$とする。

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 $\sigma_{ca}=\sqrt{(C_{aa})}$で得られる。

36.2 線形回帰

この節では直線回帰モデル $Y = c_0 + c_1 X$ による近似を行う関数について説明する。

[Function] 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)**

与えられるデータセット(x、y)と歩幅 xstride および ystride のときの長さ n の二つのベクトル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(c0、c1)を計算する。二つのパラメータ(c0、c1)の分散共分散行列は最良近似直線に対する点の分布から計算され、パラメータ(cov00、cov01、cov11)として返される。最良近似直線からの残差の二乗和は sumsq に返される。

[Function] 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)

与えられる重み付きデータセット(x、y)と長さ n の二つの歩幅ベクトル xstride および ystride 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(c0、c1)を計算する。長さ n のベクトル w と歩幅 wstride でデータの各点の重みを指定する。重みはデータ y の各点に対する分散の逆数である。

パラメータ(c0、c1)の共分散行列は重み付きデータから計算され、パラメータ(cov00、cov01、cov11)として返される。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。

[Function] int gsl_fit_linear_est (double x, double c0, double c1, double c00, double c01, double c11, double * y, double * y_err)

この関数は最良近似直線のパラメータ c0 と c1 および計算された共分散 cov00、cov01、cov11 から近似関数値 y と、モデルを $Y = c_0 + c_1 X$ とするときの点 x における標準偏差 y_err を計算する。

36.3 定数項のない線形近似

この節で説明する関数は、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。

[Function] 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)

この関数は、与えられるデータセット(x、y)と歩幅 xstride および ystride のときの長さ n の二つのベクトル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点の分布から計算され、パラメータ cov11 に返される。最良近似直線からの残差の二乗和は sumsq に返される。

[Function] 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)

この関数は、与えられる重み付きデータセット(x、y)と歩幅 xstride および ystride のときの長さ n の二つのベクトル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 のベクトル w と歩幅 wstride でデータの各点の重みを指定する。重みはデータ y の各点に対する分散の逆数である。

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。

[Function] int gsl_fit_mul_est (double x, double c1, double c11, double * y, double * y_err)

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、モデルを $Y = c_1 X$ とするときの点 x における標準偏差 y_err を計算する。

36.4 重回帰

ここでは、y が n 個の観測値からなるベクトル、X が予測変数の n×p 行列、c が p 個のパラメータ値であるときに、最適なパラメータ値を最小二乗法で求める関数について説明する。

行列 X を適切に作ることで、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、...、w_pに固定した p 個の正弦関数で近似を行うには、

$$X_{ij} = \sin(w_j x_i)$$

とすればよい。また p 個の独立した変数 x_1、x_2、 ... 、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。

この節で説明する関数はヘッダファイル ‘`gsl_multifit.h`’ で宣言されている。

汎用の最小二乗法ルーチンから計算結果を得る間には、たとえば行列 X の特異値分解を行う場合など、途中の計算結果を保持するための作業領域用のメモリーを別途使うことがある。

[Function] gsl_multifit_linear_workspace * gsl_multifit_linear_alloc (size_t n, size_t p)

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。

[Function] void gsl_multifit_linear_free (gsl_multifit_linear_workspace * work)

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。

[Function] 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)

この関数は観測データ y と予測子変数の行列 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。モデルパラメータの分散共分散行列 cov は、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 に返される。

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。特異値分解は修正ゴルブ・ラインシュ法を用いるが、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（以上機械イプシロン以下）の要素は最適化には使われない。二つ目の関数では、特異値の比 s_i/s_0が利用者の指定する値 tol 以下の時も最適化には使われず、実質的なランクが rank に返される。

[Function] 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)

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。モデルパラメータの共分散行列 cov は、近似に対する重み付きデー

タから計算される。近似モデルからの重み付き残差二乗和 χ^2 は chisq に返される。

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。特異値が零（または機械イプシロン）になる要素は近似から削除される。特異値が 0（以上機械イプシロン以下）の要素は最適化には使われない。二つ目の関数では、特異値の比 s_i/s_0が利用者の指定する値 tol 以下の時も最適化には使われず、実質的なランクが rank に返される。

[Function] int gsl_multifit_linear_est (const gsl_vector * x, const gsl_vector * c, const gsl_matrix * cov, double * y, double * y_err)

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。

36.5 例

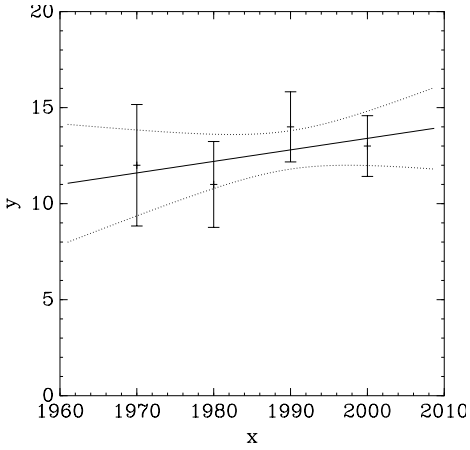
以下のプログラムは単純な（架空の）データセットに対して最小二乗線型近似を行い、近似直線と、各点での標準偏差のエラーバーを出力する。

```
#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。

$$X = \begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \dots & \dots & \dots \end{pmatrix}$$

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 の誤差(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;

    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_linear_workspace * work
         = gsl_multifit_linear_alloc (n, 3);
    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);

    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 $y = e^x$ に正規分布で誤差を加えた点を $0 < x < 2$ 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);
    }
    return 0;
}
```

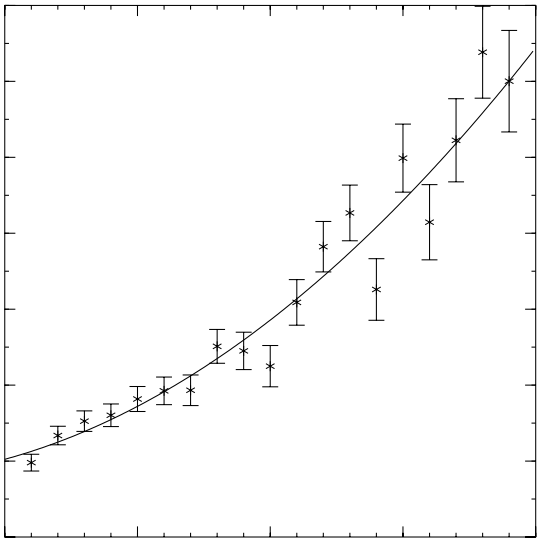
コンパイルしてできる実行ファイルは、以下のように出力する。

```
$ ./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 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、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。

- Review of Particle Properties, R.M. Barnett et al., 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 非線形最小二乗法

この章では多次元非線形最小二乗法について説明する。このライブラリには、いくつかの繰り返し計算による最小化法と収束判定を行うための低レベルルーチンを用意している。利用者は繰り返し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最小化プログラムを作る必要がある。これらのメソッドの各クラスは同じフレームワークを使用しており、利用者は実行時にこれらのメソッドを切り替えて使うことができる。その際、プログラムの再コンパイルは不要である。最小化の各インスタンスは探索点をそれぞれで常に保持しており、プログラムをマルチスレッドに対応させることができる。

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、n 個の関数 f_iの残差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を使って進んでいく。

$$\psi(p) = ||F(x+p)|| \approx ||f(x) + Jp||$$

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。他に収束範囲を広くするための手法を取り入れる。これらの手法には各ステップでノルム ||F|| を小さくすることが必要であったり、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を設定したりといった方法がある。

互いに独立な、正規分布する誤差 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。

$$f_i = \frac{Y(x, t_i) - y_i}{\sigma_i}$$

この節では非線形最小二乗法を空間中の座標を使って説明（局面上の最小値の探索など）するため、関数のパラメータを x で表す。近似対象となるデータの中の独立変数は t で表す。

上の定義に従い、Yi = 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

[Function] gsl_multifit_fsolver * gsl_multifit_fsolver_alloc (const gsl_multifit_fsolver type * T, size_t n, size_t p)

この関数は新しく生成した最小二乗法 T のインスタンスを、データ点数 n とパラメータ数 p として生成し、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ˇGSL_ENOMEM でエラーハンドラーを呼び出す。

[Function] gsl_multifit_fdfsolver * gsl_multifit_fdfsolver_alloc (const gsl_multifit_fdfsolver_type * T, size_t n, size_t p)

この関数は新しく生成した導関数を使う最小二乗法 T のインスタンスを、データ点数 n とパラメータ数 p として生成し、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。

[Function] int gsl_multifit_fsolver_set (gsl_multifit_fsolver * s, gsl_multifit_function * f, gsl_vector * x)

この関数は生成されている最小二乗法のインスタンス s に、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。

[Function] int gsl_multifit_fdfsolver_set (gsl_multifit_fdfsolver * s, gsl_function fdf * fdf, gsl_vector * x)

この関数は生成されている最小二乗法のインスタンス s に、関数と導関数 fdf、探索開始点 x を使うように設定、あるいは再設定する。

[Function] void gsl_multifit_fsolver_free (gsl_multifit_fsolver * s)

[Function] void gsl_multifit_fdfsolver_free (gsl_multifit_fdfsolver * s)

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。

[Function] const char * gsl_multifit_fsolver_name (const gsl_multifit_fdfsolver * s)

[Function] const char * gsl_multifit_fdfsolver_name (const gsl_multifit_fdfsolver * s)

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` と出力する。

```
printf("s is a '%s' solver\n", gsl_multifit_fdfsolver_name (s));
```

37.3 最小化される関数の設定

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。関数定義に汎用のパラメータを使えるようにするため、関数は以下の型で定義する必要がある。

[Data Type] gsl_multifit_function

`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。

`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。

[Data Type] gsl_multifit_function_fdf

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 nxp 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

以下の関数が各アルゴリズムの繰り返し計算を実際に行う。この関数は繰り返し計算の一回を行い、インスタンスが持つ探索点を更新する。この関数はすべての最小二乗法に使うことができ、プログラムを変更す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。

[Function] int gsl_multifit_fsolver_iterate (gsl_multifit_fsolver * s)

[Function] int gsl_multifit_fdfsolver_iterate (gsl_multifit_fdfsolver * s)

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。

[Function] gsl_vector * gsl_multifit_fsolver_position (const gsl_multifit_fsolver * s)

[Function] gsl_vector * gsl_multifit_fdfsolver_position (const gsl_multifit_fdfsolver * s)

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。

- 利用者が設定する精度で最小点が見つかったとき。
- 利用者が設定する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、利用者が設定するとこができる。以下の関数で、現時点での結果の精度の標準的な検証ができる。

[Function] int gsl_multifit_test_delta (const gsl_vector * dx, const gsl_vector * x, double epsabs, double epsrel)

この関数は許容絶対誤差が 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。

$$|dx_i| < epsabs + epsrel\ |x_i|$$

[Function] int gsl_multifit_test_gradient (const gsl_vector * g, double epsabs)

この関数は与えられる許容絶対誤差 epsabs に対する残差の勾配 g の収束を判定する。理論上、最小点では勾配は零になるはずである。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。

$$\Sigma_i |g_i| < epsabs$$

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ことができる。

[Function] int gsl_multifit_gradient (const gsl_matrix * J, const gsl_vector * f, gsl_vector * g)

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

この節で説明する最小化法は、関数と導関数の両方を使い、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適用できるような形であり、探索開始点が最小点に十分に近いことが必要である。

[Derivative Solver] gsl_multifit_fdfsolver_lmsder

これは MINPACK のスケーリング機能付き `lmder` として実装されているレベンバーグ・マルカルト法を、よりロバストで高速にしたものである。MINPACK は Jorge J. More、Burton S. Garbow、Kenneth E. Hillstrom によって書かれたものである。

このアルゴリズムはステップ幅を制御するために一般化信頼領域を設定する。新しく生成した探索点を採用するかどうかは、係数行列の対角成分を D、信頼領域の大きさを δ とするときに、その探索点が $|D(x' - x)| < \delta$ という条件を満たすかどうかで決める。D の値は関数の内部でヤコビアン行列の列ノルムを使って計算され、x の各要素の残差の感度を表す。これにより、関数値が激しく変化するような関数に対する最小化がしやすくなる。

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、信頼領域の大きさを縮小し、他のステップを生成する。

このアルゴリズムでは解の改善を見ながら、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、その時点での解よりもよい解は恐らく得られないであろうことを示す。

[Derivative Solver] gsl_multifit_fdfsolver_lmder

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

[Function] int gsl_multifit_covar (const gsl_matrix * J, double epsrel, gsl_matrix * covar)

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 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。

$$C = (J^T J)^{-1}$$

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）。

$$|R_{kk}| \le \mathrm{epsrel} |R_{11}|$$

37.9 例

以下のプログラムではバックグラウンドを含む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、40 時点分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]);
```

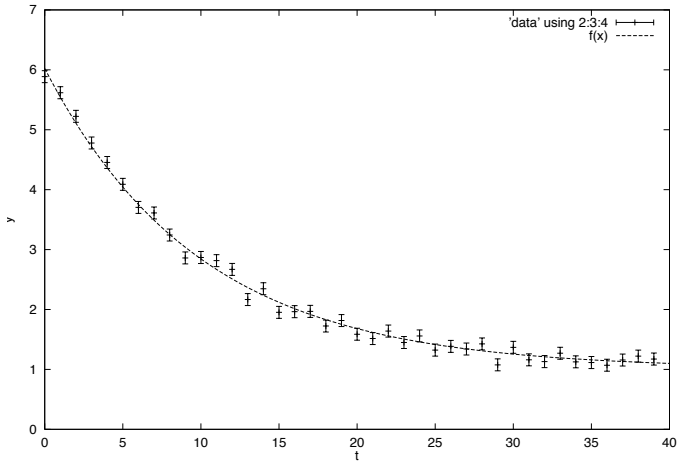
```
    };
    T = gsl_multifit_fdfsolver_lmsder;
    s = gsl_multifit_fdfsolver_alloc(T, n, p);
    gsl_multifit_fdfsolver_set(s, &f, &x.vector);
    print_state(iter, s);

    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)。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。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 $\chi^2/(n - p) >> 1$ 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.More, TheLevenberg-MarquardtAlgorithm: ImplementationandTheory, Lecture NotesinMathematics, v630(1978), ed G. Watson.

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。

- J.J. More, B.S. Garbow, K.E. Hillstrom, “TestingUnconstrainedOptimizationSoftware”, ACMTransactionsonMathematical Software, Vol 7, No1(1981), p17–41.

第 38 章 物理定数

この章では、光速 c や重力定数 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。その値は二つの単位系、標準的な MKSA（メートル、キログラム、秒、アンペア）と天文学でよく用いられる CGSM（センチメートル、グラム、秒、ガウス）で利用できる。

MKSA での定数の定義はファイル 'gsl_const_mksa.h' に書かれている。CGSM での定義は 'gsl_const_cgsm.h' にある。微細構造定数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。このライブラリで用意している定数の値そのものについては、ヘッダファイルを参照のこと。

38.1 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 $\mu 0$ 。MKSA でのみ定義されている。

GSL_CONST_MKSA_VACUUM_PERMITTIVITY

真空中の誘電率 0 。MKSA でのみ定義されている。

GSL_CONST_MKSA_PLANCKS_CONSTANT_H

プランク定数 h。

GSL_CONST_MKSA_PLANCKS_CONSTANT_HBAR

プランク定数を 2π で除した値 ħ。

GSL_CONST_NUM_AVOGADRO

アボガドロ定数 Na 。

GSL_CONST_MKSA_FARADAY

1 ファラデーの 1 モルの電荷。

GSL_CONST_MKSA_BOLTZMANN

ボルツマン定数 k。

GSL_CONST_MKSA_MOLAR_GAS

1 モルの気体定数 R0 。

GSL_CONST_MKSA_STANDARD_GAS_VOLUME

標準状態の気体の体積 V0 。

GSL_CONST_MKSA_STEFAN_BOLTZMANN_CONSTANT

ステファン・ボルツマンの放射定数 σ。

GSL_CONST_MKSA_GAUSS

1 ガウス Gauss の磁場。

38.2 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

地球上での標準的な重力加速度 g。

GSL_CONST_MKSA_SOLAR_MASS

太陽の質量。

38.3 原子物理学、核物理学

GSL_CONST_MKSA_ELECTRON_CHARGE

電子の電荷 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

エネルギー単位でのリュードベリ定数 Ry。この値はリュードベリの逆波長 R と Ry = hcR という関係がある。

GSL_CONST_MKSA_BOHR_RADIUS

ボーア半径 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

ボーアの磁子 μ_B。

GSL_CONST_MKSA_NUCLEAR_MAGNETON

核磁子 μ_N。

GSL_CONST_MKSA_ELECTRON_MAGNETIC_MOMENT

電子の磁気モーメントの絶対値 μe 。電子の物理的磁気モーメントは負である。

GSL_CONST_MKSA_PROTON_MAGNETIC_MOMENT

陽子の磁気モーメント μp 。

GSL_CONST_MKSA_THOMSON_CROSS_SECTION

トムソンの断面積 σT 。

GSL_CONST_MKSA_DEBYE

1 Debye のダイポールモーメント D。

38.4 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。

38.5 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 マイルの 1/1000）の長さ。

38.6 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 kilometer per hour の速度。

GSL_CONST_MKSA_MILES_PER_HOUR

時速 1 マイル mile per hour 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（訳注:6 フィート）。

GSL_CONST_MKSA_KNOT

1 ノット knot の速度。

38.7 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）。

38.8 長さ、面積、体積

GSL_CONST_MKSA_MICRON

1 ミクロン micron 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 liter 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。

38.9 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）。

SL_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 ouncee 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 の力。

38.10 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 のエネルギー。

GSL_CONST_MKSA_HORSEPOWER

1 馬力 hoursepower の仕事率。

38.11 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。

38.12 粘性

GSL_CONST_MKSA_POISE

1 ポアズ P, poise の粘性率。

GSL_CONST_MKSA_STOKES

1 ストークス St の動粘性率。

38.13 光と明かり

GSL_CONST_MKSA_STILB

1 スチルブ sb の輝度。

GSL_CONST_MKSA_LUMEN

1 ルーメン lumen の光束。

GSL_CONST_MKSA_LUX

1 ルクス lux の照度。

GSL_CONST_MKSA_PHOT

1 フォト phot の照度。

GSL_CONST_MKSA_FOOTCANDLE

1 フート燭 footcandle の照度。

GSL_CONST_MKSA_LAMBERT

1 ランベルト lambert の輝度。

GSL_CONST_MKSA_FOOTLAMBERT

1 フットランベルト footlambert の輝度。

38.14 放射性

GSL_CONST_MKSA_CURIE

1 キュリーの放射能。

GSL_CONST_MKSA_ROENTGEN

1 レントゲン roentgen の照射線量。

GSL_CONST_MKSA_RAD

1 ラド rad の吸収線量。

38.15 力とエネルギー

GSL_CONST_MKSA_NEWTON

SI 単位での力、1 ニュートン Newton。

GSL_CONST_MKSA_DYNE

1 ダイン dyne の力。$= 10^{-5}$ [N]

GSL_CONST_MKSA_JOULE

SI 単位でのエネルギー、1 ジュール Joule。

GSL_CONST_MKSA_ERG

1 エルグ erg のエネルギー。 $= 10^{-7}$[J]

38.16 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}

38.17 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）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、これらの値を定数で換算する。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;

     /* 距離はメートル単位になっている */
     double r_earth = 1.00 * au;
     double r_mars = 1.52 * au;
```

```
    double t_min, t_max;
    t_min = (r_mars - r_earth) / c;
    t_max = (r_mars + r_earth) / c;
    printf ("light travel time from Earth to Mars:\n");
    printf ("minimum = %.1f minutes\n", t_min / minutes);
    printf ("maximum = %.1f minutes\n", t_max / minutes);

    return 0;
}
```

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
```

38.18 参考文献

以下に示す 2002 CODATA recommended values に、信頼できる物理定数の値が載っている。ここに上げた定数について詳しくは、NIST のウェブサイトで見ることができる。

- Journal of Physical and Chemical Reference Data, 28(6), 1713-1852, 1999 Reviews of Modern Physics, 72 (2), 351-495, 2000
- http://www.physics.nist.gov/cuu/Constants/index.html
- http://physics.nist.gov/Pubs/SP811/appenB9.html

第 39 章 IEEE 浮動小数点演算

この章では浮動小数点数の内部表現を調べ、利用者が作成する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。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。

39.1 浮動小数点の内部表現

IEEE が定める標準の二値浮動小数点演算では、単精度及び倍精度実数の二値表現を定義している。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。これらが表す数値は以下の式で与えられる。

$$(-1)^s(1.\mathrm{fffff}\ldots)2^E$$

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る最小値 Emin から最大値 Emax の範囲内の値である。指数部はバイアス付き指数部 biased exponent とも呼ばれる符号なしの値 e に、バイアス bias パラメータ e = E + bias を使って変換される。上の式の fffff... 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 × Emin 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。

$$(-1)^s(0.\mathrm{fffff}\ldots)2^{E_{\min}}$$

これにより精度が p ビットのときのアンダーフローを $2^{E_{\min}-p}$ に下げることができる。零は指数部を $2^{E_{\min}-1}$ 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 ビットを以下のように分けることで表現される。

```
seeeeeeeefffffffffffffffffffff
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
```

単精度実数は 64 ビットを以下のように分けることで表現される。

```
seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
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
```

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。このライブラリ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。

[Function] void gsl_ieee_fprintf_float (FILE * stream, const float * x)

[Function] void gsl_ieee_fprintf_double (FILE * stream, const double * x)

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）が起こ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、どの数値でもない数であることを表す。

`Inf, -Inf`
: 正または負の無限大であることを表す。

`1.fffff...*2^E, -1.fffff...*2^E`
: 正規形式の浮動小数点であることを表す。

`0.fffff...*2^E, -0.fffff...*2^E`

不正規化形式の浮動小数点であることを表す。

`0, -0`

正または負の零であることを表す。

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る。これは GNU Emacs の Calc モードでそのまま利用することができる。

[Function] void gsl_ieee_printf_float (const float * x)

[Function] void gsl_ieee_printf_double (const double * x)

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; /* promote from float to double */
    printf(" f="); gsl_ieee_printf_float(&f);
    printf("\n");
    printf("fd="); gsl_ieee_printf_double(&fd);
    printf("\n");
    printf(" d="); gsl_ieee_printf_double(&d);
    printf("\n");

    return 0;
}
```

1/3 を二進数で表現すると 0.01010101... である。以下の出力例では、正規形式に仮数部の先頭ビットの 1 加えて表示している。

```
 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、二進数表現に零を付け加えることで行われることも示されている。

39.2 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がいくつか定義されている。各モードでは重要な特性、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。

これらの特性は標準関数 `fpsetround` のような関数が使えるときには、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` 関数を呼ぶことでプラットフォームに特有の初期化処理を行う。

[Function] void gsl_ieee_env_setup ()

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中の一つとする。

```
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 IEEE モードを変更せず、なにもしない。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。

以下のようなモードの組合せが便利であることが多い。

```
GSL_IEEE_MODE="double-precision,"
              "mask-underflow,"
              "mask-denormalized"
```

この組合せでは、小さい数に関するエラー(不正規化形式や零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。

丸めのモードを切り替えるとどうなるかを、以下の非常に速く収束する級数で自然対数の底 e を計算するプログラムで示す。

$$e = 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots = 2.71828182846\ldots$$

```
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 4×10^{-16} に収束している。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
```

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 epsilon を適切な値にし、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 107 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。

39.3 参考文献

IEEE 規格は以下。

- ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

もう少し教科書的な解説が “What Every Computer Scientist Should Know About Floating-Point Arithmetic” という論文にある。

- David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Computing Surveys, Vol. 23, No. 1 (March 1991), pages 5–48.
- Corrigendum: ACM Computing Surveys, Vol. 23, No. 3 (September 1991), page 413.
- B. A. Wichmann and Charles B. Dunham in Surveyor’s Forum: “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. ACM Computing Surveys, Vol. 24, No. 3 (September 1992), page

319.

SIAM 出版から、IEEE 演算と例についての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.

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 上で利用者が 書いたプログラムを実行し、ブレイク・ポイントをこの関数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・コマント゛(bt) で関数呼び出しの階層と、エラーの原因かもしれない各関数呼び出しでの引数を確認することができる。呼び出し側の関数に戻っていくことで、呼び出し時点での変数の値を確認することができる。以下の行は、プログラム fft/test_trap に含まれているものである。

```
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 では長さ零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、そのときの引数では n=0 であることがわかる。元々の呼び出しはファイル 'test_trap.c'の 94 行目であることもわかる。

元々の呼び出しの階層まで上がることで、エラーが発生した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);
```

これでエラーが発生した行を見ることができる。ここでなら、たとえば complex_wavetable のような、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の内容は info float て゛(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
```

各レジスタは、その名前を reg とすると、$reg で見ることができる。

```
(gdb) p $st1
$1 = 0.0285946445426121034771 9
```

A.3 浮動小数点例外の処理

浮動小数点例外 SIGFPE が発生したときは、プログラムの実行を終了させることができる。想定外の無限大や NaN の発生を見つけるのに便利である。その時点での設定をコマンド info signal SIGFPE で確認することができる。

```
(gdb) info signal SIGFPE
Signal  Stop  Print  Pass to program Description
SIGFPE  Yes   Yes    Yes             Arithmetic exception
```

プログラムでシグナル・ハンドラーを設定しない限りは、デフォルトの設定を変えられ SIGFPE は

プログラムに渡されなくなり、プログラムは終了する。コマンド `handle SIGFPE stop nopass` でこれを変えることができる。

```
(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、カーネルに浮動小数点例外を発生するように命令しなければならないことがある。GSL を使っているプログラムでは、環境変数 `GSL_IEEE_MODE` を設定して関数 `gsl_ieee_env_setup()`と組み合わせて使うことで、これを行うことができる (第 39 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

信頼性の高い数値計算プログラムは、C 言語では非常な注意深く書かねばならない。そこで、コンパイル時に以下に示す GCC の警告オプションを使うと、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 -O4
```

各オプションの詳細についてはマニュアル 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、コンパイルは最後まで行われない。

-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 は一般的には必要であることが多いが、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を解析を利用して、初期化されていない変数があれば警告する。最適化されないときには警告はない。

-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が困難である。

-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 `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、メモリ上で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 で開発する際には、以下の本を読むことが重要である。

- 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

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 のアイデア（James Theiler と）とその設計。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 のアイデア（Mark Galassi と）とその設計。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、乱数生成と乱数分布、求根法、最大化と最小化、多項式の求根法、複素数、物理定収、置換、ベクトルと行列の関数、ヒストグラム、統計、IEEE 関連、改訂版 CBLAS の Lebel2 と 3、行列の分解、固有値問題、累積分布関数、検定、解説、リリース。

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、ドキュメントと初期の求根法ルーチンの作成。

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・ライブラリ。

Jorma Olavi T.ahtinen

初期の複素数の算術演算関数。

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. による乱数生成法の実装。

Szymon Jaroszewicz

組み合わせの生成ルーチン。

Nicolas Darnis

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 解法の実装を改良。

このマニュアルの校訂は Nigel Lowry がしてくれた。感謝する。

付録 C Autoconf のマクロ

autoconf を使うプログラムでは、AC_CHECK_LIB マクロを使って、configure スクリプトからこのライブラリを自動的にリンクすることができる。このライブラリ自身は CBLAS や他の数値計算ライブラリに依存するので、libgsl ファイルをリンクする前にそれらをインストールしておく必要がある。こういったことを確認するためには、'configure.in' ファイルに以下の命令を書いておく。

```
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たとすると、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、ライブラリのバージョンは不問である。一般的な、関数バージョンがあまり重要ではない場合はこれでよいが、特定のバージョンのライブラリ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、これで要求するライブラリの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 GS)
```

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。

```
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、または、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ファイル 'config.h' 内で定義される。

以下の autoconf 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、移植性のある関数で自動的に他の関数を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（訳注：ファイル中で日本語が使えるかどうかは、各環境でそれぞれ確認しなければならない)。

```
/* hypot がないシステムでは gsl_hypot を使う */

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

利用者が作った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

[Function] 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)

D.2 Level 2

[Function] 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)

D.3 Level 3

[Function] 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, ...)

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。

$$\begin{pmatrix} 0.11 & 0.12 & 0.13 \\ 0.21 & 0.22 & 0.23 \end{pmatrix} \begin{pmatrix} 1011 & 1012 \\ 1021 & 1022 \\ 1031 & 1032 \end{pmatrix} = \begin{pmatrix} 367.76 & 368.12 \\ 674.06 & 674.72 \end{pmatrix}$$

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;
}
```

このプログラムをコンパイルするためには、以下のようなコマンドを実行する。

```
$ gcc -Wall demo.c -lgslcblas
```

GSL の CBLAS は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 fol lowing article was written by Richard Stal lman, 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 mod- ify 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

GNU General Public License

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) 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 this service 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 make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. 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.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0

0) This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1) You may copy and distribute 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 and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2) You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered

independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3) You may copy and distribute the Program (or a work based on it, under Section 2) in ob ject code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in ob ject code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the ma jor components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or ob ject code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the ob ject code.

4) You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program 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.

5) You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6) Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program sub ject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7) If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), 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 distribute so as to satisfy simultaneously your obligations under this License and any other pertinentobligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.
   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.
   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.
   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8) If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9) The Free Software Foundation may publish revised and/or new versions of the 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 a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10) If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11) BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, 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.

12) IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE 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 FAIL- URE 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.

END OF TERMS AND CONDITIONS

付録 G 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。

```
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（あなたがプログラマーとして会社員である場合）や学校などによる著作権放棄の文面が必要なときは、以下のサンプルのような文面を使うとよい（名前は書き換えること）。

```
Yoyodyne, Inc., hereby disclaims all copyright interest in
the program `Gnomovision' (which makes passes at compilers)

written by James Hacker.
signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

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 Library General Public License の方がよいだろう（訳注：現在は GNU Lesser General Public License を使う方がよいだろう）。

付録 H GNU 自由文書利用許諾契約

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does.

But this License is not limited to software manuals; it can be used for any textual work, regardless of sub ject 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 sub ject (or to related matters) and contains nothing that could fall directly within that overall sub ject. (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 sub ject 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 Entitles 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: 文書にこのライセンスを適用するには

あなたが作成した文書のライセンスを GNU Free Documentation License にするには、前節のライセンスの文面をあなたの文書に付け加え、また以下の著作権表示を文書の表紙の後ろに加えるとよい。

```
Copyright (C) year your name.
Permission is granted to copy, distribute and/or modify
this document under the terms of the GNU Free
Documentation License, Version 1.2 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’’.
```

もし文書内に Invariant Section、Front-Cover Text、Back-Cover Text のいずれかがある場合は、その文面内の "with ... texts." の部分を以下の文面で置き換えればよい。

```
with the Invariant Sections being list their
titles, with the Front-Cover Texts being list, and
with the Back-Cover Texts being list.
```

もし文書内に Invariant Section はあるが Cover Text はないなど、その三つのうちの一部だけが含まれる場合は、それらを合わせて一つにし、上と同じようにして置き換えをやればよい。

もし文書内に、著作権が認められるようなプログラムのコードがある場合は、文書と同時にそのプログラム・コードを別途、GNU GPL などの元で公開し、フリーソフトウェアであることを確立しておくとよい。

か

関数

そ

ふ

へ

変数、型

ま

マクロ