BNCpack

Basic Numerical Calculation package
Version 0.7
August 22, 2011

Tomonori Kouya at Kakegawa, JAPAN
http://na-inet.jp/ (tkouya@gmail.com)

1 BNC とは？

BNCpack は IEEE 単精度・倍精度浮動小数点数 (`float`, `double`)，及び GNU MP(GMP)[1] 及び GNU MPFR[2] の多倍長浮動小数点数をサポートする基本数値計算ライブラリです。全て ANSI C(C++にあらず) で記述されています。

元々は，自分が使って来た C プログラムが貯まってきたので整理しよう，という動機で始めたものです。が，近年の PC の著しい機能向上とコスト低下のおかげで多倍長計算もお手軽な環境下で使い物になってきた，じゃあ，そいつも組み込んでしまえということで，やっつけ仕事で作ってしまった代物です。

以前，私は旧労働省管轄下の特殊法人職員で，社会人向けの有料セミナーの講師をよく勤めていました。プログラミング関係で需要が多かったのは，Internet 絡みのものと Visual Basic[3] 関係でした。それで少し Visual Basic と遊んでみましたが ··· 。使い勝手は良いものの，実行プログラムのスピードが遅く，数値計算用言語としてはあまりお勧めできないことがわかりました。そこで native C で書いたライブラリを DLL にして，それを売り物にしようと目論んだこともありましたが，程なく転職してしまいましたので，Visual Basic に義理立てする必要がなくなり，その計画は頓挫しております。もっとも，BNC は GMP を切り離して構築することも可能ですので，IEEE754 浮動小数点数演算のみサポートする DLL にするのはそれほど困難ではないと思います[4]。

今回公開するものは，GMP をベースとした多倍長計算サポートというのが一番の目玉[5] です。GMP が主として UNIX 環境をターゲットにしている関係上，本ライブラリも UNIX 環境で開発してきました。

具体的な BNCpack の利用方法については機能一覧，もしくはサンプルプログラムの章を参照して下さい。

---

[1] 2011 年 8 月現在の最新版は Version 5.0.2 です。URL は `http://gmplib.org/`

[2] 2011 年 8 月現在の最新版は Version 3.0.1 です。URL は `http://www.mpfr.org/`

[3] Microsoft の商標です。

[4] 今のところ，真面目に取り組むつもりはありません。気が向けば取りかかるかも。··· でもそーゆー用途には GSL(GNU Scientific Library, `http://sources.redhat.com/gsl/`) とかありますのでそちらをお使いになった方がいいでしょう。

[5] 今となってはそれほどでもないでしょうけど，自分としては売りのつもり。

2 BNCpack のインストール

最初に述べたように，BNCpack は GMP を利用した多倍長計算が売りのライブラリですが，IEEE754 standard(単精度，倍精度) で動作する関数も用意されています。頭文字を除いて同じ名前の関数が 3 つ並んでいたら，全く同じアルゴリズムを単精度 (f/F)，倍精度 (d/D)，多倍長精度 (mpf/MPF) で実行するものだと思って間違いないでしょう。

従来多倍長計算は GMP の mpf_t 型を利用してきましたが，Version 0.3 からは MPFR パッケージも利用することが出来るように改良されました。従って，Version 0.3 以降の BNCpack は

1. IEEE754 単精度 (float)，倍精度 (double) のみ利用
2. IEEE754(float, double) 及び多倍長計算として GMP(mpf_t) を利用
3. IEEE754(float, double) 及び多倍長計算として主に MPFR(mpfr_t) を利用

という 3 種類の環境で使用可能です。但し，MPFR パッケージを使う場合，現状では mpf_t 型及びそれを使用する関数群を，全て mpfr_t 型とそれを使用する関数群に置き換える (mpf2mpfr.h[1] を利用する) ことで凌いでいますので，基本的には mpf_t 型と mpfr_t 型は共存できません[2]。

以下，この 3 つの環境で BNCpack をインストールする方法を簡単に示します。

2.1 IEEE754 単精度・倍精度のみ利用する場合

‘Makefile’をご自分の環境に合わせて設定した後,

```
% make
```

として，‘libnc.a’を作成して下さい。後は ‘bnc.h’とこの ‘libbnc.a’を適当なディレクトリにコピーしてお使い下さい。

当然のことながら，このようにして作成された ‘libbnc.a’は以下で述べる多倍長計算の機能は一切使用できません。

後は，自分のプログラムが ‘myprog.c’であれば，適切な場所で ‘bnc.h’をインクルードしておき

```
% cc -omyprog myprog.c -lbnc -lm
```

あるいはダイレクトに

```
% cc -omyprog myprog.c /libdir/libbnc.a -lm
```

としてコンパイル・リンクして使用して下さい。

2.2 GMP の mpf_t 型のみを利用する場合

GMP のインストール方法等については http://swox.com/gmp/をご参照下さい。特段変わった環境でなければ，Version 3.0.1 以降については殆どの UNIX 環境下でするようです。私の使用している RPM 系の Linux や Plamo Linux/Slackware Linux では

[1] MPFR パッケージの　マニュアル参照。

[2] 実は共用できるようにしてありますが，表では使わないようになっています。

```
% ./configure; make; su
# make install
```

でコンパイルとインストールを行うことが出来ています。

GMP がインストールしてあれば，‘bnc-x.x.x.tar.gz’をダウンロード後，‘Makefile’のコンパイルオプション等を適切に設定した後，

```
% make
```

で，‘libbnc.a’が出来上がるはずです。あとは‘bnc.h’とこの‘libbnc.a’を適当なディレクトリにコピーしてお使い下さい。

自分で作ったプログラムが‘myprog.c’であれば，適切な場所で‘bnc.h’をインクルードしておき

```
% cc -omyprog -DUSE_GMP myprog.c -lbnc -lgmp -lm
```

あるいはダイレクトに

```
% cc -omyprog -DUSE_GMP myprog.c /libdir/libbnc.a -lgmp -lm
```

としてコンパイル・リンクして下さい。この*-DUSE_GMP* というオプションがうっとうしいときには，直接プログラム中で

```
#define USE_GMP
#include "bnc.h"
```

として下さい。‘bnc.h’をインクルードする前に定義しておく必要があります。

2.3 MPFR パッケージを利用する場合

MPFR パッケージ (http://www.mpfr.org/) は GMP をベースに，IEEE754 浮動小数点数と互換性を持つように改良された多倍長浮動小数点演算ライブラリです。GMP の mpf_t 型との違いは

- IEEE754 の 4 種の丸めモード (RN, RZ, RU, RD) のサポート[3]
- オーバーフロー，非数のサポート
- 高速な初等関数 (sin, cos, log, exp 等) のサポート

という点にあります。MPFR パッケージは数値計算用途の機能を拡充したものと言えるでしょう。そのためか，少しだけ mpf_t 型の四則演算よりも速度が落ちるようです。

mpf_t 型と同様，‘bnc-x.x.x.tar.gz’をダウンロード後，‘Makefile’のコンパイルオプション等を適切に設定した後，

```
% make
```

で，‘libbnc.a’が出来上がるはずです。あとは‘bnc.h’とこの‘libbnc.a’を適当なディレクトリにコピーしてお使い下さい。

[3] mpf_t 型の丸め処理は切り捨てで行われます。

MPFR パッケージと共に，自分で作ったプログラム ‘myprog.c’をコンパイルするのであれば，適切な場所で ‘bnc.h’をインクルードしておき

```
% cc -omyprog -DUSE_GMP -DUSE_MPFR myprog.c -lbnc -lmpfr -lgmp -lm
```

あるいはダイレクトに

```
% cc -omyprog -DUSE_GMP -DUSE_MPFR myprog.c /libdir/libbnc.a \
 -lmpfr -lgmp -lm
```

としてコンパイル・リンクして下さい。この*-DUSE_GMP -DUSE_MPFR* というオプションがうっとうしいときには，直接プログラム中で

```
#define USE_GMP
#define USE_MPFR
#include "bnc.h"
```

として下さい。‘bnc.h’をインクルードする前に定義しておく必要があります。

3 機能一覧

3.1 基本データ型定義

3.1.1 多倍長浮動小数点数を含むデータ型

構造体には，その構造体ごとの精度を保持する変数 *unsigned long* `prec`を含みます。それ以外の構成は IEEE754 浮動小数点数を利用したものと変わりません。

3.1.2 複素数型

`fcmplx` [New Type]
`dcmplx` [New Type]
`mpfcmplx` [New Type]

上から順に，IEEE754 単精度浮動小数点数による複素数型，倍精度浮動小数点数による複素数型，多倍長浮動小数点数による複素数型です。

この複素数型は次のように定義されています。

```
typedef struct{
    float re; /* Real part */
    float im; /* Imaginary part */
} fcmplx;
```

これらの複素数型を利用した関数群としては，DKA 法が存在します。線型計算ではまだ利用できません。

3.1.3 多項式型

`FPoly` [New Type]
`DPoly` [New Type]
`MPFPoly` [New Type]

上から順に，単精度浮動小数点数による多項式型，倍精度係数の多項式型，多倍長係数の多項式型です。

多項式型は次のように定義されています。

```
typedef struct{
    float *coef; /* Coefficients of polynomial */
    long deg; /* Degree of polynomial */
} fpoly;

typedef fpoly *FPoly;
```

これら多項式型を利用した関数群は，DKA 法に利用されています。

3.1.4 ベクトル型

`FVector` [New Type]
`DVector` [New Type]

`MPFVector` [New Type]
上から順に，単精度浮動小数点数によるベクトル型，倍精度ベクトル型，多倍長ベクトル型です。

`CFVector` [New Type]
`CDVector` [New Type]
`CMPFVector` [New Type]
上から順に，単精度浮動小数点数の複素ベクトル型，倍精度複素ベクトル型，多倍長複素ベクトル型です。

ベクトル型は次のように定義されています。原則としては構造体へのポインタである *[F,D,MPF,CF,CD,CMPF]Vector* だけを使います。

```
typedef struct{
    float *element; /* Elements of vector */
    long dim; /* Dimension of vector */
} fvector;

typedef fvector *FVector;
```

複素ベクトル型を使った関数群はまだ提供されていません。

3.1.5 一般行列型

`FMatrix` [New Type]
`DMatrix` [New Type]
`MPFMatrix` [New Type]
上から順に，単精度一般行列型，倍精度一般行列型，多倍長一般行列型です。

`CFMatrix` [New Type]
`CDMatrix` [New Type]
`CMPFMatrix` [New Type]
上から順に，単精度複素一般行列型，倍精度複素一般行列型，多倍長複素一般行列型です。

一般行列型は次のように定義されています。原則としては構造体へのポインタである *[F,D,MPF,CF,CD,CMPF]Matrix* だけを使います。

```
typedef struct {
    float *element; /* Elements */
    long row_dim, col_dim; /* Rows, Columns */
} fmatrix;

typedef fmatrix *FMatrix;
```

複素一般行列型を扱う関数群はまだ提供されていません。

3.1.6 スタック

`FStack` [New Type]

DStack [New Type]
MPFStack [New Type]
上から順に，単精度スタック，倍精度スタック，多倍長スタックです。

スタックは次のように定義されています。原則としては構造体へのポインタである *[F,D,MPF]Stack* だけを使います。

```
typedef struct {
    float *array; /* stack */
    long size; /* Height of stack */
    long index; /* pointer to top of stack */
} fstack;

typedef fstack *FStack;
```

スタックを利用した関数群はまだ提供されていません。

3.1.7 配列

FArray [New Type]
DArray [New Type]
MPFArray [New Type]
CFArray [New Type]
CDArray [New Type]
CMPFArray [New Type]
上から順に，単精度，倍精度，多倍長，複素単精度，複素倍精度，複素多倍長配列です。

配列は次のように定義されています。原則としては構造体へのポインタである *[F,D,MPF]Array* だけを使います。

```
typedef struct {
        float *array; /* array */
        long int size; /* Length of array */
} farray;

typedef fstack *FArray;
```

配列は，DKA 法の関数群で利用されています。

3.1.8 疎行列

DRSMatrix [New Type]
MPFRSMatrix [New Type]
上から順に，倍精度，多倍長精度の疎行列用データ型です。

疎行列は次のような構造体で定義されています。

```
typedef struct {
  unsigned long prec;
  mpf_t *element;            // Elements of matrix
```

```
  long int row_dim, col_dim; // Dimensions of Row and Column
  long int **nzero_index;    // Indeces of Non-zero elements
  long int *nzero_col_dim;   // Numbers of non-zero elements in i-th row
  long int *nzero_row_dim;   // Numbers of non-zero elements in i-th column
  long int nzero_total_num;  // Total number of non-zero elements
} mpfrsmatrix;

typedef mpfrsmatrix *MPFRSMatrix;
```

疎行列は次のように格納されます。

```
      0 1 2 3 4
A = [a 0 b c 0]0
    [0 d 0 0 0]1
    [0 e f 0 0]2
    [0 0 0 g 0]3
    [0 0 h 0 i]4

<--> element = [a b c d e f g h i]
     row_dim = 5, col_dim = 5
     nzero_index[0] = [0 2 3]
     nzero_index[1] = [4]
     nzero_index[2] = [1 2]
     nzero_index[3] = [3]
     nzero_index[4] = [2 4]
     nzero_col_dim[0] = 3
     nzero_col_dim[1] = 1
     nzero_col_dim[2] = 2
     nzero_col_dim[3] = 1
     nzero_col_dim[4] = 2
     nzero_total_num =  9
     nzero_row_dim[0] = 1
     nzero_row_dim[1] = 2
     nzero_row_dim[2] = 3
     nzero_row_dim[3] = 2
     nzero_row_dim[4] = 1
```

3.2 基本関数

`long lmax` (*long* **rop**, *long* **op**) [Function]
`long lmin` (*long* **rop**, *long* **op**) [Function]
`float fmax` (*float* **rop**, *float* **op**) [Function]
`float fmin` (*float* **rop**, *float* **op**) [Function]
`double dmax` (*double* **rop**, *double* **op**) [Function]
`double dmin` (*double* **rop**, *double* **op**) [Function]
`mpf_ptr mpf_max` (*mpf_t* **rop**, *mpf_t* **op**) [Function]
`mpf_ptr mpf_min` (*mpf_t* **rop**, *mpf_t* **op**) [Function]

rop と *op* を比較して最大値，最小値を返します。

`long lpower` (*long* ***rop***, *long* ***op***) [Function]
`float fpower` (*float* ***rop***, *long* ***op***) [Function]
`double dpower` (*double* ***rop***, *long* ***op***) [Function]
`void mpf_power` (*mpf_t* ***rop***, *mpf_t* ***op1***, *long* ***op2***) [Function]
rop^{op} を計算します。

`void set_bnc_default_prec` (*unsigned long* ***rop***) [Function]
BNCpack の *mpf_...* 関数群で使用するデフォルトの精度 (仮数部の bit 数) をセットします。この関数を呼び出した後の *mpf_...* 関数が影響を受けます。MPFR パッケージを用いる場合はここでデフォルトの丸めモード (RN) もセットします。

`void set_bnc_rounding_mode` (*mp_rnd_t* ***rmode***) [Function]
MPFR パッケージを読み込んでいる場合のみ，有効となる関数です。BNCpack の *mpf_...* 関数群で使用するデフォルトの丸めモードをセットします。使用できる丸めモードは次の 4 種類です。

- *GMP_RNDN*: Round to Nearest
- *GMP_RNDU*: Round to Infinity
- *GMP_RNDD*: Round to -Infinity
- *GMP_RNDZ*: Round to Zero

この関数を呼び出した後の *mpf_...* 関数が影響を受けます。

`unsigned long get_bnc_default_prec` (*void*) [Function]
BNCpack の *mpf_...* 関数群で使用するデフォルトの精度を返します。

`void mpf_pi` (*mpf_t* ***rop***) [Function]
`void mpf_e` (*mpf_t* ***rop***) [Function]
円周率と自然対数の底の値を返します。MPFR パッケージを使用する場合は，そこで定義されている初等関数を使用することになります。

`void mpf_floor` (*mpf_t* ***rop***, *mpf_t****op***) [Function]
床関数です。GMP Ver.3.x よりサポートされましたが，Ver.2.x 以前でも有効な関数として作成してあります。

`float mpf2float` (*mpf_t* ***rop***) [Function]
`double mpf2double` (*mpf_t* ***rop***) [Function]
mpf_t 型を単精度，倍精度に変換します。

`void mpf_sin` (*mpf_t* ***rop***, *mpf_t* ***op***) [Function]
`void mpf_cos` (*mpf_t* ***rop***, *mpf_t* ***op***) [Function]
`void mpf_exp` (*mpf_t* ***rop***, *mpf_t* ***op***) [Function]
`void mpf_ln` (*mpf_t* ***rop***, *mpf_t* ***op***) [Function]
`void mpf_log10` (*mpf_t* ***rop***, *mpf_t* ***op***) [Function]
上から順に，正弦関数，余弦関数，指数関数，対数関数，常用対数関数です。Maclaurin 展開に基づくルーチンですので，ものすごく速度が遅いです。高速な多倍長初等関数群が必要でしたら mpfr パッケージの使用をご検討下さい。MPFR パッケージを使うことで，これらの関数は全て高速なものに置き換えられます。

3.3 スタック

`FStack init_fstack` (*long* **`stack_size`**) [Function]
`DStack init_dstack` (*long* **`stack_size`**) [Function]
`MPFStack init_mpfstack` (*long* **`stack_size`**) [Function]
スタックを初期化します。多倍長スタックは `set_bnc_default_prec`関数で定義された精度で初期化されます。

`MPFStack init2_mpfstack` (*long* **`stack_size`**, *unsigned long* **`prec`**) [Function]
多倍長スタックを精度指定で初期化します。

`void free_dstack` (*DStack* **`st`**) [Function]
`void free_fstack` (*FStack* **`st`**) [Function]
`void free_mpfstack` (*MPFStack* **`st`**) [Function]
スタックを消去します。

`void push_fstack` (*FStack* **`st`**, *float* **`val`**) [Function]
`void push_dstack` (*DStack* **`st`**, *double* **`val`**) [Function]
`void push_mpfstack` (*MPFStack* **`st`**, *mpf_t* **`val`**) [Function]
値をスタックに積みます。

`float pop_fstack` (*FStack* **`st`**) [Function]
`double pop_dstack` (*DStack* **`st`**) [Function]
`void pop_mpfstack` (*mpf_t* **`rval`**, *MPFStack* **`st`**) [Function]
スタックから値を取り出します。

3.4 複素数

`FCmplx init_fcmplx` (*void*) [Function]
`DCmplx init_dcmplx` (*void*) [Function]
`MPFCmplx init_mpfcmplx` (*void*) [Function]
複素数を格納する領域を確保します。

`MPFCmplx init2_mpfcmplx` (*unsigned long int* **`prec`**) [Function]
prec bit の精度を持つ複素数を格納する領域を確保します。

`void free_fcmplx` (*FCmplx* **`op`**) [Function]
`void free_dcmplx` (*DCmplx* **`op`**) [Function]
`void free_mpfcmplx` (*MPFCmplx* **`op`**) [Function]
複素数 *op* の格納されている領域を解放します。

`float get_real_fcmplx` (*FCmplx* **`op`**) [Function]
`double get_real_dcmplx` (*DCmplx* **`op`**) [Function]
`void get_real_mpfcmplx` (*mpf_t* **`rop`**, *MPFCmplx* **`op`**) [Function]
複素数 *op* の実数部を返します。多倍長型は *rop* に実数部が格納されます。

`float get_image_fcmplx` (*FCmplx* **`op`**) [Function]
`double get_image_dcmplx` (*DCmplx* **`op`**) [Function]

`void get_image_mpfcmplx` (*mpf_t* **rop**, *MPFCmplx* **op**) [Function]
複素数 *op* の虚数部を返します。多倍長型は *rop* に虚数部が格納されます。

`void set_real_fcmplx` (*FCmplx* **rop**, *float* **val**) [Function]
`void set_real_dcmplx` (*DCmplx* **rop**, *double* **val**) [Function]
`void set_real_mpfcmplx` (*MPFCmplx* **rop**, *mpf_t* **val**) [Function]
実数 *val* を複素数 *rop* の実数部に格納します。

`void set_real_mpfcmplx_ui` (*MPFCmplx* **rop**, *unsigned long int* **val**) [Function]
整数 *val* を複素数 *rop* の実数部に格納します。

`void set_image_fcmplx` (*FCmplx* **rop**, *float* **val**) [Function]
`void set_image_dcmplx` (*DCmplx* **rop**, *double* **val**) [Function]
`void set_image_mpfcmplx` (*MPFCmplx* **rop**, *mpf_t* **val**) [Function]
実数 *val* を複素数 *rop* の虚数部に格納します。

`void set_image_mpfcmplx_ui` (*MPFCmplx* **rop**, *unsigned long int* **val**) [Function]
符号なし長整数 *val* を複素数 *rop* の虚数部に格納します。

`void subst_fcmplx` (*FCmplx* **rop**, *FCmplx* **op**) [Function]
`void subst_dcmplx` (*DCmplx* **rop**, *DCmplx* **op**) [Function]
`void subst_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op**) [Function]
複素数 *op* を複素数 *rop* に代入します。

`void set0_fcmplx` (*FCmplx* **rop**) [Function]
`void set0_dcmplx` (*DCmplx* **rop**) [Function]
`void set0_mpfcmplx` (*MPFCmplx* **rop**) [Function]
複素数 *rop* に 0 をセットします。

`void add_fcmplx` (*FCmplx* **rop**, *FCmplx* **op1**, *FCmplx* **op2**) [Function]
`void add_dcmplx` (*DCmplx* **rop**, *DCmplx* **op1**, *DCmplx* **op2**) [Function]
`void add_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op1**, *MPFCmplx* **op2**) [Function]
複素数 *op1* と複素数 *op2* の和を複素数 *rop* に格納します。

`void add_fcmplx_f` (*FCmplx* **rop**, *FCmplx* **op**, *float* **val**) [Function]
`void add_dcmplx_d` (*DCmplx* **rop**, *DCmplx* **op**, *double* **val**) [Function]
`void add_mpfcmplx_mpf` (*MPFCmplx* **rop**, *MPFCmplx* **op**, *mpf_t* **val**) [Function]
複素数 *op* に実数 *val* を加え，複素数 *rop* に格納します。

`void add2_fcmplx` (*FCmplx* **rop**, *FCmplx* **op**) [Function]
`void add2_dcmplx` (*DCmplx* **rop**, *DCmplx* **op**) [Function]
`void add2_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op**) [Function]
複素数 *rop* に複素数 *op* を加え，結果を *rop* に戻します。

`void sub_fcmplx` (*FCmplx* **rop**, *FCmplx* **op1**, *FCmplx* **op2**) [Function]
`void sub_dcmplx` (*DCmplx* **rop**, *DCmplx* **op1**, *DCmplx* **op2**) [Function]

`void sub_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op1**, *MPFCmplx* **op2**) [Function]
複素数同士の差 *op1* − *op2* を計算し，結果を *rop* に格納します。

`void conj_fcmplx` (*FCmplx* **rop**, *FCmplx* **op**) [Function]
`void conj_dcmplx` (*DCmplx* **rop**, *DCmplx* **op**) [Function]
`void conj_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op**) [Function]
複素数 *op* と共約な複素数を計算し，結果を *rop* に返します。

`void sign_fcmplx` (*FCmplx* **rop**, *FCmplx* **op**) [Function]
`void sign_dcmplx` (*DCmplx* **rop**, *DCmplx* **op**) [Function]
`void sign_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op**) [Function]
複素数 *op* 全体の符号を反転し，結果を *rop* に格納します。

`void sign2_fcmplx` (*FCmplx* **op**) [Function]
`void sign2_dcmplx` (*DCmplx* **op**) [Function]
`void sign2_mpfcmplx` (*MPFCmplx* **op**) [Function]
複素数 *op* 全体の符号を反転し，結果を *rop* に格納します。

`float abs_fcmplx` (*FCmplx* **op**) [Function]
`double abs_dcmplx` (*DCmplx* **op**) [Function]
`void abs_mpfcmplx` (*mpf_t* **ret**, *MPFCmplx* **op**) [Function]
複素数 *op* の絶対値を計算して返します。

`float mul_fcmplx` (*FCmplx* **rop**, *FCmplx* **op1**, *FCmplx* **op2**) [Function]
`double mul_dcmplx` (*DCmplx* **rop**, *DCmplx* **op1**, *DCmplx* **op2**) [Function]
`void mul_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op1**, *MPFCmplx* **op2**) [Function]
複素数同士の積 *op1* ∗ *op2* を計算し，結果を *rop* に格納します。

`float mul_fcmplx_f` (*FCmplx* **rop**, *FCmplx* **op**, *float* **val**) [Function]
`double mul_dcmplx_d` (*DCmplx* **rop**, *DCmplx* **op**, *double* **val**) [Function]
`void mul_mpfcmplx_mpf` (*MPFCmplx* **rop**, *MPFCmplx* **op**, *mpf_t* **val**) [Function]
複素数 *op* と実数 *val* との積を *rop* に格納します。

`void mul_mpfcmplx_ui` (*MPFCmplx* **rop**, *MPFCmplx* **op**, *unsigned long int* **val**) [Function]
複素数 *op* と整数 *val* との積を *rop* に格納します。

`float mul2_fcmplx` (*FCmplx* **rop**, *FCmplx* **op**) [Function]
`double mul2_dcmplx` (*DCmplx* **rop**, *DCmplx* **op**) [Function]
`void mul2_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op**) [Function]
複素数同士の積 *rop* × *op* を計算し，結果を *rop* に戻します。

`float div_fcmplx` (*FCmplx* **rop**, *FCmplx* **op1**, *FCmplx* **op2**) [Function]
`double div_dcmplx` (*DCmplx* **rop**, *DCmplx* **op1**, *DCmplx* **op2**) [Function]

`void div_mpfcmplx` (*MPFCmplx* **rop**, *MPFCmplx* **op1**, *MPFCmplx* **複素数同士の除算** *op*1/*op*2 **を計算し，結果を** *rop* **に格納します。** [Function]

`void iexp_fcmplx` (*FCmplx* **rop**, *float* **op**) [Function]
`void iexp_dcmplx` (*DCmplx* **rop**, *double* **op**) [Function]
`void iexp_mpfcmplx` (*MPFCmplx* **rop**, *mpf_t* **op**) [Function]
$\exp(op \times i)$ を計算し，複素数 *rop* に格納します。

`void print_fcmplx` (*FCmplx* **op**) [Function]
`void print_dcmplx` (*DCmplx* **op**) [Function]
`void print_mpfcmplx` (*MPFCmplx* **op**) [Function]
複素数 *op* の値を標準出力に表示します。

3.5 配列

`FArray init_farray` (*long int* **array_size**) [Function]
`DArray init_darray` (*long int* **array_size**) [Function]
`MPFArray init_mpfarray` (*long int* **array_size**) [Function]
`CFArray init_cfarray` (*long int* **array_size**) [Function]
`CDArray init_cdarray` (*long int***array_size**) [Function]
`CMPFArray init_cmpfarray` (*long int* **array_size**) [Function]
array_size 個のデータが格納できる配列を確保します。

`MPFArray init2_mpfarray` (*long int* **array_size**, *unsigned long* **prec**) [Function]
`CMPFArray init2_cmpfarray` (*long int* **array_size**, *unsigned long* **prec**) [Function]
array_size 個の，*prec* bit の精度を持つデータを格納できる配列を確保します。

`void free_farray` (*FArray* **rop**) [Function]
`void free_darray` (*DArray* **rop**) [Function]
`void free_mpfarray` (*MPFArray* **rop**) [Function]
`void free_cfarray` (*CFArray* **rop**) [Function]
`void free_cdarray` (*CDArray* **rop**) [Function]
`void free_cmpfarray` (*CMPFArray* **rop**) [Function]
配列 *rop* の記憶領域を削除します。

`float get_farray_i` (*FArray* **op**, *long int* **index**) [Function]
`double get_darray_i` (*DArray* **op**, *long int* **index**) [Function]
`mpf_ptr get_mpfarray_i` (*MPFArray* **op**, *long int* **index**) [Function]
`FCmplx get_cfarray_i` (*CFArray* **op**, *long int* **index**) [Function]
`DCmplx get_cdarray_i` (*CDArray* **op**, *long int* **index**) [Function]
`MPFCmplx get_cmpfarray_i` (*CMPFArray* **op**, *long int* **index**) [Function]
配列 *op* の *index* 番目のデータを返します。

`void set_farray_i` (*FArray* **rop**, *long int* **index**, *float* **val**) [Function]
`void set_darray_i` (*DArray* **rop**, *long int* **index**, *double* **val**) [Function]
`void set_mpfarray_i` (*MPFArray* **rop**, *long int* **index**, *mpf_t* **val**) [Function]
`void set_cfarray_i` (*CFArray* **rop**, *long int* **index**, *FCmplx* **val**) [Function]

`void set_cdarray_i` (*CDArray* **rop**, *long int* **index**, *DCmplx* **val**) [Function]
`void set_cmpfarray_i` (*CMPFArray* **rop**, *long int* **index**, *MPFCmplx* **val**) [Function]
配列 *rop* の *index* 番目にデータ *val* を格納します。

`void subst_farray` (*FArray* **rop**, *FArray* **op**) [Function]
`void subst_darray` (*DArray* **rop**, *DArray* **op**) [Function]
`void subst_mpfarray` (*MPFArray* **rop**, *MPFArray* **op**) [Function]
`void subst_cfarray` (*CFArray* **rop**, *CFArray* **op**) [Function]
`void subst_cdarray` (*CDArray* **rop**, *CDArray* **op**) [Function]
`void subst_cmpfarray` (*CMPFArray* **rop**, *CMPFArray* **op**) [Function]
配列 *op* の内容を配列 *rop* に代入します。

`void print_farray` (*FArray* **op**) [Function]
`void print_cfarray` (*CFArray* **op**) [Function]
`void print_darray` (*DArray* **op**) [Function]
`void print_cdarray` (*CDArray* **op**) [Function]
`void print_mpfarray` (*MPFArray* **op**) [Function]
`void print_cmpfarray` (*CMPFArray* **op**) [Function]
配列 *op* の内容を標準出力に表示します。

3.6 多項式

`FPoly init_fpoly` (*long int* **degree**) [Function]
`DPoly init_dpoly` (*long int* **degree**) [Function]
`MPFPoly init_mpfpoly` (*long int* **degree**) [Function]
最大次数 *degree* の多項式を初期化します。係数が格納される配列の要素数が *degree* で決定されます。

`MPFPoly init2_mpfpoly` (*long int* **degree**, *unsigned long* **prec**) [Function]
係数が *prec*bit の精度を持つ最大次数 *degree* の多項式を初期化します。

`void free_fpoly` (*FPoly* **poly**) [Function]
`void free_dpoly` (*DPoly* **poly**) [Function]
`void free_mpfpoly` (*MPFPoly* **poly**) [Function]
多項式 *poly* の記憶領域を解放します。

`float get_fpoly_i` (*FPoly* **poly**, *long* **index**) [Function]
`double get_dpoly_i` (*DPoly* **poly**, *long* **index**) [Function]
`mpf_ptr get_mpfpoly_i` (*MPFPoly* **poly**, *long* **index**) [Function]
多項式 *poly* の *index* 次係数を返します。

`long setdegree_fpoly` (*FPoly* **poly**) [Function]
`long setdegree_dpoly` (*DPoly* **poly**) [Function]
多項式 *poly* の次数を返します。

`void set_fpoly_i` (*FPoly* **poly**, *long* **index**, *float* **val**) [Function]
`void set_dpoly_i` (*DPoly* **poly**, *long* **index**, *double* **val**) [Function]

`void set_mpfpoly_i` (*MPFPoly* **`poly`**, *long* **`index`**, *mpf_t* **`val`**) [Function]
多項式 *poly* の *index* 次係数に実数 *val* を格納します。

`void set_mpfpoly_i_d` (*MPFPoly* **`poly`**, *long* **`index`**, *double* **`val`**) [Function]
多項式 *poly* の *index* 次係数に倍精度実数 *val* を格納します。

`void set_mpfpoly_i_str` (*MPFPoly* **`poly`**, *long* **`index`**, *const char ** **`val`**, *int* **`base`**) [Function]
多項式 *poly* の *index* 次係数に，文字列で表現された *base* 進数の実数 *val* を格納します。

`void print_fpoly` (*FPoly* **`poly`**) [Function]
`void print_dpoly` (*DPoly* **`poly`**) [Function]
`void print_mpfpoly` (*MPFPoly* **`poly`**) [Function]
多項式 *poly* の係数を標準出力に表示します。

`void print_fdmpfpoly` (*FPoly* **`fpoly`**, *DPoly* **`dpoly`**, *MPFPoly* **`mpfpoly`**) [Function]
単精度多項式 *fpoly*, 倍精度多項式 *dpoly*, 多倍長多項式 *mpfpoly* の係数を標準出力に表示します。

`void add_fpoly` (*FPoly* **`rpoly`**, *FPoly* **`poly1`**, *FPoly* **`poly2`**) [Function]
`void add_dpoly` (*DPoly* **`rpoly`**, *DPoly* **`poly1`**, *DPoly* **`poly2`**) [Function]
`void add_mpfpoly` (*MPFPoly* **`rpoly`**, *MPFPoly* **`poly1`**, *MPFPoly* **`poly2`**) [Function]
2 つの多項式 *poly1* と *poly2* の和を *rpoly* に格納します。

`void sub_fpoly` (*FPoly* **`rpoly`**, *FPoly* **`poly1`**, *FPoly* **`poly2`**) [Function]
`void sub_dpoly` (*DPoly* **`rpoly`**, *DPoly* **`poly1`**, *DPoly* **`poly2`**) [Function]
`void sub_mpfpoly` (*MPFPoly* **`rpoly`**, *MPFPoly* **`poly1`**, *MPFPoly* **`poly2`**) [Function]
2 つの多項式 *poly1* と *poly2* の差を *rpoly* に格納します。

`void cmul_fpoly` (*FPoly* **`rpoly`**, *float* **`val`**, *FPoly* **`poly`**) [Function]
`void cmul_dpoly` (*DPoly* **`rpoly`**, *double* **`val`**, *DPoly* **`poly`**) [Function]
`void cmul_mpfpoly` (*MPFPoly* **`rpoly`**, *mpf_t* **`val`**, *MPFPoly* **`poly`**) [Function]
多項式 *poly* と実数 *val* とのスカラー積を *rpoly* に格納します。

`void subst_fpoly` (*FPoly* **`rpoly`**, *FPoly* **`poly`**) [Function]
`void subst_dpoly` (*DPoly* **`rpoly`**, *DPoly* **`poly`**) [Function]
`void subst_mpfpoly` (*MPFPoly* **`rpoly`**, *MPFPoly* **`poly`**) [Function]
多項式 *poly* を多項式 *rpoly* に代入します。

`void set0_fpoly` (*FPoly* **`poly`**) [Function]
`void set0_dpoly` (*DPoly* **`poly`**) [Function]
`void set0_mpfpoly` (*MPFPoly* **`poly`**) [Function]
多項式 *poly* の係数を全て 0 にセットします。

`long max_abscoef_fpoly` (*FPoly* **poly**) [Function]
`long max_abscoef_dpoly` (*DPoly* **poly**) [Function]
`long max_abscoef_mpfpoly` (*MPFPoly* **poly**) [Function]
多項式 *poly* の絶対値最大係数の次数を返します。

`long num_nonzero_fpoly` (*FPoly* **poly**) [Function]
`long num_nonzero_dpoly` (*DPoly* **poly**) [Function]
`long num_nonzero_mpfpoly` (*MPFPoly* **poly**) [Function]
多項式 *poly* の非零係数の個数を返します。

`void diff_fpoly` (*FPoly* **poly**) [Function]
`void diff_dpoly` (*DPoly* **poly**) [Function]
`void diff_mpfpoly` (*MPFPoly* **poly**) [Function]
多項式 *poly* の導関数を計算し，*poly* に導関数を格納します。

`float eval_fpoly` (*FPoly* **poly**, *float* **val**) [Function]
`double eval_dpoly` (*DPoly* **poly**, *double* **val**) [Function]
`void eval_mpfpoly` (*mpf_t* **eval**, *MPFPoly* **poly**, *mpf_t* **val**) [Function]
多項式 *poly* の一変数が実数 *val* である時の値を計算して返します。多倍長多項式の場合は *eval* にその値を格納します。

`float eval_diff_fpoly` (*FPoly* **poly**, *float* **val**) [Function]
`double eval_diff_dpoly` (*DPoly* **poly**, *double* **val**) [Function]
`void eval_diff_mpfpoly` (*mpf_t* **eval**, *MPFPoly* **poly**, *mpf_t* **val**) [Function]
多項式 *poly* の一変数が実数 *val* である時，その導関数の値を計算して返します。多倍長多項式の場合は *eval* にその値を格納します。

`void ceval_fpoly` (*FCmplx* **eval**, *FPoly* **poly**, *FCmplx* **val**) [Function]
`void ceval_dpoly` (*DCmplx* **eval**, *DPoly* **poly**, *DCmplx* **val**) [Function]
`void ceval_mpfpoly` (*MPFCmplx* **eval**, *MPFPoly* **poly**, *MPFCmplx* **val**) [Function]
多項式 *poly* の一変数が複素数 *val* である時の値を計算し，*eval* にその値を格納します。

`void ceval_diff_fpoly` (*FCmplx* **eval**, *FPoly* **poly**, *FCmplx* **val**) [Function]
`void ceval_diff_dpoly` (*DCmplx* **eval**, *DPoly* **poly**, *DCmplx* **val**) [Function]
`void ceval_diff_mpfpoly` (*MPFCmplx* **eval**, *MPFPoly* **poly**, *MPFCmplx* **val**) [Function]
多項式 *poly* の一変数が複素数 *val* である時，その導関数の値を計算し，*eval* に格納します。

3.7 基本線型計算

`FVector init_fvector` (*long* **dim**) [Function]
`DVector init_dvector` (*long* **dim**) [Function]
`MPFVectorx init_mpfvector` (*long* **dim**) [Function]
ベクトルを初期化します。多倍長ベクトルは `set_bnc_default_prec`関数で定義された精度で初期化されます。

`MPFVector init2_mpfvector` (*long* **`dim`**, *unsigned long* **`prec`**) [Function]
多倍長ベクトルを精度指定で初期化します。

`void free_fvector` (*FVector* **`vec`**) [Function]
`void free_dvector` (*DVector* **`vec`**) [Function]
`void free_mpfvector` (*MPFVector* **`vec`**) [Function]
ベクトルを解放します。

`gfvi` (*FVector* **`vec`**, *long* **`index`**) [Macro]
`gdvi` (*DVector* **`vec`**, *long* **`index`**) [Macro]
`gmpfvi` (*MPFVector* **`vec`**, *long* **`index`**) [Macro]
ベクトル *vec* の *index* 番目の要素を取り出します。下の `get_...`関数を短い名前でマクロにしたものです。

`float get_fvector_i` (*FVector* **`vec`**, *long* **`index`**) [Function]
`double get_dvector_i` (*DVector* **`vec`**, *long* **`index`**) [Function]
`mpf_ptr get_mpfvector_i` (*MPFVector* **`vec`**, *long* **`index`**) [Function]
ベクトル *vec* の *index* 番目の要素を取り出します。

`sfvi` (*FVector* **`vec`**, *long* **`index`**, *float* **`val`**) [Macro]
`sdvi` (*DVector* **`vec`**, *long* **`index`**, *double* **`val`**) [Macro]
`smpfvi` (*MPFVector* **`vec`**, *long* **`index`**, *mpf_t* **`val`**) [Macro]
`smpfvid` (*MPFVector* **`vec`**, *long* **`index`**, *double* **`val`**) [Macro]
`smpfvis` (*MPFVector* **`vec`**, *long* **`index`**, *const char* ***`str`**, *int* **`base`**) [Macro]
ベクトル *vec* の *index* 番目の要素に値 *val* を代入します。下の `set_...`関数を短い名前でマクロにしたものです。

`void set_fvector_i` (*FVector* **`vec`**, *long* **`index`**, *float* **`val`**) [Function]
`void set_dvector_i` (*DVector* **`vec`**, *long* **`index`**, *double* **`val`**) [Function]
`void set_mpfvector_i` (*MPFVector* **`vec`**, *long* **`index`**, *mpf_t* **`val`**) [Function]
`void set_mpfvector_i_d` (*MPFVector* **`vec`**, *long* **`index`**, *double* **`val`**) [Function]
`void set_mpfvector_i_str` (*MPFVector* **`vec`**, *long* **`index`**, *const char* ***`str`**, *int* **`base`**) [Function]
ベクトル *vec* の *index* 番目の要素に値 *val* を代入します。

`unsigned long prec_mpfvector` (*MPFVector* **`vec`**) [Function]
多倍長ベクトルの精度を返します。

`unsigned long minprec_mpfvector` (*MPFVector* **`vec`**) [Function]
`unsigned long maxprec_mpfvector` (*MPFVector* **`vec`**) [Function]
多倍長ベクトルの要素の中で，最小の精度と最大の精度をそれぞれ返します。

`void print_fvector` (*FVector* **`vec`**) [Function]
`void print_dvector` (*DVector* **`vec`**) [Function]
`void print_mpfvector` (*MPFVector* **`vec`**) [Function]
ベクトルを標準出力に書き出します。

`FMatrix init_fmatrix` (*long* **`row_dim`**, *long* **`col_dim`**) [Function]
`DMatrix init_dmatrix` (*long* **`row_dim`**, *long* **`col_dim`**) [Function]
`MPFMatrix init_mpfmatrix` (*long* **`row_dim`**, *long* **`col_dim`**) [Function]
行列を初期化します。多倍長行列は，`set_bnc_default_prec`関数で設定された精度で初期化されます。

`MPFMatrix init2_mpfmatrix` (*long* **`row_dim`**, *long* **`col_dim`**, *unsigned long* **`prec`**) [Function]
多倍長行列を精度指定で初期化します。

`void free_fmatrix` (*FMatrix* **`mat`**) [Function]
`void free_dmatrix` (*DMatrix* **`mat`**) [Function]
`void free_mpfmatrix` (*MPFMatrix* **`mat`**) [Function]
行列を解放します。

`gfmij` (*FMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**) [Macro]
`gdmij` (*DMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**) [Macro]
`gmpfmij` (*MPFMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**) [Macro]
行列 *mat* の *row_index*, *col_index* 番目の要素を取り出します。下の `get_...`関数を短い名前でマクロにしたものです。

`float get_fmatrix_ij` (*FMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**) [Function]
`double get_dmatrix_ij` (*DMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**) [Function]
`mpf_ptr get_mpfmatrix_ij` (*MPFMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**) [Function]
行列 *mat* の *row_index*, *col_index* 番目の要素を取り出します。

`sfmij` (*FMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *float* **`val`**) [Macro]
`sdmij` (*DMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *double* **`val`**) [Macro]
`smpfmij` (*MPFMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *mpf_t* **`val`**) [Macro]
`smpfmijd` (*MPFMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *double* **`val`**) [Macro]
`smpfmijs` (*MPFMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *const char* ***`str`**, *int* **`base`**) [Macro]
行列 *mat* の *row_index*, *col_index* 番目の要素に値 *val* を代入します。下の `set_...`関数を短い名前でマクロにしたものです。

`void set_fmatrix_ij` (*FMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *float* **`val`**) [Function]
`void set_dmatrix_ij` (*DMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *double* **`val`**) [Function]
`void set_mpfmatrix_ij` (*MPFMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *mpf_t* **`val`**) [Function]
`void set_mpfmatrix_ij_d` (*MPFMatrix* **`mat`**, *long* **`row_index`**, *long* **`col_index`**, *double* **`val`**) [Function]

`void set_mpfmatrix_ij_str` (*MPFMatrix* `mat`, *long* `row_index`, *long* `col_index`, *const char* `*str`, *int* `base`) [Function]
行列 *mat* の *row_index*, *col_index* 番目の要素に値 *val* を代入します。

`unsigned long prec_mpfmatrix` (*MPFMatrix* `mat`) [Function]
多倍長行列の精度を返します。

`unsigned long minprec_mpfmatrix` (*MPFMatrix* `mat`) [Function]
`unsigned long maxprec_mpfmatrix` (*MPFMatrix* `mat`) [Function]
多倍長行列成分の最小精度と最大精度をそれぞれ返します。

`void print_fmatrix` (*FMatrix* `mat`) [Function]
`void print_dmatrix` (*DMatrix* `mat`) [Function]
`void print_mpfmatrix` (*MPFMatrix* `mat`) [Function]
行列を標準出力に表示します。

`void add_fvector` (*FVector* `rvec`, *FVector* `vec1`, *FVector* `vec2`) [Function]
`void add_dvector` (*DVector* `rvec`, *DVector* `vec1`, *DVector* `vec2`) [Function]
`void add_mpfvector` (*MPFVector* `rvec`, *MPFVector* `vec1`, *MPFVector* `vec2`) [Function]
ベクトルの和を計算します。

`void sub_fvector` (*FVector* `rvec`, *FVector* `vec1`, *FVector* `vec2`) [Function]
`void sub_dvector` (*DVector* `rvec`, *DVector* `vec1`, *DVector* `vec2`) [Function]
`void sub_mpfvector` (*MPFVector* `rvec`, *MPFVector* `vec1`, *MPFVector* `vec2`) [Function]
ベクトルの差を計算します。

`void add2_fvector` (*FVector* `rvec`, *FVector* `vec`) [Function]
`void add2_dvector` (*DVector* `rvec`, *DVector* `vec`) [Function]
`void add2_mpfvector` (*MPFVector* `rvec`, *MPFVector* `vec`) [Function]
rvec + *vec* を計算し，結果を *rvec* に代入します。

`void sub2_fvector` (*FVector* `rvec`, *FVector* `vec`) [Function]
`void sub2_dvector` (*DVector* `rvec`, *DVector* `vec`) [Function]
`void sub2_mpfvector` (*MPFVector* `rvec`, *MPFVector* `vec`) [Function]
rvec - *vec* を計算し，結果を *rvec* に代入します。

`void cmul_fvector` (*FVector* `rvec`, *float* `opt`, *FVector* `vec`) [Function]
`void cmul_dvector` (*DVector* `rvec`, *double* `opt`, *DVector* `vec`) [Function]
`void cmul_mpfvector` (*MPFVector* `rvec`, *mpf_t* `opt`, *MPFVector* `vec`) [Function]
vec のスカラー *opt* 倍を計算します。

`void cmul2_fvector` (*FVector* `rvec`, *float* `opt`) [Function]
`void cmul2_dvector` (*DVector* `rvec`, *double* `opt`) [Function]
`void cmul2_mpfvector` (*MPFVector* `rvec`, *mpf_t* `opt`) [Function]
rvec のスカラー *opt* 倍を計算し，結果を *rvec* に代入します。

`float ip_fvector` (*FVector* `vec1`, *FVector* `vec2`) [Function]
`double ip_dvector` (*DVector* `vec1`, *DVector* `vec2`) [Function]
`void ip_mpfvector` (*mpf_t* `ropt`, *MPFVector*`vec1` , *MPFVector* `vec2`) [Function]
内積を計算します。

`float norm1_fvector` (*FVector* `vec`) [Function]
`float norm2_fvector` (*FVector* `vec`) [Function]
`float normi_fvector` (*FVector* `vec`) [Function]
`double norm1_dvector` (*DVector* `vec`) [Function]
`double norm2_dvector` (*DVector* `vec`) [Function]
`double normi_dvector` (*DVector* `vec`) [Function]
`void norm1_mpfvector` (*mpf_t* `ropt`, *MPFVector* `vec`) [Function]
`void norm2_mpfvector` (*mpf_t* `ropt`, *MPFVector* `vec`) [Function]
`void normi_mpfvector` (*mpf_t* `ropt`, *MPFVector* `vec`) [Function]
vec のノルムをそれぞれ計算します。

`void subst_fvector` (*FVector* `rvec`, *FVector* `vec`) [Function]
`void subst_dvector` (*DVector* `rvec`, *DVector* `vec`) [Function]
`void subst_mpfvector` (*MPFVector* `rvec`, *MPFVector* `vec`) [Function]
ベクトル *vec* をベクトル *rvec* に代入します。

`void add_fmatrix` (*FMatrix* `rmat`, *FMatrix* `mat1`, *FMatrix* `mat2`) [Function]
`void add_dmatrix` (*DMatrix* `rmat`, *DMatrix* `mat1`, *DMatrix* `mat2`) [Function]
`void add_mpfmatrix` (*MPFMatrix* `rmat`, *MPFMatrix* `mat1`, *MPFMatrix* `mat2`) [Function]
行列の和を計算します。

`void sub_fmatrix` (*FMatrix* `rmat`, *FMatrix* `mat1`, *FMatrix* `mat2`) [Function]
`void sub_dmatrix(` *DMatrix* `rmat`, *DMatrix* `mat1`, *DMatrix* `mat2`) [Function]
`void sub_mpfmatrix` (*MPFMatrix* `rmat`, *MPFMatrix* `mat1`, *MPFMatrix* `mat2`) [Function]
mat1 - *mat2* を計算し，*rmat* に代入します。

`void mul_fmatrix` (*FMatrix* `rmat`, *FMatrix* `mat1`, *FMatrix* `mat2`) [Function]
`void mul_dmatrix` (*DMatrix* `rmat`, *DMatrix* `mat1`, *DMatrix* `mat2`) [Function]
`void mul_mpfmatrix` (*MPFMatrix* `rmat`, *MPFMatrix* `mat1`, *MPFMtrix* `mat2` [Function]
mat1 * *mat2* を計算し，*rmat* に代入します。

`void subst_fmatrix` (*FMatrix* `rmat`, *FMatrix* `mat`) [Function]
`void subst_dmatrix` (*DMatrix* `rmat`, *DMatrix* `mat`) [Function]
`void subst_mpfmatrix` (*MPFMatrix* `rmat`, *MPFMatrix* `mat`) [Function]
行列 *mat* を行列 *rmat* に代入します。

`void set0_fmatrix` (*FMatrix* `rmat`) [Function]
`void set0_dmatrix` (*DMatrix* `rmat`) [Function]
`void set0_mpfmatrix` (*MPFMatrix* `rmat`) [Function]
行列 *rmat* に零行列を代入します。

`void setI_fmatrix` (*FMatrix* `rmat`) [Function]
`void setI_dmatrix` (*DMatrix* `rmat`) [Function]
`void setI_mpfmatrix` (*MPFMatrix* `rmat`) [Function]
行列 *rmat* を単位行列にします。

`void mul_fmatrix_fvec` (*FVector* `rvec`, *FMatrix* `mat`, *FVector* `vec`) [Function]
`void mul_dmatrix_dvec` (*DVector* `rvec`, *DMatrix* `mat`, *DVector* `vec`) [Function]
`void mul_mpfmatrix_mpfvec` (*MPFVector* `rvec`, *MPFMatrix* `mat`, *MPFVector* `vec`) [Function]
行列 *mat* とベクトル *vec* との積を計算し，結果をベクトル *rvec* に代入します。

`void transpose_fmatrix` (*FMatrix* `rmat`, *FMatrix* `mat`) [Function]
`void transpose_dmatrix` (*DMatrix* `rmat`, *DMatrix* `mat`) [Function]
`void transpose_mpfmatrix` (*MPFMatrix* `rmat`, *MPFMatrix* `mat`) **行列** *mat* **の転置を** *rmat* **に代入します。** [Function]

`void inv_fmatrix` (*FMatrix* `mat`) [Function]
`void inv_dmatrix` (*DMatrix* `mat`) [Function]
`void inv_mpfmatrix` (*MPFMatrix* `mat`) [Function]
行列 *mat* の逆行列を計算します。*mat* は正方行列でなければなりません。

3.8 LU 分解

`int FLUdecomp` (*FMatrix* `mat`) [Function]
`int DLUdecomp` (*DMatrix* `mat`) [Function]
`int MPFLUdecomp` (*MPFMatrix* `mat`) [Function]
行列 *mat* の LU 分解を行います。

`int SolveFLS` (*FVector* `rvec`, *FMatrix* `mat`, *FVector* `vec`) [Function]
`int SolveDLS` (*DVector* `rvec`, *DMatrix* `mat`, *DVector* `vec`) [Function]
`int SolveMPFLS` (*MPFVector* `rvec`, *MPFMatrix* `mat`, *MPFVector* `vec`) [Function]
LU 分解された行列 *mat* と定数ベクトル *vec* を使い，後退代入を行って解ベクトルを *rvec* に代入していきます。

`int FLUdecompP` (*FMatrix* `mat`, *long* `iarray`[]) [Function]
`int DLUdecompP` (*DMatrix* `mat`, *long* `iarray`[]) [Function]
`int MPFLUdecompP` (*MPFMatrix* `mat`, *long* `iarray`[]) [Function]
行列 *mat* の部分ピボット選択付き LU 分解を行います。ピボット選択後の行の状態は配列 *iarray* に保存されます。

`int SolveFLSP` (*FVector* `rvec`, *FMatrix* `mat`, *FVector* `vec`, *long* `iarray`[]) [Function]
`int SolveDLSP` (*DVector* `rvec`, *DMatrix* `mat`, *DVector* `vec`, *long* `iarray`[]) [Function]
`int SolveMPFLSP` (*MPFVector* `rvec`, *MPFMatrix* `mat`, *MPFVector* `vec`, *long* `iarray`[]) [Function]
部分ピボット選択付きで LU 分解された行列 *mat* と定数ベクトル *vec* を使い，後退代入を行って解ベクトルを *rvec* に代入していきます。

`int FLUdecompC` (*FMatrix* `mat`, *long* `row_iarray`[], *long* `col_iarray`[]) [Function]
`int DLUdecompC` (*DMatrix* `mat`, *long* `row_iarray`[], *long* `col_iarray`[]) [Function]
`int MPFLUdecompC` (*MPFMatrix* `mat`, *long* `row_iarray`[], *long* `col_iarray`[]) [Function]
行列 *mat* の完全ピボット選択付き LU 分解を行います。ピボット選択後の行の状態は配列 *row_iarray* に，列の状態は配列 *col_iarray* に保存されます。

`int SolveFLSC` (*FVector* `rvec`, *FMatrix* `mat`, *FVector* `vec`, *long* `row_iarray`[], *long* `col_iarray`[]) [Function]
`int SolveDLSC` (*DVector* `rvec`, *DMatrix* `mat`, *DVector* `vec`, *long* `row_iarray`[], *long* `col_iarray`[]) [Function]
`int SolveMPFLSC` (*MPFVector* `rvec`, *MPFMatrix* `mat`, *MPFVector* `vec`, *long* `row_iarray`[], *long* `col_iarray`[]) [Function]
完全ピボット選択付きで LU 分解された行列 *mat* と定数ベクトル *vec* を使い，後退代入を行って解ベクトルを *rvec* に代入していきます。

3.9 Conjugate-Gradient 法

`long FCG` (*FVector* `rvec`, *FMatrix* `mat`, *FVector* `vec`, *float* `reps`, *float* `aeps`, *long* `maxtimes`) [Function]
`long DCG` (*DVector* `rvec`, *DMatrix* `mat`, *DVector* `vec`, *double* `reps`, *double* `aeps`, *long* `maxtimes`) [Function]
`long MPFCG` (*MPFVector* `rvec`, *MPFMatrix* `mat`, *MPFVector* `vec`, *mpf_t* `reps`, *mpf_t* `aeps`, *long* `maxtimes`) [Function]
Conjugate-Gradient 法を実行します。係数行列を *mat* に，定数ベクトルを *vec* に入れておき，残差のノルムを使った停止条件を *reps* と *aeps* に指定します。最大反復回数は *maxtimes* で指定します。

3.10 非線型方程式

`long fnewton` (*FVector* `ans`, *FVector* `x_init`, *void* (`* func`)(*FVector* `rop`, *FVector* `op`), *void* (`* jfunc`)(*FMatrix* `rop`, *FVector* `op`), *long* `maxtimes`, *float* `abs_eps`, *float* `rel_eps`) [Function]
`long dnewton` (*DVector* `ans`, *DVector* `x_init`, *void* (`* func`)(*DVector* `rop`, *DVector* `op`), *void* (`* jfunc`)(*DMatrix* `rop`, *DVector* `op`), *long* `maxtimes`, *double* `abs_eps`, *double* `rel_eps`) [Function]
`long mpf_newton` (*MPFVector* `ans`, *MPFVector* `x_init`, *void* (`* func`)(*MPFVector* `rop`, *MPFVector* `op`), *void* (`* jfunc`)(*MPFMatrix* `rop`, *MPFVector* `op`), *long* `maxtimes`, *mpf_t* `abs_eps`, *mpf_t* `rel_eps`) [Function]
多次元の Newton 法を実行します。

`long fnewton_1` (*float* `*ans`, *float* `x_init`, *float* (`* func`)(*float* `op`), *float* (`* dfunc`)(*float* `op`), *long* `maxtimes`, *float* `abs_eps`, *float* `rel_eps`) [Function]
`long dnewton_1` (*double* `*ans`, *double* `x_init`, *double* (`* func`)(`double op`), *double* (`* dfunc`)(*double* `op`), *long* `maxtimes`, *double* `abs_eps`, *double* `rel_eps`) [Function]

`long mpf_newton_1` (*mpf_t* ***ans**, *mpf_t* **x_init**, *void* (* **func**)(*mpf_t* **rop**, *mpf_t* **op**), *void* (* **dfunc**)(*mpf_t* **rop**, *mpf_t* **op**), *long* **maxtimes**, *mpf_t* **abs_eps**, *mpf_t* **rel_eps**) [Function]

1 次元の Newton 法を実行します。

`long fsnewton` (*FVector* **ans**, *FVector* **x_init**, *void* (* **func**)(*FVector* **rop**, *FVector* **op**), *void* (* **jfunc**)(*FMatrix* **rop**, *FVector* **op**), *long* **maxtimes**, *float* **abs_eps**, *float* **rel_eps**) [Function]

`long dsnewton` (*DVector* **ans**, *DVector* **x_init**, *void* (* **func**)(*DVector* **rop**, *DVector* **op**), *void* (* **jfunc**)(*DMatrix* **rop**, *DVector* **op**), *long* **maxtimes**, *double* **abs_eps**, *double* **rel_eps**) [Function]

`long mpf_snewton(` *MPFVector* **ans**, *MPFVector* **x_init**, *void* (* **func**)(*MPFVector* **rop**, *MPFVector* **op**), *void* (* **jfunc**)(*MPFMatrix* **rop**, *MPFVector* **op**), *long* **maxtimes**, *mpf_t* **abs_eps**, *mpf_t* **rel_eps**) [Function]

多次元の簡易 Newton 法を実行します。

`long fsnewton_1` (*float* ***ans**, *float* **x_init**, *float* (* **func**)(*float* **op**), *float* (* **dfunc**)(*float* **op**), *long* **maxtimes**, *float* **abs_eps**, *float* **rel_eps**) [Function]

`long dsnewton_1` (*double* * , *double* , *double* (* **func**)(*double* **op**), *double* (* **dfunc**)(*double* **op**), *long* , *double* , *double*) [Function]

`long mpf_snewton_1` (*mpf_t* **ans**, *mpf_t* **x_init**, *void* (* **func**)(*mpf_t* **rop**, *mpf_t* **op**), *void* (* **dfunc**)(*mpf_t* **rop**, *mpf_t* **op**), *long* **maxtimes**, *mpf_t* **abs_eps**, *mpf_t* **rel_eps**) [Function]

1 次元の簡易 Newton 法を実行します。

3.11 DKA 法

`float fdka_center(FPoly` *poly*`)` [Function]

`double ddka_center(DPoly` *poly*`)` [Function]

`void mpf_dka_center(mpf_t` *center*, *MPFPoly* *poly*`)` [Function]

実係数多項式 *poly* から，Aberth の初期値決定のための円の中心を求めます。多倍長型の場合は *center* に中心の座標が格納されます。実係数なので中心は必ず実軸上にのります。

`float fdka_radius(FPoly` *poly*`)` [Function]

`double ddka_radius(DPoly` *poly*`)` [Function]

`void mpf_dka_radius(mpf_t` *rad*, *MPFPoly* *poly*`)` [Function]

実係数多項式 *poly* から，Aberth の初期値を計算するための円の半径を求めます。多倍長型の場合は *rad* に半径が格納されます。

`void fdka_init(CFArray` *init_array*, *FPoly* *poly*`)` [Function]

`void ddka_init(CDArray` *init_array*, *DPoly* *poly*`)` [Function]

`void mpf_dka_init(CMPFArray` *init_array*, *MPFPoly* *poly*`)` [Function]

実係数多項式 *poly* から，Aberth の初期値を求め，配列 *init_array* に格納します。

`long fdka(CFArray answer_array, CFArray init_array, FPoly poly, long int maxtimes, float abs_eps, float rel_eps)` [Function]
`long ddka(CDArray answer_array, CDArray init_array, DPoly poly, long int maxtimes, double abs_eps, double rel_eps)` [Function]
`long mpf_dka(CMPFArray answer_array, CMPFArray init_array, MPFPoly poly, long maxtimes, mpf_t abs_eps, mpf_t rel_eps)` [Function]
$poly = 0$ という実係数代数方程式の近似解を DKA 法で計算します。初期値 *init_array* から出発し，最大反復回数 *maxtimes* 以下で全ての近似解の変化が停止条件式 *abs_eps*+*rel_epsanswer_array*[*i*] 以下になれば，配列 *answer_array* に近似解を格納します。返り値は反復回数です。

3.12 疎行列

`DRSMatrix init_drsmatrix(long row_dim, long *nzero_col_dim, long nzero_total_num)` [Function]
`MPFRSMatrix init_mpfrsmatrix(long row_dim, long *nzero_col_dim, long nzero_total_num)` [Function]
疎行列を初期化します。

`MPFRSMatrix init2_mpfrsmatrix(long row_dim, long *nzero_col_dim, long nzero_total_num, unsigned long prec)` [Function]
精度指定付きで多倍長疎行列を初期化します。

`void free_drsmatrix(DRSMatrix mat)` [Function]
`void free_mpfrsmatrix(MPFRSMatrix mat)` [Function]
疎行列 *mat* を消去します。

`void set_nzero_row_dim(DRSMatrix mat)` [Function]
`void set_nzero_row_dim_mpf(MPFRSMatrix mat)` [Function]
行方向に疎行列 *mat* の非零成分を探索してセットします。

`void print_drsmatrix(DRSMatrix mat)` [Function]
`void print_mpfrsmatrix(MPFRSMatrix mat)` [Function]
疎行列 *mat* を標準出力に表示します。

`DRSMatrix init_set_drsmatrix_dmatrix(DMatrix org_mat)` [Function]
`MPFRSMatrix init_set_mpfrsmatrix_mpfmatrix(MPFMatrix org_mat)` [Function]
密行列 *org_mat* を疎行列形式に変換したものを初期化・格納して返します。

`int get_vars_drsmatrix_fname(long *ptr_row_dim, long **ptr_nzero_col_dim, long *ptr_nzero_total_num, const char *fname)` [Function]
`int get_vars_mpfrsmatrix_fname(long *ptr_row_dim, long **ptr_nzero_col_dim, long *ptr_nzero_total_num, const char *fname)` [Function]
ファイル名 *fname* に格納された疎行列の形式を調べて返します。疎行列初期化に必要な変数が自動的に設定できます。

`int fread_urilinkdat_fname(DRSMatrix ret, const char *fname)` [Function]
`int fread_urilinkdat_fname_mpf(MPFRSMatrix ret, const char *fname)` [Function]
ファイル名 *fname* に格納されたリンク情報に基づいて疎行列 (グラフ表現) を読み取ります。

`int mul_drsmatrix_dvec(DVector ret, DRSMatrix mat, DVector vec)` [Function]
`int mul_mpfrsmatrix_mpfvec(MPFVector ret, MPFRSMatrix mat, MPFVector vec)` [Function]
疎行列 *mat* とベクトル *vec* との積を計算し，*ret* に返します。

`int mul_drsmatrixt_dvec(DVector ret, DRSMatrix mat, DVector vec)` [Function]
`int mul_mpfrsmatrixt_mpfvec(MPFVector ret, MPFRSMatrix mat, MPFVector vec)` [Function]

`double dpower_rsmatrix(DVector evec, DRSMatrix mat, double reps, double aeps, long max_times)` [Function]
`void mpfpower_rsmatrix(mpf_t max_eig, MPFVector evec, MPFRSMatrix mat, mpf_t reps, mpf_t aeps, long max_times)` [Function]
疎行列 *mat* の絶対値最大固有値と固有ベクトルをべき乗法を用いて求めます。

`int smul_dvector(DVector ret, double scalar, DVector vec)` [Function]
`int smul_mpfvector(MPFVector ret, mpf_t scalar, MPFVector vec)` [Function]
ベクトルのスカラー倍を計算します。

`long absmax_index_dvector(double *ret, DVector vec)` [Function]
`long absmax_index_mpfvector(mpf_t ret, MPFVector vec)` [Function]
ベクトル *vec* の絶対値最大成分の番号と値を返します。

4 BNCpackを使ったサンプルプログラム集

テストプログラムの内容は以下の通りです。

‘test_efunc.c’
: 基本関数のテストプログラム

‘test_complex.c’
: 複素数のテストプログラム

‘test_poly.c’
: 多項式のテストプログラム

‘test_linear.c’
: 基本線型計算のテストプログラム

‘test_lu.c’
: LU 分解による連立一次方程式の求解

‘test_cg.c’
: CG 法による連立一次方程式の求解

‘test_newton.c’
: Newton 法による非線型方程式の求解

‘test_dka.c’
: DKA 法による代数方程式の求解

コンパイル方法は，前述の通り

1. IEEE754 のみ利用可能の場合は

   ```
   %cc test_complex.c -lbnc
   ```

   とする。-lbncの代わりに直接/libdir/libbnc.aとライブラリファイルを指定してもよい。
2. IEEE754 と GMP の mpf_t 型を利用する場合は

   ```
   %cc -DUSE_GMP test_complex.c -lbnc -lgmp
   ```

   とする。/libdir/libbnc.a /libdir/libgmp.aと直接ライブラリファイルを指定してもよい。
3. IEEE754 と GMP+MPFR パッケージを利用する場合は

   ```
   %cc -DUSE_GMP -DUSE_MPFR test_complex.c -lbnc -lmpfr -lgmp
   ```

   とする。/libdir/libbnc.a /libdir/libmpfr.a /libdir/libgmp.aと直接ライブラリファイルを指定してもよい。MPFR パッケージを利用する場合は，‘mpfr.h’と‘mpf2mpfr.h’の両方を使用するので，共に読み込むことの出来るディレクトリ位置に置いておくこと。

とします。

以下の節で，BNCpack の関数群の使い方を紹介します。

4.1 基本関数

IEEE754 の初等関数は出揃っているため，C 標準のもの (‘math.h’定義のもの) を利用します。従って，ここでは多倍長計算用のサンプルのみ示すことにします。

```
#include <stdio.h>
#include <math.h>

#include "bnc.h"
```

必要なヘッダファイルを読み込みます。bnc.h は多倍長計算を要求する定義 (USE_GMP及び USE_MPFR) が存在すれば，適宜必要となるヘッダファイル (‘gmp.h’, ‘mpfr.h’) をその中で読み込んでいます。

```
main()
{
#ifndef USE_GMP
        printf("This program are not available without GMP.\n");
#else
```

このように，多倍長計算のみで実行したい部分は，USE_GMPもしくは USE_MPFRの定義が在るかどうかをチェックする#ifdefもしくは#ifndef〜#endifで括って下さい。

```
        long int i, max_times;
        double x_min, x_max, x, h;
        mpf_t mp_x_min, mp_x_max, mp_x, mp_h, \
        mp_sin, mp_cos, mp_exp, mp_pi, mp_e, mp_ln2, mp_ln, mp_log10;
```

多倍長計算を行うための変数は全て mpf_t 型として宣言して下さい。MPFR パッケージ利用の場合は，これがマクロで自動的に全て mpfr_t 型に置き換えられます[1]。

```
        set_bnc_default_prec(128);
```

デフォルトの精度桁 (多倍長浮動小数点数の仮数部桁 (2 進)) を指定しています。この場合は 128bit(10 進 38.5 桁) となります。

```
        max_times = 128;
        x_min = -30.0;
        x_max = 30.0;
        mpf_init_set_si(mp_x_min, -30);
        mpf_init_set_si(mp_x_max, 30);

        h = (x_max - x_min) / max_times;
        mpf_init(mp_h);
        mpf_init(mp_sin);
        mpf_init(mp_cos);
        mpf_init(mp_exp);
        mpf_init(mp_pi);
        mpf_init(mp_e);
        mpf_init(mp_ln2);
```

[1] 詳細は MPFR パッケージに同封されている ‘mpf2mpfr.h’を参照のこと。

```
        mpf_init(mp_ln);
        mpf_init(mp_log10);
        mpf_sub(mp_h, mp_x_max, mp_x_min);
        mpf_div_ui(mp_h, mp_h, (unsigned long)max_times);

        mpf_init_set(mp_x, mp_x_min);
        x = x_min;
```

多倍長計算を使いこなすには当然，GMP 及び MPFR で定義されている関数群をそのまま利用する必要が出てきます。詳細については各マニュアルを参照して下さい。

```
        printf("        x             sin(x)                \
mpf_sin(x)\n");
        for(i = 0; i < max_times; i++)
        {
                printf("%25.17e %25.17e ", x, sin(x));
                mpf_sin(mp_sin, mp_x);
//              mpf_out_str(stdout, 10, 0, mp_sin);
                printf("%25.17e %25.17e", mpf2double(mp_sin), cos(x));
                mpf_cos(mp_cos, mp_x);
//              mpf_out_str(stdout, 10, 0, mp_cos);
                printf("%25.17e %25.17e", mpf2double(mp_cos), exp(x));
                mpf_exp(mp_exp, mp_x);
//              mpf_out_str(stdout, 10, 0, mp_exp);
                printf("%25.17e %25.17e", mpf2double(mp_exp), log(x));
                mpf_ln(mp_ln, mp_x);
//              mpf_out_str(stdout, 10, 0, mp_ln);
                printf("%25.17e %25.17e", mpf2double(mp_ln), \
log(x)/log(10.0));
                mpf_log10(mp_log10, mp_x);
//              mpf_out_str(stdout, 10, 0, mp_log10);
                printf("%25.17e\n", mpf2double(mp_log10));

                x += h;
                mpf_add(mp_x, mp_x, mp_h);
        }
        mpf_set_ui(mp_x, 100UL);
        printf("sin(%25.17e): ", mpf2double(mp_x));
        mpf_sin(mp_sin, mp_x);
        mpf_out_str(stdout, 10, 0, mp_sin);
        printf(" %25.17e", sin(mpf2double(mp_x)));
        printf("\n");
        printf("cos(%25.17e): ", mpf2double(mp_x));
        mpf_cos(mp_cos, mp_x);
        mpf_out_str(stdout, 10, 0, mp_cos);
        printf(" %25.17e", cos(mpf2double(mp_x)));
        printf("\n");
        printf("exp(%25.17e): ", mpf2double(mp_x));
```

```
mpf_exp(mp_exp, mp_x);
mpf_out_str(stdout, 10, 0, mp_exp);
printf(" %25.17e", exp(mpf2double(mp_x)));
printf("\n");
printf("ln (%25.17e): ", mpf2double(mp_x));
mpf_ln(mp_ln, mp_x);
mpf_out_str(stdout, 10, 0, mp_ln);
printf(" %25.17e", log(mpf2double(mp_x)));
printf("\n");
printf("log10(%25.17e): ", mpf2double(mp_x));
mpf_log10(mp_log10, mp_x);
mpf_out_str(stdout, 10, 0, mp_log10);
printf(" %25.17e", log(mpf2double(mp_x))/log(10.0));
printf("\n");

printf("PI: ");
mpf_pi(mp_pi);
mpf_out_str(stdout, 10, 0, mp_pi);
printf(" -> %25.17e", mpf2double(mp_pi));
mpf_floor(mp_pi, mp_pi);
printf(" floor(PI), floor(PI*10000):");
mpf_out_str(stdout, 10, 0, mp_pi);
mpf_pi(mp_pi); mpf_mul_ui(mp_pi, mp_pi, 10000UL);
 mpf_floor(mp_pi, mp_pi);
printf(", "); mpf_out_str(stdout, 10, 0, mp_pi);
fflush(stdout);
printf("\n");
printf("E : ");
mpf_e(mp_e);
mpf_out_str(stdout, 10, 0, mp_e);
printf(" -> %25.17e", mpf2double(mp_e));
mpf_floor(mp_e, mp_e);
printf(" floor(E):");
mpf_out_str(stdout, 10, 0, mp_e);
printf("\n");
printf("log 2 : ");
mpf_ln_2(mp_ln2);
mpf_out_str(stdout, 10, 0, mp_ln2);
printf(" -> %25.17e", mpf2double(mp_ln2));
mpf_floor(mp_ln2, mp_ln2);
printf(" floor(ln2):");
mpf_out_str(stdout, 10, 0, mp_ln2);
printf("\n");
```

ここで使われている GMP には定義されていない関数 `mpf_sin`, `mpf_cos`, `mpf_exp`, `mpf_ln`, `mpf_log10`等の関数は，前述の通り，MPFR パッケージが読み込まれると全てそこで定義されているものに置き換えられます。BNCpack で GMP 用に作成されたこれらの関数は非常に低速ですので，高速な初等関数が必要な時は MPFR パッケージとの併用をお勧めします。

```
        mpf_clear(mp_h);
        mpf_clear(mp_sin);
        mpf_clear(mp_cos);
        mpf_clear(mp_exp);
        mpf_clear(mp_pi);
        mpf_clear(mp_e);
        mpf_clear(mp_ln2);
        mpf_clear(mp_ln);
        mpf_clear(mp_log10);
#endif
}
```

最後に，使用した変数を解放します。

4.2 複素数

BNCpack で定義した複素数型は独自のもので，C++標準の complex クラスとも，GSL(`http://sources.redhat.com/gsl/`) のそれとも互換性はありません。

とある先生からは C++の complex クラスで mpf_t もしくは mpfr_t 型が利用できるようにならないか？というご要望を受けています。テンプレートですから，ちょっと苦労すれば出来そうな気もしますが … どなたかやりません？

4.2.1 倍精度の場合

```
DCmplx dca, dcb, dcc;
```

IEEE754 倍精度複素数 DCmplx 型の変数を宣言します。単精度の場合は FCmplx と宣言します。

```
/* init */
dca = init_dcmplx();
dcb = init_dcmplx();
dcc = init_dcmplx();
```

複素数 *dca*, *dcb*, *dcc* の領域を確保して初期化 (0 にセット) します。

```
/* set */
set_real_dcmplx(dca, 1.0);
set_image_dcmplx(dca, 2.0);
set_real_dcmplx(dcb, 3.0);
set_image_dcmplx(dcb, 4.0);
set_real_dcmplx(dcc, (double)rand());
set_image_dcmplx(dcc, (double)rand());

/* print */
printf("dca = "); print_dcmplx(dca);
printf("dcb = "); print_dcmplx(dcb);
printf("dcc = "); print_dcmplx(dcc);
```

$dca = 1 + 2i$, $dcb = 3 + 4i$，dcc の実数部，虚数部に乱数をセットし，表示します。

```
/* basic arithmetic */
add_dcmplx(dcc, dca, dcb);
printf("dca + dcb = "); print_dcmplx(dcc);
sub_dcmplx(dcc, dca, dcb);
printf("dca - dcb = "); print_dcmplx(dcc);
add2_dcmplx(dcc, dcb);
printf("dca - dcb + dcb = "); print_dcmplx(dcc);

mul_dcmplx(dcc, dca, dcb);
printf("dca * dcb = "); print_dcmplx(dcc);
div_dcmplx(dcc, dca, dcb);
printf("dca / dcb = "); print_dcmplx(dcc);
mul2_dcmplx(dcc, dcb);
printf("dca / dcb * dcb = "); print_dcmplx(dcc);

printf("~dca  = ");
conj_dcmplx(dcc, dca); print_dcmplx(dcc);
printf("|dca| = %25.17e\n", abs_dcmplx(dca));
printf("exp(i*dca) = "); i
exp_dcmplx(dcc, abs_dcmplx(dca)); print_dcmplx(dcc);
```

四則演算，初等関数を実行し，その結果を表示します。

```
/* clear */
free_dcmplx(dca);
free_dcmplx(dcb);
free_dcmplx(dcc);
```

最後に，確保した複素数型の変数領域を解放します。

4.2.2 多倍長の場合

```
MPFCmplx mpfca, mpfcb, mpfcc;
mpf_t tmp;
```

多倍長複素数 MPFCmplx 型の変数を宣言します。

```
/* init */
set_bnc_default_prec(128);
```

デフォルトの精度を 128bit とします。

```
mpfca = init_mpfcmplx();
mpfcb = init_mpfcmplx();
mpfcc = init2_mpfcmplx(256);
```

複素数 *mpfca*, *mpfcb*, *mpfcc* の領域を確保して初期化 (0 にセット) します。*mpfcc* はデフォルトの精度とは異なり，256bit の精度を持つように初期化されています。

```
/* set */
```

```
set_real_mpfcmplx_ui(mpfca, 1);
set_image_mpfcmplx_ui(mpfca, 2);
set_real_mpfcmplx_ui(mpfcb, 3);
set_image_mpfcmplx_ui(mpfcb, 4);
set_real_mpfcmplx_ui(mpfcc, rand());
set_image_mpfcmplx_ui(mpfcc, rand());

/* print */
printf("mpfca = "); print_mpfcmplx(mpfca);
printf("mpfcb = "); print_mpfcmplx(mpfcb);
printf("mpfcc = "); print_mpfcmplx(mpfcc);
```

$dca = 1 + 2\mathrm{i}$, $dcb = 3 + 4\mathrm{i}$，dcc の実数部，虚数部に乱数をセットし，表示します。

```
/* basic arithmetic */
add_mpfcmplx(mpfcc, mpfca, mpfcb);
printf("mpfca + mpfcb = "); print_mpfcmplx(mpfcc);
sub_mpfcmplx(mpfcc, mpfca, mpfcb);
printf("mpfca - mpfcb = "); print_mpfcmplx(mpfcc);
add2_mpfcmplx(mpfcc, mpfcb);
printf("mpfca - mpfcb + mpfcb = "); print_mpfcmplx(mpfcc);

mul_mpfcmplx(mpfcc, mpfca, mpfcb);
printf("mpfca * mpfcb = "); print_mpfcmplx(mpfcc);
div_mpfcmplx(mpfcc, mpfca, mpfcb);
printf("mpfca / mpfcb = "); print_mpfcmplx(mpfcc);
mul2_mpfcmplx(mpfcc, mpfcb);
printf("mpfca / mpfcb * mpfcb = "); print_mpfcmplx(mpfcc);

printf("~mpfca  = ");
conj_mpfcmplx(mpfcc, mpfca); print_mpfcmplx(mpfcc);
mpf_init(tmp); abs_mpfcmplx(tmp, mpfca);
printf("|mpfca| = ");
mpf_out_str(stdout, 10, 0, tmp); printf("\n");
printf("exp(i*mpfca) = ");
iexp_mpfcmplx(mpfcc, tmp); print_mpfcmplx(mpfcc);
```

四則演算，初等関数を実行し，その結果を表示します。

```
mpf_clear(tmp);

/* clear */
free_mpfcmplx(mpfca);
free_mpfcmplx(mpfcb);
free_mpfcmplx(mpfcc);
```

最後に，確保した複素数型の変数領域を解放します。

4.3 多項式

4.3.1 倍精度の場合

```
#define MAX_POLY_LEN 4096
#define MAX_DEGREE 1024

main()
{
long int i;

DPoly dpa, dpb, dpc;
DCmplx dca, dcret;
```

IEEE754 倍精度実係数を持つ多項式型 (DPoly) の変数 dpa, dpb, dpc と倍精度複素数型 (DCmplx) の変数 dca, $dcret$ を宣言します。以下，複素数型の説明は省略します。

```
/* init */
dpa = init_dpoly(MAX_POLY_LEN);
dpb = init_dpoly(MAX_POLY_LEN);
dpc = init_dpoly(MAX_POLY_LEN);

dca = init_dcmplx();
dcret = init_dcmplx();
set_real_dcmplx(dca, 1.0);
set_image_dcmplx(dca, 1.0);
```

多項式型変数を初期化します。

```
for(i = 0; i <= MAX_DEGREE; i++)

set_dpoly_i(dpa, i, (double)i);
set_dpoly_i(dpb, i, (double)rand());
set_dpoly_i(dpc, i, (double)rand());
```

多項式の係数をセットします。

```
printf("dpa: O(x^%d)\n", setdegree_dpoly(dpa));print_dpoly(dpa);
printf("dpa (1) = %25.17e\n", eval_dpoly(dpa, 1.0));
printf("dpa'(1) = %25.17e\n", eval_diff_dpoly(dpa, 1.0));
printf("dpa (1+1i) = "); ceval_dpoly(dcret, dpa, dca);
print_dcmplx(dcret);
printf("dpa'(1+i1) = "); ceval_diff_dpoly(dcret, dpa, dca);
print_dcmplx(dcret);
printf("dpa (2) = %25.17e\n", eval_dpoly(dpa, 2.0));
printf("dpa'(2) = %25.17e\n", eval_diff_dpoly(dpa, 2.0));

printf("dpb: \n");print_dpoly(dpb);
printf("dpc: \n");print_dpoly(dpc);
```

多項式及びその導関数を使った計算を行い，結果を出力します。

```
/* clear */
free_dpoly(dpa);
free_dpoly(dpb);
free_dpoly(dpc);
}
```

最後に多項式型変数領域を解放します。

4.3.2 多倍長の場合

```
#define MAX_POLY_LEN 4096
#define MAX_DEGREE 1024

main()
{
long int i;

MPFPoly mpf_pa, mpf_pb, mpf_pc;
mpf_t mpf_x, mpf_ret;
MPFCmplx mpfca, mpf_cret;
```

多倍長実係数を持つ多項式型 (`MPFPoly`) の変数 *mpf_pa*, *mpf_pb*, *mpf_pc* と多倍長複素数型 (`MPFCmplx`) の変数 *mpfca*, *mpc_cret* を宣言します。

```
/* init */
mpf_pa = init2_mpfpoly(MAX_POLY_LEN, 128);
mpf_pb = init2_mpfpoly(MAX_POLY_LEN, 256);
mpf_pc = init2_mpfpoly(MAX_POLY_LEN, 1024);

mpfca = init2_mpfcmplx(1024);
mpf_cret = init2_mpfcmplx(1024);
set_real_mpfcmplx_ui(mpfca, 1UL);
set_image_mpfcmplx_ui(mpfca, 1UL);
```

多項式型変数を初期化します。

```
for(i = 0; i <= MAX_DEGREE; i++)

set_mpfpoly_i_d(mpf_pa, i, (double)i);
set_mpfpoly_i_d(mpf_pb, i, (double)rand());
set_mpfpoly_i_d(mpf_pc, i, (double)rand());
```

多項式の係数をセットします。IEEE754 倍精度の実数をセットすると 2 進数のビットパターンがそのまま転写されますので，精度は倍精度程度で止まってしまうことに留意して下さい[2]。

```
mpf_init2(mpf_x, 128);
```

[2] それで何度失敗したことか … 。

```
mpf_init2(mpf_ret, 128);

printf("mpf_pa: O(x^%d)\n", setdegree_mpfpoly(mpf_pa));
print_mpfpoly(mpf_pa);

mpf_set_ui(mpf_x, 1UL);
printf("mpf_pa(1) = ");
eval_mpfpoly(mpf_ret, mpf_pa, mpf_x);
mpf_out_str(stdout, 0, 10, mpf_ret);
printf("\n");
printf("mpf_pa'(1) = ");
eval_diff_mpfpoly(mpf_ret, mpf_pa, mpf_x);
mpf_out_str(stdout, 0, 10, mpf_ret);
printf("\n");
mpf_set_ui(mpf_x, 1UL);

printf("mpf_pa(1+1i) = ");
ceval_mpfpoly(mpf_cret, mpf_pa, mpfca);
print_mpfcmplx(mpf_cret);
printf("mpf_pa'(1+1i) = ");
ceval_diff_mpfpoly(mpf_cret, mpf_pa, mpfca);
print_mpfcmplx(mpf_cret);

mpf_set_ui(mpf_x, 2UL);
printf("mpf_pa(2) = ");
eval_mpfpoly(mpf_ret, mpf_pa, mpf_x);
mpf_out_str(stdout, 0, 10, mpf_ret);
printf("\n");
printf("mpf_pa'(2) = ");
eval_diff_mpfpoly(mpf_ret, mpf_pa, mpf_x);
mpf_out_str(stdout, 0, 10, mpf_ret);
printf("\n");

mpf_set_ui(mpf_x, 100000UL);
printf("mpf_pa(100000) = ");
eval_mpfpoly(mpf_ret, mpf_pa, mpf_x);
mpf_out_str(stdout, 0, 10, mpf_ret);
printf("\n");
printf("mpf_pa'(100000) = ");
eval_diff_mpfpoly(mpf_ret, mpf_pa, mpf_x);
mpf_out_str(stdout, 0, 10, mpf_ret);
printf("\n");

printf("mpf_pb: \n");print_mpfpoly(mpf_pb);
printf("mpf_pc: \n");print_mpfpoly(mpf_pc);
```

多項式及びその導関数を使った計算を行い，結果を出力します。

```
mpf_clear(mpf_x);
mpf_clear(mpf_ret);

/* clear */
free_mpfpoly(mpf_pa);
free_mpfpoly(mpf_pb);
free_mpfpoly(mpf_pc);
}
```

最後に，使用した変数領域を解放します。

4.4 線型計算

線型計算のサンプルは次の LU 分解，CG 法の例で代替します。

4.5 LU 分解

4.5.1 倍精度の場合

```
DMatrix da;
DVector db, dx, dans;

long int ret_f, ret_d, ret_mpf;
long int row_ch[DIM], col_ch[DIM];
long int i, j;

/* initialize */
da = init_dmatrix(DIM, DIM);
db = init_dvector(DIM);
dx = init_dvector(DIM);
dans = init_dvector(DIM);
```

IEEE754 倍精度実行列型 DMatrix，同ベクトル型 DVectorの変数を宣言し，初期化します。

```
/* get problem */
get_dproblem(da, db, dans);
print_dmatrix(da);
```

テスト問題の係数行列，定数ベクトル，真の解を与え，係数行列を表示します。テスト問題を作る関数 get_dproblemは，例えば以下のように記述します。

```
void get_dproblem(DMatrix a, DVector b, DVector ans)
{
        long int i, j, k;
        double tmp;

        /* Lotkin Matrix */
        for(i = 0; i < a->col_dim; i++)
                set_dmatrix_ij(a, 0, i, 1.0);
        for(i = 1; i < a->row_dim; i++)
        {
```

```
                for(j = 0; j < a->col_dim; j++)
                        set_dmatrix_ij(a, i, j, 1.0 / (i + j + 1));
        }

        /* Answer */
        for(i = 0; i < ans->dim; i++)
                set_dvector_i(ans, i, (double)i);

        /* Make constant vector */
        mul_dmatrix_dvec(b, a, ans);
}

/* run DLUdecomp & SolveDLS */
// ret_d = DLUdecomp(da);
// ret_d = DLUdecompP(da, row_ch);
 ret_d = DLUdecompC(da, row_ch, col_ch);
```

係数行列の LU 分解を計算します。DLUdecompは pivoting なし，DLUdcompPは部分 pivoting のみ実行，DLUdecompCは完全 pivoting を実行します。

```
// ret_d = SolveDLS(dx, da, db);
// ret_d = SolveDLSP(dx, da, db, row_ch);
 ret_d = SolveDLSC(dx, da, db, row_ch, col_ch);

/* print */
printf("  i    row_ch[i]    col_ch[i]\n");
for(i = 0; i < DIM; i++)
printf("%5ld %25.17e %25.17e\n", i, get_dvector_i(dx, i),\
 get_dvector_i(dans, i));
```

LU 分解された係数行列を用いて後退代入を行い，解を出力します。

```
/* end */
free_dmatrix(da);
free_dvector(db);
free_dvector(dx);
free_dvector(dans);
```

最後に，使用した変数領域を解放します。

4.5.2 多倍長の場合

```
MPFMatrix mpfa;
MPFVector mpfb, mpfx, mpfans;
mpf_t reps, aeps;
long int ret_f, ret_d, ret_mpf;
long int row_ch[DIM], col_ch[DIM];
long int i, j;

set_bnc_default_prec(256);
```

```
/* initialize */
mpf_init(reps); mpf_init(aeps);
mpfa = init_mpfmatrix(DIM, DIM);
// mpfa = init2_mpfmatrix(DIM, DIM, 256);
mpfb = init_mpfvector(DIM);
// mpfb = init2_mpfvector(DIM, 256);
mpfx = init_mpfvector(DIM);
// mpfx = init2_mpfvector(DIM, 256);
mpfans = init_mpfvector(DIM);
```

多倍長実行列型 MPFMatrix，同ベクトル型 MPFVectorの変数を宣言し，デフォルトの精度を256bit にして初期化します。

```
/* get problem */
get_mpfproblem(mpfa, mpfb, mpfans);
print_mpfmatrix(mpfa);
```

テスト問題の係数行列，定数ベクトル，真の解を与え，係数行列を表示します。テスト問題を作る関数 get_mpfproblemは，例えば以下のように記述します。

```
void get_mpfproblem(MPFMatrix a, MPFVector b, MPFVector ans)
{
        long int i, j, k;
        mpf_t tmp;

        mpf_init(tmp);

        /* Lotkin Matrix */
        for(i = 0; i < a->col_dim; i++)
                set_mpfmatrix_ij_d(a, 0, i, 1.0);
        for(i = 1; i < a->row_dim; i++)
        {
                for(j = 0; j < a->col_dim; j++)
                {
                        mpf_set_ui(tmp, 1UL);
                        mpf_div_ui(tmp, tmp, \
(unsigned long)(i + j + 1));
                        set_mpfmatrix_ij(a, i, j, tmp);
                }
        }


        /* Answer */
        for(i = 0; i < ans->dim; i++)
        {
                mpf_set_si(tmp, i);
                set_mpfvector_i(ans, i, tmp);
        }
```

```
        /* Make constant vector */
        mul_mpfmatrix_mpfvec(b, a, ans);
}

/* run MPFLUdecomp & SolveMPFLS */
 ret_mpf = MPFLUdecomp(mpfa);
// ret_mpf = MPFLUdecompP(mpfa, row_ch);
// ret_mpf = MPFLUdecompC(mpfa, row_ch, col_ch);
```

係数行列の LU 分解を計算します。MPFLUdecompは pivoting なし，MPFLUdcompPは部分 pivoting のみ実行，MPFLUdecompCは完全 pivoting を実行します。

```
 ret_mpf = SolveMPFLS(mpfx, mpfa, mpfb);
// ret_mpf = SolveMPFLSP(mpfx, mpfa, mpfb, row_ch);
// ret_mpf = SolveMPFLSC(mpfx, mpfa, mpfb, row_ch, col_ch);

/* print */
for(i = 0; i < DIM; i++)

printf("%5ld ", i);
mpf_out_str(stdout, 10, 0, get_mpfvector_i(mpfx, i));
printf(" ");
mpf_out_str(stdout, 10, 0, get_mpfvector_i(mpfans, i));
printf("\n");
```

LU 分解された係数行列を用いて後退代入を行い，解を出力します。

```
/* end */
mpf_clear(reps); mpf_clear(aeps);
free_mpfmatrix(mpfa);
free_mpfvector(mpfb);
free_mpfvector(mpfx);
free_mpfvector(mpfans);
```

最後に，使用した変数領域を解放します。

4.6 Conjugate-Gradient 法

4.6.1 倍精度の場合

```
/* initialize */
da = init_dmatrix(DIM, DIM);
db = init_dvector(DIM);
dx = init_dvector(DIM);
dans = init_dvector(DIM);

/* get problem */
get_dproblem(da, db, dans);

print_dmatrix(da);
```

連立一次方程式の係数行列 *da*，定数ベクトル *db*, 近似解を格納するベクトル *dx*，真の解ベクトル *dans* を宣言し，値を代入して，係数行列を表示します。

```
/* run DCG */
itimes_d = DCG(dx, da, db, 1.0e-13, 1.0e-99, DIM * 5);

/* print */
for(i = 0; i < DIM; i++)
printf("%5ld %25.17e %25.17e\n",
i,
get_dvector_i(dx, i),
get_dvector_i(dans, i));
```

CG 法 (Conjugate-Gradient 法) を実行し，数値解を表示します。

```
/* end */
free_dmatrix(da);
free_dvector(db);
free_dvector(dx);
free_dvector(dans);
```

使用した変数領域を解放します。

4.6.2 多倍長の場合

```
set_bnc_default_prec(128);

/* initialize */
mpf_init(reps);
mpf_init2(reps2, 256);
mpf_init2(reps3, 512);
mpf_init(aeps);
mpf_init2(aeps2, 256);
mpf_init2(aeps3, 512);
mpfa  = init_mpfmatrix(DIM, DIM);
mpfa2 = init2_mpfmatrix(DIM, DIM, 256);
mpfa3 = init2_mpfmatrix(DIM, DIM, 512);
mpfb  = init_mpfvector(DIM);
mpfb2 = init2_mpfvector(DIM, 256);
mpfb3 = init2_mpfvector(DIM, 512);
mpfx  = init_mpfvector(DIM);
mpfx2 = init2_mpfvector(DIM, 256);
mpfx3 = init2_mpfvector(DIM, 512);
mpfans  = init_mpfvector(DIM);
mpfans2 = init2_mpfvector(DIM, 256);
mpfans3 = init2_mpfvector(DIM, 512);

/* get problem */
get_mpfproblem(mpfa, mpfb, mpfans);
get_mpfproblem(mpfa2, mpfb2, mpfans2);
```

```
get_mpfproblem(mpfa3, mpfb3, mpfans3);

print_mpfmatrix(mpfa);
print_mpfmatrix(mpfa2);
print_mpfmatrix(mpfa3);

/* run MPFFCG */
mpf_set_d(reps, 1.0e-20);
mpf_set_d(reps2, 1.0e-20);
mpf_set_d(reps3, 1.0e-20);
mpf_set_d(aeps, 1.0e-50);
mpf_set_d(aeps2, 1.0e-50);
mpf_set_d(aeps3, 1.0e-50);
```

連立一次方程式の係数行列 *mpfa*/*mpfa2*/*mpfa3*，定数ベクトル *mpfb*/*mpfb2*/*mpfb3*, 近似解を格納するベクトル *mpfx*/*mpfx2*/*mpfx3*，真の解ベクトル *mpfans*/*mpfans2*/*mpfans3* を，それぞれ 128bit, 256bit, 512bit の精度を持つように宣言し，値を代入して，係数行列をそれぞれ表示します。

```
itimes_mpf  = MPFCG(mpfx, mpfa, mpfb, reps, aeps, DIM * 5);
itimes_mpf2 = MPFCG(mpfx2, mpfa2, mpfb2, reps2, aeps2, DIM * 5);
itimes_mpf3 = MPFCG(mpfx3, mpfa3, mpfb3, reps3, aeps3, DIM * 5);

/* print */
for(i = 0; i < DIM; i++)

printf("%5ld ", i);
mpf_out_str(stdout, 10, 0, get_mpfvector_i(mpfx, i));
printf(" ");
mpf_out_str(stdout, 10, 0, get_mpfvector_i(mpfx2, i));
printf(" ");
mpf_out_str(stdout, 10, 0, get_mpfvector_i(mpfx3, i));
printf(" ");
mpf_out_str(stdout, 10, 0, get_mpfvector_i(mpfans, i));
printf("\n");
```

CG 法 (Conjugate-Gradient 法) を実行し，各精度の数値解を表示します。

```
/* end */
mpf_clear(reps); mpf_clear(aeps);
mpf_clear(reps2); mpf_clear(aeps2);
mpf_clear(reps3); mpf_clear(aeps3);
free_mpfmatrix(mpfa);
free_mpfmatrix(mpfa2);
free_mpfmatrix(mpfa3);
free_mpfvector(mpfb);
free_mpfvector(mpfb2);
free_mpfvector(mpfb3);
```

```
free_mpfvector(mpfx);
free_mpfvector(mpfx2);
free_mpfvector(mpfx3);
free_mpfvector(mpfans);
free_mpfvector(mpfans2);
free_mpfvector(mpfans3);
```

最後に使用した変数領域を解放します。

4.7 Newton 法

4.7.1 倍精度の場合

```
double fd(double x)
{
        /* x^2 - 2 = 0 */
        return x * x - 2.0;
}

double dfd(double x)
{
        /* 2*x */
        return 2.0 * x;
}
```

Newton 法及び簡易 Newton 法で使用する方程式 $f(x) = 0$ の左辺の関数 `f`，及びその導関数 `df`の定義をしています。

```
/* double */
times = dnewton_1(&dans, 1.0, fd, dfd, 100, 1.0e-50, 1.0e-10);
printf(" times : %ld\n", times);
printf("dnewton_1: %25.17e\n", dans);
times = dsnewton_1(&dans, 1.0, fd, dfd, 100, 1.0e-50, 1.0e-10);
printf(" times : %ld\n", times);
printf("dsnewton_1: %25.17e\n", dans);
```

一次元の Newton 法及び簡易 Newton 法を実行してその結果を出力します。

4.7.2 多倍長の場合

```
void f(mpf_t ret, mpf_t x)
{
        /* x^2 - 2 = 0 */
        mpf_mul(ret, x, x);
        mpf_sub_ui(ret, ret, 2UL);
}

void df(mpf_t ret, mpf_t x)
{
        /* 2*x */
        mpf_mul_ui(ret, x, 2UL);
```

```
}
```

Newton 法及び簡易 Newton 法で使用する方程式 $f(x) = 0$ の左辺の関数 f，及びその導関数 df の定義をしています。

```
/* mpf_t */
set_bnc_default_prec(1024);
mpf_init(ans);
mpf_init(x0);
mpf_init_set_d(aeps, 1.0e-100);
mpf_init_set_d(reps, 1.0e-50);

/* newton_1 */
mpf_set_ui(x0, 1UL);
times = mpf_newton_1(ans, x0, f, df, 100, aeps, reps);
printf(" times : %ld\n", times);
printf("mpf_newton_1:   "); mpf_out_str(stdout, 10, 0, ans);
printf("\n");

/* snewton_1 */
times = mpf_snewton_1(ans, x0, f, df, 100, aeps, reps);
printf(" times : %ld\n", times);
printf("mpf_snewton_1:   "); mpf_out_str(stdout, 10, 0, ans);
printf("\n");

mpf_sqrt_ui(ans, 2UL);
printf("mpf_sqrt: "); mpf_out_str(stdout, 10, 0, ans); printf("\n");
```

一次元の Newton 法及び簡易 Newton 法を実行してその結果を出力します。

4.8 DKA 法

4.8.1 倍精度の場合

```
/* init */
dabs_eps = 1.0e-100;
drel_eps = 1.0e-14;
df = init_dpoly(MAX_LENGTH);
cdans = init_cdarray(10);
cdinit = init_cdarray(10);
```

ここで DKA 法の停止則に必要な値をセットします。

```
/* ff = (x-1)(x-2)...(x-10) */
set_dpoly_i(df, 0, (double)3628800);
set_dpoly_i(df, 1, (double)-10628640);
set_dpoly_i(df, 2, (double)12753576);
set_dpoly_i(df, 3, (double)-8409500);
set_dpoly_i(df, 4, (double)3416930);
set_dpoly_i(df, 5, (double)-902055);
```

```
set_dpoly_i(df, 6, (double)157773);
set_dpoly_i(df, 7, (double)-18150);
set_dpoly_i(df, 8, (double)1320);
set_dpoly_i(df, 9, (double)-55);
set_dpoly_i(df,10, (double)1);

print_dpoly(df);
```

多項式の係数を低次項からセットし，表示します。

```
/* set Aberth's initial value */
ddka_init(cdinit, df);
print_cdarray(cdinit);
```

Aberth の初期値を計算し，*cdinit* にセットします。

```
/* DKA method */
dtimes = ddka(cdans, cdinit, df, 1000, dabs_eps, drel_eps);
```

DKA 法を実行します。*dtimes* に反復回数が格納されます。

```
/* print answer */
printf("Iterative times: %d\n", dtimes);
print_cdarray(cdans);
```

反復回数と近似解 (*cdans*) が表示されます。

```
/* clear */
free_dpoly(df);
free_cdarray(cdans);
free_cdarray(cdinit);
```

最後に，使用した変数を解放します。

4.8.2 多倍長の場合

```
set_bnc_default_prec(512);

/* init */
mpf_init_set_d(mpfabs_eps, 1.0e-300);
mpf_init_set_d(mpfrel_eps, 1.0e-25);
```

ここで必要となる精度 (512bit) と，DKA 法の停止則に必要な値をセットします。

```
mpff = init_mpfpoly(MAX_LENGTH);
cmpfans = init_cmpfarray(20);
cmpfinit = init_cmpfarray(20);
```

多倍長実数係数の多項式 (*mpff*)，多倍長複素数の配列 (*cmpfans*, *cmpfinit*) を初期化します。

```
/* ff = (x-1)(x-2)...(x-20) */
set_mpfpoly_i_str(mpff, 0, "2432902008176640000", 10);
set_mpfpoly_i_str(mpff, 1, "-8752948036761600000", 10);
```

```
set_mpfpoly_i_str(mpff, 2, "13803759753640704000", 10);
set_mpfpoly_i_str(mpff, 3, "-12870931245150988800", 10);
set_mpfpoly_i_str(mpff, 4, "8037811822645051776", 10);
set_mpfpoly_i_str(mpff, 5, "-3599979517947607200", 10);
set_mpfpoly_i_str(mpff, 6, "1206647803780373360", 10);
set_mpfpoly_i_str(mpff, 7, "-311333643161390640", 10);
set_mpfpoly_i_str(mpff, 8, "63030812099294896", 10);
set_mpfpoly_i_str(mpff, 9, "-10142299865511450", 10);
set_mpfpoly_i_str(mpff,10, "1307535010540395", 10);
set_mpfpoly_i_str(mpff,11, "-135585182899530", 10);
set_mpfpoly_i_str(mpff,12, "11310276995381", 10);
set_mpfpoly_i_str(mpff,13, "-756111184500", 10);
set_mpfpoly_i_str(mpff,14, "40171771630", 10);
set_mpfpoly_i_str(mpff,15, "-1672280820", 10);
set_mpfpoly_i_str(mpff,16, "53327946", 10);
set_mpfpoly_i_str(mpff,17, "-1256850", 10);
set_mpfpoly_i_str(mpff,18, "20615", 10);
set_mpfpoly_i_str(mpff,19, "-210", 10);
set_mpfpoly_i_str(mpff,20, "1", 10);

print_mpfpoly(mpff);
```

多項式の係数を低次項からセットし，表示します。

```
/* set Aberth's initial value */
mpf_dka_init(cmpfinit, mpff);
print_cmpfarray(cmpfinit);
```

Aberth の初期値を計算し，*cmpfinit* にセットします。

```
/* DKA method */
mpftimes = mpf_dka(cmpfans, cmpfinit, mpff, 1000, \
mpfabs_eps, mpfrel_eps);
```

DKA 法を実行します。*mpftimes* に反復回数が格納されます。

```
/* print answer */
printf("Iterative times: %d\n", mpftimes);
print_cmpfarray(cmpfans);
```

反復回数と近似解 (*cmpfans*) が表示されます。

```
/* clear */
free_mpfpoly(mpff);
free_cmpfarray(cmpfans);
free_cmpfarray(cmpfinit);
```

最後に，使用した変数を解放します。

5 今後の予定 (あてにしないように)

今後の予定としては

1. 常微分方程式の Solver を別パッケージにして更に拡充する。
2. gmpxx.h や mpfrxx.h といった C++クラスインターフェースを使ったサンプルプログラムの提示
3. 基本線型計算ルーチンの高速化

といったものがありますが，気が向かないと何もしない我が儘な性格なので，あまりあてにしないで下さい。

ご要望を頂いてもすぐにお答えできるかどうか分かりませんが，もし何かご意見などありましたら，マニュアル表紙のメールアドレスまでお願い致します。

謝辞・参考文献

GNU MP なかりせば本ライブラリは存在しませんでした。開発チーム一同に感謝いたします。

また，プリンタがトラブって以来，一貫して FSF を率いつつ,「自由なソフトウェア」(Free Software) の普及活動に邁進し続けている RMS にも感謝いたします。GNU Project なかりせば，開発環境に使っている Linux も存在しなかったでしょうから。

- Trobjorn Grandlund, "GNU MP: The GNU Multiple Precision Arithmetic Library", Free Software Foudation, 2000, `http://www.swox.com/gmp/`.
- Robert J. Chassell, "Texinfo: The GNU Documentation Format", Free Software Foundation, 1996, `http://www.gnu.org/manual/texinfo/index.html`.
- The MPFR Team, LORIA/INRIA Lorraine, "MPFR: The Multiple Precision Floating-Point Reliable Library", 2002, `http://www.mpfr.org/`.
- 幸谷 智紀,「ソフトウェアとしての数値計算」, 2002, `http://www.pas-net.jp/nasoft/`.

索引

(Index is nonexistent)

Table of Contents