

# **BNCpack**

---

Basic Numerical Calculation package  
Version 0.7  
September 1, 2011

**Tomonori Kouya at Kakegawa, JAPAN**

<http://na-inet.jp/> ([tkouya@gmail.com](mailto:tkouya@gmail.com))

---

Copyright © 2000-2011 Tomonori Kouya

本マニュアルを含む BNCpack は LGPL3 に則った扱いを要求するフリーソフトウェアです。また、ソフトウェアの内容及び品質について、作者は一切の保証及び責任を負わないものとします。

# 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)<sup>1</sup> 及び GNU MPFR<sup>2</sup> 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、やつ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<sup>3</sup> 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<sup>4</sup>。

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<sup>5</sup> 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。

---

<sup>1</sup> 2011 年 8 月現在の最新版は Version 5.0.2 です。URL は <http://gmplib.org/>

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

<sup>3</sup> Microsoft の商標です。

<sup>4</sup> 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。

<sup>5</sup> 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ありません。

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<sup>1</sup> 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<sup>2</sup>。

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 ‘libnc.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 ‘libnc.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/libnc.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://gmplib.org/> 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。私の使用している RPM 系の Linux や Plamo Linux/Slackware Linux では

<sup>1</sup> MPFR パッケージの マニュアル参照。

<sup>2</sup> 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が。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<sup>3</sup>
- 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。

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が。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。

<sup>3</sup> 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く

```
% 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 */
} fcplx;
```

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 farray *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;   // Indices 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

```

但し、現状では格納段階では `nzero_row_dim[]` はセットされません。設定する際には `set_nzero_row_dim` 関数を呼び出して下さい。

## 3.2 基本関数

<code>long lmax (long rop, long op)</code>	[Function]
<code>long lmin (long rop, long op)</code>	[Function]
<code>float fmax (float rop, float op)</code>	[Function]
<code>float fmin (float rop, float op)</code>	[Function]
<code>double dmax (double rop, double op)</code>	[Function]
<code>double dmin (double rop, double op)</code>	[Function]
<code>mpf_ptr mpf_max (mpf_t rop, mpf_t op)</code>	[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]  
`ropop` 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 top)` [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_fcplx (void)` [Function]  
`DCmplx init_dcplx (void)` [Function]  
`MPFCmplx init_mpf_cplx (void)` [Function]  
 複素数を格納する領域を確保します。

`MPFCmplx init2_mpf_cplx (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_fcplx (FCmplx op)` [Function]  
`void free_dcplx (DCmplx op)` [Function]  
`void free_mpf_cplx (MPFCmplx op)` [Function]  
 複素数 `op` の格納されている領域を解放します。

<code>float get_real_fcplx (FCplx op)</code>	[Function]
<code>double get_real_dcplx (DCplx op)</code>	[Function]
<code>void get_real_mpf_cplx (mpf_t rop, MPFCplx op)</code> 複素数 <i>op</i> の実数部を返します。多倍長型は <i>rop</i> に実数部が格納されます。	[Function]
<code>float get_image_fcplx (FCplx op)</code>	[Function]
<code>double get_image_dcplx (DCplx op)</code>	[Function]
<code>void get_image_mpf_cplx (mpf_t rop, MPFCplx op)</code> 複素数 <i>op</i> の虚数部を返します。多倍長型は <i>rop</i> に虚数部が格納されます。	[Function]
<code>void set_real_fcplx (FCplx rop, float val)</code>	[Function]
<code>void set_real_dcplx (DCplx rop, double val)</code>	[Function]
<code>void set_real_mpf_cplx (MPFCplx rop, mpf_t val)</code> 実数 <i>val</i> を複素数 <i>rop</i> の実数部に格納します。	[Function]
<code>void set_real_mpf_cplx_ui (MPFCplx rop, unsigned long int val)</code> 整数 <i>val</i> を複素数 <i>rop</i> の実数部に格納します。	[Function]
<code>void set_image_fcplx (FCplx rop, float val)</code>	[Function]
<code>void set_image_dcplx (DCplx rop, double val)</code>	[Function]
<code>void set_image_mpf_cplx (MPFCplx rop, mpf_t val)</code> 実数 <i>val</i> を複素数 <i>rop</i> の虚数部に格納します。	[Function]
<code>void set_image_mpf_cplx_ui (MPFCplx rop, unsigned long int val)</code> 符号なし長整数 <i>val</i> を複素数 <i>rop</i> の虚数部に格納します。	[Function]
<code>void subst_fcplx (FCplx rop, FCplx op)</code>	[Function]
<code>void subst_dcplx (DCplx rop, DCplx op)</code>	[Function]
<code>void subst_mpf_cplx (MPFCplx rop, MPFCplx op)</code> 複素数 <i>op</i> を複素数 <i>rop</i> に代入します。	[Function]
<code>void set0_fcplx (FCplx rop)</code>	[Function]
<code>void set0_dcplx (DCplx rop)</code>	[Function]
<code>void set0_mpf_cplx (MPFCplx rop)</code> 複素数 <i>rop</i> に 0 をセットします。	[Function]
<code>void add_fcplx (FCplx rop, FCplx op1, FCplx op2)</code>	[Function]
<code>void add_dcplx (DCplx rop, DCplx op1, DCplx op2)</code>	[Function]
<code>void add_mpf_cplx (MPFCplx rop, MPFCplx op1, MPFCplx op2)</code> 複素数 <i>op1</i> と複素数 <i>op2</i> の和を複素数 <i>rop</i> に格納します。	[Function]
<code>void add_fcplx_f (FCplx rop, FCplx op, float val)</code>	[Function]
<code>void add_dcplx_d (DCplx rop, DCplx op, double val)</code>	[Function]
<code>void add_mpf_cplx_mpf (MPFCplx rop, MPFCplx op, mpf_t val)</code> 複素数 <i>op</i> に実数 <i>val</i> を加え、複素数 <i>rop</i> に格納します。	[Function]

<code>void add2_fcplx (FCplx rop, FCplx op)</code>	[Function]
<code>void add2_dcplx (DCplx rop, DCplx op)</code>	[Function]
<code>void add2_mpf_cplx (MPFCplx rop, MPFCplx op)</code> 複素数 <i>rop</i> に複素数 <i>op</i> を加え、結果を <i>rop</i> に戻します。	[Function]
<code>void sub_fcplx (FCplx rop, FCplx op1, FCplx op2)</code>	[Function]
<code>void sub_dcplx (DCplx rop, DCplx op1, DCplx op2)</code>	[Function]
<code>void sub_mpf_cplx (MPFCplx rop, MPFCplx op1, MPFCplx op2)</code> 複素数同士の差 $op1 - op2$ を計算し、結果を <i>rop</i> に格納します。	[Function]
<code>void conj_fcplx (FCplx rop, FCplx op)</code>	[Function]
<code>void conj_dcplx (DCplx rop, DCplx op)</code>	[Function]
<code>void conj_mpf_cplx (MPFCplx rop, MPFCplx op)</code> 複素数 <i>op</i> と共約な複素数を計算し、結果を <i>rop</i> に返します。	[Function]
<code>void sign_fcplx (FCplx rop, FCplx op)</code>	[Function]
<code>void sign_dcplx (DCplx rop, DCplx op)</code>	[Function]
<code>void sign_mpf_cplx (MPFCplx rop, MPFCplx op)</code> 複素数 <i>op</i> 全体の符号を反転し、結果を <i>rop</i> に格納します。	[Function]
<code>void sign2_fcplx (FCplx op)</code>	[Function]
<code>void sign2_dcplx (DCplx op)</code>	[Function]
<code>void sign2_mpf_cplx (MPFCplx op)</code> 複素数 <i>op</i> 全体の符号を反転し、結果を <i>rop</i> に格納します。	[Function]
<code>float abs_fcplx (FCplx op)</code>	[Function]
<code>double abs_dcplx (DCplx op)</code>	[Function]
<code>void abs_mpf_cplx (mpf_t ret, MPFCplx op)</code> 複素数 <i>op</i> の絶対値を計算して返します。	[Function]
<code>float mul_fcplx (FCplx rop, FCplx op1, FCplx op2)</code>	[Function]
<code>double mul_dcplx (DCplx rop, DCplx op1, DCplx op2)</code>	[Function]
<code>void mul_mpf_cplx (MPFCplx rop, MPFCplx op1, MPFCplx op2)</code> 複素数同士の積 $op1 * op2$ を計算し、結果を <i>rop</i> に格納します。	[Function]
<code>float mul_fcplx_f (FCplx rop, FCplx op, float val)</code>	[Function]
<code>double mul_dcplx_d (DCplx rop, DCplx op, double val)</code>	[Function]
<code>void mul_mpf_cplx_mpf (MPFCplx rop, MPFCplx op, mpf_t val)</code> 複素数 <i>op</i> と実数 <i>val</i> との積を <i>rop</i> に格納します。	[Function]
<code>void mul_mpf_cplx_ui (MPFCplx rop, MPFCplx op, unsigned long int val)</code> 複素数 <i>op</i> と整数 <i>val</i> との積を <i>rop</i> に格納します。	[Function]
<code>float mul2_fcplx (FCplx rop, FCplx op)</code>	[Function]
<code>double mul2_dcplx (DCplx rop, DCplx op)</code>	[Function]

<code>void mul2_mpfcmplx (MPFCmplx rop, MPFCmplx op)</code> 複素数同士の積 $rop \times op$ を計算し、結果を $rop$ に戻します。	[Function]
<code>float div_fcplx (FCmplx rop, FCmplx op1, FCmplx op2)</code>	[Function]
<code>double div_dcplx (DCmplx rop, DCmplx op1, DCmplx op2)</code>	[Function]
<code>void div_mpfcmplx (MPFCmplx rop, MPFCmplx op1, MPFCmplx 複 素数同士の除算 <math>op1/op2</math> を計算し、結果を <math>rop</math> に格納します。</code>	[Function]
<code>void iexp_fcplx (FCmplx rop, float op)</code>	[Function]
<code>void iexp_dcplx (DCmplx rop, double op)</code>	[Function]
<code>void iexp_mpfcmplx (MPFCmplx rop, mpf_t op)</code> $\exp(op \times i)$ を計算し、複素数 $rop$ に格納します。	[Function]
<code>void print_fcplx (FCmplx op)</code>	[Function]
<code>void print_dcplx (DCmplx op)</code>	[Function]
<code>void print_mpfcmplx (MPFCmplx op)</code> 複素数 $op$ の値を標準出力に表示します。	[Function]

### 3.5 配列

<code>FArray init_farray (long int array_size)</code>	[Function]
<code>DArray init_darray (long int array_size)</code>	[Function]
<code>MPFArray init_mpfarray (long int array_size)</code>	[Function]
<code>CFArray init_cfarray (long int array_size)</code>	[Function]
<code>CDArray init_cdarray (long int array_size)</code>	[Function]
<code>CMPFArray init_cmpfarray (long int array_size)</code> $array\_size$ 個のデータが格納できる配列を確保します。	[Function]
<code>MPFArray init2_mpfarray (long int array_size, unsigned long prec)</code>	[Function]
<code>CMPFArray init2_cmpfarray (long int array_size, unsigned long prec)</code> $array\_size$ 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。	[Function]
<code>void free_farray (FArray rop)</code>	[Function]
<code>void free_darray (DArray rop)</code>	[Function]
<code>void free_mpfarray (MPFArray rop)</code>	[Function]
<code>void free_cfarray (CFArray rop)</code>	[Function]
<code>void free_cdarray (CDArray rop)</code>	[Function]
<code>void free_cmpfarray (CMPFArray rop)</code> 配列 $rop$ の記憶領域を削除します。	[Function]
<code>float get_farray_i (FArray op, long int index)</code>	[Function]
<code>double get_darray_i (DArray op, long int index)</code>	[Function]
<code>mpf_ptr get_mpfarray_i (MPFArray op, long int index)</code>	[Function]
<code>FCmplx get_cfarray_i (CFArray op, long int index)</code>	[Function]
<code>DCmplx get_cdarray_i (CDArray op, long int index)</code>	[Function]
<code>MPFCmplx get_cmpfarray_i (CMPFArray op, long int index)</code> 配列 $op$ の $index$ 番目のデータを返します。	[Function]

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\_mfpoly (*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\_mfpoly (*long int degree*, *unsigned long prec*) [Function]  
 係数が *precbit* 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\_mfpoly (*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\_mfpoly\_i (*MPFPoly poly*, *long index*) [Function]

多項式 *poly* の *index* 次係数を返します。



<code>long setdegree_fpoly (FPoly poly)</code>	[Function]
<code>long setdegree_dpoly (DPoly poly)</code> 多項式 <i>poly</i> の次数を返します。	[Function]
<code>void set_fpoly_i (FPoly poly, long index, float val)</code>	[Function]
<code>void set_dpoly_i (DPoly poly, long index, double val)</code>	[Function]
<code>void set_mpfpoly_i (MPFPoly poly, long index, mpf_t val)</code> 多項式 <i>poly</i> の <i>index</i> 次係数に実数 <i>val</i> を格納します。	[Function]
<code>void set_mpfpoly_i_d (MPFPoly poly, long index, double val)</code> 多項式 <i>poly</i> の <i>index</i> 次係数に倍精度実数 <i>val</i> を格納します。	[Function]
<code>void set_mpfpoly_i_str (MPFPoly poly, long index, const char * val, int base)</code> 多項式 <i>poly</i> の <i>index</i> 次係数に, 文字列で表現された <i>base</i> 進数の実数 <i>val</i> を格納します。	[Function]
<code>void print_fpoly (FPoly poly)</code>	[Function]
<code>void print_dpoly (DPoly poly)</code>	[Function]
<code>void print_mpfpoly (MPFPoly poly)</code> 多項式 <i>poly</i> の係数を標準出力に表示します。	[Function]
<code>void print_fdmpfpoly (FPoly fpoly, DPoly dpoly, MPFPoly mpfpoly)</code> 単精度多項式 <i>fpoly</i> , 倍精度多項式 <i>dpoly</i> , 多倍長多項式 <i>mpfpoly</i> の係数を標準出力に表示します。	[Function]
<code>void add_fpoly (FPoly rpoly, FPoly poly1, FPoly poly2)</code>	[Function]
<code>void add_dpoly (DPoly rpoly, DPoly poly1, DPoly poly2)</code>	[Function]
<code>void add_mpfpoly (MPFPoly rpoly, MPFPoly poly1, MPFPoly poly2)</code> 2つの多項式 <i>poly1</i> と <i>poly2</i> の和を <i>rpoly</i> に格納します。	[Function]
<code>void sub_fpoly (FPoly rpoly, FPoly poly1, FPoly poly2)</code>	[Function]
<code>void sub_dpoly (DPoly rpoly, DPoly poly1, DPoly poly2)</code>	[Function]
<code>void sub_mpfpoly (MPFPoly rpoly, MPFPoly poly1, MPFPoly poly2)</code> 2つの多項式 <i>poly1</i> と <i>poly2</i> の差を <i>rpoly</i> に格納します。	[Function]
<code>void cmul_fpoly (FPoly rpoly, float val, FPoly poly)</code>	[Function]
<code>void cmul_dpoly (DPoly rpoly, double val, DPoly poly)</code>	[Function]
<code>void cmul_mpfpoly (MPFPoly rpoly, mpf_t val, MPFPoly poly)</code> 多項式 <i>poly</i> と実数 <i>val</i> とのスカラー積を <i>rpoly</i> に格納します。	[Function]
<code>void subst_fpoly (FPoly rpoly, FPoly poly)</code>	[Function]
<code>void subst_dpoly (DPoly rpoly, DPoly poly)</code>	[Function]
<code>void subst_mpfpoly (MPFPoly rpoly, MPFPoly poly)</code> 多項式 <i>poly</i> を多項式 <i>rpoly</i> に代入します。	[Function]

<code>void set0_fpoly (FPoly poly)</code>	[Function]
<code>void set0_dpoly (DPoly poly)</code>	[Function]
<code>void set0_mpfpoly (MPFPoly poly)</code> 多項式 <i>poly</i> の係数を全て 0 にセットします。	[Function]
<code>long max_abscoef_fpoly (FPoly poly)</code>	[Function]
<code>long max_abscoef_dpoly (DPoly poly)</code>	[Function]
<code>long max_abscoef_mpfpoly (MPFPoly poly)</code> 多項式 <i>poly</i> の絶対値最大係数の次数を返します。	[Function]
<code>long num_nonzero_fpoly (FPoly poly)</code>	[Function]
<code>long num_nonzero_dpoly (DPoly poly)</code>	[Function]
<code>long num_nonzero_mpfpoly (MPFPoly poly)</code> 多項式 <i>poly</i> の非零係数の個数を返します。	[Function]
<code>void diff_fpoly (FPoly poly)</code>	[Function]
<code>void diff_dpoly (DPoly poly)</code>	[Function]
<code>void diff_mpfpoly (MPFPoly poly)</code> 多項式 <i>poly</i> の導関数を計算し, <i>poly</i> に導関数を格納します。	[Function]
<code>float eval_fpoly (FPoly poly, float val)</code>	[Function]
<code>double eval_dpoly (DPoly poly, double val)</code>	[Function]
<code>void eval_mpfpoly (mpf_t eval, MPFPoly poly, mpf_t val)</code> 多項式 <i>poly</i> の一変数が実数 <i>val</i> 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 <i>eval</i> にその値を格納します。	[Function]
<code>float eval_diff_fpoly (FPoly poly, float val)</code>	[Function]
<code>double eval_diff_dpoly (DPoly poly, double val)</code>	[Function]
<code>void eval_diff_mpfpoly (mpf_t eval, MPFPoly poly, mpf_t val)</code> 多項式 <i>poly</i> の一変数が実数 <i>val</i> 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 <i>eval</i> にその値を格納します。	[Function]
<code>void ceval_fpoly (FCmplx eval, FPoly poly, FCmplx val)</code>	[Function]
<code>void ceval_dpoly (DCmplx eval, DPoly poly, DCmplx val)</code>	[Function]
<code>void ceval_mpfpoly (MPFCmplx eval, MPFPoly poly, MPFCmplx val)</code> 多項式 <i>poly</i> の一変数が複素数 <i>val</i> である時の値を計算し, <i>eval</i> にその値を格納します。	[Function]
<code>void ceval_diff_fpoly (FCmplx eval, FPoly poly, FCmplx val)</code>	[Function]
<code>void ceval_diff_dpoly (DCmplx eval, DPoly poly, DCmplx val)</code>	[Function]
<code>void ceval_diff_mpfpoly (MPFCmplx eval, MPFPoly poly, MPFCmplx val)</code> 多項式 <i>poly</i> の一変数が複素数 <i>val</i> である時, その導関数の値を計算し, <i>eval</i> に格納します。	[Function]

### 3.7 基本線型計算

<code>FVector init_fvector (long dim)</code>	[Function]
--	------------

<code>DVector</code> <code>init_dvector</code> ( <i>long dim</i> )	[Function]
<code>MPFVectorx</code> <code>init_mpfvector</code> ( <i>long dim</i> )	[Function]
ベクトルを初期化します。多倍長ベクトルは <code>set_bnc_default_prec</code> 関数で定義された精度で初期化されます。	
<code>MPFVector</code> <code>init2_mpfvector</code> ( <i>long dim, unsigned long prec</i> )	[Function]
多倍長ベクトルを精度指定で初期化します。	
<code>void</code> <code>free_fvector</code> ( <i>FVector vec</i> )	[Function]
<code>void</code> <code>free_dvector</code> ( <i>DVector vec</i> )	[Function]
<code>void</code> <code>free_mpfvector</code> ( <i>MPFVector vec</i> )	[Function]
ベクトルを解放します。	
<code>gfvi</code> ( <i>FVector vec, long index</i> )	[Macro]
<code>gdvi</code> ( <i>DVector vec, long index</i> )	[Macro]
<code>gmpfvi</code> ( <i>MPFVector vec, long index</i> )	[Macro]
ベクトル <code>vec</code> の <code>index</code> 番目の要素を取り出します。下の <code>get_...</code> 関数を短い名前でもクロにしたものです。	
<code>float</code> <code>get_fvector_i</code> ( <i>FVector vec, long index</i> )	[Function]
<code>double</code> <code>get_dvector_i</code> ( <i>DVector vec, long index</i> )	[Function]
<code>mpf_ptr</code> <code>get_mpfvector_i</code> ( <i>MPFVector vec, long index</i> )	[Function]
ベクトル <code>vec</code> の <code>index</code> 番目の要素を取り出します。	
<code>sfvi</code> ( <i>FVector vec, long index, float val</i> )	[Macro]
<code>sdvi</code> ( <i>DVector vec, long index, double val</i> )	[Macro]
<code>smpfvi</code> ( <i>MPFVector vec, long index, mpf_t val</i> )	[Macro]
<code>smpfvid</code> ( <i>MPFVector vec, long index, double val</i> )	[Macro]
<code>smpfvis</code> ( <i>MPFVector vec, long index, const char *str, int base</i> )	[Macro]
ベクトル <code>vec</code> の <code>index</code> 番目の要素に値 <code>val</code> を代入します。下の <code>set_...</code> 関数を短い名前でもクロにしたものです。	
<code>void</code> <code>set_fvector_i</code> ( <i>FVector vec, long index, float val</i> )	[Function]
<code>void</code> <code>set_dvector_i</code> ( <i>DVector vec, long index, double val</i> )	[Function]
<code>void</code> <code>set_mpfvector_i</code> ( <i>MPFVector vec, long index, mpf_t val</i> )	[Function]
<code>void</code> <code>set_mpfvector_i_d</code> ( <i>MPFVector vec, long index, double val</i> )	[Function]
<code>void</code> <code>set_mpfvector_i_str</code> ( <i>MPFVector vec, long index, const char *str, int base</i> )	[Function]
ベクトル <code>vec</code> の <code>index</code> 番目の要素に値 <code>val</code> を代入します。	
<code>unsigned long</code> <code>prec_mpfvector</code> ( <i>MPFVector vec</i> )	[Function]
多倍長ベクトルの精度を返します。	
<code>unsigned long</code> <code>minprec_mpfvector</code> ( <i>MPFVector vec</i> )	[Function]
<code>unsigned long</code> <code>maxprec_mpfvector</code> ( <i>MPFVector vec</i> )	[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。	

<code>void print_fvector (FVector vec)</code>	[Function]
<code>void print_dvector (DVector vec)</code>	[Function]
<code>void print_mpfvector (MPFVector vec)</code>	[Function]
ベクトルを標準出力に書き出します。	
<code>FMatrix init_fmatrix (long row_dim, long col_dim)</code>	[Function]
<code>DMatrix init_dmatrix (long row_dim, long col_dim)</code>	[Function]
<code>MPFMatrix init_mpfmatrix (long row_dim, long col_dim)</code>	[Function]
行列を初期化します。多倍長行列は, <code>set_bnc_default_prec</code> 関数で設定された精度で初期化されます。	
<code>MPFMatrix init2_mpfmatrix (long row_dim, long col_dim, unsigned long prec)</code>	[Function]
多倍長行列を精度指定で初期化します。	
<code>void free_fmatrix (FMatrix mat)</code>	[Function]
<code>void free_dmatrix (DMatrix mat)</code>	[Function]
<code>void free_mpfmatrix (MPFMatrix mat)</code>	[Function]
行列を解放します。	
<code>gfmij (FMatrix mat, long row_index, long col_index)</code>	[Macro]
<code>gdmij (DMatrix mat, long row_index, long col_index)</code>	[Macro]
<code>gmpfmij (MPFMatrix mat, long row_index, long col_index)</code>	[Macro]
行列 <code>mat</code> の <code>row_index</code> , <code>col_index</code> 番目の要素を取り出します。下の <code>get_...</code> 関数を短い名前でもクロにしたものです。	
<code>float get_fmatrix_ij (FMatrix mat, long row_index, long col_index)</code>	[Function]
<code>double get_dmatrix_ij (DMatrix mat, long row_index, long col_index)</code>	[Function]
<code>mpf_ptr get_mpfmatrix_ij (MPFMatrix mat, long row_index, long col_index)</code>	[Function]
行列 <code>mat</code> の <code>row_index</code> , <code>col_index</code> 番目の要素を取り出します。	
<code>sfmij (FMatrix mat, long row_index, long col_index, float val)</code>	[Macro]
<code>sdmij (DMatrix mat, long row_index, long col_index, double val)</code>	[Macro]
<code>smpfmij (MPFMatrix mat, long row_index, long col_index, mpf_t val)</code>	[Macro]
<code>smpfmijd (MPFMatrix mat, long row_index, long col_index, double val)</code>	[Macro]
<code>smpfmij_s (MPFMatrix mat, long row_index, long col_index, const char *str, int base)</code>	[Macro]
行列 <code>mat</code> の <code>row_index</code> , <code>col_index</code> 番目の要素に値 <code>val</code> を代入します。下の <code>set_...</code> 関数を短い名前でもクロにしたものです。	
<code>void set_fmatrix_ij (FMatrix mat, long row_index, long col_index, float val)</code>	[Function]
<code>void set_dmatrix_ij (DMatrix mat, long row_index, long col_index, double val)</code>	[Function]

<code>void set_mpfmatrix_ij (MPFMatrix mat, long row_index, long col_index, mpf_t val)</code>	[Function]
<code>void set_mpfmatrix_ij_d (MPFMatrix mat, long row_index, long col_index, double val)</code>	[Function]
<code>void set_mpfmatrix_ij_str (MPFMatrix mat, long row_index, long col_index, const char *str, int base)</code> 行列 <i>mat</i> の <i>row_index</i> , <i>col_index</i> 番目の要素に値 <i>val</i> を代入します。	[Function]
<code>unsigned long prec_mpfmatrix (MPFMatrix mat)</code> 多倍長行列の精度を返します。	[Function]
<code>unsigned long minprec_mpfmatrix (MPFMatrix mat)</code>	[Function]
<code>unsigned long maxprec_mpfmatrix (MPFMatrix mat)</code> 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。	[Function]
<code>void print_fmatrix (FMatrix mat)</code>	[Function]
<code>void print_dmatrix (DMatrix mat)</code>	[Function]
<code>void print_mpfmatrix (MPFMatrix mat)</code> 行列を標準出力に表示します。	[Function]
<code>void add_fvector (FVector rvec, FVector vec1, FVector vec2)</code>	[Function]
<code>void add_dvector (DVector rvec, DVector vec1, DVector vec2)</code>	[Function]
<code>void add_mpfvector (MPFVector rvec, MPFVector vec1, MPFVector vec2)</code> ベクトルの和を計算します。	[Function]
<code>void sub_fvector (FVector rvec, FVector vec1, FVector vec2)</code>	[Function]
<code>void sub_dvector (DVector rvec, DVector vec1, DVector vec2)</code>	[Function]
<code>void sub_mpfvector (MPFVector rvec, MPFVector vec1, MPFVector vec2)</code> ベクトルの差を計算します。	[Function]
<code>void add2_fvector (FVector rvec, FVector vec)</code>	[Function]
<code>void add2_dvector (DVector rvec, DVector vec)</code>	[Function]
<code>void add2_mpfvector (MPFVector rvec, MPFVector vec)</code> <i>rvec</i> + <i>vec</i> を計算し, 結果を <i>rvec</i> に代入します。	[Function]
<code>void sub2_fvector (FVector rvec, FVector vec)</code>	[Function]
<code>void sub2_dvector (DVector rvec, DVector vec)</code>	[Function]
<code>void sub2_mpfvector (MPFVector rvec, MPFVector vec)</code> <i>rvec</i> - <i>vec</i> を計算し, 結果を <i>rvec</i> に代入します。	[Function]
<code>void cmul_fvector (FVector rvec, float opt, FVector vec)</code>	[Function]
<code>void cmul_dvector (DVector rvec, double opt, DVector vec)</code>	[Function]
<code>void cmul_mpfvector (MPFVector rvec, mpf_t opt, MPFVector vec)</code> <i>vec</i> のスカラー <i>opt</i> 倍を計算します。	[Function]

<code>void cmul2_fvector (FVector rvec, float opt)</code>	[Function]
<code>void cmul2_dvector (DVector rvec, double opt)</code>	[Function]
<code>void cmul2_mpfvector (MPFVector rvec, mpf_t opt)</code> rvec のスカラー opt 倍を計算し, 結果を rvec に代入します。	[Function]
<code>float ip_fvector (FVector vec1, FVector vec2)</code>	[Function]
<code>double ip_dvector (DVector vec1, DVector vec2)</code>	[Function]
<code>void ip_mpfvector (mpf_t ropt, MPFVector vec1, MPFVector vec2)</code> 内積を計算します。	[Function]
<code>float norm1_fvector (FVector vec)</code>	[Function]
<code>float norm2_fvector (FVector vec)</code>	[Function]
<code>float normi_fvector (FVector vec)</code>	[Function]
<code>double norm1_dvector (DVector vec)</code>	[Function]
<code>double norm2_dvector (DVector vec)</code>	[Function]
<code>double normi_dvector (DVector vec)</code>	[Function]
<code>void norm1_mpfvector (mpf_t ropt, MPFVector vec)</code>	[Function]
<code>void norm2_mpfvector (mpf_t ropt, MPFVector vec)</code>	[Function]
<code>void normi_mpfvector (mpf_t ropt, MPFVector vec)</code> vec のノルムをそれぞれ計算します。	[Function]
<code>void subst_fvector (FVector rvec, FVector vec)</code>	[Function]
<code>void subst_dvector (DVector rvec, DVector vec)</code>	[Function]
<code>void subst_mpfvector (MPFVector rvec, MPFVector vec)</code> ベクトル vec をベクトル rvec に代入します。	[Function]
<code>void add_fmatrix (FMatrix rmat, FMatrix mat1, FMatrix mat2)</code>	[Function]
<code>void add_dmatrix (DMatrix rmat, DMatrix mat1, DMatrix mat2)</code>	[Function]
<code>void add_mpfmatrix (MPFMatrix rmat, MPFMatrix mat1, MPFMatrix mat2)</code> 行列の和を計算します。	[Function]
<code>void sub_fmatrix (FMatrix rmat, FMatrix mat1, FMatrix mat2)</code>	[Function]
<code>void sub_dmatrix (DMatrix rmat, DMatrix mat1, DMatrix mat2)</code>	[Function]
<code>void sub_mpfmatrix (MPFMatrix rmat, MPFMatrix mat1, MPFMatrix mat2)</code> mat1 - mat2 を計算し, rmat に代入します。	[Function]
<code>void mul_fmatrix (FMatrix rmat, FMatrix mat1, FMatrix mat2)</code>	[Function]
<code>void mul_dmatrix (DMatrix rmat, DMatrix mat1, DMatrix mat2)</code>	[Function]
<code>void mul_mpfmatrix (MPFMatrix rmat, MPFMatrix mat1, MPFMatrix mat2)</code> mat1 * mat2 を計算し, rmat に代入します。	[Function]
<code>void subst_fmatrix (FMatrix rmat, FMatrix mat)</code>	[Function]
<code>void subst_dmatrix (DMatrix rmat, DMatrix mat)</code>	[Function]
<code>void subst_mpfmatrix (MPFMatrix rmat, MPFMatrix mat)</code> 行列 mat を行列 rmat に代入します。	[Function]

```

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, [Function]
    MPFVector vec)
    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 [Function]
    mat の転置を rmat に代入します。

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 [Function]
    iarray[])
int SolveDLSP (DVector rvec, DMatrix mat, DVector vec, long [Function]
    iarray[])

```

int SolveMPFLSP (*MPFVector rvec*, *MPFMatrix mat*, *MPFVector vec*, [Function]  
*long iarray[]*)

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),      [Function]
               float (* dfunc)(float op), long maxtimes, float abs_eps, float rel_eps)
long dnewton_1 (double *ans, double x_init, double (*          [Function]
               func)(double op), double (* dfunc)(double op), long maxtimes, double
               abs_eps, double rel_eps)
long mpf_newton_1 (mpf_t *ans, mpf_t x_init, void (* func)(mpf_t      [Function]
                  rop, mpf_t op), void (* dfunc)(mpf_t rop, mpf_t op), long maxtimes, mpf_t
                  abs_eps, mpf_t rel_eps)
1次元のNewton法を実行します。

long fsnewton (FVector ans, FVector x_init, void (* func)(FVector      [Function]
                rop, FVector op), void (* jfunc)(FMatrix rop, FVector op), long maxtimes,
                float abs_eps, float rel_eps)
long dsnewton (DVector ans, DVector x_init, void (* func)(DVector      [Function]
                rop, DVector op), void (* jfunc)(DMatrix rop, DVector op), long
                maxtimes, double abs_eps, double rel_eps)
long mpf_snewton( MPFVector ans, MPFVector x_init, void (*          [Function]
                  func)(MPFVector rop, MPFVector op), void (* jfunc)(MPFMatrix rop,
                  MPFVector op), long maxtimes, mpf_t abs_eps, mpf_t rel_eps)
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),    [Function]
                float (* dfunc)(float op), long maxtimes, float abs_eps, float rel_eps)
long dsnewton_1 (double *, double, double (* func)(double op),        [Function]
                double (* dfunc)(double op), long, double, double)
long mpf_snewton_1 (mpf_t ans, mpf_t x_init, void (* func)(mpf_t      [Function]
                    rop, mpf_t op), void (* dfunc)(mpf_t rop, mpf_t op), long maxtimes, mpf_t
                    abs_eps, mpf_t rel_eps)
1次元の簡易Newton法を実行します。

```

### 3.11 常微分方程式の初期値問題 (単段法)

常微分方程式の初期値問題を単段法で解くための関数を呼び出す時には必ず `bnc.h` に加えて `bncode.h` を呼び出して下さい。

```

#include "bnc.h"
#include "bncode.h"

```

加えて、`libbncode.a` もリンクして下さい。

常微分方程式の初期値問題は

```

dy/dx = f(x, y) <---> void f(Vector ret_val, x, Vector y)
y(x0) = y0

```

という形で与えられるものとします。陰的解法で必要となる Jacobi 行列は

```

df/dx <---> void df(Matrix ret_val, x, Vector y)

```

という形で与えられるものとします。中間出力が必要な複数ステップの計算の途中結果はファイルポインタ `fp` で指定された出力先に表示されます。

### 3.11.1 陽的 Euler 法

`void feuler_1step(FVector y, float x0, FVector y0, float h, void(* func)(FVector y, float x0, FVector y0), FVector ytmp)` [Function]

`void deuler_1step(DVector y, double x0, DVector y0, double h, void(* func)(DVector y, double x0, DVector y0), DVector ytmp)` [Function]

`void mpf_euler_1step(MPFVector y, mpf_t x0, MPFVector y0, mpf_t h, void(* func)(MPFVector y, mpf_t x0, MPFVector y0), MPFVector ytmp)` [Function]

陽的 Euler 法の 1 ステップ分の計算を行います。

`void feuler_fs(FILE *fp, float x, FVector y, float x0, FVector y0, long int div_num, void(* func)(FVector y, float x0, FVector y0))` [Function]

`void deuler_fs(FILE *fp, double x, DVector y, double x0, DVector y0, long int div_num, void(* func)(DVector y, double x0, DVector y0))` [Function]

`void mpf_euler_fs(FILE *fp, mpf_t x, MPFVector y, mpf_t x0, MPFVector y0, long int div_num, void(* func)(MPFVector y, mpf_t x0, MPFVector y0))` [Function]

陽的 Euler 法と固定ステップ  $h = (x - x0) / div\_num$  を使って  $y(x)$  の近似値  $y$  を計算します。

`void feuler_ex(FILE *fp, float x, FVector y, float x0, FVector y0, float max_h, void(* func)(FVector y, float x0, FVector y0), float rtol, float atol)` [Function]

`void deuler_ex(FILE *fp, double x, DVector y, double x0, DVector y0, double max_h, void(* func)(DVector y, double x0, DVector y0), double rtol, double atol)` [Function]

`void mpf_euler_ex(FILE *fp, mpf_t x, MPFVector y, mpf_t x0, MPFVector y0, mpf_t max_h, void(* func)(MPFVector y, mpf_t x0, MPFVector y0), mpf_t rtol, mpf_t atol)` [Function]

陽的 Euler 法と補外法に基づく局所誤差制御 ( $||y\_next - y\_old|| \leq rtol * ||y\_old|| + atol$ ) を用いて  $y(x)$  の近似値  $y$  を計算します。

### 3.11.2 陽的 Runge-Kutta 法

`void get_derk_coef(unsigned long coef_type, DVector coef_c, DMatrix coef_a, DVector coef_w)` [Function]

`void get_mpf_erk_coef(unsigned long coef_type, unsigned long prec, MPFVector coef_c, MPFMatrix coef_a, MPFVector coef_w)` [Function]

陽的 Runge-Kutta(RK) 法の係数を与えます。coef\_type には

- `ERK_OPTIMAL_2_2`: 2 段 2 次最適公式
- `ERK_CLASSICAL_4_4`: 4 段 4 次古典公式
- `ERK_BUTCHER1_7_6`: 7 段 6 次 Butcher 公式

が使用可能です。



```
long int dirk_1step(DVector y,double x0,DVector y0,double h,void(* [Function]
func)(DVector y,double x0,DVector y0), void(* jfunc)(DMatrix jacobi,double
x0,DVector y0), DVector coef_c,DMatrix coef_a,DVector coef_w,double
abs_tol,double rel_tol,long int maxtimes)
```

```
long int mpf_irk_1step(MPFVector y,mpf_t x0,MPFVector y0,mpf_t [Function]
h,void(* func)(MPFVector y,mpf_t x0,MPFVector y0), void(*
jfunc)(MPFMatrix jacobi,mpf_t x0,MPFVector y0), MPFVector
coef_c,MPFMatrix coef_a,MPFVector coef_w,mpf_t abs_tol,mpf_t
rel_tol,long int maxtimes)
```

陰的 RK 法の 1 ステップ分の計算を行います。maxtimes は内部反復 (簡易 Newton 法) の最大反復回数の指定です。

```
void dirk_fs(FILE *fp,double x,DVector y,double x0,DVector y0,long [Function]
int div_num,void(* func)(DVector y,double x0,DVector y0), void(*
jfunc)(DMatrix ymat,double x0,DVector y0), unsigned long coef_type,double
rtol,double atol,long int maxtimes)
```

```
void mpf_irk_fs(FILE *fp,mpf_t x,MPFVector y,mpf_t x0,MPFVector [Function]
y0,long int div_num,void(* func)(MPFVector y,mpf_t x0,MPFVector y0),
void(* jfunc)(MPFMatrix ymat,mpf_t x0,MPFVector y0), unsigned long
coef_type,mpf_t rtol,mpf_t atol,long int maxtimes)
```

陰的 RK 法と固定ステップ  $h = (x - x_0) / \text{div\_num}$  を使って  $y(x)$  の近似値  $y$  を計算します。

```
void dirk_ex(FILE *fp,double x,DVector y,double x0,DVector [Function]
y0,double max_h,void(* func)(DVector y,double x0,DVector y0), void(*
jfunc)(DMatrix ymat,double x0,DVector y0), unsigned long coef_type,double
rtol,double atol,long int maxtimes)
```

```
void mpf_irk_ex(FILE *fp,mpf_t x,MPFVector y,mpf_t x0,MPFVector [Function]
y0,mpf_t max_h,void(* func)(MPFVector y,mpf_t x0,MPFVector y0), void(*
jfunc)(MPFMatrix ymat,mpf_t x0,MPFVector y0), unsigned long
coef_type,mpf_t rtol,mpf_t atol,long int maxtimes)
```

陰的 RK 法と補外法に基づく局所誤差制御 ( $\|y_{\text{next}} - y_{\text{old}}\| \leq \text{rtol} * \|y_{\text{old}}\| + \text{atol}$ ) を用いて  $y(x)$  の近似値  $y$  を計算します。

### 3.11.4 補外法

調和級数 (`~_harmonic`) と Romberg 級数 (`~_nim`) に基づく補外法で任意段数 (次数) の近似値を計算します。

```
void init_dex_harmonic(long int stage, long int dimension) [Function]
```

```
void init_dex_nim(long int stage, long int dimension) [Function]
```

```
void init_mpf_ex_harmonic(long int stage, long int dimension, [Function]
unsigned long prec)
```

void init\_mpf\_ex\_nim(long int stage, long int dimension, unsigned long prec) 補外法の計算に必要な一時変数を宣言します。stage は最大段数の設定です。

```
void clear_dex_harmonic(void) [Function]
```

```
void clear_dex_nim(void) [Function]
```

```
void clear_mpf_ex_harmonic(void) [Function]
```

```

void clear_mpf_ex_nim(void) [Function]
    補外法の計算に使用した一時変数を消去します

int dex_harmonic_1step(DVector y, double x0, DVector y0, double h, void (* func)(DVector, double, DVector), double rtol, double atol) [Function]
int dex_nim_1step(DVector y, double x0, DVector y0, double h, void (* func)(DVector, double, DVector), double rtol, double atol) [Function]
int mpf_ex_harmonic_1step(MPFVector y, mpf_t x0, MPFVector y0, mpf_t h, void (* func)(MPFVector, mpf_t, MPFVector), mpf_t rtol, mpf_t atol) [Function]
int mpf_ex_nim_1step(MPFVector y, mpf_t x0, MPFVector y0, mpf_t h, void (* func)(MPFVector, mpf_t, MPFVector), mpf_t rtol, mpf_t atol) [Function]
    補外法の 1 ステップ分の計算を行います。局所誤差制御 ( $\|y_{\text{next}} - y_{\text{old}}\| \leq \text{rtol} * \|y_{\text{old}}\| + \text{atol}$ ) を用います。

void dex_harmonic_fs(FILE *fp, double x, DVector y, double x0, DVector y0, long int div_num, void (* func)(DVector, double, DVector), double rtol, double atol, long int stage) [Function]
void dex_nim_fs(FILE *fp, double x, DVector y, double x0, DVector y0, long int div_num, void (* func)(DVector, double, DVector), double rtol, double atol, long int stage) [Function]
void mpf_ex_harmonic_fs(FILE *fp, mpf_t x, MPFVector y, mpf_t x0, MPFVector y0, long int div_num, void (* func)(MPFVector, mpf_t, MPFVector), mpf_t rtol, mpf_t atol, long int stage) [Function]
void mpf_ex_nim_fs(FILE *fp, mpf_t x, MPFVector y, mpf_t x0, MPFVector y0, long int div_num, void (* func)(MPFVector, mpf_t, MPFVector), mpf_t rtol, mpf_t atol, long int stage) [Function]
    局所誤差制御を行う補外法と固定ステップ  $h (= (x - x0) / \text{div\_num})$  を使って  $y(x)$  の近似値  $y$  を計算します。。

void dex_harmonic(FILE *fp, double x, DVector y, double x0, DVector y0, double max_h, void (* func)(DVector, double, DVector), double rtol, double atol, long int stage, void (* ansfunc)(DVector, double)) [Function]
void dex_nim(FILE *fp, double x, DVector y, double x0, DVector y0, double max_h, void (* func)(DVector, double, DVector), double rtol, double atol, long int stage, void (* ansfunc)(DVector, double)) [Function]
void mpf_ex_harmonic(FILE *fp, mpf_t x, MPFVector y, mpf_t x0, MPFVector y0, mpf_t max_h, void (* func)(MPFVector, mpf_t, MPFVector), mpf_t rtol, mpf_t atol, long int stage, void (* ansfunc)(MPFVector, mpf_t), long int itimes_iv) [Function]
void mpf_ex_nim(FILE *fp, mpf_t x, MPFVector y, mpf_t x0, MPFVector y0, mpf_t max_h, void (* func)(MPFVector, mpf_t, MPFVector), mpf_t rtol, mpf_t atol, long int stage, void (* ansfunc)(MPFVector, mpf_t), long int itimes_iv) [Function]
    [テスト用関数] 局所誤差制御を行う補外法を用いて  $y(x)$  の近似値  $y$  を計算します。。  

    ansfunc 関数に真の解を設定すると誤差の計算も行えます。

```

### 3.12 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  $abs\_eps + rel\_eps \times answer\_array[i]$  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。

### 3.13 疎行列

使用時には必ず `bncsparse.h` をインクルードした上で `libbncsparse.a` をリンクして下さい。

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, nzero_row_dim[] に行ごとの非零成分数を
    セットします。

void print_drsmatrix(DRSMatrix mat) [Function]
void print_mpfmatrix(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_mpfmatrix_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 [Function]
    **ptr_nzero_col_dim, long *ptr_nzero_total_num, const char *fname)
int get_vars_mpfmatrix_fname(long *ptr_row_dim, long [Function]
    **ptr_nzero_col_dim, long *ptr_nzero_total_num, const char *fname)
    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 [Function]
    *fname)
    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 [Function]
    vec)
int mul_mpfmatrix_mpfvec(MPFVector ret, MPFRSMatrix mat, [Function]
    MPFVector vec)
    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 [Function]
    vec)
int mul_mpfmatrixt_mpfvec(MPFVector ret, MPFRSMatrix mat, [Function]
    MPFVector vec)
    疎行列 mat の転置とベクトル vec との積を計算し, ret に返します。

double dpower_rsmatrix(DVector evec, DRSMatrix mat, double [Function]
    reps, double aeps, long max_times)
void mpfpower_rsmatrix(mpf_t max_eig, MPFVector evec, [Function]
    MPFRSMatrix mat, mpf_t reps, mpf_t aeps, long max_times)
    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/test_efunc.c'
    基本関数のテストプログラム
'test/test_complex.c'
    複素数のテストプログラム
'test/test_poly.c'
    多項式のテストプログラム
'test/test_linear.c'
    基本線型計算のテストプログラム
'test/test_lu.c'
    LU 分解による連立一次方程式の求解
'test/test_cg.c'
    CG 法による連立一次方程式の求解
'test/test_newton.c'
    Newton 法による非線型方程式の求解
'ode/test_ode.c'
    Runge-Kutta 法による常微分方程式の初期値問題の求解
'test/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()
{
#ifdef 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<sup>1</sup>。

```
    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);
```

<sup>1</sup> 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、ちょっと苦勞すれば出来そうな気がします。… どなたかやりませんか?

### 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_dcplx();
dcb = init_dcplx();
dcc = init_dcplx();

```

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_dcplx(dca, 1.0);
set_image_dcplx(dca, 2.0);
set_real_dcplx(dcb, 3.0);
set_image_dcplx(dcb, 4.0);
set_real_dcplx(dcc, (double)rand());
set_image_dcplx(dcc, (double)rand());

/* print */
printf("dca = "); print_dcplx(dca);
printf("dcb = "); print_dcplx(dcb);
printf("dcc = "); print_dcplx(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_dcplx(dcc, dca, dcb);
printf("dca + dcb = "); print_dcplx(dcc);
sub_dcplx(dcc, dca, dcb);
printf("dca - dcb = "); print_dcplx(dcc);
add2_dcplx(dcc, dcb);
printf("dca - dcb + dcb = "); print_dcplx(dcc);

mul_dcplx(dcc, dca, dcb);
printf("dca * dcb = "); print_dcplx(dcc);
div_dcplx(dcc, dca, dcb);
printf("dca / dcb = "); print_dcplx(dcc);
mul2_dcplx(dcc, dcb);
printf("dca / dcb * dcb = "); print_dcplx(dcc);

printf("~dca = ");
conj_dcplx(dcc, dca); print_dcplx(dcc);
printf("|dca| = %25.17e\n", abs_dcplx(dca));
printf("exp(i*dca) = "); i
exp_dcplx(dcc, abs_dcplx(dca)); print_dcplx(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_dcplx(dca);
free_dcplx(dcb);
free_dcplx(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);

```

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

```

/* basic arithmetic */
add_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_dcplx();
dcret = init_dcplx();
set_real_dcplx(dca, 1.0);
set_image_dcplx(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: 0(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_dcplx(dcret);
printf("dpa'(1+i1) = "); ceval_diff_dpoly(dcret, dpa, dca);
print_dcplx(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, mpc_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<sup>2</sup>。

```

mpf_init2(mpf_x, 128);

```

---

<sup>2</sup> それで何度失敗したことが …。

```

mpf_init2(mpf_ret, 128);

printf("mpf_pa: 0(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 , DLUdecompP 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 = SolveDLS(dx, da, db, row_ch);
ret_d = SolveDLS(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, MPFLUdecompP 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 = MPFFCG(mpfx, mpfa, mpfb, reps, aeps, DIM * 5);
itimes_mpf2 = MPFFCG(mpfx2, mpfa2, mpfb2, reps2, aeps2, DIM * 5);
itimes_mpf3 = MPFFCG(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 Runge-Kutta 法

使用時には必ず *bncode.h* をインクルードした上で *libbncode.a* をリンクして下さい。

### 4.8.1 倍精度計算の場合

常微分方程式の定義 (*ddf*) と初期値の設定を行います。

```

void ddf(DVector y, double x0, DVector y0)

/* y' = y */
subst_dvector(y, y0);

return;

void initial_dvalue(double *lf_initx, double *lf_maxx, DVector lf_inity)

long int i;

```

```

/* x0 := 0 */
/* y := [1, ..., 1]^T */
*lf_initx = 0.0;
*lf_maxx = 1.0;
for(i = 0; i < lf_inity->dim; i++)
set_dvector_i(lf_inity, i, 1.0);

```

main 関数では次のように関数を呼び出します。

```

DVector dy, init_dy, lf_dtmp;
long int div_num;
double dx, init_dx, dstepsize, dabs_tol, drel_tol;

dy = init_dvector(DIM);
lf_dtmp = init_dvector(DIM);
init_dy = init_dvector(DIM);
initial_dvalue(&init_dx, &dx, init_dy);

dstepsize = 0.5;
dabs_tol = 0.0;
drel_tol = 1.0e-15;

/* 固定ステップ */
div_num = 128; // 128 分割
derk_fs(NULL, dx, dy, init_dx, init_dy, div_num, ddf, ERK_BUTCHER1_7_6);
/* 局所誤差制御付き */
derk_ex(NULL, dx, dy, init_dx, init_dy, dstepsize, ddf, ERK_BUTCHER1_7_6,
drel_tol, dabs_tol);

```

#### 4.8.2 多倍長精度の場合

常微分方程式の定義 (*ddf*) と初期値の設定を行います。

```

void df(MPFVector y, mpf_t x0, MPFVector y0)

/* y' = y */
subst_mpfvector(y, y0);

return;

void initial_value(mpf_t lf_initx, mpf_t lf_maxx, MPFVector lf_inity)

long int i;

/* x0 := 0 */
/* y := [1, ..., 1]^T */
mpf_init_set_str(lf_initx, "0.0", 10);

```

```
mpf_init_set_str(lf_maxx, "1.0", 10);
for(i = 0; i < lf_inity->dim; i++)
  set_mpfvector_i_str(lf_inity, i, "1.0", 10);
```

main 関数では次のように関数を呼び出します。

```
MPFVector y, init_y, lf_tmp;
mpf_t x, init_x, stepsize, abs_tol, rel_tol;

set_bnc_default_prec(256);

/* x := 1 */
mpf_init(init_x);
mpf_init(x);

y = init_mpfvector(DIM);
lf_tmp = init_mpfvector(DIM);
init_y = init_mpfvector(DIM);
initial_value(init_x, x, init_y);

mpf_init_set_d(stepsize, dstepsize);
mpf_init_set_d(abs_tol, dabs_tol);
mpf_init_set_d(rel_tol, drel_tol);

/* 固定ステップ */
div_num = 128; // 128 分割
mpf_erk_fs(NULL, x, y, init_x, init_y, div_num, df, ERK_BUTCHER1_7_6);

/* 局所誤差制御付き */
mpf_erk_ex(NULL, x, y, init_x, init_y, stepsize, df, ERK_BUTCHER1_7_6,
  el_tol, abs_tol);
```

## 4.9 DKA 法

### 4.9.1 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。

#### 4.9.2 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。

- "GNU MP: The GNU Multiple Precision Arithmetic Library", <http://gmpilib.org/>.
- Robert J. Chassel, "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

<b>1</b>	<b>BNC とは？</b> .....	<b>1</b>
<b>2</b>	<b>BNCpack のインストール</b> .....	<b>2</b>
2.1	IEEE754 単精度・倍精度のみ利用する場合 .....	2
2.2	GMP の mpf_t 型のみを利用する場合 .....	2
2.3	MPFR パッケージを利用する場合 .....	3
<b>3</b>	<b>機能一覧</b> .....	<b>5</b>
3.1	基本データ型定義 .....	5
3.1.1	多倍長浮動小数点数を含むデータ型 .....	5
3.1.2	複素数型 .....	5
3.1.3	多項式型 .....	5
3.1.4	ベクトル型 .....	5
3.1.5	一般行列型 .....	6
3.1.6	スタック .....	6
3.1.7	配列 .....	7
3.1.8	疎行列 .....	7
3.2	基本関数 .....	8
3.3	スタック .....	10
3.4	複素数 .....	10
3.5	配列 .....	13
3.6	多項式 .....	14
3.7	基本線型計算 .....	16
3.8	LU 分解 .....	21
3.9	Conjugate-Gradient 法 .....	22
3.10	非線型方程式 .....	22
3.11	常微分方程式の初期値問題 (単段法) .....	23
3.11.1	陽的 Euler 法 .....	24
3.11.2	陽的 Runge-Kutta 法 .....	24
3.11.3	陰的 Runge-Kutta 法 .....	25
3.11.4	補外法 .....	26
3.12	DKA 法 .....	28
3.13	疎行列 .....	28
<b>4</b>	<b>BNCpack を使ったサンプルプログラム集</b> .....	<b>31</b>
4.1	基本関数 .....	32
4.2	複素数 .....	35
4.2.1	倍精度の場合 .....	35
4.2.2	多倍長の場合 .....	36
4.3	多項式 .....	37
4.3.1	倍精度の場合 .....	38
4.3.2	多倍長の場合 .....	39

4.4	線型計算	41
4.5	LU分解	41
4.5.1	倍精度の場合	41
4.5.2	多倍長の場合	42
4.6	Conjugate-Gradient 法	44
4.6.1	倍精度の場合	44
4.6.2	多倍長の場合	45
4.7	Newton 法	47
4.7.1	倍精度の場合	47
4.7.2	多倍長の場合	47
4.8	Runge-Kutta 法	48
4.8.1	倍精度計算の場合	48
4.8.2	多倍長精度の場合	49
4.9	DKA 法	50
4.9.1	倍精度の場合	50
4.9.2	多倍長の場合	51
<b>5</b>	<b>今後の予定 (あてにしないように)</b>	<b>53</b>
	謝辞・参考文献	54
	索引	55