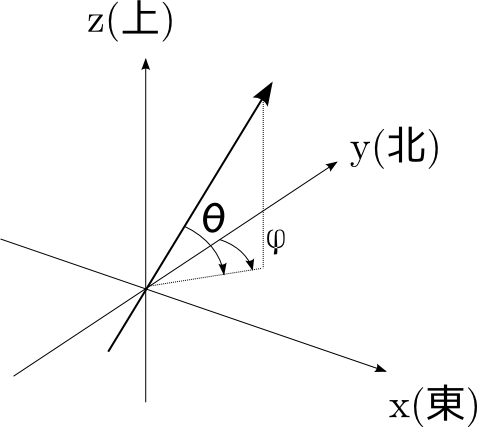
1. 座標系

構造体 grid で使用する座標系は以下の図のとおりである。



東向きに x 座標、北向に y 座標、上向きにz 座標を取り、極座標 (φ,θ) は図のとおりである。

また φは -declination, θは -inclination である。

2. datalistの仕様

datalist

prev

next

data

**struct** \_datalist

{

datapoint \*data; // datapoint = (position \*pos, double f)

datalist \*prev;

datalist \*next;

};

datalist は双方向リストで、メンバに datapoint \*data を持つ。

// datalistのメモリ確保

datalist\* datalist\_alloc ();

// datalistオブジェクトの作成(datalist\_allocと同値)

datalist\* datalist\_new ();

// datalist \*listの先頭要素ポインタ取得

datalist\* datalist\_begin (datalist \*list);

// datalist \*listの末尾要素ポインタ取得

datalist\* datalist\_end (datalist \*list);

// datalist \*listの位置に要素を挿入

**void** datalist\_insert (datalist \*list, datapoint \*data);

// datalistの末尾に要素追加

**void** datalist\_push\_back (datalist \*list, datapoint \*data);

// datalistが空(begin==end)かを判定

boolean datalist\_is\_empty (datalist \*list);

// 先頭(=0)からindex目要素のポインタ取得

datalist\* datalist\_at (datalist \*list, **int** index);

// 次の要素(next)のポインタ取得

datalist\* datalist\_get\_next (datalist \*list);

// datalist \*listの要素数

**int** datalist\_size (datalist \*list);

// datapoint\* list->dataを取得

datapoint\* datalist\_get\_datapoint (datalist \*list);

// position\* list->data->posを取得

position\* datalist\_get\_position (datalist \*list);

// list->data->pos->(x,y,z)を取得

**double** datalist\_get\_x (datalist \*list);

**double** datalist\_get\_y (datalist \*list);

**double** datalist\_get\_z (datalist \*list);

// list->data->fを取得

**double** datalist\_get\_f (datalist \*list);

// list->data->pos->(x,y,z)に代入

**void** datalist\_set\_x (datalist \*list, **double** x);

**void** datalist\_set\_y (datalist \*list, **double** y);

**void** datalist\_set\_z (datalist \*list, **double** z);

// list->data->fに代入

**void** datalist\_set\_f (datalist \*list, **double** f);

// データfを要素に持つベクトルを返す

c\_vector\* datalist\_get\_data\_vector (datalist \*list);

// datalistのメモリ解放

**void** datalist\_free (datalist \*list);

2.1 構造体の確保

datalist構造体のメモリ確保には

datalist \*datalist\_alloc ()

又は

datalist \*datalist\_new ()

を用いる。これらにより、空のdatalistオブジェクトが作成される。

item->data = NULL;

item->prev = NULL;

item->next = NULL;

2.2 リスト構造

datalistの先頭及び末尾のノード位置は

先頭 : datalist \*datalist\_begin (datalist \*list)

末尾 : datalist \*datalist\_end (datalist \*list)

で取得できる。ここで、先頭位置は list->prev = NULL の位置、末尾は list->next = NULLの位置で、何も登録されていないlistの場合、先頭と末尾のノード位置は同じになる。

datalist\_begin (list) = datalist\_end (list);

リストが空かは

boolean datalist\_is\_empty (list)

で確かめられるが、これはbeginとendが一致するかを見ている。空でないリストの場合、末尾 end は end->data = NULL となる。

begin

prev

next

data

datalist

prev

next

data

end

prev

next

2.3 アイテムの挿入

datalist \*list に新たにアイテムを登録するには

**void** datalist\_insert (datalist \*list, datapoint \*data)

を用いる。これにより、list->next にdataを持つdatalistアイテムが登録される。

リストの末尾に登録するには

**void** datalist\_push\_back (datalist \*list, datapoint \*data)

を用いる。これにより

datalist \*p = datalist\_end (list);

p->data = datapoint\_new (point...); // end のdatapoint登録

p->next = datalist\_alloc (); // end->nextのアロケート

next

data

end

prev

next

data

next

data

prev

data

next

prev

next

end

p->next->prev = p; // end->nextのprevにendを登録

の操作が行われ、endポイントが一つ後ろに移動する。

next

data

prev

data

next

prev

next

end

2.4 ユーティリティ

datalistの要素数は

int datalist\_size (datalist \*list)

で得られる。但し末尾のノード(datapointを持たない)は要素数にカウントされない。

3.　datapointの仕様

**typedef** **struct** {

position \*pos; // position = (x,y,z)

**double** f; // データ

} \_datapoint;

**typedef** \_datapoint datapoint;

// datapointのメモリ確保

datapoint\* datapoint\_alloc ();

// datapointオブジェクトの作成(data=(pos(x,y,z),f)

datapoint\* datapoint\_new (**double** x, **double** y, **double** z, **double** f);

// datapointのコピー(origをdestにコピー)

**void** datapoint\_copy (datapoint \*dest, datapoint \*orig);

// datapoint \*dataに代入

**void** datapoint\_set (datapoint \*data, **double** x, **double** y, **double** z, **double** f);

// datapointのメモリ解放

**void** datapoint\_free (datapoint \*p);

3.1 構造体の確保

datapoint構造体のメモリ確保には

datapoint \*datapoint\_alloc ()

を用いる。これにより

data->pos = NULL;

data->f = 0 (暗黙)

なる構造体が作成される。(pos=(x,y,z),f)なるオブジェクトを作成するには

datapoint \*datapoint\_new (double x, double y, double z, double f)

を用いる。

3.2 ユーティリティ

datapointをコピーするには

void datapoint\_copy (datapoint \*dest, datapoint \*orig)

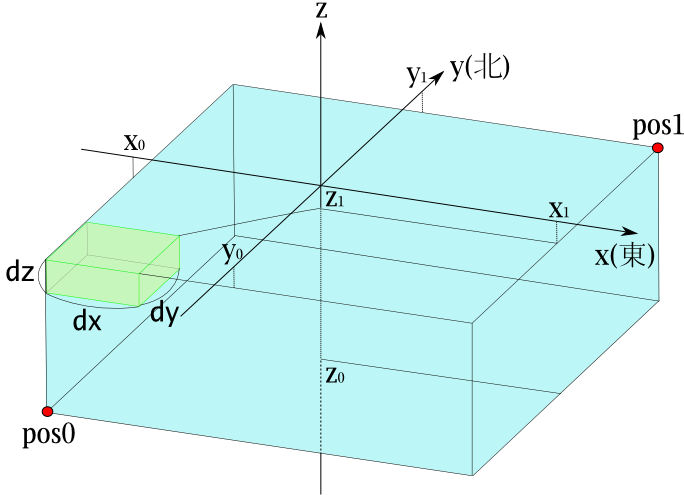
を用いる。これにより、destにorigがコピーされる。

要素にデータを代入するには

void datapoint\_set (datapoint \*data, double x, double y, double z, double f)

を用いる。

4.　グリッド



**typedef** **struct**

{

boolean size\_specified; // グリッド数設定フラグ

boolean range\_specified; // 範囲設定フラグ

boolean dimension\_specified; // ディメンジョン設定フラグ

boolean z\_constant; // z座標の一定値指定フラグ

boolean z\_array; // z座標のアレイ指定フラグ

**int** n; // グリッドの総数

**int** nh; // 水平面内のグリッド数

**int** nx; // 東西グリッド数

**int** ny; // 南北

**int** nz; // 上下

position \*pos0; // 最下層南西(左下)端座標

position \*pos1; // 最上層北東(右上)端座標

**double** \*z; // 最上層の各グリッド**上端**のz座標

dimension \*dim; // グリッドのディメンジョン

} \_grid;

**typedef** \_grid grid;

座標系は上の右手系を用いる。x座標は西→東、y座標は南→北、z座標は下→上の向きに設定される。

　一層目　　　　　　　　　　　　k+1層目



グリッド番号は、最上層南西端を0番として北向きにナンバリングしてゆく。南北グリッド数がnyの場合、西端から1列目北端がny-1番、2列目の南端がny番となる。下層グリッドも同じ順序のナンバリングで、k層目南西端がk\*nh番となる。

double\* grid->z には一層目各ブロックの上端(地表面)深度を格納する。順序はグリッドのナンバリングと同じ規則に従う。第2層目以降は、一層目のブロック深度(z[k]) - ブロックzディメンジョン(dz) × 層番号(一層目=0)で計算される。

// gridのメモリ確保

grid\* grid\_alloc ();

// gridオブジェクト作成 (東西nx、南北ny、上下グリッド数nz、西端x0、東端x1、

南端y0、北端座標y1、下端z0, 上端深度z1)

grid\* grid\_new (**int** nx, **int** ny, **int** nz, **double** x0, **double** x1,

**double** y0, **double** y1, **double** z0, **double** z1);

// グリッド数指定(東西nx、南北ny、上下グリッド数nz)

**void** grid\_set\_size (grid \*g, **int** nx, **int** ny, **int** nz);

// 水平範囲指定(西端x0、東端x1、南端y0、北端座標y1, 下端z0, 上端座標z1)

**void** grid\_set\_range (grid \*g, **double** x0, **double** x1,

**double** y0, **double** y1, **double** z0, **double** z1);

// グリッド数、範囲指定後グリッドのディメンジョンを計算

**void** grid\_cal\_dimension (grid \*g);

// グリッド上端面の高さを一定値に設定

**void** grid\_set\_z\_const (grid \*g, **double** z);

// グリッド上端面の高さを配列から設定

**void** grid\_set\_z\_array (grid \*g, **int** n, **double** \*z);

// グリッド設定フラグ取得

boolean grid\_is\_specified (grid \*g);

boolean grid\_size\_specified (grid \*g);

boolean grid\_range\_specified (grid \*g);

boolean grid\_dimension\_specified (grid \*g);

boolean grid\_z\_specified (grid \*g);

// グリッド総数を取得

**int** grid\_get\_n (grid \*g);

// 東西グリッド数を取得

**int** grid\_get\_nx (grid \*g);

// 南北グリッド数を取得

**int** grid\_get\_ny (grid \*g);

// 上下グリッド数を取得

**int** grid\_get\_nz (grid \*g);

// 水平面のグリッド数を取得

**int** grid\_get\_nh (grid \*g);

// 西端点座標取得

**double** grid\_get\_x0 (grid \*g);

// 東端点座標取得

**double** grid\_get\_x1 (grid \*g);

// 南端点座標取得

**double** grid\_get\_y0 (grid \*g);

// 北端点座標取得

**double** grid\_get\_y1 (grid \*g);

// z座標配列のポインタ取得

**double**\* grid\_get\_z (grid \*g);

// グリッドの東西・南北・上下ディメンジョン取得

**double** grid\_get\_dx (grid \*g);

**double** grid\_get\_dy (grid \*g);

**double** grid\_get\_dz (grid \*g);

// i列目のx座標取得

**double** grid\_get\_ith\_col\_x (grid \*g, **int** i);

// j行目のy座標取得

**double** grid\_get\_jth\_row\_y (grid \*g, **int** j);

// i列,j行,k層目のz座標取得

**double** grid\_get\_ith\_col\_jth\_row\_kth\_layer\_z (grid \*g, **int** i, **int** j, **int** k);

// n番目のx,y,z座標取得

**double** grid\_get\_nth\_x (grid \*g, **int** n);

**double** grid\_get\_nth\_y (grid \*g, **int** n);

**double** grid\_get\_nth\_z (grid \*g, **int** n);

// n番目のx,y,z座標取得

**void** grid\_get\_nth (grid \*g, **int** n, **double** \*x, **double** \*y, **double** \*z);

4.1グリッドオブジェクトの作成

グリッドオブジェクトの領域確保は

grid \*grid\_alloc ()

で行う。これにより

g->n = 0, g->nh = 0, g->nx, ny, nz = 0

g->pos0, pos1, z = NULL

g->dim = NULL

のグリッドオブジェクトが作成される。

グリッド数 ： nx(東西)、ny(南北)、nz(上下)

範囲　　　 ： x=[x0(西), x1(東)]、y=[y0(南) ,y1(北)]、z=[z0(下) ,z1(上)]

のグリッドオブジェクトを作成するには

grid \*grid\_new (

int nx, int ny, int nz,

double x0, double x1, double y0, double y1, double z0, double z1);

を用いる。これにより、グリッドのディメンジョン(dx, dy, dz)は自動的に計算される。また、最上層グリッドの上端深度g->zはz1で一定(g->z\_constant = TRUE)となる。

個々のパラメータは種別に指定できる。

4.2 グリッドの座標値取得

グリッドの座標値は、対応するグリッドのインデックスを通して取得できる。

k層目、西からi列目、南からj行目のグリッド座標は

double grid\_get\_ith\_col\_x (g, i);

double grid\_get\_jth\_row\_y (g, j);

double grid\_get\_ith\_col\_jth\_row\_kth\_layer\_z (g, i, j, k);

で取得できる。また、最上層南西端を0として、通しでn番目のグリッド座標は

double grid\_get\_nth\_x (g, n);

double grid\_get\_nth\_y (g, n);

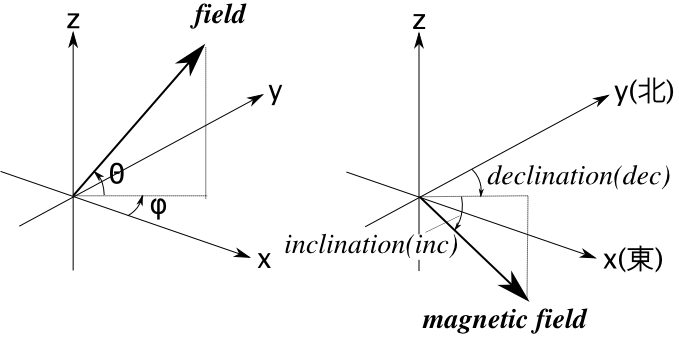
double grid\_get\_nth\_z (g, n);

又は

void grid\_get\_nth (grid \*g, int n, double \*x, double \*y, double \*z);

で取得できる。

5.　field及びsourceオブジェクト



磁場ベクトル、磁化ベクトル極形式の座標系はそれぞれ上の図に従う。

**typedef** **struct** {

// 磁化物体の位置

position \*pos;

// 磁化物体のディメンジョン

dimension \*dim;

// 磁化物体の磁化ベクトル

magnetization \*mag;

// 外部磁場の方向ベクトル

field \*exf;

} source;

// sourceオブジェクトの作成

source \*source\_new (**void**);

// sourceオブジェクトの解放

**void** source\_free (source \*s);

// 磁化物体位置の設定(x=ソースの東西中心、y=南北中心、z=上端深度)

**void** source\_set\_position (source \*s, **double** x, **double** y, **double** z);

// 磁化物体ディメンジョンの設定

**void** source\_set\_dimension (source \*s, **double** l, **double** w, **double** h);

// 磁化ベクトルの設定(intensity=磁化強度、dec=偏角、inc=伏角)

**void** source\_set\_magnetization (source \*s, **double** intensity,

**double** dec, **double** inc);

// 外部磁場方向ベクトルの設定(dec=偏角、inc=伏角)

**void** source\_set\_external\_field (source \*s, **double** dec, **double** inc);

source\_set\_magnetization (), source\_set\_external\_field ()で、磁化ベクトル、外部磁場方向ベクトルを設定できる。これらはまた

// 磁化・外部磁場ベクトルの設定(dec=偏角、inc=伏角)

**void** magnetization\_set\_from\_poler (magnetization \*mag,

**double** intensity, **double** dec, **double** inc);

**void** external\_field\_set\_from\_poler (field \*exf, **double** dec, **double** inc);

でも設定できる。

6. データ入出力

6.1 データの入力

io/read.h

// ファイル FILE \*fp からデータを読み込み

datalist \***fread\_datalist** (FILE \*fp);

datalist \***fread\_datalist\_with\_read\_specification** (FILE \*fp,

**char** \*delim, **char** \*rspec);

ファイルからデータを読み込むには

datalist \***fread\_datalist** (FILE \*fp);

を用いる。ここで仮定されているファイル内のデータフォーマットは

"x座標"(double)\t"y座標"(double)\t"z座標"(double)\t"観測値("double)

となる。

また、ファイルから順序を指定してデータを読むには

datalist \***fread\_datalist\_with\_read\_specification** (FILE \*fp,

**char** \*delim, **char** \*rspec);

を用いる。ここで **char** \*delim はファイルのセパレータ、, **char** \*rspec はデータ読み取りの順序を指定し、

x座標/y座標/z座標/観測値 をそれそれファイルの何フィールド目から読み込むかを指定する。例えば、

"y座標","x座標","z座標", "観測値"

という並びのデータを読む場合は

delim = ","

rspec = "2/1/3/4"

となる。また、rspec で0以下の値を指定されたデータは読み込まれない(デフォルトで0の値になる)。

例えば

"y座標"　"x座標"　"観測値"

なら

delim = "　"

rspec = "2/1/0/4"

となる。

6.2 データの出力

io/write.h

**void** **fprintf\_datalist** (FILE \*stream, datalist \*data, **char** \*format);

**void** **fprintf\_datalist\_with\_data\_vector** (FILE \*stream, datalist \*data,

c\_vector \*v, **char** \*format);

**void** **fprintf\_grid** (FILE \*stream, grid \*g, **char** \*format);

**void** **fprintf\_grid\_with\_data\_vector** (FILE \*stream, grid \*g, c\_vector \*v,

**char** \*format);

**void** **fprintf\_nth\_layer\_grid** (FILE \*stream, grid \*g, **int** nz, **char** \*format);

**void** **fprintf\_nth\_layer\_grid\_with\_data\_vector** (FILE \*stream, grid \*g, **int** nz, c\_vector \*v, **char** \*format);

datalist に格納されたデータを出力するには

**void** **fprintf\_datalist** (FILE \*stream, datalist \*data, **char** \*format);

を用いる。 **char** \*format は"x座標","y座標","z座標", "観測値"の出力フォーマットで、例えば

"xx.xx"\t"yy.yy"\t"zz.zz"\t"ff.ffff"\n

というフォーマットで出力するには　format="%.2f\t%.2f\t%.2f\t%.4f となる。

datalist と、c\_vector\* に格納されたデータを並べて出力するには

**void** **fprintf\_datalist\_with\_data\_vector** (FILE \*stream, datalist \*data,

c\_vector \*v, **char** \*format);

を用いる。ただし、 datalist->sizeとv->size は一致しなければいけない。この関数は、入力データと再計算された理論値を列挙する場合等に使用することができる。

グリッドの値を出力するには

**void** **fprintf\_grid** (FILE \*stream, grid \*g, **char** \*format);

を用いる。これにより、すべてのグリッドのx座標、y座標、z座標が FILE \*stream に出力される。また、グリッドに関連されるc\_vector\* を座標付きで出力するには

**void** **fprintf\_grid\_with\_data\_vector** (FILE \*stream, grid \*g, c\_vector \*v,

**char** \*format);

を用いる。これは例えば、インバージョンで求まった地下構造を座標付きで出力するときなどに使用できる。以上の関数はすべてのグリッドについて出力するが、特定の一層だけを出力するには

**void** **fprintf\_nth\_layer\_grid** (FILE \*stream, grid \*g, **int** nz,

**char** \*format);

**void** **fprintf\_nth\_layer\_grid\_with\_data\_vector** (FILE \*stream, grid \*g,

**int** nz,　c\_vector \*v, **char** \*format);

を用いる。ここでnzは出力する層を指定する(第一層目はnz=0)。

8. データ合成

datalist \***synthesize\_datalist** (**int** nx, **int** ny, **double** x0, **double** x1,

**double** y0, **double** y1, **double** zobs, **double** f);

datalist \***synthesize\_datalist\_from\_array** (**int** n, **double** \*x, **double** \*y,

**double** \*z, **double** \*f);

datalist \***synthesize\_datalist\_with\_data** (**int** nx, **int** ny,

**double** x0, **double** x1, **double** y0, **double** y1,

**double** zobs, source \*s, mgcal\_function \*func);

datalist \***synthesize\_datalist\_from\_array\_with\_data** (**int** n,

**double** \*x, **double** \*y, **double** \*z,

source \*s, mgcal\_function \*func);

範囲 [x0, x1]、[y0, y1]で高度 z=zobs の平面上のdatalist を合成するには

datalist \***synthesize\_datalist** (**int** nx, **int** ny, **double** x0, **double** x1,

**double** y0, **double** y1, **double** zobs, **double** f);

を用いる。観測値は **double** f で一定になる。観測値として理論計算の計算値を代入する場合は

datalist \***synthesize\_datalist\_with\_data** (**int** nx, **int** ny,

**double** x0, **double** x1, **double** y0, **double** y1,

**double** zobs, source \*s, mgcal\_function \*func);

を用いる。ここで mgcal\_function \*func は

**typedef struct** {

mgcal\_theoreticall function;

void \*params;

}

で、これは関数

mgcal\_function \*

mgcal\_function\_new (mgcal\_theoretical func\_ptr, void \*user\_data);

で作成する。

また mgcal\_theoretical は

**typedef** **double** (\*mgcal\_theoretical) (position\*, source\*, void \*);

で、引数に position\*, source\* 及び void \* を持つ関数となる。

例えば、

**double** total\_force\_prism (position \*pos, source \*s, void \*user\_data)

で計算させる場合は

mgcal\_function \* **func** = mgcal\_function\_new (total\_force\_prism, NULL);

**synthesize\_datalist\_with\_data** (nx, ny, x0, x1, y0, y1, s, **func**);

と呼び出す。

平面データではなくn次の配列に格納された点 (**double** \*x, **double** \*y, **double** \*z) で理論値を計算する場合は

datalist \***synthesize\_datalist\_from\_array** (**int** n, **double** \*x, **double** \*y,

**double** \*z, **double** \*f);

datalist \***synthesize\_datalist\_from\_array\_with\_data** (**int** n,

**double** \*x, **double** \*y, **double** \*z,

source \*s, mgcal\_function \*func);

を用いる。

構造体 grid \* で表されるグリッドで理論値を計算する場合は

datalist \***synthesize\_datalist\_from\_grid\_with\_data** (grid \*g,

source \*s, mgcal\_function \*func);

を用いる。この時、理論計算は g->z の高度について行われる。

9. 理論計算

magcalc.h

// 磁場ベクトル\*f及び外部磁場方向ベクトル\*exfから全磁力計算

**double** **total\_force** (field \*exf, field \*f);

// 双極子ソースによる磁気異常3成分計算

field \***dipole** (position \*obs, source \*s);

field \***calc\_dipole** (**double** x, **double** y, **double** z, source \*s);

// プリズムソースによる磁気異常3成分計算

field \***prism** (position \*obs, source \*s);

field \***calc\_prism** (**double** x, **double** y, **double** z, source \*s);

kernel\_matrix.h

c\_matrix \***kernel\_matrix** (grid \*g, field \*mag, field \*exf,

datalist \*data, mgcal\_function \*func);

**void** **set\_kernel\_matrix** (c\_matrix \*a, **int** start\_row, **int** start\_col,

grid \*g, field \*mag, field \*exf,

datalist \*list, mgcal\_function \*func);

stabilizer.h

c\_matrix \***stabilizer\_minnrm** (**int** nx, **int** ny, **int** nz, **double** alpha);

**void** **set\_stabilizer\_minnrm** (c\_matrix \*c, **int** start\_row, **int** start\_col,

**int** nx, **int** ny, **int** nz, **double** alpha);

c\_matrix \***stabilizer\_smooth** (**int** nx, **int** ny, **int** nz, **double** alpha);

**void** **set\_stabilizer\_smooth** (c\_matrix \*c, **int** start\_row, **int** start\_col,

**int** nx, **int** ny, **int** nz, **double** alpha);

equation.h

c\_matrix \***kernel\_matrix\_alloc** (datalist \*list, grid \*g);

c\_matrix \***kernel\_matrix\_minnrm\_alloc** (datalist \*list, grid \*g);

c\_matrix \***kernel\_matrix\_smooth\_alloc** (datalist \*list, grid \*g);

c\_vector \***observation\_vector\_minnrm\_alloc** (datalist \*list, grid \*g);

c\_vector \***observation\_vector\_smooth\_alloc** (datalist \*list, grid \*g);

9.1 磁場計算

ダイポール及び磁化したブロックによる磁場3成分は、それぞれ

field \***dipole** (position \*obs, source \*s);

field \***prism** (position \*obs, source \*s);

で計算される。返り値 fileld \*f の (f->x, f->y, f->z) がそれぞれ磁場3成分を表す。

観測点位置を(x,y,z)で指定するには

field \***calc\_dipole** (**double** x, **double** y, **double** z, source \*s);

field \***calc\_prism** (**double** x, **double** y, **double** z, source \*s);

を用いる。全磁力は、上記で計算された field \*f を用い

**double** **total\_force** (field \*exf, field \*f);

で求められる。この際、外部磁場の方向ベクトル field \*exf が必要となる。

9.2 核行列

インバージョンなどで用いる、各グリッドに単位磁化をおいた磁場値を成分にもつ行列を計算するには

c\_matrix \***kernel\_matrix** (grid \*g, field \*mag, field \*exf,

datalist \*data, mgcal\_function \*func);

を用いる。

ここで、

grid \*g : ソースを分布させるグリッド

field \*mag : ソースの磁化ベクトル(強度=単位)

field \*exf : 外部磁場の方向ベクトル

datalist \*data : 観測データ

mgcal\_function \*func : 核関数のmgcal\_function

である。前述のように

typedef {

mgcal\_theoretical function;

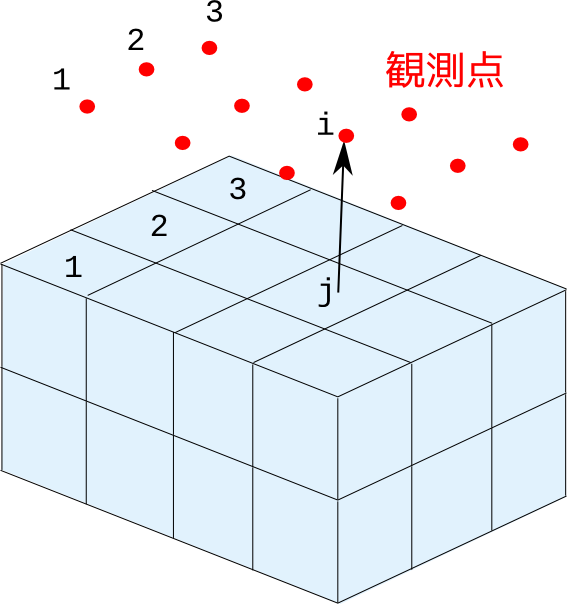
void \*params;

} mgcal\_function;

typedef (\*mgcal\_theoretical) (position \*, source \*)

である。核関数には、各グリッドの単位磁化ソースが各観測点に作る磁場を計算させる。

これにより得られる行列の i行j列成分は、下図のようにj番目のグリッドの単位ソースがi番目の観測点に作る磁場を意味する。

****

行列を新たに作らず、メモリを節約するには

**void** **set\_kernel\_matrix** (c\_matrix \*a, **int** start\_row, **int** start\_col,

　　　grid \*g, field \*mag, field \*exf,

　　　datalist \*list, mgcal\_function \*func);

を用いる。これにより c\_matrix \*a のstart\_row行start\_col列以降に核行列が上書きされる。

9.3 安定化項

インバージョンで用いられる安定化項を計算するには

c\_matrix \***stabilizer\_minnrm** (**int** nx, **int** ny, **int** nz, **double** alpha);

c\_matrix \***stabilizer\_smooth** (**int** nx, **int** ny, **int** nz, **double** alpha);

を用いる。これらはそれぞれノルム最小、平滑化の安定化項行列を返す。(nx, ny, nz) はグリッドの東西・南北・上下方向のグリッド数で、 double alpha がダンピング項である。

行列を新たに作らず、メモリを節約するには

**void** **set\_stabilizer\_minnrm** (c\_matrix \*c, **int** start\_row, **int** start\_col,

**int** nx, **int** ny, **int** nz, **double** alpha)**;**

**void** **set\_stabilizer\_smooth** (c\_matrix \*c, **int** start\_row, **int** start\_col,

**int** nx, **int** ny, **int** nz, **double** alpha);

を用いる。これにより、 c\_matrix \*c のstart\_row行、start\_col列以降に安定化行列が上書きされる。

9.4 方程式の準備

核関数、安定化項を行列に上書きする set\_\* の関数で用いる行列は

c\_matrix \***kernel\_matrix\_alloc** (datalist \*list, grid \*g);

c\_matrix \***kernel\_matrix\_minnrm\_alloc** (datalist \*list, grid \*g);

c\_matrix \***kernel\_matrix\_minderiv\_alloc** (datalist \*list, grid \*g);

c\_matrix \***kernel\_matrix\_smooth\_alloc** (datalist \*list, grid \*g);

でメモリ確保・ゼロで上書される。そのサイズは観測データ数、グリッド数から自動的に計算される。

また、安定化項付きのインバージョンを行うための観測データを格納するベクトルは

c\_vector \***observation\_vector\_minnrm\_alloc** (datalist \*list, grid \*g);

c\_vector \***observation\_vector\_minderiv\_alloc** (datalist \*list, grid \*g);

c\_vector \***observation\_vector\_smooth\_alloc** (datalist \*list, grid \*g);

でメモリ確保・ゼロで上書きされる。

これを用い、例えば観測ベクトル c\_vector \*b0 を代入するには

c\_vector b = **observation\_vector\_xxx\_alloc** (list, g);

c\_vector\_ncopy (b, b0, 0, b0->size);

を行う。