FreeFem++ ノート

桂田 祐史

2012 年 7 月 3 日, 2023 年 7 月 28 日

(このところサボっているけれど、講義の内容をこちらにフィードバックしないと。例えば https://m-katsurada.sakura.ne.jp/ana2023/ANA07_0530_handout.pdf とか。また FreeFem++ のサポート・チームも変わったようで、インストールの仕方など、ここに書いた説明が通用しなくなっているところも多い。)

FreeFem++ を見て「なんて便利なんだろう」と思う。自分が解きたい問題ほぼそのままの問題のプログラム例があればとてもハッピー。でも…似てはいるけど、そのままパクればですまない問題に突き当たってから、苦行が始まる。

必要になったときに泥縄式に調べ物をして、後のためにメモを書く。つまり本質的に自分用。

最初、内輪向けに書いた「FreeFem++の紹介」の公開版を作った (アクセス制限をするのは面倒なので)。

(2014/12) 今年、大塚・高石 [1] が出版された。「有限要素法で学ぶ現象と数理」[1] というサポート・ページがある (今はリンク切れ？直さないのかな。)。

(2016/2/11–12) 応用数理学会のチュートリアルに参加した。その講義資料は参考になる。鈴木 [2] を見よ。

(2016/6/4–15) 応用数理学会のチュートリアル (advanced course) に参加した。この資料である鈴木 [3] も公開されている。

(2021/11/8) 最近はマニュアルに “Language References” という章が用意されるようになった。最初からあれば、この文書を書かずに済んだかもしれない、という気がしないでもない。でも、率直に言ってもう少し親切に書いて欲しい。

(2023/7/28) 遅ればせながら、ParaView をインストールして使ってみた。授業で紹介したりすると良いのかもしれないけれど、とりあえずメモ (「ParaView を使ってみる (大変遅ればせながら)」)。

1 実行形式

普通 `FreeFem++` を使う。例えば `prog.edp` を実行するには、

```
FreeFem++ prog.edp
```

とする。拡張子 `.edp` を推奨しているようだが、(今のところ) 何であっても解釈実行するみたい[2]。

[1] http://comfos.org/jp/ffempp/book/

[2] edp は équation aux dérivées partielles (偏微分方程式, 英語では partial differential equations) の略だとか。

以前は FreeFem++ は /usr/local/bin の下に置かれたが、今は /usr/local/ff++ の下に置かれるようである。PATH も自動設定されるようになったので (Mac では /etc/path.d/FreeFem++ が作られる)、インストールは簡単になってトラブル・フリーになった。

2 概観

2.1 C 言語と良く似ているところ

- // から行末までは注釈、/* と */ で挟まれた部分は注釈 (共に C 言語と同じ)
- 文の最後は ;
- 四則演算 (+, -, *, /) や、代入 (=) などの演算子
- 0 は偽、0 以外の整数は真とみなす。一方、比較演算・論理演算などの結果は 0 (false) または 1 (true).
- 変数宣言の文法も C 言語と同様。型名の後に , で区切った名前のリストを書く。
- 関数呼び出しの文法も C 言語と同様。
- ブロックは { と } で複数 (0 個以上) の文を囲んで作る。
- 比較演算子 (==, !=, <, <=, >, >=)、論理演算子 (&&, ||, !)、if, if else などの制御構造。
  ただし switch はない。
- for, while などの繰り返し制御。break (ループを抜ける), continue (次の繰り返し) など。
  ただし do while はない。
- 数学関数の名前

他にもあるだろう…

脱線 実際、文法のほとんどは C, C++ のそれに近いので、簡単な C 言語の計算プログラムは比較的簡単に FreeFem++ に書き換えられる (と感じている)。論より証拠。差分法のプログラムを書いてみよう。

heat1d-e-freefem.edp — 1 次元熱方程式に対する差分法プログラム

```
// heat1d-e-freefem.edp
// 実行: FreeFem++ heat1d-e-freefem.edp
// N, x が大域的な識別子を隠すと警告が出る (つまり名前が衝突する)。

int i, N=50, n, nMax;
real h = 1.0 / N, lambda = 0.5, Tmax = 1.0;
real tau = lambda * h^2;
real[int] x(N+1);
real[int] u(N+1);
real[int] newu(N+1);

cout << "h= " << h << endl;
cout << "lambda= " << lambda << endl;
cout << "tau= " << tau << endl;

func real f(real x) {
  if (x < 0.5)
    return x;
  else
    return 1 - x;
}

for (i = 0; i <= N; i++) {
  x[i] = i * h;
  u[i] = f(x[i]);
}
plot([x,u], bb=[[-0.1,-0.1],[1.1,1.1]],aspectratio=true, wait=true);

nMax = rint(Tmax / tau);
cout << "nMax =" << nMax << endl;
for (n = 1; n <= nMax; n++) {
  for (i = 1; i < N; i++)
    newu[i] = (1.0 - 2.0 * lambda) * u[i] + lambda * (u[i+1] + u[i-1]);
  // cout << newu << endl;
  newu[0] = newu[N] = 0;
  u = newu;
  plot([x, u], bb=[[-0.1,-0.1],[1.1,1.1]],aspectratio=true);
}
```

なお、C++ ライクな cin も使うことが出来るので、N の値をキーボード入力することも可能である。

```
int i, N, n, nMax;
cout << "input N: ";
cin >> N;
real h = 1.0 / N, lambda = 0.5, Tmax = 1.0;
```

3 入出力 — cin, cout を用いた入出力

C++ににているので、付録 A を適宜参考にすること。
FreeFem++の real データの入出力の書式指定は次のようになっている。

- 何も指定しないと C 言語の %g 相当の出力になる。
- cout.precision(n); とすると、以下小数点以下の桁数は n になる。

```
cout.precision(15);
cout << "pi=" << pi << endl;
```

- 幅を指定するには `<< setw(桁数)` とする (これは毎回必要)。

```
cout << "pi=" << setw(20) << pi << endl;
```

- `cout.fixed;` とすると、以下固定小数点数形式 (C 言語の `%f` 相当) になる。

```
cout.fixed;
cout << "NA=" << NA << endl;
```

- `cout.scientific;` とすると、以下指数形式 (C 言語の `%e` 相当) になる。

```
cout.scientific;
cout << "pi=" << pi << endl;
```

- `cout.default;` とすると、以下デフォールト (C 言語の%g) に戻る。

例をあげる。

```
// testfloat.edp
real NA = 6.022e+23;
// デフォールト %g に相当
cout << "pi=" << pi << ", NA=" << NA << ", pi*NA=" << pi * NA << endl << endl;
// 幅を 20 に指定 %20g に相当
cout << "pi=" << setw(20) << pi << ", NA=" << setw(20) << NA
     << ", pi*NA=" << setw(20) << pi * NA << endl << endl;
// 小数点以下の桁数を 15 に指定 %20.15g に相当？
cout.precision(15);
cout << "pi=" << setw(20) << pi << ", NA=" << setw(20) << NA
     << ", pi*NA=" << setw(20) << pi * NA << endl << endl;
// 固定小数点数形式 %.15f に相当
cout.fixed;
cout << "pi=" << pi << ", NA=" << NA << ", pi*NA=" << pi * NA << endl << endl;
// %20.15f に相当
cout << "pi=" << setw(20) << pi << ", NA=" << setw(20) << NA
     << ", pi*NA=" << setw(20) << pi * NA << endl << endl;
// 指数形式 %20.15e に相当
cout.scientific;
cout << "pi=" << setw(20) << pi << ", NA=" << setw(20) << NA
       << ", pi*Na=" << setw(20) << pi * NA << endl << endl;
// %g 形式に戻す %.15g に相当
cout.default;
cout << "pi=" << pi << ", NA=" << NA << ", pi*Na=" << pi * NA << endl;
```

外部ファイルとの入出力については、14 節を見よ。

4 型

- 整数を表すための int がある (C 言語の `int` に相当)
- 実数を表すための real がある (C 言語の `double` に相当)
- 複素数を表すための complex がある (C 言語の `complex` に相当, 実部・虚部が `double`)
  `complex z=1.0+2i;`

- 論理を表すための bool がある (C 言語の `bool` に相当). `true`, `false` と言う値があるが、それぞれ 1, 0 の別名と考えて良い。

```
bool f=false;
bool t=true;
cout << f << endl;
cout << t << endl;
```

  これで 0 と 1 が表示される。気持ちは Boolean で、中身は int か。
  例えば `plot(u,wait=true);` は `plot(u,wait=1);` と同じ。
  要するに C や C++ と同じである。これを利用すると $(-2,-1)\times(-3,3)$ の特性関数は

```
func chi=(-2 < x) * (x < -1) * (-3 < y) * (y < 3);

あるいは

func chi=(-2 < x && x < -1) * (-3 < y && y < 3);
```

  で定義出来る。

- 文字列を表すための string がある (C++言語の `string` に相当, 日本語不可？)。
  - 2 つの string `s1`, `s2` を、(+ 演算子を用いて) `s1+s2` で連結できる。
  - string+数値 とすると、数値を文字列に変換してから連結する。

```
real a=1.23, b=4.56;
string s;
s= "a=" + a + ", b=" + b + ".";
cout << s << endl;
```

  - string を int に変換する `atoi()`, string を real に変換する `atof()` がある (C 言語の真似)。

5 配列

5.1 (添字が整数である普通の) 1 次元配列

1 次元配列は、C 言語に (少し) 似ている。

```
real[int] x(3);
```

これで `x[0]`, `x[1]`, `x[2]` あるいは `x(0)`, `x(1)`, `x(2)` が使える。`x[1]` でも `x(1)` でも違いがないのかは不明である。

2 次元配列では `[ ]` は使えないようなので、`( )` を使う、と覚える方が良いかも。

```
real[int] a1(3); // C で double a1[3]; とするのに似ている
for (int i=0;i<3;i++)
  a1[i]=i; // a1(i)=i; としても良い。

real[int] a2 = [0,1,2]; // C で double a[]={0,1,2}; とするのに似てる
real[int] a3 = 0:2; // これは少し MATLAB 風

cout << "a1=" << a1 << endl;
cout << "a2=" << a2 << endl;
cout << "a3=" << a3 << endl;
```

追記: a1 の要素数は a1.n で得られる。

まるで初心者の C++ みたいな

```
int i,j;
real[int] xx(9),yy(9);
for (i=0; i<9; i++) xx[i]=(i+1);
cout << xx << endl;
for (i=0; i<9; i++) yy(i)=(i+1);
for (i=0; i<9; i++) {
  for (j=0; j<9; j++)
    cout << (xx[i] * yy[j]) << " ";
  cout << endl;
}
```

5.2 2 次元配列

2 次元配列も定義できるようだが、こちらはカギ括弧 [ ] は使えず、丸括弧 ( ) を使う必要があるみたい。

2 次元配列はこんなふうに

```
real[int,int] a(2,2);
a(1,1)=1; a(1,2)=2;
a(2,1)=3; a(2,2)=4;
```

2 次元配列の例 (丸い括弧を使う)

```
real[int,int] a(3,2);
a(0,0)=1; a(0,1)=2;
a(1,0)=3; a(1,1)=4;
a(2,0)=5; a(2,1)=6;
cout << "a.n=" << a.n << ", a.m=" << a.m << endl; // 3 と 2 になる。
for (int i=0; i < a.n; i++) {
  for (int j=0; j < a.m; j++)
    cout << setw(4) << a(i,j) << " ";
  cout << endl;
}
cout << "output by \"cout<<a<<endl;\"" << endl;
cout << a << endl;
```

2 次元配列の例 (丸い括弧を使う)

```
$ FreeFem++ foo.edp
-- FreeFem++ v4.700001 (Jeu  5 nov 2020 11:02:28 CET - git v4.7-1)
 Load: lg_fem lg_mesh lg_mesh3 eigenvalue
    1 : real[int,int] a(3,2);
    2 : a(0,0)=1; a(0,1)=2;
    3 : a(1,0)=3; a(1,1)=4;
    4 : a(2,0)=5; a(2,1)=6;
    5 : cout << "a.n=" << a.n << ", a.m=" << a.m << endl; // 3と2になる。
    6 : for (int i=0; i < a.n; i++) {
    7 :   for (int j=0; j < a.m; j++)
    8 :     cout << setw(4) << a(i,j) << " ";
    9 :   cout << endl;
   10 : }
   11 : cout << "output by \"cout<<a<<endl;\"" << endl;
   12 : cout << a << endl;
   13 :  sizestack + 1024 =1168  ( 144 )

a.n=3, a.m=2
   1    2
   3    4
   5    6
output by "cout<<a<<endl;"
3 2
   1   2
   3   4
   5   6

times: compile 0.01818s, execution 0.000271s,  mpirank:0
 CodeAlloc : nb ptr  3611,  size :458584 mpirank: 0
Ok: Normal End
$
```

5.3 misc

行列について、掛け算や転置、逆転 (逆行列) が出来る。
連想配列っぽいのも使える。

```
real[string] a;

a["tako"] = 8.0;
a["ika"] = 10.0;
a["tsuru"] = 2.0;
a["kame"] = 4.0;
```

配列は [と] でくくって表せる。代入出来るのはもちろん、初期化にも使える。

```
real[int] c=[1,2,3];
cout << c << endl;

real[int] d;
d=[1,2,3,4];
cout << d << endl;
```

配列はソート出来る。

```
a.sort;
```

定義と同時に初期化する場合、MATLAB 風の a:b や a:dx:b が使える (dx が非整数のときは a は整数にしないこと)。

```
real[int] a(2:8);
cout << a << endl;

real[int] b(2.0:0.3:10);
cout << b << endl;
```

```
[chronos:~/work] mk% FreeFem++ foobar.edp
 EXEC of the plot  : ffglut
-- FreeFem++ v  3.190000 (date Ven 20 avr 2012 08:49:54 CEST)
 Load: lg_fem lg_mesh lg_mesh3 eigenvalue
    1 : real[int] a(2:8);
    2 : cout << a << endl;
    3 :
    4 : real[int] b(2.0:0.3:10);
    5 : cout << b << endl; sizestack + 1024 =1136  ( 112 )

7
  2   3   4   5   6
  7   8
27
  2 2.3 2.6 2.9 3.2
3.5 3.8 4.1 4.4 4.7
  5 5.3 5.6 5.9 6.2
6.5 6.8 7.1 7.4 7.7
  8 8.3 8.6 8.9 9.2
9.5 9.8
times: compile 0.005558s, execution 0.000107s,  mpirank:0
Err ReadOnePlot -1
 CodeAlloc : nb ptr  2330,  size :313288 mpirank: 0
Bien: On a fini Normalement
[chronos:~/work] mk%
```

6 便利な misc

- 円周率 pi
- clock() CPU 時間

経過時間を測る

```
real then=clock();
...
cout << clock() - then << endl;
```

- verbosity=0; とすると情報の出力を抑制する (そんなに減らないような気もするが…)。
- exec(文字列); で外部のコマンドを実行出来る。

```
dirname="datadir";
exec("mkdir " + dirname);
```

7 境界 border

問題を考える領域の境界曲線 (の一部) を定義するために border という命令がある (曲線は border と言う型の変数として定義される)。

円周全体を C とする

```
border C(t=0,2*pi) { x=cos(t); y=sin(t); }
```

円周の上半分、下半分を別々に Gamma1, Gamma2 と定義する

```
int C=1;
...
border Gamma1(t=0,pi)    { x=cos(t); y=sin(t); label=C; }
border Gamma2(t=pi,2*pi) { x=cos(t); y=sin(t); label=C; }
```

label= でラベルを定義している。0 でない値が選べる (節点にラベルをつけるとき、領域内部の点は 0 というラベルをつける)。

正方形領域 $(0,1)\times(0,1)$ の 4 つの辺 C1,C2,C3,C4 を定義

```
border C1(t=0,1) { x=t; y=0; }
border C2(t=0,1) { x=1; y=t; }
border C3(t=0,1) { x=1-t; y=1; }
border C4(t=0,1) { x=0; y=1-t; }
```

パラメータを (t=t0,t1) の形式で指定するが、t0 < t1 であるとは限らない。

```
border below(t=-1,1) { x =  t; y = -1; label=gamma; }
border right(t=-1,1) { x =  1; y =  t; label=gamma; }
border up(t=1,-1)    { x =  t; y =  1; label=gamma; }
border left(t=-1,1)  { x = -1; y =  t; label=gamma; }
```

8 メッシュ buildmesh(), readmesh(), savemesh(), ...

メッシュはかなり奥が深い。少し「変わったこと」をしようと思うと、マニュアル 5 章を読むことになると思われる。

8.1 buildmesh()

buildmesh() を使ってメッシュを作ることが出来る。

```
mesh 名前 = buildmesh(境界の名前 (分割の指定));
```

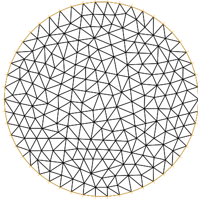


図 1: C(50)

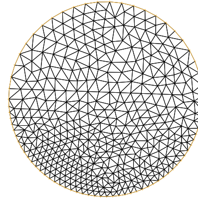


図 2: Gamma1(25)+Gamma2(50)

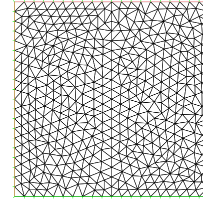


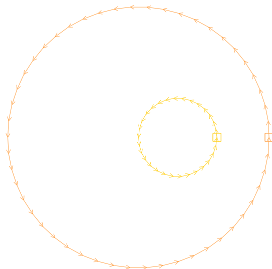
図 3: C1(20)+C2(20)+...

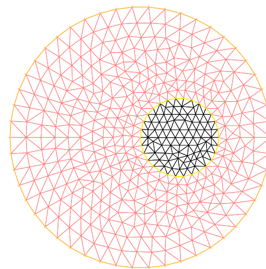
```
border C(t=0,2*pi) { x=cos(t); y=sin(t); }

int C0=1;
border Gamma1(t=0,pi)    { x=cos(t); y=sin(t); label=C0; }
border Gamma2(t=pi,2*pi) { x=cos(t); y=sin(t); label=C0; }

border C1(t=0,1) { x=t; y=0; }
border C2(t=0,1) { x=1; y=t; }
border C3(t=0,1) { x=1-t; y=1; }
border C4(t=0,1) { x=0; y=1-t; }

mesh Th1=buildmesh(C(50));
mesh Th2=buildmesh(Gamma1(25)+Gamma2(50));
mesh Th3=buildmesh(C1(20)+C2(20)+C3(20)+C4(20));

plot(Th1,wait=true,ps="Th1.eps");
plot(Th2,wait=true,ps="Th2.eps");
plot(Th3,wait=true,ps="Th3.eps");
```

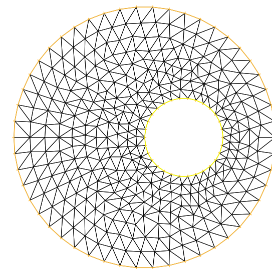
有限個の Jordan 閉曲線で囲まれた多重連結領域を三角形分割することもできる。

sampleMesh.edp

```
border a(t=0,2*pi){ x=cos(t); y=sin(t);label=1;}
border b(t=0,2*pi){ x=0.3+0.3*cos(t); y=0.3*sin(t);label=2;}
plot(a(50)+b(+30),wait=true,ps="border.eps");
mesh ThWithoutHole = buildmesh(a(50)+b(+30));
mesh ThWithHole = buildmesh(a(50)+b(-30));
plot(ThWithoutHole,wait=1,ps="Thwithouthole.eps");
plot(ThWithHole,wait=1,ps="Thwithhole.eps");
```







8.2 square()

sqaure() で $[0,1] \times [0,1]$ を 10×10 分割

```
mesh Th = square(10,10);
```

実は square() で長方形 $[a,b] \times [c,d]$ も分割できるようで、

```
mesh Th = square(m, n, [a+(b-a)*x,c+(d-c)*y])
```

とすれば良い。

```
// [1,3] × [2,5] を 20 × 30 分割
mesh Th = square(20,30, [1+2*x, 2+3*y]);
plot(Th);
```

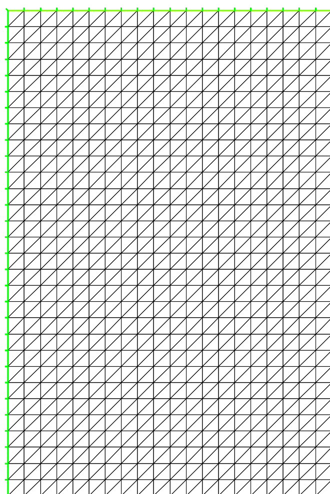


図 4: [1,3] × [2,5] を 20 × 30 分割

なお、square() で作ったメッシュは、下の辺、右の辺、上の辺、左の辺 (反時計回り) の順に 1, 2, 3, 4 というラベルがつけられている。

8.3 savemesh(), readmesh()

作ったメッシュは savemesh() を使ってファイルに保存出来る。

```
savemesh(Th, "Th.msh");
```

readmesh() を使ってメッシュをファイルから読み込むことが出来る。

```
mesh Th = readmesh("mymesh.msh");
```

region パラメーターとは？

8.4 mesh クラスの詳細

- Th をメッシュとするとき、Th.nt は三角形の数 (the number of triangles)、Th.nv は節点の数 (the number of vertices)、Th.area は領域の面積 (area) である。
- Th(i) は i 番目の節点 ($i = 0, 1, \ldots,$ Th.nv $- 1$) で、その座標は Th(i).x と Th(i).y である。Th(i).label はその接点が領域内部にあるか (0)、境界にあるか (0 以外)、境界のどの部分にあるかを示す (ラベルの値ということらしい)。
- Th[i] は i 番目の三角形 ($i = 0, 1, \ldots,$ Th.nt $- 1$)、Th[i][j] は i 番目の三角形の j 番目の節点 ($j = 0, 1, 2$) の全体節点番号、その節点の座標は Th[i][j].x と Th[i][j].y である。三角形の面積は Th[i].area である。
- 点 (x, y) を含む三角形の番号は Th(x,y).nuTriangle で得られる。

8.5 mesh ファイルの構造

ここは、リーバース・エンジニアリングする。

正方形領域 $(0, 1) \times (0, 1)$ を次のようなコードで分割してみよう。正方形の辺のうち、左と下にあるものに 1 というラベル、右と上にあるものに 2 というラベルをつけている。

```
int Gamma1=1, Gamma2=2;
border Gamma10(t=0,1) { x=0; y=1-t; label=Gamma1; }
border Gamma11(t=0,1) { x=t; y=0; label=Gamma1; }
border Gamma20(t=0,1) { x=1; y=t; label=Gamma2; }
border Gamma21(t=0,1) { x=1-t; y=1; label=Gamma2; }

int m=2;
mesh Th = buildmesh(Gamma10(m)+Gamma11(m)+Gamma20(m)+Gamma21(m));

savemesh(Th,"Th.msh");
```

として Th.msh を作ると

```
9 8 8
0 0 1
0 1 2
0 0.5 1
0.5 0 1
1 0 2
0.499999999488 0.499999999488 0
0.5 1 2
1 0.5 2
1 1 2
7 8 9 0
1 4 3 0
4 5 8 0
6 8 7 0
4 6 3 0
4 8 6 0
2 3 7 0
7 3 6 0
9 7 2
7 2 2
5 8 2
8 9 2
1 4 1
4 5 1
2 3 1
3 1 1
```

多分次のようなフォーマットになっているのであろう。

総節点数 n　総要素数 m　境界に属する辺の数 N
節点 P_0 の x,y 座標とラベル (0 は内点)
$\vdots$
節点 P_{n-1} の x,y 座標とラベル (0 は内点)
要素 e_0 の 3 節点の節点番号と 0
要素 e_1 の 3 節点の節点番号と 0
$\vdots$
要素 e_{m-1} の 3 節点の節点番号と 0
境界に属する辺 Q_0 の両端の点の節点番号とラベル
$\vdots$
境界に属する辺 Q_{N-1} の両端の点の節点番号とラベル

(FreeFem++ の内部では、番号は 0 からつけるのが基本のようである。エラーメッセージを読むときは、そのことを念頭におくこと。)

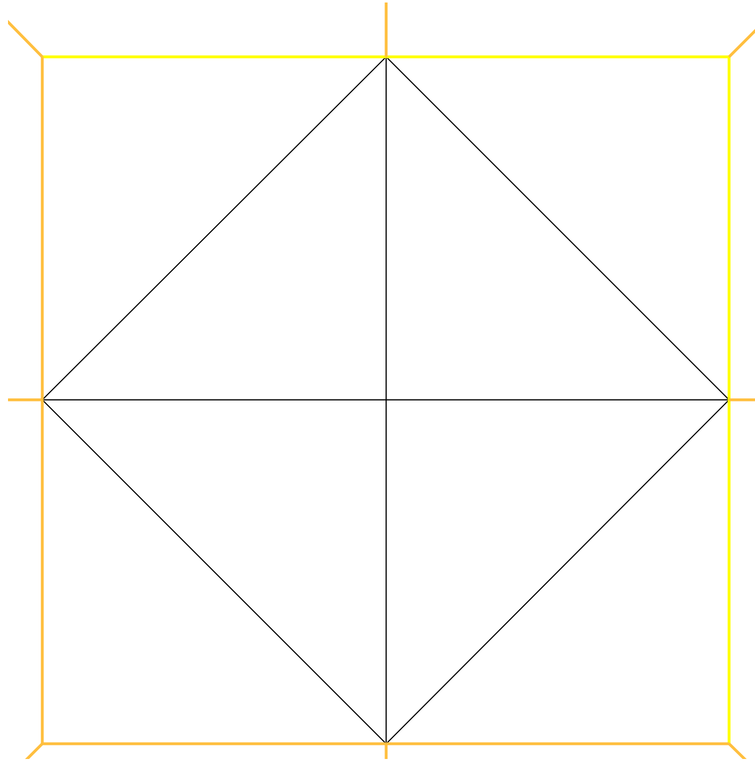


図 5: $m = 2$ の時の三角形分割の様子 (Th.msh に対応)

9 有限要素空間 fespace

既に定義しておいたメッシュと、要素の種類を表す名前 (P1, P2, ...) を用いて、有限要素空間 (数学では V_h などの記号で表すことが多い) を定義する。

有限要素空間 (型) の定義

```
fespace 名前 (メッシュの名前, 要素の種類を表す名前);
```

例えば

```
fespace Vh(Th, P1);
```

要素の種類としては、P1, P2, P1b, ... P2Morley (load "Morley"; が必要), P3 (load "Element_P3"; が必要), ··· 山のようにある。

有限要素空間は、数学的には (数ベクトルもどきの) 集合であるので、Freefem++ 的には (数ベクトルもどきを表す) 型名である。具体的に変数を宣言するには、定義した型名 変数名; とするわけだ。

有限要素空間の元 (有限要素空間型を持つ変数) の定義

```
型名 変数名;
```

例えば

```
Vh u,v;
```

有限要素空間の元は実質的に数ベクトルである (という構造を持っている) から、足したり、実数をかけたりできる。(注: u が有限要素空間の元であるとき、u[] は配列となる、らしい。)

一方で、有限要素空間の元は、単なる数ベクトルでなく、区分的多項式であり、節点以外での関数値が定義されていて、それが (例えば u が 2 次元の有限要素空間の変数である場合)

u(x,y) のようにして計算できる。

正方形領域での「格子点」上の値を出力

```
mesh Th=square(N,N);
...
fespace Vh(Th,P1);
Vh u;
...
ofstream f("u.dat");
real xi,yj; // x,y だと名前が衝突して警告されるので
real h=1.0/N;
for (int i=0; i<=N; i++) {
  xi=i*h;
  for (int j=0; j<=N; j++) {
    yj=j*h;
    f << u(xi,yj) << " "; // ここに注目
  }
  f << endl;
}
```

x, y という定義済みの名前は、節点の x 座標, y 座標を並べたベクトルになっているので、それを使って関数の値を設定出来る。

```
Vh g = sin(pi*x)+cos(pi*y);
```

メッシュ上の数ベクトルであるから、plot() で等高線を描いたり、int2d()() で数値積分したりも出来る (いずれも後述)。

10 plot()

メッシュを描く

```
Th = buildmesh(..);
...

plot(Th);
```

有限要素空間の元 (メッシュ上の数値ベクトル) の等高線を描く

```
Vh f;
...
plot(f);
```

等高線の高さを指定したい場合は、viso= オプションを用いる (等高線の高さを収めた配列の名前を指定する)。

コメントを書きたい場合は cmm= オプションを用いる。

```
Vh f;
real [int]level = [0.0];     // 高さ 0 の等高線のみ
real [int]levels = -1.0:0.1:1.0; // -1 から 0.1 刻みで 1 まで
...
plot(f, viso=level, wait=true, cmm="nodal line");
plot(f, viso=levels, value=true, wait=true, cmm="contour lines");
```

$[0,1]\times[0,1]$ 上の関数 x^2-y^2 を描く

```
mesh Th=square(20,20);
plot(Th,wait=1);
fespace Vh(Th,P1);
Vh u=x*x-y*y;
plot (u,wait=1);
```

wait= で一時停止するかどうかをコントロール

```
debug=true; // debug=1; でも可
...
plot(f,wait=debug)
```

ps=ファイル名 で描画と同時に PostScript ファイルも出力

```
plot(f,ps="graph.eps");
```

,dim=3 で 3 次元の鳥瞰図表示

```
plot(f,dim=3);
```

,fill=true で色を塗る

```
plot(f,fill=true);
```

関数のレベル 0 の等高線を描きたいのに、なぜか等高線が見えなくなるケースに遭遇した。0 の近くのレベルの範囲 $[-w,w]$ を塗りつぶすことで安直に回避した。

```
real haba=0.0001;
real [int] viso=(-haba:haba:haba);
plot(u,viso=viso,fill=true,grey=true);
```

10.1 キーボードからのコマンド

+	ズーム（イン）する
−	ズームアウトする
=	ズームの状態を元に戻す
a	ベクトルの矢印の大きさを短くする
A	ベクトルの矢印の大きさを長くする
r	リフレッシュする
f	カラーを塗りつぶすかどうか
v	
?	help
[Enter]	次のプロットへ
[ESC]	グラフィックスを閉じる

この辺は更新されているようなので、ここも書き変えないといけない。
まあ、? を打ってヘルプを見て下さい。

10.2 グラフの鳥瞰図が描きたい場合

,dim=3 という手もあるけれど…以前は、数値データをファイルに出力して、gnuplot で描画する方法が勧められていた。

gnuplot の使い方を説明する文書に書いた、「有限要素法で解いた Poisson 方程式の境界値問題の解の可視化」[3]

medit という可視化ソフトがあるということだが、マニュアルの通りにしても動かない。

```
load "medit"
mesh Th=squqre(10,10,[2*x-1,2*y-1]);
fespace Vh(Th,P1);
Vh u=2-x*x-y*y;
medit("mm",u);
```

10.3 おまけ: 1 変数関数のグラフ

```
real[int] x(0.0:0.01:1.0);
real[int] y=x.*x;
plot([x,y],aspectratio=true);
```

10.4 plot() のオプション

マニュアルの 7.1 節から。

- wait= 描画後に待つかどうか
- nbiso= 等高線の個数

[3] http://nalab.mind.meiji.ac.jp/~mk/labo/howto/intro-gnuplot/node9.html

- `nbarrow=` 等高線の色の個数
- `viso=` 等高線のレベル (配列)
- `fill=` 塗るかどうか (線だけの等高線か、等高線と等高線の間を塗りつぶすか)
- `value=` 等高線の高さの凡例を表示するかどうか
- `cmm=` ウィンドウに文字列を書き込む
- `bw=` モノクロ (白黒) にする。
- `grey=` グレースケールにする。
- `boundary=` 境界を描くかどうか
- `dim=` 2 または 3 (デフォールトは 2)
- `bb=` BoundingBox [[x1,y1],[x2,y2]]
- `ps=` 出力する PostScript ファイルの名前
- `coef=` 矢印の大きさ

10.5　サンプル・プログラム

testplot.edp

```
// testplot.edp --- plot() の使い方を試す

// メッシュを描く
int m=20;
mesh Th=square(m,m,[-1+2*x,-1+2*y]);
plot(Th,wait=1);

// [return] で次のプロット
// p        で以前のプロット

// 等高線を描く
//  N で本数を増やせる
//  n で本数を減らせる
//  f で塗りつぶしの On/Off
//  3 で 2D/3D の切り替え

func real f(real x, real y)
{
  // ここは色々複雑なことが書ける。
  return 1.0;
}

func u=x*x-y*y;
func v=2*x*y;
// plot(f); // plot() は func を描画できない。範囲指定がないので当たり前。

fespace Vh(Th,P1);
Vh fh,uh,vh;
// fh=f; // f() は代入できない
uh=u;
vh=v;
// 等高線
plot(uh,wait=1);    // plot() は fespace のオブジェクトは描画できる
plot(uh,vh,wait=1); // plot() は２つのオブジェクトを同時に描画できる

// 鳥瞰図
real [int] levels =-1.0:0.01:1.0;
plot(uh,dim=3,viso=levels,fill=true,wait=true);

// ベクトル場を描く
plot([uh,vh],wait=1);

load "plotPDF"  // without semicolon
plotPDF( "foobar", Th, uh);
```

11 int2d()(), int1d(,)()

12 弱形式を定義して解く

12.1 solve, problem

一度解けば良い定常問題などは solve() で弱形式を定義するのと同時に解いてしまえば良い。

poisson1.edp — solve で定義すると同時に解いてしまう

```
// poisson1.edp
//   Freefem++ poisson1.edp
border C(t=0,2*pi) {x=cos(t); y=sin(t);}
mesh Th = buildmesh(C(50));
fespace Vh(Th,P1);
Vh u,v;
func f=x*y;
solve Poisson(u,v,solver=LU) =
  int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v)) - int2d(Th)(f*v) + on(C,u=0);
plot(u);
```

- dx(), dy() はそれぞれ x, y での微分を表す。
  高階の微分は dxx(), dxy(), dyy() のようにする。
- int2d(Th) は、Th の領域全体の積分 (重積分) を表す。int2d()() は複数回使うことができる。2 つめのカッコ内に書く式には制限があるようで、簡単な式に分解して書く。
- int1d(Th,border) は境界のうち、 border の部分の積分 (境界積分、今の場合は線積分) を表す。int1d(Th,border1,border2) のように複数個の border が指定できる。また int1d(Th,label1,label2) のように、border の代わりに label を指定することもできる。int1d()() も複数回使うことができる。
- +on(border,u=g) とか。+on(label1,label2,u=gall) とか (+on(label1,u=g1)+on(label2,u=g2) と分けて書くことも可能)。ベクトル値関数の場合は、+on(border,u1=g1,u2=g2) のような複数の方程式を書くこともできる。

problem で定義しておいて、後で呼び出して解くことも出来る。文法は solve とほとんど同じである。

poisson2.edp — problem で定義しておいて、後から呼び出して解く

```
// poisson2.edp
//   Freefem++ poisson2.edp
border C(t=0,2*pi) {x=cos(t); y=sin(t);}
mesh Th = buildmesh(C(50));
fespace Vh(Th,P1);
Vh u,v;
func f=x*y;
problem Poisson(u,v,solver=LU) =
  int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v)) - int2d(Th)(f*v) + on(C,u=0);
Poisson;
plot(u);
```

熱方程式を解く場合などは、何度も弱形式を解く必要があるので、problem の利用は有効である。各ステップで解く連立 1 次方程式の係数行列は共通であるので、LU 分解は一度だけやっておけば良い。次の例では , init=i によって、そのあたりを制御している (i が 0 のときだけ init が false になり、そうでないとき init は true である。init は「初期化済み」を意味するのでしょう。)。

heat.edp — problem で定義しておいて…

```
// heat.edp --- 正方形領域で熱方程式 u_t=u_{xx}+u_{yy}+f を解く
int i,m=100;
real Tmax=1, tau=0.01, t;
func f=1;
func g1=0;
func g2=0;
func u0=sin(pi*x)*sin(pi*y);
mesh Th=square(m,m);
plot(Th,wait=true);
fespace Vh(Th,P1);
//
Vh u=u0,up,v;
problem heat(u,v,solver=UMFPACK,init=i)=
  int2d(Th)(u*v/tau+dx(u)*dx(v)+dy(u)*dy(v))
 -int2d(Th)(f*v)
 -int1d(Th,2,3)(g2*v)
 -int2d(Th)(up*v/tau)
 +on(1,4,u=g1);
for (i=0; i < Tmax/tau; i++) {
  up=u;
  t=(i+1)*tau;
  heat;
  // plot(u,cmm="t="+t,wait=0,ps="heat"+i+".ps");
  plot(u,cmm="t="+t,wait=0);
}
plot(u,cmm="t="+t,wait=1,ps="heat.ps");
```

12.2 varf, matrix を用いて、連立１次方程式を作って解く

solve, problem を使うと、連立 1 次方程式 $Au = f$ が「見えない」けれど、以下のようにして varf を用いると係数行列 A、右辺の既知ベクトル f が得られる。そうするのがお勧めなのだそうだ。

(おまけ: この辺の説明を読んでいて、ようやく固有値問題を解くコードが理解出来るようになった。)

poisson3.edp

```
// poisson3.edp
//   Freefem++ poisson3.edp
border C(t=0,2*pi) {x=cos(t); y=sin(t);}
mesh Th = buildmesh(C(50));
fespace Vh(Th,P1);
Vh u,v;
func f=x*y;

varf a(u,v)= int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v)) + on(C,u=0);
matrix A=a(Vh,Vh);

varf L(unused,v) = int2d(Th)(f*v) + on(C,unused=0);
Vh F; F[] = L(0,Vh);

u[]=A^-1*F[];

plot(u);
```

このプログラムでは F は Vh の元としたが、次のように配列にしても良い。

```
real [int] F=L(0,Vh);
u[] = A^-1 * F;
```

連立 1 次方程式のソルバーとしては、次のようなものがある、とのことであるが、普通は UMFPACK を使うのだそうだ。どういうように使い分けるのかは、良く知らない (個々の用語は良く使われるものなので、一応は意味が分かるのだけれど、実際にどういう条件の下でどちらを選ぶかの判断が出来ない)。

solver=		対象とする行列
UMFPACK	multi-frontal LU	疎
CG	CG 法	疎, 正値対称
LU		skyline, 非対称
Crout	Crout 法	skyline, 対称
Cholesky	Cholesky 法	skyline, 対称, 正値
GMRES	GMRES 法	疎
sparsesolver		疎

反復法では、eps=ε で停止則を指定する。$\varepsilon > 0$ のときは

$$\|Ax - b\| < \frac{\varepsilon}{\|Ax_0 - b\|}$$

$\varepsilon < 0$ のときは

$$\|Ax - b\| < |\varepsilon|\,.$$

13 convect()

(準備中)

```
border C(t=0, 2*pi) { x=cos(t); y=sin(t); }
mesh Th = buildmesh(C(100));
fespace Uh(Th,P1);
Uh cold, c = exp(-10*((x-0.3)^2+(y-0.3)^2));
real dt = 0.17,t=0;
Uh u1 = y, u2 = -x;
for (int m=0; m<2*pi/dt ; m++) {
   t += dt;
   cold=c;
   c=convect([u1,u2],-dt,cold);
   plot(c,cmm=" t="+t + ", min=" + c[].min + ", max=" + c[].max, dim=3);
}
```

14 ファイル入出力 ofstream(), ifstream()

マニュアルの §4.11 に詳しい説明があるが、非常に簡単である (嬉しい)。

- `ofstream 変数名(ファイル名文字列);`
- `ofstream 変数名(ファイル名文字列, append);`
- `ifstream 変数名(ファイル名文字列);`

マニュアルの例では、カッコ { } でくくってブロックを作って、その中で変数を宣言している。ブロックを抜けるときにファイルがクローズされるということである (出力の場合は、書き出しが完了するわけだ)。なるほどとは思うけれど、気付かない人が多いんじゃないかな。まあ、プログラムが終了時にクローズされるんだろうけど。

実例を 2 つあげる。まず、「おまけ: gnuplot で可視化」[4] に載せた例。

[4] http://nalab.mind.meiji.ac.jp/~mk/labo/text/welcome-to-freefem/node6.html

プログラム例: BiharmonicEigenvalues4.edp

```
// BiharmonicEigenvalues4.edp
// 2012/8/2

load "Morley"
verbosity=1;
int i,NN;

cout << "input N: ";
cin >> NN;
cout << "N=" << NN << endl;

real sigma=0.3;
mesh Th=square(NN,NN);
fespace Vh(Th, P2Morley);
Vh [u,ux,uy], [v,vx,vy];

macro lap2(u,v) ((dxx(u)+dyy(u))*(dxx(v)+dyy(v))) // EOM
varf aa([u,ux,uy], [v,vx,vy]) = int2d(Th)
  (lap2(u,v)-(1-sigma)*(dxx(u)*dyy(v)+dyy(u)*dxx(v)-2.0*dxy(u)*dxy(v)));

varf bb([u,ux,uy], [v,vx,vy]) = int2d(Th)(u*v);

matrix A=aa(Vh,Vh,solver=UMFPACK);
matrix B=bb(Vh,Vh,solver=UMFPACK);

int nev=40;
real[int] ev(nev); // Stockage des valeurs propres
Vh[int] [eVu,eVux,eVuy](nev);  // Stockage des vecteurs propres

int k=EigenValue(A,B,sym=true,value=ev,vector=eVu,tol=1e-10,maxit=0,ncv=0);

{
  ofstream f("BiharmonicEigenvalues-" + NN + ".txt");
  f.precision(15);
  for (i = 0; i < nev; i++) {
    f << ev[i] << endl;
  }
}

for (i=0;i<nev;i++) {
  cout << ev[i] << endl;
  //  plot(eVu[i],[eVux[i],eVuy[i]],nbiso=64,fill=true,wait=true);
  plot(eVu[i],nbiso=64,fill=true,wait=true);
}
```

この例では f.precision(15); で表示桁数を指定している。

cout のときと同じように、f.scientific;, f.fixed;, f.showbase;, f.noshowbase;, f.showpos;, f.noshowpos;, f.default; などが使える (C++ を知っていれば意味は想像出来るであろう)。

14.1 有限要素空間の元 (変数) の内容の入出力

有限要素空間 Vh の変数 u に記録されているデータをファイル (ファイル名を “u.dat” とする) に出力するには、

```
mesh Th;
...
fespace Vh(Th,何か);
Vh u;
...
savemesh(Th,"Th.msh");
ofstream f("u.dat");
...
f << u[];
```

とする。メッシュデータ Th も保存しておくことを忘れずに。

読むときは

```
mesh Th=readmesh("Th.msh");
fespace Vh(Th,何か);
Vh u;
ifstream f("output.dat");
f >> u[];
```

とすれば良い。

```
// 境界の定義（単位円)，いわゆる正の向き
border Gamma(t=0,2*pi) { x=cos(t); y=sin(t); }
// 三角形要素分割を生成（境界を 50 に分割)
mesh Th = buildmesh(Gamma(50));
savemesh(Th,"poisson.msh");
plot(Th,wait=1,ps="poisson-mesh.eps");
// 有限要素空間は P1（区分的１次多項式）要素
fespace Vh(Th,P1);
Vh u,v;
// Poisson 方程式 -△u=f の右辺
func f = x*y;
// 現在時刻をメモ
real start = clock();
// 問題を解く
solve Poisson(u,v)
 = int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)
   +on(Gamma,u=0);
// 可視化
plot(u,ps="poisson.eps");

ofstream f("poisson.dat");
f << u[];
// 計算時間を表示
cout << " CPU time= " << clock() - start << endl;
```

```
mesh Th=readmesh("poisson.msh");
fespace Vh(Th,P1);
Vh u;

ifstream f("poisson.dat");
f >> u[];

plot(u,ps="poisson2.eps");
```

15 関数定義

FreeFem++での関数定義には 2 種類ある (名前がついているのかどうか)。

BASIC 言語を知っていたら、DEF 文で定義する関数と外部関数定義に似ている、ということで伝わるかもしれないんだけど…

15.1 $\mathbf{R}^2$ (の部分集合) で定義された 2 変数関数 $f(x,y)$ を 1 行で

`func 名前=x と y の式;` という形で定義する。

```
func f=x+2*y;
func chi=(-2 < x) * (x < -1) * (-3 < y) * (y < 3);
```

引数は x, y 固定ということなのか (調べていないので信じないように)。

条件式を利用することで場合分けが出来ることに注意する。

こうして定義した関数は、弱形式の中 (int2d(), int1d() とか +on() のカッコ内) で、名前単独 (f とか ch) で指定して、f*v とか u=chi; とかして使える。

15.2 いわゆる普通の関数

```
func real f(real x, real y)
{
  if (x<y)
    return x;
  else
    return y;
}
```

こういう関数は普通のプログラマーにとってわかりやすいが、例えば弱形式の中の式に (1 行関数のように) f*v のように直接使えない。その代わりに f(x,y)*v とする必要がある。

同様に、次のようにして有限要素空間の変数に代入することができる。

```
fespace(Th,P1) Vh;
Vh ff;
ff=f(x,y);
```

あるいは

```
fespace(Th,P2) Vh2;
Vh2 ff;
ff=f(x,y);
```

こうして代入したものは (当たり前であるが fespace である Vh, Vh2 の変数なので) 例えば `plot()` で描画したり、弱形式の中の `int2d()`, `int1d()` のカッコ内や、`+on()` のカッコ内で使える。

```
plot(ff, wait=1);

... int1d(Th,Gamma1)(ff*v) ...
... +on(Gamma2,u=ff)...
```

もっとも `int1d()` や `+on()` の中では、境界上の値しか必要にならないので、`ff` に代入して使うのはバカバカしいかもしれない。`plot(ff)` は意味があるけど。

16 疑似乱数

メルセンヌツイスター

`randinit()`

`randint32()`, `randint31()`, `randreal1()`, `randreal2()`, `randreal3()`, `randres53()`

17 ARGV

C や C++ を知っていると使いたくなる `argc`, `argv[]` に相当する `ARGV` がある。

`ARGV.n` が argc の代わりになる。また、`ARGV[]` が `argv[]` の代わりになる。ただし `ARGV[0]` は FreeFem++ プログラムを実行しているプログラムの名前 (`FreeFem++` であることが多い)。

mytest.edp

```
int i;
for (i=0; i< ARGV.n; i++)
    cout << i << " " << ARGV[i] << endl;
```

というプログラム `mytest.edp` があるとき

```
[katsurada-no-MacBook-Air:~/work] mk% FreeFem++ mytest.edp two 3 yon
```

とすると

```
0 FreeFem++
1 mytest.edp
2 two
3 3
4 yon
```

という出力を得る。

ARGV[n] は string であるが、数値に変換するには、atoi(), atof() を使えば良い。

```
int n;
real mu;
n = atoi(ARGV[2]);
mu = atof(ARGV[3]);
```

オプション -某 何とか の解析は getARGV(,) で出来る。

```
include "getARGV.idp";
...
int n=getARGV("-n", 10);
```

つまり -n 20 のような指定が出来て、変数 n に代入が出来る。指定が無かった場合は 10 がデフォールト値として 変数 n に代入される。

18 菊地 [4] の Poisson 方程式の例題を解く

菊地 [4] に載っている Poisson 方程式の例題

$$-\triangle u = f \quad \text{in } \Omega, \tag{1}$$

$$u = g_1 \quad \text{in } \Gamma_1, \tag{2}$$

$$\frac{\partial u}{\partial n} = g_2 \quad \text{in } \Gamma_2 \tag{3}$$

(ただし、$\Omega = (0,1)\times(0,1)$, $\Gamma_1 = \{0\}\times[0,1]\cup[0,1]\times\{0\}$, $\Gamma_2 = \{1\}\times(0,1]\cup(0,1]\times\{1\}$, $f = 1$, $g_1 = 0$, $g_2 = 0$) を FreeFem++ を用いて解くとどうなるか。

kikuchi-poisson.edp

```
// kikuchi-poisson.edp
int Gamma1=1, Gamma2=2;
border Gamma10(t=0,1) { x=0;   y=1-t; label=Gamma1; }
border Gamma11(t=0,1) { x=t;   y=0;   label=Gamma1; }
border Gamma20(t=0,1) { x=1;   y=t;   label=Gamma2; }
border Gamma21(t=0,1) { x=1-t; y=1;   label=Gamma2; }
int m=10;
mesh Th = buildmesh(Gamma10(m)+Gamma11(m)+Gamma20(m)+Gamma21(m));
plot(Th, wait=1,ps="Th.eps");
savemesh(Th,"Th.msh"); // optional
fespace Vh(Th,P1);
Vh u,v;
func f=1;
func g1=0;
func g2=0;
solve Poisson(u,v) =
   int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
  -int2d(Th)(f*v)
  -int1d(Th,Gamma2)(g2*v)
  +on(Gamma1,u=g1);
plot(u,ps="contour.eps");
```

問　同じ問題を square() を使って解くプログラムを作成せよ (そうすると菊地先生の本と同じ三角形分割になる？)。

square でメッシュ分割したとき、下の辺、右の辺、上の辺、左の辺に 1, 2, 3, 4 というラベルがつくことが大事なポイントである。

poisson-kikuchi-square.edp

```
// poisson-kikuchi-square.edp
int m=10;
mesh Th = square(m,m);
plot(Th, wait=1,ps="Th-square.eps");
savemesh(Th,"Th-square.msh"); // optional
fespace Vh(Th,P1);
Vh u,v;
func f=1;
func g1=0;
func g2=0;
solve Poisson(u,v) =
   int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
  -int2d(Th)(f*v)
  -int1d(Th,2,3)(g2*v) // Gamma20,Gamma21
  +on(1,4,u=g1); // on(Gamma10,Gamma11,u=g1)
plot(u,ps="contour.eps");
```

A　C++のストリーム入出力

FreeFem++ の入出力は、C++のストリーム入出力の機能に良く似ている (似ているけれど同じではない。同じにすれば良いのに。)。

ここでは C++ のストリーム入出力機能の大まかな説明を行う。

A.1　標準入力 cin, 標準出力 cout, 標準エラー出力 cerr

C++のソース・プログラムで次のようにしてあることを仮定する。

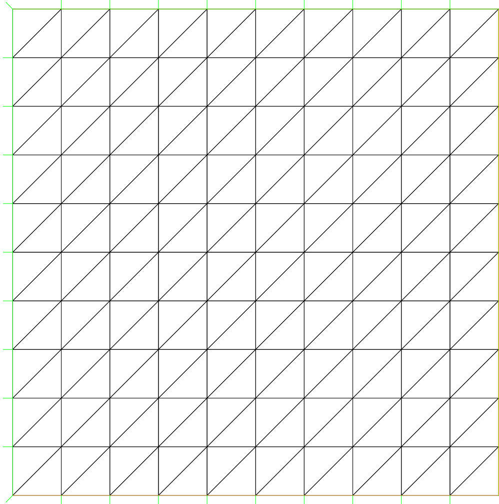


図 6: square() を用いたメッシュ分割

```
#include <iostream> // C の <stdio.h> に相当するような定番
#include <iomanip>  // setprecision() 等に必要
using namespace std;// こうしないと std::cin, std::cout, std::cerr とする必要
```

通常は、標準入力は端末のキーボードからの入力、標準出力は端末 (ターミナル) の画面への文字出力、標準エラー出力も端末の画面への文字出力、に結びつけられている (入出力のリダイレクトで、指定したファイルに結びつけることもできる)。

とりあえず、C 言語のプログラムの printf() を使う代わりに cout << 式, scanf() を使う代わりに cin >> 変数名 を使う、と覚える。

```
double a, b;
cout << "Hello, world" << endl; // endl は改行 \n である。
cout << "Please input two numbers: ";
cin >> a >> b;
cout << "a+b=" << a + b << ", a-b=" << a - b << ", a*b=" << a * b
     << ", a/b=" << a / b << endl;
```

A.2 数値の書式指定

C 言語の printf() での書式指定 "%4d", "%7.2f", "%20.15e", "%25.15g" は、C++ で使うのはあきらめることを勧める。

- 幅の指定は << setw(桁数) で行う。これは次のフィールドにしか影響しない (必要ならば毎回指定する)。
- 浮動小数点数の小数点以下の桁数の指定は << setprecision(桁数) で行う。
- 浮動小数点数で固定小数点形式での出力の指定は、<< fixed で行う。
  (C 言語の %f に相当)
- 浮動小数点数で指数形式での出力の指定は、<< scientific で行う。
  (C 言語の %e に相当)

- 浮動小数点数でデフォールト形式での出力の指定は、<< defaultfloat で行う。
(C 言語の %g に相当)

```
// testfloat.cpp --- ナンセンスな計算 (円周率とアボガドロ数の積)
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main(void)
{
  double pi, NA;
  pi = 4.0 * atan(1.0);
  NA = 6.022e+23;
  cout << setprecision(15); //double は 10 進 16 桁弱なので。cout.precision(15); も
可
  cout << fixed;
  cout << "π=" << setw(20) << pi << ", NA=" << setw(20) << NA
       << ", π Na=" << setw(20) << pi * NA << endl; // %20.15f 相当
  cout << scientific;
  cout << "π=" << setw(24) << pi << ", NA=" << setw(24) << NA
       << ", π Na=" << setw(24) << pi * NA << endl; // %24.15e 相当
  cout << defaultfloat;
  cout << "π=" << setw(20) << pi << ", NA=" << setw(20) << NA
       << ", π Na=" << setw(20) << pi * NA << endl; // %20.15g 相当
}
```

(実行してみると分かるが、意外と難しい…)

A.2.1 外部ファイルとの入出力

```
// sintable.cpp --- 0 度から 90 度までの sin の表 sin.txt を作る
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
using namespace std;

int main(void)
{
  int n=90;
  double x, dx, pi;
  pi = 4.0 * atan(1.0);
  dx = (pi / 2) / n;
  {
    ofstream out("sin.txt");
    out << fixed << setprecision(15); // %.15f
    for (int i = 0; i <= n; i++)
      out << setw(3) << i << setw(20) << sin(i * dx) << endl;
  } // ブロックを抜けるとファイルがクローズされる
  return 0;
}
```

B FreeFem++-nw — バッチ処理向きの (ffglut を呼ばない) FreeFem++

FreeFem++ で `plot()` を用いると、ffglut というプログラムを呼び出してグラフィックス描画が行われる。ffglut はユーザーが ESC キーをタイプすることではじめて終了する、FreeFem++ は ffglut が終了するまで待つ、という仕様になっているため、バッチ処理 (例えばシェル・スクリプトから FreeFem++ を起動する) では、FreeFem++ プログラムが終了できないことになる。

このような場合、FreeFem++ の代わりに FreeFem++-nw を使うとうまく行くことがある。これは FreeFem++ プログラム中に `plot()` があっても、ffglut は呼び出さず、`ps="ファイル名"` があった時に、PostScript ファイルの作成だけを行う。

`-nw` は no window (ウィンドウを出さない) という意味かな？

C MPI 対応バージョン

MacOS X 10.6, 10.7 に対しては、それぞれ

- http://www.ljll.math.upmc.fr/~hecht/ftp/FF-conf/InstallMac10.6.html
- http://www.ljll.math.upmc.fr/~hecht/ftp/FF-conf/InstallMac10.7.html

にインストール法が書いてある。

余談　2012/10/6, 電通大に Frederic Hecht がやってきて、MPI 対応バージョンの説明＆デモをしてくれた。他の人の講演中も MacBook で FreeFem++ のコンパイルをしていた (ちなみに Mac で Windows 用の FreeFem++ をコンパイルしていました)。

D　Changelog

遅ればせながら記録を書くことに。

- (2012/7/1) 一念発起マニュアルを読んで言語仕様を調べ始める。
- (2012/7/3) 数理計算特論の授業で解説。その後のこのファイルを作成する。
- (2012/8/2) 修士のゼミでプログラムを作っていて、`ofstream`, `cin`, `f.precision(15);` 等々、色々細かいことが分かる。とりあえずサンプル・プログラムの形でこの文書に取り込む。
- (2012/11/30) MPI 版についての情報を書く。
- (2012/11/30) 有限要素空間の元を表す変数があるとき、領域内の任意の点における値を補間して計算してくれる (便利です) ことを書き足す。
- (2012/11/30) アクセスフリーな場所に置くことにした。
- (2015/2/14) メッシュの説明を書いていなかったので、[5] から持って来る (少し加筆)。
- (2015/5/15) string を数値に変換しようとしてはまる。`atoi()`, `atof()` があった。こういうのマニュアルを見ても、ネットを検索しても分からない。どうにかなんないのかな。
- 有限要素空間の変数の入出力。オブジェクト指向は楽だ。
- (2021/12/13) 2020 年度は COVID19 のせいで、応用数値解析特論でオンデマンド資料を用意した。2021 年度も「対面授業」という名のハイブリッド授業をしている。そのために作った説明をこちらに追加した。

参考文献

[1] 大塚厚二, 高石武史：有限要素法で学ぶ現象と数理 — FreeFem++数理思考プログラミング —, 共立出版 (2014), https://sites.google.com/a/comfos.org/comfos/ffempp というサポート WWW サイトがある．Maruzen eBook に入っているので、 https://elib.maruzen.co.jp/elib/html/BookDetail/Id/3000018545 でアクセス出来る．

[2] Suzuki, A.: Finite element programming by FreeFem++ —intermediate course, 日本応用数理学会「産業における応用数理」研究部会のソフトウェアセミナー 「FreeFem++ による有限要素プログラミング — 中級編 —」(2016/2/11-12) の配布資料で、 https://www.ljll.math.upmc.fr/~suzukia/FreeFempp-tutorial-JSIAM2016/ から入手できる (2016).

[3] Suzuki, A.: Finite element programming by FreeFem++ —advanced course, 日本応用数理学会「産業における応用数理」研究部会のソフトウェアセミナー 「FreeFem++ による有限要素プログラミング — 中級編 —」(2016/6/4–5) の配布資料で、 https://www.ljll.math.upmc.fr/~suzukia/FreeFempp-tutorial-JSIAM2016b/ から入手できる (2016).

[4] 菊地文雄：有限要素法概説, サイエンス社 (1980), 新訂版 1999.

[5] 桂田祐史：FreeFEM++　の紹介, https://m-katsurada.sakura.ne.jp/labo/text/welcome-to-freefem/ (2007〜).