応用数値解析特論 第 7 回

～C 言語によるサンプル・プログラム, FreeFem++文法入門～

かつらだ まさし
桂田 祐史

http://nalab.mind.meiji.ac.jp/~mk/ana2021/

2021 年 11 月 8 日

目次

本日の内容

本日は二本立て。

- 菊地 [1] 掲載のプログラムを元にしたサンプル・プログラム (C 言語で記述) の紹介
- FreeFem++ の文法の説明

6 C 言語によるサンプル・プログラムの紹介

6.1 進行表

6 C 言語によるサンプル・プログラムの紹介

6.1 進行表

1. 百聞は一見しかず。まず実行例を見てもらう。

6 C 言語によるサンプル・プログラムの紹介

6.1 進行表

1. 百聞は一見しかず。まず実行例を見てもらう。
2. プログラムが何をするか、入力と出力を理解する。

   有限要素解を求めるプログラム (`naive`, `band`) では、領域や三角形分割の情報を入力データとする。そのため一般性が高くなっている。

6 C 言語によるサンプル・プログラムの紹介

6.1 進行表

1. 百聞は一見しかず。まず実行例を見てもらう。
2. プログラムが何をするか、入力と出力を理解する。
   有限要素解を求めるプログラム (`naive`, `band`) では、領域や三角形分割の情報を入力データとする。そのため一般性が高くなっている。
3. naive と band の比較をする。数学的にはやること同じ。効率の違いは？

6 C 言語によるサンプル・プログラムの紹介

6.1 進行表

1. 百聞は一見しかず。まず実行例を見てもらう。
2. プログラムが何をするか、入力と出力を理解する。
   有限要素解を求めるプログラム (`naive`, `band`) では、領域や三角形分割の情報を入力データとする。そのため一般性が高くなっている。
3. naive と band の比較をする。数学的にはやること同じ。効率の違いは？
4. プログラムの心臓部分 `assem()` と `ecm()` の読解 (説明したことの確認)。

6.2 試しに実行

参考　授業 WWW サイトの「有限要素法のサンプル C プログラム」

入手、展開、ファイル名確認

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/fem/kikuchi-fem-mac.tar.gz
tar xzf kikuchi-fem-mac.tar.gz
cd kikuchi-fem-mac
ls
```

6.2 試しに実行

参考　授業 WWW サイトの「有限要素法のサンプル C プログラム」

入手、展開、ファイル名確認

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/fem/kikuchi-fem-mac.tar.gz
tar xzf kikuchi-fem-mac.tar.gz
cd kikuchi-fem-mac
ls
```

とりあえず動作チェック (実行には、cc, cglsc, make 等が必要)

コンパイル&テスト

`make`	プログラムのコンパイル
`make test1`	naive の動作確認 (辺を 2,4,8 分割したときの有限要素解の数値データ)
`make test2`	band の動作確認 (辺を 2,4,8 分割したときの有限要素解の数値データ)
`make test3`	band の動作確認 (辺を 2,4,8,16,32 分割したときの有限要素解の等高線表示)

6.2 試しに実行

参考　授業 WWW サイトの「有限要素法のサンプル C プログラム」

入手、展開、ファイル名確認

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/fem/kikuchi-fem-mac.tar.gz
tar xzf kikuchi-fem-mac.tar.gz
cd kikuchi-fem-mac
ls
```

とりあえず動作チェック (実行には、cc, cglsc, make 等が必要)

コンパイル&テスト

`make`	プログラムのコンパイル
`make test1`	naive の動作確認 (辺を 2,4,8 分割したときの有限要素解の数値データ)
`make test2`	band の動作確認 (辺を 2,4,8 分割したときの有限要素解の数値データ)
`make test3`	band の動作確認 (辺を 2,4,8,16,32 分割したときの有限要素解の等高線表示)

途中で引っかかった場合、相談して下さい。

もしかすると、今の院生の Mac には cglsc がインストールされていないかも。その場合は `make test3` は実行できない。

ソースプログラム `naive.c`, `band.c` は、それぞれ 320 行、397 行である。

6.3 有限要素解を求めるプログラム naive, band の理解

2 次元多角形領域 Ω における Poisson 方程式の同次 Dirichlet, Neumann 境界値問題

$$-\triangle u(x,y) = f(x,y) := 1 \quad ((x,y) \in \Omega), \tag{1}$$

$$u(x,y) = g_1(x,y) := 0 \quad ((x,y) \in \Gamma_1), \tag{2}$$

$$\frac{\partial u}{\partial \boldsymbol{n}}(x,y) = g_2(x,y) := 0 \quad ((x,y) \in \Gamma_2) \tag{3}$$

を有限要素法で解くプログラムである。

6.3 有限要素解を求めるプログラム naive, band の理解

2 次元多角形領域 Ω における Poisson 方程式の同次 Dirichlet, Neumann 境界値問題

$$-\triangle u(x,y) = f(x,y) := 1 \quad ((x,y) \in \Omega), \tag{1}$$

$$u(x,y) = g_1(x,y) := 0 \quad ((x,y) \in \Gamma_1), \tag{2}$$

$$\frac{\partial u}{\partial \boldsymbol{n}}(x,y) = g_2(x,y) := 0 \quad ((x,y) \in \Gamma_2) \tag{3}$$

を有限要素法で解くプログラムである。

Q ここで Ω, Γ_1, Γ_2 は何か？

6.3 有限要素解を求めるプログラム naive, band の理解

2 次元多角形領域 Ω における Poisson 方程式の同次 Dirichlet, Neumann 境界値問題

$$-\triangle u(x,y) = f(x,y) := 1 \quad ((x,y) \in \Omega), \tag{1}$$

$$u(x,y) = g_1(x,y) := 0 \quad ((x,y) \in \Gamma_1), \tag{2}$$

$$\frac{\partial u}{\partial \boldsymbol{n}}(x,y) = g_2(x,y) := 0 \quad ((x,y) \in \Gamma_2) \tag{3}$$

を有限要素法で解くプログラムである。

Q ここで Ω, Γ_1, Γ_2 は何か？

A 実は Ω, Γ_1, Γ_2 についてはデータとして入力する。

naive, band ともに、任意の領域&境界についての計算ができる。

(f, g_1, g_2 については、簡単のため、特殊な値 1, 0, 0 が仮定されている。これを一般化するのは適度の演習問題である。)

6.3 有限要素解を求めるプログラム naive, band の理解

入力データの例 `input.dat`

```
9     8     5
0.0         0.0
0.0         0.5
0.0         1.0
0.5         0.0
0.5         0.5
0.5         1.0
1.0         0.0
1.0         0.5
1.0         1.0
 0     3     4     0     4     1
 1     4     5     1     5     2
 3     6     7     3     7     4
 4     7     8     4     8     5
 0     1     2     3     6
```

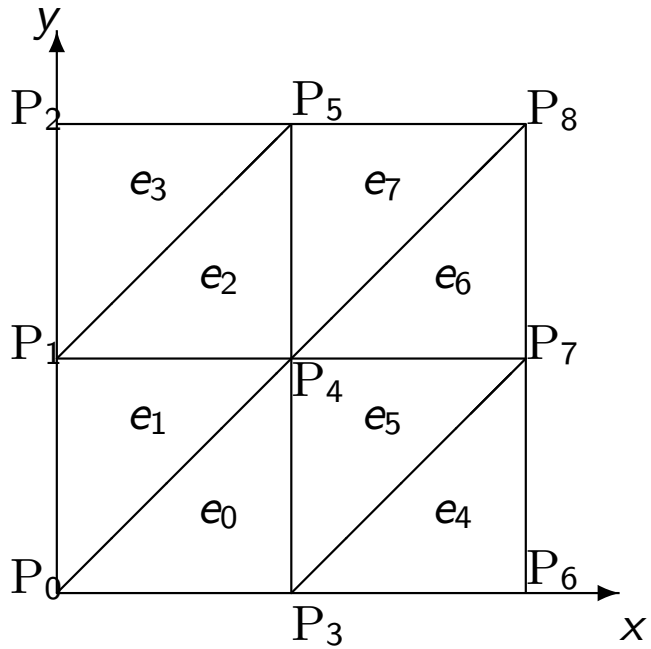


図 1: 要素分割 (各辺を 2 等分してから要素分割)

1. 1 行目には、節点数 (nnode)、要素数 (nelmt)、Γ_1 に属している節点数 (nbc)
2. 2〜10 行は、節点の座標 (x_i, y_i) $(i = 0, 1, \cdots, \mathtt{nnode} - 1)$
3. 11〜14 行は、各要素を構成する節点の全体節点番号 (0 から nelmt−1 までの通し番号)
   節点は各要素を左回りに回るように順序付けてある。
4. 最後に Γ_1 に属する節点の全体節点番号

6.3 有限要素解を求めるプログラム naive, band の理解

この形式のデータがあれば、図が描ける (幾何的状況が分かる) ことを理解しよう。

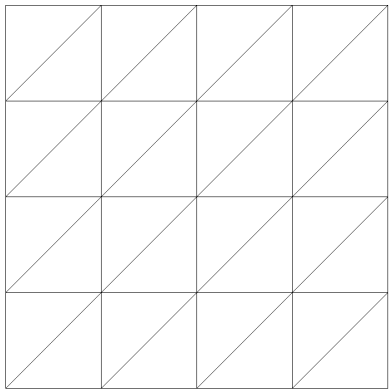
三角形と (結果として)Ω を描く

```
./disp-glsc input.dat
./disp-glsc input4.dat
cat input4.dat | ./disp-glsc
./make-input | ./disp-glsc
```

(最後のコマンドに対して、辺を何等分するか、数値 (例えば 64 とか) を入力しよう。)

コマンド 1 | コマンド 2 でコマンド 1 の出力をコマンド 2 に入力できる (パイプ機能)。

`disp-glsc` は上の形式のデータを図示するプログラム、`make-input` は正方形領域に対して上の形式のデータを作成するプログラムである。



6.3 有限要素解を求めるプログラム naive, band の理解

naive, band は上の形式の入力データから、有限要素解を計算するプログラム。

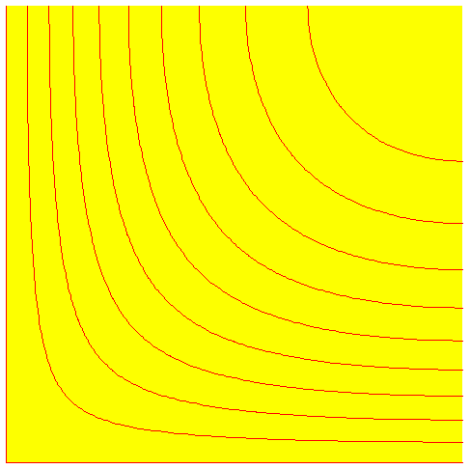
両者は同じ計算を行う。連立 1 次方程式の係数行列が**帯行列** (band matrix) であることを利用して、計算の効率化の工夫をしたのが band で、それをしないのが naive である。

一辺 64 分割で解き比べ (CPU 時間計測), 解の等高線表示

```
echo 64 | ./make-band-input > input64.dat
./disp-glsc input64.dat
time ./naive input64.dat
time ./band input64.dat
```

あるマシンで 17.5 秒 vs 0.02 秒　(naive は実際的ではない, ちなみに節点数 4225)

```
./contour band.out
```



6.4 プログラム naive の内部構造

主な関数には以下のようなものがある。

`main()`	
`input()`	入力データ読み込み
`assem()`	全体係数行列 A, 全体自由項ベクトル f の計算 (直接剛性法)
`ecm()`	要素係数行列 A_e, 要素自由項ベクトル f_e の計算
`solve()`	
`output()`	節点パラメーター (節点での解の値) を出力
`f()`	Poisson 方程式 $-\triangle u = f$ の右辺の既知関数 f

6.4 プログラム naive の内部構造

主な関数には以下のようなものがある。

`main()`	
`input()`	入力データ読み込み
`assem()`	全体係数行列 A, 全体自由項ベクトル f の計算 (直接剛性法)
`ecm()`	要素係数行列 A_e, 要素自由項ベクトル f_e の計算
`solve()`	
`output()`	節点パラメーター (節点での解の値) を出力
`f()`	Poisson 方程式 $-\triangle u = f$ の右辺の既知関数 f

主な変数名

`nnode`	節点の総数
`nelmt`	有限要素の総数
`nbc`	Γ_1 (Dirichlet 境界条件を課す) 上の節点の総数
`x[nnode], y[nnode]`	節点の座標
`ielmt[nelmt].node[3]`	各有限要素を構成する節点の番号
`ibc[nbc]`	基本境界条件を課す節点の番号
`am[][]`	全体係数行列
`fm[]`	全体自由項ベクトル

6.4 プログラム naive の内部構造　assem()

assem() は連立 1 次方程式を組み立てる関数。

$A^* := \sum_{k=0}^{N_e-1} A_k^*$ と $\boldsymbol{f}^* := \sum_{k=0}^{N_e-1} \boldsymbol{f}_k^*$ を次のように計算する。

```
/* assemblage of total matrix and vector; */
    for (k = 0; k < nelmt; k++) {
        ecm(k, ielmt, x, y, ae, fe);
        for (i = 0; i < 3; i++) {
            ii = ielmt[k].node[i];
            fm[ii] += fe[i];
            for (j = 0; j < 3; j++) {
                jj = ielmt[k].node[j];
                am[ii][jj] += ae[i][j];
            }
        }
    }
```

ielmt[k].node[i] は、要素 e_k の、局所節点番号が i の節点 N_i の全体節点番号

6.4 プログラム naive の内部構造　ecm()

ecm() は要素係数行列 A_k、要素自由項ベクトル $\boldsymbol{f}_k$ を求める関数。

```
/* 節点の座標を求める */
for (i = 0; i < 3; i++) {
    j = ielmt[k].node[i];
    xe[i] = x[j];
    ye[i] = y[j];
}
```

節点の座標さえ求まれば、A_k, $\boldsymbol{f}_k$ の成分は公式に従って計算するだけである。(それを確かめたければ、naive.c あるいは band.c を見よ。)

6.5 参考課題

以前、FreeFem++ が使えなかった頃は、授業で次のような課題を出していた。

私は「百見は一験にしかず」と考えていて、次のような実験をすることは有益と思っているが、この科目では要求しない。

(a) このプログラムで解ける問題は、境界条件が同次境界条件

$$u = 0 \quad \text{on } \Gamma_1, \qquad \frac{\partial u}{\partial \boldsymbol{n}} = 0 \quad \text{on } \Gamma_2$$

であるが、これを非同次境界条件

$$u = g_1 \quad \text{on } \Gamma_1, \qquad \frac{\partial u}{\partial \boldsymbol{n}} = g_2 \quad \text{on } \Gamma_2$$

に変える。

(b) 自分で選んだ領域を三角形分割して、このプログラムに入力できるデータを生成するプログラムを書く。

8 FreeFem++の文法

8.1 はじめに

前回 FreeFem++ のインストール手順と簡単な解説を行った。

8 FreeFem++の文法

8.1 はじめに

前回 FreeFem++ のインストール手順と簡単な解説を行った。

FreeFem++ は有限要素法によって微分方程式の数値シミュレーションを行うためのソフトウェアであり、言語処理系である (Hecht [2])。

インタープリターである (その点は MATLAB や Python と似ている)。

8 FreeFem++の文法

8.1 はじめに

前回 FreeFem++ のインストール手順と簡単な解説を行った。

FreeFem++ は有限要素法によって微分方程式の数値シミュレーションを行うためのソフトウェアであり、言語処理系である (Hecht [2])。
インタープリターである (その点は MATLAB や Python と似ている)。

今回は、プログラミング言語としての FreeFem++を説明する (マニュアルを見ても良く分からない — 少なくとも私は)。

8 FreeFem++の文法

8.1 はじめに

前回 FreeFem++ のインストール手順と簡単な解説を行った。

FreeFem++ は有限要素法によって微分方程式の数値シミュレーションを行うためのソフトウェアであり、言語処理系である (Hecht [2])。
インタープリターである (その点は MATLAB や Python と似ている)。

今回は、プログラミング言語としての FreeFem++を説明する (マニュアルを見ても良く分からない — 少なくとも私は)。

FreeFem++のことを「有限要素法専用ツール」と考える人もいる。確かに有限要素法に便利な命令が組み込まれているが、それ以外の目的のプログラミングに必要な機能も十分に備わっている (実際、有限体積法や差分法のプログラムも記述可能である)。効率を度外視すれば、C のようなプログラミング言語で出来ることは FreeFem++でも出来る、と考えよう。

8 FreeFem++の文法

8.1 はじめに

前回 FreeFem++ のインストール手順と簡単な解説を行った。

FreeFem++ は有限要素法によって微分方程式の数値シミュレーションを行うためのソフトウェアであり、言語処理系である (Hecht [2])。
インタープリターである (その点は MATLAB や Python と似ている)。

今回は、プログラミング言語としての FreeFem++を説明する (マニュアルを見ても良く分からない — 少なくとも私は)。

FreeFem++のことを「有限要素法専用ツール」と考える人もいる。確かに有限要素法に便利な命令が組み込まれているが、それ以外の目的のプログラミングに必要な機能も十分に備わっている (実際、有限体積法や差分法のプログラムも記述可能である)。効率を度外視すれば、C のようなプログラミング言語で出来ることは FreeFem++でも出来る、と考えよう。

文法は、C++に似ている (ゆえに C にも似ている)。C しか知らない人は、C++のストリームを使った入出力 (cout, cin の利用) を調べておくこと。
参考: FreeFem++ は C++ で記述されている。

8 FreeFem++の文法

8.1 はじめに

前回 FreeFem++ のインストール手順と簡単な解説を行った。

FreeFem++ は有限要素法によって微分方程式の数値シミュレーションを行うためのソフトウェアであり、言語処理系である (Hecht [2])。
インタープリターである (その点は MATLAB や Python と似ている)。

今回は、プログラミング言語としての FreeFem++を説明する (マニュアルを見ても良く分からない — 少なくとも私は)。

FreeFem++のことを「有限要素法専用ツール」と考える人もいる。確かに有限要素法に便利な命令が組み込まれているが、それ以外の目的のプログラミングに必要な機能も十分に備わっている (実際、有限体積法や差分法のプログラムも記述可能である)。効率を度外視すれば、C のようなプログラミング言語で出来ることは FreeFem++でも出来る、と考えよう。

文法は、C++に似ている (ゆえに C にも似ている)。C しか知らない人は、C++のストリームを使った入出力 (cout, cin の利用) を調べておくこと。
参考: FreeFem++ は C++ で記述されている。

マニュアル Hect[3] は事例集の性格が強く、言語仕様は整理した形では載っていない。以下の説明は、個人的なノートである桂田 [4] に基づく。

8.1 はじめに　基本的な Poisson 方程式のプログラム

```
// poisson.edp
// 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));
plot(Th,wait=true); // plot(Th,wait=true,ps="Th.eps");
// 有限要素空間は P1（区分的１次多項式）要素
real [int] levels =-0.012:0.001:0.012;
fespace Vh(Th,P1);
Vh u,v;
// Poisson 方程式 -△ u=f の右辺
func f = x*y;
// 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（等高線）
plot(u,wait=true);
//plot(u,viso=levels,fill=true,wait=true);
// 可視化（3 次元）--- マウスで使って動かせる
plot(u,dim=3,viso=levels,fill=true,wait=true);
```

→ 独特の命令ばかりで、汎用のプログラミング言語の機能があることは分かりにくい。

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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)

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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 ;

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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).

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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。

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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。

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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。

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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。

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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。

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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

8.2 汎用のプログラミング機能

8.2.1 C 言語と良く似ているところ

改めて数えるととても多い。

- // 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…

8.2.2 データ型

8.2.2 データ型

- 整数を表すための int がある (C 言語の `int` に相当)

8.2.2 データ型

- 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)

8.2.2 データ型

- 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`)

8.2.2 データ型

- 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`)
- 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。
  例えば `plot(u,wait=true);` は `plot(u,wait=1);` と同じ。

8.2.2 データ型

- 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`)
- 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。
  例えば `plot(u,wait=true);` は `plot(u,wait=1);` と同じ。
- 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？)。

8.2.2 データ型

- 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`)
- 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。
  例えば `plot(u,wait=true);` は `plot(u,wait=1);` と同じ。
- 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。

8.2.2 データ型

- 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`)
- 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。
  例えば `plot(u,wait=true);` は `plot(u,wait=1);` と同じ。
- 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;
```

8.2.2 データ型

- 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`)
- 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。
  例えば `plot(u,wait=true);` は `plot(u,wait=1);` と同じ。
- 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)。

8.2.3 配列型　1 次元

1 次元配列は、C 言語に (少し) 似ている。

```
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。

8.2.3 配列型　2 次元

2 次元配列はかなり違う。要素にアクセスするには 名前 (i,j) とする。

```
real[int,int] kuku(9,9);
int i,j;
for (i=0; i<kuku.n; i++) {
  for (j=0; j<kuku.m; j++) {
      kuku(i,j)=(i+1)*(j+1);
      cout << setw(3) << kuku(i,j);
  }
  cout << endl;
}
cout << kuku << endl;

real[int,int] kuku2=[[1,2,3,4,5,6,7,8,9],
                     [2,4,6,8,10,12,14,16,18],
                     [3,6,9,12,15,18,21,24,27],
                     [4,8,12,16,20,24,28,32,36],
                     [5,10,15,20,25,30,35,40,45],
                     [6,12,18,24,30,36,42,48,54],
                     [7,14,21,28,35,42,49,56,63],
                     [8,16,24,32,40,48,56,64,72],
                     [9,18,27,36,45,54,63,72,81]];
cout << kuku2 << endl;
```

8.3 有限要素法のための機能

8.3.1 有限要素法のプログラムの構成

有限要素法のプログラムと言っても色々あるが (この科目では、熱方程式や波動方程式などの発展方程式、流体力学の方程式、固有値問題などを取り上げる)、2 次元領域における Poisson 方程式の境界値問題のプログラムは基本的と考えられる。

1. 領域の定義と領域の三角形分割
2. 有限要素空間
3. 弱形式を次のいずれかで定義して解く。
   a. `solve()`
      弱形式を与えると同時にそれを解く (弱解を求める)。
   b. `problem()`
      弱形式を与えて問題を解く関数を定義する。発展問題で便利。
   c. `varf`, `matrix`
      弱形式を与えて連立 1 次方程式を作る。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

- 2 次元 (有界) 領域の多くは、その境界曲線を定義することで定まる。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

- 2 次元 (有界) 領域の多くは、その境界曲線を定義することで定まる。
- 境界曲線は `border` と言う型の変数として定義される。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

- 2 次元 (有界) 領域の多くは、その境界曲線を定義することで定まる。
- 境界曲線は `border` と言う型の変数として定義される。
- 三角形分割は `mesh` と言う型の変数として定義される。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

- 2 次元 (有界) 領域の多くは、その境界曲線を定義することで定まる。
- 境界曲線は `border` と言う型の変数として定義される。
- 三角形分割は `mesh` と言う型の変数として定義される。
- `buildmesh()` と言う関数は、各 border を何等分するか指定することで、border の囲む領域を三角形分割して、mesh 型のデータを作る。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

- 2 次元 (有界) 領域の多くは、その境界曲線を定義することで定まる。
- 境界曲線は `border` と言う型の変数として定義される。
- 三角形分割は `mesh` と言う型の変数として定義される。
- `buildmesh()` と言う関数は、各 border を何等分するか指定することで、border の囲む領域を三角形分割して、mesh 型のデータを作る。
- `mesh` 型のデータは、`readmesh()`, `writemesh()` と言う関数を用いて入出力できる (結果はテキスト・ファイル)。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

- 2 次元 (有界) 領域の多くは、その境界曲線を定義することで定まる。
- 境界曲線は `border` と言う型の変数として定義される。
- 三角形分割は `mesh` と言う型の変数として定義される。
- `buildmesh()` と言う関数は、各 border を何等分するか指定することで、border の囲む領域を三角形分割して、mesh 型のデータを作る。
- `mesh` 型のデータは、`readmesh()`, `writemesh()` と言う関数を用いて入出力できる (結果はテキスト・ファイル)。
- `mesh` 型のデータは、`plot()` により可視化できる。

8.3.2 領域の定義と領域の三角形分割 (2 次元の場合)

問題を考える領域を定義し、三角形分割をすることが必要である。

- 2 次元 (有界) 領域の多くは、その境界曲線を定義することで定まる。
- 境界曲線は `border` と言う型の変数として定義される。
- 三角形分割は `mesh` と言う型の変数として定義される。
- `buildmesh()` と言う関数は、各 border を何等分するか指定することで、border の囲む領域を三角形分割して、mesh 型のデータを作る。
- `mesh` 型のデータは、`readmesh()`, `writemesh()` と言う関数を用いて入出力できる (結果はテキスト・ファイル)。
- `mesh` 型のデータは、`plot()` により可視化できる。
- 矩形領域 (辺が座標軸に平行な長方形) は、`square()` と言う命令で mesh 型データが作れる (参考「FreeFem++ノート」)。

8.3.2 領域の定義と領域の三角形分割　border

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; }
```

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; }
```

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

```
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");
```

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

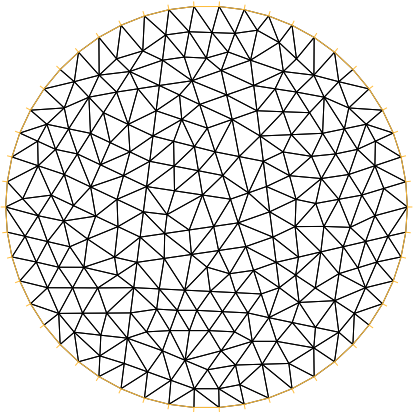


図 2: C(50)

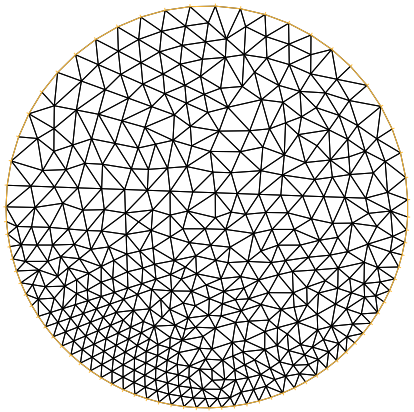


図 3: Gamma1(25)+Gamma2(50)

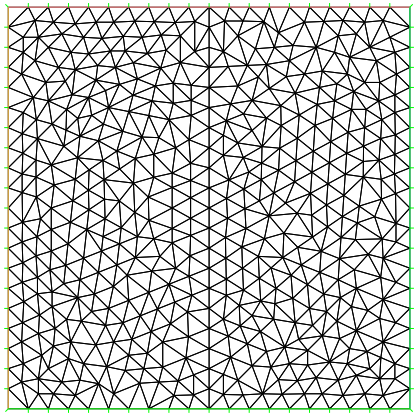


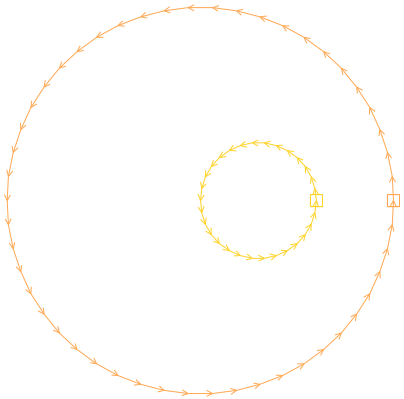
図 4: C1(20)+C2(20)+...

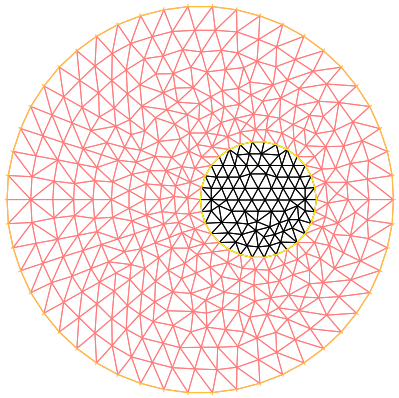
8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

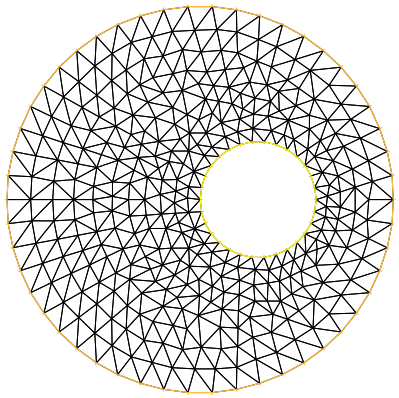
有限個の Jordan 閉曲線で囲まれた多重連結領域を三角形分割することもできる。

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");
```







8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

普通は mesh データの細かいことは見る必要がないかもしれないが…

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

普通は mesh データの細かいことは見る必要がないかもしれないが…

- `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。

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

普通は mesh データの細かいことは見る必要がないかもしれないが…

- `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。

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

普通は mesh データの細かいことは見る必要がないかもしれないが…

- `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。
- `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。

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

普通は mesh データの細かいことは見る必要がないかもしれないが…

- `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。
- `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。

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

次のような場合に `readmesh()`, `writemesh()` は有効である。

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

次のような場合に `readmesh()`, `writemesh()` は有効である。

1. FreeFem++ を用いて三角形分割を行い、得られたメッシュ・データーを外部のプログラムで利用する (有限要素法の計算は自作プログラムで行う等)。

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

次のような場合に `readmesh()`, `writemesh()` は有効である。

1. FreeFem++ を用いて三角形分割を行い、得られたメッシュ・データーを外部のプログラムで利用する (有限要素法の計算は自作プログラムで行う等)。
2. 自作のプログラムで三角形分割を行い、そのメッシュ・データーを FreeFem++ で利用する。

8.3.2 領域の定義と領域の三角形分割　メッシュ(mesh)

次のような場合に `readmesh()`, `writemesh()` は有効である。

1. FreeFem++ を用いて三角形分割を行い、得られたメッシュ・データーを外部のプログラムで利用する (有限要素法の計算は自作プログラムで行う等)。
2. 自作のプログラムで三角形分割を行い、そのメッシュ・データーを FreeFem++ で利用する。

`readmesh()`, `writemesh()` で入出力されるデータのフォーマットについては、「FreeFem++ノート §6.2 mesh ファイルの構造」を見よ。

8.3.3 有限要素空間

既に定義しておいた mesh 型データと、要素の種類を表す名前 (P1, P2, ...) を用いて、有限要素空間 (この講義では $\widetilde{X}$ のように表したが、V_h などの記号で表すことが多い) を定義する。
fespace 型の変数は関数空間を表すことになる。

例えば Th と言う mesh 型の変数があるとき、

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

とすると有限要素空間 Vh が定義される。
これは型名で

```
Vh u,v;
```

として変数 u, v が定義できる。これらが個々の関数を表す。
(数学語では $u, v \in V_h$ という調子)

(**注**　これまでの授業で、三角形要素分割して、区分的 1 次多項式 (P1 要素) しか紹介しなかったが、Poisson 方程式の境界値問題以外では、他の要素 (P1b, P2, P2Morley,...) が必要になることがある。)

8.3.3 有限要素空間

- u の節点での値を集めた配列は u[] で表す。
  u[].n (u.n でも同じ) は Th.nv と同じである。
  i 番目の節点での値 (授業中の式で $u^i = \hat{u}(P_i)$) は u[](i)
- u は補間多項式でもあり、(x, y) での値は u(x,y) で得られる。

8.3.3 弱形式を定義して解く

いよいよ弱形式を定義する方法の説明である。大きく分けて 3 通りある。

ⓐ `solve` — 弱形式を与えると同時にそれを解く (弱解を求める)。

ⓑ `problem` — 弱形式を与えて問題を解く関数を定義する。

ⓒ `varf`, `matrix` — 弱形式を与えて連立 1 次方程式を作る。

8.3.3 弱形式を定義して解く

いよいよ弱形式を定義する方法の説明である。大きく分けて 3 通りある。

ⓐ `solve` — 弱形式を与えると同時にそれを解く (弱解を求める)。

ⓑ `problem` — 弱形式を与えて問題を解く関数を定義する。

ⓒ `varf`, `matrix` — 弱形式を与えて連立 1 次方程式を作る。

これまで説明して来た次の Poisson 方程式の境界値問題を元に説明する。

$$-\triangle u(x,y) = f(x,y) \quad ((x,y) \in \Omega) \tag{4a}$$

$$u(x,y) = g_1(x,y) \quad ((x,y) \in \Gamma_1) \tag{4b}$$

$$\frac{\partial u}{\partial \boldsymbol{n}}(x,y) = g_2(x,y) \quad ((x,y) \in \Gamma_2). \tag{4c}$$

弱解 u とは、X_{g_1} に属し、次の弱形式を満たすものである。

$$\langle u, v \rangle = (f, v) + [g_2, v] \quad (v \in X). \tag{5}$$

ただし

$$X_{g_1} := \{w \mid w = g_1 \quad \text{on } \Gamma_1\}, \quad X := \{v \mid v = 0 \quad \text{on } \Gamma_1\}. \tag{6}$$

8.3.3 弱形式を定義して解く (a) solve を利用

Poisson 方程式の境界値問題を解くサンプル・プログラムでは、(a) を用いた。

solve で弱形式を定義して解く

```
solve Poisson(u,v)=
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)-int1d(Th,2,3)(g2*v)
   +on(1,4,u=g1);
```

8.3.3 弱形式を定義して解く (a) solve を利用

Poisson 方程式の境界値問題を解くサンプル・プログラムでは、(a) を用いた。

solve で弱形式を定義して解く

```
solve Poisson(u,v)=
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)-int1d(Th,2,3)(g2*v)
   +on(1,4,u=g1);
```

次はどの方法でも共通である。

8.3.3 弱形式を定義して解く (a) solve を利用

Poisson 方程式の境界値問題を解くサンプル・プログラムでは、(a) を用いた。

solve で弱形式を定義して解く

```
solve Poisson(u,v)=
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)-int1d(Th,2,3)(g2*v)
   +on(1,4,u=g1);
```

次はどの方法でも共通である。

- `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。

8.3.3 弱形式を定義して解く (a) solve を利用

Poisson 方程式の境界値問題を解くサンプル・プログラムでは、(a) を用いた。

solve で弱形式を定義して解く

```
solve Poisson(u,v)=
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)-int1d(Th,2,3)(g2*v)
   +on(1,4,u=g1);
```

次はどの方法でも共通である。

- `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。
  また `int1d(Th,2,3)` は境界のうち、ラベルが 2,3 である部分 (正方形の右と上) の積分 (境界積分、今の場合は線積分) を表す。

8.3.3 弱形式を定義して解く (a) solve を利用

Poisson 方程式の境界値問題を解くサンプル・プログラムでは、(a) を用いた。

solve で弱形式を定義して解く

```
solve Poisson(u,v)=
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)-int1d(Th,2,3)(g2*v)
   +on(1,4,u=g1);
```

次はどの方法でも共通である。

- `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。
  また `int1d(Th,2,3)` は境界のうち、ラベルが 2,3 である部分 (正方形の右と上) の積分 (境界積分、今の場合は線積分) を表す。
- `+on(1,4,u=g1)` は境界のうち、ラベルが 1,4 である部分 (正方形の下と左) で、$u = g_1$ という Dirichlet 境界条件を課すことを表す (`+on(1,u=g1)+on(4,u=g1)` と分けて書くことも可能)。
  ベクトル値関数の場合は、`+on(1,u1=g1,u2=g2)` のように複数の方程式を書くこともできる。

8.3.3 弱形式を定義して解く (a) solve を利用

Poisson 方程式の境界値問題を解くサンプル・プログラムでは、(a) を用いた。

solve で弱形式を定義して解く

```
solve Poisson(u,v)=
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)-int1d(Th,2,3)(g2*v)
   +on(1,4,u=g1);
```

次はどの方法でも共通である。

- `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。
  また `int1d(Th,2,3)` は境界のうち、ラベルが 2,3 である部分 (正方形の右と上) の積分 (境界積分、今の場合は線積分) を表す。
- `+on(1,4,u=g1)` は境界のうち、ラベルが 1,4 である部分 (正方形の下と左) で、$u = g_1$ という Dirichlet 境界条件を課すことを表す (`+on(1,u=g1)+on(4,u=g1)` と分けて書くことも可能)。
  ベクトル値関数の場合は、`+on(1,u1=g1,u2=g2)` のように複数の方程式を書くこともできる。

以下、この問題の場合に、(b), (c) がどうなるか示す。

8.3.3 弱形式を定義して解く (b) problem を利用

(b) problem を利用する方法では、次のようになる。

problem で弱形式を定義して解く

```
problem Poisson(u,v)=
  int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))-int2d(Th)(f*v)-int1d(Th,2,3)(g2*v)
 +on(1,4,u=g1);

Poisson;
```

この問題の場合は、solve と比べての利点は特に感じられないかもしれないが、時間発展の問題では、同じ形の弱形式を何度も解く必要が生じるので、有効である (効率が上がる可能性がある — 後述)。

8.3.3 弱形式を定義して解く (c) varf, matrix を利用

varf, matrix を利用

```
real Tgv=1.0e+30; // tgv と小文字でも可
varf a(u,v)=
  int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
 +on(1,4,u=g1);
matrix A=a(Vh,Vh,tgv=Tgv,solver=CG);
varf l(unused,v)=
  int2d(Th)(f*v)+int1d(Th,2,3)(g2*v)
 +on(1,4,unused=0);
Vh F;
F[]=l(0,Vh,tgv=Tgv);
u[]=A^-1*F[];
```

あらすじは、連立 1 次方程式 $A\boldsymbol{u} = \boldsymbol{f}$ の A, $\boldsymbol{f}$ を別々に計算して、$A^{-1}\boldsymbol{f}$ を計算することで $\boldsymbol{u}$ を得る、ということである (詳細は、実は現時点で把握していないので省略する)。

8.3.3 弱形式を定義して解く (c) varf, matrix を利用

varf, matrix を利用

```
real Tgv=1.0e+30; // tgv と小文字でも可
varf a(u,v)=
  int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
 +on(1,4,u=g1);
matrix A=a(Vh,Vh,tgv=Tgv,solver=CG);
varf l(unused,v)=
  int2d(Th)(f*v)+int1d(Th,2,3)(g2*v)
 +on(1,4,unused=0);
Vh F;
F[]=l(0,Vh,tgv=Tgv);
u[]=A^-1*F[];
```

あらすじは、連立 1 次方程式 $A\boldsymbol{u} = \boldsymbol{f}$ の A, $\boldsymbol{f}$ を別々に計算して、$A^{-1}\boldsymbol{f}$ を計算することで $\boldsymbol{u}$ を得る、ということである (詳細は、実は現時点で把握していないので省略する)。

tgv (terrible great value) は以前説明した。

8.3.3 弱形式を定義して解く (c) varf, matrix を利用

varf, matrix を利用

```
real Tgv=1.0e+30; // tgv と小文字でも可
varf a(u,v)=
  int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
 +on(1,4,u=g1);
matrix A=a(Vh,Vh,tgv=Tgv,solver=CG);
varf l(unused,v)=
  int2d(Th)(f*v)+int1d(Th,2,3)(g2*v)
 +on(1,4,unused=0);
Vh F;
F[]=l(0,Vh,tgv=Tgv);
u[]=A^-1*F[];
```

あらすじは、連立 1 次方程式 $A\boldsymbol{u} = \boldsymbol{f}$ の A, $\boldsymbol{f}$ を別々に計算して、$A^{-1}\boldsymbol{f}$ を計算することで $\boldsymbol{u}$ を得る、ということである (詳細は、実は現時点で把握していないので省略する)。

tgv (terrible great value) は以前説明した。

solver= は連立 1 次方程式の解法を指定する。

CG	**CG 法** (共役勾配法) (反復法, 正値対称行列 (*spd*) 用)
GMRES	**GMRES 法** (反復法, 一般の正則行列用)
UMFPACK	**UMFPACK** を利用 (直接法, 一般の正則行列用)
sparsesolver	ダイナミック・リンクで外部のソルバーを呼ぶ

参考プログラム — 有限要素解の誤差を見る

有限要素解の収束、誤差の減衰は本科目の最後に説明する予定であるが、見ておこう。
真の解 u, 有限要素解 $\hat{u}$ について

$$\|u - \hat{u}\|_{L^2} = \left(\int\int_{\Omega} |u(x,y) - \hat{u}(x,y)|^2 \, dx \, dy \right)^{1/2}$$

を L^2 誤差、

$$\|u - \hat{u}\|_{H^1} = \left(\|u - \hat{u}\|_{L^2}^2 + \|u_x - \hat{u}_x\|_{L^2}^2 + \|u_y - \hat{u}_y\|_{L^2}^2 \right)^{1/2}$$

を H^1 誤差と呼ぶ。
領域の分割を細かくしたとき、これらの誤差がどのように減衰するか、厳密解が分かる問題で調べてみよう。

```
curl -O http://nalab.mind.meiji.ac.jp/~mk/program/fem/poisson-mixedBC-mk.edp
FreeFem++ poisson-mixedBC-mk.edp
FreeFem++ poisson-mixedBC-mk.edp 2
FreeFem++ poisson-mixedBC-mk.edp 4
...
FreeFem++ poisson-mixedBC-mk.edp 64
```

正方形の辺を $20 \cdot 2^m$ $(m = 0, 1, \cdots, 6)$ 分割したときの誤差を近似計算する。

参考プログラム — 有限要素解の誤差を見る

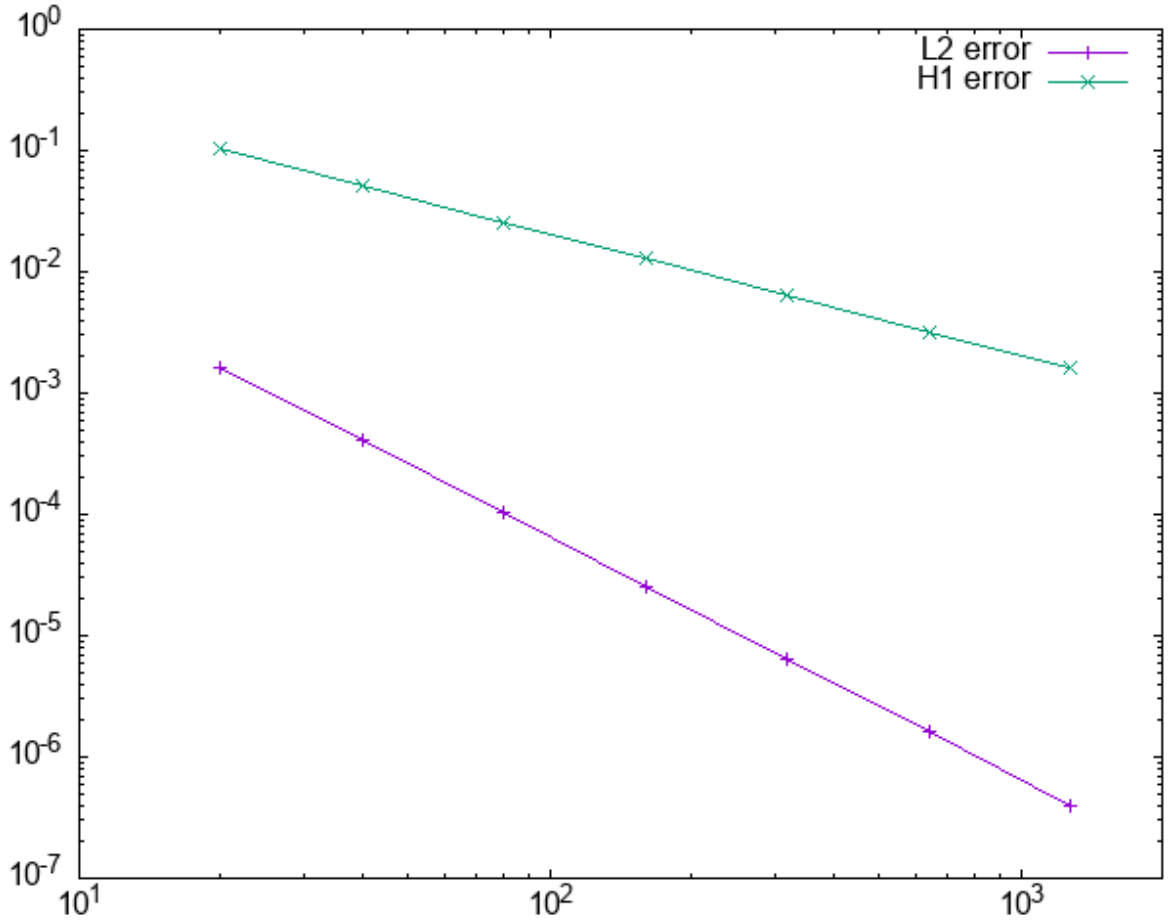


図 5: 1 辺を $n = 20, 40, 80, \cdots, 1280$ 分割したときの誤差 (横軸 n)

L^2 誤差は $O(n^{-2})$, H^1 誤差は $O(n^{-1})$ となっている (実は理論からの予想と一致)。

参考文献

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

[2] Hecht, F.: New development in FreeFem++, *J. Numer. Math.*, Vol. 20, No. 3-4, pp. 251–265 (2012).

[3] Hecht, F.: Freefem++, `https://doc.freefem.org/pdf/FreeFEM-documentation.pdf`, 以前は `http://www3.freefem.org/ff++/ftp/freefem++doc.pdf` にあった。

[4] 桂田祐史：FreeFEM++ ノート, `http://nalab.mind.meiji.ac.jp/~mk/labo/text/freefem-note.pdf` (2012〜).

授業後の追加: 図 5 はどうやって描くか

データは数が少ないので、一つ一つの分割数について、poisson-mixedBC.edp を使って誤差を得た。それを次のように記録したファイルを作った。(工夫すれば自動化できるであろう。)

error.txt

```
20   0.00162987   0.102169
40   0.000408387   0.0511309
80   0.000102155   0.0255713
160   2.55422e-05   0.0127864
320   6.38579e-06   0.00639328
640   1.59646e-06   0.00319665
1280   3.99119e-07   0.00159833
```

error.gp … gnuplot でグラフを描き、イメージ・データも作る

```
set logscale
set format y "10^{%L}"
set format x "10^{%L}"
plot [10:2000] "error.txt" using 1:2 with lp title "L2 error",\
  "error.txt" using 1:3 with lp title "H1 error"
set term png
set output "error.png"
replot
```

1, 4-5 行目が必須。2,3 行目は凡例の体裁を整える。最後の 3 行はイメージ・データ作成のため。

ターミナルで gnuplot error.gp とすれば、グラフが描かれ、errror.png が出来る。

授業後の追加: 図 5 はどうやって描くか

ちなみに (このやり方を勧めているわけではないが)、次のようなシェル・スクリプトを実行して error.txt を作成した。

make-data.sh

```
#!/bin/sh
rm -f error2.txt
for i in 1 2 4 8 16 32 64
do
  echo ${i}
  FreeFem++ poisson-mixedBC.edp ${i} | grep '^n='|grep H1 >> error2.txt
done
sed 's/n=//;s/L2-error=//;s/H1-error=//' error2.txt>error.txt
```

```
chmod +x make-data.sh
./make-data.sh
```

このように自動化すると、どのようにデータを作成したかの記録が残ることになる。結果が再現できることは非常に重要であり、データを取るときは工夫するべきである。

やり残し

鈴木厚先生の資料を参考文献に載せる。
cin, cout など入出力の話。
関数定義の話。
freefem-note へのフィードバック
gnuplot の紹介

(時間がない…)