応用複素関数 第 11 回資料

～FreeFEM の文法, 有限要素解の誤差～

かつらだ まさし
桂田 祐史

https://m-katsurada.sakura.ne.jp/complex2-2026/

2026 年 6 月 30 日

目次

本日の内容など

- FreeFEM の文法を説明する。
  文法について、桂田 [2] というメモを書いてあるけれど、講義等で説明する場合は、この文書のようなスライドを使うことにしている。
- 有限要素解の誤差についての話はおまけ。

7 FreeFEM の文法

7.1 はじめに

すでに FreeFEM のインストール手順と簡単な解説を行ってある。

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 [1])。

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

7 FreeFEM の文法

7.1 はじめに

すでに FreeFEM のインストール手順と簡単な解説を行ってある。

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 [1])。

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)。

7 FreeFEM の文法

7.1 はじめに

すでに FreeFEM のインストール手順と簡単な解説を行ってある。

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 [1])。

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。

7 FreeFEM の文法

7.1 はじめに

すでに FreeFEM のインストール手順と簡単な解説を行ってある。

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 [1])。

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 ([2] の付録に書いておいた)。

参考: FreeFEM は C++ で記述されている。

7 FreeFEM の文法

7.1 はじめに

すでに FreeFEM のインストール手順と簡単な解説を行ってある。

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 [1])。

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 ([2] の付録に書いておいた)。

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、言語仕様が整理され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 [2] に基づく。

マニュアル以外の情報源として、日本応用数理学会のチュートリアル (鈴木 [4], [5] — リンク切れ直しました)、テキスト大塚・高石 [6](本学学生は Maruzen eBook で読める)

7.1 はじめに　基本的な Poisson 方程式のプログラム

curl -O https://m-katsurada.sakura.ne.jp/program/fem/poisson.edp で入手できる。

```
// 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。

7.2 汎用のプログラミング機能

7.2.1 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。

7.2 汎用のプログラミング機能

7.2.1 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)

7.2 汎用のプログラミング機能

7.2.1 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 ;

7.2 汎用のプログラミング機能

7.2.1 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

7.2 汎用のプログラミング機能

7.2.1 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).

7.2 汎用のプログラミング機能

7.2.1 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。

7.2 汎用のプログラミング機能

7.2.1 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。

7.2 汎用のプログラミング機能

7.2.1 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。

7.2 汎用のプログラミング機能

7.2.1 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。

7.2 汎用のプログラミング機能

7.2.1 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。

7.2 汎用のプログラミング機能

7.2.1 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

7.2 汎用のプログラミング機能

7.2.1 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…

7.2.2 データ型

7.2.2 データ型

- 整数を表すための int がある (C 言語の `int` に相当)

7.2.2 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)

7.2.2 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`)
  例えば `complex a=1+2i;`　入出力は 2 次元ベクトル風の `(1,2)`

7.2.2 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`)
  例えば `complex a=1+2i;`　入出力は 2 次元ベクトル風の `(1,2)`
- 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。

7.2.2 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`)
  例えば `complex a=1+2i;`　入出力は 2 次元ベクトル風の `(1,2)`
- 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？)。

7.2.2 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`)
  例えば `complex a=1+2i;`　入出力は 2 次元ベクトル風の `(1,2)`
- 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。

7.2.2 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`)
  例えば `complex a=1+2i;`　入出力は 2 次元ベクトル風の `(1,2)`
- 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;
```

7.2.2 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`)
  例えば `complex a=1+2i;`　入出力は 2 次元ベクトル風の `(1,2)`
- 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)。

7.2.3 配列型　1 次元

1 次元配列は、C 言語に (少し) 似ている。

配列 a の第 i 要素は、a[i] としてアクセスできるが、a(i) としてアクセスするのが普通？

```
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。

7.2.3 配列型　2 次元

2 次元配列は少し違う。2 次元配列 a の (i,j) 要素にアクセスするには a(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;
```

7.2.4 FreeFEM の real データの入出力の書式指定

- 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。

7.2.4 FreeFEM の real データの入出力の書式指定　例

```
// 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;
```

7.3 有限要素法のための機能

7.3.1 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、基本的と考えられる 2 次元 Poisson 方程式の境界値問題のプログラムを例にして説明する。

1. 領域の定義と領域の三角形分割
2. 有限要素空間の定義
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。

7.3.2 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。

7.3.2 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。

7.3.2 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。

7.3.2 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。

7.3.2 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。

7.3.2 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()`, `savemesh()` と言う関数を用いて入出力できる (フォーマットはテキスト・ファイル)。

7.3.2 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()`, `savemesh()` と言う関数を用いて入出力できる (フォーマットはテキスト・ファイル)。
- `mesh` 型のデータは、`plot()` により可視化できる。

7.3.2 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()`, `savemesh()` と言う関数を用いて入出力できる (フォーマットはテキスト・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」)。

7.3.2 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;   label=1; }
border C2(t=0,1) { x=1;   y=t;   label=2; }
border C3(t=0,1) { x=1-t; y=1;   label=3; }
border C4(t=0,1) { x=0;   y=1-t; label=4; }
```

(label の値指定は必須ではない。指定する場合は 0 以外の値を選ぶ。)

7.3.2 領域の定義と領域の三角形分割　メッシュ (mesh)

それぞれ表示してみる (次のスライド)

```
// 例 1
border C(t=0,2*pi) { x=cos(t); y=sin(t); }
mesh Th1=buildmesh(C(50));
plot(Th1,wait=true,ps="Th1.eps");
// 例 2
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; }
mesh Th2=buildmesh(Gamma1(25)+Gamma2(50));
plot(Th2,wait=true,ps="Th2.eps");
// 例 3
border C1(t=0,1) { x=t;   y=0;   label=1; }
border C2(t=0,1) { x=1;   y=t;   label=2; }
border C3(t=0,1) { x=1-t; y=1;   label=3; }
border C4(t=0,1) { x=0;   y=1-t; label=4; }
mesh Th3=buildmesh(C1(20)+C2(20)+C3(20)+C4(20));
plot(Th3,wait=true,ps="Th3.eps");
```

7.3.2 領域の定義と領域の三角形分割　メッシュ (mesh)

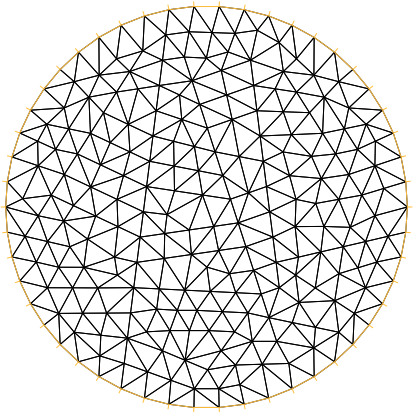

図 1: C(50)

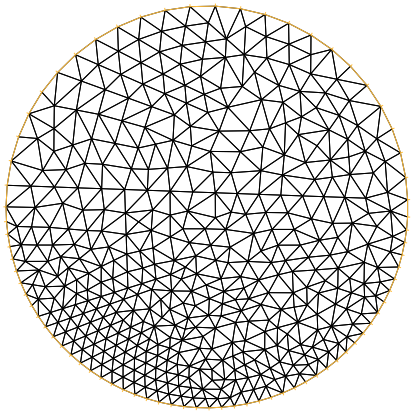

図 2: Gamma1(25)+Gamma2(50)

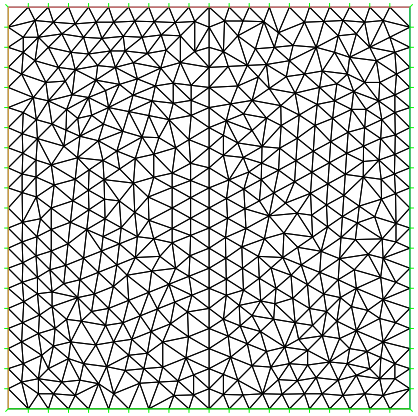

図 3: C1(20)+C2(20)+...

7.3.2 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));
plot(ThWithoutHole,wait=1,ps="Thwithouthole.eps");
plot(a(50)+b(-30),wait=true,ps="borderminus.eps");
mesh ThWithHole = buildmesh(a(50)+b(-30));
plot(ThWithHole,wait=1,ps="Thwithhole.eps");
```

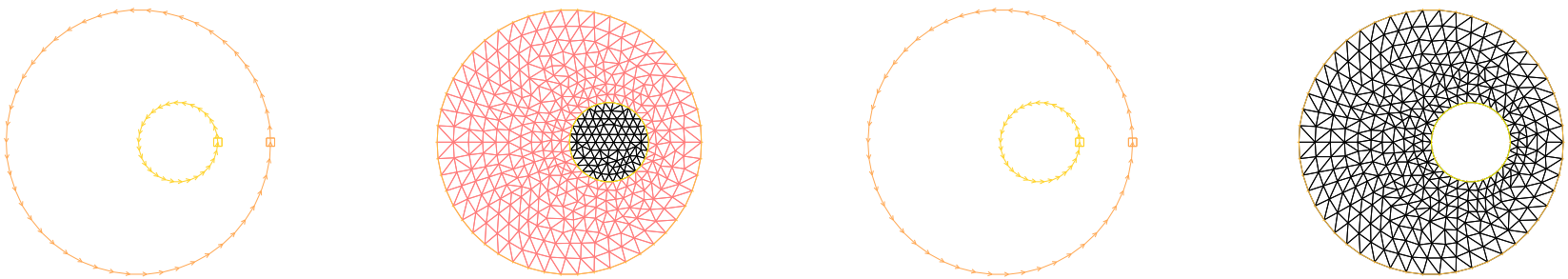

図 4: 左から 1 つ目 3 つ目は分割した border を表示。

7.3.2 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…

7.3.2 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。

7.3.2 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 ($i = 0, 1, \cdots,$ `Th.nv` $- 1$) で、その座標は `Th(`i`).x` と `Th(`i`).y` である。`Th(`i`).label` はその節点のラベル (領域内部にあれば 0, それ以外は境界のどこか) を表す。

7.3.2 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 ($i = 0, 1, \cdots,$ `Th.nv` $- 1$) で、その座標は `Th(`i`).x` と `Th(`i`).y` である。`Th(`i`).label` はその節点のラベル (領域内部にあれば 0, それ以外は境界のどこか) を表す。
- `Th[`i`]` は i 番目の三角形 ($i = 0, 1, \cdots,$ `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。

7.3.2 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 ($i = 0, 1, \cdots,$ `Th.nv` $- 1$) で、その座標は `Th(`i`).x` と `Th(`i`).y` である。`Th(`i`).label` はその節点のラベル (領域内部にあれば 0, それ以外は境界のどこか) を表す。
- `Th[`i`]` は i 番目の三角形 ($i = 0, 1, \cdots,$ `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。

7.3.2 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()`, `savemesh()` は有効である。

7.3.2 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()`, `savemesh()` 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 (有限要素解の計算は自作プログラムで行う等)。

7.3.2 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()`, `savemesh()` 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 (有限要素解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。

7.3.2 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()`, `savemesh()` 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 (有限要素解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()`, `savemesh()` で入出力される**メッシュ・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。

- `Mesh` 型の変数 `Th` の内容は、

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

のようにしてファイルに出力できる。
- そのフォーマットにのっとって作られたファイルがあれば、

```
Mesh Th=readmesh("bunkatsu.msh");
```

のようにして読み込める (自分で三角形分割することも可能)。

7.3.3 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、巷のテキストでは 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。

これは文法的には型名で

```
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。)

7.3.3 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。

7.3.3 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。

(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。

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

7.3.3 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。

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

$$u(x,y) = g_1(x,y) \quad ((x,y) \in \Gamma_1) \tag{1b}$$

$$\frac{\partial u}{\partial \boldsymbol{n}}(x,y) = g_2(x,y) \quad ((x,y) \in \Gamma_2). \tag{1c}$$

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。

$$\langle u, v \rangle = (f, v) + [g_2, v] \quad (v \in X). \tag{2}$$

ただし

$$X_{g_1} := \{w \mid w = g_1 \quad \text{on } \Gamma_1\}, \quad X := \{v \mid v = 0 \quad \text{on } \Gamma_1\}. \tag{3}$$

7.3.3 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 poisson.edp 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);
```

7.3.3 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 poisson.edp 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。

7.3.3 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 poisson.edp 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。

7.3.3 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 poisson.edp 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。

7.3.3 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 poisson.edp 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。

7.3.3 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 poisson.edp 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。

7.3.3 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)。

7.3.3 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)。

7.3.3 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、§5.5「連立 1 次方程式の具体例 Dirichlet 境界条件の考慮 ベクトル、行列の縮小を避ける方法 (2) (FreeFEM で採用)」で説明した (Tgv $= 10^{30}$)。

7.3.3 弱形式を定義して解く (d) 連立 1 次方程式のソルバー

FreeFEM ドキュメンテーションの §3.3.13 (Hecht [3]) から。

- `solver=` には LU, CG, Crout, Cholesky, GMRES, sparsesolver, UMFPACK が指定できる (最初の 5 つはアルゴリズムの名前, 説明省略)。
  デフォールトでは sparsesolver で、それは他の sparsesolver が定義されていなければ UMFPACK に等しい。それは他の直接法のソルバーが使えない場合は LU にセットされる。
  行列のメモリーへの格納の仕方は、solver により決まる。LU の場合は非対称なスカイライン (説明省略)、Crout の場合は対称なスカイライン、Cholesky の場合は正値対称なスカイライン、CG の場合は正値対称 (spd) な疎行列、その他 (GMRES, sparsesolver, UMFPACK) では疎行列。
  sparsesolver は、ダイナミック・リンクで外部のソルバーを呼ぶ

7.3.3 弱形式を定義して解く (d) 連立 1 次方程式のソルバー

FreeFEM ドキュメンテーションの §3.3.13 (Hecht [3]) から。

- `solver=` には LU, CG, Crout, Cholesky, GMRES, sparsesolver, UMFPACK が指定できる (最初の 5 つはアルゴリズムの名前, 説明省略)。
  デフォールトでは sparsesolver で、それは他の sparsesolver が定義されていなければ UMFPACK に等しい。それは他の直接法のソルバーが使えない場合は LU にセットされる。
  行列のメモリーへの格納の仕方は、solver により決まる。LU の場合は非対称なスカイライン (説明省略)、Crout の場合は対称なスカイライン、Cholesky の場合は正値対称なスカイライン、CG の場合は正値対称 (spd) な疎行列、その他 (GMRES, sparsesolver, UMFPACK) では疎行列。
  sparsesolver は、ダイナミック・リンクで外部のソルバーを呼ぶ
- `init=`論理型の式
  `false` または 0 のとき、行列が再構成 (reconstruct) される、とある。初期化されているかどうか、という意味か？

7.3.3 弱形式を定義して解く (d) 連立 1 次方程式のソルバー

FreeFEM ドキュメンテーションの §3.3.13 (Hecht [3]) から。

- `solver=` には LU, CG, Crout, Cholesky, GMRES, sparsesolver, UMFPACK が指定できる (最初の 5 つはアルゴリズムの名前, 説明省略)。
  デフォールトでは sparsesolver で、それは他の sparsesolver が定義されていなければ UMFPACK に等しい。それは他の直接法のソルバーが使えない場合は LU にセットされる。
  行列のメモリーへの格納の仕方は、solver により決まる。LU の場合は非対称なスカイライン (説明省略)、Crout の場合は対称なスカイライン、Cholesky の場合は正値対称なスカイライン、CG の場合は正値対称 (spd) な疎行列、その他 (GMRES, sparsesolver, UMFPACK) では疎行列。
  sparsesolver は、ダイナミック・リンクで外部のソルバーを呼ぶ
- `init=`論理型の式
  `false` または 0 のとき、行列が再構成 (reconstruct) される、とある。初期化されているかどうか、という意味か？
- `eps=`実数型の式
  反復法の停止則を指定する。
  $\varepsilon < 0$ の場合は $\|Ax - b\| < |\varepsilon|$, $\varepsilon > 0$ の場合は $\|Ax - b\| < \frac{|\varepsilon|}{\|Ax_0 - b\|}$
  (と書いてあるけれど、$\frac{\|Ax-b\|}{\|Ax_0-b\|} < |\varepsilon|$ の間違いではないかな？)

(連立 1 次方程式のアルゴリズムを学んだことがないと、少し分かりにくいかも…)

(補足) FreeFEM の弱形式の書き方についての注意

弱形式の書き方についての文法が詳しく説明されていない (私が探せないだけ？？) ので、良く分からないが、`int2d(Th)()` の括弧内に “複雑な式” を書くと、文法エラーが発生する。エラー・メッセージがとても分かりにくい。

以下に書くことが正しいかは自信がないが、θ 法のプログラムを書くときなど、ひっかっかった場合に参考にしてもらえると良いかも……(頼りない)

- `int2d(Th)(u*v-uold*v)` はエラーになる。
  `int2d(Th)(u*v)-int2d(Th)(uold*v)` とすると通る。一方で、
  `int2d(Th)(u*v+theta*tau*(dx(u)*dx(v)+dy(u)*dy(v)))` は通るので、
  `*v` が複数回現れるのがまずいと受け取っている (`*dx(v)` や `*dy(v)` は別カウントらしい)。
- `int2d(Th)(u*v)-int2d(Th)(uold*v)` は通るが、
  `int2d(Th)((u-uold)*v)` はエラーになる。これはどう考えるべきか。
- `int2d(Th)(tau*f*v)` を `tau*int2d(Th)(f*v)` とするとエラーになる。
  定数であっても `int2d(Th)()` の外には出さず、`+int2d()()` あるいは `-int2d()()` だけで済ませる。

私は極力分けるようにしている。±`int2d(Th)()` の個数が増えて叱られたことはない。

参考プログラム — 有限要素解の誤差を見る (1)

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

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

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 (分割が粗いとき、実行は一瞬で終わることに注意)。

```
curl -O https://m-katsurada.sakura.ne.jp/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 (2)

このプログラム (日本応用数理学会の 2016 年の鈴木厚先生のチュートリアルで紹介されたもの) で解いている問題は、先に厳密解 (真の解) u を選んで、それから f, g, h を計算して作ったのであろう (よくあるトリック)。

$$\Omega = (0,1) \times (0,1),$$
$$\Gamma_1 = \{(1,y) \mid 0 \le y \le 1\} \cup \{(x,1) \mid 0 \le x \le 1\} \cup \{(0,y) \mid 0 \le y \le 1\},$$
$$\Gamma_2 = \{(x,0) \mid 0 \le x \le 1\}.$$

$$u(x,y) = \sin(\pi x) \sin \frac{\pi y}{2}$$

より

$$f(x,y) = -\triangle u(x,y) = \frac{5\pi^2}{4} \sin(\pi x) \sin \frac{\pi y}{2},$$
$$g(x,y) = u(x,y) = \sin(\pi x) \sin \frac{\pi y}{2},$$
$$h(x,y) = \frac{\partial u}{\partial n}(x,0) = -\frac{\partial u}{\partial y}(x,0) = -\left. \frac{\pi}{2} \sin(\pi x) \cos \frac{\pi y}{2} \right|_{y=0} = -\frac{\pi}{2} \sin(\pi x).$$

参考プログラム — 有限要素解の誤差を見る (3)

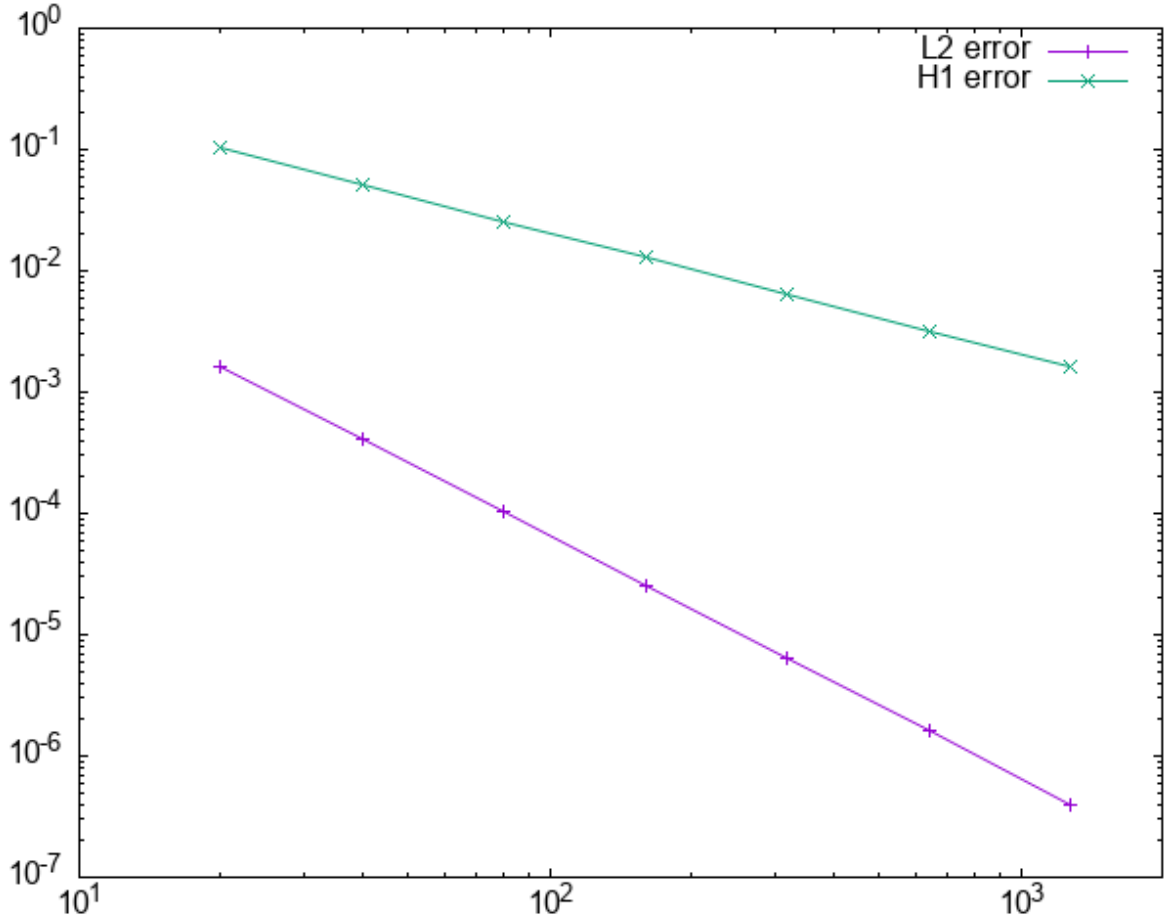

図 5: 1 辺を $n = 20, 40, 80, \cdots, 1280$ 分割したときの誤差 (横軸 n)

($h := 1/n$ として) L^2 誤差は $O(h^2)$, H^1 誤差は $O(h)$ となっている (これは理論からの予想と一致している)。

参考プログラム — 図5 の作成方法

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   3.99118e-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。

参考プログラム — 図5 の作成方法 (続き)

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 (データ作成の自動化)。

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-mk.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
```

データ作成を自動化すると、作成手順の記録が残り、結果の再現も容易になる。記録を残し、再現性を確保することは重要であり、工夫すべきである。

参考プログラム — 図5 の作成方法 (続き)

FreeFem++ があれば試せるはず

```
curl -O https://m-katsurada.sakura.ne.jp/ana2026/experiment20260602.tar.gz
tar xzf experiment20260602.tar.gz
cd experiment20260602
make
```

最後の 1 辺を $20 \times 64 = 1280$ (三角形が 320 万個) 分割する場合の計算は少し時間がかかるので、焦らず待って下さい。

やり残し

- 関数定義の話 (とりあえず「FreeFem ノート 15 関数定義」)。
- gnuplot の紹介は不要と考えているのだけど (多分知っている)、大丈夫？

A. (おまけ) C++のストリーム入出力

すでに述べたように、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。

```
#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)

A.2 数値の書式指定

```
// 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…)

参考文献 I

[1] Hecht, F.: New development in FreeFem++, *J. Numer. Math.*, Vol. 20, No. 3-4, pp. 251–265 (2012), https://hal.sorbonne-universite.fr/hal-01476313v1/document.

[2] 桂田祐史：FreeFEM++ ノート, https://m-katsurada.sakura.ne.jp/labo/text/freefem-note.pdf (2012〜).

[3] Hecht, F.: Freefem++ (マニュアル), https://github.com/FreeFem/FreeFem-doc/raw/pdf/FreeFEM-documentation.pdf, 以前は http://www3.freefem.org/ff++/ftp/freefem++doc.pdf にあった。(??).

参考文献 II

[4] 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.fr/~suzukia/FreeFempp-tutorial-JSIAM2016/ から入手できる (2016).

[5] 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 — 上級編 —」(2016/6/4–5) の配布資料で、https://www.ljll.fr/~suzukia/FreeFempp-tutorial-JSIAM2016b/ から入手できる (2016).

参考文献 III

[6] 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/musashino-u.ac.jp/freefem/home/book/OT2014 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.