数値解析についての資料

一橋大学経営管理研究科 小林健太

2025/6/8 12:30 更新

0 この資料の概要

現代においてコンピュータシミュレーションは，自然科学や工学，社会科学から医療に至るまで様々な分野で用いられ，我々の生活を基礎から支えている．講義では，コンピュータシミュレーションを行う際，現象を記述する方程式をどのようにして計算機上で計算すればよいのか，また，元の問題の解と計算機で計算した解にはどのような関係があるのか等，数値計算にまつわる諸問題について説明する．

この資料は

http://kobayashi.hub.hit-u.ac.jp/lecture/

からダウンロードできる．随時，更新する予定．

0.1 プログラミング

数値解析においては，理論の理解も大切だが，実際にプログラムを組んで理論を確かめることも重要である．この資料では，プログラムはC言語（C++）を用いる．C++はC言語の拡張となっており，C言語のプログラムはC++としても解釈できるが，C++はC言語としては実行できない．この資料では，C言語風にプログラムを書いているが，一部，C++の機能も用いている．

円マーク‘¥’とバックスラッシュ‘\’は，環境により表示が異なるが，プログラム上では同じ意味を持つ．この資料のプログラムで‘\’となっている部分は，日本語環境の多くでは‘¥’と入力するとよい．通常，キーボードの一番右上にある．

パソコン上でC++を使うにはコンパイラをインストールする必要があるが，フリーのものでも各種の選択肢がある．

初心者にとっては Embarcadero 社の Free C++ Compiler がインストールも簡単で良いだろう．

https://www.embarcadero.com/jp/free-tools/ccompiler/free-download

こちらからダウンロードできる．ダウンロードには登録が必要だが，登録しても宣伝メールが送られてくるとかの不都合は特に生じない．インストールの方法およびコンパイラの使い方について

http://kobayashi.hub.hit-u.ac.jp/topics/compiler.html

に簡単に解説しておいたので，参考にしてもらいたい（ただし，情報は古い可能性がある）．

Visual Studio は開発環境などが色々と揃っていて，Windows アプリケーションなど、そこそこ本格的なプログラミングも可能である．フリーだが登録が必要．

Cygwin は Windows 上の仮想 Linux 環境だが，インストールすると C++コンパイラが付いてくる．サーバー管理等，Linux に興味のある人はこれも良いだろう．

Cygwin からコンパイラのみを抜き出したものとして MinGW がある．非常に軽くてお勧めだが，若干，インストールに苦労することがあるかもしれない．

他にも，Open Watcom C/C++，Digital Mars C/C++ Compilers，なんてのもある．

Mac については，私は詳しくは知らないが，色々とやり方はあるようだ．Windows でなく，最初から Linux 機でプログラムするのも手かもしれない．

0.2　線形演算ライブラリ

C 言語には，行列やベクトルを扱う機能が基本機能としては付属していないので，それを実現するライブラリを用意した．

http://kobayashi.hub.hit-u.ac.jp/topics/linear.html

ここから Linear.h というファイルをダウンロードしてプログラムと同じフォルダに入れておくこと．その上で，プログラムの最初に

```
#include "Linear.h"
```

と書いておくと行列やベクトルを扱うことができる．
このライブラリを用いる上で注意すべき点は，ある変数に格納されている行列やベクトルを他の変数に代入した場合，実体としては同じものになる，ということである．すなわち，一方を変更すると他方も変化してしまうことになる（Python の list 等と同じ動作である）．別の実体を持つコピーが必要な場合は，Copy メソッドを用いる．具体的には実際に必要が生じたときに説明する．

0.3　グラフ表示

数値計算結果をグラフ化して表示するために，gnuplot をインストールする必要がある．

`http://www.gnuplot.info/`

こちらから Download from SourceForge のリンクを辿るとダウンロードサイトに行けるが，バナー広告で「Download」とか書かれているのが出たりするので間違ってクリックしないように．

32bit ウィンドウズ機なら「gp***-win32-setup.exe」を，64 ビットウィンドウズ機なら「gp***-win64-setup.exe」を選ぶと良いだろう（ただし，***はバージョン番号で数字が入る）．

0.4　この資料のプログラミング例について

参考のためにこの資料にもプログラムの例を載せてあるが，あくまで参考という事で，すべての数値解析手法についてプログラムが載っているわけではない．プログラムが載っていないものについては，各自で頑張ってプログラミングしてもらいたい．

0.5　参考文献

C 言語の参考書としては

柴田望洋「新版 明解 C 言語 入門編」（ソフトバンククリエイティブ）
皆本晃弥「やさしく学べる C 言語入門 – 基礎から数値計算入門まで」（サイエンス社）

を勧める．

数値解析について，参考にしたい，もしくはもっと詳しく勉強したいという人には

森正武「数値解析」（共立出版）
齊藤宣一「数値解析入門」（東京大学出版会）
皆本晃弥「C 言語による数値計算入門」（サイエンス社）

などの書籍を勧める．理解もさらに深まることだろう．

0.6　質問・連絡先など

この資料について，何か質問や連絡したいことがある場合は，私のメールアドレス

`kenta.k@r.hit-u.ac.jp`

まで連絡すること．

1　連立一次方程式

何か現象を表すモデル方程式があったとして，そのモデル方程式を離散化して計算機で解こうとすると，連立一次方程式が現れることが多い．そこで，まずは連立一次方程式をいかにして解くか，について説明しよう．以下では，i 行 j 列の要素が a_{ij} であるような行列を (a_{ij}) と書く．

1.1　ガウスの消去法

まず連立一次方程式の解法として思いつくのは，学部一年生の線形代数で習った掃き出し法である．これは，数値解析ではガウスの消去法として知られている．

最初に具体例を考えてみよう．連立一次方程式

$$\begin{pmatrix} -1 & -2 & 4 \\ 2 & 7 & -2 \\ -3 & -8 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ -5 \\ 14 \end{pmatrix}$$

を解きたいとする．まず，1 行目の定数倍を他の行から引いて，2 行目以降の 1 列目を零にする．

$$\begin{pmatrix} -1 & -2 & 4 \\ 0 & 3 & 6 \\ 0 & -2 & -6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 2 \end{pmatrix}$$

右辺のベクトルも変化していることに注意．線形代数で，右辺ベクトルを行列にくっつけて拡大係数行列を作ったことを思い出そう．次に，2 行目の定数倍を他の行から引いて，3 行目以降の 2 列目を零にする．

$$\begin{pmatrix} -1 & -2 & 4 \\ 0 & 3 & 6 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix}$$

このようにして，まずは上三角行列が得られた．この操作を**前進消去**という．

前進消去が終わったら，x_3 から x_1 まで順に解を求めていく．

$$\begin{aligned} x_3 &= \frac{1}{-2} \cdot 4 = -2 \\ x_2 &= \frac{1}{3} \left(3 - 6x_3\right) = 5 \\ x_1 &= \frac{1}{-1} \left(4 - (-2)x_2 - 4x_3\right) = -22 \end{aligned}$$

これで解が $x_1 = -22,\ x_2 = 5,\ x_3 = -2$ と求まる．このように，上三角行列から代入によって解を求める操作を**後退代入**という．

これを一般的に書くと以下のようになる．解きたい連立一次方程式を

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1\,n-1}^{(1)} & a_{1n}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & a_{2\,n-1}^{(1)} & a_{2n}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & \cdots & a_{3\,n-1}^{(1)} & a_{3n}^{(1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1\,1}^{(1)} & a_{n-1\,2}^{(1)} & a_{n-1\,3}^{(1)} & \cdots & a_{n-1\,n-1}^{(1)} & a_{n-1\,n}^{(1)} \\ a_{n1}^{(1)} & a_{n2}^{(1)} & a_{n3}^{(1)} & \cdots & a_{n\,n-1}^{(1)} & a_{nn}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ \vdots \\ b_{n-1}^{(1)} \\ b_n^{(1)} \end{pmatrix}$$

とし，$k = 1, 2, 3, \cdots, n-1$ について

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} a_{kj}^{(k)} \qquad (i = k+1, \cdots, n, \quad j = k+1, \cdots, n)$$

$$b_i^{(k+1)} = b_i^{(k)} - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} b_k^{(k)} \qquad (i = k+1, \cdots, n)$$

と計算すると，連立一次方程式は

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1\,n-1}^{(1)} & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2\,n-1}^{(2)} & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3\,n-1}^{(3)} & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-1\,n-1}^{(n-1)} & a_{n-1\,n}^{(n-1)} \\ 0 & 0 & 0 & \cdots & 0 & a_{nn}^{(n)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ b_3^{(3)} \\ \vdots \\ b_{n-1}^{(n-1)} \\ b_n^{(n)} \end{pmatrix}$$

と変形され，前進消去が実行される．この連立一次方程式を改めて

$$\begin{pmatrix} \tilde{a}_{11} & \tilde{a}_{12} & \tilde{a}_{13} & \cdots & \tilde{a}_{1\,n-1} & \tilde{a}_{1n} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \cdots & \tilde{a}_{2\,n-1} & \tilde{a}_{2n} \\ 0 & 0 & \tilde{a}_{33} & \cdots & \tilde{a}_{3\,n-1} & \tilde{a}_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \tilde{a}_{n-1\,n-1} & \tilde{a}_{n-1\,n} \\ 0 & 0 & 0 & \cdots & 0 & \tilde{a}_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \vdots \\ \tilde{b}_{n-1} \\ \tilde{b}_n \end{pmatrix}$$

と書けば,

$$x_n = \frac{1}{\tilde{a}_{nn}} \tilde{b}_n$$

$$
\begin{aligned}
x_{n-1} &= \frac{1}{\tilde{a}_{n-1\,n-1}} \left(\tilde{b}_{n-1} - \tilde{a}_{n-1\,n}\ x_n \right) \\
&\cdots \\
x_2 &= \frac{1}{\tilde{a}_{22}} \left(\tilde{b}_2 - \sum_{j=3}^{n} \tilde{a}_{2j} x_j \right) \\
x_1 &= \frac{1}{\tilde{a}_{11}} \left(\tilde{b}_1 - \sum_{j=2}^{n} \tilde{a}_{1j} x_j \right)
\end{aligned}
$$

によって後退代入が実行され，解が求まる．

以上の説明では，行列やベクトルの要素が変化するときに，変化前と変化後を右上の添え字で区別したが，実際のプログラムでは単に代入して値を更新していけばよい．

1.2 ピボット選択

ガウスの消去法においては，途中で $a_{kk}^{(k)}$ が零になると計算が破綻してしまう．また，$a_{kk}^{(k)}$ が零でなかったとしても，非常に小さな値のときには，計算精度が悪くなることがある．まずは実例でその様子を見てみよう．

以下の連立一次方程式を考える．解は $x_1 = x_2 = x_3 = 1$ である．

$$
\begin{pmatrix} 0.001 & 2.000 & 2.500 \\ -1.000 & 1.648 & 4.745 \\ -2.000 & 4.438 & 7.265 \end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}
=
\begin{pmatrix} 4.501 \\ 5.393 \\ 9.703 \end{pmatrix}
$$

この方程式をガウスの消去法で，4 桁の精度（丸めは四捨五入）で計算してみよう．1 段目の前進消去が終わると以下のようになる．

$$
\begin{pmatrix} 0.001 & 2.000 & 2.500 \\ 0 & 2002 & 2505 \\ 0 & 4004 & 5007 \end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}
=
\begin{pmatrix} 4.501 \\ 4506 \\ 9012 \end{pmatrix}
$$

2 段目の前進消去として，2 行目を 2 倍して 3 行目から引くと

$$
\begin{pmatrix} 0.001 & 2.000 & 2.500 \\ 0 & 2002 & 2505 \\ 0 & 0 & -3.000 \end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}
=
\begin{pmatrix} 4.501 \\ 4506 \\ 0 \end{pmatrix}
$$

となる．引き続いて後退代入を行うと，解が $x_1 = -1.000,\ x_2 = 2.251,\ x_3 = 0.000$ と得られるが，これは本当の解とは大きく異なっている．

このような現象が起きる原因は，今回の場合，前進消去の第 1 段目にある．対角成分が 0.001 と非常に小さいので，1 行目の定数倍を 2 行目と 3 行目から引く際に，絶対値の大きな数と小さな数の演算が生じ，小さい方の数の情報が失われるからである．このような現象を**情報落ち**という．

ガウスの消去法において，$a_{kk}^{(k)}$ をピボットという．情報落ちが出来るだけ生じないようにするには，計算の過程でピボットの絶対値が大きい方がよい．そこで，前進消去の k 段目を行う前に，$|a_{kk}^{(k)}|,\ |a_{k+1\,k}^{(k)}|, \cdots, |a_{nk}^{(k)}|$ を比べて一番大きなものを探し，その行と第 k 行を入れ換えると精度が良くなることが期待できる．このような操作を，ピボット選択という．

先ほどの例では，まず第 1 行と第 3 行を入れ換えて

$$\begin{pmatrix} -2.000 & 4.438 & 7.265 \\ -1.000 & 1.648 & 4.745 \\ 0.001 & 2.000 & 2.500 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 9.703 \\ 5.393 \\ 4.501 \end{pmatrix}$$

としてから計算を開始することになる．このようにして解くと，解が $x_1 = 1.002,\ x_2 = 1.001,\ x_3 = 0.9998$ となり，4 桁の精度で計算していることを考えれば妥当な結果が得られる．

1.3 ガウスの消去法のプログラム

ガウスの消去法のプログラムを以下に示そう．プログラムは C++の機能も用いているので，ファイル名は”〜.c”ではなく”〜.cpp”とする必要がある．

線形演算ライブラリ Liner.h では，既にガウスの消去法の計算が実装されており，行列 A に大して A.Gauss(b) とすることで，$Ax = b$ の解を求めることができる．そこで，既に実装されているガウスの消去法の結果と比較して答え合わせを行っている．

```
#include <stdio.h>
#include <math.h>
#include "Linear.h"

int main()
{
    printf("***** ガウスの消去法 *****\n");
```

次ページへ続く…

```
int n = 3;
Matrix A(n, n); // n × n 次元行列を宣言
Vector b(n), x(n);   // n 次元ベクトルを宣言

A[1][1] = 2; A[1][2] = 4; A[1][3] = 5; b[1] = 1;
A[2][1] = 1; A[2][2] = 6; A[2][3] = 1; b[2] = 2;
A[3][1] = 6; A[3][2] = 2; A[3][3] = 7; b[3] = 3;

// 答え合わせ用にコピーしておく
Matrix A2 = A.Copy();
Vector b2 = b.Copy();

// 前進消去
for(int k=1; k<=n-1; k++){
    // ピボット選択
    int p = k;
    double pv = fabs(A[k][k]);
    for(int i=k+1; i<=n; i++){
        double a = fabs(A[i][k]);
        if(a>pv){
            p = i; pv = a;
        }
    }
    if(p>k){
        for(int i=k; i<=n; i++){
            double a = A[k][i]; A[k][i] = A[p][i]; A[p][i] = a;
        }
        double a = b[k]; b[k] = b[p]; b[p] = a;
    }
    for(int i=k+1; i<=n; i++){
        double a = A[i][k] / A[k][k];
        for(int j=k+1; j<=n; j++){
            A[i][j] -= a*A[k][j];
        }
        b[i] -= a*b[k];
    }
}
```

次ページへ続く…

```
    // 後退代入
    for(int k=n; k>=1; k--){
        x[k] = b[k];
        for(int i=k+1; i<=n; i++){
            x[k] -= A[k][i]*x[i];
        }
        x[k] /= A[k][k];
    }
    printf("解は以下のようになりました\n");
    x.Show();

    printf("答え合わせ\n");
    x = A2.Gauss(b2);
    x.Show();

    return 0;
}
```

プログラムが正しければ，以上のように出力される筈である．

```
***** ガウスの消去法 *****
解は以下のようになりました
0.7872340425531914
0.2553191489361702
-0.3191489361702127
答え合わせ
0.7872340425531914
0.2553191489361702
-0.3191489361702128
```

1.4 定常反復法

零でない要素（非零要素）の個数が行列全体の要素の個数（n 次行列なら n^2 個）に比べて非常に少ない行列を疎行列という．方程式を離散化すると，疎行列を係数に持つ連立一次方程式になることが多い．次数の非常に大きな連立一次方程式を計算機で解くとき，行列の成分を全てメモリに保持するのは現実的ではない．例えば，100 万次元の問題を解く

場合，100 万× 100 万の変数をメモリに確保しようとすると，8T（テラ）バイトのメモリが必要になってしまう．非零要素が対角要素周辺に集まっている場合は，ガウスの消去法を用いても新たな非零成分は対角要素の周辺にしか現れないので，対角成分の周辺のみメモリに確保すればよいが，疎行列であっても非零成分が行列全体に散らばっている場合には，ガウスの消去法を用いると新たな非零成分が行列全体に多数発生してしまい，メモリ消費の点で問題が生じる．そこで用いられるのが反復法である．ここでは，反復法の中でも定常反復法という種類に属する，ヤコビ法，ガウス・ザイデル法，および SOR 法について紹介する．

連立一次方程式の i 行目は

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$$

であるが，これは a_{ii} が零でない場合には

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j\right)$$

と変形できる．そこで，

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right), \qquad i = 1, 2, \cdots, n$$

なる反復を考えてみる．このような反復によって解を求める方法を**ヤコビ法**という．

少々天下りに思えるかもしれないが，例えば高校でやった漸化式

$$a_{n+1} = pa_n + q$$

を思い出すと，この数列は $|p| < 1$ のときに収束し，収束先を a とすると

$$a = pa + q$$

を満たすので，それとの類似で考えれば，ヤコビ法の反復は，対角成分 a_{ii} が非対角成分 $a_{ij}(i \neq j)$ に比べて大きいときに収束することが期待できる．実際，対角成分が大きいような例

$$\begin{pmatrix} 3 & 1 \\ -1 & 5 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$$

について，初期値 $x_1^{(0)} = 0,\ x_2^{(0)} = 0$ に対してヤコビ法を適用すると

$$x_1^{(1)} = \frac{1}{3}(4 - x_2^{(0)}) = 1.33333\cdots \qquad x_2^{(1)} = \frac{1}{5}(4 + x_1^{(0)}) = 0.8$$

$$x_1^{(2)} = \frac{1}{3}(4 - x_2^{(1)}) = 1.06666\cdots \qquad x_2^{(2)} = \frac{1}{5}(4 + x_1^{(1)}) = 1.06666\cdots$$
$$x_1^{(3)} = \frac{1}{3}(4 - x_2^{(2)}) = 0.97777\cdots \qquad x_2^{(3)} = \frac{1}{5}(4 + x_1^{(2)}) = 1.01333\cdots$$
$$x_1^{(4)} = \frac{1}{3}(4 - x_2^{(3)}) = 0.99555\cdots \qquad x_2^{(4)} = \frac{1}{5}(4 + x_1^{(3)}) = 0.99555\cdots$$
$$x_1^{(5)} = \frac{1}{3}(4 - x_2^{(4)}) = 1.00148\cdots \qquad x_2^{(5)} = \frac{1}{5}(4 + x_1^{(4)}) = 0.99911\cdots$$

と，解 $x_1 = 1,\ x_2 = 1$ に近づいていくことがわかる．

方程式を離散化すると，比較的対角成分の大きな行列が得られることが多いので，ヤコビ法は有効に機能することが多い．ただ，いつも収束するとは限らない．収束のための厳密な十分条件については後ほど説明する．

ヤコビ法では，$x_1^{(k)},\ x_2^{(k)}, \cdots, x_n^{(k)}$ を全て求めてから，それらを元に $x_1^{(k+1)},\ x_2^{(k+1)}, \cdots, x_n^{(k+1)}$ を計算している．しかし，$x_i^{(k+1)}$ を計算しようとする時点では既に $x_1^{(k+1)}, \cdots, x_{i-1}^{(k+1)}$ が求まっているので，$x_1^{(k)}, \cdots, x_{i-1}^{(k)}$ よりも最新の結果である $x_1^{(k+1)}, \cdots, x_{i-1}^{(k+1)}$ を使おうというのが，**ガウス・ザイデル法**である．具体的に書くと

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \qquad i = 1, 2, \cdots, n$$

なる反復を行うことになる．ガウス・ザイデル法は x_i について常に最新の結果だけを保持していればいいので，計算上は同じベクトルに上書きしていけばよく，プログラムは簡単になる．

ガウス・ザイデル法は

$$y_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right),$$
$$x_i^{(k+1)} = x_i^{(k)} + (y_i^{(k+1)} - x_i^{(k)}), \qquad i = 1, 2, \cdots, n$$

と書き換えることができる．すなわち，$x_i^{(k)}$ に $(y_i^{(k+1)} - x_i^{(k)})$ を足したものが $x_i^{(k+1)}$ になっている．このとき，足す量を緩和係数 ω で調整して

$$y_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right),$$
$$x_i^{(k+1)} = x_i^{(k)} + \omega(y_i^{(k+1)} - x_i^{(k)}), \qquad i = 1, 2, \cdots, n$$

としたものが **SOR 法**である．緩和係数を調整することにより，ガウス・ザイデル法よりも高速に収束させることができる．緩和係数は $0 < \omega < 2$ の範囲に，もっとも高速に収

束する値があるということが知られているが，問題に応じて色々と試して決定する必要がある．$\omega = 1$ のとき，SOR 法はガウス・ザイデル法に一致する．

ヤコビ法および $0 < \omega \leq 1$ についての SOR 法は

$$A = (a_{ij}), \qquad \sum_{j \neq i} |a_{ij}| < |a_{ii}|, \qquad i = 1, 2, \cdots, n$$

を満たすような行列を係数に持つ連立一次方程式に対して収束することが知られている（ガウス・ザイデル法は SOR 法に含まれる）．このような行列を狭義優対角行列という．狭義優対角行列以外にもヤコビ法や SOR 法が収束するケースは色々とあるのだが，一般的な収束証明は難しいので，ここでは狭義優対角行列の場合にヤコビ法が収束することを証明してみよう．

1.5 ベクトルのノルム

ヤコビ法の収束に関する証明のためには，ベクトルや行列の大きさを測る基準が必要になる．収束というのは，近づいていくことであるが，近づいていくというのは差が小さくなっていくことである．ということは，小さいとか大きいとかを判断する基準が必要になる．そこで，ノルムというものを導入する．なお，ベクトルや行列の要素などは全て実数の範囲で考える．

任意のベクトル x, y に対して以下の 3 条件を満たすような実数値関数 $\|\cdot\|$ をノルムという．

(1) 正値性　　$\|x\| \geq 0$ であり，　$\|x\| = 0 \iff x = 0$
(2) 同次性　　任意の実数 α に対して $\|\alpha x\| = |\alpha| \|x\|$
(3) 三角不等式　$\|x + y\| \leq \|x\| + \|y\|$

さらに，$1 \leq p \leq \infty$ とし，

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

に対して

$$\|x\|_p = \begin{cases} \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} & 1 \leq p < \infty \\ \max\limits_{1 \leq i \leq n} |x_i| & p = \infty \end{cases}$$

と定義すると，$\|\cdot\|_p$ はノルムとなる．

ここで，$p=\infty$ のときだけ式の形が違っていて，おやっ，と思う人がいるかもしれないが，$p\to\infty$ のときの極限を考えれば $p=\infty$ のときの形になる．実際，$M=\max\limits_{1\le i\le n}|x_i|$ とすると，

$$M=(M^p)^{1/p}\le\left(\sum_{i=1}^n|x_i|^p\right)^{1/p}\le(nM^p)^{1/p}=n^{1/p}M\longrightarrow M\ \ (p\to\infty)$$

が成り立つ．よって挟み打ちの原理より

$$\left(\sum_{i=1}^n|x_i|^p\right)^{1/p}\ \to\ \max_{1\le i\le n}|x_i|\qquad(p\to\infty)$$

となる．

さて，$\|\cdot\|_p$ がノルムになることを $p=1,2,\infty$ の場合に証明しておこう（ここでは証明しないが，一般の $p\ge1$ 場合もそれほど難しくない）．ノルムの条件のうち，(1) と (2) は難しくないので，(3) の三角不等式のみ示す．

$$x=\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix},\qquad y=\begin{pmatrix}y_1\\y_2\\\vdots\\y_n\end{pmatrix}$$

とおくと，$p=1$ のときは

$$\|x+y\|_1=\sum_{i=1}^n|x_i+y_i|\le\sum_{i=1}^n(|x_i|+|y_i|)=\sum_{i=1}^n|x_i|+\sum_{i=1}^n|y_i|=\|x\|_1+\|y\|_1$$

$p=\infty$ のときには

$$\|x+y\|_\infty=\max_{1\le i\le n}|x_i+y_i|\le\max_{1\le i\le n}(|x_i|+|y_i|)\le\max_{1\le i\le n}|x_i|+\max_{1\le i\le n}|y_i|=\|x\|_\infty+\|y\|_\infty$$

より成り立つ．$p=2$ のときは少し難しいが，コーシー・シュワルツの不等式を用いると

$$\begin{aligned}\|x+y\|_2^2&=\sum_{i=1}^n|x_i+y_i|^2\le\sum_{i=1}^n|x_i|^2+\sum_{i=1}^n|y_i|^2+2\sum_{i=1}^n|x_i||y_i|\\&\le\sum_{i=1}^n|x_i|^2+\sum_{i=1}^n|y_i|^2+2\sqrt{\sum_{i=1}^n|x_i|^2\sum_{i=1}^n|y_i|^2}=\left(\sqrt{\sum_{i=1}^n|x_i|^2}+\sqrt{\sum_{i=1}^n|y_i|^2}\right)^2\\&=(\|x\|_2+\|y\|_2)^2\end{aligned}$$

と証明できる．

1.6 行列のノルム

行列に対してもノルムを定義する．いま，$\|\cdot\|$ をベクトルのノルムとするとき，行列 A のノルムを

$$\|A\| = \max_{x\neq 0} \frac{\|Ax\|}{\|x\|}$$

と定義する．とくに，$1 \leq p \leq \infty$ に対して

$$\|A\|_p = \max_{x\neq 0} \frac{\|Ax\|_p}{\|x\|_p}$$

と定義する．

行列ノルムの意味は，$\|Ax\|$ が $\|x\|$ に比べてどれくらい大きくなるか考え，その比率が最大になるときの比率を行列の大きさとする，ということである．

行列ノルムは以下の性質を満たす．ただし，O は零行列，I は単位行列である．

(1) $\|A\| \geq 0$ であり，　$\|A\| = 0 \iff A = O$
(2) 任意の実数 α に対して $\|\alpha A\| = |\alpha|\|A\|$
(3) $\|A + B\| \leq \|A\| + \|B\|$
(4) $\|Ax\| \leq \|A\|\|x\|$
(5) $\|AB\| \leq \|A\|\|B\|$
(6) $\|I\| = 1$

(1),(2),(3),(6) は難しくない．(4) は $x \neq 0$ のとき

$$\frac{\|Ax\|}{\|x\|} \leq \max_{y\neq 0} \frac{\|Ay\|}{\|y\|} = \|A\|$$

となることからすぐにわかり，(5) は

$$\|AB\| = \max_{x\neq 0} \frac{\|ABx\|}{\|x\|} \leq \max_{x\neq 0} \frac{\|A\|\|Bx\|}{\|x\|} = \|A\| \max_{x\neq 0} \frac{\|Bx\|}{\|x\|} = \|A\|\|B\|$$

よりわかる．

$p = 1$ のときと $p = \infty$ のときは，行列ノルムは行列の要素を用いて具体的に書くことができる．すなわち，n 次正方行列 $A = (a_{ij})$ について次が成り立つ．

(1) $\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{n} |a_{ij}|$
(2) $\|A\|_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}|$

以下，

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

とする．まず (1) を示そう．$\alpha_j = \sum_{i=1}^{n} |a_{ij}|$ とおくと

$$\|Ax\|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} a_{ij} x_j \right| \leq \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|\,|x_j| = \sum_{j=1}^{n} \left(\sum_{i=1}^{n} |a_{ij}| \right) |x_j|$$
$$= \sum_{j=1}^{n} \alpha_j |x_j| \leq \max_{1\leq j\leq n} \alpha_j \sum_{j=1}^{n} |x_j| = \max_{1\leq j\leq n} \alpha_j \|x\|_1$$

よって

$$\|A\|_1 = \max_{x\neq 0} \frac{\|Ax\|_1}{\|x\|_1} \leq \max_{x\neq 0} \frac{\max_{1\leq j\leq n} \alpha_j \|x\|_1}{\|x\|_1} = \max_{1\leq j\leq n} \alpha_j$$

一方で，$\alpha_1, \alpha_2, \cdots, \alpha_n$ のうちで最大のもの（最大のものが複数ある場合はそのうち一つ）を α_k とし，

$$v = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \leftarrow k \text{ 番目}$$

とすると，

$$\|A\|_1 = \max_{x\neq 0} \frac{\|Ax\|_1}{\|x\|_1} \geq \frac{\|Av\|_1}{\|v\|_1} = \frac{\sum_{i=1}^{n} |a_{ik}|}{1} = \alpha_k = \max_{1\leq j\leq n} \alpha_j$$

よって

$$\|A\|_1 = \max_{1\leq j\leq n} \alpha_j = \max_{1\leq j\leq n} \sum_{i=1}^{n} |a_{ij}|$$

が言えた.

(2) については，$\alpha_i = \sum_{j=1}^{n} |a_{ij}|$ とおくと

$$\|Ax\|_\infty = \max_{1\leq i\leq n} \left| \sum_{j=1}^{n} a_{ij} x_j \right| \leq \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}|\,|x_j| \leq \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}| \cdot \max_{1\leq j\leq n} |x_j|$$

$$= \max_{1\le i\le n} \alpha_i \max_{1\le j\le n} |x_j| = \max_{1\le i\le n} \alpha_i \|x\|_\infty$$

よって

$$\|A\|_\infty = \max_{x\neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty} \le \max_{x\neq 0} \frac{\max_{1\le i\le n} \alpha_i \|x\|_\infty}{\|x\|_\infty} = \max_{1\le i\le n} \alpha_i$$

一方で，$\alpha_1, \alpha_2, \cdots, \alpha_n$ のうちで最大のもの（最大のものが複数ある場合はそのうち一つ）を α_k とし，

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}, \qquad v_j = \begin{cases} 1 & (a_{kj} \ge 0) \\ -1 & (a_{kj} < 0) \end{cases}$$

とすると，

$$\begin{aligned} \|A\|_\infty = \max_{x\neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty} &\ge \frac{\|Av\|_\infty}{\|v\|_\infty} = \frac{\max_{1\le i\le n} \left| \sum_{j=1}^{n} a_{ij} v_j \right|}{1} \\ &\ge \left| \sum_{j=1}^{n} a_{kj} v_j \right| = \sum_{j=1}^{n} |a_{kj}| = \alpha_k = \max_{1\le i\le n} \alpha_i \end{aligned}$$

よって

$$\|A\|_\infty = \max_{1\le i\le n} \alpha_i = \max_{1\le i\le n} \sum_{j=1}^{n} |a_{ij}|$$

が言えた.

1.7 ヤコビ法の収束証明

ヤコビ法の反復は

$$x^{(k+1)} = Mx^{(k)} + c,$$

$$M = -\begin{pmatrix} 0 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & 0 & \frac{a_{23}}{a_{22}} & \cdots & \frac{a_{2n}}{a_{22}} \\ \frac{a_{31}}{a_{33}} & \frac{a_{32}}{a_{33}} & 0 & \cdots & \frac{a_{3n}}{a_{33}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{a_{n1}}{a_{nn}} & \frac{a_{n2}}{a_{nn}} & \frac{a_{n3}}{a_{nn}} & \cdots & 0 \end{pmatrix}, \qquad c = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \frac{b_3}{a_{33}} \\ \vdots \\ \frac{b_n}{a_{nn}} \end{pmatrix}$$

と書ける．ここで，元の連立一次方程式の解を x^* とすると

$$x^* = Mx^* + c$$

より

$$x^{(k+1)} - x^* = M(x^{(k)} - x^*)$$

が成り立つ．したがって

$$x^{(k)} - x^* = M^k(x^{(0)} - x^*)$$

より

$$\|x^{(k)} - x^*\|_\infty = \|M^k(x^{(0)} - x^*)\|_\infty \leq \|M\|_\infty^k \|x^{(0)} - x^*\|_\infty$$

が成り立つので，$\|M\|_\infty < 1$ ならばヤコビ法が収束するということがわかる．

実際に計算すると

$$\|M\|_\infty = \max_{1\leq i\leq n} \sum_{j\neq i} \left|\frac{a_{ij}}{a_{ii}}\right| = \max_{1\leq i\leq n} \frac{1}{|a_{ii}|} \sum_{j\neq i} |a_{ij}|$$

となり，A の狭義優対角性から $\|M\|_\infty < 1$ がいえる．よって，狭義優対角行列を係数に持つような連立一次方程式に対してヤコビ法が収束するということが証明できた．

1.8 ノルムの同値性

前節では無限大ノルムを用いてヤコビ法の収束を示したが，他のノルムではどうなるか，疑問に思う人もいるかもしれない。例えば，無限大ノルムでは収束しても 2 ノルムで収束していない，などということがあるだろうか？そこで，その疑問に答えるために，ノルムの同値性について説明する．

2 つのノルム $\|\cdot\|_A$ と $\|\cdot\|_B$ があったとき，正の実数 α, β が存在して，任意のベクトル x に対して

$$\alpha\|x\|_A \leq \|x\|_B \leq \beta\|x\|_A$$

が成り立つとき，この 2 つのノルムは同値であるという．この式は，ノルム $\|\cdot\|_B$ を上と下からノルム $\|\cdot\|_A$ で挟む形になっているが，書き換えると

$$\frac{1}{\beta}\|v\|_B \leq \|v\|_A \leq \frac{1}{\alpha}\|v\|_B$$

となるので，ノルム $\|\cdot\|_A$ を上と下からノルム $\|\cdot\|_B$ で挟む形にもできる．

実は，有限次元ベクトルに対するノルムは全て同値である，という事実があるので，あるノルムで収束すれば別のノルムでも収束するのである．この事実を一般のノルムについて証明するのは少し難しいので，ここでは，p ノルムが全て同値であるということを証明し

よう．特に，$1 \leq p < \infty$ を満たす任意の p について，p ノルムが無限大ノルムと同値であることを示せばよい．
既に以前，同じ式を用いたが，n 次元ベクトル x の各要素を x_i とし，$M = \max_{1 \leq i \leq n} |x_i|$ とすると，

$$M = (M^p)^{1/p} \leq \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \leq (nM^p)^{1/p} = n^{1/p} M$$

より

$$\|x\|_\infty \leq \|x\|_p \leq n^{1/p} \|x\|_\infty$$

が成り立つ．よって p ノルムと無限大ノルムは同値である．

1.9 ヤコビ法のプログラム

参考までにヤコビ法のプログラムを紹介しておく．行列は 10 次元の 3 重対角行列とした．定常反復法においては，いつ反復を終了するかが重要になるが，ここでは相対残差ノルム

$$\frac{\|b - Ax\|_\infty}{\|b\|_\infty}$$

が予め決めたある小さな値より小さくなったときに反復を終了するものとする．以下のプログラムでは，相対残差ノルムが 10^{-12} 以下になったときに反復を終了している．

```
#include <stdio.h>
#include "Linear.h"

int main()
{
    printf("***** ヤコビ法 *****\n");

    int n = 10;

    Matrix A(n, n);
    Vector b(n);
```

次ページへ続く…

```
//行列と右辺ベクトルを設定
for(int i=1; i<=n; i++){
    for(int j=1; j<=n; j++){
        if(i==j){
            A[i][j] = 4;
        } else if(i==j+1){
            A[i][j] = 1;
        } else if(i==j-1){
            A[i][j] = -2;
        } else {
            A[i][j] = 0;
        }
    }
    b[i] = i;
}

Vector x(n), x2(n);

// 初期値は0ベクトル
x.Clear();

//反復開始
int f = 0; //フラグ: 1=計算成功, 0=計算失敗
int k;
for(k=1; k<=1000; k++){
    for(int i=1; i<=n; i++){
        x2[i] = b[i];
        for(int j=1; j<=n; j++){
            if(j != i){
                x2[i] -= A[i][j] * x[j];
            }
        }
        x2[i] /= A[i][i];
    }
    x = x2.Copy();
```

次ページへ続く…

```
        // 収束判定
        Vector z = b - A*x; // 残差ベクトル
        double r = z.NormInfty() / b.NormInfty(); // 相対残差ノルム
        if(r < 1.0E-12){
            f = 1; // 成功
            break;
        }
        if(r > 1.0E100){
            break; // 失敗
        }
    }

    if(f==0){
        printf("収束しませんでした\n");
    } else {
        printf("反復%d 回目で収束\n", k);
        printf("解は以下のようになりました\n");
        x.Show();
        printf("答え合わせ\n");
        x = A.Gauss(b);
        x.Show();
    }

    return 0;
}
```

相対残差ノルムが 10^{100} を超えたり，1000 回の反復でも終了しなかった場合には,「収束しませんでした」と表示するようにしている．
x2 の値を x に上書きするのに x = x2.Copy(); と Copy メソッドを用いているが，ここを単に x = x2; としてしまうと，反復の二巡目以降，x2 と x が同じものになってしまい，ガウス・ザイデル法と同じになってしまう．

ガウス・ザイデル法のプログラムは，前にも述べた通り，常に最新の結果のみ保持していればよいので，ヤコビ法のプログラムから x2 を消し，常に x を用いるようにすればよい．すなわち，ヤコビ法のプログラムを以下のように変更すればよい．

```
--------- 前略 ---------
    for(k=1; k<=1000; k++){
        for(int i=1; i<=n; i++){
            x[i] = b[i];
            for(int j=1; j<=n; j++){
                if(j != i){
                    x[i] -= A[i][j] * x[j];
                }
            }
            x[i] /= A[i][i];
        }
--------- 後略 ---------
```

また，SOR 法については，ヤコビ法のプログラムを以下のように変更すればよい．

```
--------- 前略 ---------
    double w = 0.7;

    //反復開始
    int f = 0; //フラグ: 1=計算成功，0=計算失敗
    int k;
    for(k=1; k<=1000; k++){
        for(int i=1; i<=n; i++){
            double y = b[i];
            for(int j=1; j<=n; j++){
                if(j! = i){
                    y -= A[i][j] * x[j];
                }
            }
            y /= A[i][i];
            x[i] = x[i] + w*(y - x[i]);
        }
--------- 後略 ---------
```

実際にガウス・ザイデル法や SOR 法についてもプログラムして，収束の速さを比べてみるとよい．

1.10　補足

時間の都合で触れなかった事項についてここで述べておく．

一度きりの計算ではガウスの消去法はそこそこ効率的だが，ある決まった行列 A について，複数の右辺ベクトル $b^{(1)}, b^{(2)}, \cdots, b^{(l)}$ に対してそれぞれ $Ax = b^{(j)}$ の解を求めたい場合には，ガウスの消去法より効率の良い手法として LU 分解というアルゴリズムが知られている．

数値解析においては，行列の固有値や固有ベクトルを計算したいことも多い．そのようなときには，コレスキー分解や QR 分解が用いられることが多い．

本章では，連立一次方程式の反復解法としてヤコビ法，ガウス・ザイデル法，SOR 法について述べたが，これらの方法は歴史的には重要だが，実際にはあまり使われない（全く使われないというわけではないが）．そのかわり，反復法としては CG 法という方法が現在よく用いられている．ただ，CG 法については説明が煩雑になるので説明は省略した．

1.11　課題

課題１　ガウスの消去法の前進消去では，行列を上三角行列に変換していくことがポイントである．しかし，実際のプログラムでは行列は上三角行列には変形されていない（添え字の動く範囲を確認すること）．それでもちゃんと解が計算できる理由を説明せよ．

課題２　ガウスの消去法のプログラムにおいて，ピボット選択を用いない場合に精度が悪くなるような例を考え，計算結果を比較せよ．

課題３　10 次元以上の同じ連立一次方程式を，ヤコビ法，ガウス・ザイデル法，パラメータを色々と変えた SOR 法で解き，計算結果を比較せよ．

課題４　ヤコビ法は

$$A = (a_{ij}), \qquad \sum_{i \neq j} |a_{ij}| < |a_{jj}|, \qquad j = 1, 2, \cdots, n$$

を満たす行列に関する連立一次方程式についても収束することを示せ．

2　非線形方程式

本章では，非線形方程式の解法について考える．

2.1　二分法

1 変数の非線形方程式 $F(x) = 0$ の解を求めたいとする．一番素朴な方法は，中間値の定理（の変形バージョン）を利用する方法だろう．すなわち，$F(x)$ が連続のとき，

「$a < b$ について，$F(a)$ と $F(b)$ が異符号もしくはどちらかが 0 ならば，$[a, b]$ に $F(x) = 0$ の解が存在する」

という事実を用いる方法である．具体的には，まず，$\underline{x}^{(0)} < \overline{x}^{(0)}$ かつ $F(\underline{x}^{(0)})F(\overline{x}^{(0)}) \leq 0$ となるような初期値 $\underline{x}^{(0)}, \overline{x}^{(0)}$ を求め，$k = 0, 1, 2, \cdots$ について以下のような反復を行う．

$$
\begin{aligned}
& m^{(k)} = \frac{\underline{x}^{(k)} + \overline{x}^{(k)}}{2} \\
& (1) \quad F(\underline{x}^{(k)})F(m^{(k)}) \leq 0 \text{ のとき} \\
& \qquad \underline{x}^{(k+1)} = \underline{x}^{(k)}, \quad \overline{x}^{(k+1)} = m^{(k)}, \\
& (2) \quad F(m^{(k)})F(\overline{x}^{(k)}) \leq 0 \text{ のとき} \\
& \qquad \underline{x}^{(k+1)} = m^{(k)}, \quad \overline{x}^{(k+1)} = \overline{x}^{(k)}.
\end{aligned}
$$

(1),(2) の両方が成立するときには，どちらを選んでもよい．

二分法については，方程式の真の解を $\hat{x}$ とすると，

$$
|\hat{x} - m^{(k)}| \leq \frac{1}{2}\left(\overline{x}^{(k)} - \underline{x}^{(k)}\right) = \frac{1}{2^{k+1}}\left(\overline{x}^{(0)} - \underline{x}^{(0)}\right)
$$

が成り立つ．すなわち，$m^{(k)}$ を解の近似値とすると，反復するごとに誤差は半分になることがわかる．

二分法の利点は，$F(x)$ が微分可能でなくとも，連続でありさえすれば適用可能であること，さらに，解の範囲が明確に得られることである．

二分法を用いて $e^{-x} = x$ の解を求めるプログラムを以下に示す．

```
#include <stdio.h>
#include <math.h>

double f(double x)
{
    return exp(-x)-x;
}

int main()
{
    printf("***** 二分法 *****\n");

    // 初期値
    double xl = 0, xh = 1;

    if( f(xl)*f(xh)>0 ){
        printf("関数値が異符号となるように２つの初期値を取って下さい\n");
        return 0;
    }

    for(int k=1; k<=60; k++){
        double mid = (xl+xh)/2;
        if( f(xl)*f(mid)<=0 ){
            xh = mid;
        } else {
            xl = mid;
        }
        printf("%d 回目の反復の結果、解の範囲は [%.15f,%.15f]\n",k,xl,xh);
    }
    return 0;
}
```

2.2 一次元ニュートン法

次に，二分法より速く収束する計算方法であるニュートン法を紹介する．まずは 1 変数の場合を考えてみよう．いま，非線形方程式 $F(x) = 0$ の近似解 $\tilde{x}$ があったとして，この近似解の精度を改善することを試みる．F の微分可能性を仮定すれば，F を一次の項までテ

イラー展開することにより

$$F(x) \approx F(\tilde{x}) + (x - \tilde{x})F'(\tilde{x})$$

と近似できる．よって，右辺 $= 0$ を x について解き，

$$x = \tilde{x} - \frac{F(\tilde{x})}{F'(\tilde{x})}$$

として x を求めれば，$\tilde{x}$ よりも良い近似解になると期待できる．これは要するに，関数 $F(x)$ を，$\tilde{x}$ における接線で近似し，解を求めていることになる．よって，適当な初期値 $x^{(0)}$ から出発し，

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})}$$

なる反復を行えば，$\{x^{(k)}\}$ は解に収束することが期待できる．これが一次元ニュートン法である．ニュートン法は，適当な滑らかさの仮定の下に，初期値を真の解の近くに取れば，解へ収束することが保証されている．初期値が真の解から遠い場合は，収束しないこともある．

それでは，実際にニュートン法を用いて非線形方程式を解いてみよう．例として，二分法のときと同じ $e^{-x} = x$ の数値解を考える．

まず，方程式を $F(x) = 0$ の形に直す．この例の場合は $F(x) = e^{-x} - x$ とすればよい．よって，ニュートン法の反復は

$$x^{(k+1)} = x^{(k)} - \frac{e^{-x^{(k)}} - x^{(k)}}{-e^{-x^{(k)}} - 1}$$

となる．実際のプログラムは以下のようになる．

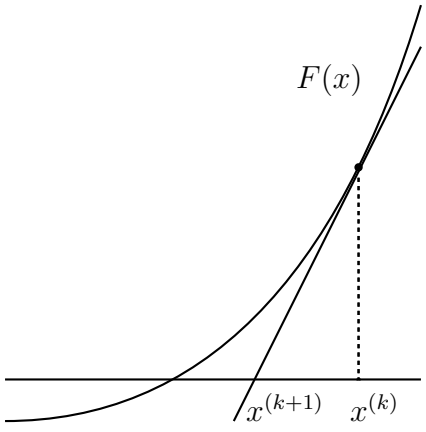

図 1: ニュートン法

```
#include <stdio.h>
#include <math.h>

double f(double x)
{
    return exp(-x)-x;
}

double df(double x)
{
    return -exp(-x)-1;
}

int main()
{
    printf("***** 一次元ニュートン法 *****\n");

    // 初期値
    double x = 0;

    for(int k=1; k<=10; k++){
        double diff = f(x)/df(x); // 変化量
        x -= diff;
        printf("x^(%d)=%.15f, 変化量=%.15f\n", k, x, diff);
    }
    return 0;
}
```

これを実行すると

$$
\begin{aligned}
x^{(1)} &= 0.537882842739990, && \text{変化量} = 0.462117157260010 \\
x^{(2)} &= 0.566986991405413, && \text{変化量} = -0.029104148665423 \\
x^{(3)} &= 0.567143285989123, && \text{変化量} = -0.000156294583710 \\
x^{(4)} &= 0.567143290409784, && \text{変化量} = -0.000000004420661 \\
x^{(5)} &= 0.567143290409784, && \text{変化量} = -0.000000000000000 \\
x^{(6)} &= 0.567143290409784, && \text{変化量} = -0.000000000000000 \\
x^{(7)} &= 0.567143290409784, && \text{変化量} = -0.000000000000000
\end{aligned}
$$

$$x^{(8)} = 0.567143290409784, \qquad \text{変化量} = -0.000000000000000$$
$$x^{(9)} = 0.567143290409784, \qquad \text{変化量} = -0.000000000000000$$
$$x^{(10)} = 0.567143290409784, \qquad \text{変化量} = -0.000000000000000$$

となり，誤差が急激に減少することがわかる．

2.3 一次元ニュートン法の収束速度

ここでは一次元ニュートン法がどれくらいの速度で収束するのか調べてみよう．

$F(x) = 0$ の真の解を $\hat{x}$ とし，$F(x)$ は $\hat{x}$ を含むある閉区間 I で C^2 級であるものとする．また，区間 I で $F'(x)$ は 0 にならないものとし，

$$M = \max_{x \in I} |F''(x)|, \qquad m = \min_{x \in I} |F'(x)| > 0$$

とする．このとき，$x^{(k)} \in I$ とし，$F(x)$ を $x^{(k)}$ のまわりで 2 次までテイラー展開し，$x = \hat{x}$ を代入すると

$$F(\hat{x}) = F(x^{(k)}) + (\hat{x} - x^{(k)})F'(x^{(k)}) + \frac{(\hat{x} - x^{(k)})^2}{2}F''(\xi)$$

となる．ここで，2 次の項はラグランジュの剰余項であり，ξ は $x^{(k)}$ と $\hat{x}$ の間の実数である．さて，$F(\hat{x}) = 0$ より

$$-F(x^{(k)}) = (\hat{x} - x^{(k)})F'(x^{(k)}) + \frac{(\hat{x} - x^{(k)})^2}{2}F''(\xi)$$

が成り立つので，これを用いてニュートン法の反復の式を変形すると

$$\begin{aligned} x^{(k+1)} - \hat{x} &= x^{(k)} - \hat{x} - \frac{F(x^{(k)})}{F'(x^{(k)})} \\ &= x^{(k)} - \hat{x} + \frac{(\hat{x} - x^{(k)})F'(x^{(k)}) + (\hat{x} - x^{(k)})^2 F''(\xi)/2}{F'(x^{(k)})} \\ &= \frac{(\hat{x} - x^{(k)})^2 F''(\xi)}{2F'(x^{(k)})} \end{aligned}$$

となる．よって

$$|x^{(k+1)} - \hat{x}| \leq \frac{M}{2m}|x^{(k)} - \hat{x}|^2$$

が成り立つ．これを変形すると

$$\frac{M}{2m}|x^{(k+1)} - \hat{x}| \leq \left(\frac{M}{2m}|x^{(k)} - \hat{x}|\right)^2$$

となるので，

$$\frac{M}{2m}|x^{(k)}-\hat{x}| \leq \left(\frac{M}{2m}|x^{(0)}-\hat{x}|\right)^{2^k}$$

なる誤差評価が得られる．つまり，

$$|x^{(0)}-\hat{x}| < \frac{2m}{M}$$

を満たすように初期値を取れば，ニュートン法は収束することがわかる．さらに，

$$\log_{10}\left(\frac{M}{2m}|x^{(k+1)}-\hat{x}|\right) \leq 2\log_{10}\left(\frac{M}{2m}|x^{(k)}-\hat{x}|\right)$$

より，ニュートン法の数値解が真の解に近いときには，一回反復を行うごとに，解の有効桁数が倍になることがわかる．

一般に，解が

$$|x^{(k+1)}-\hat{x}| \leq C|x^{(k)}-\hat{x}|^p$$

なる形で収束するとき，数列 $\{x^{(k)}\}$ もしくはその反復法は p 次収束するという．ニュートン法は $\hat{x}$ のまわりで $F'(x) \neq 0$ のとき，2 次収束することがわかる．

2.4 多次元ニュートン法

次に，一次元ニュートン法の考え方を多次元に拡張する．$F(x)$ を n 変数 n 次元関数とする．ここで，

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \qquad F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{pmatrix}$$

とする．さて，$F=0$ の近似解を

$$\tilde{x} = \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \vdots \\ \tilde{x}_n \end{pmatrix}$$

とし，この近似解のまわりで F を一次の項までテイラー展開すると

$$F(x) \approx F(\tilde{x}) + [F'(\tilde{x})](x-\tilde{x})$$

と近似できる．ここで，$[F'(x)]$ はヤコビ行列であり，

$$[F'(x)] = \begin{pmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} & \cdots & \dfrac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \dfrac{\partial f_n}{\partial x_2} & \cdots & \dfrac{\partial f_n}{\partial x_n} \end{pmatrix}$$

で定義される．よって，一次元ニュートン法の場合と同様に考えると，

$$x^{(k+1)} = x^{(k)} - [F'(x^{(k)})]^{-1} F(x^{(k)})$$

なる反復を行うことで数値解を計算することができる．

多次元ニュートン法を別の視点から考えてみよう．一変数の場合，ニュートン法の反復は，関数を近似解における接線で近似することにより，より良い近似解を求めることであった．同様に，多変数の場合，ニュートン法は，関数を接平面で近似して近似解を計算していると考えることができる．それを具体的に見ていこう．

$f_k(x)$ に $x = \tilde{x}$ で接する接平面を

$$y = f_k(\tilde{x}) + a_{k1}(x_1 - \tilde{x}_1) + a_{k2}(x_2 - \tilde{x}_2) + \cdots + a_{kn}(x_n - \tilde{x}_n)$$

とする．ここで，実際にこれが接平面となっていることから，$l = 1, 2, \cdots, n$ について

$$\frac{\partial f_k}{\partial x_l}(\tilde{x}) = \frac{\partial y}{\partial x_l} = a_{kl}$$

が成り立つ．よって，この接平面は

$$y = f_k(\tilde{x}) + \frac{\partial f_k}{\partial x_1}(\tilde{x}) \cdot (x_1 - \tilde{x}_1) + \frac{\partial f_k}{\partial x_2}(\tilde{x}) \cdot (x_2 - \tilde{x}_2) + \cdots + \frac{\partial f_k}{\partial x_n}(\tilde{x}) \cdot (x_n - \tilde{x}_n)$$

と書ける．よって，これを $f_k(x)$ の近似として用いれば，$F(x)$ は

$$F(x) \approx F(\tilde{x}) + [F'(\tilde{x})](x - \tilde{x})$$

と近似できることがわかる．

ニュートン法の反復には

$$x^{(k+1)} = x^{(k)} - [F'(x^{(k)})]^{-1} F(x^{(k)})$$

のようにヤコビ行列の逆行列が出てくるが，実際の数値計算においては，

$$\begin{cases} [F'(x^{(k)})]d^{(k)} = F(x^{(k)}) \\ \qquad x^{(k+1)} = x^{(k)} - d^{(k)} \end{cases}$$

と考え，第一式の連立一次方程式を解いて $x^{(k)}$ から $d^{(k)}$ を求め，それを元に $x^{(k+1)}$ を求める．逆行列を求めるよりも連立一次方程式を解く方が楽なので，逆行列は計算しないことに注意する．ただし，$n = 2$ の場合は逆行列を用いてもよい．

2.5　多次元ニュートン法のプログラム

3 次元ニュートン法のプログラムは以下のようになる.

```
#include <stdio.h>
#include <math.h>
#include "Linear.h"

// x^2+y^2+z^2=1
// y=sin(x)
// z=x+y
// を解く。プログラムでは x を x[1]、y を x[2]、z を x[3] とする。
// F(x) = ( x[1]^2+x[2]^2+x[3]^2-1, x[2]-sin(x[1]), x[3]-x[1]-x[2] )

// 関数値
Vector f(Vector x)
{
    Vector y(3);
    y[1] = x[1]*x[1] + x[2]*x[2] + x[3]*x[3] - 1;
    y[2] = x[2] - sin(x[1]);
    y[3] = x[3] - x[1] - x[2];
    return y;
}

// ヤコビ行列
Matrix df(Vector x)
{
    Matrix y(3, 3);
    y[1][1] = 2*x[1];     y[1][2] = 2*x[2]; y[1][3] = 2*x[3];
    y[2][1] = -cos(x[1]); y[2][2] = 1;      y[2][3] = 0;
    y[3][1] = -1;         y[3][2] = -1;     y[3][3] = 1;
    return y;
}
```

次ページへ続く…

```
int main()
{
    printf("***** 多次元ニュートン法 *****\n");

    Vector x(3);

    // 初期値
    x[1]=0.5; x[2]=0.5; x[3] = 0.5;
    for(int k=1; k<=10; k++){
        Matrix J = df(x);
        x = x - J.Gauss(f(x));
        printf("x^(%d)=(%.15f,%.15f,%.15f)\n",k,x[1],x[2],x[3]);
    }

    return 0;
}
```

計算結果が 2 次収束しているようなら，プログラムは正しいと考えられる．

2.6 補足

本稿では触れなかったが，真の解で一階微分が 0 になったり，ヤコビ行列が非正則になるような場合，ニュートン法は二次収束ではなく一次収束することが知られている．

一般的にニュートン法は，初期値を近似解の近くに取れば急速に真の解に近づくが，真の解から離れた初期値では上手く収束しないことが多い．そのため，まずは二分法などの他の方法で，ある程度，真の解に近い近似解を計算し，最後の仕上げにニュートン法を用いることが多い．

解を求めたい方程式が代数方程式，つまり

$$x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{n-1} x + a_n = 0$$

なる形の方程式の場合には，ニュートン法ではなく，DKA 法など，代数方程式に特化した方法を用いることが多い．DKA 法を用いると，n 次元方程式の n 個の解を同時に求めることができる．

2.7　課題

課題１　$\sin x = 0$ をニュートン法で解くとき，どの範囲に初期値を取れば解 $x = 0$ に収束するか，数値実験により調べなさい（-1.57〜1.57 の範囲で 0.01 刻みに初期値を取って調べよ）.

課題２　$\sin x = 0$ をニュートン法で解くとき，必ず解 $x = 0$ に収束するような初期値の範囲を本文を参考に求めなさい（まず区間 I を定める必要がある）.

課題３　4 変数以上の非線形方程式を一つ考え，多次元ニュートン法を用いて解け．その際，収束の様子を観察し，2 次収束していることを確認すること.

3　数値積分

本章では，関数 $f(x)$ の積分

$$Q = \int_a^b f(x)dx$$

の計算について考える．積分は，曲線で囲まれた部分の面積を求める場合などに必要になってくる．

関数 $f(x)$ と a, b の値によっては

$$\int_0^1 x^2 dx = \frac{1}{3}, \qquad \int_1^2 \frac{dx}{x} = \log 2, \qquad \int_{-1}^1 \frac{dx}{1+x^2} = \frac{\pi}{2}$$

などのように，公式を用いて数学的に積分値を計算することができるが，公式が無いような場合には，数学的に積分値を厳密に計算することができない場合が多い．そこで，数値的な近似計算を考える．

3.1　積分則

以下，c は a と b の中点，すなわち $c = \dfrac{a+b}{2}$ とする．

積分を近似するため，関数 $f(x)$ について次の 3 種類の近似を考える．

(a)　$f(x)$ を，a と b の中点での値 $f(c)$ を取る定数関数 $\tilde{f}_0(x) = f(c)$ で近似する．

(b)　$f(x)$ を，$(a, f(a))$ と $(b, f(b))$ を通る一次関数 $\tilde{f}_1(x)$ で近似する．ここで

$$\tilde{f}_1(x) = \frac{b-x}{b-a} f(a) + \frac{x-a}{b-a} f(b)$$

である．実際，$x = a, b$ を代入すれば $\tilde{f}_1(a) = f(a)$ かつ $\tilde{f}_1(b) = f(b)$ となることが確かめられる．

(c)　$f(x)$ を，$(a, f(a))$ と $(c, f(c))$ と $(b, f(b))$ を通る二次関数 $\tilde{f}_2(x)$ で近似する．ここで

$$\tilde{f}_2(x) = \frac{(b-x)(c-x)}{(b-a)(c-a)} f(a) + \frac{(b-x)(x-a)}{(b-c)(c-a)} f(c) + \frac{(x-c)(x-a)}{(b-c)(b-a)} f(b)$$

である．実際，$x = a, b, c$ を代入すれば $\tilde{f}_2(a) = f(a),\ \tilde{f}_2(c) = f(c),\ \tilde{f}_2(b) = f(b)$ が成り立つことが確かめられる．

これらの近似関数を用いて，積分値 Q について以下の近似を考える．

中点則：

$$R_{a,b}(f) = \int_a^b \tilde{f}_0(x)dx = (b-a)f(c) = (b-a)f\left(\frac{a+b}{2}\right)$$

台形則：

$$\begin{aligned}
T_{a,b}(f) &= \int_a^b \tilde{f}_1(x)dx \\
&= \frac{f(a)}{b-a}\int_a^b (b-x)dx + \frac{f(b)}{b-a}\int_a^b (x-a)dx \\
&= -\frac{f(a)}{b-a}\int_{b-a}^0 t\,dt + \frac{f(b)}{b-a}\int_0^{b-a} t\,dt \\
&= \frac{b-a}{2}\Big(f(a)+f(b)\Big)
\end{aligned}$$

シンプソン則：

$$\begin{aligned}
S_{a,b}(f) &= \int_a^b \tilde{f}_2(x)dx \\
&= \frac{f(a)}{(b-a)(c-a)}\int_a^b (b-x)(c-x)dx + \frac{f(c)}{(b-c)(c-a)}\int_a^b (b-x)(x-a)dx \\
&\qquad + \frac{f(b)}{(b-c)(b-a)}\int_a^b (x-c)(x-a)dx \\
&= \frac{2}{(b-a)^2}\left(f(a)\int_a^b (c-x)^2dx + 2f(c)\int_a^b (b-x)(x-a)dx \right. \\
&\qquad \left. + f(b)\int_a^b (x-c)^2dx\right) \\
&= \frac{2}{(b-a)^2}\left(2f(a)\int_0^{(b-a)/2} t^2dt + 2f(c)\cdot\frac{(b-a)^3}{6} + 2f(b)\int_0^{(b-a)/2} t^2dt\right) \\
&= \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)
\end{aligned}$$

ただし，シンプソン則の積分計算では $\displaystyle\int_a^b (x-c)dx = 0$ を用いている．
これらの近似積分公式を一般的に積分則という．実際にこれらの積分則を使用する際には，積分区間を複数の区間に分割し，個々の区間ごとに積分則を適用する．

3.2 積分則の誤差

ここでは，積分則による近似の誤差を解析しよう．簡単のため，$f(x)$ を平行移動して，中点が $x = 0$ になるようにしたものを $\varphi(x)$ とする．すなわち

$$\varphi(x) = f\left(\frac{a+b}{2} + x\right)$$

である．このとき，$k = \dfrac{b-a}{2}$ とおくと

$$\begin{aligned}
Q - R_{a,b}(f) &= \int_a^b f(x)dx - (b-a)f\left(\frac{a+b}{2}\right) \\
&= \int_{-k}^{k} \varphi(x)dx - 2k\varphi(0) \\
Q - T_{a,b}(f) &= \int_a^b f(x)dx - \frac{b-a}{2}\Big(f(a) + f(b)\Big) \\
&= \int_{-k}^{k} \varphi(x)dx - k\Big(\varphi(k) + \varphi(-k)\Big) \\
Q - S_{a,b}(f) &= \int_a^b f(x)dx - \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right) \\
&= \int_{-k}^{k} \varphi(x)dx - \frac{k}{3}\Big(\varphi(k) + \varphi(-k) + 4\varphi(0)\Big)
\end{aligned}$$

となる．そこで

$$\begin{aligned}
g_0(x) &= \int_{-x}^{x} \varphi(t)dt - 2x\varphi(0) \\
g_1(x) &= \int_{-x}^{x} \varphi(t)dt - x\Big(\varphi(x) + \varphi(-x)\Big) \\
g_2(x) &= \int_{-x}^{x} \varphi(t)dt - \frac{x}{3}\Big(\varphi(x) + \varphi(-x) + 4\varphi(0)\Big)
\end{aligned}$$

と置いて，$|g_0(k)|,\ |g_1(k)|,\ |g_2(k)|$ の大きさを考えることにする．

いま，$f(x) \in C^2[a,b]$ とし，$M_2 = \max\limits_{a\le x\le b} |f''(x)| = \max\limits_{-k\le x\le k} |\varphi''(x)|$ とする．このとき，

$$\begin{aligned}
g_0'(x) &= \varphi(x) + \varphi(-x) - 2\varphi(0), \\
g_0''(x) &= \varphi'(x) - \varphi'(-x) = \int_{-x}^{x} \varphi''(t)dt
\end{aligned}$$

が成り立つので，$0 \le x \le k$ について

$$|g_0''(x)| = \left|\int_{-x}^{x} \varphi''(t)dt\right| \le \int_{-x}^{x} M_2\, dt = 2M_2 x$$

$$|g_0'(x)| = \left|g_0'(0) + \int_0^x g_0''(t)dt\right| \le \int_0^x 2M_2 t\,dt = M_2x^2$$
$$|g_0(x)| = \left|g_0(0) + \int_0^x g_0'(t)dt\right| \le \int_0^x M_2t^2dt = \frac{1}{3}M_2x^3$$

が成り立つ．よって

$$|Q - R_{a,b}(f)| = |g_0(k)| \le \frac{1}{3}M_2k^3 = \frac{(b-a)^3}{24}\max_{a\le x\le b}|f''(x)|$$

となり，中点則の誤差評価ができた．

次に

$$\begin{aligned} g_1'(x) &= \varphi(x) + \varphi(-x) - \Big(\varphi(x) + \varphi(-x)\Big) - x\Big(\varphi'(x) - \varphi'(-x)\Big) \\ &= -x\Big(\varphi'(x) - \varphi'(-x)\Big) = -x\int_{-x}^x \varphi''(t)dt \end{aligned}$$

が成り立つので，$0 \le x \le k$ について

$$|g_1'(x)| = x\left|\int_{-x}^x \varphi''(t)dt\right| \le x\int_{-x}^x M_2\,dt = 2M_2x^2$$
$$|g_1(x)| = \left|g_1(0) + \int_0^x g_1'(t)dt\right| \le \int_0^x 2M_2t^2dt = \frac{2}{3}M_2x^3$$

が成り立つ．よって

$$|Q - T_{a,b}(f)| = |g_1(k)| \le \frac{2}{3}M_2k^3 = \frac{(b-a)^3}{12}\max_{a\le x\le b}|f''(x)|$$

となり，台形則の誤差評価ができた．台形則より中点則の方が精度が良いというのは意外かもしれない．

シンプソン則については，$f(x) \in C^4[a,b]$ とし，$M_4 = \max\limits_{a\le x\le b}|f''''(x)| = \max\limits_{-k\le x\le k}|\varphi''''(x)|$ とすると，

$$\begin{aligned} g_2'(x) &= \varphi(x) + \varphi(-x) - \frac{1}{3}\Big(\varphi(x) + \varphi(-x) + 4\varphi(0)\Big) - \frac{x}{3}\Big(\varphi'(x) - \varphi'(-x)\Big), \\ &= \frac{2}{3}\Big(\varphi(x) + \varphi(-x) - 2\varphi(0)\Big) - \frac{x}{3}\Big(\varphi'(x) - \varphi'(-x)\Big), \\ g_2''(x) &= \frac{2}{3}\Big(\varphi'(x) - \varphi'(-x)\Big) - \frac{1}{3}\Big(\varphi'(x) - \varphi'(-x)\Big) - \frac{x}{3}\Big(\varphi''(x) + \varphi''(-x)\Big), \\ &= \frac{1}{3}\Big(\varphi'(x) - \varphi'(-x)\Big) - \frac{x}{3}\Big(\varphi''(x) + \varphi''(-x)\Big), \\ g_2'''(x) &= \frac{1}{3}\Big(\varphi''(x) + \varphi''(-x)\Big) - \frac{1}{3}\Big(\varphi''(x) + \varphi''(-x)\Big) - \frac{x}{3}\Big(\varphi'''(x) - \varphi'''(-x)\Big), \end{aligned}$$

$$= -\frac{x}{3}\Big(\varphi'''(x) - \varphi'''(-x)\Big) = -\frac{x}{3}\int_{-x}^{x} \varphi''''(t)dt$$

が成り立つので，$0 \le x \le k$ について

$$|g_2'''(x)| = \frac{x}{3}\left|\int_{-x}^{x} \varphi''''(t)dt\right| \le \frac{x}{3}\int_{-x}^{x} M_4\, dt = \frac{2}{3}M_4 x^2$$
$$|g_2''(x)| = \left|g_2''(0) + \int_0^x g_2'''(t)dt\right| \le \int_0^x \frac{2}{3}M_4 t^2 dt = \frac{2}{9}M_4 x^3$$
$$|g_2'(x)| = \left|g_2'(0) + \int_0^x g_2''(t)dt\right| \le \int_0^x \frac{2}{9}M_4 t^3 dt = \frac{1}{18}M_4 x^4$$
$$|g_2(x)| = \left|g_2(0) + \int_0^x g_2'(t)dt\right| \le \int_0^x \frac{1}{18}M_4 t^4 dt = \frac{1}{90}M_4 x^5$$

が成り立つ．よって

$$|Q - S_{a,b}(f)| = |g_2(k)| \le \frac{1}{90}M_4 k^5 = \frac{(b-a)^5}{2880}\max_{a\le x\le b}|f''''(x)|$$

となり，シンプソン則の誤差評価ができた．

3.3 複合積分則

実際に積分の近似値を計算する際には，積分区間を複数の区間に分割し，個々の区間ごとに積分則を適用する．簡単のため，区間は N 等分するものとし，区間幅を h とする．具体的には，

$$h = \frac{b-a}{N}, \qquad x_k = a + kh$$

とする．そのうえで，各区間で積分則を適用した結果は次のようになる．

複合中点則：

$$R_h(f) = h\sum_{k=0}^{N-1} f(x_{k+1/2})$$
$$= h\Big(f(x_{1/2}) + f(x_{3/2}) + f(x_{5/2}) + \cdots + f(x_{N-1/2})\Big)$$

複合台形則：

$$T_h(f) = \frac{h}{2}\sum_{k=0}^{N-1}\Big(f(x_k) + f(x_{k+1})\Big)$$

$$= \frac{h}{2}\Big(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{N-1}) + f(x_N)\Big)$$

複合シンプソン則：

$$\begin{aligned} S_h(f) &= \frac{h}{6}\sum_{k=0}^{N-1}\Big(f(x_k) + 4f(x_{k+1/2}) + f(x_{k+1})\Big) \\ &= \frac{h}{6}\Big(f(x_0) + 4f(x_{1/2}) + 2f(x_1) + 4f(x_{3/2}) + 2f(x_2) + \cdots + 4f(x_{N-1/2}) + f(x_N)\Big) \end{aligned}$$

となる．

複合積分則の誤差は各区間の誤差の合計で評価することができるので

$$|Q - R_h(f)| \le N \cdot \frac{h^3}{24}\max_{a\le x\le b}|f''(x)| = \frac{(b-a)h^2}{24}\max_{a\le x\le b}|f''(x)|$$
$$|Q - T_h(f)| \le N \cdot \frac{h^3}{12}\max_{a\le x\le b}|f''(x)| = \frac{(b-a)h^2}{12}\max_{a\le x\le b}|f''(x)|$$
$$|Q - S_h(f)| \le N \cdot \frac{h^5}{2880}\max_{a\le x\le b}|f''''(x)| = \frac{(b-a)h^4}{2880}\max_{a\le x\le b}|f''''(x)|$$

と誤差評価が得られる．すなわち，刻みを細かくしていった場合の誤差限界は，複合中点則と複合台形則は刻み幅の 2 乗，複合シンプソン則は刻み幅の 4 乗のオーダーで誤差が減少することがわかる．

3.4 複合積分則のプログラム

複合中点則のプログラムは以下のようになる．

```
#include <stdio.h>
#include <math.h>

// 積分したい関数
double f(double x)
{
     return exp(x);
}
```

次ページへ続く…

```
// 原始関数
double F(double x)
{
    return exp(x);
}

// [a,b] における 2 階微分の絶対値の最大値
double M2(double a, double b)
{
    return exp(b);
}

int main()
{
    printf("***** 複合中点則 *****\n");

    double a = 1.0, b = 2.0; // 積分区間

    for(int N=5; N<=160; N*=2){  // 分割数
        double h = (b-a)/N;     // 区間幅

        double S = 0;
        for(int k=0; k<=N-1; k++){
            double x = a + (k+0.5)*h;
            S += f(x);
        }
        S *= h;

        double Q = F(b) - F(a); // 積分の理論値
        double E = (b-a)*h*h/24 * M2(a,b); // 理論的な誤差限界
        printf("N=%d、誤差は%.15f、理論誤差限界は%.15f\n", N, S-Q, E);
    }
    return 0;
}
```

積分する関数を変えるときは，最初の 3 つのユーザー定義関数を変更する．
プログラムが間違っていなければ，刻みが倍になるごとに誤差が 4 分の一になる．

複合台形則のプログラムは以下のようになる.

```
#include <stdio.h>
#include <math.h>

// 積分したい関数
double f(double x)
{
     return exp(x);
}

// 原始関数
double F(double x)
{
    return exp(x);
}

// [a,b] における 2 階微分の絶対値の最大値
double M2(double a, double b)
{
    return exp(b);
}

int main()
{
    printf("***** 複合台形則 *****\n");

    double a = 1.0, b = 2.0; // 積分区間

    for(int N=5; N<=160; N*=2){  // 分割数
        double h = (b-a)/N;     // 区間幅
        double S = 0;
        for(int k=0; k<=N; k++){
            double x = a + k*h;
            if(k==0 || k==N){
                S += f(x);
            } else {
```

次ページへ続く…

```
                S += 2*f(x);
            }
        }
        S *= h/2;

        double Q = F(b) - F(a); // 積分の理論値
        double E = (b-a)*h*h/12 * M2(a,b); // 理論的な誤差限界
        printf("N=%d、誤差は%.15f、理論誤差限界は%.15f\n", N, S-Q, E);
    }
    return 0;
}
```

N が同じ場合，中点則と比較して誤差がおよそ倍になっていればよい．

複合シンプソン則のプログラムは以下のようになる．

```
#include <stdio.h>
#include <math.h>

// 積分したい関数
double f(double x)
{
     return exp(x);
}

// 原始関数
double F(double x)
{
    return exp(x);
}

// [a,b] における 4 階微分の絶対値の最大値
double M4(double a, double b)
{
    return exp(b);
}
```

次ページへ続く…

```
int main()
{
    printf("***** 複合シンプソン則 *****\n");

    double a = 1.0, b = 2.0; // 積分区間

    for(int N=5; N<=160; N*=2){  // 分割数
        double h = (b-a)/N;     // 区間幅

        double S = 0;
        for(int k=0; k<=2*N; k++){
            double x = a + k*0.5*h;
            if(k==0 || k==2*N){
                S += f(x);
            } else if(k%2==0){
                S += 2*f(x);
            } else {
                S += 4*f(x);
            }
        }
        S *= h/6;

        double Q = F(b) - F(a); // 積分の理論値
        double E = (b-a)*h*h*h*h/2880 * M4(a,b); // 理論的な誤差限界
        printf("N=%d、誤差は%.15f、理論誤差限界は%.15f\n", N, S-Q, E);
    }
    return 0;
}
```

プログラムが間違っていなければ，刻みが倍になるごとに誤差が 16 分の一になる．

3.5 周期関数に対する複合台形則

滑らかな周期関数に対しては，複合台形則の精度が非常に良くなることが知られている．この事実は複素関数論の理論により示すことができるのだが，ここでは概略のみ説明する．

$f(x)$ を，周期 2π の滑らかな周期関数とする．ここで，滑らかというのは「解析的」という意味で用いている．「解析的」という概念については，ここでは詳細は述べないので，詳

しく知りたい人は複素関数論を勉強してもらいたい．ただ，$[0, 2\pi]$ の全ての点について定義され，絶対値などを含まない普通の関数で定義されるような関数は，大抵は解析的な周期関数となる．例えば，

$$f_1(x) = \sin x, \qquad f_2(x) = \cos x, \qquad f_3(x) = e^{\cos x}, \qquad f_4(x) = \log \frac{2 - \sin x}{1 + \cos^2 x}$$

などは解析的な周期関数である．一方，

$$f_5(x) = \tan x, \qquad f_6(x) = |\sin x|, \qquad f_7(x) = \frac{1}{1 - \cos x}, \qquad \begin{cases} f_8(x) = x \ \ (-\pi \leq x < \pi), \\ f_8(x + 2\pi) = f_8(x) \end{cases}$$

などは解析的ではない．

さて，一定の条件を満たす周期関数は

$$\begin{aligned} f(x) = \frac{a_0}{2} &+ a_1 \cos x + a_2 \cos 2x + a_3 \cos 3x + a_4 \cos 4x + \cdots \\ &+ b_1 \sin x + b_2 \sin 2x + b_3 \sin 3x + b_4 \sin 4x + \cdots \end{aligned}$$

とフーリエ級数展開できることが知られている．特に，$f(x)$ が解析的な周期関数の場合には，ある正の定数 C_1, C_2 が存在して，係数 $a_n,\ b_n$ について

$$|a_n| + |b_n| \leq C_1 e^{-C_2 n}$$

が成り立つことが知られている．

それでは，上のようにフーリエ級数展開されている関数 $f(x)$ の一周期積分を考えてみることにしよう．三角関数の一周期積分は 0 なので，積分すると a_0 だけが残り

$$Q = \int_0^{2\pi} f(x) dx = \pi a_0$$

となる．一方で，複合台形則の区間分割数を N とすると，整数 m が N の倍数ではないときには

$$\begin{aligned} \sum_{k=0}^{N-1} \cos m x_k + i \sum_{k=0}^{N-1} \sin m x_k = \sum_{k=0}^{N-1} e^{i m x_k} = \sum_{k=0}^{N-1} e^{2\pi m k i / N} &= \frac{1 - (e^{2\pi m i / N})^N}{1 - e^{2\pi m i / N}} \\ = \frac{1 - e^{2\pi m i}}{1 - e^{2\pi m i / N}} &= 0 \end{aligned}$$

より

$$\sum_{k=0}^{N-1} \cos m x_k = \sum_{k=0}^{N-1} \sin m x_k = 0$$

が成り立ち，m が N の倍数のときには

$$\sum_{k=0}^{N-1}\cos mx_k + i\sum_{k=0}^{N-1}\sin mx_k = \sum_{k=0}^{N-1}e^{imx_k} = \sum_{k=0}^{N-1}e^{2\pi mki/N} = \sum_{k=0}^{N-1}1 = N$$

より

$$\sum_{k=0}^{N-1}\cos mx_k = N, \qquad \sum_{k=0}^{N-1}\sin mx_k = 0$$

が成り立つことから，複合台形則による値は

$$\begin{aligned} T_h(f) &= \frac{2\pi}{N}\left(\frac{f(x_0)}{2} + \sum_{k=1}^{N-1}f(x_k) + \frac{f(x_N)}{2}\right) \\ &= \frac{2\pi}{N}\sum_{k=0}^{N-1}f(x_k) = \pi a_0 + 2\pi(a_N + a_{2N} + a_{3N} + \cdots) \end{aligned}$$

となる．ここで，$f(x)$ が解析的であって，フーリエ級数の係数について

$$|a_n| + |b_n| \le C_1 e^{-C_2 n}$$

なる評価が成り立つとすれば,

$$\begin{aligned} |Q - T_h(f)| &= 2\pi|a_N + a_{2N} + a_{3N} + \cdots| \\ &\le 2\pi C_1(e^{-C_2 N} + e^{-2C_2 N} + e^{-3C_2 N} + \cdots) \\ &= \frac{2\pi C_1 e^{-C_2 N}}{1 - e^{-C_2 N}} \end{aligned}$$

となり，区間分割 N に対して指数的に誤差が減少することがわかる．

実際に,

$$\int_0^{2\pi}\frac{dx}{4 + 2\sin x + \cos x} = \frac{2\pi}{\sqrt{11}}$$

などの例を複合台形則で計算してみると，誤差の指数的減少が確かめられる．

通常の閉区間上の積分も，両端をつないで周期関数とみなしたときの滑らかさが上がるほど，積分則の精度が向上することが知られている．例えば，関数

$$f(x) = \frac{1}{x} + \frac{(13 - 3x)x}{8}$$

は $f(1) = f(2) = \dfrac{9}{4},\ f'(1) = f'(2) = -\dfrac{1}{8}$ を満たすので，区間 $[1, 2]$ の両端をつないで周期関数とみなしたものは，繋いだ点で値と一階微分が連続になる．この関数の積分

$$Q = \int_1^2 f(x)dx = \log 2 + \frac{25}{16}$$

を複合台形公式で計算すると，誤差は h^4 に比例するスピードで減少し，理論的に求めた複合台形則の誤差限界よりも良い精度が実現している．

滑らかな関数 $f(x)$ に対する無限積分

$$Q = \int_{-\infty}^{\infty} f(x)dx$$

についても，複合台形則は良い精度を与えることが知られている．以下ではそれを説明しよう．

適当な $L > 0$ を取って

$$\tilde{f}(x) = \sum_{k=-\infty}^{\infty} f(x + Lk)$$

とすれば $\tilde{f}(x)$ は周期 L の滑らかな周期関数となるので，複合台形則

$$h = \frac{L}{N}, \qquad h\sum_{k=0}^{N-1} \tilde{f}(kh)$$

は

$$\int_0^L \tilde{f}(x)dx$$

を高精度に近似する．ここで，

$$h\sum_{k=0}^{N-1} \tilde{f}(kh) = h\sum_{k=-\infty}^{\infty} f(kh), \qquad \int_0^L \tilde{f}(x)dx = \int_{-\infty}^{\infty} f(x)dx$$

が成り立つので，無限積分

$$Q = \int_{-\infty}^{\infty} f(x)dx$$

は複合台形則

$$h\sum_{k=-\infty}^{\infty} f(kh),$$

を用いて高精度に計算することができる．

無限積分の場合，複合台形則は無限和になるが，$f(x)$ が e^{-x^2} のように $|x| \to \infty$ で急激に減少するような関数の場合は，無限和を有限和で打ち切ることで近似値を精度良く計算することができる．その際，打ち切りによる誤差が十分小さくなるように配慮する必要がある．

3.6　補足

本章では，複合積分則により，等間隔の分点における関数値を用いて積分を計算したが，分点を等間隔ではない特別な位置に取ることで，さらに精度を上げることができる．そのような積分公式で有名なものにガウス・ルジャンドル積分公式があり，実際にもよく用いられているが，時間の都合で説明は省略した．

既に見たように，積分する関数の両端をつないで周期関数とみなしたとき，つないだ点における滑らかさが上がると複合台形則の精度が上昇する．これはフーリエ級数論やオイラー・マクローリンの総和公式などを用いて証明することができるので，興味のある人は調べてみれば面白いと思う．解析的な周期関数の場合の複合台形則の誤差の指数的減少については，既に書いたように，複素関数論の知識を用いて証明できる．これも興味のある人は勉強してみるとよいと思う．

置換積分を利用して積分の積分変数を変換すると，被積分関数の形を変えることができる．その事実を利用し，置換積分により被積分関数を $|x| \to \infty$ で急激に減少する関数に変換して，有限項で打ち切った複合台形則で積分を計算すると，精度良く計算できる．このような考えに基づく数値積分は，変数変換型数値積分と呼ばれ，有名なものに二重指数型変数変換積分公式（DE 公式）がある．DE 公式の誤差は，多くの積分において，分点数に対して指数的減少に近いスピードで減少する．

3.7　課題

課題１　原始関数のわかっている関数を自由に選び，複合中点則，複合台形則，複合シンプソン則で計算を行い，理論的な誤差限界と比較しなさい．

課題２　$\displaystyle\int_1^2 \frac{e^x}{x} dx$ の値を小数点以下 8 桁まで正確に計算しなさい．小数点以下 8 桁まで正確な値であるということは，積分則の誤差評価を用いて確かめなさい．積分則の誤差評価を行う上で，2 階微分や 4 階微分の最大値が必要だが，これは正確な最大値でなくとも，最大値を上から押さえる値であれば誤差評価できることに注意すること．

課題３　$\displaystyle\int_0^{2\pi} \frac{dx}{4 + 2\sin x + \cos x} = \frac{2\pi}{\sqrt{11}}$ を複合台形則で計算し，精度を調べなさい．

4　常微分方程式

一変数関数とその導関数からなる方程式を**常微分方程式**という．本章では，常微分方程式の初期値問題の数値解法について説明する．

4.1　色々な常微分方程式の初期値問題

微分方程式とは，関数そのものや，その微分などの間に成り立つ関係式であるが，それだけでは解が一つに定まらない．そこで，ある地点での関数値や微分の値を与えて解を考えることが多い．このような条件を初期条件という．解が決まるには，通常は，微分方程式に出てくる微分の階数分だけ初期条件が必要である．

微分方程式の解を $y(t)$ としたとき，y を従属変数，t を独立変数という．これらの用語は，t に応じて y が決まることから名づけられている．以下，微分の記号 $'$ や $''$ や $^{(n)}$ などは独立変数による微分を表すものとする．また，本章においては，独立変数としては常に t のみを用いるものとする．

それでは，微分方程式にどのようなものがあるか見ていくことにしよう．まず，世の中には数学的に（紙と鉛筆で）解ける微分方程式がある．例えば

$$\begin{cases} y(0) = 1, \\ y' = y \end{cases}$$

は数学的に解くことができて，解は

$$y = e^t$$

となる．ここで $y(0) = 1$ が初期条件である．初期条件という用語は，時間変数を独立変数に取ることが多いという理由による．

微分方程式の右辺には独立変数が入ってもよい．例えば

$$\begin{cases} y(0) = 1, \\ y' = ty \end{cases}$$

の解は

$$y = e^{t^2/2}$$

となる．

初めにも述べたが，微分の階数が高いと，それに応じて初期条件が必要である．例えば

$$\begin{cases} y(0) = 1,\ y'(0) = 0, \\ y'' + 2y' + 2y = 0 \end{cases}$$

は数学的に解けて，解は

$$y(x) = (\cos t + \sin t)e^{-t}$$

となる．以上のような微分方程式は数学的に解が求まるが，世の中には式変形では解けない方程式も多い．

以下は，ロトカ・ボルテラ方程式という，捕食者と被食者（例えばライオンとシマウマ）の増減関係をモデル化した方程式である．

$$\begin{cases} x' = (a - by)x, \\ y' = (-c + dx)y \end{cases}$$

ここで，x と y は時間 t の関数（つまり，t が独立変数，x と y が従属変数）であり，x が被食者の，y が捕食者の個体数を表す．$a,\ b,\ c,\ d$ は正の定数である．初期条件としては，適当な時刻（例えば $t = 0$）における x と y の値を与える．

以下は，ローレンツ方程式という，大気変動モデルを元にした方程式である．

$$\begin{cases} x' = -px + py, \\ y' = -xz + rx - y, \\ z' = xy - bz \end{cases}$$

ここで，t が独立変数，$x,\ y,\ z$ が従属変数であり，$p,\ r,\ b$ は定数である．ローレンツの原論文では，$p = 10,\ r = 28,\ b = 8/3$ と取られており，現在でもこの値を用いることが多い．初期条件としては，適当な時刻における $x,\ y,\ z$ の値を与える．ローレンツ方程式は，カオスの発見につながった方程式として有名である．

ロトカ・ボルテラ方程式やローレンツ方程式の微分の階数は一階であるが，従属変数の数は複数である．しかし，これらの方程式は，変数をベクトルとして考えれば，従属変数が一つの場合と同様，

$$\begin{cases} y' = f(t, y), \\ y(t_0) = y_0 \end{cases}$$

の形で書くことができる．例えば，ロトカ・ボルテラ方程式では

$$\mathbf{y} = \begin{pmatrix} x \\ y \end{pmatrix}, \qquad f(t, \mathbf{y}) = \begin{pmatrix} (a - by)x \\ (-c + dx)y \end{pmatrix}$$

と置けば

$$\mathbf{y}' = f(t, \mathbf{y})$$

の形になる．

以下は，ファン・デル・ポール方程式という，非線形振動を表す微分方程式である．

$$x'' - \mu(1 - x^2)x' + x = 0$$

ここで，x は時間 t の関数，μ は定数である．初期条件としては，ある時刻における x および x' の値を与える．この微分方程式は，電気回路の振動現象をモデル化するために用いられた．

一般に，t を独立変数とする n 階の微分方程式

$$y^{(n)} = f(t, y, y', y'', \cdots, y^{(n-1)})$$

に対し，1 階の微分方程式

$$\mathbf{y}' = F(x, \mathbf{y}), \qquad \mathbf{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix}, \qquad F(t, \mathbf{y}) = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ f(t, y_0, y_1, y_2, \cdots, y_{n-1}) \end{pmatrix}$$

の解を考えると，y_0 は前者の n 階の微分方程式の解になっている．実際，$y = y_0$ と置くと，後者のベクトル変数に対する微分方程式の第 1 行目から第 $n-1$ 行目から $y'_{k-1} = y_k$ $(k = 1, 2, \cdots, n-1)$ が成り立ち，帰納的に $y_k = y^{(k)}$ $(k = 1, 2, \cdots, n-1)$ が成り立つので，これを第 n 行目に代入すると前者の n 階の微分方程式が得られる．
つまり，$y^{(n)} = f(t, y, y', y'', \cdots, y^{(n-1)})$ という形の微分方程式（このような形の微分方程式を**正規形の微分方程式**という）は，ベクトル変数についての 1 階の微分方程式に変形できるということがわかる．

例えば，ファン・デル・ポール方程式は

$$\frac{d}{dt}\begin{pmatrix} x \\ x' \end{pmatrix} = \begin{pmatrix} x' \\ \mu(1 - x^2)x' - x \end{pmatrix}$$

と書けるので，$x_1 = x,\ x_2 = x'$ と置き換えると

$$\mathbf{x}' = F(t, \mathbf{x}), \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \qquad F(t, \mathbf{x}) = \begin{pmatrix} x_2 \\ \mu(1 - x_1^2)x_2 - x_1 \end{pmatrix}$$

と書ける．

以下は，n 個の天体の運動を表した方程式である．

$$\mathbf{x}_i'' = -\sum_{\substack{j=1 \\ j \neq i}}^{n} G\frac{m_j}{|\mathbf{x}_i - \mathbf{x}_j|^3}(\mathbf{x}_i - \mathbf{x}_j), \qquad i = 1, 2, \cdots, n$$

ここで，i 番目の天体の質量を m_i，位置を $\mathbf{x}_i$ としている．G は万有引力定数である．この方程式も，微分の階数を減らすことで

$$\mathbf{x}' = F(t, \mathbf{x})$$

の形に書くことができる．この方程式は，万有引力の法則とニュートンの運動法則から導くことができる．

例えば，平面上を動く 2 つの天体の運動は，一つ目の天体の位置を $\mathbf{x}_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$，二つ目の天体の位置を $\mathbf{x}_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$，さらに $u_i = x_i'$, $v_i = y_i'$ $(i = 1, 2)$ と置くと

$$\mathbf{x}' = F(t, \mathbf{x}), \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ u_1 \\ v_1 \\ u_2 \\ v_2 \end{pmatrix}, \qquad F(t, \mathbf{x}) = \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ -G\dfrac{m_2(x_1 - x_2)}{\{(x_1 - x_2)^2 + (y_1 - y_2)^2\}^{3/2}} \\ -G\dfrac{m_2(y_1 - y_2)}{\{(x_1 - x_2)^2 + (y_1 - y_2)^2\}^{3/2}} \\ -G\dfrac{m_1(x_2 - x_1)}{\{(x_1 - x_2)^2 + (y_1 - y_2)^2\}^{3/2}} \\ -G\dfrac{m_1(y_2 - y_1)}{\{(x_1 - x_2)^2 + (y_1 - y_2)^2\}^{3/2}} \end{pmatrix}$$

と書ける．

以上で見たように，多くの微分方程式の初期値問題は

$$\begin{cases} y(t_0) = y_0, \\ y' = f(t, y) \end{cases}$$

の形に書くことができる．そこで，以降はこの形の微分方程式の数値解法について考えていくことにする．

4.2 ランダウの記号

今後よく使うので，ランダウの記号について説明しておこう．$|x|$ が十分小さいときに，ある定数 $M > 0$ が存在して $|f(x)| \leq M|g(x)|$ が成り立つことを

$$f(x) = O(g(x))$$

と書くことにする．この $O(\cdot)$ をランダウの記号と言う．例えば

$$\begin{aligned}
\cos x &= O(1) \\
\sin x &= O(x) \\
e^x - 1 - x &= O(x^2) \\
x^m &= O(x^n), \qquad m \geq n \geq 0 \\
\frac{1}{1+x} &= 1 - x + x^2 + O(x^3)
\end{aligned}$$

などが成り立つ．簡単に言うと，$O(x^s)$ は，x の s 乗以上のオーダーで減少する項を表している．

4.3 オイラー法

微分方程式

$$\begin{cases} y(t_0) = y_0, \\ y' = f(t, y) \end{cases}$$

の数値解法を考えるため，独立変数 x を分割幅 h で等間隔に離散化したものとして

$$t_0 < t_1 < t_2 < \cdots < t_N, \qquad t_{n+1} - t_n = h \ \ (n = 0, 1, \cdots, N-1)$$

を考え，$y(t_n)$ の近似を y_n で表すものとする．

いま，解 $y(t)$ が十分に滑らかだとすると，テイラー展開により

$$y(t_{n+1}) = y(t_n + h) = y(t_n) + hy'(t_n) + O(h^2)$$

が成り立つ．

ここで $y' = f(t, y)$ であるので，上式は

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + O(h^2)$$

と変形できる．そこで，近似スキームとして

$$y_{n+1} = y_n + hf(t_n, y_n)$$

が考えられる．具体的には，

$$\begin{cases} \ \ y_0 = y(t_0), \\ y_{n+1} = y_n + hf(t_n, y_n) \end{cases}$$

により，$y(t_1), y(t_2), \cdots, y(t_N)$ の近似である $y_1, y_2, \cdots, y_N$ を求める．これを**オイラー法**という．一般に，$y'(t) = f(t, y)$ の値は t に応じて徐々に変化していくが，$t_n \leq t \leq t_{n+1}$ の間は $y' = f(t_n, y_n)$ のまま変化しないと仮定し，接線による近似を行ったのがオイラー法であると考えることができる．

オイラー法の精度を調べるため，y_{n+1} と $y(t_{n+1})$ がどの程度近くなるか調べてみることにしよう．

$$\begin{cases} y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + O(h^2), \\ \quad y_{n+1} = y_n + hf(t_n, y_n) \end{cases}$$

より，辺々引くと

$$y(t_{n+1}) - y_{n+1} = y(t_n) - y_n + h\Big(f(t_n, y(t_n)) - f(t_n, y_n)\Big) + O(h^2)$$

となる．すなわち，仮に $y_n = y(t_n)$ となったとすると，y_{n+1} と $y(t_{n+1})$ は h の 1 次の項まで一致する．そのため，オイラー法は 1 次精度であると言われる．

実際には，y_n と $y(t_n)$ の間にも誤差があり，さらに誤差は累積していくので，$n = 1$ から $n = N$ まで計算して，トータルでどの程度の誤差が発生するかという点については，もう少し詳しく分析する必要がある．これは後ほどまた検証する．

4.4 修正オイラー法

オイラー法の考え方は，$y'(t_n) = f(t_n, y(t_n))$ を用いて，$y(t_{n+1})$ をテイラー展開の $O(h)$ の項まで近似するというものであった．

そこで，さらに精度を上げるため，$y'(t_n) = f(t_n, y(t_n))$ だけではなく $y'(t_{n+1}) = f(t_{n+1}, y(t_{n+1}))$ も用いることで，今度は $y(t_{n+1})$ を $O(h^2)$ の項まで近似してみよう．

そのためには，仮に $y_n = y(t_n)$ となったとしたときに

$$\begin{cases} y(t_{n+1}) = y(t_n) + hy'(t_n) + \dfrac{h^2}{2}y''(t_n) + O(h^3), \\ \quad y_{n+1} = y_n + h\Big(\alpha f(t_n, y(t_n)) + \beta f(t_{n+1}, y(t_{n+1}))\Big) \end{cases}$$

が $O(h^2)$ の項まで一致するように定数 α, β の値を決めたい．ここで，

$$\begin{aligned} y_n + h\Big(\alpha f(t_n, y(t_n)) + \beta f(t_{n+1}, y(t_{n+1}))\Big) &= y_n + h\Big(\alpha y'(t_n) + \beta y'(t_{n+1})\Big) \\ &= y_n + h\Big(\alpha y'(t_n) + \beta y'(t_n + h)\Big) \\ &= y_n + h\Big(\alpha y'(t_n) + \beta y'(t_n) + \beta h y''(t_n) + O(h^2)\Big) \end{aligned}$$

より，$\alpha = \beta = \dfrac{1}{2}$ と取れば目標が達成されることがわかる．よって

$$y_{n+1} = y_n + \frac{h}{2}\Big(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\Big)$$

なる数値計算スキームが考えられるが，この形では右辺に y_{n+1} が入っているので使いにくい．そこで，右辺の y_{n+1} をオイラー法を用いて $y_n + hf(t_n, y_n)$ で近似し，

$$y_{n+1} = y_n + \frac{h}{2}\Big(f(t_n, y_n) + f\big(t_{n+1}, y_n + hf(t_n, y_n)\big)\Big)$$

というスキームを考える．このままでは括弧が多くて見にくいので書き直し，初期条件も加えると

$$\begin{cases} y_0 = y(t_0), \\ k_1 = hf(t_n, y_n), \\ k_2 = hf(t_n + h, y_n + k_1), \\ y_{n+1} = y_n + \dfrac{1}{2}(k_1 + k_2) \end{cases}$$

という形になる（ここで，k_1, k_2 は n ごとに計算する）．この数値計算スキームを**修正オイラー法**（もしくはホイン法）という．

修正オイラー法を導出する際には，右辺に出てきた $y(t_{n+1})$ を y_{n+1} で置き換え，さらにそれをオイラー法で近似したので，それでもちゃんと $O(h^2)$ の精度が保たれているか調べる必要がある．それを調べるため，ここで二変数関数についての連鎖公式について復習しておこう．

いま，$f(x, y)$ を C^2 級関数とし，x および y を変数 t の関数とすると

$$\frac{df(x, y)}{dt} = \frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt}$$

が成り立つ．特に $x = t$ のときには

$$\frac{df(t, y)}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt}$$

となる．

それでは，修正オイラー法の精度を調べよう．仮に $y_n = y(t_n)$ となったとし，$f(t, y)$ が十分に滑らかだとすると，2 変数関数のテイラー展開を用いて以下のように変形できる．

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{2}\Big(f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n))\Big) \\ &= y_n + \frac{h}{2}\Big(f(t_n, y_n) + f(t_n + h, y_n + hy'(t_n))\Big) \end{aligned}$$

$$= y_n + \frac{h}{2}\Big(f(t_n, y_n) + f(t_n, y_n) + h\frac{\partial}{\partial t}f(t_n, y_n) + hy'(t_n)\frac{\partial}{\partial y}f(t_n, y_n) + O(h^2)\Big)$$
$$= y_n + hf(t_n, y_n) + \frac{h^2}{2}\Big(\frac{\partial}{\partial t}f(t_n, y_n) + y'(t_n)\frac{\partial}{\partial y}f(t_n, y_n)\Big) + O(h^3)$$

ここで，上で確認した連鎖公式を用いると

$$\begin{aligned} y_{n+1} &= y_n + hf(t_n, y_n) + \frac{h^2}{2}\cdot\frac{d}{dt}f(t_n, y_n) + O(h^3) \\ &= y_n + hf(t_n, y_n) + \frac{h^2}{2}\cdot\frac{d}{dt}y'(t_n) + O(h^3) \\ &= y_n + hy'(t_n) + \frac{h^2}{2}y''(t_n) + O(h^3) \end{aligned}$$

が成り立つ．一方，

$$y(t_{n+1}) = y(t_n + h) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + O(h^3)$$

なので，仮に y_n と $y(t_n)$ が一致したときには，y_{n+1} と $y(t_{n+1})$ は h の 2 次の項まで一致する．以上により，修正オイラー法が 2 次精度であることがわかった．

4.5　4 次のルンゲ・クッタ法

実際の応用上は，オイラー法や修正オイラー法は精度の点からあまり使われない．よく使われるのは以下の **4 次のルンゲ・クッタ法**である．

$$\begin{cases} y_0 = y(t_0), \\ k_1 = hf(t_n, y_n), \\ k_2 = hf(t_n + \dfrac{h}{2}, y_n + \dfrac{k_1}{2}), \\ k_3 = hf(t_n + \dfrac{h}{2}, y_n + \dfrac{k_2}{2}), \\ k_4 = hf(t_n + h, y_n + k_3), \\ y_{n+1} = y_n + \dfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{cases}$$

ここで，k_1, k_2, k_3, k_4 は n ごとに計算する．4 次のルンゲ・クッタ法はその名称の通り 4 次精度であることが知られている．4 次精度であることは，オイラー法や修正オイラー法と同様に示すことができるが，かなりの式変形を必要とするので容易ではない．ただ，腕力のある人は挑戦してみてもいいと思う．

数値計算スキームとしては 5 次以上のスキームも考えられるのだが，精度が 5 次以上になると手順が途端に複雑になるので，通常はあまり用いられない．

4.6 リプシッツ条件

次節以降の解析ではリプシッツ条件という概念を用いるので，ここで説明しておく．

関数 $f(x)$ が領域 D でリプシッツ条件を満たすとは，ある定数 $L > 0$ が存在して，任意の $x_1,\ x_2 \in D$ について

$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2|$$

が成り立つことを言う．$f(x)$ や x がベクトル値を取る場合には，絶対値のかわりに適当なノルムを用いる．$f(x)$ が D でリプシッツ条件を満たすとき，$f(x)$ は D でリプシッツ連続であるという．簡単に言うと，リプシッツ条件というのは，D 内の任意の 2 点間の $f(x)$ の平均変化率の絶対値が L 以下になるということである．

例えば，$f(x) = x^2$ は $D = [-1, 1]$ でリプシッツ条件を満たす．実際，$L = 2$ と取れば

$$|f(x_1) - f(x_2)| = |x_1^2 - x_2^2| = |x_1 + x_2|\ |x_1 - x_2| \leq L|x_1 - x_2|$$

が成り立つ．一般に，$f(x)$ が微分可能で，$|f'(x)|$ が D で有界，つまり $|f'(x)| \leq M$ となるような M が存在するならばリプシッツ条件を満たす．実際

$$|f(x_1) - f(x_2)| = \left|\int_{x_1}^{x_2} f'(x)dx\right| \leq \left|\int_{x_1}^{x_2} M dx\right| = M|x_1 - x_2|$$

が成り立つ．もしくは，平均値の定理より，x_1 と x_2 の間の数 c が存在して

$$|f(x_1) - f(x_2)| = |f'(c)(x_1 - x_2)| \leq M|x_1 - x_2|$$

が成り立つ．

リプシッツ連続という概念は微分可能と連続の中間の概念と考えることができる．つまり，微分不可能だがリプシッツ条件を満たす関数や，連続だがリプシッツ条件を満たさない関数が存在する．例えば，$f(x) = |x|$ は $(-1, 1)$ で微分可能ではないが，リプシッツ条件は満たす．実際，三角不等式から

$$|x_1| \leq |x_1 - x_2| + |x_2|, \qquad |x_2| \leq |x_2 - x_1| + |x_1|$$

が成り立つので，

$$|f(x_1) - f(x_2)| = ||x_1| - |x_2|| = \max\Big(|x_1| - |x_2|,\ |x_2| - |x_1|\Big) \leq |x_1 - x_2|$$

が成り立つ．一方，$f(x) = \sqrt{|x|}$ は $[-1, 1]$ で連続だがリプシッツ条件は満たさない．実際，どんなに大きく $L > 0$ を取っても，$0 < x_1 < \min(1/L^2,\ 1),\ x_2 = 0$ と取れば

$$|f(x_1) - f(x_2)| = \sqrt{x_1} = \frac{x_1}{\sqrt{x_1}} > \frac{x_1}{\sqrt{1/L^2}} = Lx_1 = L|x_1 - x_2|$$

となるので，リプシッツ条件を満たさないことがわかる．

4.7　オイラー法の大域誤差

前項までで，オイラー法と修正オイラー法の誤差について考察したが，その誤差は，$y(x_n)$ と y_n が一致したときに次のステップで生じる誤差であった．実際には，誤差は累積するものであるし，また，ある一定区間を分割して数値スキームを適用するケースを考えると，分割の数が増えるに従い，各ステップの誤差は小さくなっても，ステップの数は増えるので，総合的にみて誤差が小さくなるかどうかはすぐにはわからない．そこで，本節ではオイラー法について，総合的に誤差がどうなるかを考えてみることにする．なお，既にみたような 1 ステップで生じる誤差を**局所誤差**，トータルとして生じる誤差を**大域誤差**という．

微分方程式

$$\begin{cases} y(a) = y_0, \\ y' = f(t, y) \end{cases}$$

を区間 $[a, b]$ 上で解く．そのため，分割数を N とし，

$$t_n = a + nh, \qquad h = \frac{b-a}{N}$$

と分点を取って

$$\begin{cases} \quad y_0 = y(t_0), \\ y_{n+1} = y_n + hf(t_n, y_n) \end{cases}$$

により近似解を計算する．

誤差解析を行う上で，解 $y(t)$ および $f(t, y)$ について，ある程度の仮定を置く必要がある．まず，解 $y(t)$ は $[a, b]$ で C^2 級であるものとし，

$$M = \max_{a \le t \le b} \|y''(t)\|$$

とする．ここで $\|\cdot\|$ は任意のノルムである．$f(t, y)$ や y がベクトルではなくスカラー関数である場合には，単なる絶対値を考えればよい．

次に，全ての $a \le t \le b$ について $y(t) \in D$ となるような適当な領域 D を取り，$f(t, y)$ は $a \le t \le b,\ y \in D$ で，y に関するリプシッツ条件を満たすものとする．すなわち，定数 $L > 0$ が存在して，すべての $a \le t \le b$ および $z_1, z_2 \in D$ について

$$\|f(t, z_1) - f(t, z_2)\| \le L\|z_1 - z_2\|$$

が成り立つものとする．

さて，解 $y(t)$ は $[a, b]$ で C^2 級であるから

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(\xi) = y(t_n) + hf(t_n, y(t_n)) + \frac{h^2}{2}y''(\xi)$$

が成り立つ．ここで h^2 の項はラグランジュの剰余項であり，ξ は t_n と t_{n+1} の間の数である．一方で，オイラー法のスキームは

$$y_{n+1} = y_n + hf(t_n, y_n)$$

なので，辺々引くと

$$y_{n+1} - y(t_{n+1}) = y_n - y(t_n) + h\Big(f(t_n, y_n) - f(t_n, y(t_n))\Big) - \frac{h^2}{2}y''(\xi)$$

となる．これより

$$\|y_{n+1} - y(t_{n+1})\| \le \|y_n - y(t_n)\| + h\Big\|f(t_n, y_n) - f(t_n, y(t_n))\Big\| + \frac{Mh^2}{2}$$

が成り立つ．ここで，$y_n \in D$ を仮定すると，リプシッツ条件から

$$\begin{aligned}\|y_{n+1} - y(t_{n+1})\| &\le \|y_n - y(t_n)\| + Lh\|y_n - y(t_n)\| + \frac{Mh^2}{2} \\ &= (1 + Lh)\|y_n - y(t_n)\| + \frac{Mh^2}{2}\end{aligned}$$

が成り立つ．

一般的に，数列 $\{e_n\}$ と実数 A, B（ただし $A > 1$ とする）が

$$e_{n+1} \le Ae_n + B \quad (n \ge 0)$$

を満たすとき，

$$e_n \le A^n e_0 + \frac{A^n - 1}{A - 1}B$$

が成り立つ．証明は簡単で，$n = 0$ のときには明らかに成り立ち，n での成立を仮定すると

$$e_{n+1} \le Ae_n + B \le A\left(A^n e_0 + \frac{A^n - 1}{A - 1}B\right) + B = A^{n+1}e_0 + \frac{A^{n+1} - 1}{A - 1}B$$

より，$n + 1$ での成立も言えることから帰納的に示される．

すべての n について $y_n \in D$ を仮定するとき，上の不等式を用いると

$$\begin{aligned}\|y_n - y(t_n)\| &\le (1 + Lh)^n\|y_0 - y(t_0)\| + \frac{(1 + Lh)^n - 1}{Lh} \cdot \frac{Mh^2}{2} \\ &= \frac{((1 + Lh)^n - 1)Mh}{2L}\end{aligned}$$

が成り立つ．さらに $1 + Lh \le e^{Lh}$ を用いると

$$\|y_n - y(t_n)\| \le \frac{(e^{nLh} - 1)Mh}{2L} \le \frac{(e^{LNh} - 1)Mh}{2L} = \frac{(e^{L(b-a)} - 1)Mh}{2L}$$

が成り立つことがわかる．ここで，a, b, L, M は全て定数なので，オイラー法の大域誤差は $O(h)$ となることがわかる．

上の解析では $y_n \in D$ を仮定していたが，最初に D を解 $y(t)$ の動く範囲より大きく取っておけば，h を小さく取ることで y_n がすべて D に入るようにできるので，特に問題は生じない（厳密には論証が必要だが）．

4.8 オイラー法のプログラム

まずは，答えの分かっている問題を解いて精度を確認してみよう．N が 10 倍になるごとに誤差が 10 分の一になっていればよい．

```
#include <stdio.h>
#include <math.h>
#include "Linear.h"

// y'=y, y(0)=1 を解く。解は y=e^t
double f(double t, double y)
{
    return y;
}
// 解
double sol(double t)
{
    return exp(t);
}

int main()
{
    printf("***** オイラー法 *****\n");
    double a = 0.0, b = 1.0; // 計算範囲

    for(int N=10; N<=10000; N*=10){
        Vector t(N);
        Vector y(N);
```

次ページへ続く…

```
        double h = (b-a)/N; // 区間幅

        // 分点を作成
        for(int n=0; n<=N; n++){
            t[n] = a + n*h;
        }

        // オイラー法
        y[0] = 1; // 初期条件
        for(int n=0; n<N; n++){
            y[n+1] = y[n] + h*f(t[n], y[n]);
        }

        // 誤差を調査
        double maxerr = 0;
        for(int n=0; n<=N; n++){
            double err = fabs( y[n] - sol(t[n]) );
            if(err>maxerr){ maxerr = err;}
        }
        printf("N=%d のとき、誤差は%.15f\n", N, maxerr);
    }
    return 0;
}
```

4.9 ルンゲ・クッタ法のプログラム

次に，同じ問題を用いて 4 次のルンゲ・クッタ法の精度を確認してみよう．N が 10 倍になるごとに誤差が 1 万分の一になっていればよい．

```
#include <stdio.h>
#include <math.h>
#include "Linear.h"

// y'=y, y(0)=1 を解く。解は y=e^t
double f(double t, double y)
{
    return y;
}
// 解
double sol(double t)
{
    return exp(t);
}

int main()
{
    printf("***** 4次のルンゲ・クッタ法 *****\n");
    double a = 0.0, b = 1.0; // 計算範囲

    for(int N=10; N<=10000; N*=10){
        Vector t(N);
        Vector y(N);
        double h = (b-a)/N; // 区間幅

        // 分点を作成
        for(int n=0; n<=N; n++){
            t[n] = a + n*h;
        }
        // ルンゲ・クッタ法
        y[0] = 1; // 初期条件
        for(int n=0; n<N; n++){
            double k1 = h * f(t[n]    ,y[n]     );
            double k2 = h * f(t[n]+h/2,y[n]+k1/2);
            double k3 = h * f(t[n]+h/2,y[n]+k2/2);
            double k4 = h * f(t[n]+h  ,y[n]+k3  );
            y[n+1] = y[n] + (k1 + 2*k2 + 2*k3 + k4)/6;
        }
```

次ページへ続く…

```
        // 誤差を調査
        double maxerr = 0;
        for(int n=0; n<=N; n++){
            double err = fabs( y[n] - sol(t[n]) );
            if(err>maxerr){ maxerr = err;}
        }
        printf("N=%d のとき、誤差は%.15f\n", N, maxerr);
    }
    return 0;
}
```

4.10 色々な微分方程式への応用

ここでは，色々な微分方程式の解を，4 次のルンゲ・クッタ法を用いて計算してみよう．

まずは，ロトカ・ボルテラ方程式を解き，結果をファイルに出力する．ファイルへの出力には printf の代わりに fprintf を用いる．出力した結果は，Excel や gnuplot などのアプリケーションを用いてグラフ化することができる．

```
#include <stdio.h>
#include "Linear.h"

// x_1'=(a-b*x_2)*x_1
// x_2'=(-c+d*x_1)*x_2
// を解く。ここで、x_1, x_2 は時間 t の関数であり、a, b, c, d は正の定数。
Vector f(double t, Vector x)
{
    Vector y(2);
    y[1] = (1.0 - 1.0*x[2])*x[1];
    y[2] = (-1.0 + 0.5*x[1])*x[2];
    return y;
}

int main()
{
    printf("***** ロトカ・ボルテラ方程式 *****\n");
    double a = 0.0, b = 30.0; // 計算範囲
```

次ページへ続く…

```
    int N = 1000; // 分割数
    int d = 2; // 従属変数の次元
    Matrix S(d, N);
    Vector t(N);
    Vector x(d);

    double h = (b-a)/N; // 区間幅
    x[1] = 0.5; x[2] = 0.5; // 初期条件
    S.SetColVector(0, x);
    for(int n=0; n<=N; n++){
        t[n] = a + n*h;
    }

    for(int n=0; n<N; n++){
        Vector k1 = h * f(t[n],       x       );
        Vector k2 = h * f(t[n] + h/2, x + k1/2);
        Vector k3 = h * f(t[n] + h/2, x + k2/2);
        Vector k4 = h * f(t[n] + h  , x + k3  );
        x += (k1 + 2*k2 + 2*k3 + k4)/6;
        S.SetColVector(n+1, x);
    }
    FILE *fp;
    fp = fopen("l-v.dat", "w");
    for(int n=0; n<=N; n++){
        fprintf(fp, "%f", t[n]);
        for(int i=1; i<=d; i++){
            fprintf(fp, " %f", S[i][n]);
        }
        fprintf(fp, "\n");
    }
    fclose(fp);

    return 0;
}
```

このプログラムを実行すると l-v.dat というファイルができる．gnuplot で

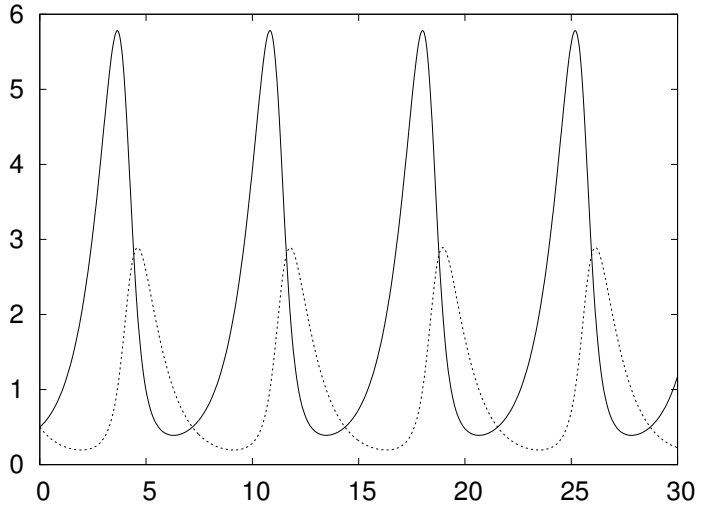

図 2: ロトカ・ボルテラ方程式の数値解

```
plot "l-v.dat" using 1:2 with lines, "l-v.dat" using 1:3 with lines
```

と入力すると図2のようなグラフが表示される．ここで，先に増加するのが非捕食者の数で，後から増加するのが捕食者の数である．2種間のロトカ・ボルテラ方程式の解は周期解になることがわかっており，数値計算結果もそれをよく表している．

ローレンツ方程式の解を計算してみよう．プログラムは以下の通り．ただし，ロトカ・ボルテラ方程式のプログラムと同じ部分は省略している（他の方程式でも同様）．

```
#include <stdio.h>
#include "Linear.h"

// x_1'= -p*x_1 + p*x_2
// x_2'= -x_1*x_3 + r*x_1 - x_2
// x_3'= x_1*x_2 - b*x_3
// を解く。ここで、x_1, x_2, x_3 は時間 t の関数で、
// p=10, r=28, b=8/3
Vector f(double t, Vector x)
{
    const double p = 10, r = 28, b = 8.0/3;
    Vector y(3);
```

次ページへ続く…

```
    y[1] = -p*x[1] + p*x[2];
    y[2] = -x[1]*x[3] + r*x[1] - x[2];
    y[3] = x[1]*x[2] - b*x[3];
    return y;
}

int main()
{
    printf("***** ローレンツ方程式 *****\n");
    double a = 0.0, b = 300.0; // 計算範囲
    int N = 50000; // 分割数
    int d = 3; // 従属変数の次元
    Matrix S(d, N);
    Vector t(N);
    Vector x(d);

    double h = (b-a)/N; // 区間幅
    x[1] = 1; x[2] = 1; x[3] = 10.0; // 初期条件
--------- 中略 ---------
    fp = fopen("lorenz.dat", "w");
--------- 後略 ---------
```

このプログラムを実行すると lorenz.dat というファイルができる．ローレンツ方程式については，時間 t と各従属変数の関係よりも，従属変数の組を一つの点と見たとき，その点がどのような軌道を描くかが興味深い．従属変数の点が動く空間を**相空間**，相空間上に従属変数の点の軌道を描いた図を**相図**という．
それでは，相図を描くために (x, y, z) の軌道をプロットしてみよう．gnuplot では

```
splot "lorenz.dat" using 2:3:4 with lines
```

と入力すればよい．グラフは次ページの図 3 のようになる．
軌道の集合は 2 つの円盤が捻じれて繋がったような図形になっている．従属変数の点は 2 つの円板上を行き来するが，それぞれの円盤を何回ずつ回る，というような明確な規則性は見られない．ローレンツ方程式は常微分方程式なので，初期値が定まれば軌道は一意に定まる（これを決定論的という）が，初期時点のわずかな差が，時間が経つと大きな差になるので，実質的には軌道の予測は不可能である．このような性質を初期値鋭敏性という．決定論的かつ初期値鋭敏性を持つが，軌道が何らかの図形へと近付いていくような系を**カオス**といい，軌道が近付いていく図形を**カオスアトラクタ**という．ローレンツ方程式はカオスについての研究の端緒になった方程式として有名である．

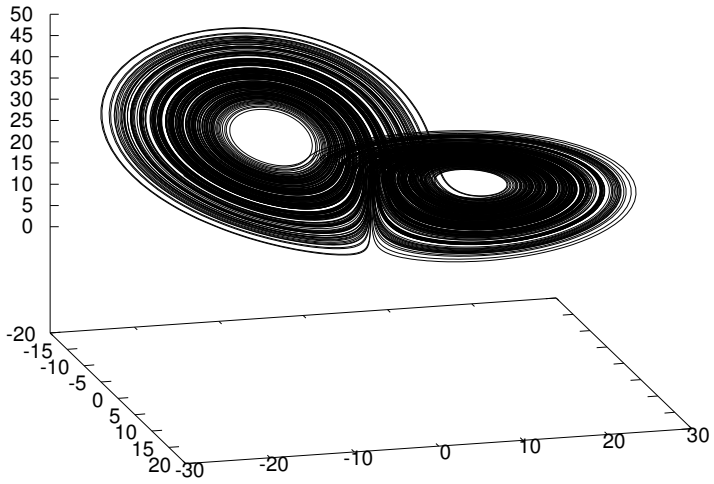

図 3: ローレンツ方程式の数値解

ファン・デル・ポール方程式は 2 階の微分方程式だが，既に述べた方法で 1 階の微分方程式に変換することができる．それをプログラムすると以下のようになる．

```
#include <stdio.h>
#include "Linear.h"

// x_1'= x_2
// x_2'= mu*(1-x_1^2)*x_2 - x_1
// を解く。ここで、x_1, x_2 は時間 t の関数で、
// mu は定数
Vector f(double t, Vector x)
{
const double mu = 1.5;
    Vector y(2);
    y[1] = x[2];
    y[2] = mu*(1-x[1]*x[1])*x[2] - x[1];
    return y;
}
```

次ページへ続く…

```
int main()
{
    printf("***** ファン・デル・ポール方程式 *****\n");
    double a = 0.0, b = 100.0; // 計算範囲
    int N = 3000; // 分割数
    int d = 2; // 従属変数の次元
    Matrix S(d, N);
    Vector t(N);
    Vector x(d);

    double h = (b-a)/N; // 区間幅
    x[1] = 0.1; x[2] = 0.0; // 初期条件
--------- 中略 ---------
    fp = fopen("vdp.dat", "w");
--------- 後略 ---------
```

ファン・デル・ポール方程式についても，点 (x, x') の軌道が興味深いため，相図を描いてみよう．gnuplot では

```
plot "vdp.dat" using 2:3 with lines
```

と入力すればよい．グラフは図 4 のようになる．ファン・デル・ポール方程式については，どのような初期値から計算を初めても，時間が経つと周期解に近付いていくことが知られている．そのよう周期解を極限軌道（リミットサイクル）という．

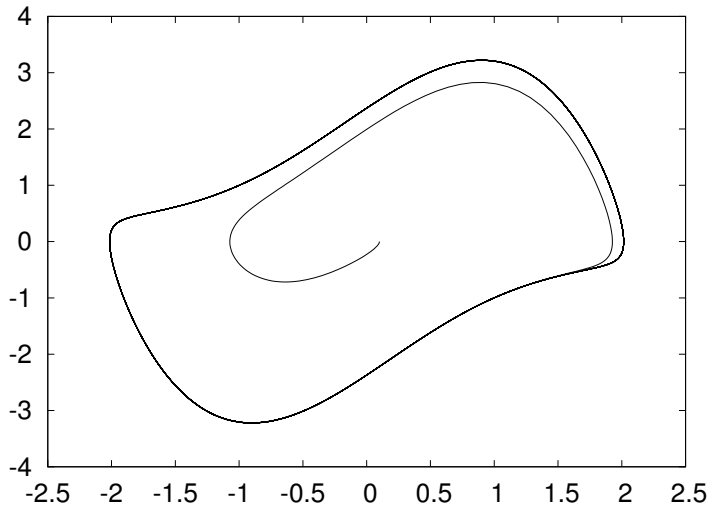

図 4: ファン・デル・ポール方程式の数値解

最後に，ロケットの軌道計算を行ってみよう．具体的には，月までロケットを飛ばし，月の外側を回って地球に戻ってくるような軌道を計算する．簡単のため，軌道は月が公転する平面に限定するものとする．平面上で考えたとしても，地球と月とロケットの 3 つの天体に関する微分方程式を考えると 12 変数の 1 階の微分方程式になってしまう．そこで，地球と月の動きは所与のものとして与え，ロケットの動きのみの微分方程式を考え，解を計算するものとする．本来であれば太陽の重力も無視できないのであるが，今回は太陽の影響は無視する．
地球と月は共通重心を中心に回転するので，その共通重心を原点とし，初期状態では地球は x 軸の負の位置，月は x 軸の正の位置にあるものとする．地球と月は共通重心の周りを反時計回りに回転するものとする（地球の北極側から見ていることになる）．
また，打ち上げは赤道上から行うとし，地球の中心から見た打ち上げ位置の方向と x 軸の正の方向がなす角度を θ とし，打ち上げ方向と x 軸の正の方向がなす角度を φ とする．
さらに，地球と月の質量をそれぞれ M_e, M_m，地球と月の距離を L，地球の半径を R_e，月の公転周期を T，万有引力定数を G，ロケットの打ち上げ速度を v_0，地球，月，ロケットの位置をそれぞれ $\mathbf{x}_e, \mathbf{x}_m, \mathbf{x}$ とすると，微分方程式は以下のようになる．

$$
\begin{cases}
\mathbf{x}_e = -\dfrac{M_m L}{M_e + M_m} \begin{pmatrix} \cos\frac{2\pi}{T}t \\ \sin\frac{2\pi}{T}t \end{pmatrix}, \\
\mathbf{x}_m = \dfrac{M_e L}{M_e + M_m} \begin{pmatrix} \cos\frac{2\pi}{T}t \\ \sin\frac{2\pi}{T}t \end{pmatrix}, \\
\mathbf{x}'' = -G\dfrac{M_e}{|\mathbf{x} - \mathbf{x}_e|^3}(\mathbf{x} - \mathbf{x}_e) - G\dfrac{M_m}{|\mathbf{x} - \mathbf{x}_m|^3}(\mathbf{x} - \mathbf{x}_m), \\
\mathbf{x}(0) = \begin{pmatrix} -\frac{M_m L}{M_e + M_m} + R_e\cos\theta \\ R_e \sin\theta \end{pmatrix}, \qquad \mathbf{x}'(0) = v_0 \begin{pmatrix} \cos\varphi \\ \sin\varphi \end{pmatrix}.
\end{cases}
$$

実際にプログラムを組む上では，単位系を統一しなければならない．ここでは mks 単位系を用いる．すなわち，長さはメートル，重さはキログラム，時間は秒を用いる．
ロケットの軌道は，地球や月と一緒に回転する座標系に変換したものをファイルに出力している．その座標系では，地球と月は静止していることになる．
以上の設定を用いてプログラムにしたものが以下である．

```
#include <stdio.h>
#include <math.h>
#include "Linear.h"

// 2次元ベクトルを回転させる関数
Vector rotate(Vector u, double theta)
```

次ページへ続く…

```
{
    Vector v(2);
    v[1] = cos(theta)*u[1] - sin(theta)*u[2];
    v[2] = sin(theta)*u[1] + cos(theta)*u[2];
    return v;
}

// t:時間変数, x:ロケットの位置と速度,
// Le:原点と地球の距離, Lm:原点と月の距離,
// Me:地球の質量, Mm:月の質量, T:月の公転周期, G:万有引力定数
Vector f(double t, Vector x, double Le, double Lm,
        double Me, double Mm, double T, double G)
{
    double s = t/T*2*M_PI; // 原点から見た月の角度
    Vector z(2);
    z[1] = -Le; z[2] = 0;
    Vector e = rotate(z, s); // 地球の位置
    z[1] = Lm; z[2] = 0;
    Vector m = rotate(z, s); // 月の位置

    // 地球からの相対位置
    double ex = x[1] - e[1]; double ey = x[2] - e[2];
    double ed = pow(ex*ex + ey*ey, 1.5); // 距離の３乗
    // 月からの相対位置
    double mx = x[1] - m[1]; double my = x[2] - m[2];
    double md = pow(mx*mx + my*my, 1.5); // 距離の３乗
    Vector y(4);
    y[1] = x[3];
    y[2] = x[4];
    y[3] = - G*Me*ex/ed - G*Mm*mx/md;
    y[4] = - G*Me*ey/ed - G*Mm*my/md;
    return y;
}

int main()
{
```

次ページへ続く…

```
printf("***** ロケットの軌道計算 *****\n");
double a = 0.0, b = 1000000.0; // 計算範囲
int N = 10000; // 分割数

// 諸量
double th = 0; // 打ち上げ位置 (度)
double ph = 30; // 打ち上げ角度 (度)
double v0 = 11090; // 打ち上げ速度 (m/s)
double Me = 5.976E24; // 地球の質量 (kg)
double Mm = 7.349E22; // 月の質量 (kg)
double Re = 6.378E6; // 地球の半径 (m)
double Rm = 1.737E6; // 月の半径 (m)
double L = 3.844E8; // 地球と月の距離 (m)
double T = 2.361E6; // 月の公転周期 (s)
double G = 6.674E-11; // 万有引力定数 (m^3/(kg*s^2))

double Le = L*Mm/(Me + Mm); // 原点と地球の距離
double Lm = L*Me/(Me + Mm); // 原点と月の距離

double h = (b-a)/N; // 区間幅
Vector t(N);
for(int n=0; n<=N; n++){
    t[n] = a + n*h;
}

Matrix S(2, N); // 位置のみ記録する
Vector x(4);

// ロケットの初期位置
x[1] = -Le + Re*cos(th/180*M_PI); x[2] = Re*sin(th/180*M_PI);
// ロケットの初期速度
x[3] = v0*cos(ph/180*M_PI); x[4] = v0*sin(ph/180*M_PI);

S[1][0] = x[1]; S[2][0] = x[2];
Vector u(2);
for(int n=0; n<N; n++){
```

次ページへ続く…

```
        Vector k1 = h * f(t[n],       x       , Le, Lm, Me, Mm, T, G);
        Vector k2 = h * f(t[n] + h/2, x + k1/2, Le, Lm, Me, Mm, T, G);
        Vector k3 = h * f(t[n] + h/2, x + k2/2, Le, Lm, Me, Mm, T, G);
        Vector k4 = h * f(t[n] + h  , x + k3  , Le, Lm, Me, Mm, T, G);
        x += (k1 + 2*k2 + 2*k3 + k4)/6;
        u[1] = x[1]; u[2] = x[2];
        // 地球や月と一緒に回転する座標系に変換
        Vector v = rotate(u, -t[n+1]/T*2*M_PI);
        S.SetColVector(n+1, v);
        // 地球および月との衝突判定
        double ex = v[1] + Le; double ey = v[2]; // 地球からの相対位置
        double mx = v[1] - Lm; double my = v[2]; // 月からの相対位置
        if(ex*ex+ey*ey<=Re*Re || mx*mx+my*my<=Rm*Rm){
            N = n+1;
            break;
        }
    }
    printf("Last time is t=%f\n", t[N]);
    FILE *fp;
    fp = fopen("rocket.dat", "w");
    // 地球および月の輪郭を描画
    for(int i=0; i<=200; i++){
        double t = i*M_PI/100.0;
        fprintf(fp, "%f %f\n", -Le + Re*cos(t), Re*sin(t));
    }
    fprintf(fp, "\n");
    for(int i=0; i<=200; i++){
        double t = i*M_PI/100.0;
        fprintf(fp, "%f %f\n", Lm + Rm*cos(t), Rm*sin(t));
    }
    fprintf(fp, "\n");
    // ロケットの軌道を描画
    for(int n=0; n<=N; n++){
        fprintf(fp, "%f %f\n", S[1][n], S[2][n]);
    }
    fclose(fp);

    return 0;
}
```

このプログラムにおいて，ロケットが地球や月と衝突した場合には，その時点で計算を中止している．また，地球と月の輪郭の曲線も描いている．
gnuplot で表示する際には，そのままだと縦横のスケールが自動で設定され，地球や月が楕円になってしまうので，まず

```
set size ratio -1
```

と入力して，縦横のスケールを等しくし、その後に

```
plot "rocket.dat" with lines
```

と入力すればよい．実際のグラフは図 5 のようになる．左側の丸が地球，右側の小さな丸が月である．
打ち上げ速度や打ち上げ位置，打ち上げ角度等を変化させてみるとわかるが，月の向こう側を通って地球に戻ってくるには，かなり細かい数値の調整が必要である．特に，打ち上げ速度は地球脱出速度（第 2 宇宙速度）である秒速 11.2Km よりわずかに小さくする必要がある．
実際の宇宙飛行では，要所要所で方向や速度の微調整が行われる（エネルギー保存のため，速度修正より方向修正の方が容易である）．本プログラムでも，特定の時刻に方向修正を行って月を周る周回軌道に乗せ，さらに次の特定の時刻に再度方向修正を行って地球に戻ってくる，等の動きを追加してみても面白いだろう．

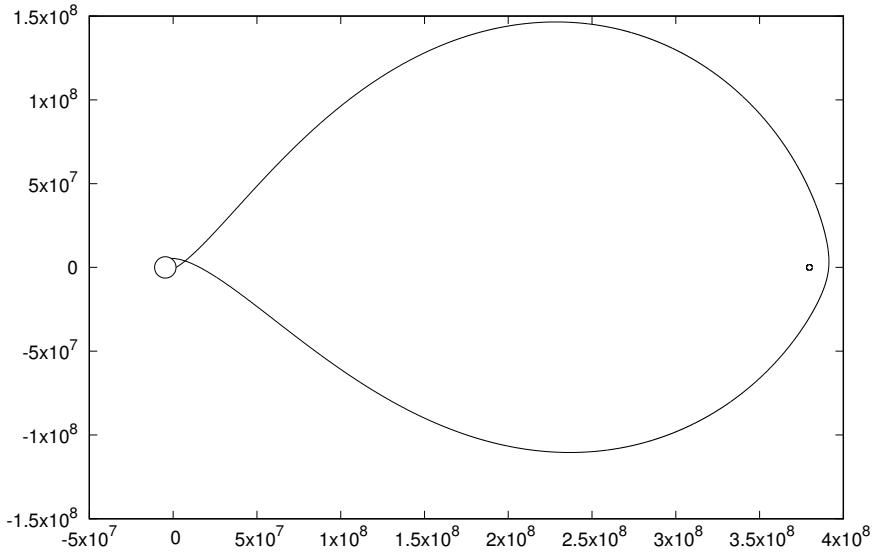

図 5: ロケットの軌道計算

4.11　補足

オイラー法，ルンゲ・クッタ法などは，正確には，陽的オイラー法，陽的ルンゲ・クッタ法と呼ばれることもある．常微分方程式の数値解法には，他にも，陰的オイラー法，陰的ルンゲ・クッタ法，ピカールの逐次近似法，予測子・修正子法など，様々な方法があり，一長一短があるが，おそらく最もよく用いられているのは，今回説明した 4 次の陽的ルンゲ・クッタ法である．

本稿では，常微分方程式の初期値問題の数値解法について説明したが，境界値問題，例えば

$$\begin{cases} y(a) = y_a, \\ y(b) = y_b, \\ y'' = f(x, y, y') \end{cases}$$

のような形の方程式も重要である．境界値問題の数値解法としては，差分法などが用いられることが多いが，時間の関係で今回は省略した．

4.12　課題

課題１　解のわかっている微分方程式を自由に選び，オイラー法，修正オイラー法，4 次のルンゲ・クッタ法で計算を行い，誤差が刻み幅に対してどのように減少していくか調べなさい．

課題２　2 階の微分方程式を自由に選び，4 次のルンゲ・クッタ法を用いて解を計算し，グラフを描きなさい．横軸に独立変数，縦軸に y と y' を取ったグラフと，横軸に y，縦軸に y' を取ったグラフ（相図）を作成すること．

5　偏微分方程式の差分解法

本章では，**偏微分方程式**の数値解法について説明する．偏微分方程式とは，関数の偏微分についての関係を表した方程式であり，最も基本的なものとして，**熱方程式**が知られている．熱方程式は，その名前からもわかる通り，元々は物理の方程式であるが，自然現象や社会現象など，様々な所に現れる極めて重要な方程式である．また，熱方程式に対する数値解法は，他の様々な偏微分方程式の数値解法へ応用が可能であるので，熱方程式の数値解法を知ることは大いに意味がある．そこで，まずは熱方程式とその数値解法を見ていくことにしよう．

5.1　熱方程式

以下の偏微分方程式が熱方程式である（導出については後ほど説明する）．

$$\begin{cases} \dfrac{\partial u}{\partial t} = \lambda \dfrac{\partial^2 u}{\partial x^2} & (t > 0,\ a < x < b), \\ u(0, x) = u_0(x) & (a \leq x \leq b), \\ u(t, a) = \alpha(t) & (t > 0) \\ u(t, b) = \beta(t) & (t > 0) \end{cases}$$

ここで，x は空間方向の位置を表し，t は時間を表す変数である．$u(t, x)$ が求めたい未知関数で，$u_0(x),\ \alpha(t),\ \beta(t)$ は与えられた関数である．また，λ は正の定数である．

熱方程式は，区間 $[a,\ b]$ で表される針金の熱分布の時間変化を表す方程式とみることができる．ここで，条件 $u(0, x) = u_0(x)$ は，時刻 $t = 0$ における温度分布を表しているので，**初期条件**と呼ばれ，条件 $u(t, a) = \alpha(t)$ と $u(t, b) = \beta(t)$ は，端点 $x = a$ および $x = b$ での温度を $\alpha(t)$ および $\beta(t)$ に固定することを意味しているので，**境界条件**と呼ばれる．特にこのような，関数値を指定する条件を**ディリクレ境界条件**という．場合によっては，端点における微分の値を指定することもあり，そのような条件を**ノイマン境界条件**と言うが，ここでは簡単のためディリクレ境界条件のみを考える．

ここで，熱方程式の導出について説明しておこう．一本の針金を考え，時刻 t，位置 $x \in [a, b]$ の温度を $u(t, x)$ とする．熱は時間とともに針金上を移動するが，その際の熱の流量は，熱分布の傾き（温度勾配）の大きさに比例することが知られている．つまり，温度変化が急な部分ほど熱が多く移動するということである．具体的には，各点における単位時間かつ単位断面積あたりの熱流量 Q は

$$Q = -K \frac{\partial u}{\partial x}$$

と表される．ここで K は針金の熱伝導率である．ここで，マイナスの符号が付いているのは，熱は温度勾配と逆方向に流れるからである．

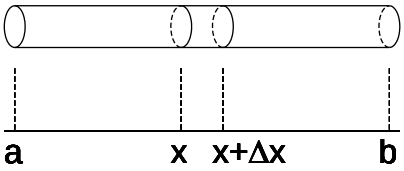

図 6: 針金上の熱分布

さて，針金上の位置 x および $x+\Delta x$ における断面で囲まれる部分を V とすると，単位時間あたりに V に流入する熱量は，断面積を S として

$$-SK\frac{\partial}{\partial x}u(t,x)+SK\frac{\partial}{\partial x}u(t,x+\Delta x)=SK\left(\frac{\partial}{\partial x}u(t,x+\Delta x)-\frac{\partial}{\partial x}u(t,x)\right)$$

となる．さらに，V の体積は $S\Delta x$ であるので，針金の比熱を c，密度を ρ とすると，単位時間あたりの V の温度変化は，平均すると

$$\frac{SK}{c\rho S\Delta x}\left(\frac{\partial}{\partial x}u(t,x+\Delta x)-\frac{\partial}{\partial x}u(t,x)\right)=\frac{K}{c\rho\Delta x}\left(\frac{\partial}{\partial x}u(t,x+\Delta x)-\frac{\partial}{\partial x}u(t,x)\right)$$

となる．ここで，$\Delta x\to 0$ とすると

$$\frac{1}{\Delta x}\left(\frac{\partial}{\partial x}u(t,x+\Delta x)-\frac{\partial}{\partial x}u(t,x)\right)\ \ \to\ \ \frac{\partial^2}{\partial x^2}u(t,x)$$

より，

$$\frac{\partial u}{\partial t}=\frac{K}{c\rho}\cdot\frac{\partial^2 u}{\partial x^2}$$

が成り立つことがわかる．$\lambda=\dfrac{K}{c\rho}$ とおけば，最初に紹介した熱方程式の形になる．

5.2 差分

微分方程式の数値計算では，微分を関数の差で近似することが多い．微分を近似する関数の差を**差分**といい，差分を用いる数値計算法を**差分法**という．それでは，いくつかの差分を紹介しよう．

$f(x)$ を C^2 級関数とすると，テイラー展開より

$$f(x+h)=f(x)+hf'(x)+O(h^2)$$

が成り立つ．よって

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

が成り立つ．すなわち，$\dfrac{f(x+h) - f(x)}{h}$ で $f'(x)$ を近似することができる．これを**（一階の）前進差分**という．

同様に，

$$f(x-h) = f(x) - hf'(x) + O(h^2)$$

より

$$f'(x) = \frac{f(x) - f(x-h)}{h} + O(h)$$

も成り立つ．$\dfrac{f(x) - f(x-h)}{h}$ を**（一階の）後退差分**という．

また，$f(x)$ が C^3 級のとき，

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$

より

$$f(x+h) - f(x-h) = 2hf'(x) + O(h^3)$$

が成り立つので，

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

が成り立つ．$\dfrac{f(x+h) - f(x-h)}{2h}$ を**（一階の）中心差分**という．$f(x)$ が滑らかな場合，中心差分は前進差分や後退差分より精度が良いということがわかる．前進差分と後退差分は，まとめて**片側差分**と呼ばれることもある．

次に，2 階微分の近似を考えてみよう．$f(x)$ が C^4 級のとき，

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4)$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4)$$

が成り立つので，

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

が成り立つ．$\dfrac{f(x+h) - 2f(x) + f(x-h)}{h^2}$ を**（二階の）中心差分**という．

5.3 陽解法

それでは，熱方程式の数値解法に入ろう．時間方向 t を刻み幅 Δt で分割し，上付きの添え字を用いて

$$t^m = m\Delta t$$

とする（m 乗ではなく添え字なので注意）．また，空間方向 x を刻み幅 $\Delta x = \dfrac{b-a}{N}$ で分割し，

$$x_n = a + n\Delta x$$

とする．さらに，$u(t^m, x_n)$ の近似を u_n^m とし，微分の近似として前進差分および中心差分

$$\begin{aligned}
\frac{\partial u}{\partial t} &\approx \frac{u(t^{m+1}, x_n) - u(t^m, x_n)}{\Delta t} \approx \frac{u_n^{m+1} - u_n^m}{\Delta t}, \\
\frac{\partial^2 u}{\partial x^2} &\approx \frac{u(t^m, x_{n+1}) - 2u(t^m, x_n) + u(t^m, x_{n-1})}{\Delta x^2} \approx \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{\Delta x^2}
\end{aligned}$$

を導入し（$\approx$ は近似を意味する），熱方程式に代入すると

$$\frac{u_n^{m+1} - u_n^m}{\Delta t} = \lambda \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{\Delta x^2}$$

となる（差分による近似と u_n^m による近似が二重に入っているので，本当に近似になっているかどうかは厳密にはわからないが，今のところはよしとしておく）．上の式を見ると，u_n^{m+1} を $u_{n-1}^m, u_n^m, u_{n+1}^m$ で書くことができるので，$t = t^0$ から順に計算していき，熱方程式の近似解を求めることができる．

初期条件や境界条件も入れてまとめて書くと，$r = \lambda \dfrac{\Delta t}{\Delta x^2}$ と置いて

$$\begin{cases}
u_n^0 = u_0(x_n) & (0 \le n \le N), \\
u_n^{m+1} = r u_{n+1}^m + (1-2r) u_n^m + r u_{n-1}^m & (m \ge 0,\ 0 < n < N), \\
u_0^{m+1} = \alpha(t^{m+1}) & (m \ge 0), \\
u_N^{m+1} = \beta(t^{m+1}) & (m \ge 0)
\end{cases}$$

となる．この計算スキームは，次の時間ステップの近似解を前の時間ステップの近似解を用いて書き下せる（これを陽に書き下せる，という）ので，差分法の中でも**陽解法**と呼ばれている．

5.4 解の収束性

陽解法は，$r \le \dfrac{1}{2}$ を満たしながら Δt と Δx を小さくしていけば，真の解に収束することが知られている．ここではこれを示そう．まず，

$$e_n^m = u(t^m, x_n) - u_n^m$$

とする．つまり，e_n^m は真の解と数値解の差とする．このとき，解が十分に滑らかだとすると，$t = t^m,\ x = x_n$ において

$$\frac{\partial u}{\partial t} = \frac{u(t^{m+1}, x_n) - u(t^m, x_n)}{\Delta t} + O(\Delta t),$$
$$\frac{\partial^2 u}{\partial x^2} = \frac{u(t^m, x_{n+1}) - 2u(t^m, x_n) + u(t^m, x_{n-1})}{\Delta x^2} + O(\Delta x^2)$$

が成り立つので，

$$\frac{u(t^{m+1}, x_n) - u(t^m, x_n)}{\Delta t} - \lambda \frac{u(t^m, x_{n+1}) - 2u(t^m, x_n) + u(t^m, x_{n-1})}{\Delta x^2} = O(\Delta t) + O(\Delta x^2)$$

となる．一方で，陽解法のスキームより

$$\frac{u_n^{m+1} - u_n^m}{\Delta t} - \lambda \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{\Delta x^2} = 0$$

なので，辺々引くと

$$\frac{e_n^{m+1} - e_n^m}{\Delta t} - \lambda \frac{e_{n+1}^m - 2e_n^m + e_{n-1}^m}{\Delta x^2} = O(\Delta t) + O(\Delta x^2)$$

となる．さらに両辺に Δt をかけて整理すると

$$e_n^{m+1} = re_{n-1}^m + (1-2r)e_n^m + re_{n+1}^m + \Delta t\Big(O(\Delta t) + O(\Delta x^2)\Big)$$

となる．ここで $r \leq \dfrac{1}{2}$ のとき，n について両辺の絶対値の最大値を取ると

$$\begin{aligned}\max_{0<n<N} |e_n^{m+1}| &\leq r \max_{0<n<N} |e_n^m| + (1-2r) \max_{0<n<N} |e_n^m| + r \max_{0<n<N} |e_n^m| + \Delta t\Big(O(\Delta t) + O(\Delta x^2)\Big) \\ &= (r + (1-2r) + r) \max_{0<n<N} |e_n^m| + \Delta t\Big(O(\Delta t) + O(\Delta x^2)\Big) \\ &= \max_{0<n<N} |e_n^m| + \Delta t\Big(O(\Delta t) + O(\Delta x^2)\Big)\end{aligned}$$

が成り立つ．ここで，時間方向の計算範囲を $0 \leq t \leq T$ とし，$\Delta t = \dfrac{T}{M}$ とすると，$0 < m \leq M$ について

$$\begin{aligned}\max_{0<n<N} |e_n^m| &\leq \max_{0<n<N} |e_n^0| + m\Delta t\Big(O(\Delta t) + O(\Delta x^2)\Big) = m\Delta t\Big(O(\Delta t) + O(\Delta x^2)\Big) \\ &\leq M\Delta t\Big(O(\Delta t) + O(\Delta x^2)\Big) = T\Big(O(\Delta t) + O(\Delta x^2)\Big)\end{aligned}$$

となる．これより，$r \leq \dfrac{1}{2}$ のとき，Δt と Δx を小さくしていけば，陽解法による数値解は真の解に収束することがわかった．

逆に，$r > \dfrac{1}{2}$ のときには収束性は必ずしも言えない．実際，このときには陽解法の数値解が収束しないような解の例を作ることができる．

5.5 安定性

実際の数値計算においては，計算過程で丸め誤差などの誤差が混入する．よって，安定して数値計算を行うには，混入した誤差が増幅されないようにしなければならない．

いま，u_n^m に ε_n^m の誤差が混入したとしよう．このとき，この誤差の影響によって u_n^{m+1} に加わる誤差 ε_n^{m+1} を求めてみると

$$\begin{aligned} u_n^{m+1} + \varepsilon_n^{m+1} &= r(u_{n-1}^m + \varepsilon_{n-1}^m) + (1-2r)(u_n^m + \varepsilon_n^m) + r(u_{n+1}^m + \varepsilon_{n+1}^m) \\ &= \left(ru_{n-1}^m + (1-2r)u_n^m + ru_{n+1}^m\right) + \left(r\varepsilon_{n-1}^m + (1-2r)\varepsilon_n^m + r\varepsilon_{n+1}^m\right) \end{aligned}$$

より

$$\varepsilon_n^{m+1} = r\varepsilon_{n-1}^m + (1-2r)\varepsilon_n^m + r\varepsilon_{n+1}^m$$

と表されることがわかる．$r \leq \dfrac{1}{2}$ のときには

$$\max_{0<n<N} |\varepsilon_n^{m+1}| \leq r \max_{0<n<N} |\varepsilon_n^m| + (1-2r) \max_{0<n<N} |\varepsilon_n^m| + r \max_{0<n<N} |\varepsilon_n^m| = \max_{0<n<N} |\varepsilon_n^m|$$

より，時間ステップを経ても混入した誤差は増大しないということがわかる．

一方で $r > \dfrac{1}{2}$ のとき，少々天下り的だが，u_n^m に $\varepsilon_n^m = \eta(-1)^n$ の誤差が混入したとすると，

$$\begin{aligned} \varepsilon_n^{m+1} &= r\varepsilon_{n-1}^m + (1-2r)\varepsilon_n^m + r\varepsilon_{n+1}^m \\ &= r\eta(-1)^{n-1} + (1-2r)\eta(-1)^n + r\eta(-1)^{n+1} \\ &= (1-4r)\eta(-1)^n \end{aligned}$$

より，u_n^{m+1} には $\varepsilon_n^{m+1} = (1-4r)\eta(-1)^n$ の誤差が混入することになる．$r > \dfrac{1}{2}$ のとき $|1-4r| > 1$ となるので，誤差は時間ステップを経るごとに増幅され，振幅が増大していくことになる．

数値計算においては，通常はありとあらゆる形の誤差が混入するので，いま述べたような $\varepsilon_n^m = \eta(-1)^n$ という形の誤差も，(ごくわずかかもしれないが) 混入してしまう．$r > \dfrac{1}{2}$ のときには，その誤差は指数関数的に増大するので，数値解を安定して計算することが困難になってしまう．

一方で，上で見たように，$r \leq \dfrac{1}{2}$ のときには混入した誤差が増加しない．このような，時間ステップが進んでも誤差が増幅されないための条件を**安定性条件**という．陽解法の安定性条件は $r \leq \dfrac{1}{2}$ である．

5.6 陰解法

陽解法において安定性条件を満たすためには，時間刻みを空間刻みの 2 乗のオーダーに取らなければならない．もし，空間方向の構造を細かく見るために Δx を細かく取ったとすると，Δt を非常に小さく取らねばならず，計算量という点で難がある．例えば，$\lambda = 1, \Delta x = 0.001$ のときには，陽解法の安定性条件を満たすために Δt を 0.0000005 以下に取らねばならず，時間ステップを進めるのに非常に多くの計算が必要になってしまう．そこで用いられるのが陰解法である．陰解法では，時刻 t^{m+1} において，差分化した関係式が成り立つように $u_\cdot^m$ を決める．具体的には

$$\frac{u_n^{m+1} - u_n^m}{\Delta t} = \lambda \frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{\Delta x^2}$$

と差分化することになる．証明には行列の固有値解析が必要なので詳しくは述べないが，陰解法は**無条件安定**，つまりどんな Δt や Δx に対しても安定に計算できることが知られている．

初期条件や境界条件を加味すると，陰解法は

$$\begin{cases} u_n^0 = u_0(x_n) & (0 \le n \le N), \\ u_0^{m+1} = \alpha(t^{m+1}) & (m \ge 0), \\ -ru_{n-1}^{m+1} + (1+2r)u_n^{m+1} - ru_{n+1}^{m+1} = u_n^m & (m \ge 0,\ 1 \le n \le N-1), \\ u_N^{m+1} = \beta(t^{m+1}) & (m \ge 0) \end{cases}$$

となる．解は毎回，連立一次方程式を解いて求めることになる．ここで，端点 u_0^{m+1} と u_N^{m+1} は境界条件から決まるのだが，この端点も未知数として連立一次方程式を立てると，プログラムが簡単になる．具体的には，$N+1$ 次の連立一次方程式

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -r & 1+2r & -r & \cdots & 0 & 0 & 0 \\ 0 & -r & 1+2r & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1+2r & -r & 0 \\ 0 & 0 & 0 & \cdots & -r & 1+2r & -r \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0^{m+1} \\ u_1^{m+1} \\ u_2^{m+1} \\ \vdots \\ u_{N-2}^{m+1} \\ u_{N-1}^{m+1} \\ u_N^{m+1} \end{pmatrix} = \begin{pmatrix} \alpha(t^{m+1}) \\ u_1^m \\ u_2^m \\ \vdots \\ u_{N-2}^m \\ u_{N-1}^m \\ \beta(t^{m+1}) \end{pmatrix}$$

を解くことで，$u_\cdot^m$ から $u_\cdot^{m+1}$ を計算する．連立一次方程式の係数行列は 3 重対角行列となり，最初と最後の行以外は，対角成分が $1+2r$，その両脇の成分が $-r$ なる．

5.7 クランク・ニコルソン法

今まで述べた陽解法や陰解法では，時間に関する片側差分の誤差が $O(\Delta t)$，空間に関する中心差分の誤差が $O(\Delta x^2)$ となっており，時間方向の精度があまり良くない．ところで，時間方向の差分

$$\frac{u(t^{m+1}, x_n) - u(t^m, x_n)}{\Delta t}$$

は，$u'(t^m, x_n)$ あるいは $u'(t^{m+1}, x_n)$ の近似として見れば片側差分となり，誤差は $O(\Delta t)$ であるが，$u'(t^{m+1/2}, x_n)$ の近似と見れば中心差分と考えることができ，誤差は $O(\Delta t^2)$ となる．よって，$t = t^{m+1/2}$ で方程式を離散化すれば，時間に関しても $O(\Delta t^2)$ の精度を実現できると考えられる．しかし，$t = t^{m+1/2}$ は時間分割の分点ではないというのが問題である．

一般に，$f(x)$ が C^2 級のとき，

$$f(x+h) = f(x) + hf'(x) + O(h^2)$$
$$f(x-h) = f(x) - hf'(x) + O(h^2)$$

より，

$$f(x) = \frac{f(x+h) + f(x-h)}{2} + O(h^2)$$

となることから，$t = t^{m+1/2}$ における熱方程式の左辺を $t = t^m$ のときと $t = t^{m+1}$ のときの平均で近似すると

$$\frac{\partial}{\partial t}u(t^{m+1/2}, x) = \frac{1}{2}\left(\frac{\partial}{\partial t}u(t^{m+1}, x) + \frac{\partial}{\partial t}u(t^m, x)\right) + O(\Delta t^2)$$

が成り立つ．よって，右辺の 2 項をそれぞれ 2 階の中心差分で近似することにより，$t = t^{m+1/2}$ での離散化を実現することができる．具体的には

$$\frac{u_n^{m+1} - u_n^m}{\Delta t} = \frac{\lambda}{2}\left(\frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{\Delta x^2} + \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{\Delta x^2}\right)$$

となる．r を用いて書き換えると

$$-ru_{n-1}^{m+1} + 2(1+r)u_n^{m+1} - ru_{n+1}^{m+1} = ru_{n-1}^m + 2(1-r)u_n^m + ru_{n+1}^m$$

となる．これを**クランク・ニコルソン法**という．クランク・ニコルソン法は $O(\Delta t^2) + O(\Delta x^2)$ の精度を持ち，さらに無条件安定であることが知られている．クランク・ニコルソン法も陰解法の一種なので，解は毎回，連立一次方程式を解いて求めることになる．具体的には，

$$
\begin{pmatrix}
1 & 0 & 0 & \cdots & 0 & 0 & 0 \\
-r & 2(1+r) & -r & \cdots & 0 & 0 & 0 \\
0 & -r & 2(1+r) & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 2(1+r) & -r & 0 \\
0 & 0 & 0 & \cdots & -r & 2(1+r) & -r \\
0 & 0 & 0 & \cdots & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
u_0^{m+1} \\ u_1^{m+1} \\ u_2^{m+1} \\ \vdots \\ u_{N-2}^{m+1} \\ u_{N-1}^{m+1} \\ u_N^{m+1}
\end{pmatrix}
$$

$$
= \begin{pmatrix}
\alpha(t^{m+1}) \\
ru_0^m + 2(1-r)u_1^m + ru_2^m \\
ru_1^m + 2(1-r)u_2^m + ru_3^m \\
\vdots \\
ru_{N-3}^m + 2(1-r)u_{N-2}^m + ru_{N-1}^m \\
ru_{N-2}^m + 2(1-r)u_{N-1}^m + ru_N^m \\
\beta(t^{m+1})
\end{pmatrix}
$$

を解けばよい.

5.8 陽解法のプログラム

陽解法のプログラムを以下に示す．熱が変化していく様子をグラフにするには，Excel や gnuplot を用いて 3 次元グラフを表示すればよい．全ての空間ステップ，時間ステップでの u の値を出力するとデータが多くなり過ぎてうまくグラフが描けないので，MM や NN でファイルに出力する間隔を指定している．初期条件 u_0 は $u_0(x) = \min(x,\, 1-x)$ という山型の関数を与えている．

全ての時間刻みと空間刻みについて数値解を計算するとデータ量が膨大になるので，時間方向は 1 ステップごとに前回の結果を上書きするようにしている．

```
#include <stdio.h>
#include "Linear.h"

#define M 1000 // 時間分割
#define MM 50 // 時間のファイル出力間隔
#define N 20 // 空間分割
#define NN 1 // 空間のファイル出力間隔

// 境界条件
double alpha(double t)
{
    return 0;
}
double beta(double t)
{
    return 0;
}
// 初期条件
double u0(double x)
{
    if(x<0.5){
        return x;
    } else {
        return 1-x;
    }
}

int main()
{
    double lambda = 1;
    double a = 0; // 左端
    double b = 1; // 右端
    double T = 0.5; // 時間の計算範囲
    double dt = T/M;
    double dx = (b-a)/N;
```

次ページへ続く…

```
    Vector x(N), u(N), u2(N);

    printf("***** 熱方程式（陽解法） *****\n");

    double r = lambda*dt/dx/dx;
    printf("r = %.15f\n", r);

    // 初期条件および x_n を設定
    for(int n=0; n<=N; n++){
        x[n] = a + n*dx;
        u[n] = u0(x[n]);
    }

    FILE *fp;
    fp = fopen("heat_explicit.dat","w");
    for(int m=0; m<=M; m++){
        double t = m*dt;
        if(m%MM==0){
            for(int n=0; n<=N; n+=NN){
                fprintf(fp, "%f %f %f\n",t, x[n], u[n]);
            }
            fprintf(fp, "\n");
        }
        t = (m+1)*dt;
        u2[0] = alpha(t);
        u2[N] = beta(t);
        for(int n=1; n<=N-1; n++){
            u2[n] = r*u[n+1] + (1-2*r)*u[n] + r*u[n-1];
        }
        u = u2.Copy();
    }
    fclose(fp);

    return 0;
}
```

グラフを表示するには，gnuplot で

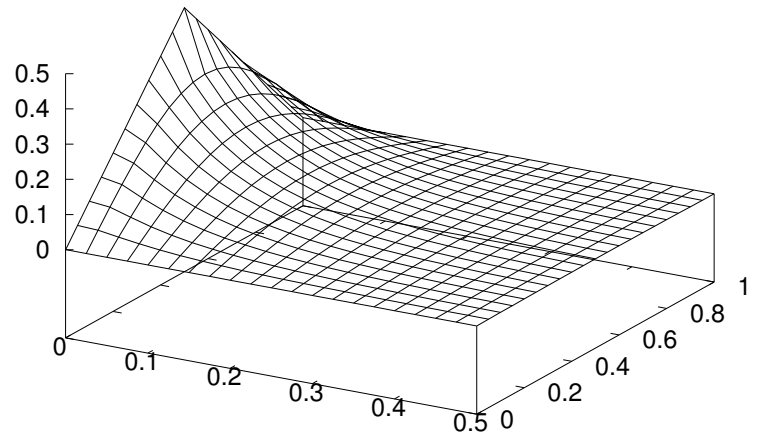

図 7: 熱方程式の解

```
splot "heat_explicit.dat" with lines
```

と入力する．解は図 7 のようになる．

Excel でグラフを表示することもできる．この場合は，プログラムの一部を以下のように書き換え，出力された csv ファイルを Excel で読み込み，グラフの種類として「ワイヤーフレーム 3D 等高線」を描けばよい．

```
--------- ここまで同じ ---------
    FILE *fp;
    fp = fopen("heat_explicit.csv","w");
    for(int n=0; n<=N; n+=NN){
        fprintf(fp, ",%f",x[n]);
    }
    fprintf(fp, "\n");
    for(int m=0; m<=M; m++){
        double t = m*dt;
        if(m%MM==0){
            fprintf(fp, "%f",t);
            for(int n=0; n<=N; n+=NN){
                fprintf(fp, ",%f",u[n]);
            }
            fprintf(fp, "\n");
        }
        t = (m+1)*dt;
--------- これ以降同じ ---------
```

陽解法では，安定性条件 $r \leq 0.5$ が満たされないときには計算が上手く行かないことも確認しておこう．

5.9 陰解法のプログラム

陰解法の場合は，毎時間ステップごとに連立一次方程式を解く必要があるが，ここではガウスの消去法を用いよう．プログラムは以下のようになる．

```
#include <stdio.h>
#include "Linear.h"

#define M 100 // 時間分割
#define MM 5 // 時間のファイル出力間隔
#define N 20 // 空間分割
#define NN 1 // 空間のファイル出力間隔

// 境界条件
double alpha(double t)
{
    return 0;
}
double beta(double t)
{
    return 0;
}
// 初期条件
double u0(double x)
{
    if(x<0.5){
        return x;
    } else {
        return 1-x;
    }
}
```

次ページへ続く…

```
int main(){
    double lambda = 1;
    double a = 0; // 左端
    double b = 1; // 右端
    double T = 0.5; // 時間の計算範囲
    double dt = T/M;
    double dx = (b-a)/N;
    Matrix A(N+1, N+1);
    Vector x(N);
    Vector u(N+1);

    printf("***** 熱方程式（陰解法） *****\n");

    double r = lambda*dt/dx/dx;
    printf("r = %.15f\n", r);

    // 行列を作成
    for(int i=1; i<=N+1; i++){
        for(int j=1; j<=N+1; j++){
            if(i==j){
                A[i][j] = 1+2*r;
            } else if(i-j==1 || i-j==-1){
                A[i][j] = -r;
            } else {
                A[i][j] = 0;
            }
        }
    }
    A[1][1]   = 1; A[1][2]     = 0;
    A[N+1][N] = 0; A[N+1][N+1] = 1;

    // 初期条件および x_n を設定
    for(int n=0; n<=N; n++){
        x[n] = a + n*dx;
        u[n] = u0(x[n]);
    }
```

次ページへ続く…

```
    FILE *fp;
    fp = fopen("heat_implicit.dat","w");
    for(int m=0; m<=M; m++){
        double t = m*dt;
        if(m%MM==0){
            for(int n=0; n<=N; n+=NN){
                fprintf(fp, "%f %f %f\n",t, x[n], u[n]);
            }
            fprintf(fp, "\n");
        }
        t = (m+1)*dt;
        u[0] = alpha(t);
        u[N] = beta(t);
        u = u.Shift(1); // 添え字を 1 始まりにするために 1 だけシフトする
        u = A.Gauss(u);
        u = u.Shift(-1); // 添え字を 0 始まりに戻す
    }
    fclose(fp);

    return 0;
}
```

Excel でグラフを描く場合は，ファイルに出力する部分を陽解法のときと同様に変更すればよい．

5.10 クランク・ニコルソン法のプログラム

クランク・ニコルソン法のプログラムを以下に示す．ただし，陰解法と同じ部分は省略する．

```
--------- 前略 ---------
    Vector u(N+1), v(N+1);

    printf("***** 熱方程式（クランク・ニコルソン法） *****\n");
```

次ページへ続く…

```
    double r = lambda*dt/dx/dx;
    printf("r = %.15f\n", r);

    // 行列を作成
    for(int i=1; i<=N+1; i++){
        for(int j=1; j<=N+1; j++){
            if(i==j){
                A[i][j] = 2*(1+r);
            } else if(i-j==1 || i-j==-1){
                A[i][j] = -r;
            } else {
                A[i][j] = 0;
            }
        }
    }
--------- 中略 ---------
    fp = fopen("heat_cn.dat","w");
--------- 中略 ---------
        v[0] = alpha(t);
        v[N] = beta(t);
        for(int i=1; i<=N-1; i++){
            v[i] = r*u[i-1] + 2*(1-r)*u[i] + r*u[i+1];
        }
        v = v.Shift(1); // 添え字を 1 始まりにするために 1 だけシフトする
        u = A.Gauss(v);
        u = u.Shift(-1); // 添え字を 0 始まりに戻す
--------- 後略 ---------
```

5.11 差分解法の応用

熱方程式以外の偏微分方程式についても，差分解法を用いて数値解を求められることが多い．ここでは，ファイナンスの分野に現れる **Black-Scholes 方程式**

$$\begin{cases} \dfrac{\partial Y}{\partial t} = -\dfrac{\sigma^2 S^2}{2} \cdot \dfrac{\partial^2 Y}{\partial S^2} - rS \cdot \dfrac{\partial Y}{\partial S} + rY \qquad (0 \leq t \leq T), \\ Y(T,S) = F(S) \end{cases}$$

の数値解法を考えてみよう．ここで，r は無リスク金利（正の定数），σ はボラティリティ—（株価の変動の大きさを表す正の定数）である．Black-Scholes 方程式は，満期 $t = T$ において，その時点での株価 S に応じた払い戻し $F(S)$ が受けられるような金融商品の，時刻 $t < T$ かつ株価が S のときの価格 $Y(t, S)$ が満たす偏微分方程式である．

K を正の定数とし，$F(S) = \max(S - K, 0)$ とする場合，すなわち，満期における株価 S が K を上回っていた場合には K と株価との差額の払い戻しが受けられ，株価 S が K を下回っていた場合には払い戻しが無いような金融商品を**コール・オプション**という．また，$F(S) = \max(K - S, 0)$ であるような金融商品を**プット・オプション**という．いずれの場合でも，K を権利行使価格という．

株価が 2 倍になれば日々の値動きの大きさも 2 倍になる，というように，一般的に株価は比率で動くので，Black-Scholes 方程式は価格そのもので見るよりも，価格の対数を見る方が自然である．そこで，$x = \log S$ とおいて，Y を x の関数として求めよう．

$$\frac{\partial Y}{\partial S} = \frac{\partial Y}{\partial x} \cdot \frac{dx}{dS} = \frac{1}{S} \cdot \frac{\partial Y}{\partial x}$$
$$\frac{\partial^2 Y}{\partial S^2} = \frac{\partial}{\partial S}\left(\frac{1}{S} \cdot \frac{\partial Y}{\partial x}\right) = -\frac{1}{S^2} \cdot \frac{\partial Y}{\partial x} + \frac{1}{S} \cdot \frac{\partial^2 Y}{\partial x^2} \cdot \frac{dx}{dS} = -\frac{1}{S^2} \cdot \frac{\partial Y}{\partial x} + \frac{1}{S^2} \cdot \frac{\partial^2 Y}{\partial x^2}$$

より，代入すると

$$\frac{\partial Y}{\partial t} = -\frac{\sigma^2}{2} \cdot \frac{\partial^2 Y}{\partial x^2} - \left(r - \frac{\sigma^2}{2}\right)\frac{\partial Y}{\partial x} + rY$$

が得られる．これを解くために，時間刻み Δt，価格刻み Δx を決め，等間隔に取った離散点を $(t^m,\ x_n)$ とし，$Y(t^m, x_n)$ の近似を Y_n^m とする．

まずは熱方程式の場合と同様，陽解法を考えよう．差分による近似として

$$\begin{aligned}
Y &\approx Y_n^{m+1}, \\
\frac{\partial Y}{\partial t} &\approx \frac{Y_n^{m+1} - Y_n^m}{\Delta t}, \\
\frac{\partial Y}{\partial x} &\approx \frac{Y_{n+1}^{m+1} - Y_{n-1}^{m+1}}{2\Delta x}, \\
\frac{\partial^2 Y}{\partial x^2} &\approx \frac{Y_{n+1}^{m+1} - 2Y_n^{m+1} + Y_{n-1}^{m+1}}{\Delta x^2},
\end{aligned}$$

を導入し，Black-Scholes 方程式に代入すると

$$\begin{aligned}
\frac{Y_n^{m+1} - Y_n^m}{\Delta t} = &-\frac{\sigma^2}{2} \cdot \frac{Y_{n+1}^{m+1} - 2Y_n^{m+1} + Y_{n-1}^{m+1}}{\Delta x^2} \\
&- \left(r - \frac{\sigma^2}{2}\right)\frac{Y_{n+1}^{m+1} - Y_{n-1}^{m+1}}{2\Delta x} + rY_n^{m+1}
\end{aligned}$$

となる．上の式を見ると，Y_n^m が $Y_{n-1}^{m+1}, Y_n^{m+1}, Y_{n+1}^{m+1}$ で書けるので，満期から時間を逆に計算して行くことで時刻 0 での値を求めることができる．時間を逆向きに計算するところが熱方程式との違いである．

Black-Scholes 方程式において x の範囲は全ての実数となるが，数値計算の際には，適当な区間に限定して計算を行う．計算する金融商品がコール・オプションやプット・オプションの場合は，権利行使価格を中心とし，適当な $R > 1$ を取って $\log K/R \leq x \leq \log KR$ とすることが多い．さらに，詳しい説明は省くが，境界条件は，コール・オプションの場合

$$Y_k^m = e^{r(T-t^m)} F(e^{x_k})$$

プット・オプションの場合

$$Y_k^m = e^{-r(T-t^m)} F(e^{x_k})$$

と取ればよいことがわかっている．ただし，k は境界の添え字である．

熱方程式の場合と同様に考えると，Black-Scholes 方程式の陽解法の安定性条件は

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{\sigma^2} + O(\Delta t)$$

であることがわかる．

Black-Scholes 方程式においても熱方程式と同様，陰解法やクランク・ニコルソン法を考えることができる．陰解法が無条件安定になること，クランク・ニコルソン法の誤差が時間刻みに対して二次精度である点も熱方程式の場合と同じである．

5.12 Black-Scholes 方程式の陽解法のプログラム

Black-Scholes 方程式を陽解法で解くプログラムを以下に示す．

```
#include <stdio.h>
#include <math.h>
#include "Linear.h"

#define M 1000 // 時間分割
#define MM 50  // 時間のファイル出力間隔
#define N 20   // 空間分割
#define NN 1   // 空間のファイル出力間隔
```

次ページへ続く…

```
// ペイオフ関数
double F(double S)
{
    double K = 100;
    if(S>K){
        return S-K;
    } else {
        return 0;
    }
}

int main()
{
// パラメータの設定
    double SL   = 50;           // 下限価格
    double SH   = 200;          // 上限価格
    double r    = 0.1;          // 無リスク金利
    double sig  = 0.2;          // ボラティリティ
    double T    = 2.0;          // 満期
    double dt   = T / M;        // 時間分割幅
    double ds   = log(SH/SL)/N; // log 価格分割幅

    printf("安定性条件チェック: dt=%f, ds=%f, sigma^2*dt/ds^2=%f\n",
                                            dt, ds, sig*sig*dt/ds/ds);

// 原資産価格
    Vector S(N);
    for(int n=0; n<=N; n++){
        S[n] = SL*exp(n*ds);
    }

    Vector Y(N);
    Vector Y2(N);
```

次ページへ続く…

```
    FILE *fp;
    fp = fopen("b-s.dat","w");

    // 満期のデータを出力
    for(int n=0; n<=N; n+=NN){
        Y[n] = F(S[n]);
        fprintf(fp,"%f %f %f\n", T, S[n], Y[n]);
    }
    fprintf(fp,"\n");

    // 係数は一定なので、あらかじめ求めておく
    double a = dt*sig*sig/ds/ds/2 - dt*(r-sig*sig/2)/2/ds;
    double b = 1 - dt*sig*sig/ds/ds - dt*r;
    double c = dt*sig*sig/ds/ds/2 + dt*(r-sig*sig/2)/2/ds;

    for(int m=M-1; m>=0; m--){
        double t = m*dt;
        double er = exp(r*(T-t));
        Y2[0] = er*F(S[0]);
        Y2[N] = er*F(S[N]);
        for(int n=1; n<N; n++){
            Y2[n] = a*Y[n-1] + b*Y[n] + c*Y[n+1];
        }
        Y = Y2.Copy();
        if(m%MM==0){
            for(int n=0; n<=N; n+=NN){
                fprintf(fp,"%f %f %f\n", t, S[n], Y[n]);
            }
            fprintf(fp,"\n");
        }
    }
    fclose(fp);

    return 0;
}
```

グラフは gnuplot で表示できる．Excel の「ワイヤーフレーム 3D 等高線」で表示する場合は，途中を

```
--------- ここまで同じ ---------
    FILE *fp;
    fp = fopen("b-s.csv","w");
    for(int n=0; n<=N; n+=NN){
        fprintf(fp,",%f", S[n]);
    }
    fprintf(fp,"\n");

    // 満期のデータを出力
    fprintf(fp,"%f", T);
    for(int n=0; n<=N; n+=NN){
        Y[n] = F(S[n]);
        fprintf(fp,",%f", Y[n]);
    }
    fprintf(fp,"\n");
--------- 中略 ---------
        if(m%MM==0){
            fprintf(fp,"%f", t);
            for(int n=0; n<=N; n+=NN){
                fprintf(fp,",%f", Y[n]);
            }
            fprintf(fp,"\n");
        }
--------- これ以降同じ ---------
```

と変更すればよい.

5.13 補足

差分法は偏微分方程式を解く上で最も基本的な解法である．今回は線形の方程式のみを取り扱ったが，例えば

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + u^2$$

のような非線形方程式も差分法で解くことができる．その場合，陰解法を適用すると非線形方程式になるので，毎時間ステップごとに多次元ニュートン法を用いることになる．

問題が線形で，右辺に時間変数が入らない場合，陰解法に現れる連立一次方程式の係数行列は時間ステップによらず毎回同じなので，そのような場合には，LU 分解などを用いる

ことで計算を効率化することができる．また，係数行列は一般的に疎行列となるので，その行列の特性に応じた連立一次方程式の解法を用いることで効率化を実現できる．特に今回述べたような空間 1 次元の問題の場合は，係数行列が 3 重対角行列となるので，3 重対角行列専用の方法を用いることができる．興味のある人は調べてもらいたい．

空間 2 次元以上の問題も差分法で解くことができるが，方程式を考える領域の形状が長方形でない場合には少々面倒である．領域が長方形ではない場合には，次章で紹介する有限要素法を用いることが多い．

5.14 課題

課題１ 初期条件を自由に定めて熱方程式の解を陽解法で計算し，解のグラフを表示しなさい．

課題２ 時間方向の計算区間を $[0,1]$，空間方向の計算区間を $[0,1]$ とし，初期条件を $u_0(x) = \sin \pi x$，境界条件を 0 とすると，$\lambda = 1$ のときの熱方程式の解は $u = e^{-\pi^2 t} \sin \pi x$ となる．この解を陰解法およびクランク・ニコルソン法で解いたときの最大誤差を，$(\Delta t, \Delta x) = (0.05, 0.0005), (0.1, 0.0005), (0.0005, 0.05), (0.0005, 0.1)$ の 4 通りについて調べ，誤差がおおよそ理論通りになっていることを確かめなさい（Δt による誤差と Δx による誤差を完全に分離することはできないので，おおよそで確かめられればよい）．

6　有限要素法

本章では，偏微分方程式の解法として，差分法とならんでよく用いられる**有限要素法**について紹介する．有限要素法は，空間の 2 次元以上の，長方形でないような領域で方程式を考える際に威力を発揮する．有限要素法は時間発展する方程式にも適用できるが，今回は簡単のため 2 次元の定常問題を考えることにする．

有限要素法の定式化や誤差評価には，関数解析の理論が大きな役割を果たしている．この資料では関数解析の知識は仮定していないので，適当な滑らかさがあれば，とか，関数の範囲を適当に制限すれば，等の曖昧な言い回しが多く出てくるが，これらは関数解析をベースに有限要素法を勉強すればクリアになる．興味のある人は是非勉強してもらいたい．

6.1　準備

2 次元領域上の偏微分方程式を扱うため，記号をいくつか導入する．まず，Ω は $\mathbb{R}^2$ の有界領域を表すとする．具体的には平面上の多角形や円の内部をイメージするとよい．ちなみに，数学で領域と言えば普通は開集合を指すので，Ω は境界を含まないものとする．また，Ω の境界を $\partial\Omega$ と表記する．

本資料にける関数は，特に断りがないかぎり実数の範囲で考えるものとする．また，下付きの添え字で偏微分を表すものとする．例えば

$$u_x = \frac{\partial u}{\partial x}, \qquad u_{xy} = \frac{\partial^2 u}{\partial x \partial y}$$

等である．

スカラー値関数 u に対し

$$\nabla u = \begin{pmatrix} u_x \\ u_y \end{pmatrix}, \qquad \Delta u = u_{xx} + u_{yy}$$

なる表記を用いる．ここで，∇ はナブラもしくはグラディエント，Δ はラプラシアンと読む．また，ベクトル値関数 $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ に対し

$$\mathrm{div}\ u = (u_1)_x + (u_2)_y$$

なる表記を用いる．div はダイバージェントと読む．

ベクトル値関数同士の積は，通常の意味でのベクトルの内積を意味するものとする．具体的には，ベクトル値関数 $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ と $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$ について

$$uv = u_1 v_1 + u_2 v_2$$

と定める．また，ベクトル値関数 u に対して，$|u|$ は通常の意味でのベクトルの長さを意味するものとする．具体的には，ベクトル値関数 $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ について

$$|u| = \sqrt{u_1^2 + u_2^2}$$

と定める．

6.2 ポアソン方程式

与えられた関数 f, g に対し

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = g & \text{on } \partial\Omega \end{cases}$$

を満たす u を求める問題を考える．この方程式は**ポアソン方程式**と呼ばれる．特に $f \equiv 0$ のときは**ラプラス方程式**という．この方程式は，熱伝導，薄膜の微小変形，流体の運動など，様々な現象の定式化に現れる．まずは，熱伝導および薄膜の微小変形について，どのようにポアソン方程式が現れるのかを説明する．

6.3 熱伝導

薄い板があり，板は発熱しているものとする．発熱量は単位時間，単位面積あたり f であるとする．f は位置の関数なので，発熱の仕方は場所によって異なっていても構わない．板の形状を Ω で表し，Ω の境界では温度が g になるように冷却されているものとする．g も場所によって異なっても構わない．状況としては，コンピュータの CPU が発熱し，周囲で冷却しているような状況をイメージするとよい．

板上の温度分布を u とする．仮定より，境界条件は $u = g$ on $\partial\Omega$ となる．ここで，板上の微小領域 $s = (x, x+h) \times (y, y+h)$ を考える．熱の移動量は熱伝導率および温度勾配に比例するので，単位時間あたりに線分 $x \times (y, y+h)$ を横切って s から流出する熱量は

$$\int_y^{y+h} \lambda u_x(x,t)dt$$

となる．ここで，λ は熱伝導率である．同じことを s の 4 辺で考えると，単位時間あたりに s から流出する熱量は

$$\begin{aligned} &\lambda \left(\int_y^{y+h} u_x(x,t)dt - \int_y^{y+h} u_x(x+h,t)dt + \int_x^{x+h} u_y(t,y)dt - \int_x^{x+h} u_y(t,y+h)dt \right) \\ &= -\lambda \left(\int_y^{y+h} \Big(u_x(x+h,t) - u_x(x,t) \Big) dt + \int_x^{x+h} \Big(u_y(t,y+h) - u_y(t,y) \Big) dt \right) \end{aligned}$$

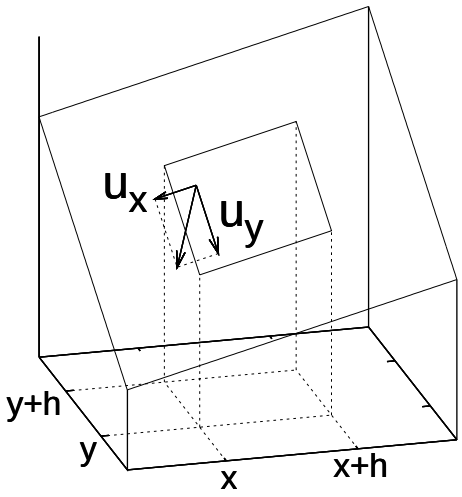

図 8: 温度分布と熱の流入の様子

となる．さて，長い時間が経過し，熱分布が定常状態に落ち着いた場合を考える．このとき，s から流出する熱量と s における発熱量が釣り合うので，

$$-\lambda\left(\frac{1}{h}\int_y^{y+h}\frac{u_x(x+h,t)-u_x(x,t)}{h}dt+\frac{1}{h}\int_x^{x+h}\frac{u_y(t,y+h)-u_y(t,y)}{h}dt\right)$$
$$=\frac{1}{h^2}\int_x^{x+h}\int_y^{y+h}f(t_1,t_2)\,dt_1dt_2$$

が成り立つ．ここで，u および f が十分に滑らかならば，テイラー展開

$$u_x(x+h,t)-u_x(x,t)=hu_{xx}(x,y)+O(h^2)$$
$$u_y(t,y+h)-u_y(t,y)=hu_{yy}(x,y)+O(h^2)$$
$$f(t_1,t_2)=f(x,y)+O(h)$$

を用いて $h\to 0$ とすることで

$$-\Delta u=\frac{f}{\lambda}$$

が得られる．

6.4 薄膜の微小変形

枠に固定された薄い膜（例えばゴム膜）があり，上向きに f の力がかかっているとする．ここでは，薄膜の縦方向の変形がごく微小な場合を考える．薄膜の縦方向の変位を u で表し，枠の上では関数 g の値で固定されているものとする．よって，境界条件は $u=g\ \ \text{on}\ \partial\Omega$ となる．

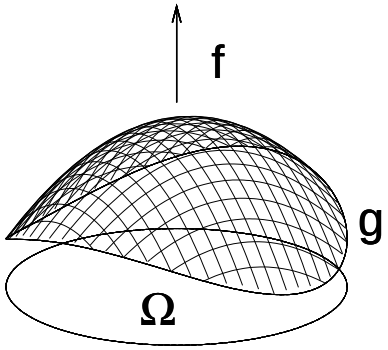

図 9: 薄膜の変形（変形の大きさは誇張して描いてある）

さて，薄膜の密度を m，厚さを d，ヤング率を Y とし，薄膜を，長さ h，バネ定数 Yd のバネが格子状に繋がり，バネの節点には重さ h^2dm の重りがついているような構造で近似する（この近似が良いかどうかは議論がある，というか本当は良くないのだが，ここでは細かいことは気にしない）．ここで，簡単のため重力は零であり，バネの自然長は零であるとする（つまり，膜は大きく引き伸ばした状態で枠に固定されているとしている）．
水平方向の変位は無視できるとすると，(x, y) の位置にある重りと $(x + h, y)$ の位置にある重りとの間に働く張力 t は

$$t = Yd\sqrt{h^2 + \Big(u(x, y) - u(x + h, y)\Big)^2}$$

となる．よって，この張力 t によって (x, y) の位置にある重りに下向きにかかる力を s とすると

$$\begin{aligned} s &= Yd\sqrt{h^2 + \Big(u(x, y) - u(x + h, y)\Big)^2}\,\frac{u(x, y) - u(x + h, y)}{\sqrt{h^2 + \Big(u(x, y) - u(x + h, y)\Big)^2}} \\ &= Yd\Big(u(x, y) - u(x + h, y)\Big) \end{aligned}$$

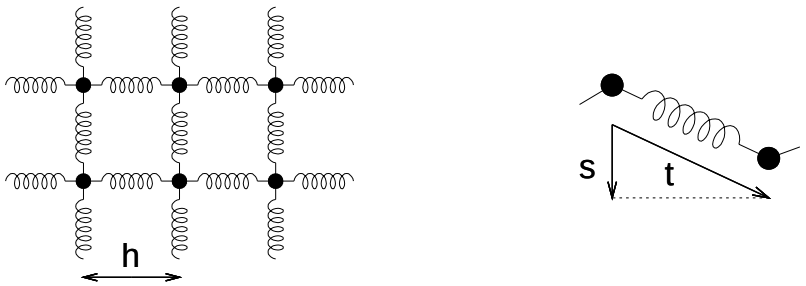

図 10: バネと重りによる近似

となる．同様にして，他の重りによって引っ張られる力も考え，それが上向きにかかっている力と釣り合うことから

$$Yd\Big(4u(x,y)-u(x+h,y)-u(x-h,y)-u(x,y+h)-u(x,y-h)\Big)=fh^2$$

すなわち

$$u(x+h,y)+u(x-h,y)+u(x,y+h)+u(x,y-h)-4u(x,y)=-\frac{f}{Yd}h^2$$

が成り立つ．ここで，u が十分に滑らかならば，テイラー展開

$$u(x+h,y)=u(x,y)+hu_x(x,y)+\frac{h^2}{2}u_{xx}(x,y)+O(h^3)$$

等を用いて $h\to 0$ とすることで

$$-\Delta u=\frac{f}{Yd}$$

が得られる．

6.5 変分原理

物理現象において，釣り合いの状態は，何らかの物理量の極小状態と対応していることが多い．これを**変分原理**という（正確な定式化等は解析力学の教科書を参照すること）．例えば，バネの一方が天井に固定され，もう一方に重りが吊るされているようなモデルを考える．
重りの重さを m，重力加速度を g，バネ定数を k，バネの伸びを x とすると，力の釣り合いから

$$kx=mg$$

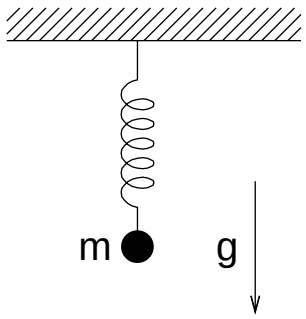

図 11: 釣り合いの位置

が成り立つ．また，バネの弾性エネルギーと重りの位置エネルギーの和を E とすると

$$E = \frac{1}{2}kx^2 - mgx$$

と表される．このとき，力の釣り合いからも，エネルギーの最小化からも，

$$x = \frac{mg}{k}$$

が求められる．

前節で考えた，薄膜をバネと重りの集合で近似するモデルについても全エネルギーを求めてみよう．そのため，$E(x,y)$ を，(x,y) の位置にある重りの位置エネルギーに，その重りの右と上につながるバネの弾性エネルギーを足したものとする．この $E(x,y)$ を足し合わせれば全エネルギーが求められる．ニュートンの運動法則から，重りに単位面積あたり f の力がかかっているということは，$f/(dm)$ の加速度がかかっていることと同じなので，

$$\begin{aligned}
E(x,y) &= \frac{1}{2}Yd\left(h^2 + \Big(u(x,y) - u(x+h,y)\Big)^2\right) \\
&\qquad + \frac{1}{2}Yd\left(h^2 + \Big(u(x,y) - u(x,y+h)\Big)^2\right) - h^2 dm \cdot \frac{f(x,y)}{dm}u(x,y) \\
&= Ydh^2 + \frac{1}{2}Ydh^2\left(\frac{u(x+h,y) - u(x,y)}{h}\right)^2 \\
&\qquad + \frac{1}{2}Ydh^2\left(\frac{u(x,y+h) - u(x,y)}{h}\right)^2 - h^2 f(x,y)u(x,y)
\end{aligned}$$

となり，これまでと同様，テイラー展開を用いると

$$E(x,y) = Ydh^2 + \frac{1}{2}Ydh^2|\nabla u(x,y)|^2 - h^2 f(x,y)u(x,y) + O(h^3)$$

が成り立つ．よって，これを足し合わせて $h \to 0$ として積分の定義を用いると，全エネルギーは

$$Yd|\Omega| + \frac{1}{2}Yd\int_\Omega |\nabla u(x,y)|^2 dxdy - \int_\Omega f(x,y)u(x,y)\,dxdy$$

となることがわかる．よって，変分原理によれば，境界上で g と一致する関数の中で

$$-\Delta u = \frac{f}{Yd}$$

を満たす u を求める問題と，汎関数

$$\frac{1}{2}\int_\Omega |\nabla u|^2 dxdy - \int_\Omega \frac{fu}{Yd}\,dxdy$$

の最小値を与える u を求める問題は同値であると期待できる．ここで，汎関数とは，関数に対してスカラーを返すような写像を意味する（普通の関数は，スカラーもしくはベクトルに対してスカラーを返す）．

さて，$\dfrac{f}{Yd}$ を改めて f と置くと，

$$-\Delta u = f$$

を満たす u を求める問題と

$$E(u) = \frac{1}{2}\int_\Omega |\nabla u|^2 dxdy - \int_\Omega fu\, dxdy$$

の最小値を与える u を求める問題が同値であると期待できる．

ポアソン方程式の解が汎関数 $E(u)$ の最小値を与える関数に一致するということは，19 世紀前半頃には知られていたようだが，厳密な証明が得られたのは 19 世紀末になってからである．現代では，関数解析の理論を用いて示すことができる．そもそも，汎関数においては，最小値の存在すら自明ではない．汎関数が下に有界なら最小値があるのは当たり前だと思うかもしれないが，これはそれほど明らかな話ではないのである．

例えば C を，$u(0) = u(1) = 0$ を満たすような一変数連続関数の集合とすると，汎関数

$$G(u) = \int_0^1 \big(u(x) - 1\big)^2 dx$$

は C の中では最小値を取らない（証明は考えてみよう）．

詳しい説明は省くが，領域 Ω や f や g にある程度の滑らかさを仮定し，関数 u の範囲を適当に定めれば，ポアソン方程式の解は汎関数 $E(u)$ の最小値を与える関数に一致することが示される．

さてここで，汎関数 $E(u)$ を最小にする u がどのような性質を持つかを考えてみよう．そこで，適当な滑らかさを持ち，$\partial\Omega$ で零となる関数の集合を H_0 とし，$\varphi \in H_0$ について $E(u + \lambda\varphi)$ を考えてみると，

$$\begin{aligned} E(u+\lambda\varphi) &= \frac{1}{2}\int_\Omega |\nabla(u+\lambda\varphi)|^2 dxdy - \int_\Omega f\ (u+\lambda\varphi) dxdy \\ &= \frac{1}{2}\int_\Omega |\nabla u|^2 + \lambda\int_\Omega \nabla u\nabla\varphi\, dxdy + \frac{\lambda^2}{2}\int_\Omega |\nabla\varphi|^2 dxdy \\ &\qquad - \int_\Omega fu\, dxdy - \lambda\int_\Omega f\varphi\, dxdy \end{aligned}$$

となる．ここで上式を λ の二次式と見ると，$E(u + \lambda\varphi)$ は $\lambda = 0$ で最小にならねばならないので，λ の一次の項は零でなければならない．すなわち

$$\int_\Omega \nabla u\nabla\varphi\, dxdy = \int_\Omega f\varphi\, dxdy \qquad \text{for} \quad \forall\varphi \in H_0$$

が成り立つ．このような条件を満たす u をポアソン方程式の**弱解**という．また，弱解に出てくる $\varphi \in H_0$ を試験関数という．

もともとポアソン方程式は

$$-\Delta u = f$$

という形で 2 階微分が入っている．それに対し，弱解は 1 階微分が存在すればいいので，弱解の方が条件が緩い．そういう意味で弱解という名称が使われている．

方程式に現れる微分の階数だけ連続微分可能で，かつ境界までこめて連続な解を**古典解**という．ポアソン方程式の場合は C^2 級かつ境界までこめて連続な解が古典解となる．一般的に，方程式の古典解は弱解になることが多いが，逆に弱解が古典解になるかどうかは状況による．方程式によっては，弱解しか存在しないこともある．しかし，古典的な意味では元の方程式を満たさないとしても，弱解には物理的に重要な意味があることが多いし，古典解を考えるよりも弱解を考える方が自然な場合も多い．

例えば，$\Omega = (0,1)^2$ において波動方程式

$$\begin{cases} u_{xx} - u_{yy} = 0 & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

を考える．このとき，$h(t)$ を周期 1 の周期関数とすると

$$u(x,y) = h(x+y) - h(x-y) - h(-x+y) + h(-x-y)$$

が波動方程式を満たすことは容易に確かめられる．このとき，h が滑らかならば u は古典解になるが，必ずしも h が滑らかでない場合にも u を解として扱いたいというのは自然な要求である．そのような場合には，弱解を考えれば，必ずしも 2 回微分できないような関数も解として考えることができる（h が不連続点を持つような関数の場合は，さらに解の範囲を広げなければならないが）．

6.6 ガウスの発散定理とグリーンの公式

今後，多用するので，ここで**ガウスの発散定理**と**グリーンの公式**について説明しておく．関数は十分に滑らかであるとする．

ガウスの発散定理　ベクトル値関数 u について

$$\int_\Omega \operatorname{div} u \, dxdy = \int_{\partial\Omega} u \cdot n \, ds$$

が成り立つ．ここで n は外向き単位法線ベクトル，ds は $\partial\Omega$ 上の線要素である．つまり，右辺は境界上における u と n の内積の積分を表す．

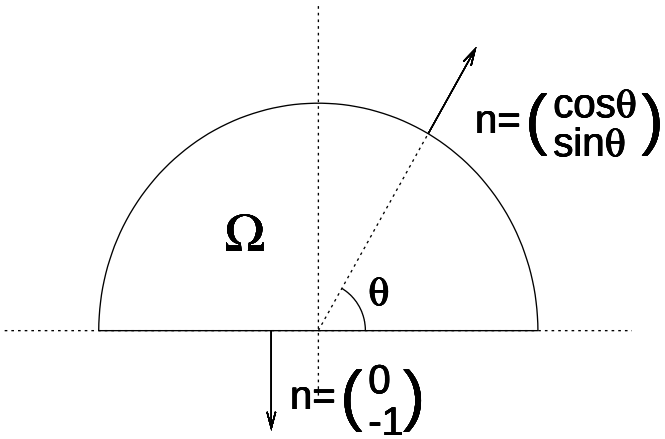

図 12: 外向き単位法線方向ベクトル

例えば，$(x, y) = (r\cos\theta, r\sin\theta)$ とし，

$$\Omega = \left\{ (x, y) \;\middle|\; r < 2,\ y > 0 \right\}, \quad u = \begin{pmatrix} x^2 \\ y^2 \end{pmatrix}$$

とすると

$$\int_\Omega \mathrm{div}\ u\, dxdy = \int_\Omega (2x + 2y)\, dxdy = 2\int_0^\pi \int_0^2 (r\cos\theta + r\sin\theta) r\, drd\theta = \frac{32}{3}$$

となり，一方で，$\partial\Omega$ のうち円弧部分を γ_1，直線部分を γ_2 とすると

$$\begin{aligned} \int_{\partial\Omega} u \cdot n\, ds &= \int_{\gamma_1} \begin{pmatrix} x^2 \\ y^2 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} ds + \int_{\gamma_2} \begin{pmatrix} x^2 \\ y^2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix} ds \\ &= \int_0^\pi \begin{pmatrix} 2^2\cos^2\theta \\ 2^2\sin^2\theta \end{pmatrix} \cdot \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} 2\, d\theta + \int_{-2}^2 \begin{pmatrix} x^2 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix} dx \\ &= 8\int_0^\pi (\cos^3\theta + \sin^3\theta)\, d\theta + 0 = \frac{32}{3} \end{aligned}$$

となり，確かに成り立っていることがわかる．ただし，線要素 ds を通常の積分変数で置き換える際には注意を要する．s を境界上の移動距離に比例して増加する変数とすると，γ_1 上では $s = 2\theta$ となるので，$ds = 2d\theta$ となる．
ガウスの発散定理の証明についてはベクトル解析の教科書で勉強してもらいたい．ここでは，ガウスの発散定理の直感的な意味について説明する．

平面上を水が流れているものとし，(x, y) における流速を $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ で表す．また，平面からは水が湧き出しているものとする（底から地下水が滲み出してくるイメージ）．さて，

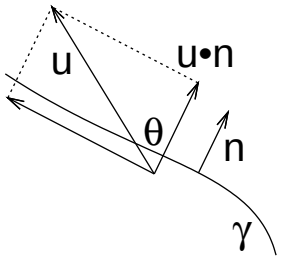

図 13: 境界を横切る流量

境界 γ を横切って流出する流量は，外向き単位法線ベクトル n を用いて，単位長さ単位時間あたり

$$|u|\cos\theta = |u||n|\cos\theta = u \cdot n$$

と表される．よって，Ω から流出する流量の合計は単位時間あたり

$$\int_{\partial\Omega} u \cdot n \, ds$$

となり，これは Ω 全体における単位時間あたりの湧き出し量と等しい．一方，Ω 内の微小領域 $(x, x+h) \times (y, y+h)$ から流出する流量は

$$\begin{aligned}
&\int_y^{y+h} u_1(x+h,t)dt - \int_y^{y+h} u_1(x,t)dt + \int_x^{x+h} u_2(t,y+h)dt - \int_x^{x+h} u_2(t,y)dt \\
&= h\left(\int_y^{y+h} \frac{u_1(x+h,t)-u_1(x,t)}{h}dt + \int_x^{x+h} \frac{u_2(t,y+h)-u_2(t,y)}{h}dt\right) \\
&= h^2 \mathrm{div}\ u + O(h^3)
\end{aligned}$$

となるので，これはこの微小領域における湧き出し量と等しい．これを Ω 全体で足し合わせると全体の湧き出し量になるので，$h \to 0$ の極限を考えると

$$\int_\Omega \mathrm{div}\ u \, dxdy = \int_{\partial\Omega} u \cdot n \, ds$$

が成り立つ．

ガウスの発散定理から，次のグリーンの公式が成り立つ．

グリーンの公式　スカラー値関数 u およびベクトル値関数 v について

$$\int_\Omega \nabla u \cdot v \, dxdy = \int_{\partial\Omega} uv \cdot n \, ds - \int_\Omega u \, \mathrm{div}\ v \, dxdy$$

が成り立つ．

グリーンの公式は，$v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$ と置くと

$$\mathrm{div}(uv) = (uv_1)_x + (uv_2)_y = u_x v_1 + u\,(v_1)_x + u_y v_2 + u\,(v_2)_y = \nabla u \cdot v + u\,\mathrm{div}\ v$$

となるので，左辺にガウスの発散定理を適用することにより示すことができる．

グリーンの公式は一変数関数の部分積分の拡張になっている．実際，Ω を長方形 $(a,b)\times(0,1)$ とし，u と $v = \begin{pmatrix} w \\ 0 \end{pmatrix}$ が x だけの関数とすると，

$$\begin{aligned}
&\int_\Omega \nabla u \cdot v\, dxdy = \int_0^1 \int_a^b u_x w\, dxdy = \int_a^b u_x w\, dx \\
&\int_{\partial\Omega} uv \cdot n\, ds = \int_0^1 u(b)v_1(b)\, dy - \int_0^1 u(a)v_1(a)\, dy = [uw]_a^b \\
&\int_\Omega u\,\mathrm{div}\ v\, dxdy = \int_0^1 \int_a^b u w_x\, dxdy = \int_a^b u w_x\, dx
\end{aligned}$$

より，グリーンの公式は一変数関数の部分積分の公式と一致する．また，グリーンの公式自体も部分積分と呼ぶことがある．

6.7 弱定式化

グリーンの公式の応用として，ポアソン方程式の古典解が弱解になることを示そう．u をポアソン方程式

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = g & \text{on } \partial\Omega \end{cases}$$

の古典解とする．ここで，第一式に任意の $\varphi \in H_0$ をかけて Ω で積分すると

$$-\int_\Omega \Delta u \varphi\, dxdy = \int_\Omega f\varphi\, dxdy$$

となる．ここで，部分積分（グリーンの公式）を適用すると

$$-\int_\Omega \Delta u \varphi\, dxdy = -\int_{\partial\Omega} \varphi \nabla u \cdot n\, ds + \int_\Omega \nabla u \nabla\varphi\, dxdy$$

が成り立つが，$\varphi \in H_0$ より φ が $\partial\Omega$ で零となることから

$$\int_\Omega \nabla u \nabla\varphi\, dxdy = \int_\Omega f\varphi\, dxdy \qquad \text{for} \qquad \forall\varphi \in H_0$$

が得られる．よって u は弱解になる．

このように，部分積分（グリーンの公式）を利用して弱解が満たす条件式を求めることを，**弱定式化**という．

6.8　リッツ法とガレルキン法

本節以降では，簡単のため境界条件は零とする．すなわち，有界領域 Ω と関数 f について，次のポアソン方程式を考える．

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

特殊な場合を除くと，この方程式は数学的には解けないので，解の様子を知りたい場合には数値計算が用いられることが多い．

さて，この方程式の近似解を求めたい場合に，近似解を有限個の関数 $\varphi_k \in H_0,\ k = 1, 2, \cdots, n$ の一次結合

$$u_h = \sum_{k=1}^{n} c_k \varphi_k$$

で近似するという発想は自然である．このような方法を有限要素法といい，得られた近似解を**有限要素解**という．また，関数 $\varphi_k\ (k = 1, 2, \cdots n)$ を**有限要素基底**という．有限要素法においては，係数 $\{c_k\}$ をどのように決めるべきかが問題となる．係数を決める代表的な方法としては，**リッツ法**と**ガレルキン法**がある．これらについて順に説明する．

まず，ポアソン方程式の解 u は汎関数

$$E(u) = \frac{1}{2}\int_\Omega |\nabla u|^2 dxdy - \int_\Omega fu\, dxdy$$

の最小値を与える関数として特徴づけられたことを思い出そう．よって，$\{\varphi_k\}$ の一次結合全体で表される関数の集合を S_h とし，S_h の中で $E(u)$ を最小にするものを近似解とすることにする．そこで，任意の $\varphi \in S_h$ を取り，以前と同様に $E(u_h + \lambda\varphi)$ を考えてみると，$\lambda = 0$ で最小値を取ることから

$$\int_\Omega \nabla u_h \nabla\varphi\, dxdy = \int_\Omega f\varphi\, dxdy \qquad \text{for} \qquad \forall\varphi \in S_h$$

が得られる．上式は，$\varphi = \varphi_j,\ (j = 1, 2, \cdots n)$ で成り立てば任意の $\varphi \in S_h$ で成り立つので，近似解を求めるための条件は

$$\int_\Omega \nabla u_h \nabla\varphi_j\, dxdy = \int_\Omega f\varphi_j\, dxdy \qquad j = 1, 2, \cdots, n$$

となる．さらに

$$u_h = \sum_{k=1}^{n} c_k \varphi_k$$

を代入すると，条件は

$$\sum_{k=1}^{n} c_k \int_\Omega \nabla\varphi_k \nabla\varphi_j \, dxdy = \int_\Omega f\varphi_j \, dxdy \qquad j = 1, 2, \cdots, n$$

となる．これは n 次元連立一次方程式になり，

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \qquad a_{jk} = \int_\Omega \nabla\varphi_k \nabla\varphi_j \, dxdy,$$

$$c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \qquad b_j = \int_\Omega f\varphi_j \, dxdy$$

と置くと

$$Ac = b$$

と書ける．このとき，行列 A を**剛性行列**，ベクトル b を**外力ベクトル**という．このように，方程式の解がある汎関数を最小化するときに，有限次元空間の中からその汎関数を最小化するものを探して近似解とする方法をリッツ法という．

さて，ここで，近似解を求めるもう一つのアプローチについて説明する．近似解を有限個の関数 $\varphi_j \in H_0,\ j = 1, 2, \cdots, n$ の一次結合

$$u_h = \sum_{k=1}^{n} c_k \varphi_k$$

で表すのはリッツ法と同様だが，今度は，近似解 u_h を，厳密解との誤差を最小にするものとして求める．誤差の測り方には色々あるが，ここでは

$$\int_\Omega |\nabla(u - u_h)|^2 dxdy$$

を最小にするように u_h を定める．このとき，任意の $\varphi \in S_h$ を取ると

$$\int_\Omega |\nabla(u - u_h + \lambda\varphi)|^2 dxdy = \int_\Omega |\nabla(u - u_h)|^2 dxdy + 2\lambda \int_\Omega \nabla(u - u_h)\nabla\varphi \, dxdy + \lambda^2 \int_\Omega |\nabla\varphi|^2 dxdy$$

が $\lambda = 0$ で最小になることから

$$\int_\Omega \nabla(u - u_h)\nabla\varphi \, dxdy = 0 \qquad \text{for} \qquad \forall\varphi \in S_h$$

が成り立つ．ここで，弱解が満たす式を用いると，$\varphi \in S_h$ のとき $\varphi \in H_0$ より

$$\int_\Omega \nabla u_h \nabla \varphi \, dxdy = \int_\Omega \nabla u \nabla \varphi \, dxdy = \int_\Omega f\varphi \, dxdy \qquad \text{for} \qquad \forall \varphi \in S_h$$

となり，リッツ法の手順と一致する．このように，誤差を最小化するものとして近似解を求める方法をガレルキン法という．

今回の問題については，リッツ法とガレルキン法は一致するので，合わせてリッツ・ガレルキン法と言う．しかし一般的には，解が何らかの汎関数を最小にするとは限らないので，リッツ法は常に適用可能ではないし，ガレルキン法においても，どのように誤差を測るかによって求まる近似解は異なるので，一般的にはリッツ法とガレルキン法が一致するとは限らない．

6.9 有限要素基底

前節において，有限要素基底 $\varphi_j \ (j = 1, 2, \cdots n)$ については，境界で零になるという以外に特に制限は無かった．そういう意味では何でもよいのだが，実際には計算を簡便にするため，領域 Ω を多数の三角形に分割し，その上で連続な区分一次関数を考えることが多い．具体的には，まず領域 Ω を多数の三角形に分割し，Ω の内部にある節点を $p_1, p_2, \cdots, p_n$ とする．また，境界上の節点を $p_{n+1}, p_{n+2}, \cdots$ とする．そして，p_j で 1 になり，他の節点では零になり，各々の三角形上では一次関数となるような連続関数を φ_j とする．ひとまずは有限要素基底には境界上で零になるという制限があるので，$\varphi_j \ (j = 1, 2, \cdots n)$ を用いるが，境界条件によっては φ_{n+1} 以降も基底として用いることがある．また，$\varphi_j \ (j = 1, 2, \cdots n)$ の一次結合全体で表される関数の集合を S_h とする．

分割されたそれぞれの三角形を有限要素，もしくは単に要素という．有限要素分割には三角形分割の他にも長方形分割などもあるが，三角形による分割であることを強調したい場合には三角形要素などともいう．また，境界がカーブしている場合には，まず領域を多角形で近似してから三角形分割を行うものとする．

有限要素基底を用いて有限要素解

$$u_h = \sum_{k=1}^{n} c_k \varphi_k$$

が得られたとすると，$u_h(p_k) = c_k$ となるので，係数 c_k そのものが $u(p_k)$ の近似になっている．

以下は三角形による領域分割と有限要素基底の例である.

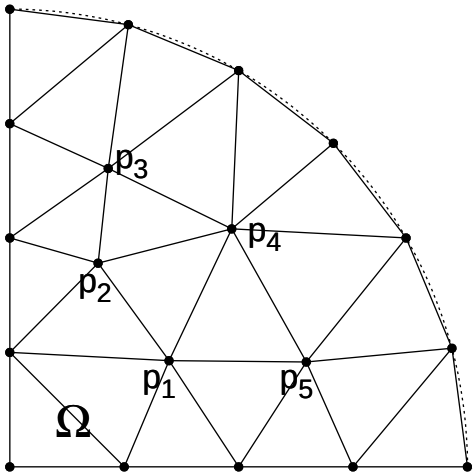

図 14: 三角形による分割

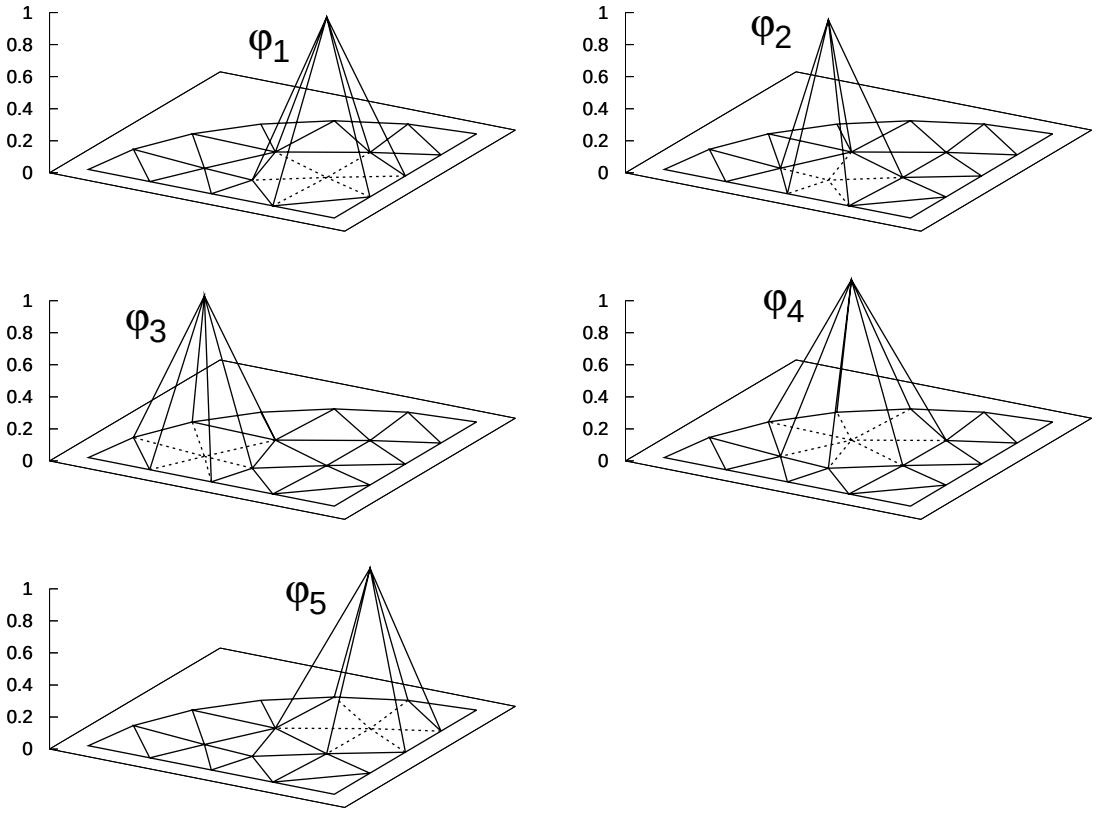

図 15: 有限要素基底

6.10　節点一覧表および要素節点対応表

有限要素法では，三角形分割のデータを扱うために，節点一覧表と要素節点対応表を用いる．例えば，$\Omega = (0,1) \times (0,1/2)$ とし，Ω を 16 個の三角形に分割し，図 16，図 17 のように三角形分割して節点および要素に番号を振ったとする．
節点番号は，内点から番号を振り，その後に境界上の節点に番号を振るとプログラムが簡単になる．ただし実際は，使い易さの点から，内点も境界上の節点も区別せずに番号を振っていくことが多い．その場合のアルゴリズムについては後で述べる．

節点ごとに，その座標と節点の属性（0:内点，1:境界上の点）を記したものが節点一覧表である．また，三角形要素ごとに，頂点の節点番号を記したものが要素節点対応表である．図 16，図 17 で例に挙げた三角形分割の場合，節点一覧表と要素節点対応表はそれぞれ，表 1，表 2 のようになる．要素節点対応表に記す頂点は，反時計回りに記しておくものとする．

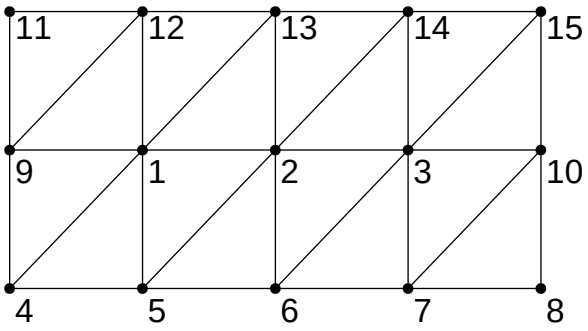

図 16: 節点番号

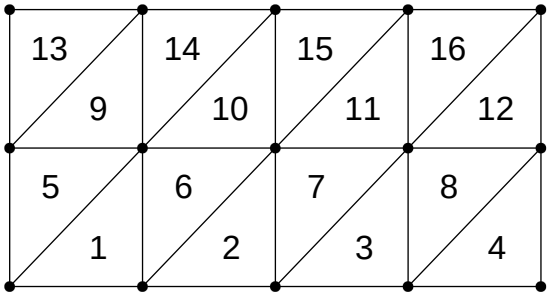

図 17: 要素番号

表 1: 節点一覧表

節点番号	x 座標	y 座標	属性
1	0.25	0.25	0
2	0.50	0.25	0
3	0.75	0.25	0
4	0.00	0.00	1
5	0.25	0.00	1
6	0.50	0.00	1
7	0.75	0.00	1
8	1.00	0.00	1
9	0.00	0.25	1
10	1.00	0.25	1
11	0.00	0.50	1
12	0.25	0.50	1
13	0.50	0.50	1
14	0.75	0.50	1
15	1.00	0.50	1

表 2: 要素節点対応表

要素番号	頂点 1	頂点 2	頂点 3
1	4	5	1
2	5	6	2
3	6	7	3
4	7	8	10
5	1	9	4
6	2	1	5
7	3	2	6
8	10	3	7
9	9	1	12
10	1	2	13
11	2	3	14
12	3	10	15
13	12	11	9
14	13	12	1
15	14	13	2
16	15	14	3

6.11　剛性行列と外力ベクトルの計算

ここでは，剛性行列

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \qquad a_{jk} = \int_\Omega \nabla\varphi_k \nabla\varphi_j \, dxdy$$

および外力ベクトル

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \qquad b_j = \int_\Omega f\varphi_j \, dxdy$$

を節点一覧表と要素節点対応表から計算する方法について述べる．本節以降，分割された三角形要素を τ_l $(l = 1, 2, \cdots, m)$ とする．

剛性行列 A や外力ベクトル b を求める際に，a_{jk} や b_j を順に求めるというのは実はあまり効率的ではない．例えば，a_{jk} を求めるためには p_j と p_k を両方含む三角形要素を検索しなければならず，手間がかかる．それよりも，各要素ごとの足し合わせで A や b を求める方法がよく用いられている．具体的には，

$$A^{(l)} = \begin{pmatrix} a_{11}^{(l)} & a_{12}^{(l)} & \cdots & a_{1n}^{(l)} \\ a_{21}^{(l)} & a_{22}^{(l)} & \cdots & a_{2n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}^{(l)} & a_{n2}^{(l)} & \cdots & a_{nn}^{(l)} \end{pmatrix}, \qquad a_{jk}^{(l)} = \int_{\tau_l} \nabla\varphi_k \nabla\varphi_j \, dxdy$$

$$b^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_n^{(l)} \end{pmatrix}, \qquad b_j^{(l)} = \int_{\tau_l} f\varphi_j \, dxdy$$

として

$$A = \sum_{l=1}^{m} A^{(l)}, \qquad b = \sum_{l=1}^{m} b^{(l)}$$

により A と b を求める．かえって複雑になったように思うかもしれないが，実はこうすることで効率的に計算を行うことができる．実際の手順としては次のようになる．

1. A および b を零行列および零ベクトルで初期化する．
2. l を順に 1 から m まで動かす．

3. 要素節点対応表により τ_l の 3 つの頂点の節点番号を求め，それを s_1, s_2, s_3 とする．
4. $a^{(l)}_{s_1s_1},\ a^{(l)}_{s_1s_2},\ a^{(l)}_{s_1s_3},\ a^{(l)}_{s_2s_1},\ a^{(l)}_{s_2s_2},\ a^{(l)}_{s_2s_3},\ a^{(l)}_{s_3s_1},\ a^{(l)}_{s_3s_2},\ a^{(l)}_{s_3s_3},$ の 9 つを計算し，A に足し込む（$a^{(l)}_{s_1s_2} = a^{(l)}_{s_2s_1},\ a^{(l)}_{s_2s_3} = a^{(l)}_{s_3s_2},\ a^{(l)}_{s_3s_1} = a^{(l)}_{s_1s_3}$ なので，実際に計算するのは 6 つ）．
5. $b^{(l)}_{s_1},\ b^{(l)}_{s_2},\ b^{(l)}_{s_3}$ の 3 つを計算し，b に足し込む．

ただし，4. と 5. については s_1, s_2, s_3 のうち境界上の節点については適用しない．

それでは具体的に $a^{(l)}_{s_1s_1}, a^{(l)}_{s_1s_2}, \cdots$ を求めてみよう．節点一覧表より座標を求め，$p_{s_1} = (x_1, y_1),\ p_{s_2} = (x_2, y_2),\ p_{s_3} = (x_3, y_3)$ とする．φ_{s_1} は τ_l 上では一次関数なので

$$\varphi_{s_1} = a_1 x + b_1 y + c_1$$

と書ける．

ここで，φ_{s_1} は p_{s_1} で 1，p_{s_2} と p_{s_3} では 0 となるので

$$\begin{aligned} a_1 x_1 + b_1 y_1 + c_1 &= 1 \\ a_1 x_2 + b_1 y_2 + c_1 &= 0 \\ a_1 x_3 + b_1 y_3 + c_1 &= 0 \end{aligned}$$

となる．ここで

$$P = \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix}$$

と置くと，クラメールの公式より

$$P^{-1} = \frac{1}{|P|} \begin{pmatrix} y_2 - y_3 & y_3 - y_1 & y_1 - y_2 \\ x_3 - x_2 & x_1 - x_3 & x_2 - x_1 \\ x_2 y_3 - x_3 y_2 & x_3 y_1 - x_1 y_3 & x_1 y_2 - x_2 y_1 \end{pmatrix}$$

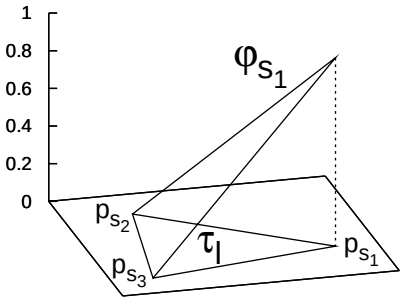

図 18: φ_{s_1} の様子

となるので

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} = P^{-1} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

より a_1 と b_1 が求まる．具体的には

$$\begin{pmatrix} a_1 \\ b_1 \end{pmatrix} = \frac{1}{|P|} \begin{pmatrix} y_2 - y_3 \\ x_3 - x_2 \end{pmatrix}$$

となる．ここで，τ_l の面積を $|\tau_l|$ で表すと，$p_{s_1}, p_{s_2}, p_{s_3}$ が反時計回りに並んでいれば

$$|P| = 2|\tau_l|$$

が成り立つので，結局，

$$a_1 = \frac{y_2 - y_3}{2|\tau_l|}, \qquad b_1 = \frac{x_3 - x_2}{2|\tau_l|}$$

が成り立つ．同様に

$$a_2 = \frac{y_3 - y_1}{2|\tau_l|}, \qquad b_2 = \frac{x_1 - x_3}{2|\tau_l|}$$
$$a_3 = \frac{y_1 - y_2}{2|\tau_l|}, \qquad b_3 = \frac{x_2 - x_1}{2|\tau_l|}$$

が成り立つ．これらを用いることにより，

$$a^{(l)}_{s_1 s_1} = \int_{\tau_l} \nabla\varphi_{s_1} \nabla\varphi_{s_1} \, dxdy = |\tau_l|(a_1^2 + b_1^2)$$
$$a^{(l)}_{s_1 s_2} = \int_{\tau_l} \nabla\varphi_{s_1} \nabla\varphi_{s_2} \, dxdy = |\tau_l|(a_1 a_2 + b_1 b_2)$$
$$\cdots\cdots\cdots$$

等と計算することができる．

図 16，図 17，表 1，表 2 で定められる三角形分割について，剛性行列を求めると，以下のようになる．

$$A = \begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix}$$

プログラムを組む前に，まずは手計算で，これと同じ結果になるか確かめておくとよい．

次に，外力ベクトルを求めるためには $b^{(l)}_j = \int_{\tau_l} f\varphi_j \, dxdy$ を計算する必要がある．しかし，f が単純な関数である場合を除くと正確に値を求めることは難しい．数値積分を用いるの

も一つの方法であるが，それよりも f を区分一次関数で近似して外力ベクトルを求める方法がよく用いられる．具体的には，f の p_k での値を f_k とし，f を φ_k（境界上の基底も含む）の一次結合で

$$f \approx \sum_k f_k \varphi_k$$

と近似して

$$b_j^{(l)} = \int_{\tau_l} f\varphi_j \, dxdy \approx \sum_k f_k \int_{\tau_l} \varphi_k \varphi_j \, dxdy$$

と計算する．このためには，τ_l の 3 つの頂点の節点番号を s_1, s_2, s_3 としたとき，

$$\int_{\tau_l} \varphi_{s_1}\varphi_{s_1} \, dxdy, \ \int_{\tau_l} \varphi_{s_1}\varphi_{s_2} \, dxdy, \ \cdots$$

等がわかればよい．これは，以下を用いて計算できる（ちゃんとそうなることは確かめてみよう）．

$$\int_{\tau} \varphi_{s_j}\varphi_{s_k} \, dxdy = \begin{cases} \dfrac{|\tau|}{6} & (j = k) \\ \dfrac{|\tau|}{12} & (j \neq k) \end{cases}$$

6.12 境界も含めて計算する方法

前節までに述べた方法は，領域の内点のみについて計算する方法であったが，この場合は，新たに境界を設定したり，領域の形状を変更したりしたときに，内点だけに番号を振り直すという操作が必要になり，面倒である．そこで，内点も境界点も同時に計算する方法について説明する．

まず最初に，内点も境界点も区別せずに，既に説明した方法を用いて剛性行列と外力ベクトルを計算する．行列およびベクトルの次数は境界点も含めた分点の数となる．その上で，境界点に関しては値が零となるように剛性行列と外力ベクトルを変形する．具体的には，p_k を境界上の点としたとき，以下のように書き換える．

$$A = \begin{pmatrix} & & & 0 & & & \\ & & & \vdots & & & \\ & & & 0 & & & \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ & & & 0 & & & \\ & & & \vdots & & & \\ & & & 0 & & & \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \\ \\ \\ \text{第 } k \text{ 列} \downarrow \\ \\ \\ \\ \end{matrix}, \qquad b = \begin{pmatrix} \vdots \\ \\ 0 \\ \\ \vdots \end{pmatrix} \leftarrow \text{第 } k \text{ 行}$$

これにより，境界上の点を有限要素基底から切り離し，零に固定することができる．

6.13　三角形分割の方法

本節では，与えられた領域を三角形分割する方法について概略を述べる．三角形分割することを，メッシュを切る，という．領域が直角だけで構成されるような単純な形状なら，図 19 のように直角三角形でメッシュを切るのが楽である．境界が曲がっている場合でも，工夫すれば図 20 のようにメッシュを切ることができる．しかしながら，メッシュの良し悪しについては，三角形の大きさが出来るだけ均一で，しかも三角形の歪が小さい方が良いとされてるので，図 21 のような規則性のないメッシュも用いられる．このようなメッシュは，境界から中心に向けて三角形で埋めていったり，互いに反発する多くの点を領域内にばらまき，定常状態になったところで線で結んでメッシュを作る（ドロネー分割という手法を用いることが多い），等の方法によって作られているが，一長一短がある．FreeFem++ というフリーソフトを用いれば，境界のデータを与えるだけで手軽にメッシュを切ることができる．

6.14　実際の数値計算

有限要素解を求める問題は，最終的に連立一次方程式に帰着された．よって，実際に数値計算する上では，連立一次方程式の効率的な解き方についても考えなければならない．一番簡単なのはガウスの消去法を用いる方法である．しかし，基底関数の数が数万以上になってくると，SOR 法や CG 法などの反復法の方がガウスの消去法よりも効率的になってくる．また，有限要素法の性質から剛性行列は疎行列（非零成分が少ない行列）になるので，メモリを出来るだけ有効に使うテクニックも色々と考えられる．ちなみに，現実の

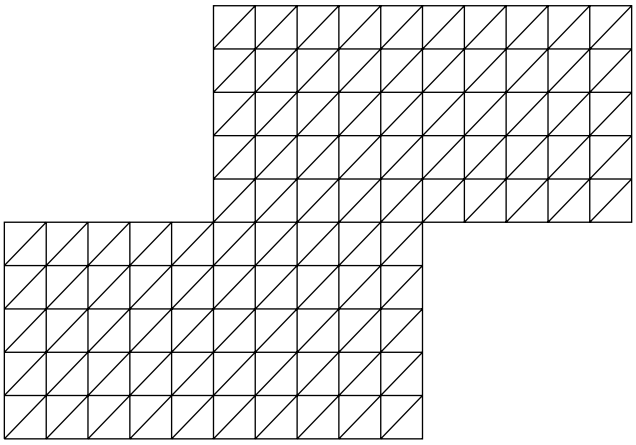

図 19: 直角三角形によるメッシュ

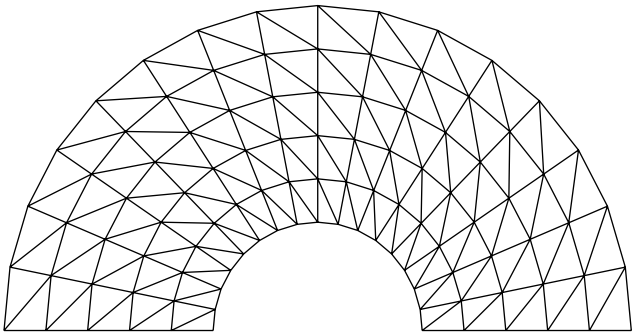

図 20: 規則性のあるメッシュ

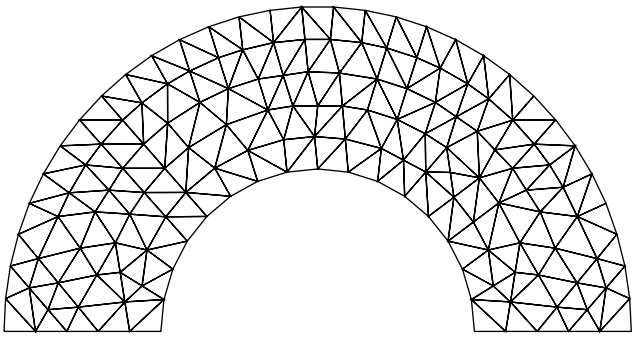

図 21: FreeFem++を用いて作成したメッシュ

問題の解析においては，基底関数の数が数百万個から数億個程度の計算は普通に行われている．

6.15　有限要素法のプログラム

以下に有限要素法のプログラムを示す．汎用性を考えると，メッシュを切って節点一覧表と要素節点対応表を作るプログラムと，節点一覧表と要素節点対応表のデータを読み込んで有限要素解を求める操作は分けた方が良いのだが，今回は簡単のため同時に行っている．領域は正方形 $[0,1]^2$ とし，直角二等辺三角形でメッシュで切っている．

```
#include <stdio.h>
#include "Linear.h"

// 右辺の関数
double f(double x, double y)
{
    return 1;
}

int main()
{
    int N = 20; // 分割数
    int np = (N+1)*(N+1); // 分点数
    int ne = 2*N*N;       // 要素数

    // 節点一覧表
    Vector px(np), py(np);
    IntVector ps(np);

    // 要素節点対応表
    IntVector ep1(ne), ep2(ne), ep3(ne);

    Vector ff(np); // 右辺の関数値を格納

    // 表の作成
    int p = 1;
    int e = 1;
    for(int j=0; j<=N; j++){
        for(int i=0; i<=N; i++){
            double x = i/(double)N, y = j/(double)N;
            ff[p] = f(x,y);
            px[p] = x, py[p] = y;
```

次ページへ続く…

```
            if(i==0 || i==N || j==0 || j==N){
                ps[p] = 1; // 境界点
            } else {
                ps[p] = 0; // 内点
            }

            if(i<N && j<N){
                ep1[e] = p;     //          ／|
                ep2[e] = p+1;   //        ／  |
                ep3[e] = p+N+2; // (x,y)  ----
                e++;
                ep1[e] = p;     //        ----
                ep2[e] = p+N+2; //       |  ／
                ep3[e] = p+N+1; // (x,y) |／
                e++;
            }
            p++;
        }
    }

    Matrix A(np, np); // 剛性行列
    Vector b(np); // 外力ベクトル

    // 剛性行列と外力ベクトルを初期化
    A.Clear();
    b.Clear();
```

次ページへ続く…

```
    for(int e=1; e<=ne; e++){
        int s1 = ep1[e], s2 = ep2[e], s3 = ep3[e];
        double x1 = px[s1], x2 = px[s2], x3 = px[s3];
        double y1 = py[s1], y2 = py[s2], y3 = py[s3];
        double tau = ((x2-x1)*(y3-y1) - (x3-x1)*(y2-y1))/2; // 面積

        double a1 = (y2-y3)/2/tau, b1 = (x3-x2)/2/tau;
        double a2 = (y3-y1)/2/tau, b2 = (x1-x3)/2/tau;
        double a3 = (y1-y2)/2/tau, b3 = (x2-x1)/2/tau;

        A[s1][s1] += tau*(a1*a1 + b1*b1); // 剛性行列に足し込む
        A[s1][s2] += tau*(a1*a2 + b1*b2);
        A[s1][s3] += tau*(a1*a3 + b1*b3);
        A[s2][s1] += tau*(a2*a1 + b2*b1);
        A[s2][s2] += tau*(a2*a2 + b2*b2);
        A[s2][s3] += tau*(a2*a3 + b2*b3);
        A[s3][s1] += tau*(a3*a1 + b3*b1);
        A[s3][s2] += tau*(a3*a2 + b3*b2);
        A[s3][s3] += tau*(a3*a3 + b3*b3);

        // 外力ベクトルの計算
        b[s1] += tau*(ff[s1]/6  + ff[s2]/12 + ff[s3]/12);
        b[s2] += tau*(ff[s1]/12 + ff[s2]/6  + ff[s3]/12);
        b[s3] += tau*(ff[s1]/12 + ff[s2]/12 + ff[s3]/6 );
    }

    // 境界の分離と固定
    for(int p=1; p<=np; p++){
        if(ps[p]>0){
            for(int i=1; i<=np; i++){
                A[p][i] = 0;
                A[i][p] = 0;
            }
            A[p][p] = 1;
            b[p] = 0;
        }
    }
```

次ページへ続く…

```
    Vector u = A.Gauss(b);    // ガウスの消去法

    FILE *fp;
    fp = fopen("fem.dat", "w");
    for(int e=1; e<=ne; e++){
        int s1 = ep1[e], s2 = ep2[e], s3 = ep3[e];
        fprintf(fp, "%f %f %f\n", px[s1], py[s1], u[s1]);
        fprintf(fp, "%f %f %f\n", px[s2], py[s2], u[s2]);
        fprintf(fp, "%f %f %f\n", px[s3], py[s3], u[s3]);
        fprintf(fp, "%f %f %f\n", px[s1], py[s1], u[s1]);
        fprintf(fp, "\n\n");
    }
    fclose(fp);

    return 0;
}
```

データは，空間内の三角形を順に出力するようになっているので，gnuplot で

```
splot "fem.dat" with lines
```

と入力すると，図 22 のように解が表示される．

メッシュ分割さえできれば，正方形領域に限らず，様々な形状の領域においても数値解を計算

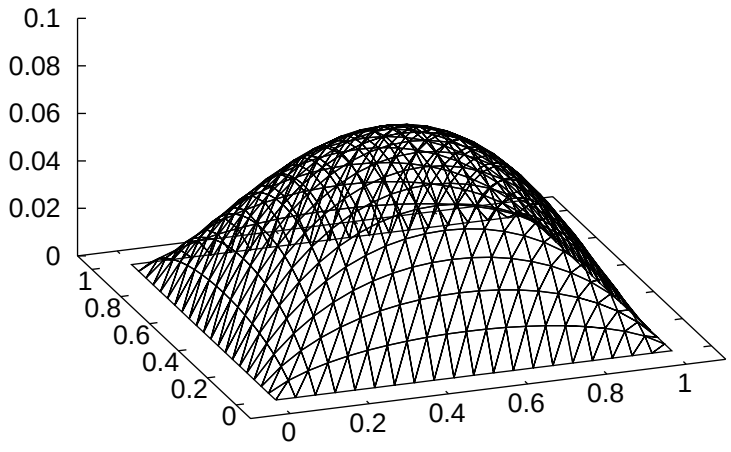

図 22: 正方領域におけるポアソン方程式の数値解

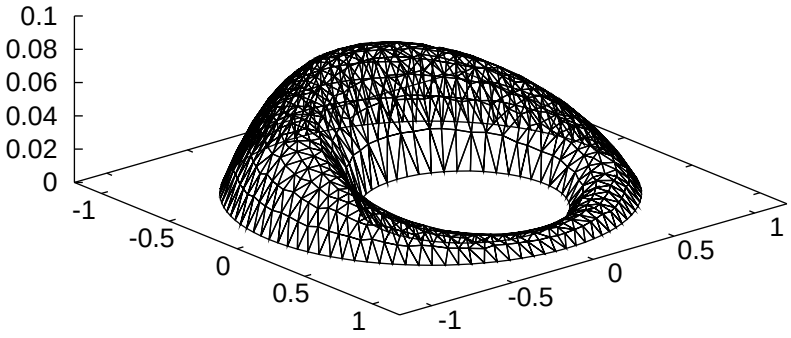

図 23: 円環領域におけるポアソン方程式の数値解

することができる．例えば図23は，偏心円環領域$\left\{(x,y)\;\middle|\;x^2+y^2<1\;(x-1/4)^2+y^2>1/4\right\}$において$f=1$のときのポアソン方程式の解を有限要素法で求めた結果である．

6.16　補足

本資料では簡単のため，境界条件を零としたが，もちろん，境界条件が零以外の問題も解くことができる．興味のある方は調べてみるとよいだろう．

有限要素法の誤差解析については概略のみ触れることにする．有限要素法のガレルキン法的な構成において，有限要素解が

$$\int_\Omega |\nabla(u-u_h)|^2 dxdy$$

を最小にするu_hとして求められたことを思い出そう．そこで，全ての分点でuと値が一致し，各三角形要素上では一次関数となるような関数Πuを考える．Πuを，uの区分線形補間，区分一次補間などと呼ぶ．Πuは有限要素基底の一次結合で書けるので，

$$\int_\Omega |\nabla(u-u_h)|^2 dxdy \le \int_\Omega |\nabla(u-\Pi u)|^2 dxdy$$

が成り立つ．これはつまり，真の解と有限要素解の間の誤差が，真の解とΠuの間の誤差で押さえられることを示している．このように，有限要素法の誤差解析は補間誤差を介して解析することが多い．

有限要素法は関数解析をベースに構築されているので，厳密な構成と誤差解析には関数解析が必要不可欠である．本章では敢えて関数解析を用いていないので，曖昧でスッキリしない部分も多かった．ちゃんと理解したいという人は，頑張って勉強してもらいたい．