計算天文学入門

牧野淳一郎

April 30, 2021

この文書の PDF 版は こちら[1]

[1] ../intro_sim.pdf

Contents

1 初めに **9**

1.1 言語について 9

1.2 参考資料 10

1.3 コンパイラのインストール等 10

2 C++入門 (1) **11**

2.1 簡単な実習 14

2.1.1 プログラムのコンパイルと実行 14

2.1.2 プログラムの修正 14

2.1.3 プログラムの実行 15

2.2 関数と制御構造 15

2.2.1 値を返さない関数（手続き） 15

2.2.2 値を返す関数 16

2.3 プログラムの説明 18

2.3.1 判断 18

2.3.2 反復 19

2.4 関数の宣言と「スコープルール」 20

2.4.1 関数のプロトタイプ宣言 20

2.5 include, namespace, define 21

2.5.1 スコープルール 23

2.6 練習 24

3 C++言語入門 II **25**

3.1 配列 25

3.2 多次元配列 26

3.3 ソート 27

3.3.1 注意 28

3.4 練習 28

3.5 再帰 28

3.5.1 もっとも簡単な例 29
3.6 練習 30

4 C++言語入門 III 31
4.1 お絵書き 31
4.2 コンパイル・実行の方法 32
4.3 グラフィックライブラリ 33
4.4 練習 35

5 放物型方程式 37
5.1 差分法 37
5.2 プログラム 39
5.3 安定性 41
5.3.1 線形安定性解析 41
5.3.2 固有値を求める 42
5.3.3 「直観的」説明 43
5.4 陰解法 44
5.5 3 重対角行列を解く 44
5.6 高精度の方法 46
5.7 練習 46

6 楕円型方程式 47
6.1 線形方程式と反復法 48
6.1.1 ガウス反復 49
6.1.2 ガウス・ザイデル法 49
6.1.3 収束の速さ 50
6.1.4 SOR 51
6.2 もっと高度な解法 51
6.2.1 ADI 51
6.2.2 共役勾配法 52
6.2.3 FFT を使った方法 52
6.3 練習 52
6.3.1 練習 1 52
6.3.2 練習 2 53
6.3.3 練習 3 53
6.3.4 練習 4 53
6.3.5 練習 5 53
6.3.6 練習 6 53

7 常微分方程式 55

7.1 ルンゲ・クッタ 55

7.2 ルンゲ・クッタ以外の方法 57

7.3 線形多段階法 57

7.3.1 アダムス法 58

7.3.2 出発公式 59

7.3.3 陰的アダムス法 60

7.4 構造体とクラス 60

7.5 練習 65

7.5.1 練習 1 65

7.5.2 練習 2 66

7.5.3 練習 3 66

7.5.4 練習 4 66

8 常微分方程式の初期値問題 (2) 67

8.1 今日の予定 67

8.2 刻み幅調節と埋め込み型公式 67

8.2.1 RK 法の場合 67

8.2.2 線形多段階法の場合 69

8.2.3 ニュートン補間とアダムス公式の一般的導出 69

8.3 ハミルトン系向けの方法 71

8.3.1 簡単な例題 71

8.3.2 リープフロッグ公式 72

8.3.3 数値例 73

8.3.4 非線形振動 75

8.3.5 シンプレクティック公式 76

8.3.6 陽解法 76

8.3.7 シンプレクティック公式の意味 77

8.3.8 シンプレクティック公式の問題点と対応 77

8.3.9 対称型公式とは 77

8.3.10 ハミルトン系用の陽的対称型 RK 公式 78

8.3.11 対称型線形多段法 79

8.3.12 エルミート型公式 79

8.3.13 対称型一段公式における時間刻みの変更 80

8.4 練習 81

8.4.1 練習 1 81

8.4.2 練習 2 81

8.4.3 練習 3 . . . 81
8.4.4 練習 4 . . . 81

9 常微分方程式の初期値問題 (3) 83

9.1 今日の予定 . . . 83
9.2 硬い微分方程式 . . . 83
9.2.1 「硬さ」の定義 . . . 85
9.2.2 A-安定性 . . . 86
9.2.3 陰的 RK 法 . . . 86
9.2.4 安定な公式を使う上での問題 . . . 88
9.2.5 完全陰的ルンゲクッタ以外の方法 . . . 89
9.2.5.1 ギアの後退差分公式 (BDF) . . . 89
9.2.5.2 半陰的ルンゲクッタ . . . 90
9.3 次週予告 . . . 90
9.4 練習 . . . 90
9.4.1 練習 1 . . . 90
9.4.2 練習 2 . . . 90
9.4.3 練習 3 . . . 91

10 最適化 (1) 93

10.1 概要 . . . 93
10.2 最適化手法の分類 . . . 94
10.3 連続関数の最適化 . . . 95
10.3.1 1 変数の場合 . . . 95
10.3.2 多変数の場合 . . . 96
10.3.3 CG 法 . . . 98
10.3.4 CG 法の応用:連立 1 次方程式 . . . 98
10.4 制約つき最適化 . . . 99
10.5 練習 . . . 99
10.5.1 練習 1 . . . 99
10.5.2 練習 2 . . . 100
10.5.3 練習 3 . . . 100
10.5.4 練習 4 . . . 100
10.5.5 練習 4 . . . 100

11 最適化 (2) 101

11.1 シミュレーテッドアニーリングの考え . . . 101
11.2 熱平衡状態の実現 – メトロポリス・モンテカルロ . . . 101

11.3 SA の実現 103
11.4 組合せ的最適化への SA の応用 103
11.4.1 SA で TSP の近似解を求める 104
11.4.2 SA のプログラムの一般的な方針 104
11.4.3 もっと高速化する方法 104
11.5 練習 105
11.5.1 問題 1 105
11.5.2 問題 2 105
11.5.3 問題 3 105
11.5.4 問題 4 105
11.5.5 問題 5 105

Chapter 1

初めに

この文書は、学部 3-4 年生から修士 1 年生くらいを対象にした、C++ によるプログラミング、数値計算、その他研究に必要な計算の使いかた解説になる予定である。牧野の研究室向けなので、FDPS を使ったプログラムを書いてシミュレーションができるようになることがとりあえずの目標である。以下のような感じで書いていきたい

- C++ 言語のミニマム
- 偏微分方程式
- 常微分方程式
  - 多体問題
  - FDPS を使った多体問題
- 他の話題
  - 線形最適化
  - 非線形最適化
  - 確率的方法：シミュレーテッド・アニーリング
- 非数値アルゴリズム
  - 基本的データ構造
  - サーチ・ソート
  - ツリー構造とその応用

1.1 言語について

神戸大学惑星学の学生であれば、今までに Fortran による数値計算の実習をしているものと思う。プログラミングの基礎としてもちろん Fortran をやることに問題はないし、分野によっては主力となる言語であるが、逆に分野によって、あるいは目的によってはは他の言語を使うことも必要になっている。その代表が C++ 言語である。

物理・天文・地球科学の分野では、Fortran も使われる一方、 C や C++ 言語での開発も広く行わている。また、色々な目的で、「スクリプト言語」といわれる、python (牧野は個人的には ruby を好むが python が一般的になっていることは否定しがたいので、少なくともこの文書ではスクリプト言語として python を使う、、、つもり、、、である) が使われる。従って、Fortran の他に

- C/C++
- python 等スクリプト言語

が使えることは、研究の他、企業での開発等においても必須になってきているといえる。

1.2 参考資料

C++ の仕様の解説は *C++*リファレンス[1] がある。非常にわかりやすいとはいえないが詳細かつ正確ではある。

1.3 コンパイラのインストール等

Windows のマシンを使っている場合、現在、C++ コンパイラ等を使う標準的な方法は、WSL をインストールして Linux 環境が使えるようにし、それに標準でついてくる GCC を使う、というものになっている。いれかたはネットで検索してそれっぽいものを発掘できるようになってほしい。

WSL で WSL をいれたあとは基本的には

```
（sudo）apt install build-essential
```

でコンパイラ関係ははいるはずである。

Mac では、標準では clang という別の系統のコンパイラがはいっている。これでも通常問題はないが、まあ、 gcc いれましょう。これもいれかたはネットで検索してそれっぽいものを発掘できるようになってほしい。MacOS GCC とかで検索して、「大丈夫そうなもの」に従う。

[1] https://ja.cppreference.com/w/

Chapter 2

C++入門(1)

本章では、簡単なプログラムを例として、「とりあえず C++ を使ってみる」ことを第一の目標にする。それから、 Fortran と対応させながら関数、制御構造などを見ていく。

```
#include <iostream>
using namespace std;
int main()
{
    double a, b, c;
    cin >>a >> b;
    c = a + b ;
    cout << "a+b=" << c << endl;
    return 0;
}
```

上のプログラムは、簡単な C++ プログラムである。まずこのプログラムを動かしてみるということが目標である。まず、このプログラムはどういうものかを説明しておく。

なお、同じことを Fortran で書くと多分こんな感じである。

```
      program sample
      real*8 a, b, c
      read(5,*) a, b
      c = a + b
      write(6,*)  'a+b=', c
      end
```

比べてみると、まあ、似ているところもあるし違うところもある。とりあえず順番に見ていこう、、、、といいたいところだが、最初の 2 行はちょっと後回し。

まず `int main()` である。これは、「プログラムの始まり」を示すものであるところは Fortran における `program sample` と変わらないが、細かくいうといろいろ違う。 Fortran では、 program 文は実はなくてもよくて、プログラム文で始まる、またはいきなり始まるプログラム単位（とはなに、、、というのは省略）が「メインプログラム」、つまり、プログラムの実行がそこから始まるものであった。

つまり、 Fortran では、まずメインプログラムがあって、それがサブルーチンとか関数を呼び出すという形になっており、メインプログラム、サブルーチン、関数のどれであるかで書き方がすこしずつ違っていた。

C/C++ でも同じようにメインプログラムがあって、それがサブルーチンや関数を呼び出していくわけだが、 Fortran と大きく違うのは「宣言のしかたに区別がなくてみな関数の形で宣言する」ということである。メインプログラムは「main という名前の関数」であり、そういう名前をつけておくとリンカがここから実行を始めるようにしてくれる。逆に、main という名前の関数がないとリンク時にエラーにになる。

関数とサブルーチンの違いは値を返すかどうかということだけなので、 C/C++ では値を返さない関数も許すことで統一的な記法を可能にしている。

元に戻ると、main 関数の宣言は

```
int main()
{
    関数の本体
}
```

という形になっている。ここで int は関数の型であり、ここでは整数型ということになる。

C/C++ で使う基本的な型には以下のようなものがある

型名	説明	Fortran との対応
int	整数型	integer
float	実数型	real (real*4)
double	倍精度実数型	double precision (real*8)
char	文字型	character

整数型では長さを指定出来る。このあたりから処理系依存になってくるが、short, long, long long といったものが指定できる。さらに、符号ありかどうかを signed/unsigned とつけることで区別できる。したがって、unsigned long long int と書くと非常に長い符号なし整数ということになり、通常 64 bit の整数で 0 から $2^{64}-1$ までを表現できることになる。

さて、メインプログラムが値を返しても受けとるところがないと思うかもしれないが、これは実は OS （というか、 UNIX/Linux/Windows の通常の環境ではプログラムを起動したシェル）が受けとる。帰ってきた値によって、プログラムが正常に終了したかどうかを判断したりするのにつかうわけである。

次に main は関数の名前で、これはメインプログラムなら main でないといけない。それ以外では好きな名前をつけていいわけだが、名前はアルファベットまたはアンダースコアで始まり、アルファベット、アンダースコアまたは数字が続く。 Fortran 77 の規格では名前は 6 文字以下という制限があったが、 C/C++ では少なくとも 31 文字までは問題なく使えることになっている。

その次の () は引数リストを書くためのものだが、今日のメイン関数は引数がないので中身は空である。そのあとの中括弧 { から } までの間に

- 変数の宣言
- 実行文

を書く。まあ、この辺はそう決めたからそう書くことになっているというだけ。

なお、 Fortran との違いとして、 改行や行内での位置が意味を持たないということがある。 Fortran だと（少なくとも昔のでは）文は 7 カラム目から 72 カラム目までに書くとか 1-5 カラムはコメント

記号／文番号であるとか 6 カラム目は継続行マークとかいうのがあったが、 C/C++ ではその辺は全く気にする必要はない。長い式なら適当に改行して構わないし、 72 カラムをはみ出したものが無視されるとかいうこともない。

そのかわり、変数名、キーワード中に空白をいれたりはできない。例えば Fortran では

```
p r o g r a m sam ple
wr ite(6,*)'Test'
r e t u r n
e n d
```

というようにプログラムの中に好きなように空白を入れることができたが、C/C++ ではそんなことはできない。

なお、この「空白が無視される」という Fortran の仕様は割合問題が多いものである。`DO 10 I = 1.10` が一体なんであると解釈されるか？を考えてみるとちょっと面白い。

次は `double a, b, c;` である。これは `a, b, c` が `double` 型の変数であると宣言している。上に書いたように改行に特別な意味がないので、宣言の終りをしめすためにセミコロン “;” をつける。これは実行文でも同様である。 Fortran では変数は宣言しなくても使えたが、C/C++ では必ず宣言しないといけない。これは、タイプミスによって妙なバグが発生することを防げるので好ましいと思う。

なお、 Fortran でも C で関数やサブルーチンの中の変数宣言は先頭にまとまっている必要があったが、 C++ ではそんな必要はなく使うところより前であればどこでも宣言できる。

次にくるのが

```
cin >>a >> b;
```

であるが、これはキーボード（正確には「標準入力」）からの入力がまず変数 a、次に変数 b に格納される。基本的には Fortran の read 文と同じようなものである。細かいことをいい出すと無限にあるのでそのあたりは参考書を見ること。

次の `c = a + b ;` は最後にセミコロンがつくのを別にすれば Fortran と同じ。式の書き方、使える数学関数などは Fortran とさして変わりはない。こまかな違いはいろいろあって、例えば

- ベキ乗演算子がない。代わりに関数 `pow(x,y)` を使う。
- いろいろな数学関数は基本的には `double` 型である。 Fortran の場合のように使われ方によって型が変わったりしない。
- Fortran にない便利な演算子が沢山ある。例えば
  - `+=` 右の式の値だけ左の変数の値を増やす。 `a+= b` は `a = a + b` と同じ。
  - 整数に対して論理演算とビット単位のブール演算をする多数の演算子がある。

整数に対する多様な操作が提供されていることは、ハードウェア制御など、普通には高級言語ではできないような操作を可能にする。これが C/C++ が広く使われる理由の一部ではある。

さて、出力の

```
cout << "a+b=" << c << endl;
```

をみてみよう。これも、とりあえずは Fortran の write 文と同じようなものと思っておいていい。文字列定数は一重ではなく二重の引用符で括る。いくつかのものを並べて書くには ¡< でつないでいく。改行には `endl` を書く。これをつけないと行が変わらないので、その次に何か書くと同じ行につながって書かれる。Fortran では、特別な制御をしない限り write 文では自動的に改行が入ったが、C/C++ ではそうではない。なお、実数の書式制御とかの細かい指定ももちろんできる。これも参考書のほうを見て欲しい。

最後は `return 0;` である。これは、この関数が 0 を返して終るということになる。 0 の代わりに int 変数や整数値になる式を書けば、もちろんその値が戻る。

通常の UNIX シェルでは、この値が `$status` というシェル変数に格納される。

2.1 簡単な実習

まず、このプログラムをエディタで作成し、例えば example1.cpp といった名前でセーブしてみよう。

ここで、いくつかの注意をしておく。

- プログラムはすべて「半角文字」で書かれる。
- プログラムの中の空白、改行には全く意味はない。(ただし、#include 何とかの後は改行または空白が必要) したがって、全く改行なしに一行に全部書くとか、空白を全部詰めて行の先頭から書くとかしても問題はない。ただし、人間が見た時の読みやすさは書き方で違い、上の例のようなやりかたはそれなりに読みやすいものであろう。
- ファイル名は、 .cpp で終っていないといけない。(.C とか .cxx でもいいかもしれない) そうなってないと、 C++ プログラムが入ったものと認識されない。別の名前でつくってしまったら、名前を付け直すこと。
- emacs の場合、プログラムを書き始めるときに、あらかじめファイル名を指定しておく(File ... Open file または Ctrl-x Ctrl-f) と、emacs の方で C++-mode というものになってプログラムを見やすい形にしてくれるので楽である。

2.1.1 プログラムのコンパイルと実行

コンパイルとは、C++ で書いたプログラムを、計算機が実際に実行できる形に翻訳する作業である。これには g++ というコマンドを使う。

コンパイルは、シェルウインドウで

```
g++ example1.cpp  -o example1
```

と入れる (例によって最後にリターンする)。すると example1 という名前の実行ファイルができる。

2.1.2 プログラムの修正

多くの場合、入力したプログラムは実行前に「エラーです」のようなメッセージがでて止まってしまう。エラーの場所に応じてエディタでプログラムを修正し、セーブしたあともう一回 g++ して見よう。

2.1.3 プログラムの実行

実行は、シェルウインドウで

```
./example1
```

と入れる。このあとリターンすると、キーボードから数字を入れるのを待っている状態になるので、数字をいれてはリターンするのを 2 回繰り返せばその 2 つの和が表示されるはずである。

少しプログラムを修正して、もう少し芸のあるものにしてみよう。

```
// program example 2
#include <iostream>
using namespace std;

int main()
{
    double a, b;
    cout <<"Enter numbers a and b:";
    cin >> a >> b ;
    cout << "a+b = " << a+b <<endl;
    cout << "a-b = " << a-b <<endl;
    cout << "a*b = " << a*b <<endl;
    cout << "a/b = " << a/b <<endl;
    return  0;
}
```

最初の // ... はコメント（注釈）といわれるもので、コンパイラは// からその行の終りまでを無視する。このため、自分や他人が後でもみて分かるようにするためのいろいろな説明などを書いておくことができる。

これは四則演算してみただけである。

2.2 関数と制御構造

2.2.1 値を返さない関数（手続き）

数学では「関数」といえば、指数関数とか三角関数のように、変数に対応して値が決まるものだが、C++言語の場合は必ずしもそうではない。以下の例で説明しよう。

```
// procedure_sample
#include <iostream>
using namespace std;

#define PI  3.14159265358979

void print_volume(double radius)
{
    cout <<"Radius = " << radius << endl;
```

```
        cout <<"Volume = " << radius*radius*radius*PI*4.0/3.0<<endl;
    }

    int main()
    {
        double x;
        cerr << "Enter radius : ";
        cin >> x;
        print_volume(x);
        return 0;
    }
```

このプログラムは、単に適当な数字を読み込んで、その値を半径とする球の体積を表示するプログラムである。このプログラムでは、実際に体積を計算して答を表示するのを、 `print_volume` という名前の関数が行なっている。

この「関数」は英語の function の訳語であるが、数学的な「関数」というより、機能とか働きとかいった意味合いに近い。ただし、すぐあとで説明するように、値を返す関数というものもあり、こちらは数学的な意味での関数に少し似ている。

値を返さない関数は、

```
void 名前 (型 引数 1 [, 引数 2, ...] [, 型 [ ... 引数 i [, 引数 i+1,...]])
{
    [変数宣言]
    実行部
}
```

という形をとる。このような記述がプログラムのなかにあると、もとのプログラム、つまり int main() で始まっているところの実行部のなかからここで新しく作った関数を「呼び出す」ことができる。Fortran では call とかがついたが、 C/C++ ではいきなり関数名を書くだけである。

2.2.2 値を返す関数

```
// bisection
#include <iostream>
using namespace std;

double f(double x)
{
    double y ;
    y = x*x*x - 2;
    return y;
}

void bisection(double &  xmin,
               double &  xmax,
               double eps)
{
```

```
    double x, f_min, f_max;
    f_min = f(xmin);
    f_max = f(xmax);
    if (f_min * f_max  > 0.0){
        cout <<"cannot find solution...\n";
    }else{
        while(xmax - xmin > eps){
            x = (xmin + xmax) *0.5;
            if (f(x) * f_min > 0.0 ){
                xmin = x;
            } else{
                xmax = x;
            }
            cout << "x= " << x << " f(x)= "<< f(x) <<endl;
        }
    }
}

int main()
{
    double x0,x1, eps;
    x0 = 0.0;
    x1 = 2.0;
    eps = 1e-10;
    bisection( x0, x1, eps);
    cout << "Final x = " << x0 << " " << x1 << endl;
    return 0;
}
```

ここでは、「関数」らしく値を返すものを使ってみている。値を返さないものとの違いは、

`void` 名前 (引数の宣言);

の代わりに

型 名前 (引数の宣言);

となることと、実行部の最後で、

`return` 式;

の形の戻すべき値を指定することである。このようにして宣言した関数は、 C++言語の標準のライブラリに入っている `sin`, `cos`, `pow` などの関数と全く同じように使うことができる。

関数では、値を一つしか返せない。したがって、上の例のように、二分法で方程式を解いて、区間の両端の値を戻したければ、引数の形で返すことになる。

とはいうものの、最初の例のところで書いたように、C++では普通に宣言すると関数の引数の値はコピーされる。で、コピーされた方を書き換えても、元の値は書き換わらない。元の変数の値を書き換えるためには、上の例のように引数の宣言のところで型と変数名の間に `&` をつける。

これは、 C の場合とは大きく違うことに（C を知っている人は）注意。もちろん、 C と同じように書くこともできる。

C++ の場合、 `&` をつけた引数は Fortran の場合とおおむね同じように使える。

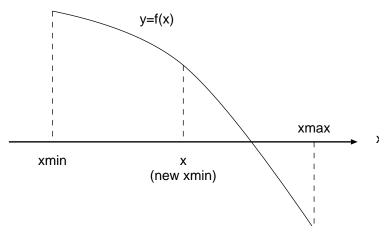


Figure 2.1:

2.3 プログラムの説明

上のプログラムは、2分法で、方程式の(近似的な)解を求めるものである。方程式は、関数=0という形になっているものとしよう。

このやりかたでは、まず最初にどの範囲に答があるかは知っているものとする。そうすると、図2.1にあるように、その範囲の両端で関数の符号が違っているはずである。

その区間の中点で関数の値を計算する。図のように、中点での値と左端での値の符号が同じなら、答えは中点と右端の間にある。この時は、中点の値で左端の値を置き換える。逆に中点での値と左端での値の符号が違えば、もちろん答えはその間にある。この時は右端の値を置き換える。いずれの場合でも、答があるとわかっている区間の幅がもとの半分に狭まる。これを繰り返していって、答をもとめる。

以下、関数 bisection の中身を見ていく。

2.3.1 判断

```
if (条件) 文1
else 文2
```

という構造は、条件が成り立っていれば文1を、そうでなければ文2を実行せよという意味になる。文2がない(条件が成り立っている時はなにかするがそうでなければ何もしない)ときには

```
if (条件) 文1
```

だけでいい。なお、上の例では `if (条件)` の後ろが { 文 文... とつながっている。このまとまりのことを複文といい、一般に文が書けるところには複文も書ける。このため、 `if (...) { .... }else { .... }` というような風にすれば条件によって違ういくつかの処理をまとめてできる。

文とは何かをちゃんと説明してなかったが、C/C++では式にセミコロンをつけたものが文である。で、式はなにかというと、代入 `a=b+c` といったものも式、関数呼びだし `bisection(xmin,xmax,eps)` も式、単なる数式 `a+b` ももちろん式である。

C/C++言語の特徴として、実行されるものはすべて式であり、(void であるということも含めて)値を持つということがある。代入式の値は代入された値それ自体なので、例えば `a = b = c+d` といったもので a と b の両方に同じ値を代入できる。

条件は、数値同士の比較式(大小、等しい)と、複数の比較式からできる論理式などが書ける。

具体的には、

`a > b`	a が b より大きければ真
`a >= b`	a が b より小さくなければ真

```
a <  b              a が b より小さければ真
a <= b              a が b より大きくなければ真
a == b              a が b と等しければ真
!条件               条件が偽なら真（否定）
(条件１ ) &&（条件 2)   両方真なら真（論理積、and）
(条件１ ) ||（条件 2)   どちらかが真なら真（論理和、or）
```

というくらいがこれから出てくることがあるであろう。

ただし、実際に条件に書かれるものは値が整数になる式ならなんでもいい。さらにおせっかいに、実数型の式でも勝手に整数に変換して評価してくれたりする。このため、条件のところに `a = b` と書いても文法的には正しいが、大抵の場合やってほしいこととはかなり違う意味を持つ。なお、処理系によってはこれに警告を出してくれるものもある。

2.3.2 反復

```
while (条件){
     文
     ......
     文
}
```

これは標準の Fortran 77 には対応するものがないが、非常に便利なものである。なお、 Fortran の DO ループに対応するものは for 文であり、

`for（変数 = 最初の値; 変数 < 最後の値 + 1; 変数 ++ ） 文`

という形に書くのが普通である。

これは、まず変数に最初の値を入れて文を実行し、次に変数を 1 増やしてまた文を実行し、以下同様に繰り返して変数が最後の値になったらおしまいにするということになる。このような繰り返し処理をループ処理という。この場合は Fortran の DO ループとほぼ同じ動作になる。

C 以外の多くのプログラム言語では、例えば `do i = 1, 10`（Fortran の場合）というように、「変数を 1 増やしては同じことを繰り返す」という上に書いた通りのことをするための特別な書き方（構文）があるが、 C/C++ の `for` を使った構文はもっとフレキシブルなものである。

`for(式 1; 条件式; 式 2){ 式 3; 式 4; ... }`というふうに書いてあると、実際に

起きることは、

- まず式 1 を実行する。
- 次に条件が成り立っているかどうか調べる。成り立っていなければおしまい。
- 次に 式 3; 式 4; ... を実行する。
- 次に 式 2 を実行して、2 番目（条件式）に戻る。

ということで、別に「変数に最初の値を入れて文を実行し、次に変数を 1 増やしてまた文を実行し、」ということしかできないわけではない。例えば

```
int i;
for(i=0; i<262144; i *= 2){
   cout <<"i = " << i << endl;
}
```

と書けば、変数 i の値を繰り返しごとに 2 倍にすることになる。

なお、ここで、i++; とか i *= 2; とかいうものが出て来たが、これは C/C++ 言語に特有の書き方で、基本的には i++ は i = i + 1 と同じ意味だし、 i*= 2 は i = i * 2 と同じである。また、 i – という表現も使える。

一般に、あらゆる演算（加減乗除の他に、論理演算なども）について、

```
i 演算記号=j
と書くのは
i  = i  演算記号 j
と書くのと同じと思っていい。
```

2.4 関数の宣言と「スコープルール」

以下に書くことは、必ずしも本質的ではないが実際にプログラムがどう動くか、あるいはどうやって書くかを理解するには結構重要なことである。

2.4.1 関数のプロトタイプ宣言

これまでの例では、

```
double f(x)
{
...
}
void bisection(...)
{
   ... = f(...);
}
int main()
{
...
   bisection(...);
}
```

といった風に、「使う関数はあらかじめその前に定義されている」という形になっている。で、例えば引数の対応が間違っているとコンパイラがチェックしてくれる。これでうまくいくのはまあいいような気がするわけだが、良く考えてみるとちょっと変である。

というのは、使うすべての関数をプログラムの中で定義できるわけではないからである。このために使っているのが、「プロトタイプ宣言」と呼ばれる機能である。プロトタイプ宣言は、例えば以下のような形をしている。

```
void bisection(double & xmin,
               double &  xmax,
               double eps);
```

つまり、関数の最初のところだけ書いて、その後に実行部をつけないでセミコロンを書いておしまいにしたものである。これは、「この名前の関数はこういう引数をもらって、こういう値を返します」ということをコンパイラに教える役割をはたしている。

この、プロトタイプ宣言というものは、要するにある関数について、「それが外からどう見えるか」を規定している。えらそうにいえば「インターフェース」を決めているということもできる。

Fortran77 では、言語仕様の中には特にこのようなチェックについて規定しているところはないので、引数の型や数が間違っていてもコンパイラはエラーを検出してくれないことが多い。

また、 C 言語の場合、プロトタイプ宣言が間違っていた時にかならずしもそれが発見されるとは限らなかった。 C++ の場合には引数が違うものは違う関数になるので、間違ったプロトタイプ宣言があるとリンク時に失敗するので発見できる。

これは C++ の重要な機能、「関数のオーバーロード」（多重定義）というものの結果である。例えば Fortran では、

```
subroutine foo(x)
real x
...
end
```

というものがあった時に、

```
subroutine foo(x)
integer x
...
end
```

というふうに引数は違うが名前は同じ関数を定義することはできない。ところが、C++ では

```
void  foo(double x){}
```

と

```
void  foo(int x){}
```

は別物として扱われる。

もちろん、だからといって必要もないのに同じ名前を使うことは混乱を招くので避けるべきだし、また引数の型が違うだけで処理の内容が同じならばテンプレート（多分後で説明する）を使って汎用の関数を作るべきであろう。

2.5 include, namespace, define

プロトタイプ宣言がでてきたついでに、最初にスキップした

```
#include <iostream>
using namespace std;
```

が何をしているかをみていく。

```
// #include <iostream>
```

と、この行をコメントアウトしてみよう。

```
g++ example1-include-commentout.cpp
example1-include-commentout.cpp: 関数 ‘int main()’ 内:
example1-include-commentout.cpp:6:5: エラー:‘cin’was not declared in this scope
     cin >>a >> b;
     ^
example1-include-commentout.cpp:8:5: エラー:‘cout’was not declared in this scope
     cout << "a+b=" << c << endl;
     ^
example1-include-commentout.cpp:8:28: エラー:‘endl’was not declared in this scope
     cout << "a+b=" << c << endl;
                            ^
```

といったエラーがでる。つまり、

```
#include <iostream>
```

によって、 cin, cout, endl といったものが何であるかがコンパイラにわかるようになる。ではこれは何をしているか、というと、計算機のどこかにある (どこかはコンパイラのほうが知っている) 場所にある iostream というファイルを読み込んで、一緒にコンパイルしてもらう、というものである。つまり、プログラムとしては、この行が、 iostream というファイルの中身に一旦置き換わる。

なお、「置き換えた結果」をコンパイラに出力させることもできる。

```
g++ -E example1.cpp
```

とすると、置き換えた結果が標準出力 (この場合画面) にでる。

なお、あとででてくるが、

```
#include "foo.h"
```

とすれば、カレントディレクトリの foo.h がインクルードされる。また、相対パスや絶対パスで指定することもできる。

(カレントディレクトリ、相対パス、絶対パスといった言葉の意味がわからない人はちゃんと調べてください)

次に、

```
using namespace std;
```

である。まず、namespace 「名前空間」というものがある、という話から始める必要がある。この、cin, cout といったものは C++ 言語仕様として提供することになっているものだが、他の追加の機能 (「ライブラリ」) を、色々な人が使えるように提供するとか、大きなプログラムを分担して開発するとかを考えると、こういった関数の名前が、ある人が作ったものと別の人が作ったものが同じになってしまうことがありえる。

それを (ある程度避けるのが、この namespace というもので、例えば cin, cout は、std という「名前空間」の中で定義されている。これはどういうことかというと、実際には、 cin とか cout とか書

いただけではコンパイラはその存在を認識できなくて、 std::cin とか std::cout とか書く必要がある、ということである。

で、それは面倒だとか、 namespace がなかった時代に書かれたプログラムを大きく書換えないですね動かしたい、といった時に使うのがこの using namespace というもので、

```
using namespace std;
```

と書いておけば、 std::cin と書かずに cin と書くだけでコンパイラはそれだとわかるようになる。

2.5.1 スコープルール

スコープというのは何かというと、例えば変数であれば、宣言した変数をどこで使うことができるかということである。宣言の有効範囲ということもできる。例えば

```
double f(x)
{
    double y;
    ...
}
int main()
{
    double y;
    .....
}
```

というプログラムを考えてみる。ここでは、 f(x) の中の変数 y と、main の中の変数 y は全く別物であって、例えば f(x) の中で y を書き換えたらその結果が main に伝わったりはしない。物理的には、これは、メモリの違う場所におかれるということを意味している。

こういうことができるのは、ある関数の中で宣言した変数は、その関数のなかでだけ有効だからである。その関数以外の関数が、勝手に変数を書き換えたりはできない。これは、不便なような気がするかもしれないが、大きなプログラムを作るという場合とか、たくさんの人が協力してプログラムを作る時とかには非常に重要な機能である。

ただし、抜け道も準備されている。例えば

```
double y;
double f(x)
{
    ...
    y = ...
}
int main()
{
    f(x) = ...
    cout << y << endl;
}
```

といったように、「関数の外」で変数を宣言することもできる。この場合には、変数宣言の下に出てくるどの関数でも、この変数を使うことができ、それらはみな同じものである。したがって、上の例のように、f の中でその変数に代入し、 main の中でその値を見れば、f で入れた値が出てくることになる。

Fortran ではこのようなことを実現するのには COMMON BLOCK を使う必要があったが、C/C++ では手軽に関数間で変数の共有ができる。これは利点でもあり欠点となることもある。

今日はこれくらい。次回は C++ 文法事項の続き。

2.6 練習

今日出たプログラムを実際に書くなり cut & paste するなりして動かしてみること。

Chapter 3

C++言語入門 II

本章では、後の話で必要になる最低限の C++ の知識ということで、配列と再帰処理について簡単にまとめる。

本来、 C++ らしい（C や Fortran ではできないような）プログラムスタイルの根幹をなすのは、「クラス」と「テンプレート」であるが、また後で触れることにしたい。

3.1 配列

Fortran でも C/C++ でも、数値計算をする上で基本になるデータ構造は多くの場合に配列である。

// calculate_sum using namespace std;

```
int main()
{
    int  n ;
    int  data[100];
    int  i,sum;
    cout << "How many numbers you want to input?";
    cin >> n;
    for (i=0;i<n;i++) cin >>data[i];
    sum = 0;
    for (i=0;i<n;i++){
        sum += data[i];
        cout << "data[" << i <<"]=" << data[i] << " sum=" << sum << endl;
    }
}
```

これは配列を使う簡単な例である。

```
int  data[100] ;
```

と変数宣言で書くと、100 個の整数型変数ができ、それぞれが `data[0]` から `data[99]` までといった風に、番号で指定して操作できる。

これはもちろん Fortran で

```
integer data(100)
```

と書くのとほとんど変わらないが、どうでもいいような違いがいくつかある。

- 括弧が丸括弧ではなくて角括弧である。
- 添字が 1 からではなく 0 から始まる。

括弧はまあなんでもいいが、添字が 0 からになるのは C/C++ の他の多くの言語とは違った特異な点である。なお、 Fortran の場合は、例えば

```
integer data(0:100)
```

といった具合に「最初と最後」を指定できるので、実は 1 から始めないといけないというわけではない。しかし、 C/C++ の場合には必ず 0 から始まり、それを変更する方法はない。

3.2 多次元配列

宣言は Fortran では

```
integer data(100,100)
```

とか書いたが、C/C++ では

```
int data[100][100];
```

と書く。使い方も同様に Fortran では `data(i,j)` のところを `data[i][j]` と書く。次元が増えても同様である。 Fortran では配列の次元は 7 次元までとかいう制限があるが、 C/C++ では特に制限はない。

なお、他にも細かく違うところはある。実用上で大事なのは、メモリ上にデータがしまわれる順番である。 1 次元配列ではもちろん配列要素が順にメモリ上におかれるが、2 次元配列では 2 つの可能性がある。つまり、最初の要素 `data(1,1)` あるいは `data[0][0]` の次に来るのがなにか？である。Fortran では `data(2,1)` が来るが、C/C++では `data[0][1]` が来る。この違いは、いろいろな場面で意識する必要がある。

なお、この C の言語仕様で定義された普通の配列に関する限り、 C/C++ の多次元配列には Fortran のそれに比べた大きな欠点がある。それは、関数の引数として配列を渡す時に、大きさを変数として渡すことができないということである。つまり、 Fortran では

```
subroutine sub1(a, n, nn)
integer n, nn
real*8 a(nn,nn)
.....
```

といった形で、次元ごとの大きさを引数として渡すことができる。このために、汎用の行列計算や連立一次方程式の解法などのサブルーチンを準備するのが容易である。ところが、 C/C++ では引数としては渡せないので話が面倒になる。もっとも、C の最新の言語仕様ではこれが可能になっているが、現在のところ存在するほとんどのプログラムはこの機能を使わないで書かれている。

もっとも、これはあくまでも汎用のライブラリを準備し、しかもそれをソースプログラムとしてではなくあらかじめコンパイルしたいわゆるバイナリライブラリとして準備する時にだけ問題になることではある。従って、自分でプログラムを書く場合にはあまり問題にはならない。

コンパイルしたライブラリを作るにはいろいろな方法がある。 C++ では、ユーザーが新しいデータ型を定義することができるので、自分で定義してしまえば Fortran の配列よりももっと便利に使えるものができる。また、既に人が作った便利なものがいろいろあるので、高度なことをするにはそういったものの利用を考えるべきである。

3.3 ソート

配列を使う例だが、計算天文学とはいえ、数値計算ばかりでも芸がないので、ここではちょっと違うこともやってみよう。

```
// sort.C

#include <iostream>
using namespace std;

double data[100];

int main()
{
    int n;
    int   i,j;
    cin >> n;
    for (i=0;i<n;i++)cin >> data[i];
    cout << "Input data\n";
    for (i=0;i<n;i++) cout << data[i]<<endl;
    for (i=0;i<n-1;i++){
        for (j=i+1;j<n;j++){
            if (data[i] > data[j]){
                double  work;
                work = data[i];
                data[i] = data[j];
                data[j]= work;
            }
        }
    }
    cout << "Sorted data\n";
    for (i=0;i<n;i++) cout << data[i] << endl;
    return 0;
}
```

このプログラムは、入力した数を小さい順に並べ変えて出力する。並べ変える原理は、以下のようなものである。

1. まず、一番小さい数を先頭に持ってくる。そのために、配列の 2 番目の要素から最後の要素までを順に見て、それが先頭の値よりも小さければ 2 つを入れ換える。

2. これで先頭にもっとも小さい数が来た。次に、2 番目の位置に残りの数の中で最小のものを持っ

てくる。これには、配列の 3 番目の要素から最後の要素までを順に見て、それが 2 番目の値よりも小さければ 2 つを入れ換える。

3. 以下、同様に、 N-1 番目まで順にやっていく。

このプログラムを実行するには、まずデータファイルを別にエディタで作っておく。これは先頭にデータ数を書き、後は一行に一つ数字を書いておけばいい。このデータファイルの名前を sample.dat、コンパイルしてできた実行ファイルの名前を simple_sort とすれば、

```
./simple_sort < sample.dat
```

とシェルウインドウで入力すれば結果が得られるはずである。

3.3.1 注意

コンパイルして作るプログラムの名前には sort は使わないこと。これは、この名前の別のプログラムがあらかじめ準備されていて、そちらが実行されてしまうからである。もちろん、 ./sort と指定すれば自分で作ったものが実行されるが、いずれにしても混乱する可能性がある。

3.4 練習

1. ソートのプログラムを手続きを使う形に書き直してみよ。メインプログラムは手続きを呼ぶだけの形になるようにすること。

   (a) ソートのプログラムについて、データの数を非常に大きくした時、実行時間がどうなるかを測定せよ。配列宣言の数字を大きくするのを忘れないこと。また理論上どうなるはずか考えてみよ。

なお、あるプログラムの実行時間を計るには time コマンドが使える。

```
time ./simple_sort < sample.dat
```

という風に入力すると、実行結果のあとにもう一行

```
0.040u 0.080s 0:00.56 21.4% 0+135k 2+0io 21pf+0w
```

というような出力が出てくる。この最初の数字が秒単位の実行時間である。なお、実際にかかった時間はもっと長いことがあり得る。これは、ファイルの読み書きの時間は正確には入らないこと、また何人かで 1 台の計算機を使っているとその分余計に時間がかかることによる。

3.5 再帰

配列に関しては C++ は Fortran に比べて必ずしも良くなっているとはいい難い、というか正直いって退化しているが、プログラムの構造という観点からはいくつか Fortran にはない有用な特色を持つ。その一つがここで述べる再帰である。なお、最近の多くの Fortran コンパイラでは、言語仕様では再帰をサポートしていなくても実際には仕様が拡張されていて再帰が使える。また、F90 は言語仕様として再帰をサポートする。

再帰とは、手続きが「自分自身を呼び出す」ことである。これは、プログラミングの理論的な扱いという観点からも、また、実用上からも非常に重要な考え方である。簡単な例から考えてみよう。

3.5.1 もっとも簡単な例

```
// factrial
#include <iostream>
using namespace std;

double factrial(int n)
{
    if (n <1){
        return 1.0;
    }else{
        return  n*factrial(n-1);
    }
}

int main()
{
    int n;
    cout << "Enter n:";
    cin >> n;
    cout <<"N = " << n << "   N! = " << factrial(n) <<endl;
    return 0;
}
```

上の例は、階乗を計算するプログラムである。階乗は、下のように定義される：

$$N! = \begin{cases} 1 & (N = 1) \\ N \cdot (N-1)! & \text{otherwize} \end{cases} \tag{3.1}$$

本当は N が整数でないと定義されないし結果も整数だが、整数だとすぐにオーバーフローしてあまり大きな数の階乗は求められないのでプログラムでは `double` を使っている。

C や Pascal のような「近代的」プログラミング言語では、この定義に従って素直にプログラムを書くことができる。上の例では、まさに、 `n` が 1 なら 1 を返し、それ以外では `n*factrial(n-1)` を返すというのが、 `factrial(n)` の中身になっている。

例えば `factrial(5)` を計算させるとすると、実際の計算は、以下のような手順で進む。

```
actrial(5)
→ 5*factrial(4)
→ 5*(4*factrial(3))
→ 5*(4*(3*factrial(2)))
→ 5*(4*(3*(2*factrial(1))))
→ 5*(4*(3*(2*1)))
→ 5*(4*(3*2))
→ 5*(4*6)
→ 5*24
→ 120
```

計算機の構造を考えると、こういうことができるというのは一見不思議ではある。つまり、関数 `factrial` の引数 `n` は同じものなのに、なぜそれが呼ばれるたびに違う値をとることが可能になるのであろうか？

再帰処理を許さない言語の場合、関数の引数や関数中で宣言された変数には「固定したアドレス」が与えられる。このために、関数が自分自身を呼ぶと、引数の値や変数の値が書き変わってしまいなんだかわからない動作をすることになる。

これに対し、 C/C++ などの再帰処理を許す言語では、関数の引数や関数中で宣言された変数は固定されたアドレスを持っているわけではなく、関数が呼ばれる度に新しいメモリ領域が割り当てられる。これは別に難しいことではなくて、ある連続したアドレス領域（通常スタック、、、書類を積み上げた山、、、と呼ばれる）をあらかじめ確保しておいて、サブルーチンが呼ばれればそこに新しく場所をとり、サブルーチンから抜ける時に使っていた場所を返すだけである。

ただこれだけのことで、「関数が自分自身を呼ぶ」というなんだか怪しげなことが実現できるわけである。

3.6 練習

1. 階乗を計算する関数を、再帰を使わない形に書き直してみよ。
2. フィボナッチ数列 $F(n) = F(n-1) + F(n-2), F(0)=0, F(1)=1$を再帰、再帰でないものの両方を使って書け。
3. 上のフィボナッチ数列について、n を大きくしていった時に計算の手間がどのように増えるかを、再帰の場合とそうでない場合のそれぞれについて考察せよ。

Chapter 4

C++言語入門 III

4.1 お絵書き

プログラムを書いて実行しても、でるものが数字ばかりではつまらない。せっかく再帰をやったので、再帰を使ってわりあい簡単なプログラムで複雑な図形を書いてみよう。

```
 1:// tree.cpp
 2:#define _USE_MATH_DEFINES
 3:#include <iostream>
 4:#include <cmath>
 5:#include <gr.h>
 6:using namespace std;
 7:
 8:void draw(double x1, double y1, double x2, double y2)
 9:{
10:    double xa[] = {x1,x2};
11:    double ya[] = {y1,y2};
12:    gr_polyline(2,xa,ya);
13:}
14:
15:void tree(int n, double x, double y, double angle, double length)
16:{
17:    double x1, y1;
18:    if (n >0){
19:        x1 = x + length*cos(angle);
20:        y1 = y - length*sin(angle);
21:        draw(x,y,x1,y1);
22:        tree(n-1,x1,y1,angle + 0.25, length*0.7);
23:        tree(n-1,x1,y1,angle - 0.25, length*0.7);
24:    }
25:}
26:
27:int main()
28:{
29:    int n;
30:    cerr << "Level of the tree? :";
```

```
31:    cin >> n;
32:    gr_setwindow(0,1,0,1);
33:    tree(n, 0.5, 0.0, -M_PI/2, 0.25);
34:    cerr <<"Enter some char + retern to finish:";
35:    char c; cin >>c;
36:}
```

これは、樹形図を書くプログラムである。原理は、指定した長さと角度で一本線を引き、その先から角度をつけて 2 本線を引き、それぞれの端からまた角度をつけて、、、というのを繰り返すだけである。

再帰を使うことで、角度、長さを変えて自分自身を呼び出すという形で簡潔にアルゴリズムが表現できている。

4.2 コンパイル・実行の方法

まず、このプログラムのコンパイルには

- GR framework

がインストールされている必要がある。

インストールは *C* ライブラリのインストールページ[1]から、Ubuntu なら *gr-latest-Ubuntu-x86_64.tar.gz*[2] を落としてきて、それをどこか適当なディレクトリで tar xzf とかで展開する。そうすると、gr というてディレクトリと、その下に lib とか include とかのサブディレクトリができる。そのディレクトリの名前を GRDIR に設定して。bash なら

```
export GRDIR=/foo/var/gr
```

csh 系なら

```
setenv  GRDIR /foo/var/gr
```

でよい (/foo/bar は実際のディレクトリの名前に置き換えること)。あとは

```
export GKS_WSTYPE=x11
```

を実行してから、*Getting Started*[3]にあるような作業をすればよい。

```
g++  tree.cpp -o tree   -std=c++11 -I $GRDIR/include -L  $GRDIR/lib -Wl,-rpath,$GRDIR/lib -lGR
```

でコンパイルし、

```
./tree
```

で実行できるはずである。上手くいけば図 4.1 のような絵がでてくるはずである。なお、環境変数 GKS_WSTYPE を x11 にしないと何もでないかもしれない。

[1] https://gr-framework.org/c.html#installation
[2] https://gr-framework.org/downloads/gr-latest-Ubuntu-x86_64.tar.gz
[3] https://gr-framework.org/c.html#getting-started

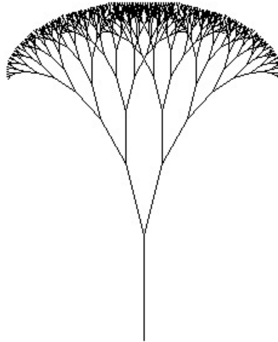


Figure 4.1: tree の出力例

4.3 グラフィックライブラリ

ここでは、 GR framework の基本的な機能を使って画面に線を書いている。

とりあえず、ここで使っている関数についてそれらが何をしているかを見ておこう。上で述べたように、

```
#include "gr.h"
```

によって、 GR framework で提供されている関数が使えるようになる。iostream は前にでてきたが、cmath は sin, cos 等の数学関数と、π のような数学でよく使われる定数を使えるようにする。その上の

```
#define _USE_MATH_DEFINES
```

がないと定数のほうは使えるようにならない。これで、 M_PI を π の値が必要なところで使うことができる。

さて、その次で

```
void draw(double x1, double y1, double x2, double y2);
```

という関数を定義している。この中身の解説はちょっとあとまわしにして、まずはこれは点 (x1,y1) から 点 (x2,y2) まで線を引くように、 GR の機能を使って作った関数である、と思ってほしい。

tree は draw を使って樹形図を書く関数なので後回しにして、 main 関数で使っている GR の機能をみていく。といっても今回

```
  gr_setwindow(0,1,0,1);
```

だけで、これは、画面（ウインドゥ）内での座標系を決めている。 4 つの引数は x 座標の最小値、x 座標の最大値、y 座標の最小値、y 座標の最大値である。

同工異曲だが、こんなものも書ける。

```
 1:// fract.cpp
 2:#define _USE_MATH_DEFINES
 3:#include <iostream>
 4:#include <cmath>
 5:#include <gr.h>
 6:using namespace std;
 7:
 8:void draw(double x1, double y1, double x2, double y2)
 9:{
10:    double xa[] = {x1,x2};
11:    double ya[] = {y1,y2};
12:    gr_polyline(2,xa,ya);
13:}
14:
15:
16:void fractal(int n,
17:       double x0,
18:       double y0,
19:       double x1,
20:       double y1)
21:{
22:    double dx, dy, xa, ya;
23:    if (n > 0){
24:  dx = (x1 - x0)/2;
25:  dy = (y1 - y0)/2;
26:  xa =  x0 + dx  - dy ;
27:  ya =  y0  + dx +dy ;
28:  fractal(n-1,x0, y0, xa, ya);
29:  fractal(n-1,xa, ya, x1, y1);
30:    }else{
31:  draw(x0,y0,x1,y1);
32:    }
33:}
34:
35:int main()
36:{
37:    int n;
38:    cerr << "Level of the tree? :";
39:    cin >> n;
40:    gr_setwindow(0,1,0,1);
41:    fractal(n, 0.6, 0.3, 0.6, 0.7);
42:    cerr <<"Enter some char + retern to finish:";
43:    char c; cin >>c;
44:}
```

なお、画像をファイルに落とすには、環境変数 GKS_WSTYPE を例えば eps にすれば eps ファイルが、 png, jpg 等でそれぞれのタイプのファイルが、gks.png みたいな名前でできる。

4.4 練習

1. この木では枝が 2 本ずつ出ているが、もっとたくさん出すにはどうすればいいか？
2. この木は枝分かれが完全に規則的で不自然である。自然にするにはどうすればいいだろうか？（ヒント：drand48() という関数を呼ぶと、0 から x までの範囲の乱数 — デタラメな数 — を返す。これを使って枝分かれの長さ、角度を変えて見よう)
3. 他のフラクタル曲線、たとえばコッホ曲線やペアノ曲線を書くプログラムを作ってみる。

これで C++ 言語のミニマムな入門は終わり。

Chapter 5

放物型方程式

物理・天文で出てくる偏微分方程式は大抵は 2 階である。で、普通は空間 3 次元と時間 1 次元の 4 次元空間で定義されるわけだが、この講義ではいくつかの特殊な場合を除いて空間 1 次元時間 1 次元の 2 次元で考える。これは、そうしておかないとプログラムを書くのも、また計算機のほうも大変になるからである。

放物型方程式ってのは何か？というのは皆さん知っているはずなので（だよね？）厳密な定義は省くが、要するに以下の形に書ける（移流）拡散方程式のことである。

$$\frac{\partial u}{\partial t} = -K\frac{\partial u}{\partial x} + D\frac{\partial^2 u}{\partial x^2} \tag{5.1}$$

ここで x, t はそれぞれ空間、時間を表す変数であり、 u は方程式が記述する量である。拡散方程式として見ればなにかの濃度ということになるし、熱伝導の方程式としてみれば温度なりエンタルピーなりということになる。K は一次の係数で、これは普通の拡散方程式や熱伝導方程式では 0 である。これが 0 でないのはどういう場合かというのは後でちょっと考える。 D が普通の拡散係数ということになる。今は D が空間・時間に依存しない場合を考える。

依存しない場合はもちろん変数分離で解けるので、数値計算することに意味があるのは D が空間・時間に依存する場合だが、まあ、とりあえずは答がわかる場合を計算してみることにしよう。

放物型方程式の場合には、初期条件と境界条件を与えないと解が定まらない。以下では、

$$\begin{aligned} \frac{\partial u}{\partial t} &= D\frac{\partial^2 u}{\partial x^2} \\ 0 < x < 1, \quad 0 < t, \quad u(0,t) &= u(1,t) = 0, \quad u(x,0) = u_0(x) \end{aligned} \tag{5.2}$$

という固定境界の場合を考える。

5.1 差分法

偏微分方程式の数値計算の方法は基本的には有限要素法と差分法に大別される。あ、あとスペクトラル法というのもあるが、これはちょっとおいておく。有限要素法の考え方はだいぶややこしいので、まず差分法を考える。

差分法とは、基本的には微分方程式に出てくる空間微分や時間微分の項を、差分で置き換えるものである。差分というのはそもそも何かというのを説明しないといけない。

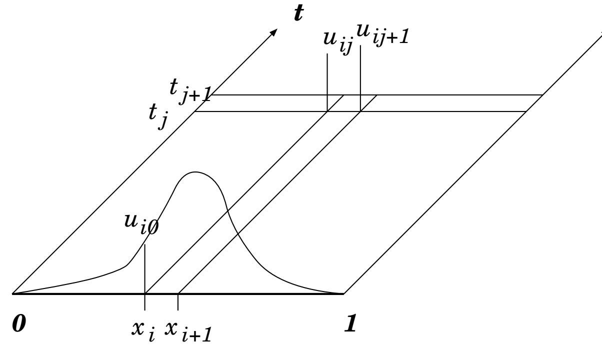


Figure 5.1: 差分法の考え

例えば区間 $[0,1]$ を n 等分して、 x_0 から x_n まで ($x_i = i\Delta x, \Delta x = 1/n$) で表すとする。時間も同様に、 ¡$t_j = j\Delta t$¿ として、これから求める近似解 $\hat{u}$ を簡単のため単に u_{ij} と書く。だいぶ記号が繁雑になるので、図 5.1 に様子を書いておく。

ここで、差分とは例えば $u_{i+1,j} - u_{i,j}$ といった具合に隣との差をとることである。微分を差分で近似するには、例えば

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + O(\Delta x^2) \tag{5.3}$$

というようなことになる。

なお、差分のとり方は一意的ではないことに注意して欲しい。例えば、$(u_{i,j} - u_{i-1,j})/\Delta x$ でもいいし、$(u_{i+1,j} - u_{i-1,j})/2\Delta x$ といったものも考えられる。さらに、もっと沢山の点を使うことも原理的には可能である。

さて、拡散方程式なので 2 階微分がいる。これはどう作ればいいかというわけだが、これはむしろ一般に高階微分を差分で表す、つまり数値微分の方法を説明しておくほうが簡単であろう。

$x_{i-k}, x_{i-k+1}, ...x_i, ...x_{i+l}$ の $k+l+1$ 点での関数値 $u_j (i-k \le j \le i+l)$ をつかって、点 x_i での m 階導関数の近似値を求める考え方は以下のようなものである。

1. 各点で $p(x_j) = u_j$ を満たす $k+l$ 次補間多項式 $p(x)$ を作る。
2. $p(x)$ を m 回微分する。
3. 微分した結果の式に x_i を代入する。

2 階導関数を作るためには最低 3 点いるので、まずもっとも簡単な以下の形を使ってみる。

$$\frac{\partial^2 u}{\partial x^2} \sim \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{5.4}$$

で、時間微分についても、もっとも安直な形

$$\frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{5.5}$$

を使ってみる。

5.2 プログラム

初期条件は $x = 0.5$ のところにだけ値がある δ 関数的なものということにする。
なお、面倒なので $D = 1$ ということにする。時間の単位が $1/D$ と思えば同じことである。

```
 1://     program parabolic1
 2:
 3:#define _USE_MATH_DEFINES
 4:#include <iostream>
 5:#include <unistd.h>
 6:using namespace std;
 7:#include <cmath>
 8:#include <gr.h>
 9:
10:void initparam(double  u[],
11:               int & nx,
12:               double &dx,
13:               double & dt,
14:               int & niter,
15:               int &gmode)
16:{
17:    cerr << "Enter nx, tmax, dt:";
18:    double tmax;
19:    cin >> nx >> tmax >> dt;
20:    dx = 1.0/nx;
21:    niter = (tmax+dt/2)/dt;
22:    cerr << "Enter graphics mode 0: no G\n"
23:         << "                   1: animation\n"
24:         << "                   2: no-erase)\n";
25:    cin >> gmode;
26:    cout << "nx = " << nx<< " dt = " << dt << " niter = "<< niter<< "\n";
27:    cout << "gmode = " <<  gmode << "\n";
28:    int i;
29:    for (i=0;i<=nx; i++) u[i] = 0;
30:    u[(nx+1)/2] = nx;
31:}
32:
33:void push_system(double u[],
34:                 double unew[],
35:                 int nx,
36:                 double dx,
37:                 double dt)
38:{
39:      double lambda = dt/(dx*dx);
40:      int i;
41:      for (i=1; i<nx; i++){
42:          unew[i] = u[i] + lambda*(u[i-1]-2*u[i]+u[i+1]);
43:      }
44:      for(i=1; i<nx; i++){
45:         u[i] = unew[i];
46:      }
```

```
47:}
48:
49:
50:void show_graphics(double u[],
51:                   int nx,
52:                   double dx,
53:                   int gmode,
54:                   double xmin,
55:                   double xmax,
56:                   double ymin,
57:   double ymax,
58:                   int first)
59:{
60:    if (gmode == 0) return;
61:    double x[nx+1];
62:    if (gmode == 1) {
63:gr_clearws();
64:    }
65:    int i;
66:    for(i=0;i<nx+1;i++){
67:x[i]=(i*dx);
68:    }
69:    gr_setwindow(xmin,xmax,ymin,ymax);
70:    gr_axes(0.25*(xmax-xmin), 0.25*(ymax-ymin), 0, 0, 1, 1, -0.01);
71:    gr_polyline(nx+1,x, u);
72:    if (gmode == 1) usleep(30000);
73:    gr_updatews();
74:}
75:
76:int main()
77:{
78:#define NMAX 10001
79:    static double u[NMAX], unew[NMAX];
80:    double dx, dt;
81:    int nx, niter, gmode, iter;
82:    initparam(u, nx, dx, dt, niter, gmode);
83:    show_graphics(u, nx, dx, gmode, 0.0, 1.0, 0.0, 10.0, 1);
84:    for(iter = 1; iter <=niter; iter++){
85:         push_system(u, unew, nx, dx, dt);
86:         show_graphics(u, nx, dx, gmode,0.0, 1.0, 0.0, 10.0, 0);
87:    }
88:    cerr << "Enter any key to finish:";
89:    char c;
90:    cin>>c;
91:    return 0;
92:
93:}
```

これをちょっと動かしてみよう。

コンパイルには

```
g++  parabolic1.cpp -o parabolic1   -std=c++11 -I $GRDIR/include\
  -L  $GRDIR/lib -Wl,-rpath,$GRDIR/lib -lGR
```

(既に書いた通り、 -I や -l 以下はライブラリをインストールした場所等に依存する)

なお、環境変数 GKS_WSTYPE を mp4 にすることで動画ファイルを作成できる。また、画面に出す時には環境変数 GKS_DOUBLE_BUF を設定する (値は 1 とか true とか適当に) と、画面のチラツキがなくなる。

5.3 安定性

ここまでで、拡散方程式に対するもっとも簡単な差分近似である、空間微分

$$\frac{\partial^2 u}{\partial x^2} \sim \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{5.6}$$

と、時間微分

$$\frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{5.7}$$

の組合せをプログラムにしてみた。これを動かしてみた人は気がついたと思うが、この方法は $\Delta t/\Delta x^2 > 0.5$ になると「うまく動かない」。

具体的には、拡散方程式なので本来は次第に滑らかになっていくべきなのに、どんどんガタガタになってしまう。しかも、これは $\Delta t/\Delta x^2 > 0.5$ であれば Δx や Δt をどんなに小さくしても起こる。

まず、なぜそういう振動が起きるのかを考え、それから振動を起こさない方法があるのかどうかを議論していこう。

5.3.1 線形安定性解析

さて、まず、なぜ振動が起きるかということだが、これは原理的には差分化された方程式の固有値を求めればわかる。

つまり、 u_j を時刻 $t_j = j\Delta t$ での数値解をベクトルとして書いたものだとすると、差分近似は、

$$u_{j+1} = Au_j \tag{5.8}$$

という形に書ける。ここで A は $(n-1)$ 次元正方行列で、その各要素は差分近似の形から決まる。この場合には、対角要素とその両側一列以外の全ての要素が 0 である 3 重対角行列といわれる形をしている。要素の値は

$$a_{ii} = 1 - 2\delta \quad (0 < i < n) \tag{5.9}$$

$$a_{i,i-1} = a_{i-1,i} = \delta \tag{5.10}$$

ここで、 $\delta = \Delta t/\Delta x^2$ である。

この行列は実対称行列なので固有値分解ができて、適当なユニタリ行列 U を持ってきて

$$A = U^{-1}\Lambda U \tag{5.11}$$

と書ける。ここで Λ は固有値行列、つまり対角要素が固有値でそれ以外が 0 の行列である。

このように分解できるので、 $\boldsymbol{v} = Uu$ とすれば最初の差分近似式は

$$\boldsymbol{v}_{j+1} = \Lambda \boldsymbol{v}_j \tag{5.12}$$

つまり、

$$v_{i,j+1} = \lambda_i \boldsymbol{v}_{ij} \tag{5.13}$$

という $n-1$ 個の独立な方程式になって、解も自明に求まる。いうまでもないが、 U の各列は A の固有ベクトルになっている。つまり、 U の i 行目を $\boldsymbol{x}_i$ と書くと、

$$\lambda \boldsymbol{x}_i = A\boldsymbol{x}_i \tag{5.14}$$

である。

不安定性が起きないための条件は、全ての固有値 λ_i の絶対値が 1 を超えないということである。

5.3.2 固有値を求める

さて、そういうわけで、固有値 λ がどうなっているか調べてみよう。式 5.14 をもともとの差分近似に入れてちょっと変形すると、

$$(\lambda + 2\delta - 1)u_i = \delta(u_{i+1} + u_{i-1}) \tag{5.15}$$

となる。ここで $\delta = \Delta t/\Delta x^2$ である。さらに、

$$\alpha = \frac{\lambda + 2\delta - 1}{\delta} \tag{5.16}$$

とおいてもうちょっと変形すると、結局

$$u_{i+1} - \alpha u_i + u_{i-1} = 0 \tag{5.17}$$

となる。

断るまでもないと思うが、これは元の偏微分を変数分離して空間方向の関数についての常微分方程式を導くのとほとんど同じ操作になっている。これを u_i についての差分方程式と見て解を求めることを考える。

線形差分方程式なので、解は $u_i = Cp^i$ の形である。これを代入すると

$$p^2 - \alpha p + 1 = 0 \tag{5.18}$$

解は

$$p = \frac{\alpha \pm \sqrt{\alpha^2 - 4}}{2} \tag{5.19}$$

となる。

真面目にやるには境界条件を満たす固有ベクトルをもとめないといけないが、面倒なので無限遠境界の場合を考える。この時、 p が実数のものはどちらかで無限大に発散するのでよろしくないので、複素数の場合を考える。この時は $|p| = 1$ になっているので無限遠でも発散しない。

なお、有限の固定境界の場合も結局境界条件を満たすような解を作るためには p が複素数でないといけないことがすぐにわかる。このため、 $|\alpha| < 2$ だけを考えればよい。

このとき、式 (5.16) は

$$\lambda = 1 - (2 - \alpha)\delta \tag{5.20}$$

と書き直せる。 $\delta > 0$ なので、 $|\alpha| < 2$ なるある α について $|\lambda| < 1$ であるためには、結局

$$\delta < \frac{2}{2 - \alpha} \tag{5.21}$$

を満たせば良く、任意の α についてなり立つためには

$$\delta < \frac{1}{2} \tag{5.22}$$

つまり

$$\frac{\Delta t}{\Delta x^2} < \frac{1}{2} \tag{5.23}$$

であればいいことになる。

上でやったことは、結局空間方向をフーリエ級数展開して、各空間波長に対する時間発展が安定である（減衰していく）ことを要求しているのと同じである。このようなやり方を von Neumann の方法による安定性解析という。

5.3.3 「直観的」説明

上の議論からわかるように、不安定条件に関係する α の値は実際には $\alpha = -2$ だけで、それ以外の α からはもっと緩い条件しかでない。 $\alpha = -2$ は $p = -1$ に対応するので、これは結局 u_i と u_{i+1} で値が逆転するようなパターンである。そのようなパターン（モード）が 1 ステップ計算するとどうなるかをもう一度もとの差分式に戻って書き直してみると、結局

$$u_{i,j+1} = (1 - 4\Delta t/\Delta x^2)u_{ij} \tag{5.24}$$

となる。括弧内の絶対値が 1 より大きいと、パターンがどんどん成長していってめちゃくちゃな答になるわけである。括弧内の絶対値が 1 より小さいためには、

$$\frac{\Delta t}{\Delta x^2} < \frac{1}{2} \tag{5.25}$$

であればよく、前節の結果と同じである。このように、偏微分方程式の場合には大抵もっとも波長の短いモードが減衰できるかどうかで安定性が決まる。

5.4 陰解法

前節の議論から、初めに作ったプログラムでまともな答が求まるためには、$\Delta t/\Delta x^2 < 1/2$ でないといけないということがわかった。これは結構厳しい条件である。というのは、空間刻みの数を増やすと、その 2 乗に比例して時間刻みの数を増やさないといけないので、全体としては空間刻みの数の 3 乗に比例して計算時間がかかるということになるからである。まあ、最近の計算機は速いので待てなくもないかもしれないが、もうちょっとなんとかならないかという気もする。

実は、なんとかなる方法というのはある。これが陰解法 (implicit method) というものである。拡散方程式の場合、もっとも簡単な陰解法は、空間微分に u_j ではなく u_{j+1} のほうを使う、つまり、

$$\frac{\partial^2 u}{\partial x^2} \sim \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{\Delta x^2} \tag{5.26}$$

というものを使うというだけである。時間微分は同じものですます。差分公式としてまとめて書くと、

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{\Delta x^2} \tag{5.27}$$

となる。最初の方法では、 $u_{i,j+1}$ は左辺に単独で現れ、右辺は時刻 j の項だけだったので、 $u_{i,j+1}$ を直接計算できた。このように、直接計算できる方法のことを陽解法 explicit method という。これに対し、上に書いた方法では時刻 $j+1$ の項が右辺に複数現れているので、これは全体として u_{j+1} についての線形連立方程式になっている。このように、新しい時刻での解が代数方程式を解かないと求められない方法のことを陰解法という。

線形連立方程式を解かないといけないのはもちろん面倒なわけだが、それだけのことはある。先ほどと全く同じように δ とか λ を定義すると、両者の関係が今度は

$$\lambda = \frac{1 + \alpha\delta}{1 + 2\delta} \tag{5.28}$$

となる。したがって、 $|\alpha| < 2$ なる任意の α と $\delta > 0$ なる任意の δ について、 $|\lambda| < 1$ が満たされているのである。つまり、どんなに Δt を大きくとっても、決して不安定にならない。この性質を無条件安定という。この場合には、安定性ではなくて計算精度の観点から Δt を決められることになり、だいぶましである。

5.5 3 重対角行列を解く

陽解法と同じように行列で書いてみると、

$$Au_{j+1} = u_j \tag{5.29}$$

で、

$$a_{ii} = 1 + 2\delta \tag{5.30}$$

$$a_{i,i-1} = a_{i-1,i} = -\delta \tag{5.31}$$

ということになる。 A は δ の符号が違うだけで前と同じ三重対角行列になっている。行列（以下、線形連立方程式のことを単に行列ということがある）を解くのは例えばガウスの消去法では計算量

が一般には次元の 3 乗になるので嬉しくないが、三重対角の場合には計算量が次元数に比例する。前進消去で消さないといけない項が常に 1 つだけだからである。

具体的には、 a_{ii} を ad, a_{ii-1} を al, a_{ii+1} を ad、右辺の値を b と表して、前進消去の部分が、

```
        double c = al[i]/ad[i-1];
        ad[i] -= c*au[i-1];
        b[i] -=  c*b[i-1];
```

というだけである。後退代入も、 au のところだけを消せばいいので

```
        b[i] = (b[i]-au[i]*b[i+1])/ad[i];
```

というふうに一つ下の値を引くだけになる。

一応、3 重対角行列を解くプログラム全体を与えておくと、

```
void solve_tridiagonal(double ad[],
                       double al[],
                       double au[],
                       double b[],
                       int n)
{
    int i;
    for(i=1;i<n;i++){
        double c = al[i]/ad[i-1];
        ad[i] -= c*au[i-1];
        b[i] -=  c*b[i-1];
    }
    b[n-1] /= ad[n-1];
    for(i=n-2;i>=0;i--){
        b[i] = (b[i]-au[i]*b[i+1])/ad[i];
    }
}
```

となる。

係数の ad 等を計算して、1 ステップ進める関数全体は、

```
void push_system(double u[],
                 double ad[],
                 double au[],
                 double al[],
                 int nx,
                 double dx,
                 double dt)
{
      double delta = dt/(dx*dx);
      int i;
      for(i=0;i<nx-1;i++){
          ad[i] = 1 + 2*delta;
          al[i] = au[i] = -delta;
      }
```

```
        solve_tridiagonal(ad, al, au, u+1, nx-1);
 }
```

である。

5.6 高精度の方法

さて、前節の簡単な陰解法は無条件安定で大変結構なのであるが、時間刻みを大きくとれるとなると、今度は計算精度のほうが気になってくる。空間方向の微分は 2 次精度で、テイラー展開の 2 次の項までが正しく入っているのに対し、時間微分は 1 次精度にしかなっていないからである。

なお、上の 1 次の陽解法のことを 1 次の前進差分、陰解法のことを 1 次の後退差分ということがある。

時間方向の精度をあげる一つの方法は、これまでに見てきた 2 つの方法を混ぜて使うことである。つまり、

$$\frac{u_{i,j+1}-u_{i,j}}{\Delta t} = \frac{u_{i+1,j+1}-2u_{i,j+1}+u_{i-1,j+1}}{2\Delta x^2} + \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{2\Delta x^2} \tag{5.32}$$

この方法をクランク・ニコルソン法という。これは時間方向に対称な形になっている。この方法では、時間方向も 2 次精度にすることができる。

5.7 練習

1. メインプログラムを付け加えて、1 次の陰解法のプログラムを完成し、 $\Delta t/\Delta x$ をいろいろ変えて安定に計算できることを確認せよ。
2. クランク・ニコルソン法が無条件安定であることを証明せよ。
3. クランク・ニコルソン法のプログラムを作り、上と同様に安定であることを確認せよ。
4. クランク・ニコルソン法の誤差が $\Delta t, \Delta x$ のそれぞれ何乗になっているか、 $\Delta t, \Delta x$ をいろいろ変えてプログラムを走らせて結果を厳密解と比べることで調べよ。厳密解と比べるには、初期条件がデルタ関数では具合が良くないのでどういうものを使うべきか考えてみること。
5. クランク・ニコルソン法が時間方向 2 次精度であることを証明せよ。
6. 前進差分、クランク・ニコルソン法、後退差分は、 時刻 j と $j+1$ での空間微分の混ぜ方を変えているだけである。これらすべてを統一的に計算できるプログラムを作成し、正しく動くことを確認せよ。
7. さらに、周期境界の場合も扱えるようにプログラムを拡張せよ。3 重対角行列の処理の部分をどう変えればいいか？

次は楕円型方程式。

Chapter 6

楕円型方程式

楕円型方程式は、典型的にはラプラス方程式

$$\nabla^2 u = 0 \tag{6.1}$$

または ポアソン方程式

$$\nabla^2 u = -4\pi G\rho \tag{6.2}$$

のようなものである。上のポアソン方程式はニュートン重力の場合で、 G は重力定数、 ρ は物質の質量密度（単位体積あたりの質量）である。ここで、 ρ はとりあえず与えられた空間座標だけの関数とする。空間 1 次元ではラプラシアンがただの 2 階微分演算子になってしまい、解くべき方程式は常微分方程式ということになる。

実際にポアソン方程式（とさらに状態方程式を連立させてでてくるレーン・エムデン方程式）を球対称の場合に解くというのは I のほうでやったと思うので、ここでは空間 2 次元の場合を考える。3 次元も考え方は同様である。

工学的な応用では、楕円型方程式の応用として重要なのはラプラス方程式の境界値問題である。これは、構造解析、電場の解析などあらゆるところで現れる。特に複雑な形状の機械部品とか機械全体の中での応力場の解析といったものが重要な応用ということになる。

が、天文・物理のいろんな問題では、複雑な境界値問題というのはあんまり大事ではない。天体は大抵重力でまとまっているわけで、固定された表面（境界）というものが与えられているとは限らないからである。もちろん、数値計算の方法によってはそういうものが出てくることもあるが、とりあえず今日は比較的に単純な境界条件だけをかんがえることにしよう。

ラプラス方程式やポアソン方程式は、基本的には拡散方程式から時間微分の項を落したものと考えることができる。つまり、なんらかの定常状態を記述しているわけである。多くの応用で、実際にそのような定常状態を求めているわけであるが、数値計算という観点からすると、ラプラス方程式やポアソン方程式に適用できる計算法は実は拡散方程式や波動方程式に陰解法を適用する時にも使える。

これは、前回見たように陰解法では空間方向の差分方程式を解く必要がでてくるからである。解くべき方程式は、ラプラス方程式やポアソン方程式の差分化からでてくるものと基本的には同じである。

というわけで、今日は、単純な 2 次元ポアソン方程式の境界値問題を例に、計算方法について考えていく。

2 次元なので、ラプラシアンは

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{6.3}$$

である。例によって変数の方を u_{ij} と書き、これが $u(x_i, y_j)$ の近似解としよう。また、前回と同じ 2 次精度の中心差分を使うとすれば、ラプラシアンの差分近似は以下で与えられる。

$$\frac{u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{ij}}{\Delta x^2} \tag{6.4}$$

但し、ここでは $\Delta x = \Delta y$ とした。いま、解くべき問題を長方形領域でのポアソン方程式の境界値問題

$$\nabla^2 u(x, y) = \rho(x, y), \quad (x_0 \le x \le x_1, y_0 \le y <\le y_1) \tag{6.5}$$

$$u(x_0, y) = u(x_1, y) = u(x, y_0) = u(x, y_1) = 0 \tag{6.6}$$

ということにすれば（以下、面倒なので $-4\pi G$ の項は 1 とする）、これは 2 次元配列 u[nx][ny] についての線形連立方程式ということになる。ここで

$$n_x = (x_1 - x_0)/\Delta x + 1 \tag{6.7}$$

$$n_y = (y_1 - y_0)/\Delta x + 1 \tag{6.8}$$

であり、境界上では $u_{ij} = 0$, それ以外では

$$\frac{u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{ij}}{\Delta x^2} = \rho_{ij} \tag{6.9}$$

となる。

6.1 線形方程式と反復法

さて、というわけで、2 次元の長方形領域上でのポアソン方程式の差分近似は、連立線形 1 次方程式 (6.9) になるので、これを何らかの方法で解けばいい。

空間 1 次元の場合には、出てくる行列が 3 重対角という特別な性質を持ったものであったので、自由度（要素数）に比例する計算量やメモリー使用量で方程式を解くことができた。

2 次元の場合はどうなるかというと、そんなにうまくはいかないということがわかる。もちろん、2 次元の場合でも、 $u_{ij} = u_{i \cdot n_x + j}$ というように添字をふりなおして (以下、添字が 1 つしかないときにはこのように付け変えたものを考えるということにする)、以下のように行列の形に書くことはできる。

$$Au = \boldsymbol{\rho} \tag{6.10}$$

ここで、 u は u_i のベクトル表示である。しかし、 A の要素を考えてみると、対角要素、その両側の他に、$\pm n_x$ だけ離れたところに 0 でない要素が現れる。

このために、計算量としては $O(n_x^2 n)$ ($n = n_x n_y$ として) になり、1 次元のときよりずっと計算量が多い。例えば $n_x = n_y$、つまり正方形領域とすれば計算量は結局 $O(n^2)$ になってしまう。

もちろん、これでも一般の場合のガウスの消去法の計算量である n^3 よりは少ないが、しかし非常に多いことには変わりはない。また、3 次元の場合にはこの状況はいっそう悪化する。

このために、ガウスの消去法のようにまともに行列を解くのではなく、適当な近似を繰り返していくことで解を求めることはできないかということが昔から考えられてきた。以下、その方法のいくつかを紹介する。これらを反復法という。

6.1.1 ガウス反復

なんらかの方法で適当な近似解 u^i （ここで上つき添字 i は i 番目の近似解ということで、別に u の i 乗という意味ではない。下につけると他の添字と混乱するので上につけてみた）があるとする。これをもうちょっとましな解にできないかということを考える。

一つの方法は、行列 A を形式的に以下のように分解することである。

$$A = D + N \tag{6.11}$$

ここで、 D は対角要素 a_{ii} だけをとりだした行列で、 N はそれ以外の残りである。これを使って、以下のような式を考える

$$Du^{i+1} = -Nu^i + \boldsymbol{\rho} \tag{6.12}$$

D は対角行列なので、これは解けていて、実際のプログラムとしては、

```
for (i=1; i<nx-1;i++){
    for (j=1; j<ny-1;j++){
        unew[i][j] = 0.25*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j])
                     - rho[i][j]*0.25*deltax*deltax;
    }
}
```

ということになる。1 で始まって $n_x - 2$ 等で終っているのは、境界は 0 固定だからである。その分の配列を節約することも考えられるが、いじましいしちょっとわかりにくくなるので上の形に書くことにしよう。

この反復で、収束すれば、つまり、 $u^{i+1} = u^i$ となれば、これらが元の方程式を満たしていることは明らかであろう。収束するかどうかは面倒なのでここでは省くが、上のような簡単なポアソン方程式の離散化で妙な非線形項とかがなければ収束するということが証明できる。

実用上は、収束したかどうか判定する必要がある。これには、普通は $|u^{i+1} - u^i|^2$ を計算してそれが適当な設定した値よりも小さくなったらいいことにする。

理論的には、数列は連続する 2 項間の差が小さくなっても収束に近付いているとは限らないが、ガウス反復の場合には、丸め誤差がなければこれで判定できることが証明できる。

が、この方法には 2 つ問題がある。一つは収束が非常に遅いことであり、もう一つは u と unew で配列が 2 ついるのでメモリがたくさん必要になることである。

6.1.2 ガウス・ザイデル法

さて、実は、上の 2 つの問題は一度に解決することができる。上のプログラムを、以下のように変えてしまうのである。

```
for (i=1; i<nx-1;i++){
    for (j=1; j<ny-1;j++){
        u[i][j] = 0.25*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j])
                  - rho[i][j]*0.25*deltax*deltax;
    }
}
```

なにをしたかというと、 unew というのをやめにして u にしただけである。つまり、一回の反復の中では u の要素を順に変えていくわけだが、その時に、既に新しい値が求まっていればそっちを使おうというだけのことである。新しい値のほうが、多分より正確であろうと考えられるからである。

形式的には、これは以下のように書くことができる。 行列 A を、対角成分 D、非対角成分の上半三角分 U、残り L の 3 個に分ける ($A = D + L + U$) と、

$$(L + D)u^{i+1} = -Uu^i + \boldsymbol{\rho} \tag{6.13}$$

ということになる。

6.1.3 収束の速さ

収束の速さをガウス反復について検討する。面倒なので 1 次元として、

$$\frac{d^2u}{dx^2} = \rho \tag{6.14}$$

に対する差分近似

$$u_{i,\mathrm{new}} = 0.5(u_{i-1} + u_{i+1}) - 0.5\rho\Delta x^2 \tag{6.15}$$

を考える。ここでは、収束の速さが問題なので、 $\rho = 0$ としてよくて、

$$u_{i,\mathrm{new}} = 0.5(u_{i-1} + u_{i+1}) \tag{6.16}$$

である。さて、これは線型の式なので固有値と固有ベクトルがあって、ある固有値 λ と固有ベクトル u_λ が

$$\lambda u_{\lambda,i} = 0.5(u_{\lambda,i-1} + u_{\lambda,i+1}) \tag{6.17}$$

という関係を満たすと考えられる。固有関数がどういうものかは自明ではないが、等間隔格子なのでフーリエ級数解が固有関数になっていることを期待すると、境界条件もいれて

$$u_{\lambda,k} = \exp(\pi lki\Delta x) \tag{6.18}$$

である。ここからの収束の議論では i は虚数単位で k が格子のインデックスとする。 l はフーリエモードのインデックスである。これを 6.17 にいれて整理すると

$$\lambda = \cos \pi l\Delta x \sim 1 - (\pi l\Delta x)^2/2 \tag{6.19}$$

となることがわかる。これは、 Δx を与えると $l = 1$ の時にもっとも 1 に近く、収束が遅いことがわかる。また、反復回数は $1/\Delta x^2$ 程度必要であることもわかる。

なお、式 6.16 は、前節で扱った拡散方程式の陽解法差分近似とみなすこともできることに注意して欲しい。なので、上のように天下りに固有関数を仮定しなくても、前節と同様に線型差分方程式であることから解を求めることもできる。線形差分であることから指数関数型に限られるため、周期解はフーリエ級数解なので、2 つの方法は数学的には等価である。

また、陽解法の安定性条件を満たすように、$\Delta t \sim \Delta x^2$ となっている。収束までは時間が 1 より十分大きくなる必要があるので、反復回数が $1/\Delta t \sim 1/\Delta x^2$ 程度必要となる。

ガウス・ザイデル法についても同様に必要な反復回数を求めることができる。次に述べる SOR についても (多分) できる。

6.1.4 SOR

というわけで、ガウス・ザイデルはガウス反復よりはましだが、まだもうちょっとなんとかならないものかという気もする。実際に近似解の挙動を見るとわかるが、どちらの方法でも、誤差の減り方にはパターンがあって、1 方向から真の解に近付いている。

というわけで、以下のようなことが考えられる：ガウス・ザイデル法で、修正量 $\Delta u = u^{i+1} - u^i$ に適当な係数 ω を掛けて水増ししてやればもうちょっとうまくいくのではないか？

こんな安直な方法で大丈夫なのかと思うであろうが、うまくいってしまうのがすばらしいところである。プログラムとしては、ガウス・ザイデルの時の式をちょっと変えて、

```
u[i][j] += omega*(0.25*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j])
               - rho[i][j]*0.25*deltax*deltax -u[i][j])
```

というだけで済む。実際的な問題は、ω の値をどうとるかということである。理論的には $1 \le \omega < 2$ でないといけないことがわかっていて、さらに本当はもちろん線形のポアソン方程式とかなら最適な ω の値が計算できるが、ここではそこまでは立ち入らないことにする。

この方法を SOR (Successive Over Relaxation, 本によっては Simultaneous ... となっているものもあるかもしれない) という。

6.2 もっと高度な解法

天文等で出てくるポアソン方程式の場合には、特にうまく ω を決められれば SOR で非常にうまくいくことが多い。が、まあ、世の中はそういう問題ばかりでもないので、それ以外の方法を紹介だけしておく。

6.2.1 ADI

ADI は alternating direction implicit の略である。ここでの基本的な考察は、「3 重対角なら簡単に解ける」ということである。これを使って、x 方向と y 方向を交代で解いているうちになんとかなって欲しいというのが ADI の考え方ということになる。

もうちょっとちゃんと書いてみると、例えば j 方向に解くときには、

$$\frac{u^{new}_{i,j+1} + u^{new}_{i,j-1} - 2u^{new}_{ij} + u_{i+1,j} + u_{i-1,j} - 2u_{ij}}{\Delta x^2} = \rho_{ij} \tag{6.20}$$

となる。ここで、 new とつけたのが更新後の値で、これを $i = 1$ から順に解いていく。 j 方向が終ったら次は i 方向をやる。

ただし、実際に使う上では、 SOR と同じように、修正量の補正をしたほうが収束が速い。 SOR とは逆に修正量をすこし減らす必要があり、どれくらい減らすべきかを決めるのにはまたややこしい理屈がいる。

ただし、適当に修正量を決めても最適な SOR よりも遅くはない程度で収束するという利点がある。

6.2.2 共役勾配法

共役勾配法は、実は、現在大規模な行列を解くのにはもっとも広く使われている方法である。これは、特にいろいろな前処理と組み合わせることで、例えば拡散方程式の定常問題で拡散係数が空間に強く依存する場合にも収束させられるとか、刻みを場所によって変えるような高度な方法でも同様になんとかできるとかいう利点があるからである。

しかし、これは安直にプログラムを書いただけではこれまでの紹介してきた方法に比べて速くなくて、いろいろ高度な変形をして初めて速くなる。また、ほとんど共役勾配法だけについて議論した良い教科書がいくつもあるので、ここでは省略する。ここでは、とりあえずそういう名前のものがあるということくらいをちょっと憶えておいてほしい。

とはいえ、極めて重要な方法ではあるので、これについては、もうちょっと数学的な準備をした後でもう一度やることにする。

6.2.3 FFT を使った方法

ポアソン方程式を解く方法の一つで、応用上は重要なのはフーリエ変換を使った方法である。いま、固定境界をやめて周期境界条件でいいことにすると、ポアソン方程式の場合、 ρ をフーリエ級数展開して、出てきた係数に $1/k^2$ (k は波数) を掛けてから逆フーリエ変換すればポアソン方程式の解になっているわけである。固定境界なら sin 関数だけで展開すればよい。フーリエ変換の計算量は多次元でも格子点の総数を n として $O(n \log n)$ の程度なので、これは反復法でいうと回数がせいぜい $\log n$ で済むということに相当する。

これは極めて強力かつ高速な方法であり、

- 長方形領域でいい
- 周期境界または固定境界でいい
- 空間微分の係数が座標によらない
- 格子が等間隔でよい

という条件がすべて満足されていれば、必ずこの方法を使うべきといってもさしつかえない。

星の構造を解くとかいう場合だと、極座標を使って球面調和関数展開ということも多い。この場合の計算には FFT のようなうまい方法があるわけではない（最近高速ルジャンドル変換の研究もあるが、これはよほど項数が大きくないと速くならない）が、 r 方向は展開しないで済ませられるのでまあつかえなくはない。

6.3 練習

6.3.1 練習 1

ガウス・ザイデル法で以下のポアソン方程式の境界値問題を解くプログラムを作成せよ。ただし、$x_0 = y_0 = 0, x_1 = y_1 = 1$ としてよい（入力パラメータとして読める汎用的なプログラムを作ってく

ればもちろんそのほうがよい)。さらに、 $\rho(x,y)=\sin\pi x\sin\pi y$ として、初期推定 $u=0$ から出発した時に

- 残差 $|Au-\boldsymbol{\rho}|$ が反復を繰り返した時にどのように 0 に近付くか
- 収束した時に、真の解との差（誤差）はどうなっているか

を空間刻み n_x ($n_y=n_x$ としてよい) のいくつかの値について調べ、それから精度が n_x の何乗になっているか議論せよ。

$$\nabla^2 u(x,y) = \rho(x,y), \quad (x_0\le x\le x_1, y_0\le y <\le y_1) \tag{6.21}$$

$$u(x_0,y)=u(x_1,y) = u(x,y_0)=u(x,y_1)=0 \tag{6.22}$$

6.3.2 練習 2

さらに、 SOR について同様に調べよ。最適な ω の値は n_x に依存するかどうかも議論すること。

6.3.3 練習 3

$\rho(x,y)=\sin k\pi x\sin k\pi y$ (k は整数) の時に、誤差はどうなるか、理論的または数値的（両方あればそれも OK）議論せよ。

6.3.4 練習 4

ρ がデルタ関数 $\delta(x-0.5,y-0.5)$ の時に、厳密解をフーリエ級数の形で表せ。

6.3.5 練習 5

上と同じデルタ関数の時に数値解を求めて、 $y=0.5$ の線上で誤差がどのように分布するか調べよ。また、誤差の等高線表示や 3 次元表示を適当なツールを使って作ってみよ。

6.3.6 練習 6

例えば 3 次元的な星の自己重力を計算するためには、境界条件を「無限遠で 0 」という形に置く必要がある。これにはどうすればよいか検討せよ。

この資料では、牧野は実際にプログラムを組んで動かしてみていないので、差分式とかには間違いがある可能性がある。プログラムを書いて動かない時には、資料の式が間違っている可能性も含めて調べること。

Chapter 7

常微分方程式

7.1 ルンゲ・クッタ

$$\frac{dy}{dx} = f(y, x), \quad y|_{x=x_0} = y_0 \tag{7.1}$$

という形の初期値問題を考える。

ルンゲクッタ法とは、非常に一般的には以下のような形に書ける方法である。

$$\begin{aligned} y_{n+1} &= y_n + h\sum_{i=1}^{s} b_i k_i \\ k_i &= f(y_n + h\sum_{j=1}^{s} a_{ij}k_j, x_n + c_i h) \end{aligned} \tag{7.2}$$

自然数 s を段数（number of stages）という。 a_{ij}, b_i, c_i はパラメータであるが、 a と c は普通

$$c_i = \sum_{j=1}^{s} a_{ij} \tag{7.3}$$

となるようにとる。これは、一般にそうでないような公式は不可能ではないがあまりいいことがない（精度がよくならない）からである。

と、こう、式に書いてしまうとすぐにはわからないが、例えば $s = 2$ の場合に書き下してみると

$$\begin{aligned} y_{n+1} &= y_n + h(k_1 b_1 + k_2 b_2) \\ k_1 &= f(y_n + h(a_{11}k_1 + a_{12}k_2), x_n + c_1 h) \\ k_2 &= f(y_n + h(a_{21}k_1 + a_{22}k_2), x_n + c_2 h) \end{aligned} \tag{7.4}$$

と、まあ、こんな感じになる。こちらを良く見ればすぐにわかるように、

1. $a_{ij} \quad (j \geq i)$ がすべて 0 ならば、 k_1 から順に計算していくこと

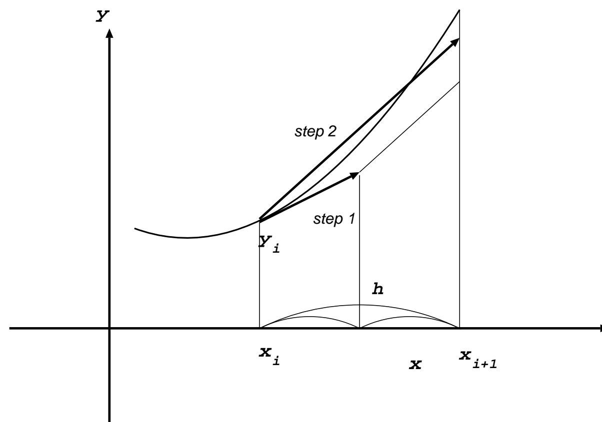


Figure 7.1: 2 次ルンゲクッタ法

ができる。つまり、「陽的」公式になっている。

1. $a_{ij} \quad (j > i)$ が 0 のときは、各 k_i についての式に k_i だけが入ってくる。これを半陰的 (semi-implicit) 公式という。この場合には、まず k_1 についての方程式をといて、次に k_2 についてのを解いて、、、と順番に計算出来る。

2. 上のような制約が全くない時は、「陰的」公式ということになる。このときは、すべての k_i に対する（一般には非線形な）方程式を一度に解く必要がある。

今日はとりあえず陽的な公式だけを考える。形式的には、もっとも簡単な前進オイラー法もルンゲ・クッタ公式の一つということになるが、まあ、普通もっとも簡単な RK 法というと、2 次の公式

$$\begin{aligned} k_1 &= f(y_i, x_i) \\ y_{i+1} &= x_i + hf(y_i + \frac{h}{2}k_1, x_i + h/2) \end{aligned} \tag{7.5}$$

である。この方法がどのように働くかについては、図的な説明というものが可能である。元の点からまずオイラー法と同様に接線を引く。が、これを次の時刻まで延ばすのではなく、ステップの半分のところで止める。で、ここでもう一回微分方程式の右辺を評価する。ここでの導関数の値を使って、もとのところ (t_i, x_i) から直線を引くわけである。

普通は、RK 法というといわゆる古典的 RK 公式

$$\begin{aligned} y_{n+1} &= y_n + h(k_1/6 + k_2/3 + k_3/3 + k_4/6) \\ k_1 &= f(y_n, t_n) \\ k_2 &= f(y_n + hk_1/2, x_n + h/2) \\ k_3 &= f(y_n + hk_2/2, x_n + h/2) \\ k_4 &= f(y_n + hk_3, x_n + h) \end{aligned} \tag{7.6}$$

のことをさす。これは、いろいろ良い性質をもつ。例えば

1. a_{ij} が $i - j = 1$ 以外すべて 0 なので、右辺の計算が楽である。

2. 次数が 4 次であり、4 段陽的公式で到達可能な最高次数を達成している
3. 係数が簡単な有理数なので、プログラムしやすい。また丸め誤差を小さくできる。

この公式が 4 次であることを示すのは、それほど簡単ではない。腕力に自信があるひとは挑戦してみて欲しい。

4 次よりも高精度な公式というのも非常によく研究されていて無数にいろんなものが知られている。ただし、4 次よりも高い次数では、次数よりも必要な段数が大きくなる。この 2 つの関係が一般にどうなるかは未解決の問題である。というようなこともあるので、よほど変わったことをしたいというのでない限り、専門家が作った公式を使うのが無難である。

Dormand と Prince は、段数／次数のさまざまな組合せについて、離散化誤差を非常に小さくした公式を求めている。これらは、

`http://www.unige.ch/math/folks/hairer/software.html`

から入手できる (Fotran と C がある)。

普通の（というのがどういう意味かは今のところ明らかではないが）常微分方程式を手軽に解くにはこれらの RK 公式を使えば十分である。

実用的には、こちらが欲しい計算精度をどうやって達成するかという問題がある。この問題を解決するためには、誤差を推定しながら積分の刻み幅を自動的に調節する必要がある。これについては次回にちょっと触れる。

7.2　ルンゲ・クッタ以外の方法

さて、「普通は」RK で十分というのは、もちろん時と場合によっては違う方法を使うべきであるという意味である。なぜそういう場合があるかというのは、以下の理由による。

- RK 法は（次数のいかんにかかわらず）ある問題を解くのにもっとも計算量が少ない方法ではないことが多い。
- 特に、問題の種類によっては、特別な RK 法を使うと非常にうまくいくことがある。
- 問題の種類によっては、普通の RK 法ではどんなに計算時間をかけてもまともな答がでない。

まあ、このへんは詳しい話をやりだすときりがないので、簡単に。以下、上の 3 つを順番に扱う。

7.3　線形多段階法

RK 法は、「一段階」法である。これはどういう意味かというと、微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{7.7}$$

の $x = x_i$ での近似解 y_i があったとして、次の x の値 ¡$x_{i+1} = x_i + \Delta x_i$¿ での近似解 y_{i+1} を求めるのに、 x_i, y_i と微分方程式そのものだけで十分であるということである。中間のややこしい値を計算する必要はあるが、それは RK 法のほうが勝手にやることで、使う方が入力を与える必要はない。

これに対して、一段階ではない方法、つまり多段階法というのは、 y_{i+1} を計算するのに、 y_i 以外の情報、例えば y_{i-1} や、そこで計算した導関数の値 f_{i-1} 、さらにもっと昔の情報を使うやりかたのことである。これは、プログラムとしてはもちろん RK 法に比べれば面倒になる。昔の値をとっておかないといけないし、また、一番最初に計算を始める時にどうするかという問題もあるからである。

さらに、「一段階でない」というだけなので、可能な計算法にあまりに多様な可能性がある。一例として、以下のような方法が考えられる。

$$y_{i+1} = y_{i-1} + 2\Delta x f_i \tag{7.8}$$

これは、陽的中点公式といわれるもので、式としては偏微分方程式の解法の時に空間微分の 1 次の項に使った中心差分と同じ形になっている。従って、理屈としては 2 次の精度をもった公式になっている。

この方法は、しかし、立派な名前までついているのにもかかわらず、実は使えない公式である。

どのような問題があるのかを示すために、以下の線形方程式

$$y = -ky \tag{7.9}$$

に陽的中点公式を適用したらなにが起きるかを考えてみる。刻みは固定の Δx とすると、離散化したものは

$$y_{i+1} = y_{i-1} - 2ky_i\Delta x \tag{7.10}$$

になる。これは線形差分方程式なので、前にもやったように固有値を調べればいい。今 $\alpha = k\Delta x$ として整理すると、固有方程式は

$$\lambda^2 + 2\alpha\lambda - 1 = 0 \tag{7.11}$$

となって、これは実解を 2 つ持つ。それらを λ_1, λ_2 とすれば、 $\lambda_1\lambda_2 = 1$ なので $\alpha \neq 0$ ならどちらかは必ず絶対値が 1 より大きい。

ちょっと式を見ればわかるように、絶対値が 1 より大きい固有値は $\alpha > 0$、つまり $k > 0$ なら負である。したがって、必ず振動的に発散することになる。

なお、上のような、安定な線形微分方程式について振舞いを調べるというのが、常微分方程式の安定性解析の基本になる。非線形な方程式では違うとかいろんなことがあるかもしれないわけだが、まあ、少なくとも線形安定でないと話にならないし、それ以上のことは一般論としていうのは難しいからである。

7.3.1 アダムス法

さて、安定でちゃんと使える線形多段階法はじゃあどんなものかというわけだが、これも無限にいろんな作り方がある。そのへんの詳しい話はそういう本に譲ることにして、ここではもっとも広く使われているアダムス法について説明する。

原理は、いくつかのステップでの導関数（微分方程式の右辺） f の値を憶えておいて、それを通る補間多項式を作り、それを積分して解を求めようというものである。

図に概念を示す。ここでは、ラグランジュ補間（ニュートン補間）をして多項式を作る。で、その作った多項式を積分する。例えば、点 i から $i+1$ まで積分するのに、点 $i-p$ から i までの関数値を使うとすれば、 p 次の多項式で

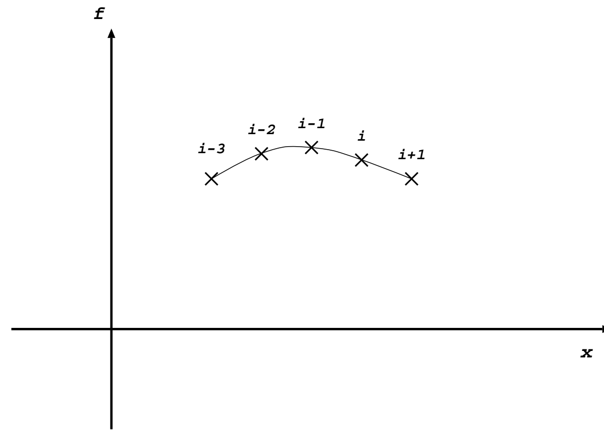


Figure 7.2:

$$P(x_j) = f_j = f(x_j, y_j) \quad (i - p \le j \le i) \tag{7.12}$$

を満たすものを作る。で、 $i+1$ での解 y_{i+1} は

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} P(x)dx \tag{7.13}$$

で与えられる。刻み h が定数であるとすれば、 p を決めれば上の式を

$$y_{i+1} = y_i + h \sum_{l=0}^{p} a_{pl} f_{i-l} \tag{7.14}$$

の形に書き直せる。

簡単な例として、 $p=1$ の場合を考えてみよう。この時、補間多項式は一次であって

$$P(x) = f_i - \frac{f_{i-1} - f_i}{h}(x - x_i) \tag{7.15}$$

となって、これを積分すれば、結局

$$y_{i+1} = y_i + \frac{h}{2}(3f_i - f_{i-1}) \tag{7.16}$$

となる。

一般に、アダムス法では任意段数の公式が構成でき、その次数は段数に等しいことがわかっている。これは、ルンゲ・クッタなどに比べればはるかによい性質をもっているということでもある。

7.3.2 出発公式

アダムス法はいくらでも高次の公式が作れ、計算量もあまり多くないということがわかっているが、必ずしも広く使われているというわけでもない。その理由はいろいろあるが、一つは、

「どうやって計算を始めるべきかよくわからない」

ということである。つまり、初期値問題としてはもちろん x_0 における y_0 しか知らないのに、多段階法ではその前の時刻での解が必要になるわけである。これに対する対応策はいくつかあるが、時間刻み一定の場合には、基本的にはルンゲ・クッタなどの別な方法で解を求めておくというやりかたが普通である。

というわけで結局プログラムを書く手間が 2 倍以上になるというのが、多段階法の実用上の問題である。

7.3.3 陰的アダムス法

さて、前に述べた公式では、補間多項式を陽的に求めた。すなわち、時刻 i とそれより以前の値だけを使っていた。これに対し、陰的な補間多項式、つまり t_{i+1} での関数の値を使った公式というものも考えられる。刻み h が定数であるとすれば、 p を決めれば前と同様に

$$x_{i+1} = x_i + h \sum_{l=-1}^{p-1} b_{pl} f_{i-l} \tag{7.17}$$

の形に書けることになる。また手をぬいて $p = 1$ の場合を考えれば、これは単に台形公式

$$x_{i+1} = x_i + \frac{h}{2}(f_i + f_{i+1}) \tag{7.18}$$

となる。

陰的公式の場合、例によってどうやって代数方程式を解くかが問題になる。通常の方法は、

- 初期値として同じ次数の陽的アダムス法の解を持ちいる。
- そのあとは直接代入法で反復する。

というものである。ただし、特に線形多段階法の場合、反復を繰り返さないことが多い。反復回数を 1 回とか 2 回に固定してしまうのである。なぜそれでいいか、また、そもそもなぜ陰的公式を使うかというあたりはレポート課題ということで。

なお、このやり方を、予測子・修正子法と呼ぶ。線形多段階法はほとんどこの形で使われるため、線形多段階法のことをさして予測子・修正子法と呼ぶ人もいる。

7.4 構造体とクラス

ここからはちょっと C++ 言語の話題というか、今回以降の話で便利な機能について。

偏微分方程式とか、あるいは連立常微分方程式の数値解法をプログラムするのに、普通は配列を使う。しかし、これは割合に面倒だし、いろいろ妙な間違いをする可能性も高い。例えば、 $\boldsymbol{x}$ に $\Delta\boldsymbol{x}$ を足すのは、数式としては $\boldsymbol{x} + \Delta\boldsymbol{x}$ で済むのに、例えば C では

```
for(i=0;i<n;i++) x[i] += dx[i];
```

Fortran なら

```
      do i = 1, n
          x(i) = x(i) + dx(i)
      enddo
```

という具合で、数式なら 4 文字で済むところをその何倍も書かないといけない。これは、 C にしても Fortran にしても、配列(あるいはベクトルとか行列)といった、数学では基本的な要素であるものに対する演算を直接に表現する記法を持っていないからである。

まあ、Fortran 95 とかそういったものを使うとそういう記法があるという話もないわけではないが、あまり普及していない。普及していない理由はいろいろあるが、その一つは C++ を使えば同様なことが実現できるからである。

C++ 言語自体は、配列に対する演算というものを用意しているわけではない。しかし、C++ では、プログラムの中で新しい「型」(実数型とか整数型というのと同じ意味での)を定義して、さらにそれに対する演算を定義することができる。これでベクトル型とかいったものを自分の使いやすいように定義すればいいことになる。

例えば、非常に基本的なベクトル型の定義は以下のようなものになる。ここでは加算と入出力くらいしか定義していないが、他の必要な演算も同様に定義できる。一応使いそうな演算を定義したものが

```
http://grape.astron.s.u-tokyo.ac.jp/~makino/pcphysics/programs/vector.h
```

にあるので、実際にプログラムを作る時にはこれを使ってもよい。牧野が書いたプログラムは信用できないという向きは自分で書くこと。

```
#ifndef  STARLAB_VECTOR_H
#  define  STARLAB_VECTOR_H
#define PR(x)  cerr << #x << " = " << x << " "
#define PRC(x) cerr << #x << " = " << x << ",  "
#define PRL(x) cerr << #x << " = " << x << "\n"

#ifndef real
# define real double
#endif
/*-----------------------------------------------------------------------------
 *  mvector  --  a class for 3-dimensional vectors
 *-----------------------------------------------------------------------------
 */
class mvector
    {
    private:
        real element[VLEN];

    public:
//Default: initialize to zero.
        mvector(real c = 0)
    {for (int i = 0; i<VLEN;i++)element[i] = c;}

//  []: the return type is declared as a reference (&), so that it can be used
//  on the left-hand side of an asignment, as well as on the right-hand side,
//  i.e.  v[1] = 3.14  and  x = v[2]  are both allowed and work as expected.

        real & operator [] (int i)       {return element[i];}

inline void print() {for(int i = 0; i<VLEN;i++)
     cout << element[i] << " ";
```

```
     cout << "\n";}

//Unary -

        const mvector operator - ()
    {mvector v;for(int i = 0; i<VLEN;i++)v.element[i] = - element[i];
          return v;}

        friend const mvector operator + (const mvector &, const mvector & );
        friend const mvector operator - (const mvector &, const mvector & );
        friend mvector operator * (real, const mvector & );
        friend real operator * (const mvector &, const mvector & );
        friend mvector operator * (const mvector &, real);
        friend mvector operator / (const mvector &, real);

//Mvector +=, -=, *=, /=

        mvector& operator += (const mvector& b)
    {for(int i = 0; i<VLEN;i++)element[i] += b.element[i];
     return *this;}

mvector& operator -= (const mvector& b)
    {for(int i = 0; i<VLEN;i++)element[i] -= b.element[i];
     return *this;}

mvector& operator *= (const real b)
    {for(int i = 0; i<VLEN;i++)element[i] *= b;
     return *this;}

mvector& operator /= (const real b)
    {for(int i = 0; i<VLEN;i++)element[i] /= b;
     return *this;}

//      Input / Output

        friend ostream & operator << (ostream & , const mvector & );

friend istream & operator >> (istream & , mvector & );
    };

inline  ostream & operator << (ostream & s, const mvector & v)
{
    for(int i = 0; i<VLEN;i++)s << v.element[i] << "  " ;
    return s;
}
inline  istream & operator >> (istream & s, mvector & v)
{for(int i = 0; i<VLEN;i++)s >> v.element[i]; return s;}

inline  const mvector operator + (const mvector &v1, const mvector & v2)
{
    mvector v3;
    for(int i = 0; i<VLEN;i++)v3.element[i] = v1.element[i]+ v2.element[i];
```

```
    return v3;
}
inline  const mvector operator - (const mvector &v1, const mvector & v2)
{
    mvector v3;
    for(int i = 0; i<VLEN;i++)v3.element[i] = v1.element[i]- v2.element[i];
    return v3;
}

inline  mvector operator * (real b, const mvector & v)
{
    mvector v3;
    for(int i = 0; i<VLEN;i++)v3.element[i] = b* v.element[i];
    return v3;
}
inline  mvector operator * (const mvector & v, real b)
{
    mvector v3;
    for(int i = 0; i<VLEN;i++)v3.element[i] = b* v.element[i];
    return v3;
}

inline  real operator * (const mvector & v1, const mvector & v2)
{
    real x = 0;
    for(int i = 0; i<VLEN;i++) x+= v1.element[i]* v2.element[i];
    return x;
}


inline  mvector operator / (const mvector & v, real b)
{
    mvector v3;
    for(int i = 0; i<VLEN;i++)v3.element[i] =  v.element[i]/b;
    return v3;
}

typedef const mvector (*vfunc_ptr)(const mvector &);
typedef const mvector (vfunc)(const mvector &);

#endif

//=========================================================================//
//  +---------------+        _\|/_        +------------------------------\\
//  |  the end of:  |         /|\         |  inc/vector.h
//  +---------------+                     +------------------------------//
//======================== STARLAB =====================================\\
```

実際にこれを使うには、

```
const int VLEN = 2;
```

```
#include  "vector.h"

double k;
mvector dxdt(mvector & x)
{
    mvector d;
    d[0] = x[1];
    d[1] = -k*x[0];
    return d;
}

   .....
   double h;
   mvector kx1;
   kx1 = dxdt(x)*h;
   ......
```

というような具合に、ベクトルの大きさを指定する（これはここでは固定である）。もちろん、可変にするとかいろんなことができるが、繁雑になるのでここでは固定サイズにする。

ここで、注意してほしいのは、 + という演算子や [] という、これも「演算子」が、新しく定義されていることである。これらは、もちろん

```
+ なら実数や整数に対してすでに定義されているが、ここではベクトル
```

同士の演算に対して新しい意味を持つように拡張されていることになる。[] も同様で、配列の他にここで定義したベクトル型について新しい意味を持つようになったわけである（といっても、こちらは普通の配列に対してとまったく同じように働くが）。

これは、偉そうにいうと C++ の演算子多重定義 (overload) という強力な機能である。まあ、落ち着いて考えてみると、この機能の実現はそんなに難しいわけではなくて、例えば + という演算子が出てきたところで、それが適用されているデータ型を見て、そのデータ型に対して定義されている演算を呼ぶようにするというだけのことである。これは、コンパイラにそういう機能を付け加えるだけで実現できる。

C++ の場合には、このような、ある意味での機能拡張は、「クラス」というものを使って実現されている。これはもともとは「オブジェクト指向」とかそういった難しい機能を実現するためのものであるが、ここではとりあえずあんまりそういうことは考えないでベクトル型を使うことにする。

なお、名前が素直に vector ではなく mvector (mathematical vector のつもり) になっているのは、C++ の新しい標準では vector という 語に別の意味を与えているからである。

上のコードの意味を簡単に説明しておく。

```
class mvector は mvector という名前のクラスの定義を始めますという
```

宣言である。その後に \{\} で囲って中身を書く。

中身には private: と書いてから書くものと public: と書いてから書くものがあって、 private: のほうは、「普通には外から見えない」ものを定義するのに使う。典型的なのはこの例のように、あるクラスの「オブジェクト」が持つデータを宣言することである。データの宣言は普通の変数宣言と同じ。

```
public:  のほうでは、主に「メンバー関数」というものを宣言するこ
```

とになる。ここで、自分の名前と同じ名前の関数は特別な意味を持っていて、(constructor)、このクラスの変数がプログラムの中で使われる時に呼ばれるものである。 上の例では、デフォルトでは vector a; みたいに書くと

`element` の各要素に `0` が代入される。`0` 以外にしたければ `vector`

a = vector(1); とか書く。

その次の double \& operator [] (int i)\{ ...\} というのははっきりいってなんだかわからないが、C++ ではいくつかの演算記号や上の配列の記号等に、クラス毎に別の意味を与えることができる。これが「演算子の多重定義」や「オーバーロード」と呼ばれる機能である。

というとなんだか恐ろしげに聞こえるが、考えてみると "+" だって実数と整数では全く違うことをするわけで、別のデータ型を定義したらそれの "+" はまた違うことをして欲しいというのは当然であろう、というより、ここで我々がしたいのはそもそもそういうことであった。 上の例では、 結局プログラムの中で

`mvector a; ... a[i] ...` という感じに書くと、 `a[i]` が `a` の要素である `element` の `i` 番目の要素、つまり

element[i] と全く同じことになる。

なお、 C 言語では構造体 struct というものがあり、例えば

```
struct mvector{
    double element[NDIM];
};
```

といったふうに使う。ここで struct mvector a; のように変数を宣言すると、 a.element[i] といった形で "." を使って要素にアクセスすることになる。これはちょっと見苦しい。また、C には演算子のオーバーロードの機能はないので、折角構造体を作っても、 "+" で足すというわけにはいかない。これはある意味みかけだけの問題ではあるが、見てわかりやすいということはプログラムを書いたりデバッグしたりする時には割合大事なことである。

その後の friend const mvector operator + ... が、実際に mvector 同士の加算を宣言する。但し、ここでは形だけを宣言して、実体は下のほうに書く。 但し、その次の += のように全部書いてもかまわない。

`mvector a, c; ... a += c;` と書くと、`‘‘ +=’’` の中で いきなり

element と書かれているものは a の要素、 b.element になっているのは c の要素になって、結局 a の各要素に c の各要素が足されるという、普通に期待したい動作が実現されることになる。なお、このような、宣言してないクラス変数を使う関数を「メンバー関数」という。

" +" も事情は同様だが、ここでは 2 つ引数を取る普通の関数として定義されている。 friend を付けると、 private: と宣言したものもこの関数の中では使えるようになるので、ここでは friend を付けている。

" ¡¡", " >¿" はメンバ関数として実現されている。

7.5 練習

7.5.1 練習 1

一次元調和振動子

$$\frac{d^2x}{dt^2} = -kx \tag{7.19}$$

について、初期条件

$$x(0) = 1, \quad \frac{dx}{dt}|_{t=0} = 0 \tag{7.20}$$

からの、 $t = 2\pi$ までの数値解を、古典的ルンゲ・クッタ公式を使って求めよ。刻み幅をいろいろ変えて、精度が刻みにどのように依存するか、また、どこまで高い精度が実現可能か調べ、なぜそうなるかを考察せよ。

なお、牧野が書いたプログラムが ここ[1] の hermonic5a.C にあるので、今回はこれをコピーしてコンパイル・実行するだけでも一応いいことにする。但し、これは古い C++ の規格にそって書かれているので色々変更が必要である。

7.5.2 練習 2

刻み幅が等しい場合について、以下の公式を導け

- 4 次の陽的アダムス公式
- 4 次の陰的アダムス公式
- 4 次の陽的シュテルマー公式

但し、シュテルマー公式とは、2 階の微分方程式用の公式である。時間の 2 階微分に、偏微分方程式の時に使ったのと同じ 3 点を使う中心差分を使う。

7.5.3 練習 3

上で導いた 4 次の陽的・陰的アダムス公式を使って、1 と同じ単振動の方程式について同様な解析を行なえ。出発公式にはルンゲ・クッタを使うか、あるいは厳密解をつかってもよい。

7.5.4 練習 4

陰的線形多段階法について、直接代入法が収束する速さが何で決まるかを考察せよ。

[1] http://jun-makino.sakura.ne.jp/pcphysics/programs/index.html

Chapter 8

常微分方程式の初期値問題(2)

8.1 今日の予定

今日はまず与えられた精度を達成するためにどのように刻みを調整するかという話をしたあと、ハミルトン系に特化した数値解法の話に移る。

8.2 刻み幅調節と埋め込み型公式

ルンゲクッタにしても線形多段階法を使うにしても、実際に問題を解こうという上ではいろいろ考えないといけないことがある。もっとも重要なことは、刻み幅 h をどうやって決めるかということである。

実際問題としては、解いていくにつれて導関数の値が大きく変わるといったことは普通に起きる。このために、刻み幅が一定のままで計算していくというのは非常に無駄なことが多い。

無駄を減らすためには、局所誤差が小さい（解が滑らかな）ところは刻みを大きく、逆に解が急にかわるところでは刻みを小さくしてやればよい。このような方法を可変刻み (variable step) 、あるいは適応刻み (adaptive step) という。

8.2.1 RK 法の場合

例えば Runge-Kutta 法の場合、 y_n から y_{n+1} にいくのに前に計算した情報とかなにかを使うわけではないので、刻み幅は任意にとることができる。問題は、どうやって刻み幅を決めるかである。

これにはいろいろな考え方があるが、通常使われるのは、局所離散化誤差を推定して、それがある値以下になるようにするという方法である。

このための一つの考え方は、以下のようなものである。

1. まず、ある幅 h で積分する。
2. 次に、その同じ幅を、 $h/2$ ずつ 2 回に分けて積分する。
3. すると、2 回に分けて積分したほうがより正確な答になっていると期待できるので、二つの解の差が誤差の推定値（大きめではあるが）となっている。

誤差の推定値が求まったら、普通は次のステップを調節する。局所離散化誤差の次数がわかっているから、それに応じてステップサイズを調整すればよい。次数プログラムによっては、現在のステッ

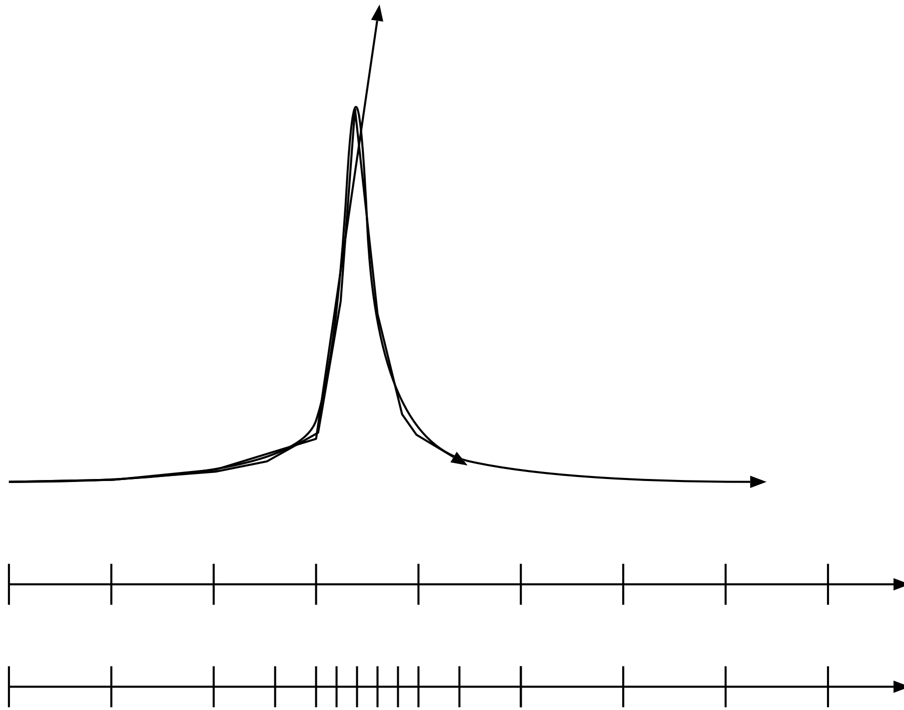


Figure 8.1: 適応的刻み幅の概念。解が急に変化する時に刻み幅が大きいままでは上手く数値解が求まらないが、変化に合わせて刻み幅を小さくすればそれらしい解が求まる。

プでの誤差が予定した値よりも大きければステップを小さくして計算し直すようになっているものもある。

実装（プログラムを作ること）が容易であることもあって、この方法はわりと広く使われている。が、かなり無駄が多い方法であるというのも確かである。というのは、単に誤差の推定のためだけに、 50\% の無駄な計算をしているからである。まあ、 50\% はたいしたことないという考え方もあるが、計算に何日も掛かるというような状況なら 50\% は無視できないので、まあ、なんとかしたくなるのが人情というものである。

というわけで、もうちょっとうまい方法はないかということで、以下のようなことを考えた人がいる。

一般に、 RK 型の公式では、最終的な値は

$$y_{n+1} = y_n + \sum_{i=1}^{s} b_i k_i \tag{8.1}$$

の形になっている。ここで、 k_i は全部そのまま使って、 b_i を別の b'_i に置き換えた

$$y' = y_n + \sum_{i=1}^{s} b'_i k_i \tag{8.2}$$

で、局所離散化誤差が元の公式よりも大きいが、むやみと大きくはない（例えば、一次次数が低い）ものがあれば、元の公式との差を誤差の推定に使うことができる。

この形の公式のことを埋め込み型 embedded 公式という。最初に提案した人の名前をとって Runge-Kutta-Fehlberg 公式ということも多い。これもいろいろ提案されているが、前回にも紹介した Dormand-

Prince 公式はすべて埋め込み型であり、これが最近はもっとも広く使われている。精度が高いものが必要な時は、 8 次の公式が使われる。これらは

`http://www.unige.ch/math/folks/hairer/software.html`

から入手できる (Fotran と C がある)。

8.2.2 線形多段階法の場合

線形多段階法の場合、 RK と違って誤差の推定は容易である。例えば、アダムス法で予測子・修正子ペアで使うなら、予測子と修正子の差は誤差の（オーダーとしては）よい推定値になっている。したがって、この部分については面倒なことは特にない。

問題は、実際に刻みを変えるほうである。もっとも一般的な方法は、「その場で公式を導出する」というものである。これについて以下簡単に説明する。

8.2.3 ニュートン補間とアダムス公式の一般的導出

アダムス法を使うには、要するに多項式補間の系数が出せればよい。以下、ニュートン補間を使う方法を説明する。

補間したい関数 $f(x)$ が、標本点 $(x_0, x_1, ...x_n)$ で、関数値 $(f_0, f_1, ...f_n)$ をとるとする。これを、以下のような形の多項式

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1)... + a_n(x - x_0)\cdots(x - x_{n-1}) \tag{8.3}$$

という形に書くことを考える。これを、 a_p までをとったものは ¡$x_0 \cdots x_{p}$¿ までを通る最低次補間多項式になっているように構成することにする。で、低い次数から順に作っていく。付け加える新しい項は、それまでに使った点すべてで 0 になるので、新しい点で標本と一致するように a_i の値を決めればいい。

まず、 $n = 0$ については、 $a_0 = f_0$ とすればいいのは明らかであろう。次に $n = 1$ であるが、

$$P(x_1) = f_0 + a_1(x_1 - x_0) = f_1 \tag{8.4}$$

から

$$a_1 = \frac{f_1 - f_0}{x_1 - x_0} \tag{8.5}$$

となる。同様に、 a_2 を求めると、

$$a_2 = \frac{f[0,2] - f[0,1]}{x_2 - x_1} \tag{8.6}$$

ただし、

$$f[i,j] = \frac{f_i - f_j}{x_i - x_j} \tag{8.7}$$

である。

さて、段々式が繁雑になるが、もうちょっとの辛抱である。上の形は、

$$a_2 = \frac{f[2,1] - f[1,0]}{x_2 - x_0} \tag{8.8}$$

というふうにも書き直せることに注意しよう。これは、 $(1,2)$ を通る補間式の一次の係数と、 $(0,1)$ を通る補間式の係数の差みたいなものになっている。

ここで、天下りに、 k 階差商 (divided difference) ¡$D_{k,l} \quad (0\le l \le n-k)$¿ というものを導入する。これは以下のように定義される。

$$\begin{aligned} D_{0,l} &= f_l \quad (0 \le l \le n) & (8.9)\\ D_{k,l} &= \frac{D_{k-1,l} - D_{k-1,l+1}}{x_l - x_{l+k}} & (8.10) \end{aligned}$$

実は、このように定義された D が、

$$D_{k,0} = a_k \tag{8.11}$$

を満たす、求める係数であることを示すことが出来る。

証明には、 $D_{k,l}$ が、 x_l から始まる $k+1$ 点を通る補間多項式の係数であるということを用いる。定義により

$$D_{k,0} = \frac{D_{k-1,0} - D_{k-1,1}}{x_0 - x_k} \tag{8.12}$$

である。ここで、帰納法を使って証明することにすると、

$$P_{k-1}(x) = D_{0,0} + D_{1,0}(x - x_0) + \cdots + D_{k-1,0}(x - x_0)\cdots(x - x_{k-2}) \tag{8.13}$$

は、 $x_0, \cdots x_{k-1}$ を通る $k-1$ 次補間多項式であり、また、

$$P'_{k-1}(x) = D_{0,1} + D_{1,1}(x - x_1) + \cdots + D_{k-1,1}(x - x_1)\cdots(x - x_{k-1}) \tag{8.14}$$

は、$x_1, \cdots x_k$ を通る $k-1$ 次補間多項式であるとしてよい。この 2 つの線形結合によって、 $x_0, \cdots x_k$ を通る補間多項式を作るには、

$$P_k = \frac{(x - x_0)P'_{k-1} - (x - x_k)P_k}{x_k - x_0} \tag{8.15}$$

とすればよい。ここで、 $D_{k-1,0}$ と $D_{k-1,1}$ は、それぞれ P_{k-1} と P'_{k-1} の最高次の係数であることに注意すると、 P_k の最高次の係数 (これは a_k に等しい) は、 $D_{k,0}$ で与えられることがわかる。

さて、

$$Q_k(x) = D_{0,0} + D_{1,0}(x - x_0) + \cdots + D_{k,0}(x - x_0)\cdots(x - x_{k-1}) \tag{8.16}$$

という多項式を考えると、これは $x_0, \cdots x_{k-1}$ で P_k と一致し、さらに最高次の係数も一致しているので、結局 $Q_k = P_k$ であることがわかる。つまり、 $Q_k(x)$ が求める補間多項式であり、 $D_{k,0}$ がその最高次の係数ということになる。

と、こんな風に差商を作っておくと、これから多項式の係数を機械的に計算して積分して、、、、ということができる。さらに、差商の形でデータをとっておくと、上の漸化式を使って D を順番に更新することができる。これは Krogh 型の公式と呼ばれ、かなり広い範囲の問題について、{\bf もっとも効率が良い公式 } であることがわかっている。

なお、上の方法で時間刻みを変えられるなら、出発公式は不必要になることに注意しておく。つまり、最初は 1 次の予測子、2 次の修正子から出発して、ステップ毎に次数をあげていけばいい。ステップサイズは次数を考慮して決めておく。

8.3 ハミルトン系向けの方法

8.3.1 簡単な例題

古典力学系のもっとも簡単な例といえば、一次元調和振動子、つまり、運動方程式では

$$\frac{d^2x}{dt^2} = -kx \tag{8.17}$$

である。これの解はもちろん単振動するものであり、それは固有値が純虚数であるからである。

さて、このような、固有値が純虚数である、いいかえれば中立安定な場合、安定性解析はちょっと厄介になる。というのは、通常の意味で安定というのは、数値解がいつかは原点にいってしまうということ、いいかえればエネルギーが保存しないということを意味するからである。というのは、安定であるとは、線型差分方程式の固有値の絶対値が全て 1 より小さいということだからである。

つまり、安定領域のある公式では、ほとんどのもので虚軸上の一定範囲では数値解が「安定」になってしまう。つまり、本当の解はいつまでも振動を続けるのに、数値解は減衰してしまうのである。もちろん、タイムステップを大きくとれば、安定領域から外れるのが普通であり、振動しながら広がっていくような解になる。安定領域に虚軸を含まないような公式（例えば前進オイラー公式）の場合では、かならず軌道が広がっていくことになる。つまり、虚数軸が安定性限界になっていない限り、必ず軌道から一方的に外れていってしまうのである。

安定領域とはそもそも何かという話をしておく。線形常微分方程式

$$\frac{dy}{dx} = kx \tag{8.18}$$

を刻み h で積分した時に i、これは結局 $y_n = C\lambda^n$ という形の一般解を持つ。ここで、すべての固有値 λ の絶対値が 1 を超えないような、複素平面上での kh の領域のことを安定性領域という。

k が複素数の場合も考えるのは、元の微分方程式が連立方程式なら固有値が実数でない場合もありえるからである。

つまるところ、線形系では安定性限界が虚数軸である（数値解に純虚数の固有値がある）というのが、数値解が厳密に振動的であるための必要十分条件ということになる。この時、数値解はもとの系の性質をある意味で良く表しているといえるであろう。

実は多変数の場合や非線形の場合でも本質的な事情は変わらない。ただし、こちらはまだいろいろ理論的にはっきりわかっていないことがいろいろある。が、大雑把にいうと同じような事情が成り立っている。つまり、もとが周期解であるときに、大抵の積分法では周期解から一方的にずれていく、つまり、保存量であるはずのエネルギー等が保存しなくなる。

もちろん、エネルギーが厳密には保存しないということが問題であるかどうかというのも実は難しい問題である。というのは、保存しないといってももちろん数値解の精度では保存しているわけだから、それでかまわないのではないかとも考えられるからである。

8.3.2 リープフロッグ公式

ハミルトン系のシミュレーションの例には例えば以下のようなものがある

- プラズマシミュレーション
- （古典）分子動力学
- 銀河動力学

これらはいずれも、古典的な多体問題として定式化される。プラズマの場合は磁場も入るのでちょっと面倒だが、それを無視すれば運動方程式が

$$m_i \frac{d^2 \boldsymbol{x}_i}{dt^2} = \sum_{j \neq i} \boldsymbol{f}_{ij} \tag{8.19}$$

つまり、ある粒子 i の加速度は他のすべての粒子からの力の合計ということになる。もちろん、古典力学系は他にもいろいろあるが、基本的なアプローチは似たようなものである。

このような問題に対して、もっとも普通に使われる公式は以下の leapfrog といわれるものである。

$$v_{i+1/2} = v_{i-1/2} + \Delta t a(x_i) \tag{8.20}$$

$$x_{i+1} = x_i + \Delta t v_{i+1/2} \tag{8.21}$$

これでは速度と位置がずれた時間でしか定義されないが、出発用公式として

$$v_{1/2} = v + \Delta t a(x_0)/2 \tag{8.22}$$

を使い、さらに終了用公式として

$$v_i = v_{i-1/2} + \Delta t a(x_i)/2 \tag{8.23}$$

を使うことで最初と最後を合わせることが出来る。この形は、実は

$$x_{i+1} = x_i + \Delta t v_i + \Delta t^2 a(x_i)/2 \tag{8.24}$$

$$v_{i+1} = v_i + \Delta t[a(x_i) + a(x_{i+1})]/2 \tag{8.25}$$

と数学的に等価である（証明してみること）また、以下のようにも書ける

$$v_{i+1/2} = v_i + \Delta t a(x_i)/2 \tag{8.26}$$

$$x_{i+1} = x_i + \Delta t v_{i+1/2} \tag{8.27}$$

$$v_{i+1} = v_{i+1/2} + \Delta t a(x_{i+1})/2 \tag{8.28}$$

さらにまた、速度を消去して x_{i-1}, x_i, x_{i+1} の関係式の形で書いてあることもあるかもしれない。なお、すべて違う名前がついていたりするが、結果は丸め誤差を別にすれば完全に同じである。時々、これらが違うものであるかのように書いてある教科書があるので注意すること。

この公式は局所誤差が $O(\Delta t^3)$、大域誤差が $O(\Delta t^2)$ である。

さて、局所誤差という観点からはこれは決して良い公式というわけではないが、現実にはこの公式は非常に広く使われている。

これは、別にもっとよい方法を知らないからとかではなく、実はこの方法がいくつかの意味で非常に良い方法であるからである。いくつかの意味とは、例えばエネルギーや角運動量のような保存量が非常によく保存するということである。これらの保存量については多くの場合に誤差がある程度以上増えない。

この、ある程度以上誤差が増えないというのは目覚しい性質である。通常の方法では、エネルギーの誤差は時間に比例して増えていく。従って、長時間計算をしようとすればそれだけ正確な計算をする必要がある。ところが、エネルギーの誤差は溜っていかないのならば、かならずしも精度を上げる必要はないとも考えられる。

もちろん、エネルギーが保存していればそれだけで計算が正しいということにはならない。が、理論的には、いくつかの重要な結果が得られている。まず、

1.leapfrog は symplectic method のもっとも簡単なものの一つである。

1. leapfrog は symmetric method のもっとも簡単なものの一つである。

Symplectic method については、以下のようなことが知られている

1. symplectic method は、すくなくともある種のハミルトニアンに対して使った場合に、それに近い別のハミルトン系に対する厳密解を与えることがある。
2. 周期解を持つハミルトン系に対して使った場合に、どんな量でも誤差が最悪で時間に比例してしか増えない。
3. 時間刻を変えると上のようなことは成り立たなくなる

これに対し、 symmetric method については以下のようなことが知られている

1. 周期解を持つ時間対称な系に対して使った場合に、どんな量でも誤差が最悪で時間に比例してしか増えない。
2. 時間刻を変えてもうまくいくようにすることも出来る。

以下、まず数値例でこれらの性質を確認し、それからなぜこのようなうまい性質を持つのかについて直観的な説明を与えよう（厳密な説明／証明には力学系の積分可能性に関するかなり深い知識が必要となる）。

8.3.3 数値例

\subsubsection{ 調和振動子 }

能書きを聞いていても良くわからないので、実例を見てみよう。まず、簡単な例として調和振動子

$$\frac{d^2x}{dt^2} = -x \tag{8.29}$$

を leap frog と 2 階のルンゲクッタで解いた例を示す。初期条件は ¡$(x,v) = (1,0)$¿ で、時間刻みは 1/4 である。

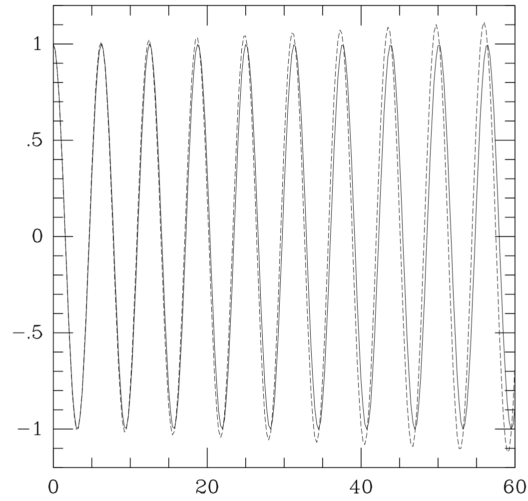


Figure 8.2: 調和振動子の数値積分。軌道。破線は 2 次のルンゲクッタ、実線は leapfrog の結果

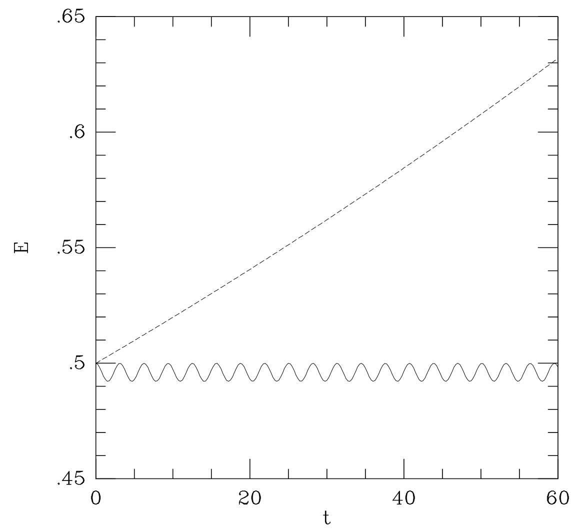


Figure 8.3: 調和振動子の数値積分。エネルギー。破線は 2 次のルンゲクッタ、実線は leapfrog の結果

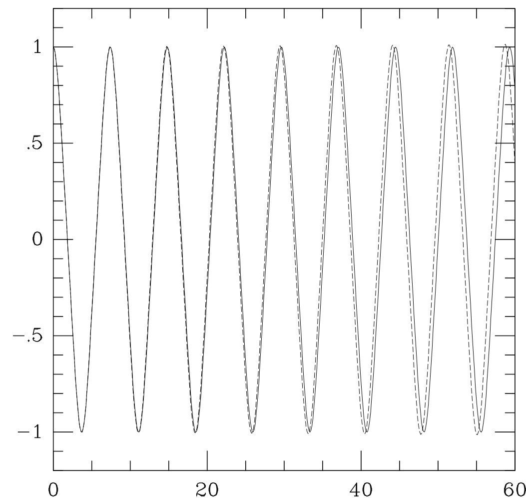


Figure 8.4: 非線形振動の数値積分。軌道。破線は 2 次のルンゲクッタ、実線は leapfrog の結果

軌道とエネルギーを図に示す。非常に特徴的なのは、 leap frog ではエネルギーが周期的にしか変化しないのに対し、ルンゲクッタでは単調に増えていることである。

ルンゲクッタでは単調に変化するというのは前に説明した通り（2 階のルンゲクッタでは虚軸を安定領域に含まないため）である。さて、これに対し、 leapfrog ではエネルギーが変化していないが、これはどういうことなのであろう？

実は、この調和振動子の場合には、 leapfrog 公式は以下の量

$$H' = \frac{1}{2}(x^2 + v^2) - \frac{h^2}{8}x^2 \tag{8.30}$$

を保存するということを確かめることが出来る。つまり、 (x, v) で与えられる位相平面の上で考えると、 leapfrog 公式の解は上の式で与えられる楕円の上にのっているのである。このために、エネルギーの誤差がある値よりも大きくなり得ないことになる。

8.3.4 非線形振動

では、非線形振動ではどうだろう？簡単な例として

$$\frac{d^2x}{dt^2} = -x^3 \tag{8.31}$$

を leap frog と 2 階のルンゲクッタで解いた例を示す。初期条件は ¡$(x,v) = (1,0)$¿ で、時間刻みは 1/8 である。

軌道とエネルギーを図に示す。調和振動の場合と同様に、 leap frog ではエネルギーが周期的にしか変化しないのに対し、ルンゲクッタでは単調に増えている。

この場合も保存量があるので、頑張れば求まるかもしれない。

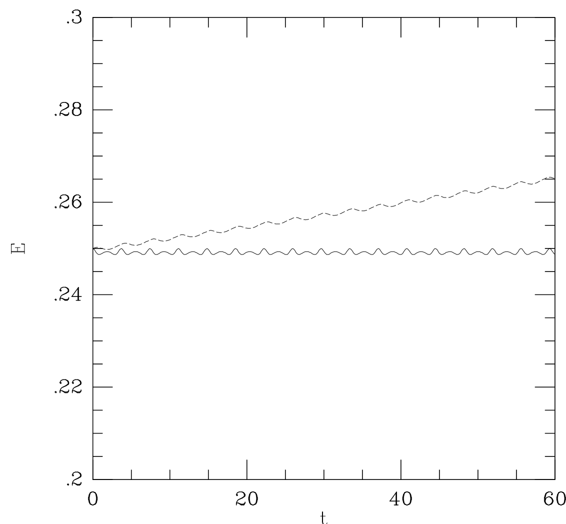


Figure 8.5: 非線形振動の数値積分。エネルギー。破線は 2 次のルンゲクッタ、実線は leapfrog の結果

8.3.5 シンプレクティック公式

さて、 leapfrog 公式は、上で見たようにハミルトン系に対してエネルギー誤差が有界に留まるという大きな特長がある。が、2 次精度でしかない。もっと高次の方法はないのだろうか、またあるとすればどういう原理で作れるのだろうか。

高次の方法を構成する一つのアプローチが、シンプレクティック公式といわれるものである。これはなにかというと、積分公式がシンプレクティック写像になるように作るということである。

シンプレクティック写像とは、ようするに正準変換のことである。正準変換とはなにかというのは解析力学を思い出してみて欲しいが、直観的には、力学系を不変に保つ、つまり変換まえの座標系で求めた軌道を変換したものと、変換後の座標系での力学系の軌道が厳密に同じになるようなものである。

ハミルトン力学系の解そのもの（ある時刻 t での座標から、 $t+h$ での座標に移す変換）もシンプレクティックである。まあ、だから、シンプレクティックになっているような積分公式は、そうでないものに比べてなんとなく力学系の性質にあっているような気はする。

で、いいたかったことは何かっていうと、上の leapfrog 公式はこのシンプレクティック性を満たしているということだった。詳しくは、「数理科学」1995 年 6 月号にのった吉田春夫さんによる解説記事でもみてもらうことにして、ここでは高次の公式にはどんなものがあるかという話をしておく。

8.3.6 陽解法

陽解法の組み立て方はいろいろある。一つは、 RK 系の公式で、係数をシンプレクティック性を満たすように決めるということである。これはここ 15 年で無数に論文がでた。4 次、あるいは 6 次の公式としては、吉田や鈴木による作用素分解に基づく公式が良く知られている。

これらの方法の原理は、要するに上の leap frog をタイムステップを変えていくつか組合わせるというものである。うまくタイムステップを組み合わせると誤差の高次の項を消すことができるわけである。3 段 4 次の公式、7 段 6 次の公式等が吉田によって導かれている。

なお、実は陽解法はハミルトニアンが $T(p)+V(q)$ の形の場合にしか使えないが、大抵の問題はこう書けることはいうまでもないであろう。

また、 RK 系の公式を力任せに構成する試みもあり、4 次から 8 次までの公式が作られている。計算精度という観点からはこちらのほうが leapfrog を組合わせるものよりもいいものもある。

8.3.7 シンプレクティック公式の意味

さて、シンプレクティックであるということと、「良い」ということの間にはちゃんとした理論的な関係があるのであろうか？一応あるということになっている。つまり、あるハミルトニアン H で表される系をシンプレクティックな p 次の公式を使って積分した解は、別のハミルトニアン H' で与えられるシステムの厳密解になっていて、 H と H' の間に

$$H = H' + h^{p+1} H_p + O(h^{p+2}) \tag{8.32}$$

という関係がある（ H' を求める数列が収束すれば）ということがわかっている。

なお、例えば上の調和振動子の場合には実際に数列が収束し、 H' が求まっている。が、これは極めて例外的な場合で、一般の場合には収束するかどうかも明らかではないようである。

収束するかどうか明らかではないのでは、使っていいことがあるという保証はないではないかと思うかもしれない。理論的にはその通りなのであるが、実際にはいろいろな問題に適用されて、従来の方法よりも高い精度が得られるということが確認されている。

8.3.8 シンプレクティック公式の問題点と対応

さて、式 (\ref{eq:shadow}) をみるとわかるように、シンプレクティック公式に付随するハミルトニアン H' には時間刻み h が入っている。従って、h をふらふら変えると H' も変わって、結果的に求まった数値解は一つの力学系の軌道ではないなんか変なものになってしまう。ということは、可変時間刻みにするとシンプレクティック公式はうまく働かないのではないかということが想像される。

じつはその通りで、例えばシンプレクティックで埋め込み型のルンゲクッタというものを作って、実際に時間刻みを変えてみた人がいる。その結果、普通のルンゲクッタよりも良くならないということが発見された (1992 年頃)。問題によってはこれは致命的な欠陥となるので、さまざまな対応法が精力的に研究されているのが現状である。が、あまり一般的にうまくいく方法というのは見つかっていないようである。つまり、万能な公式というのはシンプレクティック公式に関する限り知られていない。

もう一つのシンプレクティック公式の問題点は、「写像」であるということから一段法、具体的にはルンゲクッタ型の公式であるので、局所誤差に対する計算量という観点からは線形多段階法に比べて必ず悪いということである。

これらを解決しようという研究はいろいろあるが、一つの方向が以下に述べる対称型の公式である。

8.3.9 対称型公式とは

さて、まず対称型公式とはなにかということだが、まず時間反転に対する対称性というものを定義しておこう。時間反転に対する対称性とは、常微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{8.33}$$

に対する一段法

$$y_{i+1} = F(x_i, y_i, f, \Delta x) \tag{8.34}$$

が、

$$y_i = F(x_{i+1}, y_{i+1}, -f, \Delta x) \tag{8.35}$$

を満たすこと、つまり、直観的には、ある微分方程式系があって、それを 1 ステップ分数値解を求めたとする。で、そこから逆に戻ると厳密に元の値に戻るということである。（ここでは丸め誤差はないものとする）

で、一段法の場合には、この意味で対称なものを対称型公式ということにする。

具体例で考えてみよう。例えば前進オイラー法は対称型ではない。これは、いった先で導関数を計算すれば、一般にもとのところでの導関数とは違うから当然であろう。前進オイラー法が対称ではないのだから、その逆写像である後退オイラー法も対称型ではないことになる。

では、対称型にはどういうものがあるかということだが、明らかに対称型であるものとしては、台形公式

$$y_{i+1} = y_i + \frac{h}{2}(f_i + f_{i+1}) \tag{8.36}$$

がある。これが上の対称性を満たしていることはいうまでもない。

さて、なぜこの型の公式を考えるかということであるが、実験的には以下のようなことが知られている。

- ハミルトン系に対して対称型公式を使った場合、エネルギーや角運動量などの保存量の誤差がある範囲内にとどまる
- 時間刻みを変えても上の性質を保つことが出来る

前のほうの性質はシンプレクティック公式と同じであるが、後の方の性質はシンプレクティック公式よりもある意味でよいものである。

ここではとりあえず対称型公式にはどんなものがあるかという話をしておこう。

さて、一階の方程式に対する一段法という制限をつけた場合、つまり、ルンゲクッタ法の場合、対称な公式はかならず陰的公式になる。逆に、陰的公式であれば対称型のものを作るのは容易である。つまり、ルンゲクッタ法を与える行列 A とかベクトル b, c が対称になっていればいい。

一階の系に対する一段法では、陰的 RK のほかにうまい方法があるわけではない。それでは、

- ハミルトン系専用の解法ではどうだろうか
- 多段法ではどうなっているのだろう
- それ以外の方法はないのだろうか

以下、順番に考えていくことにする。

8.3.10 ハミルトン系用の陽的対称型 RK 公式

実は、すでに述べたように、 leapfrog 公式は対称型でありしかも陽公式である。で、 leapfrog の組合わせで作られる公式も、実はすべて対称型になっている。

8.3.11 対称型線形多段法

線形多段階法でも、対称型というものを考えることはできる。但し、これらの公式は、今のところ

- 一階の微分方程式用のものは次数が高いと不安定である。
- 二階の微分方程式用のものは、線形解析では安定であっても非線型振動では数値不安定を起こす。

ということが知られており、実用になっていない。これは国立天文台の福島さんのグループにより精力的に研究が進められている。

8.3.12 エルミート型公式

一般に対称型というわけではないが、4 次の対称型公式を導く特別な方法としてエルミート型（あるいはエルミート・オブレヒホフ型）と呼ばれる公式のクラスについてここで触れておこう。

エルミート型公式は、線形多段階法のアダムス型公式の一つの一般化である。どのような意味において一般化であるかというと、アダムス型と同じように補間多項式を積分することで新しい時刻での値を求めるところは同じである。違うところは、アダムス型公式では補間多項式を作るのに関数の値を使う（ラグランジュ・ニュートン補間）が、エルミート型公式では、関数の導関数の値も使うのである（エルミート補間)。

もっとも単純な発想としては、テイラー法というものがある。つまり、微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{8.37}$$

というものがあったとしたら、

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{dy}{dx} \tag{8.38}$$

という関係式を使って f の全微分を求めることが出来、同様に高次の導関数もどんどん計算していくことが出来る（微分が式で書ければ)。これらを使って直接テイラー級数を評価して近似解を求めるのである。

高次の導関数が簡単に求められる場合（例えば線形な系とか）には、この方法はかなり強力である。線形多段階法やルンゲクッタに比べて誤差項の係数がずっと小さく、タイムステップが大きくとれるからである。

が、多くの問題において、高階導関数の直接計算は現実的ではない。これは、計算量が指数関数的に爆発するのが普通だからである。とはいえ、 f の一階導関数や二階導関数くらいならば指数関数的といってもたかがしれている。しかしながら、これではテイラー法では 2 次や 3 次の公式しか作れない。実用的な公式としては、（leap frog のような特別に良い性質をもっているとかいうのでなければ）もうちょっと高次のものが必要である。

そこで考えられるのが、高階導関数を使ってさらにルンゲ・クッタ型とか線形多段階法のような公式を作ることである。例によってもっとも簡単な場合というのを考えると、それは y_i と y_{i+1} のところで f の一階導関数の値を使うもの（高次導関数を使わないものであれば台形公式に相当）の一般化ということになろう。

台形公式は、2 次の陰的アダムス型公式と考えることができる。つまり、 f を線形補間してそれを積分しているわけである。これに対し、 f と ¡$f' = df/dx$¿ を使う補間というのはどういうものか

ということを考えてみると、これは 3 次多項式を構成できることがわかる。具体的には、導関数の近似多項式を

$$f(x_0+h) = f_0 + ah + \frac{b}{2}h^2 + \frac{c}{3!}h^3 \tag{8.39}$$

としたとき、

$$\begin{aligned} a &= f_0' \\ b &= \frac{-6(f_0-f_1) - 2\Delta x(2f_0' + f_1')}{\Delta x^2} \\ c &= \frac{12(f_0-f_1) + 6\Delta x(f_0' + f_1')}{\Delta x^3} \end{aligned} \tag{8.40}$$

という形で係数を求めることが出来る。さらに、上の近似多項式を積分してやれば、結局

$$x_1 = x_0 + \frac{h}{2}(f_0+f_1) + \frac{h^2}{12}(f_0' - f_1') \tag{8.41}$$

という形の公式が得られる。これは台形公式と同じく陰的公式である。対称型であることは明らかであろう。

この公式はプログラムが簡単であるわりには精度が高く、また対称型であるというよい性質をもつこともあり、良く使われるようになりつつあるようである。収束させるための反復には普通の直接代入が使える。

8.3.13 対称型一段公式における時間刻みの変更

対称型一段公式には、対称性を保ったままで時間刻みを変更できるというシンプレクティック公式では（普通の意味では）なかった良い性質がある。以下では、どのようにしてそれが可能であるかを簡単に述べておく。

今、任意の対称型一段法があって、その時間刻みを与える関数として $h(x,y)$ が与えられているとする。ここで $h(x,y)$ はなんでもいいが、これも一段法である、つまり前のステップの値とかを必要としないものであるとする。

一般に、 $h(x,y)$ によって刻み幅を決めて求めていった数値解は、時間反転に対する対称性を満たしていない。つまり、1 ステップ積分してから、逆に戻すと、タイムステップが変わってしまうためにもとのところに戻らない。このことのために、例えば周期軌道の場合に誤差が周期的にならないということが起こるわけである。

刻み幅を変えつつ時間反転に対する対称性を保つ一つの方法は（これ以外の方法もあるかもしれないが、今のところ知られていない）、 $h(x,y)$ 自体に対称性を持たせることである。つまり、一段法を

$$y_1 = y_0 + h(x_0,y_0)F(x_0,y_0) \tag{8.42}$$

という形に書いた時、

$$h(x_0,y_0) = h(x_1,y_1) \tag{8.43}$$

がなりたつことを保証するように h を決めるのである。具体的には、上の対称性が成り立っていないような時間刻みの式 h があった時に、

$$h_s = \frac{h(x_0, y_0) + h(x_1, y_1)}{2} \tag{8.44}$$

によって対称化した時間刻みを作ればいいことになる。この場合時間刻み自体が陰的に決まることになるが、とにかく対称性という要求は満たされるのである。

8.4 練習

プログラムが必要なものはプログラムを提出すること。

8.4.1 練習 1

2 次元ケプラー問題

$$\frac{d^2\boldsymbol{x}}{dt^2} = -\frac{\boldsymbol{x}}{x^3} \tag{8.45}$$

について、古典ルンゲクッタと leap frog の両方で、適当な初期条件（解が楕円になるものをとること）から出発して 10, 1000, 10^6 周期後の解の厳密解からの誤差が刻み幅のどのような関数になるか調べよ。ここでは刻み幅は固定とする。

8.4.2 練習 2

4 次のシンプレクティック公式は、 $L(h)$ が刻み幅 h の leap frog を表す作用素であるとして、以下のように書ける。

$$\begin{aligned} S_4(h) &= L(d_1h)L(d_2h)L(d_1h), \\ d_1 &= 1/(2-2^{1/3}), \quad d_2 = 1-2d_1 = -2^{1/3}/(2-2^{1/3}) \end{aligned} \tag{8.46}$$

つまり、リープフロッグでまずちょっと行き過ぎて、次に逆に戻って、最後に最初と同じだけ進んで次の時刻に行くというものである。これをプログラムし、 1 と同様な解析を行なえ。

8.4.3 練習 3

4 次のエルミート公式についても同様な解析を行なえ。説明されているものは陰的公式なので、予測子に何を使えるかは各自検討すること。

8.4.4 練習 4

2.1 で述べた古典的 RK 法で誤差の調整をする方法をプログラムし、離心率の大きなケプラー問題について刻み一定の場合にくらべてどの程度得になるか調べてみよ。

Chapter 9

常微分方程式の初期値問題(3)

9.1 今日の予定

今日は、「硬い」方程式系について、それはどういうものかと、どうやって解くかという話をする。

9.2 硬い微分方程式

さて、硬い微分方程式とは何かということだが、例えば以下のような化学反応系を考えてみる

$$
\begin{aligned}
2A &\leftrightarrow B \\
B &\leftrightarrow C
\end{aligned}
\tag{9.1}
$$

各物質の濃度を y_1, y_2, y_3 と書くことにすれば、それぞれの従う方程式は、例えば

$$
\begin{aligned}
dy_1/dt &= -2k_1y_1^2 + 2k_2y_2 \\
dy_2/dt &= k_1y_1^2 - (k_2 + k_3)y_2 + k_4y_3 \\
dy_3/dt &= k_3y_2 - k_4y_3
\end{aligned}
\tag{9.2}
$$

というようなものになる。

係数がたくさんあって面倒なので、以下 ¡ $k_1 = k_2$¿, ¡ $k_3 = k_4$¿ とし、初期条件として

$$y_1 = 1, y_2 = y_3 = 0 \tag{9.3}$$

という場合だけを考えることにする。

図 9.1 に、 $k_1 = 1, k_3 = 20$ として、前進オイラー法および後退オイラー法で解いた場合の数値解の例を示す。 k_3 が大きいので、A から B に変化するには時間がかかるが B と C の間の変化は速い。この時の定常解は $y_1 = 0.3904, y_2 = y_3 = 0.1524$ というところになる。グラフに示しているのは y_1 と y_2 である。

滑らかに収束しているのは、前進オイラー法で $\Delta t = 0.025$ ととったものと後退オイラー法で $\Delta t = 0.05$ ととったものである。これに対し、振動的に発散してしまっているのは前進で $\Delta t = 0.05$

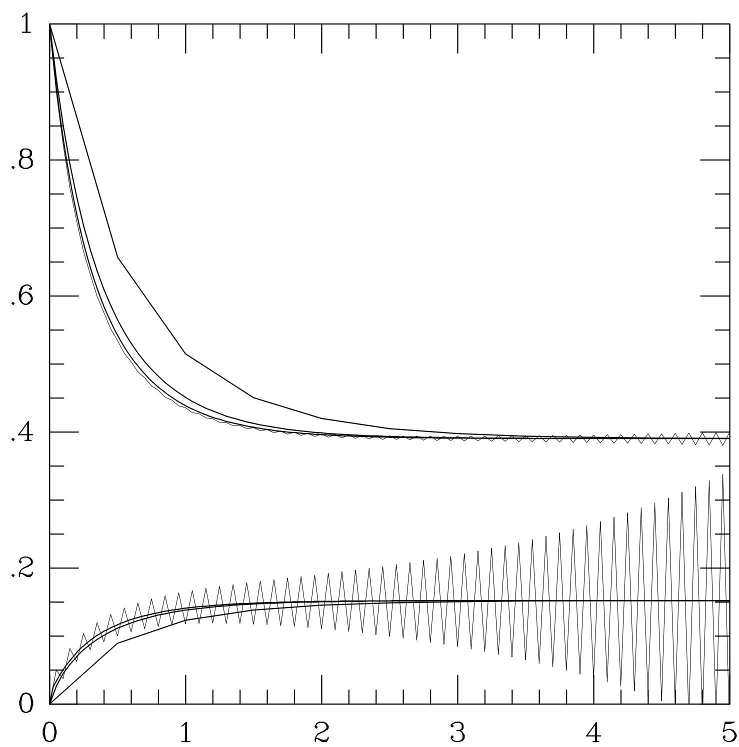


Figure 9.1: 化学反応系の数値計算

の場合である。非常に点が荒く、途中での誤差が大きいが最終的には真の値にいっているのは、後退オイラー法で $\Delta t = 0.5$ と極端に刻みを大きくしたものである。

なお、この計算例では、保存則

$$\frac{y_1}{2} + y_2 + y_3 = \frac{1}{2} \tag{9.4}$$

を利用して、方程式から y_3 を消している。

化学反応系のような非線形の方程式系では、上の例のように速度定数が大きく違うということはそれほど珍しいことではない。上の例では $k_3 = 20$ であるが、これが何桁も大きい場合というのもある。例えば、化学反応といっても天体物理で出てくるような核化学反応を考えると、速度定数が 10 桁くらい違うものが出てくることはそれほど珍しくない。

このような、非常に時間スケールが違う方程式が連立しているような系を「硬い」系と呼ぶ。こういった場合にどういう方法を使うべきかというのが今日考えていきたいことである。

9.2.1 「硬さ」の定義

最初に見たように、例えば化学反応系などでタイムスケールが非常に違うと具合の悪いことが起きるわけであるが、この具合の悪さというのは具体的にはなんだろうか？また、これはどういう意味で「硬い」のであろうか？

例によって、方程式が線形の場合について考える。非線形の場合は、局所的に線形化することで硬さを定義することになる。非斉次の線形常微分方程式

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{A}\boldsymbol{y} + \boldsymbol{\psi}(x) \tag{9.5}$$

の一般解は

$$\boldsymbol{y} = \boldsymbol{\phi}(x) + \sum_{j=1}^{N} c_j \exp \lambda_j x \tag{9.6}$$

で与えられる。ここで、 $\boldsymbol{\phi}$ は方程式 (9.5) を満たす特殊解である。

さて、この方程式が安定であるとすれば、固有値 λ_j はすべて実部が負である。ところで、前回の安定性領域についての議論でちょっと話したように、大抵の数値解法は絶対安定性領域が左側に有界である。これは、言い替えると、時間刻み h を、適当な定数 M を使って

$$h < \frac{M}{\max|\Re\lambda_j|} \tag{9.7}$$

という制限を満たすようにとらないといけないということを意味する。 M は実際には安定性領域の左側の限界値ということになる。

さて、こういった問題で、どのあたりまで計算しないといけないかというのを考えると、実部の絶対値が最小の固有値に対応する成分が十分小さくなるまでということになる。従って、必要なステップ数は、

$$s = \frac{\max|\Re\lambda_j|}{\min|\Re\lambda_j|} \tag{9.8}$$

に比例する程度ということになる。つまり、この値が大きいと、計算が非常に大変になるのである。この値のことを硬さ stiffness といい、これが「非常に大きい」ときに方程式が硬いという。

どれくらい大きい時に硬いというかは曖昧であるが、 10^4 を越えると硬いといって間違いない。実用上は、 10^6 を越えることも珍しくない。

なぜ「硬い」といわれるかという歴史的な事情を紹介しておくと、このような問題は、もともとは機械制御系、つまり、モーターで棒かなにかを回してその先の位置をセンサでフィードバックして制御するようなもので発見された。棒を振り回すとこれは変形するが、棒が「硬い」ほどその変形の周期が短くなる、つまり固有値の絶対値が大きくなるわけである。

なお、この例からわかるように、本当は固有値の虚数部分も問題である。

非線形方程式

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x, \boldsymbol{y}) \tag{9.9}$$

の場合には、そのヤコビアン $\partial\boldsymbol{f}/\partial\boldsymbol{y}$ によって局所的に線形化した方程式について硬さを定義する、つまりはヤコビアンの固有値から決まるということになる。

9.2.2 A-安定性

さて、この硬さというのが問題かどうかということであるが、仮にある公式の絶対安定領域が左半平面 $\Re z < 0$ 全体を含んでいれば、硬かろうがなんだろうが問題ではないということはすぐにわかる。この、絶対安定領域が左半平面を含むという性質を、ダールキストは A-安定性と名付けた。

A-安定性については、ダールキスト自身による次の否定的な結果がある

- 陽的ルンゲ・クッタ法は A-安定でない。
- 陽的線形多段階法は A-安定でない。
- 陰的線形多段階法で A-安定であるものの次数は 2 以下である。
- A-安定である陰的線形多段階法のうち、局所誤差がもっとも小さいものは台形公式である。

証明は面倒なので省略する。

但し、陰的ルンゲ・クッタ公式については以下の素晴らしい結果が知られている

- 対称な p 段 $2p$ 次公式（陰的ガウス公式）はすべて A-安定である。

さらに、Radau 公式、 Lobatto 公式など、端点を含む代わりに次数が低い公式についても同様な結果が得られている。

9.2.3 陰的 RK 法

陰的 RK 法はどんなものかという話を簡単にしておく。陰的 RK には陽的 RK 以上に無限にいろんなバリエーションがあり得るが、特に硬い方程式に使われる方法はほとんど collocation method といわれるものになっている。Collocation method の基本的な考え方は、「数値積分の公式をそのままなんとか常微分方程式の解法に置き換えるために、数値積分を使って RK 法の途中の点での解を求める」というものである。

といっても、これではなんだかわかりませんね。例によってまずもっとも簡単な場合というのを考えてみる。

簡単な数値積分の公式といえば、中点公式か台形公式である。どちらも 2 次精度である。台形公式はそのまま数値解法になってしまうので、以下、自明ではない例として中点公式を考える。

中点公式は、

$$\int_{x_i}^{x_i+h} f(x)dx \sim hf(x_i + h/2) \tag{9.10}$$

とするものである。微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{9.11}$$

に中点公式を使おうという時に困るのは、 y の値に何をいれればいいかわからないということである。

この困難は、数値積分が近似多項式の積分として与えられたということを思い出せば一応解決できる。中点公式の場合、数値積分ではこれは f を $f((x_0 + x_1)/2)$ で近似するということに相当した。同様なアイディアを微分方程式に適用すれば、微分方程式の場合、 f を定数で近似し、解である y を一次式で近似する、つまり

$$\hat{y}(x) = y_0 + f(x_0 + h/2, \hat{y}(x_0 + h/2),)(x - x_0) \tag{9.12}$$

という近似をする。これから、問題の中点での値を $y_{1/2}$ と書くと

$$y_{1/2} = y_0 + f(x_0 + h/2, y_{1/2})h/2 \tag{9.13}$$

と書けるわけで、これは $y_{1/2}$ に対する代数方程式になっている。これを解けば数値解も求まるわけである。これで求まった $y_{1/2}$ を使って、次の数値解 y_1 は

$$y_1 = y0 + hf(x_i + h/2, y_{1/2}) \tag{9.14}$$

と書ける。

数値積分と同じように、途中にとる点の数（RK では段数に相当）をあげれば近似の次数が上がって精度をあげられる。普通の考え方は、点が n 個なら $n-1$ 次多項式ができる。ここでは途中の点の x 座標を $x_1, ...x_n$ と書き、構成された $n-1$ 次多項式を L_{n-1} と書くことにすると x_k での「解」 y_k について

$$y_k = y_s + \int_{x_s}^{x_k} L_{n-1}(x)dx, \quad k = 1, 2, ...n \tag{9.15}$$

がなり立つ。ここで 添字 s がつくのはステップの最初の値である、さらに

$$L_{n-1}(x_k) = f(x_k, y_k), \quad k = 1, 2, ...n \tag{9.16}$$

がなり立つように y_k と L を決め、それを積分して x_e での解が求まるということになる。

と書くと非常に大変そうだが、係数をあらかじめ計算しておけば上の手続きが陰的ルンゲクッタ公式として表現できる。

途中の点のとりかたであるが、普通に考えつく方法は等間隔のラグランジュ補間で、両端を含むようなものである。これは点の数 n に対して次数が n で、まあ悪くはない。また、両端の点を含むので、左端は計算しなくていいことになりちょっと嬉しい。

が、数値積分については、うまく点をとると n 個で最大 $2n$ 次を達成できるという素晴らしい結果が知られている。もっとも高い次数を達成できるのは、 n 個の点を n 次ガウス・ルジャンドル多項式の零点にとるもので、これを普通陰的ガウス公式という。これは 2 点で 4 次とか 4 点で 8 次の精度が達成できるなかなか結構な公式である。

さらに結構なことに、上に述べた A-安定になっているわけである。

さらに、ルジャンドル多項式の性質（対称性）から、陰的ガウス公式は時間反転に対して対称であり、さらにシンプレクティックでもあるということがわかっている。なお、これらのことからわかるように、陰的ガウス公式では安定領域がちょうど左半平面全体であり、虚軸が境界になっている。

この、虚軸が境界になっているというのはまあ結構な性質ではあるが、問題によってはもっと強力に振動を押え込みたいということもある。このような場合には、次数が 1 下がるが Radau 法のような、右端の点を含む公式を使うことがある。このへんのコードも前に紹介した Web サイトにある。

9.2.4 安定な公式を使う上での問題

前節で見たように陰的ガウス法は A-安定であり、基本的にはこれを使えば方程式が硬いということによる問題は起きないといっていい。が、実用上は、この方法にはあまり嬉しくない特性、つまり、大規模な非線形代数方程式を解かなければいけないという特性がある。

もちろん、線形多段階法のところで考えたような逐次（直接）代入法で答が収束すれば、これはたいした問題ではない。しかし、実はこれはうまくいかない。つまり、陰的ルンゲ・クッタに対する逐次代入が収束する条件はリプシッツ連続条件

$$|f(x,y) - f(x,y')| \le L|y - y'| \tag{9.17}$$

が成り立っている時に、 $h < C/L$ (C は公式によって決まる定数だが、1 に近い) で与えられる。ところが、 L は定義から固有値の絶対値の最大値に等しいので、結局陽的な公式と同じように最大固有値でタイムステップが決まってしまい、逐次代入法では A-安定性が失われてしまうのである。

それではいったいどうすればいいかということだが、要するに逐次代入なんていう手抜きな方法を使わないで、もうちょっとまっとうな方法を使えばいい。通常使われるのはニュートン法による反復、つまり、方程式を局所的に線形化し、その線形方程式の解を新しい近似解にする方法である。

原理を後退オイラー法の場合について説明しよう。もちろん、高次の陰的ルンゲクッタやあとに述べる陰的線形多段階法でも理屈は同じである。後退オイラー法の場合、解くべき方程式は

$$\boldsymbol{y}_{new} - h\boldsymbol{f}(x_{new}, \boldsymbol{y}_{new}) - \boldsymbol{y}_{old} = 0 \tag{9.18}$$

である。ニュートン法による i 回目の近似値を $\boldsymbol{y}^{(i)}$ と書くことにすると、 $\boldsymbol{f}$ が $\boldsymbol{y}^{(i)}$ の回りで

$$\boldsymbol{f} \sim \boldsymbol{f}(x, \boldsymbol{y}^{(i)}) + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}(\boldsymbol{y}^{(i+1)} - \boldsymbol{y}^{(i)}) \tag{9.19}$$

と線形化されるから、 $\boldsymbol{y}^{(i+1)}$ を求めるには以下の方程式

$$\left(I - h\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}\right)(\boldsymbol{y}^{(i+1)} - \boldsymbol{y}^{(i)}) = h\boldsymbol{f}(x_{new}, \boldsymbol{y}^{(i)}) + \boldsymbol{y}_{old} - \boldsymbol{y}^{(i)} \tag{9.20}$$

を解けばいいわけである。

すぐにわかるように、ニュートン法は方程式が線形であれば硬さに無関係に一回で収束する。つまり、逐次代入法とは違って A-安定性を保てる。

方程式が非線形の場合にうまくいくか、また、どれくらいの反復回数でうまくいくかというのはなかなか難しい問題である。良く知られているように、ニュートン法は二次収束する、つまり、初期推定が十分に良ければ誤差が一回反復するとまえの誤差の 2 乗程度まで小さくなるという性質（これは、2 次の項の大きさからすぐに証明できる）があるので、うまい初期値が得られれば反復回数は少ない。

が、うまい初期値をとる一般的な方法はない。例えば陽的ルンゲクッタや陽的な線形多段階法で初期値を作ることも考えられるが、これは系が硬い時には悪い推定値を出す可能性もあるからである。比較的うまくいくのは、なにも考えないで $\boldsymbol{y}^{(0)} = \boldsymbol{y}_{old}$ とする、また陰的ガウス公式であれば途中の値についてもそうする方法である。

なお、もともとのニュートン法ではヤコビアンを反復毎に計算し直すが、これらの方法を実装する時には、「収束しているうちはヤコビアンをそのまま使う」という方法で計算量を減らすのが普通である。

9.2.5 完全陰的ルンゲクッタ以外の方法

ここまでで見たように、ガウス法などの陰的ルンゲクッタは硬い方程式にたいして非常によい性質をもつが、一つ大きな問題がある。それは、ニュートン法で解く方程式の大きさが、元の方程式の本数 N と公式の段数 p の積 Np に比例するということである。

ニュートン法で方程式を解くときに、 LU 分解（普通のガウスの消去法のことと思っていい）を使うとすれば、計算量が行列の次元の 3 乗で増える。つまり、公式の段数の 3 乗ということになる。これでは、せっかく高い次数の公式が作れるといっても実用性は低い。

ニュートン法ででてくる線形化した方程式自体を、LU 分解ではなく適当な反復法、たとえば Gauss-Seidel 法、 SOR、共役勾配 (CG) 法といったもので解くことも考えられるが、これらを使うとまた A-安定性が失われたりすることになる。

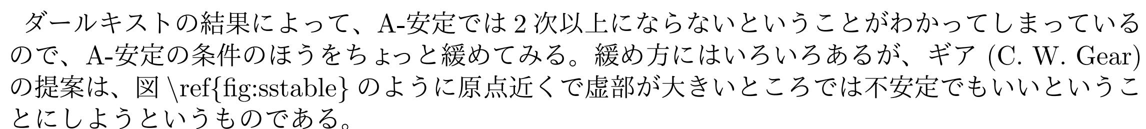
これらの問題をあるていど回避するため、非常にさまざまな公式群が研究されている。以下にその例をあげる

9.2.5.1 ギアの後退差分公式 (BDF)

前に述べたように、計算量と精度の関係を見た時に線形多段階法はルンゲクッタに比べてかなり良いのが普通である。特に陰的公式の時には、解くべき代数方程式が次数が増えても大きくならない。従って、線形多段階法で比較的よい性質をもつものはないかということを考える。

ダールキストの結果によって、A-安定では 2 次以上にならないということがわかってしまっているので、A-安定の条件のほうをちょっと緩めてみる。緩め方にはいろいろあるが、ギア (C. W. Gear) の提案は、図 \ref{fig:sstable} のように原点近くで虚部が大きいところでは不安定でもいいということにしようというものである。

ギアはこのような領域で安定であることを硬安定ということにした。さらに、以下の形の公式



$$\alpha_0 y_i + \alpha_1 y_{i+1} + \cdots \alpha_n y_{i+p} = h\beta f_{i+p} \tag{9.21}$$

であれば、 $p \leq 6$ の場合には硬安定な公式が存在することを示した。

係数の表は適当な参考書をみること。これは、現在でも多くのライブラリやパッケージソフトで使われている。

実際のプログラムの例は例えば http://www.netlib.org/ode/cvode.tar.gz といったものを見てみること。

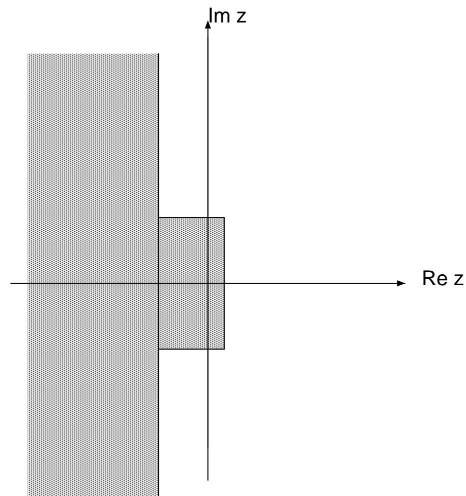


Figure 9.2: 硬安定の定義

9.2.5.2 半陰的ルンゲクッタ

これは無数にある。名前だけをあげると、ローゼンブロック法、 DIRK, MIRK, SIRK, SDIRK といったものがあり、現在盛んに研究されている。

まあ、実用上は、完全陰的なガウスや Radau 公式では問題があるという場合でなければ、わざわざこれらの方法を使う意味はあまりないように思う。

9.3 次週予告

常微分方程式の話はこれでおしまいにして、最適化の話。なお、12/4 は休講にします。

9.4 練習

9.4.1 練習 1

陰的中点公式の安定性領域を求めよ。

9.4.2 練習 2

van der Pol 方程式

$$\frac{d^2x}{dt^2}\mathrm{x}\epsilon(x^2-1)\frac{dx}{dt}=0 \tag{9.22}$$

の数値解を、 $\epsilon=0.1,1,10,100$ の時に適当な陰的公式と陽的公式（後退オイラーと前進オイラーでも OK）で求めてみよ。

9.4.3 練習 3

2 段 4 次の陰的ガウス公式について

1. ルジャンドル多項式の 0 点をとるということから公式を導け。
2. 実際に線形系に適用して、4 次精度であることと A-安定であることを確かめよ。
3. 2 変数以上の場合に適用出来るプログラムを作り、適当な非線形方程式を解いてみよ。例えば van der Pol 方程式で極度に非線形性の強い場合について、4 次の古典的ルンゲクッタと解の精度、信頼性等を比較してみよ。

Chapter 10

最適化 (1)

10.1 概要

この章と次では最適化の話をする。

これは天文学の観測・理論のあらゆる場面で必要になるとても重要な技術である。

現代の天文学の観測においては、例えばなにかを観測してデータをとったとして、それから実際に天文学的に意味があることをいうまでにはいろいろなステップがはいるのが普通である。多くの場合に、これはいろいろな自由パラメータがあるモデルを持ってきて、そのモデルのパラメータを観測データを「もっともうまく説明する」ように決めるということである。例えば銀河内のガスの速度から質量分布を推測するとか、銀河団ガスからの X 線放射から質量を推定するといった場合には、結局そういうことをやっているわけである。もっともうまく説明するとは、具体的にはなんらかの形で誤差を表現して、それを最小化するということである。

これは、形式的には例えばこういうふうな話になる：

「ある領域 D 上で定義された実数値関数 $f(x)$ がある。その最小値とそれを与える $x \in D$ を求めよ」

つまり、最適化というのは要するにこういう話である。

とはいえ、実際にどうやって上の問題の答を求めるかというのは、もちろん領域 D がどんなものかと関数 f がどんなものかによる。例えば、観測データを線形回帰して直線近似するなら、 D は直線 $y = ax + b$ の係数 (a, b) の集合ということになろう。 f は 2 乗残差である。これは 2 次形式の最小化になり、微分すれば連立一次方程式が出てきて解ける。パラメータの数が多くても、2 次形式なら話は同じである。

これに対して、同じような多次元空間内の最適化でも、もとの関数がどんなものか良くわからないとか、計算が面倒であるとかいうと、急に話がややこしくなる。

例えば、図 10.1 は連星重力レンズと推定されたものの観測結果と、それのモデルである。

横軸は時間、縦軸は明るさである。連星レンズの特徴は、単一星の場合と違ってピークが 2 つできることと、そのピークが単一星のばあいよりもずっと明るいことである。

連星レンズの場合、パラメータの数は非常に多い。連星自体の軌道要素が 6 個、その他に質量比、周期、光源の速度、それぞれの我々からの距離ということで 10 個以上ある。なお、このうちいくつかは縮退しているので、本当のパラメータは 9 個である。で、どのパラメータを変えるとなにがどう変わるかというのは簡単にはわからない。こういう時に、どうやってレンズのライトカーブの観測から、物理的な意味を引き出せるのだろうか？

というわけで、世の中には多様な最適化手法がある。これらを簡単にまとめるのが今日の話という

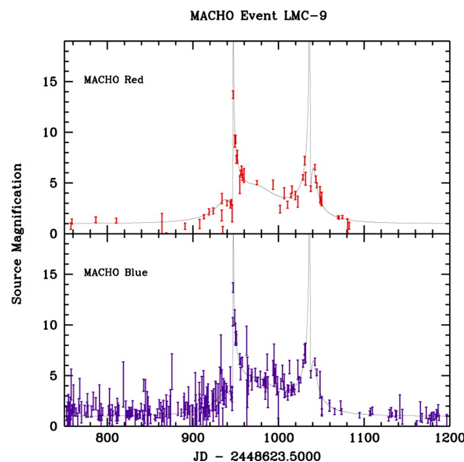


Figure 10.1: 連星重力レンズ

ことになる。

10.2 最適化手法の分類

ここでは、本来分類されるべきものは手法ではなく、問題の分類があってそれに対応して手法があるのかもしれないが、まあ、実際には、問題の定義自体が、「こういう方法で解ける」というのをかっこよくいっただけというところもある。とりあえず、ここで考える分類は以下のようなものということにしておく。

- 決定論的方法
  - 連続関数向け方法
  - 離散的関数向け方法
- 確率的方法

決定論的方法とは、文字通り問題とやり方を決めればあとは機械的に計算が進んで、いつでも同じ答がでるものである。これに対して、確率的方法とは、良さそうな答を捜す時に乱数等を使ってある意味「適当」にやるものである。

適当にやるよりも、真面目にやったほうがいいに決まってるのではないかと思うかも知れないが、必ずしもそういう問題ばかりではない。例えば、ある種の問題では、まともにやって正しい答を求めるのに必要な手間が、問題の大きさの多項式よりも速く増大する。

この一例が巡回セールスマン問題というものである。これは、名前の由来は、「あるセールスマンが一日に沢山のお客を回る時に、どの順番で回るのがもっとも効率的か?」ということであるが、例えば電話線やネットワークの線をどう引くのか効率的かといった問題にも応用される、実用上はいろんなところで出てくる問題である。

素朴な解法は、全部の順序を調べてもっとも短いのを捜すというものだが、これは手間が回る場所の数 n の階乗で増えるので n が 10 を超える当たりから実際的ではなくなる。が、例えば n^p といった、 n の冪乗の手間で正しい解が見つかるような方法は知られていない。

ところが、このような問題に対して「シミュレーテッドアニーリング」や「遺伝的アルゴリズム」といった、確率的な方法を使うと、それが本当にもっとも良い解であるという保証はないが、まあまあそんなには悪くない解が n^2 程度の手間で求まる。

まあ、この辺の詳しい話は来週にして、決定論的方法のほうの分類だが、連続関数というのは定義域が N 次元ユークリッド空間の連続な部分集合で、最適化したい目的関数も連続な実数値関数であるようなものである。そうでないものというのは要するにそれ以外の全部である。上の巡回セールスマン問題は後者の一例である。

この講義では、離散的な場合の決定論的方法はやらない。これは、問題によってあまりに沢山いろんな方法があるので、何をやるべきか良くわからないからである。

10.3 連続関数の最適化

ここで述べる手法は、とりえあえず定義域があまり変な形をしていなくて（というのをちゃんと定義することはできるが、やると話が長くなるので省略）、関数はその定義域のなかで連続で極小値を 1 つだけ持つという場合のためのものである。

この場合、目的関数 f が 2 階微分を持てば、原理的には話は極めて簡単になって、 $\nabla f = 0$ という方程式をニュートン法で解けばいい。が、多くの場合にこれはあんまりうまくいかない。というのは、変数の数 n が多くなると計算しないといけない 2 階微分の数は n^2 に比例して増えるわけで、計算量が増えるだけでなく書かないといけないプログラムの量が増えるという問題があるからである。

微分を数値的にやればいいのではとも考えられるが、これも丸め誤差等の影響があって難しい場合が多い。

もっとも、微分を数値的にやるのではなく、数式的にやる、つまり、もとの関数の数式から数式処理プログラムに生成させたり、あるいはプログラム自体を「微分」する、つまり、目的関数を計算するプログラムから微分を計算するプログラムを自動生成するといった研究もかなり進んではいる。実際に問題を解こうという時には、こういった手法が使えないかどうかも考えてはみるべきであろう。

というわけで、以下はニュートン法とかではない方法。まず 1 変数、それから多変数にいく。

10.3.1 1 変数の場合

まあ、1 変数ならニュートン法でいいのではないかとも思うわけだが、多変数の場合のための準備ということで一応説明しておく。実際には、多変数の問題を解く時に形式的には 1 変数の問題の繰り返しになって、そこでは簡単には 2 階微分が計算できなくてニュートン法というわけにはいかないので、それ以外の方法がいるわけである。

良く本に載っているのは、黄金分割法というものである。これは、以下のような方法である。区間 $[a, b]$ のなかに関数 $f(x)$ の最小値があることはわかっているとする。

1. x_1, x_2 を、 $[a, b]$ をそれぞれ $\sqrt{5} - 1 : 2$ およびその逆に内分する点とする。
2. それぞれの点で関数値 f_1, f_2 を計算する。
3. $f_1 > f_2$ なら最小値は $[x_1, b]$ にあるので、 a を x_1 で置き換え、1 に戻る。細かいことをいえば ¡x_2¿ が 次の ¡x_1¿ になるので、使い回せる。
4. そうでなければ逆に b を x_2 で置き換え、同様に 1 に戻る。
5. 上の全体を $|a - b|$ が十分小さくなるまで繰り返す。

なお、一般には別に黄金分割でなくても、区間内に適当に 2 点とってその大きい方を新しい区間の端にするというので構わない。黄金分割のミソは上の説明の「細かいこと」、つまり、分割点の一方を使い回せるので反復一度について関数計算が 1 度ですむということである。

一度ですむのはいいが、収束は遅い。一度反復した時に区間の幅が 0.62 倍にしかならないからである。これはつまり一次収束で、反復毎に定数分の 1 になるものである。

もうちょっと賢い方法としては、疑似ニュートン法的なものがある。要するに 3 点あれば 2 次関数で近似できるので、その極値を求めようというものである。最適化問題ではない非線形方程式では Secant 法と呼ばれているものの拡張である。 Secant 法程速くはないが、 一次よりも速い収束をすることが知られている。

10.3.2 多変数の場合

多変数の場合にも、黄金分割にあたるような直接探索法というのはあることはあるが、あんまり使えないので省く。大抵の本で最初に出ているのは最急降下法というものである。とりあえず、関数 f が N 次元ユークリッド空間全体で定義されているとする。この方法の原理は、「ちょっと動いた時にもっとも関数値が減る方向に動く」というのを繰り返すことである。もっとも減る方向は、

$$\Delta f = \nabla f \cdot \Delta \boldsymbol{x} \tag{10.1}$$

とテイラー展開の 1 次までとって、 $|\Delta \boldsymbol{x}|$ が一定の時に $|\Delta f|$ を最大にすればよく、もちろん $\Delta x = \alpha \nabla f$、つまり一階導関数自体が方向ということになる。 これで、あとは前節で述べた適当な方法を使ってその方向での 1 次元最適化を適当にやり、あとはまたそこで新しく勾配を求めて同じことを繰り返す。

最急降下法のよいところは、いつかは収束することであり、よくないところは収束が必ずしも速くないことである。これは、2 変数で目的関数が 2 次形式の場合についていろいろ実験したり考えてみたりすればわかるが、結局直交する 2 方向の繰り返しに陥ってしまうからである。この事情は多変数の場合でも同じで、結局 2 方向になっていくということが証明されている。

というわけで、速く答を出そうとするなら、やはり疑似ニュートン法的なものを考えたくなる。1 変数の時のように簡単に 2 次形式の近似式が構成できるわけではないが、その辺をなんとかうまくやろうというわけでいろんな方法が提案されている。

多分良くつかわれているのは DFP（Davidon-Fletcher-Powell）法や BFGS(Broyden-Fletcher-Goldfarb-Shanno) 法で、どちらも考案者の名前である。時々 Variable Metric method（可変計量法） と呼ばれることもある。

以下、基本的な考え方を説明する。

目的関数が、以下の 2 次形式

$$f(x) = \frac{1}{2} x^T Q x \tag{10.2}$$

であるとする。 x は n 次元実ベクトル、 Q は n 次元正方行列で、対称にとって良いので正定値である。

ニュートン法では、 f を 2 次形式で近似してそれの極値を求める。形式的には、近似値 x_0 の回りでのテイラー展開

$$f(x - x_0) = f(x_0) + \nabla f(x_0)^T (x - x_0) + \frac{1}{2}(x - x_0)^T H(x_0)(x - x_0) \tag{10.3}$$

の極値を求めるわけである。ここで H はヘッシアンで、2 階偏導関数を要素とする正方行列である。つまり

$$h_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} \tag{10.4}$$

で、極値を与える点は、 $\Delta x = x - x_0$ として、

$$H(x_0)\Delta x = -\nabla f(x_0) \tag{10.5}$$

である。

目的関数が上の 2 次形式なら、これはもちろん $\Delta x = -Q^{-1}Qx_0 = -x_0$ で、正しい答が求まる。

そういうわけで、考え方としては、ヘッシアンなり Q の近似値を作っていこうというのが基本になる。

ここで、ちょっと定義を続ける。まず、共役という概念を定義する。2 つのベクトル u, v が Q について共役であるとは

$$u^TQv = 0 \tag{10.6}$$

であることである。

さらに、互いに共役な k 個のベクトル $p_0, ...p_{k-1}$ を考え $(k < n)$、これらに対して、

$$H_k = \sum_{i=0}^{k-1} \frac{p_ip_i^T}{p_i^TQp_i} \tag{10.7}$$

を考えると、明らかに

$$H_kQp_i = p_i \quad (i = 0, 1, ..., k-1) \tag{10.8}$$

である。

これは、 H_k が $p_0, ..., p_{k-1}$ が張る部分空間では Q^{-1} みたいなものであるということを意味している。したがって、 p_i を順番に発生させる方法がなにかあれば、それを使って、

$$H_{k+1} = H_k + \frac{p_kp_k^T}{p_k^TQp_k} \tag{10.9}$$

で H_k を計算していけばよい。

ここでの実際的な問題は、互いに共役な p_k を作るよい方法があるかどうかよくわからないことである。一つの考え方は、反復による修正量 $s_k = x_{k+1} - x_k$ を使うことである。 s_k は共役ではないが、それでもなんとか

$$H_{k+1}Qs_k = s_k \tag{10.10}$$

が成り立つように、式 (10.9) を適当に変更する。変更のために右辺に C_k という項を追加することにすると、 これの満たすべき式は

$$(H_k + C_k)y_k = 0 \tag{10.11}$$

ここで y_k は

$$y_k = Qs_k = \nabla f(x_{k+1}) - \nabla f(x_k) \tag{10.12}$$

である。

このように C_k をとる一つの方法 (DFP 法）は、

$$C_k = -\frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} \tag{10.13}$$

とするものである。

まあ、自分でプログラムを書くならこれをつかうので OK であろう。ライブラリとかだと BFGS のほうが少しよいようである。

10.3.3 CG 法

さて、さっき共役な p_k を作る方法はよくわからないと書いたが、これは原理的には知られている。但し、いつでも上手くいくわけではないのでそれを使わない方法も研究されているわけである。

共役な p_k を直接作る方法が共役勾配法、すなわち CG(Conjugate gradient) 法である。これは形式的には簡単である。いま、$g_k = \nabla f(x_k)$ と書くことにして、:eqnarray] p_0 &=& -g_0\ p_k &=& -g_k + \beta_kp_{k-1}\quad (k=1,2, ...)

というベクトル列を考える。ここで β_k は

$$\beta_k = \frac{g_k^T Q p_{k-1}}{p_{k-1}^T Q p_{k-1}} \tag{10.14}$$

である。

f が 2 次関数の時には、これで求まる p_k は互いに共役であることを証明できる。さらに、ちょっと変形すると、

$$\beta_k = \frac{|g_k|^2}{|g_{k-1}|^2} \tag{10.15}$$

となって、これは簡単に計算できる。

なお、疑似ニュートン法、CG 法のどちらの場合でも、方向は決まるが 1 次元問題を繰り返し毎に解く必要はある。これは黄金分割なり疑似ニュートン法なりを使うことになる。多次元の反復では勾配を求めないといけないので少なくとも n 個の関数を計算することになるが、1 次元問題では反復毎に 1 度 f を計算するだけなのでここではそれほど効率に気を使う必要はないことに注意。

10.3.4 CG 法の応用:連立 1 次方程式

CG 法は現在最適化よりも大規模な線型方程式を解くのに広く使われている。今、

$$Ax = b \tag{10.16}$$

という線型連立方程式を考える。で、A は対称行列であるとする。この時、上の方程式は、以下の 2 次形式

$$f(x) = \frac{1}{2}x^t Ax - bx \tag{10.17}$$

を最小化するものと考えることができる。ここで CG 法を使うというのが基本的な考え方である。CG 法なので 1 次元方向の最小化がいるが、これは線型問題なので答がわかっている。

何故こんなややこしいことをするのかと思うかもしれない。理由はちゃんとあって、一つはこの方法は反復法であるにもかかわらず原理的には有限回で収束すること、つまり、ガウスザイデルや SOR とは違って、(計算精度が無限に高いなら) 収束が保証されていることである。もう一つはいろいろ工夫することで収束を非常に速くできることが多いということである。

収束を速くするための工夫は色々な「前処理」と言われるもので、それだけを扱った本がいっぱいあるのでここではこれ以上は扱わない。

10.4 制約つき最適化

大抵の問題では、目的関数の定義域が実数全体ということはない。例えば、銀河の回転曲線から質量分布を求めようというときには、質量はどこでも正でなければならない。

このような制約にはいろんな場合があるが、制約が等式の場合はラグランジュの未定乗数法が使えるのでこれは省略する。不等式の場合は話が難しくなる。

問題によっては厳密にできる場合もあるが、ここではペナルティ法というものを紹介しておく。これは、

$$g_i(x) \geq 0, \quad i = 1, 2, ..., n \tag{10.18}$$

という制約のもとで $f(x)$ を最小化せよという問題を、 f と g_i を適当に組み合わせて作った関数を制約なしで最小化せよという問題に置き換える。具体的には、 $\{g_i(x)\}$ を、

$$\{g_i(x)\} = \begin{cases} 0, & (g_i(x) \geq 0), \\ g_i(x), & \text{otherwize} \end{cases} \tag{10.19}$$

つまり、制約条件を満たせば 0 、そうでなければ 0 でないような関数として

$$F(x) = f(x) + p\sum\{g_i(x)\}^2 \tag{10.20}$$

を最小化する。で、答が求まる度にパラメータ p の値を適当に大きくしていって、こちらの条件に見合う解になったら止める。

この方法では、制約条件のところに解があるとそれに境界の外側から近付く。このために外点法と呼ばれる。内点法というのもあって、これは F を

$$F(x) = f(x) + p\sum\frac{1}{g_i(x)} \tag{10.21}$$

とする。こちらでは、 p を小さくするにしたがって領域の内側から真の解に近付く。

10.5 練習

プログラムが必要なものはプログラムを提出すること。

10.5.1 練習 1

1 変数関数

$$f(x) = -xe^{-x^2} \tag{10.22}$$

の区間 $[0, 2]$ での最小値を黄金分割法で求めるプログラムを作り、答を求めよ。

10.5.2 練習 2

上の関数について、3 点を使う疑似ニュートン法のプログラムを作り、収束が一次より速いことを確認せよ。

10.5.3 練習 3

2 変数の 2 次形式

$$f(x) = 0.5x_1^2 + 5x_2^2 \tag{10.23}$$

について、最急降下法がどのように収束するかをいくつかの適当な初期値について図示せよ。プログラムを書いても手で計算してもよい。

10.5.4 練習 4

式 (10.14) が互いに共役なベクトル列を与えることを証明せよ。

10.5.5 練習 4

問題 3 の収束がどうなるかを CG 法の場合についてしらべよ。

Chapter 11

最適化 (2)

本章では、確率的最適化手法を扱う。

確率的な方法といってもいろいろあるが、現在応用や研究がさかんなのはシミュレーテッドアニーリング (SA) と遺伝的アルゴリズム (GA) である。どちらも、数学的な最適化手法というよりは、物理現象 (SA) や生物の進化 (GA) を真似することで、まあまあの解が得られたらうれしいなという方法である。で、 GA はこの 2 つのなかでも新しい方法であり、理論的裏づけもいろいろはっきりしないところがあるので、今日は主に SA の話をする。

11.1 シミュレーテッドアニーリングの考え

アニーリングとは「焼きなまし」のことである。理論的にはなにかを加熱したあと、ゆっくり冷やすことで熱平衡状態を実現するということである。急に冷やすと熱平衡ではないところで固まってしまう。例えば、安定状態が結晶であるものでも、急速に冷やすと欠陥が多い結晶になったり、あるいはアモルファスで固まったりするわけである。

ここで、熱平衡状態というのは、熱力学的にはエネルギー最低の状態（何が最低になっているかはもちろん境界条件、例えば圧力一定か体積一定かによる）になっているわけで、つまり、世の中のあらゆる物質というのは、単に「ゆっくり温度を変える」というだけで、「エネルギー最低の状態を実現する」という最適化問題を解いていることになる。特に、温度 0 での熱平衡状態は、系のポテンシャルエネルギーを最小化したものになっている。これを実現するためには、系を熱平衡状態に保ちながらゆっくり冷やしていけばよい。

というわけで、そんな風にやろうというのが SA の基本的な考え方ということになる。

11.2 熱平衡状態の実現 – メトロポリス・モンテカルロ

さて、熱平衡状態を実現しないといけないわけであるが、それにはどうすればいいだろうか？とりあえず、古典多粒子系の場合を例に考えてみる。

統計力学的には、温度・体積一定の系でのある物理量 x の値は、アンサンブル平均によって与えられると考えられる。アンサンブル平均は、系のとりうるすべての状態に対して、その出現確率で重みつけして平均をとる。

ここで、ハミルトニアンが $H = T(\boldsymbol{v}) + \Psi(\boldsymbol{r})$ という風に位置の関数と速度（運動量）の関数にわけられる（大抵そうである）場合を考えて、さらに求めたい量が位置だけの関数である場合を考えると、速度については積分を落せるので以下の式がなりたつ。

$$< x >= \frac{\int \exp[-\beta\Psi(\boldsymbol{r})] x dr}{\int \exp[-\beta\Psi(\boldsymbol{r})] dr} \tag{11.1}$$

これが実際に求められればある量 x が求まるということになる。ここで $\beta = 1/kT$ である。

求めたいのは最適解とか最小値で平均でもなんでもないので、こんなのが求まってもしょうがないのではと思うかもしれないけど、もうちょっと我慢して欲しい。

ここで問題なのは、実際には上の積分は有限の計算量では評価できないということである。実際の物理系では自由度がアボガドロ数くらいあるので全く論外だが、例えば原子の数が 100 とか 1000 くらいにしたところでこれは 300 とか 3000 次元での数値積分になる。しかも、ほとんどのところでは出現確率 $\exp[-\beta\Psi(\boldsymbol{r})]$ が非常に小さく、まじめに計算してもしょうがない。

ここで、確率的積分というのを考えるわけである。確率的積分といっても、例えばランダムに粒子をばらまいて、出現確率にしたがって重みをつけて積分していっても、やはりほとんどのところで確率が非常に小さいのであまりうまくいかない。確率が低くないところだけを重点的に拾うような方法が必要である。

これを実現するのがメトロポリス・モンテカルロ法というものである。ちなみにメトロポリスは人の名前で、ロスアラモス研究所の研究スタッフであった。モンテカルロ法を提案した論文は 1953 年のもので、エドワード・テラー他 4 人との共著論文である。ちょうど水爆を作っていたころの話である。

と、そんなことはさておき、メトロポリス法の大事なところは、全くランダムに粒子をばらまくのではなく、今ある分布からちょっと変えてみてエネルギーの変化をみて、それからどうするか決めるということである。

つまり、具体的には、以下のような手順で計算を進める

- ランダムに粒子を一つ選ぶ
- 粒子を動かす向き、大きさをランダムに決める
- 動かした前と後のエネルギー差 ΔE を計算する。
- $\Delta E < 0$ なら新しい配置を採用する。そうでなければ、確率 $\exp(-\beta\Delta E)$ で新しい配置を採用する。
- 配置が新しくなったら、それを記録するなり、それを使って求めたい物理量を計算するなりしておく
- step 1 に戻る。

ここで、粒子を動かす向き、大きさが「適当」でもいいというのが大事なところで、一般には、詳細釣り合いがなりたっていれば、つまり、相互に移動可能な 2 つの状態 1 と 2 について、相互の遷移確率 P_{12} と P_{21} に以下の関係があれば

$$\exp(-\beta\Phi_1)P_{12} = \exp(-\beta\Phi_2)P_{21} \tag{11.2}$$

上のやりかたで平均をとったものは極限で統計力学的なアンサンブル平均に収束するということが証明されている。あ、もう一つ、自明な必要条件として、任意の状態から任意の状態に上の手続きを有限回くりかえすことでいける必要があるというのがある。

したがって、どのように動かすかというのは、上の対称性が成り立っているかぎりなんでもいいことになる。例えば、ある半径の球内の一様乱数でもいいし、ガウシアンとか、もっと変なものでも構わない。計算の手間と収束性を考えると、動かすのをあまり大きくするとほとんど採用されなくなっ

て無駄であるし、逆にあまり小さいと今度は時間をかけてもあまり配置が変わらないことになってやはり無駄である。というわけで、採用される確率が 1/2 くらいになるようにうまくとるのがいいということになっている。

11.3 SA の実現

さて、メトロポリス法はいいとして、それが最適化とどう関係するかという話に戻る。メトロポリス法では、とにかくある温度でのアンサンブル平均を実行できた。しかし、最適化によって我々がやりたいのは、ある関数の最小化であって別に平均でもなんでもない。

ここで、温度 0 の極限を考えると、熱力学的にはもちろん温度 0 の極限はポテンシャルエネルギーの極小値に対応する。が、温度 0 で機械的にメトロポリス法をやってもうまくいくとは限らない。というのは、温度 0 は $\beta = \infty$ に対応し、少しでもエネルギーが増えるような移動はしないということに対応する。したがって、計算を始めたところの近くに局所的な極小値があれば、そこに落ち込んで計算が止まってしまうからである。

局所的な極小値に落ち込まないようにするには、最初は高い温度にしておいて、ある程度位相空間全体を動けるようにしてメトロポリス法を適用し、それから温度を下げてはまたメトロポリス法で動かすというのを繰り返すという方法が考えられる。これが SA 法である。

実際上は、

- 最初の温度をどれくらいにするか
- どれくらいずつ温度を下げるか
- 一つの温度でどれくらいモンテカルロ計算を続けるか

といったことを考えないといけない。この辺は理論がないわけではないが、それは例えば収束性を保証するためには温度は $T_k = T_1/\log(k)$ というふうに繰り返し数 k の対数でしか温度を下げられないというようなものなのであまり役に立たない。実際には、 $T_k = T_0 r^k$ という風に指数関数的に温度を下げても意外にうまくいってしまう。そういうわけで、実際に問題を持ってきた時に上のようなパラメータ（総称してアニーリング・スケジュールということが多い）をどう決めるかは、いろいろ実験してみる必要がある。

11.4 組合せ的最適化への SA の応用

さて、もともとのメトロポリス法では、原理として考えていたのは多粒子系の熱平衡とかそういうもので、あんまり組合せ的最適化という感じはしない。

組合せ的最適化というと、代表的なのは前回にあげた巡回セールスマン問題 (TSP) のような、 $n!$ 個の組合せを調べあげないと答がわからないようなものである。この他にも、ナップザック問題とか最大充足問題とかいろんなものがあるが、 SA を使うという観点からすればどれも同じようなものである。

つまり、 SA という観点からすれば、

- ある近似解について、目的関数を計算する方法
- ある近似解の「近く」の解を「ランダム」に発生する方法

の 2 つを与えることができれば、あとはアニーリング・スケジュールを決めれば答がでる。さらに、前に述べた議論から、近くの解を発生する方法は、対称性さえ満たしていれば適当でいい。収束の速

さとかを考えると適当にするよりいろいろ考えた方がいいのは当然だが、それは収束の速さに影響するだけだしその影響もあまり大きくないのが SA のいいところである。これは、条件が悪いと実際上収束しなくなる最急降下法なんかとはだいぶ違う。

11.4.1 SA で TSP の近似解を求める

TSP は、以下のようにかける。

n 個の点 $p_1, ...p_n$ に対して、点間の移動コスト c_{ij} が与えられているとする。 n 個の点をすべて通ってもとに戻るような巡回路でコストの合計が最小になるようなものを見い出せ。

巡回路自体は、 n 個の点（の番号）の並び $q_1, ...q_n$ で与えられる。この並びは 1 から n までを並べ変えたもの、つまり、ここで ¡$1\ge q_i \ge n, \quad q_i \ne q_j (i\ne j)$¿ということになる。

目的関数は、

$$C = \sum_{i=1}^{n} c_{q_i, q_{i+1}} \tag{11.3}$$

（但し、 $q_{n+1} = q_1$ とする）であるので簡単に計算できる。近くの解の発生のさせ方だが、もっとも簡単なのは q_i と q_{i+1} を入れ換えてみるというものである。もっと大きくつなぎ変えたければ、適当に 2 箇所を選んでそこで切ってひっくり返してつなぐというのも考えられる。こちらのほうが、局所的な最適解に落ちる可能性が少ないという意味では安全である。

定義域が連続的な関数とは違って、動かす量を調整することで採用確率を調整するのは難しいというか、組合せ的最適化の場合にはそういうのを考えてもしょうがない。

実際にプログラムを作る時には、定義式にしたがって目的関数を計算するのではなく、つなぎ変えたことによる変化だけを計算すればいい。これによって一回のモンテカルロ反復に対する計算量を $O(1)$ にすることができる。

例えば、温度の変え方を指数型で $r = 0.9$ とし、一温度での反復は $100n$ とか、採用されたのが $10n$ とか適当に決めて、大抵の場合にちゃんとした答がでるようである。

11.4.2 SA のプログラムの一般的な方針

原理的には、近似解候補を与えた時に目的関数を計算する関数（手続き）と、今の近似解からランダムに新しい近似解を作る手続きの 2 つがあればプログラムは作れる。が、多くの場合に、「新しい近似解と今の近似解との差」を計算するほうが計算量が減る。 TSP の場合であれば、これは $O(n)$ と $O(1)$ で非常に大きい。また、古典粒子系のモンテカルロのように目的関数が全粒子間のポテンシャルエネルギーの和であれば、全部計算すれば $O(n^2)$ だが動かしたことによる変化は $O(n)$ である。このように、変化分を高速に計算する方法を工夫することが極めて重要になる。

11.4.3 もっと高速化する方法

SA は、まあとにかく答がでるのが利点とはいえ、「ゆっくり冷やさないといけない」という条件がつくので計算量は多い。また、メトロポリス法では配置を作っては乱数と比べるというのの繰り返しなので、普通にやったんでは並列化して沢山計算機を使って速くするのも難しい。

最近、さまざまな並列化手法が提案されている。特に注目されているのは温度並列 SA (TPSA) というもので、計算機毎に違う温度でモンテカルロをやり、時々違う温度のもの同士を交換するという方法である。温度がもっとも低いもののところに最終的な結果が求まる。これは「レプリカ交換法」とかいろんな名前がついているが、基本的な発想はみんな同じである。

11.5 練習

11.5.1 問題 1

TSP について、参考プログラム anneal.C を動かしてみて、単位正方形内にランダムに 100 個程度の点をばらまいた時にどんな答がでるか見てみよ。また、アニーリングスケジュールをいろいろ変えてみて、答がどのように変わるか調べよ。

11.5.2 問題 2

規則的に点を並べた場合には、TSP の厳密解が全数探索をしなくても求まる。100 点程度の場合にそのような厳密解がある場合を作ってみて、 SA で厳密解に到達できるかどうか調べよ。

11.5.3 問題 3

参考プログラムではコストがユークリッド距離であるとしている。これをマンハッタン距離 ($|x|+|y|$) にして、答の変化を見てみよ

11.5.4 問題 4

$x = 0.5$ のところに川があって、川を渡るのには λ のコストが余計に掛かるとして答の変化を見てみよ。もっともらしいか？また、どのようにしてもっともらしいと判断したか？

11.5.5 問題 5

「単位正方形の中に n 個の円を重ならないように詰め込めるような円の最大半径はいくつか」という問題を、「円の詰め込み問題」という。いくつかの n については厳密解が知られているが、一般の n について解かれているわけではない。これを SA で解いてみて、どれくらいもっともらしい答がでるか調べよ。目的関数として、 n 個の点間の距離と壁への距離の最小値をとればよい。知られている厳密解のいくつかはまとめられているので、比べてみること。

http://www.cg.inf.ethz.ch/~peikert/personal/CirclePackings/[1]

[1] `http://www.cg.inf.ethz.ch/~peikert/personal/CirclePackings/`