数値計算

数値計算は数学の中の数値解析に分類することができるので、数学の中に紛れ込ませておきます。
細かい理屈は飛ばして数値計算に必要な入門的な情報だけをまとめています。なので、正確性や数値計算における誤差については無視しています。
数値計算には Fortran の方が一般的なんですが (最近はそうでもなくなってきました)、Windows では C 言語の方がインストールしやすいので、C 言語を使います (今では Fortran もインストールしやすくなっていますが)。特にこだわりがないなら、コンパイラ (Borland C++ Compiler 5.5) とセットになっている Visual Windows for BC++を使うのが手っ取り早いです。
実際に実行するときコピーしやすいように、総和の計算、偶数奇数の判定、1,2 次元配列、シュウィンガー・ダイソン方程式のプログラムを source.txt に載せているので、適当なエディタで開いてください (効率性とか綺麗にとか考えて書いてないです)。

まず外観の説明をしておきます。C 言語で数値計算を行う時のプログラムの基本的な構造は

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

int main(void)
{
    return 0;
}
```

こんなかんじです。最初の include <〜> は C 言語に最初から用意されている関数を呼び出すためのものです。stdio.h にはファイルの入出力関係の関数、stdlib.h には絶対値等の関数、math.h には三角関数等の数学関数が入っています (詳しくは C 言語関数辞典 http://www.c-tipsref.com/ なんかを見てください)。int main(void) の {〜} に計算したいものを書いていき、最後に return 0 を書きます。「;」は 1 つの命令文 (ステートメント) が終わることを意味しているので、命令文を加えたら必ず書くようにしてください。
int main{〜} のように書かれている部分を関数と呼び、この場合では main が関数名で、int は戻り値のデータ型を表しています (関数はデータ型 関数名 (変数){〜} のように書く。main 関数には変数部分は必要ない)。プログラムは基本的に main 関数部分を読み取って実行されていきます。とりあえずは、int main(void) と書くのが決まりなのだと思っておけばいいです。この main 関数内に計算したいものを書き込んでいくことになります。
単純な短い計算は 10 行程度で書けてしまいますが、複雑な計算になってくると main 関数内の行数が多くなってしまいます。で、無駄に行数の多いプログラムは非常に見づらいです。なので、別の関数を作って上手いこと短くするのが大切です。例えば、積分を実行したいと思ったとき (細かい書き方は無視して)

```
int main(void)
{
    A=sekibun(〜);

    return 0;
}

double sekibun(〜)
{
    return S;
}
```

のようにして、積分を実行する関数 sekibun を main 関数の外に作るということをします (ちゃんとしたのは最後のシュウィンガー・ダイソン方程式のプログラムの説明を見てください)。この場合では、sekibun 関数内で計算された S という値を main 関数内の A に入れているという構造になっています。
これから分かるように、return は関数内で計算された値を返すためのものです。main 関数は何かの値を返す必要がないので 0 とし、main 関数内で呼び出されている値を返す必要がある関数では return S のように計算した結果を書きます。
main 関数に計算したいものを具体的に加えていきます。例えば、a=2.0,b=3.5 として a+b を計算したいなら

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

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

int main(void)
{
    double a,b,c;

    a=2.0;
    b=3.5;

    c=a+b;

    printf("% f¥n",c);

    system("PAUSE");

    return 0;
}
```

- ▷ double は実数を扱うための宣言で、変数 a,b,c を double 型で定義していることを表しています。データ型としては浮動小数点型と呼ばれます。浮動小数点型ではたとえ 2 とか 5 とかであっても、2.0、5.0 のように書きます。
- ▷ main 関数内で宣言された変数は main 関数内でのみ有効です。main 関数以外に double func{〜} みたいな関数もプログラム内にいたとしても、main 関数で宣言された変数は main 関数内のみ、func 関数内で宣言された変数は func 関数内のみで有効です。
- ▷ a=2.0,b=3.5 には、a に 2.0 を b には 3.5 を入れることを言っていて、c=a+b はそのまま a+b を計算したものが c だと言っているだけです (=の左側が計算結果になるようにする)。
- ▷ printf(〜) はコンソールに結果を出す関数です。"〜"部分にどの文字型で結果を表すかで、% f は小数点以下までを含めて出力するもので (通常は 6 桁)、% e とすれば指数表示になります。¥n は改行です。複数の結果を横並びに表示したいときは printf("% f % f % f¥n",a,b,c)、縦並びなら printf("% f¥n",a);printf("% f¥n",b); とすればいいです。横並びにするときは% f 間をタブで作ると見やすいです。
- ▷ % f の桁数は、例えば、% 10f とすれば小数点込みで 10 桁まで表示、% 7.2f とすれば全体で 7 桁、小数点以下が 2 桁までを表示とすることができます。
- ▷ system("PAUSE") はこれがないとコンソールが即閉じられてしまうので、閉じられないようにするために入れています。

実数を使った数値計算を行うときにはよっぽどの事情がない限り double 型を使うことをお勧めします。float 型というのもありますが、現在ではメモリの節約程度にしかなりません (1G 以上のメモリが標準になっている現状ではよっぽどのことがない限りメモリの節約なんか気にならないですし)。

ファイルに出力したいなら

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

int main(void)
{
    double a,b,c;
    FILE *out;

    a=2.0;
    b=3.5;

    c=a+b;

    out=fopen("A.txt","a");
    fprintf(out,"% f",c);
    fclose(out);
```

```
    return 0;
}
```

- ▷ FILE *out でファイル操作のための out というのを定義しています (out は勝手につけたものなので任意)。
- ▷ out=fopen(”A.txt”,”a”) は、A.txt というファイルを作り、そこに書き込むことを表しています。a ではファイルに追加書き込みしていくようになっていて、w にすると上書きになります。
- ▷ fprintf(out,”% f”,c) で A.txt に c の値を% f の形式で書き込むことを表し、fclose(out) で out によるファイル操作を終わらせるようになっています。
- ▷ out=fopen(”A.txt”,”a”) では追加書き込みなので、プログラムを新しく走らせるたびにどんどん同じファイルに書き込まれていくので、それが嫌なら、remove(”./A.txt”) を FILE *out の下にでも書いておくと、再実行するたびに A.txt を消してくれます。remove が (”〜”) の〜にあるファイルを消すもので、./とすると実行ファイル (拡張子が exe のファイル) と同じフォルダにある A.txt という意味になります。

基本操作はこんなものです。次に、プログラムを書く上で重要な for 文と if 文の説明をします。

- 総和の計算

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

int main(void)
{
    int i;
    double a;

    a=0.0;

    for(i=1;i<=10;i++){

        a+=2.21;
        printf(”i=% d a=% f¥n”,i,a);
    }
    printf(”結果 a=% f¥n”,a);

    system(”PAUSE”);
    return 0;
}
```

- ▷ int は整数を扱うときの宣言で、整数型と呼ばれます。今の場合では i が整数になります。
- ▷ a=0.0 で a の最初の値として 0 を入れています。
- ▷ for(){〜} が指定された回数繰り返し演算を行えという記号です。i=1 で i の最初が 1 だと指定し、i<=10 で最大値が 10 としています。i++は i=i+1 を省略したものです (i++は i=i+1 と書いてもいい)。つまり、for(i = 1; i <= 10; i + +){〜} は、i=1 から始め、i=i+1 によって i=10 になるまで {〜} 内の計算を繰り返すことを表します。<=は < にすることもできて、< では未満になります。また、for(i〜){〜} での {〜} に i が入っているなら、当然その i も ( ) 内の i の変化に対応した i の値になります。
- ▷ a+=2.21 は a=a+2.21 の省略したものです。
- ▷ printf(”% d　% f¥n”,i,a) での% d は整数を表示させるもので、どの i で a の値がどうなっているのか見やすくするために入れています。int 型で宣言されたものに対して% f、double 型に対して% d を対応させるとバグるので気を付けてください。簡単にまとめれば、整数型では% d、浮動小数点型では% f を使えばいいというだけです。
- ▷ このプログラムは a=a+2.21 を 10 回繰り返したものなので、22.1 が最終結果になります。

▷ 単純に文字を表示したいだけなら printf(”結果 % f¥n”,a) のように”〜”内に文字を入れればいいです。i=% d とすることで% d に入る数字の前に i=がつきます。

i の増加の仕方を 1 刻みにしていますが、i=i+2 にすれば 2 刻みになります。そうすると 5 回繰り返すので 11.05 になります。

- 偶数奇数の判定

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

int main(void)
{
    int a,b;

    a=0;

    for(i=0;i<=10;i++){
        a+=1;

        if(a% 2==0){
            printf(”偶数 i=% d a=% d¥n”,i,a);
        }
        if(a% 2==1){
            printf(”奇数 i=% d a=% d¥n”,i,a);
        }
    }

    system(”PAUSE”);
    return 0;
}
```

▷ if(〜){〜} が（〜）内の条件にあっているときは {〜} 内の演算を行うというものです。条件内での== が等しければというので、他にも <=,>=や <, > による大小の判定、!=での等しくないというのがあります。if 文は英語のとおり、もし (〜) であるなら {〜} を計算するというものです。条件分けで計算を行いたいときは if 文を並べていけばいいです。

▷ a% 2 は a を 2 で割った時の余りを計算することを表します。a% 3 とすれば 3 で割った時の余りになります。今は偶数、奇数の判定のために使っているので、2 で割リ切れれば偶数、余りが 1 なら奇数というように判定しています。% による演算は整数型でしかできません。

▷ if(a% 2==0) で余りが 0 なら printf(″偶数　i = %d　a = %d¥n″, i, a) を表示、if(a% 2==1) で余りが 1 なら printf(″奇数　i = %d　a = %d¥n″, i, a) を表示というようになっています。

複数の条件があるときは if 文を続けていけばいいですが、1 つの条件を与えて、それ以外は全部同じとするときは else{〜} としてしまえます。これも英語の意味通りです。今の場合では if(a%2 == 1) {〜} を else{printf(″奇数 i = %d　a = %d¥n″, i, a); } とするだけで同じ意味になります。また、1 つの if 文で複数の条件を加えることもできます。例えば、if(a == 1&&b == 2) とすれば a=1 で b=2 ならという条件になります。a == 1||b == 2 では a=1 か b=2 のときという意味になります。

C 言語の勉強するときにつまづくことで有名な 1 次元配列を簡単に説明します。数値計算上で必要な部分だけを見ることにして余計なことを考えなければ単純な話です。

- 1 次元配列

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

int main(void)
{
    int i;
    double a[5];

    for(i=0;i<=4;i++){
        a[i]=2.0*i;
    }

    for(i=1;i<=4;i++){
        printf("i=% d    % f¥n",i,a[i]);
    }

    c=a[2]+a[3];
    printf("c=% f¥n",c);

    system("PAUSE");
    return 0;
}
```

  - ▷ a[5] が 1 次元配列です。これは a[0],a[1],...a[4] の 5 個の変数を宣言しているようなものです。その各変数に for 文で値を入れて、次の for 文でコンソールに表示させています。
  - ▷ a[N] の要素はただの変数でしかないので、a[2]+a[3] のようにすることができます。
  - ▷ 注意点は a[N] と宣言したとき、a[0],a[1],...,a[N-2],a[N-1] の N 個の要素が作られる点です。そのため、for 文で i を 4 まで取るようにしていますが、5 にするとエラー落ちします。

数値計算上で必要となる知識はこんなものです。1 次元配列と言っていますが、感覚的には N 次元ベクトル (1×N 行の行列) と言ってしまった方が分かりやすいです。1 次元配列の a[5] は 5 次元ベクトル $\boldsymbol{a} = (a_0, a_1, a_2, ..., a_4)$ と同じです。

1 次元配列と言っているように多次元配列も存在していて、C 言語では 2 次元配列までは単純に作れます。

- 2 次元配列

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

int main(void)
{
    int i;
    double a[3][4],b[3][4],c[3][4];

    for(i=0;i<=2;i++){
        for(j=0;j<=3;j++){
            a[i][j]=2.0*i+5.0*j;
```

```
            printf("i=% d   j=% d   % f¥n",i,j,a[i][j]);
        }
    }

    for(i=0;i<=2;i++){
        for(j=0;j<=3;j++){
            printf("% f   ",a[i][j]);
        }
        printf("¥n");
    }

    for(i=0;i<=2;i++){
        for(j=0;j<=3;j++){
            b[i][j]=2.5*i+4.0*j+1.0;
            printf("% f   ",b[i][j]);
        }
        printf("¥n");
    }

    for(i=0;i<=2;i++){
        for(j=0;j<=3;j++){
            c[i][j]=a[i][j]+b[i][j];
            printf("% f   ",c[i][j]);
        }
        printf("¥n");
    }

    system("PAUSE");
    return 0;
}
```

- 1 次元配列はベクトルと同じものであったように、2 次元配列は行列に対応します。
- 話は 1 次元配列と同じで、最初に a[N][M] のようにして、0 ∼N-1, 0 ∼M-1 の要素があることを宣言します (N×M 行列)。その要素に for 文を使って値を入れています。
- printf("% f   ",a[i][j]) の for 文部分は行列に見えるように表示させるための部分です。b[i][j] からは b[i][j] に値を入れる for 文の中に printf を入れています。
- c[i][j]=a[i][j]+b[i][j] によって行列成分の足し算を行っています。

このようにベクトルと行列を作ることができます。ここでの作り方は静的なメモリの割り当てと呼ばれる方法です。もう 1 つ動的な割り当てと呼ばれる方法があって、実用的にはこっちのほうが便利です (これを行うには stdlib.h に入っている malloc 関数を使う必要があります)。ただ、知らなくてもどうにかなるので、この話は飛ばします。

これで大体の必要な情報は揃ったはずです。後は使用できる関数として何があるのかと、どう使えばいいのかという問題になりますが、やっていくうちに身につけていくしかないです。

最後に QED のシュウィンガー・ダイソン方程式を解くためのプログラムを使って、少し複雑になったものの説明をします。解く方程式は

$$B(x) = \frac{\alpha}{4}\frac{1}{x}\int_{\Lambda_{IR}}^{x} dy \frac{3yB(y)}{y+B^2(y)} + \frac{\alpha}{4}\int_{x}^{\Lambda_{UV}} dy \frac{3B(y)}{y+B^2(y)}$$

この方程式に関する情報は場の量子論の「カイラル対称性の力学的破れ」を見てください。
　使うプログラムは長いので、source.txt のシュウィンガー・ダイソン方程式部分を見てください。ちなみに、//はコメントアウトの記号で (c++での機能)、その行はプログラムに反映されません (c 言語で準備されているのは/**/という記号で、これで挟むことで何行でもコメントアウト出来ます)。
　# include の後の# define と書かれている部分はプログラム全体に渡って使用する変数 (というより定数) を定義しています。書き方は

define 記号 値

のようにします。IR が Λ_{IR}、UV が Λ_{UV} です。
　# define の後に int main に行く前の double〜のように書いている部分は、勝手に作った使用する関数を定義する部分です。浮動小数点の値を返す関数を作るときは double 名前 (変数) のように書きます (他のデータ型の値の場合では、double を欲しいデータ型に書き換えればいい)。変数部分では変数のデータ型も書くようにします。ここでは、このように関数を宣言するように書いていますが、int main の前に書く場合では、わざわざ宣言する必要はないです。つまり、新しく作る関数の書き方としては

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

double func(double x,int i);

int main(void)
{
    double X;

    X=2.0*func(2.0,10);

    return 0;
}

double func(double x,int i)
{
    double A;

    A=x+i;

    return A;
}
```

とするか

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

double func(double x,int i)
{
    double A;

    A=x+i;

    return A;
}

int main(void)
{
    double X;

    X=2.0*func(2.0,10);

    return 0;
```

}

とするか選べます。double func(〜) は値を返さなければいけないので、最後に return A と書いて double 型の A の値を返すようにします。そして、main 内で X に、x=2.0,i=10 とした A によって、2.0×A の値を入れるようになっています。

double $x[2*xc+1], y1[2*xc+1], ...$; の部分で 1 次元配列を使っています。$x$ の値は Λ_{UV} から Λ_{IR} の間の値なので x[i] を $x[i] = IR + (UV - IR) * i/xc$ で与えています。BB[i] が求められる $B(x)$ に対応し、B1[j],B2[j] が右辺の第一項にいる $B(y)$、第二項にいる $B(y)$ に対応します。X1[j] は積分を実行するために各 y1[j] の値を入れた第一項の被積分部分の値で、X2[j] も同様です。

各値を入れた X1[j],X2[j] を使ってシンプソン公式によって積分を実行するのが $\mathrm{sim}(IR, x[i], yc, X1)+\mathrm{sim}(x[i], UV, yc, X2)$ です。シンプソン公式は

$$\int_a^b F(x)dx = \frac{h}{3}(F_0 + F_{2n} + 4(F_1 + F_3 + \cdots F_{2n-1}) + 2(F_2 + F_4 + \cdots F_{2n-2}))$$

$$h = \frac{b-a}{2n}$$

となっています。積分範囲 $a \sim b$ を $2n$ 個に分割し、その各点に対応する $F(x)$ を $F_0, F_1, ..., F_{2n}$ としています。プログラムで積分の分割数を 2*yc としているのは、シンプソン公式は分割数 2n での公式だからです。

シンプソン公式の関数で $\mathrm{double\ sim(double\ a, double\ b, int\ n, double\ *sum)}$ として、*sum と書いているのは、ポインタによる配列が入りますよという意味です。なので、$\mathrm{sim(IR, x[i], yc, X1)}$ と書くと、X1 の全要素が入っていることになります。

最小二乗法は $y = ax + b$ での a と b が、$(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ の値が分かっているとき

$$a = \frac{n\sum_{i=1}^n x_i y_i - \sum_{i=1}^n x_i \sum_{j=1}^n y_j}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2}$$

$$b = \frac{\sum_{i=1}^n x_i^2 \sum_{j=1}^n y_j - \sum_{i=1}^n x_i y_i \sum_{j=1}^n x_j}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2}$$

によって求まるというものです。

というわけでこのプログラムは、繰り返す最初の 1 周目 (count=1) では $B(x)$ を 1 だと仮定し、2 周目からは最小二乗法によって $B(x)$ の関数形を一次関数として近似して積分を実行していき、これを繰り返すことで値を収束させるというものです (収束したことを判定するような条件式を入れていないので count=cmax で止まるようになっている)。1 次関数だと近似するのは相当荒っぽいですが、$\alpha = \alpha_{QED}/\pi \simeq 0.4$ 程度でカイラル対称性が破れる結果を出します。ちなみにシュウィンガー・ダイソン方程式を計算するプログラム CrasyDSE が http://theorie.ikp.physik.tu-darmstadt.de/ mqh/CrasyDSE/で公開されています (Mathematica が必要ですが)。

最後に C 言語のちょっとした話をしておきます。C 言語が数値計算にあまり向いていない理由として、複素数を扱う関数が用意されていないというのがよく言われます (Fortran では最初から用意されています)。しかし、C++では複素数が扱えるので、C++が使えるならこの点では困ることがないです。また、C 言語の規格で C99 というのがあって、これも複素数の関数が用意されています。ただ C99 が使えるタダのコンパイラはあまりないです。もし C99 を使いたいなら GCC のコンパイラーを用意するのが後々便利かもしれません。というわけで、複素数の扱いに関しては C 言語でもどうにかなります (今から数値計算をやろうと思う人は C より C++をやったほうがいいかも)。また、行列演算が面倒というのもあります。これはいわゆる整合配列を C 言語では簡単に扱えないという問題からきています。これに対する一般的な対処が動的なメモリ領域の割り当てを使う方法です。しかし、この方法では、関数を自分で作ってやる必要があります。Fortran ではそんな必要がないです。この面倒さが C 言語は数値計算に向いていないとされる理由です。そして、C 言語で一番致命的なのは Fortran に比べて数学関係のライブラリ (誰かが作った計算に便利なプログラムの集まり) の数が少ない点です。これはどうしようもないです。

というわけで、数値計算しかしないと決めている人は Fortran、アプリやゲームとかも作ってみたいと思ってる人は C 言語 (アプリやゲームを作る気が強いなら C++) を使うといいかもしれません。