分子シミュレーションコードの作成チュートリアル

初めて物理シミュレーションに触る研究室の後輩向けに作成したチュートリアルです。作者は流体粒子法 (MPS法)を用いて固液連成計算の研究を行う修士2年の学生です。プログラミングについては決してプロではないので、実装等はあくまでも参考程度にしてください。

目的

福井大学の古石先生のHPを参考に分子シミュレーションのプログラムを自作する。サンプルコードを研究でも活用可能なレベルに引き上げるためのヒントを得る。

対象となる読者

- 今からシミュレーションを用いて研究したい。
- C言語の基礎は勉強した or する (変数、条件分岐、for文、配列、ポインタあたりの知識は持っておきたい)
- 分子動力学法(MD)や離散要素法(DEM)、流体粒子法(SPH,MPS)、散逸粒子動力学法(DPD)といった粒子同士が相互作用する手法を使用することを考えている。

サンプルコードの入手方法

サンプルコードは自由に使用していただいて構いませんが、コードによって発生する不利益に対する責任は一切負いません。

```
git clone https://github.com/keigo-enomoto/MD_tutorial.git
```

コードを更新する場合は

```
git pull https://github.com/keigo-enomoto/MD_tutorial.git
```

チュートリアルの全体像

0. **古石先生のHPで「シミュレーションプログラム作成」まで勉強する。**
   ここまでは、HPを参考に自習してください。筆者なりの回答を`0_Koishi_simulation`のディレクトリ内にまとめています。
1. **0.で勉強した内容を踏まえて、MDシミュレーションのプログラムを作成してみる**
   実は、初学者にとっては0 -> 1が割とハードルがあるとは思いますが、まずはこの作業を試行錯誤しつつ頑張ってみてください。雑には、時間発展とともにポテンシャルエネルギーが下がりなおかつ温度が設定温度付近で揺らぐ程度であればおおよそ成功だと思います。(プログラミングがうまくいっているはずなのに、計算がうまくいかない場合は分子の密度や時間刻み幅、温度を補正する間隔等も調整してみてください。本チュートリアルではMDの計算結果自体はそこまで重要視しません。)
   筆者なりの回答を`1_basic/md_simulation_base.c`に示しています。こちらも参考にしてみてください。
2. **ファイル分割をしてみよう**
   プログラムが複雑になってくるとコード量が増えます。従って、通常は役割ごとにファイルを分割します。ここでは、ファイル分割、ヘッダーファイルの作成、makeファイルの作成を行います。
3. **配列の動的確保を行おう**
   実は`double cd[3*npa]`のように変数を用いて配列メモリを静的に確保することはコーディング規約違反です。`npa`の値はプログラム実行後に決定し、配列の大きさもその後に決定することになります。仮に`npa`の中身が定義不詳であった場合配列を確保することができません。このように配列の大きさがプログラム実行後に決定する場合は`malloc()`や`free()`を使用しましょう。
4. **インプットパラメータの読み込み**
   シミュレーションは仮想的な実験であり、様々なパラメータに対してコンピュータ上で実験を行えることが強みです。`1_basic/md_simulation_base.c`では、各パラメータは直接コード上で定義されていました。これは、パラメータの変更のたびにコンパイルをし直す必要があることを意味しており、安全性の面と効率の面からも全く推奨されません。よって、パラメータファイルを作成し、それを読み込めるようにします。

5. **初期配置の読み込み**
   多くの分子や粒子シミュレーションでは初期配置をどのように設定するのかが非常に重要となります。例えば、MPSでは粒子数密度をできるだけ一定にする必要がある一方で、全てのジオメトリーに対して格子配置を使用することができるとは限りません。このような場合、ランダムな配置から何かしらのスキームで粒子配置を緩和させて初期配置を作成する必要があります。この初期配置の作成に必要な時間はシミュレーション手法によって異なり、平衡化に時間がかかるようなスキームの場合、毎回本計算の度に初期配置を作成していては非効率です。また、同じ初期配置に対して異なるパラメータの計算を行う場合が多く、そういった計算を行う際にも本計算に初期配置の作成プログラムを組み込むことは非効率です。よって、初期配置の作成は別のプログラムで行い、得られた初期配置を記したファイルを本計算で読み取る形がベターでしょう。
6. **出力形式**
   シミュレーションで得たデータを解析することが重要です。そのためには解析のしやすい形式で出力することが重要です。同時に、出力ファイルは容量が大きすぎることが多いのでその対策も必要でしょう。また、シミュレーションでは可視化が非常に重要です。ParaViewやVMDといった有名な可視化ソフトに対応可能なように出力します。
7. **セルリスト**
   粒子シミュレーションでは、周囲の粒子との相互作用をもとに運動方程式を解いていきます。これを愚直に計算すると$N(N-1)$回の演算が必要となり、計算量がN^2に比例します。こうなると粒子数が増えるほど計算量が爆増し現実的ではありません。一般に力の計算にはカットオフ長が存在します。これをうまく利用して、必要最低限の探索範囲に留めることで計算量を$N^{1.5}$のオーダーに落とすことができるアルゴリズムがセルリストです。詳細は割愛しますが実装方法の例を示します。
8. **OpenMP**
   粒子(分子)数が数千〜数十万の単位になってくると計算時間が非常に長くなり現実的ではありません。そこで、複数のコアを用いて演算する並列計算が重要となってきます。ここでは、C言語で最も簡単に並列計算を行うことのできるOpenMPを用いてforループを並列化します。
9. **オプションを与えてみよう**
   bashのコマンドにはさまざまなオプションを与えることができます。同様に、自作プログラムにもオプションを与えて対応できるように改造することが可能です。ここでは、オプション解析について簡単に解説します。

プログラミングの勉強

そもそものコードの書く際の注意点や基礎知識の補填として、以下の参考文献をお勧めしておきます。

- 「リーダブルコード」：言わずと知れた名著です。手元におきつつプログラミングを行うといいかもしれません。
- 「プログラミングの作法」：こちらも有名な名著で、質の高いコードを作成、維持するための種々の問題についてのアドバイスが書いてあります。

↑ちなみに、これらはシステム開発とかのガチのプログラマーになるためのものであって、個人でシミュレーションプログラムの開発を行うだけなら全てマスターしておく必要があるわけではありません。(もちろんこれができていれば、効率は非常に高いんでしょうけれども、、、)。それよりはまず動くプログラムを作成し、詳しい人に自分のコードを開示して改善を続けていく方が重要な気がしています。これまでの「勉強」とは異なり、あらかじめ必要な知識を教科書のようなもので頭に入れておく、といったことは現実的ではありません。走りながら勉強しましょう。

注意点

おそらく全て書き切ることはできません。今後は榎本のGitHubに掲載する予定ですので、必要があればそちらも追ってみてください。

0. 古石先生のHPで「シミュレーションプログラム作成」まで勉強する。

非常に親切に書かれていますので、筆者が作成した解答例 (`0_Koishi_simulation`ディレクトリ内に存在) も参考に自学してみてください。ここでは、いくつかの注意点を述べます。

1. **マジックナンバーの使用は極力減らす。**
   例えば、`3_1_random_bunpu.c`ではforループの回数として`5`と`10`が指定されています。前者の`5`は配列`bunpu`の大きさであり、後者の`10`は乱数のサンプリングを行う回数です。これくらい短いコードなら問題がないかもしれませんが、ある程度長いプログラムになってくると、**変更に弱い上に数字の意味を忘れてしまう等、バグの温床となります。**数式内の数字以外はできるだけ変数として扱うようにしましょう。今回の場合だと例えば

```
int main(){
    int i, d_int;
    int seed = 1;
    double d;
    int n_array, n_sample;
    int bunpu[n_array];
    srand(seed);
    for(i = 0; i < n_array; i++){ bunpu[i] = 0; }
    for(i = 0; i < n_sample; i++){
        d = (double)rand()/RAND_MAX*n_array;
        printf("d = %f\n",d);
        d_int = (int)d;  // C言語ではdoubleをintに変換すると小数点以下は切り捨てされる
        bunpu[d_int]++;
    }
    for(i = 0; i < n_array; i++){ printf("bunpu[%d]=%d\n",i,bunpu[i]); }
    return 0;
}
```

のように`n_array`,`n_sample`、また乱数のシードも`seed`を用いることでマジックナンバーを根絶できました。(ちなみに`bunpu[n_array]`のように変数を用いて静的に配列を定義することは(動くけど)規約違反となりますので、本計算ではなるべく使用しないようにしましょう。)

2. **機能ごとに関数にまとめる**
   `3_3_4_5_init_velocity.c`を例に見てみましょう。ここでは、温度`T`の計算が2回存在します。つまり、全く同じ演算が2回存在するわけなので全く同じコードを2回繰り返すのは非効率です。よって、**このように繰り返し存在する演算は可能な限り一般化し関数にまとめます。**コードの効率面の良さもありますが、**可読性の面からも可能な限りブロックごとに関数にまとめた方がベター**です。
   `md_simulation_base.c`でも大まかなプロセスごとに関数に分割しています。（もっと分割して方がいいかもしれません）オープンソースではメインループ自体も関数にまとめていることが多いです。「関数型プログラミング」とかで検索すると解説が出てくると思います。

3. **掛け算や割り算は一次変数を用いて回数を減らす**
   特にLJポテンシャルの計算のあたりで登場します。基本的には掛け算と割り算は単純な加算減算と比較して時間がかかります。ですので、特にforループの中ではなるべく回数を減らすことによって計算時間の短縮につながります。例としては`4_6_force_Ar`の`r6`や`r12`の使い方を見てみてください。

1. 0.で勉強した内容を踏まえて、MDシミュレーションのプログラムを作成してみる

`0_Koishi_simulation`でMDシミュレーションに必要な内容の各論を勉強することができました。これらを踏まえて動くMDシミュレーションを作成してみましょう。シミュレーションプログラム全体の流れとしては、おおよそ以下の通りです。

```
int main(int argc, char **argv){

    （変数の初期化）
    （位置の初期化）
    （速度の初期化）
    （温度の補正）

    for(メインループの回数){ // whileにしてbreak等を使用してもOK
        （速度の更新(1)）
        （位置の更新）
        （周期境界条件処理）
        for(i = 0; i < 粒子数; i++){
            for(j = i+1; j < 粒子数; j++){
             （力の計算）
            }
        }
        （速度の更新(2)）
        （温度の計算）
        （温度の補正）
    }

}
```

ここで、`main()`の二つの引数は`int argc`はこのプログラムを実行した際の引数の数を示しており、`char **argv`は与えられた引数や実行ファイルのパス、名前等が入った配列です。今後使用するので、よければ覚えておいてください。(詳細については他のHPや本等を参照してください。)

1.1 変数の初期化

- `void non_dimensionalize_parameters(double *n_density, double *n_h, double *n_target_temp)`
  実装したい機能：`density`, `h`(時間刻み幅), `target_temp`を無次元化する

本来は、そもそも無次元化後のパラメータを入力すべきですが、HPと合わせるためにとりあえずこちらからやってみましょう。具体的なアルゴリズム等は`0_Koishi_simulation/1_nondimensionalized.c`に入っています。このファイルの中身を関数`non_dimensionalize_parameters()`に実装します。

ポイント：ポインタを用いて、直接的に変数の中身を更新

基本的に、C言語の`return`は一つのみです。今回の関数では三つの変数を無次元化するため、`main`関数で定義した変数のポインタを受け取り、中身を変更します。ポインタを用いた変数の更新は非常に良く使用するテクニックですので、覚えておきましょう。関数の引数はポインタ変数にしているので、main関数で関数を使用する際には`&n_h`のようにアドレスを与える必要があります。関数の定義内ではポインタ変数の中身をいじるので`*n_h`のように変数の値自体をいじります。この辺りが難しい場合は教科書等で情報を整理したり、その上で小さなプログラムで実験する等を行って理解を深めてみてください。

1.2 位置の初期化

- `void init_position_FCC(int nla, double side_unit, double *cd)`
  実装したい機能：位置を格納する配列`*cd`を初期化する

`0_Koishi_simulation/2_3_menshin3D.c`の中身を関数に実装します。引数の位置の配列`cd`はmain関数ないで「配列」として定義されているので、`cd`という変数自体はそもそもアドレスを指定します。よって、main関数内で関数の引数に与える場合は、1.1と異なり`cd`と入れるだけでOKです。

1.3 速度の初期化

- `void init_velocity(int seed, int npa, double *vl);`
  実装したい機能：速度を格納する配列`*vl`を初期化する。

`0_Koishi_simulation/3_3_4_5_init_velocity.c`の速度の初期化部分を実装します。今後の拡張も踏まえて乱数のシード値も引数`seed`として与えています。

1.4 温度の補正

- `void modify_temp(int npa, double target_temp, double *vl, double *nvl);`
  実装したい機能：設定温度`target_temp`になるように速度を規格化し、配列`nvl`を初期化する。

`0_Koishi_simulation/3_3_4_5_init_velocity.c`の中の温度補正部分を実装します。速度から現在の温度を求める部分は関数`double calc_temp(int npa, double *vl)`に実装しています。

1.5 本計算

1.5.1 速度、位置の更新

- `void update_vel_velret(int npa, double dt, double *nvl, double *fc)`, `void update_position(int npa, double dt, double *cd, double *nvl)`
  実装したい機能：速度ベルレ法で速度と位置を更新する。

非常にシンプルです。`0_Koishi_simulation/5_verlet.c`を参考に、3次元対応できるように実装してみましょう。(シンプルに`3*npa`回のループを回しても大丈夫です。可読性と並列計算、今後の拡張を考慮するとx,y,zそれぞれforループの中に入れた方がいいかもしれません。)

1.5.2 周期境界条件の適用

- `void apply_PBC(int npa, double side, double *cd)`
  実装したい機能：周期境界条件が適用されるように`cd`を修正する。

`0_Koishi_simulation/4_4_5_PBC.c`を参考に実装してみましょう。今回はxyz方向全てのシミュレーションボックスの長さが等しいので`3*npa`回のループとして実装されています。変形等によってシミュレーションボックスの形状が変化したり、直方体にすることをを考えているのであれば、各方向の長さ(例えば`side_x`,`side_y`,`side_z`みたいな変数を用意する)に対して周期境界条件を適用するような実装が好ましいでしょう。

1.5.3 力の計算

- `double calc_force_pot_LJ(int npa, double *cd, double *fc, double side)`
  実装したい機能：LJポテンシャルを用いて粒子間の力の計算を行う。ついでにポテンシャルの大きさも計算する。

`0_Koishi_simulation/4_6_force_Ar.c`を参考に実装してみましょう。**`+=`のような加算演算を行う変数は必ず初期化をしておく(値を0にしておく)ことを忘れないようにしてください。** 今回だと、`zero_force()`を用いて配列`fc`を初期化しているのと、ポテンシャルエネルギーを格納する`pot_erg`を初期化しています。初期化忘れはエラーの原因として多い実装ミスです。

1.6 コンパイルと実行について

古石先生のHPでも解説がありますが、作成したソースファイルをコンピュータが読めるような形に変更する必要があります。この作業をコンパイルと呼びます。作成された実行ファイルを実行することで、ソースファイルにプログラミングした内容をコンピュータに実行させることができます。
今回の場合、例えば以下のようにコンパイルします。

```
gcc -Wall -o md md_program_base.c
```

ここでは、いくつかコンパイルオプションを指定しています。以下に良く使用するオプションを羅列します。

オプション	説明
`-o hoge`	実行ファイルの名前を`hoge`とします。
`-Wall`	エラーメッセージ(Error)だけでなく、警告メッセージ(Warning)も出力します。
`-0`, `-02`, `-03`	コンパイラによって最適化をかけます。数字によって最適化の強さが変化します。

特に`-Wall`オプションは使用した方が良いでしょう。Warning部分はバグを引き起こす可能性があるので、それらがなくなるまでソースコードを修正することを強くお勧めします。コンパイラにWarningを指摘されることによって、C言語への理解も深まると思います。
その後以下のようにして、実行してください。

```
./md
```

以下のようにリダイレクトを使用すると、出力結果をファイルにまとめることができます。

```
./md >> output.txt
```

このようなシェルのコマンドは勉強しておくと非常に生産性が高まります。「シェルスクリプト」とか「Linux コマンド」とかで検索して勉強すると良いでしょう。また、各コマンドの使用方法はターミナル上でも確認することができます。例えば`ls`の使用方法やオプションを知るためには`man ls`とか`info ls`とかをターミナル上で打ち込むと使用方法を確認することができます。一度に教科書的に覚えても忘れるだけなので、必要な時に検索できるような体制にしておくといいでしょう。

2. ファイル分割をしてみよう

`1_basic`では、全てのソースコードを一つのファイルにまとめていました。今回のコードは400に満たないので、このようなことが可能ですが、例えば、時間積分を別のアルゴリズムに適用したい、初期配置を別のものにしたい、、、等の拡張を施していき関数が増えていくと、ファイルの中身がかなり煩雑になります。そこで、通常は機能ごとにいくつかのファイルに分割を行います。これによって、開発がよりスムーズになるはずです。
実際の実行手順として、(1) ファイルの分割とヘッダーファイルの作成、(2) 複数ファイルの際のコンパイル、(3) makeファイルの作成、の三つに分けてファイル分割を完結させていきます。

2.1 ファイルの分割

シンプルに機能ごとにファイルを分割しましょう。今回は例として以下の5つのファイルに分割します。

ファイル名	関数	機能
`cal_force.c`	`void zero_force()`, `double calc_force_pot_LJ()`, `void check_sum_force()`	力の計算
`initial_condition.c`	`void non_dimensionalize_parameters()`, `void init_position_FCC()`, `void init_velocity()`	初期条件の設定
`md_simulation.c`	`int main()`	メイン
`temp.c`	`double calc_temp()`, `void norm_temp()`, `void modify_temp()`	温度の計算、規格化
`update.c`	`void update_vel_velret()`, `void update_position()`, `void apply_PBC()`	速度、位置の更新

関数やファイルの名前の付け方については「リーダブルコード」等を参照してみてください。この作業については、シンプルに関数の定義部分をコピペすればOKです。

2.2 ヘッダーファイルの作成

`1_basic/md_simulation_base.c`では関数の定義を`main()`より下に記述したため、上部に関数の定義を羅列している箇所があります。(プロトタイプ宣言) 同様に、ファイルを分割した際にも、どのファイルにどんな関数が定義されているかを`main()`に知らせる必要があります。そのような役割を果たすのが**ヘッダーファイル**です。(かなりざっくりとした説明なので、K&R等のちゃんと書かれている本も参照してみてください。) 例えば、`printf()`関数は`stdio.h`をインクルードすることによって使用することができます。同様に、2.1で分割した別ファイルに定義されてある関数についてもそれぞれヘッダーファイルを作成します。
以下に一例として`temp.c`のヘッダーファイルである`temp.h`を示します。

```
#ifndef INCLUDE_TEMP_H
#define INCLUDE_TEMP_H
//-----------------------------------------------

//-----------------------------------------------
```

```
//  マクロ定義(Macro definition)
//-------------------------------------------------


//-------------------------------------------------
//  型定義(Type definition)
//-------------------------------------------------


//-------------------------------------------------
//  プロトタイプ宣言(Prototype declaration)
//-------------------------------------------------
double calc_temp(int npa, double *vl);
void norm_temp(int npa, double target_temp, double T, double *nvl);
void modify_temp(int npa, double target_temp, double *vl, double *nvl);
//-------------------------------------------------
#endif
```

こちらのHPに掲載されているテンプレートを拝借しています。このように、通常は関数の具体的は定義部分はソースファイル(.c)に記述し、プロトタイプ宣言をヘッダーファイルに記述します。
よって、今回はメインとなるソースファイル(`md_simulation.c`)以外の4つのファイルに対してヘッダーファイルを作成してください。

また、プログラム全体で必要な型の定義や、マクロ、定数の定義等もまとめて一つのヘッダーファイルに定義しておくといいでしょう。今回は`common.h`に`NUM_FCC_PARTICLE`の定義を記述しましょう。こうすると、例えば`initial_condition.c`でこの定数を使用したいと考えた場合に、当該ファイル内で`#include "common.h"`のようにインクルードすればOKです。このように拡張性が高まるのでオススメです。

2.3 複数ファイルの際のコンパイル

今回のように複数のファイルにまたがる際には以下のようにコンパイルします。

```
gcc -Wall cal_force.c initial_condition.c md_simulation.c temp.c update.c
```

単に、必要なファイルを全て羅列するだけです。実行ファイルを動かして、得られた結果が`1_basic`の結果と同じになっていることを確認しましょう。例えば以下のように`diff`コマンドを使用すると簡単にチェックすることができます。(出力が何もなければ二つのファイル野中身が全く同じということを示しています。)

```
./a.out > output2.txt
diff output2.txt ../1_basic/output_answer.txt
```

2.4 makeファイルの作成

大規模なシミュレーションになればなるほど、一つの実行ファイルの作成に対して多くのソースファイルが必要となります。その度に2.3のように毎回手打ちでコンパイルしていては大変ですし、依存関係を覚えておくのはほぼ不可能です。従って、**make**というツールを使用することが多いです。詳細や高度な使用方法については、例えばオライリー社からも本が出版されていたりするので、そちらを参照してみてください。ざっくりした理解としては、自動でコンパイルするための便利なツールくらいで大丈夫だと思います。

とりあえず、動かしてみましょう。以下の内容を`Makefile`という名前のファイルを作成し、コピーしてみてください。

```
CC=gcc
CFLAGS=-Wall
```

```
LDFLAGS=
LDLIBS=

CFLAGS += -O3
DEBUG_FLUG = -g3

OBJECTS = cal_force.o initial_condition.o md_simulation.o temp.o update.o
TARGETS = md_simulation

all : $(TARGETS)

.SUFFIXES:
.SUFFIXES: .c .o
.c.o:
    $(CC) -c -o $@ $(CFLAGS) $(DEBUG_FLUG) $<

md_simulation : $(OBJECTS)
    $(CC) -o $@ $(OBJECTS) $(LDFLAGS) $(LDLIBS)

clean:
    rm -f core *~ *.o $(TARGETS)
```

ざっくり解説すると、`CC`～`TARGETS`までの行は今後のコンパイルの見通しをよくするためにマクロを定義しています。
`CC`はコンパイラ名、なんちゃら`FLAGS`はコンパイルの際のオプションです。`OBJECTS`は実行ファイルの生成のためのオブジェクトファイルを格納する変数です。ここに、実行ファイルの作成に必要なソースファイルの拡張子を`.o`にしたものを羅列します。`TARGETS`は`make`というコマンドを実行した際に作成する実行ファイルの種類を指定します。今回は(どちらにしろ、一種類しかありませんが、)`md_simulation`とします。

`all : $(TARGETS)`の行以下の内容が、makeの重要な部分です。基本的な構成のイメージとしては以下のような感じです。

```
(作成する実行ファイル) : (コンポーネント (オブジェクトファイル))
    実行ファイル作成のためのレシピ
```

`.SUFFIXES: .c .o`の部分は**サフィックスルール**と呼ばれています。サフィックスとは、接尾語という意味をもち、ソースファイルの拡張子の部分(C言語なら`.c`)のことを指します。通常、`.c`のソースファイルをコンパイルし、機械が読める形になったオブジェクトファイル(`.o`)に変換されます。そして、そのオブジェクトファイルを用いて実行ファイル(例えば`a.out`)が作成され、プログラムを実行することができます。このサフィックスルールを指定することによって、「オブジェクトファイル`.o`を作成するためには`.c`を探してコンパイルしろ」ということを明示しています。

実際に`make`をしてみると以下のようなログが出力されます。

```
gcc -c -o cal_force.o -Wall -O3 -g3 cal_force.c
gcc -c -o initial_condition.o -Wall -O3 -g3 initial_condition.c
gcc -c -o md_simulation.o -Wall -O3 -g3 md_simulation.c
gcc -c -o temp.o -Wall -O3 -g3 temp.c
gcc -c -o update.o -Wall -O3 -g3 update.c
gcc -o md_simulation cal_force.o initial_condition.o md_simulation.o temp.o update.o
```

何をしているかと言うと、上の5行はそれぞれソースファイル(`.c`)をコンパイルし、オブジェクトファイル(`.o`)を生成しています。そして、最終行で生成されたオブジェクトファイル達を用いて実行ファイル`md_simulation`を作成しています。

最後の`clean :` は`make clean`と指定することによって、`rm -f core *~ *.o $(TARGETS)`が実行される、つまりmakeによって生成されたオブジェクトファイルと実行ファイルが削除されると言う操作が行われます。

このようにmakeを用いることによって、より簡便にコンパイルをすることができます。更なる応用として、環境ごとにコンパイラを変更したり(例えば、通常は`gcc`だが、クラスター内では`icc`にするとか)、それに応じてコンパイラオプションを変更したりとかを指定することもできます。

【参考文献】

- C言語 ヘッダファイルの書き方【サンプルフォーマットを公開】

3. 配列の動的確保を行おう

2_split/md_simulation.cで使用される配列は、以下のように確保されています。

```
int npa = NUM_FCC_PARTICLE*nla*nla*nla;   // 全粒子数

double cd[npa*3];   // 位置
double vl[npa*3];   // 速度
double nvl[npa*3];  // 規格化速度
double fc[npa*3];   // 力
```

このように、定義の際に全粒子数を格納する変数npaが使用されています。(ちなみにC99より前のコンパイラだと、「定数」を用いて配列のサイズを決定していないのでエラーになります。回避するためにはconst int npa = **のようにconst修飾子を使用します。)
今回のように、配列を定義する時点でnpaの値が定まっている場合は問題はないですが、npaをインプットパラメータとして読み込む場合など、配列を定義する段階でそのサイズが確定しない場合があります。そして、実用上は粒子数はインプットパラメータとしてプログラムに与える必要があります。よって、**malloc()とfree()関数を用いて配列を動的に確保する必要があります**。配列とメモリの関係等は書籍でしっかり勉強してみてください。

使用例として位置を格納する配列cdを動的に確保してみましょう。

```
double *cd;

// ~~~~~~

cd = (double *)malloc(3 * npa * sizeof(double));

// ~~~ end of the program ~~~

free(cd);
```

まず、cdはポインタ変数として定義します。そして、実際に3*npa個のdoubleの大きさのメモリをmalloc()によって確保し、(double *)のようにフォーマットをした後に変数cdに渡します。ここで、整数(int)の配列を確保する場合はもちろんsizeof(int)としてください。基本的にはmalloc()で確保したメモリは使用後にfree()で解放する必要があります。そうしないと、使用可能なメモリが不足してしまうからです。今回のcdは最後まで使用するので、メインループが終わった後にfree(cd)としてメモリを解放しています。(実際にはプログラムが終了した際に解放されはしますが、、、)

2_split/md_simulation.cでは4つの配列が間違った形で静的に確保されているので、全て動的に確保するようにしましょう。基本的には定義以外では変更点はないはずです。

また、malloc()部分を関数としてまとめることも可能です。今回は、関数内で確保した配列(ポインタ)をポインタ変数cdに渡します。よって、cdと言う変数に格納されるポインタをいじるわけになるので、関数にはポインタのアドレスを渡します(&cd)。関数内では、ポインタ変数の中身*cdにmalloc()で確保した配列を渡します。1.1 変数の初期化では変数の中身を関数内で変更するために、変数のアドレス&n_hを関数の引数として、関数内では変数の中身*n_hをいじりました。これのポインタ変数版だと考えてください。今回は、utility.cにallocate_arrays()として実装しています。この時に、必要となるオブジェクトファイルが増えたので、makeファイルのTARGETも変更することを忘れないようにしましょう。

注意

基本的には一つ変更するごとにテストを行うことが望ましいです。今回の課題は簡単なので、そこまでする必要はありませんが、最初のうちは一つ変更するごとに間違いがないかコンパイルと実行を行い、確認すると良いでしょう。変更点がありすぎるとバグが見つかった時に原因の特定が難しくなります。

【参考文献】

- メモリの動的確保（配列など）

4. インプットパラメータの読み込み

5. 初期配置の読み込み

6. 出力形式

7. セルリスト

【参考文献】

- Computer Simulation of Liquids, M.P. Allen and D.J. Tildesley, p146-155
- 短距離古典分子動力学法における加速化手法の系統的構築 新潟大M2 淡路大輔
- 粒子法入門 越塚 誠一

8. OpenMP

【参考文献】

- 並列プログラミング入門
- OpenMPの基礎 片桐孝洋