計算天文学 II 第 10 回 データ構造とアルゴリズム

牧野淳一郎

2001 年 1 月 25 日

1 データ構造

今回で最後になったので、多少データ構造とかそういう話をすることにしたい。

計算機でプログラムを書く上で基本的なデータ構造というと

1. 配列
2. リスト
3. ツリー構造

といったものであり、基本的なアルゴリズムというと

1. サーチ
2. ソート
3. その他再帰的アルゴリズム

といったところになる。これらをまんべんなくカバーしようとするとそれだけでもう 1 学期必要になるので、今日は、多少話が高度になるがツリー構造を使うすこし大がかりな計算法の話をして、それが実際にどのようにプログラムになるかを見ていくことにする。データ構造としては線形リストとツリー（木）構造、アルゴリズムとしては主に再帰的アルゴリズムを扱う。

2 重力多体問題

対象となる問題は、重力多体問題である。これはつまり沢山の質点がお互いに重力で引き合って運動するというもので、星団、銀河等の基本的なモデルということになる。

運動方程式は

$$\frac{d^2\boldsymbol{x}_i}{dt^2} = -\sum_{j\neq i, 1\leq j\leq N} Gm_j \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|^3}, \tag{1}$$

ということになる。$\boldsymbol{x}_i, m_i$ はそれぞれ粒子 i の位置、質量であり、G は重力定数である。

これをそのまま数値計算してもいいが、2 つの粒子が近付いた時に重力が発散するという問題がある。銀河のシミュレーションの場合には、実際の星の数は 10^{10} 個以上と非常に多いのに、それを多い場合でもせいぜい 100 万とかの数の粒子で表すので、粒子が重い分この問題がおきやすくなっている。この発散をまともに扱うには、粒子毎に時間刻みを変更するような特別な工夫が必要になる。

発散を避けるために、ポテンシャルとして純粋な $-Gm/r$ ではなくて、

$$\phi = -\frac{Gm}{\sqrt{r^2+\epsilon^2}} \tag{2}$$

というふうに $r=0$ で有限になるようにしたものを使うのが普通である。ϵ は定数で、計算結果に影響しないように十分小さくとる。

このようにポテンシャルをなまらしたときには、時間積分は簡単になって一定刻みでやればいい。

実際に沢山の粒子を使う上で問題になるのが、運動方程式の右辺の計算量である。一つの粒子への力を計算するのに他の全部からの力を計算しないといけないので、粒子が N 個あれば 1 ステップ当たりの計算量は N^2 になる。

これをもうちょっとうまくやろうというのが、次に説明するツリー法というものである。

3　ツリー法の原理

計算量を減らすには、遠くの粒子を適当にまとめたい。例えば遠くの方の粒子のかたまりをそれらの重心で置き換えてもいいし、高い精度が必要なら多重極展開も考えられる。遠くにいくに従ってかたまりを大きくしていけば、直観的には一つの粒子への力を $O(\log N)$ で計算でき、全粒子への力であればその N 倍の $O(N\log N)$ で計算できることになる。

もちろん、ここで問題なのはどうやってうまく粒子をかたまりに分けるかということである。ある粒子への力を計算するのに適した分け方を見つけるには、すくなくとも一度は全粒子をみないといけない。従って、その計算量は $O(N)$ より小さくはない。粒子毎に別の分け方をしたら、それだけで計算量が $O(N^2)$ になってしまう。

この問題は、すべての粒子に使える汎用の分け方というのをあらかじめ構成することができれば解決する。それをするのがツリー構造による階層的な空間分割ということになる。

図で説明しよう。3 次元の図は書きにくいし見る方も見にくいであろうから、2 次元で書くことにする（図 1）。まず、平面内の適当な領域に粒子が分布しているとしよう。粒子が有限個なので、そのすべてを含む正方形を考えることが出来る。すべて含んでいればよいので、別に最小である必要はないが、大きくても無駄なだけなので普通はそこそこ小さくとる。これを、まず 4 つの小正方形 (セル) にわける。で、そのそれぞれをさらに 4 つにわけるというのを再帰的に繰り返していく。

粒子が 0 個または 1 個しかなくなったところで止める。この空間（平面）分割に対応したツリー構造を考えると、根に全体の正方形に対応する節点があり、その下に 4 つの小正方形、さらにその下にそのそれぞれの中の 4 つ$\cdots$ と続いていって、ツリーの葉には粒子 1 つ 1 つがくることになる。この方法では、粒子の数密度が高いところでは細かく、そうでないところでは粗いという意味で、適応的な構造が実現されていることに注意してほしい。3 次元にすれば、正方形 4 個の代わりに立方体 8 個となるが、それ以外は全く同じである。

次に、どうやってこのツリー構造を使って一つの粒子への重力が計算できるかということを考えてみる。このためには、ツリーのある節点が表す立方体内の粒子全体からのある粒子への力を計算する方法を与えればよい。（なお、以下では、節点に対応する立方体、あるいはそのなかの粒子全体のことも単に節点ということがある）系全体からの力は、単に根節点からの力として計算できる。一

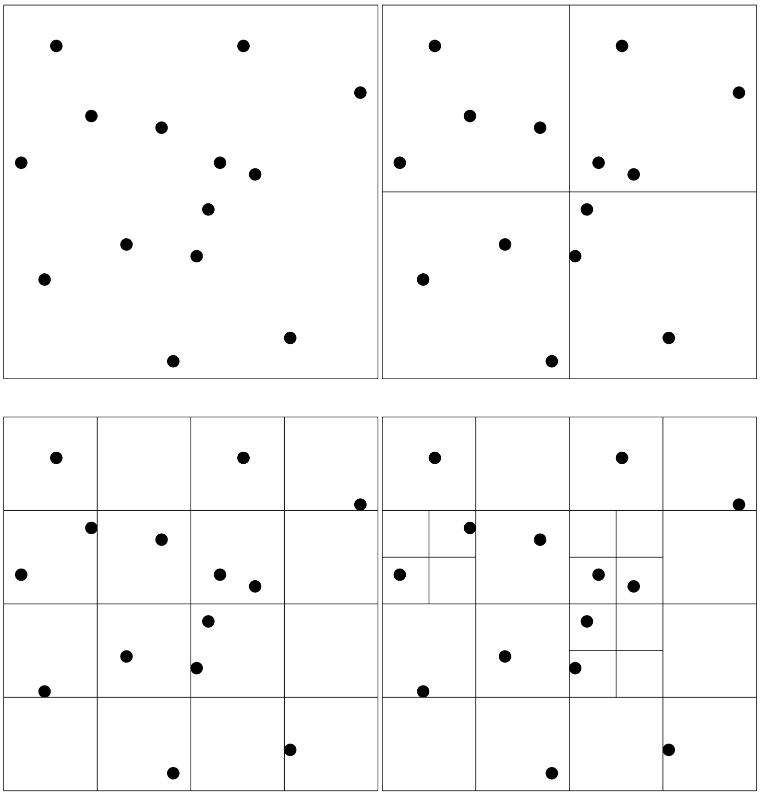

図 1: ４分木の構築

つの節点からの力を以下のように計算することにする

$$\text{ある節点からある粒子への力} = \begin{cases} \text{その節点の重心からの力} \\ \text{(節点と粒子が「十分離れている」)} \\ \text{その節点の子節点からの力の合計} \\ \text{(それ以外)} \end{cases} \tag{3}$$

「十分離れている」かどうかの判定には通常は以下のような基準を使う。

$$\frac{l}{d} < \theta \tag{4}$$

ここで d は粒子と節点の重心の距離、l は節点に対応する立方体の一辺の長さである。パラメータ θ は見込み角といわれる量であり、計算精度と計算量を制御する。高い精度を得たければ θ を小さくすればいいが、漸近的には θ^{-3} に比例して計算量が増える。計算精度をあげるもう一つの方法は、重心だけを考えるのでなく多重極展開を作っておくことである。

ツリー法を実際に使うには、各節点についてその中の粒子全体の重心の位置と質量 (あるいは多重極展開の展開係数) をあらかじめ求めておく必要がある。これらはやはり再帰的に定義することができる。つまり、子節点の中の粒子全体の重心の位置と質量がわかっていれば、親節点の情報は計算できる。多重極展開の展開係数についても (計算は繁雑であるが原理的には) おなじことである。

4 粒子データと線形リスト

というわけで、計算法をきわめて簡単に説明したが、プログラムとしてはどんなふうになるだろうか？以下、実際例を見ていくことにする。

実際のプログラムを書く関係上、どうしても準備が多くなるがまあちょっと我慢を。以下は、「粒子」のデータ構造である。

```
#ifndef  PARTICLE_H
#  define   PARTICLE_H
/*-----------------------------------------------------------------------------
 *  particle : basic class for simple nbody implementation
 *  J. Makino 2001/1/23
 *-----------------------------------------------------------------------------
 */
#include "vector.h"

class particle
    {
    public:
        vector pos;
        vector vel;
        vector acc;
        real phi;
        real mass;
        particle *  next;
        particle *  subnext;
        particle(){
            pos = 0.0;
            vel = 0.0;
            acc = 0.0;
            phi =  mass =  0.0;
        }
};

#endif
```

これは、一応 C++ の「クラス」の文法には従っているが、実際には C の構造体であると思っていい。private メンバはなくてすべてのデータメンバが外から見えるし、メンバー関数も初期設定以外にはなにも使っていないからである。

余談になるが、「粒子」クラスのような、物理的な実体とプログラムの中での構造がほぼ直接に対応するものについては、C++ で一般的に使われるような大げさなデータ構造が適切なものであるかどうかはちょっと疑問である。

ここで、`vector` は前に使ったものとほぼ同じだが、3 次元用に書き下したものを使っている。

あ、それから、`next` と `subnext` が、「粒子」クラスへのポインタになっている。ここでポインタというのは、単純には、ある変数の計算機メモリ内でのアドレスを保持するような変数である。C/C++ の場合には、Fortran とは違ってポインタ変数を極めて柔軟に使うことができ、複雑なアルゴリズムやデータ構造を表現可能にすると同時にプログラムを難解なものにすることにもなっている。

4.1 ポインタとリスト

と、ごたくを並べてもしょうがない。簡単な例で説明してみよう。以下のようなコード断片を考えてみる。

```
    particle a, b, c;
    a.next = &b;
    b.next = &c;
    c.next = NULL;
    for (particle * p = &a; p!= NULL; p = p->next){
        setup_particle(p);
        cout << p->pos << endl;
    }
```

ここで、`&` は、ある変数のアドレスを返す「演算子」であり、演算結果の型はその変数に対するポインタ型になる。逆に、`*` はある変数へのポインタ型がさしているアドレスにあるもともとの変数そのものとして使える。

うーむ、いまいち意味がわかりませんね、、、上のコードが何をするかを見ていこう。まず、`a`, `b`, `c` という `particle` 型の変数 3 個を宣言する。これらにはそれぞれどこか適当なメモリアドレスがコンパイラによって割り当てられるわけである。ここで、`a.next = &b;` とすると、`next` は `particle` 型へのポインタであったので、クラス変数 `a` のメンバ `next` の値が `b` のアドレスである、つまり `a.next` が `b` を指しているということになる。この状態では、例えば `a.next->pos` と `b.pos` は同じように `b` のメンバ `pos` になる。`NULL` は、「このポインタはどこも指していない」ということを示すために使われる特別な値である。

このように、ある構造体（クラス）のメンバに、その構造体自体を指すポインタ変数を入れておくことで、構造体変数を数珠つなぎにしておいてずるずるたどっていくようなことができる。この、数珠つなぎ構造のことを、「線形リスト」という。

この例では配列とどう違うのかわからないが、主な利点は並べかえ（途中に新しい要素を入れたり、抜いたりする）時の手間がデータの数によらないことである。欠点は、最初から順番に見ていくことしかできないことで、途中のデータをぱっと見たいというような時には具合がわるい。

このツリー法のプログラムでは、`next` や `subnext` によって作られる線形リスト構造を、ツリーの各セルの中に入っている粒子の集合を表現するのに使う。ある時点で、あるセルに入っている粒子のリストを表すのに `next` を使い、そのセルの子セルに入っている粒子のリストを表すのに `subnext` を使うということで、2 つポインタがいることになる。

詳しくはもうちょっとあとで。そのまえに、木構造の表現のほうにいこう。

5 木構造と再帰的アルゴリズム

以下が、今回のプログラムで使う木構造のためのクラス定義である。

```
#ifndef  BHTREE_H
#  define  BHTREE_H
/*-----------------------------------------------------------------------------
 *  BHtree : basic class for C++ implementation of BH treecode
 *  J. Makino 2001/1/23
```

```
 *-----------------------------------------------------------------------------
 */

#include "vector.h"
#include "particle.h"

class bhnode
{
private:
    vector cpos;
    real l;
    bhnode * child[8];
    particle * pfirst;
    int nparticle;
    vector pos;
    real mass;

public:
    bhnode(){
        cpos = 0.0;
        l = 0.0;
        for(int i = 0; i<8;i++)child[i] = NULL;
        pfirst = NULL;
        nparticle = 0;
        pos = 0.0;
        mass = 0.0;
    }
    void create_tree_recursive(bhnode * & heap_top, int & heap_remainder);
    void assign_root(vector root_pos, real length, particle * p, int nparticle);
    void dump(int indent = 0);
    int sanity_check();

    void set_cm_quantities();
    void accumulate_force_from_tree(vector & ipos, real eps2, real theta2,
                                   vector & acc,
                                   real & phi);
};

#endif
```

BH というのがやたらでてくるが、これはこのアルゴリズムを 1986 年に提唱した Barnes と Hut (当時はどちらもプリンストン高等研究所。今は Barnes はハワイ大学天文学科) の頭文字をとっているからである。 `cpos` は立方体の中心座標、 `l` は辺の長さである。 `child` は子セル (ノード) へのポインタの配列であり、最大 8 個の子セルがあるので要素数 8 の配列にしてある。これを使うことで、線形リストのように 1 つだけを指すのではなく、8 個の場所を指せるわけで、それによって木構造が表現できるわけである。

`pfirst` は中にある粒子のリストの先頭を指す。 `nparticle` はこのリストの長さである。

ここで、ちょっと注意して欲しいのは、長さは本来必要ではないということである。つまり、リストを作る時に、最後の要素が `NULL` を指しているというふうに決めたので、長さはリストをたどっていって数えればわかるわけで、本来長さを別に覚えておく必要はない。が、数えなくても長さがわかると便利なことが多いので別に覚えておくことにする。このようにした場合には、情報が冗長になっているので 2 つの情報が矛盾しないように、あるいはちゃんと意味があるほうを使うように注意する必要がある。

実際に木を作る手順を見てみる。最初にすることは、1 つ立方体をつくって、それをルート (根) とし、全粒子をその中にあるということにする、具体的には、全粒子をルートの `pfirst` から始まるリストに入れておくことである。以下がそれを実現するプログラムになる。

```
void bhnode::assign_root(vector root_pos, real length, particle * p, int np)
{
    pos = root_pos;
    l = length;
    pfirst = p;
    nparticle = np;
    for(int i=0;i<np-1;i++){
        p->next = p+1;
        p++;
    }
    p->next = NULL;
}
```

これはメンバ関数なので、`bhnode` クラス変数のメンバはメンバ名だけで使える。中心座標、辺の長さと、粒子の配列、粒子数が引数で渡ってくるので、中心座標、辺の長さをメンバ変数に代入し、`pfirst` から始まるリストを配列の要素を順にたどっていくように作っておく。

次に実際に木構造を作っていく。これは、以下のようになる

```
void bhnode::create_tree_recursive(bhnode * & heap_top, int & heap_remainder)
{
    if (heap_remainder <= 0){
        cerr << "create_tree: no more free node... exit\n";
        exit(1);
    }
    particle * psub[8];
    int nsub[8];
    for(int i = 0;i<8;i++){
        psub[i] = NULL;
        nsub[i] = 0;
    }
    particle * p = pfirst;
    for(int i=0 ; i<nparticle; i++, p = p->next){
        int subindex = 0;
        for(int k=0;k<3;k++){
            subindex <<= 1;
            if (p->pos[k] > cpos[k]){
                subindex += 1;
            }
        }
        nsub[subindex] += 1;
        p->subnext = psub[subindex];
        psub[subindex] = p;
    }

    for(int i=0; i<8;i++) if(nsub[i] >0){
        child[i] = heap_top;
        heap_top ++;
        heap_remainder -- ;
        child[i]->pfirst = psub[i];
        child[i]->cpos = cpos + vector( ((i&4)*0.5-1)*l/4,
```

```
                                        ((i&2)    -1)*l/4,
                                        ((i&1)*2  -1)*l/4);
        child[i]->l = l*0.5;
        child[i]->nparticle = nsub[i];
        for(p=psub[i]; p->subnext != NULL; p = p->subnext){
            p->next = p->subnext;
        }
        if (nsub[i] > 1){
            child[i]->create_tree_recursive(heap_top, heap_remainder);
        }else{
            child[i]->pos  = child[i]->pfirst->pos;
            child[i]->mass = child[i]->pfirst->mass;
        }
    }
}
```

ここで、渡ってきている引数は、実際には bhnode 構造体の配列の先頭アドレスと、その配列の要素数、というか正確にはその配列のどこか途中のアドレスと残りの要素数である。この配列は、木構造全部を作るのに十分な程度の大きさをとっておく。木に新しいノードを付け加えるためには配列の先頭から順番に要素を割り当てていく。

一般には、木構造のような動的な構造を作るには、 new 演算子を使って必要に応じて新しいメモリ領域を割り当てる。が、ここでは木構造は時間積分のステップ毎に新しく作るので、 new をその度に使ってはメモリが足りなくなるとかややこしい問題がある。そこで、自前でメモリを配列に確保しておいてそこにあるものを使うことにする。足りなくなったらここではエラーを出して終る。

まず、最初にすることは、自分の下にある全粒子をたどっていって、それがどの子ノードにいくかを判定し、結果に応じて対応するリストに挿入（先頭に追加）することである。ここで、 next は自分のリストなので、これを書き変えるとその後がたどれなくなる。このため、子ノード用のリストには subnext を使うことにする。判定は、各座標方向に対して、中心より上か下かを見るだけである。

全部の粒子の振り分けがすんだら、次は 1 つ以上の粒子が入っている子ノードにちゃんとクラス変数を割り当て、その中身（位置と大きさ）を設定するのと同時にリストのほうも subnext を next にコピーしなおしておく。

で、粒子の数が 2 個以上なら、今度はこの子ノードをさらに分割する。これは、ここではこの関数自身を再帰的に呼び出すことで実現されている。粒子が 1 つだけなら、その質量、位置を木構造データのほうの変数にコピーしておく。

6 ノードの物理データ

力を実際に計算するまえの最後の準備として、各ノードについてそのなかの粒子の重心と総質量を出したい。これは以下のようになる。

```
void  bhnode::set_cm_quantities()
{
    if (nparticle > 1){
        int i;
        pos = 0.0;
        mass = 0.0;
```

```
        for(i=0;i<8;i++){
            if (child[i] != NULL){
                child[i]->set_cm_quantities();
                real mchild = child[i]->mass;
                pos += mchild*child[i]->pos;
                mass += mchild;
            }
        }
        pos /= mass;
    }
}
```

粒子の数が 1 であれば、既に粒子データがコピーされているので別になにもしなくていい。1 より大きければ、実際にある子ノード全部を見て、モーメント、質量を出し、モーメントを質量で割れば重心がでる。

ここでも再帰を使っている。つまり、実際に `child[i]->mass` 等を使う前に、`child[i]` について質量、重心を計算させているのである。

7 力の計算

さて、このように木構造ができてしまうと、実際の力の計算は非常に簡単になる。以下が力を計算する関数である。

```
void bhnode::accumulate_force_from_tree(vector & ipos, real eps2, real theta2,
                                        vector & acc,
                                        real & phi)
{
    vector dx = pos - ipos;
    real r2 = dx*dx;
    if ((r2*theta2 > l*l) || (nparticle == 1)){
        accumulate_force_from_point(dx, r2, eps2, acc, phi, mass);
    }else{
        for(int i=0;i<8;i++){
            if (child[i] != NULL){
                child[i]->accumulate_force_from_tree(ipos,eps2,theta2, acc, phi);
            }
        }
    }
}
```

これはメンバ関数になっていて、あるノードから引数で渡ってくる場所への力（加速度）とポテンシャルを計算する（渡ってきた変数に足し込む）。距離の 2 乗を計算し、それが $r^2\theta^2 > l^2$ を満たしていれば実際に加速度を計算する。2 乗のままで比べるのは、単に平方根計算をしないですませるためである。また、粒子数が 1 であればやはり同様に加速度を計算する。そうでなければ、子ノードを順番にみて重力を計算する。ここでも再帰をつかう。

質点からの力、ポテンシャルの計算は以下のようになる。

```
void accumulate_force_from_point(vector dx, real r2, real eps2,
                                 vector & acc,
                                 real & phi,
                                 real jmass)
{
    double r2inv = 1/(r2+eps2);
    double rinv  = sqrt(r2inv);
    double r3inv = r2inv*rinv;
    phi -= jmass*rinv;
    acc += jmass*r3inv*dx;
}
```

ベクトル型というのを作っておいたので、記述が簡潔になっていることがわかる。

8　その他

というわけで、一応木構造をつくって重力計算をするためのデータ構造、アルゴリズムとその実現をきわめて簡単に紹介した。これを実際に多体問題の数値解を求めるプログラムにするためには、時間積分、初期条件の生成 / 読み込み、結果の表示 / 解析等いろんなことをするプログラムを作る必要がある。

また、それ以前の問題として、正しく加速度が計算できているかどうかチェックする必要ももちろんある。

これらは、今回のオプション課題ということにさせてもらう。

9　レポート課題

今回の話は、まあちょっと難しいというかいかにも講義回数が足りないので、これに関して必須のレポート問題はなし。というわけで、レポート課題は以下の通り。

1 は必須。後はオプション。 プログラムが必要なものはプログラムを提出すること。

1. この講義全体について、感想、意見、希望等をなんでも好きなだけ書いて出す。別に褒めてないと点が下がるとかそんなことはないので、率直な意見を書いてくれると参考になってありがたい。
2. 参考プログラム

   http://grape.astron.s.u-tokyo.ac.jp/~makino/
   　　　kougi/keisan_tenmongakuII/programs/index.html

   にある `BHtree.C` を動かしてみて、総当たりで計算するのと木構造で計算するのとで計算した加速度、ポテンシャルの値を比べ、θ を小さくしたときに差がどのように小さくなるか、また小さくなりかたは粒子数によってどのように変わるか調べてみよ。
3. 上の誤差が理論的にはどうなるべきか考察せよ。
4. 計算時間はどうなるか、粒子数 100 程度から 10,000 程度までで調べてみよ。
5. 時間積分や結果表示の関数を付け加えて、実際に多体問題を計算してみよ。時間積分公式としては第 6 回でやった leap frog を使おう。保存量である系全体のエネルギーや角運動量はどの程度保存するか ？時間刻みや θ を変えるとどうか ？