計算天文学 II 第 11 回 データ構造とアルゴリズム (2)

牧野淳一郎

1 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。

再帰を使うことで、ちょっと考えると大変そうな処理が非常に簡単に実現されている。

2 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;
}
```

3 各関数のテスト

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。

それらに入る前に、とりあえず今まで出来たところを動かしてみることにする。

割合いろんな関数を作っていろんなことをやらせるので、全体を動かして時にちゃんと動いているかどうかをどうやって判断し、さらに動いていない時にどこがおかしいかを見つけ出すのはそんなに簡単ではなくなる。

ある程度大きな、沢山の関数からできたプログラムをテスト・デバッグするための基本的な考え方は以下の 2 つである。

1. それぞれの関数を単体でテストする。

2. 組み合わせたプログラムをテストするが、動作が正しいかどうかを関数ごとにチェックする。

もちろん、プログラムをデバッグする時の大きな問題は、

- プログラムは書いた人の意図した通りにではなく、書いた通りに動く
- プログラムを書いた人の意図が正しいとは限らない

ということである。特に後者がデバッグを難しいものにする。

というわけで、まず、ツリー構造を作り、それが正しいかどうか調べる。それから実際に重力を計算し、それもツリー構造を使わないで総当りで計算したものとくらべてみよう。メインプログラムは以下のような感じになる。

```
#ifdef TEST
int main()
{
    particle * pp;
    int n;
    cerr << "Enter n:";
    cin >> n ;
    pp = new particle[n];
    double rsize = 1.0;
    create_uniform_sphere(pp, n, 0 , rsize);
    for(int i =0;i <n;i++){
        PRC(i); PRL(pp[i].pos);
    }
    bhnode * bn = NULL;

    int nnodes = n*2;
    bn = new bhnode[nnodes];
    bn->assign_root(vector(0.0), rsize*2, pp, n);
    int heap_remainder = nnodes-1;
    bhnode * btmp = bn + 1;
    bn->create_tree_recursive(btmp,  heap_remainder);
    PRL(bn->sanity_check());
    bn->set_cm_quantities();
    bn->dump();
    double  eps2 = 0.01;
    calculate_uncorrected_gravity_direct(pp, n, eps2);
    cerr << "Direct force \n";
    particle * p = pp;
    for(int i = 0; i<n; i++){
        PR(i); PR(p->pos);PR(p->phi); PRL(p->acc);
        p++;
    }
    cerr << "Tree   force \n";
    clear_acc_and_phi(pp, n);
    p = pp;
    for(int i = 0; i<n; i++){
        calculate_gravity_using_tree(p, bn, eps2, 0.5);
        PR(i); PR(p->pos);PR(p->phi); PRL(p->acc);
        p++;
    }
    return 0;
}
#endif
```

プログラム全体は

http://grape.astron.s.u-tokyo.ac.jp/%7Emakino/kougi/keisan_tenmongakuII/programs/simpletree/B

にあるので細かい関数については省略するが、関数 sanity_check はちょっと中を見ておこう。

```
int bhnode::sanity_check()
{
    int i;
    int iret = 0;
    if (nparticle == 1 ){
        // this is the lowest level node. Things to check:
        // particle is in the cell
        particle * p = pfirst;
        if(inbox(pos,p->pos,l)){
            cerr << "Error, particle out of box ... \n";
            dump();
            return 1;
        }
    }else{

        // This is the non-leaf node. Check the position and side
        // length of the child cells and then check recursively..
        for(i=0;i<8;i++){
            if (child[i] != NULL){
                int err = 0;
                err = child[i]->sanity_check();
                if (l*0.5 != child[i]->l) err += 2;
                vector relpos = cpos-child[i]->cpos;
                for (int k = 0 ; k<ndim;k++){
                    if (fabs(relpos[k]) !=l*0.25)err += 4;
                }
                if (err){
                    cerr << "Child " << i << " Error type = " << err << endl;
                    dump();
                }
                iret += err;
            }
        }
    }
    return iret;
}
```

ここでは、以下のようなチェックを行なっている。

1. ノードに粒子が 1 つだけなら、それがちゃんと中にあるかどうか。
2. そうでなければ、自分の子供は正しい位置、大きさを持つかどうか。

で、さらに再帰的に子供についても同じチェックをする。

inbox はこんなものである。

```
int inbox(vector  & cpos, // center of the box
          vector  & pos,  // position of the particle
          real l)         // length of one side of the box

{
    for(int  i = 0; i< ndim; i++){
        if (fabs(pos[i]-cpos[i]) > l*0.5) return 1;
    }
    return 0;
}
```

ここではベクトルの各要素についてそれぞれ調べる必要がある。

さて、この関数が「エラー」を返して来たとしても、少なくとも以下の 4 通りの可能性がある。

- ツリーを作る関数が間違っている。
- sanity_check が間違っている。
- ツリーを作る以前に既になにか間違っている。
- 上の 3 つの任意の組み合わせ。

というわけで、最初は粒子 1 つとか 2 つで何ができるかみるといったところから始めないと後が大変ではある。逆に数が少なければいろいろ書き出して人間がチェックするのも不可能ではない。以下の関数はツリー構造そのものを再帰的にプリントアウトする。 spc は単に引数の数だけ空白文字を印刷する関数である。

```
void bhnode::dump(int indent)
{
    int i;
    spc(indent); cerr << "node center pos " << cpos ;
    cerr << endl;
    spc(indent); cerr << "node cm  " << pos << " m " << mass ;
    if (nparticle == 1){
        cerr << " IS LEAF" ;PRL(nparticle);
        particle * p = pfirst;
        for(i = 0; i < nparticle; i++){
            for(int j=0;j<indent+2;j++)cerr << " ";
            PRL(p->pos);
            p= p->next;
        }
    }else{
        cerr << " IS _not_ LEAF ";PRL(nparticle);
        for(i=0;i<8;i++){
            if (child[i] != NULL){
                child[i]->dump(indent + 2);
            }
        }
    }
}
```

以下は $N = 2$ での実行結果である

```
BHtree
Enter n:2
nbody = 2,  power_index = 0
i = 0,  pp[i].pos = -0.20707  0.680971  -0.293328
i = 1,  pp[i].pos = -0.106833  -0.362614  0.772857
bn->sanity_check() = 0
node center pos 0  0  0
node cm  -0.156952  0.159178  0.239765 m 1 IS _not_ LEAF nparticle = 2
  node center pos -0.5  -0.5  0.5
  node cm  -0.106833  -0.362614  0.772857 m 0.5 IS LEAFnparticle = 1
    p->pos = -0.106833  -0.362614  0.772857
  node center pos -0.5  0.5  -0.5
  node cm  -0.20707  0.680971  -0.293328 m 0.5 IS LEAFnparticle = 1
    p->pos = -0.20707  0.680971  -0.293328
Direct force
i = 0 p->pos = -0.20707  0.680971  -0.293328 p->phi = -0.33364 p->acc = 0.014891  -0.155032  0.158389
i = 1 p->pos = -0.106833  -0.362614  0.772857 p->phi = -0.33364 p->acc = -0.014891  0.155032  -0.158389
Tree   force
i = 0 p->pos = -0.20707  0.680971  -0.293328 p->phi = -0.33364 p->acc = 0.014891  -0.155032  0.158389
i = 1 p->pos = -0.106833  -0.362614  0.772857 p->phi = -0.33364 p->acc = -0.014891  0.155032  -0.158389
```

それらしいツリー構造ができていること、力の計算結果が総当りとツリーを使ったもので一致していることがわかる。もちろん、粒子が 2 個の時にはツリーを使っても相手は 1 個なので、結果は完全に一致しなければおかしい。まあ、確かに一致しているというわけである。

2 粒子でそれらしく動いたからといって正しいということにはならない。もっと大きな粒子数でいろいろ調べ、さらに opening angle も変えて調べる必要があるがこれは課題ということにする。

4 時間積分

時間積分には前にやったリープフロッグ公式を使うことにする。どんなふうに書いてもいいが、今回は、分かりやすいと思われる

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

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

の形を使うことにしよう。以下の関数を particle クラスに入れる。

```
        void predict(real dt){
            real dt2 = dt*dt*0.5;
            pos += dt*vel + dt2*acc;
            vel +=  (dt*0.5)*acc;
        }
        void correct(real dt){
            vel +=  (dt*0.5)*acc;
        }
```

時間積分は以下のように、まず予測子を計算、それから加速度を計算し、最後に速度の修正子を計算する。

```
void integrate(bhnode * bn,
               int nnodes,
               particle * pp,
               int n,
               real eps2,
```

```
                real theta,
                real dt)
{
    for(int i = 0;i<n;i++) pp[i].predict(dt);
    calculate_gravity(bn,nnodes,pp,n,eps2,theta);
    for(int i = 0;i<n;i++) pp[i].correct(dt);
}
```

重力を計算する関数はこんな感じになる。

```
void calculate_gravity(bhnode* bn,
                       int nnodes,
                       particle * pp,
                       int n,
                       real eps2,
                       real theta)
{
    real rsize = calculate_size(pp,n);
    bn->assign_root(vector(0.0), rsize*2, pp, n);
    int heap_remainder = nnodes-1;
    bhnode * btmp = bn + 1;
    bn->create_tree_recursive(btmp,  heap_remainder);
    //    bn->dump();
    //    PRL(bn->sanity_check());
    bn->set_cm_quantities();
    clear_acc_and_phi(pp, n);

    particle * p = pp;
    for(int i = 0; i<n; i++){
        calculate_gravity_using_tree(p, bn, eps2, theta);
        //        PR(i); PR(p->pos);PR(p->phi); PRL(p->acc);
        p++;
    }
}
```

これは基本的には前のテスト用のメインプログラムから重力計算のところを切り出してまとめただけである。

5　プログラム全体の構造

さて、これまで断片的にいろいろな関数を説明してきたが、プログラム全体としてはどんな構造にすればいいだろうか？やらないといけないことは以下のようなものである。

1. ソフトニング、時間刻みなどのパラメータを読み込む。
2. 初期条件を作るなり、ファイルから読み込むなりする。
3. 重力を計算する。
4. 以下のループをシミュレーション時刻の終わりまで繰り返す。
5. 予測子を計算。
6. 重力を計算。
7. 修正子を計算。

8. 出力が必要であればなんか書く。

```
int main()
{
    particle * pp;
    int n;
    cerr << "Enter n:";
    cin >> n ;
    pp = new particle[n];
    double rsize = 1.0;
    create_uniform_sphere(pp, n, 0 , rsize);
    bhnode * bn = NULL;

    int nnodes = n*2+100;
    bn = new bhnode[nnodes];
    real eps2;
    real theta;
    real dt;
    real tend;
    int iout;
    cerr << "Enter eps2, theta, dt, tend, iout:";
    cin >> eps2 >>theta >>dt >>tend >> iout;
    cerr << "eps2=" << eps2 << " theta=" <<theta <<" dt=" << dt
         << " tend=" << tend << " iout=" << iout <<endl;
    calculate_gravity(bn,nnodes,pp,n,eps2, theta);
    cerr << "Initial data"<<endl;
    print_energies(pp,n,0);
    int istep = 0;
    for(real t=0;t<tend; t+=dt){
        integrate(bn, nnodes, pp, n, eps2, theta, dt);
        istep++;
        if (istep % iout == 0){
            cerr << "Time=" << t+dt <<endl;
            print_cm(pp, n);
            print_energies(pp,n,1);
        }
    }
    return 0;
}
```

6 計算精度のチェック

数値計算をした時には、必ず答は合っているかどうかチェックしなければならない。

解析解があるような問題であれば、解析解と比べればいいわけだが、実際に数値計算するのは解析解が求められないからするわけであって普通は解析解と比べることはできない。

微分方程式の初期値問題の場合、チェックは例えば以下のようなことをすることになる。

1. 計算精度を変えた時の解の振舞いを調べる。
2. 保存量があれば、その保存の精度を調べる。
3. 解を視覚化してみて、なにか妙なことが起こっていないかどうか見る。
4. その他。

6.1 計算精度を変えてみる

最初の、計算精度を変えた時の解の振舞いを調べるというのは、例えば今やっている重力多体問題の例なら計算機終了時の全粒子の位置、速度をとっておいて、時間刻み Δt や見込み角 θ を変えた時にどう動くかを見るというものである。原理的には、誤差は Δt や見込み角 θ の何乗かに比例するわけなので、それぞれ 3 種類くらいやってみてその差をとって、それが理屈にあっているかどうか確認する。

ただし、この方法はいつでも使えるというわけではないことに注意する必要がある。あまりうまく使えないのは、元の方程式の解がカオス的なものを長時間数値計算するような場合である。

カオスの定義にはいろいろあるが、基本的な性質の一つは、「非常に近い初期条件から出発した 2 つの系が、時間の指数関数で離れていく」というものである。ところが、大抵のハミルトン力学系で数値計算したいようなものはこの性質をもつ。

特にいま考えているような自己重力多体系の場合、軌道が指数関数的に広がる時間スケールは割合短く、典型的な粒子 (星) の軌道周期の数分の 1 程度であることがわかっている。つまり、例えば 10 周期とか計算すると、最初は非常に近くにあった粒子の位置や速度が、無限に正確に数値積分しても大きくずれてくるということになる。

そんなふうにずれてしまうのであれば数値計算しても意味がある答は得られないのではないかという気がするかもしれない。これはなかなか難しい問題で、現在のところ完全に理解されているとはいいがたい。この話を始めるといくらやっても終わらないのでちょっとこれは棚上げ。

6.2 保存量

保存量は、ハミルトン系ならエネルギーがあるし、また系の重心の速度、系の全角運動量といったものは保存するはずである。従って、こういったものがあまりに大きく変わっていてはよろしくない。問題は「あまりに大きい」というのがどれくらいかだが、これは時と場合による。つまり、数値計算から引き出したい答にエネルギー等の誤差がどれくらい影響しているかによる。これは実際にチェックしなければいけない。「これくらいの誤差ならいいだろう」と思ってなんとなく計算し、論文まで出してしまってから実は計算誤差のせいで結果が間違っていたと判明した例はいろいろある。

というわけでエネルギー等をチェックする関数は以下の通り。といってもここでは誤差等をプリントアウトするだけである。

```
real kinetic_energy(particle * pp,
                    int n)
{
    real ke = 0;
    for(int i = 0;i<n;i++) {
        ke += (pp[i].vel*pp[i].vel)*pp[i].mass*0.5;
    }
    return ke;
}

real potential_energy(particle * pp,
                      int n)
{
    real pe = 0;
    for(int i = 0;i<n;i++) {
```

```
        pe += pp[i].phi*pp[i].mass;
    }
    return pe*0.5;
}

void print_energies(particle * pp,
                    int n,
                    int check)
{
    static real etot0;
    real pe = potential_energy(pp,n);
    real ke = kinetic_energy(pp,n);
    real etot = pe+ke;
    if (check==0 ) etot0 = etot;
    cerr << "etot = " << pe+ke << " PE=" << pe << " KE="<<ke << " V.R.= " << -ke/pe;
    if(check)cerr <<  " de=" << (etot-etot0)/etot0;
    cerr <<endl;
}
```

細かく関数が分かれているが、ここまで分けなくてもいいかもしれない。運動エネルギー、ポテンシャルエネルギーはそれぞれ定義式

$$E_k = \frac{1}{2}\sum_i m_i v^2 \tag{3}$$

$$E_p = \frac{1}{2}\sum_{i\neq j} -G\frac{m_i m_j}{r_{ij}} \tag{4}$$

から計算するだけだが、力の計算の時に各粒子のポテンシャルエネルギー

$$E_{p,i} = \frac{1}{2}\sum_j -G\frac{m_j}{r_{ij}} \tag{5}$$

を力と同様近似的にではあるが計算しているのでこれを合計することにする。そうしないと、折角力の計算は速くなったのにエネルギーのチェックのほうは時間がかかるということになってしまうからである。

もちろん、近似的に計算機した結果、エネルギーが保存しない理由に 2 種類でてきてしまうことに注意する必要がある。つまり、一つは加速度の誤差がつもって粒子の位置が変わるせいだが、それとは別にポテンシャルエネルギー自体の計算誤差もあるわけである。

関数 print_energies の中では、最初に呼ばれた時にはエネルギーをとって置き、その後呼ばれた時にはそのとって置いたエネルギーと比べることで誤差を計算する。

とっておくための変数が

```
    static real etot0;
```

と宣言された変数 etot0 である。ここで static というのは、この講義の 1 回目で説明したことに資料ではなっているが記憶にないのでもう一度説明しておく。

C は再帰呼びだしが可能な言語であり、関数のローカル変数は普通は呼ばれた時に生成される (自動変数)。

static をつけておくと、どこか固定されたアドレスにとられる。このために、呼ばれた後も値が残っていて、関数が次に呼ばれた時もその値をおぼえている。従って、それと新しい値を比べることができる。static がないと関数が呼ばれた時点でメモリ上に場所がとられるので、前に呼ばれた時の値はどっかに消えてしまう。

6.3 絵を書いてみる

直感的になにかおかしいことがないかどうかを見るには、数値解を視覚化してみるのが有効なことが多い。目でみてそれらしいものだったからといって合っているとは限らないが、ひと目みておかしいとわかるということは多いからである。

というわけで、全粒子の投影図をステップ毎に書いて見ることにしよう。例によって PGPLOT を使う。

```
void initgraph()
{

    if(cpgopen("") !=1 ) exit(-1);
    cpgask(0);
}

void plot_particles(particle * p,
                    int n,
                    double xmax,
                    int first)
{
    static float *  xp;
    static float *  yp;
    static int nbuf = -1;
    cpgbbuf();
    if (first == 1){
         cpgpage();
         cpgenv(-xmax, xmax, -xmax, xmax,1,-2);
    }
    if ( nbuf < n){
        if (nbuf > 0){
            delete [] xp;
            delete [] yp;
        }
        xp = new float[n];
        yp = new float[n];
        nbuf = n;
    }
    for(int i = 0;i<n;i++){
        xp[i] = p[i].pos[0];
        yp[i] = p[i].pos[1];
    }
    cpgeras();
    cpgbox("bcnst", 0.0, 0, "bcnst", 0.0, 0);
    cpglab("X", "Y", " ");
    cpgpt(n,xp,yp,-1);
    cpgebuf();
}
```

最初の関数 initgraph はこれも第一回の講義でててきたものと同じで、出力先を選んでことが PGPLOT 自体の初期化をする関数を呼ぶだけである。

次の関数 plot_particles が実際に点々を打つ。といっても本当に点を打っているのは関数 cpgpt で、それ以外は全部そのための準備と後始末である。まず、 cpgpt は点列の座標を 2 つの 1 次元

配列で受け取るので、粒子データからそこにコピーする必要がある。さらにその前にそもそも配列のデータ領域を確保する必要があるので、最初に呼ばれた時か、または n の値が大きくなった時にはメモリを取る。

最初に呼ばれたかどうかの判断は、この配列に関しては static 変数 nbuf の値を見て行う。static 変数の宣言のところで値を設定しておくと、これはプログラムの開始の時に一度だけ値が入る。従って、その値になっていれば最初に呼ばれた時であるということになる。

なお、最初に呼ばれた時に画面サイズの設定もするが、これは別に変数を使うことにしておく。これは、実行中にプロット範囲を変えたいとかいうことに対応できるようにしておくためである。(まあ、このへんはどうでもいいような話ではある)

PGPLOT の実数型の引数は double ではなく float である。引数が配列でない場合にはこれらは区別する必要はないが、配列の場合にはちゃんと float で渡して置く必要がある。配列でない場合には、 C/C++ 言語には float と書いておいても実際には double に勝手に変換されるというよく理由が分からない規定があるので区別しなくていい、というか区別しようがないことになる。

この辺り付け加えたので、もう一度 main 関数を書いておこう。

```
int main()
{
    particle * pp;
    int n;
    cerr << "Enter n:";
    cin >> n ;
    pp = new particle[n];
    double rsize = 1.0;
    cerr << "Enter power index:";
    real power_index;
    cin >> power_index ;
    cerr << "power index = " << power_index <<endl;
    create_uniform_sphere(pp, n, power_index , rsize);
    bhnode * bn = NULL;
    int nnodes = n*2+100;
    bn = new bhnode[nnodes];
    real eps2;
    real theta;
    real dt;
    real tend;
    int iout;
    cerr << "Enter eps2, theta, dt, tend, iout:";
    cin >> eps2 >>theta >>dt >>tend >> iout;
    cerr << "eps2=" << eps2 << " theta=" <<theta <<" dt=" << dt
         << " tend=" << tend << " iout=" << iout <<endl;
    real xmax_plot;
    cerr << "Enter xmax for plot:";
    cin >> xmax_plot;
    cerr << "xmax for plot = " << xmax_plot <<endl;
    initgraph();
    plot_particles(pp,n,xmax_plot,1);
    calculate_gravity(bn,nnodes,pp,n,eps2, theta);
    cerr << "Initial data"<<endl;
    print_energies(pp,n,0);
    int istep = 0;
    for(real t=0;t<tend; t+=dt){
        integrate(bn, nnodes, pp, n, eps2, theta, dt);
```

```
        istep++;
        plot_particles(pp,n,xmax_plot,0);
        if (istep % iout == 0){
            cerr << "Time=" << t+dt <<endl;
            print_cm(pp, n);
            print_energies(pp,n,1);
        }
    }
    return 0;
}
```

段々入力するものが増えて来たので、例えば以下のようなものをファイルで準備しておいて、

```
1000
-0.5
0.2 0.8 0.02 10 25
2
```

リダイレクトを使って実行するほうが間違って変な値を入れることが少ないし手間も減るだろう。ファイル名が samplein なら以下のようにすればいい。

```
# treecode < samplein
```

というわけで今日はこれくらい。

7 練習

1. 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。

2. 上の誤差が理論的にはどうなるべきか考察せよ。

3. 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。

4. 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 treecode.C を動かしてみて、適当な時刻 (5 か 10 くらい) まで計算した時のエネルギーや運動量の誤差がソフトニング、時間刻み、見込み角によってどのように変わるかを調べよ。コンパイルするには

```
g++ -O2  -I/home/makino/utils/pgplot5.2  -o treecode   treecode.C BHtree.C \
            -L/home/makino/utils/pgplot5.2 -lcpgplot -lpgplot \
            -L/usr/X11R6/lib -lX11 -lf2c -lm
```

で (多分) 大丈夫のはず。

5. mk_plummer.C は「プラマーモデル」と言われる銀河や星団の星の分布関数の簡単なモデルに従って粒子を発生させ、標準出力に結果を書くプログラムである。

   ```
   # mk_plummer -n 100 > plummerin
   ```

   とすることで 100 個の粒子の質量、位置、速度を 1 行に 1 粒子づつ書く。なお、最初の 2 行は粒子数と現在時刻 (つねに 0) である。

   プログラム treecode.C をこの形式のファイルをを読み込んで初期条件とするように改造してみよ。

6. プラマーモデルは「力学平衡」なモデルであり、一つ一つの粒子は動くが全体としての粒子分布は時間がたっても変わらない。これでは見ていてつまらないので、読み込んだ後で位置、速度をずらした自分のコピーを作り、2 つがぶつかるような条件から計算するプログラムを作ってみよ。

7. 2 つのプラマーモデルを正面衝突させることを考える。初期に 2 つが重力的に束縛されていれば、この 2 つは最終的には合体する。逆に初期の速度が十分大きければ、正面衝突しても通り抜けてしまってそのまま無限遠にいってしまう。合体が起きるためには、初期の速度が無限遠での速度に換算した時にいくつ以下であればよいか調べてみよ。合体するかどうかどうやって判定すればいいか？また、答は粒子数や計算精度によってどんなふうに変わるか？

8 レポート課題

今回の話は、まあちょっと難しいかなという気もするのでレポートは必須というわけではないことにする。で、その代わり、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してほしい。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。

レポート (これまでにでたものを含めて) の〆切は

4 年生・他学科等 2/9

天文学科 3 年生 2/29

とする。