計算天文学 II 第 12 回 データ構造とアルゴリズム (3)

牧野淳一郎

2002 年 1 月 28 日

1 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。

1.1 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。

1.2 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

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

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

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

$$E_{p,i} = \frac{1}{2}\sum_j -G\frac{m_j}{r_{ij}} \tag{3}$$

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。

1.3 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で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。

2 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 `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。
2. `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。
3. 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。
4. 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 ?

3 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 $2/28$ とする。