# 天体物理学実習 II　6 月 21 日のレポート課題

遠山翔太

2022/6/27

完成させた N 体問題のコードは以下のようになった。加速度の計算コストを落とすためにニュートンの運動の第 3 法則、つまり作用・反作用の法則を利用し、i 粒子の加速度を求めると同時に j 粒子の加速度も計算を勧めている。

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


// 粒子構造体の定義：ID、質量、座標、速度、加速度、半径（解析用）
typedef struct particle {
  double m, x, y, z, vx, vy, vz, ax, ay, az, r;
} particle_t;



// Box Muller 法で平均 0 分散 1 の正規分布乱数を生成する
double bmrand(void) {
  double x = (double)rand()/RAND_MAX;
  double y = (double)rand()/RAND_MAX;
  // 本来２つ乱数が生成されるが、片方は捨てる
  return sqrt(-2.0*log(x))*cos(2.0*M_PI*y);
}


// 全粒子の合計の運動エネルギーを計算する
double total_ekinetic(particle_t *data, int n) {
  double ekin = 0.0;
  for (int i = 0; i < n; ++i) {
    ekin += 0.5*data[i].m*(data[i].vx*data[i].vx
```

```
        +data[i].vy*data[i].vy+data[i].vz*data[i].vz);
  }
  return ekin;
}


// 全粒子の合計のポテンシャルエネルギーを計算する
double total_potential(particle_t *data, int n, double eps) {
  double pot = 0.0;
  // 全ての i-j 粒子の組についてポテンシャルを積算する
  // 重複しないよう、ループの上限下限に注意する
  for (int i = 0; i < n-1; ++i) {
    for (int j = i+1; j < n; ++j) {
      double dx = data[j].x - data[i].x;
      double dy = data[j].y - data[i].y;
      double dz = data[j].z - data[i].z;
      double r = sqrt(dx*dx + dy*dy + dz*dz + eps*eps);
      pot -= data[i].m*data[j].m/r;
    }
  }
  return pot;
}


// 初期の粒子の空間分布を計算する関数
// rad は初期の一様球の半径（今回の例では 1.0 にしてある）
void init_position(particle_t *data, int n, double rad) {
  for (int i = 0; i < n; ++i) {
    // data[i].x,y,z が一様球になるように計算する
    // 一様乱数を生成して、それが指定された半径の球内にある場合は採用、
    // 球の外にある場合は廃棄するという手順を繰り返して作ると良い。
    double x; // -rad に 0 以上 2rad 以下の乱数を足して、-rad から rad までの一様乱数を与える。
    double y;
    double z;
    double r;
    do{
    x = -1.0*rad + (double)rand()*2.0*rad/((double)RAND_MAX); // -rad に 0 以上 2rad 以下
の乱数を足して、-rad から rad までの一様乱数を与える。
    y = -1.0*rad + (double)rand()*2.0*rad/((double)RAND_MAX);
    z = -1.0*rad + (double)rand()*2.0*rad/((double)RAND_MAX);
```

```
    r = sqrt(x*x+y*y+z*z);
    data[i].x = x, data[i].y = y,data[i].z = z;
    }while( r >= rad ); // r >= rad の場合はもう１度 do から上書き

  }
  return;
}


// 初期の粒子の速度を計算する関数
// rv はビリアル比（パラメータ)、pot は全ポテンシャルエネルギー
void init_velocity(particle_t *data, int n, double rv, double pot) {
  for (int i = 0; i < n; ++i) {
    // data[i].vx,vy,vz が平均 0、分散 σ の正規分布になるように計算する
    // 分散を σ にするには分散 1 の正規分布乱数を σ 倍すればよい。

    double sigma = sqrt(2.0*rv*fabs(pot)/3.0); //全質量 M = 1 の単位系にしていることに注意。

    data[i].vx = sigma*bmrand();
    data[i].vy = sigma*bmrand();
    data[i].vz = sigma*bmrand();

  }
  return;
}


// 粒子分布を作成する関数
void init_particles(particle_t *data, int n, double eps,
                    double rad, double rv) {
  for (int i = 0; i < n; ++i)
    data[i].m = 1.0/(double)n; // 質量をセット：全質量 M=1
  init_position(data, n, rad);
  double pot = total_potential(data, n, eps);
  init_velocity(data, n, rv, pot);
  return;
}


void calc_acceleration(particle_t *data, int n, double eps) {
```

```
  // 加速度を全てゼロでクリア
  for (int i = 0; i < n; ++i) {
    data[i].ax = 0.0;
    data[i].ay = 0.0;
    data[i].az = 0.0;
  }
  for (int i = 0; i < n; ++i) {  // 全ての i 粒子について加速度を計算する
    for (int j = i; j < n; ++j) { // j 粒子について和を取る
      // 加速度 data[i].ax,ay,az を計算する

double dx = 0.0, dy = 0.0, dz = 0.0;
dx = data[i].x -data[j].x;
dy = data[i].y -data[j].y;
dz = data[i].z -data[j].z;
double b1 =sqrt(dx*dx+dy*dy+dz*dz+eps*eps);
double b3 = b1*b1*b1;

data[i].ax -= data[j].m*dx/b3; // G=1 にしている。
data[i].ay -= data[j].m*dy/b3;
data[i].az -= data[j].m*dz/b3;

        data[j].ax += data[i].m*dx/b3;
data[j].ay += data[i].m*dy/b3;
data[j].az += data[i].m*dz/b3;

    }
  }
  return;
}



// Leapfrog 法
void leapfrog(particle_t *data, int n, double dt, double eps) {
  // t=tn での加速度は前のステップで計算されたものが ax,ay,az に入っている
  // 最初のステップの前に calc_acceleration を実行しておく必要がある
  for (int i = 0; i < n; ++i) {
    // 各粒子の速度を dt/2 分進める

    data[i].vx += 0.5*dt*data[i].ax;
    data[i].vy += 0.5*dt*data[i].ay;
```

```
    data[i].vz += 0.5*dt*data[i].az;

  }
  for (int i = 0; i < n; ++i) {
    // 各粒子の位置を dt 分進める

    data[i].x += dt*data[i].vx;
    data[i].y += dt*data[i].vy;
    data[i].z += dt*data[i].vz;



  }
  calc_acceleration(data, n, eps); // t=tn+1 での加速度を計算
  for (int i = 0; i < n; ++i) {
    // 各粒子の速度を dt/2 分進める
    // 一つ目のループと同じ形になるはず

    data[i].vx += 0.5*dt*data[i].ax;
    data[i].vy += 0.5*dt*data[i].ay;
    data[i].vz += 0.5*dt*data[i].az;

  }
  return;
}


// データをファイルに出力する
void output(particle_t *data, int n, int step) {
  char fn[32];
  sprintf(fn, "output%04d.dat", step);
  FILE *fp;
  fp = fopen(fn, "w");
  if (fp == NULL) {
    printf("Error: cannot open file %s\n", fn);
    exit(1); // プログラムを終了させる関数（return とは違う）
  }

  for (int i = 0; i < n; ++i) {
    fprintf(fp, "%d %g %g %g %g %g %g %g\n", i,
            data[i].m, data[i].x, data[i].y, data[i].z,
```

```
            data[i].vx, data[i].vy, data[i].vz);
  }

  fclose(fp);
  return;
}


// 中心からの半径 data[i].r を計算する
void calc_radii(particle_t *data, int n) {
  for (int i = 0; i < n; ++i)
    data[i].r = sqrt(data[i].x*data[i].x
              + data[i].y*data[i].y + data[i].z*data[i].z);
  return;
}


// qsort 用の比較関数：中心からの半径で並び替える
int cmp_data(const void *a, const void *b) {
  if (((particle_t*)a)->r < ((particle_t*)b)->r)
    return -1;
  else if (((particle_t*)a)->r > ((particle_t*)b)->r)
    return 1;
  else
    return 0;
}


// 粒子の密度の動径分布を計算する
void radial_profile(particle_t *data, int n, int step) {
  char fn[32];
  sprintf(fn, "radprof%04d.dat", step);
  FILE *fp;
  fp = fopen(fn, "w");
  if (fp == NULL) {
    printf("Error: cannot open file %s\n", fn);
    exit(1); // プログラムを終了させる関数（return とは違う）
  }

  calc_radii(data, n); // 中心からの半径を計算
```

```
  // データを中心からの半径で並び替える
  // これは計算に使われているデータも並び替えてしまうが、
  // 今の場合粒子に個性はないので並び替えても問題ない
  qsort(data, n, sizeof(particle_t), cmp_data);

  // 密度プロファイルを計算して表示する
  int d = 64;
  for (int i = 0; i < n-d; ++i) {
    // i 番目から i+d 番目の粒子を含む球殻を考え、
    // 球殻の半径を i 番目の粒子の半径と i+d 番目の粒子の半径の平均、
    // 球殻の密度を球殻中の粒子の質量/球殻の体積で計算する
    double radius, density;

    radius = (data[i].r + data[i+d].r)/2.0;
    double Mshell = data[i].m*d;
    double Vshell = 4.0*M_PI*(data[i+d].r*data[i+d].r*data[i+d].r-data[i].r*data[i].r*data[i].r)/3.0
    density = Mshell / Vshell;

    fprintf(fp, "%g %g\n", radius, density);
  }
  fclose(fp);
  return;
}


// データを解析して全エネルギー及びビリアル比を表示
// ポテンシャルの計算は O(N^2) なのでそれなりにコストがかかることに注意
void analyze(particle_t *data, int n, double eps, double t) {
  double ekin = total_ekinetic(data, n);
  double pot  = total_potential(data, n, eps);
  double etot = ekin + pot;
  double rv = ekin / fabs(pot);
  printf("time = %f, Etot = %.16f, Rvir = %g\n", t, etot, rv);
  return;
}


int main(void) {
  // 定数類の宣言
  const int N = 4096;       // 粒子数 - 時間がかかり過ぎる場合は減らして良い
```

```
  const int NSTEP = 1024;  // ステップ数
  const int OSTEP = 64;    // データを出力する間隔
  const double eps = 0.01; // ソフトニングパラメータ
  const double rad = 1.0;  // 初期分布である一様球の半径
  const double rv = 0.1;   // ビリアルパラメータ
  const double tend = 5.0; // 終了時間

  // 粒子・加速度データの宣言と確保
  particle_t *data;
  data = malloc(sizeof(particle_t)*N);

  srand(1); // 乱数列の初期化 これは main で一回のみ行う。何回も行うとそのたびに乱数が最初に戻ってしまう。

  init_particles(data, N, eps, rad, rv); // 粒子分布を作成

  double t = 0.0, dt = tend / NSTEP;
  analyze(data, N, eps, t); // 初期状態を確認する
  radial_profile(data, N, 0); // 初期密度プロファイルも出力

  // 最初のステップを始める前に加速度を計算しておく
  // leapfrog() 内のコメントも参照のこと
  calc_acceleration(data, N, eps);

  for (int i = 0; i < NSTEP; ++i) {
    if ((i % OSTEP) == 0) { // OSTEP 毎にデータを出力
      output(data, N, i/OSTEP);
      radial_profile(data, N, i/OSTEP);
    }

    leapfrog(data, N, dt, eps); // Leapfrog 法で dt 時間を進める
    t += dt;

    analyze(data, N, eps, t); // エネルギー等を解析する
  }

  output(data, N, NSTEP/OSTEP); // 最終状態の出力
  radial_profile(data, N, NSTEP/OSTEP); // 最終密度プロファイルを出力

  // 終了処理：粒子データを解放
```

```
  free(data);

  return 0;
}
```

ビリアル比の時間進化のグラフを作成するために、analyze の出力を以下のように変更した。

```
void analyze(particle_t *data, int n, double eps, double t) {
  double ekin = total_ekinetic(data, n);
  double pot  = total_potential(data, n, eps);
  double etot = ekin + pot;
  double rv = ekin / fabs(pot);
  printf("%f %g\n", t,rv); //時間とビリアル比のみをスペースで区切って表示させる。
  return;
}
```

得られたデータを横軸を時間、縦軸をビリアル比としてプロットしたものが図 1 である。初期値の 0.1 から 0.8 付近まで上昇し、振動しつつ約 0.55 まで下降している。ビリアル比は平衡状態で 0.5 で、大きいほどポテンシャルの影響よりも運動エネルギーの影響のほうが大きいことになるが、初期状態の 0.1 では重力の影響のほう運動の影響よりも大きいことを表しており、重力で中心に落ちていった結果、中心への引力あるいは粒子間のスイングバイを利用してポテンシャルエネルギーを運動エネルギーに変換したためにビリアル比が増大したと考えられ、その後、調和振動子の減衰振動ように再び復元力で運動エネルギーを失い、次第に平衡状態に向かっていく様子が見られる。

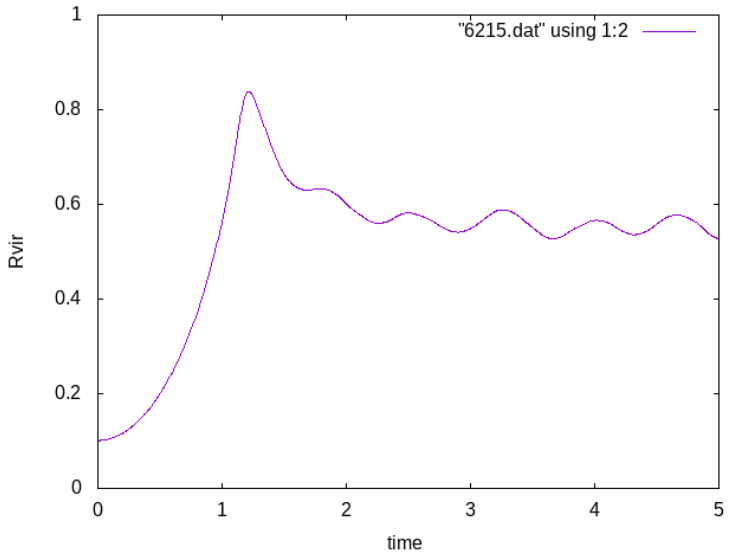


図 1

初期状態と最終状態における粒子分布を 3 次元直行座標にプロットしたグラフが図 2、3 である。図 3 では図 2 における粒子分布の中心付近に粒子が集中している半面、集団に属していない粒子も点在している。この様子は球状星団に似ているように思われた。いくつかの粒子が中心に降着していないのは、中心方向への引力よりも外向きへの加速度が上回っているからである。

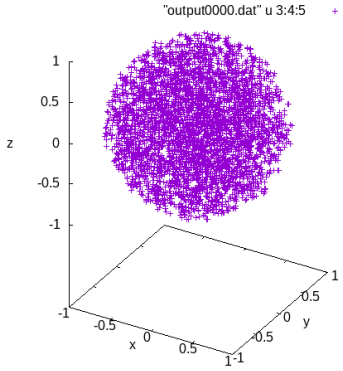


図 2　初期状態

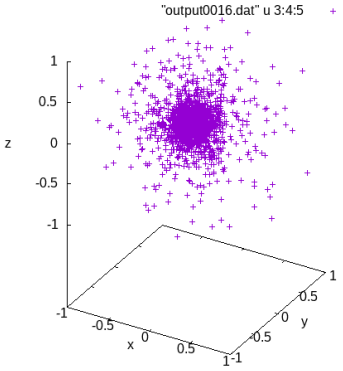


図 3　最終状態

また、初期状態と最終状態における中心からの距離と密度の関係をグラフで表したものが図 4、5 である。ただし、横軸を半径に、縦軸を密度に撮っているところは同じだが、軸の範囲は異なる。

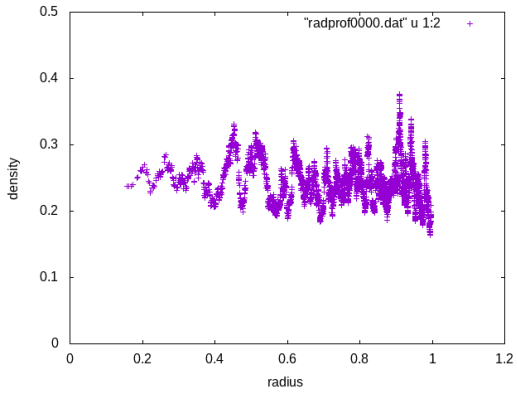


図 4　初期状態

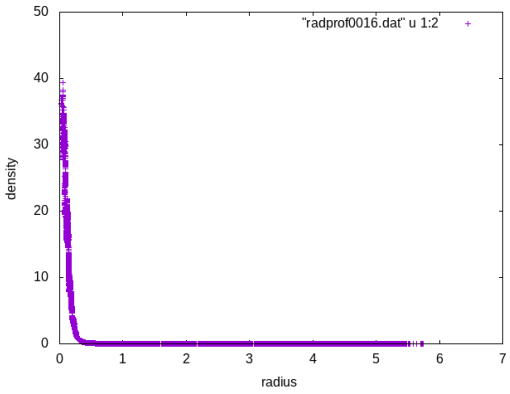


図 5　最終状態

図 4、5 を見ると、中心に粒子が集中していることがよりわかりやすい。加えて、緩和したときの密度と半径が $\rho \propto r^{-4}$ になっているかを確認するために、gnuplot に

```
stats "radprof????.dat" u (log($1)):(log($2))
```

と????にファイルの番号を入力して動かし、両対数グラフの傾きを得ると、下の表 1 のようになった。ただし、0015 までは STEP = 1024 かつ tend = 5.0 で実行したものであるが、0016 以降は STEP = 4096 かつ tend = 20.0 で実行したものである。

表 1

| ファイルの番号 | 両対数グラフの傾き |
|---|---|
| 0000 | -0.06832 |
| 0001 | -0.6414 |
| 0002 | -1.427 |
| 0003 | -2.398 |
| 0004 | -3.215 |
| 0005 | -2.481 |
| 0006 | -2.76 |
| 0007 | -2.819 |
| 0008 | -2.871 |
| 0010 | -2.946 |
| 0011 | -2.966 |
| 0012 | -2.962 |
| 00013 | -2.991 |
| 0014 | -3.006 |
| 0015 | -3.029 |
| 0016 | -3.015 |
| 0024 | -3.077 |
| 0032 | -3.097 |
| 0040 | -3.105 |
| 0048 | -3.126 |
| 0056 | -3.133 |
| 0056 | -3.139 |

次第に増加して入るが、おおよそ-3 という結果になったが、なぜ-4 にならないのかはわからなかった。ソフトニングパラメータの影響ではないかと推測する。