

The N -Body problem

Consider a set of N particles (bodies) which interact with one another by long range forces which decrease inversely as the square of the distance. The gravitational interaction between masses and the electromagnetic interaction between charges are examples of long range forces.

This N -body problem is one of the oldest problems in physics. After Newton solved the 2-body problem exactly, numerous scientists and mathematicians attempted to find exact solutions to the 3-body problem. So far, no non-trivial exact solutions to the 3-body problem have been found. Mathematicians have been able to prove that the problem is *non-integrable*, and that many 3-body trajectories are *chaotic* and cannot be computed numerically.

Why are $1/r^2$ forces called long range? Suppose a very large number of particles are distributed roughly uniformly over a large region of space. The area of a sphere of radius r surrounding any particle increases like $4\pi r^2$. Thus the number of particles at distance r times the strength of the force exerted by each of these particles is roughly independent of r . Thus *all* of the particles in the system influence the motion of any one particle.

Because the forces are long range, it is not possible to solve for the dynamics of any particle by considering a local neighborhood, as was done in speeding up the Lennard-Jones MD simulation by introducing a cut-off radius beyond which the force could be neglected.

Numerical solution of the N -body problem

Several approaches to speeding up N -body codes have been developed. The following are some examples of widely used methods:

- **Particle-mesh methods:** These are based on introducing a uniform cubic lattice of points in space. The effect of each particle in a cube of the lattice is approximated by variables located at the neighboring grid points. This grid of variables is used to compute the potential which determines the force on the particles. These methods have been very successful in solving problems in which the particles are spread out roughly uniformly over all of space. These methods typically scale like $\mathcal{O}(M \log M)$ where M is the number of grid points.
- **Tree-code methods:** These methods were developed to simulate astrophysical systems such as the motion of

stars in a galaxy. The distribution of stars is generally highly non-uniform: particle-mesh methods do not work well for non-uniform distributions. Tree-code methods partition space *hierarchically* into a tree like structure of cubic regions: where there are few particles, the tree contains only a few large cubes; but in regions with many particles, the cubes are repeatedly sub-divided into smaller cubes. Tree-code methods typically scale like $\mathcal{O}(N \log N)$.

- **Fast multipole methods:** These methods use *multipole expansions* of the long range potential. A finite number of terms in the expansions are retained, and the algorithm then allows an accurate approximate determination of the forces which scales like $\mathcal{O}(N)$. The Fast Multipole Algorithm was named one of the top ten algorithms of the twentieth century by the magazine *Computers in Science and Engineering*.

Applications of N-body methods

There are many areas in which N-body methods are used, for example:

- **Astrophysics:** The dynamical evolution of stars within galaxies, and collisions between galaxies involve thousands of bodies all interacting through long range forces. Many of the N-body algorithms were developed to solve astrophysical problems.
- **Plasma physics:** In a plasma, atoms are ionized into electrons and positively charged ions which interact through long range Coulomb forces. Particle mesh methods were developed to solve the dynamics of uniform plasmas.
- **Molecular dynamics:** Interactions between rare gas atoms such as Argon fall off like $1/r^6$. However, in materials made up of polar molecules (which have a permanent dipole moment), the Coulomb field falls off like $1/r^3$: while this is not strictly a long range force, the number of interacting particles which need to be taken into account can be very large, and N-body methods then become very useful. Such methods can also be useful in studying an electron gas confined in solid state devices.
- **Fluid dynamics:** It can be shown that elliptic partial differential equations with Dirichlet boundary conditions can be solved using N-body techniques. In this approach, the solution is determined by a finite number of discrete sources on the boundary of the region.

Hut-Makino starter code for N -body simulations

Astrophysics is an interesting area in which to learn about N -body methods. Piet Hut and Jun Makino have a website with a nice tutorial program for doing N -body simulations. It is good to learn how to locate public domain programs, to understand how they work, and to use them to solve computational problems. One can learn many useful programming techniques by studying good code written by others.

The key functions of the code are explained briefly below: the full code is explained in a PostScript file on Hut's website.

The main function

The starter code solves Newton's equations of motion for N bodies, each of which can have a different mass, given initial positions and velocities. The bodies interact through Newton's inverse square law of gravity.

```
typedef double  real;                // "real" as a general name for the
                                     // standard floating-point data type
const int NDIM = 3;                 // number of spatial dimensions

int main(int argc, char *argv[])
{
    real  dt_param = 0.03;           // control parameter to determine time step size
    real  dt_dia = 1;                // time interval between diagnostics output
    real  dt_out = 1;                // time interval between output of snapshots
    real  dt_tot = 10;               // duration of the integration
    bool  init_out = false;           // if true:  snapshot output with start at t = 0
                                     //          with an echo of the input snapshot
    bool  x_flag = false;             // if true:  extra debugging diagnostics output

    if (! read_options(argc, argv, dt_param, dt_dia, dt_out, dt_tot, init_out,
                       x_flag))
```

```
        return 1;                // halt criterion detected by read_options()

int n;                          // N, number of particles in the N-body system
cin >> n;

real t;                        // time
cin >> t;

real * mass = new real[n];      // masses for all particles
real (* pos)[NDIM] = new real[n][NDIM]; // positions for all particles
real (* vel)[NDIM] = new real[n][NDIM]; // velocities for all particles

get_snapshot(mass, pos, vel, n);

evolve(mass, pos, vel, n, t, dt_param, dt_dia, dt_out, dt_tot, init_out,
        x_flag);

delete[] mass;
delete[] pos;
delete[] vel;
}
```

The `get_snapshot` function reads the values of the masses of the bodies and the initial positions and velocities. The `evolve` function integrates Newton's equations of motion for the desired number of time steps.

Fourth-order Hermite integration algorithm

The program uses an integration algorithm that the authors have found works well for various astrophysical problems.

First some definitions: The vector distance between particles i and j is defined to be

$$\mathbf{r}_{ji} = \mathbf{r}_j - \mathbf{r}_i ,$$

and the relative velocity of the two particles is

$$\mathbf{v}_{ji} = \mathbf{v}_j - \mathbf{v}_i ,$$

According to Newton's law of gravity, the acceleration of particle i due to particle j is

$$\mathbf{a}_{ji} = \frac{M_j}{r_{ji}^3} \mathbf{r}_{ji} .$$

Here we use units such that Newton's constant $G = 1$. Note that the acceleration is directed towards particle j : gravity is an attractive force. The algorithm also makes use of the rate of change of the acceleration, which is called the “jerk”

$$\mathbf{j}_{ji} = \frac{M_j}{r_{ji}^3} \left[\mathbf{v}_{ji} - 3 \frac{\mathbf{v}_{ji} \cdot \mathbf{r}_{ji}}{r_{ji}^2} \mathbf{r}_{ji} \right] .$$

The acceleration and jerk of particle i are then given by

$$\mathbf{a}_i = \sum_{j \neq i} \mathbf{a}_{ji} , \quad \mathbf{j}_i = \sum_{j \neq i} \mathbf{j}_{ji} .$$

The Hermite algorithm used by Hut and Makino is a type of “predictor-corrector” algorithm. During a time step of size δt , the next position and velocity of the particle are “predicted” using the known acceleration and jerk:

$$\begin{aligned} \mathbf{r}_p &= \mathbf{r} + \mathbf{v}\delta t + \frac{1}{2}\mathbf{a}\delta t^2 + \frac{1}{6}\mathbf{j}\delta t^3 \\ \mathbf{v}_p &= \mathbf{v} + \mathbf{a}\delta t + \frac{1}{2}\mathbf{j}\delta t^2 \end{aligned}$$

These predicted positions and velocities are used to compute the predicted accelerations $\mathbf{a}_p$ and jerks $\mathbf{j}_p$. By making Taylor series expansions of the various formulas, it can be shown that the next two derivatives of the acceleration are given by

$$\mathbf{k} \equiv \frac{1}{2} \mathbf{a}'' \delta t^2 = 2(\mathbf{a} - \mathbf{a}_p) + \delta t(\mathbf{j} - \mathbf{j}_p)$$

$$\mathbf{l} \equiv \frac{1}{2} \mathbf{a}''' \delta t^2 = -3(\mathbf{a} - \mathbf{a}_p) - \delta t(2\mathbf{j} + \mathbf{j}_p)$$

This information is then used to get the “corrected” positions and velocities at the next time step:

$$\mathbf{r}_c = \mathbf{r}_p + \left(\frac{1}{12} \mathbf{k} + \frac{1}{20} \mathbf{l} \right) \delta t^2$$

$$\mathbf{v}_c = \mathbf{v}_p + \left(\frac{1}{3} \mathbf{k} + \frac{1}{4} \mathbf{l} \right) \delta t$$

This algorithm is implemented in the program as follows:

Taking a single time step δt

```
void evolve_step(const real mass[], real pos[][NDIM], real vel[][NDIM],
                real acc[][NDIM], real jerk[][NDIM], int n, real & t,
                real dt, real & epot, real & coll_time)
{
    real (* old_pos)[NDIM] = new real[n][NDIM];
    real (* old_vel)[NDIM] = new real[n][NDIM];
    real (* old_acc)[NDIM] = new real[n][NDIM];
    real (* old_jerk)[NDIM] = new real[n][NDIM];

    for (int i = 0; i < n ; i++)
```

```
    for (int k = 0; k < NDIM ; k++){
        old_pos[i][k] = pos[i][k];
        old_vel[i][k] = vel[i][k];
        old_acc[i][k] = acc[i][k];
        old_jerk[i][k] = jerk[i][k];
    }

    predict_step(pos, vel, acc, jerk, n, dt);
    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);
    correct_step(pos, vel, acc, jerk, old_pos, old_vel, old_acc, old_jerk,
                n, dt);
    t += dt;

    delete[] old_pos;
    delete[] old_vel;
    delete[] old_acc;
    delete[] old_jerk;
}
```

Taking the predictor step

```
void predict_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  int n, real dt)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
```

```

        pos[i][k] += vel[i][k]*dt + acc[i][k]*dt*dt/2
                      + jerk[i][k]*dt*dt*dt/6;
        vel[i][k] += acc[i][k]*dt + jerk[i][k]*dt*dt/2;
    }
}

```

Computing the accelerations and jerks

This is similar to the `computeAccelerations` function in the MD programs. It is the most time consuming part of the computation because one must examine all $N(N-1)/2$ pairs of particles. The following code also computes the total potential energy of the system

$$U = - \sum_{\text{pairs}} \frac{M_i M_j}{r_{ji}},$$

as well as two estimates of the “least collision time”, which are used to adjust the time step δt as the computation proceeds.

```

void get_acc_jerk_pot_coll(const real mass[], const real pos[][NDIM],
                          const real vel[][NDIM], real acc[][NDIM],
                          real jerk[][NDIM], int n, real & epot,
                          real & coll_time)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++)
            acc[i][k] = jerk[i][k] = 0;
    epot = 0;
    const real VERY_LARGE_NUMBER = 1e300;
    real coll_time_q = VERY_LARGE_NUMBER;    // collision time to 4th power
}

```

```

real coll_est_q;                                // collision time scale estimate
                                                // to 4th power (quartic)

for (int i = 0; i < n ; i++){
    for (int j = i+1; j < n ; j++){              // rji[] is the vector from
                                                // particle i to particle j
        real rji[NDIM];                        // vji[] = d rji[] / d t
        real vji[NDIM];
        for (int k = 0; k < NDIM ; k++){
            rji[k] = pos[j][k] - pos[i][k];
            vji[k] = vel[j][k] - vel[i][k];
        }
        real r2 = 0;                            // | rji |^2
        real v2 = 0;                            // | vji |^2
        real rv_r2 = 0;                        // ( rij .  vij ) / | rji |^2
        for (int k = 0; k < NDIM ; k++){
            r2 += rji[k] * rji[k];
            v2 += vji[k] * vji[k];
            rv_r2 += rji[k] * vji[k];
        }
        rv_r2 /= r2;
        real r = sqrt(r2);                      // | rji |
        real r3 = r * r2;                      // | rji |^3

// add the {i,j} contribution to the total potential energy for the system:

        epot -= mass[i] * mass[j] / r;

// add the {j (i)} contribution to the {i (j)} values of acceleration and jerk:

        real da[3];                            // main terms in pairwise

```

```
real dj[3];                                // acceleration and jerk
for (int k = 0; k < NDIM ; k++){
    da[k] = rji[k] / r3;                    // see equations
    dj[k] = (vji[k] - 3 * rv_r2 * rji[k]) / r3; // in the header
}
for (int k = 0; k < NDIM ; k++){
    acc[i][k] += mass[j] * da[k];           // using symmetry
    acc[j][k] -= mass[i] * da[k];           // find pairwise
    jerk[i][k] += mass[j] * dj[k];          // acceleration
    jerk[j][k] -= mass[i] * dj[k];          // and jerk
}

// first collision time estimate, based on unaccelerated linear motion:

coll_est_q = (r2*r2) / (v2*v2);
if (coll_time_q > coll_est_q)
    coll_time_q = coll_est_q;

// second collision time estimate, based on free fall:

real da2 = 0;                             // da2 becomes the
for (int k = 0; k < NDIM ; k++)            // square of the
    da2 += da[k] * da[k];                  // pair-wise accel-
double mij = mass[i] + mass[j];            // eration between
da2 *= mij * mij;                          // particles i and j

coll_est_q = r2/da2;
if (coll_time_q > coll_est_q)
    coll_time_q = coll_est_q;
```

```

    }
}
coll_time = sqrt(sqrt(coll_time_q));
}
// from q for quartic back
// to linear collision time

```

Taking the corrector step

```

void correct_step(real pos[][NDIM], real vel[][NDIM],
                 const real acc[][NDIM], const real jerk[][NDIM],
                 const real old_pos[][NDIM], const real old_vel[][NDIM],
                 const real old_acc[][NDIM], const real old_jerk[][NDIM],
                 int n, real dt)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
            vel[i][k] = old_vel[i][k] + (old_acc[i][k] + acc[i][k])*dt/2
                               + (old_jerk[i][k] - jerk[i][k])*dt*dt/12;
            pos[i][k] = old_pos[i][k] + (old_vel[i][k] + vel[i][k])*dt/2
                               + (old_acc[i][k] - acc[i][k])*dt*dt/12;
        }
}

```

Steering the calculation

The `evolve` function steers the calculation using the functions described above.

```

void evolve(const real mass[], real pos[][NDIM], real vel[][NDIM],

```

```
    int n, real & t, real dt_param, real dt_dia, real dt_out,
    real dt_tot, bool init_out, bool x_flag)
{
    cerr << "Starting a Hermite integration for a " << n
    << "-body system,\n  from time t = " << t
    << " with time step control parameter dt_param = " << dt_param
    << "  until time " << t + dt_tot
    << " ,\n  with diagnostics output interval dt_dia = "
    << dt_dia << ",\n  and snapshot output interval dt_out = "
    << dt_out << "." << endl;

    real (* acc)[NDIM] = new real[n][NDIM];          // accelerations and jerks
    real (* jerk)[NDIM] = new real[n][NDIM];          // for all particles
    real epot;                                         // potential energy of the n-body system
    real coll_time;                                   // collision (close encounter) time scale

    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);

    int nsteps = 0;                                  // number of integration time steps completed
    real einit;                                       // initial total energy of the system

    write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps, einit,
                      true, x_flag);

    if (init_out)                                     // flag for initial output
        put_snapshot(mass, pos, vel, n, t);

    real t_dia = t + dt_dia;                          // next time for diagnostics output
    real t_out = t + dt_out;                          // next time for snapshot output
    real t_end = t + dt_tot;                          // final time, to finish the integration
```

```
while (true){
    while (t < t_dia && t < t_out && t < t_end){
        real dt = dt_param * coll_time;
        evolve_step(mass, pos, vel, acc, jerk, n, t, dt, epot, coll_time);
        nsteps++;
    }
    if (t >= t_dia){
        write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps,
                        einit, false, x_flag);
        t_dia += dt_dia;
    }
    if (t >= t_out){
        put_snapshot(mass, pos, vel, n, t);
        t_out += dt_out;
    }
    if (t >= t_end)
        break;
}

delete[] acc;
delete[] jerk;
}
```

The remaining functions in the program are:

- `get_snapshot` reads an initial configuration of particles
- `put_snapshot` write the current configuration of particles

- `read_options` parses parameters specified by the user on the command line
- `write_diagnostics` writes more detailed output including the total and potential energies, etc.