N-body assignment

David Bindel

2/23/2010

Contents

1 Introduction

For the second assignment, you will be timing and tuning serial and parallel particle simulation codes in C. The parallel versions use both OpenMP and MPI. As written, these codes are not particularly high performance. Your job is three-fold:

- Characterize the performance of all three basic codes as a function of the number of particles in the system and the number of processors. Simple models are nice here, but so are empirical measurements.

- Improve the complexity by changing from the naive $O(n^2)$ force evaluation algorithm to an algorithm based on spatial partitioning. The latter algorithm should require roughly $O(n)$ work, allowing you to scale to much larger numbers of particles. You will want to modify the parallel algorithms to use the same spatial decomposition — and you will want to try to do something a little more clever to keep communication from dominating computation in the overall cost of your program!
- Improve or extend the code in some other way that appeals to you. This could be by doing something more clever with the time integrator (which is currently a basic leapfrog scheme); by thinking about how to balance the work dynamically; or even by doing some performance tuning on the serial implementation. Feel free to suggest your own ideas as well!

In what follows, I spell out some of the basic numerical components used by the base codes, describe how the serial, OpenMP, and MPI drivers work (and how they differ from each other), and note a few things about option processing and binary I/O in a Unix environment.

As always, more information may be found on the web page or the wiki.

2 System parameters

The `sim_param_t` structure holds the parameters that describe the simulation. These parameters are filled in by the `get_params` function (described later).

```
typedef struct sim_param_t {
    char* fname;   /* File name (run.out)        */
    int   npart;   /* Number of particles (500)  */
    int   nframes; /* Number of frames (200)     */
    int   npframe; /* Steps per frame (100)      */
    float dt;      /* Time step (1e-4)           */
    float eps_lj;  /* Strength for L-J (1)       */
    float sig_lj;  /* Radius for L-J   (1e-2)    */
    float G;       /* Gravitational strength (1) */
    float T0;      /* Initial temperature (1)    */
} sim_param_t;

int get_params(int argc, char** argv, sim_param_t* params);
```

3 Lennard-Jones

The Lennard-Jones potential has the form

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^6 - \left(\frac{\sigma}{r}\right)^{12} \right] = 4\epsilon \left(\frac{\sigma}{r}\right)^6 \left[1 - \left(\frac{\sigma}{r}\right)^6 \right].$$

We will also truncate the potential at $r_c = 2.5\sigma$. For our simulation, we use the following parameters for the potential:

```
#define EPS   1
#define SIG   1e-2
#define RCUT  (2.5*SIG)
```

The Lennard-Jones force on a particle from interaction with another particle at distance $\Delta\mathbf{x} = (\Delta x, \Delta y)$ is given by the negative gradient of the potential, i.e.

$$\mathbf{F} = -\nabla V_{LJ}(\|\mathbf{x}\|) = -\frac{dV_{LJ}}{dr}\nabla r = -\frac{dV_{LJ}}{dr}\frac{\mathbf{x}}{r} = C_{LJ}(r)\mathbf{x}$$

The scalar expression

$$C_{LJ}(r) = \frac{1}{r}\frac{dV_{LJ}}{dr} = \frac{24\epsilon}{r^2}\left(\frac{\sigma}{r}\right)^6\left[1 - 2\left(\frac{\sigma}{r}\right)^6\right]$$

is particularly convenient because, like the Lennard-Jones potential itself, it will only depend on even powers of r. Thus, there is no need to compute a square root during the computation of $C_{LJ}(r)$:

```
float compute_LJ_scalar(float r2, float eps, float sig2)
{
    if (r2 < 6.25*sig2) { /* r_cutoff = 2.5 sigma */
        float z = sig2/r2;
        float u = z*z*z;
        return 24*eps/r2 * u*(1-2*u);
    }
    return 0;
}
```

In order to compute the total energy for monitoring purposes, we also want the potential itself:

```
float potential_LJ(float r2, float eps, float sig2)
{
    float z = sig2/r2;
    float u = z*z*z;
    return 4*eps*u*(1-u);
}
```

4 Leapfrog integration

The leapfrog (or velocity Verlet) method takes two steps to get from t to $t+\Delta t$. In the first step, we compute estimates of $x(t+\Delta t)$ and $v(t+\Delta t/2)$ based on Taylor expansions using $x(t)$, $v(t)$, and $a(t)$:

$$v(t+\Delta t/2) = v(t) + \frac{1}{2}a(t)\Delta t$$
$$x(t+\Delta t) = x(t) + v(t+\Delta t/2)\Delta t.$$

```
void leapfrog1(int n, float dt, float* restrict x,
               float* restrict v, float* restrict a)
{
    for (int i = 0; i < n; ++i, x += 2, v += 2, a += 2) {
        v[0] += a[0]*dt/2;
        v[1] += a[1]*dt/2;
        x[0] += v[0]*dt;
        x[1] += v[1]*dt;
    }
}
```

In the second step, we compute $a(t+\Delta t)$ based on the computed $x(t+\Delta t)$, and use the new acceleration to update the velocity from $t+\Delta t/2$ to $t+\Delta t$:

$$v(t+\Delta t) = v(t+\Delta t/2) + \frac{1}{2}a(t+\Delta t)\Delta t.$$

```
void leapfrog2(int n, float dt, float* restrict v, float* restrict a)
{
    for (int i = 0; i < n; ++i, v += 2, a += 2) {
        v[0] += a[0]*dt/2;
        v[1] += a[1]*dt/2;
    }
}
```

Of course, the interesting part of the work (the part that isn't embarrassingly parallel) occurs in the acceleration computation between the two phases.

The leapfrog integration scheme is attractive and widely used because it is *symplectic* — that is, it preserves something about the Hamiltonian structure of the original problem. In particular, leapfrog integration tends to do a pretty good job at neither gaining nor losing energy over time. The catch is that these nice properties disappear if one varies the time step, or uses different time steps for different particles. See recent work by Farr and Bertschinger (2007) for an example of a (more complicated) higher-order symplectic integrator that *does* allow variations in the time steps.

5 Reflection boundaries

In order to prevent our particles from flying off to infinity, we impose reflection boundary conditions (at the end of a step). These conditions should be imposed after the first half-step of Verlet. Currently we don't check if for multiple reflections — if they occur, something is amiss!

```
static void reflect(float wall, float* restrict x,
                    float* restrict v, float* restrict a)
{
    *x = (2*wall-(*x));
    *v = -(*v);
    *a = -(*a);
}

void apply_reflect(int n, float* restrict x,
                   float* restrict v, float* restrict a)
{
    for (int i = 0; i < n; ++i, x += 2, v += 2, a += 2) {
        if (x[0] < XMIN) reflect(XMIN, x+0, v+0, a+0);
        if (x[0] > XMAX) reflect(XMAX, x+0, v+0, a+0);
        if (x[1] < YMIN) reflect(YMIN, x+1, v+1, a+1);
        if (x[1] > YMAX) reflect(YMAX, x+1, v+1, a+1);
    }
}
```

6 Serial driver

6.1 The force field abstraction

Because I wanted to play with a variety of combinations of potentials, I decided to set up a system in which the force field is not hard-wired into the integrator. Instead, the integrator calls the force field evaluation via a function pointer of type `compute_force_t`.

The assumed data layout of the `x` array is $[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]$. The `F` array is treated similarly.

```
typedef
void (*compute_force_t)(int n,                      /* Particle count */
                        const float* restrict x,    /* Positions */
                        float* restrict F,          /* Forces */
                        void* fdata);               /* Parameters */
```

Despite the function abstraction, right now there's only one force computation: Lennard-Jones potential plus a gravitational field.

```
void compute_forces(int n, const float* restrict x, float* restrict F,
                    void* fdata)
{
    sim_param_t* params = (sim_param_t*) fdata;
    float g    = params->G;
    float eps  = params->eps_lj;
    float sig  = params->sig_lj;
    float sig2 = sig*sig;

    /* Global force downward (e.g. gravity) */
    for (int i = 0; i < n; ++i) {
        F[2*i+0] = 0;
        F[2*i+1] = -g;
    }

    /* Particle-particle interactions (Lennard-Jones) */
    for (int i = 0; i < n; ++i) {
        for (int j = i+1; j < n; ++j) {
            float dx = x[2*j+0]-x[2*i+0];
            float dy = x[2*j+1]-x[2*i+1];
            float C_LJ = compute_LJ_scalar(dx*dx+dy*dy, eps, sig2);
            F[2*i+0] += (C_LJ*dx);
            F[2*i+1] += (C_LJ*dy);
            F[2*j+0] -= (C_LJ*dx);
            F[2*j+1] -= (C_LJ*dy);
        }
    }
}
```

6.2 Initial conditions

We choose the particle velocity components to be normal with mean zero and variance T_0^2, and we choose the particles uniformly at random subject to the constrain that no too particles can be closer than the basic interaction radius of Lennard-Jones lest the simulation blow up on the first time steps (we choose $r_{\min} = \sigma$). It's possible to set the parameters such that we can't place as many particles as desired subject to this constraint; if that happens, we try to place as many as possible, then bail out.

```
int init_particles_random(int n, float* x, float* v, sim_param_t* params)
{
    const int MAX_INIT_TRIALS = 1000;

    float T0     = params->T0;
```

```
    float sig    = params->sig_lj;
    float min_r2 = sig*sig;

    for (int i = 0; i < n; ++i) {
        float r2 = 0;

        /* Choose velocity components from N(0,T0) */
        double R = T0 * sqrt(-2*log(drand48()));
        double T = 2*M_PI*drand48();
        v[2*i+0] = (float) (R * cos(T));
        v[2*i+0] = (float) (R * sin(T));

        /* Choose new point via rejection sampling */
        for (int trial = 0; r2 < min_r2 && trial < MAX_INIT_TRIALS; ++trial) {
            x[2*i+0] = (float) drand48();
            x[2*i+1] = (float) drand48();
            for (int j = 0; j < i; ++j) {
                float dx = x[2*i+0]-x[2*j+0];
                float dy = x[2*i+1]-x[2*j+1];
                r2 = dx*dx + dy*dy;
                if (r2 < min_r2)
                    break;
            }
        }

        /* If it takes too many trials, bail and declare victory! */
        if (i > 0 && r2 < min_r2)
            return i;
    }
    return n;
}
```

6.3 Simulating a box

The `run_box` routine simulates the motion of a collection of unit-mass particles in a the box $[0,1]^2$ using a leapfrog integration scheme. Every `npframes` time steps, a frame is written to the output file. As described in the previous section, we use a callback function to compute the force fields at each step.

```
void run_box(FILE* fp,              /* Output file */
             int n,                 /* Number of particles */
             int npframe,           /* Number of steps between frames */
             int nframes,           /* Number of frames generated */
             float dt,              /* Time step */
             float* restrict x,     /* Initial positions */
             float* restrict v,     /* Initial velocities */
             compute_force_t force, /* Function to compute force */
             void* force_data)      /* Data used by force() */
{
    float* a = (float*) malloc(2*n*sizeof(float));
```

```
    memset(v, 0, 2*n*sizeof(float));
    memset(a, 0, 2*n*sizeof(float));

    write_header(fp, n);
    write_frame_data(fp, n, x);
    force(n, x, a, force_data);
    for (int frame = 1; frame < nframes; ++frame) {
        for (int i = 0; i < npframe; ++i) {
            leapfrog1(n, dt, x, v, a);
            apply_reflect(n, x, v, a);
            force(n, x, a, force_data);
            leapfrog2(n, dt, v, a);
        }
        write_frame_data(fp, n, x);
    }

    free(a);
}
```

6.4 The `main` event

```
int main(int argc, char** argv)
{
    sim_param_t params;
    float* x;
    float* v;
    FILE* fp;
    int npart;

    if (get_params(argc, argv, &params) != 0)
        exit(-1);

    fp = fopen(params.fname, "w");
    x = malloc(2*params.npart*sizeof(float));
    v = malloc(2*params.npart*sizeof(float));

    npart = init_particles_random(params.npart, x, v, &params);
    if (npart < params.npart) {
        fprintf(stderr, "Could not generate %d particles; trying %d\n",
                params.npart, npart);
        params.npart = npart;
    }

    run_box(fp, params.npart, params.npframe, params.nframes,
            params.dt, x, v, compute_forces, &params);

    free(v);
    free(x);
    fclose(fp);
}
```

7 Parallel force computation with OpenMP

For the force computation before, we were a *little* bit clever: we used the symmetry of the Lennard-Jones interaction to simultaneously compute the effect of i on j and the effect of j on i. This roughly doubles the speed of the computation (in practice as well as in theory) and makes me happy. On the other hand, it means that the most obvious parallel `for` construct won't work without some synchronization! For our first attempt at an OpenMP implementation, we will instead accumulate the force contributions from each thread into a local array, and we'll combine those force contributions later.

We get some speed-up this way — on my laptop, it takes about 35 seconds for the OpenMP code than the 45 seconds that the serial code takes. We can do better.

```
static float** Ftemp;

void ftemp_init(sim_param_t* params)
{
    int n = params->npart;
    Ftemp = malloc(omp_get_max_threads() * sizeof(float*));
    for (int i = 0; i < omp_get_max_threads(); ++i)
        Ftemp[i] = malloc(2*n*sizeof(float));
}

void ftemp_destroy()
{
    for (int i = 0; i < omp_get_max_threads(); ++i)
        free(Ftemp[i]);
    free(Ftemp);
}

void compute_forces(int n, const float* restrict x, float* restrict F,
                    sim_param_t* params)
{
    float g    = params->G;
    float eps  = params->eps_lj;
    float sig  = params->sig_lj;
    float sig2 = sig*sig;

    /* Global force downward (e.g. gravity) */
    for (int i = 0; i < n; ++i) {
        F[2*i+0] = 0;
        F[2*i+1] = -g;
    }

    /* Particle-particle interactions (Lennard-Jones) */
    #pragma omp parallel shared(F,x,n)
    {
        float* Ft = Ftemp[omp_get_thread_num()];
        memset(Ft, 0, 2*n*sizeof(float));

        #pragma omp for schedule(static)
        for (int i = 0; i < n; ++i) {
            for (int j = i+1; j < n; ++j) {
                float dx = x[2*j+0]-x[2*i+0];
```

```
                float dy = x[2*j+1]-x[2*i+1];
                float C_LJ = compute_LJ_scalar(dx*dx+dy*dy, eps, sig2);
                Ft[2*i+0] += (C_LJ*dx);
                Ft[2*i+1] += (C_LJ*dy);
                Ft[2*j+0] -= (C_LJ*dx);
                Ft[2*j+1] -= (C_LJ*dy);
            }
        }

        #pragma omp critical
        for (int i = 0; i < 2*n; ++i)
            F[i] += Ft[i];
    }
}
```

The rest of the OpenMP code is nearly identical to the serial code.

8 MPI driver

Somewhat remarkably, this MPI code gets better performance than my serial code on my two-core laptop. I think this is a testament to the cleverness of the OpenMPI programmers more than any sort of statement about my (deliberately naive) code. My strategy is to have each processor claim ownership of a small number of particles, and to compute their force interactions and position updates. Then the information gets exchanged via an `MPI_Allgatherv`. This was the most straightforward parallelization I could think of — and I figured it would take much more work than I have with a few hundred particles to mask the latency of an all-to-all communication that I used at each step.

8.1 Global state

Usually, global variables are a Bad Idea. In this case, however:

1. The rank, size, and data types are written only once.
2. They are read from nearly everywhere.
3. They are at least declared `static` (local to the current file).

```
static int rank;
static int nproc;
static MPI_Datatype pairtype;
```

8.2 Problem partitioning

Divide the problem more-or-less evenly among processors. Every processor at least gets `num_each` = $\lfloor n/p \rfloor$ particles, and the first few processors each get one more.

```
void partition_problem(int* iparts, int* counts, int npart)
{
    int num_each = npart/nproc;
    int num_left = npart-num_each*nproc;
    iparts[0] = 0;
    for (int i = 0; i < nproc; ++i) {
```

```
        counts[i] = num_each + (i < num_left ? 1 : 0);
        iparts[i+1] = iparts[i] + counts[i];
    }
}
```

8.3 The force computation

Right now, we take local and global arrays of positions and a local array of forces, and compute the force into a local array. The local position array corresponds to indices `istart <= i < iend` in the global array. Apart from not trying to be clever about re-using computations, this looks much the same as (though not identical to) the serial code.

```
void compute_forces(int n, const float* restrict x,
                    int istart, int iend,
                    const float* restrict xlocal, float* restrict Flocal,
                    sim_param_t* params)
{
    int nlocal = iend-istart;
    float g    = params->G;
    float eps  = params->eps_lj;
    float sig  = params->sig_lj;
    float sig2 = sig*sig;

    /* Global force downward (e.g. gravity) */
    for (int i = 0; i < nlocal; ++i) {
        Flocal[2*i+0] = 0;
        Flocal[2*i+1] = -g;
    }

    /* Particle-particle interactions (Lennard-Jones) */
    for (int i = istart; i < iend; ++i) {
        int ii = i-istart;
        for (int j = 0; j < n; ++j) {
            if (i != j) {
                float dx = x[2*j+0]-xlocal[2*ii+0];
                float dy = x[2*j+1]-xlocal[2*ii+1];
                float C_LJ = compute_LJ_scalar(dx*dx+dy*dy, eps, sig2);
                Flocal[2*ii+0] += (C_LJ*dx);
                Flocal[2*ii+1] += (C_LJ*dy);
            }
        }
    }
}
```

8.4 Initial conditions

Previously, we chose the particles and their velocities simultaneously. Now, it doesn't make sense to do so. I know how to generate a random initial particle distribution by rejection on a single node given all the point coordinates, but I don't need to do this for the velocities.

```
int init_particles_random(int n, float* x, sim_param_t* params)
```

```
{
    const int MAX_INIT_TRIALS = 1000;

    float sig    = params->sig_lj;
    float min_r2 = sig*sig;

    for (int i = 0; i < n; ++i) {
        float r2 = 0;

        /* Choose new point via rejection sampling */
        for (int trial = 0; r2 < min_r2 && trial < MAX_INIT_TRIALS; ++trial) {
            x[2*i+0] = (float) drand48();
            x[2*i+1] = (float) drand48();
            for (int j = 0; j < i; ++j) {
                float dx = x[2*i+0]-x[2*j+0];
                float dy = x[2*i+1]-x[2*j+1];
                r2 = dx*dx + dy*dy;
                if (r2 < min_r2)
                    break;
            }
        }

        /* If it takes too many trials, bail and declare victory! */
        if (i > 0 && r2 < min_r2)
            return i;
    }
    return n;
}


void init_particles_random_v(int n, float* v, sim_param_t* params)
{
    float T0 = params->T0;
    for (int i = 0; i < n; ++i) {
        double R = T0 * sqrt(-2*log(drand48()));
        double T = 2*M_PI*drand48();
        v[2*i+0] = (float) (R * cos(T));
        v[2*i+0] = (float) (R * sin(T));
    }
}
```

8.5 Simulating a box

The `run_box` code looks similar to the serial code, except for a communication phase (`MPI_Allgatherv`) immediately after updating the positions. If MPI-2 had nonblocking collective operations, this would have been a perfect place to use them.

Note that we output whenever the output file `fp` is non-`NULL`.

```
void run_box(FILE* fp,                  /* Output file (at 0) */
             int n, int nlocal,         /* Counts (all and local) */
             int* iparts, int* counts,  /* Offsets and counts per proc */
```

```
              int npframe,               /* Steps per frame */
              int nframes,               /* Frames */
              float dt,                  /* Time step */
              float* restrict x,         /* Global position vec */
              float* restrict xlocal,    /* Local part of position */
              float* restrict vlocal,    /* Local part of velocity */
              sim_param_t* params)       /* Simulation params */
{
    float* alocal = (float*) malloc(2*nlocal*sizeof(float));
    memset(vlocal, 0, 2*nlocal*sizeof(float));
    memset(alocal, 0, 2*nlocal*sizeof(float));

    if (fp) {
        write_header(fp, n);
        write_frame_data(fp, n, x);
    }

    compute_forces(n, x, iparts[rank], iparts[rank+1],
                   xlocal, alocal, params);

    for (int frame = 1; frame < nframes; ++frame) {
        for (int i = 0; i < npframe; ++i) {
            leapfrog1(nlocal, dt, xlocal, vlocal, alocal);
            apply_reflect(nlocal, xlocal, vlocal, alocal);
            MPI_Allgatherv(xlocal, nlocal, pairtype,
                           x, counts, iparts, pairtype,
                           MPI_COMM_WORLD);
            compute_forces(n, x, iparts[rank], iparts[rank+1],
                           xlocal, alocal, params);
            leapfrog2(nlocal, dt, vlocal, alocal);
        }
        if (fp)
            write_frame_data(fp, n, x);
    }

    free(alocal);
}
```

8.6 The `main` event

As with many MPI codes, this code has a lot of bookkeeping, most of which I currently have in `main`. Other than the usual initialization, finalization, and inquiries, we have the following more interesting features:

1. We set up an MPI type (`pairtype`) to denote a pair of floating point numbers. Note that we have to do an `MPI_Type_commit` before we can make such a constructed data type useable to the rest of the system.
2. We initialize on the first processor, then send information out to the others using an `MPI_Bcast`.

```
int main(int argc, char** argv)
{
    sim_param_t params;
    float* x;
```

```
float* xlocal;
float* vlocal;
FILE* fp = NULL;
int npart;
int* iparts;
int* counts;
int nlocal;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Type_vector(1, 2, 1, MPI_FLOAT, &pairtype);
MPI_Type_commit(&pairtype);

if (get_params(argc, argv, &params) != 0) {
    MPI_Finalize();
    exit(-1);
}

/* Get file handle and initialize everything on P0 */
x = malloc(2*params.npart*sizeof(float));
if (rank == 0) {
    fp = fopen(params.fname, "w");
    npart = init_particles_random(params.npart, x, &params);
    if (npart < params.npart) {
        fprintf(stderr, "Could not generate %d particles; trying %d\n",
                params.npart, npart);
    }
}

/* Broadcast initial information from root */
MPI_Bcast(&npart, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(x, npart, pairtype, 0, MPI_COMM_WORLD);
params.npart = npart;

/* Decide who is responsible for which item */
counts = malloc( nproc   *sizeof(int));
iparts = malloc((nproc+1)*sizeof(int));
partition_problem(iparts, counts, npart);
nlocal = counts[rank];

/* Allocate space for local storage and copy in data */
xlocal = malloc(2*nlocal*sizeof(float));
vlocal = malloc(2*nlocal*sizeof(float));
memcpy(xlocal, x+2*iparts[rank], 2*nlocal*sizeof(float));
init_particles_random_v(nlocal, vlocal, &params);

run_box(fp, npart, nlocal, iparts, counts,
        params.npframe, params.nframes,
        params.dt, x, xlocal, vlocal, &params);
```

```
    free(vlocal);
    free(xlocal);
    free(iparts);
    free(x);
    if (fp)
        fclose(fp);

    MPI_Finalize();
    return 0;
}
```

9 Option processing

The `print_usage` command documents the options to the `nbody` driver program, and `default_params` sets the default parameter values. You may want to add your own options to control other aspects of the program. This is about as many options as I would care to handle at the command line — maybe more! Usually, I would start using a second language for configuration (e.g. Lua) to handle anything more than this.

```
static void print_usage()
{
    fprintf(stderr,
            "nbody\n"
            "\t-h: print this message\n"
            "\t-o: output file name (run.out)\n"
            "\t-n: number of particles (500)\n"
            "\t-F: number of frames (200)\n"
            "\t-f: steps per frame (100)\n"
            "\t-t: time step (1e-4)\n"
            "\t-e: epsilon parameter in LJ potential (1)\n"
            "\t-s: distance parameter in LJ potential (1e-2)\n"
            "\t-g: gravitational field strength (1)\n"
            "\t-T: initial temperature (1)\n");
}

static void default_params(sim_param_t* params)
{
    params->fname   = "run.out";
    params->npart   = 500;
    params->nframes = 400;
    params->npframe = 50;
    params->dt      = 1e-4;
    params->eps_lj  = 1;
    params->sig_lj  = 1e-2;
    params->G       = 1;
    params->T0      = 1;
}
```

The `get_params` function uses the `getopt` package to handle the actual argument processing. Note that `getopt` is *not* thread-safe! You will need to do some synchronization if you want to use this function safely with threaded code.

```
int get_params(int argc, char** argv, sim_param_t* params)
{
    extern char* optarg;
    const char* optstring = "ho:n:F:f:t:e:s:g:T:";
    int c;

    #define get_int_arg(c, field) \
        case c: params->field = atoi(optarg); break
    #define get_flt_arg(c, field) \
        case c: params->field = (float) atof(optarg); break

    default_params(params);
    while ((c = getopt(argc, argv, optstring)) != -1) {
        switch (c) {
        case 'h':
            print_usage();
            return -1;
        case 'o':
            strcpy(params->fname = malloc(strlen(optarg)+1), optarg);
            break;
        get_int_arg('n', npart);
        get_int_arg('F', nframes);
        get_int_arg('f', npframe);
        get_flt_arg('t', dt);
        get_flt_arg('e', eps_lj);
        get_flt_arg('s', sig_lj);
        get_flt_arg('g', G);
        get_flt_arg('T', T0);
        default:
            fprintf(stderr, "Unknown option\n");
            return -1;
        }
    }
    return 0;
}
```

10 Binary output

There are two output file options for our code: text and binary. Originally, I had only text output; but I got impatient waiting to view some of my longer runs, and I wanted something a bit more compact, so added a binary option

The viewer distinguishes the file type by looking at the first few characters in the file: the tag `NBView00` means that what follows is text data, while the tag `NBView01` means that what follows is binary. If you care about being able to read the results of your data files across multiple versions of a code, it's a good idea to have such a tag!

```
#define VERSION_TAG "NBView01"
```

Different platforms use different byte orders. The Intel Pentium hardware is little-endian, which means that it puts the least-significant byte first – almost like if we were to write a hundred twenty as 021 rather than 120. The Java system (which is where our viewer lives) is big-endian. Big-endian ordering is also the so-called "wire standard" for sending data over a network, so UNIX provides functions `htonl` and `htons` to convert long (32-bit) and short (16-bit) numbers from the host representation to the wire representation. There is no corresponding function `htonf`

for floating point data, but we can construct such a function by pretending floats look like 32-bit integers — the byte shuffling is the same.

```
uint32_t htonf(void* data)
{
    return htonl(*(uint32_t*) data);
}
```

The header data consists of a count of the number of balls (a 32-bit integer) and a scale parameter (a 32-bit floating point number). The scale parameter tells the viewer how big the view box is supposed to be in the coordinate system of the simulation; right now, it is always set to be 1 (i.e. the view box is $[0, 1] \times [0, 1]$)

```
void write_header(FILE* fp, int n)
{
    float scale = 1.0;
    uint32_t nn = htonl((uint32_t) n);
    uint32_t nscale = htonf(&scale);
    fprintf(fp, "%s\n", VERSION_TAG);
    fwrite(&nn,     sizeof(nn),     1, fp);
    fwrite(&nscale, sizeof(nscale), 1, fp);
}
```

After the header is a sequence of frames, each of which contains $n_{\text{particles}}$ pairs of 32-bit int floating point numbers. There are no markers, end tags, etc; just the raw data. The `write_frame_data` routine writes n pairs of floats; note that writing a single frame of output may involve multiple calls to `write_frame_data` Frame data just consists integer) and a scale parameter (a 32-bit floating point number). The scale parameter tells the viewer how big the view box is supposed to be in the coordinate system of the simulation; right now, it is always set to be 1 (i.e. the view box is $[0, 1] \times [0, 1]$)

```
void write_frame_data(FILE* fp, int n, float* x)
{
    for (int i = 0; i < n; ++i) {
        uint32_t xi = htonf(x++);
        uint32_t yi = htonf(x++);
        fwrite(&xi, sizeof(xi), 1, fp);
        fwrite(&yi, sizeof(yi), 1, fp);
    }
}
```

11 Postscript

This document is partly meant to introduce the code for our second homework assignment, and partly as an exercise (for me) in working with a documentation tool called `dsbweb` that I developed a couple years ago. This documentation tool provides a minimalist system to support Knuth's idea of "literate programming," or writing programs as one might write essays.