1 The SPH equations

Smoothed particle hydrodynamics (SPH) is a particle-based method for simulating the behavior of fluids. Each computational particle carries along information about the fluid in a little region, such as the velocity and density; and during the course of the simulation, these particles interact with each other in a way that models the dynamics of a fluid. In this project, we will tune a simple 2D version of an SPH method described by Müller et al for use in interactive graphics [1].

Our simulation basically solves a system of ordinary differential equations[1] for a collection of particles with equal masses m and interaction radii h. Each particle i has a position $\mathbf{r}_i$, a velocity $\mathbf{v}_i$, and a density ρ_i. Particle i interacts with the set N_i of particles within radius h of i. The density is computed at each step by

$$\rho_i = \frac{4m}{\pi h^8} \sum_{j \in N_i} (h^2 - r^2)^3.$$

The acceleration is computed by the rule

$$\mathbf{a}_i = \frac{1}{\rho_i} \sum_{j \in N_i} \mathbf{f}_{ij}^{\text{interact}} + \mathbf{g},$$

where

$$\mathbf{f}_{ij}^{\text{interact}} = \frac{m_j}{\pi h^4 \rho_j}(1 - q_{ij}) \left[15k(\rho_i + \rho_j - 2\rho_0)\frac{(1-q_{ij})}{q_{ij}}\mathbf{r}_{ij} - 40\mu \mathbf{v}_{ij} \right],$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$, $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$, and $q_{ij} = \|\mathbf{r}_{ij}\|/h$. The parameters in these expressions are

$$\begin{aligned} \rho_0 &= \text{reference mass density} \\ k &= \text{bulk modulus} \\ \mu &= \text{viscosity} \\ \mathbf{g} &= \text{gravitational vector} \end{aligned}$$

By default, we choose most of these parameters to be appropriate to a liquid like water. The exception is the bulk modulus, which is chosen so that the computational speed of sound

$$c_s = \sqrt{\frac{k}{\rho_0}}$$

is large relative to the typical velocities we expect to see in the simulation, but not too large. Choosing k to be very large (e.g. on the scale of the bulk modulus for water) severely limits the time step size needed for stable simulation.

[1]We describe the derivation of the 2D equations in a separate document. It may interest those of you who care about fluid dynamics, but it is not critical to understand the derivation in order to do the assignment.

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          */
    int   nframes; /* Number of frames   */
    int   npframe; /* Steps per frame    */
    float h;       /* Particle size      */
    float dt;      /* Time step          */
    float rho0;    /* Reference density  */
    float k;       /* Bulk modulus       */
    float mu;      /* Viscosity          */
    float g;       /* Gravity strength   */
} sim_param_t;

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

3 System state

The sim_state_t structure holds the information for the current state of the system and of the integration algorithm. The array x has length $2n$, with x[2*i+0] and x[2*i+1] representing the x and y coordinates of the particle positions. The layout for v, vh, and a is similar, while rho only has one entry per particle.

The alloc_state and free_state functions take care of storage for the local simulation state.

```
typedef struct sim_state_t {
    int n;                /* Number of particles    */
    float mass;           /* Particle mass          */
    float* restrict rho;  /* Densities              */
    float* restrict x;    /* Positions              */
    float* restrict vh;   /* Velocities (half step) */
    float* restrict v;    /* Velocities (full step) */
    float* restrict a;    /* Acceleration           */
} sim_state_t;

sim_state_t* alloc_state(int n);
void free_state(sim_state_t* s);
```

3.1 Density computations

The formula for density is

$$\rho_i = \frac{4m}{\pi h^8} \sum_{j \in N_i} (h^2 - r^2)^3.$$

We search for neighbors of node i by checking every particle, which is not very efficient. We do at least take advange of the symmetry of the update (i contributes to j in the same way that j contributes to i).

```
void compute_density(sim_state_t* s, sim_param_t* params)
{
    int n = s->n;
    float* restrict rho = s->rho;
    const float* restrict x = s->x;

    float h  = params->h;
    float h2 = h*h;
    float h8 = ( h2*h2 )*( h2*h2 );
    float C  = 4 * s->mass / M_PI / h8;

    memset(rho, 0, n*sizeof(float));
    for (int i = 0; i < n; ++i) {
        rho[i] += 4 * s->mass / M_PI / h2;
        for (int j = i+1; j < n; ++j) {
            float dx = x[2*i+0]-x[2*j+0];
            float dy = x[2*i+1]-x[2*j+1];
            float r2 = dx*dx + dy*dy;
            float z  = h2-r2;
            if (z > 0) {
                float rho_ij = C*z*z*z;
                rho[i] += rho_ij;
                rho[j] += rho_ij;
            }
        }
    }
}
```

3.2 Computing forces

The acceleration is computed by the rule

$$\mathbf{a}_i = \frac{1}{\rho_i} \sum_{j \in N_i} \mathbf{f}_{ij}^{\text{interact}} + \mathbf{g},$$

where the pair interaction formula is as previously described. Like `compute_density`, the `compute_accel` routine takes advantage of the symmetry of the interaction forces ($\mathbf{f}_{ij}^{\text{interact}} = -\mathbf{f}_{ji}^{\text{interact}}$) but it does a very expensive brute force search for neighbors.

```
void compute_accel(sim_state_t* state, sim_param_t* params)
{
    // Unpack basic parameters
    const float h    = params->h;
    const float rho0 = params->rho0;
    const float k    = params->k;
    const float mu   = params->mu;
    const float g    = params->g;
    const float mass = state->mass;
    const float h2   = h*h;

    // Unpack system state
    const float* restrict rho = state->rho;
    const float* restrict x   = state->x;
    const float* restrict v   = state->v;
    float* restrict a         = state->a;
    int n = state->n;

    // Compute density and color
    compute_density(state, params);

    // Start with gravity and surface forces
    for (int i = 0; i < n; ++i) {
        a[2*i+0] = 0;
        a[2*i+1] = -g;
    }

    // Constants for interaction term
    float C0 = mass / M_PI / ( (h2)*(h2) );
    float Cp =  15*k;
    float Cv = -40*mu;

    // Now compute interaction forces
    for (int i = 0; i < n; ++i) {
        const float rhoi = rho[i];
        for (int j = i+1; j < n; ++j) {
            float dx = x[2*i+0]-x[2*j+0];
            float dy = x[2*i+1]-x[2*j+1];
            float r2 = dx*dx + dy*dy;
            if (r2 < h2) {
                const float rhoj = rho[j];
                float q = sqrt(r2)/h;
                float u = 1-q;
                float w0 = C0 * u/rhoi/rhoj;
                float wp = w0 * Cp * (rhoi+rhoj-2*rho0) * u/q;
                float wv = w0 * Cv;
                float dvx = v[2*i+0]-v[2*j+0];
                float dvy = v[2*i+1]-v[2*j+1];
                a[2*i+0] += (wp*dx + wv*dvx);
                a[2*i+1] += (wp*dy + wv*dvy);
                a[2*j+0] -= (wp*dx + wv*dvx);
                a[2*j+1] -= (wp*dy + wv*dvy);
            }
        }
    }
}
```

4 Leapfrog integration

The leapfrog time integration scheme is frequently used in particle simulation algorithms because

- It is explicit, which makes it easy to code.
- It is second-order accurate.
- It is *symplectic*, which means that it conserves certain properties of the continuous differential equation for Hamiltonian systems. In practice, this means that it tends to conserve energy where energy is supposed to be conserved, assuming the time step is short enough for stability.

Of course, our system is *not* Hamiltonian – viscosity is a form of damping, so the system loses energy. But we'll stick with the leapfrog integration scheme anyhow.

The leapfrog time integration algorithm is named because the velocities are updated on half steps and the positions on integer steps; hence, the two leap over each other. After computing accelerations, one step takes the form

$$\begin{aligned}\mathbf{v}^{i+1/2} &= \mathbf{v}^{i-1/2} + \mathbf{a}^i \Delta t \\ \mathbf{r}^{i+1} &= \mathbf{r}^i + \mathbf{v}^{i+1/2} \Delta t,\end{aligned}$$

This is straightforward enough, except for two minor points.

1. In order to compute the acceleration at time t, we need the velocity at time t. But leapfrog only computes velocities at half steps! So we cheat a little: when we compute the half-step velocity velocity $\mathbf{v}^{i+1/2}$ (stored

in vh), we simultaneously compute an approximate integer step velocity $\tilde{\mathbf{v}}^{i+1}$ (stored in v) by taking another half step using the acceleration $\mathbf{a}^i$.

2. We don't explicitly represent the boundary by fixed particles, so we need some way to enforce the boundary conditions. We take the simple approach of explicitly reflecting the particles using the reflect_bc routine discussed below.

```
void leapfrog_step(sim_state_t* s, double dt)
{
    const float* restrict a = s->a;
    float* restrict vh = s->vh;
    float* restrict v  = s->v;
    float* restrict x  = s->x;
    int n = s->n;
    for (int i = 0; i < 2*n; ++i) vh[i] += a[i]  * dt;
    for (int i = 0; i < 2*n; ++i) v[i]   = vh[i] + a[i] * dt / 2;
    for (int i = 0; i < 2*n; ++i) x[i]  += vh[i] * dt;
    reflect_bc(s);
}
```

At the first step, the leapfrog iteration only has the initial velocities $\mathbf{v}^0$, so we need to do something special.

$$\begin{aligned}\mathbf{v}^{1/2} &= \mathbf{v}^0 + \mathbf{a}^0 \Delta t/2 \\ \mathbf{r}^1 &= \mathbf{r}^0 + \mathbf{v}^{1/2}\Delta t.\end{aligned}$$

```
void leapfrog_start(sim_state_t* s, double dt)
{
    const float* restrict a = s->a;
    float* restrict vh = s->vh;
    float* restrict v  = s->v;
    float* restrict x  = s->x;
    int n = s->n;
    for (int i = 0; i < 2*n; ++i) vh[i]  = v[i] + a[i] * dt / 2;
    for (int i = 0; i < 2*n; ++i) v[i]  += a[i]  * dt;
    for (int i = 0; i < 2*n; ++i) x[i]  += vh[i] * dt;
    reflect_bc(s);
}
```

5 Reflection boundary conditions

Our boundary condition corresponds to hitting an inelastic boundary with a specified coefficient of restitution less than one. When a particle hits a vertical barrier (`which = 0`) or a horizontal barrier (`which = 1`), we process it with `damp_reflect`. This reduces the total distance traveled based on the time since the collision reflected, damps the velocities, and reflects whatever solution components should be reflected.

```
static void damp_reflect(int which, float barrier,
                         float* x, float* v, float* vh)
{
    // Coefficient of resitiution
    const float DAMP = 0.75;

    // Ignore degenerate cases
    if (v[which] == 0)
        return;

    // Scale back the distance traveled based on time from collision
    float tbounce = (x[which]-barrier)/v[which];
    x[0] -= v[0]*(1-DAMP)*tbounce;
    x[1] -= v[1]*(1-DAMP)*tbounce;

    // Reflect the position and velocity
    x[which]  = 2*barrier-x[which];
    v[which]  = -v[which];
    vh[which] = -vh[which];

    // Damp the velocities
    v[0] *= DAMP;  vh[0] *= DAMP;
    v[1] *= DAMP;  vh[1] *= DAMP;
}
```

For each particle, we need to check for reflections on each of the four walls of the computational domain.

```
static void reflect_bc(sim_state_t* s)
{
    // Boundaries of the computational domain
    const float XMIN = 0.0;
    const float XMAX = 1.0;
    const float YMIN = 0.0;
    const float YMAX = 1.0;
    float* restrict vh = s->vh;
    float* restrict v  = s->v;
    float* restrict x  = s->x;
    int n = s->n;
    for (int i = 0; i < n; ++i, x += 2, v += 2, vh += 2) {
        if (x[0] < XMIN) damp_reflect(0, XMIN, x, v, vh);
        if (x[0] > XMAX) damp_reflect(0, XMAX, x, v, vh);
        if (x[1] < YMIN) damp_reflect(1, YMIN, x, v, vh);
        if (x[1] > YMAX) damp_reflect(1, YMAX, x, v, vh);
    }
}
```

6 Initialization

We've hard coded the computational domain to a unit box, but we'd prefer to do something more flexible for the initial distribution of fluid. In particular, we define the initial geometry of the fluid in terms of an *indicator function* that is one for points in the domain occupied by fluid and zero elsewhere. A `domain_fun_t` is a pointer to an indicator for a domain, which is a function that takes two floats and returns 0 or 1. Two examples of indicator functions are a little box of fluid in the corner of the domain and a circular drop.

```
typedef int (*domain_fun_t)(float, float);

int box_indicator(float x, float y)
{
    return (x < 0.5) && (y < 0.5);
}

int circ_indicator(float x, float y)
{
    float dx = (x-0.5);
    float dy = (y-0.3);
    float r2 = dx*dx + dy*dy;
    return (r2 < 0.25*0.25);
}
```

The `place_particles` routine fills a region (indicated by the `indicatef` argument) with fluid particles. The fluid particles are placed at points inside the domain that lie on a regular mesh with cell sizes of $h/1.3$. This is close enough to allow the particles to overlap somewhat, but not too much.

```
sim_state_t* place_particles(sim_param_t* param,
                             domain_fun_t indicatef)
{
    float h  = param->h;
    float hh = h/1.3;

    // Count mesh points that fall in indicated region.
    int count = 0;
    for (float x = 0; x < 1; x += hh)
        for (float y = 0; y < 1; y += hh)
            count += indicatef(x,y);

    // Populate the particle data structure
    sim_state_t* s = alloc_state(count);
    int p = 0;
    for (float x = 0; x < 1; x += hh) {
        for (float y = 0; y < 1; y += hh) {
            if (indicatef(x,y)) {
                s->x[2*p+0] = x;
                s->x[2*p+1] = y;
                s->v[2*p+0] = 0;
                s->v[2*p+1] = 0;
                ++p;
            }
        }
    }
    return s;
}
```

The place_particle routine determines the initial particle placement, but not the desired mass. We want the fluid in the initial configuration to exist roughly at the reference density. One way to do this is to take the volume in the indicated body of fluid, multiply by the mass density, and divide by the number of particles; but that requires that we be able to compute the volume of the fluid region. Alternately, we can simply compute the average mass density assuming each particle has mass one, then use that to compute the particle mass necessary in order to achieve the desired reference density. We do this with normalize_mass.

```
void normalize_mass(sim_state_t* s, sim_param_t* param)
{
    s->mass = 1;
    compute_density(s, param);
    float rho0 = param->rho0;
    float rho2s = 0;
    float rhos  = 0;
    for (int i = 0; i < s->n; ++i) {
        rho2s += (s->rho[i])*(s->rho[i]);
        rhos  += s->rho[i];
    }
    s->mass *= ( rho0*rhos / rho2s );
}

sim_state_t* init_particles(sim_param_t* param)
{
    sim_state_t* s = place_particles(param, box_indicator);
    normalize_mass(s, param);
    return s;
}
```

7 The `main` event

The `main` routine actually runs the time step loop, writing out files for visualization every few steps. For debugging convenience, we use `check_state` before writing out frames, just so that we don't spend a lot of time on a simulation that has gone berserk.

```
void check_state(sim_state_t* s)
{
    for (int i = 0; i < s->n; ++i) {
        float xi = s->x[2*i+0];
        float yi = s->x[2*i+1];
        assert( xi >= 0 || xi <= 1 );
        assert( yi >= 0 || yi <= 1 );
    }
}

int main(int argc, char** argv)
{
    sim_param_t params;
    if (get_params(argc, argv, &params) != 0)
        exit(-1);
    sim_state_t* state = init_particles(&params);
    FILE* fp    = fopen(params.fname, "w");
    int nframes = params.nframes;
    int npframe = params.npframe;
    float dt    = params.dt;
    int n       = state->n;

    tic(0);
    write_header(fp, n);
    write_frame_data(fp, n, state->x, NULL);
    compute_accel(state, &params);
    leapfrog_start(state, dt);
    check_state(state);
    for (int frame = 1; frame < nframes; ++frame) {
        for (int i = 0; i < npframe; ++i) {
            compute_accel(state, &params);
            leapfrog_step(state, dt);
            check_state(state);
        }
        write_frame_data(fp, n, state->x, NULL);
    }
    printf("Ran in %g seconds\n", toc(0));

    fclose(fp);
    free_state(state);
}
```

8 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 default_params(sim_param_t* params)
{
    params->fname   = "run.out";
    params->nframes = 400;
    params->npframe = 100;
    params->dt      = 1e-4;
    params->h       = 5e-2;
    params->rho0    = 1000;
    params->k       = 1e3;
    params->mu      = 0.1;
    params->g       = 9.8;
}

static void print_usage()
{
    sim_param_t param;
    default_params(&param);
    fprintf(stderr,
            "nbody\n"
            "\t-h: print this message\n"
            "\t-o: output file name (%s)\n"
            "\t-F: number of frames (%d)\n"
            "\t-f: steps per frame (%d)\n"
            "\t-t: time step (%e)\n"
            "\t-s: particle size (%e)\n"
            "\t-d: reference density (%g)\n"
            "\t-k: bulk modulus (%g)\n"
            "\t-v: dynamic viscosity (%g)\n"
            "\t-g: gravitational strength (%g)\n",
            param.fname, param.nframes, param.npframe,
            param.dt, param.h, param.rho0,
            param.k, param.mu, param.g);
}
```

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:F:f:t:s:d:k:v:g:";
    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('F', nframes);
        get_int_arg('f', npframe);
        get_flt_arg('t', dt);
        get_flt_arg('s', h);
        get_flt_arg('d', rho0);
        get_flt_arg('k', k);
        get_flt_arg('v', mu);
        get_flt_arg('g', g);
        default:
            fprintf(stderr, "Unknown option\n");
            return -1;
        }
    }
    return 0;
}
```

9 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 "SPHView01"
```

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 and an optional flag which is used to determine the color. There are no markers, end tags, etc; just the raw data. The `write_frame_data` routine writes n point records; note that writing a single frame of output may involve multiple calls to `write_frame_data`.

```
void write_frame_data(FILE* fp, int n, float* x, int* c)
{
    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);
        uint32_t ci0 = c ? *c++ : 0;
        uint32_t ci = htonl(ci0);
        fwrite(&ci, sizeof(ci), 1, fp);
    }
}
```

References

[1] M. Müller, D. Charypar, and M. Gross. *Particle-based fluid simulation for interactive applications*, in Proceedings of Eurographics/SIGGRAPH Symposium on Computer Animation.