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 3D SPH method described by Müller et al for use in graphics [1]. There are better methods for this problem (and this implementation is arguably incomplete – we left out the surface tension forces), but this method does illustrate common issues in particle-based methods.

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

$$\mathbf{f}_{ij}^{\text{interact}} = \frac{45}{\pi h^5} \frac{m_j}{\rho_j} (1 - q_{ij}) \left[\frac{k}{2} (\rho_i + \rho_j - 2\rho_0) \frac{(1 - q_{ij})}{q_{ij}} \mathbf{r}_{ij} - \mu \mathbf{v}_{ij} \right],$$

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

```
typedef struct particle_t {
    float rho;         /* Particle density   */
    float x[3];        /* Particle positions */
    float v[3];        /* Particle velocities (full step) */
    float vh[3];       /* Particle velocities (half step) */
    float a[3];        /* Particle accelerations */
    struct particle_t* next;  /* List link for spatial hashing */
} particle_t;

typedef struct sim_state_t {
    int n;                /* Number of particles    */
    float mass;           /* Particle mass          */
    particle_t* part;     /* Particles              */
    particle_t** hash;    /* Hash table             */
} 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

$$\rho_i = \sum_j m_j W_{p6}(r_i - r_j, h) = \frac{315m}{64\pi h^9} \sum_{j \in N_i} (h^2 - r^2)^3.$$

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).

```
inline
void update_density(particle_t* pi, particle_t* pj, float h2, float C)
{
    float r2 = vec3_dist2(pi->x, pj->x);
    float z  = h2-r2;
    if (z > 0) {
        float rho_ij = C*z*z*z;
        pi->rho += rho_ij;
        pj->rho += rho_ij;
    }
}

void compute_density(sim_state_t* s, sim_param_t* params)
{
    int n = s->n;
    particle_t* p = s->part;
    particle_t** hash = s->hash;

    float h  = params->h;
    float h2 = h*h;
    float h3 = h2*h;
    float h9 = h3*h3*h3;
    float C  = ( 315.0/64.0/M_PI ) * s->mass / h9;

    // Clear densities
    for (int i = 0; i < n; ++i)
        p[i].rho = 0;

    // Accumulate density info
#ifdef USE_BUCKETING
    /* BEGIN TASK */
    /* END TASK */
#else
    for (int i = 0; i < n; ++i) {
        particle_t* pi = s->part+i;
```

```
        pi->rho += 4 * s->mass / M_PI / h3;
        for (int j = i+1; j < n; ++j) {
            particle_t* pj = s->part+j;
            update_density(pi, pj, h2, C);
        }
    }
#endif
}
```

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.

```
inline
void update_forces(particle_t* pi, particle_t* pj, float h2,
                   float rho0, float C0, float Cp, float Cv)
{
    float dx[3];
    vec3_diff(dx, pi->x, pj->x);
    float r2 = vec3_len2(dx);
    if (r2 < h2) {
        const float rhoi = pi->rho;
        const float rhoj = pj->rho;
        float q = sqrt(r2/h2);
        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 dv[3];
        vec3_diff(dv, pi->v, pj->v);

        // Equal and opposite pressure forces
        vec3_saxpy(pi->a,  wp, dx);
        vec3_saxpy(pj->a, -wp, dx);

        // Equal and opposite viscosity forces
```

```
        vec3_saxpy(pi->a,  wv, dv);
        vec3_saxpy(pj->a, -wv, dv);
    }
}

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
    particle_t* p = state->part;
    particle_t** hash = state->hash;
    int n = state->n;

    // Rehash the particles
    hash_particles(state, h);

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

    // Start with gravity and surface forces
    for (int i = 0; i < n; ++i)
        vec3_set(p[i].a,  0, -g, 0);

    // Constants for interaction term
    float C0 = 45 * mass / M_PI / ( (h2)*(h2)*h );
    float Cp = k/2;
    float Cv = -mu;

    // Accumulate forces
#ifdef USE_BUCKETING
    /* BEGIN TASK */
    /* END TASK */
#else
    for (int i = 0; i < n; ++i) {
        particle_t* pi = p+i;
        for (int j = i+1; j < n; ++j) {
            particle_t* pj = p+j;
            update_forces(pi, pj, h2, rho0, C0, Cp, Cv);
```

```
        }
    }
#endif
}
```

4 Spatial hashing

We conceptually partition our computational domain into bins that are at least h on a side, and label each with integer coordinates (i_x, i_y, i_z). In three dimensions, the bin size doesn't have to be all that small before the number of such bins is quite large, and most bins will be empty. Rather than represent each bin explicitly, we will map the bins to locations into a hash table, allowing the possibility that different bins can map to the same location in the hash table (though ideally this should not happen too often). We compute the hash function by mapping (i_x, i_y, i_z) to a Z-Morton integer index, which we then associate with a hash bucket. By figuring out the hash buckets in which the neighbor of a given particle could possibly lie, we significantly reduce the cost of checking interactions.

In the current implementation, we set the bin size equal to h, which implies that particles interacting with a given particle might lie in any of 27 possible neighbors. If you use a bin of size $2h$, you may have more particles in each bin, but you would only need to check eight bins (at most) for possible interactions. In order to allow for the possibility of more or fewer possible bins containing neighbors, we define `MAX_NBR_BINS` to be the maximum number of bins we will ever need to check for interactions, and let `particle_neighborhood` return both which bins are involved (as an output argument) and the number of bins needed (via the return value).

We also currently use the last few bits of each of the i_x, i_y, i_z indices to form the Z-Morton index. You may want to change the number of bits used, or change to some other hashing scheme that potentially spreads the bins more uniformly across the table.

```
#define HASH_DIM 0x10
#define HASH_SIZE (HASH_DIM*HASH_DIM*HASH_DIM)
#define MAX_NBR_BINS 27

unsigned particle_bucket(particle_t* p, float h);
unsigned particle_neighborhood(unsigned* buckets, particle_t* p, float h);
void hash_particles(sim_state_t* s, float h);
```

4.1 Z-Morton encoding

We use a Z-Morton encoding to map a triple of integer indices (i_x, i_y, i_z) to a single integer. If you want to see a picture of the Z-Morton ordering, I recommend the Wikipedia page! In terms of computation, though, the Z-Morton ordering simply interleaves the bits of the three independent indices. That is, if i_x^j is the bit in the 2^j place for index i_x, the bit pattern for the Z-Morton code looks like

$$c = (\ldots i_z^1 \; i_y^1 \; i_x^1 \; i_z^0 \; i_y^0 \; i_x^0)_2.$$

While this is not as good a space-filling curve as a Hilbert curve, it's very cheap.

The concrete code combines three 10-bit (max) indices into a single 32-bit Z-Morton code. The code is adapted from

http://fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/

```
inline unsigned zm_part1by2(unsigned x)
{
    x &= 0x000003ff;
    x = (x ^ (x << 16)) & 0xff0000ff;
    x = (x ^ (x <<  8)) & 0x0300f00f;
    x = (x ^ (x <<  4)) & 0x030c30c3;
    x = (x ^ (x <<  2)) & 0x09249249;
    return x;
}

inline unsigned zm_compact1by2(unsigned x)
{
    x &= 0x09249249;
    x = (x ^ (x >>  2)) & 0x030c30c3;
    x = (x ^ (x >>  4)) & 0x0300f00f;
    x = (x ^ (x >>  8)) & 0xff0000ff;
    x = (x ^ (x >> 16)) & 0x000003ff;
    return x;
}

inline unsigned zm_encode(unsigned x, unsigned y, unsigned z)
{
    return (zm_part1by2(z) << 2) + (zm_part1by2(y) << 1) + zm_part1by2(x);
}

inline void zm_decode(unsigned code, unsigned* x, unsigned* y, unsigned* z)
{
    *x = zm_compact1by2(code);
    *y = zm_compact1by2(code >> 1);
    *z = zm_compact1by2(code >> 2);
```

```
}
```

4.2 Spatial hashing implementation

In the current implementation, we assume `HASH_DIM` is 2^b, so that computing a bitwise of an integer with `HASH_DIM` extracts the b lowest-order bits. We could make `HASH_DIM` be something other than a power of two, but we would then need to compute an integer modulus or something of that sort.

```
#define HASH_MASK (HASH_DIM-1)

unsigned particle_bucket(particle_t* p, float h)
{
    unsigned ix = p->x[0]/h;
    unsigned iy = p->x[1]/h;
    unsigned iz = p->x[2]/h;
    return zm_encode(ix & HASH_MASK, iy & HASH_MASK, iz & HASH_MASK);
}

unsigned particle_neighborhood(unsigned* buckets, particle_t* p, float h)
{
    /* BEGIN TASK */
    /* END TASK */
}

void hash_particles(sim_state_t* s, float h)
{
    /* BEGIN TASK */
    /* END TASK */
}
```

5 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)
{
    int n = s->n;
    for (int i = 0; i < n; ++i) {
        particle_t* p = s->part + i;
        vec3_saxpy(p->vh, dt,  p->a);
        vec3_copy(p->v, p->vh);
        vec3_saxpy(p->v, dt/2, p->a);
        vec3_saxpy(p->x, dt,   p->vh);
    }
    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)
{
```

```
    int n = s->n;
    for (int i = 0; i < n; ++i) {
        particle_t* p = s->part + i;
        vec3_copy(p->vh, p->v);
        vec3_saxpy(p->vh, dt/2, p->a);
        vec3_saxpy(p->v,  dt,   p->a);
        vec3_saxpy(p->x,  dt,   p->vh);
    }
    reflect_bc(s);
}
```

6 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 barrier, 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];
    vec3_saxpy(x, -(1-DAMP)*tbounce, v);

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

    // Damp the velocities
    vec3_scalev(v,  DAMP);
    vec3_scalev(vh, 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;
    const float ZMIN = 0.0;
    const float ZMAX = 1.0;

    int n = s->n;
    for (int i = 0; i < n; ++i) {
        float* vh = s->part[i].vh;
        float* v  = s->part[i].v;
        float* x  = s->part[i].x;
        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);
        if (x[2] < ZMIN) damp_reflect(2, ZMIN, x, v, vh);
        if (x[2] > ZMAX) damp_reflect(2, ZMAX, x, v, vh);
    }
}
```

7 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, float);

int box_indicator(float x, float y, float z)
{
    return (x < 0.5) && (y < 0.75) && (z < 0.5);
}

int circ_indicator(float x, float y, float z)
{
```

```
    float dx = (x-0.5);
    float dy = (y-0.5);
    float dz = (z-0.5);
    float r2 = dx*dx + dy*dy + dz*dz;
    return (r2 < 0.25*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)
                for (float z = 0; z < 1; z += hh)
                        count += indicatef(x,y,z);

    // 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) {
            for (float z = 0; z < 1; z += hh) {
                if (indicatef(x,y,z)) {
                    vec3_set(s->part[p].x, x, y, z);
                    vec3_set(s->part[p].v, 0, 0, 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;
    hash_particles(s, param->h);
    compute_density(s, param);
    float rho0 = param->rho0;
    float rho2s = 0;
    float rhos  = 0;
    for (int i = 0; i < s->n; ++i) {
        rho2s += (s->part[i].rho)*(s->part[i].rho);
        rhos  += s->part[i].rho;
    }
    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;
}
```

8 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->part[i].x[0];
        float yi = s->part[i].x[1];
        float zi = s->part[i].x[2];
        assert( xi >= 0 || xi <= 1 );
        assert( yi >= 0 || yi <= 1 );
```

```
        assert( zi >= 0 || zi <= 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;

    double t_start = omp_get_wtime();
    //write_header(fp, n);
    write_header(fp, n, nframes, params.h);
    write_frame_data(fp, n, state, 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);
        }
        printf("Frame: %d of %d - %2.1f%%\n",frame, nframes,
               100*(float)frame/nframes);
        write_frame_data(fp, n, state, NULL);
    }
    double t_end = omp_get_wtime();
    printf("Ran in %g seconds\n", t_end-t_start);

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

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 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;
}
```

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.