ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computational Engineering

Spring semester 2014

Prof. P. Koumoutsakos, G. Tauriello
ETH Zentrum, CLT F 12, E 11
CH-8092 Zürich

Solution 6

Issued: 6.5.2014

Question 1: Code reading

Question 2: Implement bonded particles

Define the data structure that stores the bonds and provides the functionality to perform update operations and diagnostics on the bonded particles

```
1  class Bonds
2  {
3  public:
4          // setup structure with given stiffness
5          void setup(ParticleVector* pv, const double a_strength, const
               double a_domain_size);
6          // add new twin with length at rest from initial distance
7          void addBond(const int idx1, const int idx2);
8          // get number of bonds (bond indices in [0,size
9          int getNumberOfBonds() const;
10         // add accelerations to particles from spring force for bond i
11         void addAccelerations(const int i) const;
12         // compute pairwise potential energy of bond i
13         double computePairwiseEnergy(const int i) const;
14
15 private:
16         // private structure to store bonds
17         struct Info {
18                 int idx1;
19                 int idx2;
20                 double r0;
21         };
22
23         // storage for bonds
24         vector<Info> bonds;
25         // link to particle vector
26         ParticleVector* particles;
27         // strength for all springs -> k = strength/length_at_rest
28         double strength;
29         // domain size to take care of periodic boundaries
30         double domain_size;
31         double inv_domain_size;
32 };
```

Listing 1: Bonds.h

```
void Bonds::setup(ParticleVector* pv, const double a_strength, const
    double a_domain_size) {
    assert(a_domain_size > 0);
    particles = pv;
    strength = a_strength;
    domain_size = a_domain_size;
    inv_domain_size = 1.0/a_domain_size;
}

void Bonds::addBond(const int idx1, const int idx2) {
    // Extract particles
    const Particle& p1 = (*particles)[idx1];
    const Particle& p2 = (*particles)[idx2];
    // Compute vector connecting particles (xi - xj)
    const double r[3] = {
        p1.x[0] - p2.x[0],
        p1.x[1] - p2.x[1],
        p1.x[2] - p2.x[2]
    };
    // fill info
    Info info;
    info.idx1 = idx1;
    info.idx2 = idx2;
    info.r0 = sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
    bonds.push_back(info);
}

int Bonds::getNumberOfBonds() const {
    return bonds.size();
}

void Bonds::addAccelerations(const int i) const {
    assert(i >= 0 && i < getNumberOfBonds());

    // Get info on bond
    Particle& p1 = (*particles)[bonds[i].idx1];
    Particle& p2 = (*particles)[bonds[i].idx2];
    const double r0 = bonds[i].r0;

    // Compute vector connecting particles (x1 - x2)
    const double _r[3] = {
        p1.x[0] - p2.x[0],
        p1.x[1] - p2.x[1],
        p1.x[2] - p2.x[2]
    };

    // Periodic boundary conditions - take shortest distance
    const double r[3] = {
        _r[0] - domain_size*floor(0.5 + _r[0]*inv_domain_size),
        _r[1] - domain_size*floor(0.5 + _r[1]*inv_domain_size),
        _r[2] - domain_size*floor(0.5 + _r[2]*inv_domain_size)
    };

    // Compute spring forces Fij = -k*(r-r0)/r * (xi - xj) = -Fji
    Force f;
    const double rij = sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
```

```
56      if (rij > 0) {
57          const double k = strength/r0;
58          const double magnitude = k * (r0-rij);
59          const double factor = magnitude/rij;
60          for(int i=0; i<3; i++) f[i] = factor*r[i];
61          p1.addAcceleration(f);
62          p2.addAcceleration(f.opposite());
63      }
64  }
65
66  double Bonds::computePairwiseEnergy(const int i) const {
67      assert(i >= 0 && i < getNumberOfBonds());
68
69      // Get info on bond
70      Particle& p1 = (*particles)[bonds[i].idx1];
71      Particle& p2 = (*particles)[bonds[i].idx2];
72      const double r0 = bonds[i].r0;
73
74      // Compute vector connecting particles (x1 - x2)
75      const double _r[3] = {
76          p1.x[0] - p2.x[0],
77          p1.x[1] - p2.x[1],
78          p1.x[2] - p2.x[2]
79      };
80
81      // Periodic boundary conditions - take shortest distance
82      const double r[3] = {
83          _r[0] - domain_size*floor(0.5 + _r[0]*inv_domain_size),
84          _r[1] - domain_size*floor(0.5 + _r[1]*inv_domain_size),
85          _r[2] - domain_size*floor(0.5 + _r[2]*inv_domain_size)
86      };
87
88      // Compute energy Vij = k/2 * (r-r0)^2
89      const double rij = sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
90      const double k = strength/r0;
91      const double dr = rij - r0;
92      return 0.5*k * dr*dr;
93  }
```

Listing 2: Bonds.cpp

We add a private member of type `Bonds` to the `Simulation` class:

```
1   class Simulation
2   {
3       // Private class variables
4       // ...
5
6       // FILL HERE IF MORE PRIVATE ELEMENTS NEEDED
7       Bonds bonds;
8
9       // Private class methods (helpers)
10      // ...
11  };
```

Listing 3: Simulation.h

Inside the `Simulation.cpp` file:

- `Simulation::Simulation`: initialize your data structure storing storing the bonds and bond length in the constructor of the `Simulation`.

```
Simulation::Simulation(string filename, int a_ncells, double
    a_domain_size, double strength)
: particles(), cell_lists(a_ncells), ncells(a_ncells), domain_size
    (a_domain_size)
{
    // FILL HERE: initialize stiffness for bonds
    bonds.setup(&particles, strength, a_domain_size);

    // Allocate and initialized particles given initial conditions
    _createParticlesFromFile(filename);
}
```

- `Simulation::_createParticlesFromFile`: initialize the bonds where the particle positions are read from file.

```
void Simulation::_createParticlesFromFile(string filename)
{
    // ...

    // Parse input file for bonded particles
    assert((N - Nsingle) % 2 == 0);    // even number of remaining
        particles
    for(int i=Nsingle; i<N; i+=2)
    {
        assert(filestream.good());

        // read in two particles i and i+1
        filestream >> particles[i].x[0];
        filestream >> particles[i].x[1];
        filestream >> particles[i].x[2];
        filestream >> particles[i+1].x[0];
        filestream >> particles[i+1].x[1];
        filestream >> particles[i+1].x[2];

        // FILL HERE
        // create new bond and pass indices of connected particles
        bonds.addBond(i, i+1);
    }
}
```

- `Simulation::_computeAccelerations`: compute and add the accelerations caused by the bonds to the particles.

```
void Simulation::_computeAccelerations()
{
    if (ncells == 0) {
        _computeAccelerationsNoCL();
    } else {
        _resetAccelerations();
        _buildCellList();
        _computeAccelerationsCL();
    }
```

```
    // FILL HERE
    // iterate through all bonds and add the accelerations
    // exerted by the bond forces
    const int nb = bonds.getNumberOfBonds();
    for (int i = 0; i < nb; ++i) bonds.addAccelerations(i);
}
```

- `Simulation::_diag`: add the potential energies off the bonds to the system energy.

```
void Simulation::_diag(size_t stepid, double t)
{
    // ...

    // go through all bonds and update potential energy (H)
    const int nb = bonds.getNumberOfBonds();
    for (int i = 0; i < nb; ++i) H += bonds.computePairwiseEnergy(
        i);

    // ...
}
```

Question 3: Run code

Visualizing the particle locations in VMD, we observe the bonded yellow particles localized in the center of a cloud of white particles. The particle systems starts forming a droplet, a consequence of surface tension taking effect (Fig. 1).We further observe that total energy H and linear momentum $||\mathbf{P}||$ are conserved for this system (Fig. 2).

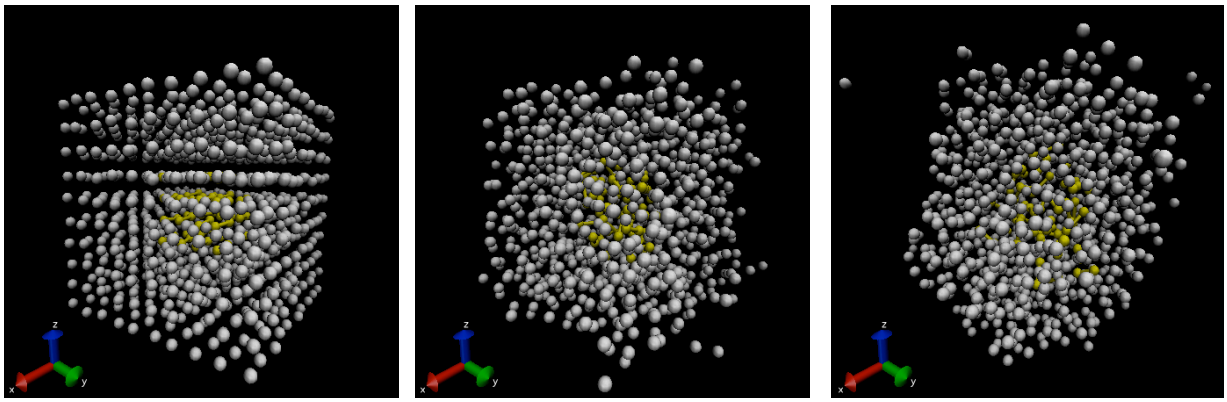

Figure 1: Snapshots of solution at $t = 0$, $t = 1$ and $t = 5$.

Question 4: Timing

Clearly, it is not possible to assume an arbitrary number of cells. The minimal cell size is bounded by the cut-off. Creation of cells with sizes smaller than the interaction cut-off ends up in missing pairwise interaction.

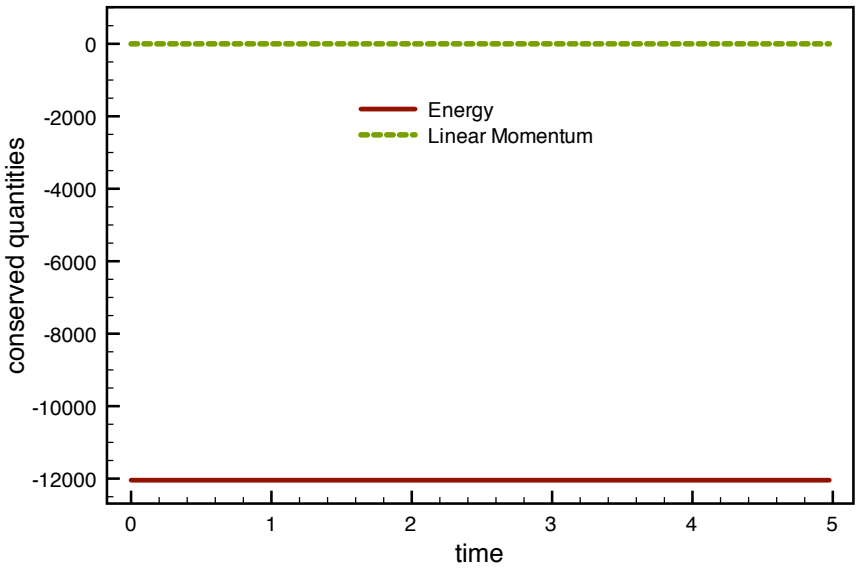

Figure 2: Total energy H and linear momentum $||\mathbf{P}||$.

Figure 3 presents the speed-ups for different cells per dimension over the naive method. The acceleration factor for each execution with respect to the naive algorithm can be computed as:

$$A = \frac{t_{naive}}{t} \ , \tag{1}$$

where t_{naive} is the real time for the execution of the naive algorithm and t is the real time for the implementation we compare to. Note that for too few cells you are slower than the naive method, since the cells are so large that all of the particle pairs must be checked anyway, plus the overhead from the construction of the cell lists.

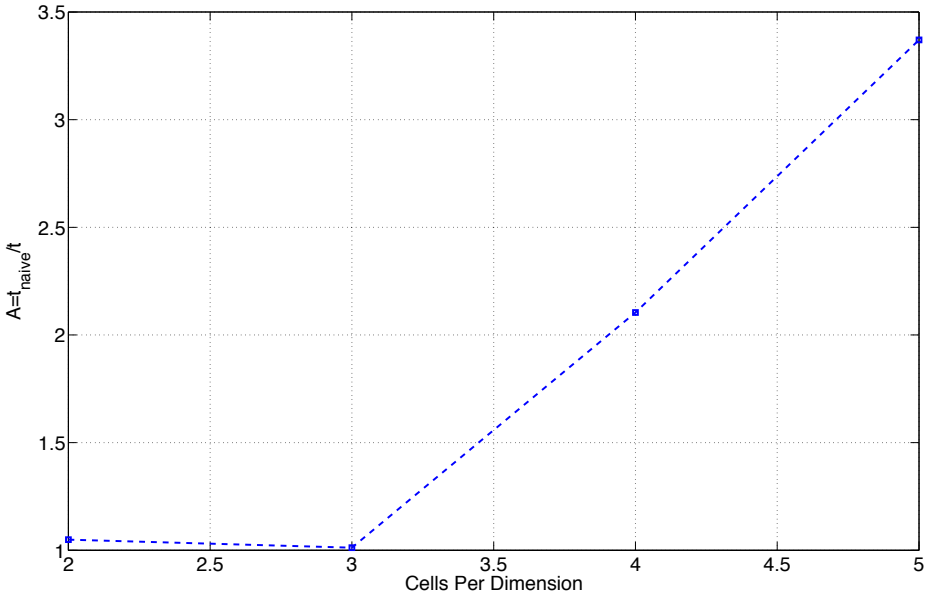

Figure 3: Relative Speed-Up over the naive N^2 implementation with varying cells per dimension.