






 [dilevin](#) / [CSC417-a2-mass-spring-3d](#)

Assignment 2 for CSC2549

 3 stars  1 fork Star Watch ▾ Code Issues 5 Pull requests Actions Projects Wiki Security master ▾

...

**abhimadan** Fix readme typos ...

✓ 9 hours ago ⌚ 33

[View code](#)

README.md

# Physics-Based Animation – Mass-Spring Systems on Three Dimensions

 Build-CSC2549-Assignment-Two passing**To get started:** Clone this repository and all its [submodule](#) dependencies using:

```
git clone --recursive https://github.com/dilevin/CSC2549-a2-mass-spring-3d.git
```

**Do not fork:** Clicking "Fork" will create a *public* repository. If you'd like to use GitHub while you work on your assignment, then mirror this repo as a new *private* repository:<https://stackoverflow.com/questions/10065526/github-how-to-make-a-fork-of-public-repository-private>

## Introduction

The second assignment has two purposes is really where we start doing Computer Graphics. The main focus will be on pushing the ideas we explored in 1D (the previous assignment) to larger scale examples (bunnies!) in 3D.

## Prerequisite installation

On all platforms, we will assume you have installed cmake and a modern c++ compiler on Mac OS X<sup>1</sup>, Linux<sup>2</sup>, or Windows<sup>3</sup>.

We also assume that you have cloned this repository using the `--recursive` flag (if not then issue `git submodule update --init --recursive`).

**Note:** We only officially support these assignments on Ubuntu Linux 18.04 (the OS the teaching labs are running) and OSX 10.13 (the OS I use on my personal laptop). While they *should* work on other operating systems, we make no guarantees.

**All grading of assignments is done on Linux 18.04**

## Layout

All assignments will have a similar directory and file layout:

```
README.md
CMakeLists.txt
main.cpp
include/
  function1.h
  function2.h
  ...
src/
  function1.cpp
  function2.cpp
  ...
data/
  ...
...
```

The `README.md` file will describe the background, contents and tasks of the assignment.

The `CMakeLists.txt` file setups up the cmake build routine for this assignment.

The `main.cpp` file will include the headers in the `include/` directory and link to the functions compiled in the `src/` directory. This file contains the `main` function that is executed when the program is run from the command line.

The `include/` directory contains one file for each function that you will implement as part of the assignment.

The `src/` directory contains *empty implementations* of the functions specified in the `include/` directory. This is where you will implement the parts of the assignment.

The `data/` directory contains *sample* input data for your program. Keep in mind you should create your own test data to verify your program as you write it. It is not necessarily sufficient that your program *only* works on the given sample data.

## Compilation for Debugging

---

This and all following assignments will follow a typical cmake/make build routine. Starting in this directory, issue:

```
mkdir build
cd build
cmake ..
```

If you are using Mac or Linux, then issue:

```
make
```

## Compilation for Testing

---

Compiling the code in the above manner will yield working, but very slow executables. To run the code at full speed, you should compile it in release mode. Starting in the **build directory**, do the following:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

Followed by:

```
make
```

Your code should now run significantly (sometimes as much as ten times) faster.

If you are using Windows, make sure to use `x64`. Running `cmake ..` should have created a Visual Studio solution file called `a2-mass-spring-3d.sln` that you can open and build from there. Building the project will generate an `.exe` file. Building the project will generate an `.exe` file. Move the `.exe` file to the `build` folder so that links to the input mesh aren't broken.

Why don't you try this right now?

## Execution

---

Once built, you can execute the assignment from inside the `build/` using

```
./a2-mass-spring-3d
```

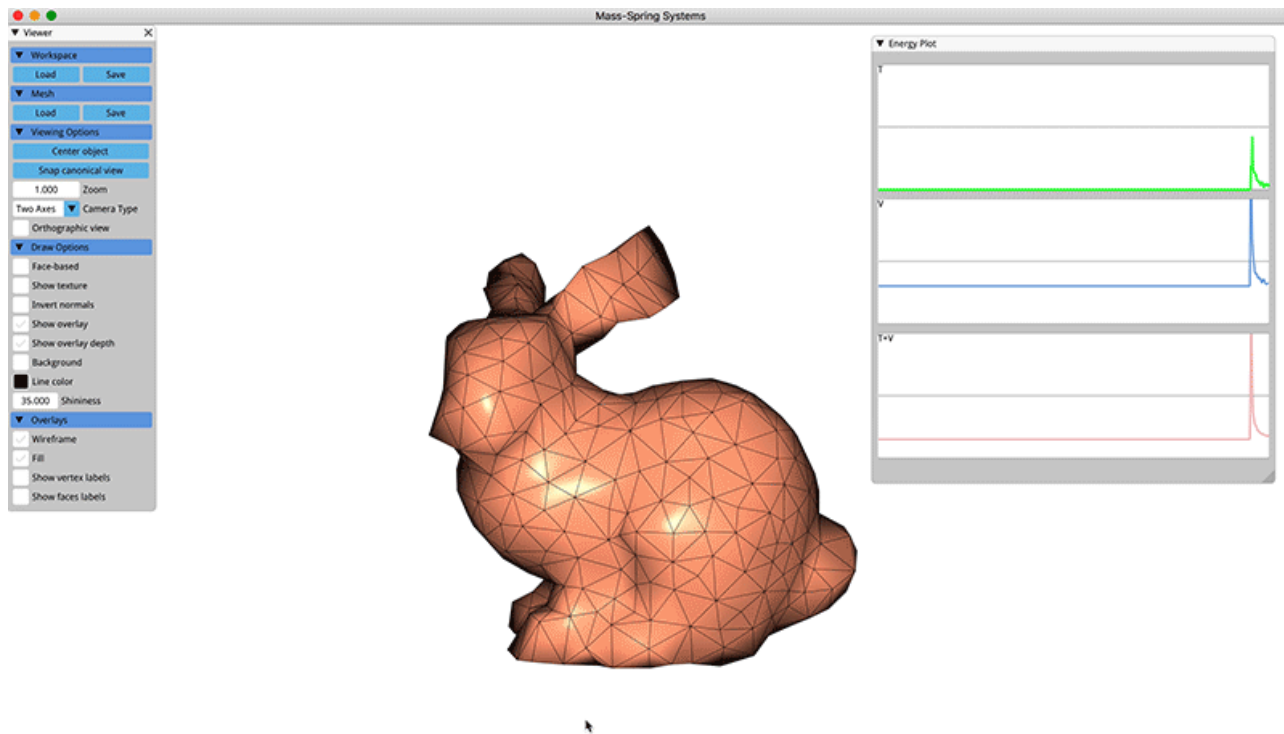
## Background

---

Its happening! We are finally doing some computer graphics ! You can tell because this assignment has a bunny in it (their name is Terry).

The goal of this assignment is to take what you've learned about 1D mass-spring systems and move it to 3D. While much of what we previously covered on variational mechanics and time integration still applies, you will soon see that implementations in 3D become much more complicated. Crucially, this assignment will tackle three concepts which will be important for the remainder of the course

1. the potential energy of a 3D, non-zero rest length spring
2. the notion of assembly. Assembly is a process which builds global linear algebra operations describing the motion of a whole object (bunny) from smaller operators describing the behaviour of individual pieces (in this case springs).
3. The linearly implicit time integrator in 3D -- arguably the most popular time integration scheme in computer graphics.



Github does not render the math in this Markdown. Please look at [README.html](#) to see the equations in their proper form

## A Spring and Two Masses in 3D

The [previous assignment](#) had you implement a simple simulation of a coupled mass and spring in one-dimension. In this assignment we are going to level that up to three dimensions. Specifically, the assignment 2 code reads in a 3D *tetrahedral* mesh (created using [TetWild](#)) and interprets all the edges of this tetrahedral mesh as springs. The user (you!!) will be able to poke and prod this bunny by clicking on vertices, in order to elicit, springy wobbly motion.

In 1D, we used the one-dimensional position of the mass particle as our generalized coordinate. In 3D we will use the obvious extension, which is that the generalized coordinate of each mass particle will be its 3D position. For the simple case of a spring connecting two masses we will define the  $6 \times 1$  vector

$$\mathbf{q} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ x_1 \\ y_1 \\ z_1 \end{bmatrix},$$

where subscripts are used to denote the particle each variable is associated with. Analogous with the 1D case,  $\dot{\mathbf{q}} = \frac{d\mathbf{q}}{dt}$ , where the time derivative is taken component-wise.

The first **important** difference between the 1D and 3D cases, is the nature of our spring itself. in 1D we assumed the spring has *zero rest-length* -- in other words, it has zero potential energy when the distance between the 1D particle, and the wall to which it is affixed, is zero. This won't work for 3D shapes since it will cause them to collapse to a point (like a bunny in a blackhole). So we need to define a new potential energy, one which encourages the spring to maintain its original length.

We are going to define a way to measure the deformation of our spring, relative to its rest, or undeformed, length -- such a measure is called a *strain* measure.

Lets define  $l^0$  to be the original length of of our spring. Then we can define a simple strain measure as  $l - l^0$ . This has all the properties we want in an effective strain measure:

1. It's 0 when the spring has length  $l^0$  (rather than being 0 when length is 0)
2. It's invariant to rigid body transformations (i.e the strain isn't effected by translating or rotating the spring)

Now in keeping with the variational approach to mechanics, we can use this strain to define a potential energy which we can use to construct our equations of motion.

Let's define the potential energy for a single spring to be

$$V = \frac{1}{2}k (l - l^0)^2,$$

where  $k$  is the mechanical stiffness of the spring, like in 1D.

Let's express  $V$  in terms of  $\mathbf{q}$ , which stores the current positions of the end-points of the spring. We are going to do this by introducing some useful matrices that will come in handy later when we need to take derivatives of the potential energy.

We can compute the vector  $\mathbf{dx}$  which points from one end of the spring to the other as

$$\mathbf{dx} = \underbrace{(-\mathbf{I} \quad \mathbf{I})}_B \mathbf{q},$$

so that means the length of the spring can be written as

$$l = \sqrt{\mathbf{q}^T B^T B \mathbf{q}}$$

Thus  $V(\mathbf{q}) = \frac{1}{2}k \left( \sqrt{\mathbf{q}^T B^T B \mathbf{q}} - l^0 \right)^2$  which gives us the potential energy as a function of the generalized coordinates.

The kinetic energy,  $T$  is defined analogously to the 1D case. In 1D, the kinetic energy was  $\frac{1}{2}m\dot{q}^2$  where  $\dot{q}$  was the scalar, 1D velocity. We can interpret this as the magnitude squared of the 1D velocity and thus, a reasonable 3D substitute would be

$$T(\dot{\mathbf{q}}) = \frac{1}{2}m\dot{\mathbf{q}}^T \dot{\mathbf{q}}.$$

Because  $T$  is only a function of the generalized velocity and  $V$  is just a function of the generalized coordinate, you should be able to convince yourself that the equations of motion for a single, 3D spring are

$$m\ddot{\mathbf{q}} = \underbrace{-\frac{\partial V(\mathbf{q})}{\partial \mathbf{q}}}_{\text{Generalized Force}}$$

Now that we've seen the basics of how a single spring can be simulated, let's see what happens when we consider a whole system consisting of many springs!

## Mass-Spring Systems in 3D

Now let's consider the case where we have  $m$  springs and  $n$  particles (like in the case of the tetrahedral mesh representing Terry the bunny).

Whereas in the previous section, our generalized coordinate was  $2 \times 3$  scalar values for two particles, in the general case, it becomes an  $n \times 3$  vector

$$\mathbf{q} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ x_1 \\ y_1 \\ z_1 \\ \vdots \\ x_n \\ y_n \\ z_n \end{bmatrix}.$$

We'll start with the kinetic energy of a mass-spring system because its the same formula as for a single spring ! However, because it will be convenient later on, we will replace the scalar mass  $m$  with an  $n \times n$  diagonal matrix  $M$  (with the diagonal entries set to  $m$ ) which gives us the slightly modified kinetic energy

$$T(\dot{\mathbf{q}}) = \frac{1}{2} \dot{\mathbf{q}}^T M \dot{\mathbf{q}},$$

where  $M$  is called the *mass matrix* or *inertia tensor*. We'll see later in the course that mass matrices can get a lot more complicated than just diagonal.

It's really the force computation that gets effected most by the presence of multiple springs, but its easiest to see this by starting with the potential energy.

The potential energy of our entire mass spring system, is the sum of the potential energy of all the springs

$$V(\mathbf{q}) = \sum_{i=0}^{m-1} V_i(\mathbf{q}_0, \mathbf{q}_1),$$

where  $\mathbf{q}_0$  and  $\mathbf{q}_1$  are the generalized coordinates (the world space positions) of the two end points of the  $i^{th}$  spring.

To make things a little easier conceptually, we are going to introduce *another* set of convenience matrices  $S_i$ . Each  $S_i$  performs the following useful function

$$\begin{bmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \end{bmatrix} = S_i \mathbf{q}.$$

An  $S_i$  matrix ( $6 \times n$ ) selects, from all the particle positions, the end-point positions of the  $i^{th}$  spring. Why is this convenient ? Because we can now rewrite the potential energy of the mass-spring system in terms of  $\mathbf{q}$ .

So we can express the potential energy of the mass-spring system like this

$$V(\mathbf{q}) = \sum_{i=0}^{m-1} V_i(S_i \mathbf{q}),$$

and now we can take derivatives of this in order to compute, say the forces. Let's see what happens if we compute the gradient of this energy with respect to the generalized coordinates. Then we get

$$\frac{\partial V(\mathbf{q})}{\partial \mathbf{q}} = \sum_{i=0}^{m-1} S_i^T \frac{\partial V_i(S_i \mathbf{q})}{\partial \mathbf{q}},$$

and because we know that the energy, in this case, only depends on the end points of the spring ( $\mathbf{q}_0$  and  $\mathbf{q}_1$ ) we can make one final modification:

$$\frac{\partial V(\mathbf{q})}{\partial \mathbf{q}} = \sum_{i=0}^{m-1} S_i^T \frac{\partial V_i(\mathbf{q}_0, \mathbf{q}_1)}{\partial \mathbf{q}_0, \mathbf{q}_1}.$$



Let's unpack this a bit. First, you should be convinced that  $\frac{\partial V_i(\mathbf{q}_0, \mathbf{q}_1)}{\partial \mathbf{q}_0, \mathbf{q}_1}$  is just computing the potential energy gradient of one spring with respect to its end-points, exactly like what we discussed in the previous section. The only mysterious piece is our  $S_i^T$ . That matrix is distributing the local end-point forces into the global force vector, so that they act on the proper mass particles in the system. This summation and distribution of local forces is called **assembly** because it assembles the global force vector from local contributions. Assembly of local quantities comes up all the time in simulation, in fact we are going to see it used one more time in the next section.

**Implementation Note:** You could implement assembly by mirroring the formula above. In other words, instantiate a sparse  $S_i$  matrix for each spring. However that is a waste of space and resources. Instead it's better to implement the *action* of  $S_i^T$  by indexing directly into the global force vector.

## Linearly-Implicit Time Integration

As we discussed in the previous assignment, making the right choice of time integrator is crucial for the stability, appearance and performance of a simulation. In this assignment we are going to implement one of the most famous time integrators in computer graphics, the *Linearly-Implicit Time Integrator*. This integrator is not fully-implicit like backward Euler, but it is efficient to compute and reasonably stable, thus it is a good initial place to start when integrating an elastic system like these mass-springs.

You should be able to convince yourself that, given the potential and kinetic energies above, that the backward Euler update for a 3D, mass-spring system is

$$M\dot{\mathbf{q}}^{t+1} = M\dot{\mathbf{q}}^t + \Delta t \mathbf{f}(\mathbf{q}^{t+1})$$

$$\mathbf{q}^t + \Delta t \dot{\mathbf{q}}^{t+1}$$

Our big problem is estimating the effect of the forces at  $\mathbf{q}^{t+1}$ . Rather than do this fully, the linearly-implicit integrator linearizes the system around the current state via [Taylor expansion](#) and uses this approximation to update the velocity.

We proceed by exploiting the fact that  $\mathbf{q}^{t+1} = \mathbf{q}^t + \Delta t \dot{\mathbf{q}}^{t+1}$  and making the assumption that  $\Delta t$  is sufficiently small (i.e we try and find a  $\Delta t$  so that the simulation doesn't explode). Then we can replace the above update equations with

$$M\dot{\mathbf{q}}^{t+1} = M\dot{\mathbf{q}}^t + \Delta t \mathbf{f}(\mathbf{q}^t) + \Delta t^2 \underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{q}}}_{K} \dot{\mathbf{q}}^{t+1} \mathbf{q}^{t+1}$$

$$\mathbf{q}^t + \Delta t \dot{\mathbf{q}}^{t+1}$$

Importantly we see the appearance of the force gradient,  $K$ , called the *Stiffness matrix* (this will come up again and again :) ) which encodes the change in the forcing behaviour of the object, in a local region around the current state. Because our *generalized forces* are the negative gradient of the potential energy function, the Stiffness matrix is given by

$$K = -\frac{\partial^2 V(\mathbf{q})}{\partial \mathbf{q}^2},$$

or  $K$  is the negative Hessian of the potential energy. **Warning:** missing negative signs are the number one affliction for otherwise happy physics simulators.

Ok, let's do one more rearrangement of these update equations to get them in their final, solvable form:

$$(M - \Delta t^2 K) \dot{\mathbf{q}}^{t+1} = M\dot{\mathbf{q}}^t + \Delta t \mathbf{f}(\mathbf{q}^t) \mathbf{q}^{t+1}$$

$$\mathbf{q}^t + \Delta t \dot{\mathbf{q}}^{t+1}$$

Now we see where linearly-implicit time integration gets its performance, it requires solving a single, sparse, symmetric linear system. **Note:** A fun game is to convince yourself that  $K$  is both sparse (argue from the connectivity of the mass spring system) and symmetric (argue from Newton's 3rd law).

In practice, once we have *assembled* (there's that word again)  $M$  and  $K$  we can form  $(M - \Delta t^2 K)$  and solve this linear system using off-the-shelf tools. For this assignment you can use the *Simplicial LDLT* solver built into the [Eigen](#) library.

Let's end this section by talking about the assembly of  $K$ . In the same way as for the forces, we can directly compute the second derivative of our energy

$$\frac{\partial^2 V(\mathbf{q})}{\partial \mathbf{q}^2} = \sum_{i=0}^{m-1} S_i^T \underbrace{\frac{\partial^2 V_i(\mathbf{q}_0, \mathbf{q}_1)}{\partial (\mathbf{q}_0, \mathbf{q}_1)^2}}_{H_i} S_i.$$

Where  $H_i$  is the local spring energy Hessian (so the Hessian of the  $i^{th}$  spring energy wrt to its end points).

**WARNING:** to assemble the stiffness matrix replace  $H_i$  with  $K_i$ , the per-spring stiffness matrix.

Again, you could implement this using a whole bunch of  $S_i$  matrices, but instead its better to implement it via direct indexing. In particular, in Eigen, you can construct a list of `Eigen::Triplet` objects that can be used to directly initialize a sparse matrix.

**Hint:** a good test for any integrator is to test that your object doesn't move when no forces are exerted on it.

## Fixed Displacement Boundary Conditions

A Mass-Spring system acting purely under gravity is very boring, it just falls straight down (a rigid motion!). What's the point of doing all this work if we don't get a wiggly bunny out of it? In order to generate wiggleness, we need to, at a minimum, fix parts of our object to the environment. These are the *boundary conditions* for the physics simulation. Because we are fixing the position of particles, these are *Dirichlet* boundary conditions. There are a few ways to go about this, but we are going to take a projection approach -- we will express our generalized coordinates using a smaller set of variables.

Given that a fixed vertex, by definition, does not move, it makes sense to choose all other vertex positions as our subspace variables. We can now construct a linear subspace which rebuilds all the positions of our particles from this subspace --

$$\mathbf{q} = P^T \hat{\mathbf{q}} + \mathbf{q}_{fixed}.$$

Let's say we have  $l$  fixed vertices. Then  $\hat{\mathbf{q}}$  is an  $3(n - l)$  length vector which includes the positions of all our non-fixed particles.  $\mathbf{q}_{fixed}$  is a  $3n$  vector which stores a 0's in the positions of our non-fixed particles and the fixed position of fixed particles. It's the matrix  $P$  that performs the magic of stitching these things together.

$P$  is a bit like the selection matrices, the  $S_i$ 's used to explain assembly. It selects out non-fixed vertex positions from  $\hat{\mathbf{q}}$  and places them in the correct position in the global  $\mathbf{q}$  vector. Fixed positions are then filled in with values from  $\mathbf{q}_{fixed}$ . If we take  $\hat{\mathbf{q}}$  as the generalized coordinates of our physical system, we will directly encode these fixed boundary conditions into the simulator.

This is exactly what the assignment 2 simulator does. In practice, it is useful to implement  $P$  as a sparse matrix, and you will get the chance to do just that in assignment 2.

## Groundrules

Implementations of nearly any task you're asked to implemented in this course can be found online. Do not copy these and avoid googling for code; instead, search the internet for explanations. Many topics have relevant wikipedia articles. Use these as references. Always remember to cite any references in your comments.

## Implementation Notes

For this course most functions will be implemented in **.cpp** files. In this assignment the only exception is that time integrators are implemented in **.h** files. This is due to the use of lambda functions to pass force data to the time integration algorithms.

### **src/T\_particle.cpp**

Compute the kinetic energy of a single mass particle.

### **src/V\_gravity\_particle.cpp**

Compute the gravitational potential energy of a single mass particle.

### **src/V\_spring\_particle\_particle.cpp**

Compute the potential energy of a spring which connects two particles.

### **src/dV\_gravity\_particle\_dq.cpp**

Compute the gradient of the gravitational potential energy for a single particle.

### **src/dV\_spring\_particle\_particle\_dq.cpp**

Compute the forces exerted by a single spring on its end-points.

### **src/d2V\_spring\_particle\_particle\_dq2.cpp**

Compute the per-spring hessian of the spring potential energy.

### **src/mass\_matrix\_particles.cpp**

Compute the sparse, diagonal mass matrix that stores the mass of each particle in the mass-spring on its diagonal.

### **src/assemble\_forces.cpp**

Iterate through each spring in the mesh, compute the per-spring forces and assemble the global force vector.

### **src/assemble\_stiffness.cpp**

Iterate through each spring in the mesh, compute the per-spring stiffness matrix and assemble the global, sparse stiffness matrix. To do this, you should construct a list of `Eigen::Triplet` objects and then set your sparse matrix using these [triplets](#).

### **src/fixed\_point\_constraints.cpp**

Compute the sparse projection matrix which projects out fixed point vertices.

### **src/pick\_nearest\_vertices.cpp**

Given a point on the screen (i.e a mouse position clicked by the user), find all vertices within a given radius. **For this method, and this method alone** you are allowed to use the `igl::unproject` and `igl::ray_mesh_intersect` functions. I have provided the appropriate ray shooting function for you to use in the code as well.

### **include/linearly\_implicit\_euler.h**

Implement the linearly implicit Euler time integrator.

## **Releases**

No releases published

## **Packages**

No packages published

## **Contributors** 6



## Languages

● **C++** 66.4%    ● **CSS** 30.0%    ● **CMake** 3.6%