karansher / computer-graphics-mass-spring-systems Public

forked from alecjacobson/computer-graphics-mass-spring-systems

Computer Graphics Assignment – Mass Spring Systems

0 stars 21 forks

Star Watch

Code Issues 3 Pull requests Actions Projects Security Insights

master

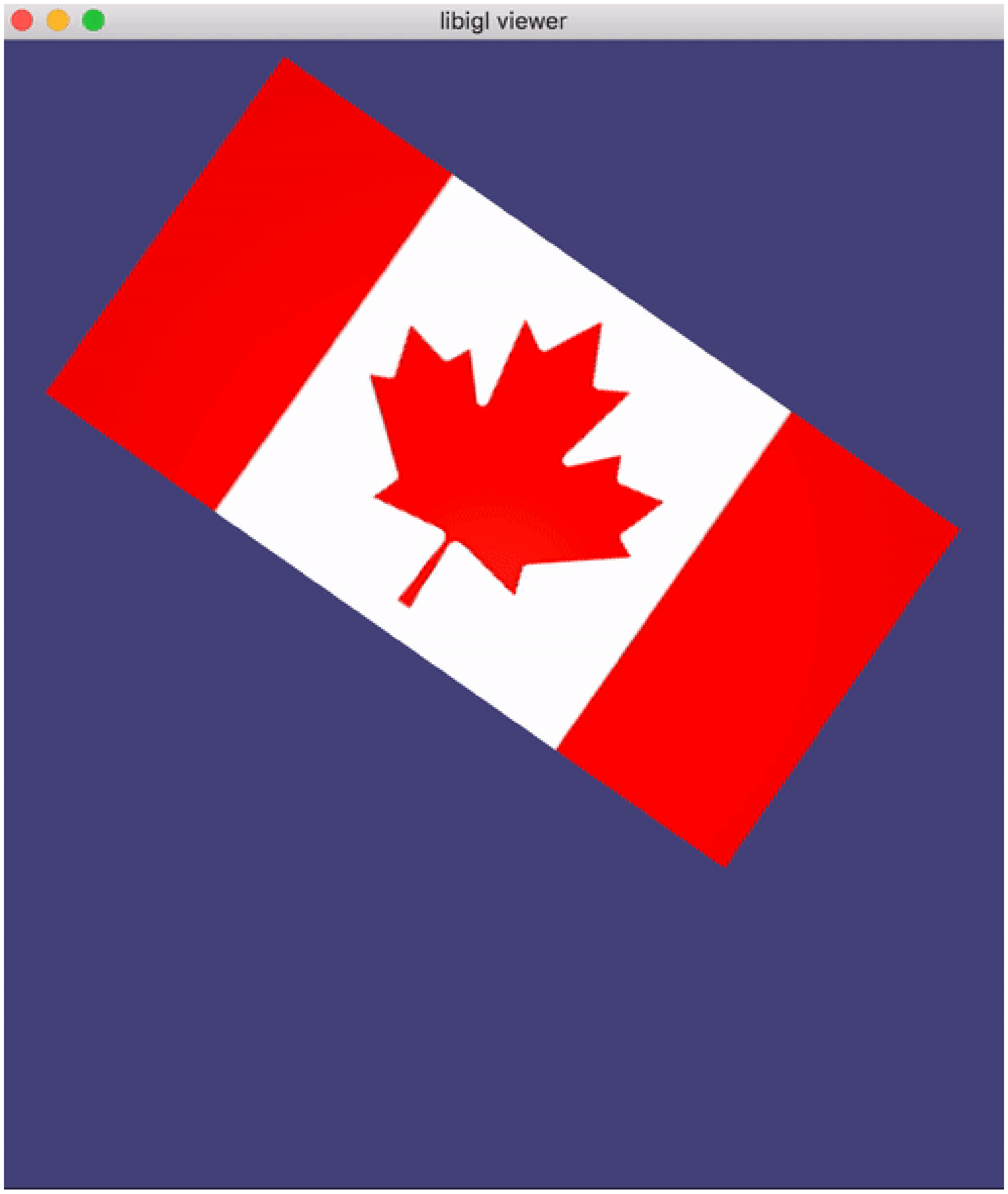
This branch is up to date with alecjacobson/computer-graphics-mass-spring-systems:master.

alecjacobson Update README.md … on Mar 21 23

View code

Computer Graphics – Mass-Spring Systems

> **To get started:** Clone this repository using
>
> ```
> git clone --recursive http://github.com/alecjacobson/computer-graphics-mass-spring-systems.git
> ```



Background

Read Chapter 16.5 of *Fundamentals of Computer Graphics (4th Edition)*.

Read "Fast Simulation of Mass-Spring Systems" [Tiantian Liu et al. 2013]

Mass-Spring Systems

In this assignment we'll consider animating a deformable shape.

We *model* the shape's physical behavior by treating it as a network of point masses and springs. We can think of our shape as a graph where each vertex is a point mass and each edge is a spring.

Given *initial conditions* (each point's starting position and starting velocity, if any) we will create an animation following the laws of physics forward in time. In the real world, physics is deterministic: if we know the current state, we can be sure of what the next state will be (at least at the scales we're considering). This will also be true of our physical simulation.

The law that we start with is Newton's second law, which states that the forces $\mathbf{f} \in \mathbf{R}^3$ acting on a body must equal its mass m times its acceleration $\mathbf{a} \in \mathbf{R}^3$:

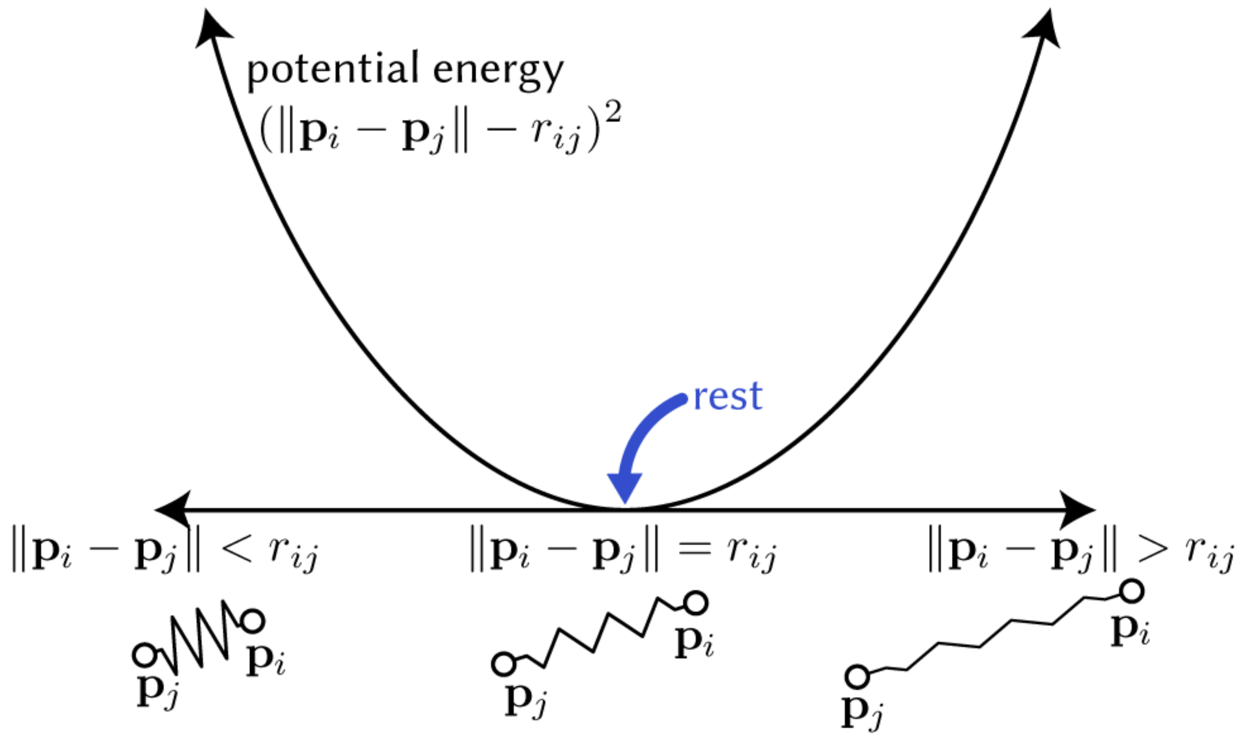
$$\mathbf{f} = m\mathbf{a}.$$

Notice that $\mathbf{f}$ and $\mathbf{a}$ are vectors, each having a magnitude and a direction. We will build our computational simulation by asking for this equation to be true for each point mass in our network. The forces $\mathbf{f}_i$ acting on the i-th point mass are simply the sum of forces coming from any incident spring edge ij and any external force (such as gravity).

Personifying physical objects, we say that they are *at rest* when their potential energy is zero. When the object is *not at rest* then it exerts a force pushing it toward its rest state (elastic force), decreasing its potential energy as fast as possible. The force is the negative gradient of the potential energy.

A simple spring is defined by its stiffness $k > 0$ and *rest* length $r_{ij} \in \mathbf{R}$. Its potential energy measures the squared difference of the current length and the rest length times the stiffness:

$$V(\mathbf{p}_i, \mathbf{p}_j) = \frac{1}{2}k(\|\mathbf{p}_i - \mathbf{p}_j\| - r_{ij})^2.$$


potential energy
$(\|\mathbf{p}_i - \mathbf{p}_j\| - r_{ij})^2$
rest
$\|\mathbf{p}_i - \mathbf{p}_j\| < r_{ij}$
$\|\mathbf{p}_i - \mathbf{p}_j\| = r_{ij}$
$\|\mathbf{p}_i - \mathbf{p}_j\| > r_{ij}$
$\mathbf{p}_j$
$\mathbf{p}_i$

The force exerted by the spring on each mass is the partial derivative of the potential energy V with respect to the corresponding mass position. For example, for $\mathbf{p}_i$ we have

$$\mathbf{f}_{ij} = -\frac{\partial V}{\partial \mathbf{p}_i} \in \mathbf{R}^3.$$

For now, we can postpone expanding $\partial V / \partial \mathbf{p}_i$, and just recognize that it is a 3D vector.

Our problem is to determine *where* all of the mass will be after a small duration in time (Δt).

> **Question:** What is a reasonable choice for the value of Δt ?
>
> **Hint:** 🎞 or 🖥

We'll assume we know the current positions for each mass $\mathbf{p}_i^t \in \mathbf{R}^3$ at the current time (t) and the current velocities $\dot{\mathbf{p}}_i^t = \partial \mathbf{p}_i(t)/\partial t \in \mathbf{R}^3$. When $t = 0$ then we call these the initial conditions of the entire simulation. For $t \geq 0$, we can still think of these values as the initial conditions for the remaining time.

In the real world, the trajectory of an object follows a continuous curve as a function of time. In our simulation, we only need to know the position of each pass at discrete moments in time. We use this to build discrete approximation of the time derivatives (velocities and accelerations) that we encounter. Immediately, we can replace the current velocties $\dot{\mathbf{p}}_i^t$ with a *backward* finite difference of the positions over the small time step:

$$\dot{\mathbf{p}}_i^t = \frac{\mathbf{p}_i^t - \mathbf{p}_i^{t-\Delta t}}{\Delta t}$$

where $\mathbf{p}_i^{t-\Delta t} \in \mathbf{R}^3$ is the position at the *previous* time.

We can also use a *central* finite difference to define the acceleration at time t:

$$\mathbf{a}_i^t = \ddot{\mathbf{p}}_i^t = \frac{\partial^2 \mathbf{p}_i(t)}{\partial t^2} = \frac{\dot{\mathbf{p}}_i^{t+\Delta t} - \dot{\mathbf{p}}_i^t}{\Delta t} = \frac{\mathbf{p}_i^{t+\Delta t} - \mathbf{p}_i^t}{\Delta t} - \frac{\mathbf{p}_i^t - \mathbf{p}_i^{t-\Delta t}}{\Delta t} = \frac{\mathbf{p}_i^{t+\Delta t} - 2\mathbf{p}_i^t + \mathbf{p}^{t-\Delta t}}{\Delta t^2}.$$

This expression mentions our *unknown* variables $\mathbf{p}_i^{t+\Delta t}$ for the first time. We'll soon that based on definition of the potential spring energy above and the acceleration here we can *solve* for the values of these unknown variables.

Time integration as energy optimization

In the equation $\mathbf{f} = m\mathbf{a}$, the acceleration term $\mathbf{a}$ depends *linearly* on the unknowns $\mathbf{p}^{t+\Delta t}$. Unfortunately, even for a simple spring the forces $\mathbf{f} = \partial V / \partial \mathbf{p}^{t+\Delta t}$ depend *non-linearly* on $\mathbf{p}^{t+\Delta t}$. This means we have a *non-linear* system of equations, which can be tricky to satisfy directly.

> **Question:** We've *chosen* to define $\mathbf{f}$ as the forces that implicitly depend on the unknown positions $\mathbf{p}^{t+\Delta t}$ at the end of the time step $t + \Delta t$. What would happen if we defined the forces to explicitly depend on the (known) current positions $\mathbf{p}^t$?

An alternative is to view physics simulation as an optimization problem. We will define an energy that will be minimized by the value of $\mathbf{p}^{t+\Delta t}$ that satisfies $\mathbf{f} = m\mathbf{a}$. The minimizer $\mathbf{p}$ of some function $E(x)$ will satisfy $\partial E / \partial \mathbf{p} = 0$. So we construct an energy E such that $\partial E / \partial \mathbf{p} = \mathbf{f} - m\mathbf{a}$:

$$\mathbf{p}^{t+\Delta t} = \underset{\mathbf{p}}{\operatorname{argmin}} \underbrace{\left(\sum_{ij} \frac{1}{2} k(\|\mathbf{p}_i - \mathbf{p}_j\| - r_{ij})^2\right) - \Delta t^2 \left(\sum_i m_i \left(\frac{\mathbf{p}_i - 2\mathbf{p}_i^t + \mathbf{p}_i^{t-\Delta t}}{\Delta t^2}\right)^2\right) - \left(\sum_i \mathbf{p}_i^\top \mathbf{f}_i^{\text{ext}}\right)}_{E(\mathbf{p})}$$

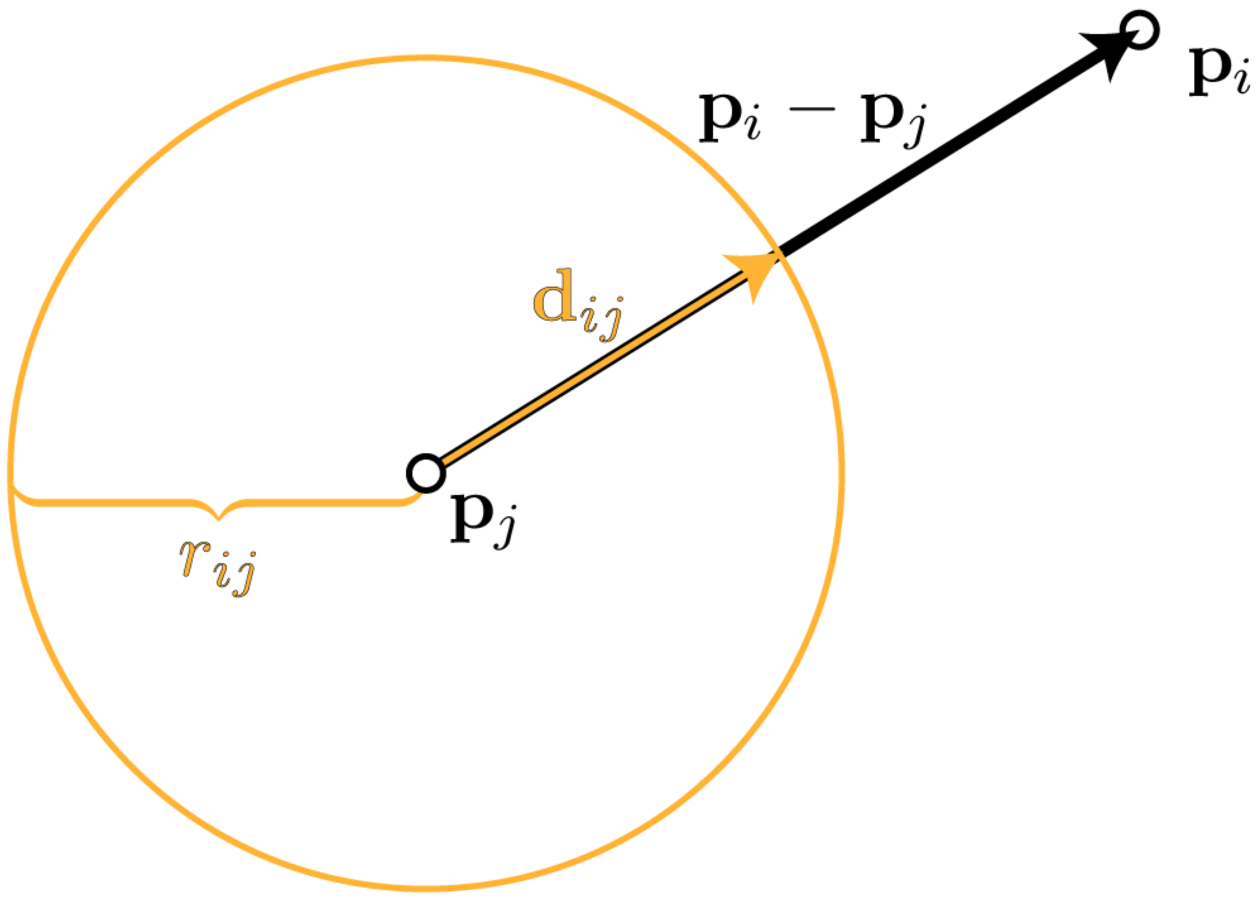
Keen observers will identify that the first term is potential energy and the second term resembles kinetic energy. Intuitively, we can see the first term as trying to return the spring to rest length (elasticity) and the second term as trying to keep masses moving in the same direction.

Because of the $\|\mathbf{p}_i - \mathbf{p}_j\| - r_{ij}$ term, minimizing E is a non-linear optimization problem. The standard approach would be to apply gradient descent (slow), Gauss-Newton method, or Newton's Method (too complicated for this assignment).

In a relatively recent SIGGRAPH paper "Fast Simulation of Mass-Spring Systems", Tiantian Liu et al. made a neat observation that makes designing an algorithm to minimize E quite simple and fast. For each spring ij, they observe that the non-linear energy can be written as a small optimization problem:

$$(\|\mathbf{p}_i - \mathbf{p}_j\| - r_{ij})^2 = \min_{\mathbf{d}_{ij} \in \mathbf{R}^3, \|\mathbf{d}\| = r_{ij}} \|(\mathbf{p}_i - \mathbf{p}_j) - \mathbf{d}_{ij}\|^2.$$

It may seem like we've just created extra work. We took a closed-form expression (left) and replaced it with an optimization problem (right). Yet this optimization problem is small ($\mathbf{d}_{ij}$ is a single 3D vector) and can be easily solved *independently* (and even in parallel) for each spring (i.e., $\mathbf{d}_{ij}$ doesn't depend on $\mathbf{d}_{\ell k}$ etc.). Reading the right-hand side in English it says, find the vector of length r_{ij} that is as close as possible to the current spring vector $\mathbf{p}_i - \mathbf{p}_j$.



Now, suppose we somehow *knew already* the vector $\mathbf{d}_{ij}$ corresponding to the *unknown* optimal solution $\mathbf{p}^{t+\Delta t}$, then treating $\mathbf{d}_{ij}$ as a *constant* we could find the optimal solution by solving the *quadratic* optimization problem:

$$\mathbf{p}^{t+\Delta t} = \mathop{\text{argmin}}_{\mathbf{p}} \underbrace{\left(\sum_{ij} \frac{1}{2} k \|(\mathbf{p}_i - \mathbf{p}_j) - \mathbf{d}_{ij}\|^2\right) - \Delta t^2 \left(\sum_i m_i \left(\frac{\mathbf{p}_i - 2\mathbf{p}_i^t + \mathbf{p}_i^{t-\Delta t}}{\Delta t^2}\right)^2\right) - \left(\sum_i \mathbf{p}_i^\top \mathbf{f}_i^{\text{ext}}\right)}_{\tilde{E}(\mathbf{p})}.$$

The modified energy $\tilde{E}(\mathbf{p})$ is *quadratic* with respect to the unknowns $\mathbf{p}$, therefore the solution is found when we set the first derivative equal to zero:

$$\frac{d\tilde{E}}{d\mathbf{p}} = 0.$$

This leads to a straightforward "local-global" iterative algorithm:

- Step 1 (local): Given current values of $\mathbf{p}$ determine $\mathbf{d}_{ij}$ for each spring.
- Step 2 (global): Given all $\mathbf{d}_{ij}$ vectors, find positions $\mathbf{p}$ that minimize quadratic energy $\tilde{E}$.
- Step 3: if "not satisfied", go to Step 1.

For the purposes of this assignment we will assume that we're "satisfied" after a fixed number of iterations (e.g., 50). More advanced *stopping criteria* could (should) be employed in general.

Matrices

The subtext of this assignment is understanding the computational aspects of large matrices. In the algorithm above, Step 1 is easy and relies on "local" information for each spring.

Step 2 on the otherhand involves all springs simultaneously. Matrices are our convenient notation for representing both the linear operators (e.g., in the equation $\frac{d\tilde{E}}{d\mathbf{p}} = 0$) and the quadratic forms (e.g., in the energy $\tilde{E}$).

Let's begin by being precise about some notation. We will stack up all of the n unknown mass positions $\mathbf{p}_i \in \mathbf{R}^3$ as the rows of a matrix $\mathbf{p} \in \mathbf{R}^{n\times 3}$. We can do the same for the *known* previous time steps' positions $\mathbf{p}^t, \mathbf{p}^{t-\Delta t} \in \mathbf{R}^{n\times 3}$.

We can then express the inertial term using matrices:

$$\Delta t^2 \left(\sum_i m_i \left(\frac{\mathbf{p}_i - 2\mathbf{p}_i^t - \mathbf{p}_i^{t-\Delta t}}{\Delta t^2} \right)^2 \right) = \frac{1}{\Delta t^2} \left(\sum_i \left(\mathbf{p}_i - 2\mathbf{p}_i^t - \mathbf{p}_i^{t-\Delta t}\right)^\top m_i \left(\mathbf{p}_i - 2\mathbf{p}_i^t - \mathbf{p}_i^{t-\Delta t}\right) \right)$$
$$= \frac{1}{\Delta t^2} \operatorname{tr}\left(\mathbf{p} - 2\mathbf{p}^t + \mathbf{p}^{t-\Delta t}\right)^\top \mathbf{M} \left(\mathbf{p} - 2\mathbf{p}^t + \mathbf{p}^{t-\Delta t}\right),$$

where $\operatorname{tr}(\mathbf{X})$ computes the trace of $\mathbf{X}$ (sums up the diagonal entries: $\mathbf{X}_{11} + \mathbf{X}_{22} + \ldots$).

and the entries of the square matrix $\mathbf{M} \in \mathbf{R}^{n \times n}$ are set to

$$\mathbf{M}_{ij} = \begin{cases} m_i & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} .$$

The potential energy term can be similarly written with matrices. We'll start by introducing the *signed incidence* matrix of our mass-psring network of n vertices and m edges $\mathbf{A} \in \mathbf{R}^{m \times n}$. The *rows* of $\mathbf{A}$ correspond to an arbitrary (but fixed) ordering of the edges in the network. In a mass-spring network, the edges are un-oriented in the sense that the spring acts symmetrically on its vertices. For convenience, we'll pick an orientation for edge anyway. For the e-th edge ij, we should be sure to use the same orientation when computing $\mathbf{d}_{ij}$ and for the following entries of $\mathbf{A}$. So, for the e-th row of $\mathbf{A}$ corresponding to edge connecting vertices i and j we'll assign values:

$$\mathbf{A}_{ek} = \begin{cases} +1 & \text{if } k = i \\ -1 & \text{else if } k == j \\ 0 & \text{otherwise.} \end{cases}$$

Using this matrix $\mathbf{A}$ as a linear operator we can compute the spring vectors for each edge:

$$\mathbf{v} = \mathbf{A}\mathbf{p} \leftrightarrow \mathbf{v}_{ij} = \mathbf{p}_i - \mathbf{p}_j.$$

We can now write the modified potential energy of $\tilde{E}$ in matrix form:

$$\left(\sum_{ij} \frac{1}{2} k \|(\mathbf{p}_i - \mathbf{p}_j) - \mathbf{d}_{ij}\|^2 \right) = \frac{k}{2} \operatorname{tr}\left((\mathbf{A}\mathbf{p} - \mathbf{d})^\top (\mathbf{A}\mathbf{p} - \mathbf{d})\right),$$

where we stack the vector $\mathbf{d}_{ij}$ for each edge in the corresponding rows of $\mathbf{d} \in \mathbf{R}^{m \times 3}$.

Combining our two matrix expressions together we can write $\tilde{E}$ entirely in matrix form:

$$\begin{aligned}\tilde{E}(\mathbf{p}) &= \frac{k}{2}\operatorname{tr}\left((\mathbf{A}\mathbf{p}-\mathbf{d})^\top(\mathbf{A}\mathbf{p}-\mathbf{d})\right) + \operatorname{tr}\left(\mathbf{p}-2\mathbf{p}^t+\mathbf{p}^{t-\Delta t}\right)^\top \mathbf{M}\left(\mathbf{p}-2\mathbf{p}^t+\mathbf{p}^{t-\Delta t}\right)\operatorname{tr}\left(\mathbf{p}^\top\mathbf{f}^{\text{ext}}\right)\\ &= \frac{1}{2}\operatorname{tr}\left(\mathbf{p}^\top(k\mathbf{A}^\top\mathbf{A}+\frac{1}{\Delta t^2}\mathbf{M})\mathbf{p}\right) - \operatorname{tr}\left(\mathbf{p}^\top(k\mathbf{A}^\top\mathbf{d}+\frac{1}{\Delta t^2}\mathbf{M}(2\mathbf{p}^t-\mathbf{p}^{t-\Delta t})+\mathbf{f}^{\text{ext}})\right) + \text{ constants }.\end{aligned}$$

> **Question:** Why do we not bother to write out the terms that are constant with respect to $\mathbf{p}$?

We can clean this up by introducing a few auxiliary matrices:

$$\begin{aligned}\mathbf{Q} &:= (k\mathbf{A}^\top\mathbf{A}+\frac{1}{\Delta t^2}\mathbf{M}) \in \mathbf{R}^{n\times n}\\ \mathbf{y} &:= \frac{1}{\Delta t^2}\mathbf{M}(2\mathbf{p}^t-\mathbf{p}^{t-\Delta t})+\mathbf{f}^{\text{ext}} \in \mathbf{R}^{n\times 3}\\ \mathbf{b} &:= k\mathbf{A}^\top\mathbf{d}+\mathbf{y} \in \mathbf{R}^{n\times 3}.\end{aligned}$$

Now our optimization problem is neatly written as:

$$\mathbf{p}^{t+\Delta t} = \underset{\mathbf{p}}{\operatorname{argmin}}\ \frac{1}{2}\operatorname{tr}\left(\mathbf{p}^\top\mathbf{Q}\mathbf{p}\right) - \operatorname{tr}\left(\mathbf{p}^\top\mathbf{b}\right).$$

> **Recall:** The trace operator behaves very nicely when differentiating.
>
> $$\frac{\partial \operatorname{tr}\left(\mathbf{x}^\top\mathbf{y}\right)}{\partial\mathbf{x}} = \mathbf{y}$$
>
> and
>
> $$\frac{\partial \frac{1}{2}\operatorname{tr}\left(\mathbf{x}^\top\mathbf{Y}\mathbf{x}\right)}{\partial\mathbf{x}} = \mathbf{Y}\mathbf{x}$$

Taking a derivative with respect to $\mathbf{p}$ and setting the expression to zero reveals the minimizer of this quadratic energy:

$$\mathbf{Q}\mathbf{p} = \mathbf{b}.$$

Since $\mathbf{Q}$ is a square invertible matrix we can *solve* this system, which we often write as:

$$\mathbf{p} = \mathbf{Q}^{-1}\mathbf{b}.$$

Solving as the *action* of multiplying by a matrix's inverse

From an algorithmic point of view the notation $\mathbf{p} = \mathbf{Q}^{-1}\mathbf{b}$ is misleading. It might suggest first constructing `Qinv = inverse(Q)` and then conducting matrix multiply `p = Qinv * b`. This is almost always a bad idea. Constructing `Qinv` be very expensive $O(n^3)$ and numerically unstable.

Instead, we should think of the *action* of multiplying by the inverse of a matrix as a single "solve" operation: `p = solve(Q,b)` . Some programming languages (such as MATLAB) indicate using operator overloading "matrix division": `p = Q \ b` .

All good matrix libraries (including Eigen) will implement this "solve" action. A very common approach is to compute a factorization of the matrix into a lower triangular matrix times it's transpose:

$$\mathbf{Q} = \mathbf{L}\mathbf{L}^{\top}.$$

Finding this $\mathbf{L}$ matrix takes $O(n^3)$ time in general.

The action of solving against a triangular matrix is simple forward-/back-substitution and takes $O(n^2)$ time. We can conceptually rewrite our system as $\mathbf{Q}\mathbf{p} = \mathbf{b}$ with $\mathbf{L}\mathbf{L}^{\top}\mathbf{p} = \mathbf{b}$.

README.md

positions for). We can split our solve routine into two steps: precomputation done once when the mass-spring system is loaded in and fast substitution at run-time:

```
// Once Q is known
L = precompute_factorization(Q)
// ... each time step
// ... ... each iteration
p = back_substitution(transpose(L),forward_substitution(L,b))
```

Sparse Matrices

For small mass spring systems, $O(n^3)$ at loading time and $O(n^2)$ at runtime may be acceptable. But for even medium sized systems this will become intractable $(n = 1000 \rightarrow n^3 = 1,000,000,000.)$

Fortunately, we can avoid this worst-case behavior by observing a special structure in our matrices. Let's start with the mass matrix $\mathbf{M} \in \mathbf{R}^{n \times n}$. All of the values of this matrix are zero except the diagonal. Storing this as a general matrix we would be storing $n^2 - n$ zeros. Instead, we can acknowlede that this matrix is sparse and store only the non-zeros along the diagonal.

Similarly, the matrix $\mathbf{A}^{m\times n}$ has $2m$ non-zeros (a $+1$ and -1 per edge) and the other $mn-2m$ entries are zero. Furthermore, the result of the product $\mathbf{A}^{\top}\mathbf{A}$ and by extension $\mathbf{Q}\in\mathbf{R}^{n\times n}$ will mostly contain zeros. The number of non-zeros is in fact $O(m+n)$. Large mass-spring systems tend to have $m=O(n)$ edges, so we can happily think of the number of non-zeros as $O(n)$.

We've reduced the storage required from $O(n^2)$ to $O(n)$. What's the catch? General (or "dense") matrices can be easily mapped to memory linearly. For a an arbitrary sparse matrix, we need store additional information to know *where* each non-zero entry is. The most common general approach is to stored a sorted list of values in each column (or row) of the matrix. This is a rather awkward data-structure to manipulate directly. Similar to the pitfalls of bubble sort, inserting values one at a time can be quite slow since we'd have to keep the lists sorted after each operation.

Because of this most sparse matrix libraries require (or prefer) to insert all entries at once and presort non-zeros indices prefer creating the datastructure. Friendly sparse matrix libraries like Eigen, will let us create a list list of (i,j,v) triplets for each non-zero and then insert all values.

So if our dense matrix code looked something like:

```
Afull = zero(m,n)
for each pair i j
  Afull(i,j) += v
end
```

> By convention we use `+=` instead of `=` to allow for repeated (i,j) pairs in the list.

then we can replace this with

```
triplet_list = []
for each pair i j
  triplet_list.append( i, j, v)
end
Asparse = construct_from_triplets( triplet_list )
```

> **Warning:**
>
> Do not attempt to set values of a sparse matrix directly. That is, ***do not*** write:

```
A_sparse(i,j) = v
```

Storing only the non-zero entries means we must rewrite all basic matrix operations including (matrix-vector product, matrix addition, matrix-matrix product, transposition, etc.). This is outside the scope of our assignment and we will use Eigen's `SparseMatrix` class.

Most important to our mass spring system is the *solve action* discussed above. Similar to the dense case, we can precompute a factorization and use substitution at runtime. For our sparse matrix, these steps will be $O(n^{\approx 1.5})$, with substitution faster and nearly $O(n)$.

Pinned Vertices

Subject to the external force of gravity in $\mathbf{f}^{\text{ext}}$ our spring networks will just accelerate downward off the screen.

We can pin down vertices (e.g., those listed in `b`) at their intial positions, by requiring that their corresponding positions values $\mathbf{p}_i$ are always forced to be equal to their initial values $\mathbf{p}_b^{\text{rest}}$:

$$\mathbf{p}_i = \mathbf{p}_i^{\text{rest}} \ \forall i \text{ in pinned vertices.}$$

There are various ways we can introduce this simple linear equality constraint into the energy optimization above. For this assignment, we'll use the easy-to-implement penalty method. We will add an additional quadratic energy term which is minimized when our pinning constraints are satisfied:

$$\frac{w}{2} \sum_{i \text{ in pinned vertices}} \|\mathbf{p}_i - \mathbf{p}_i^{\text{rest}}\|^2,$$

where the w should be set to some large value (e.g., `w=1e10`). We can write this in matrix form as:

$$\frac{w}{2}\operatorname{tr}\left((\mathbf{C}\mathbf{p} - \mathbf{C}\mathbf{p}^{\text{rest}})^\top (\mathbf{C}\mathbf{p} - \mathbf{C}\mathbf{p}^{\text{rest}})\right) = \frac{1}{2}\operatorname{tr}\left(\mathbf{p}^\top (w\mathbf{C}^\top \mathbf{C})\mathbf{p}\right) - \operatorname{tr}\left(\mathbf{p}^\top w\mathbf{C}^\top \mathbf{C}\mathbf{p}^{\text{rest}}\right) + \text{constant}$$

where $\mathbf{C} \in \mathbf{R}^{|\text{pinned}| \times n}$ has one row per pinned vertex with a $+1$ in the corresponding column.

We can add these quadratic and linear coefficients to $\mathbf{Q}$ and $\mathbf{b}$ above correspondingly.

Tasks

White List

- `Eigen::Triplet`

Black List

- `igl::edge_lengths`
- `igl::diag`
- `igl::sparse`
- `igl::massmatrix`
- `.sparseView()` on `Eigen::MatrixXd` types

Write your dense code first. This will be simpler to debug.

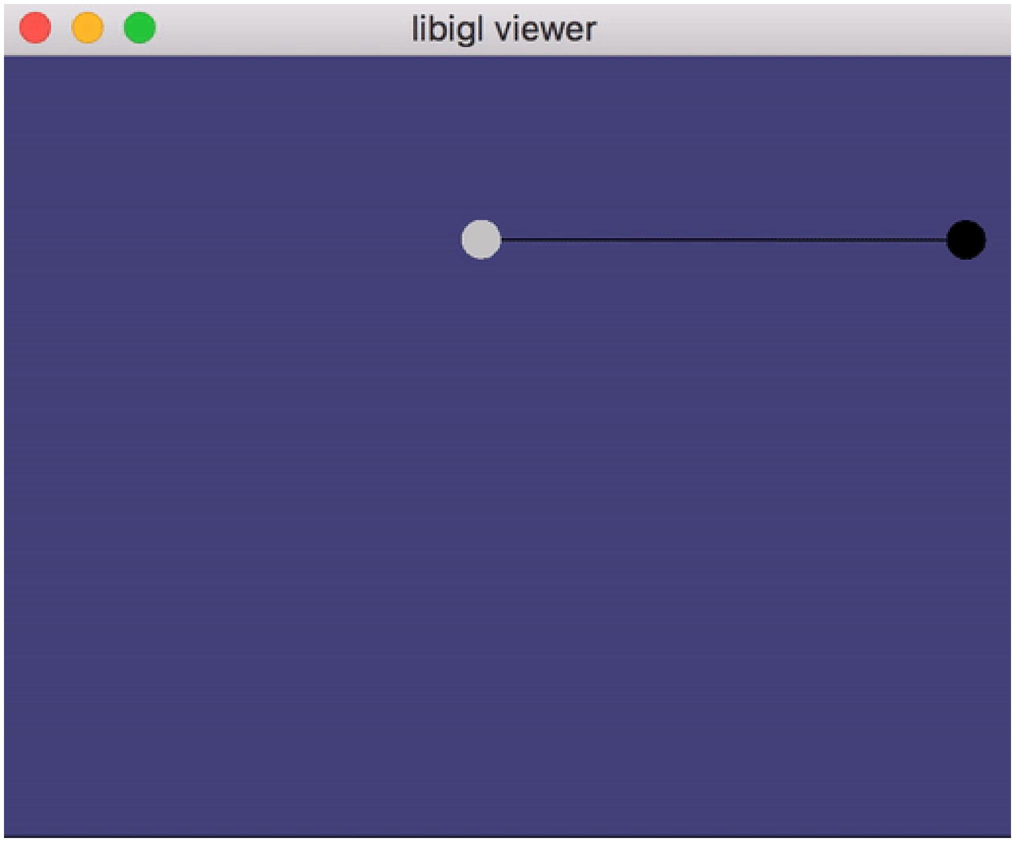
`src/signed_incidence_matrix_dense.cpp`

`src/fast_mass_springs_precomputation_dense.cpp`

`src/fast_mass_springs_step_dense.cpp`

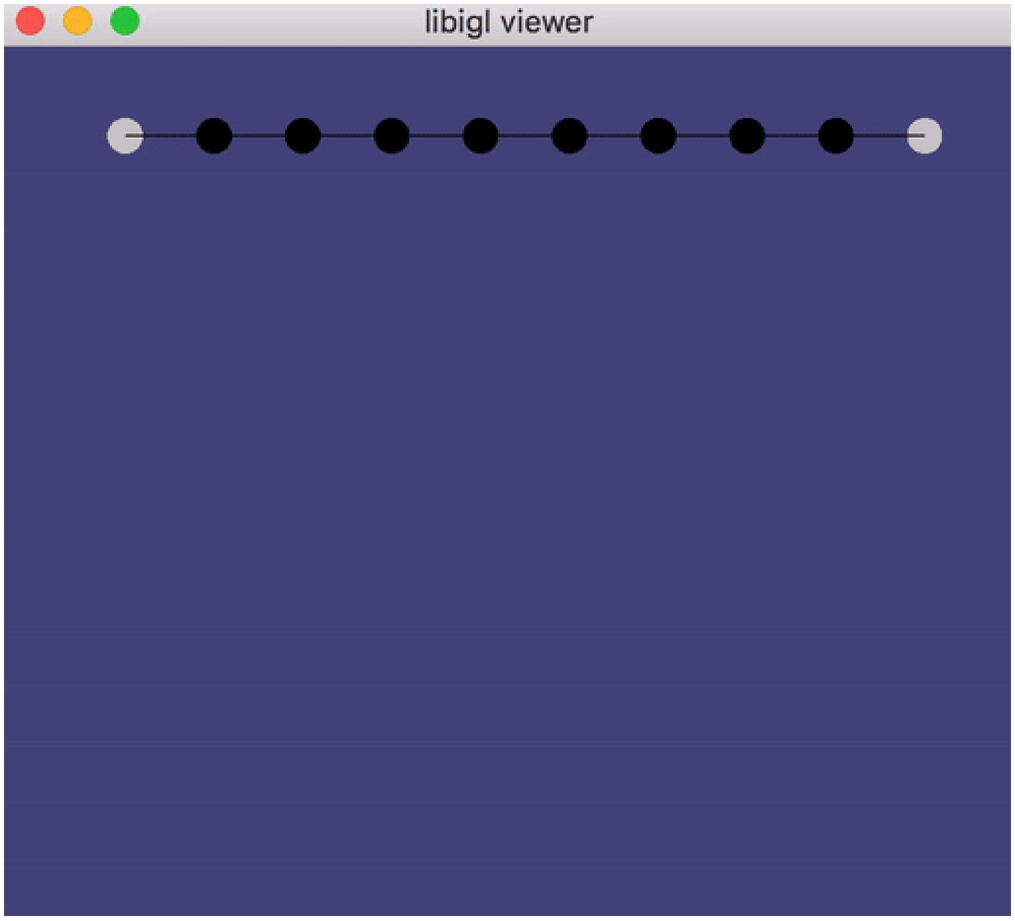
At this point you should be able to run on small examples.

For example, running `./masssprings_dense ../data/single-spring-horizontal.json` should produce a swinging, bouncing spring:

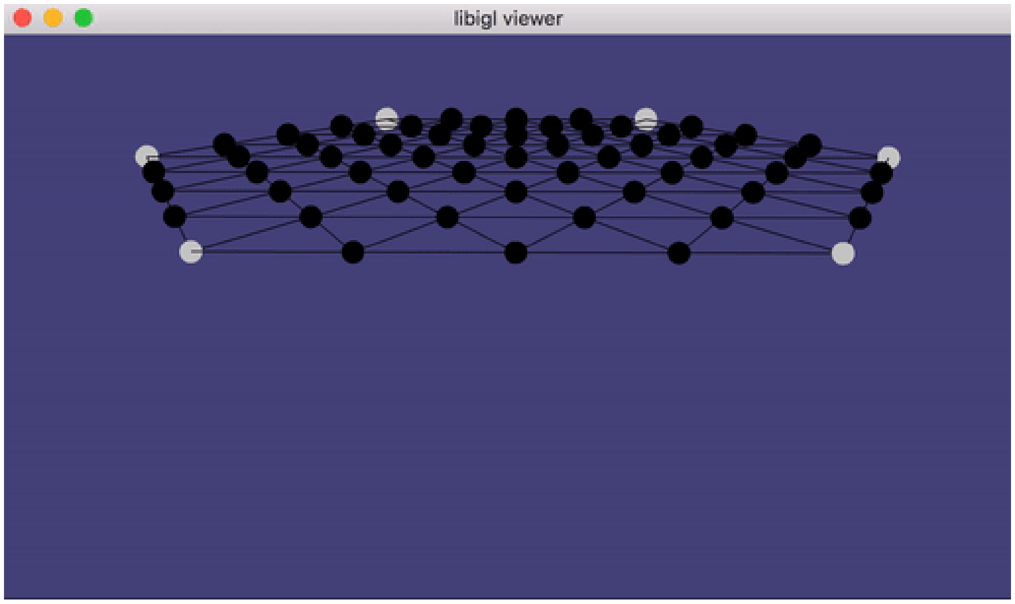


If the single spring example is not working, debug immediately before proceeding to examples with more than one spring.

Running `./masssprings_dense ../data/horizontal-chain.json` will produce a hanging catenary chain:



Running `./masssprings_dense ../data/net.json` will produce a hanging catenary chain:



If you try to run `./masssprings_dense ../data/flag.json` you'll end up waiting a while.

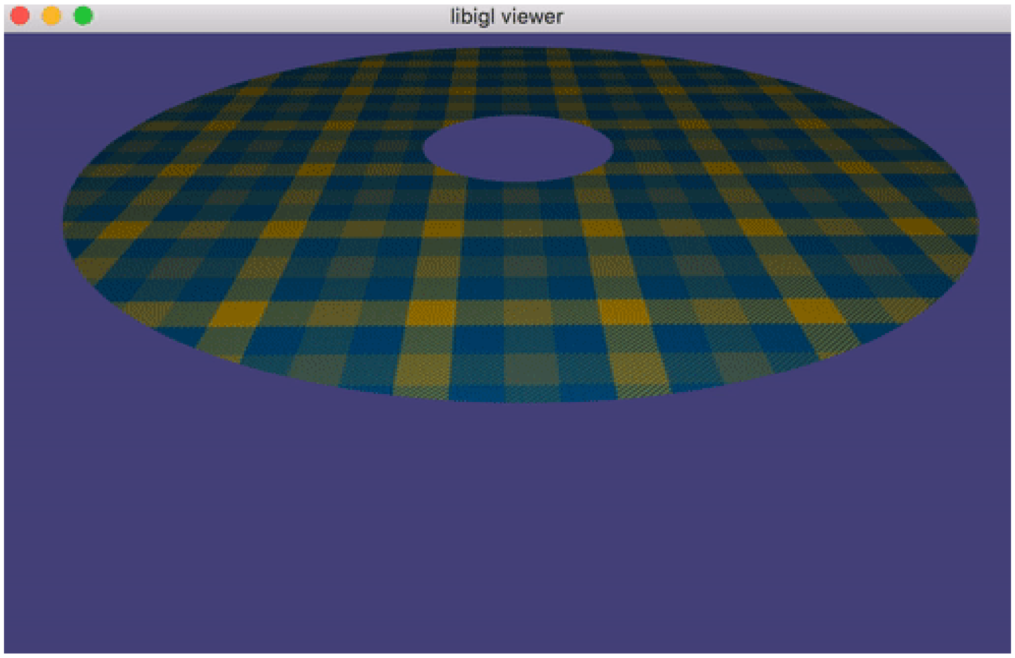
Start your sparse implementations by copying-and-pasting your correct dense code. Remove any dense operations and construct all matrices using triplet lists.

`src/signed_incidence_matrix_sparse.cpp`

`src/fast_mass_springs_precomputation_sparse.cpp`

`src/fast_mass_springs_step_sparse.cpp`

Now you should be able to see more complex examples, such as running `./masssprings_sparse ../data/flag.json` or `./masssprings_sparse ../data/skirt.json`:



> ## Notes for TAs editing the README
>
> This README file is too complex for texify to render. Use readme2tex locally to render the TeX to SVGs.
>
> `python -m readme2tex --output README.md README.tex.md --nocdn`
>
> `sed -i 's/invert_in_darkmode\"/invert_in_darkmode\&sanitize=true\"/g' README.md`

Releases

No releases published

Packages

No packages published

Languages

- ● **C++** 97.0%
- ● **CSS** 2.0%
- ● **Other** 1.0%