CMM-a4: FEM and Soft Manipulation

In this assignment we'll learn about modeling and controlling soft systems.

Reading

Preliminaries

We'll be manipulating a soft bar in 2D. The bar is modeled as a finite element mesh. Call the number of nodes N. Call the vector of nodal positions $\boldsymbol{x}$.

Note that $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_N \end{bmatrix}$, where $\boldsymbol{x}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$ is the Cartesian position of the i-th node.

To access the i-th length 2 segment of $\boldsymbol{x}$ in Eigen we use the code

```
Vector2d x_i = x.segment<2>(2 * i);
```

which is too much typing, and also you're going to forget that second `2` some day and make bugs. Feel free to use my macro instead.

```
Vector2d x_i = seg2(x, i);
```

Simulation

The simulator I'm giving you can also do dynamics, but I'll just summarize how it does statics. The extension to dynamics is actually very simple, and I include at the very end of the assignment as optional reading if you're interested.

Our overall system is a simulation of a soft bar glued to handles. This includes finite elements (triangles) to model the bar itself, and additional spring terms to model the interface between the bar and the handles. We pack the handles' translations and rotations into a vector of control parameters $\boldsymbol{u}$.

For now consider the control parameters as constant. Our goal is to find a statically stable position of the mesh $\boldsymbol{x}$. The total energy of this system is of the form

$$E = E(\boldsymbol{x}; \boldsymbol{u})$$

That is to say, the energy maps the mesh's position $\boldsymbol{x}$ to a scalar energy E. I've put $\boldsymbol{u}$ after the semi-colon to indicate that it is being held constant. You could also leave it out entirely and write $E = E(\boldsymbol{x})$.

We can stack the force on each node into a vector, and consider the *nodal forces*

$$\boldsymbol{F} = \begin{bmatrix} \boldsymbol{F}_1 \\ \vdots \\ \boldsymbol{F}_N \end{bmatrix},$$

where $\boldsymbol{F}_i$ is the force acting on the i-th node. Like energy, the forces are a function of deformed position $\boldsymbol{F} = \boldsymbol{F}(\boldsymbol{x})$, and can be derived from energy[1] using the equation

$$\boldsymbol{F} = -\frac{\partial E}{\partial \boldsymbol{x}}. \tag{1}$$

A statically stable position $\boldsymbol{x}^*$ of the mesh satisfies the equation

$$\boldsymbol{F}(\boldsymbol{x}^*) = \boldsymbol{0}.$$

We could solve this equation using e.g. Newton's method for root finding. An equivalent approach[2] (that tends to be popular in computer graphics) is to solve

$$\boldsymbol{x}^*(\boldsymbol{u}) = \arg\min_{\boldsymbol{x}} \boldsymbol{E}(\boldsymbol{x}; \boldsymbol{u}), \tag{2}$$

using e.g. Newton's method for minimization with line search. This is how the simulator I've given you works!–Just set $\boldsymbol{u}$, and minimize the total energy of the system.

Here is a modified version of the high-level code we use to solve statics (your version works for both statics and dynamics, which makes it look a bit more complicated).

```
// 1) Build E(x; u)
E = buildEnergyFunction();
updatePinAnchorsAccordingToHandles(u);
// 2) Solve x(u) = arg min E(x; u);
VectorXd x_solve = x_prev; // warm-start solve @ x_prev
NewtonFunctionMinimizer solver;
solver.minimize(&E, x_solve);
```

That was the general formulation. Now just a couple specifics for your implementation. The total energy of the system is

$$E(\boldsymbol{x}; \boldsymbol{u}) = E_{\text{FEM}}(\boldsymbol{x}) + E_{\text{pins}}(\boldsymbol{x}; \boldsymbol{u}),$$

where E_{FEM} sums over the triangle energies, and E_{pins} sums over the pin energies.

[1] The idea of the proof is as follows (we will drop transposes, rigorous notions of path integrals, etc.) The work done by a conservative force $\bar{\boldsymbol{F}}$ on a point with position $\boldsymbol{x}$ is given by $W = \int \bar{\boldsymbol{F}} d\boldsymbol{x}$. Differentiating we obtain that $\frac{dW}{d\boldsymbol{x}} = \bar{\boldsymbol{F}}$. In our case the work done on the system stores positive deformation energy E, and the nodal forces $\boldsymbol{F} = \bar{\boldsymbol{F}}$ are internal forces. So we obtain $\frac{dE}{d\boldsymbol{x}} = -\boldsymbol{F} \implies$ Equation (1).

[2] Proof: As a minimizer of E, $\boldsymbol{x}^*$ necessarily has $\frac{\partial E}{\partial \boldsymbol{x}}(\boldsymbol{x}^*) = \boldsymbol{0} \implies \boldsymbol{F}(\boldsymbol{x}^*) = \boldsymbol{0}$ by Equation (1).

You went over FEM energies in class, and I'll just summarize the energy of a pin (which we could also call a *zero length spring*).

A pin has *deformed position* $\boldsymbol{p} = \boldsymbol{p}(\boldsymbol{x})$. If a pin is attached to the i-th node, then $\boldsymbol{p}(\boldsymbol{x}) = \boldsymbol{x}_i$. A pin has *anchor position* $\boldsymbol{p}' = \boldsymbol{p}'(\boldsymbol{u})$. By writing it as a function of $\boldsymbol{u}$ we are indicating the fact that the way we control the mesh is by moving the pins' anchor positions. This is what the handles are actually *doing* to the simulation.

The energy of a pin is $E_{\text{pin}} = \frac{1}{2}k||\boldsymbol{p}(\boldsymbol{x}) - \boldsymbol{p}'||^2$.

Control

We control the mesh with two handles (denoted by the small arrows). You can click and drag to specify a handle's position and rotation (to specify rotation click on the point of the handle). Retina users may have problems with this, please let me know.

We call the vector of control signals

$$\boldsymbol{u} = \begin{bmatrix} x_L & y_L & x_R & y_R & \theta_L & \theta_R \end{bmatrix}^T,$$

where e.g. x_L is the translation of the left handle in the x-direction, and θ_R is the rotation of the right handle.

IK Objective: Given target position $\boldsymbol{x}'$, find optimal control $\boldsymbol{u}^*$ that brings the deformed position $\boldsymbol{x}(\boldsymbol{u}^*)$ as close as possible to the target position.

We can encode this notion into an objective

$$\mathcal{O}(\boldsymbol{u}) = \frac{1}{2}\left\|\boldsymbol{x}^*(\boldsymbol{u}) - \boldsymbol{x}'\right\|^2,$$

which penalizes deviation between $\boldsymbol{x}^*(\boldsymbol{u})$ and $\boldsymbol{x}'$. We minimize $\mathcal{O}$ to find to obtain optimal control inputs $\boldsymbol{u}^*$.

```
auto u = sim.stackControl(...); // warm-start solve
GradientDescentLineSearch minimizer(...);
minimizer.minimize(&O, u);
```

NOTE: In practice we do not specify a target position for the entire mesh, but rather for only a small number of *feature points*. To see how this is done see `ManipObjective.get_O(...)`.

(3 pt) FEM (fem-app.cpp)

The mesh should fall far, far away if you run things now. Please implement the energy model of the pins which secures the mesh to the handles.

1. **(0.5 pt)** Implement `Pin.energy`.
2. **(0.5 pt)** Implement `Pin.gradient`.
3. **(0.5 pt)** Implement `Pin.hessian`.

Things should look a lot better now, but if you drag the mesh around enough you'll notice a pretty big problem. The energy density of a neo-Hookean triangle element is

$$\Psi(\boldsymbol{x}) = \frac{\mu}{2}\text{tr}(\mathcal{F}^\mathsf{T}\mathcal{F} - I) - \mu \ln J + \frac{\kappa}{2}(\ln J)^2,$$

but in the code we see it has been set to zero (by me). You could implement the above function as is, but since we're already here let's do a bit of math math.

4. **(0.5 pt)** Assuming $\mathcal{F}$ is a 2×2 matrix, prove $\text{tr}(\mathcal{F}^\mathsf{T}\mathcal{F} - I) = ||\mathcal{F}||_F^2 - 2$, where $||\cdot||_F$ denotes the Frobenius norm. Write your proof in the README .
5. **(1 pt)** Calculate the energy density in `Triangle.energy`. For full points you **must** use `F.squaredNorm()` to help you calculate the first term.

The simulation should now behave well. To test your implementation you can press `sim.CHECK_dEdx` and `sim.CHECK_d2Edx2`. These will test $\frac{\partial E}{\partial \boldsymbol{x}}$ (the gradient) and $\frac{\partial^2 E}{\partial \boldsymbol{x}^2}$ (the Hessian) with finite differences respectively. Please note that these functions test the gradient and Hessian **at the current mesh position** $\boldsymbol{x}(\boldsymbol{u})$.

(6 pt) Soft manipulation (manip-app.cpp)

In this section we will build up to implementing the full analytic gradient $\frac{d\mathcal{O}}{d\boldsymbol{u}}$ in `ManipObjective.get_DODu(...)`.

The IK objective has the form $\mathcal{O} = \mathcal{O}(\boldsymbol{x}(\boldsymbol{u}))$. I am dropping the $*$'s in this section for readability, but please note that by $\boldsymbol{x}(\boldsymbol{u})$ we mean the statically stable position of the mesh we get by solving Equation (2).

We expand using the chain rule.

$$\frac{d\mathcal{O}}{d\boldsymbol{u}} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{u}}^T \frac{\partial \mathcal{O}}{\partial \boldsymbol{x}}$$

6. **(0.5 pt)** Implement `if (step == FD_BIG_PIECES){...}`.

We can compute $\frac{\partial \mathcal{O}}{\partial \boldsymbol{x}}$ analytically.

7. **(0.5 pt)** Implement `if (step == ANALYTIC_dOdx){...}`.

The relationship $\boldsymbol{x}(\boldsymbol{u})$ is not analytic. Rather it is the result of performing the minimization in Equation (2), so we cannot differentiate $\boldsymbol{x}(\boldsymbol{u})$ directly. Instead we leverage the fact that $\boldsymbol{x}(\boldsymbol{u})$ solves statics, i.e. that $\boldsymbol{F}(\boldsymbol{x}(\boldsymbol{u}); \boldsymbol{u}) = \boldsymbol{0}$.

$$\implies \frac{d\boldsymbol{F}}{d\boldsymbol{u}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{u}} + \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{u}} = \boldsymbol{0}.$$

You can solve this system for $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{u}}$.

8. **(0.5 pt)** Implement `if (step == SOLVE_dxdu){...}`.

For computing $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}$...

HINT: Recall Equation (1).

HINT: `get_H(u, x)` computes the Hessian $\frac{d^2E}{d\boldsymbol{x}^2}$

For solving the system...

HINT: `solve_AX_EQUALS_B(...)` solves a matrix system.

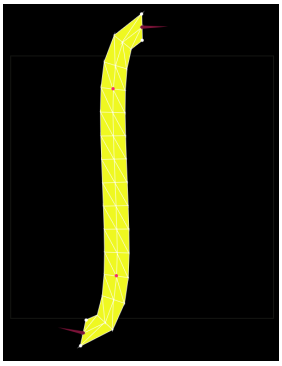
We can compute $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}$ analytically.

9. **(0.5 pt)** Implement `if (step == ANALYTIC_dFdu){...}`.

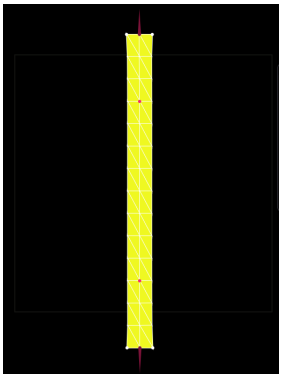
NOTE: This question is a bit of a challenge. You'll have to look around the code and improvise. Once you're done, that's it! The whole gradient computed analytically.

Challenge problem: regularizers

There is something strange about our controller. When we specify arrange the two targets in a vertical line, the optimizer finds this solution



as opposed to this one



which we would likely expect/prefer.

In your README please answer the following questions.

10. **(1 pt)** Why does the optimizer find the first solution?
11. **(1.5 pt)** Propose a regularizer (sub-objective) that we can add to $\mathcal{O}$ so that the optimizer finds the second solution instead.

Finally in the code...

12. **(1.5 pt)** Implement that regularizer inside the `if (QUESTION_12){...}` block of `ManipObjective.get_O(...)`.

NOTE: Feel free to use `step = FD_O_of_u;` for this last part. This estimates the gradient by finite differencing across `evaluate(const VectorXd &u)`, meaning you don't have to worry about implementing the gradient of your regularizer.

Optional reading: dynamics

Implicit integration methods are nice and stable, and we like them for FEM.

We simulate our robot forward in time using implicit Euler update rule

$$\begin{cases} \boldsymbol{x}_k = \boldsymbol{x}_{k-1} + h\boldsymbol{v}_k \\ \boldsymbol{v}_k = \boldsymbol{v}_{k-1} + h\boldsymbol{a}_k, \end{cases} \tag{3}$$

from which we can derive the discretized acceleration

$$\boldsymbol{a}_k = \frac{\boldsymbol{x}_k - 2\boldsymbol{x}_{k-1} + \boldsymbol{x}_{k-2}}{h^2}, . \tag{4}$$

Integrating physics forward from time t_{k-1} to time t_k requires solving (a discretized version of) Newton's second law

$$\boldsymbol{F}_k = \boldsymbol{M}\boldsymbol{a}_k, \tag{5}$$

where $\boldsymbol{F}_k = \boldsymbol{F}_k(\boldsymbol{x}_k)$, $\boldsymbol{a}_k = \boldsymbol{a}_k(\boldsymbol{x}_k)$, and $\boldsymbol{M}$ is the constant mass matrix. The previous two positions $\boldsymbol{x}_{k-1}$ and $\boldsymbol{x}_{k-2}$ are also constant, because *they have already happened*.

As with statics, we can either solve Equation (5) as a root finding problem, or perform an equivalent minimization. A cute observation is that the quantity in between the big parentheses below

$$\boldsymbol{x}_k^* = \arg\min_{\boldsymbol{x}_k} \left(E + \frac{h^2}{2}\boldsymbol{a}_k^T \mathbf{M} \boldsymbol{a}_k \right) \tag{6}$$

has the gradient $-\boldsymbol{F}_k + \boldsymbol{M}\boldsymbol{a}_k$, implying $\boldsymbol{F}(\boldsymbol{x}_k^*) = \boldsymbol{M}\boldsymbol{a}_k(\boldsymbol{x}_k^*)$.

This isn't particularly fundamental, but it lets us use basically the same code for solving statics and dynamics which is nice :) To solve dynamics we just tack on the extra "inertial term" $\frac{h^2}{2}\boldsymbol{a}_k^T \mathbf{M} \boldsymbol{a}_k$.

```
E += .5 * pow(h, 2) * a.transpose() * M.asDiagonal() * a;
```