MAE 259B Group 2 Progress Report

Siyuan Chen, Xiangzhou Kong, Long Chen

05/09/2018

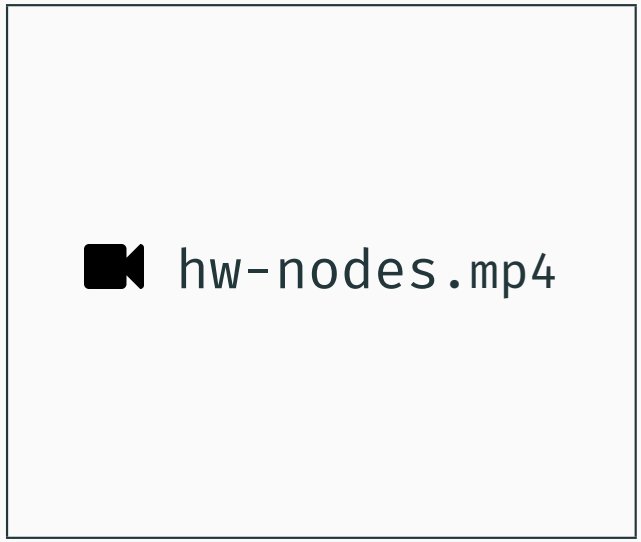
What we did - Starting point

Start from the homework code

Adapted from the MATLAB sample code, translated into Python, with minor changes and optimizations.

Build utilities

Command line interface, 3D visualization tool, code snapshot tool, etc..



Not all PDF viewers can play videos. If not playable here, the videos can be found at `https://github.com/kmxz/mae259b/tree/master/mid-presentation/video`.

What we did - Performance optimization

Profiling shows 90% of time is spent on calculating *F* and *J*.

```
def gradEbAndHessEb(xkm1, ykm1, xk, yk, xkp1, ykp1, φk0):
    itm1 = 2 * tan(0.5 * φk0)
    itm2 = ((-xk + xkp1) * (xk - xkm1) + (-yk + ykp1) * (yk - ykm1))
    itm3 = ((-xk + xkp1) * (yk - ykm1) - (xk - xkm1) * (-yk + ykp1))
    itm4 = itm3 / itm2 ** 2
    itm5 = tan(0.5 * atan(itm3 / itm2))
    itm6 = (1 + itm3 ** 2 / itm2 ** 2)
    itm7 = ((ykm1 - ykp1) / itm2 + itm3 * (2 * xk - xkm1 - xkp1) / itm2 ** 2)
    itm8 = ((-xkm1 + xkp1) / itm2 + itm3 * (2 * yk - ykm1 - ykp1) / itm2 ** 2)
    itm9 = ((-xk + xkm1) / itm2 + (-yk + ykm1) * itm4)
    itm10 = ((-xk + xkm1) * itm4 + (yk - ykm1) / itm2)
    itm11 = (-itm1 + 2 * itm5) * (itm5 ** 2 + 1) * itm5 / itm6 ** 2
    itm12 = (-itm1 + 2 * itm5) * (itm5 ** 2 + 1) / itm6
    itm13 = itm12 / itm6
    itm14 = itm3 ** 2 / itm2 ** 3

    F = np.empty(6)
    F[0] = 2 * ((-xk + xkp1) * itm4 + (-yk + ykp1) / itm2) * itm12
    F[1] = 2 * ((xk - xkp1) / itm2 + (-yk + ykp1) * itm4) * itm12
    F[2] = 2 * itm7 * itm12
    F[3] = 2 * itm8 * itm12
    F[4] = 2 * itm10 * itm12
    F[5] = 2 * itm9 * itm12

    J11 = 2 * ((-2 * xk + 2 * xkp1) * (-xk + xkp1) * itm3 / itm2 ** 3 + 2 * (-xk + xkp1) * (-yk +
ykp1) / itm2 ** 2) * itm12 + 2 * (-(-2 * xk + 2 * xkp1) * itm14 - (-2 * yk + 2 * ykp1) * itm4) *
((-xk + xkp1) * itm4 + (-yk + ykp1) / itm2) * itm13 + 2 * ((-xk + xkp1) * itm4 + (-yk + ykp1) /
itm2) ** 2 * itm11 + 2 * ((-xk + xkp1) * itm4 + (-yk + ykp1) / itm2) ** 2 * (itm5 ** 2 + 1) ** 2
/ itm6 ** 2
    J12 = 2 * (-itm1 + 2 * itm5) * (itm5 ** 2 + 1) * ((-xk + xkp1) * (xk - xkp1) / itm2 ** 2 +
(-xk + xkp1) * (-2 * yk + 2 * ykp1) * itm3 / itm2 ** 3 + (-yk + ykp1) ** 2 / itm2 ** 2) / itm6 +
2 * ((xk - xkp1) / itm2 + (-yk + ykp1) * itm4) * ((-xk + xkp1) * itm4 + (-yk + ykp1) / itm2) *
itm11 + 2 * ((xk - xkp1) / itm2 + (-yk + ykp1) * itm4) * ((-xk + xkp1) * itm4 + (-yk + ykp1) /
itm2) * (itm5 ** 2 + 1) ** 2 / itm6 ** 2 + 2 * ((-xk + xkp1) * itm4 + (-yk + ykp1) / itm2) *
(-(2 * xk - 2 * xkp1) * itm4 - (-2 * yk + 2 * ykp1) * itm14) * itm13
    J13 = 2 * (-itm1 + 2 * itm5) * (itm5 ** 2 + 1) * ((-xk + xkp1) * (ykm1 - ykp1) / itm2 ** 2 +
(-xk + xkp1) * itm3 * (4 * xk - 2 * xkm1 - 2 * xkp1) / itm2 ** 3 + (-yk + ykp1) * (2 * xk - xkm1
- xkp1) / itm2 ** 2 - itm4) / itm6 + 2 * itm7 * ((-xk + xkp1) * itm4 + (-yk + ykp1) / itm2) *
```

30.6 s

⇓

7.6 s

4× faster

What we did - Natural curvature

When calculating bending energy, replace $\frac{1}{2}EI(\phi_i)^2$ with $\frac{1}{2}EI(\phi_i - \phi_{k0})^2$.

The formulas for calculating F and J need to be changed (do differentiation again).

To verify the result:

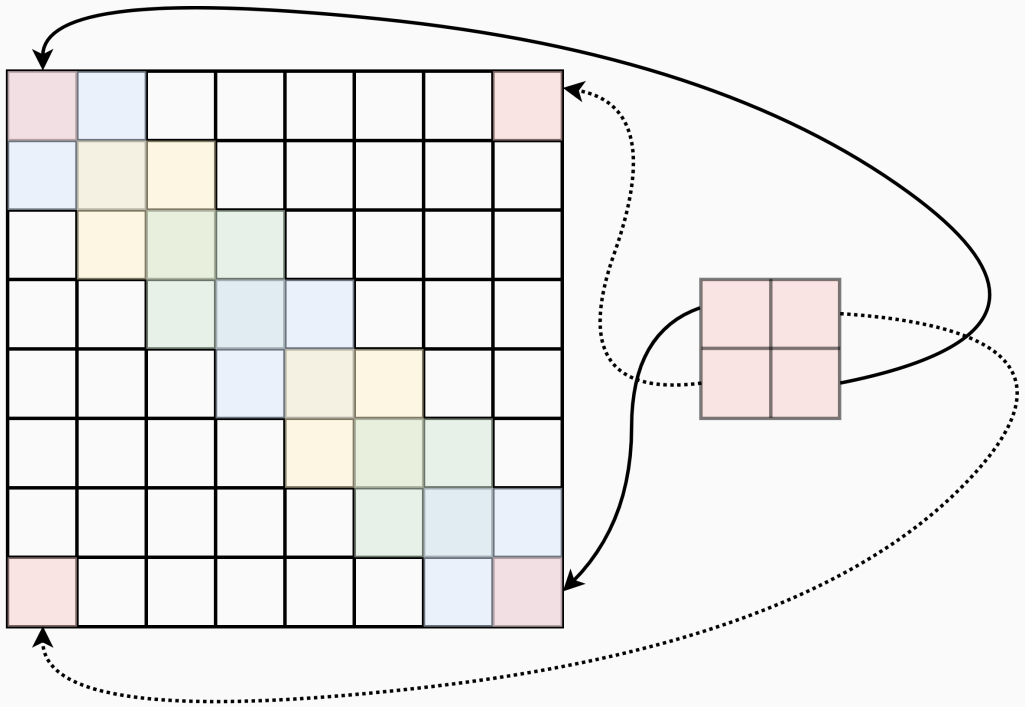
- Expect same result as previous code when $\phi_0 = 0$;
- Expect a straight beam to recover natural curvature when no external force applied.

```
from sympy import symbols, diff, tan, atan

def φk(xkm, xk, xkp, ykm, yk, ykp):
    return atan(((xkp - xk) * (yk - ykm)
 - (xk - xkm) * (ykp - yk)) / ((xkp -
 xk) * (xk - xkm) + (ykp - yk) * (yk -
 ykm)))

def Ebk(xkm, xk, xkp, ykm, yk, ykp, φko):
    return (2 * tan(φk(xkm, xk, xkp,
 ykm, yk, ykp) / 2.0) - 2 * tan(φko /
 2.0)) ** 2

xkm, xk, xkp, ykm, yk, ykp, φko =
 symbols('xkm1 xk xkp1 ykm1 yk ykp1 φk0')

Eb = Ebk(xkm, xk, xkp, ykm, yk, ykp, φko)

F1 = diff(Eb, xkm)
F2 = diff(Eb, ykm)
F3 = diff(Eb, xk)
F4 = diff(Eb, yk)
F5 = diff(Eb, xkp)
F6 = diff(Eb, ykp)

J11 = diff(F1, xkm)
J12 = diff(F1, ykm)
J13 = diff(F1, xk)
J14 = diff(F1, yk)
J15 = diff(F1, xkp)
```

What we did - Circular structure

Instead of $nv - 1$ edges, we have nv edges.
For bending, instead of $nv - 2$ components, we have nv components.
For stretching, instead of $nv - 1$ components, we have nv components.

When assembling the Jacobians, new components added to connect two ends together:



What we did - Circular structure

Verify our code by running the “hanging circle”:

$Y = 10^6\,\mathrm{Pa}$

1e6.mp4

$Y = 10^7\,\mathrm{Pa}$

1e7.mp4

$Y = 10^8\,\mathrm{Pa}$

1e8.mp4

What we did - Inflation pressure

With $\underline{x}_i$ and $\underline{x}_{i+1}$, we can easily calculate force exerted on those two points.

As force is dependent on actual positions of nodes, we need to add the variation into the Jabobian matrix. We simply take derivatives on the forces.

What we did - Surface contact

Predictor-corrector method is used.

Assume a surface at $y = 0$, When doing time-marching, on each time step :

1. Compute $\underline{q}(t)$ as before;
2. Check if there exists any node whose $y < 0$. If there is any, set it as a temporarily constrained DOF with $y = 0$, and recompute current time step;
3. Check if there exists any temporarily constrained DOF, such that the normal *force* between the surface and the node is negative. Remove such temporary constraint, and recompute current time step.

```
Fb, Jb = getFb(qCurrentIterate, EI, nv, voronoiRefLen, -2 * pi / nv, isCircular=True)
Fs, Js = getFs(qCurrentIterate, EA, nv, refLen, isCircular=True)
Fg = m * garr
Fp, Jp = getFp(qCurrentIterate, nv, refLen, InflationPressure)

Forces = Fb + Fs + Fg + Fp

# Equation of motion
f = m * (qCurrentIterate - q0) / dt ** 2 - m * u / dt - Forces
fUncons = dofHelper.unconstrained_v(f)
```

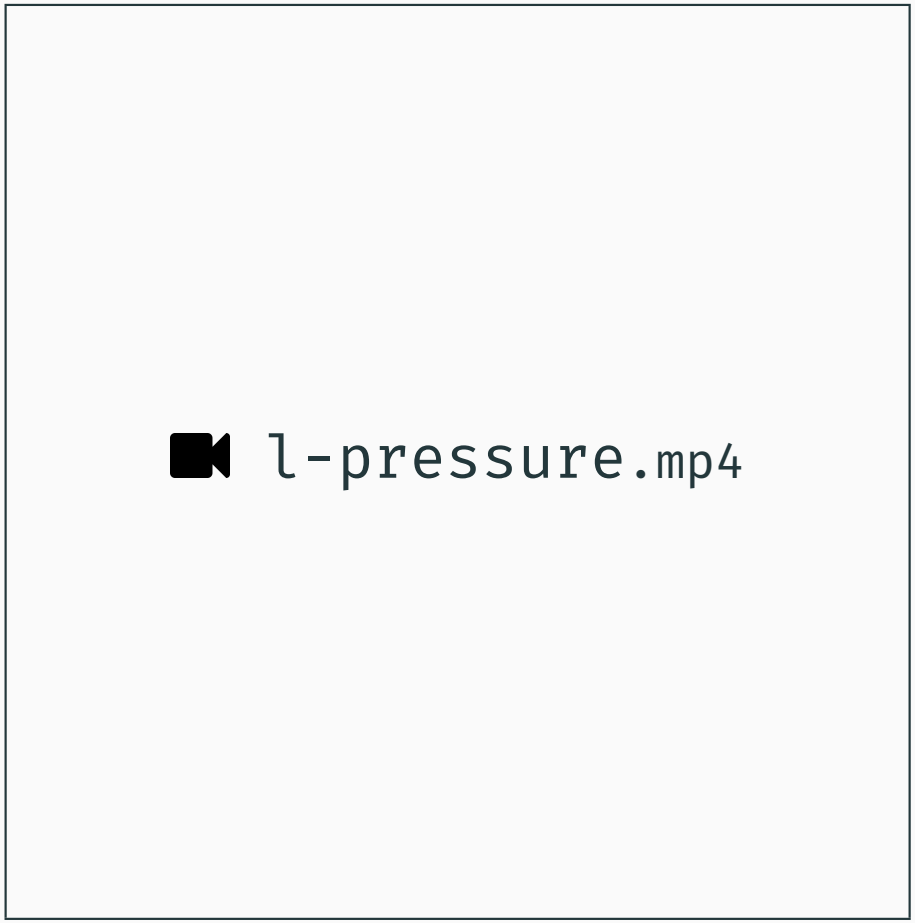
What we did - Surface contact

In each frame:

```
        qNew, reactionForces = objfun(q0)

        # inspect reactionForces to see if any one is negative, which should be UNCONSTRAINED
        needToFree = [unconsInd for unconsInd in dofHelper._constrained if (reactionForces[
unconsInd] < 0)]
        if needToFree:
            dofHelper.unconstraint(needToFree)
            print('Contact condition updated. Remove constraints and recompute')
            qNew, reactionForces = objfun(q0)

        # inspect qNew to see if any one falls below ground, which should be CONSTRAINED
        while True:
            q0Effective = None
            for c in range(nv):
                index = 2 * c + 1
                if qNew[index] < 0:  # y < 0: bad!
                    if q0Effective is None:
                        q0Effective = q0.copy()
                    q0Effective[index] = 0
                    dofHelper.constraint([index])
            if q0Effective is None:
                break
            else:
                print('Contact condition violated. Add constraints and recompute')
                qNew, reactionForces = objfun(q0Effective)
```

Objective completed!

Low pressure



High pressure

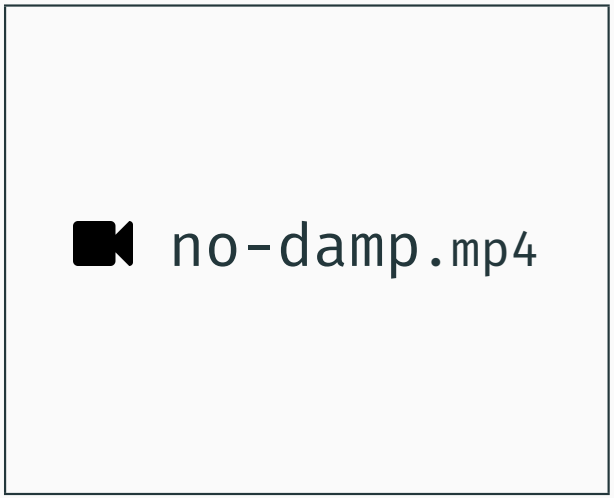
What we did - Damping

Without damping, we have oscillations.

Naïve damping $\underline{F}_{d,i} \propto -\underline{u}_i$ ruins bulk motion (as our bulk velocity is high)

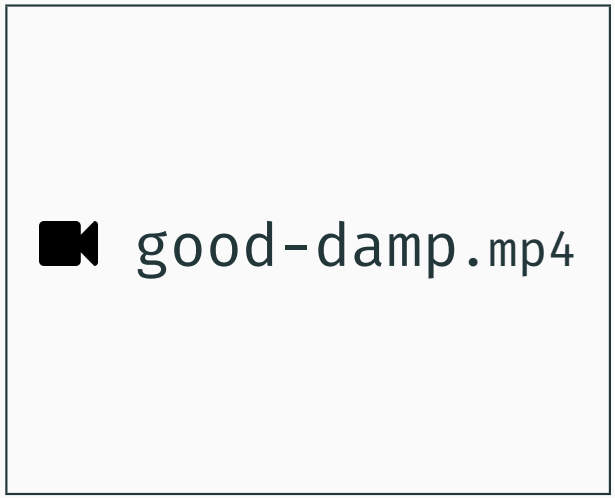
So we use velocity relative to center of mass: $\underline{F}_{d,i} \propto -(\underline{u}_i - \underline{u}_{cm})$

No damping



Naïve damping

CM-Relative damping



Problem: Shape Jitter

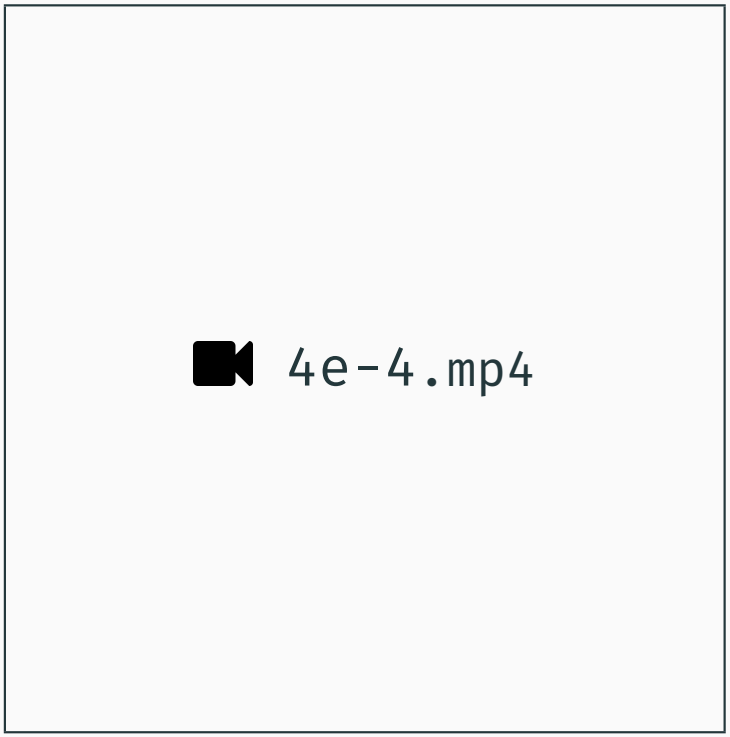
- Step size too large, everything bashed to ground before reacting
- Self-intersection occurs

Problem: Shape Jitter - Observations

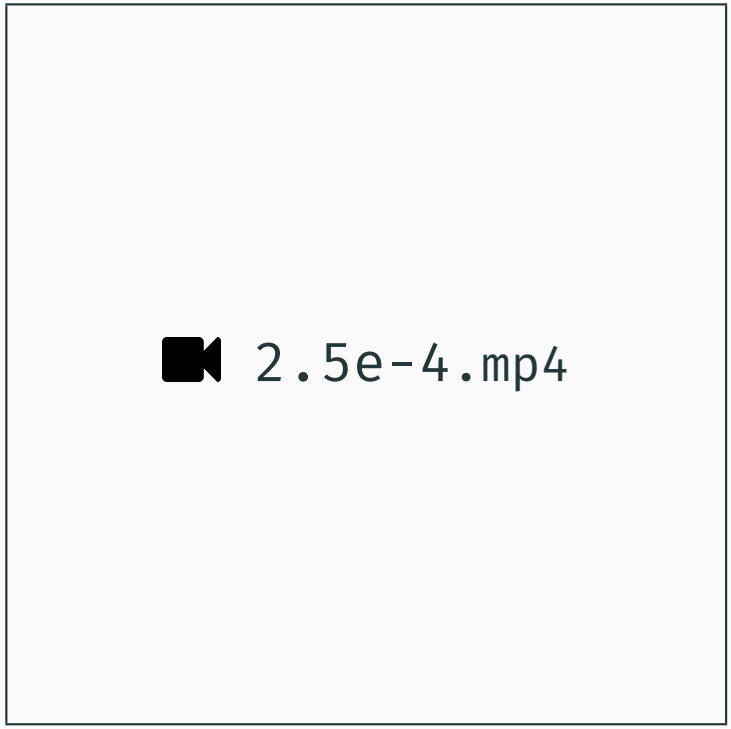
Damping does not really help.

Reducing time-step size helps. For the previous setup:

$dt = 4 \times 10^{-4}\,\mathrm{s}$



$dt = 2.5 \times 10^{-4}\,\mathrm{s}$



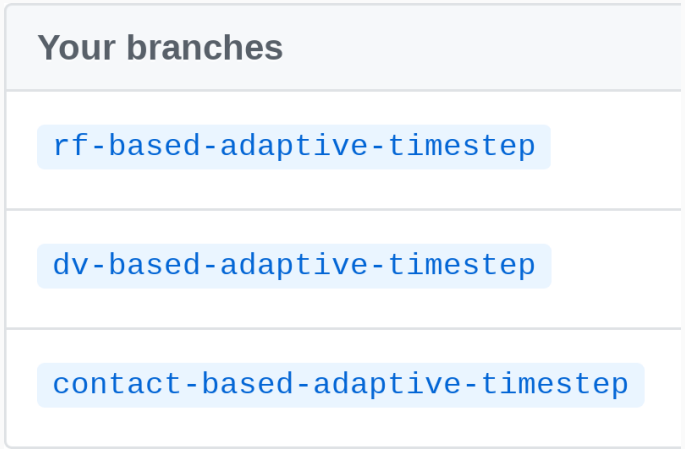
Compared to initial time step size $5 \times 10^{-3}\,\mathrm{s}$. This takes $20\times$ more computational time! Almost unacceptable.

Adaptive Time Stepping

“Adaptive Time Stepping” in the assigned reading gave us some inspiration.

When the time step is found too large by some criteria, we reduce the step size and try again.

When the adverse conditions are gone, step size is increased again.



Adaptive Time Stepping

Δv-based adaptive time stepping

They use a Δx threshold as criteria for deciding time steps. As our bulk velocity is high, we use the Δv threshold instead.

Not very effective: Only saves 20% steps. The oscillation also gives rapidly changing local velocity, we cannot distinguish the "bashing".

Contact-based adaptive time stepping

Use small step when in contact with the surface. Otherwise use large step.

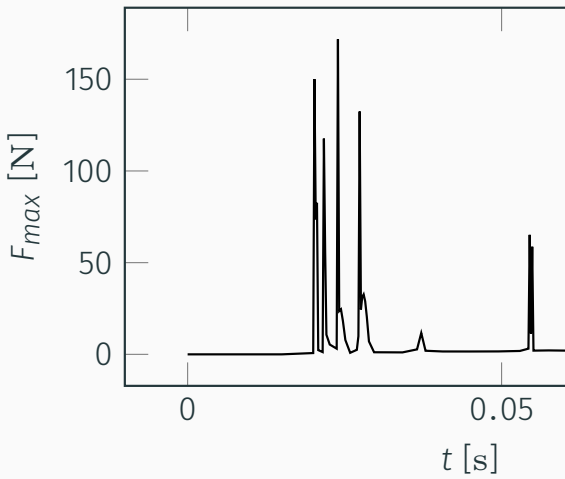
Not very effective: Not flexible enough (only two step sizes); step size is small even when resting on the surface.

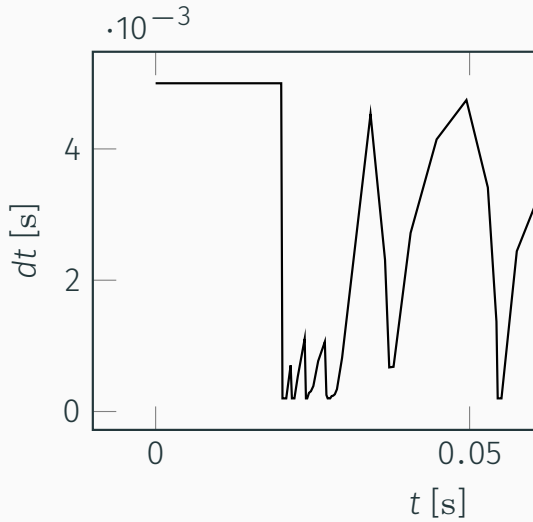
Adaptive Time Stepping

Reaction force-based adaptive time stepping

Step size should be inverse proportional to the maximum nodal reaction force from the ground.
We use $F_{max}dt \leq 0.008\,\mathrm{N}\cdot\mathrm{s}$ as the threshold for previous setup.

Instead of "decreasing by half, increasing by doubling" (as used in the paper), we examine $\frac{F_{max}dt}{0.008\,\mathrm{N}\cdot\mathrm{s}}$ to decide how much we should increase/decrease.





Adaptive Time Stepping

For a $2\,\mathrm{s}$ simulation, only 487 steps attempted, resulting in 437 valid steps.

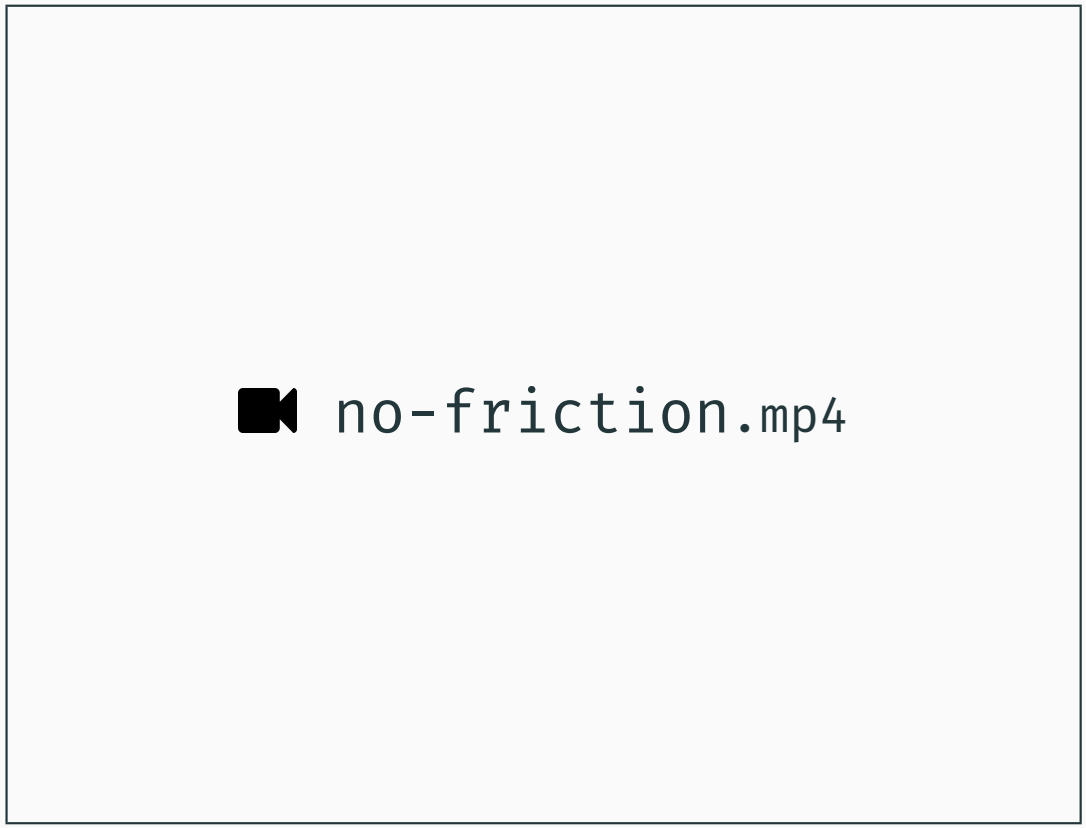
Compared to 400 steps (initial setup $dt = 5 \times 10^{-3}\,\mathrm{s}$), we only need $\sim 20\%$ more steps, instead of $\sim 2000\%$.

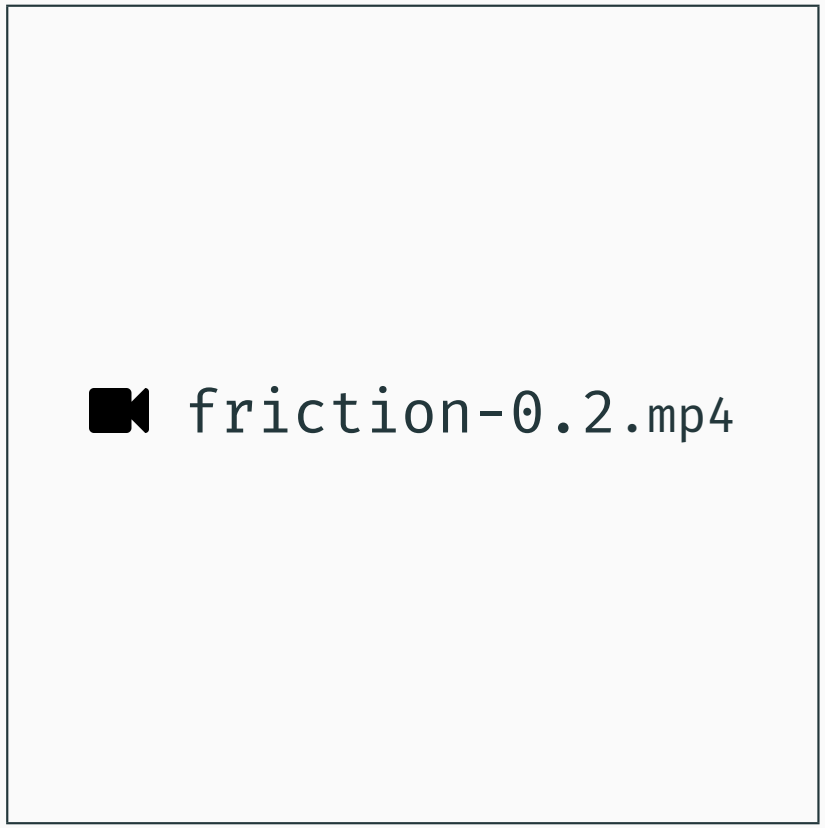
ats.mp4

ats-slow.mp4

Surface friction considered

For all nodes in contact with the surface, a friction force proportional to its velocity is applied: $F_{f,i} \propto -u_i$.

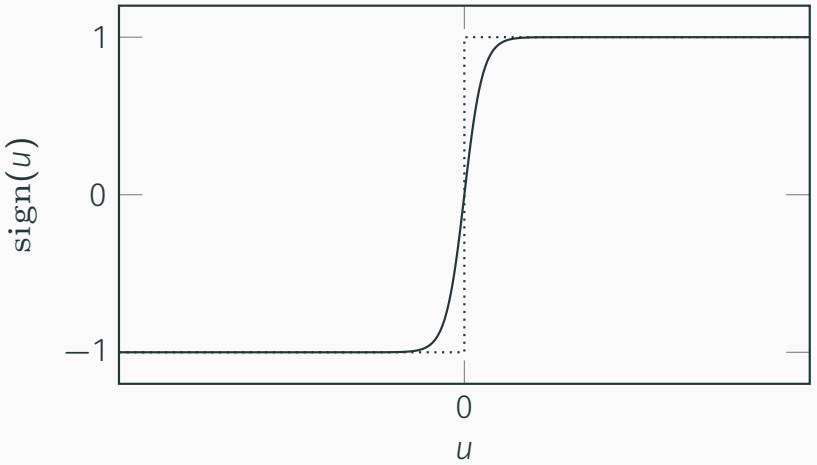




More realistic friction

Model local friction be proportional to the local normal force: $F_{f,i} = -\mathrm{sign}(u_i)\mu F_{N,i}$.

However, normal force is acquired after solving EOM. We don't know it when forming EOM with $F_{f,i}$.



What if we use the normal force F_N acquired in previous Newton-Raphson iteration?

- The result is still accurate when iteration converges.
- However, convergence is not guaranteed, because we don't have a reliable J_f.

More realistic friction

However, convergence is possible with an approximation, using $\mathrm{sign}(u_{t=t_{k-1}})$ instead of $\mathrm{sign}(u_{t=t_k})$.

What's next

- Better simulation of friction
- Self-intersection prevention

  "When collisions between a cloth vertex and triangle, or two cloth edges are detected, we insert a strong damped spring force to push the cloth apart" (Baraff 1998)
- Compare with results from literature
- 3-D sphere (with discrete shells)

Project on GitHub now!

All code, results (and these slides) uploaded to

`https://github.com/kmxz/mae259b`

See our results interactively at

`https://kmxz.github.io/mae259b`