

# 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..



hw-render.mp4



hw-nodes.mp4

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

# What we did - Performance optimization

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

```
def gradEbAndHessEb(xkml, ykml, xk, yk, xkpl, ykpl, φk0):
```

```
    itm1 = 2 * tan(0.5 * φk0)
    itm2 = ((-xk + xkpl) * (xk - xkml) + (-yk + ykpl) * (yk - ykml))
    itm3 = ((-xk + xkpl) * (yk - ykml) - (xk - xkml) * (-yk + ykpl))
    itm4 = itm3 / itm2 ** 2
    itm5 = tan(0.5 * atan(itm3 / itm2))
    itm6 = (1 + itm3 ** 2 / itm2 ** 2)
    itm7 = ((ykml - ykpl) / itm2 + itm3 * (2 * xk - xkml - xkpl) / itm2 ** 2)
    itm8 = ((-xkml + xkpl) / itm2 + itm3 * (2 * yk - ykml - ykpl) / itm2 ** 2)
    itm9 = ((-xk + xkml) / itm2 + (-yk + ykml) * itm4)
    itm10 = ((-xk + xkml) * itm4 + (yk - ykml) / 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 + xkpl) * itm4 + (-yk + ykpl) / itm2) * itm12
F[1] = 2 * ((xk - xkpl) / itm2 + (-yk + ykpl) * itm4) * itm12
F[2] = 2 * itm7 * itm12
F[3] = 2 * itm8 * itm12
F[4] = 2 * itm10 * itm12
F[5] = 2 * itm9 * itm12
```

```
Jl1 = 2 * ((-2 * xk + 2 * xkpl) * (-xk + xkpl) * itm3 / itm2 ** 3 + 2 * (-xk + xkpl) * (-yk + ykpl) / itm2 ** 2) * itm12 + 2 * ((-2 * xk + 2 * xkpl) * itm14 - (-2 * yk + 2 * ykpl) * itm4) * ((-xk + xkpl) * itm4 + (-yk + ykpl) / itm2) * itm13 + 2 * ((-xk + xkpl) * itm4 + (-yk + ykpl) / itm2) ** 2 * itm11 + 2 * ((-xk + xkpl) * itm4 + (-yk + ykpl) / itm2) ** 2 * (itm5 ** 2 + 1) ** 2 / itm6 ** 2
```

```
Jl2 = 2 * (-itm1 + 2 * itm5) * (itm5 ** 2 + 1) * ((-xk + xkpl) * (xk - xkpl) / itm2 ** 2 + (-xk + xkpl) * (-2 * yk + 2 * ykpl) * itm3 / itm2 ** 3 + (-yk + ykpl) ** 2 / itm2 ** 2) / itm6 + 2 * ((xk - xkpl) / itm2 + (-yk + ykpl) * itm4) * ((-xk + xkpl) * itm4 + (-yk + ykpl) / itm2) * itm11 + 2 * ((xk - xkpl) / itm2 + (-yk + ykpl) * itm4) * ((-xk + xkpl) * itm4 + (-yk + ykpl) / itm2) * (itm5 ** 2 + 1) ** 2 / itm6 ** 2 + 2 * ((-xk + xkpl) * itm4 + (-yk + ykpl) / itm2) * (-2 * xk - 2 * xkpl) * itm4 - (-2 * yk + 2 * ykpl) * itm14) * itm13
```

```
Jl3 = 2 * (-itm1 + 2 * itm5) * (itm5 ** 2 + 1) * ((-xk + xkpl) * (ykml - ykpl) / itm2 ** 2 + (-xk + xkpl) * itm3 * (4 * xk - 2 * xkml - 2 * xkpl) / itm2 ** 3 + (-yk + ykpl) * (2 * xk - xkml - xkpl) / itm2 ** 2 - itm4) / itm6 + 2 * itm7 * ((-xk + xkpl) * itm4 + (-yk + ykpl) / itm2) *
```

30.6 s



7.6 s

4x faster

# What we did - Natural curvature

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

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  $\phi_k(x_{km}, x_k, x_{kp}, y_{km}, y_k, y_{kp})$ :
    return atan(((xkp - xk) * (yk - ykm)
    - (xk - xkm) * (ykp - yk)) / ((xkp -
    xk) * (xk - xkm) + (ykp - yk) * (yk -
    ykm)))

def Eb( $x_{km}, x_k, x_{kp}, y_{km}, y_k, y_{kp}, \phi_{k0}$ ):
    return (2 * tan( $\phi_k(x_{km}, x_k, x_{kp},$ 
    ykm, yk, ykp) / 2.0) - 2 * tan( $\phi_{k0}$  /
    2.0)) ** 2

xkm, xk, xkp, ykm, yk, ykp,  $\phi_{k0}$  =
    symbols('xkm1 xk xkp1 ykm1 yk ykp1  $\phi_{k0}$ ')

Eb = Eb( $x_{km}, x_k, x_{kp}, y_{km}, y_k, y_{kp}, \phi_{k0}$ )

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)
```

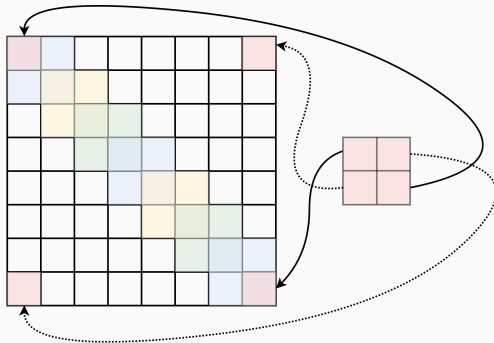
# What we did - Circular structure

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”:

$$\gamma = 10^6 \text{ Pa}$$

 1e6.mp4

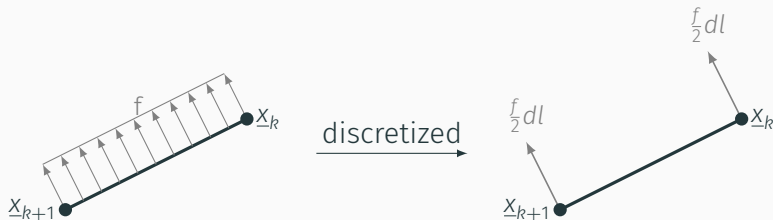
$$\gamma = 10^7 \text{ Pa}$$

 1e7.mp4

$$\gamma = 10^8 \text{ Pa}$$

 1e8.mp4

## What we did - Inflation pressure



With  $\underline{x}_k$  and  $\underline{x}_{k+1}$ , 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 Jacobian 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 frame;
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 frame.

```
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

 l-pressure.mp4

High pressure

 h-pressure.mp4

# What we did - Damping

Without damping, we have oscillations.

Naïve damping  $\underline{F}_{d,k} \propto -\underline{v}_k$  ruins bulk motion (as our bulk velocity is high)

So we use velocity relative to center of mass:

$$\underline{F}_{d,k} \propto -(\underline{v}_k - \underline{v}_{cm})$$

No damping

 no-damp.mp4

Naïve damping

 bad-damp.mp4

CM-Relative damping

 good-damp.mp4

## Problem: Shape Jitter

🎥 jitter.mp4

🎥 jitter-slow.mp4

- Step size too large, everything bashed to ground before reacting
- No self-intersection detection

## Project on GitHub now!

All code, results (and these slides) uploaded to

<https://github.com/kmxz/mae259b-project>

