Final Report

CAO Anh Quan & DROZDOVA Mariia

INF643 - Soft Robots: Simulation, Fabrication and Control
AI-VIS

February 10, 2020

Contents

1 Lab #1: Pendulum

Let us assume we have n springs and n+1 balls. We solved the following system of equations that are basically Newton laws. We have a gravity force and a spring force(elasticity). We solve it through an explicit scheme.

$$\Delta l_i = |\vec{r}_i - \vec{r}_{0i}|$$
$$\vec{n}_i = (\vec{r}_i - \vec{r}_{0i})/\Delta l_i$$
$$\vec{F}_{si} = k\Delta l_i \vec{n}$$
$$\vec{F}_{gi} = m\vec{g}$$
$$\vec{F}_i = \vec{F}_{gi} + \vec{F}_{si}$$
$$\vec{a}_i = \vec{-}F_{i-1} + \vec{F}_i$$
$$\vec{v}_i = \vec{v}_i + \vec{a}_i\Delta t$$
$$\vec{r}_i = \vec{r}_i + \vec{v}_i\Delta t$$

We can modify parameters k and m. With smaller k, the springs stretch more. With smaller m the movement stops faster as there is less inertia.

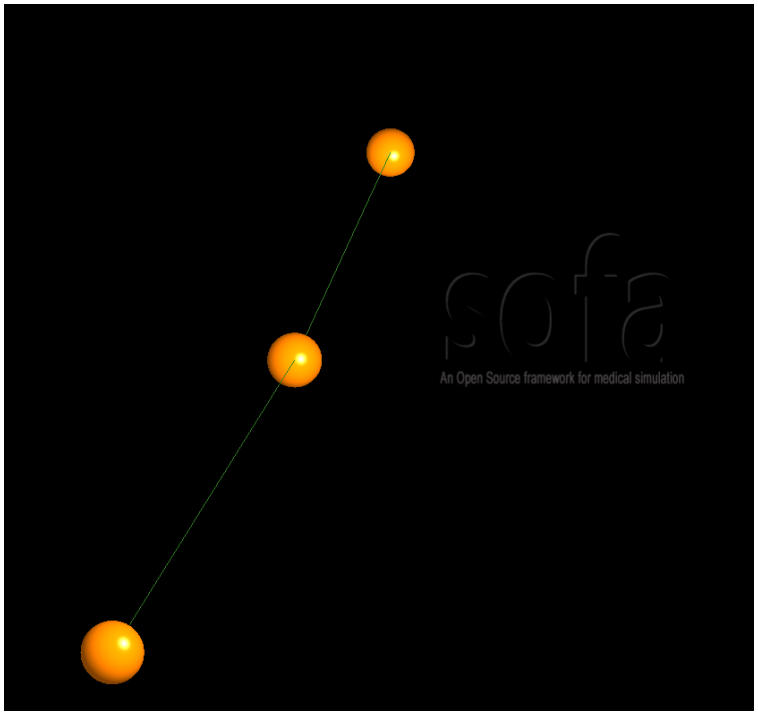

Figure 1: A pendulum.

```
        vecs = self.MechaObject.velocity
        spring_forces = self.computeSpringForce()
        for i in range(1, len(self.nodes)):
            if i != 0:
                total_force = -self.m* self.gravity

                for force in spring_forces[i]:
                    total_force += np.array(force)
                acc = total_force / self.m

                vecs[i] = np.array(vecs[i])
                vecs[i] += acc * deltaTime
                self.nodes[i] += vecs[i] * deltaTime

                self.nodes[i] = self.nodes[i].tolist()
                vecs[i] = vecs[i].tolist()

        self.MechaObject.position = self.nodes
        self.MechaObject.velocity = vecs
```

Listing 1: Code for the animation loop of a pendulum.

2 Lab #3: Tripod

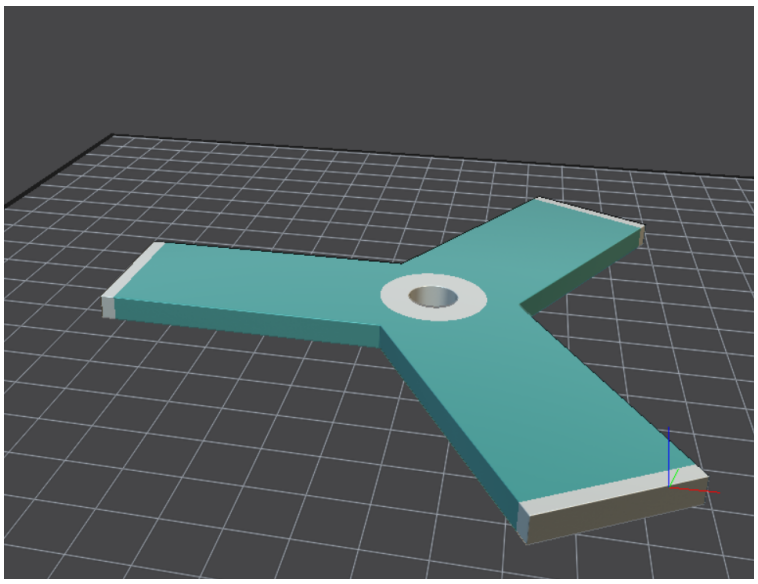

Figure 2: A final tripod.

In this Lab we designed a tripod. We created it using IceSL. Thanks to 3D textures we coded density and anglefield for the material. You can see the corresponding values on the 2D map in the Figures 3 and 4

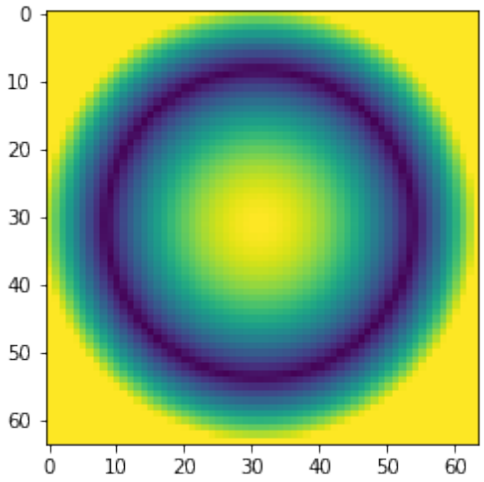

Figure 3: Desnity Map.

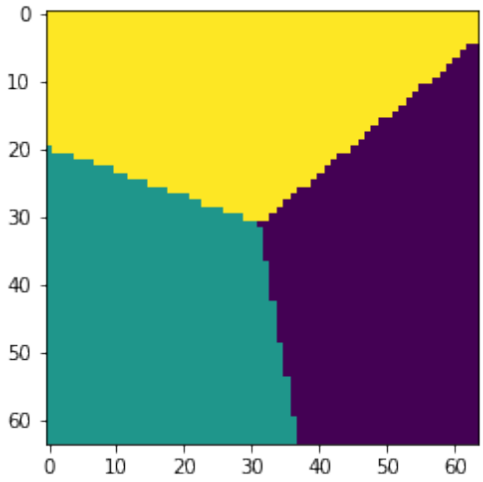

Figure 4: Anglefield.

Density

We wanted to make the density not uniform but higher near the end of the legs and the ring as this elements should hold better. We decided to use square augmentation of the density near these elements while keeping it uniform in the middle.

Anglefield

Anglefield was created from the following assumptions: we want the tripod to be easily bending in its legs, thus, it should consist of blocks which are parallel to its width. The ring has different anglefield: its normals follow the radius.

Unfortunately we did not succeed to slice with both anglefield and density because of the error. In the figure you can see the tripod with a constant density and anglefield from Figure 5

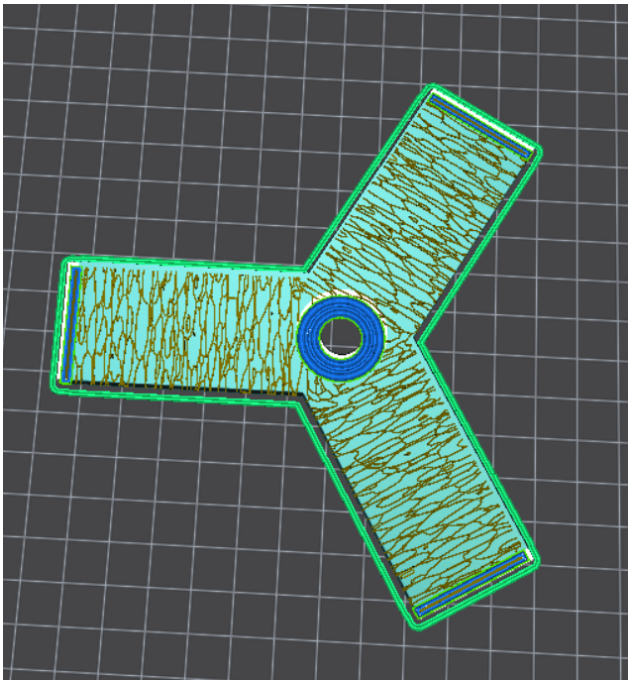

Figure 5: A sliced tripod with anglefield.

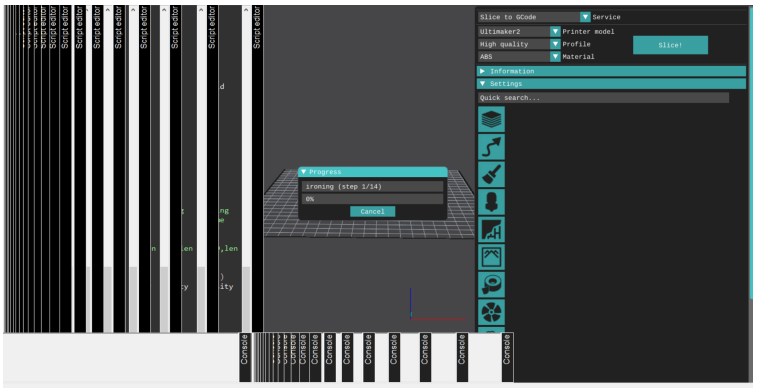

Figure 6: Example of the bug when we try to run density code.

3 Lab #4: Control of the maze

During the lab we were asked to come up with deterministic algorithm to "solve" the maze. "Solving" the maze means that the ball makes cycles in the maze in an infinite loop.

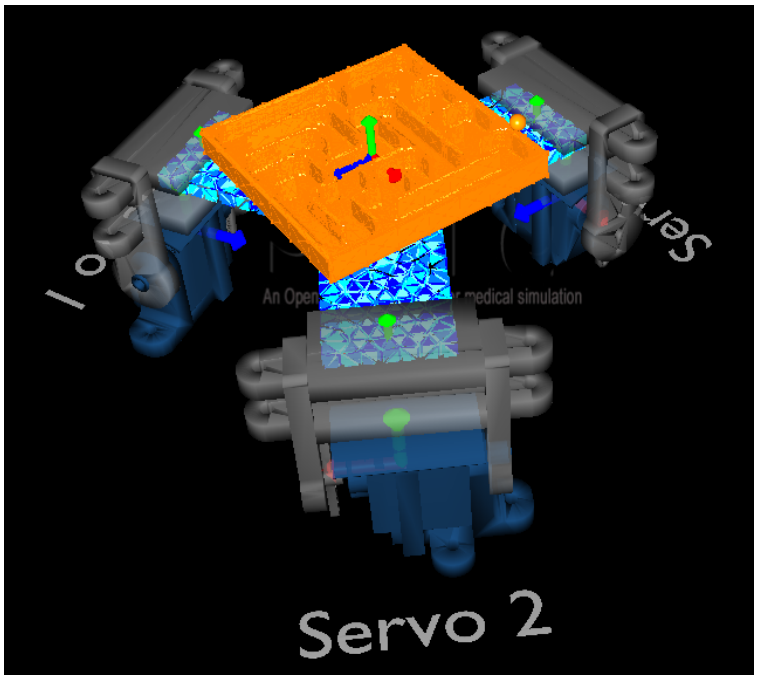

Figure 7: A robot simulation with a maze.

We were provided with a simulation of a real soft robot. In model-based control we use our knowledge about the concrete model and its dynamics to perform a control. For example, in this task we did not have to specify the movements of the robot (of its free legs). Instead, we give the desired output expressed in the angles of maze rotating around two axes x and z and the robot moves accordingly.(inverse control) The angles correspond to `self.theta_x` and `self.theta_z`. In the beginning of simulation they are equal to zeros and the axes lay in the same plane as the maze.

To come up with a sequence of actions that will allow an infinite loop, we manually made one loop during which we automatically registered the values of angles and the corresponding time in a `.txt` file.

```
0   0   0
0.33   0   -0.1
0.39   0   0.0
0.56   0.1   0.0
...
14.14   0.0   -0.1
14.51   0.0   0.0
```

Listing 2: Registered values of time, `self.theta_x` and `self.theta_z` from `actions4.txt`.

Then we introduced a parameter `flag` that is controlled by **L**-button. It is initialized to False and each **Ctrl+L** changes its value to the opposite one.

```
#with open('actions4.txt', 'a+') as f:
#    f.write(str(self.time) + '  ' + str(self.theta_x) + '  ' + str(self.
    theta_z) + '\n' )
```

Listing 3: Code for writing down the sequence of action from `onKeyPressed` function.

```
if key == Key.L:
    self.flag = not self.flag
    #remember time when animation started
    self.time0 = self.time + 0.1 - self.time0
```

Listing 4: Modified part of the code from `onKeyPressed` function.

Another variable `self.time0` stores the moment of the time when **L** was pressed. We need it as during the loop we will subtract this value from the current time to get the relative time. This relevant time will determine which action we should do based on the the values in the `.txt` file. We add a 0.1 for a small delay. Anyway, at least `dt` should be added as out first time value from the file corresponds to 0.0 while in reality we will compare with it only at the moment `dt` after the button is pressed as we call for a loop in `onBeginAnimationStep` function. We subtract the previous value of `self.time0` in case the control was stopped by **Ctrl+L** and then renewed again.

Note: It is important not to give any other commands to the maze as the control is predetermined and does not include response to not predicted interventions.

In `onBeginAnimationStep` function we call for a function `next_operation` which excute the prerecorded sequence of actions from zero state. Zero state corresponds to the maze after **Ctrl+A** and **Ctrl+I** are pressed and no rotations are performed.

```
if self.flag:
    self.next_operation(dt)
```

Listing 5: Modified part of the code from `onBeginAnimationStep` function.

The idea is simple. When we create an instance of class GoalController we read the prerecorded sequence of states from the file which path is given by `path_to_file`. (**You should modify a path to an absolute path of `actions4.txt`!**)

```
        self.theta_xs, self.theta_zs, self.times = [],[],[]
        self.time0, self.counter = 0, 0
        self.flag = False
        #an absolute path to the file
        path_to_file = r'C:\Users\Etudient1\Desktop\SOFA_v19.06.99
    _custom_Win64_v8.1\bin\actions4.txt'
        if not os.path.exists(path_to_file):
            print("File is not found : " + str(path_to_file))
            return
        with open(path_to_file, 'r') as f:
            for line in f:
                t, _, theta_x, _, theta_z, _ = (re.split(' |\n|\t||', line))
                self.times.append(float(t))
                self.theta_xs.append(float(theta_x))
                self.theta_zs.append(float(theta_z))
```

Listing 6: Lines added to the end of init function of GoalController class.

We read the time moments, rotation angles around x and z axes and save them to the three lists. We create a `self.counter` which tracks what is the next index from the lists we need to take.

In the `next_operation` function, first we check if we already made a loop. If so, we reset the relevant time (`self.time0`) and the counter(`self.counter`). Then, we check until which moment we should no change anything (`sleeping_time`). It corresponds to the moment of time when we will perform next action from the lists and is determined by the counter. We check if the current time is not further than (`dt/2`) (`dt` is the time step) from the next action time. If so, we update the angles and set a counter to its next value.

```
def next_operation(self, dt):
        if self.counter >= len(self.times):
            self.counter = 0
            self.time0 = self.time
        sleeping_time = self.times[self.counter]
        #time to push a new action
        if (self.time - self.time0 >= sleeping_time - dt/2 and self.time -
    self.time0 < sleeping_time + dt/2):
            theta_x = self.theta_xs[self.counter]
            theta_z = self.theta_zs[self.counter]
            self.counter += 1
            self.theta_x = theta_x
            self.theta_z = theta_z
            print('done')
        return
```

Listing 7: Function for executing the prerecorded sequence of actions.

We tested the chosen sequence for more than 10 loops in the simulation and it succeeded to accomplish them continuously. We did not test them with the real soft robot.