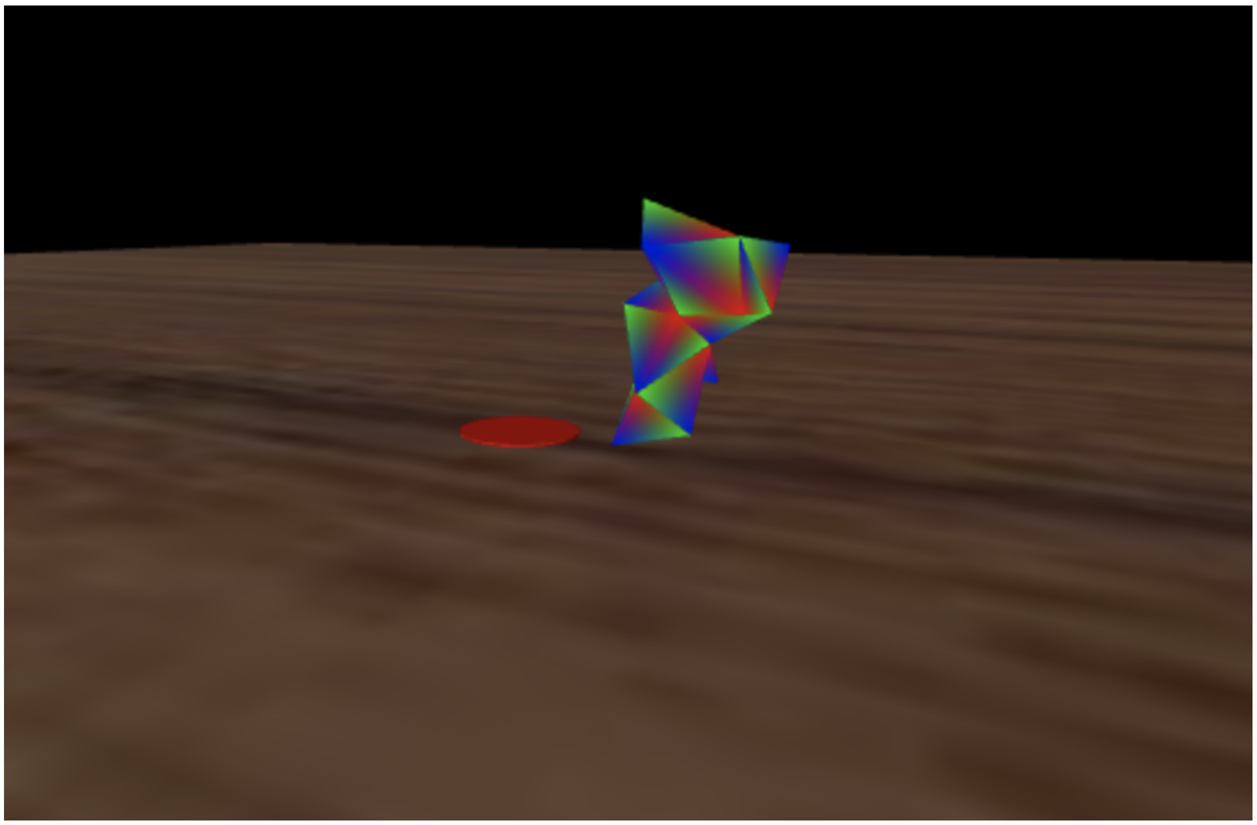
MECS E4510: Evolutionary Computation & Design Automation

Dr. Hod Lipson

Assignment 3c - Evolved Robot

Abhinit Kothari, ak4429
Connor Finn, cmf2196



December 9, 2019

9:00 PM

Grace hours before submission: 104

Grace hours used for submission: 21

Grace hours after submission: 83

1 Results Summary

1.1 Evolved Robot Performance

The fasted robot across three cycles of motion held an average speed of 1.4 m/s and a performance of 0.43 diameters / cycle.

Table 1 provides the velocity and distance traveled and by the top robot generated from each algorithm. A cycle is defined as the robot travelling for one full gait. Because the representation is done via sine waves with frequency, w = 62.8, a full cycle occurs after 0.1 seconds. The diameter of the robot is considered to be the largest distance between any 2 masses; 0.327m for both EA generated robots, 0.298m for the HC robot, and 0.261m for the random search robot.

Algorithm	Distance After 3 Cycles (meters)	Velocity (m/s)	Velocity (m/diameter
Simple EA	0.4217	1.40	0.43
Diverse EA	0.3961	1.32	0.4
Hill Climber	0.2628	0.876	0.29
Random Search	.1222	0.41	0.16

Table 1: Velocities and Distances travelled by the robots generated by each algorithm over 10,000 robot evaluations.

1.2 Video Representations of the Parametric Robot

- Video of the best performing robot:
  `https://youtu.be/yBPrXmlqBFY`
- Video of robot bouncing to validate simulation:
  `https://youtu.be/dJKbS_ADAmU`
- Multiple Robots Shown in one simulation:
  `https://youtu.be/bhF5fu3i4qc`
- Parade of the best four robots:
  `https://youtu.be/sp_NahPGa2M`

1.3 Image of Fastest Robot: Multiple Frames

Three images of the best performing robot's motion are shown in Figure 1. The three frames is depict stages of one robot cycle in the order (a) , (b) , (c). While the robot comprises only point masses and springs, multicolored cube faces and a wood plane floor were added in order to give the cube a solid appearance. The red circle shown in the Figure 1 is a region representing the starting location of the robot. It is assumed to have zero mass; thus, it does not impact the physics of the simulation.

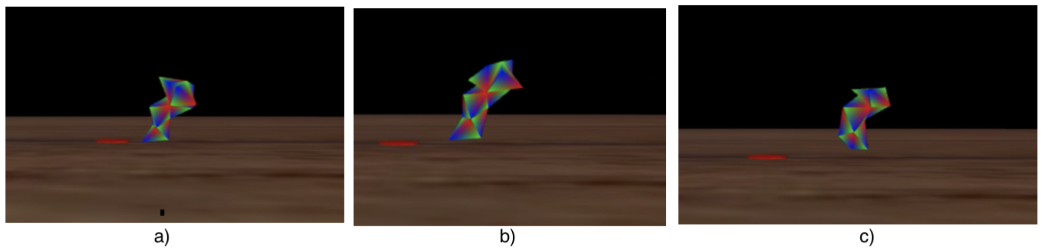


Figure 1: Three frames of the evolved robot moving in the simulator. The images provide a demonstration of the robot's single step forward.

2 Methods

2.1 Simulation Parameters

The built simulator uses discretized time to apply a numerical simulation of motion for a tetrahedron based robot. The robot, and all other objects modeled by this simulator, comprise point masses and springs that connect them. If, at any point, a mass were to fall below the ground, a force equal to the spring constant of the ground multiplied by the distance the mass had traveled below the plane would be applied in the upward direction. For this assignment, metric units were used and gravity was applied in the negative y direction. The parameters used to define each component of the simulator are listed below.

- Point masses were assigned a mass of 0.4 kg
- Four types of spring materials used: Muscle 1 (k = 11,000 $\frac{N}{m}$); Muscle 2 (k = 9000$\frac{N}{m}$); Bone (k = 1,000,000$\frac{N}{m}$); Tissue (k = 500$\frac{N}{m}$).
- Coefficient of Static friction (Leather on Wood) = 0.61
- Coefficient of Dynamic friction (Leather on Wood)= 0.52
- Gravity: -9.8 $\frac{m}{s^2}$
- Rest Length: $Lo = a + b \cdot sin(w \cdot T + c)$
  - a: The initial rest length for each mass when the cube is at rest.
  - b: The magnitude of the sine curve.
  - c: The phase shift of the sine curve.
  - W: this is the frequency of the sine wave. It is held to 62.8$\frac{m}{s}$ for all springs in the simulation.

2.2 Actuation Parameters

The main parameters that make the robot move in the pattern that it does involve the spring type and tetrahedron baseline of the robot. The spring type has a major effect in making the robot ”gallop” forward because some springs (such as bone) produce a much larger force when compressed and usually bone type springs are located at the rear end of the robot to make sure the force it produces is to the right in the positive direction thereby increasing the fitness.

2.3 Robot Parameters

- Point masses were assigned a mass of 0.4 kg
- Four types of spring materials used: Muscle 1 (k = 11,000$\frac{N}{m}$), Muscle 2 (k = 9000$\frac{N}{m}$), Bone (k = 1,000,000$\frac{N}{m}$), Tissue (k = 500$\frac{N}{m}$).
- The ground spring constant: 100,000 $\frac{N}{m}$
- Gravity: -9.8 $\frac{m}{s^2}$
- Rest Length: $Lo = a + b \cdot sin(w \cdot T + c)$
  - a: The initial rest length for each mass when the cube is at rest.

  - b: The magnitude of the sign curve.
  - c: The phase shift of the sign curve.

- Time Step: dt = 0.0001
- Coefficient of Static Friction (wood on wood): 0.35
- Coefficient of Dynamic Friction (wood on wood): 0.3
- Damping: 0.999

2.3.1 Alternate Representation

The robots were generated by starting with a 'seed' tetrahedron, and building on top of it according to a genome which indicated the location and material for the next tetrahedron. All simulation parameters were the same as in assignment 3b except for the addition of different spring material types. 2 different muscle materials; championing oscillatory natures, a bone material with a large k value, and a tissue material, with a soft material were introduced. The generated robots were assumed to have a leathery feel, so the friction surface condition used was Leather on Wood, in order to be consistent with the wooden simulation environment surface.

2.3.2 Innovative Robots

The representation, discussed in Section 2.3.1 was inspired by the Generative Blueprint representation discussed in class. As a result, the generated robots varied greatly in design, shown in Figure 2 depicts three different robots generated; where white represents the bone material, blue and pink are muscle materials, and yellow is a tissue, as described in Section 2.3.

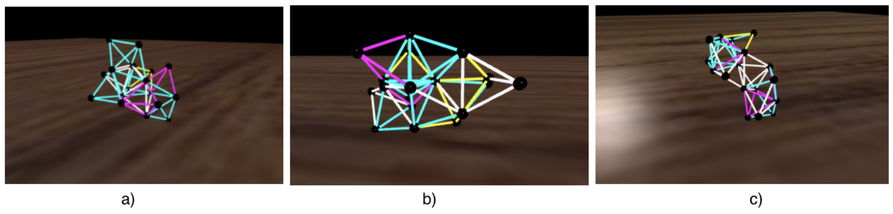


Figure 2: Three depictions of the innovative robots generated via evolutionary algorithms

2.4 Evolutionary Parameters

For the evolution part of this assignment, the genome representation was a 2 by 28 array of pointers for material type and surface index (location to add the new tetrahedron). The materials were given numbers between 1-4 since there were four different materials to choose from. After adding an additional tetrahedron, 3 new sides that are available , and one of the robot's previous sides is no longer open. The total number of sides available for attachment is 2*(number of tetrahedrons to be attached) + 3 (from the seed). Four approaches were made to generate the fastest and the longest traveling robot. Random search and Hill Climber algorithms were used as a baseline for comparison to the evolutionary algorithms. Random search involved created robots at random and calculating their distance traveled over a certain period of time, keeping track of the best robot. For hill climber, a random robot was generated and small mutations were made to the spring material and tetrahedron locations by changing a single number in the genome. For the EA, two variations were used: simple and diverse. For all the EAs, the genome with the spring material and side indices was evolved. These values are used for crossover and mutation. Mutation was done in the same manner as the hill climber, and the adjusted genome was kept only if the robot's performance improved.

For crossover, some parts of the genome of one robot are replaced by parts of a genome from another robot in the population. The new robot is then added back to the population. Fitness used to evaluate the EAs is based on how far horizontally a robot travels. Diversity is also calculated to compare the variance in the genomes of the different robots in a population.

- *Simple EA with crossover followed by mutation:* In this method, a population of a random parents was generated. A random number for a cut length was then obtained which was used for crossover. The cuts would come from the genome of different robots which contained the spring material and tetrahedron side indices. Two randomly chosen parents were then crossed over to produce a child. This crossover robot was then mutated 'm' number of times to obtained 'm' children. The fitness was compared to every previous iteration and only the good values were kept or else the robot parameters were reverted back for reevaluation. This evolution was carried out to produce 'n' generations of robots where only the best ones were carried over to the next generation.

- Diverse EA with crossover followed by mutation: For assignment 3c, diverse EA was modified a little bit. Instead of truncation selection, deterministic crowding was used where any child similar to the parent is better than the parent replaces the parent in the population. This was done to make the diversity management more effective since it was not that good in assignment 3b. In this method, the same approach was taken as the simple EA. Other than this, the diverse EA was same as the simple EA. This introduced variation in the robots that go through to another generation adding diversity to the population.

2.5 Discussion of What Worked and what did not work

In terms of the results, it can be seen from the performance plots that the simple EA produced the best fitness over 3 cycles. Diverse EA was very similar to the simple EA in terms of fitness over 10,000 evaluations. However, 10,000 evaluations is very small in the grand scheme of things and over a longer number of evaluations the diversity maintenance EA would play a larger effect on fitness. Both EAs produced robots that had good horizontal motion. In all the robots, the visualization shows a very similar motion of "galloping forward". This could be due to the interaction between different spring materials which are of varying stiffness values. Symmetry is something that could not be achieved. This could be why we have leaping robots instead of running robots. Symmetry would have reduced leaping motion and promote walking and running motions as the robots would tend to stay on ground. Being able to include symmetry would have allowed for actual walking/running robots. In addition to this, increasing the number of cycles and evaluations would have resulted in a larger fitness.

2.6 Performance Plots

2.7 Learning Curves

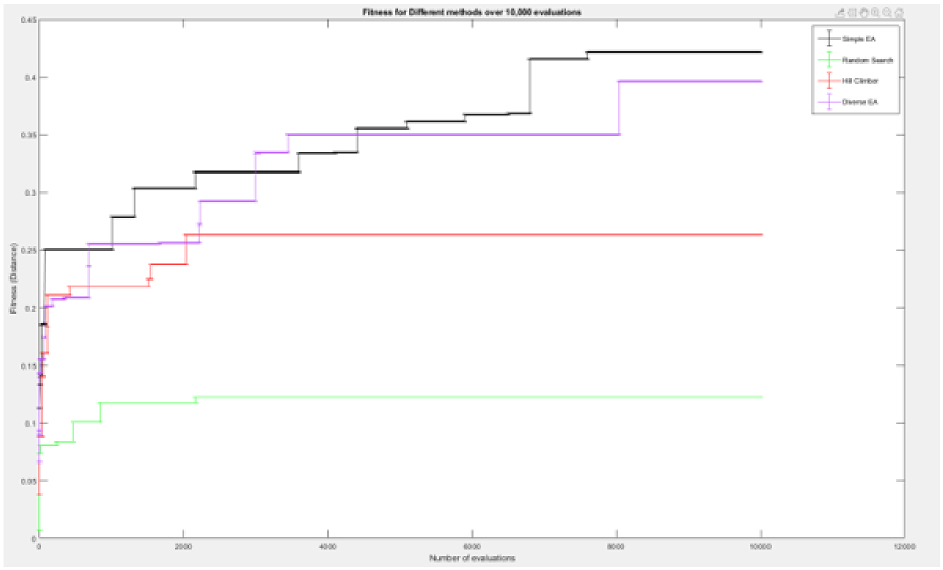


Figure 3: Fitness comparison between simple EA (crossover + mutation) and diverse EA over 10,000 robot evaluations. The simple EA produces better results.

Figure 3 demonstrates the fitness of the robots generated by each algorithm as a function of robot evaluations. The fitness is the horizontal distance travelled over three cycles of motion. Over 10,000 evaluations, the simple EA performed the best, and both variations of the evolutionary algorithms outperformed the baseline metrics. As seen in Figure 4, the diversity of both evolutionary algorithms is large (description in Section 2.8); thus, the diversity maintenance did not play a large role in producing high fitness. Over larger numbers of evaluations, the deterministic crowding that was used in the diverse ea is predicted to have greater effect.

2.8 Diversity Curve

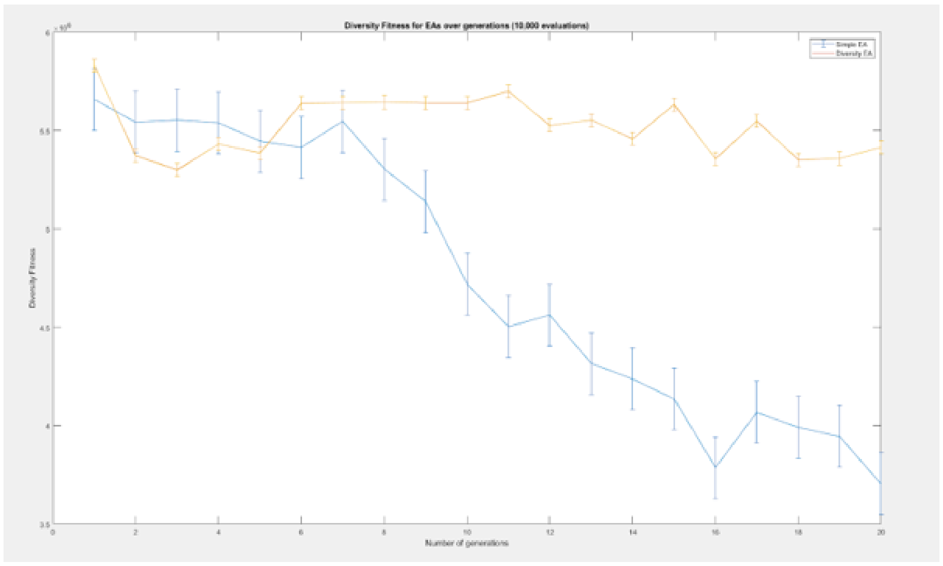


Figure 4: Diversity comparison between the Simple EA and Diverse EA over 10,000 evaluations.

The genomic diversity measure of the population was calculated by determining the euclidean distance between the genomes of every robot pair in the population. The EA which incorporated deterministic crowding maintained a much higher diversity measure over the 10,000 evaluations; however, both EAs ended with a large amount of diversity. This is likely due to the large population size (100 individuals) and comparatively small number of robot evaluations.

3 References and Bibliography

1. StackOverflow, https://stackoverflow.com/
2. Hod Lipson - class notes
3. Joni Mici - Outside Consultation

4 Appendix

4.1 Physics Simulator

```
"""
* this script is the physics simulator class
* we will import this for use with robots.py

"""

import numpy as np
from vpython import *
import matplotlib.pyplot as plt
import time



## constants we will be using

T = 0                                   # global time variable
g = np.array((0 , -9.8 , 0))    # gravity
dt = 0.0001                             # time step
damp = 0.999                            # add dampening to make it more realistic
mu_s = .61                                      # coeficient of static friction: oak on leather
mu_d = .52                              # coeficient of dynamic frriction: oak on leather
K_g = -100000                           # spring constant of the ground
#w = 2 * 3.14 / (1000 * dt )                    # frequency ( we want 100 time steps in a sine
    ↪ wave .001 * 2pi = dt * w )
w = 62.8

# we will start by creating classes for the point_mass and spring
#  * these are our building blocks for the robots

# make the robot class - going to have all kinds of useful functions
class simulator:

        def __init__(self):
                self.T_stop = 1
                self.springs_shown = []
                self.spheres_shown = []
                self.triangles_shown = []
                self.robot_list = []
                self.mass_list = []
                self.spring_list = []
        def set_robot(self , robot):
                """
                * this function will allow us to simulate more than one mass at a time
                * it takes in a list of robots, and sets the simulator's mass and spring lists
                  to be the sum of the masses and springs of the robots

                """
                mass_list = self.mass_list                                                  # get the
                    ↪  simulator's mass list
                spring_list = self.spring_list                                      # get the
                    ↪ simulator's spring list
                self.robot_list += [robot]                                                  # add
                    ↪ robot to the list of robots
                for mass in robot.mass_list:
                        mass_list += [mass]                                         # add the masses
                            ↪ and springs to the simulator
                for spring in robot.spring_list:
                        spring_list += [spring]




        def remove_triangles(self):
                self.triangles_shown = []
                self.triangles_to_make = []
        def remove_robot(self , robot):
                """
                * this takes in a robot, and removes it from the robot list
                * will return an error statement if the robot is not in the list
                """



                if robot in self.robot_list:                                                        #
                    ↪  check if robot is in the list
                        self.mass_list = [item for item in self.mass_list if item not in robot.
                            ↪ mass_list]                        # remove masses
                        self.spring_list = [item for item in self.spring_list if item not in
                            ↪ robot.spring_list]          # remove springs
                        self.robot_list.remove(robot)                                               #
                            ↪  remove the robot
                else:
                    ↪
                        print( "this robot is not in the simulator!")           # if it isn't
                            ↪ return an error message

# _____ Graphics methods:
        #method used to visualize
        def show_solid(self):
                """
        * this function takes in the mass array and will show them using vpython
                """
                num_robots = len(self.robot_list)                                   # get the
                    ↪  number of robots
                rad = np.sqrt(2) * .05
                interest_box=box(pos=vector(0,-.005,0),size=vector(10,.005,10), texture = 'https
                    ↪ ://s3.amazonaws.com/glowscript/textures/wood_texture.jpg')
                #offline_box=box(pos=vector(0,-.005,0),size=vector(10,.005,10), color = color.
                    ↪ white)

                rod = cylinder(pos=vector(0,-.005,0),  axis=vector(0,.005,0), radius=rad , color
                    ↪ = color.red)

                triangles = []

                print(len(self.robot_list))
                for robot in self.robot_list:
                        triangles_to_make = [[robot.mass_list[0] , robot.mass_list[1] , robot.
                            ↪ mass_list[2]]]
                        for side in robot.sides:
                                triangles_to_make += [side.mass_list]

                        for j in range(len(triangles_to_make)):
                                curr_triangle = triangles_to_make[j]
                                    ↪

                                m1 = curr_triangle[0]
                                m2 = curr_triangle[1]
                                m3 = curr_triangle[2]

                                m1_pos = m1.state[0]
                                m2_pos = m2.state[0]
                                m3_pos = m3.state[0]

                                v1 = vector( m1_pos[0] , m1_pos[1] , m1_pos[2])
                                v2 = vector( m2_pos[0] , m2_pos[1] , m2_pos[2])
                                v3 = vector( m3_pos[0] , m3_pos[1] , m3_pos[2])

                        # shows material
                        #       a = vertex( pos=v1 , color = m1.color)
                        #       b = vertex( pos=v2 , color = m1.color)
                        #       c = vertex( pos=v3 , color= m1.color )

                        # variable colors
```

```
                        a = vertex( pos=v1 , color = color.red)
                        b = vertex( pos=v2 , color = color.green)
                        c = vertex( pos=v3 , color= color.blue )


                        T = triangle( vs = [a , b , c])
                        triangles += [T]

        self.triangles_shown = triangles


def robot_zoo(self , robot_list):
        pass


        # this method will be used to update the image during simulation
def update_solid_image(self ):
        """
* this function takes in the mass array and will show them using vpython
        """


        triangles = self.triangles_shown

        for i in range(len(self.robot_list)):
                robot = self.robot_list[i]
                triangles_to_make = [[robot.mass_list[0] , robot.mass_list[1] , robot.
                    ↪ mass_list[2]]]
                for side in robot.sides:
                        triangles_to_make += [side.mass_list]

                for j in range(len(triangles_to_make)):
                        curr_triangle = triangles_to_make[j]
                            ↪

                        m1 = curr_triangle[0]
                        m2 = curr_triangle[1]
                        m3 = curr_triangle[2]


                        m1_pos = m1.state[0]
                        m2_pos = m2.state[0]
                        m3_pos = m3.state[0]

                        vec1 = vector( m1_pos[0] , m1_pos[1] , m1_pos[2])
                        vec2 = vector( m2_pos[0] , m2_pos[1] , m2_pos[2])
                        vec3 = vector( m3_pos[0] , m3_pos[1] , m3_pos[2])
                        t = triangles[ 32 * i + j]
                        # print(" num , "  ,  len(triangles_to_make))
                        t.v0.pos.x = m1_pos[0]
                        t.v0.pos.y = m1_pos[1]
                        t.v0.pos.z = m1_pos[2]
                        t.v1.pos.x = m2_pos[0]
                        t.v1.pos.y = m2_pos[1]
                        t.v1.pos.z = m2_pos[2]
                        t.v2.pos.x = m3_pos[0]
                        t.v2.pos.y = m3_pos[1]
                        t.v2.pos.z = m3_pos[2]
        # print(T , len(triangles))
#method used to visualize
def show_masses(self ):
        """
* this function takes in the mass array and will show them using vpython
        """
        num_masses = len(self.mass_list)
        num_springs = len(self.spring_list)
        spring_list = self.spring_list
        mass_list = self.mass_list

        scene = canvas(background = color.black )

        rad = np.sqrt(2) * .05
        interest_box=box(pos=vector(0,-.005,0),size=vector(10,.005,10), texture = 'https
            ↪ ://s3.amazonaws.com/glowscript/textures/wood_texture.jpg')
        #offline_box=box(pos=vector(0,-.005,0),size=vector(10,.005,10), color = color.
            ↪ white)
        #rod = cylinder(pos=vector(0,-.005,0),  axis=vector(0,.005,0), radius=rad , color
            ↪  = color.red)


        r =.01# radius of masses
        spheres = []
        springs = []

        for i in range(num_masses):
                m = self.mass_list[i]
                curr_pos = m.state[0]
                ball = sphere(pos = vector(curr_pos[0] , curr_pos[1] , curr_pos[2]),
                    ↪ radius = r,color = color.black)


                spheres += [ball]
        for j in range(num_springs):
                spr = self.spring_list[j]
                    ↪                                                # get the spring
                    ↪ object
                m1 = spr.mass_1
                m2 = spr.mass_2

                m1_pos = m1.state[0]
                m2_pos = m2.state[0]



                v1 = vector( m1_pos[0] , m1_pos[1] , m1_pos[2])
                v2 = vector( m2_pos[0] , m2_pos[1] , m2_pos[2])
                c = curve(pos=[v1,v2], color=spr.color, radius=0.0025)
                springs += [c]
        self.spheres_shown = spheres
        self.springs_shown = springs



def delete_everything(self ):
        """
        * this function deletes the vpython image
        * only works with the solid image.
        """
        for i in range(len(self.triangles_shown)):      # this loop will remove all
            ↪ triangles from window
                triangle = self.triangles_shown[i]
                triangle.visible = False
                del triangle




        # this method will be used to update the image during simulation
def update_image(self):
        """
* this function takes in the mass array and will show them using vpython
        """
        r =.01# radius of masses
        spheres = self.spheres_shown
        springs = self.springs_shown

        num_masses = len(self.mass_list)
        num_springs = len(self.spring_list)
```

```
                for i in range(num_masses):
                        m = self.mass_list[i]
                        curr_pos = m.state[0]
                        ball = spheres[i]
                        ball.pos = vector(curr_pos[0] , curr_pos[1] , curr_pos[2])


                for j in range(num_springs):
                        spr = self.spring_list[j]
                            ↪                                               # get the spring
                            ↪ object
                        m1 = spr.mass_1
                        m2 = spr.mass_2

                        m1_pos = m1.state[0]
                        m2_pos = m2.state[0]


                        v1 = vector( m1_pos[0] , m1_pos[1] , m1_pos[2])
                        v2 = vector( m2_pos[0] , m2_pos[1] , m2_pos[2])
                        c = springs[j]
                        c.clear()
                        c.append([v1,v2])






#  _____ The Simulation:


        def simulate(self ,  T_stop , show):
                global T
                # putting this here allows for it to be input dependant
                self.T_stop = T_stop
                self.kinetic_energy =np.zeros(int(T_stop / dt) + 2)
                self.potential_energy =np.zeros(int(T_stop / dt) + 2)
                self.spring_energy=np.zeros(int(T_stop / dt) + 2)
                self.total_energy =np.zeros(int(T_stop / dt) + 2)
                self.time =np.zeros(int(T_stop / dt) + 2)


                num_springs= len(self.spring_list)
                num_masses = len(self.mass_list)
```

```
# for x in range(num_masses):
#       print(x , self.mass_list[x].state[0])

time.sleep(20)
T =0

# need a list of forces for each mass
force_list = [0] * num_masses
ground_force_list = [0] * num_masses
ground_energy_list = [0] * num_masses

tally = 0
start = time.time()
while T < T_stop:
        self.time[tally] = time.time() - start
        temp_force_list = [np.zeros(3)] * num_masses
        # evaluate the springs
        for k in range(num_springs):
                spring_temp = self.spring_list[k]       # get the current spring
                    ↪

                # get the two masses attattched to the spring
                m1 = spring_temp.mass_1
                m2 = spring_temp.mass_2
                # get the length of the spring and the unit vector
                curr_length = m1.state[0] - m2.state[0]
                curr_length2 = m2.state[0] - m1.state[0]
                dist = np.linalg.norm(curr_length)
                unit_vec1 = curr_length / dist
                unit_vec2 = -1 * unit_vec1
                #unit_vec2 = curr_length2 / dist
                # get the rest length
                rest = spring_temp.rest_length
                Lo = rest[0]
                rest_dist = np.linalg.norm(Lo) + rest[1] * np.sin(rest[2] + T * w
                    ↪ )

                # get the force scalar ( may not need this -1)
                force =  -1 * spring_temp.spring_constant * (dist - rest_dist)

                # multiply by the unit vector
                F_spring1 = force * unit_vec1
                F_spring2 = force * unit_vec2


                # add spring force to the foce on each masss
                index_1 = self.mass_list.index(m1)
                index_2 = self.mass_list.index(m2)


                # add it in!
                temp_force_list[index_1] = temp_force_list[index_1] +  F_spring1
                temp_force_list[index_2] = temp_force_list[index_2] +  F_spring2

                #add to the potential energy
                self.spring_energy[tally] = self.spring_energy[tally] +
                    ↪ spring_temp.spring_constant * 0.5 * (dist - rest_dist)**2
                    ↪                                # add pot from spring

        # add gravity & the ground force
        static = [0] * num_masses
        for i in range(num_masses):

                mass_temp = self.mass_list[i]                    # get the point
                    ↪ mass object
                temp_force_list[i] = temp_force_list[i] + g * mass_temp.mass
                    ↪          # add gravity


                # check if it's below ground - if it is, add ground force and
                    ↪ friction
                if mass_temp.state[0][1] <= 0:

                        # Friction
                        normal = abs(temp_force_list[i][1])
                            ↪          # get the normal force
                        horizontal = np.delete(temp_force_list[i] , 1)
                            ↪                  # get the horizontal forces
                        horizontal_magnitude = np.linalg.norm(horizontal)

                        if horizontal_magnitude < normal * mu_s:        # this is
                            ↪ the static friction
                                static[i] = 1
                                temp_force_list[i][0] = 0
                                temp_force_list[i][2] = 0

                        else:
                                horizontal_unit = horizontal /
                                    ↪ horizontal_magnitude
                                friction_force = normal * mu_d
                                    ↪                          # dynamic friction
                                    ↪  force
                                friction = horizontal_unit * friction_force * -1
                                    ↪         # in the oposite direction of the
                                    ↪ horziontal force
                                friction_array = np.array((friction[0] , 0 ,
                                    ↪ friction[1]))
                                    ↪                         # need to put y
                                    ↪ force of zero back in
                                temp_force_list[i] = temp_force_list[i] -
                                    ↪ friction_array


                        # model the ground as a spring with a large K
                        bounce_force = mass_temp.state[0][1] * K_g
                            ↪
                        adjust_force = np.array((0 , bounce_force , 0))
                        temp_force_list[i] = temp_force_list[i] + adjust_force
                        ground_force_list[i] = bounce_force
                        self.spring_energy[tally] = self.spring_energy[tally] +
                            ↪ -.5 * K_g * mass_temp.state[0][1] ** 2


        for p in range(num_masses):
                force_list[p] = temp_force_list[p]



        # now update the kinematics

        for j in range( num_masses):
                curr_force = force_list[j]
                curr_mass = self.mass_list[j]
                ground_energy = ground_energy_list[j]


                self.kinetic_energy[tally] = self.kinetic_energy[tally] + .5 *
                    ↪ curr_mass.mass * np.linalg.norm(curr_mass.state[1])**2
                #self.potential_energy[tally] = self.potential_energy[tally] +
                    ↪ ground_energy
                self.potential_energy[tally] = self.potential_energy[tally] +
                    ↪ curr_mass.mass * -1 *  g[1] * curr_mass.state[0][1]  # add
                    ↪  pot from height

                # update the state
                mass = curr_mass.mass
                arr1 = np.array(((1 , dt , 0) , ( 0 , 1 , 0) , ( 0 , 0 , 0)) )

                # so i left it as just mass
                arr2 = np.array((dt * dt / (mass) , dt / mass  , 1/ mass ) ).
                    ↪ reshape(3 , 1)
```

```
                                curr_mass.state = np.dot(arr1 , curr_mass.state ) + np.dot(arr2 ,
                                    ↪  curr_force.reshape(1 , 3))
                                curr_mass.state[1] = curr_mass.state[1] * damp

                                if static[j] == 1:
                                    ↪                         # in the event we can't overcome
                                    ↪ static friction
                                        curr_mass.state[1][0] = 0                               #
                                            ↪  x velocity = 0
                                        curr_mass.state[1][2] = 0                               #
                                            ↪  z velocity = 0

                        self.total_energy[tally] = self.kinetic_energy[tally] + self.
                            ↪ potential_energy[tally] + self.spring_energy[tally]
                        tally += 1
                        T += dt

                        # print("take " , T)
                        # for x in range(num_masses):

                        #       print(x , self.mass_list[x].state[0])

                        if show:
                                if T*100000 %20:
                                        #self.update_image()
                                        self.update_solid_image()


# _____ Plotting methods:

        def plot_energy(self):
                """
                * the potential energy needs to be adjusted to account for spring potential
                    ↪ energy again

                """



                x = np.arange(int(self.T_stop / dt) + 2)
                plt.plot( x[:20000] , self.kinetic_energy[:20000], color = 'blue', linewidth = 2
                    ↪ ,   label="kinetic_energy")
                plt.plot( x[:20000] , self.potential_energy[:20000],  color='red', linewidth=2 ,
                    ↪ label="potential_energy")
                plt.plot( x[:20000], self.total_energy[:20000],   color='black', linewidth=2,
                    ↪ label="total_energy")
                plt.xlabel('Simulation Time')
```

```
            plt.ylabel('Energy (Newtons)')
            plt.title('Energy')
            plt.legend()
            plt.show()

    def plot_time(self):
            x = np.arange(int(self.T_stop / dt) + 2) * 28
            plt.plot( x[:-2] , self.time[:-2], color = 'blue', linewidth = 2 ,   label="Time
                ↪ Elapsed (seconds) ")
            plt.xlabel('Spring Evaluations')
            plt.ylabel('Elapsed Time (seconds)')
            plt.title('Computational Efficiency')
            plt.show()
```

4.2 Robot Classes

```
"""
This script bounces a cube off of a floor using a custom built physics simulator and v python


"""
import numpy as np
from vpython import *


## constants we will be using

K = 10000                               # spring constant N/M
musc_1_const = 11000    # muscle type 1
musc_2_const = 9000             # muscle type 2
tissue_const = 500              # soft support
bone_const = 1000000    # hard support
tall = 0.08164965809277261






class point_mass:

    def __init__(self, mass , state ):
        self.mass = mass                                        # flaot in kg
        self.state = state                                      # state inclues position,
            ↪ velocity, and accelerate ( 3 by 3 np array)
        self.color = vector(0, 0, 0)

```

```
class spring:
        def __init__(self,  m1 , m2):
                self.spring_constant = K                                                 #
                self.rest_length = [m1.state[0] - m2.state[0] , 0 , 0]                           #
                    ↪  this is an array [Lo, b , c] feed into eq Lo + b * sin(wt + c)
                self.mass_1 = m1                                                         # mass
                    ↪ object
                self.mass_2 = m2                                                         # mass
                    ↪ object
                self.color = vector(0, 0, 0)                             # use http://www.rgbtool.
                    ↪ com/ to select color for this

"""
 * the following will all be subclasses of the spring class
 * they represent the types of springs that a robot could have
"""

class muscle_1(spring):

        def __init__(self, m1 , m2):
                super().__init__(m1 , m2)
                    ↪        # get typical spring initiation
                self.color = vector(0 , 252 , 30)                                                #
                    ↪  change the color
                self.rest_length[1] = 0.025
                    ↪                # change the b value
                self.rest_length[2] = 0
                    ↪        # change the c value
                self.spring_constant = musc_1_const



class muscle_2(spring):

        def __init__(self, m1 , m2):
                super().__init__(m1 , m2)                                                        #
                    ↪  get typical spring initiation
                self.color = vector(252, 0 , 30)
                self.rest_length[1] = 0.01
                    ↪                # change the b value
                self.rest_length[2] = 1.25 * np.pi
                    ↪                        # change the c value
                self.spring_constant = musc_2_const


class tissue(spring):
        def __init__(self, m1 , m2):
                super().__init__(m1 , m2)                                                  #
                    ↪  get typical spring initiation
                self.color = vector(252, 252 , 0)                                  # Yellow
                self.rest_length[1] = 0
                    ↪          # change the b value
                self.rest_length[2] = 0
                    ↪          # change the c value
                self.spring_constant = tissue_const

class bone(spring):

        def __init__(self, m1 , m2):
                super().__init__(m1 , m2)                                                  #
                    ↪  get typical spring initiation
                self.color = vector(252, 252 , 252)
                self.rest_length[1] = 0
                    ↪          # change the b value
                self.rest_length[2] = 0
                    ↪          # change the c value
                self.spring_constant = bone_const


class robot:
        """
        * this is our big class

        """

        def __init__(self):
                self.mass_list = [0] * self.masses_num                          # masses_num ,
                    ↪ spring_num set in subclass
                self.spring_list = [0] * self.spring_num
                self.center_of_mass = 0
                self.breathing_list = np.zeros(self.spring_num)         # list of rest lengths



        def get_com(self, mass_list):
                """
                * this function takes in a mass list
                * it returns the com

                """
                num_points = len(mass_list)             # get the number of masses
                pos_sum = np.zeros(3)                   # place holder
                for mass in mass_list:                  # add to the place holder
                        pos = mass.state[0]
                        pos_sum +=pos
                com = pos_sum / num_points              # divide by number of masses

                return com


        def center_object(self, com):
                """
                * this function takes in the com of the object, and moves every mass so that the
                    ↪ com is hovering about (0 , val , 0)

                """
                movement = -1* com              # want to get com to (0 , val , 0)
                movement[1] = 0                 # dont change height

                masses = self.mass_list
                for mass in masses:
                        mass.state[0] += movement

        def set_breathing_aray(self):
                """
                * this function sets the spring values to those in self.breathing_list
                * self.breathing_list is an array that contains elements ( b , c) in the formula
                    ↪ rest_length = Lo + b * sin(w*T + c)

                """
                num_springs = self.spring_num
                    ↪

                for i in range(num_springs):
                    ↪                          # set the values of breathing array equal to the
                    ↪ inputs
                        self.spring_list[i].rest_length[1:] = self.breathing_list[i]
                            ↪          # the first value of rest lenght does not change

        def make_breathing_array(self):
                """
                * this function takes in a robot and creates a new list of (b , c) values (in
                    ↪ arrays)
                * The array will be used for the breath function
                * by definition of the period for a sin function c is in (0 , 2pi)
                * b can be as large or small as we want. Let's just make it a random percentage
                    ↪ of the rest length Lo so b in (0 , 1)
                """
                num_springs = self.spring_num
                    ↪                   # get spring number
                breath_list = [0] * num_springs                 # make an empty array of that
                    ↪ size
                for i in range(num_springs):
                    ↪                   # fill empty array with values from apropriate domains
                        b = np.random.uniform(0 , .03)
                        c = np.random.uniform(0 , 2 * np.pi)

                        breath_list[i] = np.array((b,c))

                self.breathing_list = breath_list
                    ↪                   # return the array we generated


        def mutate(self):
                """
                * this function makes a small mutation to our breathing cube
                * the mutation is a simple adjustment to one of the springs.
                """

                num_springs = self.spring_num
                spring_index = np.random.randint(num_springs)                   # select a random
                    ↪  spring index

                spring_temp = self.spring_list[spring_index]                    # get the spring
                mutate_items = spring_temp.rest_length[1:]                          # get the
                    ↪  things we will mutate

                length = len(mutate_items)
                    ↪          # this is a list

                pick_one = np.random.randint(length)                                # pick
                    ↪ either b , c ect..
                val = mutate_items[pick_one]

                adjust = np.random.uniform(0.99 , 1.01)                                     #
                    ↪  get adjustment
                spring_temp.rest_length[pick_one + 1] = val * adjust    # update the spring

                return spring_index
                    ↪                   # this will be useful for updating self.breathing_array

        # this gets the center of mass at
        def init_com(self):
                print("here" , self.masses_num)
                masses = 0
                position = np.zeros(3)
```

```
                temp = 0
                for i in range(self.masses_num):
                        temp_mass = self.mass_list[i]

                        masses += temp_mass.mass
                        position += temp_mass.state[0]
                        temp += temp_mass.mass * temp_mass.state[0]

                com = temp / masses
                self.center_of_mass = com


        def get_diameter(self):
                for i in range(len(self.mass_list)):
                        for j in range(len(self.mass_list)):
                                m1 = self.mass_list[i]
                                m2 = self.mass_list[j]


# now we will make the class called a cube,  a subclass of robot
class cube(robot):

        def __init__(self):
                self.spring_num = 28
                self.masses_num = 8
                super().__init__()
                self.type = 0                                   # this is a pointer for use in a
                    ↪ switch statement









        def change_pos(self , location , theta , spin):


                # build the transformation matrix. This will set the cube to the euler angles
                    ↪ provided.
                theta_x = theta[0]
                theta_y = theta[1]
                theta_z = theta[2]

                trans_x = np.array(((1 , 0 , 0 , location[0] ), (0 , np.cos(theta_x) , -1 * np.
                    ↪ sin(theta_x) , 0) , (0 , np.sin(theta_x) , np.cos(theta_x) , 0) ,(0 , 0 ,
                    ↪ 0 , 1)) )
                trans_y = np.array(((np.cos(theta_y) , 0 , np.sin(theta_y) , 0 ), (0 , 1 , 0 ,
                    ↪ location[1]) , (-1 * np.sin(theta_y) , 0 , np.cos(theta_y) , 0) ,(0 , 0 ,
                    ↪ 0 , 1)) )
                trans_z = np.array(((np.cos(theta_z) , -1 * np.sin(theta_z) , 0 , 0 ), (np.sin(
                    ↪ theta_z) ,  np.cos(theta_z), 0 , 0) , (0 , 0 , 1 , location[2]) ,(0 , 0 ,
                    ↪ 0 , 1)) )

                transformation_matrix = np.dot(trans_x , np.dot(trans_y , trans_z))



                # first make the masses
                index = 0
                    ↪                                                   # used to keep track of
                    ↪  loc
                # these nested for loops are used to fill a position vertor
                for p in range(2):
                        for l in range(2):
                                for n in range(2):
                                        # this constructs a cube centered at the origin
                                        pos = np.array((p / 10 -.05 , l/ 10 -.05  , n / 10 -.05 )
                                            ↪ )

                                        pos_m = np.hstack((pos , 1))

                                        pos_m_alt = np.dot( transformation_matrix , pos_m)

                                        pos_alt = pos_m_alt[0:3]

                                        if spin:
                                                if p == 0:
                                                        kinematics = np.vstack((pos_alt , np.
                                                            ↪ array((0 , 1 , 0)) , np.zeros(3)))
                                                        m = point_mass(.4 , kinematics)
                                                if p ==1:
                                                        kinematics = np.vstack((pos_alt , np.
                                                            ↪ array((0 , 0 , 0)) , np.zeros(3)))
                                                        m = point_mass(.4 , kinematics)
                                        else:
                                                kinematics = np.vstack((pos_alt , np.zeros(3) ,
                                                    ↪ np.zeros(3)))
                                                self.mass_list[index].state = kinematics
                                                index += 1
```

```
def set_cube(self , location , theta , spin ):
        """
 * This method will make the mass and spring objects we need for a cube
 * location is a 1 by 3 numpy array, it gives the adjustmetn from 0 , 0
 * theta is the angle rotated in radians
 * this cube will not have breathing components.

        """

        # build the transformation matrix. This will set the cube to the euler angles
            ↪ provided.
        theta_x = theta[0]
        theta_y = theta[1]
        theta_z = theta[2]

        trans_x = np.array(((1 , 0 , 0 , location[0] ), (0 , np.cos(theta_x) , -1 * np.
            ↪ sin(theta_x) , 0) , (0 , np.sin(theta_x) , np.cos(theta_x) , 0) ,(0 , 0 ,
            ↪ 0 , 1)) )
        trans_y = np.array(((np.cos(theta_y) , 0 , np.sin(theta_y) , 0 ), (0 , 1 , 0 ,
            ↪ location[1]) , (-1 * np.sin(theta_y) , 0 , np.cos(theta_y) , 0) ,(0 , 0 ,
            ↪ 0 , 1)) )
        trans_z = np.array(((np.cos(theta_z) , -1 * np.sin(theta_z) , 0 , 0 ), (np.sin(
            ↪ theta_z) ,  np.cos(theta_z), 0 , 0) , (0 , 0 , 1 , location[2]) ,(0 , 0 ,
            ↪ 0 , 1)) )

        transformation_matrix = np.dot(trans_x , np.dot(trans_y , trans_z))


        # first make the masses
        index = 0
            ↪                                                    # used to keep track of
            ↪  loc
        # these nested for loops are used to fill a position vertor
        for p in range(2):
                for l in range(2):
                        for n in range(2):
                                # this constructs a cube centered at the origin
                                pos = np.array((p / 10 -.05 , l/ 10 -.05  , n / 10 -.05 )
                                    ↪ )

                                pos_m = np.hstack((pos , 1))

                                pos_m_alt = np.dot( transformation_matrix , pos_m)

                                pos_alt = pos_m_alt[0:3]
```

```
                    if spin:
                        if p == 0:
                            kinematics = np.vstack((pos_alt , np.
                               ↪ array((0 , 1 , 0)) , np.zeros(3)))
                            m = point_mass(.4 , kinematics)
                        if p ==1:
                            kinematics = np.vstack((pos_alt , np.
                               ↪ array((0 , 0 , 0)) , np.zeros(3)))
                            m = point_mass(.4 , kinematics)
                    else:
                        kinematics = np.vstack((pos_alt , np.zeros(3) ,
                           ↪ np.zeros(3)))
                        m = point_mass(.4 , kinematics)
                    self.mass_list[index] = m
                    index += 1

        # now make the springs
        num = self.masses_num
        count = 0
    # this loop will get all the spring stuff

        for j in range(self.spring_num):
            for i in range(j + 1 , num):
                # add this spring to the mass objects it touches
                m1 = self.mass_list[i]
                m2 = self.mass_list[j]

                if self.type == 0:
                    s = spring( m1 , m2)

                elif self.type == 1:
                    s =muscle_1(m1 ,m2)

                elif self.type == 2:
                    s = muscle_2(m1 ,m2)

                elif self.type == 3:
                    s = bone(m1 , m2)

                elif self.type ==4:
                    s = tissue(m1 , m2)




                self.spring_list[count] = s
```

```
                                count += 1




class tetrahedron(robot):


        def __init__(self):
                self.spring_num = 6
                self.masses_num = 4
                super().__init__()
                self.type = 0                                   # this is a pointer for use in a
                    ↪ switch statement
                self.sides = []                                 # this list tells which sides are
                    ↪  availible to build off
                self.genome = []
                    ↪         # this is used for the adding tetrahedron function




        def set_genome(self , num_tetr_added):

                genome = np.zeros((num_tetr_added , 2))                         # first column is
                    ↪  side to add tetr to,
















                for i in range(num_tetr_added):
                        side_select = np.random.randint(2 * i + 3)                      # start
                            ↪ with 3 sides, add two more each time ( add three subtract 1)
                        type_select = np.random.randint(4)                                      #
                            ↪  bone, tissue, muscle 1 , muscle 2
                        genome[i] = (side_select , type_select)
                self.genome = genome



        def mutate_genome(self):
                """
                * this function makes a small mutation to the genome
                * it will return the indices of the adjustment - that way we can keep or revert
                    ↪ the change after evaluating the fitness
                """

                genome = self.genome                                            # get the genome
                    ↪ and its dimensions
                num_rows = np.size(genome , 1)
                num_cols = np.size(genome ,0)

                column = np.random.randint(num_cols)            # get a random position in the
                    ↪ genome
                row = np.random.randint(num_rows)
                old_val = genome[column , row]
                if row ==0:
                    ↪                 # reselecting the side to build off
                        val = np.random.randint(column * 2 + 3)
                else:
                    ↪                 # reselecting the material for this item
                        val = np.random.randint(4)

                genome[column , row] = val                                      # update the
                    ↪ genome


                return [column , row] , old_val                                         # return
                    ↪ the indices of the genome that we mutated




        def remove_old(self):
                self.mass_list = [0] * self.masses_num
                self.spring_list = [0] * self.spring_num
                self.sides = []



        def build_robot(self):
                """
                * this function is ran after set_genome
                * it will add the tetrahedron to the seed robot in accordance to the genome
                * the function then ensures no masses start below zero
                * finally it sets the center of mass to have a 0 x and z value
                """
                genome = self.genome
                for i in range(np.size(genome , 0)):
                        v1 = int(genome[i , 0 ])
                        v2 = int(genome[i , 1])
                        self.add_tetr(v1 , v2)


                # need to adjust so no masses are below zero
                min_y = 0
                for mass in self.mass_list:
                        if mass.state[0][1] < min_y:
                                min_y = mass.state[0][1]
                if min_y < 0:
                        for mass in self.mass_list:
                                mass.state[0][1] += -1* min_y

        def set_tetr(self , location , theta ):
                """
         * This method will make the mass and spring objects we need for a cube
         * location is a 1 by 3 numpy array, it gives the adjustmetn from 0 , 0
         * theta is the angle rotated in radians
         * this cube will not have breathing components.

                """
                """
         * This method will make the mass and spring objects we need for a cube
         * location is a 1 by 3 numpy array, it gives the adjustmetn from 0 , 0
         * theta is the angle rotated in radians

                """

                theta_x = theta[0]
                theta_y = theta[1]
                theta_z = theta[2]

                trans_x = np.array(((1 , 0 , 0 , location[0] ), (0 , np.cos(theta_x) , -1 * np.
                    ↪ sin(theta_x) , 0) , (0 , np.sin(theta_x) , np.cos(theta_x) , 0) ,(0 , 0 ,
                    ↪ 0 , 1)) )
                trans_y = np.array(((np.cos(theta_y) , 0 , np.sin(theta_y) , 0 ), (0 , 1 , 0 ,
                    ↪ location[1]) , (-1 * np.sin(theta_y) , 0 , np.cos(theta_y) , 0) ,(0 , 0 ,
                    ↪ 0 , 1)) )
                trans_z = np.array(((np.cos(theta_z) , -1 * np.sin(theta_z) , 0 , 0 ), (np.sin(
                    ↪ theta_z) ,  np.cos(theta_z), 0 , 0) , (0 , 0 , 1 , location[2]) ,(0 , 0 ,
                    ↪ 0 , 1)) )



                transformation_matrix = np.dot(trans_x , np.dot(trans_y , trans_z))

        # get the positions that would put a tetrahedron with the centroid at the origin
                side_length1 = 0.1
                depth = np.sqrt(0.1**2 - 0.05**2)
                mid = np.sqrt((depth/3)**2 + 0.05**2)
                height = np.sqrt(0.1**2 - mid**2)
                mass_positions = [ np.array((- 0.5 * side_length1 , 0.0 , -depth / 3)) , np.array
                    ↪ ((0.5 * side_length1 , 0.0 , - depth / 3) ), np.array((  0.0 , 0.0 , 2*
                    ↪ depth / 3)) , np.array((0.0, height , 0.0))]
                # first make the masses
                index = 0
                    ↪                                                    # used to keep track of
                    ↪  loc
                # these nested for loops are used to fill a position vertor
                for p in range(self.masses_num):
                        # this constructs a cube centered at the origin
                        pos = mass_positions[p]
                        pos_m = np.hstack((pos , 1))
                        pos_m_alt = np.dot( transformation_matrix , pos_m)
                        pos_alt = pos_m_alt[0:3]
                                        #pos = pos - np.array((.05 , 0.005 , .05))
                                        #pos = pos * rot_matrix
                                                # add the starting location
                        kinematics = np.vstack((pos_alt , np.zeros(3) , np.zeros(3)))
                        m = point_mass(.4 , kinematics)
                        self.mass_list[index] = m
                        index += 1

                # now make the springs
                num = self.masses_num
                count = 0
        # this loop will get all the spring stuff
                for j in range(self.spring_num):
                        for i in range(j + 1 , num):
                                # add this spring to the mass objects it touches
                                m1 = self.mass_list[i]
                                m2 = self.mass_list[j]
                                #build the spring object
                                s = muscle_1( m1 , m2)
                                self.spring_list[count] = s
                                count += 1
                # get the sides; the bottom side will not be added bc it is the base

                mass_list = self.mass_list

                # we would like to provide the center of mass as well / will help with
                    ↪ calculating normal
                com = self.get_com(mass_list)
                s1 = side([mass_list[0] , mass_list[1] , mass_list[3]] , com)
                s2 = side([mass_list[0] , mass_list[2] , mass_list[3]] , com)
                s3 = side([mass_list[1] , mass_list[2] , mass_list[3]] , com)



                self.sides = [s1 , s2 , s3]




        def get_diameter(self , mass_list):
                print("here2")

                di = 0
                for mass in mass_list:
                        for i in range(len(mass_list)):
                                alt = mass_list[i]
                                pos_sum = np.zeros(3)                   # place holder
                                pos1 = mass.state[0]
                                pos = alt.state[0]
                                pos_sum = pos - pos1
                                dist = np.linalg.norm(pos_sum)
                                if dist > di:
                                        di = dist

                print(di)


        def get_triangle(self, points , com ):
                """
                * this function gets the coordinates for the centroid and normal of a triangle
                * it takes in a list of 3 point masses

                """
                # get the x,y,z coordinates for each of the point masses
                p1 = points[0].state[0]
                p2 = points[1].state[0]
                p3 = points[2].state[0]
```

```
        # get the centroid coordinates
        Cx = (p1[0] + p2[0] + p3[0] ) / 3
        Cy = (p1[1] + p2[1] + p3[1] ) / 3
        Cz = (p1[2] + p2[2] + p3[2] ) / 3

        # get two sides
        s1 = p2 - p1
        s2 = p3 - p1

        # take the cross product
        n = np.cross(s1 , s2)
        size = np.linalg.norm(n)
        n = n / size

        # need to make sure direction is ok.  # this only works for the first triangle #
            ↪ ( may need to take in the com)
        if Cx > com[0]:
                n[0] = abs(n[0])
        else:
                n[0] = -1 * abs(n[0])

        if Cy > com[1]:
                n[1] = abs(n[1])
        else:
                n[1] = -1 * abs(n[1])

        if Cz > com[2]:
                n[2] = abs(n[2])
        else:
                n[2] = -1 * abs(n[2])

        return Cx , Cy , Cz , n



def add_tetr(self , side_num , type_spring):
        side1 = self.sides[side_num]                            # get the side
        masses = side1.mass_list                                        # get the masses
        com = side1.com
        Cx , Cy , Cz , n = self.get_triangle(masses , com)


        # build the point mass
        height = tall * n                               # empirically find tall (length)
            ↪ # this could be a point of error
```

```
                height += np.array((Cx , Cy , Cz))


                kinematics = np.vstack((height , np.zeros(3) , np.zeros(3)))
                m = point_mass(.4 , kinematics)
                self.mass_list += [m]

                for i in range(len(masses)):
                        m1 = masses[i]
                        if type_spring == 0:
                                s = tissue(m1 , m)

                        elif type_spring == 1:
                                s =muscle_1(m1 ,m)

                        elif type_spring == 2:
                                s = muscle_2(m1 ,m)

                        elif type_spring == 3:
                                s = bone(m1 , m)


                        self.spring_list += [s]

                #self.spring_num += len(masses)
                #self.masses_num += 1

                # update the sides
                self.sides.pop(side_num)
                # get com:
                masses +=[m]
                new_com = self.get_com(masses)

                for mass in masses:
                        mass.color = s.color

                s1 = side([masses[0] , masses[1] , m ] , new_com)
                s2 = side([masses[0] , masses[2] , m ], new_com)
                s3 = side([masses[1] , masses[2] , m ] , new_com)


                self.sides += [s1 , s2 , s3]


class side:
        def __init__(self , masses , com):
                self.mass_list = masses
                self.com = com                                          # this will help
                    ↪ calculate the normal ( com of the tetrahedron attatched)
                self.spring_list = []
```

4.3 Evolutionary Algorithms

```

import numpy as np
from physics_simulator import simulator
from robots import *


# ___________________ Operations library

def crossover(r1 , r2):
        """
        * this function takes in two robots and performs a crossover
        * the cross over will occur between the lists of (b , c) values
        * a new robot object will be generated.
        """
        child_robot = tetrahedron()                     # make a cube object
        child_robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))

        genome1 = r1.genome     # get the two lists of (b , c) values
        genome2 = r2.genome

        length = len(genome2)                           # get the length of the genome

        cut1 = np.random.randint(length)        # make two unique cuts
        cut2 = np.random.randint(length)

        while cut1 == cut2:                                     # make sure the cuts are
            ↪ different
                cut2 = np.random.randint(length)

        first = min(cut1 , cut2)                        # get the smaller and larget cuts
        second = max(cut1 , cut2)

        new_list = np.vstack((genome1[:first] , genome2[first : second] , genome1[second :]))   #
            ↪  make new list with cuts
        child_robot.genome = new_list                   # set as the new list

        child_robot.build_robot()                               # actuate our list to the springs
        # center the robot about the origin
        mass_list = child_robot.mass_list
        com = child_robot.get_com(mass_list)
        child_robot.center_object(com)

        return child_robot                                      # return the robot with the
            ↪ crossed over list


def make_population(num_robots , num_tetr_added):
        """
        * this function makes a population of robots
        * it takes the population size as an input
        * it returns a list of these robots
        """
        num_robots = int(num_robots)                                    # these need to be
            ↪ integers
        num_tetr_added = int(num_tetr_added)

        robot_list = [0] * num_robots                                    # empty list
        for i in range(num_robots):
                robot = tetrahedron()
                robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))                  #
                    ↪  make the seed
                robot.set_genome(num_tetr_added)
                    ↪         # build the genome
                robot.build_robot()                     # builds the robot and sets it about (0,
                    ↪ val , 0)
                # center the robot about the origin
                mass_list = robot.mass_list
                com = robot.get_com(mass_list)
                robot.center_object(com)

                robot_list[i] = robot           # fill list with breathing robots

        return robot_list


def calc_fitness( r):
        """
        * this function calculates the fitness of the robot
        * the fitness is defined by how far it travels
        """
        sim = simulator()
        sim.set_robot(r)
        com = r.get_com(r.mass_list)                    # center the object
        r.center_object(com)
        com1 = r.get_com(r.mass_list)
        sim.set_robot(r)
        print(" com1"  , com1)
        sim.show_solid()
        sim.simulate(.3 , True)                         # run simulator for three robot cycles
        com2 = r.get_com(r.mass_list)
        print(" com2"  , com2)
        distance = com2 - com1
        horizontal = np.delete(distance , 1)
        fit = np.linalg.norm(horizontal)
        sim.show_solid()
        sim.remove_robot(r)


                                # remove the robot from the simulator

        return fit

# this bubble sort function was modeled off of " https://www.geeksforgeeks.org/bubble-sort/"
def bubbleSort(population , fitness_list):
    n = len(population)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if fitness_list[j] < fitness_list[j+1] :
                fitness_list[j], fitness_list[j+1] = fitness_list[j+1], fitness_list[j]
                population[j] , population[j+1] = population[j+1] , population[j]





def diversity_val(population):
        """
        * this function takes in a population and returns an overall fitness score by evaluating
            ↪ every combination of robots
        * the diversity value between two robots is decided by the function div_distance
        """
        divScore = 0
        for i in range(len(population)):
                for j in range(len(population)):
                        if i ==j:
                                pass
                        else:
                                divScore = divScore + div_distance(population[i] ,population[j])
        return divScore

def div_distance(r1 , r2):
        """
        * this function generates a score representing the genomic diversity of two robots.
        * b is a value in  (0 , .03) , c is a value in (0 , 2 * np.pi).
        * because the variation will be larger for c we cannot use (c2-c1) + (b2-b1) as a
            ↪ determinant for diversity bc diff in c will outweigh diff in b
        * instead use  a normalized version (b2 - b1) / .03 + (c2 - c1) / 2*pi

        """

        diversity_score = 0
        array1 = r1.genome                              # get the genome of robot 1
        array2 = r2.genome                              # get the genome of robot 2
        length = np.size(array1 , 0)    # get the number of columns ( genome is an array of
            ↪ values (b,c))

        for i in range(length):
                b1 = array1[i][0]                                       # get the shape ( b) and
                    ↪ type (c ) vals for each robot
                c1 = array1[i][1]                                       #               ....
                b2 = array2[i][0]                                       #               ....
                c2 = array2[i][1]                                       #               ....

                val = abs(b2 -  b1) / 0.03 + abs(c2 - c1) / (2 *np.pi)          # get value of
                    ↪ interest for each spring
                diversity_score += val
                    ↪          # update the diversity score


        return diversity_score



# _______________ Optimization algorithms


def hill_climber():
        """
        * this function performs a hill climber on a single robot.
        *it will save the

        """
```

```
num_climb = 9999
num_tetr_added = 14
hc_fit = np.zeros(num_climb+1)
best_fit = 0
best_array = np.empty(8)
# make a random robot
robot = tetrahedron()
robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))                  # make
    ↪ the seed
robot.set_genome(num_tetr_added)                                                        #
    ↪  build the genome
robot.build_robot()                      # builds the robot and sets it about (0, val , 0)
# center the robot about the origin
mass_list = robot.mass_list
com = robot.get_com(mass_list)
robot.center_object(com)

best_fit = calc_fitness( robot)
#print("best fit" , best_fit)

hc_fit[0] = best_fit
for i in range(1 ,num_climb + 1):
        print('mut' , i)
        mut_loc , old_val = robot.mutate_genome()                              # this is
            ↪  a list for the coordinates of the item mutated
        robot.remove_old()
        robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))
        robot.build_robot()                      # builds the robot and sets it about (0,
            ↪ val , 0)
        # center the robot about the origin
        mass_list = robot.mass_list
        com = robot.get_com(mass_list)
        robot.center_object(com)

        fit_new = calc_fitness( robot)
        if fit_new > best_fit:
                best_fit = fit_new
                best_array = np.copy(robot.genome)

        else:
                # get robot back to old state
                robot.genome[mut_loc[0] , mut_loc[1]] = old_val
        hc_fit[i] = best_fit


a = robot.genome
```

```
        np.savetxt("hc2_robot.csv", best_array, delimiter=",")
        np.savetxt("hc2_help.csv", a, delimiter=",")
        np.savetxt("hc2_learning.csv", hc_fit, delimiter=",")




def random_search():
        """
        * this function will perform a random search for the best robot
        """

        num_search = 20
        num_tetr_added = 14

        rs_fit = np.zeros(num_search)
        best_fit = 0
        best_array = np.empty(8)
        for i in range(num_search):
                print('counter: ', i)
# make a random robot
                robot = tetrahedron()
                robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))                  #
                    ↪  make the seed
                robot.set_genome(num_tetr_added)                                #  build the
                    ↪ genome
                robot.build_robot()                     # builds the robot
                # center the robot about the origin
                mass_list = robot.mass_list
                com = robot.get_com(mass_list)
                #print( " com 1" , com)
                robot.center_object(com)
                # sim.set_robot(robot)
                # sim.show_masses()
                fitness = calc_fitness( robot)          # evaluate its fitness
                #print(" calc fit " , fitness)
                if fitness > best_fit:
                        best_fit = fitness
                        #print("update")
                        best_array = np.copy(robot.genome)
                        #print(best_array)


                #print(" best " , best_fit)
                rs_fit[i] = best_fit
                robot2 = tetrahedron()
                robot2.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))                 #
                    ↪  make the seed
                robot2.genome = best_array                                          # build
                    ↪ the genome
                robot2.build_robot()                    # builds the robot
                # center the robot about the origin
                mass_list = robot2.mass_list
                com2 = robot.get_com(mass_list)
                #print( " com 2" , com2)
                robot2.center_object(com2)

                #sim.show_masses()

                fitness2 = calc_fitness( robot2)                # evaluate its fitness
                #print(" second " , fitness2)

        np.savetxt("rs_robot.csv", best_array, delimiter=",")
        np.savetxt("rs_learning.csv", rs_fit, delimiter=",")




def simple_evolution():
        """
        * this function takes in an initial population of robots in the form of a list
        * it then performs evolutionary algorithm consisting of simple mutations and crossovers
        """
        num_tetr_added = 14
        population_size = 100                                           # how many robots
        num_mutations = 10                                                      # mutations
            ↪ between generation
        num_generations = 10                                                    # how many
            ↪ generations

        evals = population_size * num_generations * num_mutations
        diversity_vals = np.zeros(num_generations)

        overall_fitness = np.zeros(evals + 1)                   # the best fitness at each robot
            ↪ evaluation
        ticker = 0                                                                      # keeps
            ↪ track of which robot eval we are on
        parent_size = int(population_size / 2)
                                                        # make a simulator object
        parents = make_population(parent_size , num_tetr_added )                # make the
            ↪ population

        population_fitness = [0] * population_size
        for m in range(num_generations):
                print('generation: ' , m)
```

```
# crossover
                population = parents.copy()
                for k in range(parent_size):
                        p1 = np.random.randint(population_size/2)
                        p2 = np.random.randint(population_size/2)
                        while p2 == p1:
                                p2 = np.random.randint(population_size/2)               # note:
                                    ↪ if pop size = 2; stuck here forever

                        pr1 = parents[p1]
                        pr2 = parents[p2]
                        c = crossover(pr1 , pr2)
                        population = population + [c]

                # now we have a full population




                                # update the population fitness array ( only need to if there are
                                    ↪  zeros)
                #print(" before crossover " , population_fitness)
                for z in range(len(population_fitness)):
                        if population_fitness[z] ==0:
                                robot = population[z]
                                robot.remove_old()
                                robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))
                                robot.build_robot()                     # builds the robot and
                                    ↪ sets it about (0, val , 0)
                                # center the robot about the origin
                                mass_list = robot.mass_list
                                com = robot.get_com(mass_list)
                                robot.center_object(com)
                                population_fitness[z] = calc_fitness(robot)
                #print(" after crossover " ,population_fitness)
        # Mutation:
                diversity_vals[m] = diversity_val(population)                   # get diversity
                    ↪ before mutation
                for j in range(num_mutations):
                        print('mut' , j)

                        for p in range(population_size):
                                robot = population[p]

                                if ticker == 0:
```

```
                                overall_fitness[ticker] = population_fitness[p]
                                ticker += 1


                        print("robot # " , p)

                        mut_loc , old_val = robot.mutate_genome()
                            ↪         # this is a list for the coordinates of the item
                            ↪ mutated
                        robot.remove_old()
                        robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))
                        robot.build_robot()                     # builds the robot and
                            ↪ sets it about (0, val , 0)
                        # center the robot about the origin
                        mass_list = robot.mass_list
                        com = robot.get_com(mass_list)
                        robot.center_object(com)

                        fit_new = calc_fitness( robot)
                        #print(" fitness " , fit_new)

                        if fit_new > population_fitness[p]:
                                #print("here" , population_fitness[p] , fit_new)
                                population_fitness[p]= fit_new

                                best_array = np.copy(robot.genome)

                        else:
                                # get robot back to old state
                                robot.genome[mut_loc[0] , mut_loc[1]] = old_val


                        if overall_fitness[ticker - 1] < population_fitness[p]:
                                overall_fitness[ticker] = population_fitness[p]
                        else:
                                overall_fitness[ticker] = overall_fitness[ticker - 1]
                        ticker +=1
                        #print(" end of mut" , population_fitness)
# Selection:
        # sort the population in order

        bubbleSort(population , population_fitness)
        #print("after sort" , population_fitness)
        # use truncation selection
        parents = population[: parent_size]
        second_half = [0] * parent_size
        population_fitness = population_fitness[: parent_size] + second_half
        best_robot = population[0]
        # best_robot.remove_old()
        # best_robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))
        # best_robot.build_robot()                      # builds the robot and sets it about (0,
            ↪ val , 0)
        # # center the robot about the origin
        # mass_list = best_robot.mass_list
        # com = robot.get_com(mass_list)
        # best_robot.center_object(com)

        a = best_robot.genome
        np.savetxt("ea2_robot.csv", a, delimiter=",")
        np.savetxt("ea2_learning.csv", overall_fitness, delimiter=",")
        np.savetxt("ea2_diversity.csv", diversity_vals, delimiter=",")





def diverse_evolution():
        """
        * this function takes in an initial population of robots in the form of a list
        * it then performs evolutionary algorithm consisting of simple mutations and crossovers
        * this evolutionary algorithm is enhanced by adding diversity maintenance ( deterministic
            ↪  Crowding )
        """
        num_tetr_added = 14
        population_size = 50                                            # how many robots
        num_mutations = 6                                                       # mutations
            ↪ between generation
        num_generations = 10                                                    # how many
            ↪ generations

        evals = population_size * num_generations * num_mutations
        overall_fitness = np.zeros(evals + 1)                   # the best fitness at each robot
            ↪ evaluation
        ticker = 0                                                                      # keeps
            ↪ track of which robot eval we are on
        diversity_vals = np.zeros(num_generations)
        parent_size = int(population_size / 2)
        parents = make_population(parent_size , num_tetr_added)         # make the population

        population_fitness = [0] * population_size
        for m in range(num_generations):
                print('generation: ' , m)
```

```
# crossover:

                population = parents.copy()
                for k in range(parent_size):
                        p1 = k
                            ↪                         # want to ensure, each robot is the
                            ↪ primary parent for a child robot
                        p2 = np.random.randint(population_size/2)
                        while p2 == p1:
                                p2 = np.random.randint(population_size/2)               # note:
                                    ↪ if pop size = 2; stuck here forever

                        pr1 = parents[p1]
                        pr2 = parents[p2]
                        c = crossover(pr1 , pr2)
                        population = population + [c]




                # update the population fitness array ( only need to if there are zeros)
                #print(" before crossover" , population_fitness)
                for z in range(len(population_fitness)):
                        if population_fitness[z] ==0:
                                robot = population[z]
                                robot.remove_old()
                                robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))
                                robot.build_robot()                     # builds the robot and
                                    ↪ sets it about (0, val , 0)
                                # center the robot about the origin
                                mass_list = robot.mass_list
                                com = robot.get_com(mass_list)
                                robot.center_object(com)
                                population_fitness[z] = calc_fitness(robot)
                #print(" after crossover " , population_fitness)
                # now we have a full population
        # Mutation:
                diversity_vals[m] = diversity_val(population)                   # get diversity
                    ↪ before mutation
                for j in range(num_mutations):
                        print('mut' , j)

                        for p in range(population_size):
                                robot = population[p]

                                if ticker == 0:
                                        overall_fitness[ticker] = population_fitness[p]
        ticker += 1


print("robot # " , p)

mut_loc , old_val = robot.mutate_genome()
    ↪         # this is a list for the coordinates of the item
    ↪ mutated
robot.remove_old()
robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))
robot.build_robot()                     # builds the robot and
    ↪ sets it about (0, val , 0)
# center the robot about the origin
mass_list = robot.mass_list
com = robot.get_com(mass_list)
robot.center_object(com)
fit_new = calc_fitness(robot)
#print(" fitness , " , fit_new)
if fit_new > population_fitness[p]:
        population_fitness[p]= fit_new
        best_array = np.copy(robot.genome)

else:
                                        # get robot back to old
                                            ↪ state
        robot.genome[mut_loc[0] , mut_loc[1]] = old_val
```

```
                        if overall_fitness[ticker - 1] < population_fitness[p]:
                                overall_fitness[ticker] = population_fitness[p]
                        else:
                                overall_fitness[ticker] = overall_fitness[ticker - 1]
                        ticker +=1

                        #print(" after mut " , population_fitness)
# Selection:
        # use a selection process with diversity management
        for i in range(parent_size):
                if population_fitness[parent_size + i] > population_fitness[i]:         #
                    ↪  compare child to primary parent
                        parents[i] = population[parent_size +i]
                            ↪                                  # if child is better,
                            ↪ replace parent
                        population_fitness[i] = population_fitness[parent_size +i]
                        population[i] =  population[parent_size +i]
                population_fitness[parent_size +i] = 0

bubbleSort(population , population_fitness)
    ↪                  # this just to keep track of progress
#print("after sort " , population_fitness)

best_robot = population[0]
# best_robot.remove_old()
# best_robot.set_tetr( np.array((0 , 0 , 0)) , np.array((0 , 0 , 0)))
# best_robot.build_robot()                      # builds the robot and sets it about (0,
    ↪ val , 0)
# # center the robot about the origin
# mass_list = best_robot.mass_list
# com = robot.get_com(mass_list)
# best_robot.center_object(com)
#print(" in fucn " , calc_fitness(best_robot))
a = best_robot.genome
np.savetxt("div_ea_3k_best_robot2.csv", a, delimiter=",")
np.savetxt("div_ea_3k_learning2.csv", overall_fitness, delimiter=",")
np.savetxt("div_ea_3k_diversity2.csv", diversity_vals, delimiter=",")
```