Eduardo Mintzias (eam2285)
Pol Bernat Belenguer (pb2750)

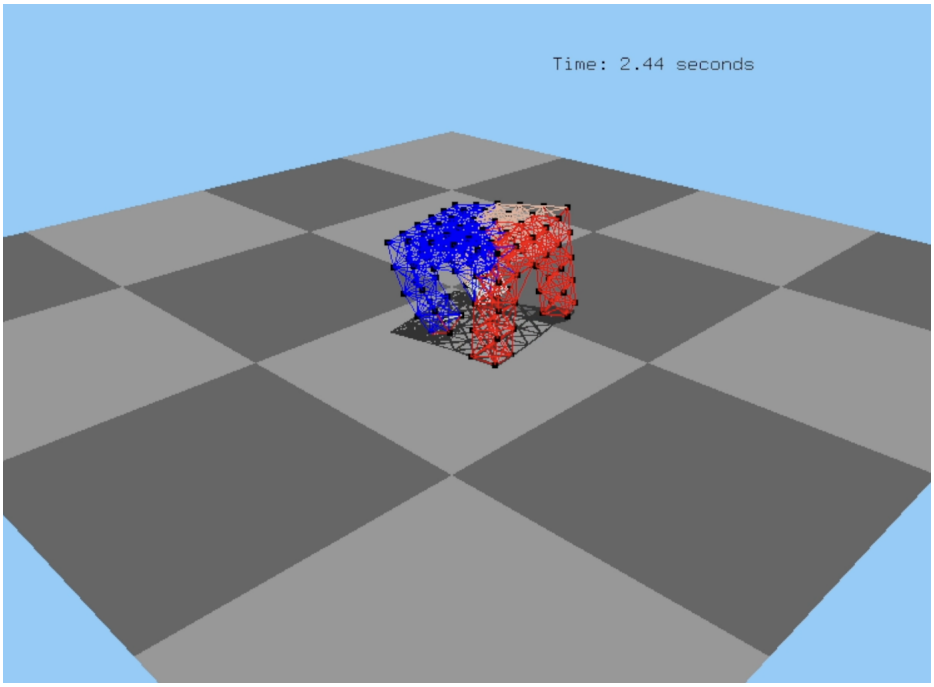
MECS 4510: EVOLUTIONARY COMPUTATION AND DESIGN AUTOMATION

Professor Hod Lipson

Date Submitted: 11/16/2023
Grace Hours Used: 45
Grace Hours Gained: 1
Grace Hours Remaining: 51

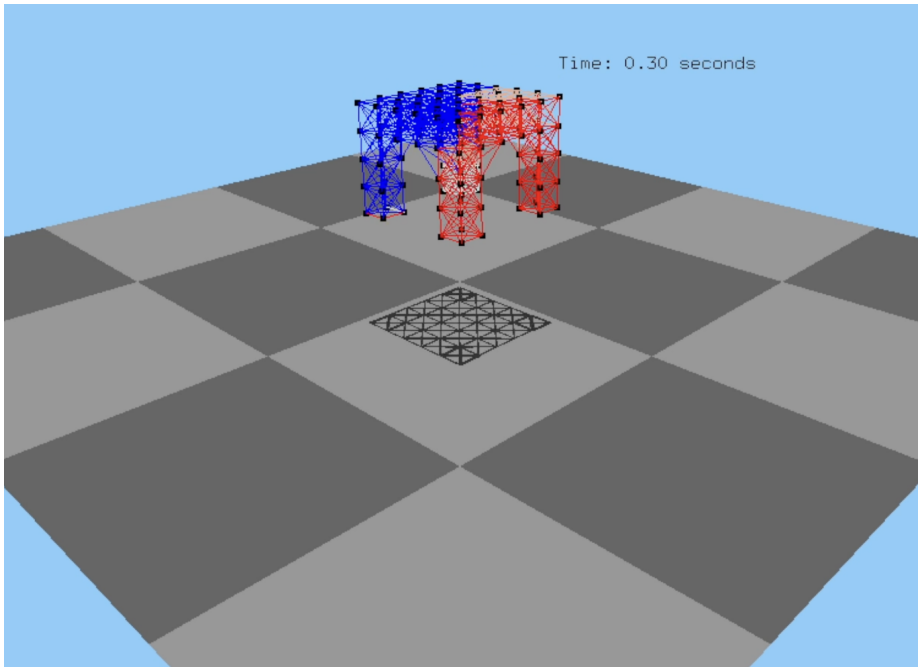
SUMMARY

VIDEO OF FASTEST ROBOT WALKING (4 CYCLES)



https://youtu.be/UOjn6RgGtDM

VIDEO OF FASTEST ROBOT BOUNCING ON THE GROUND



https://youtu.be/b0Hx9RR7q7I

As you can see from the above videos, we have made sure to have nice renderings of the **innovative robot design** that we came up with, we call it *table bot*

We are rendering each spring with a different color, representing each of the material types (white is bone, light orange is soft tissue, and red & blue are opposite muscles)

METHODS

DESCRIPTION OF ALL ROBOT PARAMETERS

Our robot's basic building blocks are Masses and Springs, which are defined with the following parameters:

MASS:

For each of the masses in our robot we keep track of the following information:

- 'm' - the mass in kg
- 'p' - the [x, y, z] position in meters
- 'v' - the [x, y, z] velocity in meters/second
- 'a' - the [x, y, z] acceleration in meters/second^2
- 'springs' - an array of all of the spring objects connected to that mass

SPRING:

For each of the springs in our robot we keep track of the following information:

- 'm1' - the first mass object the spring is connected to
- 'm2' - the second mass object the spring is connected to
- 'L0' - the rest length of the spring
- 'tissue_type' - an indicator of the tissue type (an int between 1-4, used to render the it correctly)
- 'k' - the spring constant, dependent on the 'tissue type'
- 'b' - the amplitude of oscillation for muscle 'tissue_types' that provides actuation
- 'c' - the phase shift in oscillation for muscle'tissue_types'
- 'center' - the [x, y, z] position of the center point of the spring

CUSTOM BODY:

Given the above definitions, we describe our robot in the following way:

- 'masses' - an array of all of the mass objects contained in that body
- 'springs' - an array of all of the spring objects connecting all of the masses (without repetition)
- 'tissue_dict' - a dictionary with the (k, b, c) values characteristic to each tissue_type (namely, bone has a high k & no b nor c, soft tissue has a low k & no b nor c, muscles have a medium k & values for b and/or c)
- 'genome' - we initialize springs to be a certain 'tissue_type' is by picking 8 random mass points in our body and, for all of the springs, calculating which is closest (by using the 'center' position of

each spring), we then assign the ‘tissue_type’ from the ‘tissue_dict’ as well as the ‘k’, ‘b’ and ‘c’ values to those springs - the genome is simply an array of the 3D point and the tissue type we are initializing fo our Custom body
- ‘COM’ - the Center-of-Mass value for our robot, used to evaluate the fitness in locomotion

DESCRIPTION OF ALL EVOLUTIONARY PARAMETERS

Currently only doing random search due to the runtime limitation of our simulation - we are simulating a structure of ~750 springs, the equivalent to ~28 cubes, with python - therefore, for this assignment we have only been able to apply a random search as we can not realistically complete more than 10s of evaluations even in 8 hours of running our code

Therefore, for now we are simply generating our fixed ‘Custom Body’ robots with the random ‘genome’ (ie. the positions where different tissue_types are initialized for our robot)

For the next assignment, once we have properly sped up our runtime with Numba, we will apply evolutionary parameters like mutation (changing the location where tissues are initalized to an adjacent mass location) and cross-over (splitting the genome at two locations and crossing over the description)

A thought about linkage - we will be reorganizing the genome to place points that are closer to each other closer in the genome, therefore when crossing over, genes that work well together are more likely to be passed on

The fitness function currently is the absolute distance the robot moves in the x-y plane from the initial position to the final position. We are planning to use co-evolution whereby we will be optimizing the muscle weights (values ‘k’, ‘b’ and ‘c’) as we optimize the morphology of the robot

DESCRIPTION OF WHAT WORKS AND DID NOT WORK

The main issue we are encountering is that our simulation is running very slow, therefore optimizing on such a large robot is extremely difficult and attempting fancier optimization techniques is simply unfeasible

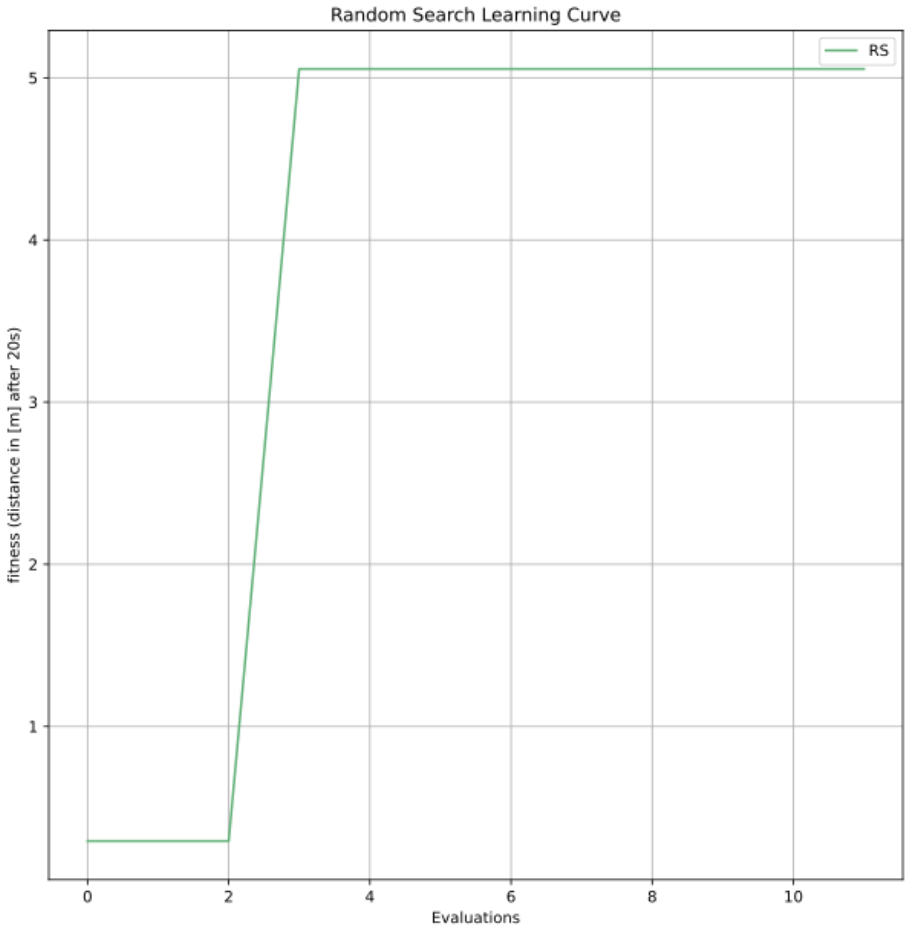
All of our functionality works and we are excited to implement Numba to be able to have our evolutionary algorithms shine

NUMBER OF SPRING EVALS/SECOND

~140,000 (on my personal laptop pre-NUMBA, running python)

RESULTS

LEARNING CURVE



BIBLIOGRAPHY

ChatGPT

PyOpenGL - https://pyopengl.sourceforge.net/index.html

Stack Overflow

APPENDIX

```
# %%
from Libraries import *

#%%
# Mass class
class Mass:
    def __init__(self, m, p, p_0 = np.zeros(3)):
        self.m = m
        self.p = np.array(p) + p_0
        self.v = np.array([0, 0, 0], dtype=np.float64)
        self.a = np.array([0, 0, 0], dtype=np.float64)
        self.springs = []

    def add_spring(self, spring):
        self.springs.append(spring)

# Spring class
class Spring:
    def __init__(self, m1, m2, k, tissue_type = None):
        self.m1 = m1
        self.m2 = m2
        self.L0 = np.linalg.norm(m1.p - m2.p)
        self.center = (m1.p+m2.p) /2

        self.tissue_type = tissue_type
        self.k = k
        self.b = 0
        self.c = 0

    def update_center(self):
        self.center = (self.m1.p+self.m2.p)/2


class Custom_body_1:
    def __init__(self, cube_size=0.1,  mass_value=0.1, k_value=9000, p_0=np.dot([0, 0,
0], 0.0), Genome_size = 8, prev_genome=None, only_bounce=False):
        self.fitness = 1e-7
        self.only_bounce = only_bounce
```

```
        # maps to properties (k,b,c)
        self.tissue_dict = {1:(1000,0,0), 2:(20000,0,0), 3:(5000,.125,0),
4:(5000,-.125,0)}
        self.reverse_tissue_dict = {value: key for key, value in
self.tissue_dict.items()}

        points = np.genfromtxt("table_body.txt", delimiter=',')
        #self.masses = np.zeros((len(points),3))
        masses_dict = defaultdict(lambda: None)

        for i,point in enumerate(points):
            masses_dict[tuple(point)] = Mass(mass_value, point * cube_size, p_0=p_0)

        self.masses = list(masses_dict.values())

        springs =[]
        for point in points:
            x,y,z = point
            cube_masses = [masses_dict[(x+dx, y+dy, z+dz)] for dx in range(2) for dy in
range(2) for dz in range(2)]
            cube_masses = [mass for mass in cube_masses if mass is not None]
            for i, mass1 in enumerate(cube_masses):
                for mass2 in cube_masses[i+1:]:
                    # Check if a spring already exists between these two masses
                    if not any(spring.m1 == mass1 and spring.m2 == mass2 or
                            spring.m1 == mass2 and spring.m2 == mass1 for spring in
mass1.springs):
                        spring = Spring(mass1, mass2, k_value)
                        springs.append(spring)
                        mass1.add_spring(spring)
                        mass2.add_spring(spring)
        self.springs = springs


        #Genome is a set of points with a corresponding tuple of properties (K,b,c)
        genome_points = [m.p for m in np.random.choice(self.masses, size=Genome_size,
replace=False)]

        if prev_genome is not None:
            self.genome = prev_genome
            for i in range(len(self.genome)):
                self.genome[i][0] += p_0
```

```
        else:
            self.genome = np.array([[pt ,
self.tissue_dict[np.random.choice([1,2,3,4])]] for pt in genome_points ])

        self.COM_update()
        self.Update_springs()



    def COM_update(self):
        rslt = np.zeros(3)
        total_mass = 0
        for mass in self.masses:
            rslt  += mass.p * mass.m
            total_mass += mass.m
        self.COM = rslt/total_mass
        return rslt/total_mass


    def Update_springs(self):
        for s in self.springs:
            s.update_center() #should not be necessary since this is called after
initialized but more robust here
            dist = [np.linalg.norm(s.center - g_pt) for g_pt in self.genome[:,0]]
            min_ind = np.argmin(dist)
            s.tissue_type = self.reverse_tissue_dict[tuple(self.genome[min_ind,1])]
            if self.only_bounce:
                s.k,s.b,s.c = self.genome[min_ind,1][0], 0, self.genome[min_ind,1][1]
                #s.b = 0
            else:
                s.k,s.b,s.c = self.genome[min_ind,1]

            pass


if __name__ == "__main__":
  table = Custom_body_1()
  #print(table.reverse_tissue_dict)
  #print(table.springs[0].center)

# %%
#lattice = CubeLattice(lattice_size=2, k_value=9000, p_0 = [0,0,0])
```

```
#print(lattice.genome)

genome = np.array([[[0.2, 0.4, 0.30000000000000004], [1000.0, 0.0, 0.0]], [[0.0, 0.2,
0.30000000000000004], [20000.0, 0.0, 0.0]], [[0.1, 0.30000000000000004, 0.4], [1000.0,
0.0, 0.0]], [[0.1, 0.0, 0.0], [20000.0, 0.0, 0.0]], [[0.4, 0.4, 0.2], [5000.0, -0.125,
0.0]], [[0.4, 0.5, 0.2], [20000.0, 0.0, 0.0]], [[0.4, 0.0, 0.0], [1000.0, 0.0, 0.0]],
[[0.1, 0.2, 0.4], [5000.0, -0.125, 0.0]]])

body = Custom_body_1(prev_genome = genome)



#%%
from Libraries import *
from Datastructures import Custom_body_1 as custom
from MAIN import *
#%%
class Simulate:
   def __init__(self, body):
       self.body = body
       self.Initial_pos = body.COM
       #GVLs
       self.b  = 0.999 # Dampening constant
       self.Kc = 100000  # Ground force constant
       self.G  = np.array([0, 0, -9.81])  # Gravity
       self.mu_s = 0.89 # Static friction
       self.mu_k = 0.70 # Kinetic friction
       self.dt = 0.00007  # Time-step
       self.T  = 0  # Global time variable
       self.omega = 2*np.pi
       self.global_step = 0
       self.sixty_Hz = int(0.01666 / self.dt) # dynamic plotting for 60 FPS
       self.four_Hz = int(.25 / self.dt)

   ########################  PLOTTING METHODS ####################################
   def draw_cube_faces(self):
       base_color = (0/255, 120/255, 200/255)  # Blue color
       border_color = (0, 0, 0)  # Black color for the border
       border_width = 1  # Width of the border
       for i, face in enumerate(self.body.faces):
           # Draw face
           glBegin(GL_QUADS)
```

```
            glColor3f(base_color[0], base_color[1], base_color[2])
            for mass in face:
                glVertex3fv(mass.p)
            glEnd()

            # Draw border
            glLineWidth(border_width)
            glColor3fv(border_color)
            glBegin(GL_LINES)
            for i in range(len(face)):
                glVertex3fv(face[i].p)
                glVertex3fv(face[(i + 1) % len(face)].p)  # Loop back to the start for
the last line
            glEnd()

    def draw_masses_and_springs(self):
        mass_color = (0, 0, 0)
        spring_color = {1:(232/255, 177/255, 155/255), 2:(206/255, 222/255, 220/255),
3:(1, 0, 0), 4:(0, 0, 1)}  # Color springs for different types
        mass_size = 5  # Size of the mass points
        spring_width = 1  # Width of the springs
        # Draw masses
        glPointSize(mass_size)
        glColor3fv(mass_color)
        glBegin(GL_POINTS)
        for mass in self.body.masses:
            glVertex3fv(mass.p)
        glEnd()
        # Draw springs
        glLineWidth(spring_width)
        glBegin(GL_LINES)
        for spring in self.body.springs:
            glColor3fv(spring_color[spring.tissue_type])
            glVertex3fv(spring.m1.p)
            glVertex3fv(spring.m2.p)
        glEnd()

    def draw_shadow(self):
        shadow_color = (0.2, 0.2, 0.2)  # Dark gray color for shadow
        spring_shadow_width = 1  # Width of the spring shadows
        # Set the color and line width for the shadows
        glColor3fv(shadow_color)
```

```
        glLineWidth(spring_shadow_width)
        # Draw shadows of the springs
        glBegin(GL_LINES)
        for spring in self.body.springs:
            # Project the mass positions onto the ground plane (z = 0)
            glVertex3f(spring.m1.p[0], spring.m1.p[1], -0.001)
            glVertex3f(spring.m2.p[0], spring.m2.p[1], -0.001)
        glEnd()

    def draw_ground(self):
        dark_grey = (0.4, 0.4, 0.4)  # Dark grey color
        light_grey = (0.6, 0.6, 0.6)  # Lighter grey color
        side_length = 1  # Length of the side of each square
        num_squares = 4  # Number of squares in each row and column
        glBegin(GL_QUADS)
        for i in range(num_squares):
            for j in range(num_squares):
                # Determine the color of the square
                if (i + j) % 2 == 0:
                    glColor3fv(light_grey)
                else:
                    glColor3fv(dark_grey)
                # Calculate the coordinates of the square
                x1 = i * side_length - 1.45
                y1 = j * side_length - 1.45
                x2 = x1 + side_length
                y2 = y1 + side_length
                # Draw the square
                glVertex3f(x1, y1, -0.005)
                glVertex3f(x2, y1, -0.005)
                glVertex3f(x2, y2, -0.005)
                glVertex3f(x1, y2, -0.005)
        glEnd()

    def render_text(self, x, y, text):
        glMatrixMode(GL_PROJECTION)
        glPushMatrix()
        glLoadIdentity()
        glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0)
        glMatrixMode(GL_MODELVIEW)
        glPushMatrix()
        glLoadIdentity()
```

```
        glColor3f(0, 0, 0)
        glRasterPos2f(x, y)
        for char in text:
            glutBitmapCharacter(GLUT_BITMAP_9_BY_15, ord(char))
        glPopMatrix()
        glMatrixMode(GL_PROJECTION)
        glPopMatrix()
        glMatrixMode(GL_MODELVIEW)

    def draw_cube(self):
        # Draw shadow first
        self.draw_shadow()
        # Then draw the cube's faces
        #draw_cube_faces(self.body)
        self.draw_masses_and_springs()

    ########################## RUN METHODS ###################################
    # TODO run faster!
    def update_mass(self, mass):
        # Initial force is 0
        # GRAVITATIONAL FORCE
        # Update F
        F = mass.m * self.G
        # SPRING FORCE
        # Loop over all springs
        for spring in mass.springs:
            # If spring connected to that mass
            if spring.m1 == mass or spring.m2 == mass:
                # Calculate spring force and actuator length based on time
                L = np.linalg.norm(spring.m1.p - spring.m2.p)
                Lo = spring.L0*(1+spring.b * np.sin(self.omega*self.T + spring.c))
                # Update F_spring in the vector direction from m2 to m1 (unit vect
(p1-p1)/dist)
                F_spring = spring.k * (L - Lo) * (spring.m1.p - spring.m2.p) / L

                # Update F with appropriate sign
                if spring.m1 == mass:
                    F -= F_spring
                else:
                    F += F_spring
        # FRICTION FORCE
        if mass.p[2] <= 0:
```

```
            Fp = [F[0], F[1], 0]
            Fn = [0, 0, F[2]]
            if np.linalg.norm(Fp) < np.linalg.norm(Fn)*self.mu_s:
                F -= Fp
            else:
                F += Fp/np.linalg.norm(Fp)*(-1)*np.linalg.norm(Fn)*self.mu_k
        # GROUND COLLISION FORCE
        # Ground collision check and response
        if mass.p[2] < 0:
            F += np.array([0, 0, -self.Kc * mass.p[2]])

        # UPDATE ACCELERATION
        mass.a = F / mass.m
        # UPDATE VELOCITY
        mass.v += mass.a * self.dt
        mass.v *= self.b # dampening
        # UPDATE POSITION
        mass.p += mass.v * self.dt
        return mass

    def Body_Advance_step(self):
        #Update masses
        self.body.masses = [self.update_mass(mass) for mass in self.body.masses]

        #update simulation
        self.T+=self.dt
        self.global_step +=1
        pass

    def evaluate(self, T = .05):
        dist = np.linalg.norm(self.Initial_pos - self.body.COM_update())
        return dist

    def print_update(self):
        out = f'Time = {round(self.T,2)}  |  '
        out+= f'Position  = {np.round(self.body.masses[72].p,2)}  |  '
        out+= f'Vel  = {np.round(self.body.masses[72].v,2)}  |  '
        print(out)

    def plot_frame(self):
        pygame.init()
        #glutInit()
```

```
        display = (800,600)
        pygame.display.set_mode(display, DOUBLEBUF|OPENGL)
        gluPerspective(45, (display[0]/display[1]), 0.1, 100.0)
        gluLookAt(-3.5, -3.5, 2.5, 0, 0, 0, 0, 0, 1)
        glClearColor(0.53, 0.81, 0.98, 1)
        glDisable(GL_CULL_FACE)
        glEnable(GL_DEPTH_TEST)
        for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        pygame.quit()
                        return

        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
        self.draw_ground()
        self.draw_cube()

        # Render the text in the top-right corner using GLUT
        text_string = f"Time: {self.T:.2f} seconds"
        self.render_text(0.6, 0.9, text_string)

        pygame.display.flip()
        #pygame.time.wait(10)

    def run_simulation(self, Plot = False, Actuator_on = False, Verbose = False, max_T
= 1):

        if Plot:
            start_time = time.time()
            pygame.init()
            #glutInit()
            display = (800,600)
            pygame.display.set_mode(display, DOUBLEBUF|OPENGL)
            gluPerspective(45, (display[0]/display[1]), 0.1, 100.0)
            gluLookAt(-2., -2., 1.5, 0, 0, 0, 0, 0, 1)
            glClearColor(0.53, 0.81, 0.98, 1)
            glDisable(GL_CULL_FACE)
            glEnable(GL_DEPTH_TEST)

            while True:
                #__________SIMULATION _________
                # Loop over all masses and update them, advance simulation one step
                #start_time = time.time()
```

```
                self.Body_Advance_step()
                #print(f"Running {len(self.body.springs)/(time.time() -
start_time):.6f} springs/sec")

                # _________PLOTING_________
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        pygame.quit()
                        return

                # _________RENDERING_________
                if self.global_step % 100 == 1:
                    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
                    self.draw_ground()
                    self.draw_cube()
                    # Render the text in the top-right corner using GLUT
                    text_string = f"Time: {self.T:.2f} seconds"
                    self.render_text(0.6, 0.9, text_string)
                    pygame.display.flip()
                    #pygame.time.wait(10)
                if self.T> max_T:
                    print(self.T)
                    print(f"Took {(time.time() - start_time):.6f} sec for {self.T} sec
in sim")
                    break


        else:
            # Run simulation without plotting
            start_time = time.time()
            while True:
                self.Body_Advance_step()

                if Verbose and (self.global_step % self.sixty_Hz == 1):
                     self.print_update()

                if self.T> max_T:
                    print(f"Took {(time.time() - start_time):.6f} sec for {self.T} sec
in sim")
                    break
        #desired output
        return self.evaluate()
```

```
if __name__ == "__main__":
    with open('1700148827.3789582_random_search.pkl', 'rb') as file:
        data = pickle.load(file)
    #print(data[1])
    genome = data[0][5].genome
    #genome = data[0][0].genome
    NEW_BODY = custom(p_0=[0, 0, 0.7], prev_genome=np.array(genome), only_bounce=True)
    sim1 = Simulate(body = NEW_BODY)
    fitness = sim1.run_simulation(Plot = True, Verbose = True, max_T = 20)
    #cProfile.run('main(cubes)', 'profiling.out')
    #print(fitness)
    print('done')


#%%
from Libraries_cloud import *
from Datastructures_cloud import *
from Simulation_cloud import *

#%%
class EvolvingGait:
    def __init__(self, pop_size):
        self.pop_size = pop_size
        self.population = np.empty(pop_size, dtype=object)
        self.fitnesses = np.zeros(pop_size, dtype=float)

        self.random_population()
        self.update_pop_fitness()

    # Populate with random bodies
    def random_population(self):
        for i in range(self.pop_size):
            self.population[i] = self.random_individual()

    def random_individual(self):
        return Custom_body_1()

    def update_pop_fitness(self):
```

```
        for i in tqdm(range(self.pop_size), desc='Evaluating:'):
            self.fitnesses[i] =
Simulate(body=self.population[i]).run_simulation(Plot=False, max_T=2)

    def fitness_prop_selection(self):
        pass

    def mutate(self):
        pass

    def crossover(self):
        pass

    def results(self):
        return [self.population, self.fitnesses]


if __name__ == "__main__":

    t = EvolvingGait(pop_size = 12)

    pop_n_fit = t.results()

    filename = '{}_random_search.pkl'.format(time.time())
    with open(filename, 'wb') as file:
        pickle.dump(pop_n_fit, file)
```