Projet de Simulation robotique

Juyeon KIM
Master SAR
Student number 28611849

I. INTRODUCTION

The main objective of this project is to build a simulation using classes and methods. The project consists of creating Paricle, Univers, and Generators Classes used for various mechanical simulations. This report aims to present the different codes, and architectures used and the different successes and failures of the project. The source code of the project is here https://github.com/judy3378/Simulation-Physique.git. Prior to code implementation, we first created a virtual environment where we installed pyGame and pylab.

II. INFRASTRUCTURE OF MECHANICAL SIMULATION

`Class Particle`, `Univers` and force generators can be found in `Particule.py`.

A. Class Particle

The Class Particle contains methods to compute the position, velocity, and acceleration of the particles, as well as drawing the elements into the Pygame screen.

A particle is initialized with its initial position 'pos0', velocity 'vit0', mass, name, color and whether it is affected by the forces, 'fix'. Acceleration of the particles and forces in the universe are in a list.

```
def __init__(self, pos0 = Vecteur3D(),
    vit0=Vecteur3D(), mass = 0, name = 'p1',
    color = 'blue', fix=False):
      """attribute 'fix' can be used to not
          adhere to the PFD"""
      self.position = [pos0]
      self.velocity=[vit0]
      self.acc = [Vecteur3D()]
      self.forces = [Vecteur3D()]
      self.mass = mass
      self.name = name
      self.color = color
      self.fix = fix
```

`getPosition`, `getSpeed` returns current position and velocity. They are used to update the next position of the particle :

```
def move (self,step):
   """for a time 'step', resolves the PFD
       with external forces sum, then
       calculates acceleration, velocity and
       position"""
   V = Vecteur3D()
   V = self.getSpeed()
   self.velocity += [V+self.acc[-1]* step]
   self.position += [V * step +
       0.5*self.acc[-1]*step**2 +
       self.getPosition()]
```

The following methods are used to set forces in the universe, velocity and position of the particle disregarding the PFD.

```
def setForce(self, force = Vecteur3D()):
   """adds a force to a list of forces"""
   self.forces.append(force)

def setSpeed(self, speed=Vecteur3D()):
   """disregards PFD and enforces velocity
       only when 'fix'=True"""
   self.velocity = speed

def setPosition(self,
    position=Vecteur3D()):
   """disregards PFD and enforces position
       only when 'fix'=True"""
   self.position = position

def getForces(self):
   """returns the last force applied"""
   return self.forces[-1]

def getAcc(self):
   """returns the current acceleration"""
   a = Vecteur3D()
   a = self.getForces() * (1/self.mass)
   self.acc.append(a)
```

In order to see the trajectories of the particles in 2D and 3D, we use the following methods `plot` and `plot3d` :

```
def plot(self):
   """plots 2D trajectory"""
   X, Y, Z = zip(*[(p.x, p.y, p.z) for p
       in self.position])
   return plot(X, Z, color=self.color,
       label=self.name) #Or X,Y,

def plot3d(self):
   """3d plot"""
   fig = plt.figure()
   ax=fig.add_subplot(111,projection='3d')

   X, Y, Z = zip(*[(p.x, p.y, p.z) for p
       in self.position])
   ax.scatter3D(X,Y,Z,color=self.color,
       label=self.name)

   ax.set_xlabel('X')
   ax.set_ylabel('Y')
   ax.set_zlabel('Z')
```

```
        ax.set_title('3D Trajectory')

        plt.legend()
        plt.show()
```

The method `gameDraw` draws the circles, representing the particles.

```
def gameDraw(self, screen, scale):
    size = 5
    pos = self.getPosition()
    X, Y, Z = int(scale * pos.x), int(scale *
        pos.y), int(scale * pos.z)
    r, g, b = random(), random(), random()
    rgb = (r, g, b, 1)
    pygame.draw.circle(screen, rgb, (X, Y),
        size * 2, size)

    # draw lines of director vectors
    vit = self.getSpeed()
    VX, VY = int(scale * vit.x) + X, int(scale
        * vit.y) + Y
    pygame.draw.line(screen, self.color, (X,
        Y), (VX, VY))
```

```
def __str__(self):
    return f"Particle {self.name}: Position =
        {self.position}, Velocity =
        {self.velocity}"

def __repr__(self):
    return self.__str__()
```

B. Class Universe

Class Universe contains methods to create the scene where particles interact, to add particles and forces to the environment, to run the simulation, and to update according to the changes such as new particles or forces. We initialize the dimensions of the screen to (1024, 768) pixels, which we will modify later according to the tasks.

```
class Univers(object) :

def __init__(self, name="la plage", t0=0,
    step=0.1, dimensions=(1024,768), scale=1,
    *args):
    self.name = name
    self.temps = [t0]
    self.population = []
    self.step = step
    self.dimensions = dimensions
    self.listForce = []
    self.scale=scale
```

We can add particles, namely Agent, and forces, namely Source.

```
def addAgent(self, *agents):
    """add one or multiple agents"""
    self.population.extend(agents)

def addSource(self, *generators):
    """add one or multiple sources/forces"""
    self.listForce.extend(generators)
```

We then perform a simulation for one step (time of simulation) for the entire population. We apply force to the particle which is not fixed.

```
    def simule(self):
        """perform a simulation for one step
            for the entire population"""
        for part in self.population:
            totalForce = Vecteur3D()
            for f in self.listForce:
                totalForce += f.apply(part)
            if part.fix==False:
                part.setForce(totalForce)
                part.getAcc()
                part.move(self.step)
            else:
                part.move(self.step)
        self.temps.append(self.temps[-1]+self.step)

    def simuleAll(self, time):
        """perform a simulation for a duration
            of 'time'=> multiple simule()
            steps"""
        while self.temps[-1] <time:
            self.simule()
```

We plot the trajectories of particles in the population using the `plot` method from class Particle.

```
    def plot(self):
        for a in self.population:
            a.plot()

    def plot3d(self):
        for a in self.population:
            a.plot3d()
```

We have `gameInit` method to start the simulation with display in a Pygame window where we set the framerate to 60 fps, meaning the screen updates 60 times per second, the grey background color. `gameUpdate` method handles keyboard/mouse input and simulates display for a step of simulation time.

```
    def gameInit(self):
        """ Start a real-time simulation with
            display in a Pygame window.
            set the display framerate to
                60fps."""
        pygame.init()
        self.t0 = time.time()
        self.screen =
            pygame.display.set_mode(self.dimensions)
        self.clock = pygame.time.Clock()
        self.background=(200,200,200)
        self.fps=60
        self.run=True
        self.gameKeys = pygame.key.get_pressed()

    def gameUpdate(self):
        """handles keyboard/mouse input and
```

```
        simulates display for a step"""
    now = time.time()-self.t0
    while self.temps[-1] < now:
       self.simule()
    self.screen.fill(self.background)
    # display current time
    font_obj =
        pygame.font.Font('freesansbold.ttf',
        24)
    text_surface_obj =
        font_obj.render(f'{now:.2f}', True,
        'red', self.background)
    text_rect_obj =
        text_surface_obj.get_rect()
    text_rect_obj.center = (25, 10)
    self.screen.blit(text_surface_obj,
        (50,20))
    # display particles
    for p in self.population:
       p.gameDraw(self.screen,self.scale)
    pygame.display.update()
    self.clock.tick(self.fps)
    # check for window close event
    for event in pygame.event.get():
       if event.type == pygame.QUIT:
          self.run = False
    pygame.display.flip
    self.gameKeys = pygame.key.get_pressed()
    #update particle forces based on
        keyboard input or mouse
```

C. Class Generators

Class Generators consists of various types of forces that are added to the environment. There are constant force, harmonic force, viscosity, field force, spring dumper, a rod. Each class of force consists of a method `__init__`to initialize containing information about the specificities of forces such as the norm, coefficients, and also the particles to be affected, and another method `apply` to apply the force to the particle.

```
class ForceConst(object):
   def __init__(self, value = Vecteur3D(),
       *particles):
      self.value = value
      self.particles = particles

   def apply(self, p):
      return self.value
```

```
class ForceHarmonic(object):
   def __init__(self, value=Vecteur3D(),
       pulsation =0, *particles):
      self.value = value
      self.pulsation = pulsation
      self.particles = particles

   def apply(self, p):
      return self.value *
          math.cos(self.pulsation*time.time())
```

```
class Viscosity(object):
   def __init__(self, coef = 0, *particles):
      self.coef = coef
      self.particles = particles

   def apply(self, p):
      return self.coef * (-p.getSpeed())
```

```
class ForceField(object):
   def __init__(self, pos0=Vecteur3D(),
       amplitude = Vecteur3D(), *particles):
      self.pos0 = pos0
      self.amplitude = amplitude
      self.particles = particles

   def apply(self, p):
      displacement = self.pos0 -
          p.getPosition()
      return self.amplitude*displacement
```

Spring damper:

$$F_x = -kx - cv_x$$

$$F_y = -ky - cv_y$$

where x is the displacement from the rest length, k is the spring constant, cv_x is a damping force proportional to the velocity of the particle.

```
class dampingSpring(object):
   def __init__(self, p0, p1, k=0, c=0, l0=0):
      self.k = k #stiffness
      self.c = c #damping
      self.l0 = l0
      self.p0 = p0
      self.p1 = p1

   def apply(self, p):
      if p is self.p0:
         dist = self.p1.getPosition() -
             self.p0.getPosition() # mass
             position
         vit = (self.p1.getSpeed() -
             p.getSpeed()) ** dist

      elif p is self.p1:
         dist = self.p0.getPosition() -
             self.p1.getPosition() #position
             de la masse
         vit = (self.p0.getSpeed() -
             p.getSpeed()) ** dist

      else:
         return Vecteur3D()

      Fr = (dist.mod() - self.l0) * self.k
          #force de rappel
      Fv = self.c * vit #force de viscosite
      Fnormalise = (Fr+Fv)*dist.norm()

      force =((dist.mod() - self.l0) * self.k
          + self.c * vit ) * dist.norm()

      return force
```

```
class Rod(object):
    def __init__(self, particle1=None,
        particle2=None):
        self.particle1 = particle1
        self.particle2 = particle2
        self.stiffness = 0
        self.dumping = 0

    def apply(self,p):
        return dampingSpring(self.stiffness,
            self.dumping, 0, self.particle1,
            self.particle2).apply(p)
```

```
class PrismJoint(object):
    def __init__(self, axis=Vecteur3D(),
        particle=None):
        self.axis = axis # direction of the
            joint
        self.particle = particle

    def apply(self, p):
        displacement = p.getPosition() -
            self.particle.getPosition()
        projected_displacement = displacement -
            displacement.proj(self.axis)
        return projected_displacement *
            -self.particle.mass*9.81 #assuming
            gravity
```

III. Validation

A. Task 1: Free fall with magnetic force in the center

In a scene of 10m x 7m, 10 particles of mass and position (x,y), randomly distributed, are in free fall along the -z-axis, with an attractive force field in the center of the screen at z = - 5m. Within the main of Task1.py :

```
    # set up screen
    # top left corner is (0,0) top right
        (1000,0) bottom left (0,700)
    # bottom right (1000,700).
    scene_width = 10
    scene_height = 7
    screen = pygame.display.set_mode((1000,
        700))
```

Then we create the Universe named `plage`:

```
Univers(step = 0.01, scale = 100)
```

Forces are added into the Universe as follows:

```
# Adding an attractive field force centered
    at z=-5m
center_force =
    ForceField(Vecteur3D(scene_width/2,
    scene_height/2, -5), amplitude=5)
plage.addSource(center_force)
# Adding gravity
plage.addSource(Gravity(Vecteur3D(0, 0,
    -9.8)))
```

Ten particles are added in random positions and colors:

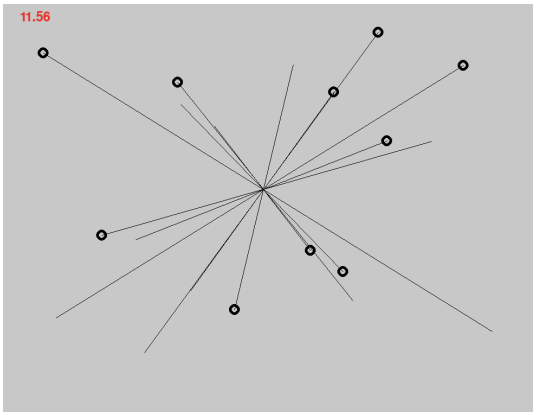


Fig. 1. Task1: 2D simulation in (x,y) of particles represented in circles and their director vectors in lines with simulation time on the top left.

```
# Parameters for the particles
num_particles = 10
# Creating random particles
for _ in range(num_particles):
    name = 'Particle'+str(_)
    x = random()*scene_width
    y = random()*scene_height
    z = random()*pi*2
    r = random()
    g = random()
    b = random()
    rgb = (r,g,b,1)
    position = Vecteur3D(x, y, z)
    particle = Particule(pos0=position,
        name=name, mass=1, color=rgb)
    plage.addAgent(particle)
```

Running the simulation is as follows :

```
(py311) mac:simphysique 23 kimjuyeon$ python
    run_task1.py
pygame 2.3.0 (SDL 2.26.5, Python 3.11.3)
Hello from the pygame community.
    https://www.pygame.org/contribute.html
--------------------------------
Usage:
Press (q) to quit simulation.
and see figure of trajectories.
--------------------------------
2023-09-12 07:43:38.421 python[87441:1863132]
    TSM
    AdjustCapsLockLEDForKeyTransitionHandling
    - _ISSetPhysicalKeyboardCapsLockLED
    Inhibit
```

Figure 1 shows ten hollow circles representing particles sparsed on the (x,y) plane screen. At that moment, particles are going toward the center of the screen, as the lines, which represent their directions, are going inwards. Trajectories of ten particles in the (x,z) plane after simulation are presented in Figure 2. We observe that the particles come together at a point located slightly below the z = -5 m due to the effect of gravity force. Figure 3 presents the trajectory of one particle in 3d.

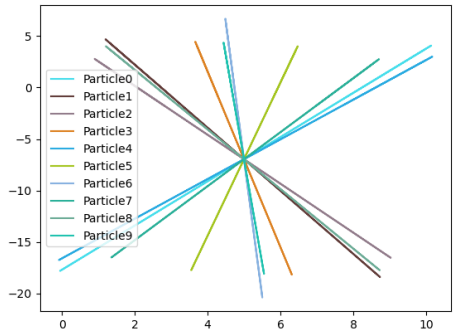


Fig. 2. Task 1: 2D plot in (x,z) of ten particles' trajectories.

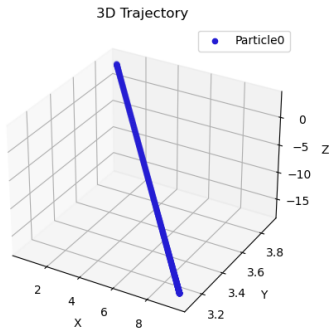


Fig. 3. Task 1: 3D plot of particle 0's trajectory.

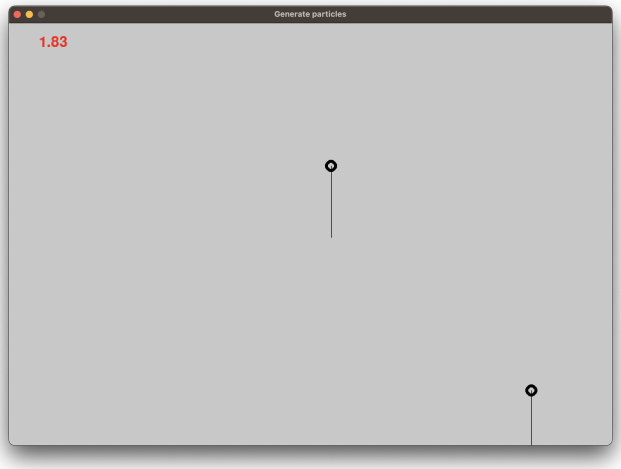
B. Task 2: Viscosity and generation of particles

In a scene of 100mx70m of (x,y,) plane, the particle is under the force of gravity and viscosity.

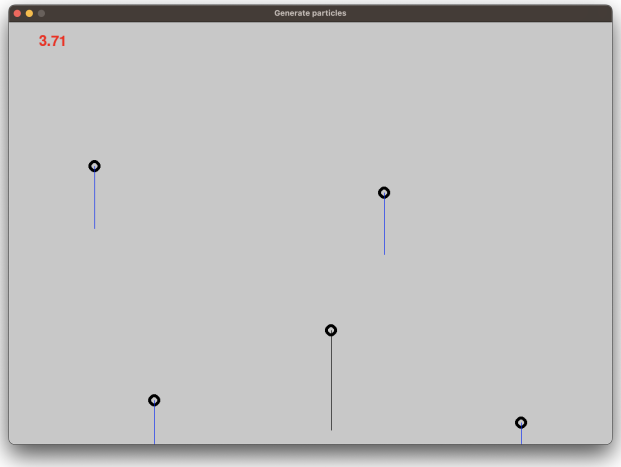
```
    scale=10
    scene_width = 100
    scene_height = 70
    plage = Univers(step=0.01,
        dimensions=(scene_width*scale,
        scene_height*scale), scale=scale)
    # add gravity and viscosity
    plage.addSource(Viscosity(0.5))
    plage.addSource(Gravity(Vecteur3D(0,9.8,0)))
```

To add a particle, (space) is pressed. However, the key is pressed multiple times on the computer, therefore multiple particles are created at once (see Figure 4). Attempts were made to solve this problem, however, the proposed code can generate either multiple particles at a time or a single particle at each simulation.

```
while plage.run:
    clock.tick(60)
    plage.gameUpdate()
    if plage.gameKeys[K_SPACE]:
        particle =
            Particule(pos0=Vecteur3D(random()*
                scene_width, random()*scene_height,
                random()*pi*2), mass=1)
        plage.addAgent(particle)
```



(a) Initially two particles.



(b) Three particles are added.

Fig. 4. Task 2: Generated particles under gravity and viscosity force.

We observe that the particles are drawn towards the negative y-axis in Figure 5. The velocity of a particle along the y-axis increases non-linearly (slightly curved in Figure 6) due to the effect of viscosity coupled with gravity.

C. Task 3: Mass-spring-damper system with constant force or harmonic force

The mass experiences a damping force that is proportional to its current velocity (see Figure 7), which mathematically is $F = -kx - cv$, where c is the damping constant.

D. Task 4: Three pendulums

The goal is to simulate a pendulum of varying rest length $l = 10, 20, 30$cm. The idea is to add one particle p_0 that is fixed, and a second particle p_1 so that the

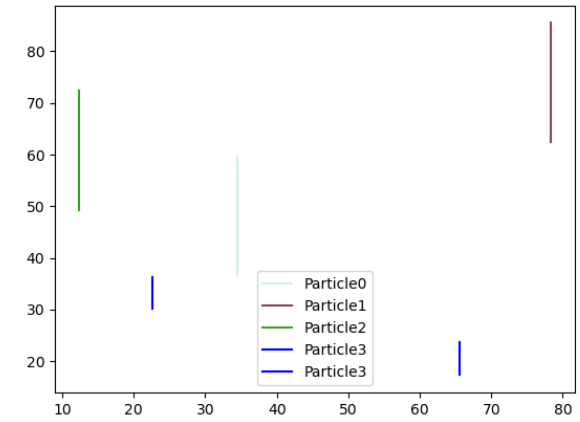


Fig. 5. Task 2: Plot (x,y) of particles. Particle3 are generated. **Different simulation that Fig. 4*

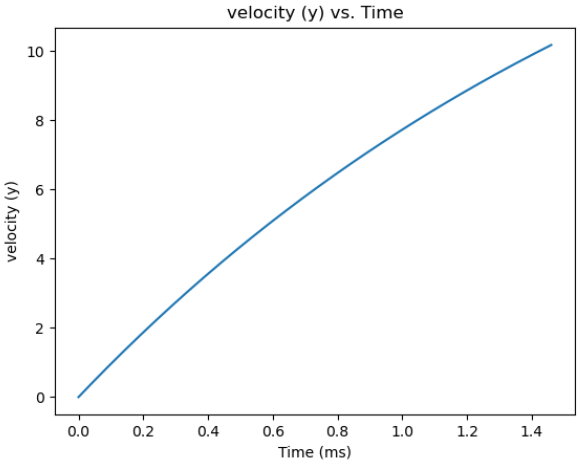


Fig. 6. Task 2: Velocity (y) vs. time of a particle

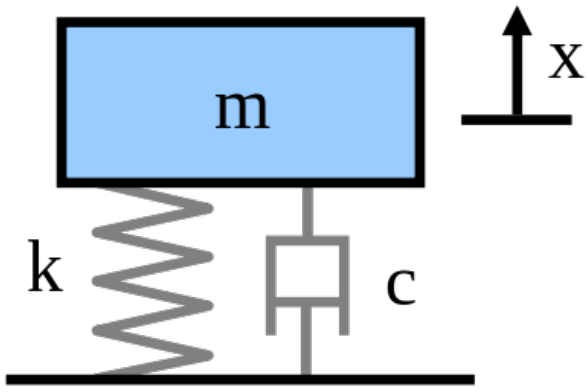


Fig. 7. Mass Spring Damper

distance between two particles is: $l = |p_1 - p_0|$, equivalent to `dist = self.p1.getPosition() - self.p0.getPosition`. Then we add gravity and a rod, which can be considered as a damping spring with 0 stiffness and damping.

E. Task 5: Double pendulum

F. Task 6: 2 DOF system with two masses and three springs

G. Task 7: Inverted pendulum on a moving base

IV. Conclusion and perspectives

We build simulations to validate a model or a theory. It allows us to perform experiments that are too expensive or dangerous to carry out in real life. We mostly worked with continuous system simulations where time is the continuous variable as well as the speed of the particles. From a technical point of view, pyGame is a visual and interactive framework for simulations. It is possible to set the time using a clock object to ensure that the simulation is the same on all machines, regardless of the actual machine speed. Users can also add or eventually remove particles from the environment. The project can be further developed by adding keys to pause and resume simulation, observing step by step.

V. Évaluation et commentaires