Odd Semester (2021)



**BINUS UNIVERSITY**

**BINUS INTERNATIONAL**

**Assignment Cover Letter**

**(Individual Work)**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | |  | |  | |
| **Student Information**: **Surname** | | | | | **Given Names**  **Samuel** | | **Student ID Number**  **2101693630** | |
| 1. | | **Putra** |  | |
|  |  |
| **Course Code** | **: COMP6065** |  |  | | **Course Name** | | **: Introduction to Programming** | |
| **Class** | **: L1BC** |  |  | | **Name of Lecturer(s)** | | **:** 1. Ida Bagus Kerthyayana | |
|  |  |  |  | |  | | 2. Minaldi Loeis  3. Jude Joseph Lamug Martinez | |
| **Major** | **: CS** |  |  | |  | |  | |
| **Title of Assignment**  (if any) | : Brachistochrone Simulator | |  |  | |  | |  | |
| **Type of Assignment**    **Submission Pattern** | **: Final Project** |  |  | |  | |  | |
| **Due Date** | **: 6-11-2017** |  |  | | **Submission Date** | | **: 6-11-2017** | |

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer’s instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

# Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

# Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: (Name of Student)

1. Samuel Putra

**“Brachistochrone Simulator”**

**Name : Samuel Christian Veda Putra**

**ID : 2101693630**

1. **Description**

**The function of this program:**

The purpose of this program is to simulate and visualize the trajectory of an object from point A to point B at different heights, calculate and visualize the path with the shortest amount of time, given that the object moving from point A to point B experiences a constant gravitational acceleration with no friction.

This program also compares the time travelled with different gravitational acceleration from different planets and on top of that, the user can add their own planet to simulate various kinds of gravitational acceleration. To give perspective to the user, this program will also plot several other trajectories and compare their times.

**II.a. Design/Plan**

**Project’s Hierarchy Chart**

Plot the Final Graph **(*main.py*)**

Main Menu **(*main.py*)**

Add Planet **(*main.py*)**

Run Simulation **(*main.py*)**

Exit **(*main.py*)**

Show Planet Details **(*main.py*)**

Add Planet Attributes **(*main.py*)**

Get Value of Gravity **(*main.py*)**

Display Planet’s Attributes **(*main.py*)**

Planets Class **(*planets.py*)**

Compute the Formula **(*functions.py*)**

**II.b. Explanation of Each function**

**Main Menu: ( *main.py* )**

* **Outside main ():**
* Imports all the modules and classes; “functions”, “planets”, “matplotlib”, “numpy”, “time”.
* Initializes the class “Planets()” as a variable called “planets”.
* Sets the end points of the bead as “x2” and “y2”.
* Create message variables; “welcome\_message”, “exit\_message” and “main\_menu” to be printed in the main() function later on.
* Assign the current list of planets to a variable called “planets\_list”.
* **Inside animate (i):**
* This function iterates the value of the X coordinates and the Y coordinates for all four of the graph’s lines, so that a value can be returned to displayed as an animation in the animation.FuncAnimation() function.
* **Inside main ():**
* Using a while loop with input() to prompt user for an action, and breaks the loop/ ends the program when “EXIT” is selected.
* If the user input chooses the add planet section, the user will be further prompted with the name and characteristics of the planet that is to be added in the list of planets.
* If the user input chooses the run simulation section, the program will request for a name of planet on which the simulation will occur. After a planet is chosen, the program will display the gravitational acceleration of that planet and plots a trajectory graph in respect to the planet’s gravitational acceleration.
* If the user input chooses the show planet details section, the user will be requested to enter a planet from the existing list of planets. After a planet name has been entered, a dictionary item which correspond to that planet will be displayed. This dictionary contains the characteristics of that specific planet.

**Planets Class: ( *planets.py* )**

* Contains a class named Planets that consist of:

1. Private member:

- planets\_list:planets\_list[]

**-** planets:planets{}

**-** name:String

**-** gravity:double

**-** distance\_from\_sun:int

**-** surface:String

**-** moons:String

**-** rings:String

**-** mass\_to\_earth:int

1. Public member:

* double getGravityt(), returns the value of gravity.
* planets getPlanet(), returns the planet dictionary.
* planets\_list getPlanetList(), returns the list of existing planets.
* addPlanet(name:String, gravity:double, distance\_from\_sun:int, surface:String, moons:Strings, rings:String, mass\_to\_earth:int)

**Functions: ( *functions.py* )**

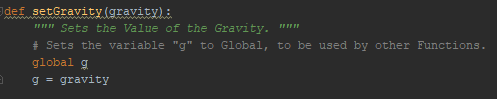
* **Inside setGravity (gravity):**
* This function sets the value of gravity and assigns it to variable “g”, which will then set the variable “g” to global to be able to be used by other functions with in this python file.
* **Inside Linear (x2, y2, N=100):**
* Returns Straight line from point (0, 0) to (x2, y2), with a gradient of -0.5.
* The value of N determines how smooth the curve will later be.
* **Inside Func (x, f, fp):**
* This function computes the minimized integrand of the time integral in respect to f(x) with the formula square root of ((1+fp(x)^2)/(2\*g\*f(x))), and returns that value.
* **Inside Circle (x2, y2, N=100):**
* Returns a circle curve from point (0, 0) to (x2, y2), as comparison to the Brachistochrone trajectory, and the time of travel as well.
* Using scipy, the time of travel of the circle curve is calculated by numerical integration.
* The value of N determines how smooth the curve will later be.
* **Inside Parabola (x2, y2, N=100):**
* Returns a parabolic curve from point (0, 0) to (x2, y2) as comparison to the Brachistochrone trajectory, and the time travel as well.
* It also calculates the time of travel of the object in the parabolic curve by the use of numerical integration.
* The value of N determines how smooth the curve will later be.
* **Inside Fp (x):**
* Returns the value from the formula (r-x)/F(x).
* **Inside F (x):**
* Returns the value from the formula sqrt (2\*r\*x – x^2).
* **Inside findTheta (theta):**
* Returns the value of y2/x2- (1-cos(theta)/theta – sin(theta)), to further be computed in the newton function provided by the scipy module.
* **Inside Brachistochrone (x2, y2, N=100):**
* Returns a Brachistochrone curve from point (0, 0) to (x2, y2).
* The value of N determines how smooth the curve will later be.
* It will first find the radius of the circle generating the Brachistochrone curve using the findTheta(theta) function. It will then find the time of travel for the Brachistochrone, and prints it in 4 significant figures. In addition to that it will return the value of x, y and T to be plotted in the main.py.

|  |
| --- |
| **Planets** |
| **-** planets\_list:planets\_list[]  **-** planets:planets{}  **-** name:String  **-** gravity:double  **-** distance\_from\_sun:int  **-** surface:String  **-** moons:String  **-** rings:String  **-** mass\_to\_earth:int |
| **+ Planets()**  + addPlanet(name:String, gravity:double, distance\_from\_sun:int, surface:String, moons:Strings, rings:String, mass\_to\_earth:int) |
| + getGravity(name:String):double |
| + getPlanet(name:String):planets  + getPlanetList():planets\_list |

**UML Class Diagram:**

**III.a. Lessons that Have Been Learned**

1. ***The use of “global”:***



Going into this project, I did some research on how I can set a variable from one function to be used and manipulated in another function, at that time I didn’t know an efficient way of doing it, but then I discovered the “global” keyword.

1. ***Changing the opacity of the line:***

A picture containing clock, outdoor

Description generated with high confidence

When plotting the graph for the first time, I thought that the graph looked boring and monochromatic. I thought that matplotlib was the library to blame for this mishab, but then later on I did a bit of research and found that I could change the opacity of the lines being plotted which in the end result gave a more minimalistic representation of the color I was looking for.

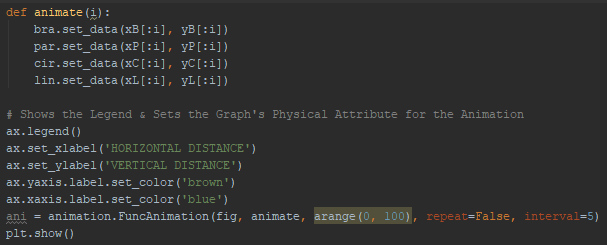
1. ***The existence of Scipy:***

A close up of a screen

Description generated with high confidence

Moving on with this project I took a considerable amount of time trying to find a way to compute for calculus problems like for example the “euler-lagrange differential equation” which was used to find the values which makes T stationary, without the use of math. I did a bit of research in forums and library documentation and found scipy, which already has calculus functions built-in, so that all I have to do is just set the arguments.

1. ***The use of animation in matplotlib:***



Into this project, I thought that matplotlib was a barebones library and could only plot basic graphs and shapes in the cartesian plane. I wanted my program to be able to not only visualize the trajectory, but also simulate it in real-scaled time. After asking a few friends of mine and did my research online, I came across an animation module built in the matplotlib library. This was surprising to me as I thought that matplotlib was not much but a basic plotter.

1. ***Sleep function in the time library:***

A close up of a logo

Description generated with high confidence

I wanted to find a way to make the text-based part of my program a little bit more interactive. I wandered online (mostly in stackoverflow), and found that there is a time library with a sleep function which creates a delay of my choosing, which I think gives a more user friendly experience while using the program.

1. ***Format in printing strings:***

A picture containing object

Description generated with very high confidence

I wanted to find a way to print the numerical values of my calculations in an efficient manner. That was when I came across the .format() function, which in my case lets me print my T values in the significant figures that I want, which was new to me.

**III.b. Problem that Have Been Overcome**

Making this program is far from easy, even though I thought it was straight forward. The first problem I encountered was understanding the formula of the Brachistochrone itself. I had to read an article from <http://webdev.physics.harvard.edu/academics/undergrad/probweek/sol83.pdf> which explains step by step the formula of the solution to the Brachistochrone problem. The next problem was to interpret the formula to python to begin with, and as I mentioned in part III.a., I found that scipy and numpy was the solution to that, since there were already prebuilt-in calculus functions in the library. The biggest problem for me however, is to animate the graph plot in relation to the respective time traveled in the trajectory. After researching and consulting with one of my friends, even though last minute I managed to make the plotting animation work. A lot of the other problems are small careless mistakes, which I expected since there is a bit of dictionary-work involved.

**V. Source Code**

1. ***main.py***

**import** functions  
**from** planets **import** Planets  
**import** matplotlib.pyplot **as** plt  
**import** matplotlib.animation **as** animation  
**from** numpy **import** arange  
**import** time  
  
*# Declares the Final Position of the Bead.  
# Declares All the Variables & the Dictionary of Places & their Respective Gravitational Acceleration*planets = Planets()  
x2, y2 = 1, 0.7  
welcome\_message = **"=============================\n"** \  
 **" BRACHISTOCHRONE SIMULATOR \n"** \  
 **"============================="**exit\_message = **"=============================\n"** \  
 **" GOODBYE \n"** \  
 **"============================="**planets\_list = planets.getPlanetsList()  
  
main\_menu = **"\n"** \  
 **"-----------------------------\n"** \  
 **" MAIN MENU \n"** \  
 **"-----------------------------"  
  
def** main():  
 *# Prints Welcoming Message  
 # Initiates Main Menu* print(welcome\_message)  
 **while True**:  
 **for** i **in** planets\_list:  
 print(i)  
 print(main\_menu)  
 action = input(**"[1] ADD PLANET\n"  
 "[2] RUN SIMULATION\n"  
 "[3] SHOW PLANET DETAILS\n"  
 "[4] EXIT\n"**)  
  
 **if** action == **"1"**:  
 **while True**:  
 *# Add a New Planet with User's Predetermined Attributes* name = input(**"PLANET NAME: "**).title()  
 gravity = float(input(**"GRAVITY (m/s2): "**))  
 distance\_from\_sun = int(input(**"DISTANCE FROM SUN: "**))  
 surface = input(**"SURFACE TYPE: "**)  
 moons = input(**"MOONS? Y/N: "**)  
 rings = input(**"RINGS? Y/N: "**)  
 mass\_to\_earth = float(input(**"HOW MANY TIMES EARTH MASS? "**))  
 *# Calls a Function from Planets Module* planets.addPlanet(name, gravity, distance\_from\_sun, surface, moons, rings, mass\_to\_earth)  
 *# Prompts for Next Action* next\_action = input(**"ADD ANOTHER PLANET? Y/N"**).upper()  
 **if** next\_action == **"N"**:  
 **break  
  
 if** action == **"2"**:  
 **while True**:  
 *# Displays the Existing Lists of Planets* print(planets\_list)  
 user\_input = input(**"\nPICK A PLANET: "**).title()  
 *# Declares "g" by Calling a Function from the Planets Module* g = planets.getGravity(user\_input)  
 print(user\_input, **":"**, g, **"m/s2\n"**)  
 *# Sets Gravity by Calling on the Functions Module for Computation* functions.setGravity(g)  
  
 *# Plot a Figure Comparing the 4 Trajectory  
 # Displays the Time in 4 Significant Figures* fig = plt.figure()  
 ax = fig.add\_subplot(111, autoscale\_on=**False**, xlim=(0, 1), ylim=(0.8, -0.1))  
  
 xB, yB, TB = functions.Brachistochrone(x2, y2)  
 xP, yP, TP = functions.Parabola(x2, y2)  
 xC, yC, TC = functions.Circle(x2, y2)  
 xL, yL, TL = functions.Linear(x2, y2)  
  
 bra, = ax.plot([], [], lw=3, alpha=0.5, label=**'{}: {:.4f} s'**.format(**"Brachistochrone"**, TB))  
 par, = ax.plot([], [], lw=3, alpha=0.5, label=**'{}: {:.4f} s'**.format(**"Parabola"**, TP))  
 cir, = ax.plot([], [], lw=3, alpha=0.5, label=**'{}: {:.4f} s'**.format(**"Circle"**, TC))  
 lin, = ax.plot([], [], lw=3, alpha=0.5, label=**'{}: {:.4f} s'**.format(**"Linear"**, TL))  
  
 **def** animate(i):  
 bra.set\_data(xB[:i], yB[:i])  
 par.set\_data(xP[:i], yP[:i])  
 cir.set\_data(xC[:i], yC[:i])  
 lin.set\_data(xL[:i], yL[:i])  
  
 *# Shows the Legend & Sets the Graph's Physical Attribute for the Animation* ax.legend()  
 ax.set\_xlabel(**'HORIZONTAL DISTANCE'**)  
 ax.set\_ylabel(**'VERTICAL DISTANCE'**)  
 ax.yaxis.label.set\_color(**'brown'**)  
 ax.xaxis.label.set\_color(**'blue'**)  
 ani = animation.FuncAnimation(fig, animate, arange(0, 100), repeat=**False**, interval=5)  
 plt.show()  
 end\_action = input(**"EXIT TO MAIN MENU? Y/N "**).upper()  
 **if** end\_action == **"Y"**:  
 **break  
  
 if** action == **"3"**:  
 **while True**:  
 planet\_name = input(**"PLANET NAME: "**).title()  
 *# Calls a Function from the Planets Module with the Argument "planet\_name"* planet\_details = planets.getPlanet(planet\_name)  
 *# Displays the Attribute of the Planet* print(planet\_details)  
 mid\_action = input(**"EXIT TO MAIN MENU? Y/N "**).upper()  
 **if** mid\_action == **"Y"**:  
 **break  
  
 if** action == **"4"**:  
 print(**"EXITING..."**)  
 *# Set a 2 Seconds Delay Before Exiting or Breaking the While Loop* time.sleep(2)  
 print(exit\_message)  
 **break**main()

1. ***planets.py***

**class** Planets(object):  
 *# Initialize the Lists & Dictionaries for the Planets* **def** \_\_init\_\_(self):  
 self.planets\_list = [**'Mercury'**, **'Venus'**, **'Earth'**, **'Mars'**, **'Jupiter'**, **'Saturn'**, **'Uranus'**, **'Neptune'**]  
 self.planets = {**'Mercury'**: {**'name'**: **'Mercury'**, **'gravity'**: 3.59, **'distance\_from\_sun'**: 58000000, **'surface'**: **'Rocky'**, **'moons'**: **'No'**, **'rings'**: **'No'**, **'mass\_to\_earth'**: 0.055},  
 **'Venus'**: {**'name'**: **'Venus'**, **'gravity'**: 8.87, **'distance\_from\_sun'**: 108000000, **'surface'**: **'Rocky'**, **'moons'**: **'No'**, **'rings'**: **'No'**, **'mass\_to\_earth'**: 0.815},  
 **'Earth'**: {**'name'**: **'Earth'**, **'gravity'**: 9.81, **'distance\_from\_sun'**: 150000000, **'surface'**: **'Rocky'**, **'moons'**: **'Yes'**, **'rings'**: **'No'**, **'mass\_to\_earth'**: 1},  
 **'Mars'**: {**'name'**: **'Mars'**, **'gravity'**: 3.77, **'distance\_from\_sun'**: 228000000, **'surface'**: **'Rocky'**, **'moons'**: **'Yes'**, **'rings'**: **'No'**, **'mass\_to\_earth'**: 0.10744},  
 **'Jupiter'**: {**'name'**: **'Jupiter'**, **'gravity'**: 25.95, **'distance\_from\_sun'**: 778000000, **'surface'**: **'Gaseous'**, **'moons'**: **'Yes'**, **'rings'**: **'Yes'**, **'mass\_to\_earth'**: 317.82},  
 **'Saturn'**: {**'name'**: **'Saturn'**, **'gravity'**: 11.08, **'distance\_from\_sun'**: 1427000000, **'surface'**: **'Gaseous'**, **'moons'**: **'Yes'**, **'rings'**: **'Yes'**, **'mass\_to\_earth'**: 95.16},  
 **'Uranus'**: {**'name'**: **'Uranus'**, **'gravity'**: 10.67, **'distance\_from\_sun'**: 2871000000, **'surface'**: **'Gaseous'**, **'moons'**: **'Yes'**, **'rings'**: **'Yes'**, **'mass\_to\_earth'**: 14.371},  
 **'Neptune'**: {**'name'**: **'Neptune'**, **'gravity'**: 14.07, **'distance\_from\_sun'**: 4498000000, **'surface'**: **'Gaseous'**, **'moons'**: **'Yes'**, **'rings'**: **'Yes'**, **'mass\_to\_earth'**: 17.147}}  
  
 **def** getGravity(self, name):  
 *""" Returns the Gravitational Acceleration of that Specific Planet. """* **return** self.planets[name][**'gravity'**]  
  
 **def** addPlanet(self, name, gravity, distance\_from\_sun, surface, moons, rings, mass\_to\_earth):  
 *""" Adds a Planet to the List and to the Planets Dictionary. """* self.planets\_list.append(name)  
 self.planets[name] = {**'name'**: name, **'gravity'**: gravity, **'distance\_from\_sun'**: distance\_from\_sun, **'surface'**: surface,  
 **'moons'**: moons, **'rings'**: rings, **'mass\_to\_earth'**: mass\_to\_earth}  
  
 **def** getPlanet(self, name):  
 *""" Returns the Attributes of a Specific Planet. """* **return** self.planets[name].items()  
  
 **def** getPlanetsList(self):  
 *""" Return a List of the Existing Planets. """* **return** self.planets\_list

1. ***functions.py***

**import** numpy **as** np  
**from** scipy.optimize **import** newton  
**from** scipy.integrate **import** quad  
*# Acceleration Due to Gravity.  
# Final Position of the Bead.*x2, y2 = 1, 0.7  
**def** setGravity(gravity):  
 *""" Sets the Value of the Gravity. """  
 # Sets the variable "g" to Global, to be used by other Functions.* **global** g  
 g = gravity  
  
**def** Brachistochrone(x2, y2, N=100):  
 *""" Returns a Brachistochrone Curve from point (0,0) to (x2, y2). """  
 # Find the value of Theta.* **def** findTheta(theta):  
 **return** y2/x2 - (1-np.cos(theta))/(theta-np.sin(theta))  
 theta2 = newton(findTheta, np.pi/2)  
  
 *# Find the Radius of Circle Generating the Brachistochrone.* R = y2 / (1 - np.cos(theta2))  
  
 theta = np.linspace(0, theta2, N)  
 x = R \* (theta - np.sin(theta))  
 y = R \* (1 - np.cos(theta))  
  
 *# Find the Time of Travel of Brachistochrone & Prints It.* T = theta2 \* np.sqrt(R / g)  
 print(**'T(Brachistochrone) = {:.4f}'**.format(T))  
 **return** x, y, T  
  
**def** Linear(x2, y2, N=100):  
 *""" Return a Straight Line from (0,0) to (x2, y2). """  
  
 # Gradient of Equation* m = y2 / x2  
 x = np.linspace(0, x2, N)  
 *# Linear Equation* y = m\*x  
  
 *# Find the Time of Travel of a Linear Equation.* T = np.sqrt(2\*(1+m\*\*2)/g/m \* x2)  
 print(**'T(Linear) = {:.4f}'**.format(T))  
 **return** x, y, T  
  
**def** Func(x, f, fp):  
 *""" The Integrand of the Time Integral that Minimizes in Respect to f(x). """* **return** np.sqrt((1+fp(x)\*\*2) / (2 \* g \* f(x)))  
  
**def** Circle(x2, y2, N=100):  
 *""" Returns a Circle Curve from point (0,0) to (x2, y2). """  
  
 # Find the Circle's Radius* r = (x2\*\*2 + y2\*\*2)/2/x2  
  
 **def** F(x):  
 **return** np.sqrt(2\*r\*x - x\*\*2)  
 **def** Fp(x):  
 **return** (r-x)/F(x)  
  
 x = np.linspace(0, x2, N)  
 y = F(x)  
  
 *# Calcualte the Time of Travel by Numerical Inetegration.* T = quad(Func, 0, x2, args=(F, Fp))[0]  
 print(**'T(Circle) = {:.4f}'**.format(T))  
 **return** x, y, T  
  
**def** Parabola(x2, y2, N=100):  
 *""" Returns a Parabolic Curve from point (0,0) to (x2, y2). """  
  
 # Set the Constant for the Curve* c = (y2/x2)\*\*2  
  
 **def** F(x):  
 **return** np.sqrt(c\*x)  
 **def** Fp(x):  
 **return** c/2/F(x)  
  
 x = np.linspace(0, x2, N)  
 y = F(x)  
  
 *# Calculate the time of travel by numerical integration.  
 # Print T(parabola) in 3 significant figures points.* T = quad(Func, 0, x2, args=(F, Fp))[0]  
 print(**'T(Parabola) = {:.4f}'**.format(T))  
 **return** x, y, T