AMATH 271: Final Project

Omer Sipra, Kamalesh Reddy

December 10, 2020

Abstract

The following project falls in the mathematical domain of Calculus of Variation. It exhibits a computer program that is designed to provide a path of least "insert independent variable here" for any problem in Theoretical Mechanics. The computer program uses principles from Lagrangian Mechanics to achieve this. The problem's lagrangian (specifically, it's arguments) is the input to the program and, using the Euler-Lagrange equation, the programs provides the user with a plot of the minimized quantity vs the independent variable that they defined.

Introduction

The problem that inspired this Project is one of the oldest problems in Calculus of Variations. The problem was a challenge by the mathematician Johann Bernoulli. He described the problem as: "Given two points A and B in a vertical plane, what is the curve traced out by a particle under the influence of gravity (constant), which starts at A and reaches B in the shortest time."

In other words: 'A' is any arbitrary point (x_0, y_0) in 2 dimensional space and B is defined as (x_1, y_1) where $y_1 \leq y_0$ and $x_0 \neq x_1$. This results in B being a translation of A which is under (but not directly under A). The system includes a particle which is supposed to "fall" from A to B because of a constant gravitational force. The objective of the problem is to find the path that would require the least time for the particle to get to B.

The problem was attempted and successfully solved by several mathematicians. Isaac Newton famously solved the problem overnight. The solution of this problem was named the "Brachistochrone". Johann Bernoulli's solution being the first, identified this brachistochrone to be the common **cycloid**.

A cycloid is the path traced out by a point on the circumference of a circle as it rolls on a horizontal surface without slipping.

Different solutions to this problem were developed until as early as the 2010's. One of these solutions is what inspired this project. It was developed by Mark Levi [2] in 2014. His solution shows this symmetry in the path of least time called the “tautochrone” property of a brachistochrone, which means that the path a particle would take to fall from any point on the cycloid to B would all be the same! This made us realize that for any mechanical problem there exists 1 solution that describes the path of a least quantity which can be scaled to any magnitude of the problemâĂŹs dimensions.

Another solution to the brachistochrone problem is provided in the textbook [1]. It involves finding the Lagrangian of the problem and using the Euler-Lagrange equation to find a DE that can be plotted. The resulting plot, as expected, reveals a brachistochrone.

If this method works for the brachistochrone problem, then it should work for any arbitrary problem in mechanics where a path can be traced. We hence decided to create a program to find the path of least time/distance of any problem that involves a Lagrangian. To achieve this we used the solutions mentioned above to design a program written in Python that can take a Lagrangian input (which corresponds to a mechanics problem of course) and then said code will show you the path of least distance/time. To demonstrate this project we tested the program for a couple of problems including problems from Mini-Project 2, AMATH 271 and problems from chapter 6 of Taylor. The first two problems of the project are borrowed from Taylor. The first problem is the simplest on the project and helped us understand how to setup the mechanics of the code. It requires us to find the shortest possible path between two points using the Euler-Lagrange method. We know that the answer to this is a straaight line. The problem verifies this by using the arc length measurement and minimizing the distance through that, On the other hand, the second problem is the brachistochrone problem itself, but following the same method as problem 1. We are supposed to find the path of least time to get from A to B (A and B being the points defined above). This problem also involves inverting out independent and dependent variable which is something the code would have to account for to work in all problems.

The last 2 problems of the project were borrowed from Mini-Project 2. These problems are a lot more complex than the first two and use conventional method of finding a Lagrangian to then minimize a quantity. Since we already knew the solutions to these, they served as a good way to find potential shortcomings of our method and also verified anything we did right.

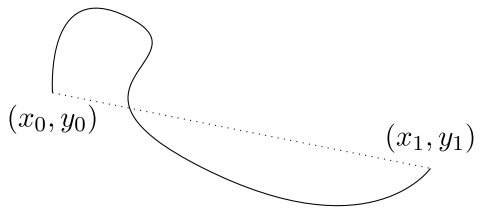
We could actually improve on this, granted it could take quite a long time, and publish a full-fledged Python libraries. But, most ‘generealized’ Python libraries have around 2000+ lines of code and multiple files; this would also require contribution of sevral people; but this is a start, since, as we will see, it does actually: solve a Lagrangian by substituting into the Euler-Lagrange equation $\longrightarrow$ simplifies the equation to give us a DE of the form $f(\cdots) = 0 \longrightarrow$ use this DE to plot required quantities (which was done in Maple for Mini-Project 1). The only part which can be improved upon is having the program manipulate the DE into the system, this way the user will not even have to manually enter a system of DE’s.

Following are our findings and the code:

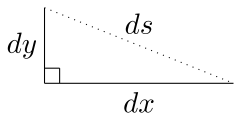
1

Taylor Example 6.1: Shortest Path Between 2 Points (warm-up)

1.1 Equations of problems



Now, consider a small section of the curve, ds (an arbitrarily small ds is a line):



We can hence write ds in terms of small steps in the x and y direction using the Pythagorean theorem:

$$ds^2 = dx^2 + dy^2 \implies ds = \sqrt{dx^2 + dy^2} \implies ds = \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx \implies ds = \sqrt{1 + y'(x)^2}\, dx$$

Integrating on both sides from (x_0, y_0) to (x_1, y_1):

$$\int_{(x_0,y_0)}^{(x_1,y_1)} ds = \int_{x_0}^{x_1} \sqrt{1 + \dot{y}^2}\, dx.$$

1.2 Analysis

Our goal is to minimize the integral, $\int_{(x_0,y_0)}^{(x_1,y_1)} ds$. To minimize this, observe that, equivalently, we can take the integrand, $\sqrt{1 + \dot{y}^2}$, to be our Lagrangian (the proof of which is omitted)[1]. This is the essence of Lagrangian mechanics, it says nature is an optimization problem and depicts “nature is lazy" in equations. We have,

$$\mathcal{L}(\dot{y}, y, x) = \sqrt{1 + \dot{y}^2}$$

[1]Look into "Calculus of variations", Euler-Lagrange equation", and "Principle of stationary action".

1.3 Solutions - *SPBTP.py*

Now, on to solving this problem! Now that we have the Lagrangian, we can feed it to our Python program while making necessary changes (passing parameters (here, constants), variables 'names', and the Lagrangian itself as parameters).
The mechanism of this program will be explained in detail for this warm-up problem:

1. We import the required libraries; in this program, we will only need `sympy` to output the derivative. However, we will not use it to solve the derivative, since most derivatives cannot be solved numerically, which is how sympy solves them. We will use a second program to solve the derivative numerically.
2. Declare the function, `givemeDE`. The function does not explicitly take the lagrangian as an argument (for reasons that I will state later); BUT, the lagrangian is defined locally in the very next step, this makes it easier for the user to define the lagrangian, which will be required either way.
3. This problem's lagrangian does not require us to define any parameter (clearly).
4. The user must define the lagrangian, `L`, manually.
   Note: Make sure the Lagrangian is a function of ***three*** *variables.*
5. We then assign each variable; the dependent variable's derivative, the dependent variable, and the independent variable; with strings. This displays the differential equation with appropriate strings in place of `dvarp`, `dvar`, and `ivar` (the general variables our program uses).
6. The next step is where Python solves for each term of the lagrangian and `prints` it to the console.
7. Then, it plugs the two terms into the Euler-Lagrange equation (defined implicitly) and isolates all terms.
8. Finally, it prints the `DE`. The final differential equation will be of the form $f(\dot{q}, q, t) = 0$.
9. Keep the format of the function `givemeDE` in mind.

Crux: the program take the lagrangian as "input" and gives us the DE: $f(\ddot{q}, \dot{q}, q, t) = 0$. This DE will now be solved, again, using Python. Running the code below (`derog.py`) as is, we get:

```
∂ℒ/∂y       = 0


∂ℒ/∂y'      = Derivative(y(x), x)/sqrt(Derivative(y(x), x)**2 + 1)


d/dx(∂ℒ/∂y') = Derivative(y(x), (x, 2))/sqrt(Derivative(y(x), x)**2 + 1) -
    Derivative(y(x), x)**2*Derivative(y(x), (x, 2))/(Derivative(y(x), x)**2 + 1)**(3/2)
Required DE: y''/(y'**2 + 1)**(3/2) = 0
```

`y''/(y'**2 + 1)**(3/2) = 0` $\implies$ $\ddot{y} = 0$: (This is, as expected, a straight line!)

$$y(x) = C_1 + C_2 x$$

Python Code:

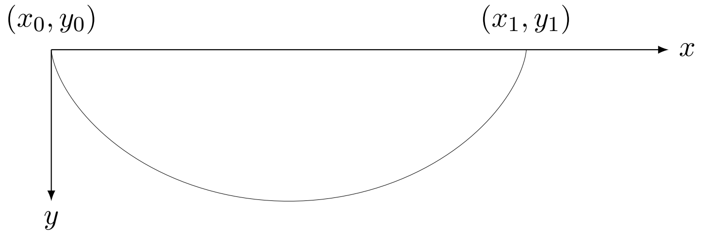
```
# LIBRARIES
import sympy as sp

# FUNCTION
def givemeDE(*args):

  #PARAMETERS

  #LAGRANGIAN
  def L(dvar, dvarp, ivar):
      return sp.sqrt(1 + dvarp**2)

  if len(args) == 3:
      # AESTHETICS
      def convertEq(x):
          x = x.replace("Eq(", "")
          x = x.replace(", 0)", "")
          x = x + " = 0"
          return x
      # AESTHETICS
      def convertDE(de):
          de_str = str(de)
          de_str = de_str.replace("Derivative(" + args[-2] + "(" + args[-1] + "), (" +
              args[-1] + ", 2))", args[-3] + "'" + "'")
          de_str = de_str.replace("Derivative(" + args[-2] + "(" + args[-1] + "), " +
              args[-1] + ")", args[-3] + "'")
          de_str = de_str.replace(args[-2] + "(" + args[-1] + ")", args[-3])
          return de_str

      # DEFINES ASSIGNED SYMBOLS
      ivar = sp.Symbol(args[-1])
      dvar = sp.Function(args[-2])(ivar)
      dvarp = sp.Derivative(dvar, ivar)

      # PRINTS EACH TERM OF THE LAGRANGIAN
      print("∂ℒ/∂" + args[-3] + "    =", sp.diff(L(dvar, dvarp, ivar), dvar))
      print('\n')
      print("∂ℒ/∂" + args[-3] + "'" + "   =", sp.diff(L(dvar, dvarp, ivar), dvarp))
      print('\n')
      print("d/d" + args[-1] + "(∂ℒ/∂" + args[-3] + "')" + " =",
          sp.diff(sp.diff(L(dvar, dvarp, ivar), dvarp), ivar))

      # FINDS DE USING TERMS ABOVE
      de = sp.simplify(sp.Eq(sp.diff(L(dvar, dvarp, ivar), dvar) -
          sp.diff(sp.diff(L(dvar, dvarp, ivar), dvarp), ivar), 0))

      # PRINTS OUT DE
      print("Required DE:", convertDE(convertEq(str(de))))

# EXECUTE PROGRAM!
givemeDE("y", "y", "x")
```

2

Taylor Example 6.2: Brachistochrone

"Given two points A and B in a vertical plane, what is the curve traced out by a point acted on only by gravity, which starts at A and reaches B in the shortest time." - *Johann Bernoulli*

2.1 Equations of problems



Now that we know how to minimize a certain quantity numerically using the Euler-Lagrange equations, we will follow the same method in solving this problem. We have:

$$v = \sqrt{2gy} \implies \frac{ds}{dt} = \sqrt{2gy}$$

Since we want to minimize time, we isolate 'dt'':

$$dt = \frac{ds}{\sqrt{2gy}}$$

From Section 1.1, we know that $ds = \sqrt{1 + x'(y)^2}$. On substitution:

$$dt = \frac{\sqrt{1 + x'(y)^2}\,dx}{\sqrt{2gy}}$$

Integrating on both sides from initial point to final point:

$$\int_{t_0}^{t_1} dt = \int_{x_0}^{x_1} \frac{\sqrt{1 + x'(y)^2}}{\sqrt{2gy}} dx \implies \int_{t_0}^{t_1} dt = \frac{1}{\sqrt{2g}} \int_{x_0}^{x_1} \sqrt{\frac{1 + x'(y)^2}{y}} dx.$$

2.2 Analysis

According to our problem statement, we want to minimize the time, $\int_{t_0}^{t_1} dt$. We procced in the same way (Section 1.2), by choosing $\sqrt{\frac{1+x'(y)^2}{y}}$ to be our lagrangian:

$$\mathcal{L}(\dot{x}, x, y) = \sqrt{\frac{1+x'(y)^2}{y}}$$

2.3 Solutions - *Brach.py*

The entire program will not be explained since most of it remains the same; the only parts that change are: the definition of the lagrangian and the final DE that we will plot this time.
The Lagrangian is defined as follows:

```
  def L(dvar, dvarp, ivar):
      return sp.sqrt((dvarp**2 + 1)/ivar)
```

Also note the change in variables, the independent variable here is y and the dependent variable is x, so we change the arguments of the function as follows:

```
givemeDE("x", "x", "y")
```

We must also comment out the line that prints the DE's solution. This is because sympy solves DE's analytically to output a cartesian equation, however, in the case of a brachistochrone there is no such equation (or it is rather complicated); we instead verify by solving it numerically and plotting the solution using the following code from `DEnumerical.py`:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def brach(a):
    def diffeqs(w, y, p):
        [x] = w # d matrix
        [] = p # parameters
        der = [np.sqrt(y/(2*a - y))] #Spoiler Alert!
        return der

    # y values
    y = np.linspace(0.001, 2.1, 100000)

    # containing parameters and initial conditions
    p = []
    w0 = [0]

    wsol = odeint(diffeqs, w0, y, args = (p, ))
    plt.plot(wsol[:, 0], -1*y)
brach(1)
plt.show()
```

Brief explanation of the code above:

1. Again, we import the required libraries; in this program, we will need `numpy` and `scipy` to solve the DE numerically, and then we plot it using `matplotlib.pyplot`.
2. `brach(a)` is a function that takes 'a', the parameter, as an argument.
3. `diffeqs(w, y, p)` then solves the system of differential equations as per it's definition inside `diffeqs` and returns the derivative of the matrix.
4. `odeint` then uses `diffeqs`; the initial conditions, `w0`; the y (independent variable) values `y`; and the parameters, here, '`a`' to solve the system and store them in `wsol`
5. Then, we simply plot $y vs. x$ or, here, `(wsol[:, 0], -1*y)` after adjusting 'direction' of y.

Now that we have the code out of the way; let's find the DE numerically, we run the program to get:

```
∂ℒ/∂x        = 0


∂ℒ/∂x'       = sqrt((x'**2 + 1)/y)*x'/(x'**2 + 1)


d/dy(∂ℒ/∂x') = y*sqrt((x'**2 + 1)/y)*(x'*x''/y
               - (x'**2 + 1)/(2*y**2))*x'/(x'**2 + 1)**2
               + sqrt((x'**2 + 1)/y)*x''/(x'**2 + 1)
               - 2*sqrt((x'**2 + 1)/y)*x'**2*x''/(x'**2 + 1)**2


Required DE    : (-2*y*x'' + x'**3 + x')/(2*y**2*sqrt((x'**2 + 1)/y)*(x'**2 + 1)) = 0
```

Now, observe that `d/dy`$(\partial\mathcal{L}/\partial x')$ looks rather complicated. Instead, as suggested in Taylor we can say that since $\partial\mathcal{L}/\partial x = 0$, `d/dy`$(\partial\mathcal{L}/\partial x') = 0 \implies \partial\mathcal{L}/\partial x' = C$, where C is the constant of integration;

$$\implies \texttt{sqrt((x'**2 + 1)/y)*x'/(x'**2 + 1)} = \sqrt{\frac{x'^2+1}{y}} \cdot \frac{x'}{x'^2+1} = \frac{x'}{\sqrt{y(x'^2+1)}} = C$$

Setting $C = \sqrt{\dfrac{1}{2r}}$ (as we will see later, r is the radius of the cycloid's circle):

$$\frac{x'^2}{y(x'^2+1)} = C^2 \implies \frac{x'^2}{y(x'^2+1)} = \frac{1}{2r} \implies 2rx'^2 = y(x'^2+1) \implies x'^2(2r-y) = y$$

$$\boxed{x' = \sqrt{\frac{y}{2r-y}}}$$

We solve the following “system” using `DEnumerical.py`:

$$[x] = \left[\sqrt{\frac{y}{2r-y}}\right]$$

That is:

```
[x] = w
[a] = p
der = [np.sqrt(y/((2*a) - y))]
```

For $a = 1$:

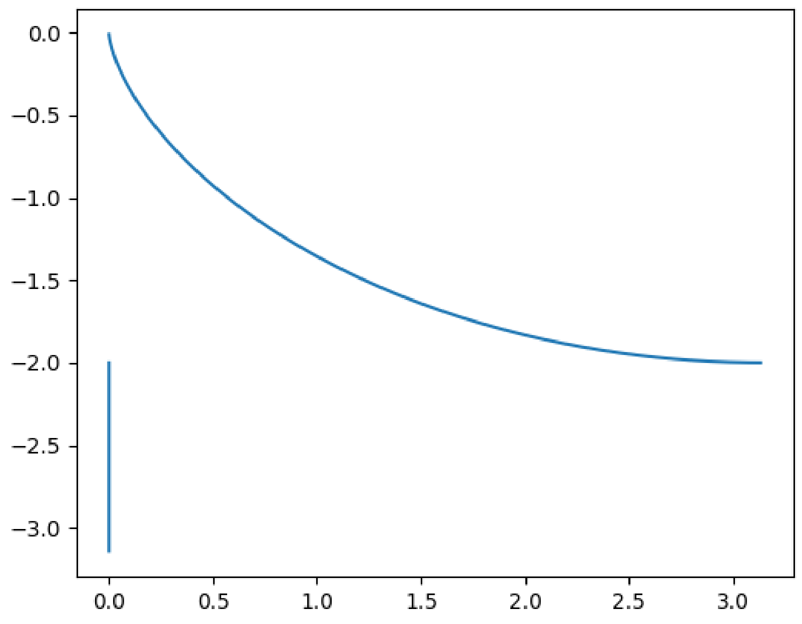
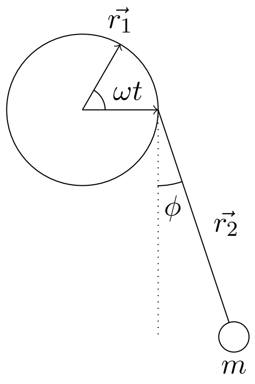


Figure 2.1: Cycloid with a circle of radius 1.

3

Mini-Project 2: Question 1

3.1 Equations of problems



3.2 Analysis

From Mini-project 1, question 1; we have the following lagrangian for the above system:

$$\mathcal{L} = \mathcal{L}(\phi', \phi, t) = \frac{m}{2}(R^2\omega^2 + L^2\dot{\phi}^2 + 2R\omega L\dot{\phi}\sin(\phi - \omega t)) - mg(R\sin(\omega t) - L\cos(\phi))$$

3.3 Solutions - *.py*

Plugging this Lagrangian into `MP2q1.py`, the program retruns the DE, we get (for $\omega = 0$ and 0.2):

```
∂ℒ/∂ϕ       = -1.0*sin(ϕ) # w = 0


∂ℒ/∂ϕ'      = 1.0*ϕ'


d/dt(∂ℒ/∂ϕ') = 1.0*ϕ''

Required DE    : 1.0*sin(ϕ) + 1.0*ϕ'' = 0
```

```
∂ℒ/∂ϕ       = -1.0*sin(ϕ) + 0.04*cos(0.2*t - ϕ)*ϕ' # w = 0.2


∂ℒ/∂ϕ'      = -0.04*sin(0.2*t - ϕ) + 1.0*ϕ'


d/dt(∂ℒ/∂ϕ') = -0.04*(0.2 - ϕ')*cos(0.2*t - ϕ) + 1.0*ϕ''

Required DE    : 1.0*sin(ϕ) - 0.008*cos(0.2*t - ϕ) + 1.0*ϕ'' = 0
```

$\omega = 0$:

```
1.0*sin(phi) + 1.0*phi'' = 0
```

$\implies \phi'' = -\sin\phi$

$\omega = 0.2$:

```
1.0*sin(phi) - 0.008*cos(0.2*t - phi) + 1.0*phi'' = 0
```

$\implies \phi'' = -\sin\phi + 0.008\cos(0.2t - \phi)$

We hence get the following plots: (just as in Question 1!)

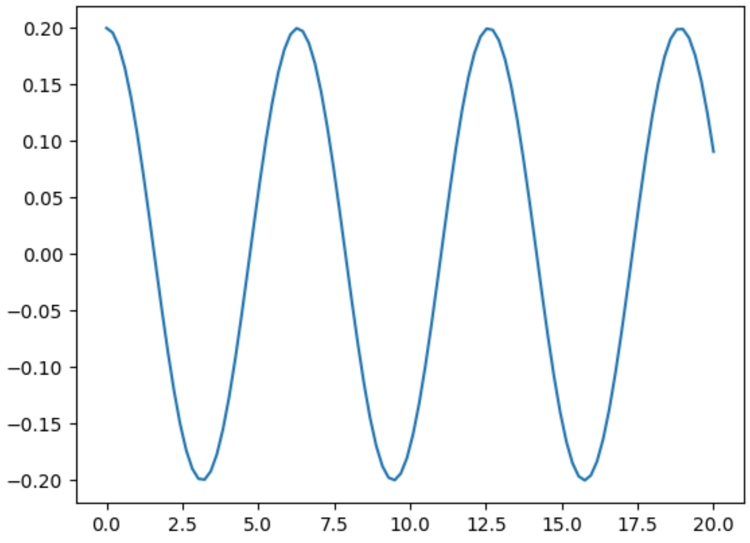


Figure 2.1: $\implies \phi'' = -\sin\phi$.

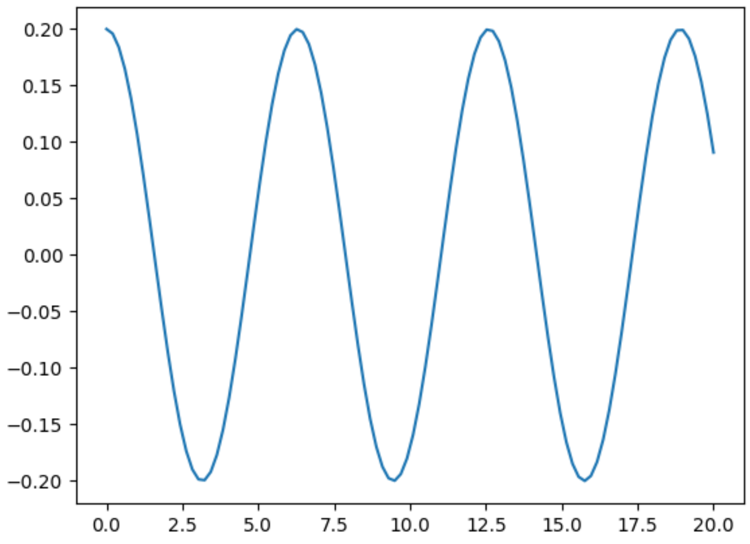
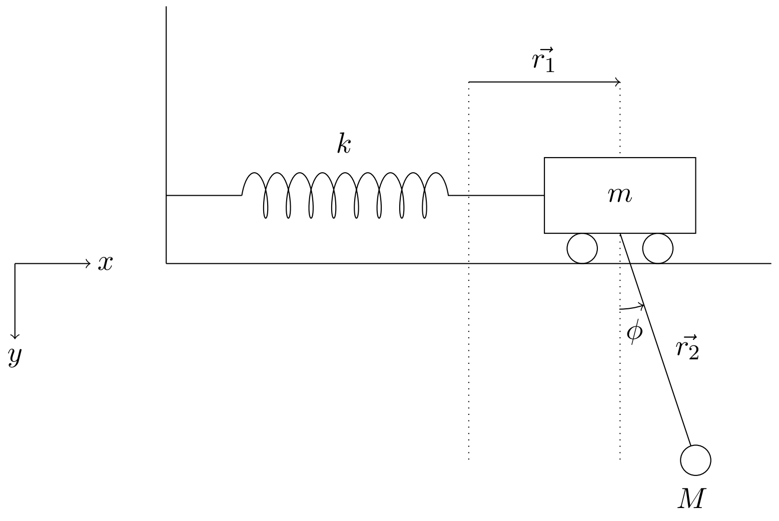


Figure 2.1: $\implies \phi'' = -\sin\phi + 0.008\cos(0.2t - \phi)$.

4

Mini-Project 2: Question 2

4.1 Equations of problems



4.2 Analysis

For the second question, we had the following Lagrangian:
$\mathcal{L} = \mathcal{L}(\dot{x}, x, \dot{\phi}, \phi, t) = \frac{1}{2}m\dot{x}^2 + \frac{1}{2}M\dot{x}^2 + \frac{1}{2}ML^2\dot{\phi}^2 + M\dot{x}L\dot{\phi}\cos(\phi)) - \frac{1}{2}kx^2 + MgL\cos(\phi)$
However, observe that this has five arguments, but our program works for three arguments. We could in theory generalize our program, but we will slightly modify it to work for five argument lagrangians (this limitation will be explained in the conclusion). The only limitation is not being able to convert the equation's "raw" form to a nicer looking equation; but, we do get an equation that we can plot.

4.3 Solutions - *.py*

Plugging this Lagrangian into `MP2q2.py`, the program retruns the DE, we get

```
∂ℒ/∂x         = 0


∂ℒ/∂x'        = 1.0*cos(phi(t))*Derivative(phi(t), t) + 1.0*x'


d/dt(∂ℒ/∂x') = -1.0*sin(phi(t))*Derivative(phi(t), t)**2 +
    1.0*cos(phi(t))*Derivative(phi(t), (t, 2)) + 1.0*x''
```

```
  Required DE     : -1.0*sin(phi(t))*Derivative(phi(t), t)**2 +
      1.0*cos(phi(t))*Derivative(phi(t), (t, 2)) + 1.0*x'' = 0
```

`-1.0*sin(phi(t))*Derivative(phi(t), t)**2 + 1.0*cos(phi(t))*Derivative(phi(t), (t, 2)) + 1.0*x'' = 0`

$\implies -\phi'^2 \sin\phi + \phi'' \cos\phi + x'' = 0$

We hence get the following plots: (just as in Question 2!)

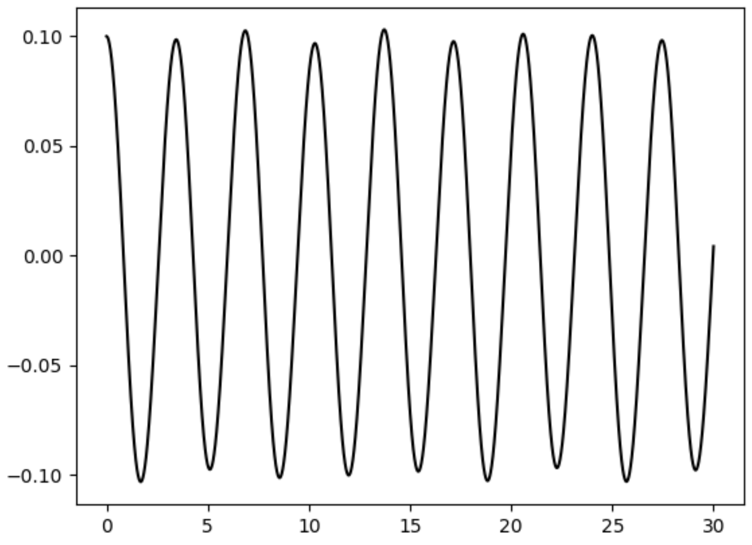


Figure 2.1: $-\phi'^2 \sin\phi + \phi'' \cos\phi + x'' = 0$.

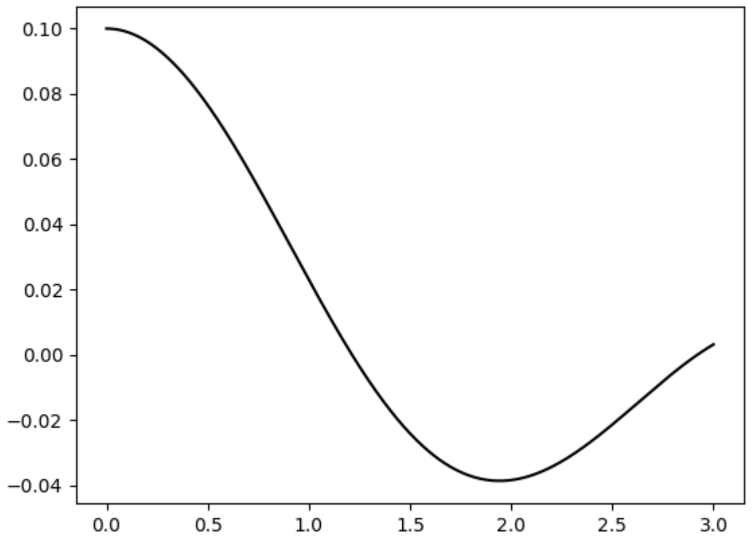


Figure 2.2: $-\phi'^2 \sin\phi + \phi'' \cos\phi + x'' = 0$, with initial conditions as in the Mini-Project; we see that this is the exact same graph from the second question.

1 Conclusions

The report shows, the program involves a 3 step process that can find how the minimization of one quantity affects a system. The system starts with the Lagrangian of the system defined with its independent variable, dependent variable and the derivative of the dependent variable. The code will then utilize the Euler-Lagrange equations to minimize the quantity that the user defines. The relationship is then used to obtain a differential equation. For simpler problems these DE's can be solved analytically. For Problem 1 this DE states that the double derivative of the indepent variable is 0. This is correct since the solution to that DE is of the same form as the equation of a line. Which is the correct solution to the problem (the shortest line between two points is indeed a line).

Problem 2 does the same thing as above but instead of minimzing distance it minimizes time, and as the plot shows: the solution is the original brachistochrone that inspired this project. This verified out method and results.

Problems 3 and 4 were added to expand the complexity of the problems that the program can solve. Solutions and plots produced for those problems also match the correct solutions. For further development of this project, a user can potentially expand this code to include problems that involve several dependent variables parametrized by a single quantity or potentially be expanded to any arbitraty n bodies or n dimensional problems; this was what we were referring to by this project being the start of a potential **Python Library**! Although this can, depending on comlexity, take months, if not, years to develop.

References

[1] Taylor, J. R. *Classical Mechanics.* University Science Books, 2005.

[2] Mark Levi *Quick! Find a Solution to the Brachistochrone Problem.* Mathematical Curiosities, 2005.

[3] Rodrigo Matos Carnier *Energy-efficient optimal control of robotic gait by indirect methods.* Research Gate , 2017