MA202 Project Report
Automatic Car Braking System Using Numerical Methods

Anavart Pandya
20110016

Ruchit Chudasama
20110172

Varun Chandwani
20110221

Vikash Vishnoi
20110226

Yash Adhiya
20110235

24 APRIL, 2022

Contents

Abstract

Being able to predict the future conditions and state of something using the given data is the need of today' world. Prediction helps making process more structured and strategic. Although the predictions are not always 100 percent accurate, but the results with least possible errors are also of great use to many industries. All these predictions can be done using some numerical methods like Lagrange's Interpolation, Newton's Method etc.

For the past several years, research and development in the field of self-driving automobiles has progressed dramatically, resulting in various intuitive and creative developments. However, in self-driving automobiles, the issue of safety has always been a major problem. Various driver-less car trials on the road have resulted in serious accidents. Working around the safety characteristics of self-driving automobiles is thus a critical requirement.

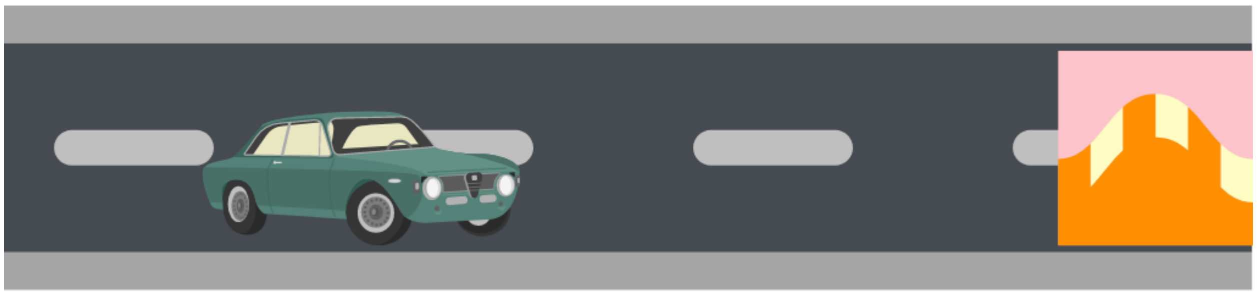


Figure 1.1: Car approaching a speed-breaker

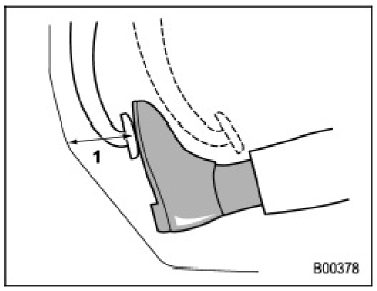


Figure 1.2: Braking Angle θ

Introduction

Our project is based on the working of the cars with automatic braking systems. Automatic braking systems consists of a sub system which helps in calculating the distance of the vehicle from the incoming obstacles like speed breakers. Basically this sub system is just a group of sensors which are fixed in front of the car. These sensors helps in calculating the distance of the car from the speed breaker after every fixed distance travelled.

The data from these sensors and the current speed of the car is used to command the braking system so that the vehicle stops before the obstacle or attains a required speed before reaching the speed breaker.

For this, we have defined a function G(θ) which displays the acceleration at any angle θ. Here, θ is the angle by which the car's brake pad of the car has been pressed.

Here, we have made sure that the value of the function G(θ) does stay negative and is practical enough so that it can be thought of as deceleration of a vehicle.

$$G(\theta) = -\ 10(sin^2\theta\ +\ 0.6\,sin(\theta^2)\ +\ 0.5\,sin\theta)\ -\ 0.12$$

$$G_1(\theta) = -\ 10(sin^2\theta\ +\ (0.6\ +\ a)\,sin(\theta^2)\ +\ 0.5\,sin\theta\ -\ sin(\frac{\theta^3}{100}))\ -\ b$$

$$a \in (0, 0.1)$$

$$b \in (0, 0.2)$$

Figure 2.1: Functions for G(θ) and G1(θ)

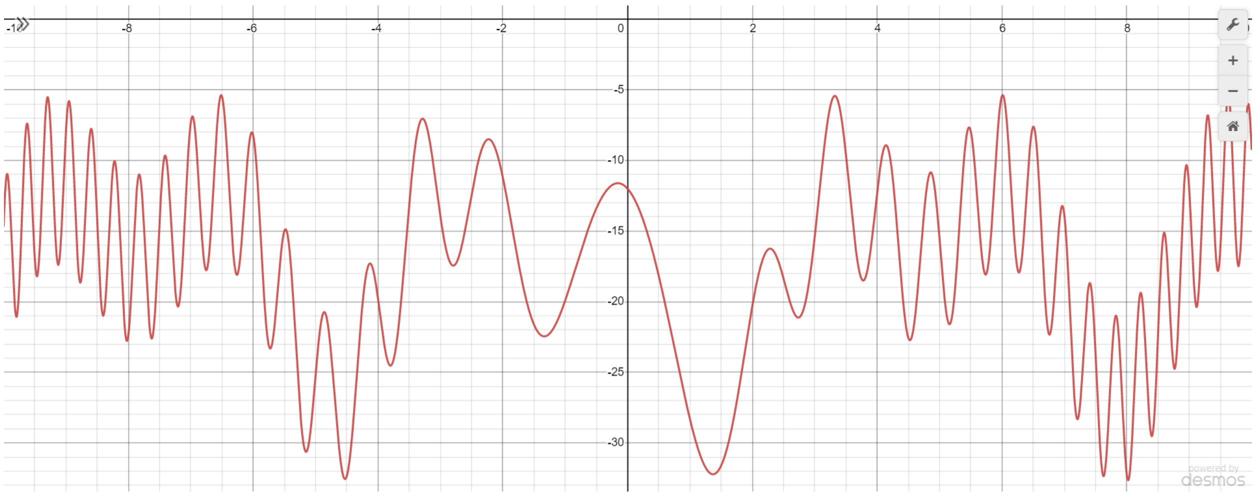
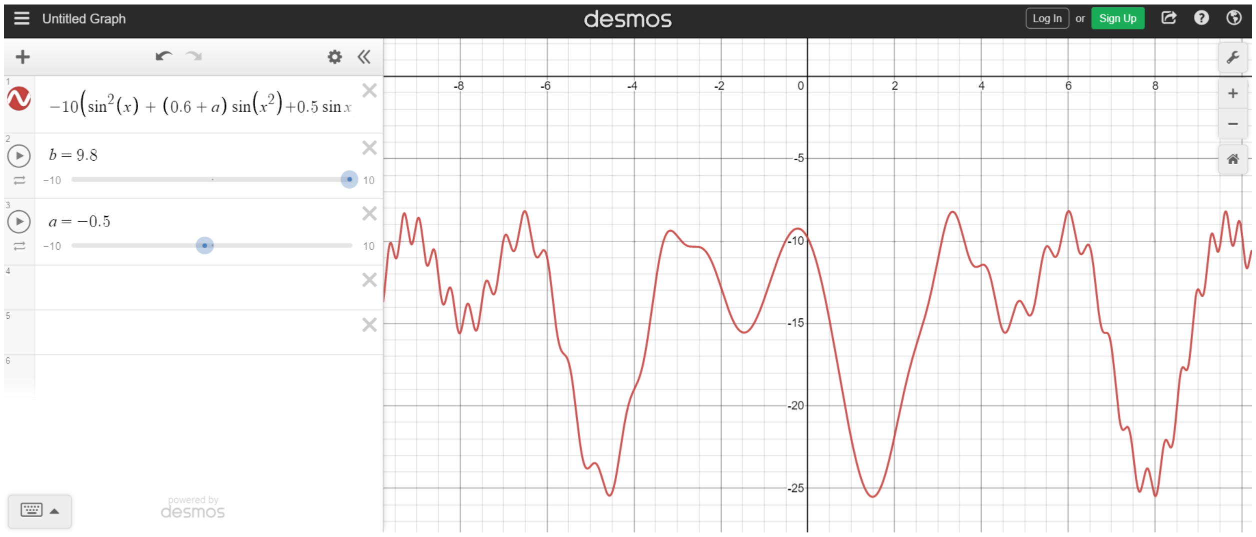


Figure 2.2: G(θ)



$-10\left(\sin^2(x) + (0.6 + a)\sin\left(x^2\right) + 0.5\sin x\right.$

$b = 9.8$ -10 … 10

$a = -0.5$ -10 … 10

Figure 2.3: G1(θ)

Mathematical Formulation

3.1 Defining the functions

3.1.1 Importing necessary libraries of python

```
import numpy as np
import math
import random
import matplotlib.pyplot as plt
```

3.1.2 G(θ)

```
def G(theta):
  accl = -10*((math.sin(math.radians(theta)))**2 +
      0.6*(math.sin((math.radians(theta))**2)) +
  0.5*math.sin(math.radians(theta))) - 0.12
  #accl = -3*math.sin(theta)
  return accl
```

This is a relation between the braking angle and the resulting acceleration of the car. This function solely depends on the mechanical parameters of the car such as the engine, fuel, and the dynamics of the system. Here we have taken any arbitrary function as
G(θ) = $\sin^2(\theta) + 0.5\sin(\theta^2) + 0.5\sin(\theta)$ - 0.12

We have taken G(0) as -0.12 because we are assuming that the acceleration due to friction is -0.12 m/s^2. The above function does not take into account the factors that are not certain such as environment, weather, etc.

To compensate this, we define a new function that helps us to compare the ideal value of acceleration with the real value of acceleration. And this further helps us to predict the optimum theta for braking.

3.1.3 G1(θ)

```
def G1(theta):
  accl = -10*((math.sin(math.radians(theta)))**2 +
      (0.6+random.randrange(0,10,1)/100)*(math.sin((math.radians(theta))**2)) +
      0.5*math.sin(math.radians(theta)) -
      math.sin(((math.radians(theta))**3)/100)) - (random.randrange(0,20,1))/100
  #accl = -3*math.sin(theta) + random.randrange(0,20)/100
  return accl
```

This function is very similar to G(θ) but this has random coefficients and hence it is realistic.

3.1.4 dG(θ)

```
def dG(theta):
  daccl = -10*(2*math.sin(math.radians(theta))*math.cos(math.radians(theta)) +
      0.6*math.sin((math.radians(theta))**2)
     *math.cos((math.radians(theta))**2)*2*(math.radians(theta)) +
         0.5*math.cos(math.radians(theta)))
  return daccl
```

3.1.5 Calculating Real Velocity

```
def real_vel(accl,s,v1):
  v_square = (v1**2 + 2*accl*s)
  if v_square >= 0:
    v_root = v_square**0.5
  else:
    v_root = 0
  return v_root
```

We calculated the real value of acceleration using the function G1(θ). This real value of acceleration is used in the above function to calculate the real velocity at a distance s.
We need real velocity because we need to take the difference of real velocity with the velocity predicted by interpolation.

3.1.6 Calculating derivative of velocity

```
def dvel(vel_list,interval):
  return (vel_list[len(vel_list)-1] - vel_list[len(vel_list)-2])/interval
```

This function will be useful while calculating the value of acceleration using $a = v\frac{dv}{ds}$.

3.1.7 Main Function

```
initial_vel = 10
initial_dist = 1000
dist_int = 1
final_vel = 0
final_dist = 0
n=3
dist = initial_dist
vel = initial_vel
pred_vel_list = [initial_vel]
real_vel_list = [initial_vel,final_vel]
dist_list = [initial_dist,final_dist]
error = 0
thetas = []
y = [[0 for i in range(1000)]
       for j in range(1000)];


while True:
  dist -= dist_int
  if dist <= final_dist:
    print('velocity error = ' + str(error))
    print('absolute velocity error = ' + str(real_v-final_vel))
    break
  #pred_vel =
     lagrangian(dist,1,dist_list[(len(dist_list)-2):],real_vel_list[(len(real_vel_list)-2):])
  #pred_vel = lagrangian(dist,(len(real_vel_list)-1),dist_list,real_vel_list)
  for i in range((len(dist_list))):
    y[i][0]=real_vel_list[i]
  pred_vel=applyFormula(dist, dist_list, y, n)
  y=dividedDiffTable(dist_list, y, len(dist_list))
  #print(pred_vel)
  pred_vel_list.append(pred_vel)

  accl = abs(pred_vel)*(dvel(pred_vel_list,dist_int))
  #print(accl)
  if accl >= G(0):
    accl = G(0)
    #theta = 0
  if accl <= G(30):
    accl = G(30)
    #theta = 30
  theta = Newton(0.1,0.001,accl)
  if theta <= 0:
    theta = 0
  if theta >= 30:
    theta = 30
  thetas.append(theta)
  #print(theta)
  real_accl = G1(theta)
  real_vel_list.remove(final_vel)
  real_v = real_vel(real_accl,dist_int,real_vel_list[len(real_vel_list)-2])
  if real_v <= final_vel:
    real_vel_list.append(real_v)
    print('dist_error = ' + str(dist_list[len(dist_list)-2]))
    break
  real_vel_list.append(real_v)
  real_vel_list.append(final_vel)
  dist_list.remove(final_dist)
  dist_list.append(dist)
  dist_list.append(final_dist)
  error = real_v - pred_vel
print('thetas:')
print(thetas)
```

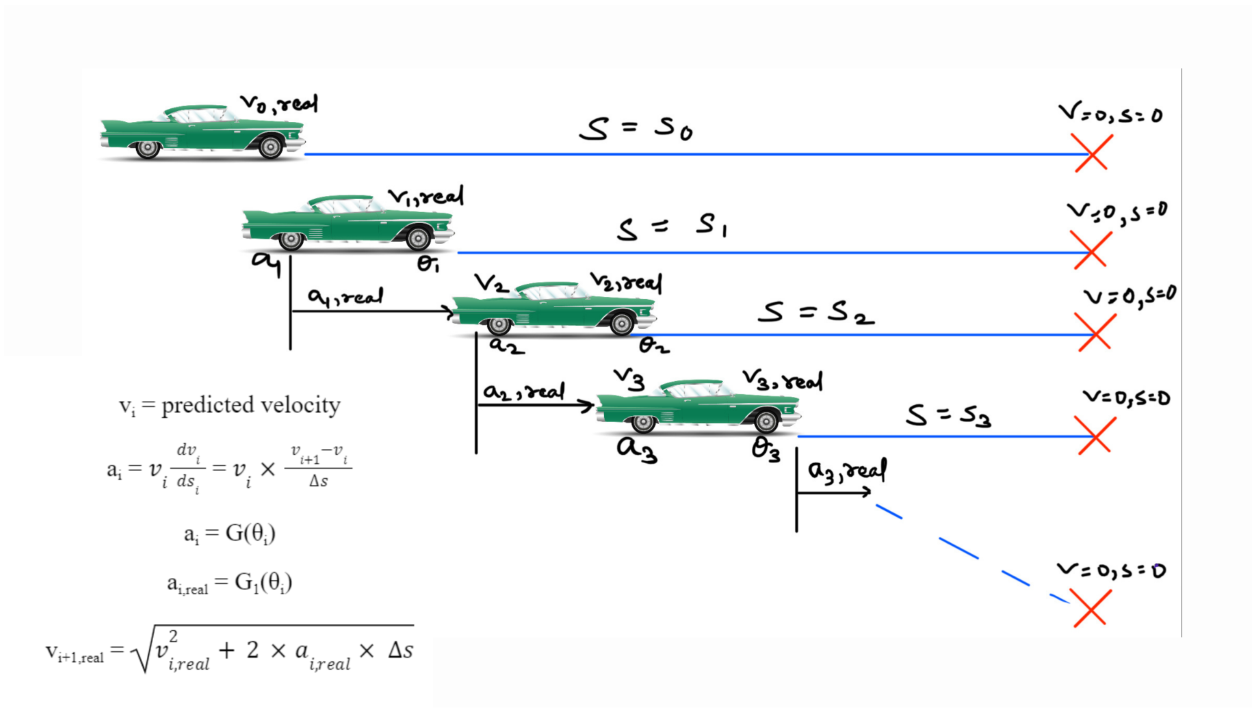


Figure 3.1: Analysis of car approaching a speed-breaker

Numerical Analysis

4.1 Newton's Method Code

```
def Newton(theta,error,accelaration):
  if abs(G(theta)-accelaration) <= error:
    return theta
  else:
    while True:
      theta_new = theta - ((G(theta)-accelaration)/(dG(theta)-accelaration))
      theta = theta_new
      if abs(G(theta_new)-accelaration) <= error:
        break
    return(abs(theta_new))
```

We have used Newton's method to find out the root of the function G(θ).
To recall, Newton's method is a technique for solving equations of the form f(x)=0 by successive approximation. The idea is to pick an initial guess x0 such that f(x0) is reasonably close to 0. We then find the equation of the line tangent to y=f(x) at x=x0 and follow it back to the x axis at a new (and improved!) guess x1.

4.2 Lagrange's Interpolation

```
def lagrangian(x,n,list_x,list_y):
  f = 0
  for i in range(n+1):
    f_ = list_y[i]
    for j in range(n+1):
      if j != i:
        f_ = f_*((x-list_x[j])/(list_x[i]-list_x[j]))
    f += f_
  return f
```

We have taken the velocity and distance as our inputs to predict the velocity of the vehicle at certain points using Lagrange's Interpolation.

4.3 Newton's divided difference interpolation

```
# Function to find the product term
def proterm(i, value, x):
   pro = 1;
   for j in range(i):
      pro = pro * (value - x[j]);
   return pro;

# Function to calculate divided difference table
def dividedDiffTable(x, y, n):
   for i in range(1, n):
      for j in range(n - i):
         y[j][i] = ((y[j][i - 1] - y[j + 1][i - 1]) /(x[j] - x[i + j]))
   return y;

# Function for applying Newtons divided difference formula
def applyFormula(value, x, y, n):
   sum = y[0][0]
   for i in range(1, n):
      sum = sum + (proterm(i, value, x) * y[0][i])
   return sum

# Function to display divided difference table
def printDiffTable(y, n):
   for i in range(n):
      for j in range(n - i):
         print(round(y[i][j], 4), "\t",end = " ")
      print("");

# calculating divided difference table
# y=dividedDiffTable(x, y, n)
```

This method was suggested by Prof. Tanya to our group at the time of project presentation. We will eventually see that the Newton's interpolating polynomial is better than the Lagrange's interpolating polynomial. The special feature of the Newton's polynomial is that the coefficients a_i can be determined using a very simple mathematical procedure.

We can see one beauty of the method is that, once the coefficients are determined, adding new data points won't change the calculated ones, we only need to calculate higher differences continues in the same manner. The whole procedure for finding these coefficients can be summarized into a divided differences table.

Results

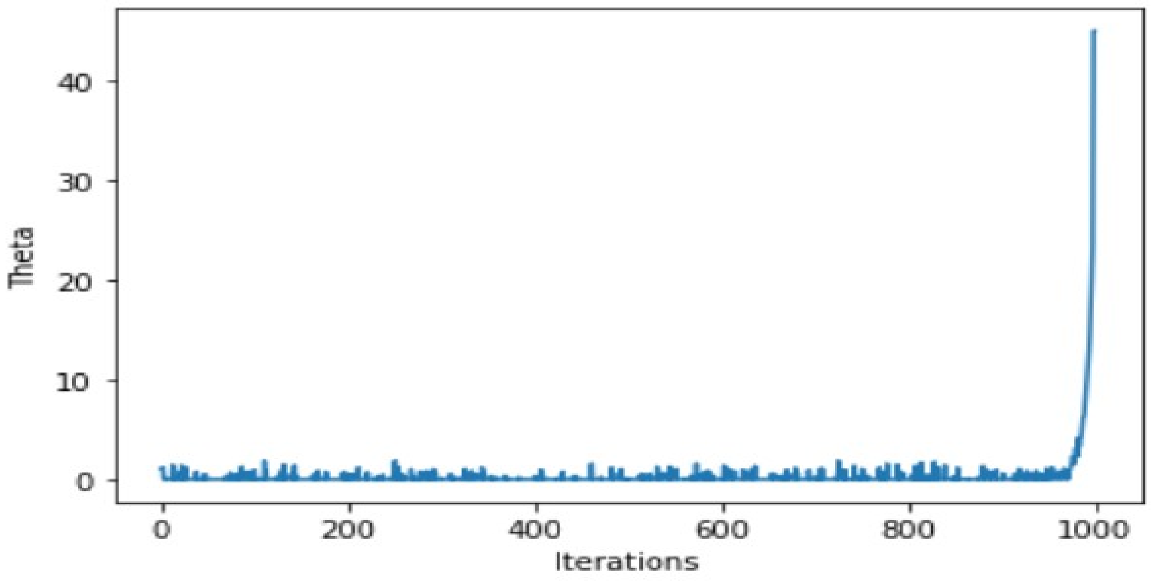


Figure 5.1: Results for first order interpolation

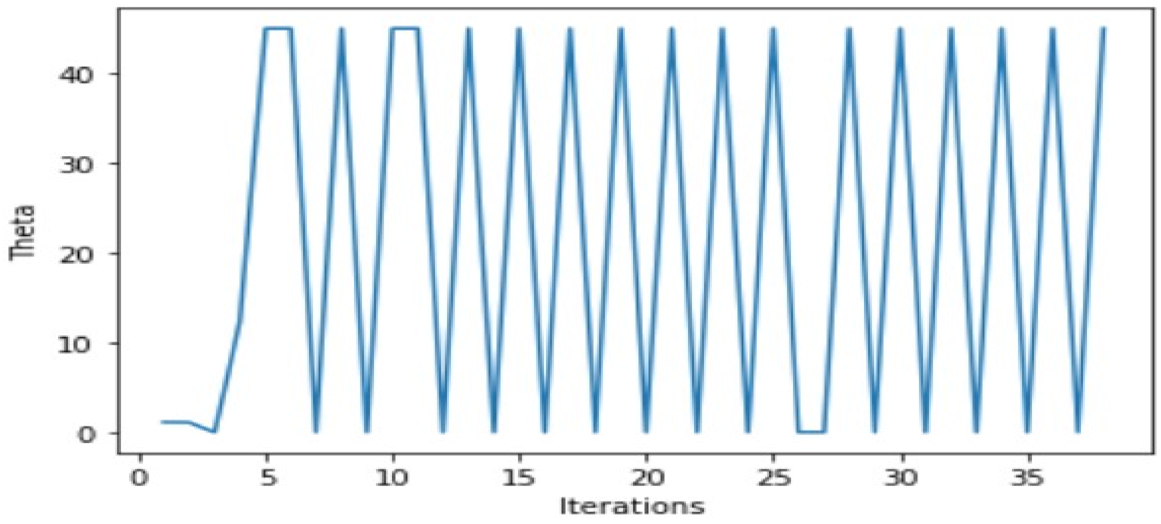


Figure 5.2: Results for nth order interpolation

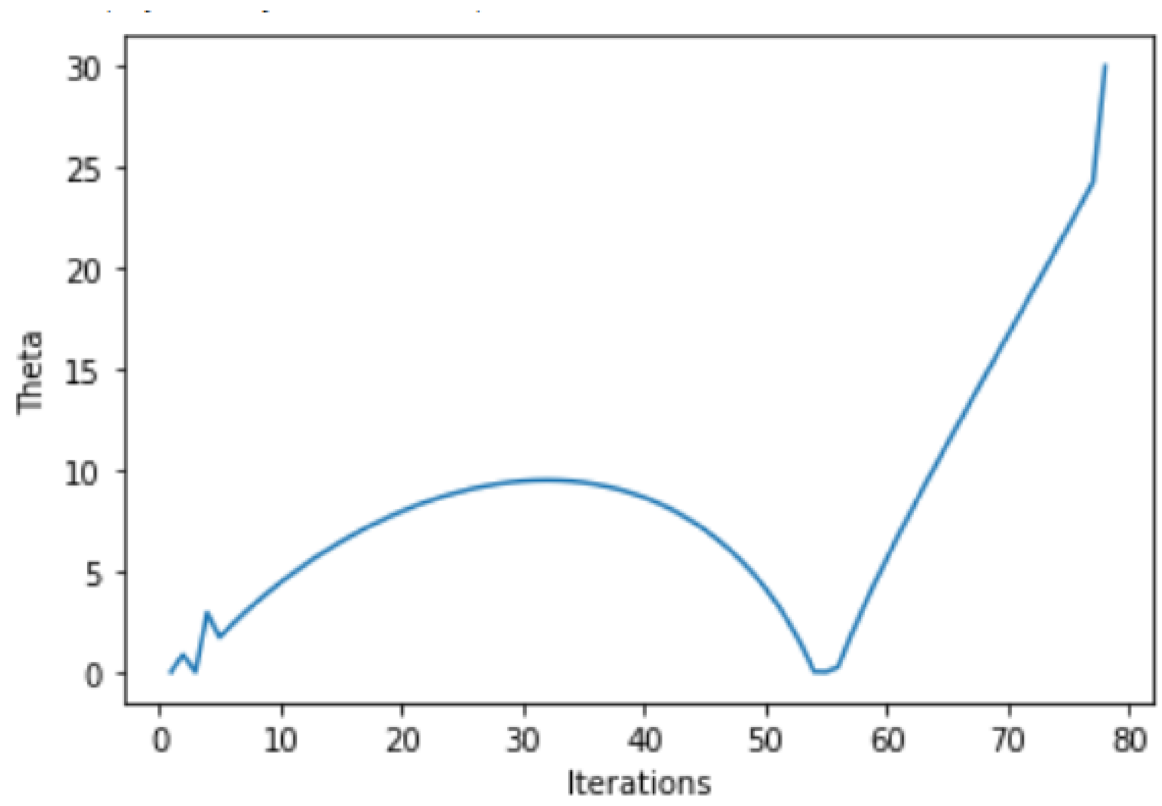


Figure 5.3: Results for Newton-Divided Interpolation

Observations and Conclusions

- For first order Lagrange interpolation, the value of θ very less (close to zero) for most of the duration. This means that the numerical method is not able to properly interpolate the values of θ that should be applied by the car for optimal braking. This leads to sudden braking towards the end, which is not desirable.
- For nth order Lagrange interpolation, the values of θ suddenly increases, reaches a threshold, remains there for some time, and again decreases. Such oscillations occur repeatedly. We also observe that frequency of these oscillations of value of θ increases over the number of iterations as the car moves closer to the obstacle. This shows that the numerical method is over estimating what is supposed to be done to get optimal braking angle. Hence, this is also not desirable.
- When Newton divided interpolation is used, we see that the results are much better. Initially there is a steep increase in θ for a short instant of time. Then, there is a smooth increase in θ for some iterations. In the middle, θ smoothly decrease a bit and then it linearly increases toward the end. This is not the best result, but surely the best we have achieved so far.
- Lagrange interpolation is particularly convenient when the same values of points are repeatedly used in several applications. The data values can be stored in computer memory and number of computations can thus be reduced.
- Limitations of the Lagrange interpolation occur when additional data points are added or removed to improve the appearance of the interpolating curve. The data set should be completely re-calculated every time when the data points are added or removed.
- It is thus harder to control the optimal appearance of the curve with the Lagrange interpolation algorithm. Hence, we use the Newton polynomial interpolation to get better results.

Authors' Contribution

The entire report was studied and written collaboratively by all members of the Group. Members of the association had many comprehensive meetings and discussions. The study's work was equally distributed among the participants. To better grasp the topic of the paper, the whole team took part in the analysis.Yash and Vikash together contributed to the research and study, while Anavart, Varun and Ruchit collaboratively contributed on writing the code and the approach. Finally, all of the group members worked tirelessly to grasp the project's topic and ensuring that the study was completed successfully.

References Used

- https://analyticsindiamag.com/10-creative-safety-features-for-driverless-car/
- https://dmpeli.math.mcmaster.ca/Matlab/Math4Q3/NumMethods/Lecture2-2.html
- https://electrek.co/2021/11/15/tesla-serious-phantom-braking-problem-autopilot/
- https://www.geeksforgeeks.org/newtons-divided-difference-interpolation-formula/
- https://www.teendriversource.org/learning-to-drive/self-driving-cars-adas-technologies