Big Homework 1

Mohamed Nguira

Problem:

We have a mobile vehicle, which should survive after the track. We have some predefined trajectory, which is given in y(x) format — our goal to pass this trajectory as fast as possible. But at the end of the course, there is a drop-off. It means that we should stop our machine at the end of the trajectory. We have to establish some constraints, such as max tangent acceleration (max power on the motor) and normal (road adhesion). Despite it, the best way is to represent our robot as a particle. How should we move (speed and acceleration) for solving such a task?

Parameters:

$$a_{t_{max}} = 10m/s^2$$
$$a_{n_{max}} = 6m/s^2$$
$$v_{max} = 1.5m/s$$

$y(x) = A * sin(om \cdot x + \theta_0)$, where $A = 1, om = 3, \theta_0 = 0.2$, x exists $[0...4]$

1) Tools:

Python with matplotlib

2) Solution description:

First, let's write y as a function of x:

$$y(x) = sin(3x + 0.2)$$

its derivative with respect to x:

$$y'(x) = 3cos(3x + 0.2)$$

its second derivative with respect to x:

$$y''(x) = -9sin(3x + 0.2)$$

We know that tangential acceleration is responsable to make the object move faster and normal acceleration makes it change direction faster. Hypothetically, if there was no limit on the normal acceleration we can put the tangential acceleration on the maximum for the first part of the movement, 0 for the second part and on the minimum for the third part so that the object stops.

To compute the normal acceleration needed so that the object follow the path, we use the radius of curvature formula:

$$r(x) = \frac{(1 + y'(x)^2)^{1.5}}{|y''(x)|}$$

$$a_n = \frac{v^2}{r}$$

However since we have a limit over the normal acceleration, there will also be a second limit over the maximum velocity. The maximum velocity will then be:

$$v_{max} = min(1.5, \sqrt{6r})$$

Note: While experimenting, my tangential acceleration was sometimes exceeding the limit by around 2% so I reduced the maximum velocity by 5% which solved the problem while keeping a good achieved time.

Let us define all the functions mentioned:

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure
from math import *

def y(x):
  return sin(3 * x + 0.2)

def y_d(x):
  return 3 * cos(3 * x + 0.2)

def y_d_d(x):
  return -9 * sin(3 * x + 0.2)

def r(x):
  return (1 + y_d(x) ** 2) ** 1.5 / abs(y_d_d(x))

def vmax(x):
  return min(1.5,sqrt(6 * r(x))) * 0.95

l = 8.7287

vinit = 0
```

Starting from this point, we will split the trajectory into several parts considered linear, as small as possible. For each small path we want to either increase the tangential acceleration to the maximum, keep the current value or decrease it to the minimum. This will depend on the constraints put on the velocity for the given time.

The following code computes the time needed for the acceleration part numerically by calculating the distance needed to traverse for each little path and using it to find resultant velocity after the robot traverse it and from that calculating the time.

I used the following relation to calculate the final velocity:

$$v_f^2 - v_0^2 = 2as$$

```
x0 = 0
dx = 1e-5
l = 0
v = 0
time = 0
at = 10
yplot = []
xplot = []
timeplot = []
velplot = []
acctplot = []
accnplot = []

#acceleration:
while v < vmax(x0):
  yplot += [y(x0)]
  xplot += [x0]
  timeplot += [time]
  velplot += [v]
  accnplot += [v*v / r(x0)]

  dy = y(x0 + dx) - y(x0)
  ds = (dx **2 + dy ** 2) ** 0.5
  vf = (2 * 10 * ds + v**2)**0.5
  x0 += dx
  time += (vf - v)/at
  v = vf
  acctplot += [at]
  l += ds
```

Same approach was used to find the deceleration time and the specific x when to start deceleration. We just assume the robot start in the end point and want to reach the maximum velocity from there traversing the reverse trajectory.

```
#Finding Time for deceleration:
x1 = 4
v1 = 0
while v1 < vmax(x1):
  dy = y(x0 - dx) - y(x0)
  ds = (dx **2 + dy ** 2) ** 0.5
  vf = (2 * 10 * ds + v1**2)**0.5
  x1 -= dx
  v1 = vf
```

Compute the time needed, accelerations and velocities for the middle part of the trajectory. Everything is done numerically.

```
while x0 < x1:
  yplot += [y(x0)]
  xplot += [x0]
  timeplot += [time]
  velplot += [v]
  accnplot += [v*v / r(x0)]

  dy = y(x0 + dx) - y(x0)
  ds = (dx **2 + dy ** 2) ** 0.5

  dvpotential = (2 * 10 * ds + v**2)**0.5 - v
  dv = vmax(x0 + dx)  - v
  vf = 0
  if dv < 0:
      vf = v + dv
  else:
    vf = min(dvpotential,dv) + v
  a = abs(vf**2 - v**2) / (2 * ds)
  acctplot += [(vf**2 - v**2) / (2 * ds)]
  #print(a, " ", ds, " ", vf, " ", v)
  if a == 0:
    time += ds / v
  else :
    time += abs((vf - v)) / a

  v = vf
  l += ds
  x0 += dx


while x0 < 4:
  yplot += [y(x0)]
  xplot += [x0]
  timeplot += [time]
  velplot += [v]
  accnplot += [v*v / r(x0)]

  dy = y(x0 - dx) - y(x0)
  ds = (dx **2 + dy ** 2) ** 0.5
  vf = (2 * 10 * ds + v**2)**0.5
  x0 += dx
  time += (vf - v)/at
  v -= vf - v
  acctplot += [-at]
  l += ds



figure(figsize=(8, 4), dpi=80)
plt.plot(xplot,yplot,label = "y(x)")
plt.legend()

figure(figsize=(8, 4), dpi=80)
plt.plot(timeplot,yplot,label = "y(t)")
plt.legend()

figure(figsize=(8, 4), dpi=80)
plt.plot(timeplot,xplot,label = 'x(t)')
plt.legend()

figure(figsize=(8, 4), dpi=80)
plt.plot(timeplot,velplot,label = 'v(t)')
plt.legend()

figure(figsize=(8, 4), dpi=80)
plt.plot(timeplot,acctplot,label = 'at(t)')
plt.legend()

figure(figsize=(8, 4), dpi=80)
plt.plot(timeplot,accnplot,label = 'an(t)')
plt.legend()

print("Curve length is: ", l)
print("Time needed is: ", time)
```

```
Curve length is:  8.72823942817396
Time needed is:  6.597176177360877
```