Name- Mangirish Sanjeev Kulkarni.

ASU ID- 1223229852.

Project 1- Rocket Landing (Atlantis STS-129).

Introduction-

The given project's objective is demonstrated with the below mentioned formulation considering a simlpe case of optimizing the trajectory for a rocket landing.

Consider a simple formulation of rocket landing where the rocket state $x(t)$ is represented by its distance to the ground $d(t)$ and its velocity $v(t)$, i.e., $x(t) = [d(t), v(t)]^T$, where t specifies time. The control input of the rocket is its acceleration $a(t)$. The discrete-time dynamics follows

$$d(t+1) = d(t) + v(t)\Delta t,$$
$$v(t+1) = v(t) + a(t)\Delta t,$$

where Δt is a time interval. Further, let the closed-loop controller be

$$a(t) = f_\theta(x(t))$$

where $f_\theta(\cdot)$ is a neural network with parameters θ, which are to be determined through optimization.

For each time step, we assign a loss as a function of the control input and the state: $l(x(t), a(t))$. In this example, we will simply set $l(x(t), a(t)) = 0$ for all $t = 1, \ldots, T-1$, where T is the final time step, and $l(x(T), a(T)) = ||x(T)||^2 = d(T)^2 + v(T)^2$. This loss encourages the rocket to reach $d(T) = 0$ and $v(T) = 0$, which are proper landing conditions.

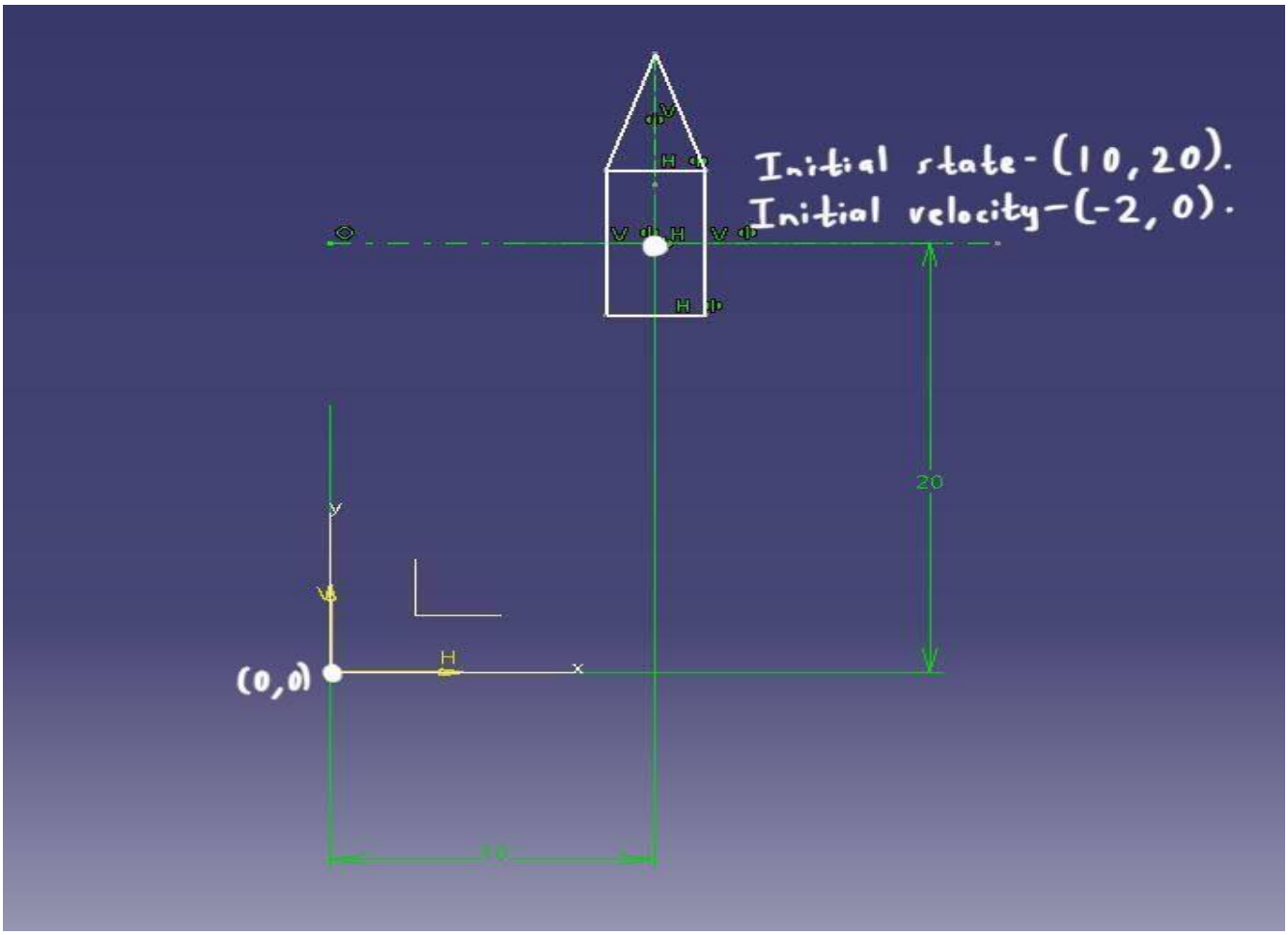
The optimization problem is now formulated as

$$\min_\theta \quad amp; ||x(T)||^2$$
$$d(t+1) = d(t) + v(t)\Delta t,$$
$$v(t+1) = v(t) + a(t)\Delta t,$$
$$a(t) = f_\theta(x(t)),\ \forall t = 1, \ldots, T-1$$

While this problem is constrained, it is easy to see that the objective function can be expressed as a function of $x(T-1)$ and $a(T-1)$, where $x(T-1)$ as a function of $x(T-2)$ and $a(T-2)$, and so on. Thus it is essentially an unconstrained problem with respect to θ.

In the following, we code this problem up with PyTorch, which allows us to only build the forward pass of the loss (i.e., how we move from $x(1)$ to $x(2)$ and all the way to $x(T)$) and automatically get the gradient $\nabla_\theta l(x(T), a(T))$.

1. Problem Formulation-



For my problem formulation, I have chosen a 2-D scenario for rocket landing with the addition of drag force causing deceleration in Y axis alone. The initial co-ordinates of the rocket are (10,20) with initial velocities in x and y direction being (-2,0) respectively. The rocket has thrusters in both X and Y directions with the same thrust constant to control its descent.

The Project objective is to land the rocket as close to the target landing site i.e., point (0,0) with zero velocity at landing. For the loss function the chosen control parameters are Y, V_y, X and V_x.

Objective function -

$$\min_{\theta} \; ||s(T)||^2$$

where,

$$s(T) = [Y(T), V_y(T), X(T), V_x(T)]$$

Variables -

S: State
T: Time
X: X-co-ordinate
Y: Y-co-ordinate
Vx: Velocity in x direction
Vy: Velocity in y direction
ax: Acceleration in x-direction
ay: Acceleration in y-direction

Constraints -

$$X(T+1) = X(T) + V_x(T)\Delta T;$$
$$Y(T+1) = Y(T) + V_y(T)\Delta T;$$

$$V_x(T+1) = V_x(T) + a_x(T)\Delta T;$$
$$V_y(T+1) = V_y(T) + a_y(T)\Delta T - a_{y\,drag}(T)\Delta T;$$

$$a_x(T), a_y(T) = f_\theta[x(T)], \; \forall t = 1, \cdots, T-1.$$

Calculations for Drag force and Drag Deceleration-

Rocket model- Atlantis STS-129

Ref: http://www.spacefacts.de/mission/english/sts-129.htm...(Link 1)
Ref: https://en.wikipedia.org/wiki/STS-129... (Link 2)

Drag effect calculations-

Drag force: **DF = Cd * A * 0.5 * row * V^2** ... (Eqn 1)

Ref: https://www.grc.nasa.gov/www/k12/rocket/drageq.html#:~:text=The%20drag%20equation%20states%20that,times%20the%20reference%20area%20A.&text=For%20given%20air%20conditions%2C%20shape,for%20Cd%20to%20determine%20drag.)

Cd = 0.5 (Drag co-efficient) ... (Link 2)

A = pi * 16 feet ^2 (16 feet=4.8768m) = 74.71705804 m^2 ... (Link 2)

row = 8050 kg/m^3 (steel density, Ref: https://www.toppr.com/guides/physics/fundamentals/density-of-steel-how-to-calculate-the-density-of-metal/#:~:text=That's%20why%20the%20density%20of,the%20densest%20alloy%20or%20metal.)

V = velocity = delta state (Link 1; Landing Vel= 365 km/hr.=101.389 m/s)

m = 93063 Kg (Link 1; Mass at landing of Atlantis STS-129)

Inputting in eqn 1, we get,
DF = 0.5 * 74.71705804 * 0.5 * 8050 * V^2
DF = 150368.0793 * V2**

Declaration due to drag-

DRAG_DECELERATION (deceleration due to drag): **DD = DF/m.**
DD = 1.615766516 * V^2.

Assumption- After calculating the drag force, the rocket is assumed to be a point mass and only its movement is controlled.

2. CODE-

In [19]:
```
#Project 1
#Mangirish Kulkarni
#ASU Id- 1223229852

import logging
import math
import random
import numpy as np
import time
import torch as t
import torch.nn as nn
from torch import optim
from torch.nn import utils
import matplotlib.pyplot as plt

logger = logging.getLogger(__name__)
```

In [20]:
```
# environment parameters-

FRAME_TIME = 0.1  # time interval
GRAVITY_ACCELERATION = 9.81  # gravity const g = 9.81 m/s^2
BOOST_ACCELERATION = 14.715  # assuming thrust const to be 14.715 m/s^2 (Usually >g)

# # Unused parameters-
# PLATFORM_WIDTH = 0.25  # landing platform width
# PLATFORM_HEIGHT = 0.06  # landing platform height
# ROTATION_ACCEL = 20  # rotation constant
```

In [21]:
```
# define system dynamics-
# Notes:
# 0. You only need to modify the "forward" function
# 1. All variables in "forward" need to be PyTorch tensors.
# 2. All math operations in "forward" has to be differentiable, e.g., default PyTorch functions.
# 3. Do not use inplace operations, e.g., x += 1. Please see the following section for an example that does not work.

class Dynamics(nn.Module):

    def __init__(self):
        super(Dynamics, self).__init__()

    @staticmethod
    def forward(state, action):

        """
        action: thrust or no thrust
        action[0] = y-direction thrust
        action[1] = x-direction thrust
        state[0] = y
        state[1] = y_dot
        state[2] = x
        state[3] = x_dot
        """

        # Apply gravity
        # Note: Here gravity is used to change velocity which is the second element of the state vector
        # Normally, we would do x[1] = x[1] + gravity * delta_time
        # but this is not allowed in PyTorch since it overwrites one variable (x[1]) that is part of the computational g
raph to be differentiated.
        # Therefore, I define a tensor dx = [0., gravity * delta_time], and do x = x + dx. This is allowed...

        delta_state_gravity = t.tensor([0., GRAVITY_ACCELERATION * FRAME_TIME, 0., 0.]) #2D conversion

        # For Thrust
        # Note: Same reason as above. Need a 2-by-1 tensor.
        delta_state_x = BOOST_ACCELERATION * FRAME_TIME * t.tensor([0., 0., 0., 1.]) * action[1]  #2D conversion in x
        delta_state_y = BOOST_ACCELERATION * FRAME_TIME * t.tensor([0., -1., 0., 0.]) * action[0] #2D conversion in y
        V = delta_state_y

        """""
        Rocket model- Atlantis STS-129
        Ref: http://www.spacefacts.de/mission/english/sts-129.htm   ...(Link 1)
        Ref: https://en.wikipedia.org/wiki/STS-129  ...(Link 2)

        Drag effect calculations-
        Drag force: DF = Cd * A * 0.5 * row * V^2   ...(Eqn 1; Ref:https://www.grc.nasa.gov/www/k-12/rocket/drageq.html
```

```
#:~:text=The%20drag%20equation%20states%20that,times%20the%20reference%20area%20A.&text=For%20given%20air%20conditions%2C%20shape,for%20Cd%20to%20determine%20drag.)

        Cd = 0.5 (Drag co-efficient) ...(Link 2)
        A = pi * 16 feet ^2 (16 feet=4.8768m) = 74.71705804 m^2 ...(Link 2)
        row = 8050 kg/m^3 (steel density, Ref:https://www.toppr.com/guides/physics/fundamentals/density-of-steel-how-to-calculate-the-density-of-metal/#:~:text=That's%20why%20the%20density%20of,the%20densest%20alloy%20or%20metal.)
        V = velocity = delta state (Link 1; Landing Vel= 365 km/hr=101.389 m/s)
        m = 93063 Kg (Link 1; Mass at landing of Atlantis STS-129)

        Inputing in eqn 1, we get,
        DF = 0.5 * 74.71705804 * 0.5 * 8050 * V**2
        DF = 150368.0793 * V**2

        Now Decl due to drag-

        DRAG_DECELERATION (deceleration due to drag) = DF/m

        """
        #Drag-
        DRAG_DECELERATION = 1.615766516 * V**2

        NET_ACCELERATION = BOOST_ACCELERATION - DRAG_DECELERATION # Net accln in Y direction

        #Now new Velocity in Y-direction-
        delta_state_y = NET_ACCELERATION * FRAME_TIME * t.tensor([0., -1., 0., 0.]) * action[0]

        # Update velocity-
        state = state + delta_state_y + delta_state_gravity + delta_state_x

        # Update state-
        # Note: Same as above. Use operators on matrices/tensors as much as possible. Do not use element-wise operators as they are considered inplace.
        step_mat = t.tensor([[1., FRAME_TIME, 0., 0.],
                             [0., 1., 0., 0.],
                             [0., 0., 1., FRAME_TIME],
                             [0., 0., 0., 1.]])
        state = t.matmul(step_mat, state)

        return state
```

In [22]:
```
# a deterministic controller
# Note:
# 0. You only need to change the network architecture in "__init__"
# 1. nn.Sigmoid outputs values from 0 to 1, nn.Tanh from -1 to 1
# 2. You have all the freedom to make the network wider (by increasing "dim_hidden") or deeper (by adding more lines to
 nn.Sequential)
# 3. Always start with something simple

class Controller(nn.Module):

    def __init__(self, dim_input, dim_hidden, dim_output):
        """
        dim_input: # of system states
        dim_output: # of actions
        dim_hidden: up to you
        """
        super(Controller, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(dim_input, dim_hidden),
            nn.Tanh(),
            nn.Tanh(),#added new layer
            nn.Linear(dim_hidden, dim_hidden),#added new layer
            nn.Tanh(),#added new layer
            nn.Tanh(),#added a new layer
            nn.Linear(dim_hidden, dim_output),#added new hidden layer
            # You can add more layers here
            nn.Sigmoid()
        )

    def forward(self, state):
        action = self.network(state)
        return action
```

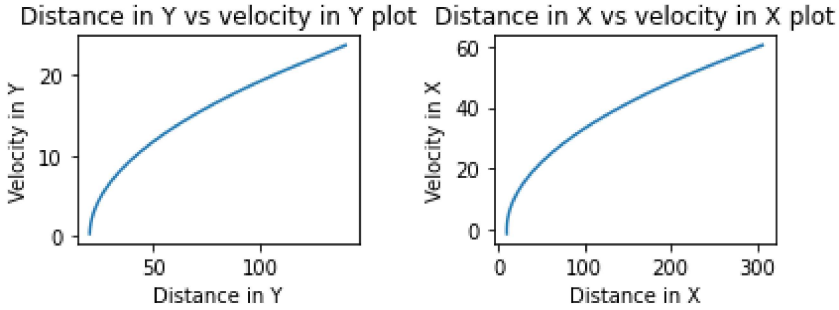
In [23]:
```
# the simulator that rolls out x(1), x(2), ..., x(T)
# Note:
# 0. Need to change "initialize_state" to optimize the controller over a distribution of initial states
# 1. self.action_trajectory and self.state_trajectory stores the action and state trajectories along time

class Simulation(nn.Module):

    def __init__(self, controller, dynamics, T):
        super(Simulation, self).__init__()
        self.state = self.initialize_state()
        self.controller = controller
        self.dynamics = dynamics
        self.T = T
        self.action_trajectory = []
        self.state_trajectory = []

    def forward(self, state):
        self.action_trajectory = []
        self.state_trajectory = []
        for _ in range(T):
            action = self.controller.forward(state)
            state = self.dynamics.forward(state, action)
            self.action_trajectory.append(action)
            self.state_trajectory.append(state)
        return self.error(state)

    @staticmethod
    def initialize_state():
        #Rocket is coming down from a height of 10 m with 0 m/s initial velocity in Y
        #Rocket is 5 m to the right of target landing side with -1 m/s initial velocity in X
        state = [20., 0., 10., -2.]  # TODO: need batch of initial states
        return t.tensor(state, requires_grad=False).float()

    def error(self, state):
        return state[0]**2 + state[1]**2 + state[2]**2 + state[3]**2
```

In [24]:
```
# set up the optimizer
# Note:
# 0. LBFGS is a good choice if you don't have a large batch size (i.e., a lot of initial states to consider simultaneous
ly)
# 1. You can also try SGD and other momentum-based methods implemented in PyTorch
# 2. You will need to customize "visualize"
# 3. loss.backward is where the gradient is calculated (d_loss/d_variables)
# 4. self.optimizer.step(closure) is where gradient descent is done

class Optimize:
    def __init__(self, simulation):
        self.simulation = simulation
        self.parameters = simulation.controller.parameters()
        self.optimizer = optim.LBFGS(self.parameters, lr=0.01)

    def step(self):
        def closure():
            loss = self.simulation(self.simulation.state)
            self.optimizer.zero_grad()
            loss.backward()
            return loss
        self.optimizer.step(closure)
        return closure()

    def train(self, epochs):
        for epoch in range(epochs):
            loss = self.step()
            print('_____________________________________________________________________________________________________
__________')
            print('\nIteration number:\t',epoch+1)
            print('loss: %.3f' % (loss))
            self.visualize()

    def visualize(self):
        data = np.array([self.simulation.state_trajectory[i].detach().numpy() for i in range(self.simulation.T)])
        a = data[:, 0]
        b = data[:, 1]
        c = data[:, 2]
        d = data[:, 3]

        plt.figure()
        plt.subplot(221)
        plt.title('Distance in Y vs velocity in Y plot')
        plt.xlabel('Distance in Y')
        plt.ylabel('Velocity in Y')
        plt.plot(a, b)
```

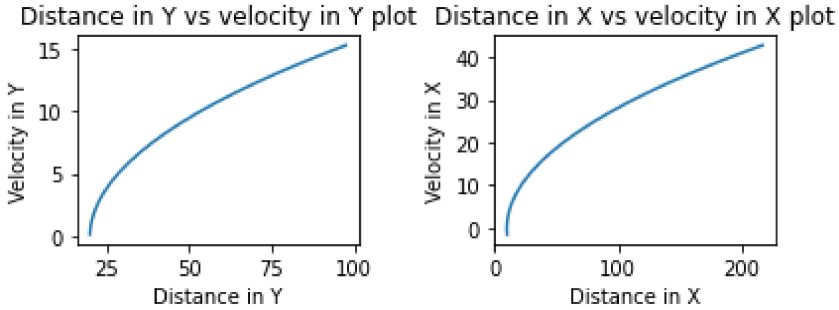
```
plt.subplot(222)
plt.title('Distance in X vs velocity in X plot')
plt.xlabel('Distance in X')
plt.ylabel('Velocity in X')
plt.plot(c, d)
plt.tight_layout()
plt.show()
```

In [25]:
```
# Now it's time to run the code!

T = 100  # number of time steps
dim_input = 4  # state space dimensions (x, x_dot, y, y_dot)
dim_hidden = 6  # latent dimensions
dim_output = 2  # action space dimensions (action_x, action_y)
d = Dynamics()  # define dynamics
c = Controller(dim_input, dim_hidden, dim_output)  # define controller
s = Simulation(c, d, T)  # define simulation
o = Optimize(s)  # define optimizer
o.train(60)  # solve the optimization problem
```

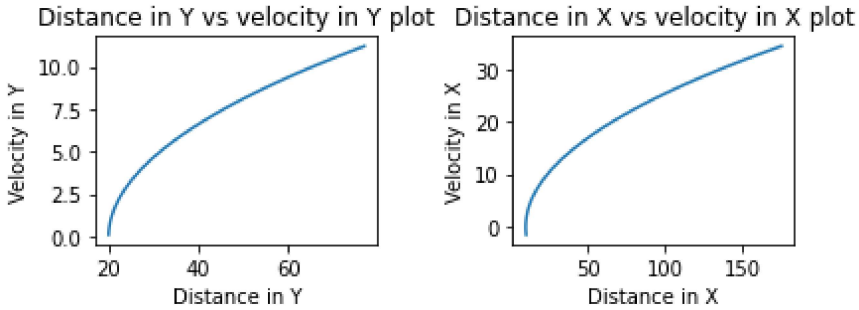
```
Iteration number:        1
loss: 117812.922
```



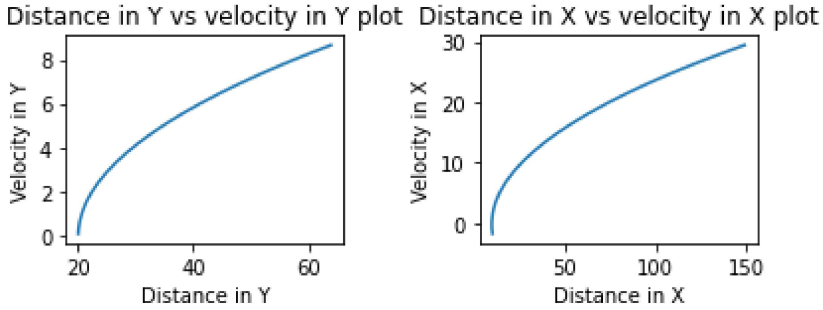
```
Iteration number:        2
loss: 58451.883
```



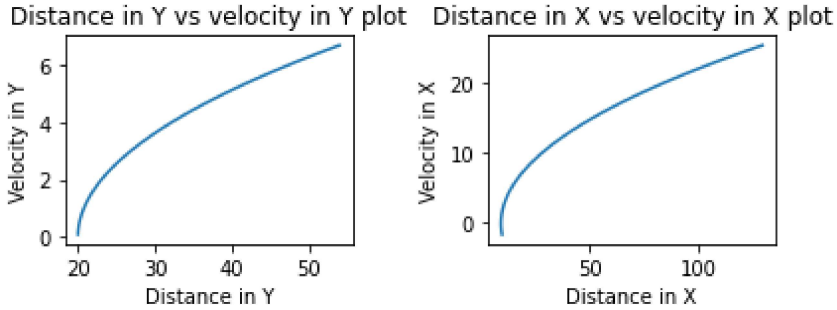
```
Iteration number:        3
loss: 37866.066
```



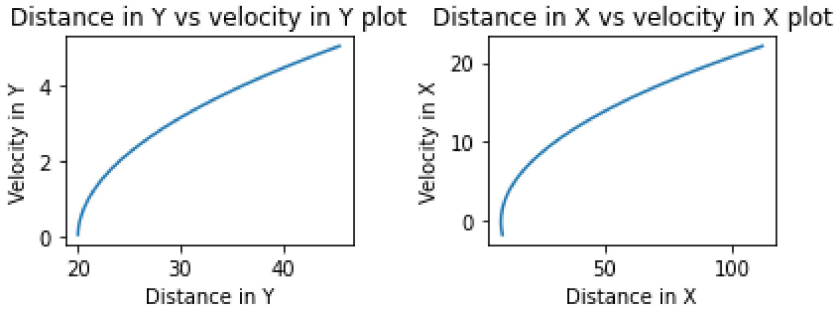
Iteration number: 4
loss: 27253.021



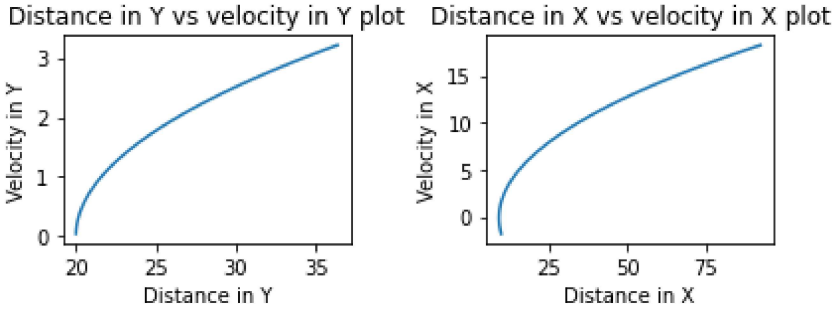
Iteration number: 5
loss: 20314.717



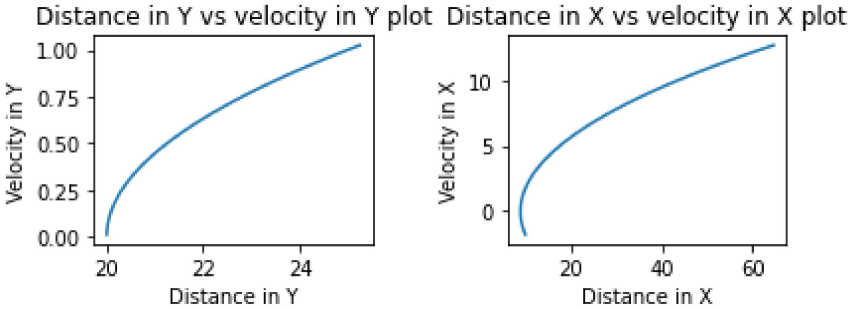
Iteration number: 6
loss: 15145.119



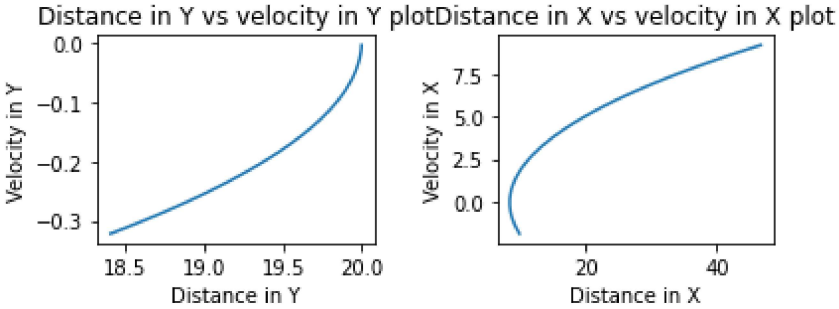
```
Iteration number:        7
loss: 10218.019
```



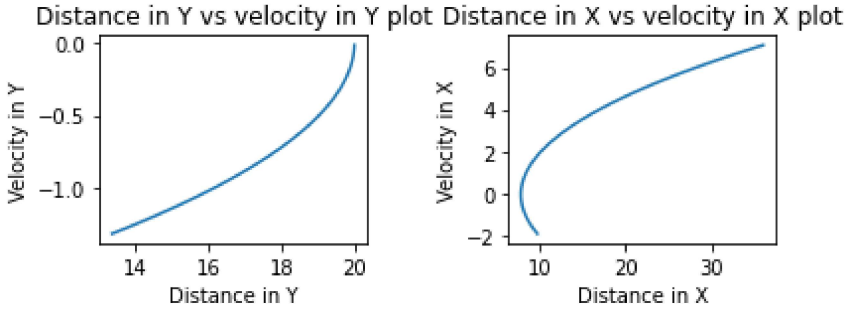
```
Iteration number:        8
loss: 4964.897
```



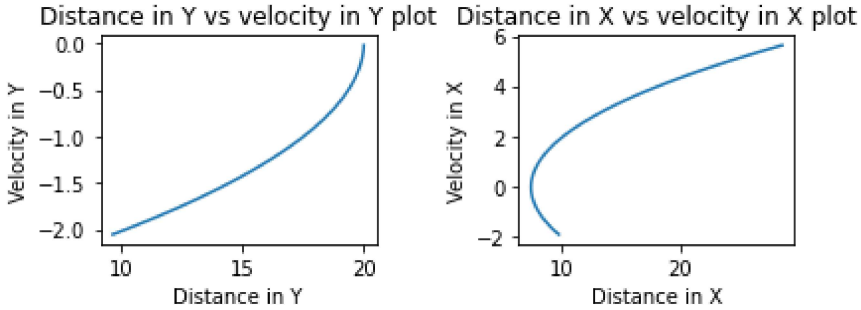
```
Iteration number:        9
loss: 2605.194
```



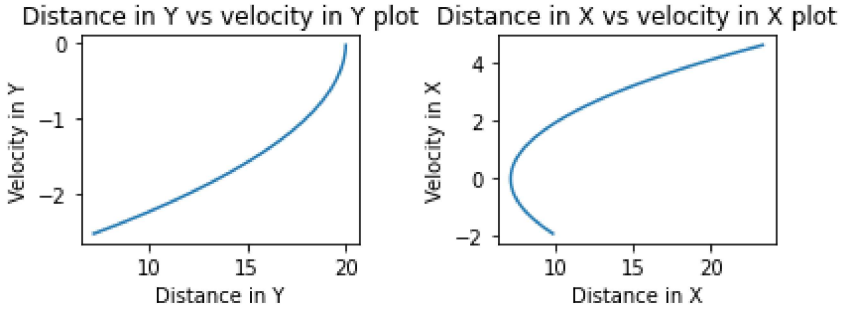
Iteration number: 10
loss: 1522.937



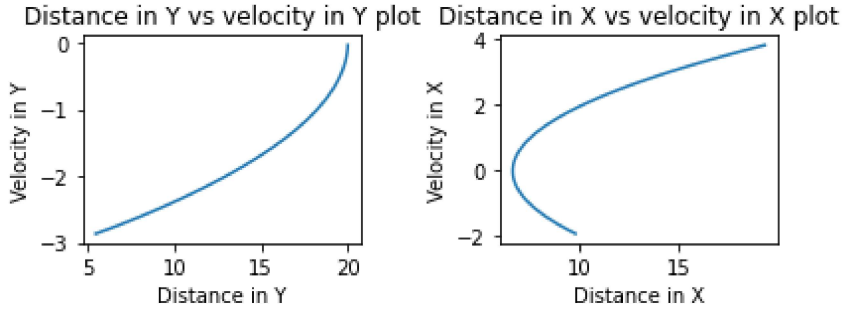
Iteration number: 11
loss: 944.113



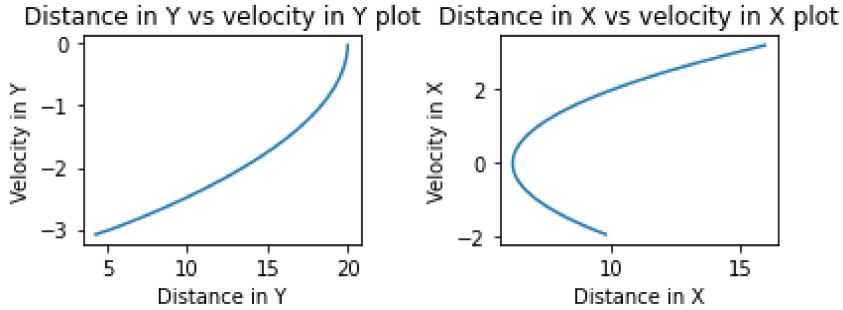
Iteration number: 12
loss: 625.791



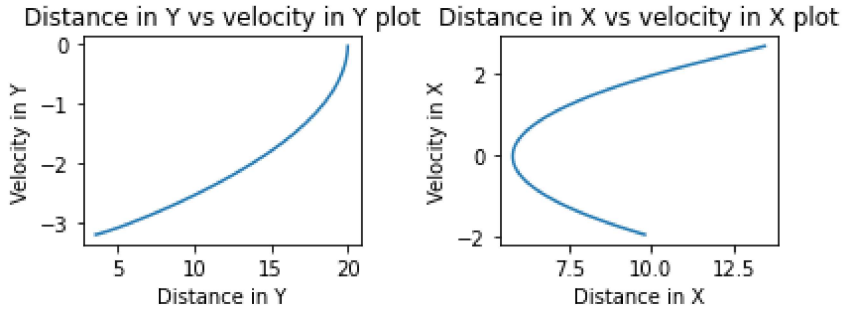
Iteration number: 13
loss: 425.820



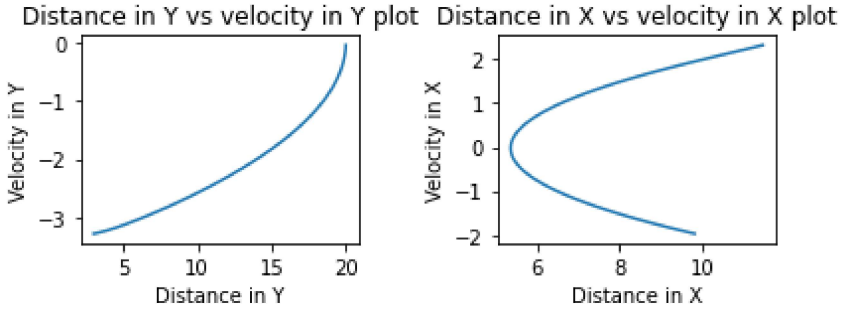
Iteration number: 14
loss: 293.219



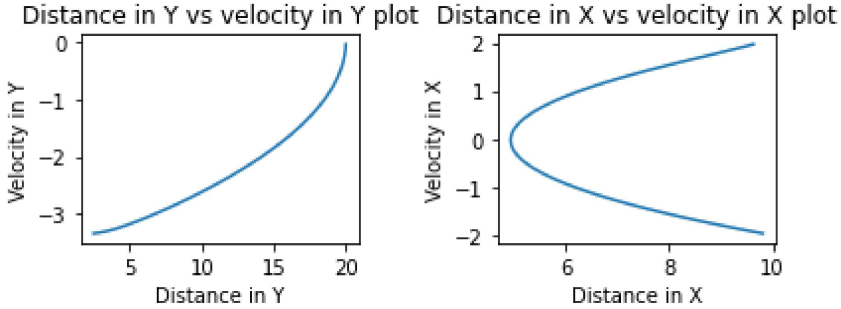
Iteration number: 15
loss: 211.271



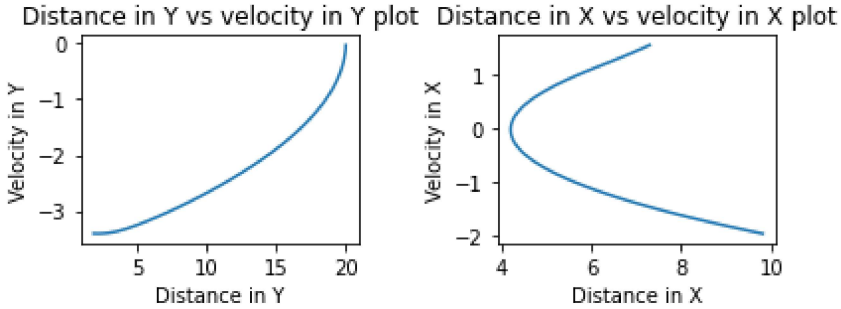
Iteration number: 16
loss: 156.190



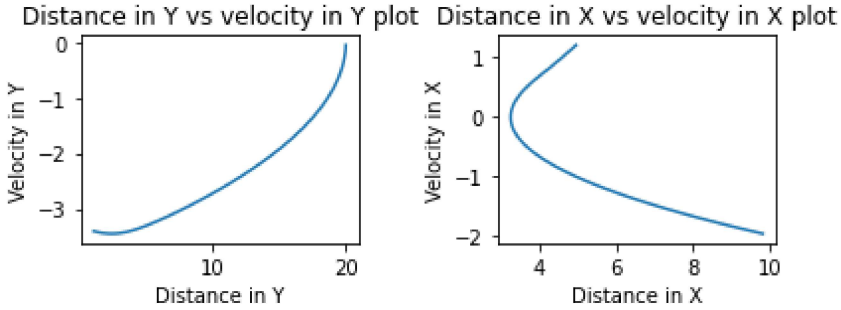
Iteration number: 17
loss: 113.921



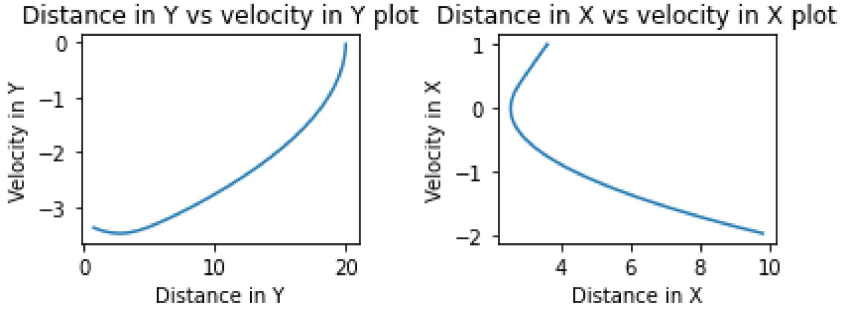
Iteration number: 18
loss: 70.593



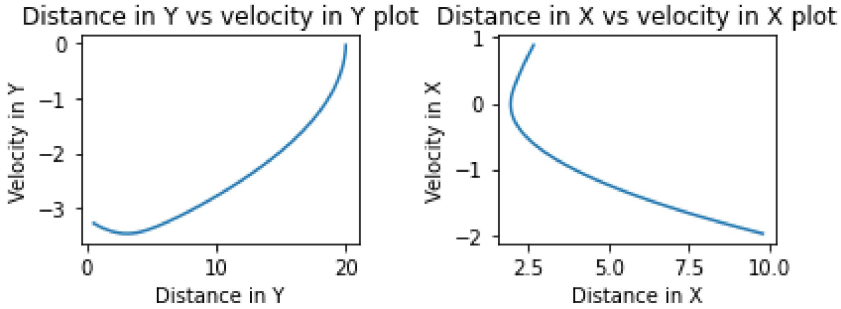
Iteration number: 19
loss: 39.016



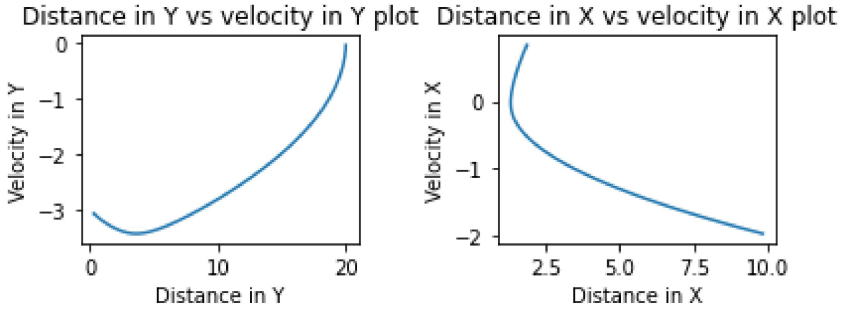
Iteration number: 20
loss: 25.951



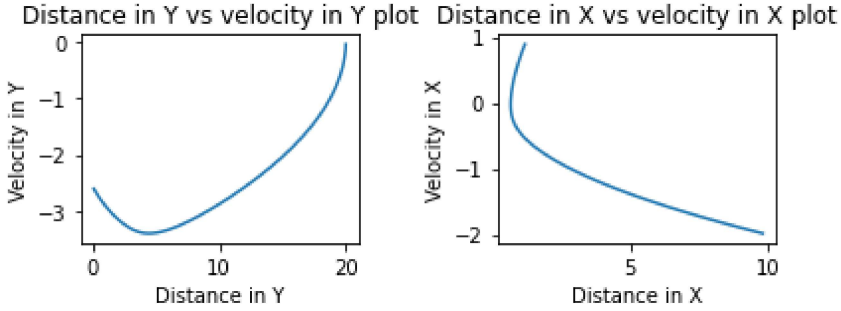
Iteration number: 21
loss: 18.920



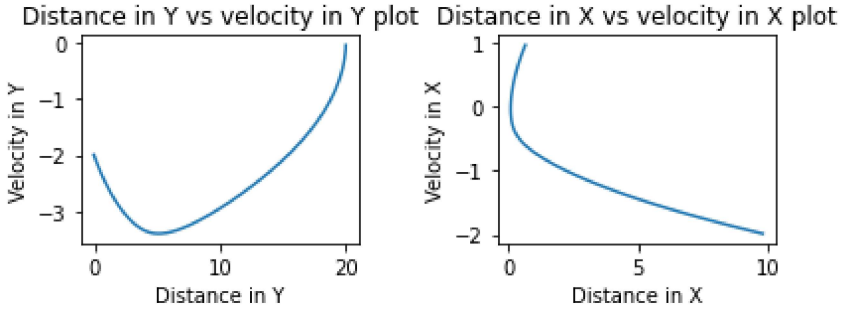
Iteration number: 22
loss: 13.783



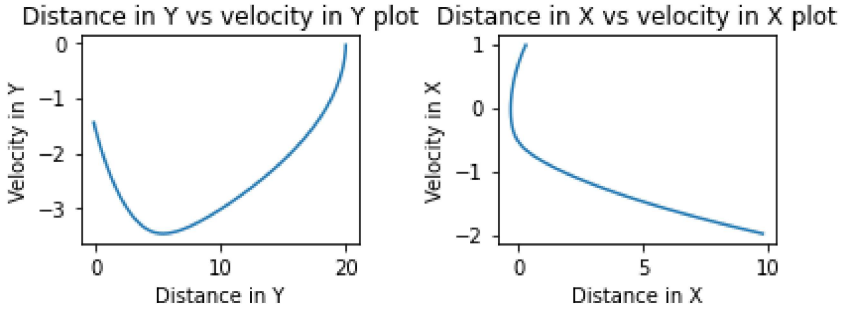
Iteration number: 23
loss: 8.922



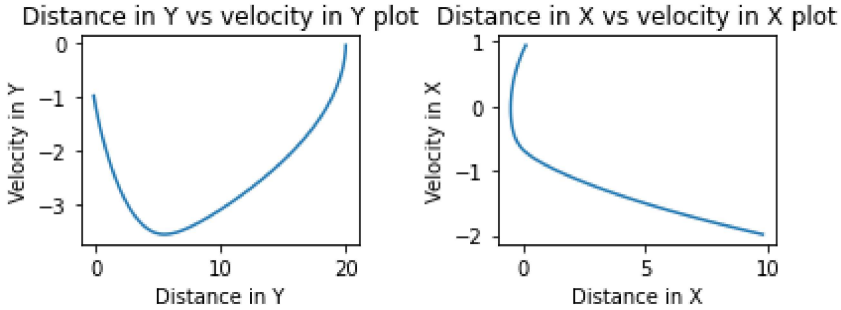
Iteration number: 24
loss: 5.347



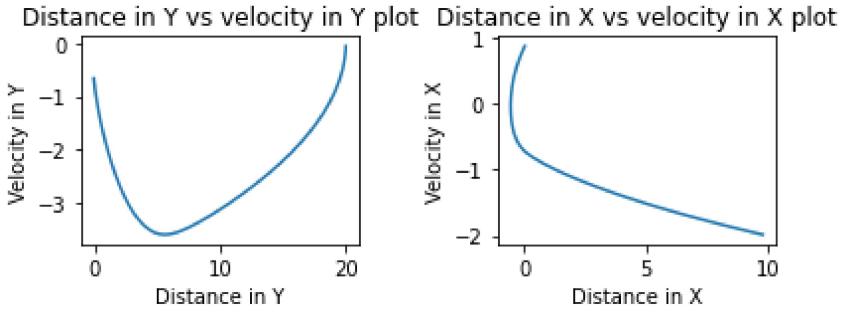
Iteration number: 25
loss: 3.179



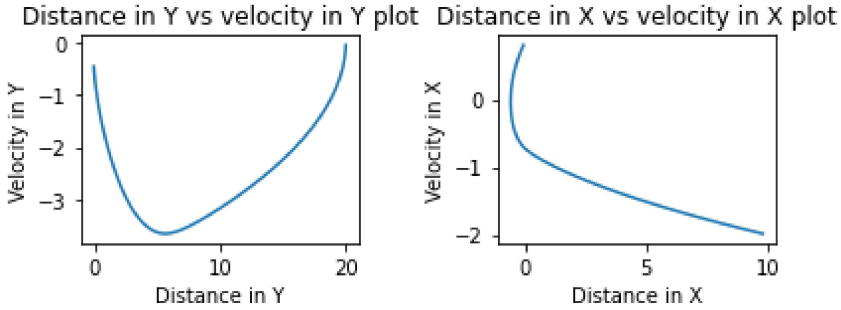
Iteration number: 26
loss: 1.892



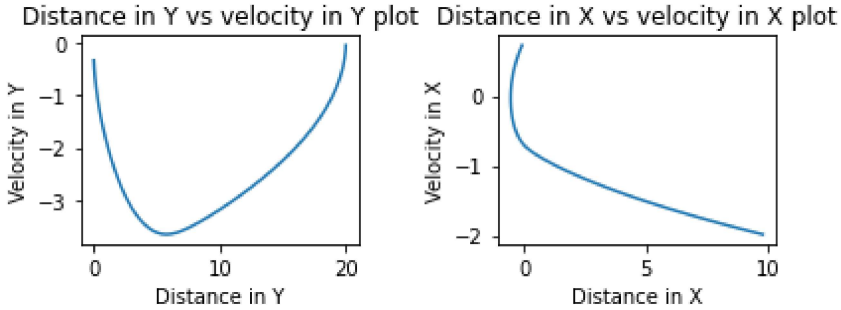
Iteration number: 27
loss: 1.218



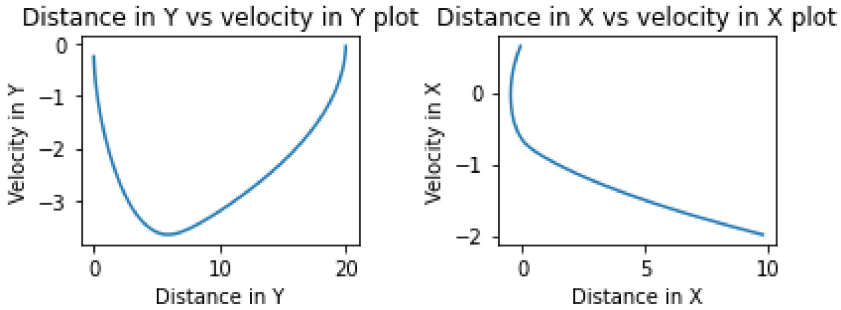
Iteration number: 28
loss: 0.865



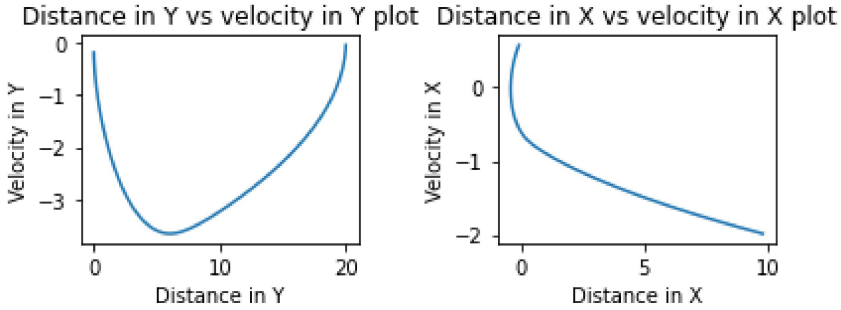
Iteration number: 29
loss: 0.650



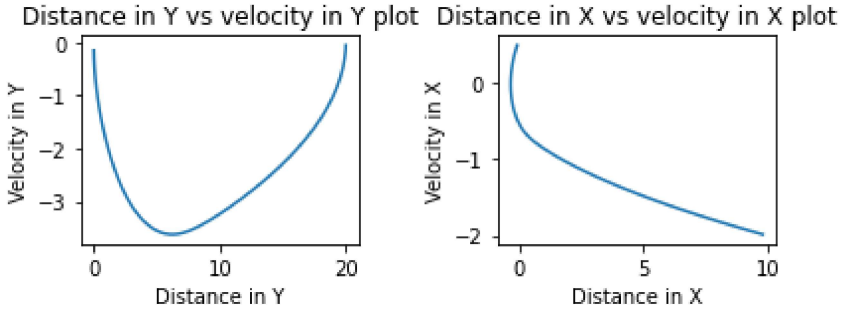
Iteration number: 30
loss: 0.494



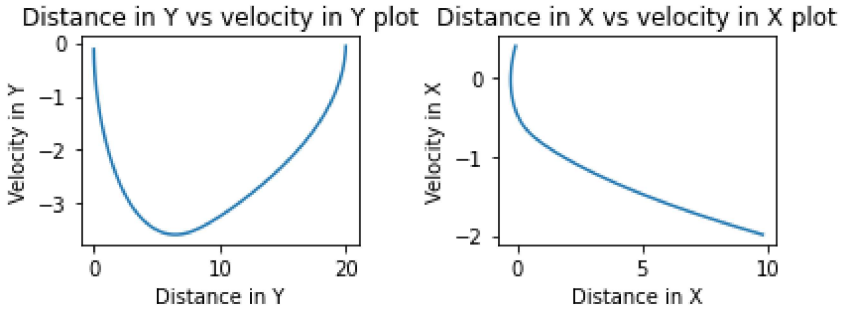
Iteration number: 31
loss: 0.366



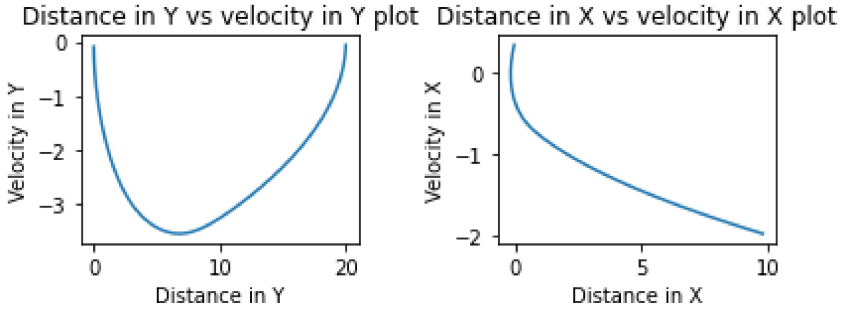
Iteration number: 32
loss: 0.263



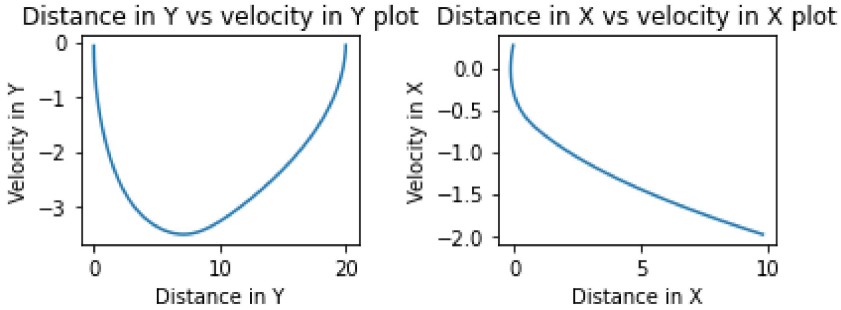
Iteration number: 33
loss: 0.182



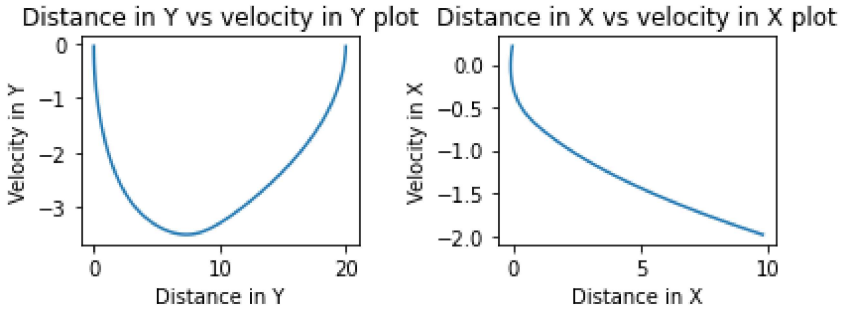
Iteration number: 34
loss: 0.121



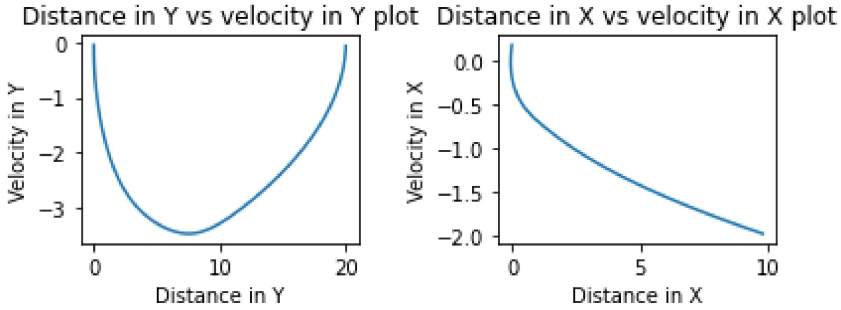
Iteration number: 35
loss: 0.079



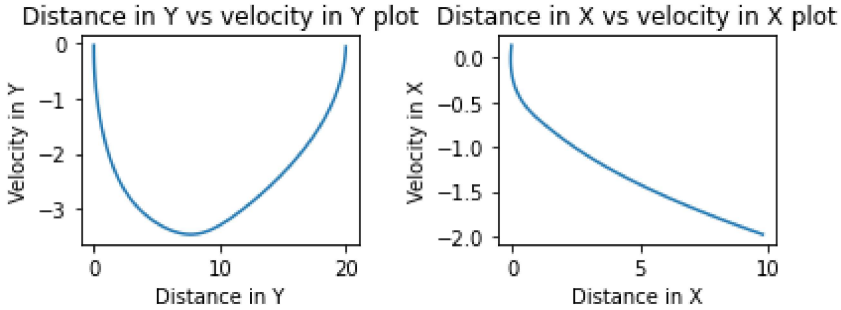
Iteration number: 36
loss: 0.051



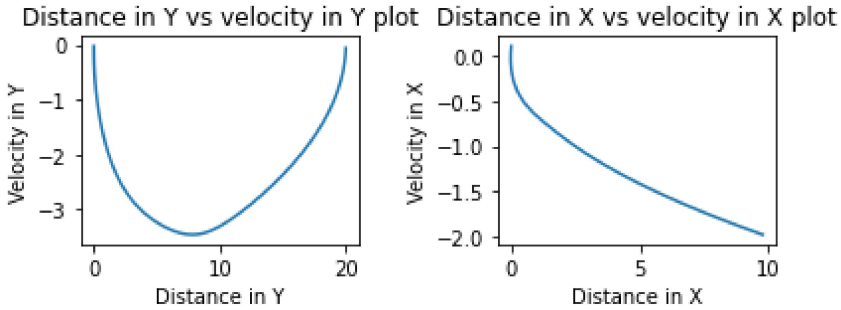
Iteration number: 37
loss: 0.033



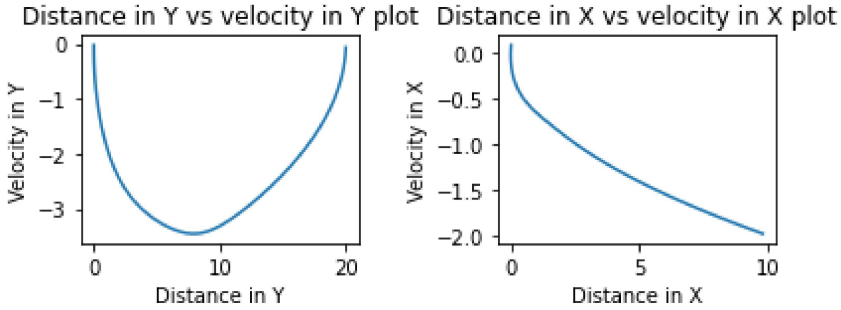
Iteration number: 38
loss: 0.021



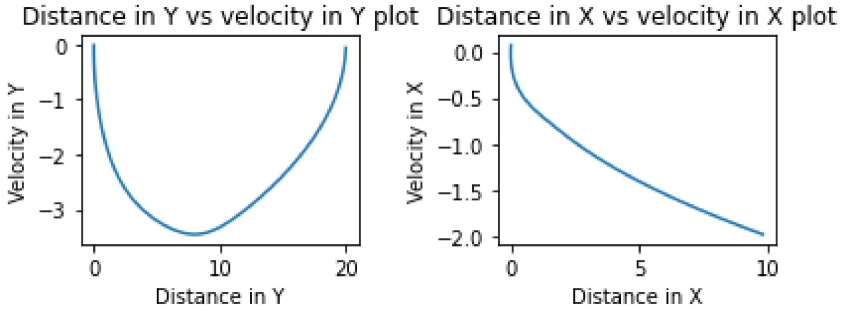
Iteration number: 39
loss: 0.014



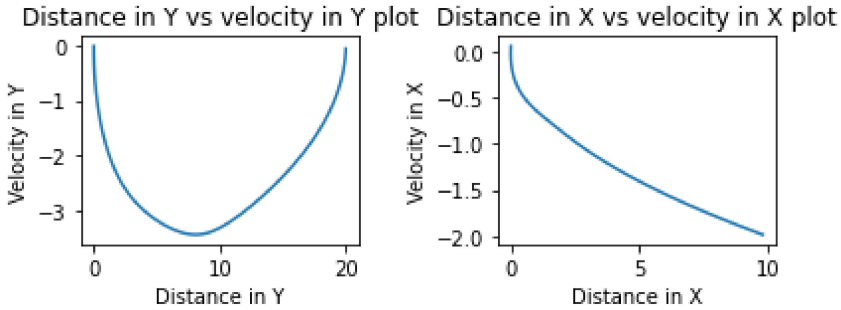
```
Iteration number:        40
loss: 0.009
```



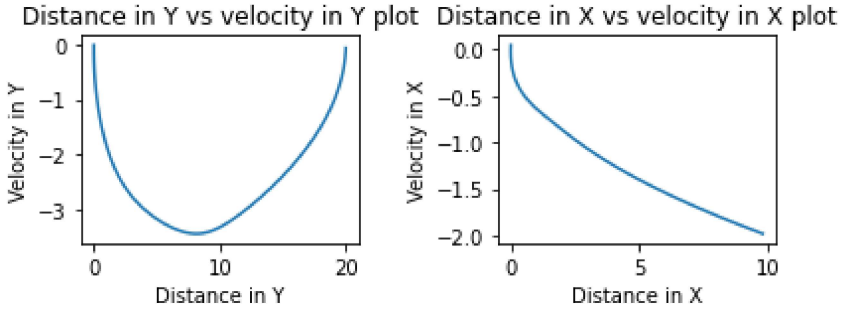
```
Iteration number:        41
loss: 0.006
```



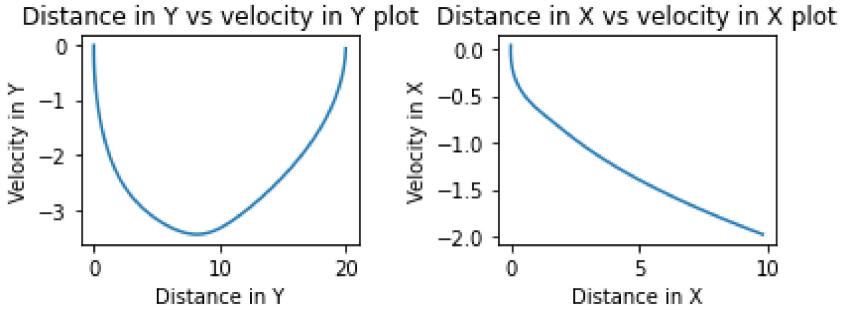
```
Iteration number:        42
loss: 0.004
```



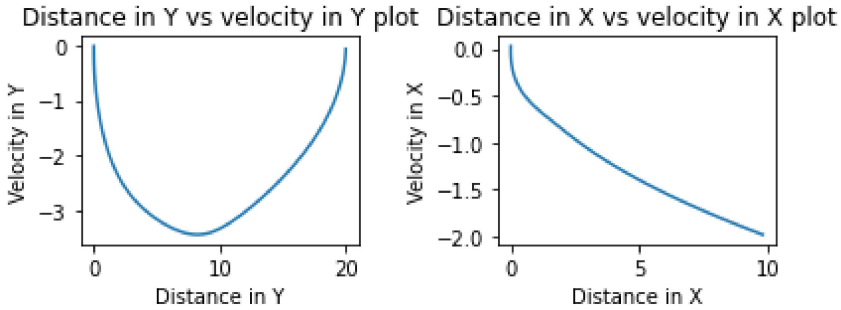
Iteration number: 43
loss: 0.002



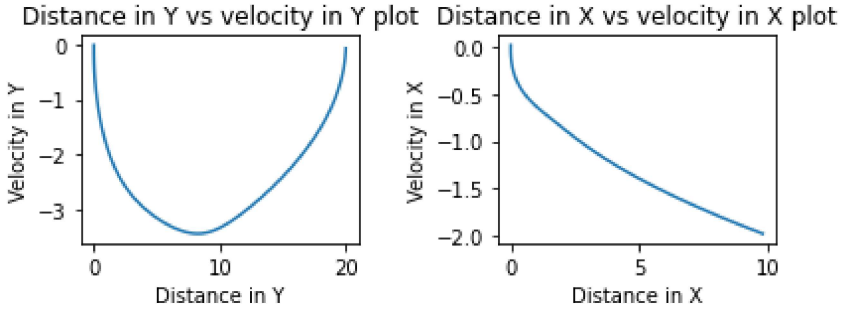
Iteration number: 44
loss: 0.002



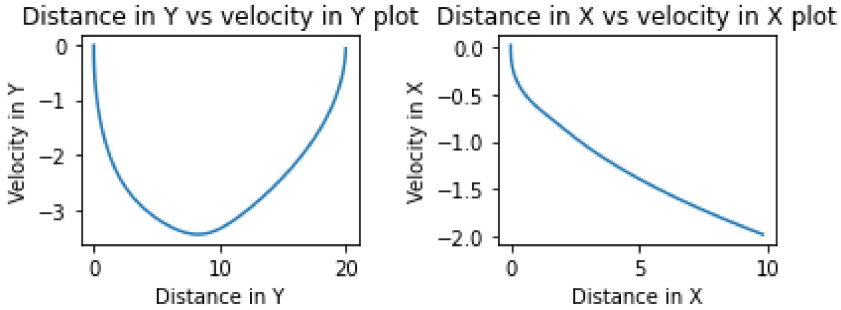
Iteration number: 45
loss: 0.001



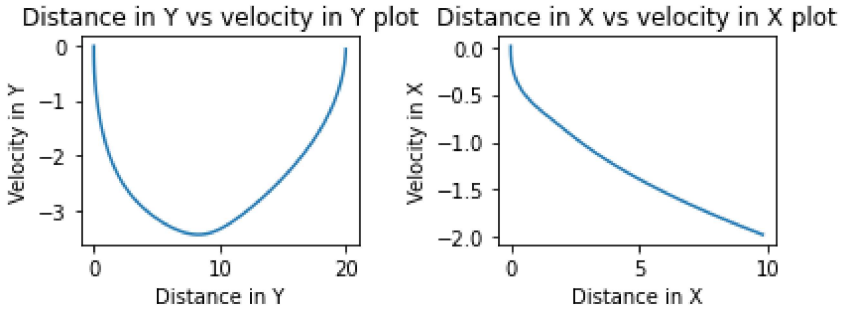
```
Iteration number:        46
loss: 0.001
```



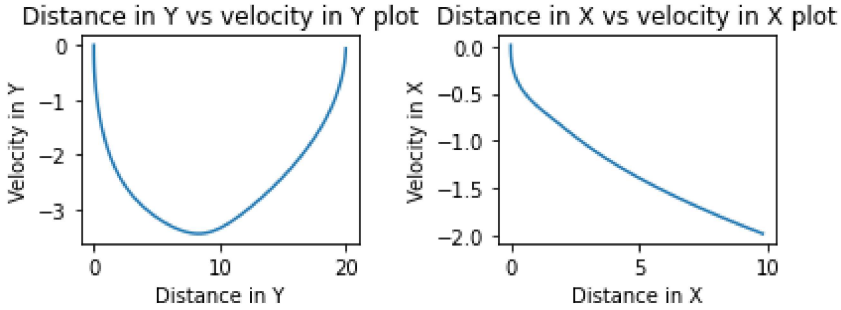
```
Iteration number:        47
loss: 0.000
```



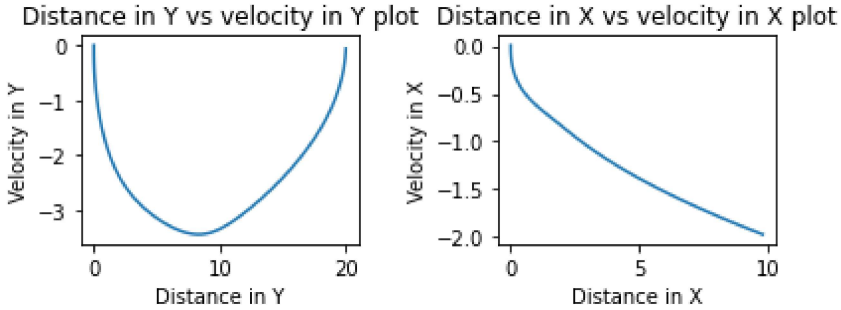
```
Iteration number:        48
loss: 0.000
```



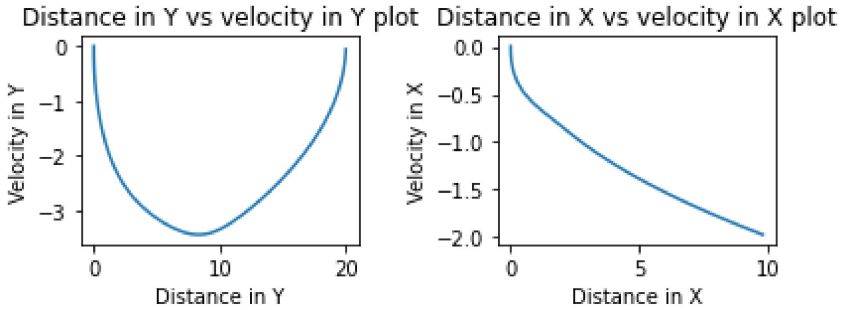
```
Iteration number:        49
loss: 0.000
```



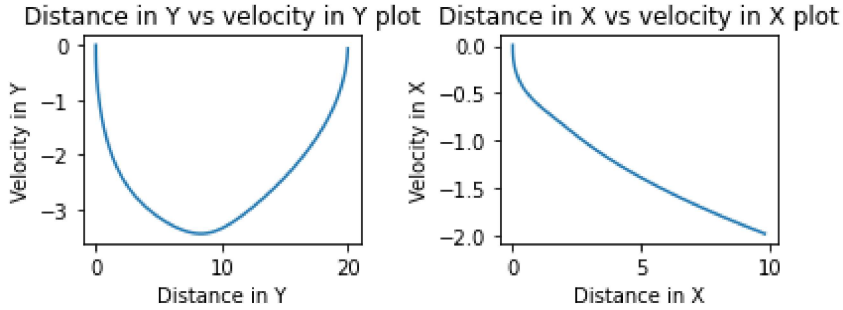
```
Iteration number:        50
loss: 0.000
```



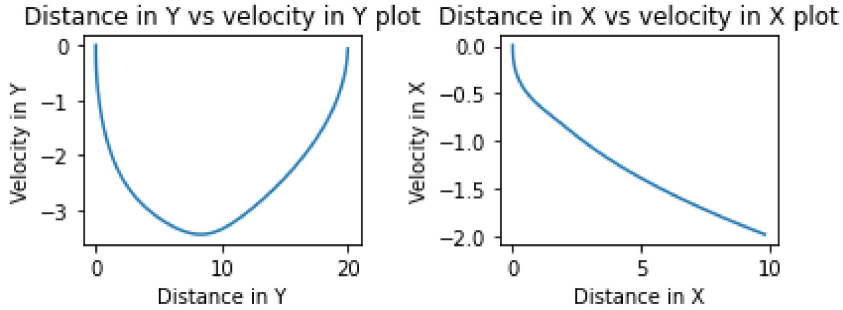
```
Iteration number:        51
loss: 0.000
```



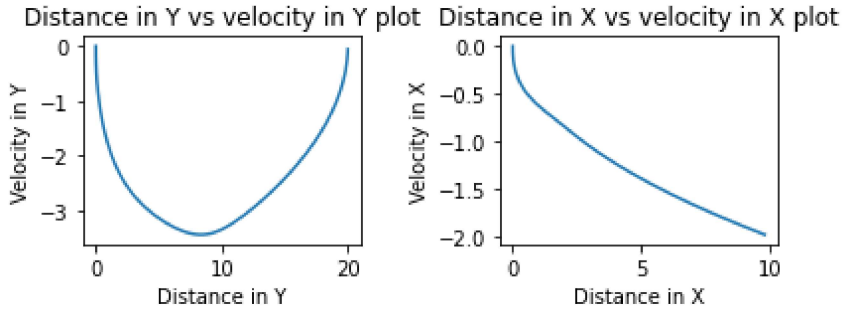
Iteration number: 52
loss: 0.000



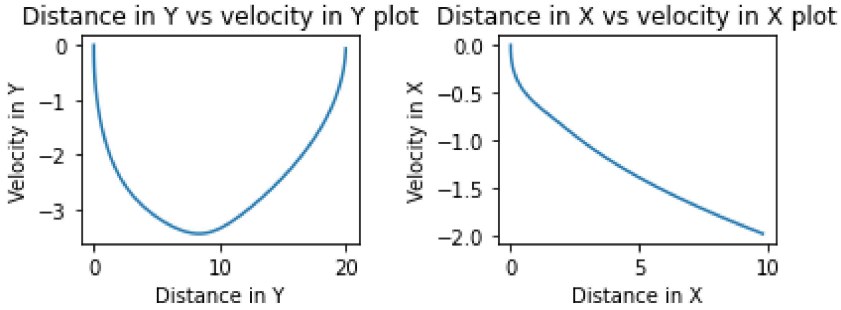
Iteration number: 53
loss: 0.000



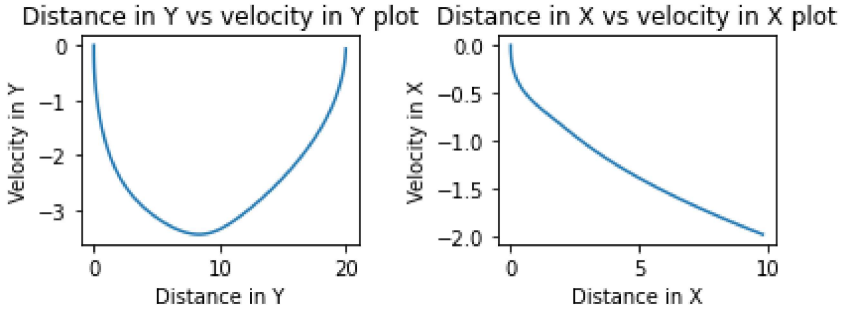
Iteration number: 54
loss: 0.000



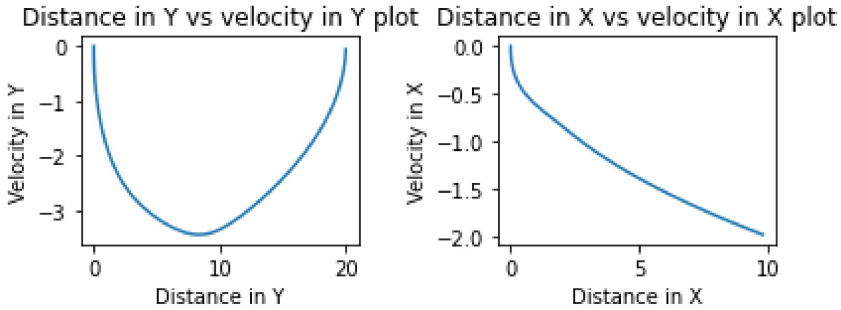
```
Iteration number:        55
loss: 0.000
```



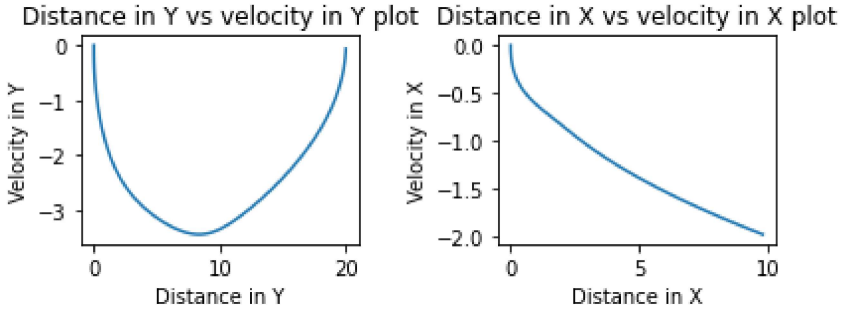
```
Iteration number:        56
loss: 0.000
```



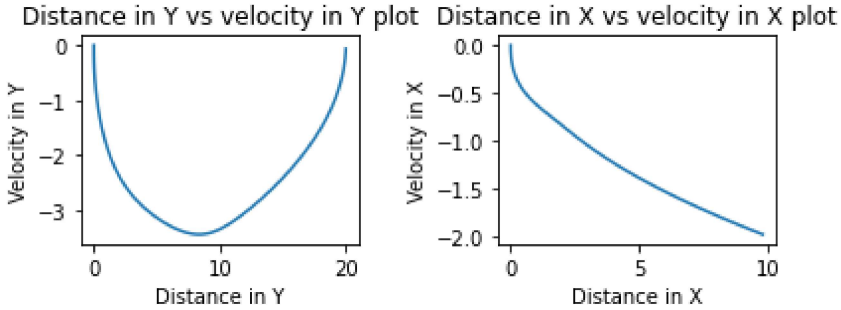
```
Iteration number:        57
loss: 0.000
```



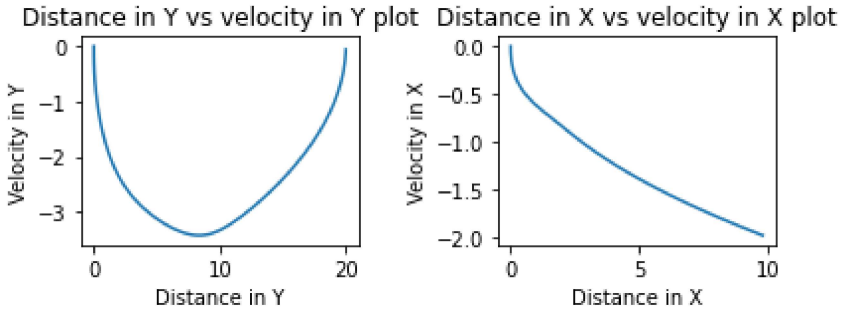
Iteration number: 58
loss: 0.000



Iteration number: 59
loss: 0.000



Iteration number: 60
loss: 0.000



3. Convergence Study of Results & Optimal Solution-

We can visualize the Distance Vs. Velocity plot for the rocket landing for both X and Y direction after every iteration along with its corresponding loss function value. As evident, the result is achieved within 48 iterations by the neural network and the entire code is run for 60 iterations. The "Velocity vs Distance" plots for both X-axis and Y-axis shows the change in velocity with distance when looked from the right to the left.

- 1st iteration- Indicates very high loss of about ~ 117812 which is also the highest recorded loss that suggest that the rocket hasn't landed as per requirement.

- 11th iteration- As seen there is a drastic change in the graph indicating that the neural network might be trying a new approach.

- 27th iteration- The loss has reduced close to ~1 marking a reduction of around 99% in loss when compared to 1st iteration.

- 45th iteration- The loss function is equal to 0.01 and the rocket is able to land at the correct position but minor value for velocity.

- 47th iteration and onwards- We have achieved loss = 0 and thus, our objective function has been properly minimized and the target landing conditions are satisfied. Thus, we can say that with the currently formulated neural network and dynamic model, we are able to achieve convergence and minimize our objective function in around 46th iterations each with a time step of 0.1 seconds.