Jesus Pena

Documentation:

ASU ID: 1212651308

$$\min_{\theta} \quad \frac{1}{N}\sum_{i=1}^{N} ||x^{(i)}(T)||^2$$

$$d(t+1)^{(i)} = d(t)^{(i)} + v(t)^{(i)} + \underline{w(t)}^{(i)}\Delta t, \quad + C_D \cdot \Delta t^2 \cdot 1/2$$

$$v(t+1)^{(i)} = v(t)^{(i)} + a(t)^{(i)} + \underline{u(t)}^{(i)}\Delta t, \quad + C_D \cdot \Delta t$$

$$a(t)^{(i)} = f_\theta(x(t)^{(i)}), \ \forall t = 1, \ldots, T-1, \ i = 1, \ldots, N$$

noise

Drag

rocket's position

The objective function is: $\frac{1}{N}\sum_{i=1}^{N} ||x^{(i)}(T)||^2$

The variable is: $\theta \equiv$

1) $w(t)$, $v(t)$ $\equiv$ noise
2) Drag
3) Thrust
4) velocity
5) gravity

Parametes

Assumptions:

1) Rocket lands vertically so we apply gravity, thrust, and drag

Constraints:

$$d(t+1)^{(i)} = d(t)^{(i)} + v(t)^{(i)} + \underline{w(t)}^{(i)}\Delta t, \quad + \text{Drag} \cdot \Delta t^2 \cdot 1/2$$

$$v(t+1)^{(i)} = v(t)^{(i)} + a(t)^{(i)} + \underline{u(t)}^{(i)}\Delta t, \quad + \text{Drag} \cdot \Delta t$$

$$a(t)^{(i)} = f_\theta(x(t)^{(i)}), \ \forall t = 1, \ldots, T-1, \ i = 1, \ldots, N$$

noise

Drag

* This model adds drag and random noise, the drag is calculated by using the same formula as thrust but in the opposite direction with a drag coefficient of either: 0, 0.005, 0.01, 0.015 It also implements random noise using np.random.normal(mean, std. deviation, shape) where

the mean = 0 and Std. deviation = 0.00015, anything above this number greatly affects the plots. Shape = 1

* Frame time = 0.1s = Δt, gravity accel = 0.12, thrust constant = 0.18

- $X(t)$ is the rocket state defined by $d(t)$ = distance to ground and $V(t)$ = velocity. $X(t) = [d(t) \quad V(t)]^T$.
- Δt = time interval
- $a(t) = f_\theta(X(t))$ where $f_\theta \equiv$ neural network with parameters θ
- $X_T^2 = d(T)^2 + V(T)^2$ where T = Final time step

Analysis:

In the code (shown at the end of the PDF), I have it run with 4 drag coefficient values: 0, 0.005, 0.01, 0.015 AND with the noise on or off. Below are 8 figures, figures 1-4 shows varying drag coeffs with noise on. As expected, the noise has an effect on the loss per iteration, either a large effect like in figure 2 or 3 or smaller "bumps" in the plot like figure 1 or 4. Compared to figures 5-8, which had no noise and a more noticeable smoothness to the loss iteration plots.

When it comes to the trajectory, the drag coeffs didn't have a large effect since figures 5-8 look very similar with the difference being how "close" the trajectory for each iteration is to each other but keeping a similar overall shape/trajectory. The larger impact is noise, as from figure 3 with a large noise jump, the overall trajectory is the most varied of figures 1-4.
The results of the state trajectory shows that while drag doesn't have a large difference to the trajectory, noise does. This makes sense as drag doesn't change direction of the rocket, it simply slows it down, drag can also be thought as resistance. Also, the implementation of drag was "same direction as thrust but with a negative sign" (seen in ✡✡ in the code below), this also shows that drag won't alter the trajectory, just slow it down. Noise is random, meaning that the trajectory will have a high probability of being different every time.

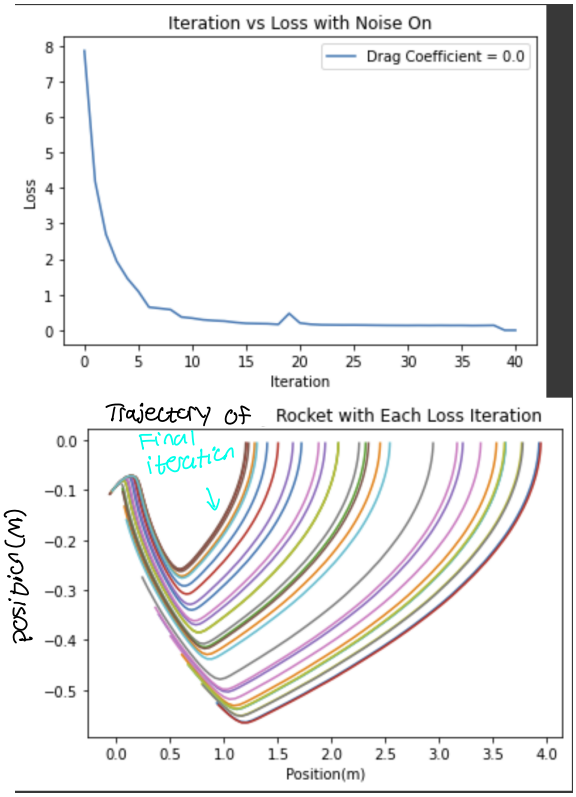


Figure 1

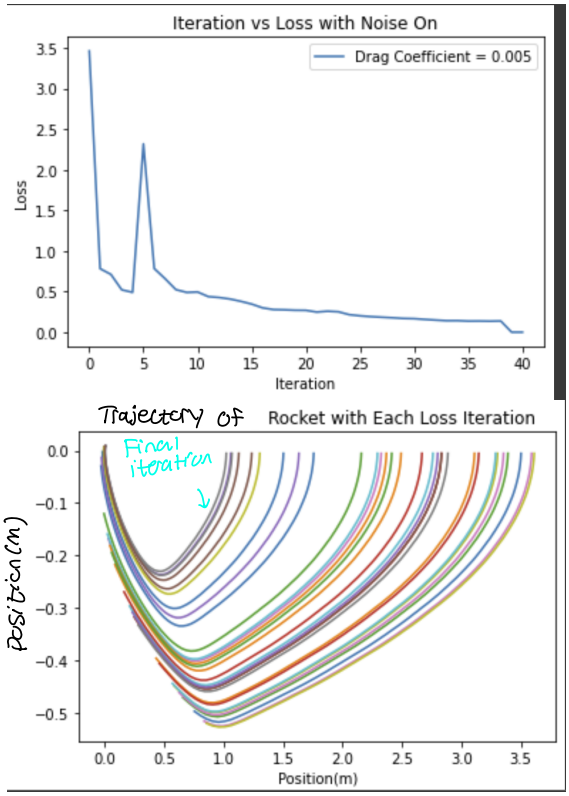


Figure 2

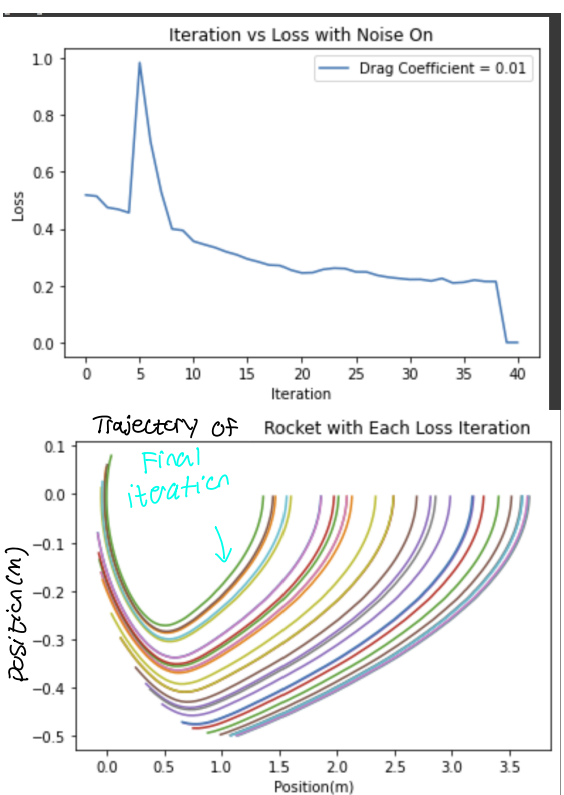


Figure 3

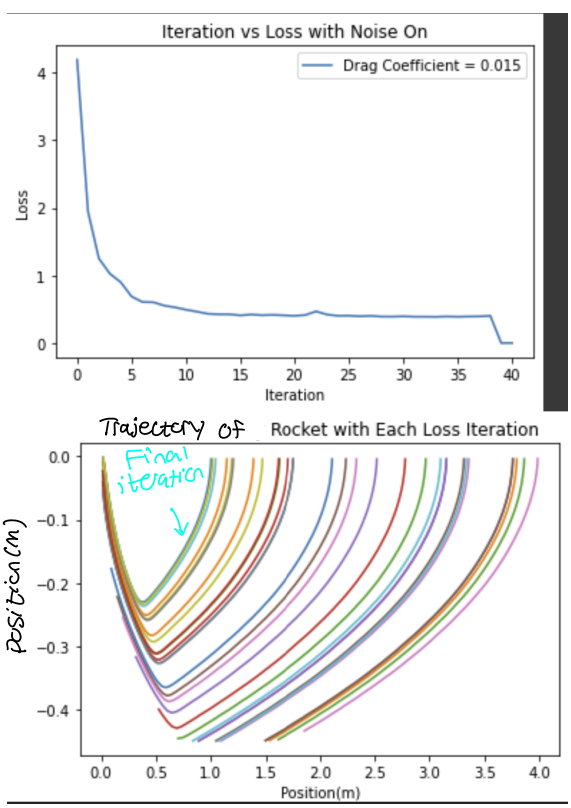


Figure 4

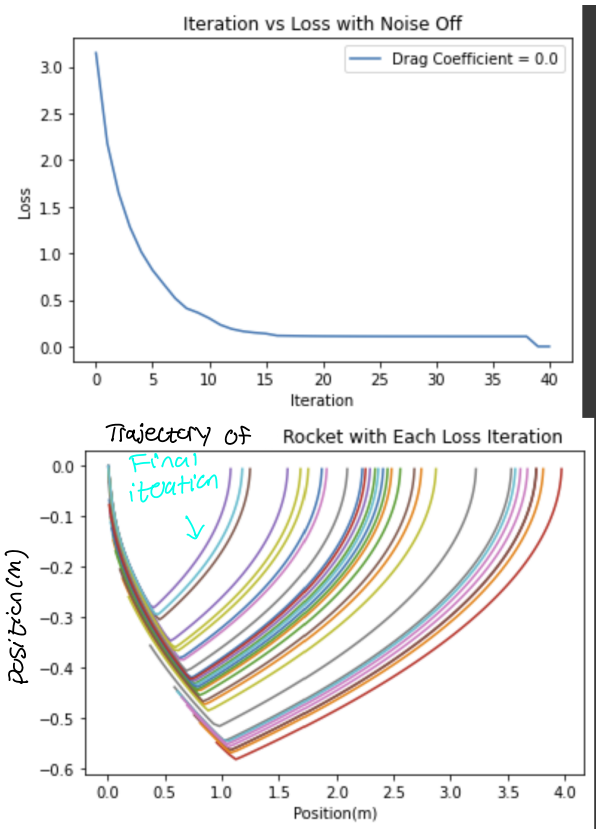


Figure 5

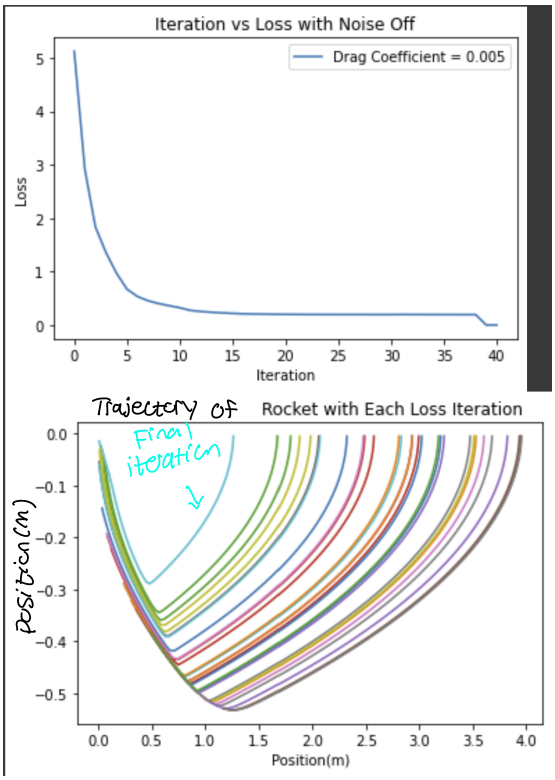


Figure 6

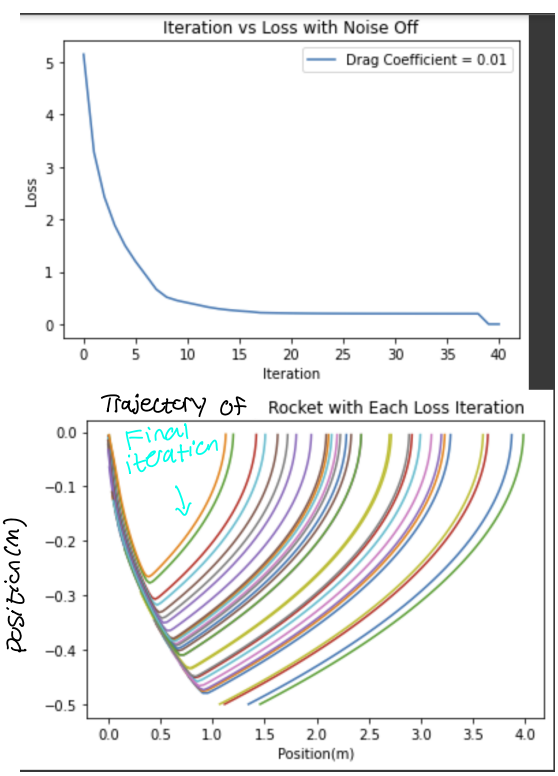


Figure 7

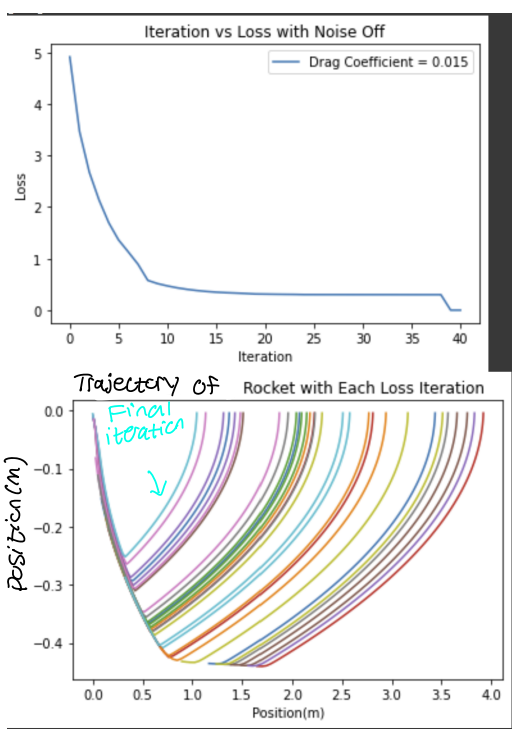


Figure 8

```
#TitleSolutionV2

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__)

# this is for the "while" loops to build the plots
ii = 1
ff = 1
last_ii = 4
D_C = np.zeros(last_ii+1)

# while loop for turning noise on and off
while ff < 3:

  if ff == 1:
      Noise = False
  else:
      ii = 1
      Noise = True

  # while loop for increasing drag coeffs
  while ii < last_ii+1:

    def my_relu(x):
        return torch.maximum(x, torch.zeros_like(x))

    ## Environment Parameters
    FRAME_TIME = 0.1  # time interval
    GRAVITY_ACCEL = 0.12  # gravity constant
    BOOST_ACCEL = 0.18  # thrust constant

    ## System Dynamics
    class Dynamics(nn.Module):

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

        def forward(self, state, action, Drag_On, Noise):
            # code whether drag is on or off
            if ii == last_ii:
              Drag_Coeff = 0
              D_C[ii] = Drag_Coeff
            else:
              Drag_Coeff = 0.005*ii
              D_C[ii] = Drag_Coeff

            """
```

```
            action: thrust or no thrust
            state[0] = y
            state[1] = y_dot
            """

            # Apply gravity
            # Note: Here gravity is used to change velocity which is the second element of the state vec
            # 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
            # Therefore, I define a tensor dx = [0., gravity * delta_time], and do x = x + dx. This is a
            delta_state_gravity = t.tensor([0., GRAVITY_ACCEL * FRAME_TIME])

            # Thrust
            # Note: Same reason as above. Need a 2-by-1 tensor.
            delta_state1 = BOOST_ACCEL * FRAME_TIME * t.tensor([0., -1.]) * action

            # Drag (calculated the same as thrust for simplification,
            # but in the opposite direction)
            # Note: Same reason as above. Need a 2-by-1 tensor.
            delta_state2 = -1 * Drag_Coeff * FRAME_TIME * t.tensor([0., -1.]) * action

            # Update velocity
            state = state + delta_state1 + delta_state2 + delta_state_gravity

            # Update state
            # Note: Same as above. Use operators on matrices/tensors as much as possible. Do not use ele
            step_mat = t.tensor([[1., 0.],
                                 [FRAME_TIME, 1.]])

            # add noise (w and u from problem formulation). To add noise, I will use
            # np.random.norm(mean,standard deivation,shape) implementation found here:
            # https://www.adamsmith.haus/python/answers/how-to-add-noise-to-a-signal-using-numpy-in-pyth
            # but will transform it into a tensor
            w = t.tensor([1, 0]) * t.tensor(np.random.normal(0, 0.00015, 1))
            u = t.tensor([0, 1]) * t.tensor(np.random.normal(0, 0.00015, 1))
            noise = w+u

            # noise will be added to the state when it is on, .float() fixed
            # an error I was having when I added just "noise" vs "noise.float()"
            if Noise == True:
                state = t.matmul(state, step_mat) + noise.float()
            else:
                state = t.matmul(state, step_mat)

            return state

## Controller
# since I am not adding anything to the controller, this class will
# stay the same
  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.Linear(dim_hidden, dim_output),
            nn.Sigmoid()
        )
    def forward(self, state):
        action = self.network(state)
        return action

## Simulation
class Simulation(nn.Module):

    # add Drag and Noise to the initialized states
    def __init__(self, controller, dynamics, T, Drag_On, Noise):
        super(Simulation, self).__init__()
        self.state = self.initialize_state()
        self.controller = controller
        self.dynamics = dynamics
        self.T = T
        self.action_trajectory = []
        self.state_trajectory = []
        self.Drag_On = Drag_On
        self.Noise = Noise

    # add Drag and Noise to the states
    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.Drag_On, self.Noise)
            self.action_trajectory.append(action)
            self.state_trajectory.append(state)
        return self.error(state)

    @staticmethod
    def initialize_state():
      # create intial states
        state = np.ones((40, 2))
        for j in range(state.shape[0]):
            state[j][0] = np.random.uniform(1, 4, 1)
            state[j][1] = 0
        return t.tensor(state, requires_grad=False).float()

    def error(self, state):
        # since there are multiple states, I will take the mean of all of
        # the errors
        return t.mean(state**2)


## Optimizer
class Optimize:

    def __init__(self, simulation):
        self.simulation = simulation
```

```
        self.parameters = simulation.controller.parameters()
        self.optimizer = optim.LBFGS(self.parameters, lr=0.01)
        self.loss_list = []

    def step(self):
        def closure():
            loss = self.simulation(self.simulation.state)
            self.optimizer.zero_grad()
            # add .sum() to fix some issues I was having with the gradients
            loss.sum().backward()
            return loss
        self.optimizer.step(closure)
        return closure()

    def train(self, epochs):
      # these lines of code store loss variables so I can plot it in one graph
        x = 1
        step = 0
        loss_y = np.zeros(41)
        for epoch in range(epochs):
            loss = self.step()
            print('[%d] loss: %.3f' % (epoch + 1, loss))
            x = x+1
            loss_y[step] = loss
            step = step + 1
            if x == 40:
                self.visualize(loss_y)

    def visualize(self,loss_y):
      # creates plots with different drag coeffs and if noise on or off
        plt.figure()
        x1 = np.arange(41)
        if ff == 1:
           plt.title('Iteration vs Loss with Noise Off')
        else:
           plt.title('Iteration vs Loss with Noise On')
        plt.plot(x1,loss_y,label="Drag Coefficient = {}".format(D_C[ii]))
        plt.ylabel('Loss')
        plt.xlabel('Iteration')
        plt.legend()
        plt.figure()

      # creates plots of position and velocity of rocket
        data = np.array([[self.simulation.state_trajectory[i][j].detach().numpy() for i in range(sel
        for i in range(30):
            x = data[i, :, 0]
            y = data[i, :, 1]
            plt.plot(x, y)

        plt.title('Position vs Velocity Rocket with Each Loss Iteration')
        plt.xlabel('Position(m)')
        plt.ylabel('Velocity(m/s)')
        plt.show()


## Now it's time to run the code!
t.autograd.set_detect_anomaly
```

this should "trajectory of"

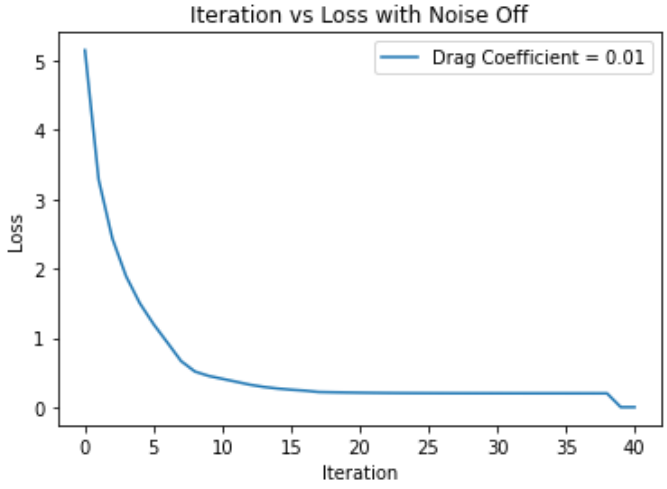
this should be "position"

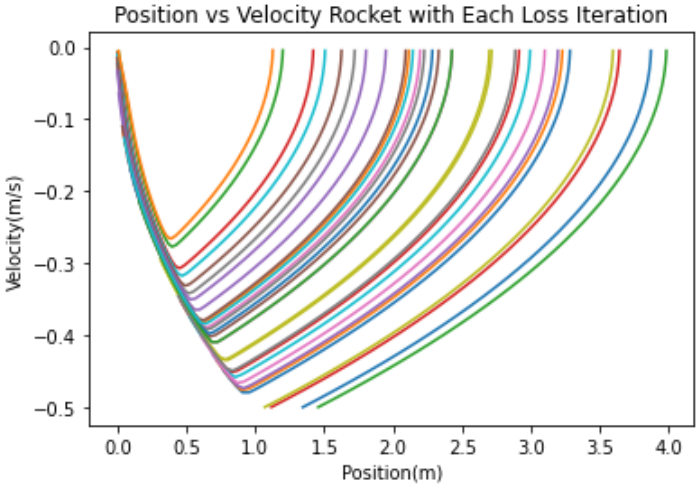
```
torch.autograd.set_detect_anomaly

T = 100  # number of time steps
dim_input = 2  # state space dimensions
dim_hidden = 6  # latent dimensions
dim_output = 1  # action space dimensions
d = Dynamics()  # define dynamics
if ii == last_ii:
  Drag_On = False
if ii != last_ii:
  Drag_On = True


c = Controller(dim_input, dim_hidden, dim_output)  # define controller
s = Simulation(c, d, T, Drag_On, Noise)  # define simulation
o = Optimize(s)  # define optimizer
o.train(40)  # solve the optimization problem
ii = ii + 1
if ii == last_ii+1:
   ff = ff + 1
```

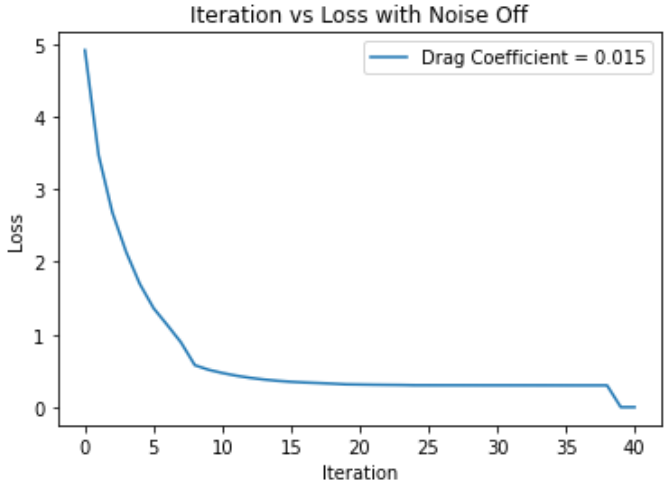
```
[12] loss: 0.367
[13] loss: 0.324
[14] loss: 0.292
[15] loss: 0.269
[16] loss: 0.252
[17] loss: 0.238
[18] loss: 0.219
[19] loss: 0.215
[20] loss: 0.211
[21] loss: 0.209
[22] loss: 0.207
[23] loss: 0.205
[24] loss: 0.203
[25] loss: 0.203
[26] loss: 0.202
[27] loss: 0.201
[28] loss: 0.201
[29] loss: 0.201
[30] loss: 0.200
[31] loss: 0.200
[32] loss: 0.200
[33] loss: 0.200
[34] loss: 0.199
[35] loss: 0.199
[36] loss: 0.199
[37] loss: 0.199
[38] loss: 0.199
[39] loss: 0.199
```

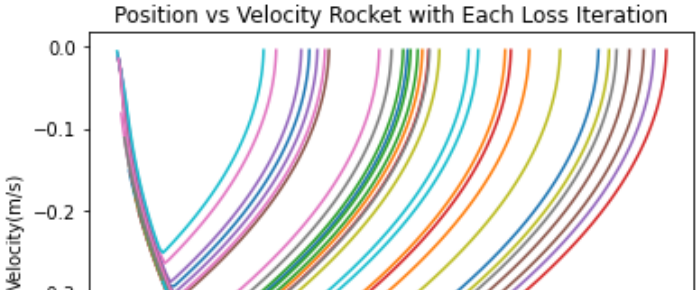


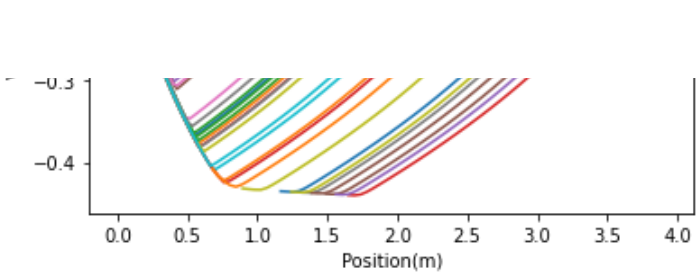



```
[40] loss: 0.199
[1] loss: 4.915
[2] loss: 3.467
```

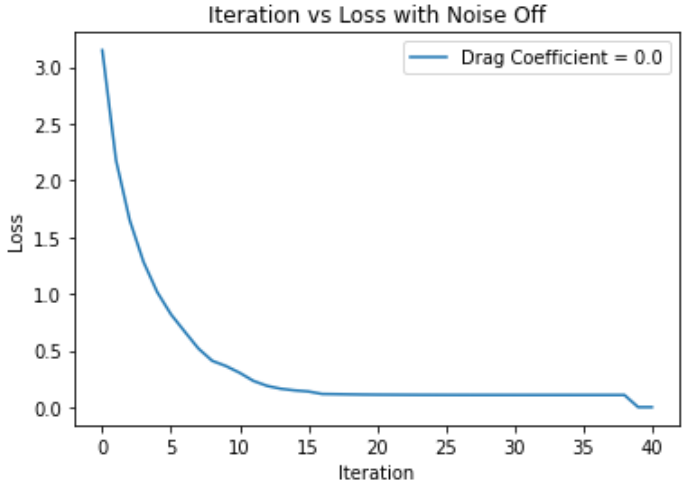
```
[3] loss: 2.679
[4] loss: 2.137
[5] loss: 1.695
[6] loss: 1.362
[7] loss: 1.130
[8] loss: 0.890
[9] loss: 0.578
[10] loss: 0.515
[11] loss: 0.471
[12] loss: 0.434
[13] loss: 0.404
[14] loss: 0.381
[15] loss: 0.363
[16] loss: 0.349
[17] loss: 0.340
[18] loss: 0.332
[19] loss: 0.324
[20] loss: 0.314
[21] loss: 0.312
[22] loss: 0.309
[23] loss: 0.307
[24] loss: 0.305
[25] loss: 0.302
[26] loss: 0.302
[27] loss: 0.302
[28] loss: 0.302
[29] loss: 0.301
[30] loss: 0.301
[31] loss: 0.301
[32] loss: 0.301
[33] loss: 0.301
[34] loss: 0.301
[35] loss: 0.301
[36] loss: 0.300
[37] loss: 0.300
[38] loss: 0.300
[39] loss: 0.300
```

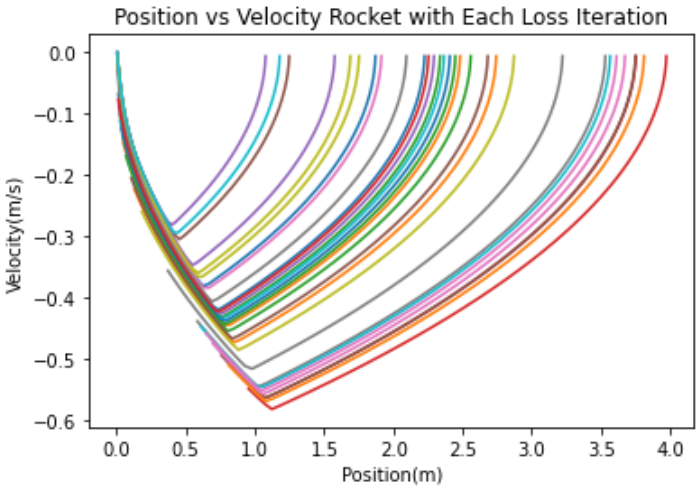




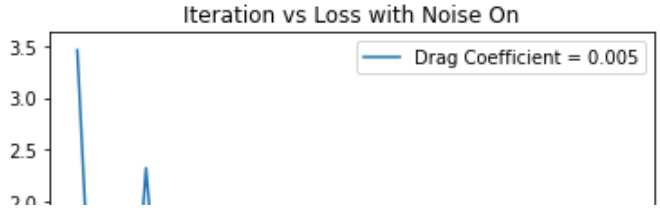


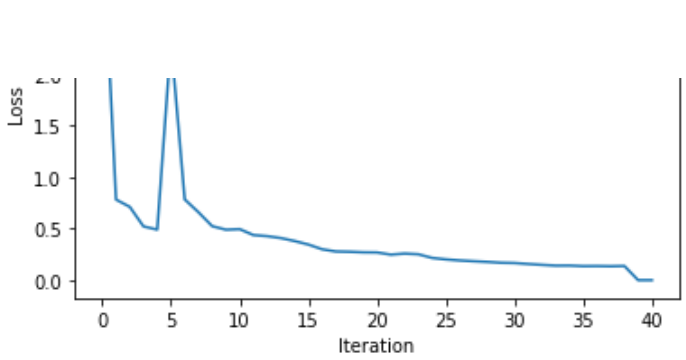

```
[40] loss: 0.300
[1] loss: 3.150
[2] loss: 2.179
[3] loss: 1.648
[4] loss: 1.282
[5] loss: 1.015
[6] loss: 0.819
[7] loss: 0.667
[8] loss: 0.517
[9] loss: 0.409
[10] loss: 0.363
[11] loss: 0.303
[12] loss: 0.232
[13] loss: 0.188
[14] loss: 0.162
[15] loss: 0.148
[16] loss: 0.139
[17] loss: 0.116
[18] loss: 0.115
[19] loss: 0.113
[20] loss: 0.111
[21] loss: 0.110
[22] loss: 0.110
[23] loss: 0.109
[24] loss: 0.109
[25] loss: 0.109
[26] loss: 0.108
[27] loss: 0.108
[28] loss: 0.108
[29] loss: 0.108
[30] loss: 0.108
[31] loss: 0.108
[32] loss: 0.108
[33] loss: 0.108
[34] loss: 0.108
[35] loss: 0.108
[36] loss: 0.108
[37] loss: 0.108
[38] loss: 0.108
[39] loss: 0.108
```

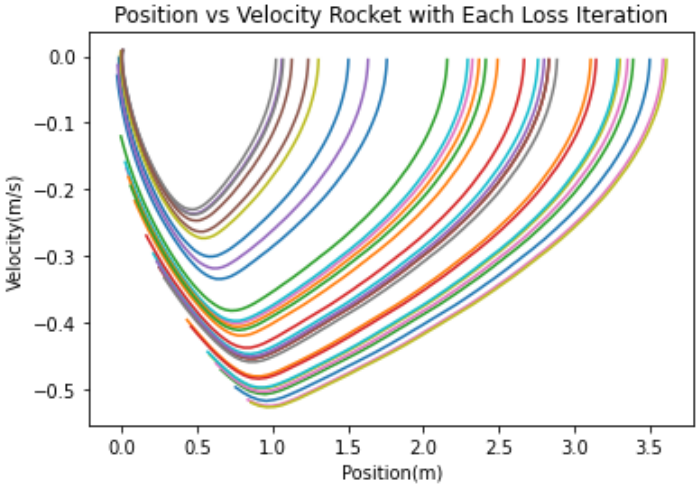





```
[40] loss: 0.108
[1] loss: 3.461
[2] loss: 0.782
[3] loss: 0.711
[4] loss: 0.521
[5] loss: 0.490
[6] loss: 2.318
[7] loss: 0.783
[8] loss: 0.658
[9] loss: 0.523
[10] loss: 0.489
[11] loss: 0.495
[12] loss: 0.437
[13] loss: 0.427
[14] loss: 0.408
[15] loss: 0.379
[16] loss: 0.345
[17] loss: 0.299
[18] loss: 0.278
[19] loss: 0.275
[20] loss: 0.270
[21] loss: 0.269
[22] loss: 0.248
[23] loss: 0.258
[24] loss: 0.251
[25] loss: 0.216
[26] loss: 0.201
[27] loss: 0.191
[28] loss: 0.184
[29] loss: 0.177
[30] loss: 0.169
[31] loss: 0.166
[32] loss: 0.157
[33] loss: 0.149
[34] loss: 0.141
[35] loss: 0.142
[36] loss: 0.136
[37] loss: 0.137
[38] loss: 0.135
[39] loss: 0.137
```

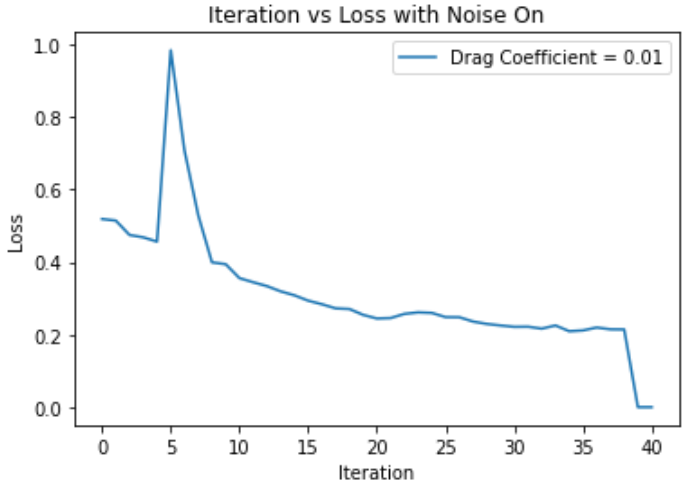


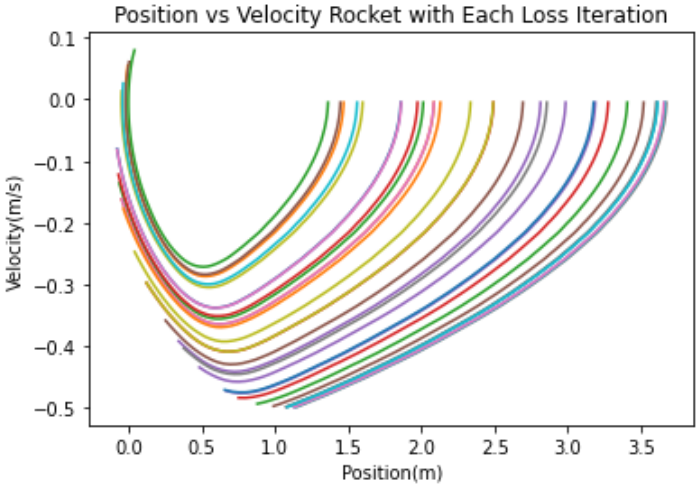





```
[40] loss: 0.137
[1] loss: 0.519
[2] loss: 0.515
[3] loss: 0.475
[4] loss: 0.468
[5] loss: 0.457
[6] loss: 0.985
[7] loss: 0.708
[8] loss: 0.529
[9] loss: 0.399
[10] loss: 0.394
[11] loss: 0.356
[12] loss: 0.345
[13] loss: 0.334
[14] loss: 0.319
[15] loss: 0.309
[16] loss: 0.294
[17] loss: 0.284
[18] loss: 0.273
[19] loss: 0.271
[20] loss: 0.255
[21] loss: 0.245
[22] loss: 0.246
[23] loss: 0.257
[24] loss: 0.261
[25] loss: 0.260
[26] loss: 0.248
[27] loss: 0.248
[28] loss: 0.236
[29] loss: 0.230
[30] loss: 0.225
[31] loss: 0.222
[32] loss: 0.222
[33] loss: 0.217
[34] loss: 0.225
[35] loss: 0.209
[36] loss: 0.212
[37] loss: 0.220
```

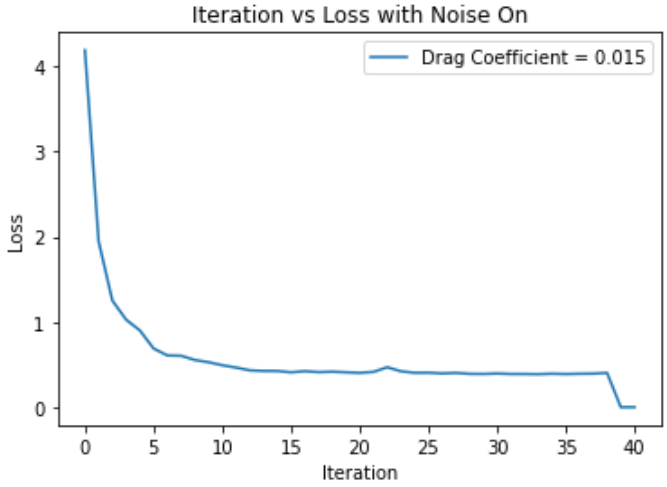
[37] loss: 0.220
[38] loss: 0.215
[39] loss: 0.214

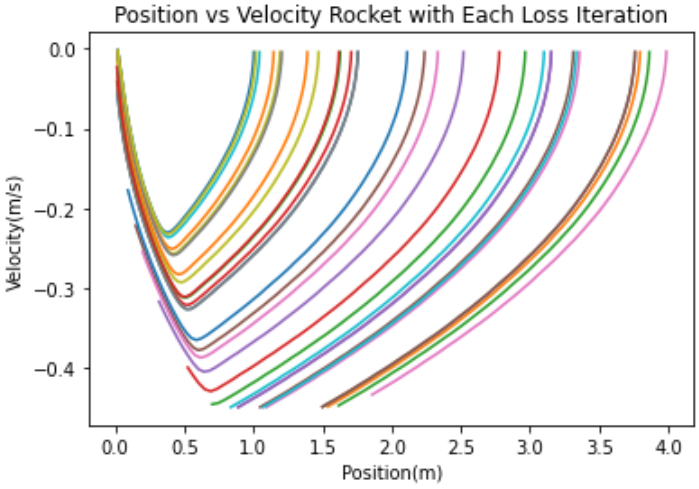




[40] loss: 0.219
[1] loss: 4.188
[2] loss: 1.945
[3] loss: 1.251
[4] loss: 1.026
[5] loss: 0.900
[6] loss: 0.688
[7] loss: 0.607
[8] loss: 0.603
[9] loss: 0.553
[10] loss: 0.528
[11] loss: 0.492
[12] loss: 0.464
[13] loss: 0.433
[14] loss: 0.425
[15] loss: 0.424
[16] loss: 0.409
[17] loss: 0.422
[18] loss: 0.412
[19] loss: 0.416
[20] loss: 0.409
[21] loss: 0.403
[22] loss: 0.414
[23] loss: 0.469
[24] loss: 0.421
[25] loss: 0.403
[26] loss: 0.405
[27] loss: 0.396

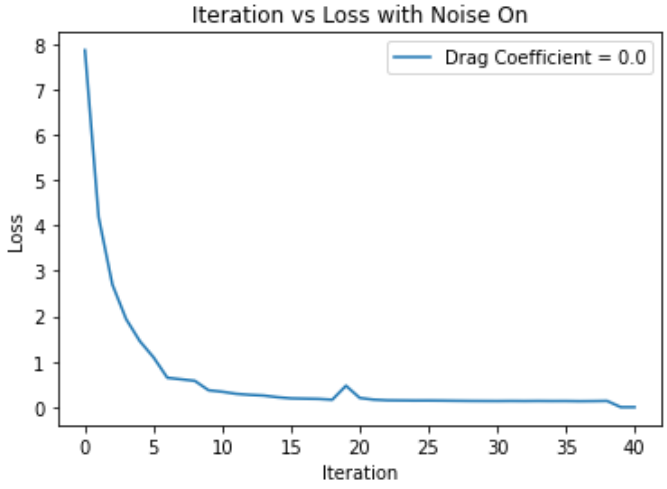
```
[28] loss: 0.402
[29] loss: 0.391
[30] loss: 0.390
[31] loss: 0.395
[32] loss: 0.389
[33] loss: 0.389
[34] loss: 0.387
[35] loss: 0.392
[36] loss: 0.389
[37] loss: 0.392
[38] loss: 0.394
[39] loss: 0.402
```

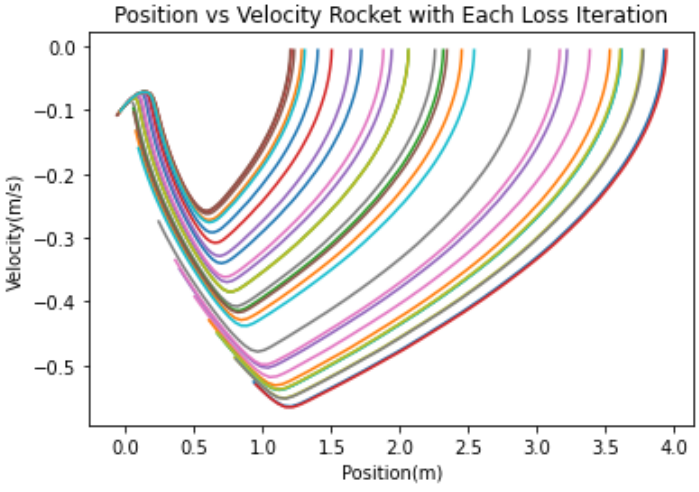





```
[40] loss: 0.400
[1] loss: 7.858
[2] loss: 4.181
[3] loss: 2.697
[4] loss: 1.939
[5] loss: 1.452
[6] loss: 1.095
[7] loss: 0.647
[8] loss: 0.614
[9] loss: 0.579
[10] loss: 0.372
[11] loss: 0.340
[12] loss: 0.294
[13] loss: 0.271
[14] loss: 0.256
[15] loss: 0.220
[16] loss: 0.194
[17] loss: 0.187
[18] loss: 0.182
```

```
[19] loss: 0.164
[20] loss: 0.472
[21] loss: 0.203
[22] loss: 0.166
[23] loss: 0.151
[24] loss: 0.148
[25] loss: 0.145
[26] loss: 0.146
[27] loss: 0.144
[28] loss: 0.140
[29] loss: 0.137
[30] loss: 0.136
[31] loss: 0.135
[32] loss: 0.137
[33] loss: 0.135
[34] loss: 0.137
[35] loss: 0.135
[36] loss: 0.135
[37] loss: 0.131
[38] loss: 0.134
[39] loss: 0.138
```






```
[40] loss: 0.133
```