Application of RL to Simulate Performance of an Unguided Hypersonic Glide Vehicle

Justine John A Serdoncillo
Aerospace Engineering and Mechanics
University of Minnesota - Twin Cities
Minneapolis, MN 55414
serdo004@umn.edu

Darryl Williams
Aerospace Engineering and Mechanics
University of Minnesota - Twin Cities
Minneapolis, MN 55414
will7322@umn.edu

Abstract

.

In this paper, we present a novel approach for simulating and controlling a high-speed aircraft using simplified physics and reinforcement learning (RL). We focus on the case of an unguided hypersonic glide vehicle (HGV), which is a challenging and relevant problem for aerospace engineering. Our goal is to design a control policy that can keep the vehicle on a straight heading, without relying on analytical expressions or numerical approximations of the dynamics. To achieve this, we use neural networks to model the continuous state space and the action space of the system, and apply RL algorithms to learn the optimal policy from data. We compare our approach with conventional methods and demonstrate its effectiveness and robustness in various scenarios. Our results show that RL can offer a viable and efficient alternative for controlling high-speed aircraft.

Keywords: Q-Learning, Neural Networks, Glide Vehicles

1 Problem Introduction

Hypersonic glide vehicles (HGVs) are capable of maneuvering within earth's atmosphere as they glide at high-speeds in excess of Mach 5. As a result of their speed, HGVs are able to travel several thousand kilometers within a short amount of time. Their ability to achieve long distances in short time frames and their use of aerodynamic forces for maneuvering have made them a subject of extensive research, particularly for defense applications. A typical trajectory for HGVs involves a boost phase that brings the vehicle high in the atmosphere, a reentry phase, and a glide phase at speeds of Mach 5-20.

In this paper we use a deep Q-learning approach with experience reply to perform guidance on an HGV during its glide phase. In the implemented framework, an aerodynamic code is integrated into the equation of motion over a non-rotating spherical Earth:

$$\frac{\partial}{\partial t}\begin{bmatrix}\gamma\\ V\\ x\\ h\end{bmatrix} = \begin{bmatrix}\frac{1}{V}\left[\frac{L}{m} + g\cos(\gamma) - \frac{V^2}{R}\cos(\gamma)\right]\\ \frac{D}{m} + g\sin(\gamma)\\ -V\cos(\gamma)\\ -V\sin(\gamma)\end{bmatrix}. \tag{1}$$

We then use an optimal policy π^*, to preform guidance on the HGV initial conditions of 60 km in altitude traveling at 6 km/s. Newtonian Aerodynamic (NA) was used as a lower-fidelity tool to compute the aerodynamic forces acting on the HGV. This approximate method is much cheaper than computational fluid dynamics enabling rapid exploration of the simulated environment but for lower levels of accuracy. Newtonian aerodynamics assumes tangential velocities are unchanged, while the normal velocity results in aerodynamic forces acting on the vehicle as shown in figure 1.

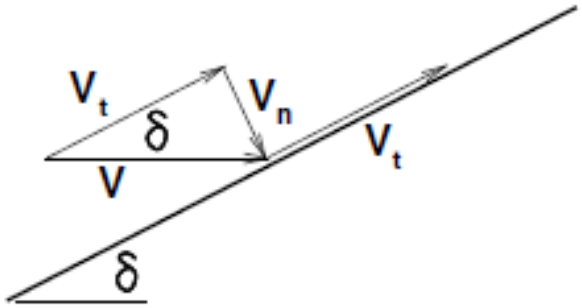


Figure 1: Illustration of tangential and normal velocities on surface.

This approximation is more accurate as speeds are increased into higher mach numbers. The algorithm to compute forces over a tessellated surface is described in the NA Procedure below.

$N =$ number of triangles elements
procedure NA(α)
  $i \leftarrow 1$
  $\hat{v} = \{\cos(\alpha), sin(\alpha), 0\}$
  Forces $=\{0, 0, 0\}$
  while $i \leq N$ **do**
    if $(\hat{v} \cdot \hat{n}_i) < 0$ **then**
      $cp_i = 2\,(\hat{v} \cdot \hat{n}_i)^2$
    else
      $cp_i = 0$ ▷ Flow is hidden from velocity vector
    end if
    $P_i = (cp_i \cdot q) + P_\infty$
    $F = F + (P_i \cdot A_i) \cdot (-\hat{n}_i)$ ▷ A_i computed from cross product
    $i \leftarrow i + 1$
  end while
  $D = F(2)\sin(\alpha) + F(1)\cos(\alpha)$
  $L = F(2)\cos(\alpha) - F(1)\sin(\alpha)$
end procedure

We then use the aerodynamic forces(L,D) to compute the derivatives in Equation 1. Time was integrated using the 4-th order Runga-Kutta method with a fixed time-step of $\Delta t = 1$ seconds as the time discretization.

2 Methodology

2.1 Simulation Environment

Our policy π(a = $\alpha|s$), when given a state, selects an angle of attack (α) such that forces on the vehicle, lift (L) and drag (D), are modulated. With these forces, we can then integrate the equation of motion for a fixed δt to arrive at out new state. We note that the simulated environment is not representative of actual flight. Rather, this model for system dynamics is sought as a proof of concept and to have tractable training times of the algorithm used. The relative disparity between system dynamics comparing a computational fluid dynamics approach to NA is shown in Figure 3.

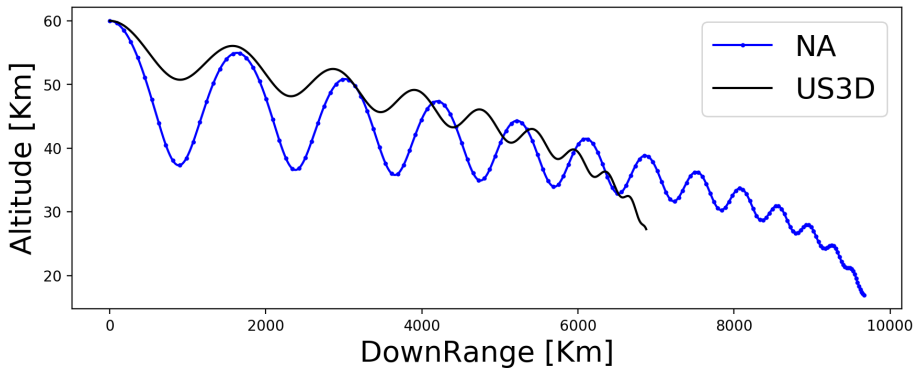


Figure 2: Comparison between full trajectory using US3D (CFD) and NA.

2.2 Markov Decision Process model

We implemented an epsilon greedy approach for exploration. Modeled as an episodic problem that terminated once the altitude went below 1 km or velocity decreased below 1 km/s. A summary of major hyperparameters and state space below:

- State Space: γ, h,V
- Action: α
- $\epsilon_{i+1} = 1 + \log(\epsilon_i/(0.6 * n))$
- Problem: Episodic with gamma = 0.9
- Rewards: $-\gamma^2$ + d

2.3 Neural Networks for continuous State Space RL

Artificial neural networks have been used for extensively for nonlinear function approximation. It characterized by applying a linear model to a feature vector then applying a non-linear mapping. This neural networks are often convoluted within each other its to have several instances of linear and non-linear mappings applied to them. This type of neural network architecture is known as a deep neural network. For the implemented method we decided on a neural network composed of two 24 neuron hidden layers, The non-linear activation decided was the use of the rectified linear unit.

In our case, we are using a deep Q-learning methodology by using an input tuple of the state, action, new state, and the reward. Our compute time was broken into the following:

- Exploration: first 25%
  - choosing random actions to explore the state transitions
  - store [state, action, state prime, reward] for each combination
- Training: second 65%
  - reduction of epsilon to reduce exploration
  - trains using stored memory
  - input is the state and action, output is the next state and reward
- Testing: last 10%
  - test neural net by running on new intial conditions and trajectories

The figure below shows the algorithm that we based our neural network on. The code for the environment and neural network as seen in the appendices below.

Algorithm 4 Deep Q-Learning with Experience Replay

Initialize replay memory D with capacity N
Initialize Q-values with randoms weights
for $t = 1 \dots T$ **do**
    With probability ϵ select a random action a_t,
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a, w_{t-1})$
    Execute action a_t
    Receive reward r_t and new state s_{t+1}
    Store transition $(\phi(s_t), a_t, r_t, \phi(s_{t+1}))$ into D
    Sample experience $(\phi(s_j), a_j, r_j, \phi(s_{j+1}))$ from D
    Set $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal state} \\ r_j + \gamma max_{a_{j+1}} Q(\phi(s_{j+1}), a_{j+1}, w_{t-1}) & \text{otherwise} \end{cases}$
    Perform gradient descent update using $[y_j - Q(\phi(s_j, a_j, w_t)]^2$ as loss
end for

Figure 3: Deep Q-Learning algorithm

Figure 4

3 Results

To tackle this project, there were lots of iterations that was needed in order to create a proper model environemtn and for it to achieve our desired objectives. At first, the initial parameters were used to simulate and run the environment but adding a negative reward at the end of the episode. At the end, our test cases showed the HGV to glide in a really flat trajectory and then descends down relatively fast. This seemed like what we initially wanted but from the data, it can be seen that it ended the episode early at around 1/10 of the original downrange using a constant angle of attack.

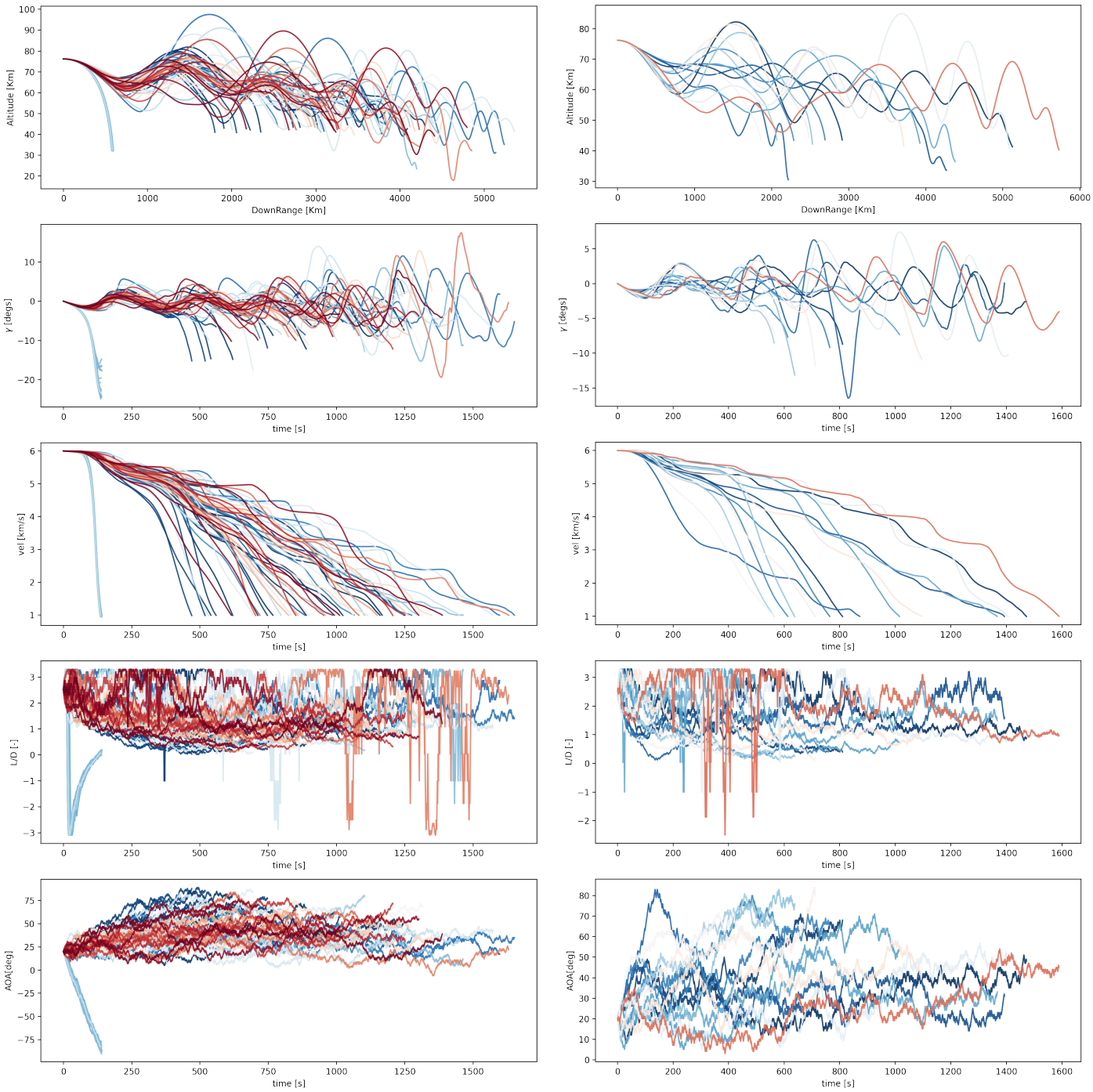
Due to the performance of the trajectory for the heading angle, thoughts on if the issue was more on the neural net was done. The episodes was increased to 1000 from 100, the gamma changed from 0.7 to 0.85 to highlight the importance of future actions so it doesn't try to crash earlier. Lastly due to the increase in episode number, the minibatch was increased to 64. After doing all of these changes, the effect on the trajectories was still the same so the focus changed not on the neural network but on the rewards structure.

With the same effect but much higher number of episodes, the number of episodes was greatly decreaed to 100 again. At the same time, the gamma was increased to 0.95. To reward going higher distances, an extra reward of the down-range was added while also penalizing if the terminating downrange was below 1000km. This had better results, and incremental improvements was desired.

In order to address this, splits in the exploration, training and testing were done. To have a bigger source of memory for training, the exploration was increased to 35% of the episodes, while training account for 65% of the episodes. At the same time, some limitations were removed which was now just the altitude because a low velocity eventually gets a low altitude soon anyway. The number of episodes was increased to 250 with a minibatch of 32 as well. There were some further explorations with our rewards structure by being more lenient with the heading angle so an absolute value was used instead.

Lastly, due to a recent collaboration and ideation, adding generality for the controller would be great so there were now different initial conditions added so the state space may be greatly increased. At the same time, extra limitations were added due to the physical representatio of the model. This should have been added earlier but was disregarded for the sake of exploration.

We now summarize the runs of the DNN Q-learning. Two of our runs are listed Figure 5. We see consistency that between the two figures, as the network progresses in training, the algorithm learns slowly to keep the heading angle near zero while maximizing range. Its evident by subfigure 5b that it has developed a policy for incentivizing max range. This is supported by the early failed regions in subfigure 5a where the trajectories did not exceed 1,000 km in down range distance.

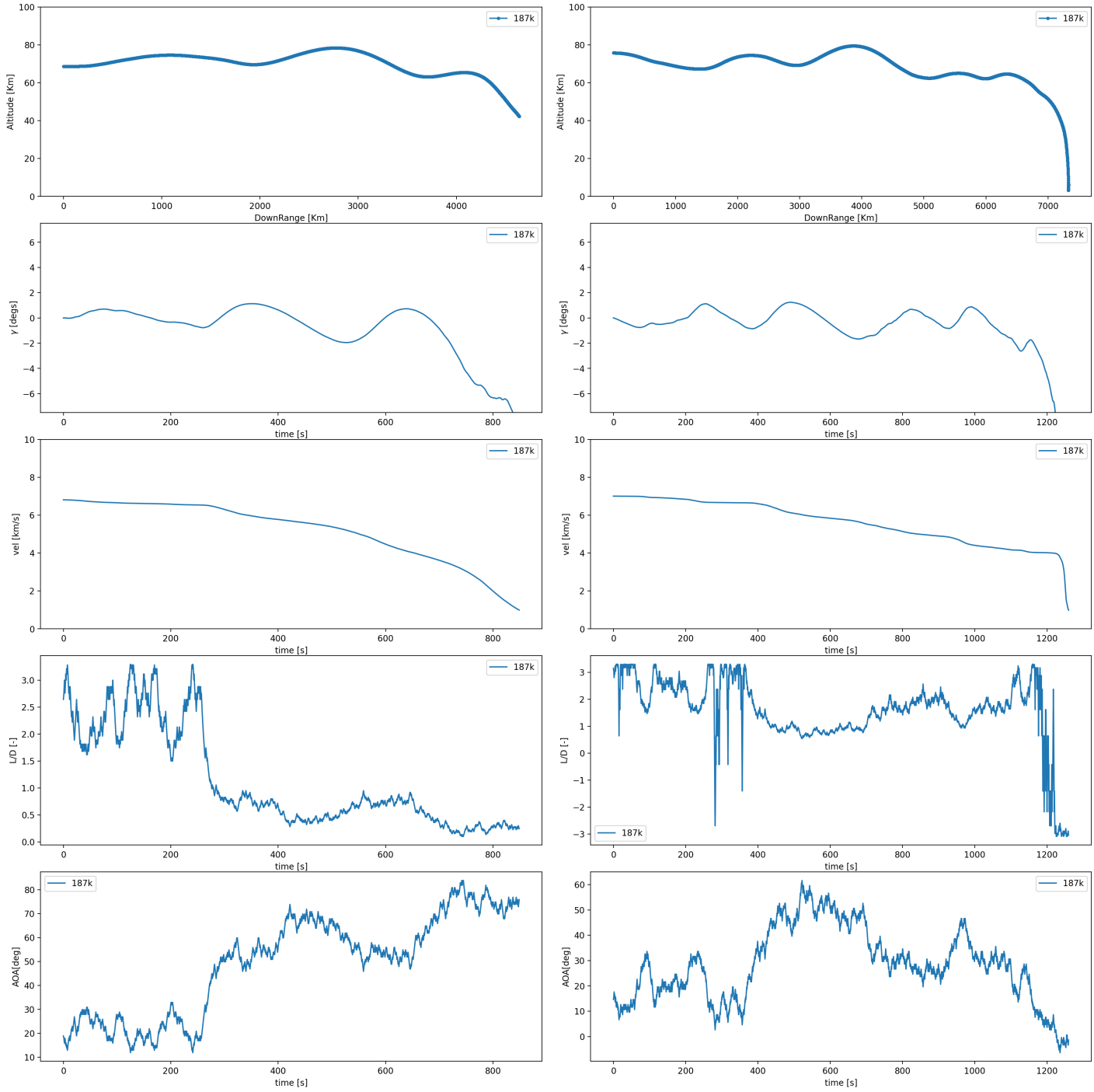


(a) Run 2

(b) Run 3

Figure 5: Output runs of DNN Q-learning. Blue signifies earlier portions of the training while red signifies later in training

When we implement the policy after updating the results of two runs are shown in Figure 6. We see that the policy performance is acceptable for the earlier portions of the trajectory. However, as expected, once velocities are lower more lift is needed to maintain level fligth. Hence, we see notable trends upward in subfigure 6a. We see in subfigure 6b just how sensitive initial chooses effect guidance for the policy. We note this by the subtle decrease in subfigure 6a as opposed to the increase in 6b leading to very different angle of attack progression but similar trajectory profiles in altitudes.



(a) Testing optimal policy run 1

(b) Testing optimal policy run 2

Figure 6: Online run of $\pi*$ after training

4 Discussion

Compared with other MDP problems, this is more complex due to the usage of the ODE and there is no analytical solution for the controls problem. However, we have showed a new approach for simulating and controlling a hypersonic glide vehicle using simplified physics and reinforment learning. Some of the key insights that we have discovered while doing this project is that firstly, Choosing a suitable neural network architecture for the policy and value functions was not trivial, as different architectures had different trade-offs in terms of complexity, performance, and generalization. We experimented with various architectures and compared their results to find the best one for our problem. Additionally,

we have experienced that uning the hyperparameters of the neural network and the reinforcement learning algorithm was also a difficult task, as they affected the convergence, stability, and robustness of the learning process. We used a trial-and-error approach and followed some best practices to find the optimal values for the hyperparameters.

We would report that our main issue was designing a proper reward function which was crucial for achieving our objective of maintaining a flat heading angle. We found that different reward functions had different impacts on the final output, and that some reward functions did not encourage the desired behavior. We also had to consider the trade-off between accuracy and computation time, as higher accuracy required more computation time and vice versa.

Some key limitations include the number of episodes that the model was trained on, even if it was then modeled to be generalizable for any initial condition. At the same time, we also had difficulty in balancing multiple goals, such as minimizing the heading angle but also not having it to terminate too early. Afterwards, due to the scope of the simplified physics it terminates early after the altitude reaches a high height. Qualitatively, this trajectory is good because there was not that much reduction in velocity while also having a straight heading angle. Unfortunately, this better controls can't be used or it won't be physically intuitive. In the end, we had to settle but something mediocre but was still able to perform better than the constant angle-of-attack.

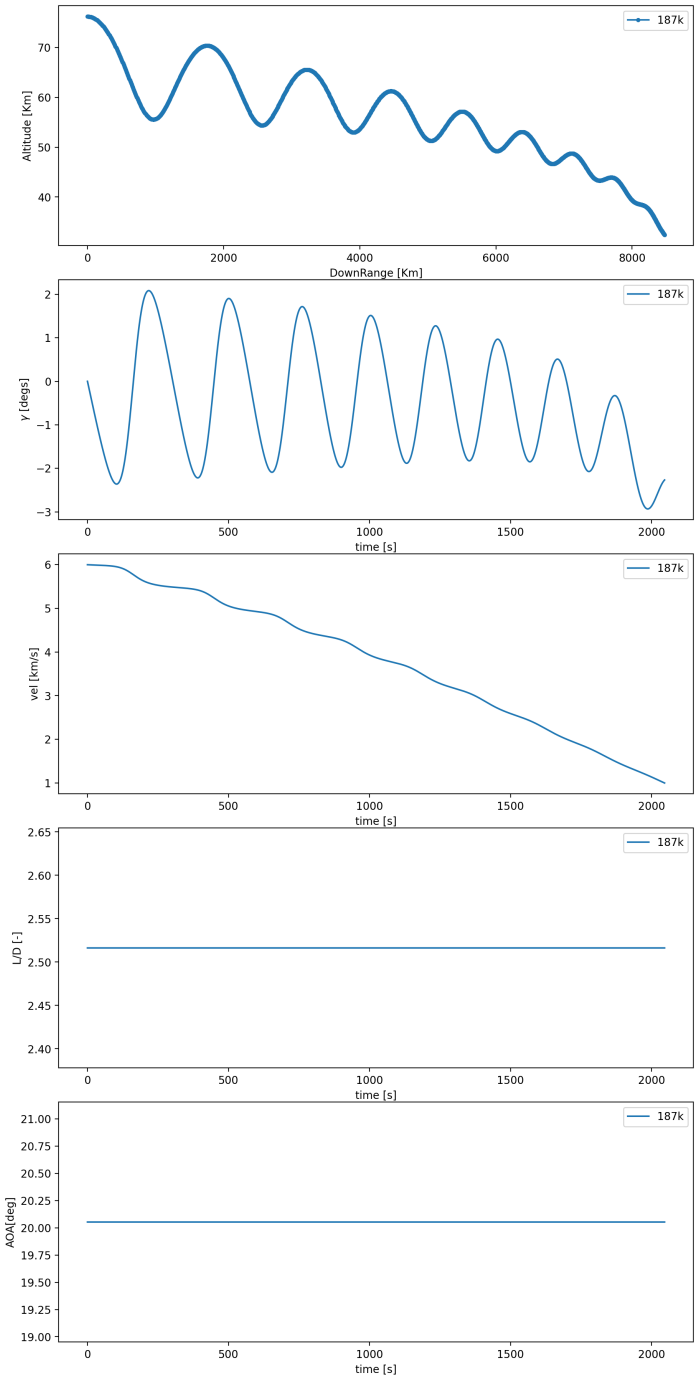
5 Conclusions

In conclusion, we have showed the feasibility of using a neural net for reinforcement learning of a hypersonic glide vehicle. This was achieved by running multiple episodes and was split into an exploration, training, and testing state. In the exploration stage, the action taken was completely random, while in the training stage, the randomness of the action choosing was slowly decreasing. With our goal of having a flat heading angle, the new trajectory of the HGV is much different than the constant angle of attack case. There were lots of issues that was faced during the simulation for each run because of the limitations on some of the state space values so it is still physically representative and accurate. Overall, in addition with training hyperparameters, this was addressed by changing the rewards, termination and neural net hyperparameters. And was able to get a proper control system using the first few timesteps of the gliding.

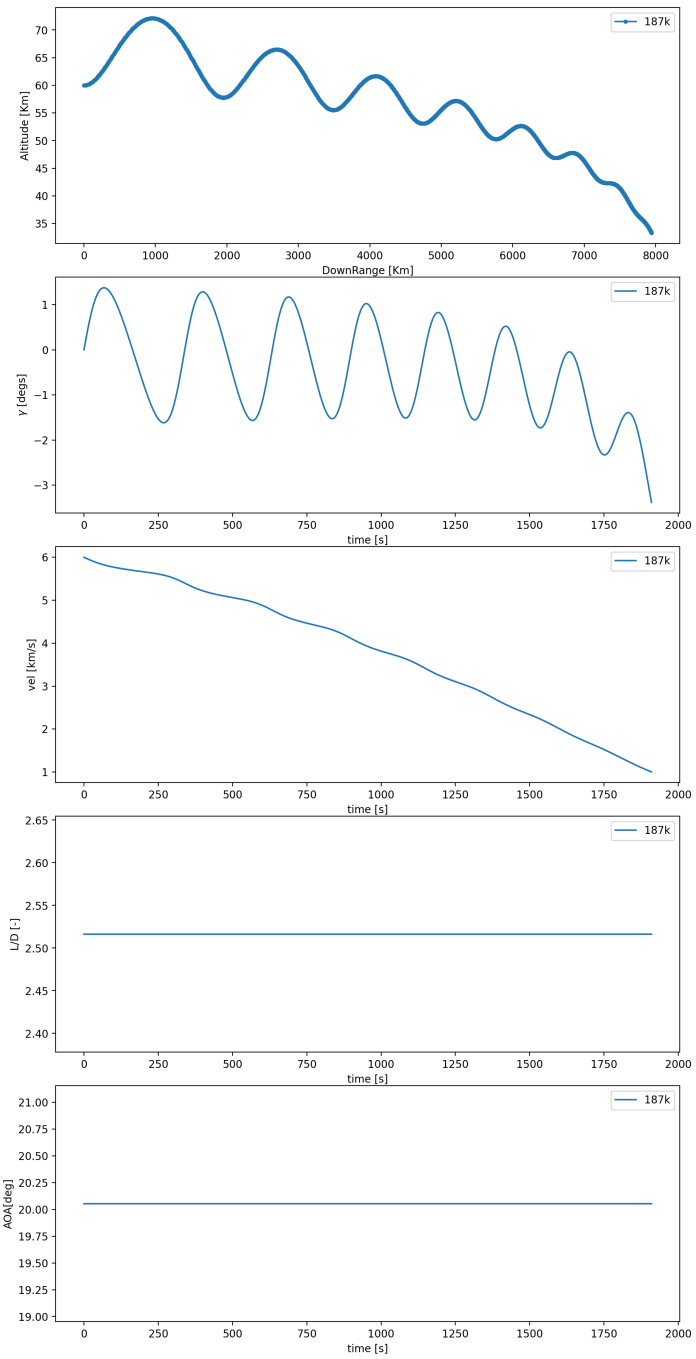
For future work, extended training time and a more complex neural network architecture may train the controller better to fit our objectives more accurately. A neural network generally gets more complex with various combinations of sequential blocks and activation functions as the model to be trained gets more complicated. In this case, with a 3 layer fully connected layer, this is definitely not enough as the number of nodes cannot fully represent a complex 4 state ordinary differential equation. Different work in the realm of machine learning has been done with the use of LSTMs or Residual Blocks which add complexity. Additionally, it can be noted that neural networks have a tendency to overfit the data so the use of dropout layers may benefit the model.

Appendix

Trajectories with constant angle-of-attack



(a) vel = 6km/s, alt=76.2 km, α=20.05



(b) vel = 6km/s, alt=60.0 km, α=20.05

Hypersonic Glide Vehicle Environment

```python
import numpy as np
from na_python_version import *
neqn = 5
import os

class plane():
    def __init__(self):
        os.chdir(r'C:\Users\GreenFluids_VR\Documents\GitHub\CUDA__Project\code\python')

    def render(self, eps=999, state=None):
        fname = "../geometry/glider_187k.stl"
        self.fresults = "../data/Run 9/trajNA187_PY_RK4_NN_" + str(eps) + ".txt"
        rho, P_inf, L_D = 0.0, 0.0, 0.0
        self.t = 0
        self.dt = 0.5
        self.t1 = np.copy(self.dt)

        y = np.zeros(5)
        yp = np.zeros(5)
        ntri=187140
        na = NA(ntri)
        read_STL(fname,na)
        self.cumreward = 0
        if state is None:
            y[0] = 0.0    # gamma (0 rads)
            y[1] = np.random.randint(5000, 8000) #6000.0  # vel (6 km/s)
            y[2] = np.random.randint(50000, 76200) #76200  # altitude (60 km)
            y[3] = 0.0    # downrange (0 km)
            y[4] = np.random.uniform(0.10, 0.35) #0.35  # alpha
        else:
            y = np.copy(state)

        self.na = na
        self.y = y
        self.yp = yp
        EOM(self.na, self.t, self.y, self.yp)
        self.L_D = na.L / na.D
        self.startwrite = False
        self.write()

        return self.y

    def step(self, action):
        del_alpha = np.pi/180 if action == 1 else -np.pi/180
        state = np.copy(self.y)
        self.y[4] += del_alpha
        done = False
        self.y = runge_kutta_4(self.na, self.t, self.y, self.dt, self.yp)
        reward = -(180/np.pi * self.y[0]) ** 2
        if self.y[1] < 1e3 or self.y[2] > 85e3 or self.y[4] < -np.pi/2 or self.y[4] > np.pi/2:
            done = True
            reward += -1e5
        if self.y[3] < 1e3 or self.y[2] > 85e3:
            reward += -1e6
        self.t = self.t1
        self.t1 += self.dt
        self.L_D = self.na.L / self.na.D
        self.cumreward += reward
        self.write()
        return state, self.y, reward, done

    def write(self):
        if self.startwrite == False:
            with open(self.fresults, 'w') as outputFile:
                outputFile.write(f"{self.t} {self.y[0]} {self.y[1]} {self.y[2]} {self.y[3]} {self
    .y[4]} {self.L_D} {self.cumreward}\n")
```

```
            self.startwrite = True
        else:
            with open(self.fresults, 'a') as outputFile:
                outputFile.write(f"{self.t} {self.y[0]} {self.y[1]} {self.y[2]} {self.y[3]} {self
    .y[4]} {self.L_D} {self.cumreward}\n")
```

Neural Network with Deep Q-Learning

```

# %% Importing
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
import os

from sim_NeuralNet import plane

EPISODES = 100
TRAIN_START = 0.5
TRAIN_DUR = 0.5
EXPLORE_EPI_END = int(TRAIN_START*EPISODES)
TEST_EPI_START = int((TRAIN_START + TRAIN_DUR)*EPISODES )
EPS_START = 1.0
EPS_END = 0.05
EPS_DECAY = 1+np.log(EPS_END)/(TRAIN_DUR*EPISODES)
GAMMA = 0.9
LR = 0.001
MINIBATCH_SIZE = 25
ITERATIONS = 40
REP_MEM_SIZE = 10000

neqn = 5

use_cuda = torch.cuda.is_available()
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor

class QNet(nn.Module):
    def __init__(self, state_space_dim, action_space_dim):
        nn.Module.__init__(self)
        self.l1 = nn.Linear(state_space_dim, 24)
        self.l2 = nn.Linear(24, 24)
        self.l3 = nn.Linear(24, action_space_dim)
    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x
class ReplayMemory: ## (D)
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
    def push(self, s, a , r, ns):
        self.memory.append((FloatTensor([s]),
        a,
        FloatTensor([ns]),
        FloatTensor([r])))
        if len(self.memory) > self.capacity:
            del self.memory[0]
    def sample(self, MINIBATCH_SIZE):
        return random.sample(self.memory, MINIBATCH_SIZE)
    def __len__(self):
        return len(self.memory)

class QNetAgent:
    def __init__(self, stateDim, actionDim):
        self.sDim = stateDim
        self.aDim = actionDim
        self.model = QNet(self.sDim, self.aDim)
        if use_cuda:
```

```
            self.model.cuda()
        self.optimizer = optim.Adam(self.model.parameters(), LR)
        self.lossCriterion = torch.nn.MSELoss()
        self.memory = ReplayMemory(REP_MEM_SIZE)

        self.steps_done = 0
        self.episode_durations = []
        self.avg_episode_duration = []
        self.epsilon = EPS_START
        self.epsilon_history = []
        self.mode = ""
        self.eps_done = 0

    def select_action(self, state):
        self.steps_done += 1
        if random.random() > self.epsilon:
            with torch.no_grad():
                return self.model(FloatTensor(state)).data.max(1)[1].view(1, 1)
        else:
            return LongTensor([[random.randrange(2)]])


    def run_episode(self, e, environment):
        state = environment.render(self.eps_done)
        self.eps_done += 1
        done = False
        steps = 0

        if e < EXPLORE_EPI_END:
            self.epsilon = EPS_START
            self.mode = "Exploring"
        elif EXPLORE_EPI_END <= e <= TEST_EPI_START:
            self.epsilon = self.epsilon*EPS_DECAY
            self.mode = "Training"
        elif e > TEST_EPI_START:
            self.epsilon = 0.025
            self.mode = "Testing"
        self.epsilon_history.append(self.epsilon)
        while True:
            action = self.select_action(FloatTensor([state]))
            c_action = action.data.cpu().numpy()[0,0]
            state, next_state, reward, done = environment.step(c_action)
            self.memory.push(state, action, reward, next_state)
            if EXPLORE_EPI_END <= e <= TEST_EPI_START:
                self.learn()
            state = next_state
            steps += 1
            if done:
                print("{2} Mode: {4} | Episode {0} Duration {1} steps | epsilon {3}"
                .format(e, steps, "\033[92m" if steps >= 195 else "\033[99m", self.epsilon, self.
    mode))
                self.episode_durations.append(steps)
                break
        return environment.cumreward

    def finaltest(self, eps, state):
        state = environment.render(eps=eps, state=state)
        self.epsilon = 0.025
        while True:
            action = self.select_action(FloatTensor([state]))
            c_action = action.data.cpu().numpy()[0,0]
            state, next_state, reward, done = environment.step(c_action)
            state = next_state
            steps += 1
            if done:
                print("| Duration {0} steps | epsilon {1}"
                .format(steps, self.epsilon))
```

```
                break


    def learn(self):
        if len(self.memory) < MINIBATCH_SIZE:
            return
        for i in range(ITERATIONS):

            experiences = self.memory.sample(MINIBATCH_SIZE)
            batch_state, batch_action, batch_next_state, batch_reward = zip(*experiences)

            batch_state = torch.cat(batch_state)
            batch_action = torch.cat(batch_action)
            batch_reward = torch.cat(batch_reward)
            batch_next_state = torch.cat(batch_next_state)

            current_q_values = self.model(batch_state).gather(1, batch_action)

            max_next_q_values = self.model(batch_next_state).detach().max(1)[0]
            expected_q_values = batch_reward + (GAMMA * max_next_q_values)

            loss = self.lossCriterion(current_q_values, expected_q_values.unsqueeze(1))

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

if __name__ == "__main__":
    environment = plane()
    state_size = 5
    action_size = 2
    rewards = np.zeros(EPISODES)

    agent = QNetAgent(state_size, action_size)
    for e in range(EPISODES):
        reward = agent.run_episode(e, environment)
        rewards[e] = reward
    print('Complete')
    test_epi_duration = agent.episode_durations[TEST_EPI_START-EPISODES+1:]
    print("Average Test Episode Duration",np.mean(test_epi_duration))

    os.chdir(r'C:\Users\GreenFluids_VR\Documents\GitHub\CUDA__Project\code\data')
    torch.save(agent.model.state_dict(), "weights1.h5")
    with open('rewards.txt', 'w') as outputFile:
        for e in range(EPISODES):
            outputFile.write(f"{rewards[e]}\n")
    agent.finaltest(state=[0,6e3, 60e3, 0, 0.35])
    agent.finaltest(state=[0,6e3, 75e3, 0, 0.35])
```