Aircraft Path Planning with Power Beaming

Nicholas Orndorff

MAE242 Course Project
12/02/2022

1 Abstract

Power beaming, that is, sending energy to an aircraft with a laser or other electromagnetic beam, is transformative because it decouples an aircraft from its energy source. Aircraft designed to exploit this technology are limited by the base station locations which provide these beams since the available power typically varies inversely with the distance from the power source. Because of this, there is a strong coupling between the mission and the aircraft design. Optimal mission trajectories might be intuitive for single power source. However, missions containing multiple power sources yield increasingly complex trajectories. This project creates a framework for finding optimal trajectories through a state space containing multiple power sources. Aircraft dynamics are modeled with a Markov decision process, and two methods of trajectory optimization are applied and compared over the same state space: policy iteration and SARSA.

2 Problem formulation

Since policy iteration is a model-based algorithm, the aircraft and its environment must be described as a typical Markov decision process with a state space $\mathcal{S}$, an action space $\mathcal{A}$, probabilities P, and discount factor γ ($\langle \mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma \rangle$). The SARSA algorithm is model free, but still requires a method of modeling aircraft dynamics so the trajectories can be simulated. The aircraft dynamics utilize a simplified model where altitude and velocity v are held constant:

$$\dot{x} = v \sin \psi \tag{1}$$

$$\dot{y} = v \cos \psi \tag{2}$$

$$\dot{\psi} = \theta. \tag{3}$$

This yields a three-dimensional state space $s = (x, y, \psi) \in \mathcal{S}$ where x and y are the aircraft position in an Earth reference frame and ψ is the current heading of the aircraft. In discrete time, subsequent states are calculated as a function of the time step (dt):

$$x_{t+1} = x_t + v \sin \psi dt \tag{4}$$

$$y_{t+1} = y_t + v \cos \psi dt \tag{5}$$

$$\psi_{t+1} = \psi_t + \theta. \tag{6}$$

The action set $\mathcal{A}$ is comprised of individual actions a at each state such that $a \in \mathcal{A}$. For the given flight dynamics model, the actions are comprised of a fixed set of angles θ. These angles are discretized from $-\pi/6$ to $\pi/6$ radians in six increments, and the probability $p(a)$ of a given heading is described by a normal distribution (Fig. 1) as a function of the standard deviation $\sigma = 1$:

$$p(a) = \frac{1}{\sigma\sqrt{2\pi}} e^{-0.5(a/\sigma)^2} \ \forall a \in \mathcal{A} \tag{7}$$

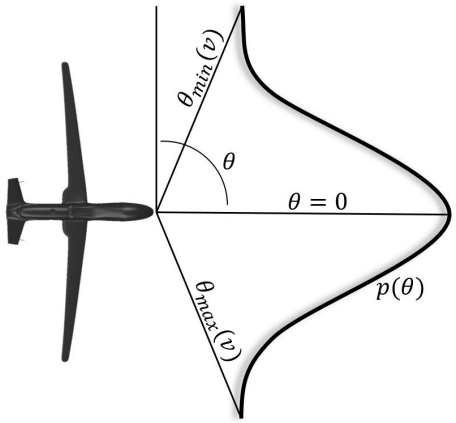

Figure 1: Action probability diagram.

This probability distribution replicates the physics of a true aircraft by assigning the greatest action probability to an angle of zero. This mimics the momentum properties of an actual aircraft with mass and velocity.

The state space $\mathcal{S}$ of the aforementioned model is comprised of a Cartesian grid $s = (x, y, \psi) \in \mathcal{S}$. Since the dynamics are time dependent, the aircraft can jump across states depending on the velocity, heading angle, and time step. This means that actions are not restricted to mapping current states to neighboring states.[AGS13]. Because the state-space is discrete, there is no guarantee that the new calculated state will perfectly match the discretization. Because of this, the new states $(x_{t+1}, y_{t+1}, \psi_{t+1})$ are calculated from equations 4-6, and rounded to the nearest existing state in $\mathcal{S}$.

The environment consists of two power stations at specific (x, y) coordinates, but the algorithms and code are written to function with any (n) power stations. The Euclidean distance between the aircraft and each station is calculated as $d = \sqrt{(x - x_s)^2 + (y - y_s)^2}$ for a power station located at (x_s, y_s). It is assumed that the power station is at roughly the same altitude as the aircraft. The reward is modeled by an inverse exponential relationship $r = k/d^2$ as an approximation to the Friis transmission equation:

$$P_r = P_t G_t G_r \left(\frac{\lambda}{4\pi d}\right)^2 \approx \frac{k}{d^2}. \tag{8}$$

This equation calculates the power received P_r by the aircraft given the transmitted power P_t, the transmitter and receiver gains (G_t, G_r), and the wavelength λ.

Using this reward model, the total reward (r_t) for an aircraft at position (x, y) can be summed over n power stations. To prevent the aircraft from circling a power station forever, the minimum of the reward and a fixed constant h is considered:

$$r_t = \min\left(h, \sum_{i=1}^{n} \frac{k}{(x - x_i)^2 + (y - y_i)^2}\right). \tag{9}$$

The values of h and k are empirically determined to be 10 and 100 respectively. Interestingly, the dynamics of the aircraft are such that any circling behavior is unlikely even without the minimization function. It requires a very poorly tuned reward function to achieve circling behavior (see Figure 6).

Within the grid-based environment there is a single goal state (x_g, y_g) with a substantial scalar reward. This reward is sized such that the aircraft eventually reaches the goal regardless of the discount factor γ. The Euclidean distance from the current state to the goal state d_g is added to the reward function to further prevent the aircraft from circling the power sources. This creates the final reward function as follows:

$$r = {d_g}^2 + \sum_{i=1}^{n} \min\left(h, \frac{k}{{d_i}^2}\right) \tag{10}$$

The discount factor is generally quite small in order to allow the aircraft to maximize the power received from the power stations in the near-term.

3 Algorithms

State-of-the-art aircraft trajectory optimization typically uses nonlinear programming (NLP). The NLP formulation is typically solved through gradient-based optimization techniques. This method is well suited for accurate physics-based models, but does not satisfy global path optimality. By using policy iteration we can create a framework that guarantees convergence to an optimal solution. The policy iteration experiments are repeated with a SARSA algorithm using the same environment and reward function. Convergence properties and resulting trajectories are then compared.

Partially observable Markov processes have been implemented in similar scenarios with success by Ragi and Chong[RC13] and Kawano[Kaw11]. These models combine state estimation and stochastic models to represent entire systems. These papers provide background for this project, and present some methodology that is used here to translate the continuous flight dynamics model to a Markov decision process.

Two algorithms are used here: policy iteration and SARSA. Policy iteration updates the value function for the entire state-space in a 'brute-force' method that iteratively converges on the optimal values and policy for every state. We compare this with the SARSA model-free method, which attempts to find the optimal policy through exploration. The SARSA algorithm is modified slightly from that presented by Sutton and Barto[SB18]. In step 2, a variable named *max-episodes* is introduced that limits the total number of episode evaluations. This is a practical limitation, since the state-space is too large to reasonably explore in its entirety. Similarly, the aircraft dynamics are complex enough that early policies may take large quantities of iterations to reach the terminal state. Because of this, the maximum number of time steps within an episode are limited by the variable *max-time-steps*.

For the results presented here, the maximum number of SARSA episodes is set to $\mathcal{O}(10^4)$ and the maximum number of time steps is set to 50.

Policy Iteration

1. Initialize $V(s) \in \mathcal{R}$, $\pi(s) \in \mathcal{A}(s)\ \forall s \in \mathcal{S}$

2. Policy evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop $\forall s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* = *false*
   If *policy-stable*, then stop and return $V = v_*$ and $\pi = \pi_*$; else return to 2

SARSA

1. Initialize $\alpha \in (0, 1]$, $Q(s, a) \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$

2. While episode < max-episodes
   Initialize S
   Choose A from S using policy derived from Q (e.g., ϵ-greedy)
   While t < max-time-steps:
   Take action A, observe R,S'
   Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
   $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
   $S \leftarrow S'$; $A \leftarrow A'$
   $t = t + 1$
   Until S is terminal

4 Implementation approach

The state space is three dimensional ($s = (x, y, \psi)$), and the resulting trajectory is plotted in a two-dimensional Cartesian form (similar to Fig. 2). Since the dynamics are time dependent, each coordinate pair also represents a specific time (x_t, y_t). The gradient of the line connecting two coordinate pairs represents the current heading angle.

The optimal policy is simulated based on the same models used to generate the results. This is particularly necessary with the SARSA algorithm which returns values for each state. After the algorithm completes a separate step is used to choose policies and simulate trajectories. The trajectories computed with policy iteration and SARSA are both simulated in a deterministic fashion, that is, the policies choose actions corresponding with the maximum value functions.

5 Results

Results are presented here for both policy iteration and SARSA algorithms. All plots use the same state space of dimensions (60, 30, 20) for states x, y, and ψ respectively. Two power sources are included (purple dots in the figures), located at x, y coordinates of (5, 15) and (25, 35). The starting state for all plots is (15, 1), the terminal state is (15, 55), the time step is 1s, and the velocity is 4m/s. Because the aircraft dynamics can skip over states, there is no guarantee that the aircraft will end each episode directly on top of the terminal state. To solve this, a radius of three meters is placed at the terminal state, within which the episode will terminate and the terminal reward is applied.

5.1 Policy Iteration

The results of the policy iteration algorithm are presented in Figure 2. The discount factor γ is swept from 0.1-0.9 in an effort to validate the model and understand the time-dependency of the reward function. Figure 2 shows that the aircraft's trajectory becomes more linear as the discount factor increases. This is as expected, since larger discount factors mean the aircraft looks farther ahead, and prioritizes the large reward at the target. With a smaller discount factor the aircraft flies closer to the power sources, thereby maximizing the immediate reward.

The convergence time and number of iterations are summarized in the following table. The iteration number includes both policy evaluation iterations and policy improvement iterations. A clear conclusion is that the iteration number and the convergence time increase with the discount factor. This is likely due to increased policy evaluation steps, since the states become more interconnected with higher discount factors. The convergence time is generally quite high due to the large number of possible state combinations (36,000 possibilities) that the algorithm must sweep through multiple times for each iteration.

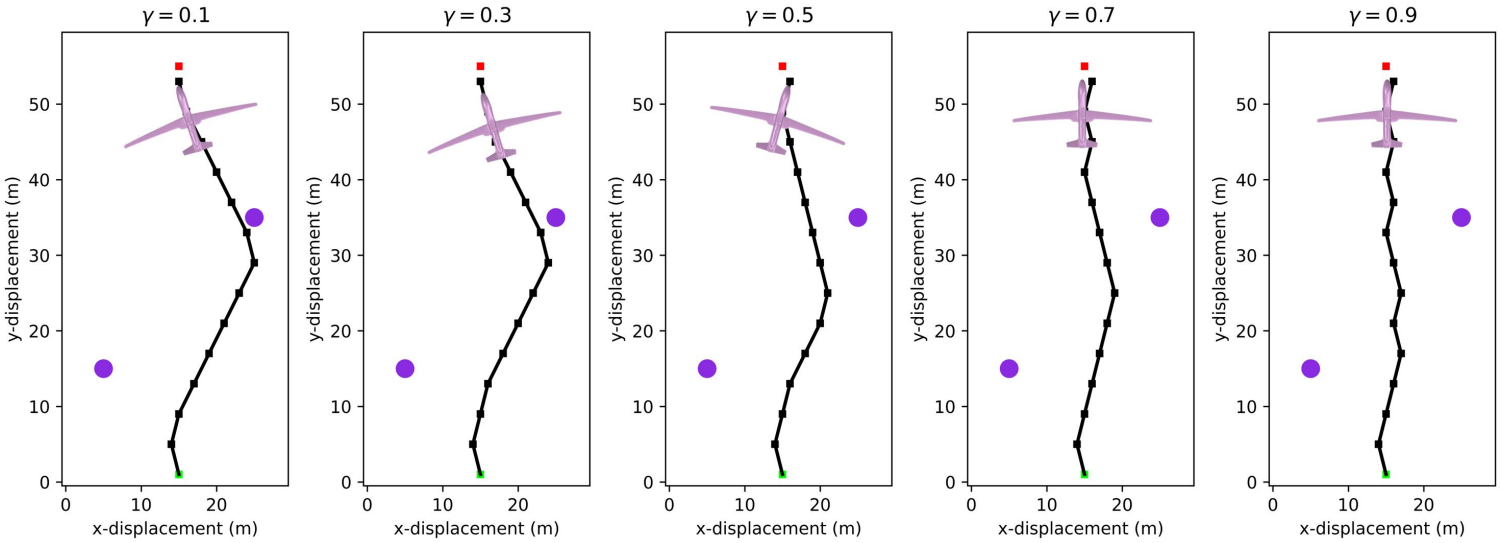

Figure 2: Policy iteration results for $\gamma \in (0.1, 0.3, 0.5, 0.7, 0.9)$.

Table 1: Policy iteration summary.

γ	iterations	time (s)
0.1	10	228
0.3	12	230
0.5	14	234
0.7	21	358
0.9	33	389

5.2 SARSA

The results of the SARSA implementation are shown in figures 3-6. Figures 3-5 depict five plots containing trajectories with various discount factors. Each of these figures presents results for a different value of α, in an attempt to characterize the effect of both these parameters.

The general trend in all figures shows the aircraft takes a more direct path towards the terminal state as the discount factor increases. Like the results from policy iteration, this is because the algorithm is prioritizing the large reward for reaching the terminal state.

As the learning rate (α) increases, the trajectory becomes increasingly affected by the power sources across the entire range of discount factors. This is understandable, since α effectively weights the subsequent action values for a given state. For the results shown here, ϵ is large (0.9). This is a practical limitation, since low values of ϵ prioritize exploration and take a very long time to converge to a reasonable trajectory in such a large state space.

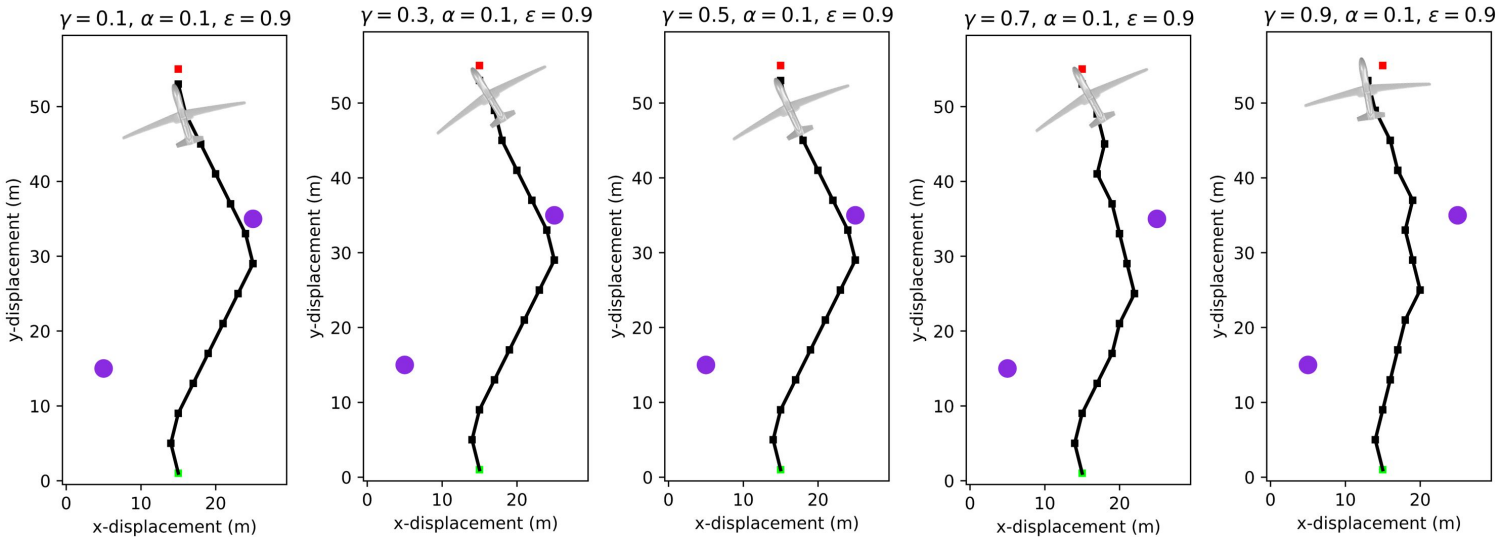

Figure 3: SARSA results for $\gamma \in (0.1, 0.3, 0.5, 0.7, 0.9)$, $\alpha = 0.1$, and $\epsilon = 0.9$.

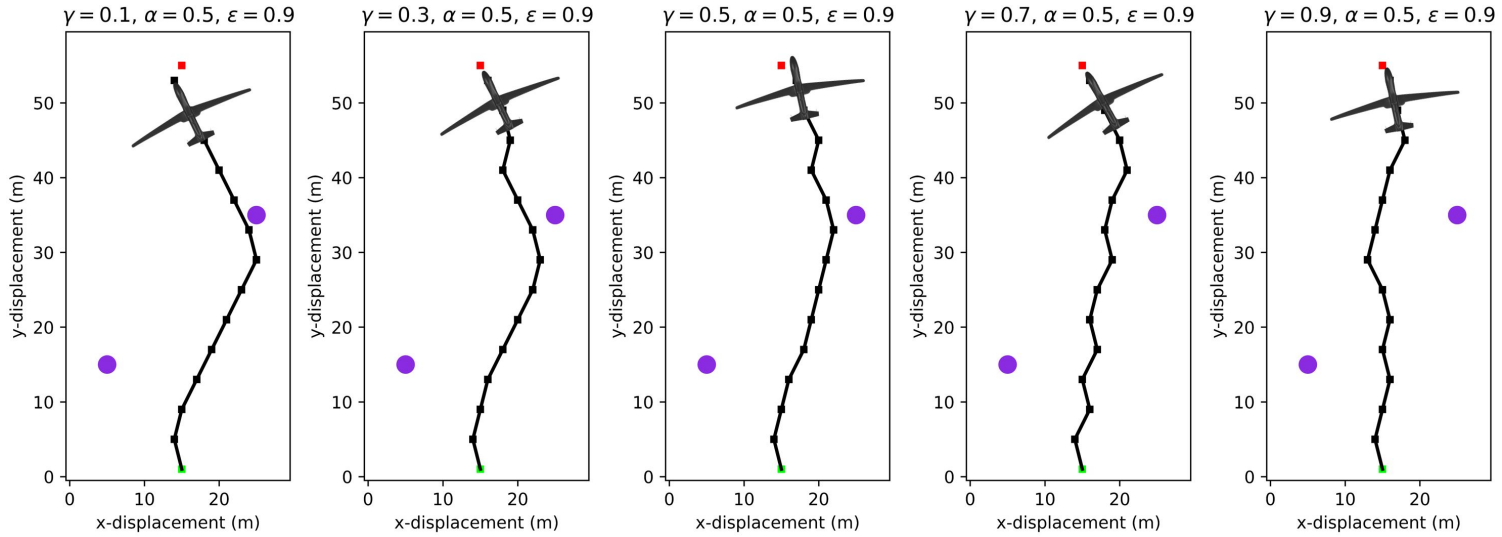

Figure 4: SARSA results for $\gamma \in (0.1, 0.3, 0.5, 0.7, 0.9)$, $\alpha = 0.5$, and $\epsilon = 0.9$.

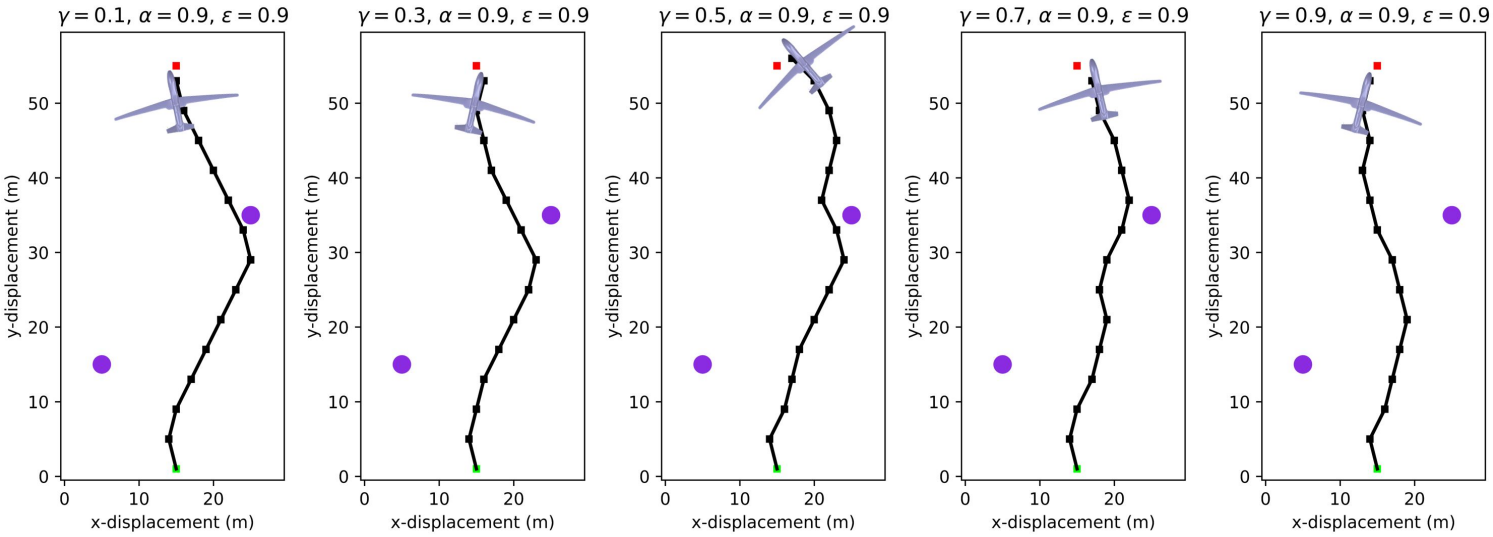

Figure 5: SARSA results for $\gamma \in (0.1, 0.3, 0.5, 0.7, 0.9)$, $\alpha = 0.9$, and $\epsilon = 0.9$.

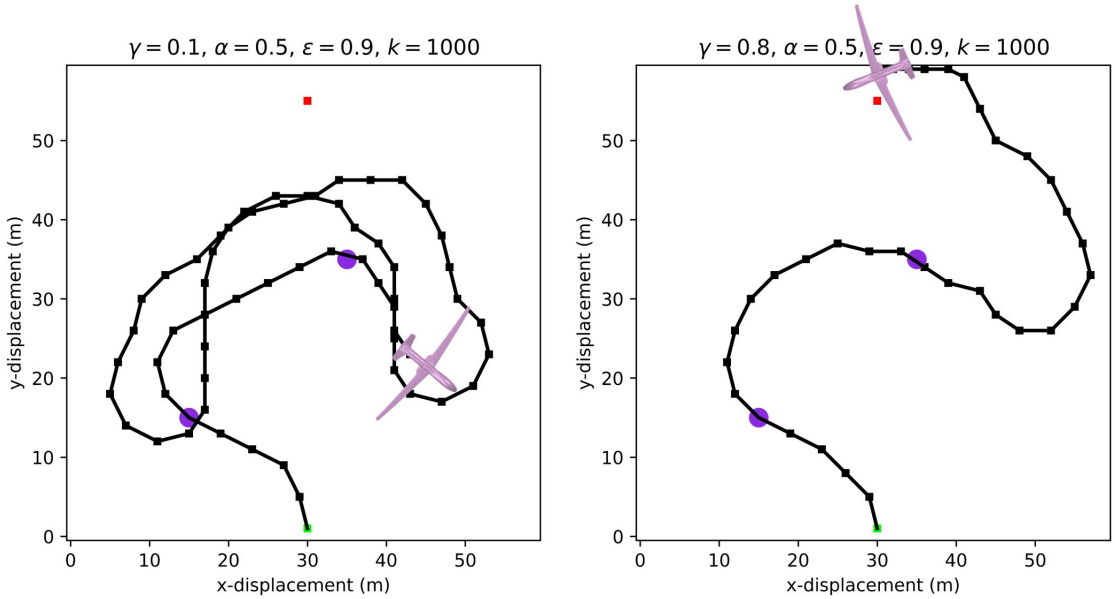

Figure 6: SARSA results for poorly weighted reward functions.

As an experiment to evaluate the effect of the constant k in the reward function, the state-space is temporarily enlarged to (60, 60, 20) and k is set to 1000. The results of this test are shown in Figure 6. Clearly, the reward at the power sources is now so large that the aircraft circles the power stations. If the discount factor is low, the aircraft may never reach the terminal state before the time step limit is reached (see left plot of Figure 6). With a larger discount factor, the aircraft does eventually reach the terminal state, but takes a more circuitous path (see right plot of Figure 6). With 10,000 episodes, the SARSA algorithm completes in roughly one minute for any choice of γ, α, or ϵ.

5.3 Algorithm Comparison

The two algorithms produce similar results but function in very different ways. To understand this, the episode rewards are computed for both algorithms and plotted with respect to the episode number for the

SARSA algorithm and the policy improvement iteration number for the policy iteration algorithm (Figure 7).

The leftmost plot in figure 7 depicts the cumulative reward for a simulated trajectory for each major iteration. Major iterations are computed as each evaluation-improvement combination, resulting in far fewer iterations than the iteration number in Table 1 which counts individual policy evaluation steps. Clearly, policy iteration converges to the optimal reward with few iterations, however, each of these iterations is quite time consuming and requires multiple sweeps of the entire state space.

The right plot in Figure 7 shows the reward for each episode of the SARSA algorithm for 50,000 episodes. The range of the values is roughly similar to that for policy iteration, but the data jumps around somewhat randomly. This is because each episode is determined based upon an ϵ-greedy policy, which intentionally introduces some stochastic behavior to allow for state exploration. If the action choice was purely deterministic and exploitative, the algorithm might miss a possible episode with a greater reward.

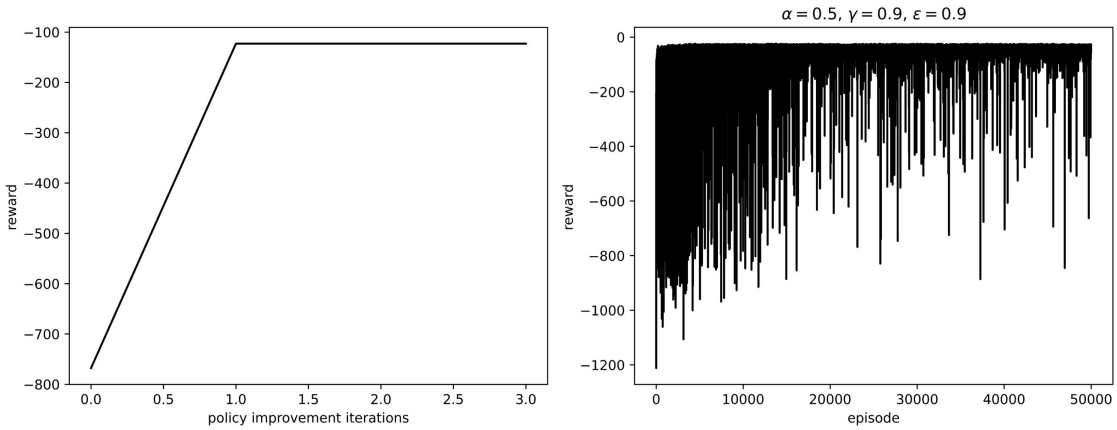

Figure 7: Example policy iteration cumulative reward versus major iterations (left). Example SARSA reward versus episode (right).

The effect of increasing learning rate for SARSA is shown in Figure 8 for α =0.1, 0.5, and 0.9. With a low learning rate the algorithm struggles to converge over 20,000 episodes. A learning rate of 0.5 or greater seems necessary to guarantee good convergence in a reasonable amount of episodes.

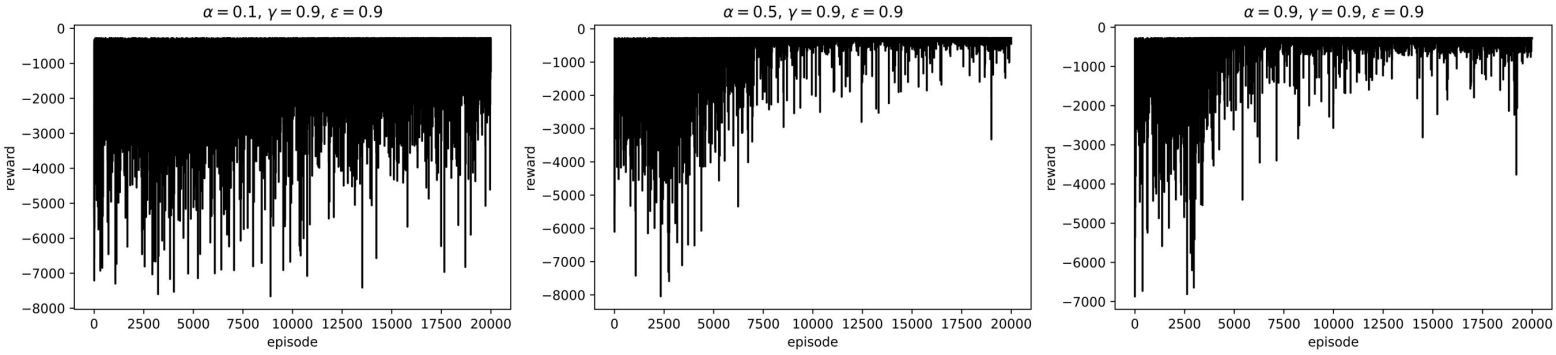

Figure 8: The effect of the learning rate on SARSA cumulative episode rewards.

Since the resulting trajectories are similar, and the computational time for SARSA is much less than policy iteration (approximately one minute versus six minutes) a compelling argument can be made for using the SARSA algorithm rather than policy iteration when using large state spaces. The time difference is inherent to the method, that is, policy iteration contains multiple for-loops which iterate over the entire state-space. These loops are computationally expensive, and are the reason for the increased convergence

time. SARSA is a model-free method and only loops over the action space which is substantially smaller than the state space.

For the SARSA algorithm, the total number of episode time steps can be plotted with respect to the episode iteration number (Figure 9). For this plot, the maximum number of time steps is set at 300 and the episodes resulting from an ϵ-greedy policy are simulated. Clearly, the episode length decreases over the 10,000 simulated episodes. This is indicative of the aircraft taking a shorter, more direct path to the target state as the value function converges to the true value function.

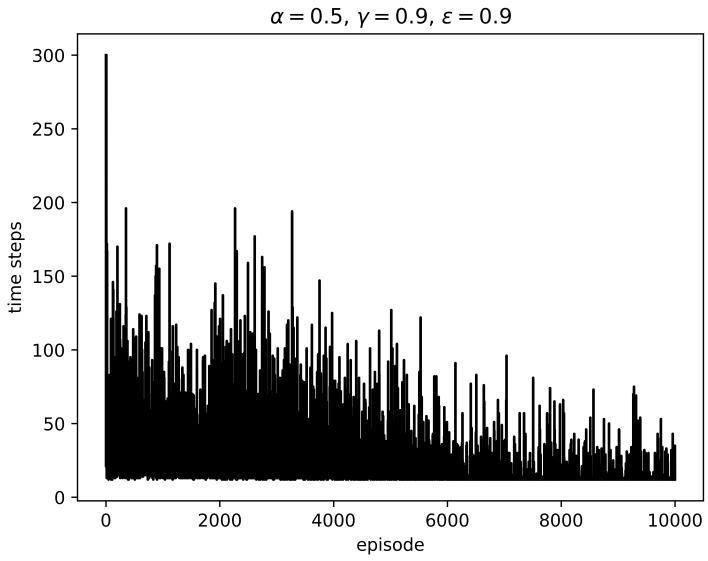

Figure 9: Timesteps versus episodes for SARSA.

6 Conclusion

This paper presents a method for finding optimal aircraft trajectories in an environment with multiple remote power sources. Two algorithms are applied: policy iteration and SARSA. The trajectories are shown to be similar despite the differences between the algorithms.

Both algorithms are computationally expensive when applied to the state space used here. The state space is large, primarily because the aircraft model is developed in a continuous space, and then discretized to work within policy iteration and SARSA algorithms. Perhaps a combination of a coarse SARSA/policy iteration algorithm paired with a continuous gradient-based NLP method would warrant further research.

All code for the two algorithms can be found at: `https://github.com/nichco/mae242_project.git`.

7 Appendix

7.1 SARSA Code

```
# SARSA algorithm
import numpy as np
import random
import matplotlib.pyplot as plt
import time

# choose next action (epsilon-greedy)
def choose_action(state):

    # find the max Q value in the action space for the current state
    max_q = -1*float('inf')
    greedy_actions = []
    for action in A:
        next_state, done = get_state(state, action)
        next_theta_index = (np.where(theta_space == next_state[2])[0]).item()
        q_val = Q[next_state[0], next_state[1], next_theta_index]
        #if q_val > max_q:
        #    max_q, greedy_action = q_val, action

        if q_val > max_q:
            max_q = q_val
            greedy_actions = [] # remove other actions from the list
            greedy_actions.append(action)
        elif q_val == max_q:
            max_q = q_val
            greedy_actions.append(action)

    if np.random.uniform(0, 1) > epsilon:
        a = random.choice(A) # exploration
    else:
        #a = greedy_action # exploitation
        a = random.choice(greedy_actions)
    return a



# choose next action (deterministic)
def choose_deterministic_action(state):

    # find the max Q value in the action space for the current state
    max_q = -1*float('inf')
    greedy_actions = []
    for action in A:
        next_state, done = get_state(state, action)
        next_theta_index = (np.where(theta_space == next_state[2])[0]).item()
        q_val = Q[next_state[0], next_state[1], next_theta_index]
        #if q_val > max_q:
        #    max_q, greedy_action = q_val, action

        if q_val > max_q:
            max_q = q_val
            greedy_actions = [] # remove other actions from the list
            greedy_actions.append(action)
        elif q_val == max_q:
            max_q = q_val
            greedy_actions.append(action)

    # a = greedy_action
    a = random.choice(greedy_actions)
    return a


# round value to closest value in given array
```

```
def round(val, arr):
    index = (np.abs(val - arr)).argmin()

    return arr[index]


# get the next state
def get_state(state, action):
    x, y, theta = state[0], state[1], state[2]

    # rounding takes care of boundaries and strange angles
    next_theta = round(theta + action, theta_space) # rounded to the nearest angle in theta-space
    next_x = round(x + dt*v*np.cos(theta), x_space) # rounded to nearest integer in x-space
    next_y = round(y + dt*v*np.sin(theta), y_space) # rounded to nearest integer in y-space

    next_state = [next_x, next_y, next_theta]

    done = False
    # if [next_x, next_y] == end: done = True
    if next_x > end[0] - b and next_x < end[0] + b and next_y > end[1] - b and next_y < end[1] + b: done=T

    return next_state, done



def get_reward(state,done):

    de = ((end[0] - state[0])**2 + (end[1] - state[1])**2)**0.5 # Euclidean distance to end state
    ds1 = ((source_1[0] - state[0])**2 + (source_1[1] - state[1])**2)**0.5 # Euclidean distance to power s
    ds2 = ((source_2[0] - state[0])**2 + (source_2[1] - state[1])**2)**0.5 # Euclidean distance to power s

    if done: reward = 1
    # else: reward = -de + np.min([2,10/(ds1**2)]) + np.min([2,10/(ds2**1)])
    else: reward = -de + 100/(ds1**2 + 1) + 100/(ds2**2 + 1)

    return reward



#Function to learn the Q-value
def update(state, next_state, reward):
    theta_index = (np.where(theta_space == state[2])[0]).item()
    next_theta_index = (np.where(theta_space == next_state[2])[0]).item()

    term = reward + gamma*Q[next_state[0], next_state[1], next_theta_index] - Q[state[0], state[1], theta_i
    Q[state[0], state[1], theta_index] = Q[state[0], state[1], theta_index] + alpha*term


def sarsa():
    ep_num = 0
    for episode in range(total_episodes):
        print('episode: ',ep_num)
        r = 0 # cumulative episode reward sum
        state = start
        action = choose_action(state)

        # while True:
        t = 0
        while t < max_time_steps:
            next_state, done = get_state(state, action)
            reward = get_reward(next_state,done)
            r = r + reward
            next_action = choose_action(next_state)

            # update the Q value
            update(state, next_state, reward)

            # update current state and action
```

```
            state = next_state
            action = next_action

            # if state is terminal, break
            if done: break

            t += 1

        ep_num += 1
        rvec[episode] = 1*r
        tvec[episode] = 1*t



# simulate epsilon-greedy or deterministic trajectories
def simulate():
    img = np.ones((n,m,3))
    img[start[0], start[1], :] = [0,1,0] # start is green

    fig, ax = plt.subplots()
    ax.scatter(source_1[0],source_1[1],color='blueviolet',s=100)
    ax.scatter(source_2[0],source_2[1],color='blueviolet',s=100)
    # ax.legend(['power source'])

    state = start
    t = 0
    while t < max_time_steps:
        action = choose_deterministic_action(state) # deterministic
        # action = choose_action(state) # stochastic
        next_state, done = get_state(state, action)

        ax.plot([state[1], next_state[1]], [state[0], next_state[0]], color='k', linewidth=2)

        state = next_state

        img[state[0], state[1], :] = [0,0,0] # path is black

        t += 1
        if done: break

    img[end[0], end[1], :] = [1,0,0] # end is red

    ax.imshow(img)
    ax.invert_yaxis()
    ax.set_xlabel('x-displacement (m)')
    ax.set_ylabel('y-displacement (m)')
    ax.set_title(r'$\gamma = 0.8$, $\alpha = 0.5$, $\epsilon = 0.9$, $k=1000$')
    # plt.savefig('sarsa_g0.8_loop_2.png', dpi=1200, bbox_inches='tight')
    plt.show()



# run SARSA learning:
total_episodes = 20000
alpha = 0.9
gamma = 0.9 # 0.5
epsilon = 0.9
max_time_steps = 200

min_steer, max_steer = -np.pi/6, np.pi/6
A = np.linspace(min_steer, max_steer, 7)

n = 60 # x discretizations
m = 30 # y discretizations
o = 20 # heading angle discretizations
dt = 1 # timestep
v = 4 # velocity
b = 3 # done radius
```

```
x_space = np.arange(n)
y_space = np.arange(m)
theta_space = np.linspace(-np.pi, np.pi, o)

end = [55,15]
start = [1,15,theta_space[9]]
source_1, source_2 = [5,15], [25,35]

# initialize Q matrix
Q = np.zeros((n,m,o))
# initialize cumulative reward vector
rvec = np.zeros((total_episodes))
tvec = np.zeros((total_episodes))

t1 = time.perf_counter()
sarsa()
t2 = time.perf_counter()
print('time: ',t2-t1)

plt.plot(rvec,color='k')
plt.title(r'$\alpha=0.9$, $\gamma=0.9$, $\epsilon=0.9$')
plt.xlabel('episode')
plt.ylabel('reward')
plt.savefig('sarsa_reward_0.9.png', dpi=1200, bbox_inches='tight')
plt.show()
"""
plt.plot(tvec,color='k')
plt.title(r'$\alpha=0.5$, $\gamma=0.9$, $\epsilon=0.9$')
plt.xlabel('episode')
plt.ylabel('time steps')
plt.savefig('sarsa_tvec.png', dpi=1200, bbox_inches='tight')
plt.show()
"""

simulate()
```

7.2 Policy Iteration Code

```
# policy iteration algorithm
import numpy as np
import random
import matplotlib.pyplot as plt
import time

# round value to closest value in given array
def round(val, arr):
    index = (np.abs(val - arr)).argmin()

    return arr[index]

# get the next state
def get_state(state, action):
    x, y, theta = state[0], state[1], state[2]

    # rounding takes care of boundaries and strange angles
    next_theta = round(theta + action, theta_space) # rounded to the nearest angle in theta-space
    next_x = round(x + dt*v*np.cos(theta), x_space) # rounded to nearest integer in x-space
    next_y = round(y + dt*v*np.sin(theta), y_space) # rounded to nearest integer in y-space

    next_state = [next_x, next_y, next_theta]

    done = False
    # if [next_x, next_y] == end: done = True
    if next_x > end[0] - b and next_x < end[0] + b and next_y > end[1] - b and next_y < end[1] + b: done=T
```

```
    return next_state, done


def get_reward(state,done):

    de = ((end[0] - state[0])**2 + (end[1] - state[1])**2)**0.5 # Euclidean distance to end state
    ds1 = ((source_1[0] - state[0])**2 + (source_1[1] - state[1])**2)**0.5 # Euclidean distance to power s
    ds2 = ((source_2[0] - state[0])**2 + (source_2[1] - state[1])**2)**0.5 # Euclidean distance to power s

    # if [state[0], state[1]] == end: reward = 100
    if done: reward = 100 #100
    else: reward = -de + 100/(ds1**2 + 1) + 100/(ds2**1 + 1)
    # else: reward = -de + np.min([2,20/(ds1**2 + 1)]) + np.min([2,20/(ds2**1 + 1)])

    return reward


def prob():
    #sigma = 1
    #p = (1/(sigma*((2*np.pi)**0.5)))*np.exp(-0.5*((pi/sigma)**2))
    n = len(A)
    return 1/n


# simulate pi within policy iteration to calculate reward values
def sim_pi():
    state1 = start
    t, rval = 0, 0

    while t < max_time_steps:
        theta_index = (np.where(theta_space == state1[2])[0]).item()
        pi = policy[state1[0],state1[1],theta_index]
        next_state, done = get_state(state1, pi)
        reward = get_reward(next_state,done)
        rval += reward
        t += 1
        state1 = next_state
        if done: break

    return rval


def policy_iteration():
    print('policy_iteration')
    iterations = 0

    for iter in range(max_iter):
        rval = sim_pi()
        rvec.append(rval)
        while True: # evaluation
            print('evaluation')
            delta = 0
            for i in range(n):
                for j in range(m):
                    for k in range(o):
                        old_value = values[i,j,k]
                        new_value = 0
                        for a in A:
                            nx, done = get_state([i,j,theta_space[k]],a)
                            theta_index = (np.where(theta_space == nx[2])[0]).item()
                            reward = get_reward(nx,done)
                            new_value += prob()*(reward + gamma*values[nx[0],nx[1],theta_index])
                        values[i,j,k] = new_value
                        delta = max(delta, abs(old_value - new_value))
            print('delta: ',delta)
            iterations += 1
            if delta < error: break
```

```
            policy_stable = True # improvement
            print('improvement')
            for i in range(n):
                for j in range(m):
                    for k in range(o):
                        old_action = policy[i,j,k]
                        new_action, max_val = None, -1*float('inf')
                        for a in A:
                            val = 0
                            for new_a in A:
                                nx, done = get_state([i,j,theta_space[k]],a)
                                theta_index = (np.where(theta_space == nx[2])[0]).item()
                                reward = get_reward(nx,done)
                                val += prob()*(reward + gamma*values[nx[0],nx[1],theta_index])
                            if val > max_val: max_val, new_action = val, a
                        policy[i,j,k] = new_action
                        if old_action != new_action: policy_stable = False

            iterations += 1
            print(iterations)
            if policy_stable == True: break

        return values, policy, iterations


    # choose next action (deterministic)
    def choose_deterministic_action(state):

        # find the max value in the action space for the current state
        max_q = -1*float('inf')
        greedy_actions = []
        for action in A:
            next_state, done = get_state(state, action)
            next_theta_index = (np.where(theta_space == next_state[2])[0]).item()
            q_val = values[next_state[0],next_state[1],next_theta_index]
            #if q_val > max_q:
            #    max_q, greedy_action = q_val, action
            if q_val > max_q:
                max_q = q_val
                greedy_actions = [] # remove other actions from the list
                greedy_actions.append(action)
            elif q_val == max_q:
                max_q = q_val
                greedy_actions.append(action)

        # a = greedy_action
        a = random.choice(greedy_actions)
        return a

    # simulate trajectories
    def simulate():
        img = np.ones((n,m,3))
        img[start[0], start[1], :] = [0,1,0] # start is green

        fig, ax = plt.subplots()
        ax.scatter(source_1[0],source_1[1],color='blueviolet',s=100)
        ax.scatter(source_2[0],source_2[1],color='blueviolet',s=100)
        # ax.legend(['power source'])

        state = start
        t = 0
        while t < max_time_steps:
            action = choose_deterministic_action(state)
            next_state, done = get_state(state, action)

            ax.plot([state[1], next_state[1]], [state[0], next_state[0]], color='k', linewidth=2)

            state = next_state
```

```
            img[state[0], state[1], :] = [0,0,0] # path is black
            t += 1
            if done: break

        img[end[0], end[1], :] = [1,0,0] # end is red

        ax.imshow(img)
        ax.invert_yaxis()
        ax.set_xlabel('x-displacement (m)')
        ax.set_ylabel('y-displacement (m)')
        ax.set_title(r'$\gamma = 0.9$')
        # plt.savefig('pi_g0.9.png', dpi=1200, bbox_inches='tight')
        plt.show()


# policy iteration
gamma = 0.5
min_steer, max_steer = -np.pi/6, np.pi/6
A = np.linspace(min_steer, max_steer, 7).tolist()

n = 60 # x discretizations
m = 30 # y discretizations
o = 20 # heading angle discretizations
dt = 1 # timestep
v = 4 # velocity
b = 3 # done radius

x_space = np.arange(n)
y_space = np.arange(m)
theta_space = np.linspace(-np.pi, np.pi, o)

end = [55,15]
start = [1,15,theta_space[9]]
source_1, source_2 = [5,15], [25,35]

# initialize values and policy
values = np.zeros((n,m,o))
policy = np.zeros((n,m,o))
error = 1E0
max_iter = 10 # for policy iteration
max_time_steps = 50 # for simulation

rvec = []
t1 = time.perf_counter()
values, policy, iterations = policy_iteration()
t2 = time.perf_counter()
print('time: ', t2 - t1)
print('iter: ',iterations)
# print(np.array2string(values,separator=','))
# print(np.array2string(policy,separator=','))
plt.plot(rvec,color='k')
print(rvec)
plt.xlabel('policy improvement iterations')
plt.ylabel('reward')
plt.savefig('pi_reward.png', dpi=1200, bbox_inches='tight')
plt.show()

simulate()
```

References

[Kaw11] Hiroshi Kawano. "Study of path planning method for under-actuated blimp-type UAV in stochastic wind disturbance via augmented-MDP". In: *2011 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*. 2011, pp. 180–185. DOI: 10.1109/AIM.2011.6027018.

[RC13] Shankarachary Ragi and Edwin K. P. Chong. "UAV Path Planning in a Dynamic Environment via Partially Observable Markov Decision Process". In: *IEEE Transactions on Aerospace and Electronic Systems* 49.4 (2013), pp. 2397–2412. DOI: 10.1109/TAES.2013.6621824.

[AGS13] Wesam H. Al-Sabban, Luis F. Gonzalez, and Ryan N. Smith. "Wind-energy based path planning for Unmanned Aerial Vehicles using Markov Decision Processes". In: *2013 IEEE International Conference on Robotics and Automation*. 2013, pp. 784–789. DOI: 10.1109/ICRA.2013.6630662.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.