Caleb Logemann
AERE 504 Intelligent Air Systems
Project 2

1 Description of Methods

1.1 Part 1 - Discrete Grid World

For this problem I formed the transition and reward function from the data using the maximum liklihood approach. After forming these function I used the Gauss-Seidel Value iteration to compute the maximal Utility of each state and then extracted the policy from U. This algorithm took approximately 2 seconds to run on the given data.

1.2 Part 2 - Continuous Grid World

For this problem I followed the approach of Xuxi Yang in his research. I implemented a tree search to compute the proper policy on the fly during the simulation. The algorithm searched out to all possible sequences of 6 actions. At each state, s the reward is computed as

$$r_s = 1 - d_s/d_{max}$$

where d_s is the distance to the goal position at this state and d_{max} is the maximum distance possible that is

$$d_{max} = \sqrt{500^2 + 500^2} = 500\sqrt{2}$$

If the state is invalid, that is the state violates the boundaries of the region, then the reward is -1. There is also a reward associated with each node in the tree. Each node in the tree has a state associated with it, that state is generated by updating the state of the parent node assuming a given action was taken. The state of the root node is the current state. The reward for a leaf node is just the reward for the state associated with that node. If a state is not a leaf state the the reward for the node is an average of the reward at that state and it children's rewards. The reward is computed as

$$r_n = \frac{1}{2} r_s + \frac{1}{2} \frac{r_l + r_r + r_0}{3}$$

where r_s is the reward of the state of the node and r_l, r_r, r_0 are the rewards of the nodes children.

The algorithm computes the reward of all three possible actions and then chooses the action with the highest reward. Since this policy is computed as the simulation runs, it takes no longer then the original simulation took to run and no precomputation is necessary. However if the tree depth is increased some slow down of the simulation is present. The depth of the tree can go to 7 or 8 actions before the simulation is noticeably slowed.

The algorithm almost always reaches the goal state. The only cases where it does not is when the goal state is very close to the initial position of the plane and the plane cannot turn quickly enough to reach the goal. In this case the plan runs into the wall close to the goal state.

2 Appendix

2.1 Part 1 - Discrete Grid World

This first Julia file implements, the Value Iteration, QLearning, and SARSA algorithms for finding a policy.

```
using LinearAlgebra

function valueIteration(T, R, states, actions, discountFactor, tol,
     maxIterations)
    k = 0;
    U = Dict(s => 0.0 for s in states)
    oldnorm = norm(U)
    while (k <= maxIter)
        k = k+1;
        for s in states
            U[s] = maximum([R[s,a] + discountFactor*sum([T[sp,s,a]*U[sp] for sp
                 in states]) for a in actions])
        end
        newnorm = norm(U);
        if (abs(newnorm - oldnorm) < tol)
            break;
        end
        oldnorm = newnorm;
    end
    return U;
end

function policyExtraction(states, actions, U, T, R)
    policy = Dict();
    for s in states
        maxValue = nothing;
        maxAction = 0;
        for a in actions
            value = R[s,a] + sum([T[sp,s,a].*U[sp] for sp in states]);
```

```
            if (maxValue == nothing || value >= maxValue)
                maxValue = value;
                maxAction = a;
            end
        end
        policy[s] = maxAction;
    end
    return policy;
end

function policyExtractionQ(states, actions, Q)
    policy = Dict();
    for s in states
        maxValue = nothing;
        maxAction = 0;
        for a in actions
            value = Q[s, a];
            if (maxValue == nothing || value >= maxValue)
                maxValue = value;
                maxAction = a;
            end
        end
        policy[s] = maxAction;
    end
    return policy;
end

function qLearning(states, actions, data, learningRate, discountFactor)
    t = 0;
    numRows = size(data,1);
    Q = Dict((s, a) => 0.0 for s in states, a in actions);
    for i = 1:numRows
        s = df[i,:s];
        a = df[i,:a];
        r = df[i,:r];
        sp = df[i,:sp];
        Q[s, a] = Q[s, a] + learningRate*(r + discountFactor*maximum([Q[sp, ap]
             for ap in actions]) - Q[s, a]);
        t = t+1;
    end
    return Q;
end

function sarsa(states, actions, data, learningRate, discountFactor)
    t = 0;
    numRows = size(data,1);
    Q = Dict((s, a) => 0.0 for s in states, a in actions);
    for i = 1:numRows-1
        s = df[i,:s];
        at = df[i,:a];
        atp1 = df[i+1,:a];
        r = df[i,:r];
        sp = df[i,:sp];
        Q[s, at] = Q[s, at] + learningRate*(r + discountFactor*Q[sp, atp1] - Q[s
             , at]);
        t = t+1;
    end
    return Q;
```

```
end
```

This script uses the previous function to actually compute the policy for the given data.

```
using CSV, DataFrames, Printf

df = CSV.read("transitions.csv");
states = unique(df[:s]);
actions = unique(df[:a]);
numRows = size(df,1);
counts = Dict((s, a, sp) => 0 for s in states, a in actions, sp in states);
rewardsum = Dict((s, a) => 0.0 for s in states, a in actions);

for i = 1:numRows
    s = df[i,1];
    a = df[i,2];
    r = df[i,3];
    sp = df[i,4];

    counts[s, a, sp] = counts[s, a, sp] + 1;
    rewardsum[s, a] = rewardsum[s, a] + r;
end

T = Dict((sp, s, a) => 0.0 for s in states, a in actions, sp in states);
R = Dict((s, a) => 0.0 for s in states, a in actions);

for s in states
    for a in actions
        countSum = sum([counts[s, a, sp] for sp in states]);
        if(countSum != 0)
            R[s, a] = rewardsum[s, a]/countSum;
            for sp in states
                T[sp, s, a] = counts[s, a, sp]/countSum;
            end
        end
    end
end

include("ReinforcementLearning.jl")
# Value Iteration
discountFactor = 0.95;
tol = 1e-8;
maxIter = 100;
U = valueIteration(T, R, states, actions, discountFactor, tol, maxIter);
policy = policyExtraction(states, actions, U, T, R);

# write policy to file
filename = "transitions.policy";
file = open(filename, "w");
for i = 1:size(states,1)
    @printf(file, "%i\n", policy[i]);
end
close(file);
```

```
# Qlearning
#learningRate = 0.5;
#Q = qLearning(states, actions, df, learningRate, discountFactor);
#policy2 = policyExtractionQ(states, actions, Q);
# SARSA
#Q = sarsa(states, actions, df, learningRate, discountFactor);
#policy3 = policyExtractionQ(states, actions, Q);
```

2.2 Part 2 - Continuous Grid World

The following python code implements my tree search policy for part 2 of the project. In order to use this code, "from part2policy import *" needs to be added to the import statements of the simulator file.

```
import math
import pdb
width = 500
height = 500

def getNextState(current_state, action):
    phip = current_state[4] + action
    vxp = -2*math.sin(math.radians(phip))
    vyp = -2*math.cos(math.radians(phip))
    xp = current_state[0] + vxp
    yp = current_state[1] + vyp
    next_state = [xp, yp, vxp, vyp, phip,
            current_state[5], current_state[6]]
    return next_state

def isStateValid(state):
    isXValid = state[0] > 0 and state[0] < width
    isYValid = state[1] > 0 and state[1] < height
    return isXValid and isYValid

def distanceToGoal(state):
    x_distance = state[5] - state[0]
    y_distance = state[6] - state[1]
    d = math.sqrt(x_distance**2 + y_distance**2)
    return d

def reward(state):
    if(isStateValid(state)):
        max_d = math.sqrt(2)*width
        d = distanceToGoal(state)
        r = 1 - d/max_d
    else:
        r = -1
    return r

def tree_search(state, action, depth):
    next_state = getNextState(state, action)
```

```
    r = reward(next_state)
    if (depth == 0 or ~isStateValid(next_state)):
        return r
    else:
        rp2 = tree_search(next_state, 2, depth-1)
        r0  = tree_search(next_state, 0, depth-1)
        rm2 = tree_search(next_state, -2, depth-1)
        return 0.5*r + 0.5*(rp2 + r0 + rm2)/3

def policy(current_state):
    tree_depth = 6
    rp2 = tree_search(current_state, 2, tree_depth)
    r0 = tree_search(current_state, 0, tree_depth)
    rm2 = tree_search(current_state, -2, tree_depth)
    # print([rp2, r0, rm2])
    if(rp2 >= r0 and rp2 >= rm2):
        return 2
    elif (r0 >= rm2 and r0 >= rp2):
        return 0
    else:
        return -2
```