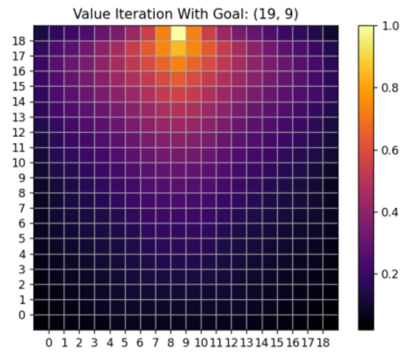
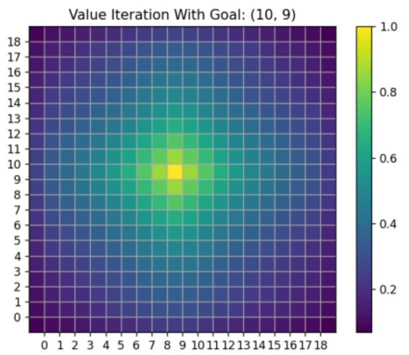
HW 3: MDP, Gridworld, Bellman's Equation

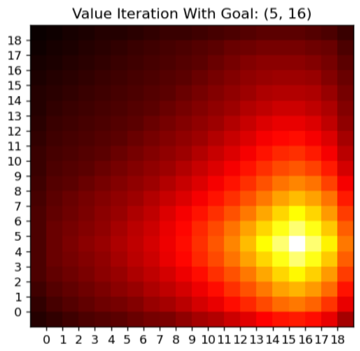
Nicholas Strohmeyer (nas3882)

Part 1: Value Iteration

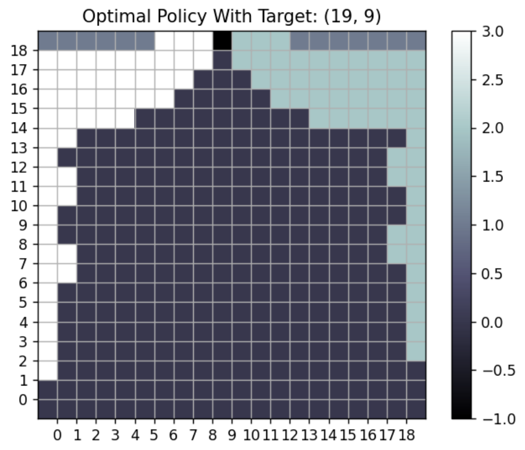
- Various heatmaps allow us to visualize value distributions given different end goal states
- In all cases value emanates out from the center target state (where the reward is densest)
- State space sweeps to converge: **54** (case 1: when goal = (19,9))



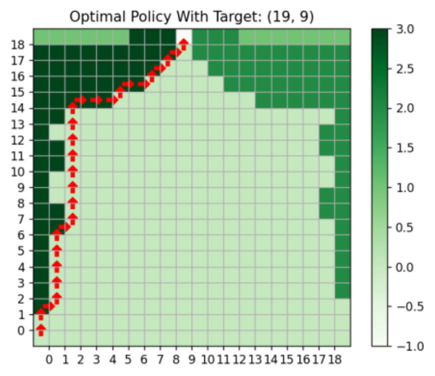


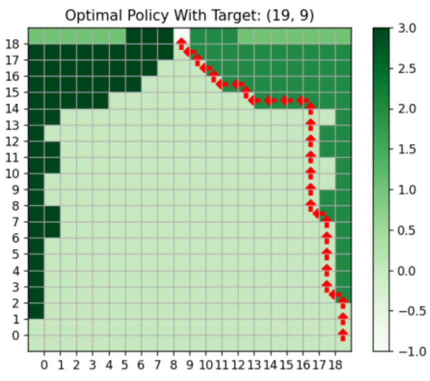


Below is the policy generated by value iteration in case 1 where the goal = (19,9). In part 2, we will look at the policy function using policy iteration and some smoother/ cleaner lines will emerge. However, this policy is generally correct. If we are to the right, the policy says go left (sky blue), if we are to the left, the policy says go right (white). For most of the area under the goal state, the policy prescribes an up action. We will see that sometimes it will be slightly more efficient to go right or left (depending on the state) when we look at the policy iteration map. The silver line across the top indicating "down" is interesting. I explain why I believe the alogrithms prescribed these actions in part 2 (it has to do with the boundary). Lastly, I include a plot of the prescribed trajectory from 2 different starting states: (0,0) and (0,19):



Steps taken = 28



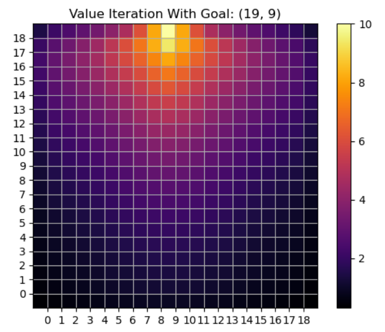


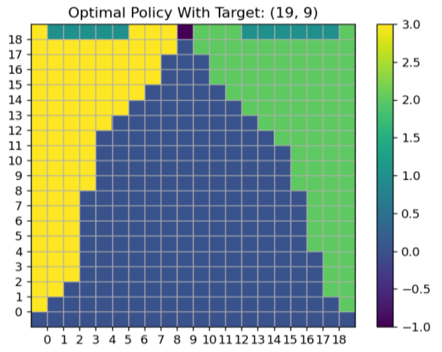
Part 2: Policy Iteration

- State Space Sweeps when goal (19,9): **49**

We see policy iteration did better(i.e. required less sweeps of the state space to converge) We could likely converge even faster with a better initial guess for the policy (right now it is randomized). For example, we could instead try an initial policy such as: "go right if column < 9 , go left if column > 9, and go up if column = 9" (based on our knowledge of starting state and goal state)

- I come back to test this hypothesis in part 3





Left Map:

The inferno map on left = optimal value function. I arbitrarily made the target state a much higher reward than surrounding states in order to help convergence along (that is why the scale goes up to 10). We can see that value emanates outward from grid square (19,9) almost like a vector field pulling surrounding particles in towards the target state. We can see this function is essentially identical to the value iteration algorithm result.

Right Map:

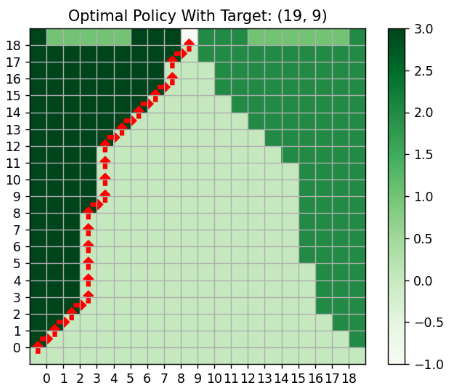
4-toned map represents 4 actions (the reward state is marked -1, colored purple to indicate no action taken once reached) 0 = up (blue) , 1 = down (teal) , 2 = left (green) , 3 = right (yellow)

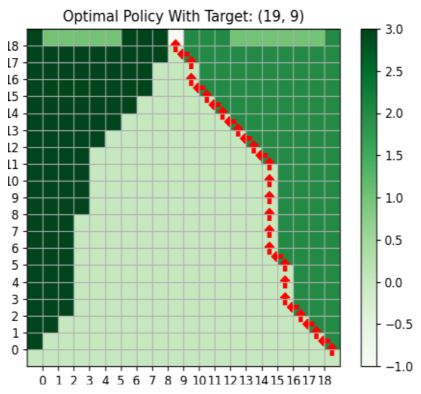
Generally the optimal policy is doing as we would expect. If we are directly below the target state, the optimal actionis to move up. If we are further off to the right, the optimal action is to go left and vice versa.

Why do we move down along the top row ?

The interesting exception to this rule is across the top row where our target state lives (that is y = 19). Unless we are right next to (19,9), the optimal policy actually tells us to go down. The reason for this is not intuitive at first, but after thinking about it for a little while, I think it makes sense. It is the effect of calculating value along the boundary where one of the actions is illegal (we cannot go up). These states end up with one less grid square contributing to the overall expectation in bellman's equation. The effect is that states immediately below (19,9) have slightly higher reward values than those to the immediate left and right. This effect emanates to the left and right (in row y = 18) and thus gives the down action a slightly higher reward in those aforementioned squares (19, 1-5) & (19, 12-18). Fortunately, closer to (19,9) the policy still correctly identifies the optimal move to the left and right.

Using the optimal policy function, I simulated a trajectory starting from 2 states again: (0,0) and (19,0).



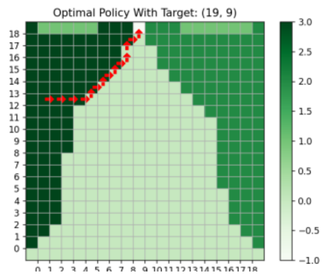


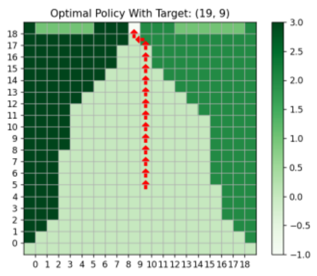
It is interesting to see how this moves exactly along the boundary of the "up" and "right" regions. This makes sense. Deviating too deeply into either region would represent inefficieny (ie wasted movements). This is a satisfying visual representation of the optimality of bellman's equation. The agent is not only finding the target state but doing so in as few of steps as possible. Note: this explanation is also true of the value iteration trajectories we saw in part 1. In part 3, I consider the differing geometries between the optimal policy functions of the 2 algorithms

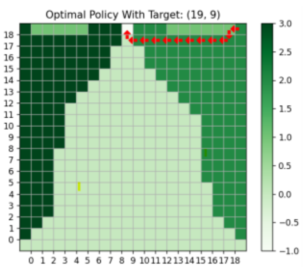
Steps taken = 28

Just for fun, here a few other simulated trajectories

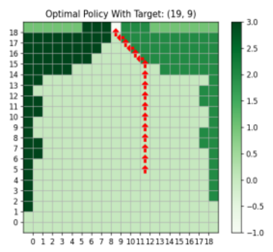
Policy iteration

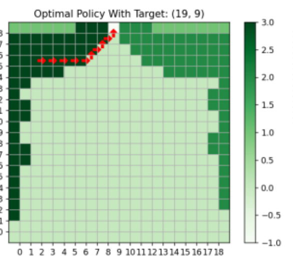


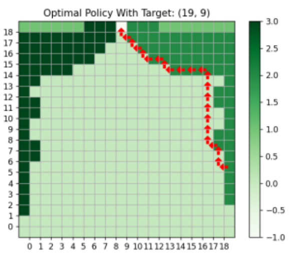


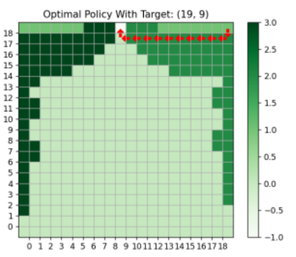


Value Iteration

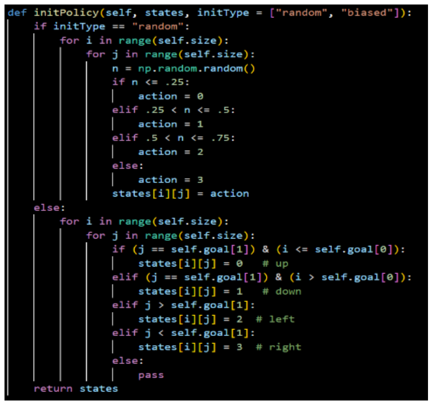








Part 3: Additional Considerations



```
def initPolicy(self, states, initType = ["random", "biased"]):
    if initType == "random":
        for i in range(self.size):
            for j in range(self.size):
                n = np.random.random()
                if n <= .25:
                    action = 0
                elif .25 < n <= .5:
                    action = 1
                elif .5 < n <= .75:
                    action = 2
                else:
                    action = 3
                states[i][j] = action
    else:
        for i in range(self.size):
            for j in range(self.size):
                if (j == self.goal[1]) & (i <= self.goal[0]):
                    states[i][j] = 0   # up
                elif (j == self.goal[1]) & (i > self.goal[0]):
                    states[i][j] = 1   # down
                elif j > self.goal[1]:
                    states[i][j] = 2  # left
                elif j < self.goal[1]:
                    states[i][j] = 3  # right
                else:
                    pass
    return states
```

1. Convergence Times

In part 2, I hypothesized that using a more informed initial guess would make policy iteration converge faster. But after switching to the biased policy shown on the right, and yet did not observe faster convergence. I still think it is reasonable that this would be true in larger state spaces (although did not test that here)

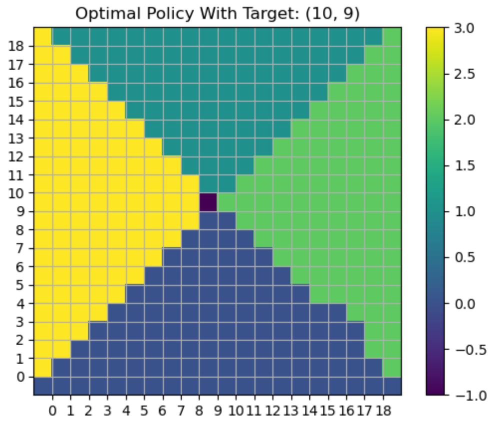
On the other hand, I found that changing the target state did highlight a more significant difference between policy iteration and value iteration and affected speed of convergence greatly. Specifically, value iteration consistently requires about 50-55 state space sweeps in order to converge, regardless of the goal state. However, the closer the goal state is to our initial, policy iteration can converge significantly faster. The table to the right displays this data.

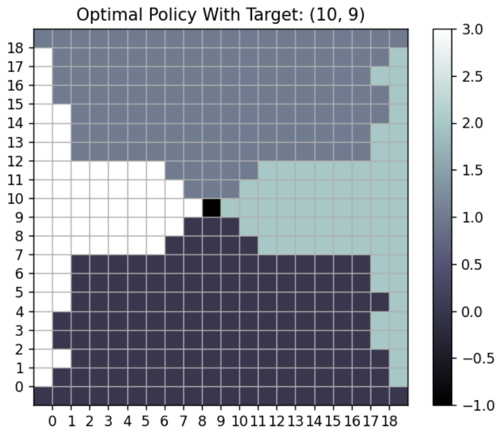
Goal	Value Iteration	Policy Iteration	Delta
19,9	54	49	5
10,9	47	29	18
19, 0	53	1	52
5,6	52	22	30
19,19	53	55	-2
0,19	54	49	5
1,0	55	21	34
11,1	54	38	16

2. Policies with different Goal States

Goal = (10,9) i.e. center-grid target state

I was curious to see what would happen and how the 2 algorithms would compare when we placed the goal state exactly at the center of the grid (or close to exact center since even # of squares). What type of policy would each algorithm come up with? We saw that policy iteration created slightly smoother boundaries before, but if the target were in a more "balanced" position, would the 2 policies differ so greatly?





The figures on the right show that policy iteration does indeed create cleaner boundaries again (above) as value iteration (below) is again a more "choppy" policy. This is evidenced by the rotational symmetry in the top diagram. No matter where we are in gridworld, we always tend to funnel down towards the center along a sort of "uniform vector field" on the diagonals if you will. However, value iteration is optimal as well although not as perfectly symmetric in its action space. No matter what region we are in, we always take the minimum number of steps to get to the center (that is if we need to go right, we will never go left afterwards and if we need to go up, we will never go down etc.)

My hypothesis for why the policy iteration comes up with a "smoother" action space is that it could be due to the symmetry that was built into the initial guess. It would be interesting to explore further

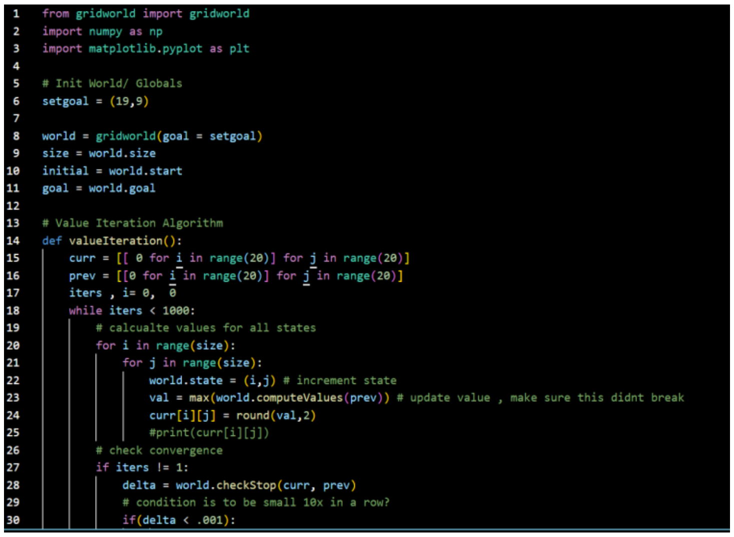
Part 4: Project Code

I created 2 python files to simualte gridworld: A gridworld **class** that contains about half of the methods and the parameters for a particular construct of gridworld (ie size, intial conditions etc.) and **runGW.py**, where I put the main() routine, the plotting functions **as well as the actual policy and value iteration algorithms** (although policy/value iteration both use methods that are contained in the gridworld class). It would make more sense to keep policy/value iteration algorithms in the same file as the methods they depend on, so hopefully this is not too confusing. I added method descriptions/documentation to hoepfully make a little clearer/ easier to read

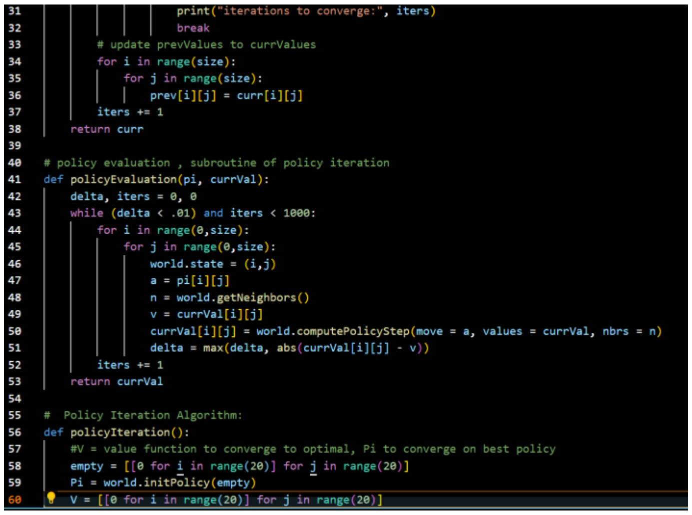
1st File

runGW.py:

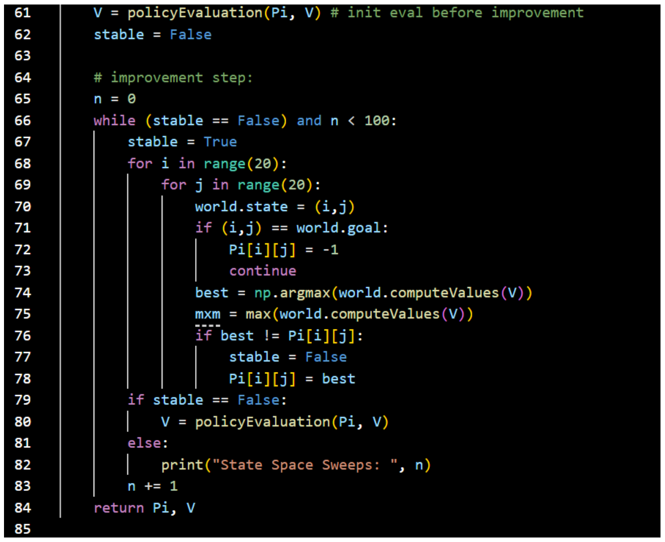
- Globals : allows us to set some attributes about gridworld (I use what the assignment calls for)
- plotvalueMap()
- plotPolicyMap()
- plotSimPath() - plots a simualted trajectory through gridworld
- main() -- gives user choice of running policy or value iteration – computes path starting from initial (0,0) given either case … based on the computed value function



```python
from gridworld import gridworld
import numpy as np
import matplotlib.pyplot as plt

# Init World/ Globals
setgoal = (19,9)

world = gridworld(goal = setgoal)
size = world.size
initial = world.start
goal = world.goal

# Value Iteration Algorithm
def valueIteration():
    curr = [[ 0 for i in range(20)] for j in range(20)]
    prev = [[0 for i in range(20)] for j in range(20)]
    iters , i= 0,  0
    while iters < 1000:
        # calcualte values for all states
        for i in range(size):
            for j in range(size):
                world.state = (i,j) # increment state
                val = max(world.computeValues(prev)) # update value , make sure this didnt break
                curr[i][j] = round(val,2)
                #print(curr[i][j])
        # check convergence
        if iters != 1:
            delta = world.checkStop(curr, prev)
            # condition is to be small 10x in a row?
            if(delta < .001):
```

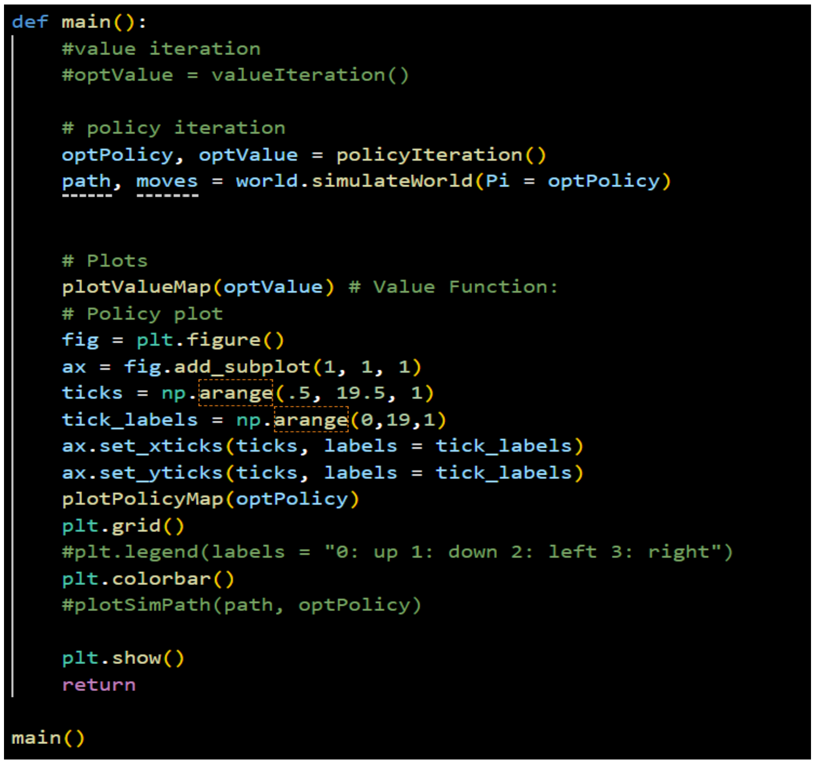


```python
                    print("iterations to converge:", iters)
                    break
        # update prevValues to currValues
        for i in range(size):
            for j in range(size):
                prev[i][j] = curr[i][j]
        iters += 1
    return curr

# policy evaluation , subroutine of policy iteration
def policyEvaluation(pi, currVal):
    delta, iters = 0, 0
    while (delta < .01) and iters < 1000:
        for i in range(0,size):
            for j in range(0,size):
                world.state = (i,j)
                a = pi[i][j]
                n = world.getNeighbors()
                v = currVal[i][j]
                currVal[i][j] = world.computePolicyStep(move = a, values = currVal, nbrs = n)
                delta = max(delta, abs(currVal[i][j] - v))
        iters += 1
    return currVal

#  Policy Iteration Algorithm:
def policyIteration():
    #V = value function to converge to optimal, Pi to converge on best policy
    empty = [[0 for i in range(20)] for j in range(20)]
    Pi = world.initPolicy(empty)
    V = [[0 for i in range(20)] for j in range(20)]
```



```python
    V = policyEvaluation(Pi, V) # init eval before improvement
    stable = False

    # improvement step:
    n = 0
    while (stable == False) and n < 100:
        stable = True
        for i in range(20):
            for j in range(20):
                world.state = (i,j)
                if (i,j) == world.goal:
                    Pi[i][j] = -1
                    continue
                best = np.argmax(world.computeValues(V))
                mxm = max(world.computeValues(V))
                if best != Pi[i][j]:
                    stable = False
                    Pi[i][j] = best
        if stable == False:
            V = policyEvaluation(Pi, V)
        else:
            print("State Space Sweeps: ", n)
        n += 1
    return Pi, V

```

- Skipping past some of the plotting functions

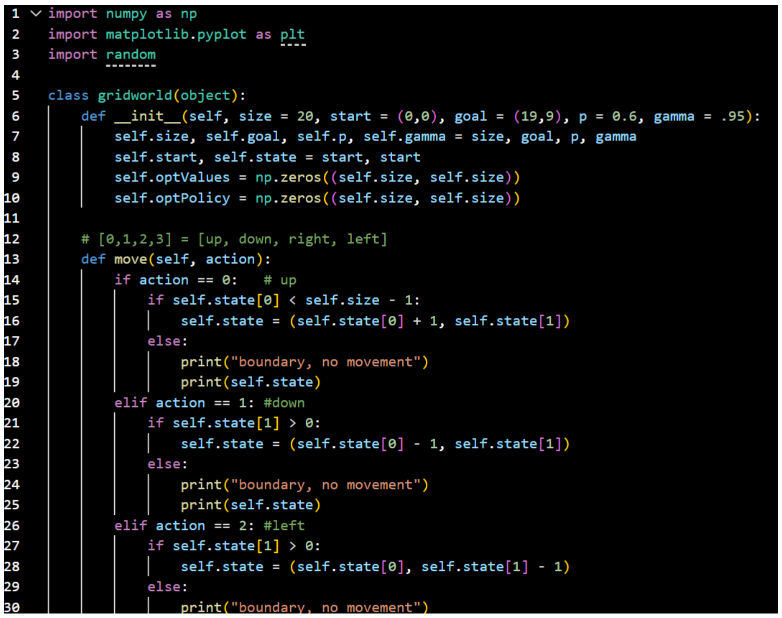


```python
def main():
    #value iteration
    #optValue = valueIteration()

    # policy iteration
    optPolicy, optValue = policyIteration()
    path, moves = world.simulateWorld(Pi = optPolicy)


    # Plots
    plotValueMap(optValue) # Value Function:
    # Policy plot
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ticks = np.arange(.5, 19.5, 1)
    tick_labels = np.arange(0,19,1)
    ax.set_xticks(ticks, labels = tick_labels)
    ax.set_yticks(ticks, labels = tick_labels)
    plotPolicyMap(optPolicy)
    plt.grid()
    #plt.legend(labels = "0: up 1: down 2: left 3: right")
    plt.colorbar()
    #plotSimPath(path, optPolicy)

    plt.show()
    return

main()
```

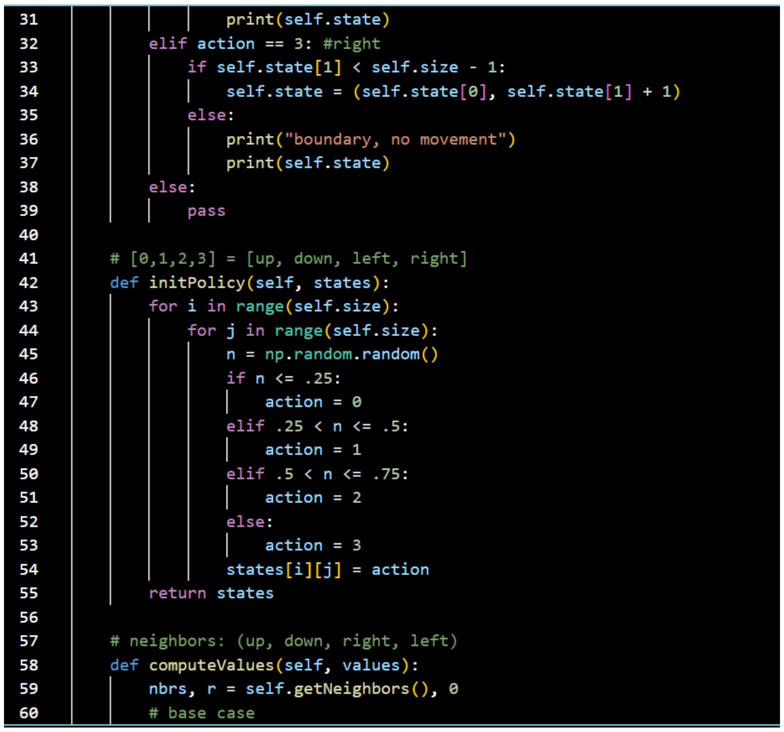
2nd File:

gridworld.py (class)

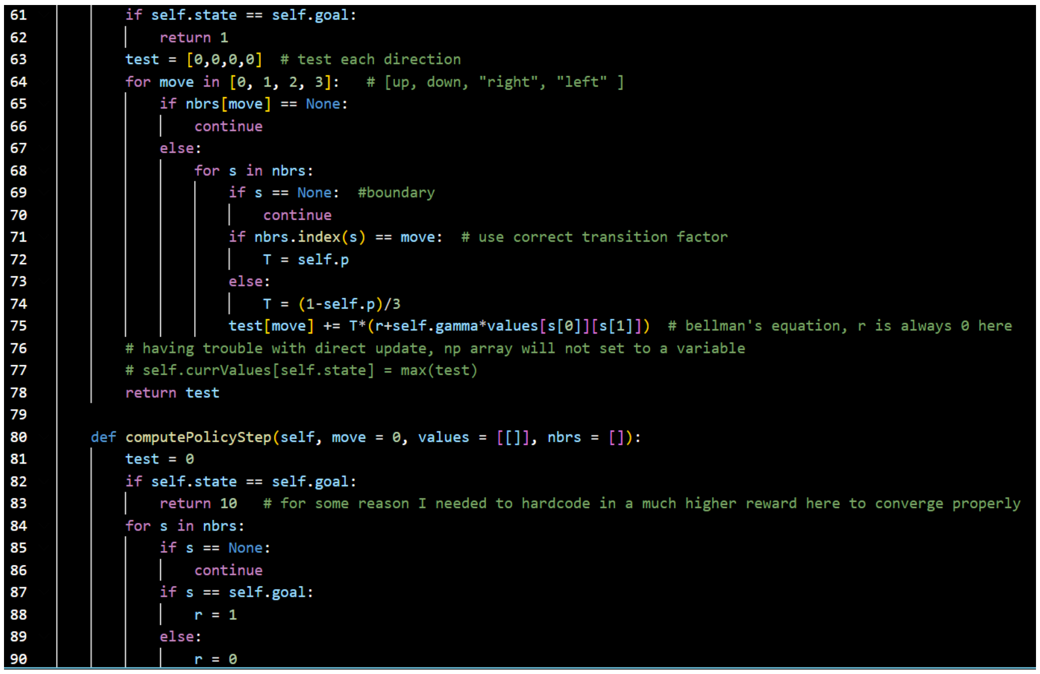
- **_init_** -- constructor
- **move()** -- given an action input, update the state of gridworld. If action is invalid (i.e boundary) do nothing
- **initPolicy()** – randomizes a starting policy configuration for the policy iteration algorithm (could use an initial guess to get better performance) Allows a choice between a random intial policy or directed one (see code above in part 3 for “biased” version
- **extractPolicy()** – using only an optimal value function as input, creates the optimal policy function by taking argmax of the nearest neighbors in each state
- **computeValues()** – computes bellman’s equation for all possible actions at a given state (returns all values and then we take argmax() in outer routine
- **computeStep()** -- computes bellman’s equation for a single constrained policy at a given state
- **checkStop()** -- checks a convergence criterion for value iteration algorithm
- **simulateWorld()** – given a policy function and initial state, simulates a trajectory. Built-in stopping condition in case the trajectory is periodic (infinite loop)
- **getNeighbors()** -- given a state, find the coordinates of each of its negihbors. If adjacent to boundary in some direction, returns None for that direction



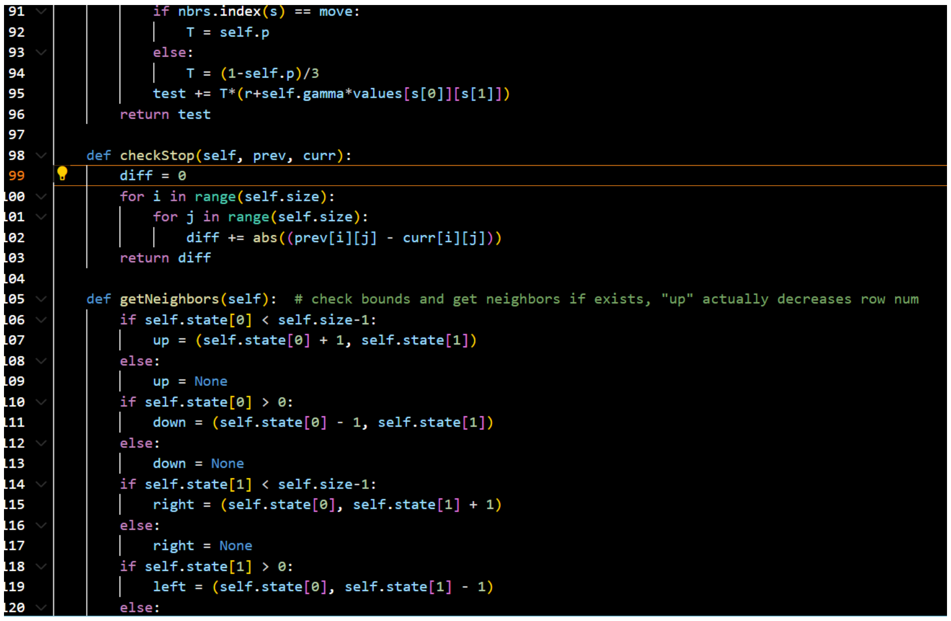
```python
import numpy as np
import matplotlib.pyplot as plt
import random

class gridworld(object):
    def __init__(self, size = 20, start = (0,0), goal = (19,9), p = 0.6, gamma = .95):
        self.size, self.goal, self.p, self.gamma = size, goal, p, gamma
        self.start, self.state = start, start
        self.optValues = np.zeros((self.size, self.size))
        self.optPolicy = np.zeros((self.size, self.size))

    # [0,1,2,3] = [up, down, right, left]
    def move(self, action):
        if action == 0:   # up
            if self.state[0] < self.size - 1:
                self.state = (self.state[0] + 1, self.state[1])
            else:
                print("boundary, no movement")
                print(self.state)
        elif action == 1: #down
            if self.state[1] > 0:
                self.state = (self.state[0] - 1, self.state[1])
            else:
                print("boundary, no movement")
                print(self.state)
        elif action == 2: #left
            if self.state[1] > 0:
                self.state = (self.state[0], self.state[1] - 1)
            else:
                print("boundary, no movement")
```



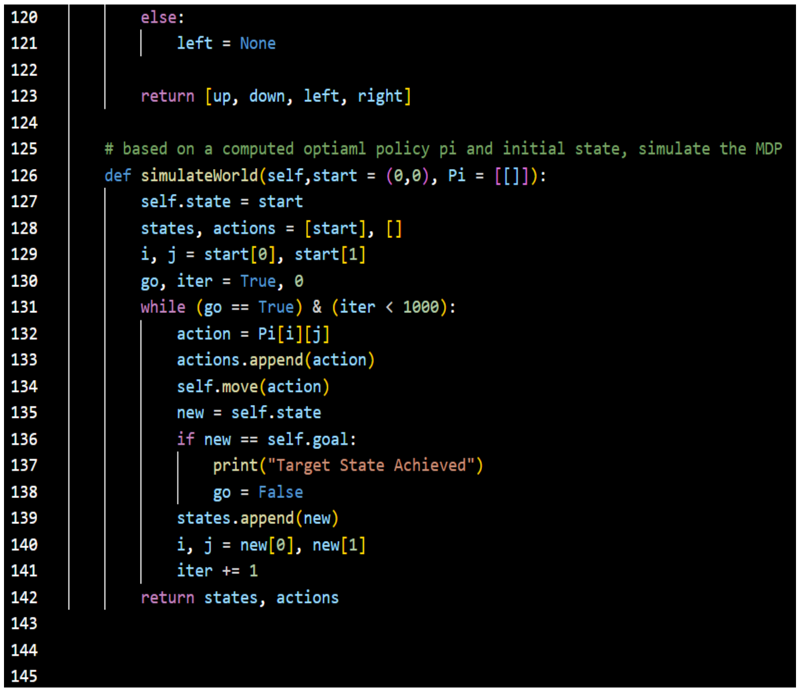
```python
                print(self.state)
        elif action == 3: #right
            if self.state[1] < self.size - 1:
                self.state = (self.state[0], self.state[1] + 1)
            else:
                print("boundary, no movement")
                print(self.state)
        else:
            pass

    # [0,1,2,3] = [up, down, left, right]
    def initPolicy(self, states):
        for i in range(self.size):
            for j in range(self.size):
                n = np.random.random()
                if n <= .25:
                    action = 0
                elif .25 < n <= .5:
                    action = 1
                elif .5 < n <= .75:
                    action = 2
                else:
                    action = 3
                states[i][j] = action
        return states

    # neighbors: (up, down, right, left)
    def computeValues(self, values):
        nbrs, r = self.getNeighbors(), 0
        # base case
```



```python
        if self.state == self.goal:
            return 1
        test = [0,0,0,0]  # test each direction
        for move in [0, 1, 2, 3]:   # [up, down, "right", "left" ]
            if nbrs[move] == None:
                continue
            else:
                for s in nbrs:
                    if s == None:  #boundary
                        continue
                    if nbrs.index(s) == move:  # use correct transition factor
                        T = self.p
                    else:
                        T = (1-self.p)/3
                    test[move] += T*(r+self.gamma*values[s[0]][s[1]])  # bellman's equation, r is always 0 here
        # having trouble with direct update, np array will not set to a variable
        # self.currValues[self.state] = max(test)
        return test

    def computePolicyStep(self, move = 0, values = [[]], nbrs = []):
        test = 0
        if self.state == self.goal:
            return 10   # for some reason I needed to hardcode in a much higher reward here to converge properly
        for s in nbrs:
            if s == None:
                continue
            if s == self.goal:
                r = 1
            else:
                r = 0
```

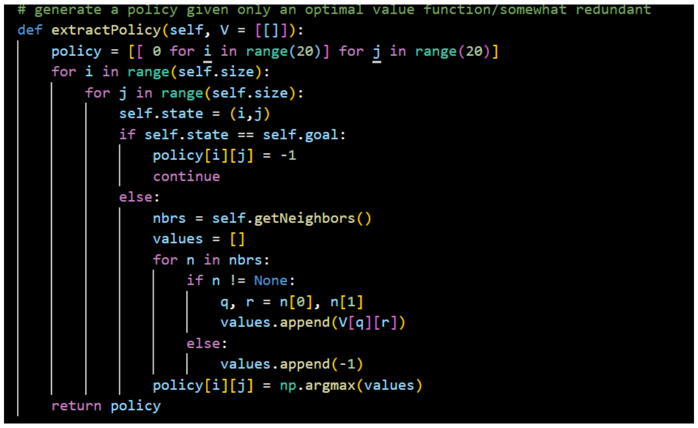


```python
            if nbrs.index(s) == move:
                T = self.p
            else:
                T = (1-self.p)/3
            test += T*(r+self.gamma*values[s[0]][s[1]])
        return test

    def checkStop(self, prev, curr):
        diff = 0
        for i in range(self.size):
            for j in range(self.size):
                diff += abs((prev[i][j] - curr[i][j]))
        return diff

    def getNeighbors(self):  # check bounds and get neighbors if exists, "up" actually decreases row num
        if self.state[0] < self.size-1:
            up = (self.state[0] + 1, self.state[1])
        else:
            up = None
        if self.state[0] > 0:
            down = (self.state[0] - 1, self.state[1])
        else:
            down = None
        if self.state[1] < self.size-1:
            right = (self.state[0], self.state[1] + 1)
        else:
            right = None
        if self.state[1] > 0:
            left = (self.state[0], self.state[1] - 1)
        else:
```



```python
        else:
            left = None

        return [up, down, left, right]

    # based on a computed optiaml policy pi and initial state, simulate the MDP
    def simulateWorld(self,start = (0,0), Pi = [[]]):
        self.state = start
        states, actions = [start], []
        i, j = start[0], start[1]
        go, iter = True, 0
        while (go == True) & (iter < 1000):
            action = Pi[i][j]
            actions.append(action)
            self.move(action)
            new = self.state
            if new == self.goal:
                print("Target State Achieved")
                go = False
            states.append(new)
            i, j = new[0], new[1]
            iter += 1
        return states, actions
```

(added the below function after the others)



```python
# generate a policy given only an optimal value function/somewhat redundant
def extractPolicy(self, V = [[]]):
    policy = [[ 0 for i in range(20)] for j in range(20)]
    for i in range(self.size):
        for j in range(self.size):
            self.state = (i,j)
            if self.state == self.goal:
                policy[i][j] = -1
                continue
            else:
                nbrs = self.getNeighbors()
                values = []
                for n in nbrs:
                    if n != None:
                        q, r = n[0], n[1]
                        values.append(V[q][r])
                    else:
                        values.append(-1)
                policy[i][j] = np.argmax(values)
    return policy
```