

Jack's Capitalism Optimization

Exercise 4.9: Original Problem

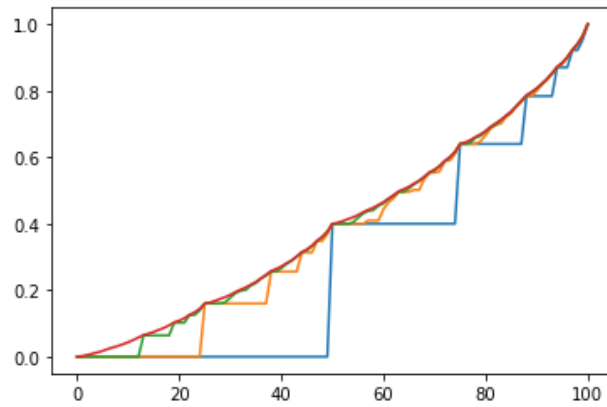


Figure 1: Value

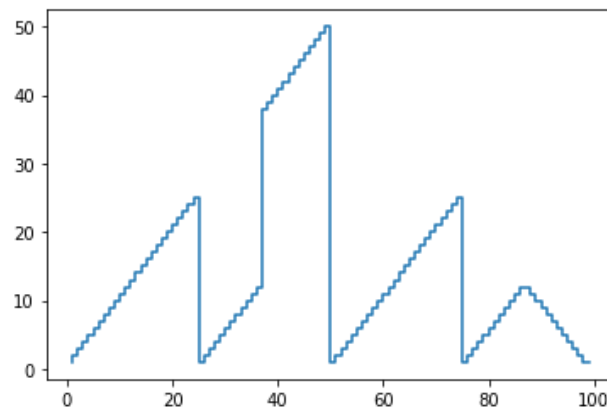


Figure 2: Policy

Exercise 4.9: $\text{ph} = 0.25$

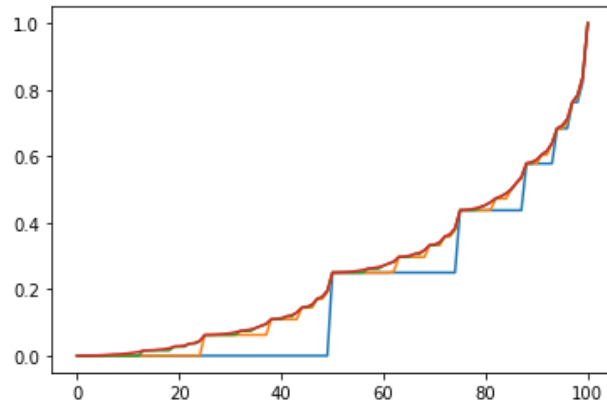


Figure 3: Value

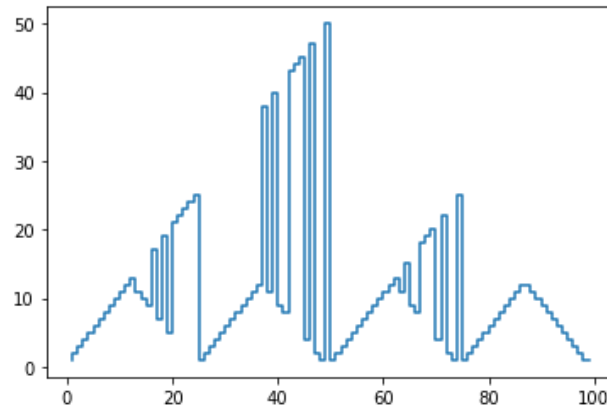


Figure 4: Policy

Exercise 4.9: $\text{ph} = 0.55$

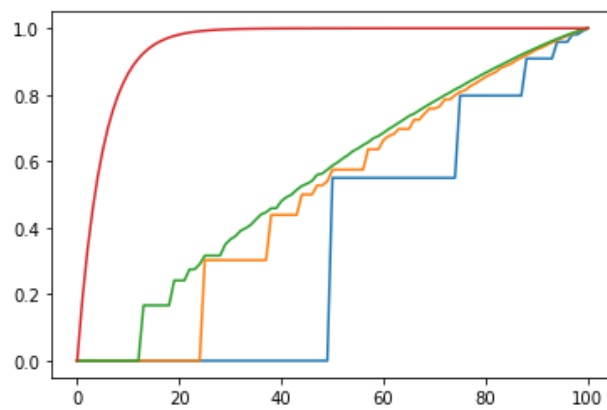


Figure 5: Value

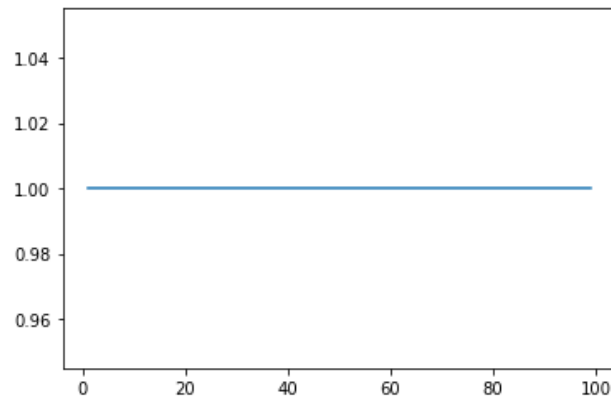


Figure 6: Policy

Appendix

Python Code for Problem 1

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Oct 23 09:06:23 2023
4
5 @author: justi
6 """
7
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import random
11
12 # Value Iteration for the Gambler's problem
13
14 # %% Gambler function
15
16 class Gambler:
17     def __init__(self, ph):
18         self.ph = ph
19         self.S = np.arange(1, 100)
20         self.V = np.zeros(101)
21         self.V[0] = 0
22         self.V[100] = 1
23         self.Vs = []
24         self.pi = None
25         self.sweep_count = None
26
27     def valueIteration(self):
28         self.sweep_count = 0
29         while True:
30             delta = 0
31             for s in self.S:
32                 v = self.V[s]
33                 self.V[s] = np.max([self.V_eval(s, a) for a in self.A(s)])
34                 delta = np.maximum(delta, abs(v - self.V[s]))
35             if self.sweep_count < 3:
36                 self.Vs.append(self.V.copy())
37             self.sweep_count += 1
38             if delta < 1E-10:
39                 break
40             print('Sweeps needed:', self.sweep_count)
41             self.Vs.append(self.V.copy())
42             self.pi = [self.A(s)[np.argmax([self.V_eval(s, a) for a in self.A(s)])] for s in
self.S]
43             plt.figure()
44             plt.plot(self.Vs[0])
45             plt.plot(self.Vs[1])
46             plt.plot(self.Vs[2])
47             plt.plot(self.Vs[3])
48             plt.figure()
49             plt.step(self.S, self.pi)
50
51     def A(self, s):
52         return np.arange(1, np.minimum(s, 100 - s) + 1)
53
54     def V_eval(self, s, a):
55         return 1 * self.V[s + a] * self.ph + 1 * self.V[s - a] * (1 - self.ph)
56
57 Orig = Gambler(0.4)
58 Orig.valueIteration()
59
60 Low = Gambler(0.25)
61 Low.valueIteration()
62
63 High = Gambler(0.55)
64 High.valueIteration()
```

Python Code for Problem 2

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Oct 18 10:14:27 2023
4
5 @author: just1
6 """
7
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import random
11 import math
12
13 states = [(x,y) for x in range(2) for y in range(5)]
14 policy = np.zeros((2,5), dtype=np.int8)
15 values = np.zeros((2,5), dtype=np.float16)
16 reward = np.array([500, 600, 700, 800, 1000])
17 prob = np.array([1/4, 1/4, 1/6, 1/6, 1/6])
18 gamma = 0.97
19
20 def sprimes(state, action):
21     bought = state[0]
22     price = state[1]
23
24     if bought == 0:
25         if action == 0:
26             sprimes = [(0,y) for y in range(5)]
27         elif action == 1:
28             sprimes = [(1,y) for y in range(5)]
29     elif bought == 1:
30         sprimes = [(1,y) for y in range(5)]
31
32     return sprimes
33
34 def probs(sprime, state, action):
35     bought = state[0]
36     price = state[1]
37     bought2 = sprime[0]
38
39     if bought + action == bought2:
40         probs = prob[price]
41     else:
42         probs = 0
43
44     return probs
45
46 def rewards(state, action):
47     bought = state[0]
48     price = state[1]
49
50     if bought == 0:
51         if action == 0:
52             rewards = -60
53         elif action == 1:
54             rewards = reward[price]
55     elif bought == 1:
56         rewards = 0
57
58     return rewards
59
60 def possibleActions(state):
61     bought = state[0]
62     price = state[1]
63
64     if bought == 0:
65         actions = [0,1]
66     else:
67         actions = [0]
68
69     return actions
70
```

```

71
72 def evalVal(state, action):
73     value = 0
74     psa = sprimes(state, action)
75
76     r = rewards(state, action)
77     for sp in range(len(psa)):
78         p = probs(psa[sp], state, action)
79         nv = values[psa[sp]]
80         value += p * ( r + gamma * nv )
81     return value
82
83 theta = 1E-5
84 maxIte = 10000
85 ite = 0
86 print("Init Values")
87 print(values)
88 print("Init Policy")
89 print(policy)
90
91 def policyEvaluation():
92     while True:
93         delta = 0
94         for s in range(len(states)):
95             b, p = states[s]
96             v = values[b,p]
97             #breakpoint()
98             values[b,p] = evalVal(states[s], policy[b,p])
99             delta = max(delta, abs(v - values[b,p]))
100         if delta < theta:
101             break
102
103 def policyImprovement():
104     vvalues = {a: evalVal(s,a) for a in possibleActions(s)}
105     bestActions = [a for a,value in vvalues.items() if value == np.max(list(vvalues.values()
106     ))]
107     #print(bestActions)
108     policy[s] = np.random.choice(bestActions)
109
110 while True:
111     print(f"ite: {ite}")
112     print(f"policy: {policy[0]}")
113     ite += 1
114     policyEvaluation()
115     print(f"values: {values[0]}")
116
117     policy_stable = True
118     for s in range(len(states)):
119         b, p = states[s]
120         old = policy[b,p].copy()
121         vvalues = {a: evalVal([b,p],a) for a in possibleActions([b,p])}
122         bestActions = [a for a,value in vvalues.items() if value == np.max(list(vvalues.
123         values()))]
124         #print(bestActions)
125         policy[b,p] = np.random.choice(bestActions)
126
127         if old != policy[b,p]:
128             policy_stable = False
129     if policy_stable:
130         break
131
132 print("Final Values")
133 print(values)
134 print("Final Policy")
135 print(policy)

```

Python Code for Problem 3

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Oct 18 10:14:27 2023
4
5 @author: just1
6 """
7
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import random
11 import math
12
13 states = np.arange(0,4)
14 policy = np.array([0,0,0,1])
15 values = np.zeros(4, dtype=np.float16)
16 prob = np.array([[0, 7/8, 1/16, 1/16],
17                  [0, 3/4, 1/8 , 1/8 ],
18                  [0,  0, 1/2 , 1/2 ],
19                  [0,  0,  0, 1   ]])
20 gamma = 0.95
21
22 rewards = np.array([[0, -1000, -3000,  0],
23                    [0,    0, -2000, -6000]])
24
25 def possibleActions(state):
26     if state == 2:
27         actions = [0,1]
28     elif state == 3:
29         actions = [1]
30     else:
31         actions = [0]
32     return actions
33
34 def sprimes(state, action):
35     if state == 2 and action == 1:
36         statePrimes = [1]
37     elif state == 3 and action == 1:
38         statePrimes = [0]
39     else:
40         statePrimes = [0,1,2,3]
41
42     return statePrimes
43
44 """
45 def evalVal(state, action):
46     value = 0
47     r = rewards[action, state]
48     for i in range(4):
49         p = prob[state, i]
50         value += p * (r + gamma * values[i])
51
52     return value
53 """
54
55 def evalVal(state, action):
56     value = 0
57     psa = sprimes(state, action)
58
59     r = rewards[action, state]
60     for sp in range(len(psa)):
61         p = prob[state, psa[sp]]
62         nv = values[psa[sp]]
63         value += p * ( r + gamma * nv )
64     return value
65
66 theta = 1E-2
67 maxIte = 10000
68 ite = 0
69 print("Init Values")
70 print(values)
```

```

71 print("Init Policy")
72 print(policy)
73
74 def policyEvaluation():
75     while True:
76         print(values)
77         delta = 0
78         for s in range(len(states)):
79             v = values[s]
80             values[s] = evalVal(states[s], policy[s])
81             delta = max(delta, abs(v - values[s]))
82         if delta < theta:
83             break
84
85 #policyEvaluation()
86
87 while True:
88     print(f"ite: {ite}")
89     print(f"policy: {policy}")
90     ite += 1
91     policyEvaluation()
92     print(f"values: {values}")
93
94     policy_stable = True
95     for s in range(len(states)):
96         old = policy[s].copy()
97         vvalues = {a: evalVal(s,a) for a in possibleActions(s)}
98         bestActions = [a for a,value in vvalues.items() if value == np.max(list(vvalues.
99         values()))]
100         policy[s] = np.random.choice(bestActions)
101
102         if old != policy[s]:
103             policy_stable = False
104     if policy_stable:
105         break
106
107 print("Final Values")
108 print(values)
109 print("Final Policy")
110 print(policy)

```


Python Code for Problem 4

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Oct 23 23:12:10 2023
4
5 @author: justi
6 """
7
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import random
11 import math
12
13 states = np.arange(3)
14 policy = np.array([2,2,2])
15 values = np.zeros(3, dtype=np.float16)
16
17 gamma = 0.95
18
19 actions = np.array([0,1,2])
20
21 """
22 def evalVal(state, action):
23     value = 0
24     r = rewards[action, state]
25     for i in range(4):
26         p = prob[s, i]
27         value += p * (r + gamma * values[i])
28
29     return value
30 """
31
32 def evalVal(state, action):
33     value = 0
34     sprime = state + action
35     cost = action * 5 #+ 10*np.sign(action)
36
37     for i in range(3):
38         nv = sprime-i
39
40         if nv >= 0 and nv <= 2:
41             r = -4 * nv
42             nvalue = values[nv]
43         elif nv == -1:
44             r = -8
45             nvalue = 0
46         elif nv == -2:
47             r = -40
48             nvalue = 0
49         elif nv > 2:
50             r = -8
51             nvalue = values[2]
52
53         value += -cost + 1/3 * ( r + gamma * nvalue )
54     return value
55
56 theta = 1E-2
57 maxIte = 10000
58 ite = 0
59 print("Init Values")
60 print(values)
61 print("Init Policy")
62 print(policy)
63
64 def policyEvaluation():
65     while True:
66         print(values)
67         delta = 0
68         for s in range(len(states)):
69             v = values[s]
70             values[s] = evalVal(states[s], policy[s])
```

```

71         delta = max(delta, abs(v - values[s]))
72         if delta < theta:
73             break
74
75 #policyEvaluation()
76
77 while True:
78     print(f"ite: {ite}")
79     print(f"policy: {policy}")
80     ite += 1
81     policyEvaluation()
82     print(f"values: {values}")
83
84     policy_stable = True
85     for s in range(len(states)):
86         old = policy[s].copy()
87         vvalues = {a: evalVal(s,a) for a in [0,1,2]}
88         bestActions = [a for a,value in vvalues.items() if value == np.max(list(vvalues.
values()))]
89         policy[s] = np.random.choice(bestActions)
90
91         if old != policy[s]:
92             policy_stable = False
93     if policy_stable:
94         break
95
96 print("Final Values")
97 print(values)
98 print("Final Policy")
99 print(policy)

```

EXERCISE 4.9 (PROGRAMMING EXERCISE)

GAMBLER'S PROBLEM for $P_h = 0.25$ & $P_h = 0.55$

stable if $\theta \rightarrow 0$?

EXERCISE 5.6 WHAT IS THE EQUATION ANALOGOUS TO (5.6) FOR ACTION VALUES $Q(s,a)$

INSTEAD OF STATE VALUES $V(s)$, AGAIN GIVEN RETURNS GENERATED USING B ?

not sure but now instead of

$$V(s) = \frac{\sum_{t \in T(s)} p_t: T(t) - 1 G_t}{\sum_{t \in T(s)} p_t: T(t) - 1}$$

$\tau(s)$ now becomes $\tau(s,a)$

$$Q(s,a) = \frac{\sum_{t \in \tau(s,a)} p_t: T(t) - 1 G_t}{\sum_{t \in \tau(s,a)} p_t: T(t) - 1}$$

EXERCISE 5.8 THE RESULTS W/ EX. 5.5 & SHOWN IN FIG 5.4 USED A 1^{st} VISIT MC METHOD.
SUPPOSE THAT INSTEAD AN EVERY-VISIT MC METHOD WAS USED ON THE SAME
PROBLEM, WOULD THE VARIANCE OF THE ESTIMATOR STILL BE INFINITE?
WHY OR WHY NOT?

estimator will still be infinite because reward is at terminal still.
& the visits to the state will still run to infinity
expected reward at every state = 1

Problem 3 A MANUFACTURER RELIES ON ONE KEY MACHINE. DUE TO HEAVY USE, THE MACHINE DETERIORATES RAPIDLY. AT THE END OF EACH WEEK A THOROUGH INSPECTION IS DONE THAT CLASSIFIES THE MACHINE INTO ONE OF FOUR POSSIBLE STATES.

0 - GOOD AS NEW

1 - OPERABLE MINOR DETERIORATION

2 - OPERABLE MAJOR DETERIORATION

3 - INOPERABLE

WITHOUT ANY REPAIRS THE STATE OF THE MACHINE EVOLVES AS A MARKOV CHAIN W/ A TRANSITION MATRIX

$$P = \begin{bmatrix} 0 & 7/8 & 1/16 & 1/16 \\ 0 & 3/4 & 1/8 & 1/8 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

IF IN STATE 3, MANUFACTURER REPLACES MACHINE & COSTS 6000\$

IF IN STATE 1 & 2 \$1000/week in 1

\$3000/week in 2

repair cost

USING DISCOUNT FACTOR OF $\gamma = 0.95$ FIND OPTIMAL POLICY VIA POLICY ITERATION

Stages 4 months
infinite problem hence discount

States: $\{0, 1, 2, 3\}$

actions: $\{\text{continue}, \text{repair}\}$

Rewards $\begin{Bmatrix} 0 & -1000 & -3000 & \text{X} \\ 0 & 0 & -2000 & -6000 \end{Bmatrix}$

Dynamics of repair

convert to a solvable linear system for Policy Evaluation

initialize w/ $\{\text{continue}, \text{continue}, \text{continue}, \text{repair}\}$

only thing possible to change

to do policy iteration

start w/ policy evaluation first

from code this evaluates to $[-5187, -5432, -5712, 0]$

using policy improvement next

$\{\text{continue}, \text{continue}, \text{repair}, \text{repair}\}$

& new values = $[-2892, -3480, 0, 0]$

Final policy	
0	continue
1	continue
2	repair
3	repair

PROBLEM 4 CONSIDER AN INFINITE-PERIOD INVENTORY SYSTEM w/ A SINGLE PRODUCT WHERE, AT THE BEGINNING OF EACH PERIOD, A DECISION IS TO BE MADE ABOUT HOW MANY ITEMS TO PRODUCE DURING THAT PERIOD. THE SETUP COST IS \$10 & THE UNIT PRODUCTION COST IS \$5. THE HOLDING COST FOR EACH ITEM NOT SOLD DURING IS \$4, AND A MAXIMUM OF 2 ITEMS CAN BE STORED. DURING EACH PERIOD, DEMAND IS 0, 1, 2 ITEMS EACH WITH PROBABILITY $\frac{1}{3}$. IF DEMAND EXCEEDS THE SUPPLY AVAILABLE DURING THAT PERIOD, THOSE SALES ARE LOST AND A SHORTAGE COST IS INCURRED { 1 UNIT: \$8
2 UNITS: \$32

$K=0.95$, Get optimal policy using policy iteration

$$X_{k+1} = X_k - d_k + \text{prod}_{0,1,2}$$

STATES: # of items in inventory {2, 1, 0}

ACTIONS: PRODUCE {0, 1, 2} use code

initialize value function = (0, 0, 0)

initialize policy: produce (2, 2, 2)

policy evaluation:

not sure about code but final policy is

{0, 1, 2}

produce 0, 0, 0

b) using policy iteration, see code attached.

Started w/ $V = \{0, 0, 0, 0, 0\}$

& $\Pi = (\text{wait}, \text{wait}, \text{wait}, \text{wait}, \text{wait})$

1st iteration policy evaluation: $[-2476, -2476, -1651, -1651, -1651]$
policy improvement $[\text{sell}, \text{sell}, \text{sell}, \text{sell}, \text{sell}]$

2nd iteration policy evaluation: $[625, 750, 583.5, 666.5, 833.5]$
policy improvement = $[\text{wait}, \text{wait}, \text{sell}, \text{sell}, \text{sell}]$
converged

best policy = \$500	wait
\$600	wait
\$700	sell
\$800	sell
\$900	sell