Submitted by: Clifton Roozendal (*2534824*) and Venkat Mohit Sornapudi (*2721697*)

Assignment – 2

Introduction

The assignment has two Markov chain models to be solved. One for Parts 1 and 2 (shown in *Fig 1* in Appendix) and another for Part 3 (shown in *Fig 2* in Appendix). In both the models, State 0 represents new system, and the next consecutive states (State 1 to State 90) represent the consecutive deteriorated states of the system. At State 0, the probability of system failure is 0.1 and this probability increases by 0.01 for the next consecutive states. When the system fails, it must be replaced (forcing it to go to State 0) at a cost of 1 unit. In case of Part 3, a preventive replacement can be done at any state at a cost of 0.5 units.

Part 1: Finding the stationary distribution by recursion

Here, we defined *P, r* as numpy arrays that represent transition probabilities and rewards obtained (cost is represented as negative reward) as shown in *Fig 3, 4* respectively in Appendix. *P* and *r* are of the shapes 91x91 and 1x91 respectively. We initialized *pi*, represents the probability distribution of the system being at State 0, with 1x91 array ([1,0,0,…till 91 terms]).

To find the stationary distribution of the system, we did recursion on time as follows (@ represents matrix multiplication):

$$pi_{t+1} = pi_t \ @ \ P$$

Thus, we get new *pi* value after each step. The recursion stopped when the *pi* value converges (deviates by a very small amount). The obtained limiting *pi* is shown in *Fig 5* in Appendix. We then found the long-run average reward (*phi*) to be - 0.1461, by performing the following operation with the resultant *pi* (▪ represents dot product):

$$phi = \ pi \blacksquare \ r$$

Part 2: The average-cost Poisson equation

The Poisson equation in vector form is expressed as (@ represents matrix multiplication, *e* represents column vector of ones of similar shape as *r*, and $*$ represents scalar-vector multiplication):

$$V \ + \ phi * e \ = \ r \ + P \ @ V$$

Here, we used P, r that were defined in Part 1. To solve the system of equations we used numpy.linalg.solve method. But first, to use this method, we collected coefficients of *V* and *phi*. Note that we have 92 unknowns i.e., *V(0), V(1), … V(90), phi* and 91 equations. So, we set *V(0)* as zero and excluded it in coefficients of *V* and *phi* matrix since *V(0)* has no significance in the equation once it is substituted. Finally, after solving for *V* values we appended *V(0)* (=0).

We implemented the following python function to solve the Poisson equation:

```
def solve_poisson(r, P):

  V_coeff = np.identity(91)-P
  V_coeff_and_phi_coeff = np.c_[V_coeff, np.ones((91))]

  # set V(0) as zero so excluded first row in calculation
  V_and_phi = np.linalg.solve(V_coeff_and_phi_coeff[:,1:], r.T)

  # limiting values i.e. V*(1),V*(2)... V*(90)
  V_1_to_90 = V_and_phi[:-1]
  V_0_to_90 = np.append(0, V_1_to_90)
  phi = V_and_phi[-1]

  return V_0_to_90, phi
```

Thus, the obtained limiting average replacement cost (*phi*) is 0.1461 (same as in Part 1). The obtained *V* values are presented in *Fig 6* in the appendix.

---

* *np stands for numpy (a python library)*

Part 3: Policy iteration and Value iteration

We initialized two 91x91 matrices, *P_action0* and *P_action1*. The matrix *P_action0* is the same as matrix *P* which was used in Parts 1,2 (*Fig 3*). The matrix *P_action1* has the first column ones and zeros in the rest of the matrix. We also initialized two vectors of length 91 named *r_action0* and *r_action1*. The array named *r_action0* is the same as array *r* that was used in Parts 1,2 (*Fig 4*). The array named r_action1 has the value 0.5 in all positions.

Policy iteration

For the optimal policy iteration, we used the following steps:

1. Fix a policy α
2. Find a solution $(V_\alpha, phi_\alpha)\ of\ V\ +\ phi\ =\ r_\alpha + P_\alpha V$
3. $\alpha\tilde{}\ =\ arg\ max\{r_\alpha,\ +\ P_\alpha, V_\alpha\}$
4. If $\alpha\ =\ \alpha\tilde{}$: terminate with $(V_\alpha,, phi_\alpha)$ optimal. Else $\alpha\ =\ \alpha\tilde{}$ and go back to step 2.

The above steps are implemented by the following python loop:

```
for i in range(91): # loop to compare actions at each state (reference state)

  r = np.zeros((91))
  P = np.zeros((91.91))

  for j in range(91): # constructing vector r and matrix P

       if best_actions[j] ==0:
         r[j] = r_action0[j]
         P[j] = P_action0[j]
       else: # if action 1 is found to be the best action in previous iterations
               (for states before the reference state)
         r[j] = r_action1[j]
         P[j] = P_action1[j]

  # V's for the present combination of actions
    (got after updations to the initialized combination from previous iterations)
  V_0_to_90, _ = solve_poisson(r,P)

  action0_value = r_action0 + P_action0 @ V_0_to_90
  action1_value = r_action1 + P_action1 @ V_0_to_90
  action_values = np.vstack((action0_value,action1_value))
  best_actions = np.argmin(action_values,axis=0)

  if i == 90:
       _, policy_phi = solve_poisson(r,P)
```

In the above code, we initialized a policy by taking action 0 at all states. We then improved this policy by comparing $r_\alpha,\ +\ P_\alpha, V_\alpha$ values for action 0 and action 1 at each state. For doing so, we solved Poisson equations by the function constructed in Part 2 and obtained *V* values for each policy.

We thus obtained the average long term replacement cost as 0.14491. This result is lower compared to the result obtained in parts 1,2. We also observe that from state 13 onwards the optimal action is 1 i.e. preventive replacement. All the other results obtained are displayed in *Fig 7* in Appendix.

Value iteration

For the average optimal value iteration, we used the following formula: $V_{t+1}\ =\ max\{r_a\ +\ P\}$.

We implemented the optimal value iteration as follows:

```
V = np.zeros((91))
for i in range(91):
  V_action0 = r_action0 + P_action0@V
  V_action1 = r_action1 + P_action1@V
  V_decision = np. minimum(V_action0.V_action1)
  if i == 90:
      Value_phi = (V_decision - V)
  V = V_decision
```

In the above code, we initialized *V* with a vector of zeros of size 91. We looped over all the states and compared $r_a\ +\ P$ values for action 0, action 1 and taken the minimum of them as the new *V* vector. The obtained new *V* vector is used in place of V in the subsequent iterations of the loop.

As obtained in policy iteration, we obtained the same average long term replacement cost i.e. 0.14491. All the other results obtained are displayed in *Fig 8* in Appendix.

Submitted by: Clifton Roozendal (*2534824*) and Venkat Mohit Sornapudi (*2721697*)

Appendix

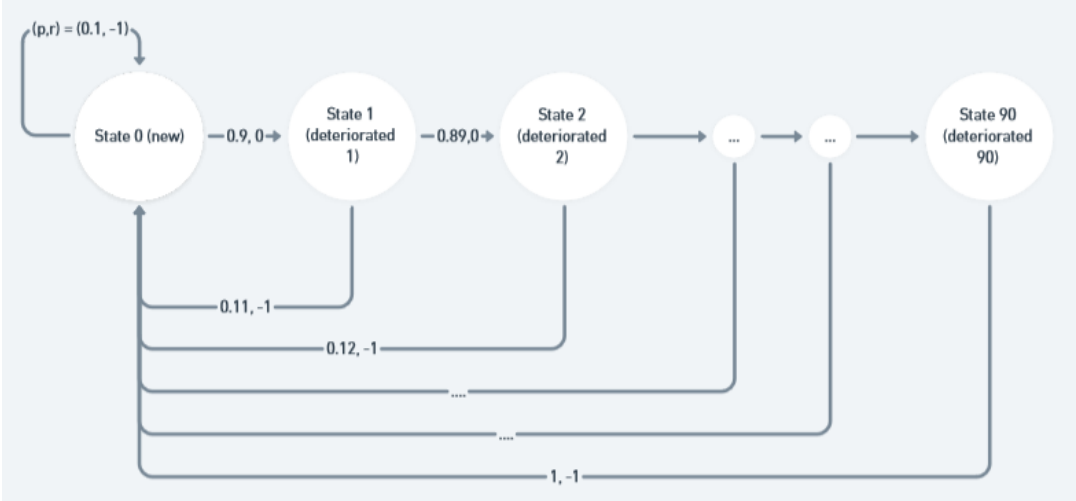

Fig 1. Stochastic model of the system given for Parts 1 and 2

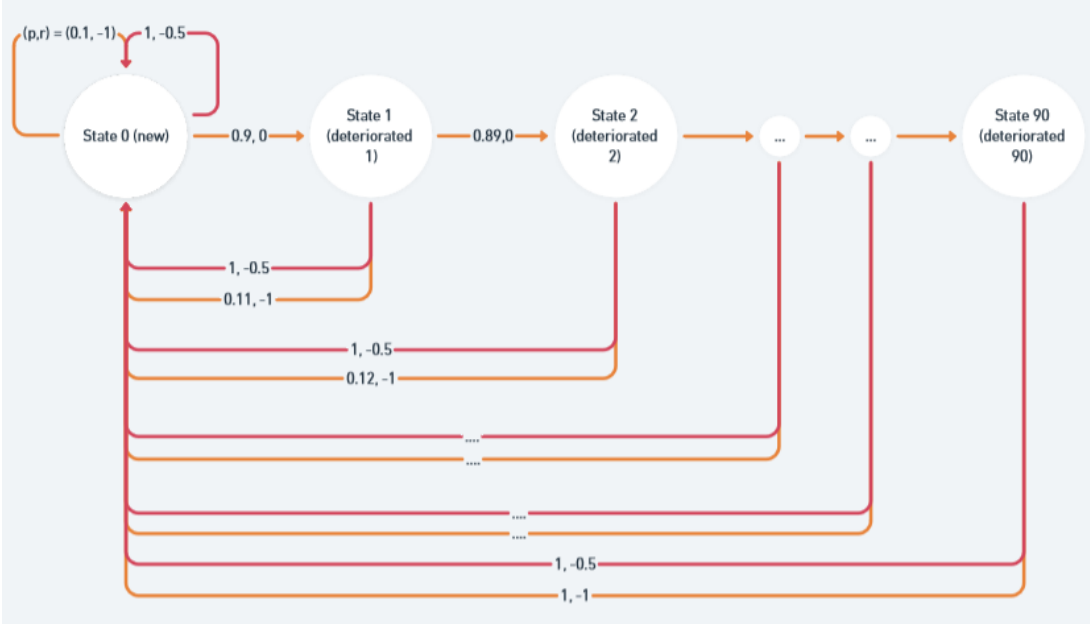

Fig 2. Stochastic model of the system given for Part 3. Orange lines represent action 0. Red lines represent action 1.

	0	1	2	3	4	5	6	7	8	9	...	81	82	83	84	85	86	87	88	89	90
0	0.10	0.9	0.00	0.00	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.00	0.00
1	0.11	0.0	0.89	0.00	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.00	0.00
2	0.12	0.0	0.00	0.88	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.00	0.00
3	0.13	0.0	0.00	0.00	0.87	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.00	0.00
4	0.14	0.0	0.00	0.00	0.00	0.86	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.00	0.00
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
86	0.96	0.0	0.00	0.00	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.04	0.00	0.00	0.00
87	0.97	0.0	0.00	0.00	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.03	0.00	0.00
88	0.98	0.0	0.00	0.00	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.02	0.00
89	0.99	0.0	0.00	0.00	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.00	0.01
90	1.00	0.0	0.00	0.00	0.00	0.00	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.00	0.00

Fig 3. P

```
array([[-1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000]])
```

Fig 4. r

```
array([[0.1461, 0.1315, 0.1170, 0.1030, 0.0896, 0.0771, 0.0655, 0.0550,
        0.0457, 0.0374, 0.0303, 0.0243, 0.0192, 0.0150, 0.0115, 0.0087,
        0.0066, 0.0049, 0.0035, 0.0026, 0.0018, 0.0013, 0.0009, 0.0006,
        0.0004, 0.0003, 0.0002, 0.0001, 0.0001, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000]]))
```

Fig 5. Limiting pi

	0
0	0.948781
1	0.901891
2	0.858856
3	0.819263
4	0.782750
...	...
85	0.152120
86	0.150569
87	0.149049
88	0.147559
89	0.146098

Fig 6. V(1) (at index 0) to V(90) (at index 89), *(V(0) =0)*

```
best actions:
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
preventive actions are taken in states:
 [[13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
  37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
  61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
  85 86 87 88 89 90]]
Policy iteration Phi:  0.14491799482409004
```

Fig 7. Outputs of policy iteration

```
Final V's:
 [13.0229 13.0728 13.1182 13.1596 13.1971 13.2312 13.2620 13.2895 13.3139
 13.3350 13.3526 13.3661 13.3750 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780 13.3780
 13.3780]
Value iteration phi:  0.1449179984802651
```

Fig 8. Outputs of value iteration