Problem 1.

Solution:

(1). State space: $X = \{0, 1, 2, \cdots\}$ state of equipment $x = 0$ – new.

Control space: $U = \{0, 1\}$ 0 - wait; 1 – replace.

Reward function for one period: $r(x,u) = \begin{cases} R - K(1 - \gamma e^{-\mu x}) - c_0 & u = 1 \\ R - (c_0 + c_1 x) & u = 0 \end{cases}$

Dynamic programming equation:

$$v_\gamma{}^*(x) = \max\{r(x,1) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(j);\ r(x,0) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(x+j)\}$$

(2) If there is no salvage value, the reward function for one period is:

$$r(x,u) = \begin{cases} R - K - c_0 & u = 1 \\ R - (c_0 + c_1 x) & u = 0 \end{cases}$$

So the $r(x,u)$ is non-increasing because $c_1 > 0$ in the case $u = 0$; and $R - K - c_0$ is a constant in the case $u = 1$.

Further, p_j and $\gamma(0 < \gamma < 1)$ are constant and do not depend on x. The function $v_\gamma{}^*(\cdot) v_\gamma{}^*(\cdot + j)$ are non-increasing with respect to x, by induction assumption.

Denote

$$h(x) = r(x,1) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(j),$$

$$g(x) = r(x,0) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(x+j),$$

We conclude that $h(\cdot)$ and $g(\cdot)$ are non-increasing because they are sums of non-increasing functions.

Thus, for $x > y$

$$\max\{h(x);\ g(x)\} \le \max\{h(y);\ g(y)\}$$

We conclude that $v_\gamma{}^*(x)$ is non-increasing.

(3) The reward function for one period is $r(x,u) = \begin{cases} -6 + 8e^{-0.2x} & u = 1 \\ 4 + x & u = 0 \end{cases}$,

The probabilities of deterioration by j steps in one period are given by Poisson distribution:

$$p_j = \frac{1}{j!\, e}$$

Value iteration:

$$\begin{cases} v^*(0) = max\{2 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots, 4 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots\} \\ v^*(1) = max\{0.55 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots, 5 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots\} \\ v^*(2) = \max\{-0.64 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots, 6 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots\} \\ v^*(3) = \max\{-2.41 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots, 7 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots\} \\ v^*(4) = \max\{-3.06 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots, 8 + 0.37v^*(0) + 0.37v^*(1) + 0.18v^*(2) \cdots\} \\ \cdots \end{cases}$$

```
MATLAB:
% [v_lo,n_it] = dne1_value_iteration_revised (0.9,10000);
function [v_lo,n_it] = dne1_value_iteration_revised (alpha,max_it)
i = 0;
n_it = max_it;
v=[0,0,0,0,0];
vv=[0,0,0,0,0];
v_lo=[0,0,0,0,0];
v_up=[0,0,0,0,0];
while (i < n_it)
    vv(1) =
max(2+alpha*(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)),4+alpha
*(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)));
    vv(2) =
max(0.55+alpha*(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)),5+al
pha*(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)));
    vv(3) = max(-
0.64+alpha*(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)),6+alpha*
(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)));
    vv(4) = max(-
2.41+alpha*(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)),7+alpha*
(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)));
    vv(5) = max(-
3.06+alpha*(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)),8+alpha*
(0.37*v(1)+0.37*v(2)+0.18*v(3)+0.06*v(4)+0.02*v(5)));
    v_lo = vv + min(vv-v)*alpha/(1-alpha);
    v_up = vv + max(vv-v)*alpha/(1-alpha);
    if (isequal(v_lo,v_up))
        n_it=i;
    end
    i=i+1;
    v=v_lo;
end
end
```

```
>> dne1_value_iteration_revised(0.9, 10000)

ans =

   48.9100    49.9100    50.9100    51.9100    52.9100
```

Policy iteration:

In [9]:

```
def policy_evaluation(self):
        next_value_table = [[0.00] * self.env.width
                                    for _ in range(self.env.heig
ht)]
        # Bellman Expectation Equation for the every states
        for state in self.env.get_all_states():
            value = 0.0
            for action in self.env.possible_actions:
                next_state = self.env.state_after_action(state,
action)
                reward = self.env.get_reward(state, action)
                next_value = self.get_value(next_state)
                value += (self.get_policy(state)[action] *
                          (reward + self.discount_factor * next_
value))
            next_value_table[state[0]][state[1]] = round(value,
2)
        self.value_table = next_value_table
```

In [14]:

```
def policy_improvement(self):
        next_policy = self.policy_table
        for state in self.env.get_all_states():
            value = -99999
            max_index = []
            result = [0.0, 0.0, 0.0, 0.0,0.0]  # initialize the
policy

            # for every actions, calculate
            # [reward + (discount factor) * (next state value fu
nction)]
            for index, action in enumerate(self.env.possible_act
ions):
                next_state = self.env.state_after_action(state,
action)
                reward = self.env.get_reward(state, action)
                next_value = self.get_value(next_state)
                temp = reward + self.discount_factor * next_valu
e
                if temp == value:
                    max_index.append(index)
                elif temp > value:
                    value = temp
                    max_index.clear()
                    max_index.append(index)

            # probability of action
            prob = 1 / len(max_index)

            for index in max_index:
                result[index] = prob

            next_policy[state[0]][state[1]] = result

        self.policy_table = next_policy
```

Linear programming:

```
% v = dne1_LP
function v = dne1_LP
clear();
f=[1;1;1;1;1];

A=[0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02;
   0.37,0.37,0.18,0.06,0.02];

b=[2;4;0.55;5;-0.64;6;-2.41;7;-3.06;8];

v=linprog(f,A,b);

end
```

Problem 2.

Solution:

(1) State space: $X = \{x_t, t = 0, 1, 2, \cdots | x_{t+1} = \max(0, x_t + u_t - d_t)\}$

Control space: $U = \{u_t, t = 0, 1, 2, \cdots | 0 \le u_t \le 12, u_t \in Z\}$

Dynamic programming equation:

$$v^*(x) = \max_{u \ge 0}\{-5u - 2(x+u) + E[10 \times min(x+u, d) + 0.8v^*(x+u-d)_+]\}$$

(2) value iteration:

```
% [v_lo,n_it] = dne1_value_iteration_revised (0.8,10000);
function p=p(~)
   alphabet=[0,1,2,3,4];
   prob= [0.1,0.2,0.3,0.2,0.2];
   p=randsrc(1,1,[alphabet;prob]);
end
function [v_lo,n_it] = dne1_value_iteration_revised (alpha,max_it)
i = 0;
n_it = max_it;
v=[0,0,0,0,0,0,0,0,0,0,0,0,];
```

```
vv=[0,0,0,0,0,0,0,0,0,0,0,];
v_lo=[0,0,0,0,0,0,0,0,0,0,0,];
v_up=[0,0,0,0,0,0,0,0,0,0,0,];
while (i < n_it)
    vv(1) =
max(2+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)+p*
v(9)+p*v(10)+p*v(11)+p*v(12)),4+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(
5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(2) =
max(0.55+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)
+p*v(9)+p*v(10)+p*v(11)+p*v(12)),5+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p
*v(5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(3) =
max(2+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)+p*
v(9)+p*v(10)+p*v(11)+p*v(12)),4+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(
5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(4) =
max(0.55+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)
+p*v(9)+p*v(10)+p*v(11)+p*v(12)),5+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p
*v(5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(5) =
max(2+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)+p*
v(9)+p*v(10)+p*v(11)+p*v(12)),4+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(
5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(6) =
max(0.55+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)
+p*v(9)+p*v(10)+p*v(11)+p*v(12)),5+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p
*v(5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(7) =
max(2+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)+p*
v(9)+p*v(10)+p*v(11)+p*v(12)),4+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(
5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(8) =
max(0.55+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)
+p*v(9)+p*v(10)+p*v(11)+p*v(12)),5+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p
*v(5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(9) =
max(2+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)+p*
v(9)+p*v(10)+p*v(11)+p*v(12)),4+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(
5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(10) =
```

```
max(0.55+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)
+p*v(9)+p*v(10)+p*v(11)+p*v(12)),5+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p
*v(5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(11) =
max(2+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)+p*
v(9)+p*v(10)+p*v(11)+p*v(12)),4+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(
5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    vv(12) =
max(0.55+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p*v(5)+p*v(6)+p*v(7)+p*v(8)
+p*v(9)+p*v(10)+p*v(11)+p*v(12)),5+alpha*(p*v(1)+p*v(2)+p*v(3)+p*v(4)+p
*v(5)+p*v(6)+p*v(7)+p*v(8)+p*v(9)+p*v(10)+p*v(11)+p*v(12)));
    v_lo = vv + min(vv-v)*alpha/(1-alpha);
    v_up = vv + max(vv-v)*alpha/(1-alpha);
    if (isequal(v_lo,v_up))
        n_it=i;
    end
    i=i+1;
    v=v_lo;
end
end
```

Policy iteration:

In [9]:

```
def policy_evaluation(self):
        next_value_table = [[0.00] * self.env.width
                                    for _ in range(self.env.heig
ht)]
        # Bellman Expectation Equation for the every states
        for state in self.env.get_all_states():
            value = 0.0
            for action in self.env.possible_actions:
                next_state = self.env.state_after_action(state,
action)
                reward = self.env.get_reward(state, action)
                next_value = self.get_value(next_state)
                value += (self.get_policy(state)[action] *
                          (reward + self.discount_factor * next_
value))
            next_value_table[state[0]][state[1]] = round(value,
2)
        self.value_table = next_value_table
```

In [14]:

```
def policy_improvement(self):
        next_policy = self.policy_table
        for state in self.env.get_all_states():
            value = -99999
            max_index = []
            result = [0.0, 0.0, 0.0, 0.0,0.0]  # initialize the
policy

            # for every actions, calculate
            # [reward + (discount factor) * (next state value fu
nction)]
            for index, action in enumerate(self.env.possible_act
ions):

                next_state = self.env.state_after_action(state,
action)
                reward = self.env.get_reward(state, action)
                next_value = self.get_value(next_state)
                temp = reward + self.discount_factor * next_valu
e
                if temp == value:
                    max_index.append(index)
                elif temp > value:
                    value = temp
                    max_index.clear()
                    max_index.append(index)

            # probability of action
            prob = 1 / len(max_index)

            for index in max_index:
                result[index] = prob

            next_policy[state[0]][state[1]] = result

        self.policy_table = next_policy
```

Problem 3.

Solution:

State space: $X = \{x_{ut}, t = 0, 1, 2, \cdots; u = 1,2,3\ | x_{t+1} = \ x_t - r(t)\}$

Control space: $U = \{u, u = 1, 2, 3\}$

Reward function is $r(t) = r_{ut}x_{ut}$

Dynamic programming equation:

$$v^*(x) = \max\{r(x,1) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(j);\ r(x,2) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(x+j); r(x,3) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(j)\}$$

Policy:

$$\pi^*(x) = arg\max\{r(x,1) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(j);\ r(x,2) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(x+j); r(x,3) + \gamma \sum_{j=0}^{\infty} p_j\, v_\gamma{}^*(j)\}$$