HW #1 CS159

Greg Stroot, Zhuofang Li, Francesca-Zhoufan Li
0002135380, 2136043, 2136030

Due on April 8th, 2021

Instructions

Please LaTeX your solutions using the attached template. Fill in each section with your answers and please do not change the order of sections and subsection.

You need to submit both a PDF and your code.

1 Problem 1

1.1 Transition Matrices and Cost Function (3 points)

```
P_move_forward:
[[0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]]
P_park:
[[0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]]
C_move_forward:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 4. 0.]
C_park:
[5. 0. 4. 0. 3. 0. 2. 0. 1. 0. 4. 0.]
```

Figure 1: Compute Values for $N = 5, p = 0.05$

1.2 Closed-loop system Matrices (2 points)

```
Ppi:
[[0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.05 0.95 0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]]
Cpi:  [0. 0. 0. 0. 0. 0. 2. 0. 1. 0. 4. 0.]
```

Figure 2: Compute Values for $N = 5, p = 0.05$

1.3 Policy Evaluation (2 points)

```
Policy Evaluation Terminated at iteration:  6
Value Function:   [3.7575 3.7575 3.7575 3.7575 3.7575 3.7575 2.     3.85   1.     4.
 4.     0.    ]
```

Figure 3: Compute Values for $N = 5, p = 0.05$

1.4 Value Iteration (2 points)

```
====== Start Value Iteration
Value Iteration Terminated at iteration :  6
Park from free parking spot iThreshold =  2
Value Function:  [3.719625 3.719625 3.719625 3.719625 3.       3.7575   2.       3.85
 1.       4.       4.       0.      ]
```

Figure 4: Compute Values for $N = 5, p = 0.05$

Code used to compute the above figure may be seen below:

```python
def bellmanRecursion(self, s, V):
        # Bellman recursion of the value function.
        # Recall that there are 2 actions --> First, evaluate expected cost for action 1 and action 2. Then select

        # Hint: here you need to evaluate and store the cost for all actions a.
        cost = []
        for a in range(0, self.actions):
                C_a_s = self.C[a][s]
                P_a_s = self.P[a][s,:]
                val = C_a_s + np.dot(P_a_s, V)
```

```
                cost.append(val)

        # s-th component of the value function vector
        Vn_s = np.min(cost)
        # s-th component of the action vector
        An_s = np.argmin(cost)


        return Vn_s, An_s

def valueIteration(self):
        if self.printLevel >= 0: print("====== Start Value Iteration")
        # Initialize the value function vector and action vector
        Vn = [np.zeros(self.states)]
        An = [np.zeros(self.states)]

        # Value Iteration loop
        for j in range(0, self.maxIt):
                # Initialize new value function vector and action vector
                Vnext = np.zeros(self.states)
                Anext = np.zeros(self.states)
                Vcurrent = Vn[-1]

                # Run Bellman recursion for all states using the value function vector at the previous iterate
                # Notice that self.states = 2N+2 = total number of states
                # Hint: You need to update the vectors Vnext \in \mathbb{R}^{self.states} and Anext \in \mathbb{R}
                for s in range(0, self.states):
                        Vnext[s], Anext[s] = self.bellmanRecursion(s, Vcurrent)

                Vn.append(Vnext)
                An.append(Anext)

                # Check if the algorithm has converged
                if ((j>1) and np.sum(Vn[-1]-Vn[-2])==0):
                        print('Value Iteration Terminated at iteration : ',j)
                        print('Park from free parking spot iThreshold = ', self.computeIndex(An[-1]))
                        if self.printLevel >= 1: print('Value Function: ', Vn[-1])
                        break
```

1.5 Policy Iteration (2 points)

```
Policy Iteration Terminated at iteration :  2
Park from free parking spot iThreshold =  2
Value Function:  [3.719625 3.719625 3.719625 3.719625 3.       3.7575   2.       3.85
 1.       4.       4.       0.      ]
```

Figure 5: Compute Values for $N = 5, p = 0.05$

Code used to compute the above figure may be seen below:

```python
def policyImprovement(self, V):
        # Initialize value function vector and action vector
        Vn = np.zeros(self.states)
        An = np.zeros(self.states)

        # Run Bellman recursion for all states
        # Notice that self.states = 2N+2 = total number of states
        # Hint: Here you need to run a for loop to update the vectors Vn \in \mathbb{R}^{self.states} and An \in \r
        for s in range(0, self.states):
                Vn[s], An[s] = self.bellmanRecursion(s, V)

        iThreshold = self.computeIndex(An)
        return iThreshold

def policyIteration(self):
        if self.printLevel >= 0: print("====== Start Policy Iteration")
        # Initialize list of value function vectors
        Vn = []
        iThreshold = self.N

        # Policy Iteration Loop
        for j in range(0, self.maxIt):
                # Hint: use the functions that you have already developed
                [Ppi, Cpi] = self.computePolicy(iThreshold) # First compute a policy for the threshild value iThre
                Vnext      = self.policyEvaluation(Ppi, Cpi) # Policy evaluation step
                iThreshold = self.policyImprovement(Vnext) # Policy improvement step;
                Vn.append(Vnext)

                # Check if the algorithm has converged
                if ((j>1) and np.sum(Vn[-1]-Vn[-2])==0):
                        print('Policy Iteration Terminated at iteration : ',j)
                        print('Park from free parking spot iThreshold = ', iThreshold)
                        if self.printLevel >= 1: print('Value Function: ', Vn[-1])
                        break
```

1.6 Testing on a bigger example (4 points)

```
====== Start Value Iteration
Value Iteration Terminated at iteration :  201
Park from free parking spot iThreshold =  165
====== Start Policy Iteration
Policy Iteration Terminated at iteration :  3
Park from free parking spot iThreshold =  165
```

Figure 6: Compute Values for $N = 200, p = 0.05, Cg = 100$

2 Problem 2

2.1 Cost Reformulation (1 points)

BarQ:

$$\begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{matrix}$$

Figure 7: BarQ

BarR:

$$\begin{matrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{matrix}$$

Figure 8: BarR

```
def buildCost(self):
      # Hint: you could use the function "linalg.block_diag"
      barQ=linalg.block_diag(*([self.Q] * (self.N-1)))

      barQ=linalg.block_diag(barQ, self.Qf)

      barR=linalg.block_diag(*([self.R] * (self.N)))

      H = linalg.block_diag(barQ, barR)
      q = np.zeros(H.shape[0])

      if self.printLevel >= 2:
          print("H: ")
          print(H)
          print("q: ", q)

      self.q = q
      self.H = sparse.csc_matrix(2 * H)
```

2.2 Inequality Constraint Reformulation (3 points)

G_in and E_in:

```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1. -0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1. -0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -0. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1. -0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -0. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -0. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.]]
```

```
[[-1. -0.]
 [-0. -1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

w_0:

[15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 5 5 5 5 5 5 5 5]

```python
def buildIneqConstr(self):
        # Hint 1: consider building submatrices and then stack them together
        # Hint 2: most likely you will need to use auxiliary variables
        G1=linalg.block_diag(*([self.Fx] * (self.N-1)))
        G2=linalg.block_diag(*([self.Fu] * self.N))
        G1=linalg.block_diag(G1,self.Ff)
        G_in=linalg.block_diag(G1,G2)

        G_in=np.concatenate((np.zeros((self.bx.shape[0],G_in.shape[1])),G_in))

        w_in=np.reshape([self.bx]*self.N,-1)
        w_in=np.concatenate((w_in,self.bf))
        w_in2=np.reshape([self.bu]*self.N,-1)
        w_in=np.hstack((w_in,w_in2)).T

        E_in=-self.Fx.T
        E_in=np.hstack((E_in,np.zeros((self.n,w_in.shape[0]-self.Fx.shape[0]))))
        E_in=E_in.T

        if self.printLevel >= 2:
            print("G_in: ")
            print(G_in)
            print("E_in: ")
            print(E_in)
            print("w_in: ", w_in)

        self.G_in = sparse.csc_matrix(G_in)
        self.E_in = E_in
        self.w_in = w_in.T
```

2.3 Equality Constraint Reformulation (3 points)

$$
\begin{matrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
-1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & -1
\end{matrix}
$$

Figure 9: G_eq

$$
\begin{matrix}
1 & 1 \\
0 & 1 \\
0 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0
\end{matrix}
$$

Figure 10: E_eq

```python
def buildEqConstr(self):
      # Hint 1: consider building submatrices and then stack them together
      # Hint 2: most likely you will need to use auxiliary variables
      ...
      Gu=linalg.block_diag(*([-self.B]*self.N))
      Gx1=linalg.block_diag(*([np.eye(self.n)]*self.N))
      Gx2=linalg.block_diag(*([-self.A]*(self.N-1)))
      Gx2=np.vstack((np.zeros((self.n,self.n*(self.N-1))),Gx2))
      Gx2=np.hstack((Gx2,np.zeros((self.n*self.N,self.n))))
      G_eq=np.hstack((Gx2+Gx1,Gu))

      E_eq=self.A.T
      E_eq=np.hstack((E_eq,np.zeros((self.n,self.N*self.n-self.n))))
      E_eq=E_eq.T

      if self.printLevel >= 2:
          print("G_eq: ")
          print(G_eq)
          print("E_eq: ")
          print(E_eq)

      self.G_eq = sparse.csc_matrix(G_eq)
      self.E_eq = E_eq
```

2.4 Solve the FTOCP (3 points)

```
Optimal State Trajectory:
[[-1.50000000e+01  1.50000000e+01]
 [-8.26014053e-16  1.00000000e+01]
 [ 1.00000000e+01  5.00000000e+00]
 [ 1.50000000e+01  9.84385122e-15]
 [ 1.50000000e+01  8.98749851e-15]]
Optimal Input Trajectory:
[[-5.00000000e+00]
 [-5.00000000e+00]
 [-5.00000000e+00]
 [-8.98749851e-16]]
Solver Time:  0.007988  seconds.
```

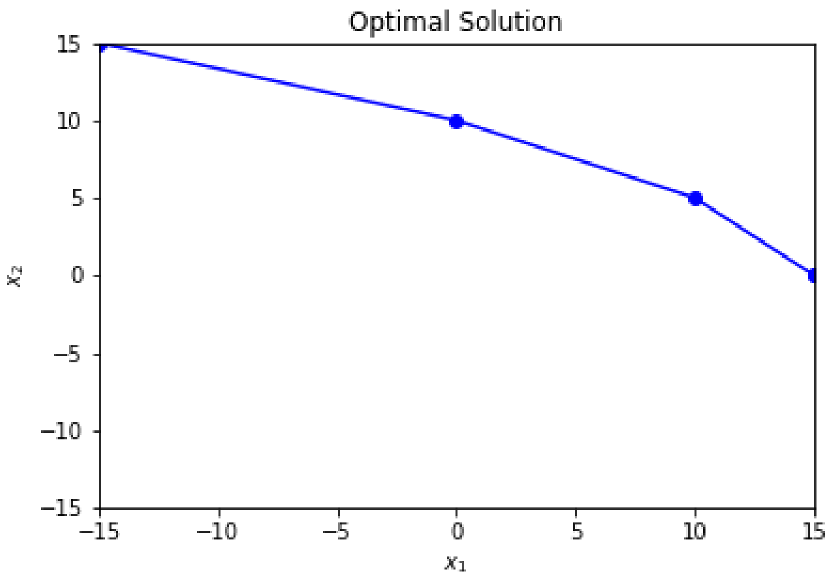

Figure 11: problem 2 optimal solution figure

3 Problem 3

3.1 QP formulation (4 points)

We know to minimize the cost function should be

$$J_0^*(x(0)) = \min_{U_0, X_0} X_0^\top \overline{Q} X_0 + U_0^\top \overline{R} U_0$$

or in the matrix format as

$$J_0^*(x(0)) = \min_{U_0, X_0} \begin{bmatrix} X_0 \top & U_0 \top \end{bmatrix} \begin{bmatrix} \overline{Q} & 0 \\ 0 & \overline{R} \end{bmatrix} \begin{bmatrix} X_0 \\ U_0 \end{bmatrix}$$

subject to the following constrains:

$$G_{0,in} \begin{bmatrix} X_0 \\ U_0 \end{bmatrix} \leq E_{0,in} x(0) + w_{0,in}$$

and

$$G_{0,eq} \begin{bmatrix} X_0 \\ U_0 \end{bmatrix} = E_{0,eq} x(0) + C_{eq}$$

where $\overline{Q} \triangleq \text{blockdiag}(Q, \ldots, Q, Q_F)$ and $\overline{R} \triangleq \text{blockdiag}(R, \ldots, R)$

$$C_{eq} = \begin{bmatrix} C_0 \\ C_1 \\ \vdots \\ C^{N-1} \end{bmatrix}$$

where

$$C_k = \begin{bmatrix} 0 \\ 0.1^k \end{bmatrix}, \forall k \in \{0, \cdots, N-1\}$$

Note now we have A corresponds to different time steps where

$$A_k = \begin{bmatrix} 1 & 0.5^k \\ 0 & 1 \end{bmatrix}, \forall k \in \{1, \cdots, N-1\}$$

$$B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$G_{0,eq} = \begin{bmatrix} I & & & & & -B & & & & \\ -A_1 & I & & & & & -B & & & \\ & -A_2 & I & & & & & -B & & \\ & & \ddots & \ddots & & & & & \ddots & \\ & & & -A_{k+1} & I & & & & & -B \end{bmatrix}$$

For $G_{0,in}$, $E_{0,in}$, and $w_{0,in}$, it would be the same as the previous question as the following

$$G_{0,\text{in}} = \begin{bmatrix} 0 & & \cdots & & 0 & 0 & & \cdots & & 0 \\ F_x & & & & & 0 & & & & \\ & F_x & & & & & 0 & & & \\ & & \ddots & & & & & \ddots & & \\ & & & F_x & & & & & 0 & \\ & & & & F_f & & & & & 0 \\ 0 & & & & & F_u & & & & \\ & 0 & & & & & F_u & & & \\ & & \ddots & & & & & \ddots & & \\ & & & 0 & & & & & F_u & \\ & & & & 0 & & & & & F_u \end{bmatrix}$$

$$E_{0,\text{in}} = [-F_x^\top, 0, \ldots, 0]^\top \in \mathbb{R}^{(n_{bx}N + n_{bf} + n_{bu}N)\times n},$$

$$w_{0,\text{in}} = [b_x^\top, \ldots, b_x^\top, b_f^\top, b_u^\top, \ldots, b_u^\top]^\top \in \mathbb{R}^{(n_{bx}N + n_{bf} + n_{bu}N)}$$

for the vectors $b_x \in \mathbb{R}^{n_{bx}}$, $b_f \in \mathbb{R}^{n_{bf}}$ and $b_u \in \mathbb{R}^{n_{bu}}$ that are used to define the constraint sets $\mathcal{X}$, $\mathcal{X}_F$ and $\mathcal{U}$, respectively.

3.2 Solve the QP (6 points)

```
def buildCost(self):
    # Hint: Are the matrices H and q constructed using  A and B?
    barQ  = linalg.block_diag(*([self.Q] * (self.N-1)))
    barQ = linalg.block_diag(barQ, self.Qf)
    barR = linalg.block_diag(*([self.R] * (self.N)))

    H = linalg.block_diag(barQ, barR)
    q = np.zeros(H.shape[0])

    if self.printLevel >= 2:
        print("H: ")
        print(H)
        print("q: ", q)

    self.q = q
    self.H = sparse.csc_matrix(2 * H)

def buildIneqConstr(self):
    # Hint: Are the matrices G_in, E_in and w_in constructed using  A and B?
    G1   = linalg.block_diag(*([self.Fx] * (self.N-1)))
    G2   = linalg.block_diag(*([self.Fu] * self.N))
    G1   = linalg.block_diag(G1,self.Ff)
    G_in = linalg.block_diag(G1,G2)
    G_in = np.concatenate((np.zeros((self.bx.shape[0],G_in.shape[1])),G_in))

    w_in  = np.reshape([self.bx]*self.N,-1)
    w_in  = np.concatenate((w_in,self.bf))
    w_in2 = np.reshape([self.bu]*self.N,-1)
    w_in  = np.hstack((w_in,w_in2)).T

    E_in = -self.Fx.T
    E_in = np.hstack((E_in,np.zeros((self.n,w_in.shape[0]-self.Fx.shape[0]))))
    E_in = E_in.T

    if self.printLevel >= 2:
        print("G_in: ")
        print(G_in)
        print("E_in: ")
        print(E_in)
        print("w_in: ", w_in)

    self.G_in = sparse.csc_matrix(G_in)
    self.E_in = E_in
    self.w_in = w_in.T
```

```
def buildEqConstr(self):
    Gu   = linalg.block_diag(*([-self.B[0]]*self.N))
    Gx1  = linalg.block_diag(*([np.eye(self.n)]*self.N))
    Gx2  = linalg.block_diag(*[-a for a in self.A[:1]])
    Gx2  = np.vstack((np.zeros((self.n,self.n*(self.N-1))),Gx2))
    Gx2  = np.hstack((Gx2,np.zeros((self.n*self.N,self.n))))
    G_eq = np.hstack((Gx2+Gx1,Gu))

    E_eq = self.A[0].T
    E_eq = np.hstack((E_eq,np.zeros((self.n,self.N*self.n-self.n)))).T

    C_eq = np.concatenate(self.C).T

    if self.printLevel >= 2:
        print("G_eq: ")
        print(G_eq)
        print("E_eq: ")
        print(E_eq)
        print("C_eq: ", C_eq)

    self.C_eq = C_eq
    self.G_eq = sparse.csc_matrix(G_eq)
    self.E_eq = E_eq
\end{lstlisting}

\begin{lstlisting}[language=Python]
def solve(self, x0):
    """Computes control action
    Arguments:
        x0: current state
    """

    # Solve QP
    startTimer = datetime.datetime.now()
    self.osqp_solve_qp(self.H, self.q, self.G_in,
                       np.add(self.w_in, np.dot(self.E_in,x0)),
                       self.G_eq, np.dot(self.E_eq,x0)+self.C_eq)
    endTimer = datetime.datetime.now(); deltaTimer = endTimer - startTimer
    self.solverTime = deltaTimer

    # Unpack Solution
    self.unpackSolution(x0)

    self.time += 1

    return self.uPred[0,:]
```

```
G_in:                                                              E_in:
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]   [[-1. -0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [-0. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  1.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [-1. -0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [-0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0. -1. -0.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0. -0. -1.  0.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0. -1. -0.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0. -0. -1.  0.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -0.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0. -0. -1.  0.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]    [ 0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.]]   [ 0.  0.]]
w_in:  [15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15  5  5  5  5
  5  5  5  5]
```

```
G_eq:
[[ 1.      0.      0.      0.      0.      0.      0.      0.      0.      0.
   0.      0.    ]
 [ 0.      1.      0.      0.      0.      0.      0.      0.     -1.      0.
   0.      0.    ]
 [-1.     -0.5     1.      0.      0.      0.      0.      0.      0.      0.
   0.      0.    ]
 [ 0.     -1.      0.      1.      0.      0.      0.      0.      0.     -1.
   0.      0.    ]
 [ 0.      0.     -1.     -0.25    1.      0.      0.      0.      0.      0.
   0.      0.    ]
 [ 0.      0.      0.     -1.      0.      1.      0.      0.      0.      0.
  -1.      0.    ]
 [ 0.      0.      0.      0.     -1.     -0.125   1.      0.      0.      0.
   0.      0.    ]
 [ 0.      0.      0.      0.      0.     -1.      0.      1.      0.      0.
   0.     -1.    ]]
E_eq:
[[1. 1.]
 [0. 1.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
C_eq:  [0.     1.     0.     0.1    0.     0.01   0.     0.001]
```

```
Optimal State Trajectory:
[[-1.50000000e+01  1.50000000e+01]
 [-8.59158911e-21  1.10954174e+01]
 [ 5.54770871e+00  8.48879377e+00]
 [ 7.66990716e+00  7.04656538e+00]
 [ 8.55072783e+00  6.40687762e+00]]
Optimal Input Trajectory:
[[-4.90458258]
 [-2.70662365]
 [-1.4522284 ]
 [-0.64068776]]
Solver Time:  0.001907  seconds.
```

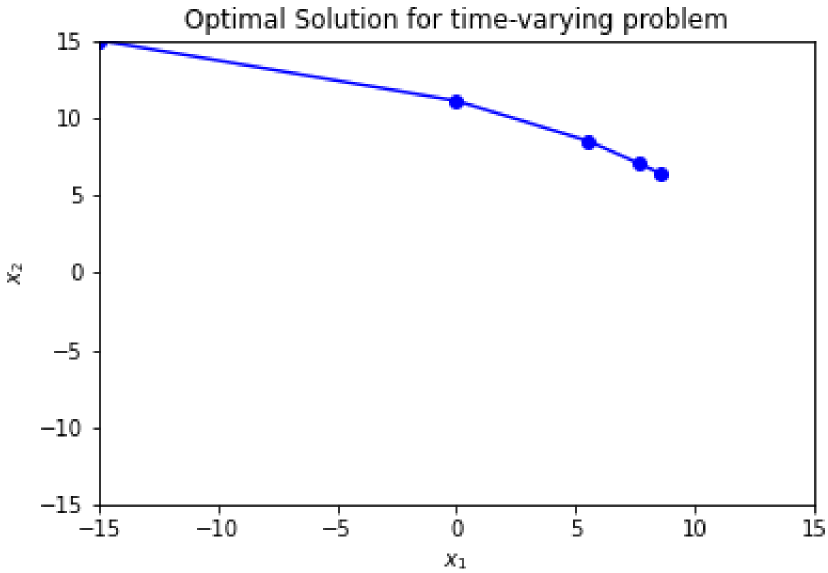

Figure 12: problem 3 optimal solution figure

4 Appendix

4.1 Problem 1 code

4.1.1 MDP.py

```python
import numpy as np
import pdb

class MDP(object):
	""" Markov Decision Process (MDP)
	Methods:
		- buildTransitionMatrices:
		- computePolicy:
		- valueIteration:
		- policyIteration:

	"""
	def __init__(self, N, p, Cg, printLevel):
		self.N  = N
		self.p  = p
		self.Cg = Cg
		self.printLevel = printLevel

		self.states  = 2*self.N + 2
		self.actions = 2
		self.maxIt   = 500

	def bellmanRecursion(self, s, V):
		# Bellman recursion of the value function.
		# Recall that there are 2 actions --> First, evaluate expected cost for action 1 and action 2. Then select

		# Hint: here you need to evaluate and store the cost for all actions a.
		cost = []
		for a in range(0, self.actions):
			C_a_s = self.C[a][s]
			P_a_s = self.P[a][s,:]
			val = C_a_s + np.dot(P_a_s, V)

			cost.append(val)

		# s-th component of the value function vector
		Vn_s = np.min(cost)
		# s-th component of the action vector
		An_s = np.argmin(cost)


		return Vn_s, An_s

	def policyIteration(self):
		if self.printLevel >= 0: print("====== Start Policy Iteration")
		# Initialize list of value function vectors
		Vn = []
		iThreshold = self.N

		# Policy Iteration Loop
		for j in range(0, self.maxIt):
			# Hint: use the functions that you have already developed
			[Ppi, Cpi] = self.computePolicy(iThreshold) # First compute a policy for the threshild value iThre
			Vnext      = self.policyEvaluation(Ppi, Cpi) # Policy evaluation step
```

```
                iThreshold = self.policyImprovement(Vnext) # Policy improvement step;
                Vn.append(Vnext)

                # Check if the algorithm has converged
                if ((j>1) and np.sum(Vn[-1]-Vn[-2])==0):
                        print('Policy Iteration Terminated at iteration : ',j)
                        print('Park from free parking spot iThreshold = ', iThreshold)
                        if self.printLevel >= 1: print('Value Function: ', Vn[-1])
                        break

def policyImprovement(self, V):
        # Initialize value function vector and action vector
        Vn = np.zeros(self.states)
        An = np.zeros(self.states)

        # Run Bellman recursion for all states
        # Notice that self.states = 2N+2 = total number of states
        # Hint: Here you need to run a for loop to update the vectors Vn \in \mathbb{R}^{self.states} and An \in \
        for s in range(0, self.states):
                Vn[s], An[s] = self.bellmanRecursion(s, V)

        iThreshold = self.computeIndex(An)
        return iThreshold

def computeIndex(self, An):
        return np.argmax(An[::2])

def valueIteration(self):
        if self.printLevel >= 0: print("====== Start Value Iteration")
        # Initialize the value function vector and action vector
        Vn = [np.zeros(self.states)]
        An = [np.zeros(self.states)]

        # Value Iteration loop
        for j in range(0, self.maxIt):
                # Initialize new value function vector and action vector
                Vnext = np.zeros(self.states)
                Anext = np.zeros(self.states)
                Vcurrent = Vn[-1]

                # Run Bellman recursion for all states using the value function vector at the previous iterate
                # Notice that self.states = 2N+2 = total number of states
                # Hint: You need to update the vectors Vnext \in \mathbb{R}^{self.states} and Anext \in \mathbb{R}
                for s in range(0, self.states):
                        Vnext[s], Anext[s] = self.bellmanRecursion(s, Vcurrent)

                Vn.append(Vnext)
                An.append(Anext)

                # Check if the algorithm has converged
                if ((j>1) and np.sum(Vn[-1]-Vn[-2])==0):
                        print('Value Iteration Terminated at iteration : ',j)
                        print('Park from free parking spot iThreshold = ', self.computeIndex(An[-1]))
                        if self.printLevel >= 1: print('Value Function: ', Vn[-1])
                        break

def policyEvaluation(self, P, C):
        # Initialize value function
        Vn = [np.zeros(self.states)]
```

```python
        # Run iterative strategy
        for j in range(0,self.maxIt):
                Vnext = C + Vn[-1] @ P.T  # Hint: here you need to use only Vn[-1], P and C.
                Vn.append( Vnext )

                # Check if algorithm has converged
                if ((j>1) and np.sum(Vn[-1]-Vn[-2])==0):
                        if self.printLevel >= 2: print("Policy Evaluation Terminated at iteration: ",j)
                        break

        Vout = Vn[-1]
        if self.printLevel >= 2: print("Value Function:  ", Vout)
        return Vout

def computePolicy(self,iThreshold = 0):
        # Compute the state sThreshold such that
        # if a state s <  sThreshold --> the transition probabilities are are given by the matrix self.P[0] associ
        # if a state s >= sThreshold --> the transition probabilities are are given by the matrix self.P[1] associ
        sThreshold = 2*iThreshold


        # You need to combine the matrices self.P[0] and self.P[1] which are assocaied
        # with the move forward acton and parking action, respectively
        # (hint: use the variable sThreshold and the command vstack)
        Ppi = np.vstack((self.P[0][0:sThreshold-1,:], self.P[1][sThreshold-1:,:]))

        # You need to combine the vectors self.C[0] and self.C[1] which are assocaied
        # with the move forward acton and parking action, respectively (hint: use the variable sThreshold)
        # (hint: use the variable sThreshold and the command hstack)
        Cpi = np.hstack((self.C[0][0:sThreshold-1], self.C[1][sThreshold-1:]))

        if self.printLevel >= 3:
                print("Ppi: ")
                print(Ppi)
                print("Cpi: ", Cpi)

        return Ppi, Cpi

def buildTransitionMatrices(self):
        # First 2*N states are assocaited with the N parking spots being free or occupied.
        # Last tow state are Garage and Theater
        P_move_forward = np.zeros((self.states, self.states))
        P_park         = np.zeros((self.states, self.states))

        # Make Theater an absorbing state
        P_move_forward[-1,-1] = 1
        P_park[-1,-1]         = 1

        # Set Transition From Garage to Theater
        P_move_forward[-2,-1] = 1
        P_park[-2,-1]         = 1


        # Move Forward
        for i  in range(0, self.N):
                i_f = 2*i  # i-th parking spot free
                i_o = 2*i+1 # i-th parking spot occupied
                s_n = 2*(i+1) # (i+1)-th parking spot free for (i+1) < N and garage for (i+1) = N

                if i == self.N-1:
```

```
                P_move_forward[i_f, s_n] = 1
                P_move_forward[i_o, s_n] = 1
        else:
                P_move_forward[i_o, s_n+0] = self.p
                P_move_forward[i_o, s_n+1] = 1 - self.p
                P_move_forward[i_f, s_n+0] = self.p
                P_move_forward[i_f, s_n+1] = 1 - self.p

# Parking
for i  in range(0, self.N):
        i_f = 2*i  # i-th parking spot free
        i_o = 2*i+1 # i-th parking spot occupied
        s_n = 2*(i+1) # (i+1)-th parking spot free

        if i == self.N-1:
                P_park[i_f, -1] = 1
                P_park[i_o, s_n] = 1
        else:
                P_park[i_f, -1] = 1
                P_park[i_o, s_n+0] = self.p
                P_park[i_o, s_n+1] = 1 - self.p

# Compute cost vector associated with each action
C_move_forward = np.zeros(2*self.N+2);
C_park         = np.zeros(2*self.N+2);

# Cost of being at the Garage
C_move_forward[-2] = self.Cg
C_park[-2]         = self.Cg

for i in range(0, self.N):
        i_f = 2*i # i-th parking spot free
        C_park[i_f] = self.N - i

if self.printLevel >= 2:
        print("P_move_forward:")
        print(P_move_forward)

        print("P_park:")
        print(P_park)

        print("C_move_forward:")
        print(C_move_forward)

        print("C_park:")
        print(C_park)

self.C = [C_move_forward, C_park]
self.P = [P_move_forward, P_park]
```

4.2 Problem 2 code

4.2.1 ftocp.py

```
import pdb
import numpy as np
from cvxopt import spmatrix, matrix, solvers
from numpy import linalg as la
from scipy import linalg
from scipy import sparse
from cvxopt.solvers import qp
import datetime
from numpy import hstack, inf, ones
from scipy.sparse import vstack
from osqp import OSQP
from dataclasses import dataclass, field


class FTOCP(object):
    """ Finite Time Optimal Control Problem (FTOCP)
    Methods:
        - solve: solves the FTOCP given the initial condition x0 and terminal contraints
        - buildNonlinearProgram: builds the ftocp program solved by the above solve method
        - model: given x_t and u_t computes x_{t+1} = f( x_t, u_t )

    """

    def __init__(self, N, A, B, Q, R, Qf, Fx, bx, Fu, bu, Ff, bf, printLevel):
        # Define variables
        self.printLevel = printLevel

        self.A  = A
        self.B  = B
        self.N  = N
        self.n  = A.shape[1]
        self.d  = B.shape[1]
        self.Fx = Fx
        self.bx = bx
        self.Fu = Fu
        self.bu = bu
        self.Ff = Ff
        self.bf = bf
        self.Q  = Q
        self.Qf = Qf
        self.R  = R

        print("Initializing FTOCP")
        # ftocp.buildCost()
        # ftocp.buildIneqConstr()
        # ftocp.buildEqConstr()
        self.buildCost()
        self.buildIneqConstr()
        self.buildEqConstr()
        print("Done initializing FTOCP")

        self.time = 0


    def solve(self, x0):
        """Computes control action
        Arguments:
```

```python
        x0: current state
    """

    # Solve QP
    startTimer = datetime.datetime.now()
    self.osqp_solve_qp(self.H, self.q, self.G_in,
                np.add(self.w_in, np.dot(self.E_in,x0)),
                self.G_eq, np.dot(self.E_eq,x0))
    endTimer = datetime.datetime.now(); deltaTimer = endTimer - startTimer
    self.solverTime = deltaTimer

    # Unpack Solution
    self.unpackSolution(x0)

    self.time += 1

    return self.uPred[0,:]

def unpackSolution(self, x0):
    # Extract predicted state and predicted input trajectories
    self.xPred = np.vstack((x0, np.reshape((self.Solution[np.arange(self.n*(self.N))]),(self.N,self.n))))
    self.uPred = np.reshape((self.Solution[self.n*(self.N)+np.arange(self.d*self.N)]),(self.N, self.d))

    if self.printLevel >= 2:
        print("Optimal State Trajectory: ")
        print(self.xPred)

        print("Optimal Input Trajectory: ")
        print(self.uPred)

    if self.printLevel >= 1: print("Solver Time: ", self.solverTime.total_seconds(), " seconds.")

def buildIneqConstr(self):
    # Hint 1: consider building submatrices and then stack them together
    # Hint 2: most likely you will need to use auxiliary variables
    G1=linalg.block_diag(*([self.Fx] * (self.N-1)))
    G2=linalg.block_diag(*([self.Fu] * self.N))
    G1=linalg.block_diag(G1,self.Ff)
    G_in=linalg.block_diag(G1,G2)


    G_in=np.concatenate((np.zeros((self.bx.shape[0],G_in.shape[1])),G_in))



    w_in=np.reshape([self.bx]*self.N,-1)
    w_in=np.concatenate((w_in,self.bf))
    w_in2=np.reshape([self.bu]*self.N,-1)
    w_in=np.hstack((w_in,w_in2)).T


    E_in=-self.Fx.T
    E_in=np.hstack((E_in,np.zeros((self.n,w_in.shape[0]-self.Fx.shape[0]))))
    E_in=E_in.T

    if self.printLevel >= 2:
        print("G_in: ")
        print(G_in)
        print("E_in: ")
        print(E_in)
```

```python
            print("w_in: ", w_in)

        self.G_in = sparse.csc_matrix(G_in)
        self.E_in = E_in
        self.w_in = w_in.T

    def buildCost(self):
        # Hint: you could use the function "linalg.block_diag"
        barQ=linalg.block_diag(*([self.Q] * (self.N-1)))

        barQ=linalg.block_diag(barQ, self.Qf)

        barR=linalg.block_diag(*([self.R] * (self.N)))

        H = linalg.block_diag(barQ, barR)
        q = np.zeros(H.shape[0])

        if self.printLevel >= 2:
            print("H: ")
            print(H)
            print("q: ", q)

        self.q = q
        self.H = sparse.csc_matrix(2 * H)  #  Need to multiply by two because CVX considers 1/2 in front of quadratic cost

    def buildEqConstr(self):
        # Hint 1: consider building submatrices and then stack them together
        # Hint 2: most likely you will need to use auxiliary variables

        Gu=linalg.block_diag(*([-self.B]*self.N))
        Gx1=linalg.block_diag(*([np.eye(self.n)]*self.N))
        Gx2=linalg.block_diag(*([-self.A]*(self.N-1)))
        Gx2=np.vstack((np.zeros((self.n,self.n*(self.N-1))),Gx2))
        Gx2=np.hstack((Gx2,np.zeros((self.n*self.N,self.n))))
        G_eq=np.hstack((Gx2+Gx1,Gu))
        print(Gu.shape, Gx1.shape, Gx2.shape, G_eq.shape)

        E_eq=self.A.T
        E_eq=np.hstack((E_eq,np.zeros((self.n,self.N*self.n-self.n))))
        E_eq=E_eq.T

        if self.printLevel >= 2:
            print("G_eq: ")
            print(G_eq)
            print("E_eq: ")
            print(E_eq)

        self.G_eq = sparse.csc_matrix(G_eq)
        self.E_eq = E_eq

    def osqp_solve_qp(self, P, q, G= None, h=None, A=None, b=None, initvals=None):
        """
        Solve a Quadratic Program defined as:
        minimize
            (1/2) * x.T * P * x + q.T * x
        subject to
            G * x <= h
            A * x == b
        using OSQP <https://github.com/oxfordcontrol/osqp>.
        """
```

```
        qp_A = vstack([G, A]).tocsc()
        l = -inf * ones(len(h))
        qp_l = hstack([l, b])
        qp_u = hstack([h, b])

        self.osqp = OSQP()
        self.osqp.setup(P=P, q=q, A=qp_A, l=qp_l, u=qp_u, verbose=False, polish=True)

        if initvals is not None:
            self.osqp.warm_start(x=initvals)
        res = self.osqp.solve()
        if res.info.status_val == 1:
            self.feasible = 1
        else:
            self.feasible = 0
            print("The FTOCP is not feasible at time t = ", self.time)

        self.Solution = res.x
```

4.2.2 main.py

```
import numpy as np
from utils import system
import pdb
import matplotlib.pyplot as plt
from ftocp import FTOCP
from matplotlib import rc
from numpy import linalg as la
# =====================================================
# Initialize system parameters
A = np.array([[1, 1],
                  [0, 1]]);
B = np.array([[0],
                          [1]]);

x0    = np.array([-15.0,15.0])   # initial condition

# Initialize ftocp parameters
printLevel = 3
N  = 4; n = 2; d = 1;
Q      = np.eye(n)
R      = 10*np.eye(d)
Qf     = np.eye(n)

# State constraint set X = \{ x : F_x x \leq b_x \}
Fx = np.vstack((np.eye(n), -np.eye(n)))
bx = np.array([15,15]*(2))

# Input constraint set U = \{ u : F_u u \leq b_u \}
Fu = np.vstack((np.eye(d), -np.eye(d)))
bu = np.array([5]*2)

Ff = Fx
bf = bx

# =====================================================
# Solve FTOCP and plot the solutiob
ftocp = FTOCP(N, A, B, Q, R, Qf, Fx, bx, Fu, bu, Ff, bf, printLevel)
```

```
ftocp.solve(x0)

# +
plt.figure()
plt.plot(ftocp.xPred[:,0], ftocp.xPred[:,1], '-ob')
plt.title('Optimal Solution')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.xlim(-15,15)
plt.ylim(-15,15)

plt.savefig("problem_2.png")
plt.show()
```

4.2.3 utils.py

```
import numpy as np
import pdb
from scipy.spatial import ConvexHull


class system(object):
        """docstring for system"""
        def __init__(self, A, B, w_inf, x0):
                self.A     = A
                self.B     = B
                self.w_inf = w_inf
                self.x            = [x0]
                self.u            = []
                self.w            = []
                self.x0    = x0

                self.w_v = w_inf*(2*((np.arange(2**A.shape[1])[:,None] & (1 << np.arange(A.shape[1]))) > 0) - 1)

        def applyInput(self, ut):
                self.u.append(ut)
                self.w.append(np.random.uniform(-self.w_inf,self.w_inf,self.A.shape[1]))
                xnext = np.dot(self.A,self.x[-1]) + np.dot(self.B,self.u[-1]) + self.w[-1]
                self.x.append(xnext)

        def reset_IC(self):
                self.x = [self.x0]
                self.u = []
                self.w = []
```

4.3 Problem 3 code

4.3.1 ftocp.py

```python
import pdb
import numpy as np
from cvxopt import spmatrix, matrix, solvers
from numpy import linalg as la
from scipy import linalg
from scipy import sparse
from cvxopt.solvers import qp
import datetime
from numpy import hstack, inf, ones
from scipy.sparse import vstack
from osqp import OSQP
from dataclasses import dataclass, field

class FTOCP(object):
    """ Finite Time Optimal Control Problem (FTOCP)
    Methods:
        - solve: solves the FTOCP given the initial condition x0 and terminal contraints
        - buildNonlinearProgram: builds the ftocp program solved by the above solve method
        - model: given x_t and u_t computes x_{t+1} = f( x_t, u_t )

    """

    def __init__(self, N, A, B, C, Q, R, Qf, Fx, bx, Fu, bu, Ff, bf, printLevel):
        # Define variables
        self.printLevel = printLevel

        self.A  = A
        self.B  = B
        self.C  = C
        self.N  = N
        self.n  = A[0].shape[1]
        self.d  = B[0].shape[1]
        self.Fx = Fx
        self.bx = bx
        self.Fu = Fu
        self.bu = bu
        self.Ff = Ff
        self.bf = bf
        self.Q  = Q
        self.Qf = Qf
        self.R  = R

        print("Initializing FTOCP")
        self.buildIneqConstr()
        self.buildCost()
        self.buildEqConstr()
        print("Done initializing FTOCP")

        self.time = 0


    def solve(self, x0):
        """Computes control action
        Arguments:
            x0: current state
        """
```

```python
        # Solve QP
        startTimer = datetime.datetime.now()
        self.osqp_solve_qp(self.H, self.q, self.G_in,
                           np.add(self.w_in, np.dot(self.E_in,x0)),
                           self.G_eq, np.dot(self.E_eq,x0)+self.C_eq)
        endTimer = datetime.datetime.now(); deltaTimer = endTimer - startTimer
        self.solverTime = deltaTimer

        # Unpack Solution
        self.unpackSolution(x0)

        self.time += 1

        return self.uPred[0,:]

    def unpackSolution(self, x0):
        # Extract predicted state and predicted input trajectories
        self.xPred = np.vstack((x0, np.reshape((self.Solution[np.arange(self.n*(self.N))]),(self.N,self.n))))
        self.uPred = np.reshape((self.Solution[self.n*(self.N)+np.arange(self.d*self.N)]),(self.N, self.d))

        if self.printLevel >= 2:
            print("Optimal State Trajectory: ")
            print(self.xPred)

            print("Optimal Input Trajectory: ")
            print(self.uPred)

        if self.printLevel >= 1: print("Solver Time: ", self.solverTime.total_seconds(), " seconds.")

    def buildIneqConstr(self):
        # Hint: Are the matrices G_in, E_in and w_in constructed using  A and B?
        G1   = linalg.block_diag(*([self.Fx] * (self.N-1)))
        G2   = linalg.block_diag(*([self.Fu] * self.N))
        G1   = linalg.block_diag(G1,self.Ff)
        G_in = linalg.block_diag(G1,G2)
        G_in = np.concatenate((np.zeros((self.bx.shape[0],G_in.shape[1])),G_in))

        w_in  = np.reshape([self.bx]*self.N,-1)
        w_in  = np.concatenate((w_in,self.bf))
        w_in2 = np.reshape([self.bu]*self.N,-1)
        w_in  = np.hstack((w_in,w_in2)).T

        E_in = -self.Fx.T
        E_in = np.hstack((E_in,np.zeros((self.n,w_in.shape[0]-self.Fx.shape[0]))))
        E_in = E_in.T

        if self.printLevel >= 2:
            print("G_in: ")
            print(G_in)
            print("E_in: ")
            print(E_in)
            print("w_in: ", w_in)

        self.G_in = sparse.csc_matrix(G_in)
        self.E_in = E_in
        self.w_in = w_in.T


    def buildCost(self):
        # Hint: Are the matrices H and q constructed using  A and B?
```

```
        barQ  = linalg.block_diag(*([self.Q] * (self.N-1)))
        barQ = linalg.block_diag(barQ, self.Qf)
        barR = linalg.block_diag(*([self.R] * (self.N)))

        H = linalg.block_diag(barQ, barR)
        q = np.zeros(H.shape[0])

        if self.printLevel >= 2:
            print("H: ")
            print(H)
            print("q: ", q)

        self.q = q
        self.H = sparse.csc_matrix(2 * H)  #  Need to multiply by two because CVX considers 1/2 in front of quadratic cost

    def buildEqConstr(self):
        Gu   = linalg.block_diag(*([-self.B[0]]*self.N))
        Gx1  = linalg.block_diag(*([np.eye(self.n)]*self.N))
        Gx2  = linalg.block_diag(*[-a for a in self.A[1:]])
        Gx2  = np.vstack((np.zeros((self.n,self.n*(self.N-1))),Gx2))
        Gx2  = np.hstack((Gx2,np.zeros((self.n*self.N,self.n))))
        G_eq = np.hstack((Gx2+Gx1,Gu))

        E_eq = self.A[0].T
        E_eq = np.hstack((E_eq,np.zeros((self.n,self.N*self.n-self.n)))).T

        C_eq = np.concatenate(self.C).T

        if self.printLevel >= 2:
            print("G_eq: ")
            print(G_eq)
            print("E_eq: ")
            print(E_eq)
            print("C_eq: ", C_eq)

        self.C_eq = C_eq
        self.G_eq = sparse.csc_matrix(G_eq)
        self.E_eq = E_eq

    def osqp_solve_qp(self, P, q, G= None, h=None, A=None, b=None, initvals=None):
        """
        Solve a Quadratic Program defined as:
        minimize
            (1/2) * x.T * P * x + q.T * x
        subject to
            G * x <= h
            A * x == b
        using OSQP <https://github.com/oxfordcontrol/osqp>.
        """

        qp_A = vstack([G, A]).tocsc()
        l = -inf * ones(len(h))
        qp_l = hstack([l, b])
        qp_u = hstack([h, b])

        self.osqp = OSQP()
        self.osqp.setup(P=P, q=q, A=qp_A, l=qp_l, u=qp_u, verbose=False, polish=True)

        if initvals is not None:
            self.osqp.warm_start(x=initvals)
```

```
        res = self.osqp.solve()
        if res.info.status_val == 1:
            self.feasible = 1
        else:
            self.feasible = 0
            print("The FTOCP is not feasible at time t = ", self.time)

        self.Solution = res.x
```

4.3.2 main.py

```
import numpy as np
from utils import system
import pdb
import matplotlib.pyplot as plt
from ftocp import FTOCP
from matplotlib import rc
from numpy import linalg as la

# =====================================================
# Initialize system parameters
A = np.array([[1, 1],
                  [0, 1]]);
B = np.array([[0],
                          [1]]);
C = []
x0    = np.array([-15.0,15.0])   # initial condition

# Initialize ftocp parameters
printLevel = 3
N  = 4
n = 2
d = 1
Q      = np.eye(2)
R      = 10*np.eye(1)
Qf     = np.eye(2)

# State constraint set X = \{ x : F_x x \leq b_x \}
Fx = np.vstack((np.eye(n), -np.eye(n)))
bx = np.array([15,15]*(2))

# Input constraint set U = \{ u : F_u u \leq b_u \}
Fu = np.vstack((np.eye(d), -np.eye(d)))
bu = np.array([5]*2)

Ff = Fx
bf = bx
# =====================================================
# Solve FTOCP and plot the solution
A = []
B = []
C = []
for i in range(0, N):
        A.append(np.array([[1, 0.5**i],
                           [0, 1]]))
        B.append(np.array([[0],
                                            [1]]))

        C.append(np.array([0, 0.1**i]))
```

```
ftocp = FTOCP(N, A, B, C, Q, R, Qf, Fx, bx, Fu, bu, Ff, bf, printLevel)
ftocp.solve(x0)

plt.figure()
plt.plot(ftocp.xPred[:,0], ftocp.xPred[:,1], '-ob')
plt.title('Optimal Solution for time-varying problem')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.xlim(-15,15)
plt.ylim(-15,15)
plt.savefig("problem_3.png")
plt.show()
```

4.3.3 utils.py

```
import numpy as np
import pdb
from scipy.spatial import ConvexHull


class system(object):
	"""docstring for system"""
	def __init__(self, A, B, w_inf, x0):
		self.A     = A
		self.B     = B
		self.w_inf = w_inf
		self.x 		= [x0]
		self.u 		= []
		self.w 		= []
		self.x0    = x0

		self.w_v = w_inf*(2*((np.arange(2**A.shape[1])[:,None] & (1 << np.arange(A.shape[1]))) > 0) - 1)

	def applyInput(self, ut):
		self.u.append(ut)
		self.w.append(np.random.uniform(-self.w_inf,self.w_inf,self.A.shape[1]))
		xnext = np.dot(self.A,self.x[-1]) + np.dot(self.B,self.u[-1]) + self.w[-1]
		self.x.append(xnext)

	def reset_IC(self):
		self.x = [self.x0]
		self.u = []
		self.w = []
```