Assignment 4: Finite-Time Optimal Control (Solution)

ME C231A, EECS C220B, UC Berkeley

1. Finite-Time Optimal Control of a Vehicle

This question is about solving a nonlinear finit-time optimal control problem and it is similar to the unicycle example discussed in class. For each part of the question, please print out 4 subplots (one for each state vs time). Also print a simple plot showing the vehicle motion (x_k vs y_k). In addition, plot 2 subplots (one for each control input vs time). Also, turn in the code for both parts (a) and (b), which formulate the optimization problems described below. For both parts, specify IPOPT as the solver in `pyomo`.

Consider the same simplified kinematic bicycle model used in Homework 1.

$$
\begin{aligned}
\dot{x} &= v\cos(\psi+\beta) \\
\dot{y} &= v\sin(\psi+\beta) \\
\dot{v} &= a \\
\dot{\psi} &= \frac{v}{l_r}\sin(\beta) \\
\beta &= \tan^{-1}\left(\frac{l_r}{l_f+l_r}\tan(\delta_f)\right)
\end{aligned}
$$

where

$$
\begin{aligned}
x &= \text{global x CoG coordinate} \\
y &= \text{global y CoG coordinate} \\
v &= \text{speed of the vehicle} \\
\psi &= \text{global heading angle} \\
\beta &= \text{angle of the current velocity with respect to the longitudinal axis of the car} \\
a &= \text{acceleration of the center of mass into this direction} \\
l_r &= \text{distance from the center of mass of the vehicle to the rear axle} \\
l_f &= \text{distance from the center of mass of the vehicle to the front axle} \\
\delta_f &= \text{steering angle of the front wheels with respect to the longitudinal axis of the car}
\end{aligned}
$$

Collect the states in one vector $z = [x, y, v, \psi]^T$, and the inputs as $u = [a, \beta]^T$. Obtain a discrete-time model by using Forward Euler Discretization with sampling time $\Delta t = 0.2$. Use $l_f = l_r = 1.738$ in the simulation. You are asked to formulate and solve a parking problem as a finite-time optimal control problem. The vehicle starts from the initial state $\bar{z}_0 = [0,\ 3,\ 0,\ 0]^T$. Our goal is to park the vehicle in the terminal state $\bar{z}_N = [0, 0, 0, -\pi/2]^T$. Set the horizon $N = 70$ and the state constraints to be: $[-20,\ \ -5,\ \ -10,\ \ -2\pi]^T \leq z(k) \leq [20,\ \ 10,\ \ 10,\ \ 2\pi]^T$. Solve the finite time optimal control problem and plot the results.

Part (a)

Parking Problem Formulation 1

Consider the finite time optimal control problem defined below. Formulate and solve the optimzation problem in `pyomo`.

$$
\begin{aligned}
\min_{z_0,\dots,z_N,u_0,\dots,u_{N-1}} \quad & \sum_{k=N-2}^{N} \|z_k - \bar{z}_N\|_2^2 \\
& z_{k+1} = z_k + f(z_k, u_k)\Delta t && \forall k = \{0, \dots, N-1\} \\
& z_{min} \leq z_k \leq z_{max} && \forall k = \{0, \dots, N\} \\
& u_{min} \leq u_k \leq u_{max} && \forall k = \{0, \dots, N-1\} \\
& |\beta_{k+1} - \beta_k| \leq \beta_d && \forall k = \{0, \dots, N-2\} \\
& z_0 = \bar{z}_0 \\
& z_N = \bar{z}_N
\end{aligned}
$$

Consider the following constraints:

1. The difference of current and previous steering commands are bounded by ± 0.2 rad. (i.e. $\beta_d = 0.2$)
2. The accelerations are bounded by $|a(k)| \leq 0.3\ \mathrm{m/s^2}$.
3. The steering control inputs are limited to $|\beta(k)| \leq 0.6$ rad.

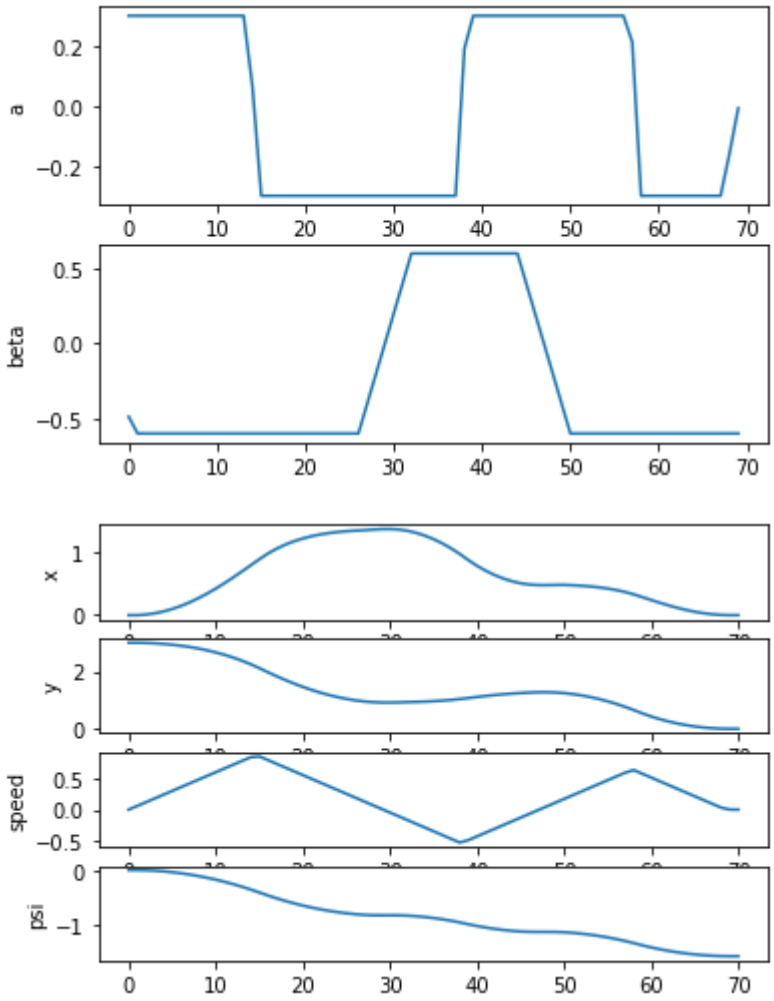
In [1]:
```
# System parameters and simulation setting
import matplotlib.pyplot as plt
import numpy as np
import pyomo.environ as pyo

# simulation parameters
Ts = 0.2
N = 70
TFinal = Ts*N

# z is state vector, u is input, Ts is sampling period.

z0Bar = np.array([0,3,0,0])
zNBar = np.array([0,0,0,-np.pi/2])
zMax = np.array([20,10,10,2*np.pi])
zMin = np.array([-20,-0.2,-10,-2*np.pi])

nz = 4        # number of states
nu = 2         # number of inputs

l_r   = 1.738

model = pyo.ConcreteModel()
model.tidx = pyo.Set(initialize=range(0, N+1)) # length of finite optimization
problem
model.zidx = pyo.Set(initialize=range(0, nz))
model.uidx = pyo.Set(initialize=range(0, nu))

# Create state and input variables trajectory:
model.z = pyo.Var(model.zidx, model.tidx)
model.u = pyo.Var(model.uidx, model.tidx)

# Objective:

model.cost = pyo.Objective(expr = sum((model.z[i, t] - zNBar[i])**2 for i in m
odel.zidx for t in model.tidx if (N-3 < t) & (t < N+1)), sense=pyo.minimize)

# Constraints:

model.constraint1 = pyo.Constraint(model.zidx, rule=lambda model, i: model.z[i
, 0] == z0Bar[i])
model.constraint2 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[0
, t+1] == model.z[0, t] + Ts*(model.z[2, t]*pyo.cos(model.z[3, t] + model.u[1,
t]))
                                   if t < N else pyo.Constraint.Skip)
model.constraint3 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[1
, t+1] == model.z[1, t] + Ts*(model.z[2, t]*pyo.sin(model.z[3, t] + model.u[1,
t]))
                                   if t < N else pyo.Constraint.Skip)
model.constraint4 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[2
, t+1] == model.z[2, t] + Ts*model.u[0, t]
                                   if t < N else pyo.Constraint.Skip)
model.constraint5 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[3
, t+1] == model.z[3, t] + Ts*(model.z[2, t]/l_r*pyo.sin(model.u[1, t]))
                                   if t < N else pyo.Constraint.Skip)
model.constraint6 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[0
```

```
, t] <= 0.3
                                    if t < N else pyo.Constraint.Skip)
model.constraint7 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[0
, t] >= -0.3
                                    if t < N else pyo.Constraint.Skip)
model.constraint8 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[1
, t+1] - model.u[1, t] <= 0.2
                                    if t < N-1 else pyo.Constraint.Skip)
model.constraint9 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[1
, t+1] - model.u[1, t] >= -0.2
                                    if t < N-1 else pyo.Constraint.Skip)
model.constraint10 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[
1, t] >= -0.6
                                    if t < N else pyo.Constraint.Skip)
model.constraint11 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[
1, t] <= 0.6
                                    if t < N else pyo.Constraint.Skip)
model.constraint12 = pyo.Constraint(model.zidx, rule=lambda model, i:model.z[i
, N] == zNBar[i])

model.constraint13 = pyo.Constraint(model.zidx, model.tidx, rule=lambda model,
i, t: model.z[i, t] <= zMax[i]
                                    if t < N else pyo.Constraint.Skip)

model.constraint14 = pyo.Constraint(model.zidx, model.tidx, rule=lambda model,
i, t: model.z[i, t] >= zMin[i]
                                    if t < N else pyo.Constraint.Skip)

# Now we can solve:
results = pyo.SolverFactory('ipopt').solve(model).write()
```

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Lower bound: -inf
  Upper bound: inf
  Number of objectives: 1
  Number of constraints: 1266
  Number of variables: 424
  Sense: unknown
# ----------------------------------------------------------
#   Solver Information
# ----------------------------------------------------------
Solver:
- Status: ok
  Message: Ipopt 3.13.2\x3a Optimal Solution Found
  Termination condition: optimal
  Id: 0
  Error rc: 0
  Time: 1.2279338836669922
# ----------------------------------------------------------
#   Solution Information
# ----------------------------------------------------------
Solution:
- number of solutions: 0
  number of solutions displayed: 0
```

In [2]:

```
# plot results
x = [pyo.value(model.z[0,0])]
y = [pyo.value(model.z[1,0])]
v = [pyo.value(model.z[2,0])]
psi = [pyo.value(model.z[3,0])]
a = [pyo.value(model.u[0,0])]
beta = [pyo.value(model.u[1,0])]

for t in model.tidx:
    if t < N:
        x.append(pyo.value(model.z[0,t+1]))
        y.append(pyo.value(model.z[1,t+1]))
        v.append(pyo.value(model.z[2,t+1]))
        psi.append(pyo.value(model.z[3,t+1]))
    if t < N-1:
        a.append(pyo.value(model.u[0,t+1]))
        beta.append(pyo.value(model.u[1,t+1]))

plt.figure()
plt.subplot(2,1,1)
plt.plot(a)
plt.ylabel('a')
plt.subplot(2,1,2)
plt.plot(beta)
plt.ylabel('beta')

plt.figure()
plt.subplot(4,1,1)
plt.plot(x)
plt.ylabel('x')
plt.subplot(4,1,2)
plt.plot(y)
plt.ylabel('y')
plt.subplot(4,1,3)
plt.plot(v)
plt.ylabel('speed')
plt.subplot(4,1,4)
plt.plot(psi)
plt.ylabel('psi')
plt.show()
```



Part (b)

Parking Problem Formulation 2

Use formulation 1 and add the additional constraints on the difference between current and previous acceleration to get a smoother parking maneuver:

$$|a(k+1) - a(k)| \leq a_d \quad \forall k = 0, \ldots, N-2,$$

where $a_d = 0.06$ m/s^2. Formulate the optimzation problem in `pyomo`.

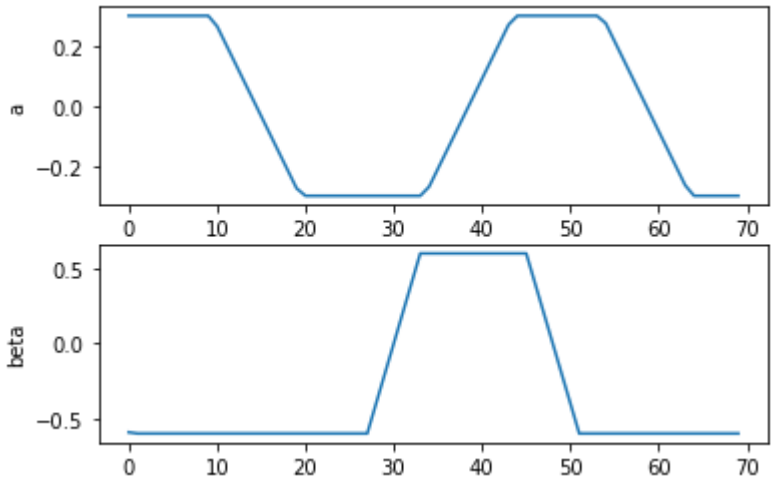
Show by using plots that you get a smoother control action.

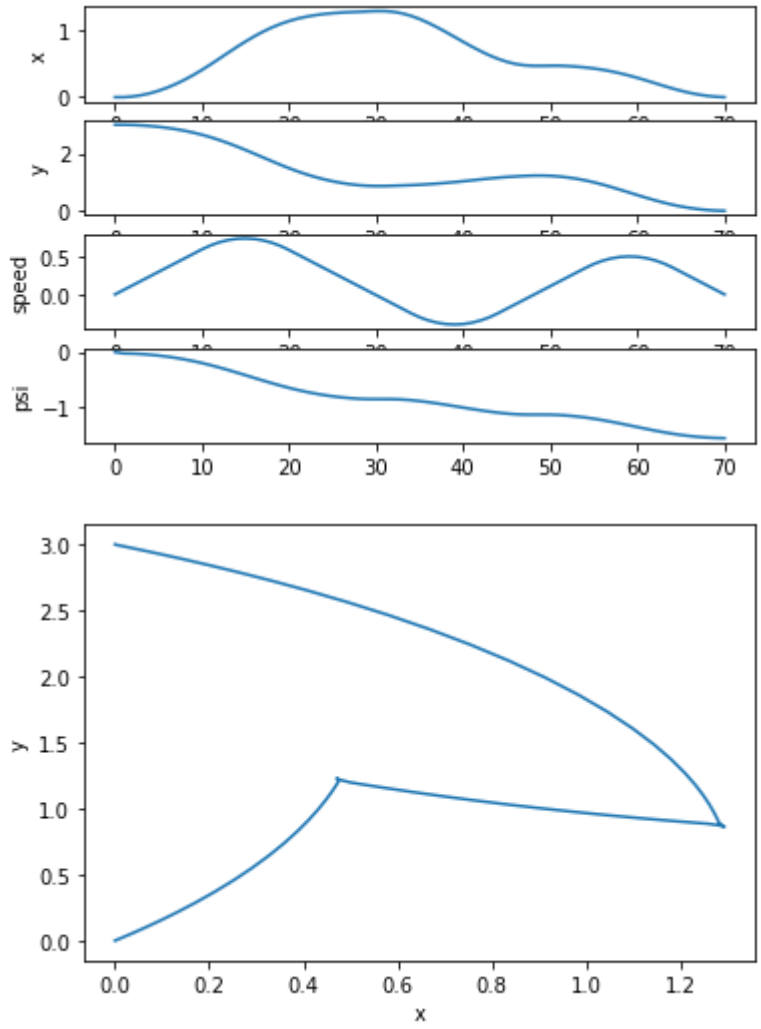
In [3]:
```
# simulation parameters
Ts = 0.2
N = 70
TFinal = Ts*N

nz = 4        # number of states
nu = 2         # number of inputs

l_r   = 1.738

z0Bar = np.array([0,3,0,0])
zNBar = np.array([0,0,0,-np.pi/2])
zMax = np.array([20,10,10,2*np.pi])
zMin = np.array([-20,-0.2,-10,-2*np.pi])

model = pyo.ConcreteModel()
model.tidx = pyo.Set(initialize=range(0, N+1)) # length of finite optimization
problem
model.zidx = pyo.Set(initialize=range(0, nz))
model.uidx = pyo.Set(initialize=range(0, nu))

# Create state and input variables trajectory:
model.z = pyo.Var(model.zidx, model.tidx)
model.u = pyo.Var(model.uidx, model.tidx)

# Objective:

model.cost = pyo.Objective(expr = sum((model.z[i, t] - zNBar[i])**2 for i in m
odel.zidx for t in model.tidx if (N-3 < t) & (t < N+1)), sense=pyo.minimize)

# Constraints:

model.constraint1 = pyo.Constraint(model.zidx, rule=lambda model, i: model.z[i
, 0] == z0Bar[i])
model.constraint2 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[0
, t+1] == model.z[0, t] + Ts*(model.z[2, t]*pyo.cos(model.z[3, t] + model.u[1,
t]))
                                    if t < N else pyo.Constraint.Skip)
model.constraint3 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[1
, t+1] == model.z[1, t] + Ts*(model.z[2, t]*pyo.sin(model.z[3, t] + model.u[1,
t]))
                                    if t < N else pyo.Constraint.Skip)
model.constraint4 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[2
, t+1] == model.z[2, t] + Ts*model.u[0, t]
                                    if t < N else pyo.Constraint.Skip)
model.constraint5 = pyo.Constraint(model.tidx, rule=lambda model, t: model.z[3
, t+1] == model.z[3, t] + Ts*(model.z[2, t]/l_r*pyo.sin(model.u[1, t]))
                                    if t < N else pyo.Constraint.Skip)
model.constraint6 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[0
, t] <= 0.3
                                    if t < N else pyo.Constraint.Skip)
model.constraint7 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[0
, t] >= -0.3
                                    if t < N else pyo.Constraint.Skip)
model.constraint8 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[1
, t+1] - model.u[1, t] <= 0.2
```

```
                                    if t < N-1 else pyo.Constraint.Skip)
model.constraint9 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[1
, t+1] - model.u[1, t] >= -0.2
                                    if t < N-1 else pyo.Constraint.Skip)
model.constraint10 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[
1, t] >= -0.6
                                    if t < N else pyo.Constraint.Skip)
model.constraint11 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[
1, t] <= 0.6
                                    if t < N else pyo.Constraint.Skip)
model.constraint12 = pyo.Constraint(model.zidx, rule=lambda model, i:model.z[i
, N] == zNBar[i])

model.constraint13 = pyo.Constraint(model.zidx, model.tidx, rule=lambda model,
i, t: model.z[i, t] <= zMax[i]
                                    if t < N else pyo.Constraint.Skip)

model.constraint14 = pyo.Constraint(model.zidx, model.tidx, rule=lambda model,
i, t: model.z[i, t] >= zMin[i]
                                    if t < N else pyo.Constraint.Skip)
model.constraint15 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[
0, t+1] - model.u[0, t] >= -0.06
                                    if t < N-1 else pyo.Constraint.Skip)
model.constraint16 = pyo.Constraint(model.tidx, rule=lambda model, t: model.u[
0, t+1] - model.u[0, t] <= 0.06
                                    if t < N-1 else pyo.Constraint.Skip)

# Now we can solve:
results = pyo.SolverFactory('ipopt').solve(model).write()


# plot results
x = [pyo.value(model.z[0,0])]
y = [pyo.value(model.z[1,0])]
v = [pyo.value(model.z[2,0])]
psi = [pyo.value(model.z[3,0])]
a = [pyo.value(model.u[0,0])]
beta = [pyo.value(model.u[1,0])]

for t in model.tidx:
    if t < N:
        x.append(pyo.value(model.z[0,t+1]))
        y.append(pyo.value(model.z[1,t+1]))
        v.append(pyo.value(model.z[2,t+1]))
        psi.append(pyo.value(model.z[3,t+1]))
    if t < N-1:
        a.append(pyo.value(model.u[0,t+1]))
        beta.append(pyo.value(model.u[1,t+1]))

plt.figure()
plt.subplot(2,1,1)
plt.plot(a)
plt.ylabel('a')
plt.subplot(2,1,2)
plt.plot(beta)
plt.ylabel('beta')
```

```
plt.figure()
plt.subplot(4,1,1)
plt.plot(x)
plt.ylabel('x')
plt.subplot(4,1,2)
plt.plot(y)
plt.ylabel('y')
plt.subplot(4,1,3)
plt.plot(v)
plt.ylabel('speed')
plt.subplot(4,1,4)
plt.plot(psi)
plt.ylabel('psi')
plt.show()

plt.figure()
plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
WARNING: Loading a SolverResults object with a warning status into
    model=unknown;
        message from solver=Ipopt 3.13.2\x3a Converged to a locally infeasibl
e
        point. Problem may be infeasible.
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Lower bound: -inf
  Upper bound: inf
  Number of objectives: 1
  Number of constraints: 1404
  Number of variables: 424
  Sense: unknown
# ----------------------------------------------------------
#   Solver Information
# ----------------------------------------------------------
Solver:
- Status: warning
  Message: Ipopt 3.13.2\x3a Converged to a locally infeasible point. Problem
may be infeasible.
  Termination condition: infeasible
  Id: 200
  Error rc: 0
  Time: 2.4326138496398926
# ----------------------------------------------------------
#   Solution Information
# ----------------------------------------------------------
Solution:
- number of solutions: 0
  number of solutions displayed: 0
```





2. Unconstrained Linear Finite Time Optimal Control - Batch

Consider the discrete-time dynamic system with the following state space representation:

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} 0.77 & -0.35 \\ 0.49 & 0.91 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} + \begin{bmatrix} 0.04 \\ 0.15 \end{bmatrix} u(k)$$

We want to design a linear quadratic optimal control for this system with a finite horizon $N = 50$. We set the following cost matrices:

$$Q = \begin{bmatrix} 500 & 0 \\ 0 & 100 \end{bmatrix}, \quad R = 1, \quad P = \begin{bmatrix} 1500 & 0 \\ 0 & 100 \end{bmatrix},$$

and assume that the initial state is $x(0) = [1, -1]^T$;

Part (a)

Determine the optimal set of inputs

$$U_0 = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix}$$

through the Batch Approach, i.e. by writing the dynamic equations as follows:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} I \\ A \\ \vdots \\ \vdots \\ A^N \end{bmatrix} x(0) + \begin{bmatrix} 0 & \cdots & \cdots & 0 \\ B & 0 & \cdots & 0 \\ AB & B & \cdots & 0 \\ \vdots & \ddots & \ddots & 0 \\ A^{N-1}B & \cdots & AB & B \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ \vdots \\ u_{N-1} \end{bmatrix}$$
$$= \mathcal{S}^x x(0) + \mathcal{S}^u U_0 ,$$

and using the formula:

$$U_0^*(x(0)) = -(\mathcal{S}^{uT}\overline{Q}\mathcal{S}^u + \overline{R})^{-1}\mathcal{S}^{uT}\overline{Q}\mathcal{S}^x x(0) ,$$

and calculate the optimal cost $J_0^*(x(0))$:

$$J_0^*(x(0)) = x(0)^T(\mathcal{S}^{xT}\overline{Q}\mathcal{S}^x - \mathcal{S}^{xT}\overline{Q}\mathcal{S}^u(\mathcal{S}^{uT}\overline{Q}\mathcal{S}^u + \overline{R})^{-1}\mathcal{S}^{uT}\overline{Q}\mathcal{S}^x)x(0) .$$

In [4]:

```
import numpy as np
import scipy as sp
from scipy.linalg import block_diag
from numpy.linalg import inv


def Sx_Su(A, B, N):

    nX = np.size(A,0)
    nU = np.size(B,1)

    Sx = np.eye(nX)
    A_tmp = A
    for i in range(N):
        Sx = np.vstack((Sx, A_tmp))
        A_tmp = A_tmp @ A

    SxB = Sx @ B
    Su = np.zeros((nX*(N+1),nU*N))

    for j in range(N):
        Su_tmp = np.vstack((np.zeros((nX, nU)), SxB[:-nX,:]))
        Su[:, j] = Su_tmp.reshape(Su_tmp.shape[0], )
        SxB = Su_tmp

    return Sx, Su

def lqrBatch(A,B,Q,R,PN,N):
    Sx, Su = Sx_Su(A, B, N)
    Qbar  = sp.linalg.block_diag(np.kron(np.eye(N),Q),PN)
    Rbar = np.kron(np.eye(N),R)
    QSu = Qbar@Su
    H = Su.T@QSu+Rbar
    F = Sx.T@QSu
    K = -np.linalg.inv(H)@F.T
    P0 = F@K + Sx.T @Qbar@Sx
    return K,P0
# Solve the finite-horizon, LQR problem for a time-invariant discrete-time
# system.  A is nX-by-nX, B is nX-by-nU, (Q,R,PN) are symmetric and positive
# semidefinite (R positive-definite), and of dimension nX-by-nX, nU-by-nU,
# and nX-by-nX, respectively.   N denotes the number of time-steps.  The
# output argument K is N*nU-by-nX, so that K*x0 is the (vertically
# concatenated) sequence of optimal inputs, {u_0, u_1, ..., u_{N-1}}.  The
# optimal cost, from any initial condition x0, is x0'*P0*x0.

A = np.array([[0.77, -0.35],
              [0.49, 0.91]])
B = np.array([[0.04],
              [0.15]])
Q = np.diag((500,100))
R = 1
PN = np.diag((1500,100))
# x0 = np.array([[1],[-1]])
x0 = np.array([1,-1])
N = 5
```

```
K,P0 = lqrBatch(A,B,Q,R,PN,N)
U0_star = K@x0
J0_star = x0.T@P0@x0

print('K= ', K)
print('P0= ',P0)
print('U0_star = ',U0_star)
print('J0_star = ',J0_star)
```

```
K=  [[-2.03227264 -6.14516735]
 [-2.51745006  1.25826235]
 [-1.49314732  1.47593134]
 [-0.79280627  0.9213694 ]
 [-0.70590212  0.88342561]]
P0=  [[ 842.67961988 -299.18321182]
 [-299.18321182  433.23464536]]
U0_star =  [ 4.11289471 -3.77571241 -2.96907866 -1.71417567 -1.58932773]
J0_star =  1874.2806888890866
```

Print out $U_0^*(x(0))^T$ and $J_0^*(x(0))$. *Hint:* To efficiently concatenate the matrices use `scipy.linalg.block_diag` and `numpy.kron`.

Part (b)

Verify the results of the previous point by solving a numerical optimization problem. In fact, the cost can be written as a function of U_0 as follows:

$$\begin{aligned} J_0(x(0), U_0) &= (\mathcal{S}^x x(0) + \mathcal{S}^u U_0)^T \overline{Q} (\mathcal{S}^x x(0) + \mathcal{S}^u U_0) + U_0^T \overline{R} U_0 \\ &= U_0^T H U_0 + 2x(0)^T F U_0 + x(0)^T \mathcal{S}^{xT} \overline{Q} \mathcal{S}^x x(0), \end{aligned}$$

where $H := \mathcal{S}^{uT} \overline{Q} \mathcal{S}^u + \overline{R}$ and $F := \mathcal{S}^{xT} \overline{Q} \mathcal{S}^u$, and then minimized by solving an quadratic minimization problem. Check that the optimizer U_0^* and the optimum $J_0(x(0), U_0^*)$ correspond to the ones determined analytically in the previous point.

Note: Make sure you compute the unconstrained solution here and do not have any linear constraints.

In [5]:
```
# use cvxopt to solve quadratic program
# H and F are given and no constraint is included
import cvxopt
from cvxopt import matrix, solvers

Qbar  = sp.linalg.block_diag(np.kron(np.eye(N),Q),PN)
Rbar = np.kron(np.eye(N),R)
Sx, Su = Sx_Su(A, B, N)
QSu = Qbar@Su
H = Su.T@QSu+Rbar
F = Sx.T@QSu

P = 2*H
q = 2*x0.T@F

P = cvxopt.matrix(P, tc='d')
q = cvxopt.matrix(q.T, tc='d')
sol = cvxopt.solvers.qp(P,q)

print('U0_star =', sol['x'])
print('J0_star =', sol['primal objective'] + x0.T@(Sx.T @Qbar@Sx)@x0)
```

```
U0_star = [ 4.11e+00]
[-3.78e+00]
[-2.97e+00]
[-1.71e+00]
[-1.59e+00]

J0_star = 1874.2806888890866
```

---

3. Unconstrained Linear Finite Time Optimal Control - Recursive

Part (a)

Design the optimal controller through the recursive approach and determine the optimal state-feedback matrices F_k. Start from the Riccati Difference Equations, assuming that $P_N = P$, and compute recursively the P_k:

$$P_k = A^T P_{k+1} A + Q - A^T P_{k+1} B (B^T P_{k+1} B + R)^{-1} B^T P_{k+1} A,$$

and then calculate F_k as a function of P_{k+1}:

$$F_k = -(B^T P_{k+1} B + R)^{-1} B^T P_{k+1} A.$$

Compare the optimal cost $J_0^*(x(0)) = x(0)^T P_0 x(0)$ with question 2.a and check that they are equal.

In [6]:
```
nX = np.size(A,0)
nU = np.size(B,1)

P = np.zeros((nX,nX,N+1))
F = np.zeros((nU,nX,N))
P[:,:,N] = PN

for i in range(N-1,-1,-1):
    F[:,:,i] = -np.linalg.inv(R + B.T @ P[:,:,i+1] @ B) @ B.T @ P[:,:,i+1] @ A
    P[:,:,i] = Q + A.T @ P[:,:,i+1] @ A + A.T @ P[:,:,i+1] @ B @F[:,:,i]


Jopt_DP = x0.T @ P[:,:,0] @ x0

print('The optimal cost from the recursive approach is:')
print(Jopt_DP)
```

```
The optimal cost from the recursive approach is:
1874.2806888890864
```

Part (b)

We want to understand the effect of model uncertainty when using the two approaches in simulations (batch and recursive). Hence, let us add an additive process disturbance $Dw(k)$ to the right-hand side of dynamic equation, $x(k+1) = Ax(k) + Bu(k) + Dw(k)$. Assume the matrix D to be:

$$D = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix},$$

while the process $w(k)$ a Gaussian white noise, with mean $m = 0$ and variance $\sigma^2 = 10$.

Consider the simulation length to be equal to N time steps and that the input u_k to the system are defined as follows:

$$u_k = \begin{cases} U_0[k] & \text{for Batch Approach} \\ F_k x_k & \text{for Recursive Approach,} \end{cases}$$

where $U_0[k]$ is the k-th component of the vector U_0.

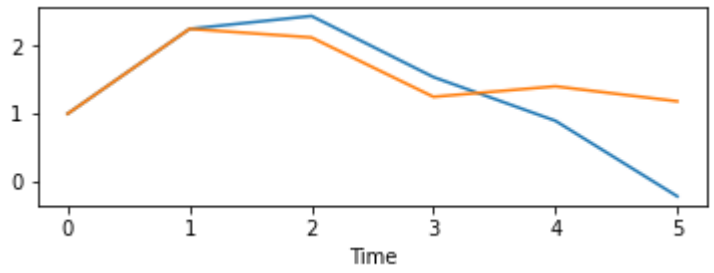
Plot a graph of the state evolution over time. What is the difference in the dynamic evolution? What happens if you modify the variance of the disturbance?

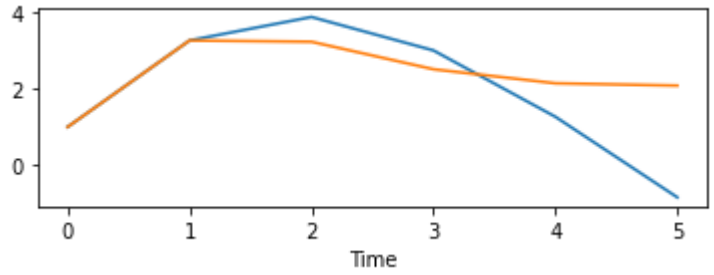
A function `sysSim` that simulates the dynamic system is given to you. This function outputs the next state $x(k+1)$ based on the current states $x(k)$, inputs $u(k)$ and the distubance $w(k)$. For the generation of the white noise, use `noise = numpy.random.normal((mean, var))`.

In [7]:

```
import matplotlib.pyplot as plt

# xNext is the next state x(k+1), xCurr and uCurr are the current state x(k) a
nd input u(k), respectively.

def sysSim(A, B, D, w, xCurr, uCurr):
    xNext = A@xCurr + B@uCurr + D@w
    return xNext

# Put your code here:
D = np.array([0.1, 0.1])
mean = 0
Uopt_batch = U0_star
t = range(N+1)

# For variance of 10
var = np.array([10])
x_batch = []
x_batch.append(x0)
x_batch_j = np.reshape((x0),2,)
x_DP = []
x_DP.append(x0)
x_DP_j = np.reshape((x0),2,)
noise = np.zeros((2),)
for j in range(N):
    noise = np.random.normal((mean, var))
    x_batch_j = sysSim(A, B, D, noise, x_batch_j, np.reshape((Uopt_batch[j]),1
,))
    x_DP_j = sysSim(A, B, D, noise, x_DP_j, np.reshape(F[:,:,j]@np.reshape((x_
DP_j),2,),1,))
    x_batch.append(x_batch_j)
    x_DP.append(x_DP_j)

x_batch_10 = (np.asarray(x_batch)).T
x_DP_10 = (np.asarray(x_DP)).T


# Double the variance
var = np.array([20])
x_batch = []
x_batch.append(x0)
x_batch_j = np.reshape((x0),2,)
x_DP = []
x_DP.append(x0)
x_DP_j = np.reshape((x0),2,)
noise = np.zeros((2),)
for j in range(N):
    noise = np.random.normal((mean, var))
    x_batch_j = sysSim(A, B, D, noise, x_batch_j, np.reshape((Uopt_batch[j]),1
,))
    x_DP_j = sysSim(A, B, D, noise, x_DP_j, np.reshape(F[:,:,j]@np.reshape((x_
DP_j),2,),1,))
    x_batch.append(x_batch_j)
    x_DP.append(x_DP_j)

x_batch_20 = (np.asarray(x_batch)).T
```

```
x_DP_20 = (np.asarray(x_DP)).T


plt.subplot(2,1,1)
plt.plot(t,x_batch_10[0,:])
plt.plot(t,x_DP_10[0,:])
plt.xlabel('Time')
plt.show()


plt.subplot(2,1,2)
plt.plot(t,x_batch_20[0,:])
plt.plot(t,x_DP_20[0,:])
plt.xlabel('Time')
plt.show()
```





4. Constrained Finite Time Optimal Control - Sparse vs Dense QP Formulations

Consider CFTOC of a discrete-time double-integrator system:

$$\begin{aligned} \min_{\substack{u_0,\dots,u_{N-1}\\ x_1,\dots,x_N}} \quad & x_N^T P x_N + \sum_{k=0}^{N-1} x_k^T Q x_k + u_k^T R u_k \\ \text{subject to} \quad & x_{k+1} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_k \\ & -1 \leq u_k \leq 1, \ k \in \{0, \dots, N-1\} \\ & \begin{bmatrix} -15 \\ -15 \end{bmatrix} \leq x(k) \leq \begin{bmatrix} 15 \\ 15 \end{bmatrix}, \ k \in \{1, \dots, N\} \end{aligned}$$

where $N = 3$, $P = Q = \mathcal{I}_{2\times 2}$, $R = 0.1$.

Part (a)

Let $x_0 = [-1, -1]^T$. Determine the QP **dense** formulation when you substitute the dynamics in the CFTOC. That is, determine H, f, c, A, b for the following problem:

$$\begin{aligned} \min_{U_0} \quad & \frac{1}{2} U_0^T H U_0 + f^T U_0 + c \\ \text{subject to} \quad & AU_0 \leq b. \end{aligned}$$

where $U_0 = \begin{bmatrix} u_0^T, u_1^T, \ldots, u_{N-1}^T \end{bmatrix}^T$. Write the code to synthesize these matrices and then compute the solution using `cvxopt`. Note that you can drop the c term when using `cvxopt`. You should NOT have any Aeq or beq. Make sure to turn in your code and outputs (synthesized matrices, solver solution and optimal value). How many 0's are in A? In H?

In [8]:

```
import numpy as np
import scipy as sp
from scipy.linalg import block_diag
from numpy.linalg import inv
import time
from ttictoc import tic,toc

# Define state and controller matricies
Ax = np.array([[1, 1],
               [0, 1]])
Bx = np.array([[0],
              [1]])
Q = np.eye(2)
P = np.eye(2)
R = 0.1
N = 3
x0 = np.array([[-1],[-1]])


# Define state and input constraints
ULlim = -1
UUlim = 1
xLlim = np.array([-15, -15])
xUlim = np.array([15, 15])

Sx, Su = Sx_Su(Ax, Bx, N)

Qbar  = sp.linalg.block_diag(np.kron(np.eye(N),Q),P)
Rbar = np.kron(np.eye(N),R)
QSu = Qbar@Su
H = Su.T@QSu+Rbar
F = Sx.T@QSu
K = -np.linalg.inv(H)@F.T
P0 = F@K + Sx.T @Qbar@Sx

c = x0.T@Sx.T@Qbar@Sx@x0

A = np.concatenate([np.kron(np.array([[1],[-1]]),np.eye(3)), Su, -Su], axis =
0)
b = np.concatenate([np.ones((nX*N,1)), 15*np.ones((2*nX*(N+1),1))-np.concatena
te([Sx,-Sx], axis = 0)@x0], axis = 0)

P = 2*H
q = 2*x0.T@F

P = cvxopt.matrix(P, tc='d')
q = cvxopt.matrix(q.T, tc='d')
G = cvxopt.matrix(A, tc='d')
h = cvxopt.matrix(b, tc='d')

tic()
sol = cvxopt.solvers.qp(P,q,G,h)
t_dense = toc()

Uopt_dense = sol['x']
Jopt_dense = sol['primal objective']+c
```

```
print('The optimal control sequence=', Uopt_dense)
print('The optimal cost=', Jopt_dense)
print('The H matrix=', H)
print('The F matrix=', F)
print('The c matrix=', c)
print('The A matrix=', A)
print('The b matrix=', b)
print('Number of zeros in A matrix=', np.size(A,0)*np.size(A,1) - np.count_non
zero(A))
print('Number of zeros in H matrix=', np.size(H,0)*np.size(H,1) - np.count_non
zero(H))
```

```
     pcost       dcost       gap    pres   dres
 0: -2.4779e+01 -8.2232e+02  1e+03  1e-01  7e-16
 1: -1.9633e+01 -8.5580e+01  7e+01  3e-03  2e-16
 2: -2.0823e+01 -2.6044e+01  5e+00  2e-04  1e-16
 3: -2.1589e+01 -2.2124e+01  5e-01  1e-05  3e-16
 4: -2.1717e+01 -2.1764e+01  5e-02  4e-08  3e-16
 5: -2.1726e+01 -2.1729e+01  4e-03  2e-16  9e-17
 6: -2.1726e+01 -2.1726e+01  5e-05  1e-16  8e-17
 7: -2.1726e+01 -2.1726e+01  5e-07  2e-16  4e-17
Optimal solution found.
The optimal control sequence= [ 1.00e+00]
[ 9.13e-01]
[-8.30e-01]

The optimal cost= [[12.27427386]]
The H matrix= [[8.1 4.  1. ]
 [4.  3.1 1. ]
 [1.  1.  1.1]]
The F matrix= [[ 3.  1.  0.]
 [11.  5.  1.]]
The c matrix= [[34.]]
The A matrix= [[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]
 [-1. -0. -0.]
 [-0. -1. -0.]
 [-0. -0. -1.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 2.  1.  0.]
 [ 1.  1.  1.]
 [-0. -0. -0.]
 [-0. -0. -0.]
 [-0. -0. -0.]
 [-1. -0. -0.]
 [-1. -0. -0.]
 [-1. -1. -0.]
 [-2. -1. -0.]
 [-1. -1. -1.]]
The b matrix= [[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [16.]
 [16.]
 [17.]
 [16.]
 [18.]
 [16.]
 [19.]
 [16.]
```

```
 [14.]
 [14.]
 [13.]
 [14.]
 [12.]
 [14.]
 [11.]
 [14.]]
Number of zeros in A matrix= 42
Number of zeros in H matrix= 0
```

Part (b)

Let $x_0 = [-1, -1]^T$. Determine the QP **sparse** formulation when you do NOT substitute the dynamics in the CFTOC. That is, what are H, f, A, b, Aeq, beq for the following problem:

$$\begin{aligned} \min_{z} \quad & \frac{1}{2} z^T H z + f^T z \\ \text{subject to} \quad & Az \leq b \\ & A_{eq} z = b_{eq}. \end{aligned}$$

where $z = [x_1^T, \ldots, x_N^T, u_0^T, \ldots, u_{N-1}^T]^T$. This requires using Aeq and beq. Again, write a script to synthesize these matrices and compute the solution using `cvxopt`. Make sure to turn in the outputs (synthesized matrices, `cvxopt.solvers.qp` solution and optimal value). How many 0's are in A? In H? In Aeq?

Part (c)

Compare the `cvxopt.solvers.qp` solver times compare using `tic toc` commands. Check your solutions to make sure the two methods get the same answer.

In [9]:

```
Q = np.eye(2)
P = np.eye(2)
R = 0.1
N = 3
x0 = np.array([[-1],[-1]])

H = sp.linalg.block_diag(np.kron(np.eye(N-1),Q),P,np.kron(np.eye(N),R))

Aeq = np.concatenate([np.concatenate([np.eye(2), np.zeros((2,2)), np.zeros((2,
2)), -Bx, np.zeros((2,1)), np.zeros((2,1))], axis = 1),
       np.concatenate([-Ax, np.eye(2), np.zeros((2,2)), np.zeros((2,1)), -Bx,
np.zeros((2,1))], axis = 1),
       np.concatenate([np.zeros((2,2)), -Ax, np.eye(2), np.zeros((2,1)), np.ze
ros((2,1)), -Bx], axis = 1)], axis = 0)

beq = np.concatenate([Ax, np.zeros((2,2)), np.zeros((2,2))], axis = 0)@x0

A = sp.linalg.block_diag(np.kron(np.concatenate([np.eye(2), -np.eye(2)], axis
= 0), np.eye(3)), np.kron(np.array([[1], [-1]]), np.eye(3)))
b = np.concatenate([15*np.ones((2*nX*N,1)), np.ones((2*N,1))], axis = 0)

P = 2*H
q = np.zeros((np.size(H,1)))
c = x0.T@Q@x0

P = cvxopt.matrix(P, tc='d')
q = cvxopt.matrix(q, tc='d')
G = cvxopt.matrix(A, tc='d')
h = cvxopt.matrix(b, tc='d')
Aeq = cvxopt.matrix(Aeq, tc='d')
beq = cvxopt.matrix(beq, tc='d')

tic()
sol = cvxopt.solvers.qp(P,q,G,h,Aeq,beq)
t_sparse = toc()

Uopt_sparse = sol['x']
Jopt_sparse = sol['primal objective']+c
Uopt_sparse_array = np.asarray(Uopt_sparse)

print('Uopt_sparse=', sol['x'])
print('Jopt_sparse=', Jopt_sparse)

print('The optimal control sequence=', np.asarray(Uopt_sparse_array[6:]))
print('The optimal cost=', Jopt_sparse)
print('The H matrix=', H)
print('The F matrix=', q)
print('The A matrix=', A)
print('The b matrix=', b)
print('Number of zeros in A matrix=', np.size(A,0)*np.size(A,1) - np.count_non
zero(A))
print('Number of zeros in H matrix=', np.size(H,0)*np.size(H,1) - np.count_non
zero(H))

print('time to solve the dense formulation=', t_dense)
print('time to solve the sparse formulation=', t_sparse)
```

```
     pcost       dcost       gap    pres   dres
 0:  7.2214e+00 -6.1432e+02  9e+02  1e-01  5e-15
 1:  1.2377e+01 -4.7259e+01  6e+01  3e-03  8e-15
 2:  1.1177e+01  6.4328e+00  5e+00  2e-04  1e-15
 3:  1.0410e+01  9.9089e+00  5e-01  1e-05  5e-15
 4:  1.0283e+01  1.0237e+01  5e-02  2e-08  3e-15
 5:  1.0274e+01  1.0271e+01  3e-03  1e-16  2e-15
 6:  1.0274e+01  1.0274e+01  5e-05  2e-16  5e-16
 7:  1.0274e+01  1.0274e+01  5e-07  1e-16  2e-15
Optimal solution found.
Uopt_sparse= [-2.00e+00]
[-4.11e-10]
[-2.00e+00]
[ 9.13e-01]
[-1.09e+00]
[ 8.30e-02]
[ 1.00e+00]
[ 9.13e-01]
[-8.30e-01]

Jopt_sparse= [[12.27427386]]
The optimal control sequence= [[ 1.        ]
 [ 0.91286202]
 [-0.82987426]]
The optimal cost= [[12.27427386]]
The H matrix= [[1.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  1.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  1.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  1.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  1.  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.  0.1 0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.1]]
The F matrix= [ 0.00e+00]
[ 0.00e+00]
[ 0.00e+00]
[ 0.00e+00]
[ 0.00e+00]
[ 0.00e+00]
[ 0.00e+00]
[ 0.00e+00]

The A matrix= [[ 1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [-1. -0. -0. -0. -0. -0.  0.  0.  0.]
 [-0. -1. -0. -0. -0. -0.  0.  0.  0.]
 [-0. -0. -1. -0. -0. -0.  0.  0.  0.]
 [-0. -0. -0. -1. -0. -0.  0.  0.  0.]
 [-0. -0. -0. -0. -1. -0.  0.  0.  0.]
 [-0. -0. -0. -0. -0. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.]
```

```
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0. -1. -0. -0.]
 [ 0.  0.  0.  0.  0.  0. -0. -1. -0.]
 [ 0.  0.  0.  0.  0.  0. -0. -0. -1.]]
The b matrix= [[15.]
 [15.]
 [15.]
 [15.]
 [15.]
 [15.]
 [15.]
 [15.]
 [15.]
 [15.]
 [15.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]]
Number of zeros in A matrix= 144
Number of zeros in H matrix= 72
time to solve the dense formulation= 0.021612400000094567
time to solve the sparse formulation= 0.017535499999894455
```