North South University

Department of Electrical & Computer Engineering

Homework 2

Course Instructor: Dr. Shahnewaz Siddique
Course Code: CSE495
Course Title: Introduction to Robotics
Section: 01
Semester: Summer 2023
Date of Submission: November 5, 2023

Submitted By:

Name	ID
Ahmed Kiser	2021280042

Ans to the Ques No 1

(a)

The differential flatness method involves solving differential equations that describe the dynamics of the system. The differential flatness method is suited for continious time control and trajectory tracking. It needs to execute in real time.

On the other hand, A* method is used for planning paths. It used to searching through a grid to find the right path. The computational complexity of A* depends on the size of the search space, the presence of obstacles, and

Comparing those method, the differential flatness method requires more computational resources due to the need to solve complex

differential equations. A* method is only about searching paths and it computational demand depends on the size of grid. Therefore, differential flatness requires more resources than A* algorithm.

(b)

Differential flatness models the system as a flat output system, which enables the generation of smooth trajectories. A* is a search based path planning which excels in finding optimal paths avoiding obstacles.

Comparing those method, A* method would be better for obstacle avoidance. A* method has the ability to find the optimal path, balance path optinality

and computational cost. The differential flatness method is suited for continious time system and effective for non grid environment. A* method can find path that covers long distances and navigate through intricate obstacles, making it ideal when the robot must plan routes over extended areas, A* will be best choice for obstracle avoidance

(c)

I would use differential flatness method for movement in obstacle free 3d space.

For movement in 3d space, which was obstacle free, there is no need for grid-based path planning. As the space are obstacle free, the smoothness and

accuracy is important. Differential flatness are perfect for continious environment allowing smoother and efficient path planning. In space without obstacles, differential flatness method simplify the process. The differential flatness method allows for the generation of precise control inputs.

Finally a space without obstacles, the path planning process is simplified with differential flatness.

As I didn't choose A* method,

Drawbacks of A* method:

① A* method in open 3d space. will require unnecessary computations there was no obstacle to avoid.

② It's a grid based approach where the environment divided into nodes. But for 3d system, it was more inefficient.

③ A* method will consume more memory resources and take more time.

④ A* method resulting more computational time without benefits for 3d space which was obstacle free.

Differential method offers the advantage of continious and smooth control and more efficient for obstacle free 3d environment. A* is not the best option for discrete path planning.

Ans to the Ques No 2

Unicycle robot model represented by,

$$\dot{x}(t) = v(t)\cos\theta(t)$$
$$\dot{y}(t) = v(t)\sin\theta(t)$$
$$\dot{\theta}(t) = w(t)$$

now, expressing in polar coordinate,

$$\rho = \sqrt{x^2+y^2}$$
$$\alpha = \operatorname{atan2}(y,x) - \theta + \pi$$
$$\delta = \alpha + \theta$$

polar coordinates dynamics equation,

$$\dot{\rho}(t) = -v(t)\cos\alpha(t)$$
$$\dot{\alpha}(t) = \frac{v(t)\sin\alpha(t)}{\rho(t)} - w(t)$$
$$\dot{\delta}(t) = \frac{v(t)\sin\alpha(t)}{\rho(t)}$$

Performing Lyapunov stability analysis

$$v(\rho,\alpha,\theta) = \frac{1}{2}\rho^2 + \frac{1}{2}(\alpha^2 + k_3\delta^2)$$

Considering closed-loop control

$$v = k_1 \rho \cos\alpha$$

$$\omega = k_2 \alpha + k_1 \frac{\sin\alpha \cos\alpha}{\alpha} (\alpha + k_3 \delta)$$

$k_1, k_2, k_3 > 0$

now taking ~~tan~~ derivative of V,

$$\dot{V} = \rho \dot{\rho} + \alpha \dot{\alpha} + k_3 \delta \dot{\delta}$$

$$= \rho [-v \cos\alpha] + \alpha \left[\frac{v \sin\alpha}{\rho} - \omega\right] + k_3 \delta \frac{v \sin\alpha}{\rho}$$

$$= -k_1 \rho^2 \cos^2\alpha - k_2 \alpha^2 \Rightarrow \dot{V} < 0$$

for, $\rho \neq 0$, $\alpha \neq 0$, $\delta \neq 0$

Ans to the Ques No 3

Lyapunov stability theory:

Given a system $\dot{x} = f(x, u)$, if there exists a function $V(x, u)$ such that,

(1) $V(x, u) = 0$ for $x = 0$

(2) $V(x, u) > 0$ for $x \neq 0$

(3) $\dot{V}(x, u) < 0$ for $x \neq 0$

Then the system is stable

Given that, $\dot{x}_1 = -x_1 + x_2^3$

$\dot{x}_2 = -x_2 + u$

Lyapunov function, $V = \frac{1}{2} x_1^2 + \frac{1}{4} x_2^4$

Here, $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

for

(1) $x = 0$ then $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ for that,

$V = \frac{1}{2} x_1^2 + \frac{1}{4} x_2^4 = \frac{1}{2} 0 + \frac{1}{4} 0 = 0$

$\therefore$ for, $x=0$, $V(0,0)=0$

② for, $x_1 \neq 0$ and $x_2 = 0$

$$V(x_1, x_2) = \frac{1}{2}x_1^2 + \frac{1}{4}x_2^4$$

$$= \frac{1}{2}x_1^2 > 0$$

for, $x_1 = 0$ and $x_2 \neq 0$

$$V(x_1, x_2) = \frac{1}{2}x_1^2 + \frac{1}{4}x_2^4$$

$$= 0 + \frac{1}{4}x_2^4$$

$$= \frac{1}{4}x_2^4 > 0$$

for, $x_1 \neq 0$ and $x_2 \neq 0$

$$V(x_1, x_2) = \frac{1}{2}x_1^2 + \frac{1}{4}x_2^4 > 0$$

③ $\dot{V}(x_1, x_2) = \frac{d}{dt}\left(\frac{1}{2}x_1^2\right) + \frac{d}{dt}\left(\frac{1}{4}x_2^4\right)$

$$= \left(\frac{1}{2} \cdot 2x_1 \frac{d}{dt}x_1\right) + \frac{1}{4} \cdot 4x_2^3 \frac{d}{dt}x_2$$

$$= x_1\dot{x}_1 + x_2^3\dot{x}_2$$

from $\dot{V} = x_1(-x_1 + x_2^3) + x_2^3(-x_2 + u)$

$= -x_1^2 + x_1 x_2^3 - x_2 x_2^3 + u x_2^3$

the system $= -x_1^2 + x_1 x_2^3 - x_2^4 + u x_2^3$

to stabilize the sysem,

$-x_1^2 + x_1 x_2^3 - x_2^4 + u x_2^3 < 0$

$\Rightarrow u x_2^3 < x_1^2 - x_1 x_2^3 + x_2^4$

$\Rightarrow u < \dfrac{x_1^2 - x_1 x_2^3 + x_2^4}{x_2^3}$ the condition

that the condition to stabilize the system.

Ans to the Ques No 4

A* is a label correcting algorithm that is modified version of Dijkstra's algorithm. In Dijkstra's algorithm the goal vertex q_G is not taken into account, potentially leading to wasted effort in cases where the greedy choice make no progress towards the goal.

While Dijkstra algorithm only prioritizes a vertex q based on its cost-of-arrival $c(q)$, A* prioritizeds based on cost-of-arrival $c(q)$ plus an approximate cost-to-go $h(q)$. This provides a better estimate of the total quality of a path than just using the cost-of-arrival alone. A* algorithm is grid-based approach which is simple, and fast.

A* algorithm,

$f(q) = c(q) + h(q)$; where, $f(q)$, $c(q)$ and $h(q)$ are respectively estimated cost, cost of arrival and minimum cost to go.

Data: q_I, q_G, G

$q_I \rightarrow$ Initial/start vertex,

$q_G \rightarrow$ Goal vertex

$G \rightarrow$ Graph

Result: path

$c(q) = \infty$, $f(q) = \infty$, $\forall q$

for all vertex q, cost of arrival and cost of estimated cost are infinite.

$c(q_I) = 0$, $f(q_I) = h(q_I)$

for initial vertex q_I, cost of arrival are zero and estimated cost and minimum cost to go are equal.

$Q = \{ q_I \}$

The vertex only contain Initial vertex

While Q is not empty do

$q = \arg\min_{q' \in Q} f(q')$

if $q = q_a$ then

return path

Q.remove(q)

for $q' \in \{ q' \mid (q, q') \in E \}$ do

$\tilde{c}(q') = c(q) + c(q, q')$

if $\tilde{c}(q') < c(q')$ then

q'.parent = q

$c(q') = \tilde{c}(q')$

$f(q') = c(q') + h(q')$

if $q' \notin Q$ then

Q.add

return failure

The Algorithm shows,

① Enter starting node in graph

(ii) If the graph is empty, return failure

(iii) Select node from graph which has smallest value and if node = goal, return path.

(iv) Expend node and generate succession. Compute (g+h) for each node

(v) attach next node, to back point

(vi) go to (iv) (ii)

Cons of A* Algorithm:

(i) Resolution dependent: In that method, not guaranteed to find solution if grid resolution is not small enough.

(ii) Limited to simple robots: Grid size is exponential in the number of DOFs.

Ans to the Ques No 5

Probabilistic Road Map (PRM) is conceptually quite similar to combinatorial planneres. The PRM algorithm also generates a topological graph G called a roadmap where the vertices are configurations q in the free part of configuration space C_{free} and edges connected to the vertices.

The Algorithm

1. Randomly sample n configurations q_i from the configuration space.
2. Query a collision checker for each q_i to determine if $q_i \in C_{free}$ if $q_i \notin C_{free}$ then it is removed from the sample set.
3. Create a Graph $G = (V, E)$, with vertices from the sampled configurations $q_i \in C_{free}$.

Define a radius r and create edges for every pair of vertices q and q' where (i) $\|q - q'\| \leq r$ and (ii) the straight line path between q and q' is collision free.

PRM is a multi-query planner, which generates a roadmap (graph) G embedded in the free space.

Cons of PRM

(1) Requires a large number of samples n to sufficiently cover configuration space.

(2) Insufficient sampling may result in wrong path.

~~(3) Required large number of samples to sufficient~~

(3) PRM can be computationally intensive, especially in high-dimentional space.

Ans to the Ques No 6

Rapidly-exploring random tree (RRT) is a single query planner, which grows a tree T, rooted at the start configuration s, embedded in C_{free}. The RRT algorithm solves problem by incrementally sampling and building the graph, starting at the initial configuration q_I, until the goal configuration q_a is reached.

RRT algorithm begins by initializing a tree, $T = (V, E)$ with a vertex at initial configuration. At each iteration the RRT algorithm then performs the following steps:

1. Randomly sample a configuration $q \in C$.
2. Find the vertex $q_{near} \in V$ that is closest to the sampled configuration q.
3. Compute a new configuration q_{new} that

lies on the line connecting q_{near} and q such that the entire line from q_{near} to q_{new} is contained in the free configuration space C_{free}.

4. Add a vertex q_{new} and edge to the tree.

After a iteration only a single point is sampled and potentially added to the tree.

Cons of RRT:

① RRT can be arbitary bad with non-negligible probability.

② According to this method, generate sub optimal in only terms of length

③ Difficult for handling dynamic obstacle.

ANSWER TO THE QUESTION NO. 7

(a)

In [1]:
```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# dt = time step
dt = 0.01
ti= 0; tf= 15;
t = np.arange(ti, tf, dt)
t
```

Out[2]:
```
array([0.000e+00, 1.000e-02, 2.000e-02, ..., 1.497e+01, 1.498e+01,
       1.499e+01])
```

In [3]:
```
T= 15
Tsq = np.power(T,2)
Tcb = np.power(T,3)
#initialized A
A = np.array([[1, 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, 1, 0, 0],
              [1, T, Tsq, Tcb,  0, 0, 0, 0],
              [0, 1, 2*T, 3*Tsq,  0, 0, 0, 0],
              [0, 0, 0, 0,  1, T, Tsq, Tcb,],
              [0, 0, 0, 0,  0, 1, 2*T, 3*Tsq]
              ]
            )
A
```

Out[3]:
```
array([[   1,    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,    1,    0,    0],
       [   1,   15,  225, 3375,    0,    0,    0,    0],
       [   0,    1,   30,  675,    0,    0,    0,    0],
       [   0,    0,    0,    0,    1,   15,  225, 3375],
       [   0,    0,    0,    0,    0,    1,   30,  675]])
```

In [4]:
```
# A pesudo inverse
Ainv = np.linalg.pinv(A)
#initialized b
b = np.array([0, 0, 0, -0.5, 5, 0, 5, -0.5])
#matrix multiplication x = Ainv * b
x= np.matmul(Ainv, b)
Ainv
```

Out[4]:
```
array([[ 1.00000000e+00,  6.58070820e-13,  0.00000000e+00,
         0.00000000e+00, -8.64325972e-16,  3.91841507e-15,
         0.00000000e+00,  0.00000000e+00],
       [ 4.16333634e-17,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00180281e-16,  1.21430643e-17,
         0.00000000e+00,  0.00000000e+00],
       [-1.33333333e-02, -1.33333333e-01,  0.00000000e+00,
         0.00000000e+00,  1.33333333e-02, -6.66666667e-02,
         0.00000000e+00,  0.00000000e+00],
       [ 5.92592593e-04,  4.44444444e-03,  0.00000000e+00,
         0.00000000e+00, -5.92592593e-04,  4.44444444e-03,
         0.00000000e+00,  0.00000000e+00],
       [ 4.38264820e-17, -1.56199568e-17,  1.00000000e+00,
         4.13738488e-13, -4.38264820e-17,  2.19459069e-16,
         3.73236598e-16, -9.14524532e-16],
       [-3.59628264e-18, -8.65405541e-17, -1.66533454e-16,
         1.00000000e+00,  3.59628264e-18, -1.80076414e-17,
        -3.85975973e-17,  7.75421394e-16],
       [-1.03997980e-19,  1.18523646e-17, -1.33333333e-02,
        -1.33333333e-01,  1.03997980e-19, -5.20842611e-19,
         1.33333333e-02, -6.66666667e-02],
       [ 9.93178042e-21, -4.00814858e-19,  5.92592593e-04,
         4.44444444e-03, -9.93178042e-21,  4.97355363e-20,
        -5.92592593e-04,  4.44444444e-03]])
```

In [5]:
```
# Extracting coefficients
a11, a12, a13, a14, a21, a22, a23, a24 = x
x
```

Out[5]:
```
array([-4.32162986e-15, -5.00901404e-16,  6.66666667e-02, -2.96296296e-03,
       -2.04764931e-13, -5.00000000e-01,  1.66666667e-01, -7.40740741e-03])
```

In [6]:
```
# Calculating X and Y with polynomial equations
xDesired = a11 + a12 * t + a13 * t**2 + a14 * t**3
yDesired = a21 + a22 * t + a23 * t**2 + a24 * t**3
```

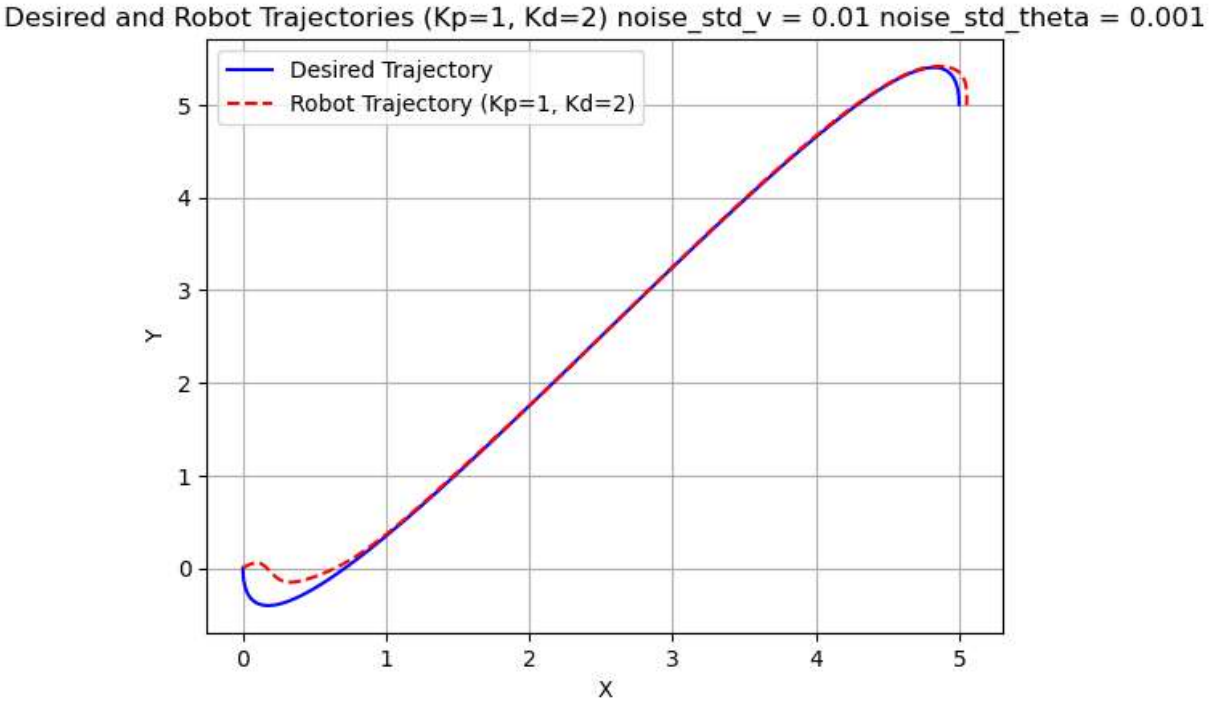
In [7]:
```
# Calculating x_double_dot and y_double_dot from the trajectory
xdd = np.gradient(np.gradient(xDesired, t), t)
ydd = np.gradient(np.gradient(yDesired, t), t)

# new states
x_new = xDesired[0]
y_new = yDesired[0]
theta = np.arctan2(np.gradient(xDesired, t)[0], np.gradient(xDesired, t)[0])
V = np.sqrt(np.gradient(xDesired, t)[0]**2 + np.gradient(yDesired, t)[0]**2)

# store robot states
x_states = [x_new]
y_states = [y_new]

noise_std_v = 0.01
noise_std_theta = 0.001
```

Kp = 1 , Kd = 2 noise_std_v = 0.01 noise_std_theta = 0.001

In [8]:
```
kp = 1
kd = 2
```

In [9]:

```python
# the unicycle equations using Euler's method
for i in range(1, len(t)):
    # Calculating error terms
    x_err = x_new - xDesired[i]
    y_err = y_new - yDesired[i]
    x_derr = V * np.cos(theta) - np.gradient(xDesired, t)[i]
    y_derr = V * np.sin(theta) - np.gradient(yDesired, t)[i]

    # Calculating control inputs using the closed-loop controller with matrix inverse
    A_inv = np.linalg.inv(np.array([
        [np.cos(theta), -V * np.sin(theta)],
        [np.sin(theta), V * np.cos(theta)]
    ]))
    b = np.array([
        xdd[i] - (1 * x_err) - (2 * x_derr),
        ydd[i] - (1 * y_err) - (2 * y_derr)

    ])
    control_inputs = np.matmul(A_inv, b)

    # Extracting control inputs
    a = control_inputs[0]
    omega = control_inputs[1]

    # Adding noise to V and theta
    V += a * (t[i] - t[i-1]) + np.random.normal(0, noise_std_v)
    theta += omega * (t[i] - t[i-1]) + np.random.normal(0, noise_std_theta)

    # Update robot states
    x_new += V * np.cos(theta) * (t[i] - t[i-1])
    y_new += V * np.sin(theta) * (t[i] - t[i-1])

    x_states.append(x_new)
    y_states.append(y_new)

# Plot the desired trajectory and robot trajectory for Kp = 4 and Kd = 4
plt.figure()
plt.plot(xDesired, yDesired, label='Desired Trajectory', color='blue')
plt.plot(x_states, y_states, label='Robot Trajectory (Kp=1, Kd=2)', linestyle='--', color='red
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Desired and Robot Trajectories (Kp=1, Kd=2) noise_std_v = 0.01 noise_std_theta = 0.0
plt.grid(True)
plt.show()
```



In [10]:
```
T= 15
Tsq = np.power(T,2)
Tcb = np.power(T,3)
#initialized A
A = np.array([[1, 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, 1, 0, 0],
              [1, T, Tsq, Tcb,  0, 0, 0, 0],
              [0, 1, 2*T, 3*Tsq,  0, 0, 0, 0],
              [0, 0, 0, 0,  1, T, Tsq, Tcb,],
              [0, 0, 0, 0,  0, 1, 2*T, 3*Tsq]
              ]
             )
# A pesudo inverse
Ainv = np.linalg.pinv(A)
#initialized b
b = np.array([0, 0, 0, -0.5, 5, 0, 5, -0.5])
#matrix multiplication x = Ainv * b
x= np.matmul(Ainv, b)
Ainv
a11, a12, a13, a14, a21, a22, a23, a24 = x
# Calculating X and Y with polynomial equations
xDesired = a11 + a12 * t + a13 * t**2 + a14 * t**3
yDesired = a21 + a22 * t + a23 * t**2 + a24 * t**3
# Calculating x_double_dot and y_double_dot from the trajectory
xdd = np.gradient(np.gradient(xDesired, t), t)
ydd = np.gradient(np.gradient(yDesired, t), t)

# new states
x_new = xDesired[0]
y_new = yDesired[0]
theta = np.arctan2(np.gradient(xDesired, t)[0], np.gradient(xDesired, t)[0])
V = np.sqrt(np.gradient(xDesired, t)[0]**2 + np.gradient(yDesired, t)[0]**2)

# store robot states
x_states = [x_new]
y_states = [y_new]

```

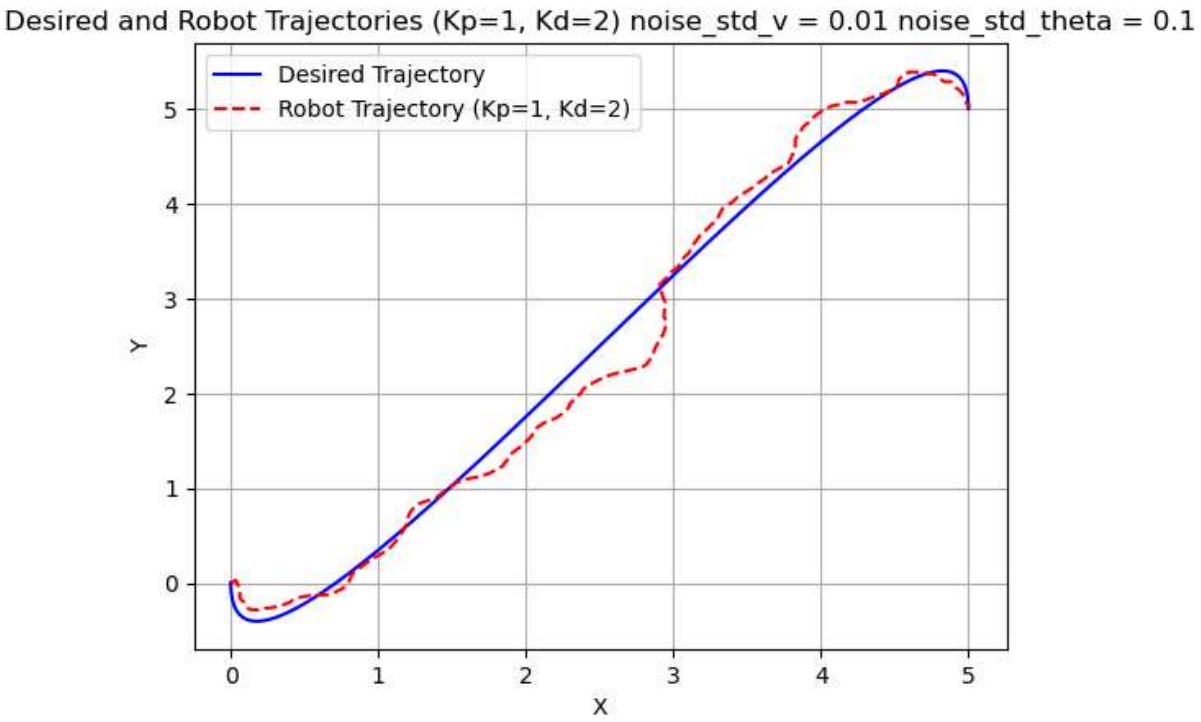
In [11]:
```
noise_std_v = 0.01
noise_std_theta = 0.1
```

In [12]:
```
kp = 1
kd = 2
```

Kp = 1 , Kd = 2 noise_std_v = 0.01 noise_std_theta = 0.1

In [13]:

```
# the unicycle equations using Euler's method
for i in range(1, len(t)):
    # Calculating error terms
    x_err = x_new - xDesired[i]
    y_err = y_new - yDesired[i]
    x_derr = V * np.cos(theta) - np.gradient(xDesired, t)[i]
    y_derr = V * np.sin(theta) - np.gradient(yDesired, t)[i]

    # Calculating control inputs using the closed-loop controller with matrix inverse
    A_inv = np.linalg.inv(np.array([
        [np.cos(theta), -V * np.sin(theta)],
        [np.sin(theta), V * np.cos(theta)]
    ]))
    b = np.array([
        xdd[i] - (4 * x_err) - (4 * x_derr),
        ydd[i] - (4 * y_err) - (4 * y_derr)

    ])
    control_inputs = np.matmul(A_inv, b)

    # Extracting control inputs
    a = control_inputs[0]
    omega = control_inputs[1]

    # Adding noise to V and theta
    V += a * (t[i] - t[i-1]) + np.random.normal(0, noise_std_v)
    theta += omega * (t[i] - t[i-1]) + np.random.normal(0, noise_std_theta)

    # Update robot states
    x_new += V * np.cos(theta) * (t[i] - t[i-1])
    y_new += V * np.sin(theta) * (t[i] - t[i-1])

    x_states.append(x_new)
    y_states.append(y_new)

# Plot the desired trajectory and robot trajectory for Kp = 4 and Kd = 4
plt.figure()
plt.plot(xDesired, yDesired, label='Desired Trajectory', color='blue')
plt.plot(x_states, y_states, label='Robot Trajectory (Kp=1, Kd=2)', linestyle='--', color='red
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Desired and Robot Trajectories (Kp=1, Kd=2) noise_std_v = 0.01 noise_std_theta = 0.1
plt.grid(True)
plt.show()
```



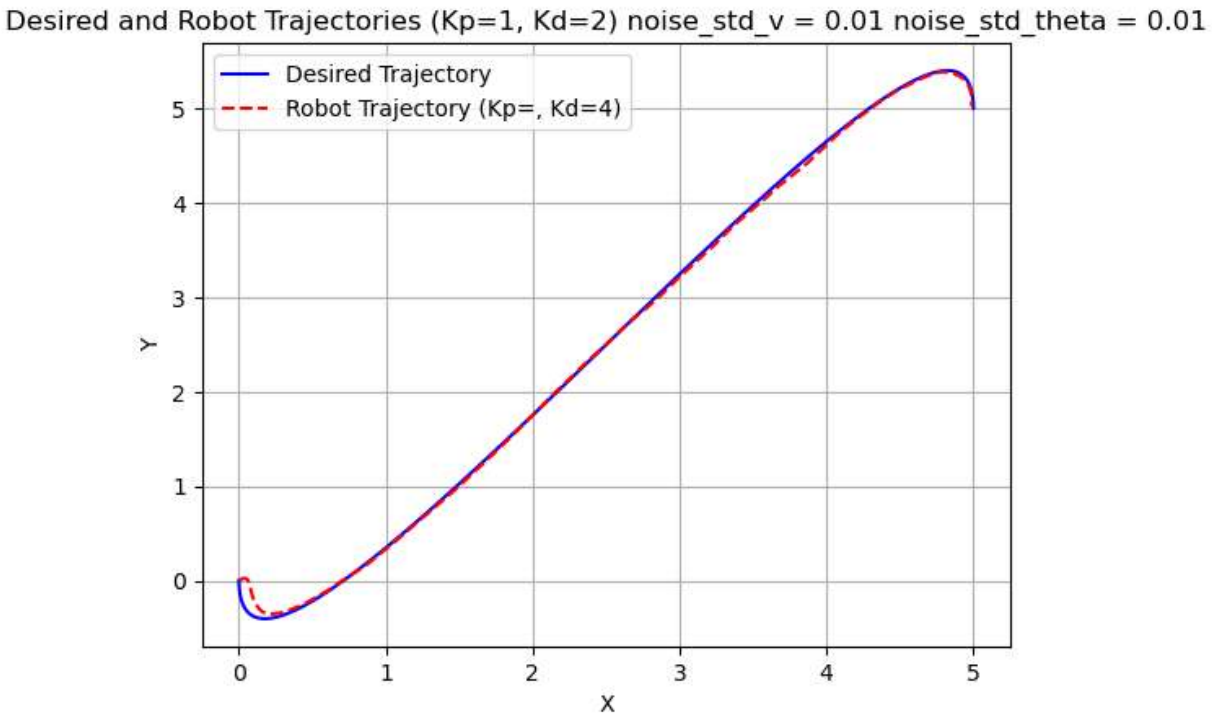
In [14]:

```
T= 15
Tsq = np.power(T,2)
Tcb = np.power(T,3)
#initialized A
A = np.array([[1, 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, 1, 0, 0],
              [1, T, Tsq, Tcb,  0, 0, 0, 0],
              [0, 1, 2*T, 3*Tsq,  0, 0, 0, 0],
              [0, 0, 0, 0,  1, T, Tsq, Tcb,],
              [0, 0, 0, 0,  0, 1, 2*T, 3*Tsq]
              ]
             )
# A pesudo inverse
Ainv = np.linalg.pinv(A)
#initialized b
b = np.array([0, 0, 0, -0.5, 5, 0, 5, -0.5])
#matrix multiplication x = Ainv * b
x= np.matmul(Ainv, b)
Ainv
a11, a12, a13, a14, a21, a22, a23, a24 = x
# Calculating X and Y with polynomial equations
xDesired = a11 + a12 * t + a13 * t**2 + a14 * t**3
yDesired = a21 + a22 * t + a23 * t**2 + a24 * t**3
# Calculating x_double_dot and y_double_dot from the trajectory
xdd = np.gradient(np.gradient(xDesired, t), t)
ydd = np.gradient(np.gradient(yDesired, t), t)

# new states
x_new = xDesired[0]
y_new = yDesired[0]
theta = np.arctan2(np.gradient(xDesired, t)[0], np.gradient(xDesired, t)[0])
V = np.sqrt(np.gradient(xDesired, t)[0]**2 + np.gradient(yDesired, t)[0]**2)

# store robot states
x_states = [x_new]
y_states = [y_new]
```

Kp = 1 , Kd = 2 noise_std_v = 0.01 noise_std_theta = 0.01

In [15]:

```
noise_std_v = 0.01
noise_std_theta = 0.01
# the unicycle equations using Euler's method
for i in range(1, len(t)):
    # Calculating error terms
    x_err = x_new - xDesired[i]
    y_err = y_new - yDesired[i]
    x_derr = V * np.cos(theta) - np.gradient(xDesired, t)[i]
    y_derr = V * np.sin(theta) - np.gradient(yDesired, t)[i]

    # Calculating control inputs using the closed-loop controller with matrix inverse
    A_inv = np.linalg.inv(np.array([
        [np.cos(theta), -V * np.sin(theta)],
        [np.sin(theta), V * np.cos(theta)]
    ]))
    b = np.array([
        xdd[i] - (4 * x_err) - (4 * x_derr),
        ydd[i] - (4 * y_err) - (4 * y_derr)

    ])
    control_inputs = np.matmul(A_inv, b)

    # Extracting control inputs
    a = control_inputs[0]
    omega = control_inputs[1]

    # Adding noise to V and theta
    V += a * (t[i] - t[i-1]) + np.random.normal(0, noise_std_v)
    theta += omega * (t[i] - t[i-1]) + np.random.normal(0, noise_std_theta)

    # Update robot states
    x_new += V * np.cos(theta) * (t[i] - t[i-1])
    y_new += V * np.sin(theta) * (t[i] - t[i-1])

    x_states.append(x_new)
    y_states.append(y_new)

# Plot the desired trajectory and robot trajectory for Kp = 4 and Kd = 4
plt.figure()
plt.plot(xDesired, yDesired, label='Desired Trajectory', color='blue')
plt.plot(x_states, y_states, label='Robot Trajectory (Kp=, Kd=4)', linestyle='--', color='red'
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Desired and Robot Trajectories (Kp=1, Kd=2) noise_std_v = 0.01 noise_std_theta = 0.(
plt.grid(True)
plt.show()
```


Desired and Robot Trajectories (Kp=1, Kd=2) noise_std_v = 0.01 noise_std_theta = 0.01
Desired Trajectory
Robot Trajectory (Kp=, Kd=4)
Y
X
0
1
2
3
4
5

ANSWER TO THE QUESTION NO. 7

(b)

In [1]:
```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# dt = time step
dt = 0.01
ti= 0; tf= 15;
t = np.arange(ti, tf, dt)
t
```

Out[2]:
```
array([0.000e+00, 1.000e-02, 2.000e-02, ..., 1.497e+01, 1.498e+01,
       1.499e+01])
```

In [3]:
```
T= 15
Tsq = np.power(T,2)
Tcb = np.power(T,3)
#initialized A
A = np.array([[1, 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, 1, 0, 0],
              [1, T, Tsq, Tcb,  0, 0, 0, 0],
              [0, 1, 2*T, 3*Tsq,  0, 0, 0, 0],
              [0, 0, 0, 0,  1, T, Tsq, Tcb,],
              [0, 0, 0, 0,  0, 1, 2*T, 3*Tsq]
              ]
             )
A
```

Out[3]:
```
array([[   1,    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,    1,    0,    0],
       [   1,   15,  225, 3375,    0,    0,    0,    0],
       [   0,    1,   30,  675,    0,    0,    0,    0],
       [   0,    0,    0,    0,    1,   15,  225, 3375],
       [   0,    0,    0,    0,    0,    1,   30,  675]])
```

In [4]:
```
# A pesudo inverse
Ainv = np.linalg.pinv(A)
#initialized b
b = np.array([0, 0, 0, -0.5, 5, 0, 5, -0.5])
#matrix multiplication x = Ainv * b
x= np.matmul(Ainv, b)
Ainv
```

Out[4]:
```
array([[ 1.00000000e+00,  6.58070820e-13,  0.00000000e+00,
         0.00000000e+00, -8.64325972e-16,  3.91841507e-15,
         0.00000000e+00,  0.00000000e+00],
       [ 4.16333634e-17,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00180281e-16,  1.21430643e-17,
         0.00000000e+00,  0.00000000e+00],
       [-1.33333333e-02, -1.33333333e-01,  0.00000000e+00,
         0.00000000e+00,  1.33333333e-02, -6.66666667e-02,
         0.00000000e+00,  0.00000000e+00],
       [ 5.92592593e-04,  4.44444444e-03,  0.00000000e+00,
         0.00000000e+00, -5.92592593e-04,  4.44444444e-03,
         0.00000000e+00,  0.00000000e+00],
       [ 4.38264820e-17, -1.56199568e-17,  1.00000000e+00,
         4.13738488e-13, -4.38264820e-17,  2.19459069e-16,
         3.73236598e-16, -9.14524532e-16],
       [-3.59628264e-18, -8.65405541e-17, -1.66533454e-16,
         1.00000000e+00,  3.59628264e-18, -1.80076414e-17,
        -3.85975973e-17,  7.75421394e-16],
       [-1.03997980e-19,  1.18523646e-17, -1.33333333e-02,
        -1.33333333e-01,  1.03997980e-19, -5.20842611e-19,
         1.33333333e-02, -6.66666667e-02],
       [ 9.93178042e-21, -4.00814858e-19,  5.92592593e-04,
         4.44444444e-03, -9.93178042e-21,  4.97355363e-20,
        -5.92592593e-04,  4.44444444e-03]])
```

In [5]:
```
# Extracting coefficients
a11, a12, a13, a14, a21, a22, a23, a24 = x
x
```

Out[5]:
```
array([-4.32162986e-15, -5.00901404e-16,  6.66666667e-02, -2.96296296e-03,
       -2.04764931e-13, -5.00000000e-01,  1.66666667e-01, -7.40740741e-0
3])
```

In [6]:
```
# Calculating X and Y with polynomial equations
xDesired = a11 + a12 * t + a13 * t**2 + a14 * t**3
yDesired = a21 + a22 * t + a23 * t**2 + a24 * t**3
```

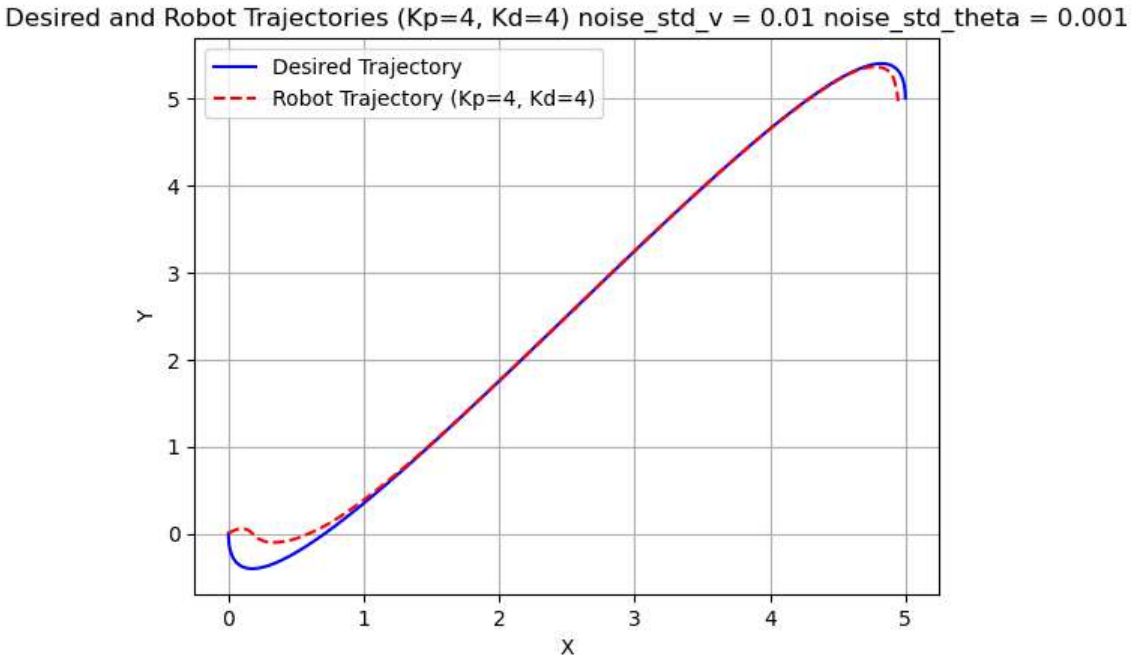
In [7]:
```
# Calculating x_double_dot and y_double_dot from the trajectory
xdd = np.gradient(np.gradient(xDesired, t), t)
ydd = np.gradient(np.gradient(yDesired, t), t)

# new states
x_new = xDesired[0]
y_new = yDesired[0]
theta = np.arctan2(np.gradient(xDesired, t)[0], np.gradient(xDesired, t
V = np.sqrt(np.gradient(xDesired, t)[0]**2 + np.gradient(yDesired, t)[0

# store robot states
x_states = [x_new]
y_states = [y_new]

noise_std_v = 0.01
noise_std_theta = 0.001
```

Kp = 4 , Kd = 4 noise_std_v = 0.01 noise_std_theta = 0.001

In [8]:
```
kp = 4
kd = 4
```

In [9]:

```
# the unicycle equations using Euler's method
for i in range(1, len(t)):
    # Calculating error terms
    x_err = x_new - xDesired[i]
    y_err = y_new - yDesired[i]
    x_derr = V * np.cos(theta) - np.gradient(xDesired, t)[i]
    y_derr = V * np.sin(theta) - np.gradient(yDesired, t)[i]

    # Calculating control inputs using the closed-loop controller with
    A_inv = np.linalg.inv(np.array([
        [np.cos(theta), -V * np.sin(theta)],
        [np.sin(theta), V * np.cos(theta)]
    ]))
    b = np.array([
        xdd[i] - (1 * x_err) - (2 * x_derr),
        ydd[i] - (1 * y_err) - (2 * y_derr)

    ])
    control_inputs = np.matmul(A_inv, b)

    # Extracting control inputs
    a = control_inputs[0]
    omega = control_inputs[1]

    # Adding noise to V and theta
    V += a * (t[i] - t[i-1]) + np.random.normal(0, noise_std_v)
    theta += omega * (t[i] - t[i-1]) + np.random.normal(0, noise_std_th

    # Update robot states
    x_new += V * np.cos(theta) * (t[i] - t[i-1])
    y_new += V * np.sin(theta) * (t[i] - t[i-1])

    x_states.append(x_new)
    y_states.append(y_new)

# Plot the desired trajectory and robot trajectory for Kp = 4 and Kd =
plt.figure()
plt.plot(xDesired, yDesired, label='Desired Trajectory', color='blue')
plt.plot(x_states, y_states, label='Robot Trajectory (Kp=4, Kd=4)', lin
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Desired and Robot Trajectories (Kp=4, Kd=4) noise_std_v = 0.
plt.grid(True)
plt.show()
```


Desired and Robot Trajectories (Kp=4, Kd=4) noise_std_v = 0.01 noise_std_theta = 0.001
Desired Trajectory
Robot Trajectory (Kp=4, Kd=4)
Y
X
0
1
2
3
4
5

In [10]:
```
T= 15
Tsq = np.power(T,2)
Tcb = np.power(T,3)
#initialized A
A = np.array([[1, 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, 1, 0, 0],
              [1, T, Tsq, Tcb,  0, 0, 0, 0],
              [0, 1, 2*T, 3*Tsq,  0, 0, 0, 0],
              [0, 0, 0, 0,  1, T, Tsq, Tcb,],
              [0, 0, 0, 0,  0, 1, 2*T, 3*Tsq]
              ]
             )
# A pesudo inverse
Ainv = np.linalg.pinv(A)
#initialized b
b = np.array([0, 0, 0, -0.5, 5, 0, 5, -0.5])
#matrix multiplication x = Ainv * b
x= np.matmul(Ainv, b)
Ainv
a11, a12, a13, a14, a21, a22, a23, a24 = x
# Calculating X and Y with polynomial equations
xDesired = a11 + a12 * t + a13 * t**2 + a14 * t**3
yDesired = a21 + a22 * t + a23 * t**2 + a24 * t**3
# Calculating x_double_dot and y_double_dot from the trajectory
xdd = np.gradient(np.gradient(xDesired, t), t)
ydd = np.gradient(np.gradient(yDesired, t), t)

# new states
x_new = xDesired[0]
y_new = yDesired[0]
theta = np.arctan2(np.gradient(xDesired, t)[0], np.gradient(xDesired, t
V = np.sqrt(np.gradient(xDesired, t)[0]**2 + np.gradient(yDesired, t)[0

# store robot states
x_states = [x_new]
y_states = [y_new]

```

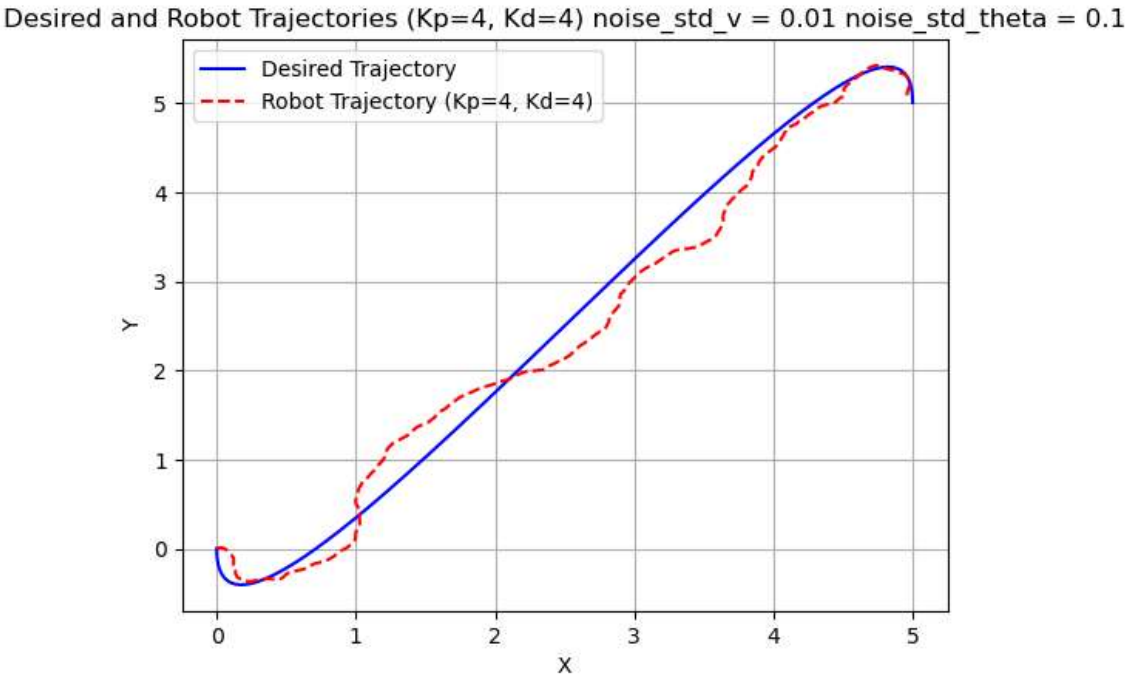
In [11]:
```
noise_std_v = 0.01
noise_std_theta = 0.1
```

In [12]:
```
kp = 4
kd = 4
```

Kp = 4 , Kd = 4 noise_std_v = 0.01 noise_std_theta = 0.1

In [13]:

```
# the unicycle equations using Euler's method
for i in range(1, len(t)):
    # Calculating error terms
    x_err = x_new - xDesired[i]
    y_err = y_new - yDesired[i]
    x_derr = V * np.cos(theta) - np.gradient(xDesired, t)[i]
    y_derr = V * np.sin(theta) - np.gradient(yDesired, t)[i]

    # Calculating control inputs using the closed-loop controller with
    A_inv = np.linalg.inv(np.array([
        [np.cos(theta), -V * np.sin(theta)],
        [np.sin(theta), V * np.cos(theta)]
    ]))
    b = np.array([
        xdd[i] - (4 * x_err) - (4 * x_derr),
        ydd[i] - (4 * y_err) - (4 * y_derr)

    ])
    control_inputs = np.matmul(A_inv, b)

    # Extracting control inputs
    a = control_inputs[0]
    omega = control_inputs[1]

    # Adding noise to V and theta
    V += a * (t[i] - t[i-1]) + np.random.normal(0, noise_std_v)
    theta += omega * (t[i] - t[i-1]) + np.random.normal(0, noise_std_th

    # Update robot states
    x_new += V * np.cos(theta) * (t[i] - t[i-1])
    y_new += V * np.sin(theta) * (t[i] - t[i-1])

    x_states.append(x_new)
    y_states.append(y_new)

# Plot the desired trajectory and robot trajectory for Kp = 4 and Kd =
plt.figure()
plt.plot(xDesired, yDesired, label='Desired Trajectory', color='blue')
plt.plot(x_states, y_states, label='Robot Trajectory (Kp=4, Kd=4)', lin
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Desired and Robot Trajectories (Kp=4, Kd=4) noise_std_v = 0.
plt.grid(True)
plt.show()
```


Desired and Robot Trajectories (Kp=4, Kd=4) noise_std_v = 0.01 noise_std_theta = 0.1
Desired Trajectory
Robot Trajectory (Kp=4, Kd=4)
Y
X
0
1
2
3
4
5

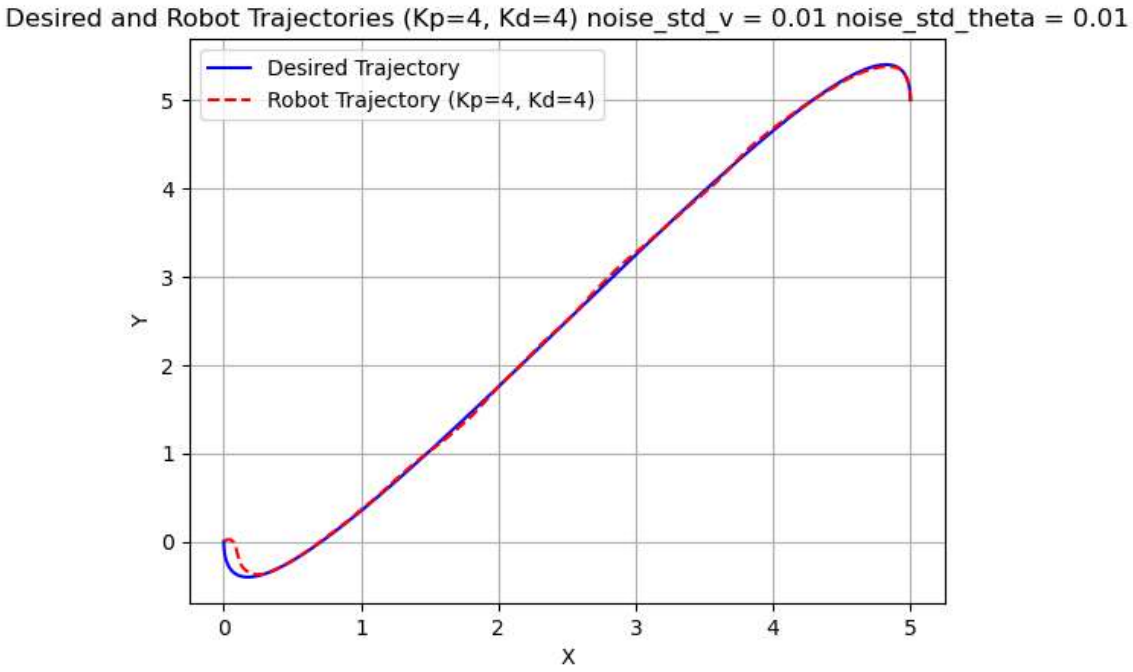
In [14]:

```
T= 15
Tsq = np.power(T,2)
Tcb = np.power(T,3)
#initialized A
A = np.array([[1, 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, 1, 0, 0],
              [1, T, Tsq, Tcb,  0, 0, 0, 0],
              [0, 1, 2*T, 3*Tsq,  0, 0, 0, 0],
              [0, 0, 0, 0,  1, T, Tsq, Tcb,],
              [0, 0, 0, 0,  0, 1, 2*T, 3*Tsq]
              ]
             )
# A pesudo inverse
Ainv = np.linalg.pinv(A)
#initialized b
b = np.array([0, 0, 0, -0.5, 5, 0, 5, -0.5])
#matrix multiplication x = Ainv * b
x= np.matmul(Ainv, b)
Ainv
a11, a12, a13, a14, a21, a22, a23, a24 = x
# Calculating X and Y with polynomial equations
xDesired = a11 + a12 * t + a13 * t**2 + a14 * t**3
yDesired = a21 + a22 * t + a23 * t**2 + a24 * t**3
# Calculating x_double_dot and y_double_dot from the trajectory
xdd = np.gradient(np.gradient(xDesired, t), t)
ydd = np.gradient(np.gradient(yDesired, t), t)

# new states
x_new = xDesired[0]
y_new = yDesired[0]
theta = np.arctan2(np.gradient(xDesired, t)[0], np.gradient(xDesired, t
V = np.sqrt(np.gradient(xDesired, t)[0]**2 + np.gradient(yDesired, t)[0

# store robot states
x_states = [x_new]
y_states = [y_new]
```

Kp = 4 , Kd = 4 noise_std_v = 0.01 noise_std_theta = 0.01

In [15]:

```
noise_std_v = 0.01
noise_std_theta = 0.01
# the unicycle equations using Euler's method
for i in range(1, len(t)):
    # Calculating error terms
    x_err = x_new - xDesired[i]
    y_err = y_new - yDesired[i]
    x_derr = V * np.cos(theta) - np.gradient(xDesired, t)[i]
    y_derr = V * np.sin(theta) - np.gradient(yDesired, t)[i]

    # Calculating control inputs using the closed-loop controller with
    A_inv = np.linalg.inv(np.array([
        [np.cos(theta), -V * np.sin(theta)],
        [np.sin(theta), V * np.cos(theta)]
    ]))
    b = np.array([
        xdd[i] - (4 * x_err) - (4 * x_derr),
        ydd[i] - (4 * y_err) - (4 * y_derr)

    ])
    control_inputs = np.matmul(A_inv, b)

    # Extracting control inputs
    a = control_inputs[0]
    omega = control_inputs[1]

    # Adding noise to V and theta
    V += a * (t[i] - t[i-1]) + np.random.normal(0, noise_std_v)
    theta += omega * (t[i] - t[i-1]) + np.random.normal(0, noise_std_th

    # Update robot states
    x_new += V * np.cos(theta) * (t[i] - t[i-1])
    y_new += V * np.sin(theta) * (t[i] - t[i-1])

    x_states.append(x_new)
    y_states.append(y_new)

# Plot the desired trajectory and robot trajectory for Kp = 4 and Kd =
plt.figure()
plt.plot(xDesired, yDesired, label='Desired Trajectory', color='blue')
plt.plot(x_states, y_states, label='Robot Trajectory (Kp=4, Kd=4)', lin
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Desired and Robot Trajectories (Kp=4, Kd=4) noise_std_v = 0.
plt.grid(True)
plt.show()
```


Desired and Robot Trajectories (Kp=4, Kd=4) noise_std_v = 0.01 noise_std_theta = 0.01
Desired Trajectory
Robot Trajectory (Kp=4, Kd=4)
Y
X
0
1
2
3
4
5