Homework 1 Report

Neeraj Laul

Problem 1:

1.1 Rear Wheel Drive Modeling

Write a Python program to plot the 2D trajectory of point O on a rear-wheel drive vehicle, given the initial pose (x_i, y_i, ϕ_i), drive speed ω, steering angle α, and duration T.

Assume that all the wheels have a diameter of 0.5 m, chassis length to be 4 m, and distance between the wheels is 1.5 m (see Fig. 1). Assume that none of the wheels slip and the drive speed is split among both the wheels as the following equation:

$$\omega_{\text{left}} + \omega_{\text{right}} = 2\omega$$

Please show all your work for the derivation of the state-space model.

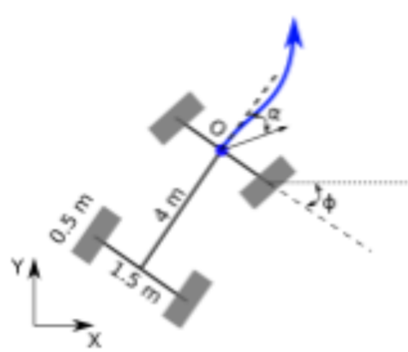

Figure 1: Rear Wheel Drive Model

Figure 1:

Answer 1:

We consider a rear-wheel driven car-like vehicle with wheel radius r, wheelbase L, and steering angle α. The forward velocity of the vehicle is defined as:

$$v = r\omega$$

where ω is the wheel's angular velocity. This relation arises from the no-slip rolling condition: the linear velocity at the contact point of a rolling wheel equals its angular velocity multiplied by its radius. Hence, as the wheel rotates with angular velocity ω, a point on the rim moves forward with velocity $v = r\omega$.

The problem statement also provides the relationship between the left and right wheel angular velocities:

$$\omega_{\text{left}} + \omega_{\text{right}} = 2\omega$$

This equation indicates that the overall forward motion of the vehicle (i.e., its average drive speed) is based on the mean of the left and right wheel rotational speeds. In a rear-wheel drive configuration, this averaged angular velocity ω determines the translational speed of the vehicle through $v = r\omega$. When both wheels rotate at the same rate, the vehicle moves straight; when they differ (due to steering), the vehicle follows a curved path.

The continuous kinematic model is:

$$\dot{x} = v\cos\phi, \quad \dot{y} = v\sin\phi, \quad \dot{\phi} = \frac{v}{L}\tan\alpha$$

where ϕ represents the vehicle heading angle.

Simulation Approach

Initial conditions (x_0, y_0, ϕ_0) are provided by the user. Using Euler integration with step size Δt, the trajectory is updated as:

$$x_{k+1} = x_k + \dot{x}\,\Delta t, \quad y_{k+1} = y_k + \dot{y}\,\Delta t, \quad \phi_{k+1} = \phi_k + \dot{\phi}\,\Delta t$$

for $k = 0, 1, \ldots, N-1$, where $N = T/\Delta t$.

Code Used

```python
import numpy as np
import matplotlib.pyplot as plt
from sympy import Matrix, cos, sin, tan, symbols, lambdify


class Derivation:
    def __init__(self):
        self.WRAD = 25
        self.WSEP = 150
        self.WBASE = 400
        self.x, self.y, self.phi = symbols("x y phi")
        self.omega, self.alpha, self.L, self.r = symbols("omega 
            alpha L r")
        self.state = None
        self.x0, self.y0, self.phi0 = None, None, None
```

```python
    def start(self):
        vals = input("Enter Initial Vals")
        x0, y0, phi0 = map(float, vals.split(","))
        self.x0, self.y0, self.phi0 = x0, y0, phi0
        T = 100
        dt = 0.1

    def eqn(self):
        v = self.r * self.omega
        dx = v * cos(self.phi)
        dy = v * sin(self.phi)
        dphi = (v / self.L) * tan(self.alpha)
        self.state = Matrix([dx, dy, dphi])


class Sim:
    def __init__(self, der):
        self.der = der
        self.f = lambdify(
            (self.der.r, self.der.omega, self.der.phi, self.der.L,
                self.der.alpha),
            self.der.state,
            "numpy",
        )

    def plots(self, omega_val, alpha_val, T=10, dt=0.1):
        r_val = self.der.WRAD / 100.0
        L_val = self.der.WBASE / 100.0
        phi_val = self.der.phi0
        x_val, y_val = self.der.x0, self.der.y0
        N = int(T / dt) + 1
        traj = np.zeros((N, 3))
        traj[0] = [x_val, y_val, phi_val]
        for k in range(1, N):
            dx, dy, dphi = [
                val.item()
                for val in self.f(r_val, omega_val, phi_val, L_val
                    , alpha_val)
            ]
            x_val += dx * dt
            y_val += dy * dt
            phi_val += dphi * dt
            traj[k] = [x_val, y_val, phi_val]
        plt.plot(traj[:, 0], traj[:, 1])
        plt.xlabel("X")
```

```python
        plt.ylabel("Y")
        plt.axis("equal")
        plt.show()


if __name__ == "__main__":
    der = Derivation()
    der.start()
    der.eqn()
    sim = Sim(der)

    choice = input("Use manual values (M) or random values (R)? ")
        .strip().lower()
    if choice == "m":
        omega_val = float(input("Enter omega (wheel angular
            velocity): "))
        alpha_val = float(input("Enter alpha (steering angle in
            radians): "))
    else:
        omega_val = np.random.uniform(0, 5.0)  # random omega in
            [0.5, 5.0]
        alpha_val = np.random.uniform(-5, 5)  # random alpha in
            [-0.5, 0.5]
        print(
            f"Randomly chosen values -> omega: {omega_val:.2f},
                alpha: {alpha_val:.2f}"
        )

    sim.plots(omega_val=omega_val, alpha_val=alpha_val)
```

Verification and Results

To verify correctness, two simulation cases were tested using known values of ω and α:

- **Straight line motion:** For $\alpha = 0$, the steering is neutral, $\tan\alpha = 0$, and $\dot{\phi} = 0$, meaning the vehicle maintains its initial heading. The simulation parameters were:

$$\omega = 2.0, \quad \alpha = 0.0, \quad T = 100$$

  This produced a straight trajectory along the x-axis, confirming correct linear motion.

- **Circular motion:** When α is constant and nonzero, the vehicle follows a circular trajectory with radius:

$$R = \frac{L}{\tan\alpha}$$

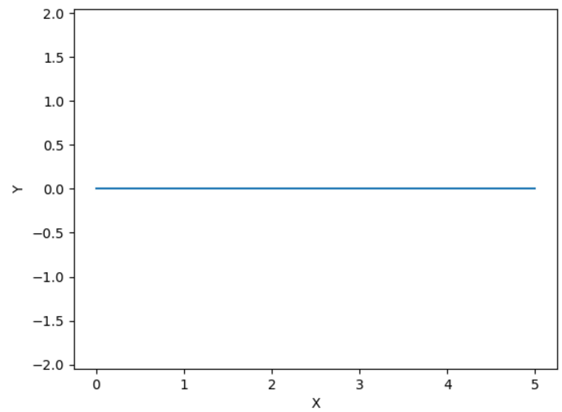

Figure 2: Straight Line Verification

Using the following parameters:

$$\omega = 20.0, \quad \alpha = 0.7854 \text{ radians } (45^\circ), \quad T = 100$$

the vehicle completed a full circular loop. The simulation results matched the analytical expectation of circular motion, verifying the correctness of the derived state-space model and numerical implementation.

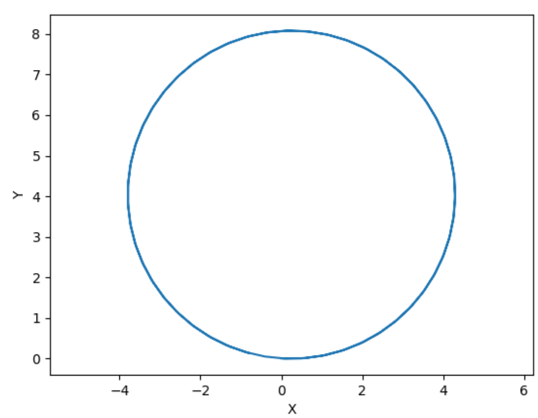

Figure 3: Circular Motion Verification

Problem 2:

1.2 Derive the Kinematics Equations for a 3-DOF Manipulator Using Geometrical Method

Consider a serial manipulator with 3 links connected by revolute joints as shown in Fig. 2, with the link lengths l_1, l_2, l_3.

1. Derive the *(position and velocity) forward kinematics* equations, given joint angles $\theta_1, \theta_2, \theta_3$ and joint velocities.
2. Derive the *(velocity) inverse kinematics* equations in matrix format using the geometrical method, given velocities of the end-effector $\dot{x}, \dot{y}, \dot{\phi}$ and joint angles $\theta_1, \theta_2, \theta_3$. Use Python's `SymPy` library to take derivatives.

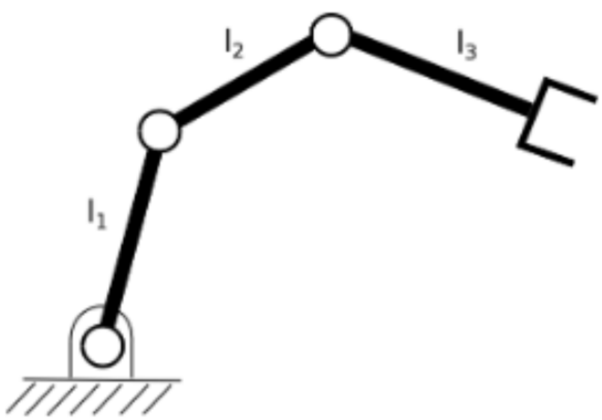

Figure 2: 3-link serial manipulator

Figure 4:

Answer 2:

Forward Kinematics

The end-effector position and orientation are:

$$x = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3)$$

$$y = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\theta_1 + \theta_2 + \theta_3)$$

$$\phi = \theta_1 + \theta_2 + \theta_3$$

Thus,

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ \phi \end{bmatrix}$$

Velocity Kinematics

Differentiating,

$$\dot{\mathbf{p}} = J(\theta_1, \theta_2, \theta_3) \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix}$$

Jacobian

$$J = \begin{bmatrix} -l_1 \sin\theta_1 - l_2 \sin(\theta_1+\theta_2) - l_3 \sin(\theta_1+\theta_2+\theta_3) & -l_2 \sin(\theta_1+\theta_2) - l_3 \sin(\theta_1+\theta_2+\theta_3) & -l_3 \sin(\theta_1 + \theta \\ l_1 \cos\theta_1 + l_2 \cos(\theta_1+\theta_2) + l_3 \cos(\theta_1+\theta_2+\theta_3) & l_2 \cos(\theta_1+\theta_2) + l_3 \cos(\theta_1+\theta_2+\theta_3) & l_3 \cos(\theta_1 + \theta \\ 1 & 1 & 1 \end{bmatrix}$$

Inverse Velocity Kinematics

$$\begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix} = J^{-1} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \end{bmatrix}$$

Code Used

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Symbols
theta1, theta2, theta3 = sp.symbols("theta1 theta2 theta3")
l1, l2, l3 = sp.symbols("l1 l2 l3")
theta1_dot, theta2_dot, theta3_dot = sp.symbols("theta1_dot
    theta2_dot theta3_dot")

# Forward kinematics (position)
x = (
     l1 * sp.cos(theta1)
    + l2 * sp.cos(theta1 + theta2)
    + l3 * sp.cos(theta1 + theta2 + theta3)
)
y = (
     l1 * sp.sin(theta1)
    + l2 * sp.sin(theta1 + theta2)
    + l3 * sp.sin(theta1 + theta2 + theta3)
)
phi = theta1 + theta2 + theta3
pos = sp.Matrix([x, y, phi])
```

```python
# Jacobian (velocity kinematics)
q = sp.Matrix([theta1, theta2, theta3])
q_dot = sp.Matrix([theta1_dot, theta2_dot, theta3_dot])
J = pos.jacobian(q)
vel = J * q_dot

# Inverse velocity kinematics
xdot, ydot, phidot = sp.symbols("xdot ydot phidot")
end_effector_vel = sp.Matrix([xdot, ydot, phidot])
qdot_from_task = J.inv() * end_effector_vel

# Display results
print("Forward Kinematics (Position):")
sp.pprint(pos)

print("\nVelocity Kinematics:")
sp.pprint(vel)

print("\nJacobian Matrix:")
sp.pprint(J)

print("\nInverse Velocity Kinematics (Joint velocities):")
sp.pprint(qdot_from_task)

# Determinant of Jacobian
det_J = J.det()
print("\nDeterminant of Jacobian:")
sp.pprint(det_J)

# Check if the Jacobian is invertible
if det_J != 0:
    print("\nJacobian is invertible.")
else:
    print("\nJacobian is singular, cannot compute inverse.")

# Simplify symbolic outputs
print("\nSimplified Forward Kinematics (Position):")
sp.pprint(sp.simplify(pos))

print("\nSimplified Velocity Kinematics:")
sp.pprint(sp.simplify(vel))

print("\nSimplified Inverse Velocity Kinematics (Joint velocities)
    :")
sp.pprint(sp.simplify(qdot_from_task))
```

```python
# Define specific numerical values
values = {
    theta1: sp.pi / 2,
    theta2: sp.pi / 2,
    theta3: sp.pi / 2,
    l1: 1.0,
    l2: 1.0,
    l3: 0.5,
    xdot: 0.1,
    ydot: 0.1,
    phidot: 0.05,
}

# Evaluate forward kinematics numerically
pos_numeric = pos.evalf(subs=values)
print("\nNumerical Forward Kinematics (Position):")
sp.pprint(pos_numeric)

# Evaluate Jacobian numerically
J_numeric = J.evalf(subs=values)
print("\nNumerical Jacobian Matrix:")
sp.pprint(J_numeric)

det_val = J_numeric.det()
print("\nDeterminant of numeric Jacobian:", det_val)

try:
    J_inv_numeric = J_numeric.inv()
    end_effector_vel_numeric = end_effector_vel.evalf(subs=values)
    qdot_numeric = J_inv_numeric * end_effector_vel_numeric
    print("\nNumerical Joint Velocities from Inverse Velocity
        Kinematics:")
    sp.pprint(qdot_numeric)
except:
    print("Jacobian is singular at this configuration. Skipping
        inverse kinematics.")

theta1_val = float(values[theta1])
theta2_val = float(values[theta2])
theta3_val = float(values[theta3])

l1_val, l2_val, l3_val = values[l1], values[l2], values[l3]

# Joint coordinates
```

```python
x0, y0 = 0, 0
x1, y1 = l1_val * np.cos(theta1_val), l1_val * np.sin(theta1_val)
x2, y2 = x1 + l2_val * np.cos(theta1_val + theta2_val), y1 +
    l2_val * np.sin(theta1_val + theta2_val)
x3, y3 = x2 + l3_val * np.cos(theta1_val + theta2_val + theta3_val
    ), y2 + l3_val * np.sin(theta1_val + theta2_val + theta3_val)

plt.figure()
plt.plot([x0, x1], [y0, y1], "r-", marker="o", linewidth=2, label=
    "Link 1")
plt.plot([x1, x2], [y1, y2], "g-", marker="o", linewidth=2, label=
    "Link 2")
plt.plot([x2, x3], [y2, y3], "b-", marker="o", linewidth=2, label=
    "Link 3")
plt.title("Manipulator Configuration")
plt.xlabel("X")
plt.ylabel("Y")
plt.axis("equal")
plt.grid(True)
plt.legend()
plt.show()

# ------------------------------------------------
# Plot 2: End-effector trajectory
# ------------------------------------------------
theta1_vals = np.linspace(0, np.pi / 2, 50)
traj_x, traj_y = [], []

for th1 in theta1_vals:
    x_val = (
        l1_val * np.cos(th1)
        + l2_val * np.cos(th1 + theta2_val)
        + l3_val * np.cos(th1 + theta2_val + theta3_val)
    )
    y_val = (
        l1_val * np.sin(th1)
        + l2_val * np.sin(th1 + theta2_val)
        + l3_val * np.sin(th1 + theta2_val + theta3_val)
    )
    traj_x.append(x_val)
    traj_y.append(y_val)

plt.figure()
plt.plot(traj_x, traj_y, "r-", linewidth=2)
plt.title("End-Effector Trajectory (theta1 sweep)")
```

```
plt.xlabel("X")
plt.ylabel("Y")
plt.axis("equal")
plt.grid(True)
plt.show()

# Sweep theta2
theta2_vals = np.linspace(0, np.pi / 2, 50)
traj_x, traj_y = [], []

for th2 in theta2_vals:
    x_val = (
        l1_val * np.cos(theta1_val)
        + l2_val * np.cos(theta1_val + th2)
        + l3_val * np.cos(theta1_val + th2 + theta3_val)
    )
    y_val = (
        l1_val * np.sin(theta1_val)
        + l2_val * np.sin(theta1_val + th2)
        + l3_val * np.sin(theta1_val + th2 + theta3_val)
    )
    traj_x.append(x_val)
    traj_y.append(y_val)

plt.figure()
plt.plot(traj_x, traj_y, "g-", linewidth=2)
plt.title("End-Effector Trajectory (theta2 sweep)")
plt.xlabel("X")
plt.ylabel("Y")
plt.axis("equal")
plt.grid(True)
plt.show()

# Sweep theta3
theta3_vals = np.linspace(0, np.pi / 2, 50)
traj_x, traj_y = [], []

for th3 in theta3_vals:
    x_val = (
        l1_val * np.cos(theta1_val)
        + l2_val * np.cos(theta1_val + theta2_val)
        + l3_val * np.cos(theta1_val + theta2_val + th3)
    )
    y_val = (
        l1_val * np.sin(theta1_val)
```

```
        + l2_val * np.sin(theta1_val + theta2_val)
        + l3_val * np.sin(theta1_val + theta2_val + th3)
    )
    traj_x.append(x_val)
    traj_y.append(y_val)

plt.figure()
plt.plot(traj_x, traj_y, "b-", linewidth=2)
plt.title("End-Effector Trajectory (theta3 sweep)")
plt.xlabel("X")
plt.ylabel("Y")
plt.axis("equal")
plt.grid(True)
plt.show()
```

Results

The following figures illustrate the numerical simulation results obtained from the derived forward and inverse velocity kinematics. The manipulator configuration plot shows the spatial arrangement of the three links for the given joint angles, while the trajectory plots demonstrate the end-effector motion produced by independently varying each joint angle (θ_1, θ_2, and θ_3) within a 90° range. These visualizations confirm the correctness and physical consistency of the derived equations.

Forward Kinematics (Position):

$$\begin{bmatrix} l_1 \cdot \cos(\theta_1) + l_2 \cdot \cos(\theta_1 + \theta_2) + l_3 \cdot \cos(\theta_1 + \theta_2 + \theta_3) \\ l_1 \cdot \sin(\theta_1) + l_2 \cdot \sin(\theta_1 + \theta_2) + l_3 \cdot \sin(\theta_1 + \theta_2 + \theta_3) \\ \theta_1 + \theta_2 + \theta_3 \end{bmatrix}$$

Figure 5: Forward Kinematics(Position)

Velocity Kinematics:

$$\begin{bmatrix} -l_3 \cdot \theta\_3\_dot \cdot \sin(\theta_1 + \theta_2 + \theta_3) + \theta\_1\_dot \cdot (-l_1 \cdot \sin(\theta_1) - l_2 \cdot \sin(\theta_1 + \theta_2) - l_3 \cdot \sin(\theta_1 + \theta_2 + \theta_3)) + \theta\_2\_dot \cdot (-l_2 \cdot \sin(\theta_1 + \theta_2) - l_3 \cdot \sin(\theta_1 + \theta_2 + \theta_3)) \\ l_3 \cdot \theta\_3\_dot \cdot \cos(\theta_1 + \theta_2 + \theta_3) + \theta\_1\_dot \cdot (l_1 \cdot \cos(\theta_1) + l_2 \cdot \cos(\theta_1 + \theta_2) + l_3 \cdot \cos(\theta_1 + \theta_2 + \theta_3)) + \theta\_2\_dot \cdot (l_2 \cdot \cos(\theta_1 + \theta_2) + l_3 \cdot \cos(\theta_1 + \theta_2 + \theta_3)) \\ \theta\_1\_dot + \theta\_2\_dot + \theta\_3\_dot \end{bmatrix}$$

Figure 6: Velocity Kinematics

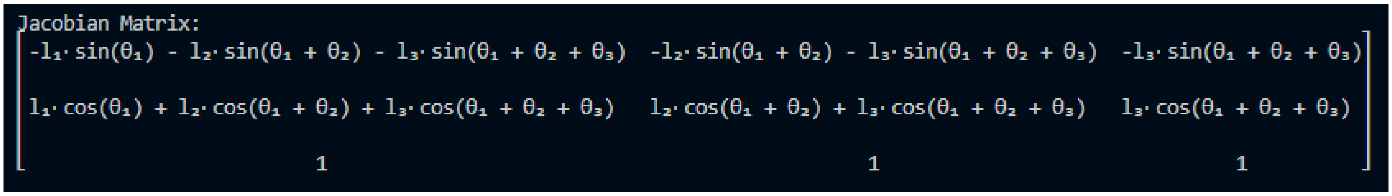

Figure 7: Jacobian Matrix

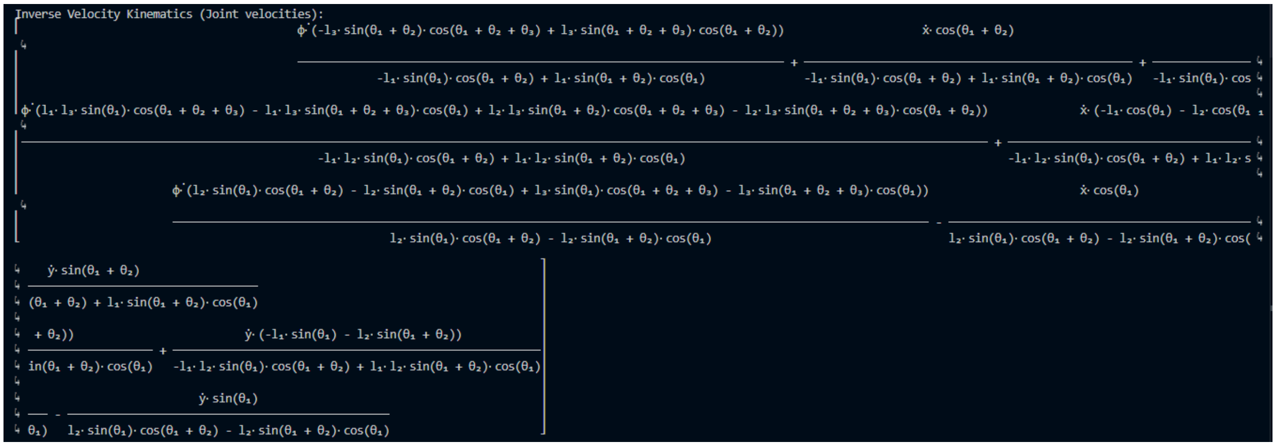

Figure 8: Inverse Kinematics

The code can also be found at the repository : Github

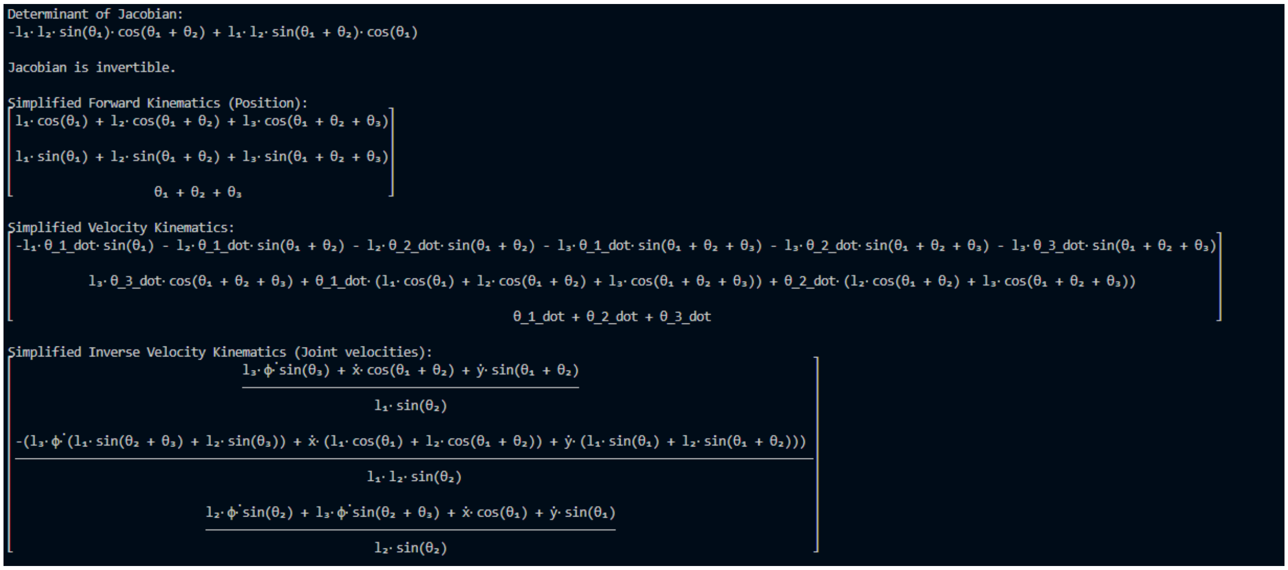

Figure 9: Proof Of Invertiblity and Simplifications

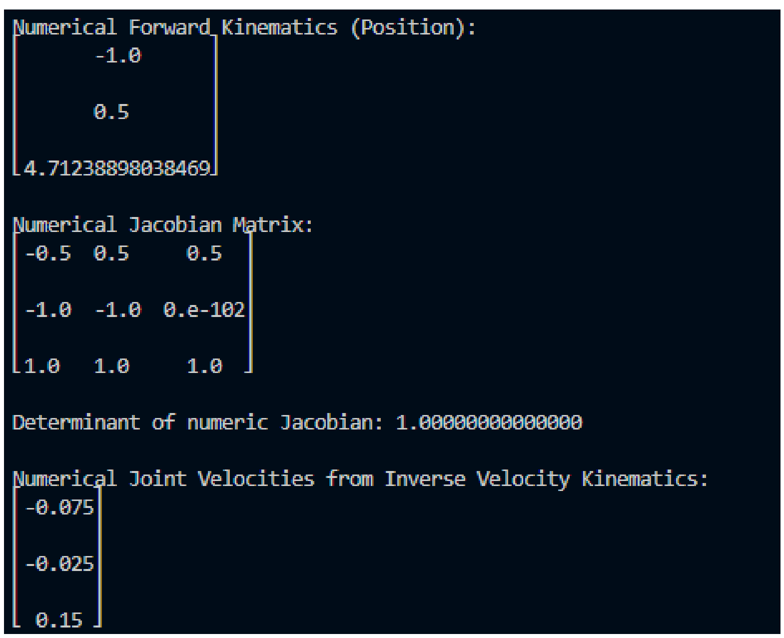

Figure 10: Numerical Substitutions

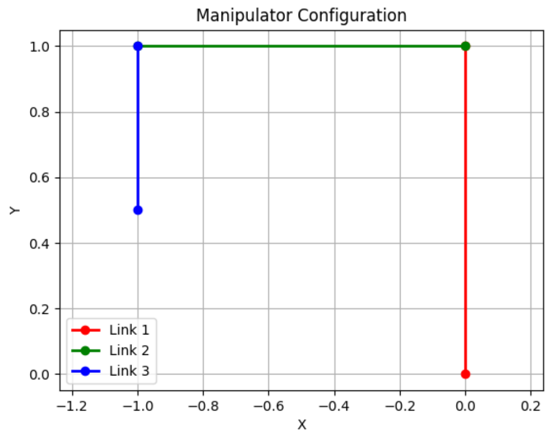

Figure 11: Manipulator Spatial Position

Figure 12: Theta 1 Sweep

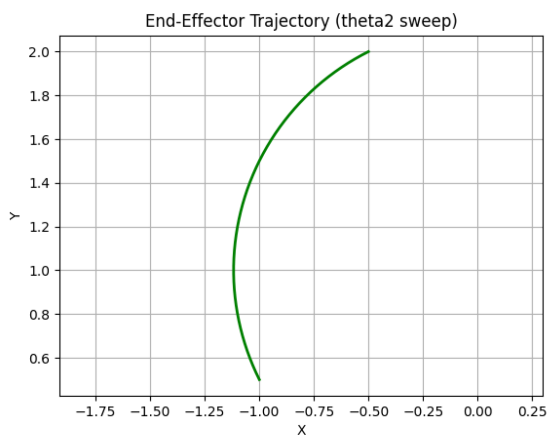

Figure 13: Theta 2 Sweep

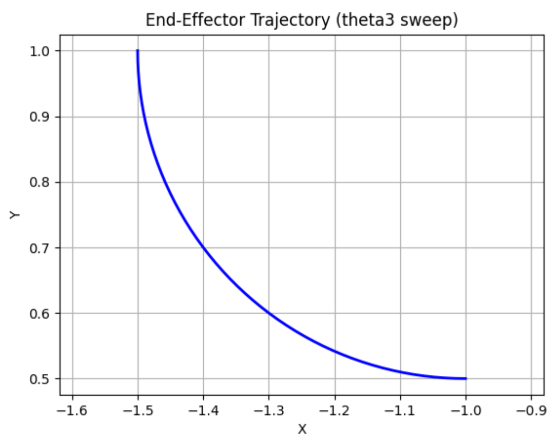

Figure 14: Theta 3 Sweep