AAE 590LGM: Lie Group Methods

Problem Set 2

Travis Hastreiter

10 February, 2026

Contents

Code Listing

P1) Normal Subgroups and Quotient Groups

Before diving into SE(2), let's understand why normal subgroups matter.

1.a) Definition and Intuition:

Question:
A subgroup $N \leq G$ is **normal** (written $N \trianglelefteq G$) if $gNg^{-1} = N$ for all $g \in G$.

- Explain why every subgroup of an abelian group is automatically normal.
- For non-abelian groups, conjugation can "twist" a subgroup. Give geometric intuition: if H is a set of translations and g is a rotation, what does gHg^{-1} represent?

Solution:

1.b) A Matrix Group Example:

Question:
Consider the determinant map $\det : \mathrm{GL}(2, \mathbb{R}) \to \mathbb{R}^*$ (where $\mathbb{R}^* = \mathbb{R} \setminus \{0\}$ under multiplication).

- Verify det is a homomorphism: $\det(AB) = \det(A)\det(B)$.
- What is ker(det)? (This group has a name—what is it?)
- What information about a matrix is "forgotten" when we apply det? What's preserved?

Solution:

1.c) First Isomorphism Theorem (Preview):

Question:
For a homomorphism $\phi : G \to H$:

- **Kernel:** $\ker(\phi) = \{g \in G : \phi(g) = e_H\}$ (elements mapped to identity)
- **Image (Range):** $\mathrm{im}(\phi) = \{\phi(g) : g \in G\}$ (elements that ϕ "hits")

First Isomorphism Theorem: $G/\ker(\phi) \cong \mathrm{im}(\phi)$.
In words: quotienting by what ϕ "kills" gives you what ϕ "sees."

- For SE(()2): if we define $\pi : \mathrm{SE}(()2) \to \mathrm{SO}(()2)$ by $\pi(\mathbf{t}, R) = R$, what is $\ker(\pi)$?
- What does the First Isomorphism Theorem tell us about $\mathrm{SE}(()2)/\ker(\pi)$?

Solution:

P2) SE(2) and the Semi-Direct Product

The group SE(2) of planar rigid motions is our first example of a **semi-direct product**. This structure—where one subgroup "twists" another—is fundamental to robotics.

The Translation Group $\mathbb{T}(2)$: The set of 2D translations forms a group $\mathbb{T}(2)$ under composition. As a matrix Lie group:

$$\mathbb{T}(2) = \left\{ \begin{bmatrix} I & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} : \mathbf{t} \in \mathbb{R}^2 \right\} \cong (\mathbb{R}^2, +)$$

This group is abelian (translations commute) and isomorphic to $\mathbb{R}^2$ with vector addition. Its Lie algebra is $\mathfrak{t}(2) \cong \mathbb{R}^2$.

2.a) Matrix Representation:

Question:
Write $X \in \mathrm{SE}(()2)$ as a 3×3 matrix:

$$X = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}, \quad R \in \mathrm{SO}(2),\ \mathbf{t} \in \mathbb{R}^2$$

Tuple notation: We can also write $X = (\mathbf{t}, R)$ as a compact tuple. The correspondence is:

$$(\mathbf{t}, R) \longleftrightarrow \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

We order as $(\mathbf{t}, R)$ (translation first) to match the semi-direct product $\mathbb{T}(2) \rtimes \mathrm{SO}(2)$ convention. Both notations represent the same rigid motion: rotate by R, then translate by $\mathbf{t}$ (in the world frame).

- Derive the composition $X_1 X_2$ and inverse X^{-1} formulas using block matrix multiplication. You don't need to expand 2×2 blocks—leave products like $R_1 R_2$ as is.
- Express your results in tuple form: $(\mathbf{t}_1, R_1) \cdot (\mathbf{t}_2, R_2) = (?, ?)$ and $(\mathbf{t}, R)^{-1} = (?, ?)$.
- Implement `se2_compose(X1, X2)` and `se2_inverse(X)`.

Check your work: $(\mathbf{t}_1, R_1) \cdot (\mathbf{t}_2, R_2) = (\mathbf{t}_1 + R_1 \mathbf{t}_2,\ R_1 R_2)$ and $(\mathbf{t}, R)^{-1} = (-R^T \mathbf{t},\ R^T)$.
Solution:
Code snippet 1 shows the implementation of `se2_compose(X1, X2)` and `se2_inverse(X)`

P3) SE(2) Lie Algebra, Exp/Log, and Adjoint

The Lie algebra $\mathfrak{se}(2)$ consists of twists. We use Translation-first ordering $\xi = \begin{pmatrix} v_x & v_y & \omega \end{pmatrix}^T \in \mathbb{R}^3$

3.a) Lie Algebra Structure:

Question:
With $\xi = (v_x, v_y, \omega)^T = (\mathbf{v}^T, \omega)^T$, the wedge map is:

$$\xi^\wedge = \begin{bmatrix} \omega^\wedge & \mathbf{v} \\ \mathbf{0}^T & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\omega & v_x \\ \omega & 0 & v_y \\ 0 & 0 & 0 \end{bmatrix}$$

- Verify this is block upper-triangular (rotation block in upper-left, translation in upper-right).
- Implement `se2_wedge(xi)` and `se2_vee(Xi)`.

Solution:
Code snippet 1 shows the implementation of `se2_wedge(xi)` and `se2_wedge(Xi)`

3.b) Exponential Map:

Question:
The closed-form expression is:

$$\text{Exp}(()\xi^\wedge) = \begin{bmatrix} R(\omega) & V(\omega)\mathbf{v} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $V(\omega) = \frac{\sin\omega}{\omega} I + \frac{1-\cos\omega}{\omega} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$.

- Derive by hand using the matrix exponential power series (show key steps).
- Implement `se2_exp(xi)` with Taylor series for $\omega \approx 0$ (use 5 terms).

Solution:
Code snippet 1 shows the implementation of `se2_exp(xi)`

3.c) Logarithm Map:

Question:
The inverse of $V(\omega)$ is:

$$V(\omega)^{-1} = \frac{\omega}{2} \cot\frac{\omega}{2} I + \frac{\omega}{2} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

- Implement `se2_log(X)`. Use Taylor series for $\omega \approx 0$ (use 5 terms).
- Verify numerically: $\text{Exp}(()\text{Log}(()X)) = X$ for random $X \in \text{SE}(()2)$.

Solution:
Code snippet 1 shows the implementation of `se2_log(X)`.
Code snippet 2 shows the implementation of the unit test verifying $\text{Exp}(()\text{Log}(()X)) = X$ for random $X \in \text{SE}(()2)$ named `test_se2_exp_log()` and Fig 1 shows the test passing.

3.d) Adjoint Representation Ad_X:

Question:
The adjoint $\mathrm{Ad}_X : \mathfrak{se}(2) \to \mathfrak{se}(2)$ is defined by $(\mathrm{Ad}_X \xi)^\wedge = X \xi^\wedge X^{-1}$. *Note: You don't need to multiply out* 2×2 *blocks explicitly. Showing block form (e.g.,* $R\omega^\wedge R^T$*,* $R\mathbf{v}$*) is sufficient.*

Translation-first ordering $\xi = (v_x, v_y, \omega)^T$:

- Compute $X \xi^\wedge X^{-1}$ explicitly for $X = (\mathbf{t}, R)$.
- Extract the 3×3 matrix Ad_X such that $\mathrm{Ad}_X \xi$ gives the transformed twist.
  You should get: $\mathrm{Ad}_X = \begin{bmatrix} R & \mathbf{t}^\odot \\ \mathbf{0}^T & 1 \end{bmatrix}$ where $\mathbf{t}^\odot = \begin{bmatrix} t_y \\ -t_x \end{bmatrix} = -J\mathbf{t}$ with $J = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$.

Rotation-first ordering $\xi = (\omega, v_x, v_y)^T$:

- Repeat the derivation with this ordering. The wedge map produces the same 3×3 matrix, but extracting Ad_X requires matching to the new vector ordering.
- You should get: $\mathrm{Ad}_X = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{t}^\odot & R \end{bmatrix}$ (block *lower*-triangular).

Verification:

- Verify numerically: $\mathrm{Ad}_{X_1 X_2} = \mathrm{Ad}_{X_1} \mathrm{Ad}_{X_2}$ for random $X_1, X_2 \in \mathrm{SE}(()2)$.
- Verify numerically: $\mathrm{Ad}_X^{-1} = \mathrm{Ad}_{X^{-1}}$ (the adjoint respects inverses).
- **Connection to Problem 2:** Confirm that R acts on velocity components, as you predicted from the normal subgroup structure.

Solution:
Code snippet 2 shows the implementation of the unit test verifying $\mathrm{Ad}_{X_1 X_2} = \mathrm{Ad}_{X_1} \mathrm{Ad}_{X_2}$ for random $X_1, X_2 \in \mathrm{SE}(()2)$ for random $X \in \mathrm{SE}(()2)$ named `test_se2_Ad_composition` and Fig 1 shows the test passing. Code snippet 2 shows the implementation of the unit test verifying $\mathrm{Ad}_X^{-1} = \mathrm{Ad}_{X^{-1}}$ for random $X \in \mathrm{SE}(()2)$ named `test_se2_Ad_inv()` and Fig 1 shows the test passing.

3.e) Lie Bracket and ad_ξ:

Question:
The Lie bracket on vectors is defined by $[\xi_1, \xi_2]^\wedge := \xi_1^\wedge \xi_2^\wedge - \xi_2^\wedge \xi_1^\wedge$ (i.e., compute the matrix commutator, then apply vee). The small adjoint ad_ξ is the 3×3 matrix such that $[\xi_1, \xi_2] = \mathrm{ad}_{\xi_1} \xi_2$.

Translation-first ordering $\xi = (v_x, v_y, \omega)^T$:

- Compute $[\xi_1^\wedge, \xi_2^\wedge]$ for $\xi_i = (v_{ix}, v_{iy}, \omega_i)^T$. Extract the result as a vector.
- Derive by hand the 3×3 matrix ad_ξ.
  You should get: $\mathrm{ad}_\xi = \begin{bmatrix} \omega^\wedge & \mathbf{v}^\odot \\ \mathbf{0}^T & 0 \end{bmatrix}$ where $\mathbf{v}^\odot = \begin{bmatrix} v_y \\ -v_x \end{bmatrix} = -J\mathbf{v}$.

Solution:

Implementation and Unit Tests

Code Snippet 1: SE(2) and $\mathfrak{se}(2)$ Function Implementations

```python
import numpy as np
from Code.SO2.SO2_maps import so2_wedge, so2_vee, so2_exp, so2_log

def se2_compose(X1, X2):
    """
    Composition of elements of SE(2)

    :param X1: element of SE(2)
    :param X2: element of SE(2)
    """
    X1_t = X1[0:2, 2].reshape((2, 1))
    X1_R = X1[0:2, 0:2]

    X2_t = X2[0:2, 2].reshape((2, 1))
    X2_R = X2[0:2, 0:2]

    X12_R = X1_R @ X2_R
    X12_t = X1_R @ X2_t + X1_t

    X12 = np.block([[X12_R, X12_t],
                    [np.zeros((1, 2)), 1]])

    return X12

def se2_inverse(X):
    """
    Inverse of SE(2) element

    :param X: element of SE(2)
    """
    X_t = X[0:2, 2].reshape((2, 1))
    X_R = X[0:2, 0:2]

    X_inv_t = -X_R.T @ X_t
    X_inv_R = X_R.T

    X_inv = np.block([[X_inv_R, X_inv_t],
                      [np.zeros((1, 2)), 1]])

    return X_inv

def se2_wedge(xi):
    """
    Maps from se2 real number parametrization to se2 Lie algebra

    :param xi: twist vector (v_x, v_y, omega) in R3
    """
    xiwedge = np.block([[so2_wedge(xi[2]), xi[0:2].reshape((2, 1))],
                        [np.zeros((1, 2)),       0]])

    return xiwedge

def se2_vee(xiwedge):
    """
    Maps from se2 Lie algebra to its real number parametrization

    :param xiwedge: 3x3 se2 Lie algebra element
    """

    xi = np.block([xiwedge[[0, 1], [2, 2]], so2_vee(xiwedge[0:2, 0:2])]);

    return xi

def se2_exp(xi):
```

```
    """
    Maps from parametrization of se2 Lie algebra to SE2 Lie group

    :param xi: twist vector (v_x, v_y, omega) in R3
    """
    # Extract elements
    v = xi[0:2]
    omega = xi[2]

    # SO2 exponential map
    R = so2_exp(omega)

    # Translation-orientation coupling
    if abs(omega) > 1e-8:
        V = np.sin(omega) / omega * np.eye(2) + (1 - np.cos(omega)) / omega * np.array([[0, -1], [1, 0]])
    else:
        V = ((1 - omega ** 2 / 6 + omega ** 4 / 120) * np.eye(2)
           + (omega / 2 - omega ** 3 / 24) * np.array([[0, -1], [1, 0]]))

    X = np.block([[R,    V @ v.reshape((2, 1))],
                  [np.zeros((1, 2)),     1]])

    return X

def se2_log(xiwedge):
   """
   Maps from SE2 Lie group to the parametrization of its Lie algebra se2

   :param xiwedge: 3x3 se2 Lie algebra element
   """
   omega = so2_log(xiwedge[0:2, 0:2])

   if abs(omega) > 1e-8:
       V_inv = omega / 2 * (np.cos(omega / 2) / np.sin(omega / 2) * np.eye(2) + np.array([[0, 1], [-1, 0]]))
   else:
       V_inv = 1 / 2 * ((2 - omega ** 2 / 6 - omega ** 4 / 360 - 2 * omega ** 6 / 30240 + omega ** 8 / 604800) * np.eye(2) + omega * np.array([[0, -1], [1, 0]]))

   v = V_inv @ xiwedge[0:2, 2]

   xi = np.vstack((v.reshape((2, 1)), np.reshape(omega, (1,1))))

   return xi

def se2_Ad(X):
    """
    Maps from se2 to se2 defined by (Ad_X xi)**wedge = X xi**wedge X**-1
    Shifts tangent spaces

    :param X: SE2 Lie group element
    """
    Ad = np.copy(X)
    Ad[0:2, 2] = -np.array([[0, -1], [1, 0]]) @ X[0:2, 2]

    return Ad

def se2_ad(xi):
    """
    Derivative of Adjoint at identity

    :param xi: se(2) Lie algebra element parametrization
    """
    ad = se2_wedge(xi)
    ad[0:2, 2] = -np.array([[0, -1], [1, 0]]) @ ad[0:2, 2]

    return ad
```

Code Snippet 2: Unit Tests for SE(2) Using Pytest

```python
import pytest as pt
import numpy as np
from Code.SE2.SE2_maps import se2_compose, se2_inverse, se2_exp, se2_log, se2_vee, se2_wedge, se2_Ad, se2_ad

def test_se2_exp_log():
    """
    Test that exp(log(X)) = X for random SE2 elements
    """
    n_samples = 100

    rng = np.random.default_rng()
    xi_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))

    checks = np.zeros([n_samples, 1])
    for i in range(n_samples):
        X_random = se2_exp(xi_random[:, i])

        X_log = se2_log(X_random)
        X_tilde = se2_exp(X_log)

        checks[i] = np.all(X_tilde == pt.approx(X_random))

    assert np.all(checks)

def test_se2_Ad():
    """
    Test that Ad_X(xi) = (X xi^wedge X^-1)^vee for random SE2, se2 elements
    """
    n_samples = 100

    rng = np.random.default_rng()
    xi_X_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))
    xi_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))

    checks = np.zeros([n_samples, 1])
    for i in range(n_samples):
        X_random = se2_exp(xi_X_random[:, i])

        xi_prime = se2_vee(X_random @ se2_wedge(xi_random[:, i]) @ se2_inverse(X_random))
        xi_prime_ad = se2_Ad(X_random) @ xi_random[:, i]

        checks[i] = np.all(xi_prime_ad == pt.approx(xi_prime))

    assert np.all(checks)

def test_se2_Ad_composition():
    """
    Test that Ad_{X_1 X_2} = Ad_{X_1}Ad_{X_2} for random SE2 elements
    """
    n_samples = 100

    rng = np.random.default_rng()
    xi1_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))
    xi2_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))

    checks = np.zeros([n_samples, 1])
    for i in range(n_samples):
        X1_random = se2_exp(xi1_random[:, i])
        X2_random = se2_exp(xi2_random[:, i])

        Ad_X1X2 = se2_Ad(se2_compose(X1_random, X2_random))
        Ad_X1_Ad_X2 = se2_Ad(X1_random) @ se2_Ad(X2_random)

        checks[i] = np.all(Ad_X1X2 == pt.approx(Ad_X1_Ad_X2))

```

```python
    assert np.all(checks)

def test_se2_Ad_inv():
    """
    Test that (Ad_X)^-1 = Ad_{X^-1} for random SE2 elements
    """
    n_samples = 100

    rng = np.random.default_rng()
    xi_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))

    checks = np.zeros([n_samples, 1])
    for i in range(n_samples):
        X_random = se2_exp(xi_random[:, i])

        Ad_X_inv = np.linalg.inv(se2_Ad(X_random))
        Ad_Xinv = se2_Ad(se2_inverse(X_random))

        a = se2_Ad(X_random) @ Ad_X_inv
        b = se2_Ad(X_random) @ Ad_Xinv

        checks[i] = np.all(Ad_X_inv == pt.approx(Ad_Xinv))

    assert np.all(checks)

def test_se2_ad():
    """
    Test that ad_{xi_1}(xi_2) = [xi_1, xi_2] for random se2 elements
    """
    n_samples = 100

    rng = np.random.default_rng()
    xi1_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))
    xi2_random = np.reshape(rng.uniform(-10, 10, 3 * n_samples), (3, n_samples))

    checks = np.zeros([n_samples, 1])
    for i in range(n_samples):
        Lie_bracket = se2_vee(se2_wedge(xi1_random[:, i]) @ se2_wedge(xi2_random[:, i])
                              - se2_wedge(xi2_random[:, i]) @ se2_wedge(xi1_random[:, i]))
        Lie_bracket_ad = se2_ad(xi1_random[:, i]) @ xi2_random[:, i]

        checks[i] = np.all(Lie_bracket_ad == pt.approx(Lie_bracket))

    assert np.all(checks)
```

Figure 1: Pytest unit test output showing tests passing

```
================================== test session starts ==================================
platform win32 -- Python 3.10.11, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\thatf\OneDrive\Documents\Purdue Classes\AAE 590LGM\Lie Group Methods
collected 5 items

Tests\SE2\test_SE2.py .....                                                      [100%]

=================================== 5 passed in 0.56s ===================================
```

P4) SE(()2) Kinematics Simulation

The kinematic equation $\dot{X} = X\xi^\wedge$ describes rigid body motion on SE(2)

4.a)

Question:
Unicycle Model: A unicycle has forward velocity v and yaw rate ω. In the body frame, there is no lateral motion ($v_y = 0$) so $v_x = v$.

- Write the body-frame twist for the unicycle
- Expand the KE to derive the ODEs
- For constant $v, \omega > 0$: What shape is the trajectory? What is the radius?

Solution:
The body frame twist is

$$\boxed{\xi = \begin{pmatrix} v_x \\ v_y \\ \omega \end{pmatrix} = \begin{pmatrix} v \\ 0 \\ \omega \end{pmatrix}}$$

Deriving the equations of motion

$$\dot{X} = X\xi^\wedge = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & -\omega & v \\ \omega & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} -\omega\sin(\theta) & -\omega\cos(\theta) & v\cos(\theta) \\ \omega\cos(\theta) & -\omega\sin(\theta) & v\sin(\theta) \\ 0 & 0 & 0 \end{pmatrix}$$

$$\implies \dot{x} = v_x\cos(\theta),\ \dot{y} = v\sin(\theta),\ \dot{\cos(\theta)} = -\omega\sin(\theta),\ \dot{\sin(\theta)} = \omega\cos(\theta)$$

$$\dot{\cos(\theta)} = -\sin(\theta)\dot{\theta} = -\omega\sin(\theta) \implies \dot{\theta} = \omega$$

$$\dot{\sin(\theta)} = \cos(\theta)\dot{\theta} = \omega\cos(\theta) \implies \dot{\theta} = \omega$$

$$\boxed{\dot{x} = v\cos(\theta),\ \dot{y} = v\sin(\theta),\ \dot{\theta} = \omega}$$

For constant $v, \omega > 0$ the trajectory follows a circle with a radius found by computing the following integral

$$r = \int_0^{\frac{\pi}{2\omega}} v\cos(\omega t)\mathrm{d}t = \frac{v}{\omega}\int_0^{\frac{\pi}{2}} \cos(\theta)\mathrm{d}t = \frac{v}{\omega}$$

The trajectory is a circle with radius $\frac{v}{\omega}$

4.b)

Question:

Solution:

Code Snippet 3: SE(2) Integrator Implementation

```
import numpy as np
from Code.SE2.SE2_maps import se2_compose, se2_exp, se2_wedge

def se2_euler_integrator(X_k, xi_k, dt):
    """
    Integrate using first order Euler
```

```
    :param X_k: group element at t_k
    :param xi_k: twist at t_k
    :param dt: timestep
    """
    X_kp1 = se2_compose(X_k, np.eye(3) + dt * se2_wedge(xi_k))

    return X_kp1

def se2_Lie_group_integrator(X_k, xi_k, dt):
    """
    Integrate using exponential map (assuming twist is constant accross timestep)

    :param X_k: group element at t_k
    :param xi_k: twist at t_k
    :param dt: timestep
    """
    X_kp1 = se2_compose(X_k, se2_exp(dt * xi_k))

    return X_kp1
```

Code Snippet 4: SE(2) Integrator Test

```
import numpy as np
import matplotlib.pyplot as plt
from Code.SE2.SE2_maps import se2_compose, se2_exp, se2_log, se2_wedge, se2_vee, se2_inverse
from Code.SE2.SE2_integrators import se2_euler_integrator, se2_Lie_group_integrator

## Unicycle sim

# Initialize
X_0 = np.eye(3)
v = 1 # [m / s]
omega = 0.5 # [rad / s]
xi_k = np.array([[v], [0], [omega]])
dt = 0.1 # [s]
tf = 20
time = np.arange(0, tf + dt, dt)

# Simulation loop
X_k_euler = X_0
X_k_Lie = X_0

positions_euler = np.zeros([2, time.size])
positions_Lie = np.zeros([2, time.size])

error_euler = np.zeros_like(time)
error_Lie = np.zeros_like(time)

for k in range(time.size - 1):
   X_k_euler = se2_euler_integrator(X_k_euler, xi_k, dt)
   X_k_Lie = se2_Lie_group_integrator(X_k_Lie, xi_k, dt)

   positions_euler[:, k + 1] = X_k_euler[0:2, 2]
   positions_Lie[:, k + 1] = X_k_Lie[0:2, 2]

   R_euler = X_k_euler[0:2, 0:2]
   R_Lie = X_k_Lie[0:2, 0:2]

   error_euler[k + 1] = np.linalg.norm(R_euler.T @ R_euler - np.eye(2), "fro")
   error_Lie[k + 1] = np.linalg.norm(R_Lie.T @ R_Lie - np.eye(2), "fro")

# Plot
# Trajectories
plt.plot(positions_euler[0, :], positions_euler[1, :], label = "Euler")
plt.plot(positions_Lie[0, :], positions_Lie[1, :], label = "Lie Group")
plt.xlabel("X Position [m]")
plt.ylabel("Y Position [m]")
```

```
plt.title("Unicycle Trajectories vs Integrator")
plt.legend()
plt.grid(True)
plt.gca().set_aspect('equal', adjustable='box')
plt.savefig("./HW2/Parts/Q4/AAE590LGM_HW2_Q4_traj.png")
plt.show()

# Error
plt.semilogy(time, error_euler, "o", label = "Euler")
plt.semilogy(time, error_Lie, "o", label =  "Lie Group")
plt.xlabel("Time [s]")
plt.ylabel("Rotation Error")
plt.title("Rotation Error vs Time for Both Integrators")
plt.legend()
plt.grid(True)
plt.savefig("./HW2/Parts/Q4/AAE590LGM_HW2_Q4_error.png")
plt.show()

# Compare with Exp map
X_f = se2_compose(X_0, se2_exp(tf * xi_k))
print(np.linalg.norm(X_f - X_k_Lie, "fro"))
```

Figure 2: Trajectories of both integrators for unicycle

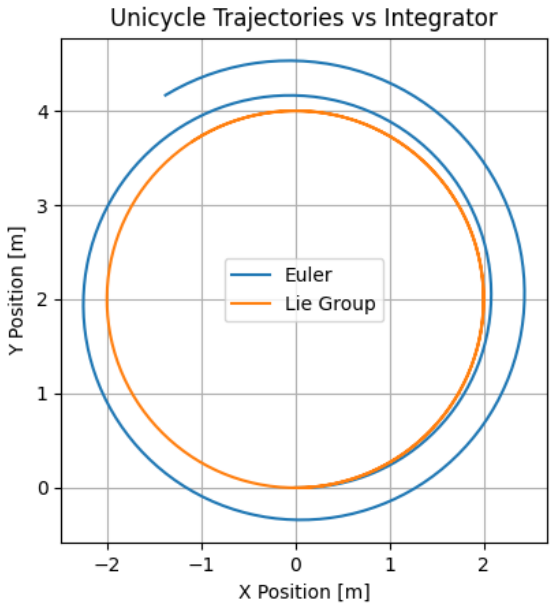
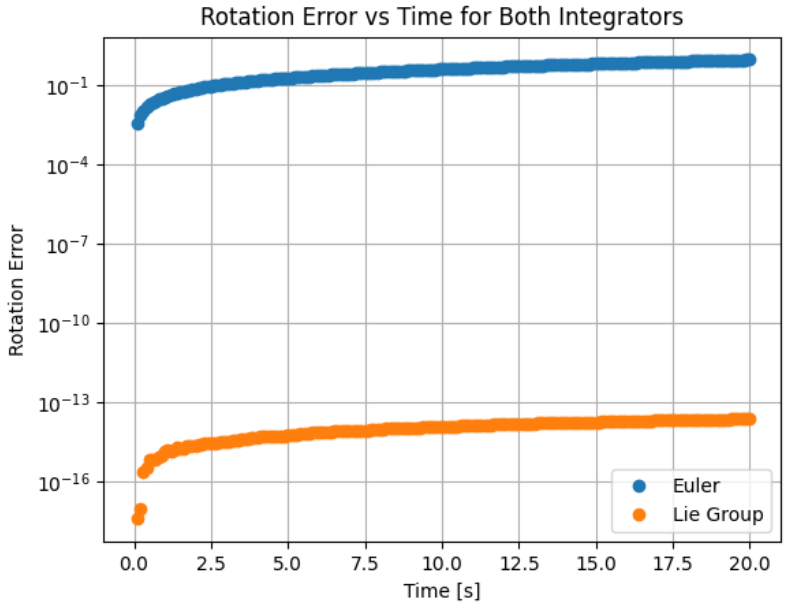


Figure 3: Error over time for both integrators



The error between the exact propagation using the exponential map and the Lie group integrator at the final time is 1.8289049996250573e-14. This means that within numerical precision they are the same whic makes sense given that the Lie group integrator uses the exponential map.