ENEE 467: Robotics Project Laboratory
Spatial Math and Rigid Body Transformations
Lab 1

Learning Objectives

The purpose of this lab is to provide an introduction to the mathematical representations used to describe rigid body motions as well as programmatic tools useful for computing these representations. After completing this lab students should be able to:

- Describe rigid motions using homogeneous transformation matrices
- Compose and parameterize transformation frames relative to one another
- Use Python tools to compute transformations
- Compute arbitrary transformation frames given a kinematic tree.

The theory of modern robotics and manipulation is based on the principles of rigid body motion and the establishment of various coordinate systems to represent the position and orientation of rigid bodies. Robots are approximated as chains of rigid links (non-flexible and non-deformable) whose actuation occurs at the joints between links. In this lab we study the operations of rotations and translation, and introduce the notion of homogeneous transformations. Homogeneous transformations combine the operations of rotation and translation into a single matrix multiplication. Furthermore, homogeneous transformation matrices can be used to perform coordinate transformations. Such transformations allow us to represent various quantities in different coordinate frames, a facility that we will often exploit in subsequent labs.

Related Reading

1. Robot Dynamics and Control
2. tf: The Transform Library

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab Questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab Exercises at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

List of Questions

List of Exercises

How to Read This Manual

Throughout the manual, boxes beginning with a **Question #** title contain questions to be answered as you work through the lab:

> **Question 0.** A question for you.

Boxes with a 🍃 symbol (like the one below) are extra theoretical or implementation details provided for your enrichment and insight. We recommend that you read them during the lab:

> 🍃 Some extra context and info.

Terminal commands and code blocks will be set in gray-colored boxes (usually for Python syntax)

```
import math # some important comment you should totally read
```

while terminal outputs will be set in yellow-colored boxes

```
some terminal output
```

Files and directories will be set in magenta color with a teletype font, e.g., `~/some/dir/to/a/file.txt`, code variable names and in-text commands will be set in black teletype font, e.g., `a = [0, 0, 0]`, while Python type hints will be set in green teletype font, e.g., `save(path: str)` signifies that the argument to the `save` function is a string. Lastly, figure captions go below figures, table captions appear above tables, and links to an external online reference are highlighted in blue, eg. see: Wikipedia.

1. Lab Procedure

1.1 Prerequisites

This lab requires a few Python scientific libraries for spatial math computations. Namely NumPy, Spatial Maths for Python, and the Robotics Toolbox for Python.

Ensure that the `lab-1` image is build on your computer by running the following command in a terminal to list all built images on the host computer:

```
docker image ls
```

Make sure that you see the `lab-1` image in the output:

```
lab-1-image   latest    cddcf72cb40a   2 days ago     2.12GB
```

If the `lab-1-image` is not seen, ask an instructor to help you with the build process. Instructions for building the image yourself are also provided in the lab repository.

Navigate to the `HOME/Labs` directory and then clone the `lab-1` repo:

```
cd ~/Labs
git clone https://github.com/ENEE467-F2025/lab-1.git
cd lab-1
```

Launch the Docker container with the following commands:

```
cd docker
docker compose -f lab-1-compose.yml run --rm lab-1-docker
```

You should be greeted with the following prompt indicating you are now inside the Docker container:

```
(lab-1) robot@desktop:~$
```

where `desktop` may be replaced with the name of the host machine you are currently using.

⚠ The remainder of the lab is intended to be completed from within the Docker container to prevent any mishaps happening on the host machine. Whenever you are inside a terminal, ensure that the Docker container is active indicated by the above prompt. To reactivate the container should you close the terminal, follow the above instructions. If you're ever unsure of what machine your terminal is connected to, look at the shell prefix:

Docker Container:

```
(lab-1) robot@desktop:~$
```

Host Machine:

```
ras_enee467@desktop:~$
```

Containers are isolated development environments. To pass files between the host machine and the container, the `lab-1/src` directory you downloaded on the host machine is mounted in the container at `/home/robot/lab-1/src`.

2. Rigid Rotations

2.1 The Special Orthogonal Group

Rotations in 3D are canonically represented by rotation matrices, 3×3 matrices belonging to the *Special Orthogonal Group* $\mathrm{SO}(3)$. A matrix $R \in \mathrm{SO}(3)$ is *orthonormal* meaning it is orthogonal, $R^T R = I$, and normalized, $\det R = 1$. The Special Orthogonal Group is called a group because it satisfies the properties of a mathematical group. A group contains a set of elements and a binary operation "$\cdot$" such that for all A, B, and C in the group, the following properties are satisfied:

- **closure:** $A \cdot B$ is also in the group.
- **associativity:** $(A \cdot B) \cdot C = A \cdot (B \cdot C)$.
- **identity element existence:** There exists an element I in the group such that $A \cdot I = I \cdot A = A$.
- **inverse element existence:** There exists an element A^{-1} in the group such that $A \cdot A^{-1} = A^{-1} \cdot A = I$.

For the Special Orthogonal Group, the binary operator is matrix multiplication, the identity element is the 3×3 Identity matrix, and for any $R \in \mathrm{SO}(3)$, the inverse element is its transpose, $R^{-1} = R^T$.

Question 1. Suppose you have two rotation matrices R_1, $R_2 \in \mathrm{SO}(3)$. Is the sum of these two matrices $R_1 + R_2$ an element of $\mathrm{SO}(3)$? Provide a proof for your answer.

There are three major uses for a rotation matrix R:

- to represent an orientation;

- to change the reference frame in which a vector or frame is represented;
- to rotate a vector or frame.

In the first use, R is thought of as a representing a (right-hand) frame; in the second and third uses, R is though of as an operator that acts on a vector or frame (changing its reference frame or rotating it).

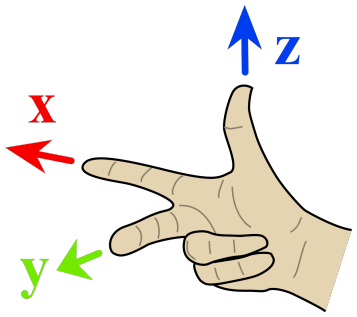


Figure 1: The right-handed coordinate frame.

2.2 Basic 3D Rotations

A basic 3D rotation is a rotation about one of the axes of a coordinate system. In three dimensions, these matrices have the convenient form:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \quad R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

It is easy to verify that the basic rotation matrices have the following properties:

$$R_z(0) = I \tag{1}$$

$$R_z(\theta)R_z(\varphi) = R_z(\theta + \varphi) \tag{2}$$

which together imply

$$R_z(\theta)^{-1} = R_z(-\theta). \tag{3}$$

Question 2. Prove properties (1), (2), and (3) for the basic rotation matrix $R_z(\theta)$.

The `spatialmath` package provides class methods for generating basic rotation matrices. Open an Interactive Python shell (`ipython`) inside the container and import all `spatialmath` utilities:

```
from spatialmath import *
from math import pi
```

Create a rotation matrix representing a rotation of $\pi/4$ radians about the z-axis:

```
R_1 = SO3.Rz(pi/4)
print(R_1)
```

Which should return the following matrix:

```
   0.7071   -0.7071    0
   0.7071    0.7071    0
   0         0         1
```

Recall that one interpretation of a rotation matrix is a rigid frame in which the orthogonal coordinate axes of the frame are the columns of the rotation matrix. To extract the coordinate axes of a rotation matrix, use the following commands:

```
R_1.n
R_1.o
R_1.a
```

where n represents the normal (x) axis of the frame, o represents the orientation (y) axis of the frame, and a represents the approach (z) axis of the frame, and the rotation matrix is expressed as $R_1 = \begin{bmatrix} n & o & a \end{bmatrix}$. The naming convention is derived from a popular frame convention in robotics called the *orientation-approach* or 2-axis representation where a coordinate frame is expressed by two orthogonal vectors. See Figure 2 for a visual representation of the 2-axis frame.

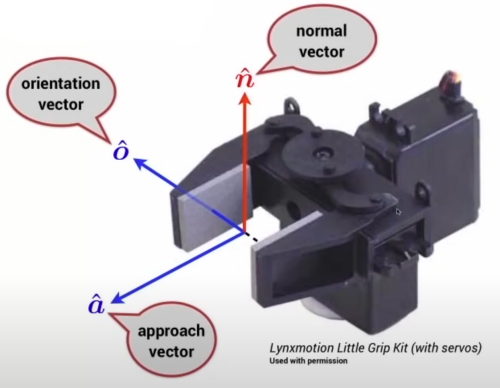


Figure 2: A gripper frame using the orientation-approach convention.

Question 3. The orientation of a right-hand coordinate frame is represented by three orthogonal vectors. Why is it the case that you can fully define a coordinate frame with just two orthogonal vectors? (As we will see soon, you can fully parameterize a rotation matrix with just three numbers!)

To generate basic rotation matrices about the x and y axes, use the following class methods:

```
R_2 = SO3.Rx(pi/4)
print(R_2)
R_3 = SO3.Ry(pi/4)
print(R_3)
```

To visualize the rotation matrices, use the plotting function:

```
import matplotlib.pyplot as plt
R_1.plot(frame="1", color="black")
R_2.plot(frame="2", color="blue")
R_3.plot(frame="3", color="red")
plt.show()
```

Save the figure by selecting the save icon and include it in your lab writeup.

Suppose now we have a vector in the world frame $v_0 = [1,\ 0,\ 0]^T$. To apply a rotation to a vector in the world frame, we *premultiply* the vector with the rotation matrix. Try rotating the vector using the three basic rotation matrices:

```
v_0 = [1., 1., 0.]
print( R_1 * v_0 )
print( R_2 * v_0 )
print( R_3 * v_0 )
```

Another important property is that the action of a rotation matrix on a vector (e.g., rotating the vector) does not change the length of the vector. Verify that this is the case:

```
import numpy as np
print(f"Length of unrotated vector: {np.linalg.norm(v_0):0.3f}")
print(f"Length after applying R_1:  {np.linalg.norm(R_1*v_0):0.3f}")
print(f"Length after applying R_2:  {np.linalg.norm(R_2*v_0):0.3f}")
print(f"Length after applying R_3:  {np.linalg.norm(R_3*v_0):0.3f}")
```

Suppose now that the previous rotation matrices represent three rotated coordinate frames with respect to the world frame. How do we get the coordinates of v_0 in the frames represented by R_1, R_2, and R_3? Right now the coordinates of v_0 are expressed in the world frame. The relationship between a vector in two coordinate frames a and b given the rotation matrix R_{ab} that associates them is:

$$v_a = R_{ab} v_b$$

Therefore to transform the coordinates of v_0 into another frame, we premultiply v_0 by the inverse of the rotation matrix associated with the desired frame. Therefore:

$$v_1 = R_1^{-1} v_0$$
$$v_2 = R_2^{-1} v_0$$
$$v_3 = R_3^{-1} v_0$$

The spatialmath package provides an inverse function for elements of SO(3). To compute the new coordinates of v_0, try the following:

```
v_0 = [1., 1., 0.]
v_1 = R_1.inv()*v_0
v_2 = R_2.inv()*v_0
v_3 = R_3.inv()*v_0
```

Question 4.

a. Try playing around with some more vectors using the above transformations. What do you notice about the z-component of vectors transformed into the R_1 frame? Respectively the x-component for R_2 and y-component for R_3.

b. Suppose you were given a vector v whose coordinates are expressed in the frame R_1. What transformation would you use to express the coordinates of v in the world frame?

2.3 Composition of Rotation Matrices

Recall that R_1 represents a rotational transformation between the world frame and frame 1. Suppose now R_2 represents a rotational transformation between frame 1 and frame 2 (as opposed to R_2 representing a rotation from the world frame to frame 2). To make this a little more intuitive, let's rename the matrices to indicate their relationships. $R_{01} \leftarrow R_1$ and $R_{12} \leftarrow R_2$ where the matrix R_{ab} represents a transformation from frame a to frame b. Create two new matrices in your Python shell:

```
R_01 = R_1
R_12 = R_2
```

A given vector v can then be represented by coordinates specified with respect to any of theses frames: v_0, v_1, and v_2. The relationship among these representations of v is:

$$v_0 = R_{01} v_1$$
$$v_1 = R_{12} v_2$$

Substituting these two equations results in:

$$v_0 = R_{01} R_{12} v_2$$

and we can express a new matrix representing the transformation from the world frame to frame 2 with:

$$R_{02} = R_{01} R_{12}$$

Two conventions exist for enumerating rotation/transformation matrices. One convention represents a transformation from frame a to frame b by R_b^a. The alternative choice is to have both frames as subscripts, i.e. R_{ab}. These two representations are equivalent and will both be used in some capacity when convenient. The subscript cancellation rule for compositions in either convention is:

$$R_b^a \cdot R_c^b = R_{\not b}^a \cdot R_c^{\not b} = R_c^a\,, \qquad R_{ab} \cdot R_{bc} = R_{a\not b} \cdot R_{\not b c} = R_{ac}$$

Another helpful property is that the inverse of a rotation/transformation matrix reverses the order of the subscripts, i.e.:

$$(R_{ab})^{-1} = R_{ba}$$

To compose elements of SO(3) in Python, use the multiplication "$*$" operator:

```
R_02 = R_01 * R_12
```

The order in which a sequence of rotations is carried out, and consequently the order in which the rotation matrices are multiplied together is crucial. Rotation is not a vector quantity[1] and so rotational transformations do not commute in general.

Suppose we have two frames related by the rotation matrix R_{01}. Then we are given a rotation matrix R in which we would like to do two different things: 1) Apply R to R_{01} relative to the world frame, and 2) apply R to R_{01} relative to the frame 1. To apply R in the world frame, we premultiply R_{01} with R. To apply R in frame 1, we post multiply R_{01} with R. Create a new unique rotation matrix R using the `Rand` class method:

```
R = SO3.Rand()
```

Then apply the two transformations to a vector starting along the world x-axis and compare the results:

```
print( R * R_01 * [1,0,0] )
print( R_01 * R * [1,0,0] )
```

Notice that you result in two different vectors after applying the two different transformations even though they use the same two matrices. This highlights the noncommutativity of the SO(3) group.

[1] Meaning belonging to a vector space and following certain axioms including commutativity. Moreover, SO(3) is also a smooth manifold, so it is in fact a Lie group.

Summary of Rotation Composition
Given a rotation matrix R_{01} representing a rotation from the world frame to frame 1, if another frame is to be obtained by a rotation R relative to the **current frame** (frame 1) then **postmultiply** R_{01} by R to obtain

$$R_{02} = R_{01}R$$

If the second rotation is to be performed relative to the **fixed frame** (world frame) then **premultiply** R_{01} by R to obtain

$$R_{02} = RR_{01}$$

Question 5. Suppose R is defined by the following sequence of basic rotations in the order specified:

1. A rotation of $\pi/4$ about the current x-axis
2. A rotation of $\pi/2$ about the current z-axis
3. A rotation of $\pi/6$ about the fixed z-axis
4. A rotation of $\pi/3$ about the current y-axis
5. A rotation of $\pi/2$ about the fixed x-axis

Determine the correct product of basic rotation matrices to obtain the rotation matrix R. Compute the rotation matrix R using the `spatialmath` library and include a screenshot in your lab writeup.

2.4 Parameterization of Rotations

A rigid body possesses at most three rotational degrees-of-freedom and thus at most three quantities are required to specify its orientation. This can be deduced from the orthonormality of rotation matrices in that their columns are normal vectors and these columns are mutually orthogonal. Together these constraints define six independent equations with nine unknowns, which implies that there are three free variables.

A rotation matrix R can be described as a product of successive rotations about the principle coordinate axes x_0, y_0, and z_0 taken in a specific order. These rotations define the **roll**, **pitch**, and **yaw** angles. The order of rotation is specified as first a roll about x_0, then a pitch about y_0, and finally a yaw about z_0. The successive rotations are relative to the fixed frame, and the resulting transformation matrix is given by:

$$R_{\text{rpy}} = R_z(\text{yaw})R_y(\text{pitch})R_x(\text{roll})$$

Create a rotation matrix using the RPY convention with roll = $\pi/2$, pitch = $\pi/3$, and yaw = $\pi/4$:

```
R_rpy = SO3.Rz(pi/4) * SO3.Ry(pi/3) * SO3.Rx(pi/2)
print(R_rpy)
```

The `spatialmath` library also provides a class method for creating a rotation matrix from RPY angles directly:

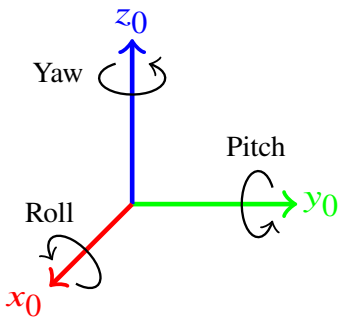


Figure 3: Roll, pitch, and yaw angles.

```
R = SO3.RPY(pi/2, pi/3, pi/4)
print(R)
```

Verify that the two matrices are indeed the same.

Be mindful here since the arguments to the class method `SO3.RPY` are in the order roll-pitch-yaw whereas the order when composing basic rotation matrices is apparently backwards. Remember, this isn't actually the case. Since we are rotating with respect to the fixed frames, successive rotations are premultiplied.

To extract the RPY angles from a given rotation matrix R, use the method `rpy()`:

```
# generate a random rotation matrix
R = SO3.Rand()
# get the RPY angles
print( R.rpy() )
```

Interestingly, there is always more than one sequence of rotations (RPY angles) that result in the same rotation matrix. In the degenerate case, there are an infinite number of solutions. Such caveats lead us to alternative ways of parameterizing rotations.

We are often interested in a rotation about an arbitrary axis in space. This provides both a convenient way to describe rotations and an alternative parameterization for rotation matrices. Let $k = [k_x, k_y, k_z]^T$, expressed in the world frame, be a unit vector defining an axis. To construct a rotation matrix given an angle and a vector, use the class method `SO3.AngleAxis`:

```
v = [1.2, -2., 0.5] # need not be normalized
theta = pi/2
R = SO3.AngleAxis(theta, v)
print(R)
```

An interesting property of rotations is that every rotation matrix $R \in \mathrm{SO}(3)$ has an invariant linear subspace consisting of all vectors along the *axis of rotation*, i.e. $k \in \mathbb{R}^3$, $Rk = k$. Euler's rotation theorem states that for every rotation matrix R, there exists a unit vector $\hat{k}$ and angle θ such a

rotation of θ about $\hat{k}$ results in an equivalent rotation as the matrix R. To compute the unit axis and angle given a rotation matrix, use the `SO3` class method `angvec()`:

```
print( R.angvec() )
```

The vector $\hat{k}$ is the *eigenvector* of R corresponding to the eigenvalue $\lambda = 1$. Each rotation matrix has three eigenvalues, $\lambda_1 = 1$, $\lambda_2 = e^{i\theta}$, and $\lambda_3 = e^{-i\theta}$, where θ is the angle about $\hat{k}$ corresponding to the rotation represented by R.

3. Rigid Motions

3.1 The Special Euclidean Group

Rigid body motion is a pair (R, d) where $R \in \mathrm{SO}(3)$ is a rotation matrix and $d \in \mathbb{R}^3$ is a position representing pure rotation and pure translation. The group of all rigid motions is known as the *Special Euclidean Group* $\mathrm{SE}(3)$. An element $T \in \mathrm{SE}(3)$ is a 4×4 matrix of the form:

$$T = \begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix}$$

Similarly to the Special Orthogonal Group, the binary operator is matrix multiplication, the identity element of $SE(3)$ is the 4×4 Identity matrix, and the inverse of an element $T \in \mathrm{SE}(3)$ is given by:

$$T^{-1} = \begin{bmatrix} R^T & -R^T d \\ 0 & 1 \end{bmatrix} \tag{4}$$

Question 6. Verify that the inverse form for an element of $\mathrm{SE}(3)$ given by Eq. (4) is in fact the inverse. I.e. prove that $TT^{-1} = T^{-1}T = I$.

3.2 3D Translation

Translation in 3D can be represented by a *homogeneous transformation* matrix $T \in \mathrm{SE}(3)$ whose rotation component is identity. To create a transformation matrix with a translation of 1 m in the x-direction, 2 m in the y-direction, and 3 m in the z-direction w.r.t. the world frame, try:

```
T = SE3.Trans(1, 2, 3)
print(T)
```

Which should return the following 4×4 matrix:

```
   1          0          0          1
   0          1          0          2
   0          0          1          3
   0          0          0          1
```

To visualize the frame represented by T, try the following commands:

```
SE3().plot(frame="0", color="black")
T.plot(frame="1")
plt.show()
```

The spatialmath library also provides class methods for basic translations about the principle axes. To create basic translation matrices, use the following commands:

```
T1 = SE3.Tx(1)
print(T1)
T2 = SE3.Ty(2)
print(T2)
T3 = SE3.Tz(3)
print(T3)
```

Unlike rotations, translations always commute. To illustrate this, try composing different permutations of the three basic translation matrices and verify that their product is the same matrix:

```
print( T1 * T2 * T3 )
print( T1 * T3 * T2)
print( T2 * T1 * T3 )
# etc...
```

3.3 Composition of Homogeneous Transformation Matrices

Suppose a rigid motion consists of a rotation about the world z-axis, followed by a translation along the current z-axis, followed by a translation along the current x-axis, and finally followed by a rotation about the current x-axis. How can we go about computing the frame associated with this rigid motion? The spatialmath library provides class methods for all basic translations and rotations about the principle axes. The transformation can be computed with the following composition:

$$T = R_z(\theta)T_z(d)T_x(a)R_x(\alpha)$$

The above frame convention is known as the Denavit-Hartenberg or DH convention in which rigid transformations are parameterized by four scalar quantities: θ, d, a, and α. The geometric interpretation of these parameters (DH parameters) will be discussed in the next lab. For now, lets write a function to compute this frame given the four parameters:

```
def DH_frame(theta, d, a, alpha):
    T = SE3.Rz(theta) * SE3.Tz(d) * SE3.Tx(a) * SE3.Rx(alpha)
    return T
```

Note that an element of SE(3) is parameterized by six free variables (three for rotations and three for translations). However, the DH frame only has four parameters. Therefore it is not possible to

represent any arbitrary homogeneous transformation using this convention. Though, as we will see next week, four parameters is sufficient for describing the individual joints of a robot comprised of a chain of rigid links.

One might also wish to fully parameterize a rigid motion with six free variables. A standard transformation can therefore be thought of as a rigid translation followed by a rigid rotation in the translated frame. Create a function to generate a standard transformation:

```
def transform(x, y, z, R, P, Y):
    T = SE3.Trans(x, y, z) * SE3.RPY(R, P, Y)
    return T
```

The above transformation can be used to fully represent the degrees of a rigid body in free space.

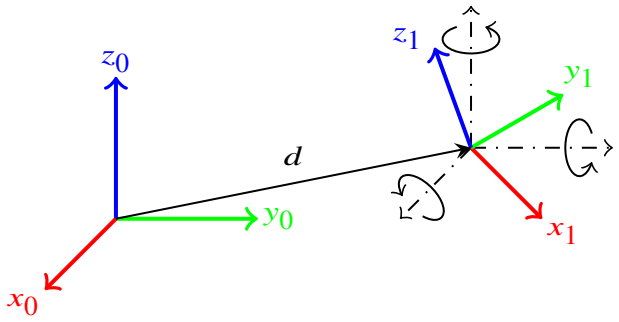


Figure 4: A standard transformation.

The combined position and orientation of a rigid body is often referred to as a *pose* (position + orientation).

Summary of Homogeneous Transformation Composition

The same interpretation regarding composition and ordering of transformations holds for 4×4 homogeneous transformations as for 3×3 rotations. Given a homogeneous transformation T_{01} relating the world frame to frame 1, if another frame is to be obtained by a rigid motion T relative to frame 1, then

$$T_{02} = T_{01}T$$

whereas if the second rigid motion is to be performed relative to the world frame, then

$$T_{02} = TT_{01}$$

Question 7. Consider the matrix product:

$$T = R_x(\alpha)T_x(b)T_z(d)R_z(\theta)$$

where R_x is a homogeneous rotation about the x-axis, T_x is a homogeneous translation along the x-axis, T_z is a homogeneous translation along the z-axis, and R_z is a homogeneous rotation about the z-axis.

a. Determine all permutations of these four matrices that yield the same homogeneous transformation matrix, T.
b. Let $\alpha = \pi/3$ rad, $b = 0.75$ m, $d = 0.35$ m, and $\theta = \pi/4$ rad. Compute the homogeneous transformation matrix T and include it in your lab writeup.

3.4 Relative Transformations

So far we have dealt with transformation frames expressed in the world frame. What if we wish to calculate a *relative* transformation between two arbitrary frames? This is often a desired quantity in robotics. An example application would be to calculate the relative transformation between a gripper and a target object.

Suppose we are given two transformation matrices, T_{01} and T_{02}, representing frames relative to the world frame. Using these two matrices, we would like to calculate the transformation T_{12}. Recall the composition property of transformations:

$$T_{02} = T_{01}T_{12}$$

which then implies:

$$T_{12} = (T_{01})^{-1}T_{02} = T_{10}T_{02}$$

To illustrate this, create two simple transformations from the world frame:

```
T_01 = transform(1, 2, 3, pi/4, pi/2, pi/3)
T_02 = transform(-0.5, 4, -1, pi, -pi/2, pi/6)
```

and compute their relative transformation:

```
T_12 = T_01.inv() * T_02
```

This rule can be applied to arbitrary relative transformations. Suppose we have two transformations represented as a sequence of basic transformation matrices:

$$T_{03} = T_{01}T_{12}T_{23}$$
$$T_{07} = T_{05}T_{56}T_{67}$$

Now we wish to calculate, say, the relative transformation from frame 7 to frame 3. First we need to identify a sequence of transformations that will take us from frame 7 to frame 3. The only common frame between the transformations is the world frame (frame 0). Therefore we will first calculate

the relative transform from frame 7 to the world frame. This can be done simply by inverting the matrix T_{07}:

$$T_{70} = (T_{07})^{-1} = (T_{05}T_{56}T_{67})^{-1} = (T_{67})^{-1}(T_{56})^{-1}(T_{05})^{-1}$$

Recall that inverting a product of matrices reverses the multiplication order. Then to calculate the entire transform from frame 7 to frame 3, multiply the two relative transforms together to get:

$$T_{73} = T_{70}T_{03} = (T_{67})^{-1}(T_{56})^{-1}(T_{05})^{-1}T_{01}T_{12}T_{23}$$

Question 8. Suppose you are given the following set of transformation matrices:

$$\{T_{01},\ T_{12},\ T_{13},\ T_{24},\ T_{05}\}$$

Calculate the following relative transformations using only the above transformations and their inverses: T_{34}, T_{23}, and T_{45}.

4. Exercises

Exercise 1. Spherical Pendulum Frame

Consider a spherical pendulum as shown in Figure 5 parameterized by two variables with respect to the world frame: $\theta \in [0, \pi]$ representing the polar angle and $\varphi \in [0, 2\pi)$ representing the azimuthal angle. The world frame is centered at the base of the pendulum post, and the pendulum frame is centered on the pendulum bob with the z-axis aligned with the pendulum string and the tangent y-axis pointing in the direction of increasing φ.

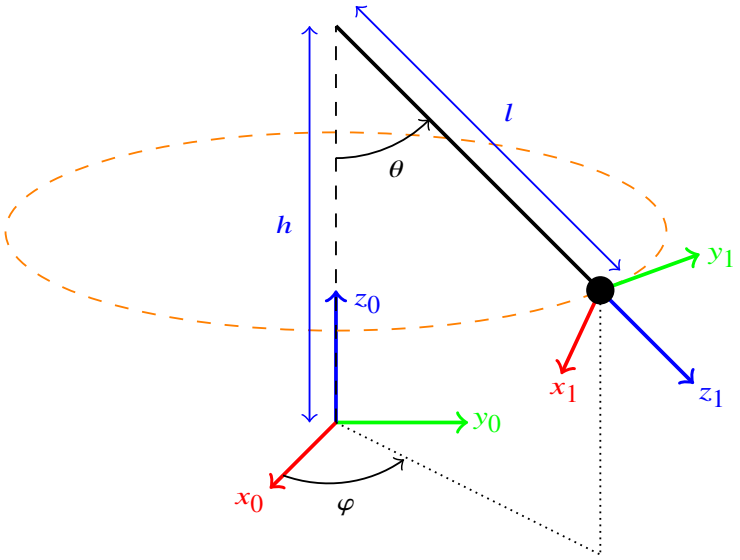


Figure 5: A spherical pendulum.

Deliverables:

a. Write the homogeneous transformation matrix representing the pendulum frame with respect to the world frame using the basic homogeneous transformation matrices. The final product should depend on both angle parameters as well as the height of the pendulum post h and the length of the pendulum string l.

b. Write a Python function for computing the pendulum frame parameterized by θ and φ for $h = l = 3$ m. Plot the pendulum frame and the world frame for three different values of θ and φ within their ranges. Include the plots in your lab writeup. Include the Python script along with your lab report submission. An example Python script is provided for you below:

```python
from spatialmath import *
import matplotlib.pyplot as plt
from math import pi

def pendulum_frame(theta, phi, h=3., l=3.):
    # compute the pendulum frame
    # T = ...
    # return T

# plot the world frame
SE3().plot(frame="0", color="black")
# plot the pendulum frame
pendulum_frame(pi/6, pi/4).plot(frame="1")
plt.show()
```

Exercise 2. Satellite Orbit

Tracking objects in space requires the definition of a frame of reference. In orbital mechanics, an inertial frame is usually defined with respect to the fixed stars. This is an approximation of course since the position of the stars does change over time. However, in the time scale that most problems are defined (on the order of months or years), this approximation proves to be adequate. A commonly used inertial frame is the **Earth-centered inertial** (ECI) frame which is centered at the origin of the Earth and fixed with respect to the stars. Now that we have a fixed inertial

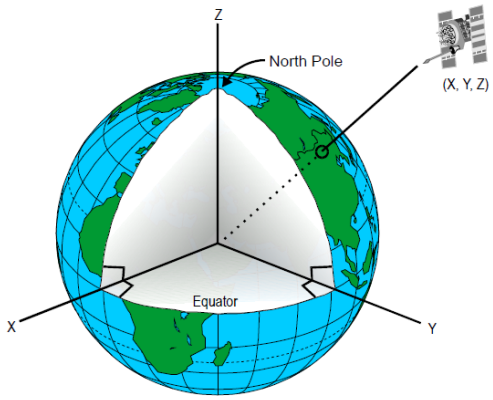


Figure 6: The Earth-centered inertial reference frame.

frame, we can begin to define non-inertial frames. The **Earth-centered, Earth-fixed** (ECEF) frame is a non-inertial (rotating) frame centered at the origin of the Earth, but rotating at a fixed angular velocity equal to that of the rotation of the Earth. Therefore the ECI and ECEF frames are coincident once every 24 hours. Suppose the ECI and ECEF frames are coincident every day at UTC 00:00. The ECEF frame is therefore given by:

$$T_{\text{ecef}}^{\text{eci}}(t) = R_z(\omega_e t)$$

where t is given in hours since UTC 00:00, ω_e is the rotational velocity of the Earth in rad/hr, $\omega_e \approx 0.26179$ rad/hr, and $R_z(\theta) \in \text{SE}(3)$. Now consider a satellite in a circular low Earth orbit with the following orbital parameters:

Epoch (UTC):	01:30
Inclination (i):	51.6331°
Orbital height (h):	413
Right ascension of ascending node (Ω):	231.5214°
Revolutions per day:	15.50263553
Mean anomaly at epoch (ν):	27.0500°

For more information about what these parameters mean and how they determine an orbit, see: Orbital Parameters.

Suppose we want to calculate the frame of the satellite with respect to the ECEF frame. All orbital parameters are defined with respect to the ECI frame. Let's define the satellite frame such that the x-axis is constantly pointing towards the center of the Earth and the z-axis is aligned with

the direction of travel (tangent to the orbital path). The satellite's frame w.r.t. the ECI frame is therefore:

$$T_{\text{sat}}^{\text{eci}}(t) = R_z(\Omega)R_x(i)R_z(w_s(t - \text{Epoch}) + \nu)T_x(h + r_e)R_z(\pi)R_x(\pi/2) \qquad (\dagger)$$

where t is hours elapsed since UTC 00:00, all angular parameters are in radians, the angular velocity of the satellite $w_s = 2\pi(\text{Revolutions per day})/24$ rad/hour, and r_e is the radius of the Earth $r_e \approx 6,378$ km.

Deliverables:

a. Open the Python file `src/satellite/satellite.py`. Complete the method `sat_transform(t, orbital_params)` by implementing Eq. (†) for the computation of the satellite frame in the ECI coordinates. For your convenience, the orbital parameters are provided as a dictionary. There is a `TODO:` flag in the method indicating where you should add your code. In your lab report, qualitatively describe the sequence of transformations in Eq. (†) from the ECI frame that result in the final satellite frame.

b. We want to track the satellite's position over time in the ECEF frame. To do this, you will need to calculate the relative frame $T_{\text{sat}}^{\text{ecef}}$. A method `ecef_transform` is provided for you for computing the ECEF frame as a function of time. In the loop within the `main` method, compute the relative transformation given `T_ecef` and `T_sat`. There is a `TODO:` flag in the loop indicating where you should add your code. Store the relative transformation in a variable called `T`.

c. Once we have the position of the satellite in the ECEF frame, we can calculate the polar coordinates of the satellite and plot its longitude and latitude over an atlas of the Earth. To compute the polar coordinates, we first extract the position vector of the satellite in the ECEF frame with the `SE3.t` property. Once we have the vector d, the polar coordinates can be computed using the following equations:

$$\text{lon} = \text{atan2}(d_y, d_x)$$
$$\text{lat} = \cos^{-1}(d_z/\|d\|) - \pi/2$$

Implement the calculation of the polar coordinates in the main loop. There is a `TODO:` flag indicating where you should add your code. Use the NumPy methods `np.atan2` and `np.arccos` to perform the computation.

d. Finally, we want to plot the angular position of the satellite over an atlas of the Earth. A plotting utility is provided for you already so no additional changes need to be made to the `main` function after the for loop. Once you make the previous changes, run the file and verify that the following plot is generated. Save it and include it in your lab report.

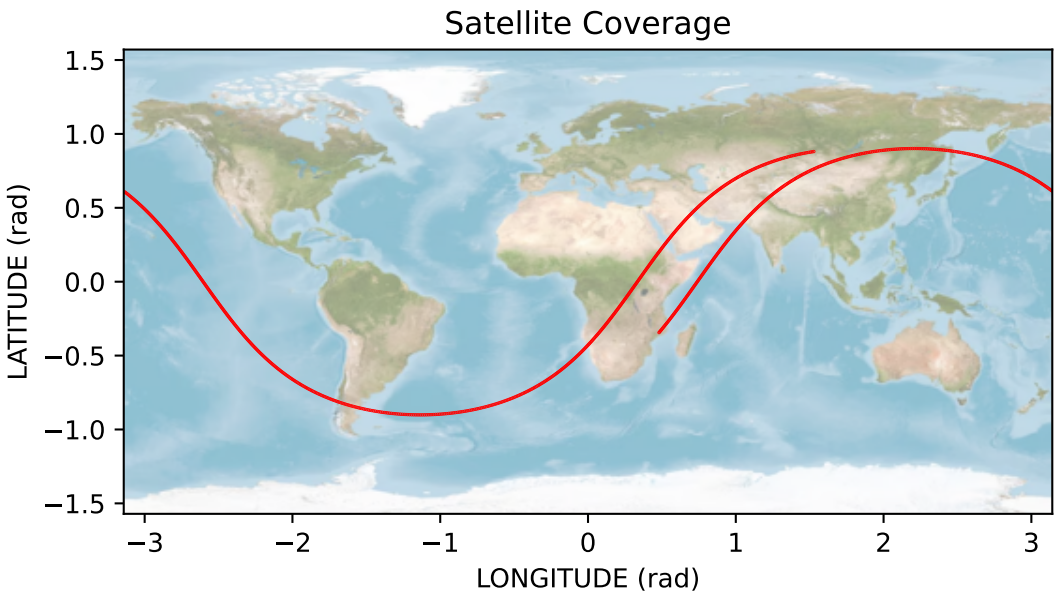
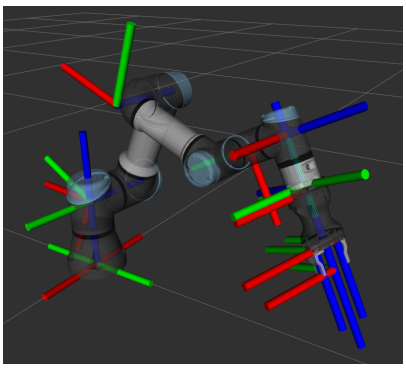


Figure 7: Orbital position of the satellite in polar coordinates.

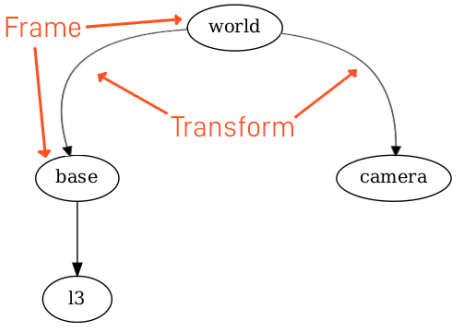
Exercise 3. Kinematic Tree Transformation Lookup

Many kinematic structures consist of a "tree" of rigid bodies. To reduce the computational complexity of storing all the transformations between rigid frames in a kinematic tree, frames are stored in a tree-like structure. Note for a robot consisting of N rigid links, there are $N \times (N-1)$ possible transformations that can be computed. Storing all of these transformations (and updating them over time) is computationally inefficient. Instead, we can store N instantaneous transformations between links and compute relative transformations on request.

Each node in the tree represents a frame tied to a rigid link, and the branches represent a transformation from the parent node to the child node. A kinematic tree is a special type of graph. Closed loops are not allowed. Each node (with the exception of the root node) only has one parent, but can have multiple children. Try to imagine a kinematic tree starting at your shoulder and extending through your arm down to your finger tips. To compute transformations between arbitrary nodes



(a) Frames for the links of a UR3e with a Robotiq Hand-E gripper.



(b) An example transfer frame tree.

in the tree, first a path is found connecting the two nodes. Next the individual transforms are multiplied starting at the source node along the path until the target node is reached. The computed transformation then represents the transformation between the source and target node, $T_{\text{target}}^{\text{source}}$. Recall the tree is directed, and transformations point from the parent node to the child node. When propagating up the tree, we simply take the inverse of the transformation. For example, the relative transform between the frames `base` and `camera` in Figure 8b is given by:

$$T_{\text{camera}}^{\text{base}} = \left(T_{\text{base}}^{\text{world}}\right)^{-1} \cdot T_{\text{camera}}^{\text{world}} = T_{\text{world}}^{\text{base}} \cdot T_{\text{camera}}^{\text{world}}$$

In the `lab-1` repo you are provided the skeleton of a transfer frame lookup class. A screenshot of the UR3e and Robotiq Hand-E frames are provided in a YAML file as well as a class method for loading a transfer frame tree from YAML files. Take a look at the file `frames_2025-08-23_21.47.46.pdf` for the entire TF tree from the actual UR3e and Robotiq Hand-E robot. The transfer frame tree is converted into a dictionary containing the following key-value pairs:

```
    "frame_name": {
        "parent": "parent_frame_name",
        "children": ["child1_frame_name", "child2_frame_name"],
        "xyzrpy": [0., 0., 0., 0., 0., 0.],
        "frame": SE3()
    }
```

Your task is to implement the method:

```
def get_transform(self, source: str, target: str):
```

which takes as input the `source` and `target` frame names and computes the relative transformation matrix between them. To do this you will need to accomplish two tasks:

1. Perform a tree search to find the shortest directed path between the `source` and `target` frames. Return the shortest path as a list of frame names as well as the direction (either up or down the tree indicating whether or not to invert the transformation along the branch).
2. Once you have the shortest directed path, compute the relative transformation by multiplying the branch transformation matrices along the path.

The method should handle requests for transformations between frames that are not included in the tree, and any request where the `source` and `target` frames are the same should return the identity transformation.

To perform the tree search, consider a simple tree with four nodes defined by the following dictionary:

```
tree = {
    "root_node": {
        "parent": None,
        "children": ["node_a", "node_b"]
    },
    "node_a": {
        "parent": "root_node",
        "children": []
    },
    "node_b": {
        "parent": "root_node",
        "children": ["node_c"]
    },
    "node_c": {
        "parent": "node_b",
        "children": []
    }
}
```

Finding a path between two nodes can be done in linear time. Given two nodes `source` and `target`, first we want to find a path from each node to the root node:

```
source = "node_a"
target = "node_c"

def get_path_to_root(tree, node: str) -> list[str]:
    # make sure the node is a key in the tree
    assert node in tree

    if tree[node]["parent"] == None:
        return [node]
    else:
        return [node] + get_path_to_root(tree, tree[node]["parent"])

source_path = get_path_to_root(tree, source)
target_path = get_path_to_root(tree, target)
```

Note that the addition + operator in Python concatenates lists. Now given the two paths, we simply work our way down the lists until we no longer have a common parent:

```
while (
        len(source_path) > 0 and
        len(target_path) > 0 and
        source_path[-1] == target_path[-1]
    ):
    common_node = source_path.pop()
    target_path.pop()

target_path.reverse()
path = source_path + target_path
print(path)
```

Since the kinematic tree is directed, you will have to alter the above code to include the direction of the path (up the tree until a common node is reached, then back down) so your transfer frame computation is correct.

Deliverables: Open the file `src/tf_tree/tf_tree.py` and complete the method `get_transform`. There is a `TODO:` flag indicating where you should add your code. Once completed, verify that your transformation lookup works by querying some relative transformations in the `main` method. Include the completed Python file along with your lab report submission.

The ROS `tf2` library provides an interface for looking up relative transformations. There code (written in C++) works identically to the code you implemented. Given a tf tree, a tree search is performed and then the relative transformation is computed by multiplying the transformation matrices along the path. For more information, see the their 2013 TePRA paper: tf: The Transform Library.

5. Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Your team is only responsible for submitting one lab report, make sure to include all members' names. Include your solutions to all of the question boxes in the lab report. Refer to the List of Questions to ensure you don't miss any. Include your full solution to the exercises in your lab report. Each exercise includes a box breaking down the required deliverables you must complete. If you wrote any Python programs during the exercises, include these files as a separate attachment. Submit your lab report as a `.pdf` along with any code to ELMS under the assignment for that weeks lab. You have one week to complete the lab report and submit it on the day of your next lab session. See the files section on ELMS for a lab report template. Below is a rubric breaking down the grading for this week's lab assignment:

Component	Points
Lab Report and Formatting	10
Question 1.	5
Question 2.	5
Question 3.	5
Question 4.	5
Question 5.	5
Question 6.	5
Question 7.	5
Question 8.	5
Exercise 1.	15
Exercise 2.	15
Exercise 3.	20
Total	100