

Spatial Math

Peter Corke

June 10, 2026

```
[3]: # import matplotlib.pyplot as plt
import numpy as np
np.set_printoptions(linewidth=100, formatter={'float': lambda x: f"{x:8.3g}" if_
↪abs(x) > 1e-10 else f"{0:8.3g}"})
```

1 Spatial Math

In robotics and computer vision we frequently need to represent position, orientation or pose in 3D space. These representations are a matrix or a vector, and NumPy is great for vectors and matrices. However, for rotations and poses we need matrices and vectors that have **special** structure.

- Pose is often represented by a 4×4 matrix.
 - but not all 4×4 matrices are represent poses
 - Pose is represented by a subset of all possible 4x4 matrices that belong to
 - * the **special** Euclidean group of order 3.
 - * the tangent space $se(3)$ which has 6 unique elements we can represent in a **special** 6-vector
- Orientation in 3D space be represented by a unit quaternion, a 4-vector with one constraint
- Lines in 3D can be represented by a 6-vector with 2 constraints

The objects in this package serve as gatekeepers that respect the constraints on the underlying matrices and vectors.

For example, the `SE3` class contains a NumPy array of shape $(4,4)$ but it only admits matrices that belong to the group $SE(3)$

Let's start simply

```
[4]: from spatialmath import SE3

SE3()
```

```
[4]:  1          0          0          0
      0          1          0          0
      0          0          1          0
      0          0          0          1
```

which is the identity matrix.

We could write this more laboriously as

```
[5]: SE3(np.eye(4))
```

```
[5]:  1      0      0      0
      0      1      0      0
      0      0      1      0
      0      0      0      1
```

The identity matrix was admitted by the SE3 gatekeeper because it belongs to SE(3).

Here's an example of the gatekeeper at work, keeping out the unworthy

```
[6]: try:
      SE3(np.zeros((4, 4)))
    except ValueError as e:
      print("bad thing")
```

bad thing

There are various class methods that act like constructors. Here's a very complex way to compound motion in the X-, Y-, and Z-directions.

```
[7]: SE3.Tx(1) * SE3.Ty(2) * SE3.Tz(3)
```

```
[7]:  1      0      0      1
      0      1      0      2
      0      0      1      3
      0      0      0      1
```

Note that `*` is the group operator and means composition. In Python terms it's matrix multiplication, the `@` operator, inside the box.

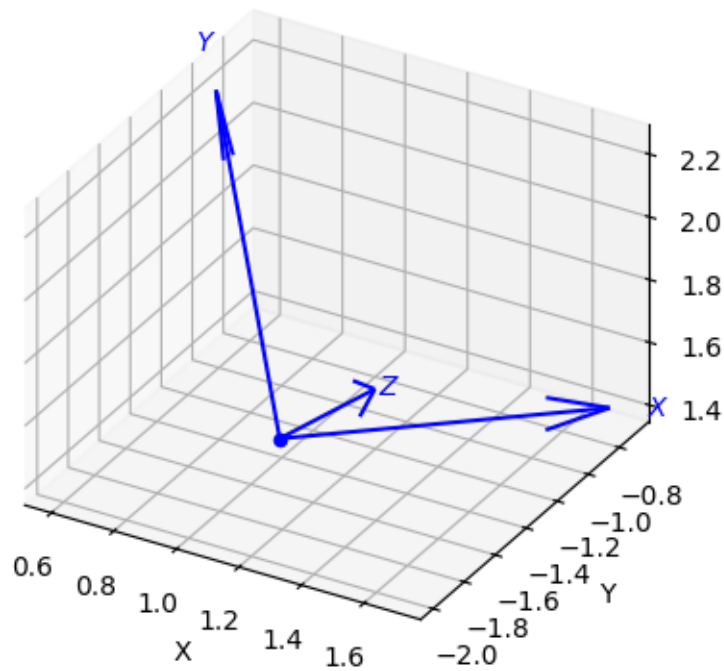
Now a sequence of translations and rotations about the canonical axes

```
[8]: T = SE3.Tx(1) * SE3.Rx(np.pi/4) * SE3.Tz(2) * SE3.Ry(np.pi/5) * SE3.Rz(np.pi/6)
      print(T)
```

```
  0.7006  -0.4045   0.5878   1
  0.7135   0.4046  -0.5721  -1.414
 -0.00639  0.8202   0.5721   1.414
  0        0        0        1
```

To understand the position and orientation of this pose it's very helpful to plot it

```
[9]: T.plot()
```



All objects can also print a compact single-line representation of themselves (see also `strline`)

```
[10]: T.printline()
```

```
t = 1, -1.41, 1.41; rpy/zyx = 55.1°, 0.366°, 45.5°
```

The upper left 3×3 matrix belongs to the group $SO(3)$ and that has its own class

```
[11]: from spatialmath import S03
```

```
R = S03(T)
R
```

```
[11]:  0.7006  -0.4045  0.5878
       0.7135   0.4046 -0.5721
      -0.00639  0.8202  0.5721
```

and we can convert it to various other representations. Roll/pitch/yaw angles are common, but it's really important to specify the order of rotation. In this example it's: $R_x(\text{yaw}) R_y(\text{pitch}) R_z(\text{roll})$

```
[12]: R.rpy(order="xyz")
```

```
[12]: array([ 0.524, 0.628, 0.785])
```

or in angle-axis form: a rotation about an axis specified as a unit vector

```
[13]: R.angvec()
```

```
[13]: (1.2253413396896766, array([ 0.74, 0.316, 0.594]))
```

or as a unit-quaternion (specifically a right-handed quaternion, not a left-handed/JPL quaternion)

```
[14]: from spatialmath import UnitQuaternion

q = UnitQuaternion(R)
q
```

```
0.8181 << 0.4254, 0.1816, 0.3416 >>
```

```
[14]:
```

The `*` operator also performs composition of `S03` and `UnitQuaternion` objects

```
[15]: UnitQuaternion(R*R)
```

```
0.3386 << 0.6961, 0.2971, 0.5590 >>
```

```
[15]:
```

```
[16]: q*q
```

```
0.3386 << 0.6961, 0.2971, 0.5590 >>
```

```
[16]:
```

and we see the equivalence of rotational composition using rotation matrices or unit quaternions.

The rotational conversions above have equivalent methods for `SE3` object, they just ignore the translation component

```
[17]: print(T.rpy())
print(UnitQuaternion(T))
```

```
[ 0.962 0.00639 0.794]
0.8181 << 0.4254, 0.1816, 0.3416 >>
```

We can interpolate between rotations, here's an example for unit quaternions

```
[18]: q1 = UnitQuaternion.Rx(np.pi/2)
q2 = UnitQuaternion.Ry(-np.pi)
q1.interp(q2, 0.5)
```

```
0.5000 << 0.5000, -0.7071, 0.0000 >>
```

```
[18]:
```

which is done using SLERP. Interpolating `S03` objects operates in unit quaternion space.

All the objects are subclasses of `list` and so can do all the things you can do with a list: slice, iterate, insert, append, pop etc.

```
[19]: R = S03([S03.Rx(np.pi/2), S03.Ry(np.pi/3), S03.Rz(np.pi/4)])
print(len(R))
print(R[1])

for _ in R:
    print(_.rpy())
```

```
3
  0.5      0      0.866
  0        1      0
 -0.866    0      0.5

[ 1.57      0      0]
[ 0        1.05    0]
[ 0        0      0.785]
```

Let's create a longer list, rotations around the Y-axis from 0 to $\pi/2$ radians

```
[20]: Rt = S03.Ry([theta for theta in np.linspace(0, np.pi/2, 50)])
print(len(Rt))
print(Rt[0])
print(Rt[49])
```

```
50
  1      0      0
  0      1      0
  0      0      1

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

In one line we can convert that to a listy quaternion

```
[21]: qt = UnitQuaternion(Rt)
print(len(qt))
print(qt[0])
print(qt[49])
```

```
50
1.0000 << 0.0000, 0.0000, 0.0000 >>
0.7071 << 0.0000, 0.7071, 0.0000 >>
```

We can also do broadcasting very easily. Here we apply $R_x(\pi/2)$ to every element of Rt

```
[22]: Rtx = S03.Rx(np.pi/2) * Rt
print(len(Rtx))
print(Rtx[0])
print(Rtx[49])
```

50

1	0	0
0	0	-1
0	1	0
0	0	1
1	0	0
0	1	0

Let's return to the somewhat random SE(3) matrix we computed earlier

```
[23]: T
```

```
[23]: 0.7006  -0.4045  0.5878  1
      0.7135  0.4046 -0.5721 -1.414
      -0.00639 0.8202  0.5721  1.414
      0        0        0        1
```

and compute its logarithm

```
[24]: T.log()
```

```
[24]: array([[ 0, -0.728,  0.387,  0.191],
            [ 0.728,  0, -0.907, -0.91],
            [-0.387,  0.907,  0,  2.15],
            [ 0,  0,  0,  0]])
```

which is another 4x4 matrix with special (different special) structure. The bottom row is zero, and the upper left 3×3 matrix is skew symmetric. It has only 6 unique values.

We can represent this as a twist (v, ω) , where $v \in \mathbb{R}^3$ is the moment and encodes the position of the screw axis in space and the pitch of the screw, and $\omega \in \mathbb{R}^3$ is the direction of the screw axis.

```
[25]: from spatialmath import Twist3
```

```
S = Twist3(T)
print(S)
```

```
(0.19095 -0.90969 2.1536; 0.90655 0.38689 0.72798)
```

which contains all the information needed to reconstruct the original SE(3) matrix

```
[26]: S.SE3()
```

```
[26]: 0.7006  -0.4045  0.5878  1
      0.7135  0.4046 -0.5721 -1.414
      -0.00639 0.8202  0.5721  1.414
      0        0        0        1
```

The twist encodes motion as a rotation of a frame around a screw axis. For this example, the pitch of the screw is

```
[27]: # h = S.pitch  
      # print(h)
```

that is, for every radian of rotation about the axis, we translate `h` along the axis.

The axis of the screw is

```
[28]: # S.line()
```

which is of type

```
[29]: # type(_)
```

which is a Plücker representation of a 3D line, another 6-vector, but this one has 2 constraints.

The pose represented by `T` can be obtained by rotating a frame from the origin around a screw, with the pitch and axis described above, by an angle of

```
[30]: print(S.theta, "radians")
```

1.2253413396896766 radians

2 The classes are highly polymorphic

- all the objects that include rotation share common methods like `Rx Ry Rz rpy interp` etc.
- all the objects that include translation share common methods `Tx Ty Tz` etc.
- they can all plot themselves
- their constructors will convert from other types, eg. `S03(UnitQuaternion(...))`
- with no argument, the constructor generates a null motion
- they all have a `Rand` constructor
- they all inherit from `list`

3 There's a lot more

We've barely scratched the surface. There are also:

- many functions are SymPy friendly
- lots more conversions back and forth between the representations we've touched on.
- 2D versions of all of this: `SE2`, `S02` and `Twist2`
- classes for 3D planes and lines (which we touched on), and methods for point-line distance, line-line and line-plane intersection.
- a handy graphics library for lines, arrows, boxes, circle, ellipses, etc.