Part II

Rigid Systems

The study of rigid systems is an essential starting point for foldable mechanisms, even for flexible systems, because the representations we use to understand rigid systems can be adapted and expanded in order to help us understand flexible systems as well. This part will take you through introductory concepts like vectors and rotations, as well as highlight different implementations you can use to work with geometry in Python.

Chapter 12

Reference Frames

When analyzing a system, sometimes it's convenient or simple to represent a system in a specific way. We all know of the Cartesian system of coordinates which span a three dimensional space, but did you know that there are an infinite number of ways to describe that same space? Just as we are familiar with the x,y, and z directions, we can use different reference frames to represent the same coordinates and vectors in different ways.

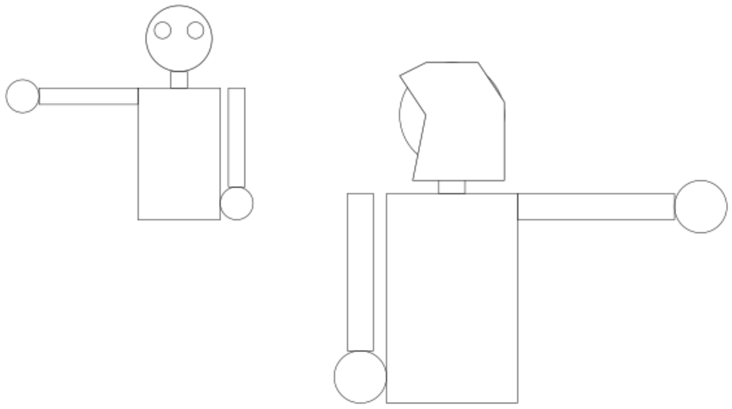


Figure 12.1: Picture of Professor and Students

Take the example of your professor standing in front of you, giving a lecture. If both of you were to stretch your right hand out to your right, you wouldn't be pointing in the same direction. To get to the door *behind you*, your professor would need to go *forward*. Direction is relative to each of you, based on your frame of reference.

Sometimes the ability to select your directional representation is useful in kinematics and dynamics, because depending on the representation, it may take many more or far fewer variables to do so. For example, when discussing the motion of a vehicle such as a plane or car, it can be useful to describe the forces acting on that vehicle from the perspective of the vehicle, even if you're more interested where the vehicle goes relative to your own perspective standing on the ground. The vehicle's perspective and the ground's perspective, in terms of their directional components, would be considered separate reference frames.

In three-dimensional space, frames provide three orthogonal unit basis vectors, which can be used to construct vectors. **There are infinite ways to describe the same 3D space.**

- Frames Are
  - Containers for holding basis vectors
- Frames Are Not
  - Attached to a point in space. **Only hold rotation information**
- Frames Do Not
  - Translate. **Only rotate.**
  - Have mass.
  - Flex or stretch
  - Have dimensions

12.1 Inertial Reference (Newtonian) Frames

Newtonian frames are reference frames that are not accelerating.

Newtonian reference frames are useful because many algorithms / expressions (such as the equations for vector derivatives) assume a non-accelerating reference frame.

https://en.wikipedia.org/wiki/Inertial_frame_of_reference

Chapter 13

Basis Vectors

13.1 Definition

A reference frame contains a set of unique *basis vectors*, which, in linear algebra terms, span an R^3 space. As long as these basis vectors span the space, they are capable of describing any vector within that space. However, there are some other useful qualities of these basis vectors, which typically make life easier, which must also be enforced when operating on and between vectors.

Orthogonal: The set of basis vectors in a reference frame are orthogonal, or mutually perpendicular to each other. This means that there are no shared components of vectors; each basis vector is completely independent of each other.

A more mathematical equivalent is that

$$\begin{aligned}
0 =& \hat{b}_1 \cdot \hat{b}_2 && (13.1)\\
=& \hat{b}_2 \cdot \hat{b}_3 && (13.2)\\
=& \hat{b}_3 \cdot \hat{b}_1 && (13.3)\\
& && (13.4)
\end{aligned}$$

, where $\hat{b}_1$, $\hat{b}_2$, and $\hat{b}_3$ are the three basis vectors of a reference frame.

Normal: Basis vectors in a reference frame are *normal*, meaning their length is 1, or

$$\begin{aligned}
1 =& \hat{b}_1 \cdot \hat{b}_1 && (13.5)\\
=& \hat{b}_2 \cdot \hat{b}_2 && (13.6)\\
=& \hat{b}_3 \cdot \hat{b}_3 && (13.7)\\
& && (13.8)
\end{aligned}$$

. In other words, they are *unit vectors*.

13.2 Vectors

Vectors are multidimensional geometric entities that indicate a *direction* and a *magnitude* in their respective space.

In other words, vectors are linear combinations of basis vectors from one or more reference frames. When reference frames are defined, their associated basis vectors may then be scaled and added together to create vector expressions.

A Note on Notation:

- Regular vectors use $\vec{v}$
- Unit vectors use $\hat{v}$

Chapter 14

Rotations

14.1 Introduction

While there may be many ways to navigate and describe the same three-dimensional space using reference frames, it is also necessary and desireable to be able to change representations; this can be useful for interpreting motion from a different perspective, for adding forces or torques to a system using directional components which are a more natural description, or in order to perform mathematical operations between vectors which are represented by different basis vectors. The method by which we represent one frame to another is through the concept of rotations.

Rotations define the relationship between two *frames*, whether those frames are explicitly used or not in your problem. Though rotations are represented as 3x3 matrices, they may be defined a number of ways, including (axis, angle) representations, quaternions, Euler parameters, Euler vectors, Rodrigues' parameters, etc. Each representation has its benefits and drawbacks, but at the end of the day, each of these methods is simply a way to define the rotational relationships between reference frames and the basis vectors they contain. A rotation is a specific type of vector transformation that **1)** preserves length and **2)**, preserves angles between vectors. Generically, we may think of rotational transformations as permitting the same vector to be represented using a new set of basis vectors, or, in another way of thinking, to actually rotate a body into a new orientation with relation to some other frame.

As stated above, rotations can be defined in a number of ways. You may:

- supply a 3x3 matrix directly
- generate one using an axis, angle pair
- create one by defining one or more simpler rotations along an x,y,or z axis.

Rotations can also hold variables as well as constants, and can be differentiated.

14.2 Theory

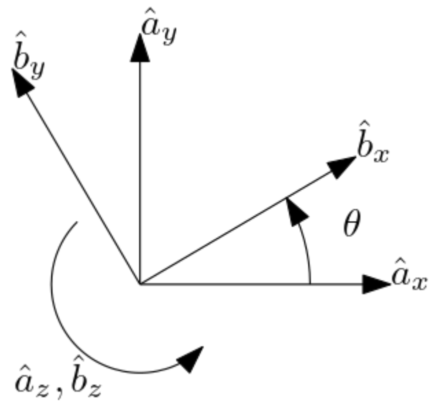


Figure 14.1: caption

14.3 Usage

Rotations are used extensively in kinematics and dynamics. Some use cases include

- Generating basis vectors for use in general-purpose vector creation
- They can be used with constants to generate fixed changes of reference
- They can be used conjunction with differentiable state variables to determine rotational velocity and acceleration between frames.

Rotations are typically generated in a sequential order from a fixed, or Newtonian reference frame (a non-accelerating world frame).

14.4 Two perspectives on the same tool

A rotation can be used and thought of simultanesoulsy as

- something that physically rotates a vector, point, or other geometry in physical space
- something that expresses the same vector, point, or other geometry in from a different reference frame.

Both of these operations are critical operations in the field of robotics, but the expresssions one uses for performing both tasks are the same, and the words we use can sometimes be ambiguous. For example:

The robot's arm rotates 30 degrees about the z axis:

The robots end effector, expressed in the N frame:

14.5 Rotation Tables / matrices

A rotation table more explicitly helps us relate vectors in two frames together. In this case ${}^aR^b$, when multiplied by a vector containing basis vectors in the B frame, results in an expression consisting of basis vectors in the A frame.

$$\vec{v}_{(a)} = {}^aR^b\vec{v}_{(b)}$$

$${}^aR^b = \begin{array}{c|ccc} & b_x & b_y & b_z \\ \hline a_x & a_x \cdot b_x & a_x \cdot b_y & a_x \cdot b_z \\ a_y & a_y \cdot b_x & a_y \cdot b_y & a_y \cdot b_z \\ a_z & a_z \cdot b_x & a_z \cdot b_y & a_z \cdot b_z \end{array} = \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} v_4 \\ v_5 \\ v_6 \end{bmatrix}$$

14.6 Rotations are Orthonormal[1]

$$R^T R = R R^T = I$$

$$R^T = R^{-1}$$

14.6.1 Unit Length row and column vectors

$$1 = |\vec{v}_1| = |\vec{v}_2| = |\vec{v}_3| = |\vec{v}_4| = |\vec{v}_5| = |\vec{v}_6|$$

14.6.2 Orthogonal row and column vectors

$$0 = \vec{v}_1 \times \vec{v}_2 = \vec{v}_2 \times \vec{v}_3 = \vec{v}_3 \times \vec{v}_1$$

$$0 = \vec{v}_4 \times \vec{v}_5 = \vec{v}_5 \times \vec{v}_6 = \vec{v}_6 \times \vec{v}_4$$

[1] https://en.wikipedia.org/wiki/Orthogonal_matrix

14.7 Simple rotations about major axes

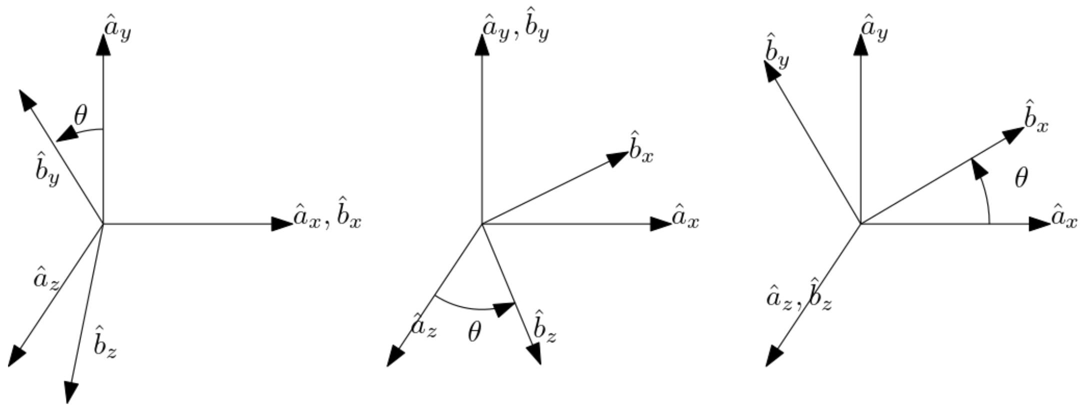


Figure 14.2: caption

$$ {}^aR^b_x(\theta) = \begin{array}{c|ccc} & b_x & b_y & b_z \\ \hline a_x & 1 & 0 & 0 \\ a_y & 0 & \cos\theta & -\sin\theta \\ a_z & 0 & \sin\theta & \cos\theta \end{array} $$

$$ {}^aR^b_y(\theta) = \begin{array}{c|ccc} & b_x & b_y & b_z \\ \hline a_x & \cos\theta & 0 & \sin\theta \\ a_y & 0 & 1 & 0 \\ a_z & -\sin\theta & 0 & \cos\theta \end{array} $$

$$ {}^aR^b_z(\theta) = \begin{array}{c|ccc} & b_x & b_y & b_z \\ \hline a_x & \cos\theta & -\sin\theta & 0 \\ a_y & \sin\theta & \cos\theta & 0 \\ a_z & 0 & 0 & 1 \end{array} $$

14.8 Compound Rotations

Rotations can be chained together and resolved into a single expression:

$$ {}^aR^c = {}^aR^{b\,b}R^c $$

14.9 External References

- https://en.wikipedia.org/wiki/Rotation_matrix

Chapter 15

Using the `numpy` Package For Kinematics

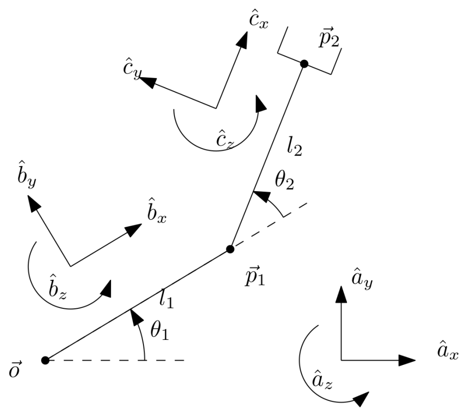


Figure 15.1: two-dof, serial, planar robot

Numpy is a general-purpose Python package for working with arrays and matrices. It, alongside is companion library `scipy`, contains a lot of similar functionality you might find when working with matrices in Matlab. Today we are going to learn how to manipulate vectors and express them in different reference frames using rotation matrices

We will need to import the numpy library, the math library, and the matplotlib library to plot our results. Numpy allows us to create and manipulate arrays, math contains functions like sin and cos, as well as constants like π, and matplotlib is our plotting library.

```
import numpy
import math
import matplotlib.pyplot as plt
```

To define a rotation matrix, we want to define theta, the angle of rotation, as well as the axis of rotation.

```
theta = 30*math.pi/180
```

A rotation between two frames about the z axis can be described for three dimensional vectors as

```
R = numpy.array([[math.cos(theta),-math.sin(theta),0],[math.sin(theta),math.cos(theta),0],[0,0,1]])
R
```

```
array([[ 0.8660254, -0.5      ,  0.       ],
       [ 0.5      ,  0.8660254,  0.       ],
       [ 0.       ,  0.       ,  1.       ]])
```

Within the b frame, we can define the unit vector along the x axis as a matrix that looks like

$$\hat{b}_x = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Similarly, the $\hat{c}_x$ unit vector will look quite similar in its own frame:

$$\hat{c}_x = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

```
b_x_in_b = numpy.array([[1,0,0]]).T
c_x_in_c = numpy.array([[1,0,0]]).T
b_x_in_b
```

```
array([[1],
       [0],
       [0]])
```

However, when expressed in another frame, the $\hat{b}_x$ vector has components in the $\hat{a}_x$ and $\hat{a}_y$ direction

```
b_x_in_a = R@b_x_in_b
b_x_in_a
```

```
array([[0.8660254],
       [0.5      ],
       [0.       ]])
```

If we want to create a second rotation matrix to express the B and C frames to each other, we would have to write out another expression in Python, with a separate value for theta. Or, we can write a function to regenerate a new Rz rotation matrix whenever we want.

```
def R_z(theta):
    R = numpy.array([[math.cos(theta),-math.sin(theta),0],
                     [math.sin(theta),math.cos(theta),0],
                     [0,0,1]])
    return R
```

By running it, we see the `Rz()` function only needs a new theta value to generate a new matrix. This is a handy reuse of code.

```
R_z(60*math.pi/180)
```

```
array([[ 0.5      , -0.8660254,  0.       ],
       [ 0.8660254,  0.5      ,  0.       ],
       [ 0.       ,  0.       ,  1.       ]])
```

Now let's apply this to create two rotation matrices, one for theta_1 and one for theta_2.

```
theta1 = 30*math.pi/180
theta2 = 30*math.pi/180
R_z_1 = R_z(theta1)
R_z_2 = R_z(theta2)
```

We can define constants, such as the length of a vector, as a literal value, saved in a Python variable. Combining with our numpy vectors, this helps us define the vector components of a robot arm in each local coordinate frame

```
l1 = 1
l2 = 1.1
v1_in_b = l1*b_x_in_b
v2_in_c = l2*c_x_in_c
```

We need to express vectors in the same frame before we can operate on them together using numpy literals, though. For example, `v2_in_c` is initially expressed in the C frame. We can use `R2` to express it in the B frame

```
v2_in_b = R_z_2@v2_in_c
v2_in_b
```

```
array([[0.95262794],
       [0.55      ],
       [0.        ]])
```

And then in the A frame.

```
v2_in_a = R_z_1@v2_in_b
v2_in_a
```

```
array([[0.55      ],
       [0.95262794],
       [0.        ]])
```

Likewise, we can express `v1_in_b` , initially expressed as a linear combination of unit vectors from the B frame, in A.

```
v1_in_a = R_z_1@v1_in_b
v1_in_a
```

```
array([[0.8660254],
       [0.5      ],
       [0.       ]])
```

`v1_in_a` and `v2_in_a` only describe each link's travel from proximal to distal joint, not its position in space from an origin. If we define the origin of our robot as $\vec{o}$, or

```
o = numpy.array([[0,0,0]]).T
o
```

```
array([[0],
       [0],
       [0]])
```

We can then express $\vec{p}_1$ and $\vec{p}_2$ as points in space, whose distance is relative to the origin.

```
p1 = o+v1_in_a
p1
```

```
p2 = o+v1_in_a+v2_in_a
p2
```

If we want to plot all the points in our system at once, it becomes convenient to collect them into a list, and then a numpy array. The `squeeze()` function helps eliminate dimensions of length zero.

```
points = numpy.array([o,p1,p2])
points = points.squeeze()
points
```

```
array([[0.        , 0.        , 0.        ],
       [0.8660254 , 0.5       , 0.        ],
       [1.4160254 , 1.45262794, 0.        ]])
```

You can see the first column is the x components (in our base reference frame) of $\vec{o}$, $\vec{p}_1$, and $\vec{p}_2$. The second column is the y components, and the third column of zeros, indicates the z components. Ignoring the z column, we can plot all three x-y points at once using the following commands, which first plots the points, and then scales the axis so that they are scaled equivalently.

```
plt.plot(*(points[:,:2].T))
plt.axis('equal')
```

```
(-0.07080127018922196,
 1.486826673973661,
 -0.07263139720814413,
 1.5252593413710265)
```

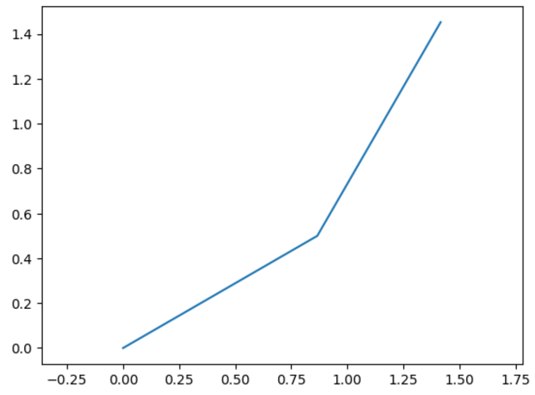


Figure 15.2: caption

This is a quick example of working with numerical rotation matrices using the most basic of packages in Python. Future sections will show us how to improve working with vectors so that common mistakes, such as operations between vectors in two different bases, is gracefully handled.

Chapter 16

Quaternions

16.1 Complex numbers

$$i^2 = -1$$

16.2 Quaternion Structure

$$q = a + bi + cj + dk$$

where i, j, and k represent an orthogonal set of complex numbers.

$$ii = jj = kk = -1$$

16.3 Operations between i, j, and k

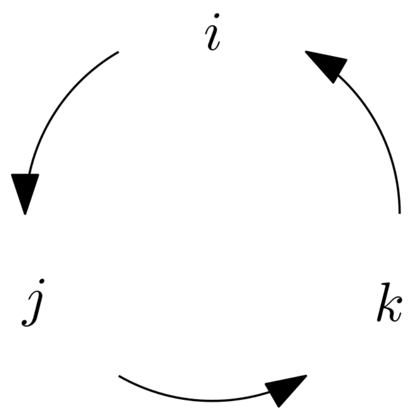


Figure 16.1: The right-hand rule applied to i, j, and k

$$ij = -ji = k$$

$$jk = -kj = i$$

$$ki = -ik = j$$

$$ijk = -1$$

proof: $ijk = (ij)k = (k)k = -1$

Note: This means that quaternion complex components are non-commutative, eg $ij \neq ji$, $jk \neq kj$, $ki \neq ik$.

16.4 Pure Quaternion

A pure quaternion is a quaternion with no real component, eg $bi + cj + dk$

16.5 Quaternion Operations

Given

$$q_1 = a + bi + cj + dk$$

$$q_2 = e + fi + gj + hk$$

16.5.1 Addition

$$\begin{aligned} q_1 + q_2 &= a + bi + cj + dk + e + fi + gj + hk \\ &= (a+e) + (b+f)i + (c+g)j + (d+h)k \end{aligned} \tag{16.1}$$

16.5.2 Multiplication

$$\begin{aligned} q_1 q_2 =& (a + bi + cj + dk)(e + fi + gj + hk) && (16.2) \\ =& ae + afi + agj + ahk + bei + bfii + bgij + bhik + cej \\ & + cfji + cgjj + chjk + dek + dfki + dgkj + dhkk && (16.3) \\ =& (ae - bf - cg - dh) + (af + be + ch - dg)i \\ & + (ag - bh + ce + df)j + (ah + bg - cf + de)k && (16.4) \end{aligned}$$

Note: quaternion multiplication is non-commutative, ie $q_1 q_2 \neq q_2 q_1$

16.6 Complex Conjugate

for $q_1 = a + bi + cj + dk$, its conjugate q^* may be represented by

$$q_1^* = a - bi - cj - dk$$

16.6.1 Conjugate properties

$$(q_1^*)^* = q_1$$

$$q_1 q_1^* = q_1^* q_1 = |q_1|^2$$

$$(q_1 q_2)^* = q_2^* q_1^*$$

$$|q_1 q_2|^2 = |q_1|^2 |q_2|^2$$

16.6.2 Inverse

$$q_1^{-1} = \frac{q_1^*}{|q_1|^2}$$

$$q_1 q_1^{-1} = q_1^{-1} q_1$$

16.7 Unit Quaternions

Unit quaternions observe the additional constraint that their length is 1. In other words, for a quaternion $q_1 = a + bi + cj + dk$,

$$|q_1| = \sqrt{q_1 q_1^*} = \sqrt{a^2 + b^2 + c^2 + d^2} = 1$$

thus

$$a^2 + b^2 + c^2 + d^2 = 1$$

This additional constraint ensures that mutliplying any quaternion by a unit quaternion preserves the length of the result, much like how multiplying an normal matrix with a vector preserves length of the vector.

16.7.1 Unit Quaternion Inverse

For a unit quaternion q, its inverse can be simply represented by

$$q^{-1} = q^*$$

because $|q| = 1$

16.7.2 Rotations with unit quaternions

A unit quaternion may be used to represent a rotation in 3D space. For a cartesion point (u_1, u_2, u_3) and its *pure* quaternion representation $p = u_1 i + u_2 j + u_3 k$, the operation

$$p' = rpr^{-1} = rpr^* \tag{16.5}$$

can be used to transform point p to p', or to express a point p in a different frame as p'. In this expression, r represents a unit quaternion and p' represents the rotated / expressed point.

Remember, you can only use a quaternion's conjugate in place of its inverse when its length is one (a *unit* quaternion).

16.7.3 Unit quaternions as half rotations

Any unit quaternion can be decomposed in the following manner

$$r = \cos(\theta/2) + \sin(\theta/2)(u_x i + u_y j + u_z k)$$

where θ represents the rotation about a unit vector $\vec{u}$, where u_x, u_y, and u_z represent the x, y, and z components, with $u_x^2 + u_y^2 + u_z^2 = 1$.

16.7.4 Constructing a Rotation Matrix from a unit quaternion

For a unit quaternion $r = \cos(\theta/2) + \sin(\theta/2)(u_x i + u_y j + u_z k)$,

$$v_1 = rir^* \tag{16.6}$$
$$v_2 = rjr^* \tag{16.7}$$
$$v_3 = rkr^* \tag{16.8}$$

$$R = \begin{bmatrix} \text{imag}(v_1) & \text{imag}(v_2) & \text{imag}(v_3) \end{bmatrix}$$

16.8 Compound Rotations

A point can be rotated using two successive unit quaternions (or expressed in two successive frames) like this:

$$p' = r_1 p r_1^*$$

$$p'' = r_2 p' r_2^*$$

$$p'' = r_2 r_1 p r_1^* r_2^*$$

16.9 Example: Calculate a unit quaternion from a supplied axis and angle

Reference frame B is rotatated relative to reference frame A (basis vectors $\hat{a}_x$, $\hat{a}_y$, and $\hat{a}_z$) by $\theta = \pi/3$, about the axis defined by an arbitrary vector $\vec{w} = 3\hat{a}_x + 4\hat{a}_y + 2\hat{a}_z$. ($\vec{w}$ has not been normalized).

Find r, the unit quaternion that represents this rotation.

$$r = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}(u_x i + u_y j + u_z k)$$

$$\hat{u} = u_x\hat{a}_x + u_y\hat{a}_y + u_z\hat{a}_z = \frac{\vec{w}}{|\vec{w}|} = \frac{3\hat{a}_x + 4\hat{a}_y + 2\hat{a}_z}{3^2 + 4^2 + 2^2} = \frac{3}{29}\hat{a}_x + \frac{4}{29}\hat{a}_y + \frac{2}{29}\hat{a}_z$$

$$r = \cos\frac{\pi}{6} + \sin\frac{\pi}{6}\left(\frac{3}{29}i + \frac{4}{29}j + \frac{2}{29}k\right)$$

$$r^* = \cos\frac{\pi}{6} - \sin\frac{\pi}{6}\left(\frac{3}{29}i + \frac{4}{29}j + \frac{2}{29}k\right)$$

16.10 Example: Express a vector in another reference frame

Vector $\vec{p}$ is expressed in the B frame, with $\vec{p} = 1\hat{b}_x + 2\hat{b}_y + 3\hat{b}_z$. Express $\vec{v}$ in A.

The quaternion representation for $\vec{p}$ is simply

$$p = i + 2j + 3k$$

The expression for

$$p_{\text{in A}} = rpr^*$$

If we use the value for r from the last expression, this can be expanded to

$$p_{\text{in A}} = \left(\cos\frac{\pi}{6} + \sin\frac{\pi}{6}\left(\frac{3i}{29} + \frac{4j}{29} + \frac{2k}{29}\right)\right)(i + 2j + 3k)\left(\cos\frac{\pi}{6} - \sin\frac{\pi}{6}\left(\frac{3i}{29} + \frac{4j}{29} + \frac{2k}{29}\right)\right)$$

The final answer can then be obtained by expanding this expression, ensuring that the order of i,j, and k are preserved.

16.11 Example

Say we have a vector $\vec{p} = \begin{pmatrix}2\\1\\0\end{pmatrix}$ that we want to rotate about the vector $\vec{r} = \begin{pmatrix}1\\0\\0\end{pmatrix}$ by the amount $\theta = 90^\circ$. We form two quaternions:

$$p = 2i + 1j + 0k$$

$$r = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)(1i + 0j + 0k)$$

Noting that this quaternion is already a unit quaternion ($|r| = 1$), We can then compute the quaternion p' with the equation

$$p' = rpr^*$$

The result in vector notation is $\vec{p}' = \begin{pmatrix}2\\0\\1\end{pmatrix}$.

16.12 Example

Say we have a vector $\vec{p} = \begin{pmatrix}2\\1\\0\end{pmatrix}$ that we want to rotate about the vector $\vec{r} = \begin{pmatrix}1\\1\\0\end{pmatrix}$ by the amount $\theta = 180^\circ$. We form two quaternions:

$$p = 2i + 1j + 0k$$

$$r = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\frac{(1i + 1j + 0k)}{\sqrt{2}}$$

Noting that the original vector needed to be normalized, (hence dividing by $\sqrt{2}$) or else the resulting quaternion would have needed to be normalized ($r_{new} = \frac{r}{|r|}$), we can then compute the quaternion p' with the equation

$$p' = rpr^*$$

The result in vector notation is $\vec{p}' = \begin{pmatrix}1\\2\\0\end{pmatrix}$.

16.13 Example

Say we have a vector v representing a robot arm, initially expressed in its native (non-inertial) reference frames, as in

$$\vec{v}_1 = l_b \hat{b}_x$$

$$\vec{v}_2 = l_c \hat{c}_x$$

$$\vec{p}_{end} = \vec{v}_1 + \vec{v}_2$$

In quaternion form,

$$v_{1(B)} = l_b i$$

$$v_{2(C)} = l_c i$$

If frame B is rotated from the base inertial frame A by θ_1 about $\hat{a}_z$, we can create a unit quaternion which represents this rotation.

$$r_{ab} = \cos\left(\frac{\theta_1}{2}\right) + \sin\left(\frac{\theta_1}{2}\right)(0i + 0j + 1k)$$

We can then express $v_{1(B)}$ in the A frame:

$$v_{1(A)} = r_{ab} v_{1(B)} r_{ab}^*$$

$$v_{1(A)} = r_{ab}(l_b i) r_{ab}^*$$

Similarly, if frame C is rotated from B by θ_2 about $\hat{b}_z$, we can create a second unit quaternion which represents this rotation.

$$r_{bc} = \cos\left(\frac{\theta_2}{2}\right) + \sin\left(\frac{\theta_2}{2}\right)(0i + 0j + 1k)$$

We can then express $v_{2(C)}$ in the B frame, and then the A frame, respectively:

$$v_{2(B)} = r_{bc} v_{2(C)} r_{bc}^*$$

$$v_{2(A)} = r_{ab} v_{2(B)} r_{ab}^*$$

$$v_{2(A)} = r_{ab} r_{bc} v_{2(C)} r^*_{bc} r^*_{ab}$$

$$v_{2(A)} = r_{ab} r_{bc} (l_c i) r^*_{bc} r^*_{ab}$$

We can now express $\vec{p}_{end}$ in A

$$\begin{aligned} p_{end(A)} =& v_{1(A)} + v_{2(A)} && (16.9) \\ =& r_{ab}(l_b i + r_{bc}(l_c i) r^*_{bc}) r^*_{ab} && (16.10) \end{aligned}$$

16.14 External resources

- https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
- https://en.wikipedia.org/wiki/Quaternion
- https://www.youtube.com/watch?v=d4EgbgTm0Bg&t=1s&pp=ygUTM2Jsa3VlMWJyb3duIHF1YXRlcg%3D%3D

Chapter 17

Quaternion Implementation

In this chapter we will create an implementation of a quaternion, as a Python class.

Classes in Python are useful when you need to group data and functions together into a logical grouping. Functions inside classes are called methods.

```
import numpy
import math
import sympy
```

Let's create a new class and call it "Quaternion".

```
class Quaternion(object):
    pass
```

Python's fundamental data type is an object. All data in Python is a sub-class of the object class, so when we create a new class, we want to specify what class we are inheriting from. Hence the `(object)` notation

We use `pass` on the second line to indicate that we are creating an empty class to start with.

```
q = Quaternion()
```

Great, the class gets instantiated, but it doesn't do anything yet. Now lets add some functionality.

Remember from the last chapter that quaternions are defined by:

$$q = a + bi + cj + dk$$

where i, j, and k are orthogonal complex numbers and a, b, c, d are four coefficients supplied to define the quaternion. In Python, we want to *initialize* an *instance* of the `Quaternion` class. This is done by defining an `__init__()` function.

`__init__()` is a special function used by Python for the purpose of creating an instance of a class. Think of a class as the design specification for a container to hold custom information (and functions), where an instance of a class actually holds custom data.

For example,

```
class MyClass(object):
    def __init__(self,value):
        self.value = value

my_instance1 = MyClass('a string')
my_instance2 = MyClass(3.4)

print(my_instance1.value)
print(my_instance2.value)
```

```
a string
3.4
```

To initialize `Quaternion`, we define the `__init__()` function with five variables. The first variable is a special variable called `self`. This variable is the instance *itself*, hence the name. We can use `self` to allow us to access the instances data in order to set and retrieve it. The four other variables are the four coefficients that define a quaternion. When we initialize the Quaternion class with four numbers, they need to get saved. Therefore, we need to put them inside `self`, as child variables.

```
class Quaternion(object):

    def __init__(self,a,b,c,d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
```

Now lets instantiate it again and see what happens

```
q = Quaternion(1,2,3,4)
q
```

```
<__main__.Quaternion at 0x73dc41d75ed0>
```

It ran without error, but the result it returns doesn't tell us much. Did we successfully save a, b, c, and d?

```
q.d
```

```
4
```

Yes. But we will have to provide some helper functions later in order to make the Quaternion class a bit more legible.

17.1 Overloading Python Operators

As discussed in the last chapter, several kinds of Quaternion operations are permitted, including multiplication. Python's operator for multiplication, the `*` symbol, can be redefined by a custom class to work differently. The way this works is by *operator overloading*. If we define a custom function in our new class, we can perform operations on custom classes using normal math operators! So, for our first example, let's define our class so that it can handle multiplication natively with the `*` operator. The custom function we need to redefine is the `__mul__` function. It takes two variables, `self` –

the variable that represents the instance *itself* – and the other operand involved in the multiplication operation, which we will call `other` in our example

Say you have the `a * b` operation in Python. In this case, the `*` operator calls the `__mul__` function defined by `a`, if it exists, supplying it with `(a,b)` as the two inputs to that function.

Lets do a quick, simple example before coming back to our Quaternion class

```python
class MyClass(object):
    def __init__(self,value):
        self.value = value
    def __mul__(self,other):
        return self.value*other

myinstance = MyClass(3)
myinstance*5
```

```
15
```

So, now that we see that we can multiply two things together with the `__mul__` function, lets assume that we are trying to multiply two quaternions together. Looking at the previous chapter, we see that when two quaternions $q_1 = a + bi + cj + dk$ and $q_2 = e + fi + gj + hk$ are multiplied, the result, when expanded and simplified, is

$$q_1q_2 = (ae - bf - cg - dh) + (af + be + ch - dg)i + (ag - bh + ce + df)j + (ah + bg - cf + de)k$$

Implemented in our class, we need to

1. extract the coefficients, $a - h$ from `self` and `other`
2. perform the element-wise multiplication for the four new coefficients
3. create and return a new quaternion result.

```python
class Quaternion(object):

    def __init__(self,a,b,c,d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def __mul__(self,other):
        a = self.a
        b = self.b
        c = self.c
        d = self.d
        e = other.a
        f = other.b
        g = other.c
        h = other.d
        a2 = a*e - b*f - c*g - d*h
        b2 = a*f + b*e #...finish this expression
        c2 = a*g #+ ...finish this expression
        d2 = a*h #+ ...finish this expression
        result = Quaternion(a2,b2,c2,d2)
        return result
```

Important: Please note that the example above is not complete

```
q1 = Quaternion(1,2,3,4)
q2 = Quaternion(4,3,2,1)
result = q1*q2
result
```

```
<__main__.Quaternion at 0x73dc5418e010>
```

Because the `__mul__()` function is not yet complete, this result will produce the wrong answer. Can you complete the method above yourself?

```
q1*q2
```

```
<__main__.Quaternion at 0x73dc41d89c90>
```

17.2 Other functionality

Other examples of functionality include the conjugate operation, where for a quaternion $q = a + bi + cj + dk$, its conjugate is

$$q^* = a - bi - cj - dk$$

In Python, you can define a method that returns a new quaternion class according to that equation:

```
class Quaternion(object):

    def __init__(self,a,b,c,d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    #...

    def conjugate(self):
        return Quaternion(self.a,-self.b,-self.c,-self.d)
```

Now lets test it.

```
q1 = Quaternion(1,2,3,4)
result = q1.conjugate()
print(result.a,result.b,result.c, result.d)
```

```
1 -2 -3 -4
```

17.3 Length

The norm, or length of a quaternion q is defined in the previous chapter as $|q|$, where

$$|q| = \sqrt{qq^*} = \sqrt{a^2 + b^2 + c^2 + d^2}$$

We can calculate this length quite easily as well

```
class Quaternion(object):

    def __init__(self,a,b,c,d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    #...

    def len(self):
        a = self.a
        b = self.b
        c = self.c
        d = self.d
        l = (a**2+b**2+c**2+d**2)**.5
        return l
```

With the `len()` function and `conjugate()` functions complete, it is trivial to create an `inverse()` function for quaternions. Can you do this?

17.4 Printing

by implementing the `__str__(self)` and the `__repr__(self)`, we can output more useful information about our custom class when it is returned in the Python command line, or when we `print()` it.

```
class Quaternion(object):

    def __init__(self,a,b,c,d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    #...

    def __str__(self):
        s = 'Quaternion({0},{1} i,{2} j,{3} k)'.format(self.a,self.b,self.c,self.d)
        return s

    def __repr__(self):
        return str(self)
```

Now let's test our printing and outputting:

```
q1 = Quaternion(1,2,3,4)
print(q1)
q1
```

```
Quaternion(1,2 i,3 j,4 k)
```

```
Quaternion(1,2 i,3 j,4 k)
```

17.5 Other tasks

We have only begun the `Quaternion` class. Other important things to do could include:

- implementing an inverse function
- normalizing
- testing whether a given quaternion is a "pure" quaternion or not.
- creating a quaternion from a specified axis, angle pair
- computing the axis angle pair corresponding to a unit quaternion's coefficients
- rotating a pure quaternion by a given unit quternion.
- converting quaternions to vectors and back.

Furthermore, given the variety of math packages, such as `sympy` and `math`, one could envision a quaternion class that can perform symbolic or numeric operations on quaternions based on the received input. While this is outside the scope of this introduction, I hope you can see how quickly classes such as `Quaternion` can start to help you perform math and compute kinematics.

Chapter 18

Expressing Vectors in other Reference Frames

18.1 Introduction

You cannot combine the coefficients of two vectors together if those coefficients are attached to different sets of basis vectors.

$$u_1\hat{a}_x + u_2\hat{b}_y \neq (u_1 + u_2)\hat{a}_x \neq (u_1 + u_2)\hat{b}_y$$

They must instead, be *expressed* in terms of one or the other basis vectors before you can combine them. Vectors can be expressed in other reference frames using rotations, which define the length and angle-preserving transformation that describes one set of basis vectors in a different frame's basis vectors.

18.2 Sequence

1. First, define your frames
    1. Define at least one Fixed (Newtonian) frame
2. Create Rotations between frames
3. Use frames' basis vectors to compose vectors
4. Operate on them.
    - `dot()` , `cross()` , `+` , `-` , `*`

18.3 Expressing vectors in other frames

Let's say that

$$p = u_1\hat{b}_x + u_2\hat{b}_y + u_3\hat{b}_z$$

and that

$$^{a}R^{b} = \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix}$$

Then

$$p' = ^{a}R^{b} p \tag{18.1}$$

$$= \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \tag{18.2}$$

$$= (r_1 u_1 + r_2 u_2 + r_3 u_3) a_x \tag{18.3}$$

$$+ (r_4 u_1 + r_5 u_2 + r_6 u_3) a_y \tag{18.4}$$

$$+ (r_7 u_1 + r_8 u_2 + r_9 u_3) a_z \tag{18.5}$$

18.4 multiple stages

$$p_{1(a)} = u_1 \hat{a}_x + u_2 \hat{a}_y + u_3 \hat{a}_z$$

$$p_{2(b)} = u_4 \hat{b}_x + u_5 \hat{b}_y + u_6 \hat{b}_z$$

$$p_{3(c)} = u_7 \hat{c}_x + u_8 \hat{c}_y + u_9 \hat{c}_z$$

$$p_1 + p_{2(b)} + p_{3(c)} = p_{1(a)} + {}^{a}R^{b} p_{2(b)} + {}^{a}R^{bb}R^{c} p_{3(c)}$$

Chapter 19

Vector Math

19.1 Remind me: What are vectors?

Definition: A vector describes a *magnitude* and a *direction* within the N-dimensional space it is defined.

Vectors are ubiquitous in engineering; you have probably worked with them before. They can be used in a number of contexts in mechanism design and robotics, from describing the kinematics of a structure to defining the magnitude and direction of a force or axis of rotation of a torque.

In undergraduate textbooks, you may have encountered 2D vector problems. 2D vectors, while easy to understand and work with, often hide some of the complexity that comes with three-dimensional spaces. Problems in dynamics and physics that deal with "handedness", or the specific sequence or ordering of vector operations, often require 3D vectors (or higher dimensional spaces) in order to more easily capture the math contained within. In this section we will be defining and working with 3D vectors and their operations.

19.2 Vectors and Basis Vectors

Vectors are a separate concept from basis vectors. Any number of frames, and their affiliated basis vectors, can be used to describe the same vector. Because a vector simply describes a magnitude and a direction within a given N-dimensional space, any number of basis vectors can be used to describe them. Thus, the expression of a vector depends on the set of basis vectors you use to describe it with.

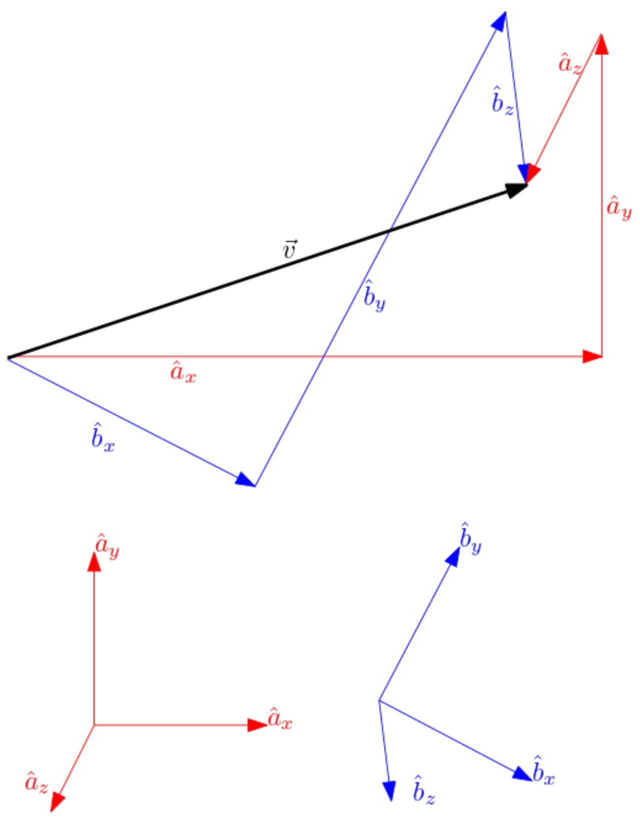


Figure 19.1: Different basis vectors can be used to describe the same vector.

Consider two reference frames A and B along with two different sets of basis vectors, $(\hat{a}_x, \hat{a}_y, \hat{a}_z)$, and $(\hat{b}_x, \hat{b}_y, \hat{b}_z)$, respectively. Both sets of basis vectors span a three-dimensional space. But the same vector $\vec{v}$ will be described using two completely different equations, depending on which set you use to describe it with. In the case of the A frame, c_1, c_2, and c_3 represent the signed length of $\vec{v}$ along $\hat{a}_x$, $\hat{a}_y$, and $\hat{a}_z$, respectively, while d_1, d_2, and d_3 represent a different set of lengths. The choice of *representation* impacts the *description*.

$$\vec{v} = c_1\hat{a}_x + c_2\hat{a}_y + c_3\hat{a}_z$$

$$\vec{v} = d_1\hat{b}_x + d_2\hat{b}_y + d_3\hat{b}_z$$

Question: Are these two descriptions related? Yes, of course, but they just *look* different.

What is nice about the equations above? Both equations describe the same vector, yet each also supplies the reference frame used to describe it with. This is important, because as you will see below, operations are usually defined for vectors within the same frame.

Vectors can also be described using matrix notation:

$$v = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

.

It must be noted, however, that this description is incomplete, because it omits the reference frame the scalar values are defined in. This issue can be commonly found in textbooks and research papers. Care must then be taken to ensure such meaning is not lost in this representation.

Note: Always include the reference frame of vectors that are written in matrix notation.

19.3 Representations vs Information

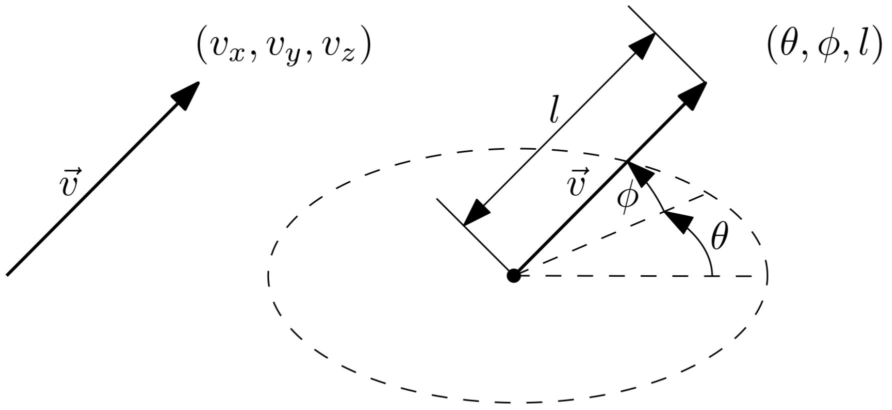


Figure 19.2: The same vector using Cartisian vs. Spherical representation

Vectors do not need to use Cartesian basis vectors in their description. Any number of coordinate systems can be used. The example above shows the same vector $\vec{v}$ represented with both a cartesian and spherical coordinate system.

Note: Regardless of the coordinate system used, a vector in three dimensions contains three independent numbers.

See the note above, because this is important. Regardless of the system used to describe it, a vector contains the same amount of information regardless. You may use three cartesian magnitudes (v_x, v_y, v_z), or spherical coordinates (θ, ϕ, l). Either way, a three-dimensional vector holds **three** numbers.

19.4 Vector Operations

Just like scalar math, you can write mathematical expressions and equations with vectors. There are new kinds of operations that you can perform on and between vectors that take into account that these are multi-dimensional entities. Furthermore, some operations produce lower-dimensional results, while other operations result in answers in higher dimensions. Unlike scalar operations, the order of operations becomes much more important with vectors.

Performing a dot or cross product with vectors composed of different basis vectors does not work unless a relationship can be defined between them, expressing one set of basis vectors in terms of the other. This can be addressed by defining a relationship between frames through *rotations*.

19.4.1 Scalar Multiplication

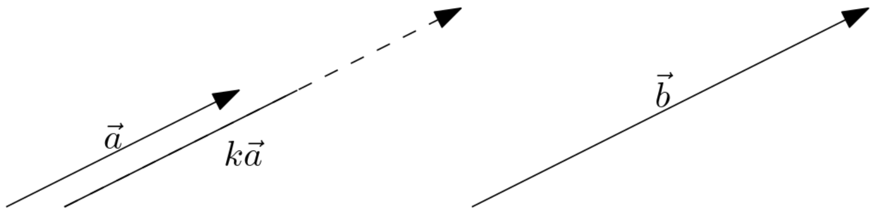


Figure 19.3: Scalar Multiplication

$$k\vec{a} = \vec{b}$$

Any vector multiplied by a scalar will result in a vector with the same direction but different magnitude. The order of operands does not matter, or is *commutative*. In other words, $k\vec{b} = \vec{b}k$

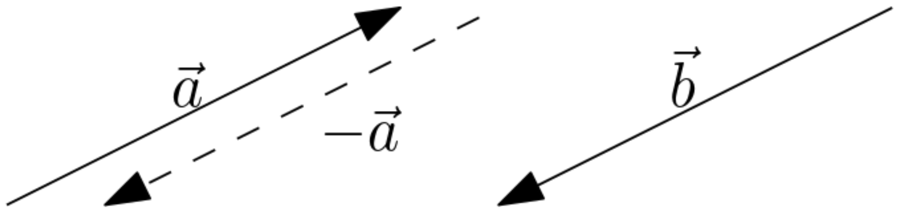


Figure 19.4: Scalar multiplication is distributive

$$\vec{b} = (-1) * \vec{a} = -\vec{a}$$

You can negate a vector, because a negative vector is still simply a vector multiplied by a negative scalar value.

Note: There is some ambiguity in the concept of a negative vector, because a negative vector can be simultaneously thought of either **1)** as a vector of the *same* direction with the *opposite* magnitude, or **2)** the *same* magnitude in the *opposite* direction. Both are true, and is the source of some mathematical ambiguity later as well.

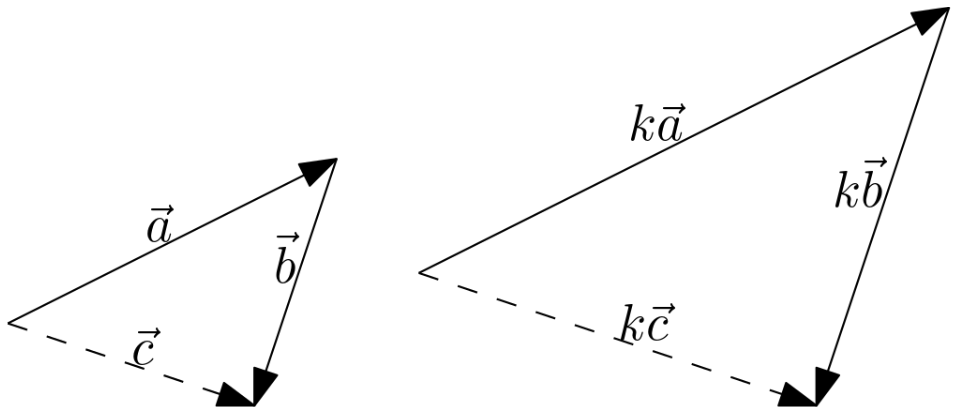


Figure 19.5: caption

Scalar multiplication can be distributed as well, as shown above. In other words,

$$k\vec{c} = k(\vec{a} + \vec{b}) = k\vec{a} + k\vec{b} = k\vec{c}$$

19.4.2 Scalar Division

Scalar division can always be converted to a multiplication problem and is thus a trivial extension to multiplication. For example,

$$\frac{\vec{v}}{k} = \underbrace{\left(\frac{1}{k}\right)}_{c}\vec{v} = c\vec{v}$$

19.4.3 Addition

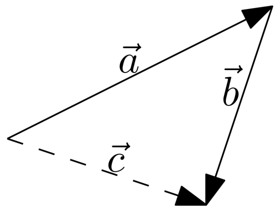


Figure 19.6: caption

Two vectors can be added together, and the result is another vector of the same dimension, or $\vec{c} = \vec{a} + \vec{b}$.

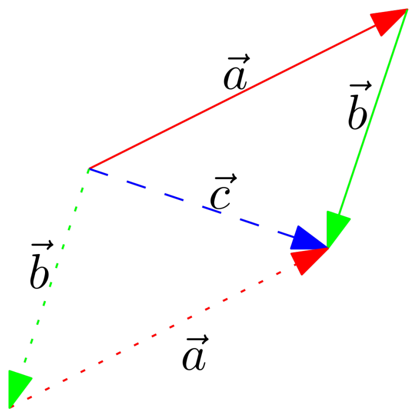


Figure 19.7: Vector addition is *commutative*

The order of operations also does not matter – in this way, vector addition is *commutative*. In other words,

$$\vec{c} = \vec{a} + \vec{b} = \vec{b} + \vec{a}$$

This is shown in the following figure. Starting from the beginning of $\vec{c}$, you can travel along $\vec{a}$ to $\vec{b}$ in order to arrive at the tail of $\vec{c}$ (solid lines), or travel in the opposite sequence, from $\vec{b}$ to $\vec{a}$ (dotted lines). Both arrive at the same spot.

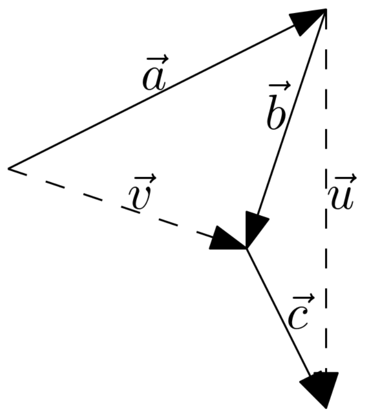


Figure 19.8: Vector addition is *associative*

Vector addition is also *associative*, in that the grouping or sequencing of addition operations does not matter. In other words, using the figure above,

$$\vec{a} + \underbrace{(\vec{b} + \vec{c})}_{\vec{u}} = \underbrace{(\vec{a} + \vec{b})}_{\vec{v}} + \vec{c}$$

19.4.4 Vector Subtraction

Vector subtraction is also a simple extension to vector addition, because you can always recast a vector difference as the summation of a negative vector; summation and scalar multiplication – and their associated properties – have both been described above. For example,

$$\vec{a} - \vec{b} = \vec{a} + (-1)\vec{b}$$

19.4.5 Dot product

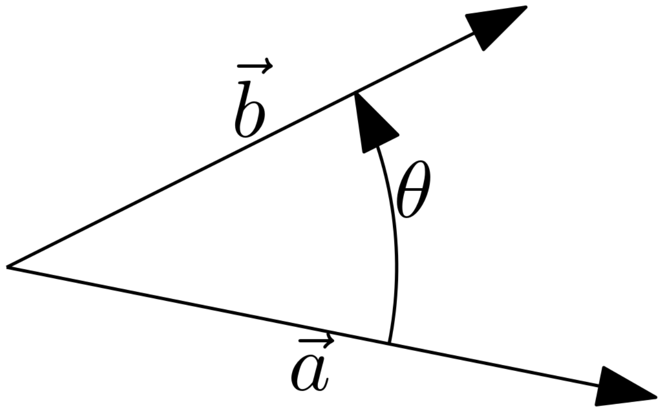


Figure 19.9: The vectors and angles associated with the dot product.

The dot product is used to measure how well one vector's direction is aligned with another's. This uses the concept of two vectors' *inner angle*, which is itself defined as the smallest angle that can be measured between two vectors within the plane the two vectors share. The dot product is thus defined in terms of this inner angle θ, or

$$\vec{a} \cdot \vec{b} \triangleq |\vec{a}| \left|\vec{b}\right| \cos\theta \tag{19.1}$$

Because it produces a scalar result, the dot product is *commutative*, or

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$$

The dot product can also be *distributed* to terms grouped within parentheses, or

$$\vec{a} \cdot \left(\vec{b} + \vec{c}\right) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}$$

This becomes especially useful when working with vectors represented by more than one basis vector or reference frame.

The definition above provides useful insights but is not easy to work with if representing vectors using Cartesian components. When you represent $\vec{a}$ and $\vec{b}$ with a common set of Cartesian basis vectors, you can expand and distribute the dot product to all of the basis vector pairs formed, as in

$$\vec{a}\cdot\vec{b}=(a_x\hat{n}_x+a_y\hat{n}_y+a_z\hat{n}_z)\cdot(b_x\hat{n}_x+b_y\hat{n}_y+b_z\hat{n}_z)$$

Expanding this expression you get

$$\begin{aligned}\vec{a}\cdot\vec{b}=&a_x\hat{n}_x\cdot b_x\hat{n}_x+a_x\hat{n}_x\cdot b_y\hat{n}_y+a_x\hat{n}_x\cdot b_z\hat{n}_z && (19.2)\\ &a_y\hat{n}_y\cdot b_x\hat{n}_x+a_y\hat{n}_y\cdot b_y\hat{n}_y+a_y\hat{n}_y\cdot b_z\hat{n}_z\\ &a_z\hat{n}_z\cdot b_x\hat{n}_x+a_z\hat{n}_z\cdot b_y\hat{n}_y+a_z\hat{n}_z\cdot b_z\hat{n}_z\\ & && (19.3)\end{aligned}$$

Grouping scalars, you get:

$$\begin{aligned}\vec{a}\cdot\vec{b}=&a_xb_x(\hat{n}_x\cdot\hat{n}_x)+a_xb_y(\hat{n}_x\cdot\hat{n}_y)+a_xb_z(\hat{n}_x\cdot\hat{n}_z) && (19.4)\\ &a_yb_x(\hat{n}_y\cdot\hat{n}_x)+a_yb_y(\hat{n}_y\cdot\hat{n}_y)+a_yb_z(\hat{n}_y\cdot\hat{n}_z)\\ &a_zb_x(\hat{n}_z\cdot\hat{n}_x)+a_zb_y(\hat{n}_z\cdot\hat{n}_y)+a_zb_z(\hat{n}_z\cdot\hat{n}_z)\\ & && (19.5)\end{aligned}$$

Here we see opportunities for simplification, using the definition of the dot product itself. Because $(\hat{n}_x,\hat{n}_y,\hat{n}_z)$ are *orthonormal* to each other, this means that their inner angle is $\frac{\pi}{2}$. Plugging this into the definition above, we get the following relations:

$$\begin{aligned}\hat{n}_x\cdot\hat{n}_y=0 && (19.6)\\ \hat{n}_y\cdot\hat{n}_z=0 && (19.7)\\ \hat{n}_z\cdot\hat{n}_x=0 && (19.8)\\ && (19.9)\end{aligned}$$

Furthermore, again using the definition of the dot product, any unit vector $\hat{v}$ dotted with itself is equal to 1, because its magnitude $|\hat{v}|$ is and the inner angle between it and itself is 0 and thus $\cos 0=1$. This means that

$$\begin{aligned}\hat{n}_x\cdot\hat{n}_x=1 && (19.10)\\ \hat{n}_y\cdot\hat{n}_y=1 && (19.11)\\ \hat{n}_z\cdot\hat{n}_z=1 && (19.12)\\ && (19.13)\end{aligned}$$

This simplifies our expression even further to:

$$\vec{a}\cdot\vec{b}=a_xb_x+a_yb_y+a_zb_z$$

19.4.6 Length of a Vector

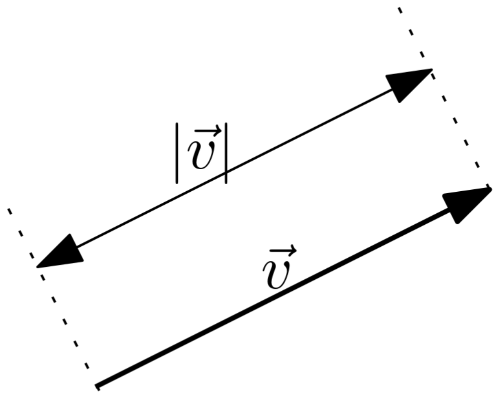


Figure 19.10: The length of a vector

We can use the definition of the dot product to find a vector's *length*. The length of a vector $\vec{v}$ is the scalar, absolute value of a vector's magnitude. It is typically written as $|\vec{v}|$. A vector's length is defined as:

$$|\vec{v}| \triangleq \sqrt{\vec{v} \cdot \vec{v}}$$

Note the difference between a vector's *magnitude* and *length*. Its length is always positive. Its magnitude depends on the relative direction of the vector.

19.5 Cross Product

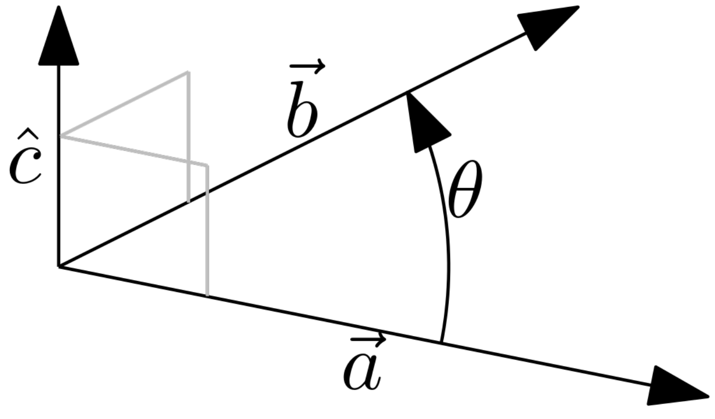


Figure 19.11: caption

The cross product may be defined as

$$\vec{a} \times \vec{b} \triangleq |\vec{a}| \left|\vec{b}\right| \sin\theta \hat{c}$$

where $\hat{c}$ is a unit vector normal to $\vec{v}$ and $\vec{w}$.

Note: There are two valid unit vectors normal to $\vec{a}$ and $\vec{b}$, but the cross product is valid for only one. $\hat{c}$ is determined using the right hand rule and the inner angle between $\vec{a}$ and $\vec{b}$. Starting from $\vec{a}$, sweep your right hand *through* the inner angle from $\vec{a}$ to $\vec{b}$. The normal indicated by the right hand rule indicates the direction of $\hat{c}$. This will be discussed later when we delve into angles

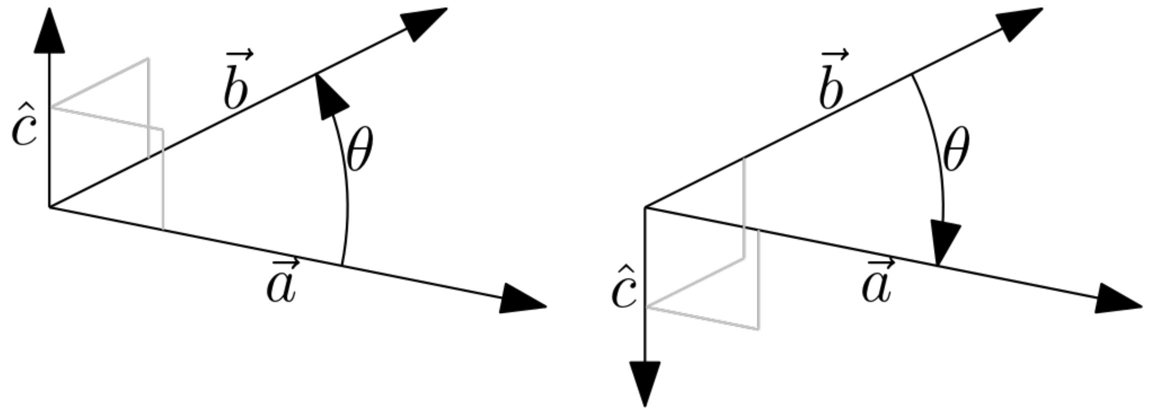


Figure 19.12: caption

Because the sequence of vectors in the cross product determines the direction of the resulting $\hat{c}$ normal, its order matters. Though the cross product *is not* commutative $\vec{a} \times \vec{b} \neq \vec{b} \times \vec{a}$, the cross product *is* anti-commutative, or

$$\vec{a} \times \vec{b} = -(\vec{b} \times \vec{a})$$

The cross product operator can also be distributed, or

$$\vec{a} \times \left(\vec{b} + \vec{c}\right) = \vec{a} \times \vec{b} + \vec{a} \times \vec{c}$$

Like the dot product, this becomes useful when resolving vectors into their basis vector components.

Like the dot product, the definition of the cross product also provides useful insights but is not easy to work with if representing vectors using Cartesian components. When you represent $\vec{a}$ and $\vec{b}$ with a common set of Cartesian basis vectors, you can expand and distribute the dot product to all of the basis vector pairs formed, as in

$$\vec{a} \times \vec{b} = (a_x\hat{n}_x + a_y\hat{n}_y + a_z\hat{n}_z) \times (b_x\hat{n}_x + b_y\hat{n}_y + b_z\hat{n}_z)$$

Expanding this expression you get

$$\begin{aligned}\vec{a}\times\vec{b} =& a_x\hat{n}_x\times b_x\hat{n}_x + a_x\hat{n}_x\times b_y\hat{n}_y + a_x\hat{n}_x\times b_z\hat{n}_z \quad (19.14)\\ & a_y\hat{n}_y\times b_x\hat{n}_x + a_y\hat{n}_y\times b_y\hat{n}_y + a_y\hat{n}_y\times b_z\hat{n}_z\\ & a_z\hat{n}_z\times b_x\hat{n}_x + a_z\hat{n}_z\times b_y\hat{n}_y + a_z\hat{n}_z\times b_z\hat{n}_z\\ & \quad (19.15)\end{aligned}$$

Grouping scalars, you get:

$$\begin{aligned}\vec{u}\times\vec{v} =& a_xb_x(\hat{n}_x\times\hat{n}_x) + a_xb_y(\hat{n}_x\times\hat{n}_y) + a_xb_z(\hat{n}_x\times\hat{n}_z) \quad (19.16)\\ & a_yb_x(\hat{n}_y\times\hat{n}_x) + a_yb_y(\hat{n}_y\times\hat{n}_y) + a_yb_z(\hat{n}_y\times\hat{n}_z)\\ & a_zb_x(\hat{n}_z\times\hat{n}_x) + a_zb_y(\hat{n}_z\times\hat{n}_y) + a_zb_z(\hat{n}_z\times\hat{n}_z)\\ & \quad (19.17)\end{aligned}$$

Here, like with the dot product, we see opportunities for simplification, using the definition of the cross product itself. Because $(\hat{n}_x, \hat{n}_y, \hat{n}_z)$ are *orthonormal* to each other, this means that their inner angle is $\frac{\pi}{2}$. Plugging this into the definition for cross product above, we get the following relations:

$$\hat{n}_x\cdot\hat{n}_y = \hat{n}_z \quad (19.18)$$
$$\hat{n}_y\cdot\hat{n}_z = \hat{n}_x \quad (19.19)$$
$$\hat{n}_z\cdot\hat{n}_x = \hat{n}_y \quad (19.20)$$
$$(19.21)$$

Furthermore, again using the definition of the cross product, any unit vector $\hat{v}$ dotted with itself is equal to 0, because the inner angle between it and itself is 0 and thus sin $0 = 0$. This means that

$$\hat{n}_x\cdot\hat{n}_x = 0 \quad (19.22)$$
$$\hat{n}_y\cdot\hat{n}_y = 0 \quad (19.23)$$
$$\hat{n}_z\cdot\hat{n}_z = 0 \quad (19.24)$$
$$(19.25)$$

This further simplifies the expression above to

$$\vec{u}\times\vec{v} = a_xb_y\hat{n}_z - a_xb_z\hat{n}_y - a_yb_x\hat{n}_z + a_yb_z\hat{n}_x + a_zb_x\hat{n}_y - a_zb_y\hat{n}_x$$

19.6 Vector Math No-No’s

Please avoid the following expressions, as – per this book’s definitions – they are either undefined or just plain wrong.

- Don’t multiply a vector by another vector (with the $*$ symbol)

  - $\vec{a} * \vec{b} = ?$
- Don't sum or equate vectors and scalars together
  - $\vec{a} \neq 3$
  - $\vec{a} - 4 = ?$
  - $\vec{a} \neq 0$

  Instead, use the length of vectors, as in $|\vec{a}| = 0$

- Don't use a cross product on a scalar:
  - $\vec{a} \times 4 = ?$
  - $\vec{a} \times (\vec{b} \cdot \vec{c}) = ?$
- Don't mix matrix and vector notation.
  - $3\hat{n}_x + 4\hat{n}_y \neq \begin{bmatrix} 3 \\ 4 \end{bmatrix}$

  As discussed, the change in notation includes a difference in meaning. Changing representations should be done on separate lines, with explicit callouts for the basis vectors used in matrix notation.

19.7 Other Properties

These essential definitions and properties extend to many other helpful expressions. This chapter will not explicitly define more of these properties, but some of the problems at the end of the chapter will expose other useful properties

There are a number of more applicable books on the subject. The references are listed at the end.

19.8 Implementation

Vectors can be represented as a Python class, and can leverage Python's ability to overload mathematical operators with other functionality. In this way, common operators such as $+$,$-$,$*$ and $/$ take on their own meaning when used in expressions with scalars, vectors, dyads, or dyadics. One can also use packages like numpy to represent vectors, though **you must take care not to linearly combine two vectors defined in different reference frames without first expressing one in terms of the others' basis vectors.**

Chapter 20

Mechanisms, Linkages, and Degrees of Freedom

20.1 Introduction

In this chapter we will review the definitions of linkages, mechanisms, and discuss some basic joint types and how that contributes to the degrees of freedom of a system.

20.2 Links

Links are rigid bodies that can be connected to other links through joints

20.3 Joints

Joints are connections between links that permit motion about or along specified directions, or degrees of freedom.

Conversely, joints can be thought of as connections between links that *restrict* motion about or along degrees of freedom. For example, consider a pin joint. It permits rotation about one axis, and *prevents* rotation about all others.

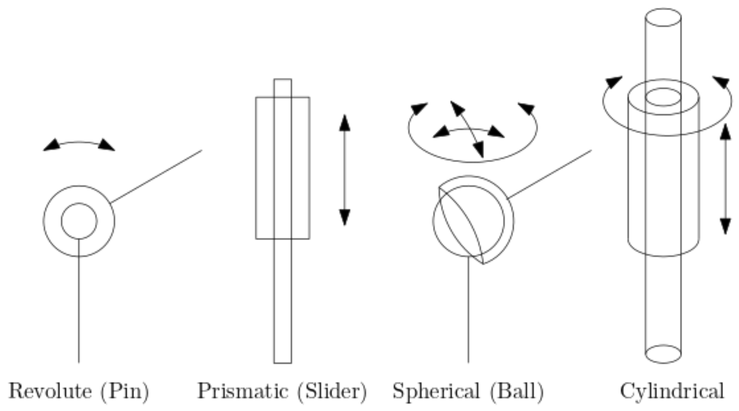


Figure 20.1: caption

Common types of joints:

- Pin / Hinge / Revolute Joint
- Slider / Linear / Prismatic Joint
- Ball / Spherical Joint
- Cylindrical Joint
- Screw Joint
- ...

20.4 Degree of Freedom

A joint's degrees of freedom counts the number of independent ways a link can move about and/or along another link.

A mechanism's degrees of freedom describes the number of independent ways a linkage or machine can move relative to a base link, fixed frame, or other selected link with regard to its system of joints,

A mechanism's degrees of freedom are determined by the complex geometric relationships between a mechanism's joints, constraints, and joint limits, as well as any external constraints enforced via contact with the environment.

20.5 Planar Mechanisms

Planar mechanisms are mechanisms whose motion can be completely described in two dimensions. A planar mechanisms joints are all parallel to each other and share a common vertex at infinity.

20.6 Spherical Mechanisms

Spherical mechanisms are mechanisms in which the joint axes all share a common vertex in 3D space.

20.7 Spatial Mechanisms

A spatial mechanism is any mechanism that moves in 3D space. Planar mechanisms and Spherical mechanisms are two subsets of the more general class of spatial mechanisms.

20.8 Serial Mechanisms

A serial mechanism is a mechanism whose links are arranged *one after the other*, in a row. Serial mechanisms can be planar, spherical, or, more generally, spatial (3D).

20.9 Parallel Mechanisms

A parallel mechanism is a mechanism whose links are arranged alongside one another, and join to form a closed loop. Parallel mechanisms can also be planar, spherical, or spatial (3D).

20.10 Structure

A structure can be thought of as any mechanism whose degrees of freedom is less than or equal to its constraints.

20.11 Linkage Topologies

Linkages and origami-inspired mechanisms are a special class of mechanisms, because can all be described by just one type of joint – the pin joint – and a link. This makes the analysis and synthesis of linkages relatively easy, because you can use graph theory to represent the connection between links and joints

Chapter 21

Vector Expressions & Constraint Equations

21.1 Introduction

In this chapter we discuss how to use basic vector operations to create vector-based expressions, and how to form these expressions into systems of constraint equations.

21.2 Vector Expressions

Expressions using vectors can be composed in order to add geometric meaning for use in mechanism analysis.
Vector expressions help us measure and express the relationships visible in mechanisms, origami-inspired folding, and structures. They can be used to measure the distance and angle between points, lines, and planes. They can be used to enforce constraints. They can also be used in dynamics code to interpret the conditions of your robot.

21.2.1 Expressions for Measuring and Specifying Angles

Much of the text based on angles is based on the expressions for dot product and cross product, found previously.

Specifying the angle between vectors can be tricky because ambiguities exist in the definition of angles between vectors.

Often, we care about measuring the *smallest* angle between two vectors, without regard to vector order. We can use the dot product for this.

$$\theta = \cos^{-1}\left(\frac{\vec{v}_1 \cdot \vec{v}_2}{|\vec{v}_1||\vec{v}_2|}\right) \forall\theta \in \{0 \leq \theta \leq \pi\}$$

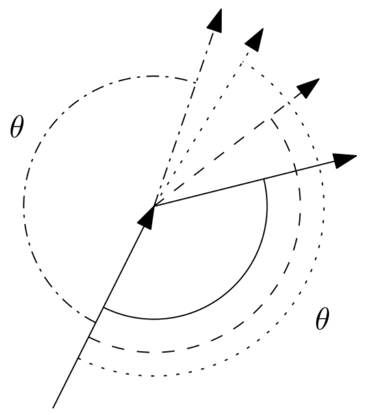


Figure 21.1: The inner angle measured by θ in a plane changes definition as the relative angle of the vectors pass through co-linearity

This is because the inner angle – along with the formula above – does not depend on the sequence of the vectors supplied, and because it always returns the minimum angle. For mechanism design and analysis, however, the internal angle has a problem: As a physical hinge rotates, the definition of the measured angle changes. This can be seen in the figure above. As two vectors pass co-linearity, the definition of θ changes to adopt whatever the minimum angle is. For mechanisms that continuously rotate past $\theta = \pi$, this can make analysis challenging.

Can we use the cross product to find an angle that doesn't change? As you recall, the cross product *does* depend on vector order:

$$\vec{v}_1 \times \vec{v}_2 \triangleq |\vec{v}_1|\,|\vec{v}_2| \sin\theta\hat{u}$$

It turns out that for Python's standard math library, the unit vector $\hat{u}$ is consistent only for the range $0 < \theta < \pi$. For $-\pi < \theta < 0$, $-\hat{u}$ is assumed. Thus, the $\hat{u}$ vector specified by the cross product operation is *consistent with the inner angle as it is defined by the dot product*. (For the values $\theta = \{0, \pi\}$, $\hat{u} = \vec{0}$).

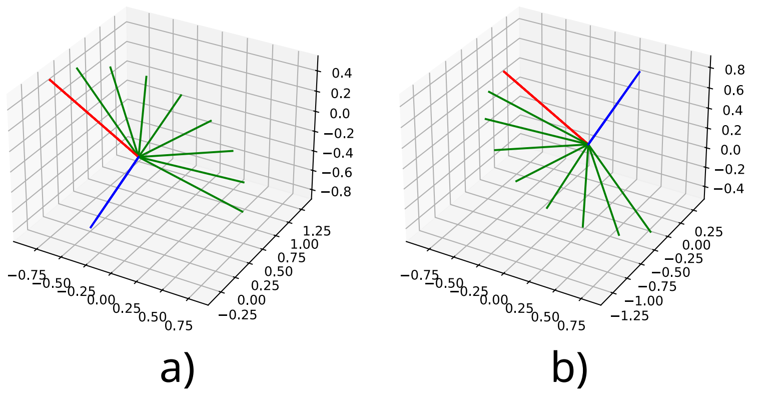


Figure 21.2: The unit vectors resulting from a cross-product operation in Python across two separate domains. a) \$0 < ▯< ▯\$ b) $-\pi < \theta < 0$.

Referring to this figure, for example, for two vectors $\vec{v}_{red}$ and $\vec{v}_{green}$ lying in an arbitrary plane, the cross product $\vec{v}_{red} \times \vec{v}_{green} = \vec{v}_{blue}$. The orientation of $\vec{v}_{blue}$ depends on which half of the plane the inner angle between $\vec{v}_{red}$ and $\vec{v}_{green}$ exists, (by applying the right hand rule in the direction from $\vec{v}_{red}$ to $\vec{v}_{green}$ through the inner angle, the orientation switches).

This definition helps math libraries such as that found in Python to resolve the ambiguity in definition for the ordered angle between one vector and another. This measurement is not unique, because it depends both on the from-to sequence of vectors, as well as an external reference orientation, as supplied by a normal unit vector. The figure below highlights the issue.

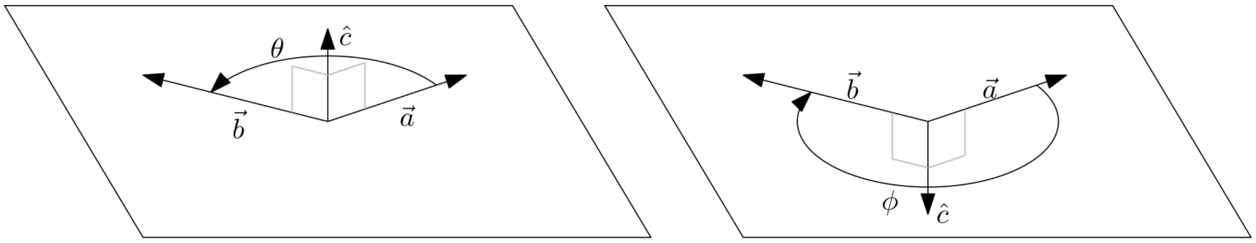


Figure 21.3: The absolute angle between two vectors depends on the direction of the unit vector used as a reference.

Though the cross-product (or right hand rule) can be used to identify a vector perpendicular to two arbitrary vectors, there are two equal and opposite vectors along which the right-hand rule can be applied. Consider two vectors in the x-y plane. Two unit vectors, (one in the direction of the +z axis and the other in the direction of the -z axis) both serve as the normal to the x-y plane. Finding the angle *from $\vec{a}$ to $\vec{b}$* using each of these unit vectors produces two different answers: θ, or ϕ, where $\theta + \phi = 2\pi$. Depending on the orientation of the normal vector selected and the order of vectors you swept your hand *from* and *to*, you are measuring two different angles. Therefore, the only way to measure a consistent angle between two vectors is to supply a third (normal unit) vector which can serve as a static reference.

21.2.2 Unwrapping Angles and Trignometric Domains

In real mechanical systems, it is often important to count the total number of revolutions of a joint between one link and another. However, in the math libraries supplied for many programming languages, trigonometric functions repeat over a domain of 2π. Thus with those functions alone, it is impossible to determine, simply from vectors, exactly how many times a body has rotated about an axis.

Looking at the following table, we see that the range of $\sin^{-1}$ and $\cos^{-1}$ do not span $2\pi = 360°$. This means that the dot product and cross product alone are not sufficient for finding the angle between vectors.

function	range
`asin(y)`	$-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$
`acos(x)`	$0 \leq \theta \leq \pi$

function	range
`atan(y/x)`	$-\frac{\pi}{2} < \theta < \frac{\pi}{2}$
`atan2(y,x)`	$-\pi < \theta \leq \pi$

However, `atan2()` has a valid range from $-\pi < \theta \leq \pi$, and is thus quite useful for resolving full 360 degree rotations. However, when θ goes past that range in a true mechanical system, it is still necessary to track the number of transitions in order to get the absolute travel. For a practical example, see section X.Y to learn more about *unwrapping* angular values. A practical application of unwrapping a joint angle is provided in a later section of the book.

21.2.3 Distance between two points

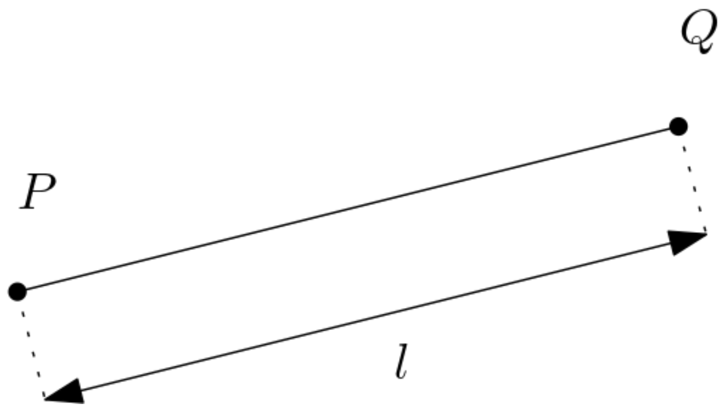


Figure 21.4: Distance between two points

The distance between two points can be reformulated as a vector expression. Consider two points P and Q. These points may be considered vectors $\vec{p}$ and $\vec{q}$, respectively, representing the vectors pointing from a common origin to P and Q.

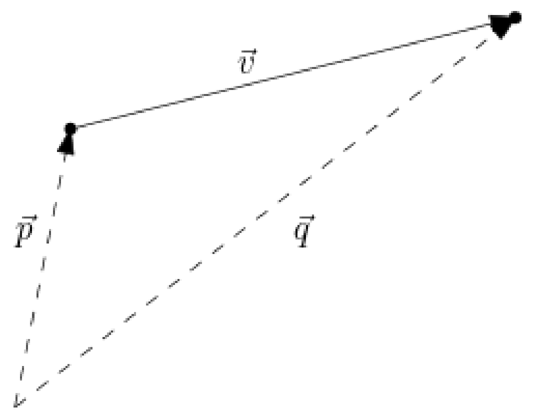


Figure 21.5: Distance between two points, formulated as a vector problem.

Written in basis vector forms, they can be expressed as

$$\vec{p} = p_x\hat{a}_x + p_y\hat{a}_y + p_z\hat{a}_z$$

$$\vec{q} = q_x\hat{a}_x + q_y\hat{a}_y + q_z\hat{a}_z$$

Where

$$\vec{v} = \vec{q} - \vec{p}$$

The simplest form of the expression for the distance between two points is thus

$$|\vec{v}| = l$$

It can be observed that this is a scalar equation, but how do we write it using the coefficients of the basis vectors? First, we need to expand

$$|\vec{v}| = |\vec{q} - \vec{p}| = l$$

Collecting terms on one side is often important when writing vector expressions as constraint equations, as we will see later in this section.

$$0 = |\vec{q} - \vec{p}| - l$$

By expanding the expressions, we see that

$$0 = (\vec{q} - \vec{p}) \cdot (\vec{q} - \vec{p}) - l^2$$

$$0 = (q_x - p_x)^2 + (q_y - p_y)^2 + (q_z - p_z)^2 - l^2$$

21.3 v,w perpendicular

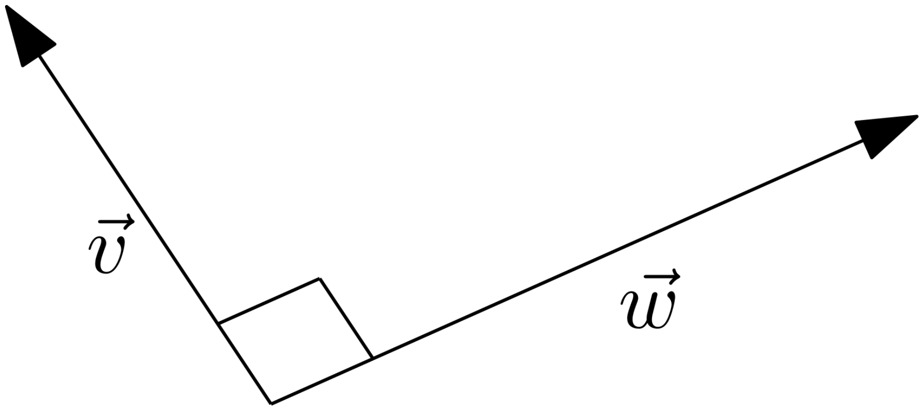


Figure 21.6: caption

Here the intuition is that perpendicular lines have an angle of θ between them. This means that the expression for dot product or cross product can be used, as both use θ. The dot product produces a scalar result, however, while the result of a cross product is a vector, meaning that all three elements of the vector must equal zero; hence we pick the dot product expression. Using , we substitute $\theta = \frac{\pi}{2}$ into the expression, resulting in the simplified

$$\vec{v} \cdot \vec{w} = 0.$$

When broken out into basis vector coefficients, the expression expands to

$$v_x w_x + v_y w_y + v_z w_z = 0$$

21.4 v,w parallel

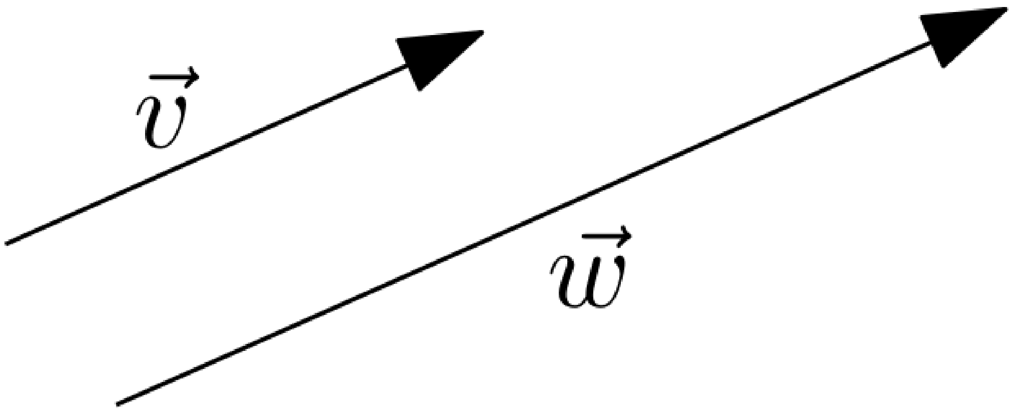


Figure 21.7: caption

The intuition here is the same as above, except that we substitute the expression $\theta = 0$ into the definition of the dot product. This results in

$$\vec{v} \cdot \vec{w} = |v||w|$$

When we expand this form into the coefficients of the basis vectors, the expression becomes:

$$v_x w_x + v_y w_y + v_z w_z = \sqrt{(v_x^2 + v_y^2 + v_z^2)(w_x^2 + w_y^2 + w_z^2)}$$

21.5 point P a distance l from Q on line QR

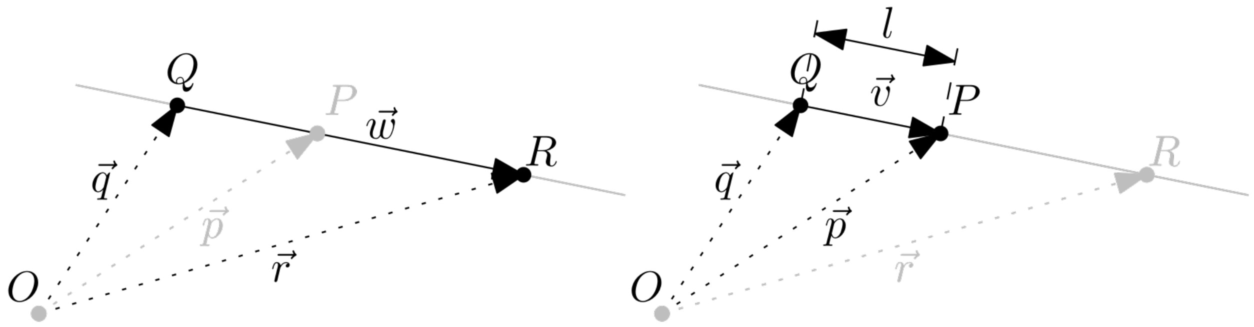


Figure 21.8: caption

$$Q = \vec{q}$$

$$R = \vec{r}$$

$$P = \vec{p} = \vec{q} + \vec{v}$$

using the definition of a dot product,

$$\vec{v} \cdot \vec{w} = |\vec{v}||\vec{w}| \cos\theta$$

Because $\vec{v}$ and $\vec{w}$ are colinear, $\theta = 0$, and the following unit vectors are equal.

$$\frac{\vec{w}}{|\vec{w}|} = \frac{\vec{v}}{|\vec{v}|}$$

Remembering that P is a distance l from Q and that $|\vec{v}| = l$

$$\frac{\vec{w}}{|\vec{w}|} = \frac{\vec{v}}{l}$$

$$\vec{v} = l\frac{\vec{w}}{|\vec{w}|}$$

Therefore

$$\vec{p} = \vec{q} + l\frac{\vec{w}}{|\vec{w}|}$$

21.6 Loop Vector Expressions

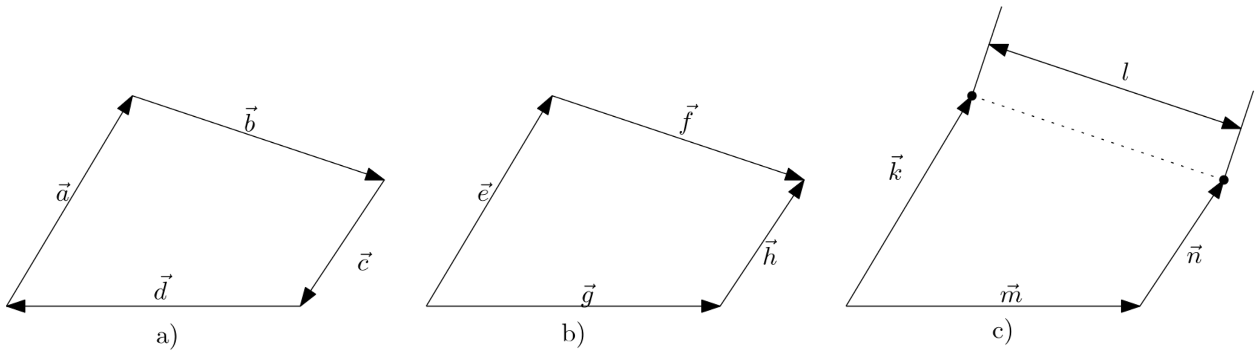


Figure 21.9: Three vector-based expressions defining a loop. a) $\vec{a}+\vec{b}+\vec{c}+\vec{d}=\vec{0}$ b) $\vec{g}+\vec{h}=\vec{e}+\vec{f}$, c) $|\vec{m}+\vec{n}-\vec{k}|=l$

A loop is a set of vectors whose lengths sum to zero, or where two independent vector sums equal each other. If you think about generic mechanisms as branching articulated bodies, sometimes those branches meet back up at their ends to create closed kinematic chains with interesting motion. There are various ways to express those loops, as seen in the figure above. Some expressions produce longer expanded forms, while others are more efficient to compute. Can you tell which form, when expanded to basis vector form, produces the shortest equation?

21.7 Constraints

A constraint is a mathematical description that that imposes limits on the motion of a mechanism. Informally, it can be thought of as the opposite of the freedom of motion of a body, link, or system.

Constraints can be expressed as equalities. Constraint expressions can be scalar or vector-based, although many computing environments require constraints to be described in scalar terms

21.8 Limits

A concept related to a constraint is a limit. Limits are constraints that are only active within a certain domain. They can often be expressed as one or more inequalities.

Limits may be described about specific joints (a “joint limit”), or in addition to a mechanism’s principal degrees of freedom.

One kind of limit, typically imposed by the environment, is the effect of contact. Contact is a one-sided limit that is typically the result of the surface geometry of a linkage contacing something. The point of contact often determines what motion is constrained.

21.9 Constraint Equations

Constraint equations are a mathematical approach used to describe the physical limits imposed by the geometric constraints of joints, mechanisms, and contact.

Specific geometric conditions can be used to describe constraints, such as

- Distances
- Perpendicular, Parallel Vectors
- Point on a line
- Point a distance from a line
- Loops

Constraints can also be used to reduce the freedom of a system’s description. A convenient example is the unit quaternion, which uses *four* parameters to describe something that requires *three* numbers.

- $a^2 + b^2 + c^2 + d^2 = 1$

21.9.1 Solving Systems of Constraint Equations

Constraint equations can be solved in as many ways as you can describe the geometry of mechanisms. Some representations are easier to code, while some representations simplify to much shorter expressions. In this chapter we will describe several ways you can represent the same system using vastly different descriptions

21.10 Assembling constraint equations

1. The first set of equations is generated by directly defining the locations of one triangular face. For each vertex v_i on such a fixed face,

$$\begin{aligned} v_{ix} - x_i &= 0 \\ v_{iy} - y_i &= 0 \\ v_{iz} - z_i &= 0, \end{aligned}$$

where x_i, y_i and z_i are the positions of each point in x, y, and z coordinates. Repeated over the three vertices, this should produce 9 equations.

2. For each line l_i that is not on the edge of the fixed face, there should be two vertices, v_j and v_k connected to it. Using the expanded form of the length constraint , create an equation which should look like
$$v_{jx}v_{kx} + v_{jy}v_{ky} + v_{jz}v_{kz} - \ell_i = 0,$$
where ℓ_i is the length of line l_i. This should be done $m - 3$ times.

3. For each green line, there is an additional constraint that the two neighboring faces are parallel to each other. let vertices v_i and v_j be the two vertices shared by the fixed(green) line, and vertices v_a and v_b be the remaining vertices of the two faces.
$$\vec{a} = \vec{v}_a - \vec{v}_i$$
$$\vec{b} = \vec{v}_b - \vec{v}_i$$
$$\vec{c} = \vec{v}_j - \vec{v}_i$$
$$\vec{d} = \vec{c} \times \vec{a}$$
$$\vec{e} = \vec{b} \times \vec{c}$$
Using the parallel constraint equation generates an extra equation per rigid line,
$$\vec{d} \cdot \vec{e} - |\vec{d}||\vec{e}| = 0.$$
this should be done r times.

There should be a total of $c = 9 + (m - 3) + r$ equations derived from the above three steps. Alternative derivations are possible, but this recipe is one of the more straightforward.

21.10.1 Solved Nonlinearly

For most code you write, data is not *differentiable* or *symbolic*, it is *numeric*. Differentiation is also done *numerically*.

- https://en.wikipedia.org/wiki/Gradient_descent
- https://en.wikipedia.org/wiki/Hill_climbing

21.11 Numerical Solving: Advantages and Disadvantages

21.11.1 Advantages

- Don't need to differentiate variables, only output.
- Faster and Easier to code

21.11.2 Disadvantages

- Initial guess must be close.
- Numerical solving is slow for big systems with many variables.
- Have to solve for valid positions.
  - Continuity between solutions was not guaranteed
  - No knowledge if we skipped through a singularity.

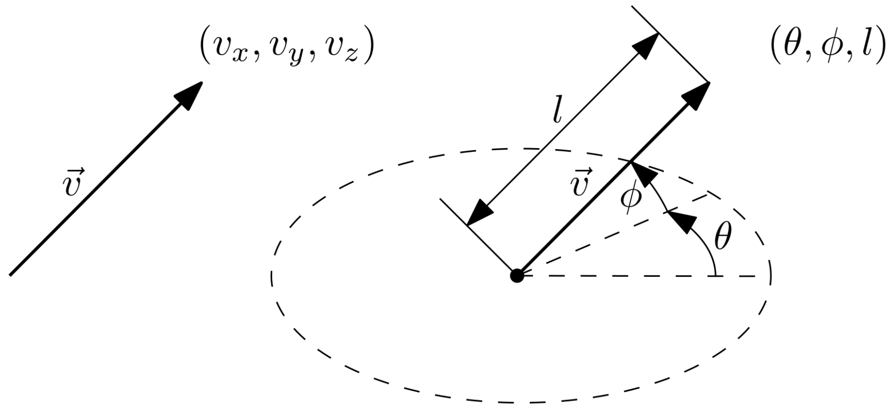
21.12 More Info on `minimize()`

Many Algorithms

- Nelder-Mead
- Powell
- CG
- BFGS
- SLSQP
- ... many more

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

21.12.1 Represenations



Title

- Many ways to represent the same constraint
- Variables don't necessarily map to DOF

21.12.2 Description Choice

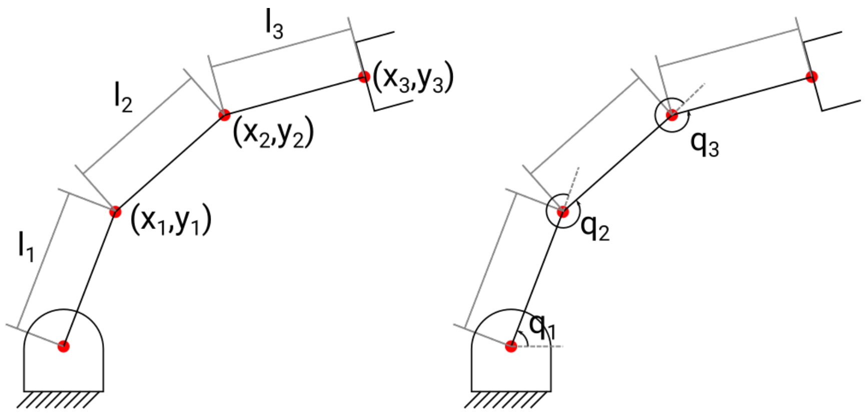


Figure 21.10: caption

Cartesian state variables: $\begin{cases} x_1, \dot{x}_1, \\ y_1, \dot{y}_1, \\ x_2, \dot{x}_2, \\ y_2, \dot{y}_2, \\ x_3, \dot{x}_3, \\ y_3, \dot{y}_3 \end{cases}$

Cartesian State Equations:

$$p_1 = x_1\hat{n}_x + y_1\hat{n}_y$$
$$p_2 = x_2\hat{n}_x + y_2\hat{n}_y$$
$$p_3 = x_3\hat{n}_x + y_3\hat{n}_y$$

Cartesian constraint equations:

$$0 = |\vec{p}_1 - \vec{p}_0| - l_1$$
$$0 = |\vec{p}_2 - \vec{p}_1| - l_2$$
$$0 = |\vec{p}_3 - \vec{p}_2| - l_3$$

Polar state variables: $\begin{cases} q_1, \dot{q}_1, \\ q_2, \dot{q}_2, \\ q_3, \dot{q}_3 \end{cases}$

Polar State Equations

$$\vec{p}_1 = \vec{p}_0 + l_1\hat{a}_x$$
$$\vec{p}_2 = \vec{p}_1 + l_2\hat{b}_x$$
$$\vec{p}_3 = \vec{p}_2 + l_3\hat{c}_x$$

where $\hat{a}_x$, $\hat{b}_x$, $\hat{c}_x$ are unit vectors directed along links 1, 2, 3, and l_1, l_2, l_3 are constants. No constraint equations required.

equations	Cartesian	Polar
state variables	6	3
constraint equations	3	0
DOF	**3**	**3**

Discussion	Cartesian	Polar
Advantages	- Simple representation	- Representation matches physical meaning (structure, motors)
	- Easy to represent all constraints as length of vector	- Fewer representational singularities
Disadvantages	- “Representational” singularites exist	- Requires intuition to select representation
	- State variable redundancy	

21.13 Origami Systems

Using the results of the last section, we can finally start to make a system for analyzing generic origami structures. You can think of an idealized origami system as a set of rigid faces which are permitted to rotate about each other at the fold lines. An ideal origami face is rigid, *i.e.* any motion in the system occurs between faces, along the fold lines. Rigidity in the faces preserves face edge lengths, and as discussed in , the distance between two points can be expressed with . This allows us to create a system of equations which expresses the motion of the origami system with regard to those fold lines.

21.14 Counting Degrees of Freedom

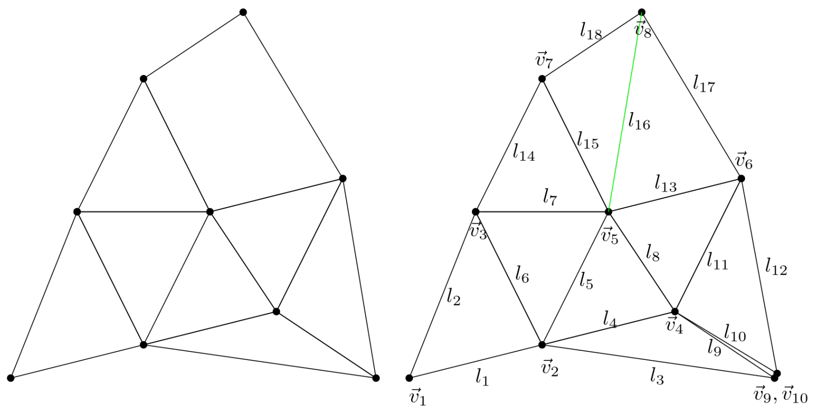


Figure 21.11: Couting Degrees of Freedom in an Origami Mechanism

There is a fairly straightforward way to count the number of degrees of freedom in an origami design, Assuming that your pattern has no holes.

Here are the steps:

1. Draw your origami structure
2. Split any polygons with > 3 sides into triangles by adding extra lines. (green in our example)
3. Separate vertices and group lines for cut lines. Sometimes it makes sense to redraw the mechanism with two vertices next to each other.
4. Number each line as l_i. Let m equal the total number of lines (black & green)
5. Number each vertex (where two or more lines cross or meet) as v_i. Set $n = $ the total number of vertices.
6. Count g, the number of non-hinge lines (in green)
7. Determine f, the degrees of freedom of your system using $f = 3n - m - g - 6$

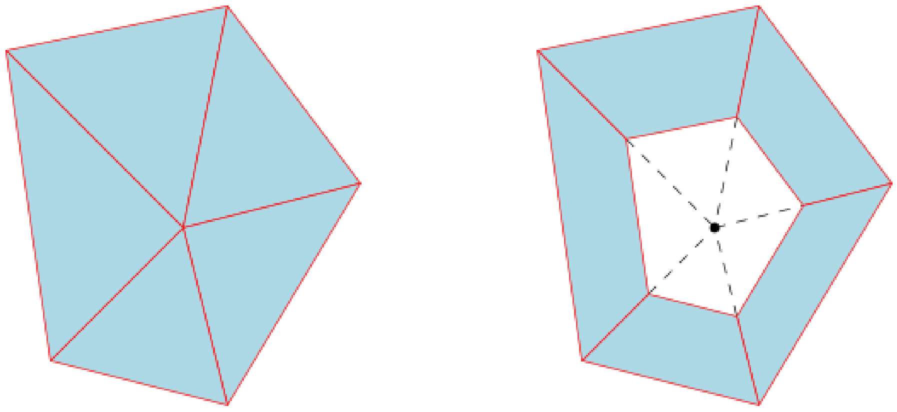


Figure 21.12: Two Equivallent Mechanisms

Why do holes make it harder? They obscure shared vertices, as in the example of equivalent mechanisms, above.

Chapter 22

Duplicate Functionality in Numpy and Scipy

Python makes it possible to do the same thing a number of different ways. In addition, packages like numpy and scipy permit you to manipulate data in a number of ways that are often redundant with each other. As a case study, let's take a look at the many different ways to find the length of a vector.

First, let's import some packages

```
import numpy
import scipy.linalg
```

Numpy lets you create one-dimensional arrays or two-dimensional arrays (matrices) using nearly the same functions, with minor differences. Can you tell the difference between a one-dimensional array and a 2D array?

```
v1a = numpy.array([1,2,3])
v1m = numpy.array([[1,2,3]]).T
```

you can better tell the difference by inspecting their shapes.

```
print(v1a.shape, v1m.shape)
```

```
(3,) (3, 1)
```

v1a is vector 1 as an array, and v1m is the same information stored in a 2D array, or matrix. the shape of v1a lists only one dimension, of length 3. the second one, lists 2 dimensions, of shape 3x1.

If you want to find the length of v1a, you could multiply the elements of v1a, sum them together, and take the square root:

```
((v1a*v1a).sum())**.5
```

```
3.7416573867739413
```

You could also use the `dot()` method, then sum and find the square root. Note that the dot method works on one-dimensional arrays without needing to transpose or massage the shape in any way

```
((v1a.dot(v1a)).sum())**.5
```

```
3.7416573867739413
```

with a matrix, however, you need to transpose v1m in order for the `dot()` method to work without throwing a dimensionality-related error

```
(v1m.T.dot(v1m)[0,0])**.5
```

```
3.7416573867739413
```

because `v1m` is also a matrix, you can use matrix multiplication (`@`) to multiply in a standard row-by-column fashion permitted for matrix multiplication. Be sure to transpose `v1m` in the first case, though. Also note that the result is returned as a 2 dimensional array of length (1,1). To get the scalar value to return, you have to index the [0,0]th element.

```
((v1m.T@v1m)[0,0])**.5
```

```
3.7416573867739413
```

As before, you can also use element-wise multiplication and sum each element's result together. Note that this method is a 1:1 match for the 1-dimensional array case

```
((v1m*v1m).sum())**.5
```

```
3.7416573867739413
```

You can also use scipy's linalg package to find the vector norm of both the matrix and 1D array using the same terminology, which can be an advantage when you are not sure what form will be supplied.

```
scipy.linalg.norm(v1m)
```

```
3.7416573867739413
```

```
scipy.linalg.norm(v1a)
```

```
3.7416573867739413
```