Using the Posemath Library

by

Karl Murphy

Updated 4/14/99

This gives examples of how to use the posemath library, part of the RCS libary. Most of the examples are given in C. Posemath also supports C++ and the corresponding code is more compact due to operator overloading.

See Also RCS lib Document on posemath

Introduction

A **Vector** is a 3 dimentional quantity that has magnitude and direction. A vector is not a **point** although a vector can be used to represent the relative position between two points. A vector is not fixed in space, it only has magnitude and direction. Imagine an arrow that can be slid all around (although not rotated).

A **Reference Frame** is a set of three mutually perpendicular unit vectors, often denoted **x**, **y**, and **z**, and an associated point called the origin.

Vectors

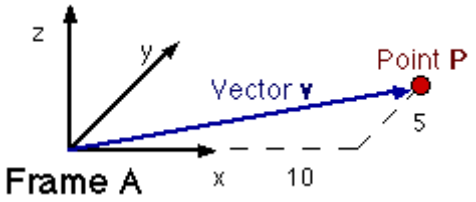
Given a reference frame **A** and a vector **v**, there exists 3 unique scalars, vx, vy, and vz such that

v = vx **x** + vy **y** + vz **x**

The **representation** of the vector v in the reference frame A is the three scalar values, [vx, vy, vz]T. Note that vx is the dot product of **v** and **x**, etc. The values [vx, vy, vz] are called **v** in **A** and are the representation used most often in posemath. The three scalars are meaningless without an associated reference frame.

Example 1

The vector **v** is the relative position of point **P** from the origin of frame **A**. Express the vector in posemath notation.



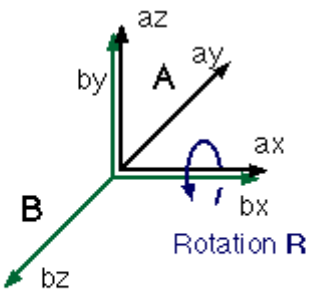

```
PmCartesian v = {10, 5, 0}; /* vector v in frame A */
```

The position of a point relative to a reference frame is the vector from the origin of the frame to the point expressed in the frame. The representation of **v** (the vector from the origin of **A** to **P**) is different in different frames that have different orientations but always has the same magnitude. The position of **P** in different frames might also have different magnitudes.

Posemath also supports cylindrical and spherical representations.

Reference Frame Rotations

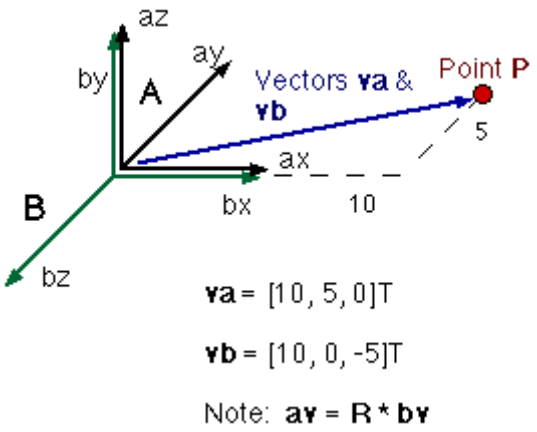
Coordinate frames can be rotated relative to one another. There are many ways to represent rotations. A rotation matrix is a 3x3 matrix where the columns are the representations of the **x**, **y**, and **z** axis of the rotated frame expressed in the unrotated frame. See the figure below.



$$\mathbf{R} = \text{Rotate}(X, 90°) = [\mathbf{bx}, \mathbf{by}, \mathbf{bz}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

The elements of the matrix are the dot products of the various axis (**ay** dot **bz** = -1).

The vector representation in a rotated frame can be calculated by using normal vector - matrix multiplication as shown below.



Strictly speaking, the vectors **va** and **vb** are the same, they both span from the origin to the point **P**, even thought their representations in the two frames are different.

Besides the rotation matrix, posemath supports many representations for rotations. One of which, the quaternion, is a strange but computationally nice representation. It has a scalar and vector part. The scalar is the cosine of half angle of rotation and the vector is the unit rotation vector multiplied by the sine of the half angle. I told you it was weird. Because of the computational niceties, posemath routines have a bias toward using quaternions. When you use them, think of them as a rotation matrix. pmPrintf() will even convert them to a matrix and print them that way.

Example 2

Rotate **bv** to get **av** as shown in the figure above.

```
PmCartesian av; /* vector av in frame A */

PmCartesian bv = {10, 0, -5}; /* vector bv in frame B */

PmQuaternion rq; /* the rotation expressed as

a quaternion */

PmRotationMatrix rm; /* the rotation expressed as

a matrix */


/* Method 1 */

rm = (PmRotationMatrix) {{1,0,0},{0,0,1},{0,-1,0}};


/* When reading a matrix, quaterian, unit vector, etc.

as ASCII you should normalize it if not all +-1 & 0's.

We will do it anyway */
```

```
pmMatNorm( rm, &rm );


pmMatCartMult( rm, bv, &av );


/* Method 2 */

/* represent the rotation as a quaternion */

pmAxisAngleQuatConvert( PM_X, PM_PI / 2, &rq);

/* or */

pmMatQuatConvert( rm, &rq);

pmQuatCartMult( rq, bv, &av );


/* print rm as a quaternion and as a rotation matrix */

pmPrintf("As a quat %q, as a matrix \n%Q\n", rq, rq);


/* in C++ */

av = rq * bv;
```

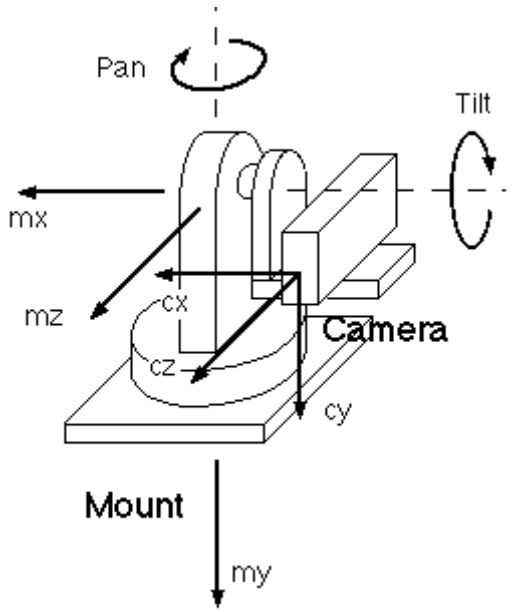
Example 2b

Now the other way: Rotate **av** to get **bv** as shown in the figure above.

```
PmQuaternion rqInv; /* the rotation inverse */

pmQuatInv(rq, &rqInv); /* invert rq */

pmQuatCartMult( rqInv, av, &bv );


/* in C++ */

bv = (-rq) * av; // (-rq) is the inverse of rq
```

Example 3

Given the pan tilt unit and the coordinate frames as shown, compute the rotation from the mount to the camera. Neglect translations




```
double pan, tilt; /* pan and tilt in rads */

PmQuaternion panTilt; /* pan tilt rotation */



pmAxisAngleQuatConvert( PM_Y, pan, &panTilt);

pmQuatAxisAngleMult( panTilt, PM_X, tilt, &panTilt);
```

Reference Frame Transforms

The transformation from one coordinate frame to another requires translation and rotation. One representation is the homogeneous transformation, a 4 x 4 matrix with the rotation matrix in the upper left, the translation vector on the right, and a row of 3 zeros and a 1 along the bottom. The bottom row can be used to represent other transformations such as a change in scale, but these are not currently supported in posemath. The position of a point relative to a new reference frame can be calculated using normal vector - matrix multiplication as shown below. (A 1 is added to the end of each vector representation for proper matrix multiplication.)

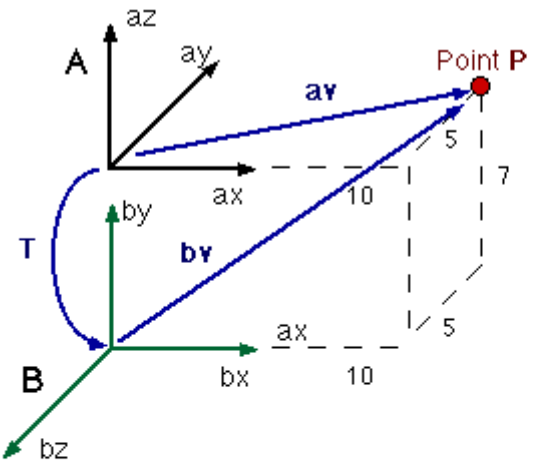
The translation is expressed in the non-rotated frame.

av = **T** * **bv**

where

av = [10, 5, 0, 1]T

bv = [10, 7, -5, 1]T



$$\mathbf{T} = \text{Tran}(0, 0, -7)\ \text{Rotate}(X, 90°) = \begin{bmatrix} \mathbf{bx} & \mathbf{by} & \mathbf{bz} & \mathbf{v} \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & -7 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The posemath library supports homogeneous transforms but is biased toward a different representation, the **pose**, a quaternion and a cartesian translation. It is often easier to think of the pose as a homogeneous transform.

Example 4

What is the pose for the above transform. Calculate **av** given **bv**.

```
PmCartesian av; /* vector av in frame A */

PmCartesian bv = {10, 7, -5}; /* vector bv in frame B */

PmPose t; /* transform w/ the rotation

expressed as a quaternion */

PmPose t_inv; /* inverse t */


/* represent the rotation as a quaternion */

pmAxisAngleQuatConvert( PM_X, PM_PI / 2, &t.rot);


/* tran in un-rotated frame. One element at a time */

t.tran.x = t.tran.y = 0;

t.tran.z = -7;
```

```
pmPoseCartMult( t, bv, &av ); /* calc av given bv */


/* print t with a quaternion and with a rotation matrix */

pmPrintf("As a quat %p, as a matrix \n%P\n", t, t);
```

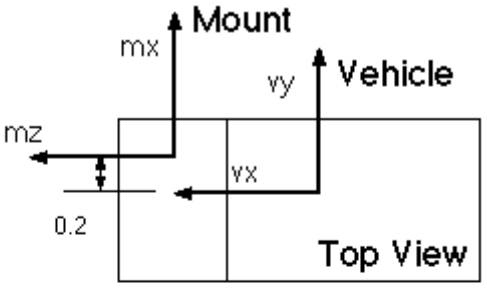
Example 4b

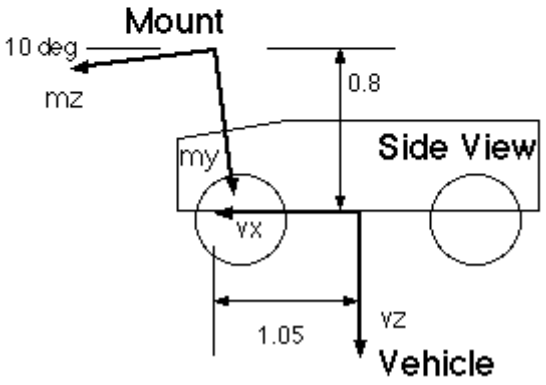
Now the other way: Calculate **bv** given **av**.

```
pmPoseInv( t, &t_inv); /* invert t */

pmPoseCartMult( t, av, &bv ); /* calc bv given av */


/* in C++ */

bv = (-t) * av;
```

Example 5

Given the two frames, **Vehicle** and **Mount** as shown in the figure below calculate the transformation from **Vehicle** to **Mount**.






```
PmPose mount; /* transform from Vehicle to Mount */


/* the translation is in the base frame (ie vehicle) */

mount.tran.x = 1.05;

mount.tran.y = 0.2;

mount.tran.z = -0.8;


/* rotate to align X axis */

pmAxisAngleQuatConvert( PM_Z, PM_PI / 2, &mount.rot);


/* rotate to almost align Y & Z axis */

pmQuatAxisAngleMult( mount.rot, PM_X, PM_PI / 2, &mount.rot);


/* rotate for 10 deg offset */

pmQuatAxisAngleMult( mount.rot, PM_X, 10.0 * TO_RAD, &mount.rot);
```

Multiple Transforms

Multiple transformation are computed using matrix multiplication when using

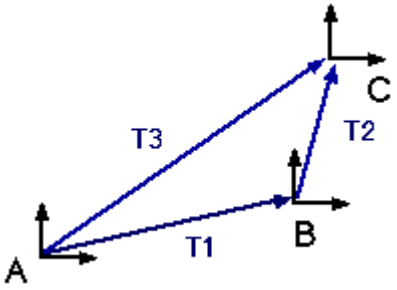
Homogeneous Transformations and by using pose multiplication when using poses. Either way, the rules are the same. Multiplication is associative,

(**T1** * **T2**) * **T3** = **T1** * (**T2** * **T3**)

but not commutative (in general)

T1 * **T2** != **T2** * **T1**

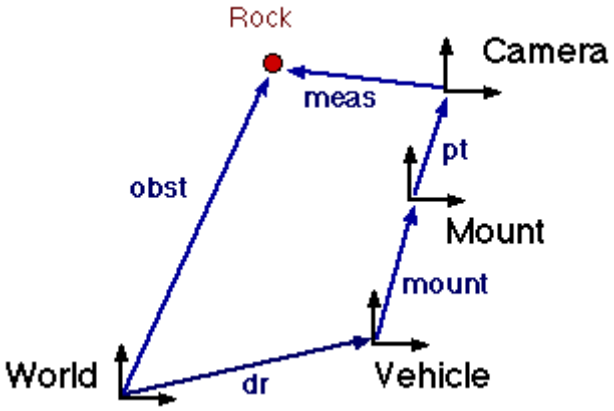
Manipulate equations by pre- and post-multiplying by inverses. Drawing a simple sketch showing the frames and relative transforms can help. As you traverse from one frame to the next, post-multiply by the transform if you are traversing in the direction of the arrow, and post-multiply by the INVERSE of the transform if you are traversing in the opposite direction.



A to C: T3 = T1 * T2
A to B: T1 = T3 * INV(T2)
B to C: T2 = INV(T1) * T3

Example 6

Given the pose of the vehicle in world frame, the pose of the mount in vehicle frame, the pose of the camera due to the pan tilt, and the x, y, z measurement of a rock in camera coordinates (see above examples), what is the position of the rock in the world frame?

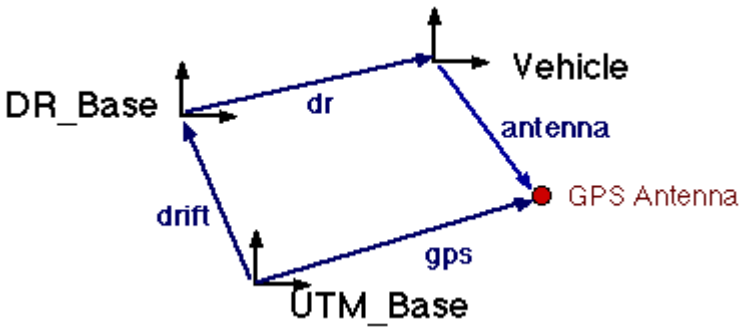



```
PmPose dr, mount, pt; /* these are set elsewhere */

PmCartesian obst, meas; /* meas is set elsewhere */


PmPose ttt; /* world to camera, temp pose*/
```

```
pmPosePoseMult( dr, mount, &ttt); /* mount in world */

pmPosePoseMult( ttt, pt, &ttt); /* camera in world */


pmPoseCartMult( ttt, meas, &obst); /* rock in my world */


/* in c++ */

obst = dr * mount * pt * meas; // rock on!
```

Example 7

A vehicle has two navigation sensors, a dead reckoning unit and a gps unit. The DR unit measures vehicle orientation and distance traveled since start up (in northing, easting, and down). The GPS measures the position of the antenna relative to the UTM zone (in northing, easting, and down). The DR_Base and the UTM_Base have the same orientation. Calculate the drift, ie, the offset of the DR_Base. In practice, this drift would be filtered to take out noise in the gps measurement.




```
PmPose dr;

PmCartesian drift, antenna, gps;

PmCartesian ttt; /* dr_base to antenna vector */


pmPoseCartMult( dr, antenna, &ttt);

pmCartCartSub( gps, ttt, &drift);

/* note we can subtract because there is no rotation between

the two bases, */
```