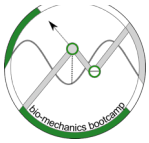


TUTORIAL 6 – Rotation Matrices & Euler Angles

Why do we care? Learning how to compose and manipulate rotation matrices will make your code run faster and more smoothly. This will be important if you are looking at complicated models with many xyz points. A basic understanding of rotation matrices (via some basic linear algebra) will reinforce your understanding of how rotations work in terms of geometry and visualization. Most importantly, rotation matrices will organize the Sine and Cosine terms automatically for you (assuming you composed your rotation matrix correctly, which you'll learn shortly). So now that you understand Sines and Cosines, you don't have to worry about them again for the next few tutorials - they will come back for you when we talk about waveforms.

For the purposes of this and following tutorials, a reference frame is an xyz coordinate system (aka frame, aka basis) that is centered and oriented with respect to another coordinate system. We will call 'Ground' the global xyz coordinate system with origin = {0x, 0y, 0z}.

Aims of this tutorial: *Do a mini Linear Algebra review *Become comfortable with right-hand rotations and rotation order *Euler Angles *Practice building rotation matrices in *MM* *Matrix multiplication

Skills required prior to this tutorial: General Mathematica *MM* workflow. 2D and 3D interactive plotting and styling. Building custom functions. 3D plotting using ListPointPlot3D and Graphics3D.

Background and theory: You should be comfortable with the right-hand coordinate system by now. If not, please review the background from the previous two tutorials.

■Euler Angles involve a sequence of 3 rotations that can be put together (composed) with three rotation matrices multiplied together. They describe how to get from one orientation to any other orientation with a sequence of 3 different angles called 'Euler Angles'. Beyond this, the internet has many opinions on what is meant by 'Euler Angles'. There are many conventions of Euler Angles – I won't attempt to add any clarity to the topic. We will discard most nuances and use two alternative conventions: 1). Z-X-Z: This represents rotations about the z axis followed by x axis followed by the z axis again. This means: first rotate about the 'ground' Z axis, then rotate about the 'ground' X axis then do another final rotation about the 'ground' Z axis. 2). Z-X'-Z": As an alternative to the 2nd and 3rd rotations, they can be made about the rotated frame of reference. In other words, after the first rotation, ground XYZ becomes X'Y'Z'. So the 2nd rotation would be about X' (not X). Likewise, the 3rd rotation will be about Z" which is the new position of the Z axis after it has been rotated twice (ground Z → Z' → Z"). To avoid confusion, I'll avoid the term "Euler Angles" altogether and simply talk about rotations as they are, e.g. Z-X-Z or Z-X'-Z" to indicate whether we're talking about convention 1 or 2 (among many other possibilities we'll ignore). By the end of the next tutorial, you'll know enough to select your own rotation convention that's most convenient for you.

■ **REVIEW - Coordinate system review** As you know, a Cartesian coordinate system is a set of 3 orthogonal 'unit basis vectors' pointing away from a common origin. They are orthogonal by necessity and unit vectors for convenience. The global reference frame to which all other frames refer we will call 'Ground '. It is based at xyz = {0, 0, 0} and the axes are the

following vectors. I'll denote vectors in lowercase bold blue. I'll denote matrices in capital bold blue. As we learned before, these vectors mean: start at the origin, move x distance then move y distance than move z distance.

x = {1, 0, 0}
y = {0, 1, 0}
z = {0, 0, 1}

■**Linear algebra review: CRUCIAL things you need to know about matrices** -- This is basic linear algebra, some of which is review from the previous tutorial. Don't skip this section unless you've studied linear algebra before!

1. Matrix multiplication works as follows:

$$\mathbf{A}_{(m \times k)}.\mathbf{X}_{(k \times n)} = \mathbf{B}_{(m \times n)}$$

Where the dot is the dot product (representing matrix multiplication). If this doesn't make sense, please review this now in the previous tutorial. If n =1 then you are doing matrix-vector multiplication which produces a vector output.

►**Try it In *MM*:** Create a 4x3 matrix of random numbers using RandomReal[] command. For example, RandomReal[{0,4}] gives a random number between 0 and 4.

```
A = Table[RandomReal[…], {rows, 1, 4}, {columns, 1, 3}];
```

Do the same for X, except pick *k* and *n* dimensions such that the above equation is satisfied. In *MM* the Dot product function is simply a dot. A.X (same as Dot[A, X])
If X has the wrong shape, it won't work. Try to produce B matrices of different sizes.
Now try to reverse the order of multiplication. Notice how it doesn't work (or gives a different answer). This is because reversing the order violates the equation above. Specifically, X(kxn).A(mxk) which will only work if m = n. Finally, try it with A and X both as 3x3 matrices and notice how reversing the order gives different answers. Please remember: Order matters in matrix multiplication.

2. A rotation matrix is a 3x3 matrix usually of Sine and Cosine terms with angle(s) that represent rotations about a known vector (e.g. the z axis, {0, 0, 1}). If you compose the matrix correctly and do the correct linear algebra (not hard), your Sines and Cosines will end up in the right place without having to worry about them any more. E.g. rotating around the origin about the y axis ({0, 1, 0}) gives:

$$\boldsymbol{R} = \begin{matrix} Cos(\theta) & 0 & Sin(\theta) \\ 0 & 1 & 0 \\ -Sin(\theta) & 0 & Cos(\theta) \end{matrix}$$

This is just an example. Don't even think about memorizing it because there are an infinite number of different R's, depending on the rotation axis. To use R, we just use the dot product:

$$\mathbf{R}_{(3 \times 3)}.\mathbf{p}_{(3 \times 1)} = \mathbf{p'}_{(3 \times 1)}$$

Where p is a single xyz point and p' is the rotated point (in this case rotated by θ about the y

axis).

►**Try it In *MM***: Create an R using RotationMatrix[]. For example, rotating about the y axis Pi/2 degrees: R = RotationMatrix[Pi/2, {0, 1, 0}]. Make an xyz point, e.g. p = {1, 0, 0}. Dot p with R and look at the result. It should make sense – we're rotating about the y by 90 ° using the right-hand-rule. Remember that if an axis is pointing from the origin towards your eye, positive rotation will be left (counterclockwise). Where should point {1, 0, 0} end up? Try different points.

3. R can be the composition of a sequence of rotations (e.g. an "Euler" Z-X-Z rotation). This is done by dotting each R matrix. For example, what if we want to rotate about z by angle1 then about x by angle2 then z by angle3. We would find R1, R2 and R3 then multiply them together.

!The order of dot products is the reverse of the order of rotations!

$$R = R3 \cdot R2 \cdot R1$$

So, a single matrix-vector multiplication can rotate in 3 angles simultaneously, given the proper R matrix. Importantly, these rotations are all about the global 'ground' xyz axis. Each subsequent rotation refers to ground, not to the rotated axis. Next tutorial, we'll go over Z-X'-Z" (or 3-1'-3") rotations in which each subsequent rotation in a sequence refers to the rotated frame.

►**Try it In *MM***: Using what you learned above, generate 3 different R matrices (different angles and/or different axes). Multiply them together to get a single R then rotate a point using it. Change the order and appreciate that Order Matters In Matrix Multiplication.

4. R matrices are orthogonal. Let me first explain two concepts: Firstly, the inverse of a matrix is analogous to the reciprocal of a number (1/number). So we can undo a multiplication by multiplying by the inverse. For example, if we want to solve for A in B=A.X, we would simply write $A=B.X^{-1}$ where X^{-1} is the inverse of X. Secondly, the transpose of a matrix is when we swap rows for columns. An orthogonal matrix is a matrix whose transpose is equal to its inverse. Who cares? This is crucial because to undo a rotation, all we have to do is transpose R. In other words, R^T (transpose of R) is the un-rotation matrix. So if we want to reverse a rotation we've already done, we use R^T in place of R.

5. If we arrange our points properly, we can rotate multiple xyz points simultaneously. For convenience, we will always organize our xyz points in a Nx3 matrix where N is the number of points. Let's call our Nx3 matrix P. Here's how to rotate multiple points:

$$P'^T = R.P^T$$

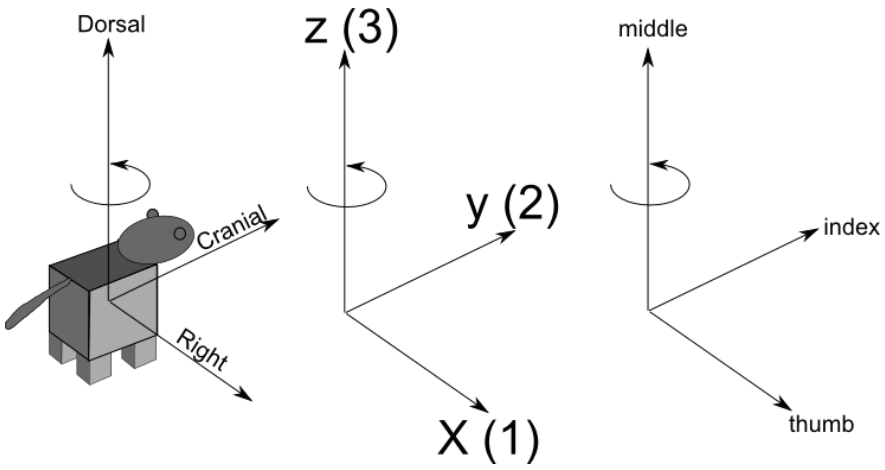
In other words, dot R with the transposed point matrix (= 3xN) which gives the transpose of the rotated points. In practice, simply transpose P'^T to get to P'. Don't worry, it will make more sense when we write a function that does this in *MM*. This doesn't work with translation, so we will use a convenient translation function built into *MM*. If you don't have *MM*, you have to use for loops and row-by-row addition (works fine but is slow) or 4x4 matrices which is fast, but a bit tricky to implement. I'll have an appendix tutorial on this at

some point.

►**Try it In *MM*:** Generate an R matrix and an Nx3 matrix of random xyz points, P. Transpose P and call it Pt: Pt = Transpose[P]. Now, try rotating all of the N points in bulk by using the equation above. Transpose the result to get P'.

Next, try composing a rotation through multiplication of 3 different R's. Rotate P → P' Then, undo the rotation using the transpose of the rotation matrix to get back your original points P.

■ **REVIEW - Right hand system:**
For all 3D problems in these tutorials, we'll use the right hand coordinate system where z is up. Anatomically, +y is cranial, -y is caudal, +x is right of the midline, -x is left of the midline and +z is dorsal, -z is ventral.



Visual tool: Before beginning with exercises, I recommend making a simple tool of a right-hand coordinate system with axes labeled '1' '2' '3' or 'x' 'y' 'z'. I made mine from a paper clip with tape for labels.

PART 1: Practice

Exercise 1: If the x axis is rotated by positive 90 ° **(Pi/2)** about the z axis, what would be the vectors describing the rotated coordinate system (i.e. x', y', z') assuming we start with:
x = {1, 0, 0}
y = {0, 1, 0}
z = {0, 0, 1}

You should be able to do this in your head or using your paper clip tool. Verify your result performing the rotation of x, y, z in *MM* using a rotation matrix.

Exercise 2: Similar to the above, what would x', y', z' be if x, y, z were rotated by Pi/2 about the y axis?

Exercise 3: Similar to the above, what would be the vectors describing the rotated **x, y, z** if you performed a –Pi/2 rotation about y followed by a Pi/2 rotation about the rotated x axis (which was rotated by the first rotation).

PART 2: Building *MM* code for rotation and reference frame analysis

See included code for hints if you get stuck on coding

IMPORTANT: Before we begin, I'd like to define a matrix called a 'frame' or a 'reference frame'. For now, this will be a 4x3 matrix of xyz points (vectors). It is defined as follows: frame = {axis 1, axis 2, axis 3, origin} where axes 1, 2, 3 are unit vectors (their magnitude = 1) pointed in x, y, z directions respectively. Get comfortable with axes xyz ←→ 123. So, to pick out the x axis vector we write frame[[1]] and frame[[2]] for y, etc. Unit length vectors simplifies the math. The ground frame = {{1, 0, 0}, {0, 1, 0], {0, 0, 1}, {0, 0, 0}}. If we shift it's origin, we change frame[[4]] → new origin.

Exercise 1 (*MM*): Write a custom function that performs a rotation applied to N number of points (see point 5 above). Let's call the function transform3x3. It will be come the basis for code we write in future tutorials. We will modify it as we go. For now, it should have two arguments: Rmatrix3x3 and Pnx3 representing the 3x3 rotation matrix and an Nx3 matrix of xyz points.

transform3x3[Rmatrix3x3_, Pnx3_]:= …

Write the code in the function such that the output is P'. Test your function by replicating your answer from exercise 1: Input a rotation matrix rotating Pi/2 about the y axis. P = {**x, y, z**} (like in exercise 1). You should get the answer **P'**. Do your transposing such that P' is Nx3 (not 3xN). Try it for an 4x3 matrix (**P** = 4 xyz points). For example, try rotating P = {**x, y, z,** origin} just to make sure it works on Nx3 matrices. Look at the code pdf if you get stuck.

Exercise 2 (*MM*):

Step1: Create a matrix composed of 3 vectors representing ground **x**, **y**, **z**. As you should know by now, that will look like {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}. This can be done by making a 3x3 identity matrix:

xyzvectors = IdentityMatrix[3];

Let's add it to our matrix (you'll see why soon) to make it a 4x3 'reference frame' matrix.

origin = {0, 0, 0};
xyzframe0 = Join[xyxvectors, {origin}]; (*frame 0 for 'ground'*)

Plot your frame using Plot3D, etc. (see Appendix 2 if you're hazy on 3D plotting). You'll see that it looks fairly useless. We would like to see a wire frame for our coordinate system with three axes sticking out from the origin. You have to tell *MM* to do this explicitly by starting at the origin, going to the tip of the x axis then back to the origin then to the tip of the y axis, etc. In other words, insert the origin between each vector like this {{x1, y1, z1}, {0, 0, 0}, {x2, y2, z2}, {0, 0, 0}…etc}.

Step2: Make a function called frameDraw that does the above automatically.

I.e.
frameDraw[xyzframe_]:= …
See code pdf if you get stuck.

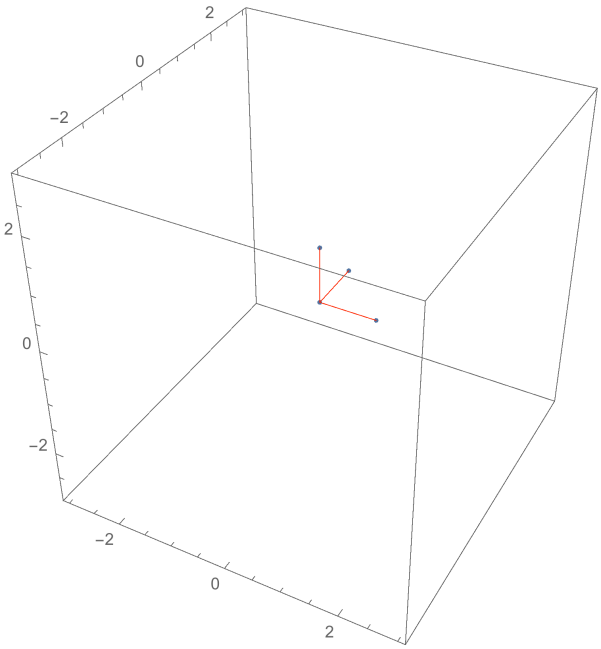
frameDraw[xyzframe0] → {axis1, origin, axis2, origin, axis3}

Step 3: Now, define a new variable for your frame points for plotting:
forplotting0 = frameDraw[xyzframe0];(*your frame points organized for plotting convenience*)

Use ListLinePlot3D[…] and Graphics3d[Line[…]] to plot your frame.

frameplot0 = Show[…];

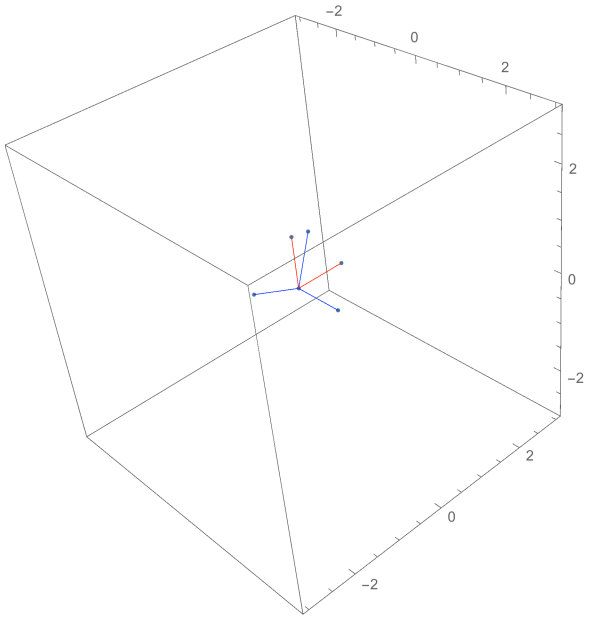
It should look like this:



Step 4: Define a rotation matrix (any will do) and call it r1. For example, rotate by 1.2 radians about the y axis. Rotate xyzframe0 by r1 and call it xyzframe1. Use transform3x3 to do this.

r1 = RotationMatrix[…];
xyzframe1 = transform3x3[…];
frameplot1 = Show[…];

Superimpose plots of xyzframe1 and xyzframe0 like this:
Show[frameplot0, frameplot1]



Step 5: Modify the previous plot to make it interactive where you can change the angle of rotation. Place all of the necessary code for xyzframe1 inside the manipulate cell. Also try rotating about different axes.

Exercise 3 (*MM*): Multiple rotations

Step 1: Based on the code from Step 5 above compose three rotations **ra, rb, rc** about Z, X, Z axes using angles anglea, angleb, and anglec respectively. This breaks xyzframe1 into 3 rotations:

r1a = RotationMatrix[anglea, …];
xyzframe1a = transform3x3[r1a, …];
r1b = …
xyzframe1b = …
r1c = …
xyzframe1 = … (*this is the final rotation, giving us xyzframe1*)

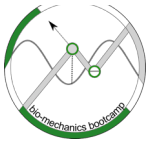
Play with your rotations and get a feel for how the different angles can entirely control the orientation of xyzframe1.

Step 2: Copy the code from above, but lump **ra**, **rb** and **rc** into a single **R** using matrix multiplication. Mind the rotation order. And use transform3x3[…] to create a copy of xyzframe1 called xyzframe1alt. This replicates step 1 with a single matrix multiplication. Plot xyzframe1alt on top of xyzframe1 (hint, use dashed lines so you can see them both) to convince yourself that they produce the same result.

Appendix 1: How to make a custom function

If you want a single command to do any arbitrary number of arbitrarily complex operations, define an actual function. functionName[argument1_, argument 2_, etc.]:= Some operations on argument1 and argument2. You always need the square brackets and underscore (_) after each argument, separated by commas. You also will need ':=' to let *MM* know it's a function.
E.g. add two numbers:
addTwo[number1_, number2_]:= number1 + number2
Input: addTwo[8, 4]
Output: 12
You can define default arguments. For example, addTwo[number1_, number2_:=3] means that unless otherwise specified, the second argument will be 3:
Input: addTwo[8]
Output: 11

You can do something more complicated.
E.g.
doSomething[a_, b_, c_]:= c*Table[i, {i,a,b}]
This will generate a list going from a to b and multiply each list element by c.



You can also execute several lines of code by defining intermediate variables that are used within (and only within) the function. For this, you should use the Module[] command which declares variables that stay local and can't interact with any other code outside of the Module brackets:

Module[{list of variables}, your code]

E.g.
doSomething[a_, b_, c_]:=Module[{intermediatevar1, intermediatevar2, intermediatevar3, output},
intermediatevar1 = a*Sin[b];
intermediatevar2 =Table[Cos[intermediatevar1]* i, {i, 1, c}];
intermediatevar3 = {intermediatevar2, intermediatevar2/2};
output = intermediatevar3
](*end module*)

Note that all lines except the last have a semicolon.

Appendix 2: 3D point plotting

To plot data points in 3D, use ListPointPlot3D[…] on data in the form N x 3 where N is the number of rows (each row is one data point) and 3 is number of columns for xyz. It's not super useful on its own. You can overlay lines to connect the dots using Graphics3D[Line[…]] accepting data in the same form.

Pnx3 = matrix of xyz points…
pointplot = ListPointPlot3D[Pnx3];
lineplot = Graphics3D[Line[Pnx3]];
Show[pointplot, lineplot]

For line styling, you can add arguments within the Graphics3D using curly brackets.
Graphics3D[{…., Line[]}]

For example, Graphics3D[{Blue, Line[Pnx3]}]. You can do other things such as change the thickness, etc. Search MM help for Line[] for more details.

Finally, force the plot range by putting a PlotRange->{{xmin, xmax}, {ymin, ymax}, {zmin, zmax}}. Also use AspectRatio->1 and BoxRatios->{1, 1, 1} as arguments in the Show[] function to keep the plot from resizing in annoying ways.

Links:

http://mathworld.wolfram.com/EulerAngles.html