TUTORIAL 4 – Trigonometry C (building a 3D morphological model)

Why do we care? Please see statement in previous tutorial.

Where are we going with this? There are two tools we will learn over the next few tutorials:

1. Working forward from known angle and segment length inputs, we will solve for unknown xyz limb/body position so that we can make predictions and understand moving anatomy better. We'll also use these tools to animate data from actual numerical models. To start, think of it like building a toy model that helps you visualize how the motion/position of one part influences motion/position of another. E.g. if the knee extends, how much does this push the body up in the ZY plane, etc.

2. Working backward from known xyz position data (e.g. from high speed video or motion tracking) we will solve for unknown angles and segment lengths so that we can estimate muscle movements or calculate joint torques (rotational forces).

Aims of this tutorial:
***Convert our 2D stick figure model into 3D**
***Get introduced to 3D polar coordinates**
***Learn how to manipulate angles in 3D (basics)**

Skills required prior to this tutorial: General Mathematica *MM* workflow. Plotting with ListPlot and manipulating the arguments such as PlotRange and PlotStyle. Use of Manipulate[]. Making custom functions in *MM*. PRIOR COMPLETION OF TUTORIAL 3 is recommended.

Skills gained from this tutorial: * Mathematica '*MM*' software:
-3D plotting of points using ListPointPlot3D[] and Graphics3D[].

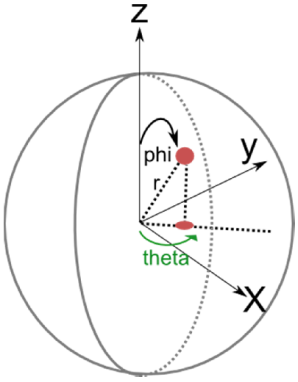
Spoiler alert: This tutorial does not teach you the easiest way to model a limb (i.e. a linked chain of rigid bodies). For now, we will use the inefficient method because it will reinforce geometry principles. Following this tutorial you'll know the foundation such that it will be easier to appreciate and learn more sophisticated geometry tools such as rotation matrices and reference frames (following tutorials).

Background: 2D and 3D polar coordinates

2D Polar: In a nutshell, polar coordinates are an alternative way besides Cartesian to represent the position of a point. Instead of {x, y}, polar is {r, theta} where r is the distance to the origin and theta is the counterclockwise angle as if 'r' is rotating. For example, we tell time in polar coordinates by giving theta as hours or minutes with the clock face in the XY plane. Since the hands of a clock are constant length, r is irrelevant. You can also tell time in X Y coordinates, but this is silly because both X and Y would be changing. The motivation for representing data in polar vs. Cartesian depends on the shape being described. A circle is very simple in polar (in terms of r and theta) and more complicated in Cartesian (in terms of x & y). Conversely, a straight line is most easily represented in Cartesian vs polar. Nothing is fundamentally different between the two systems – they each work on an xy (or xyz) axis system. They simply offer alternative ways of describing the same geometry. Your 2D morphological model from the prior tutorial could be represented in polar easily. For

example, a bone rotating about the origin with length = r1 and an angle (theta1) with respect to horizontal will have the coordinates joint1 = {r1, theta1}. Conversion from polar to Cartesian (and back) is simple: x = r*Cos(theta); y = r*Sin(theta). Not by coincidence, these have the same exact form as the x, y coordinates of your 2D limb model { l*Cos(theta), l*Sin(theta}. For more details, see the Wolfram Mathworld article on polar coordinates (link at bottom).

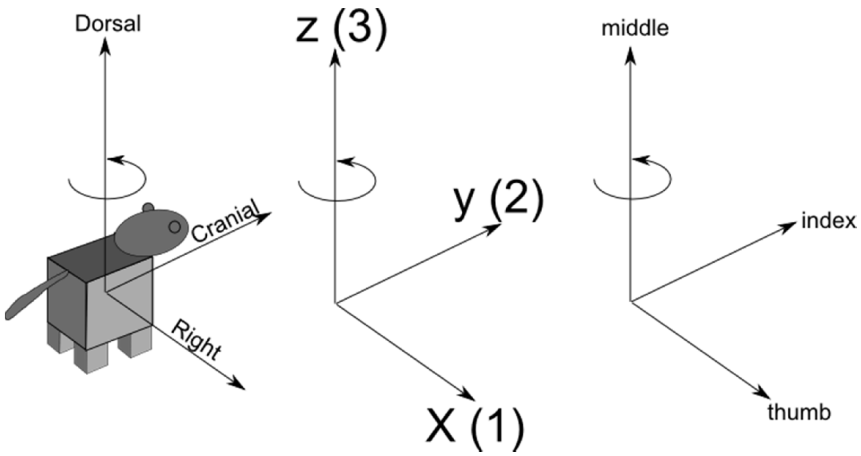
3D Polar: We can easily extend from 2D into 3D by adding another angle. So, if we're in the XY plane we have {r, theta}. If we move up and down we need a second angle, phi. So, the point coordinate is {r, theta, phi).



These are spherical polar coordinates which are useful for describing geometry or motion on or within a sphere. If you think of the coordinate system as defining a volume, the volume in this case would be a sphere. Essentially, theta is longitude and phi is latitude (except starting at the north pole, not at the equator). If you live in NY city, your address could be in Cartesian e.g. xyz = {350, 5th avenue, floor 103}. However if you want someone coming from somewhere deep near the center of the Earth, it may be more useful to give the address in spherical polar rtp = {R, 73.9857° W, 40.7484° N} where R = height of Empire State Building + radius of the Earth.

In my view, spherical polar coordinates are also useful for describing motion about a 'ball-and-socket' joint such as the vertebrate hip or shoulder (more later). For example, if your shoulder is the origin and your head points up along the z axis, you can move your arms up and down (changing phi) or forward-backward (changing theta) corresponding to anatomical axes of adduction/adduction (phi) vs. protraction/retraction (theta). For more details, see the Wolfram Mathworld article on spherical polar coordinates (link at bottom).

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. We will explore coordinate systems more deeply in later tutorials. Note: Positive rotation about any given axis is counterclockwise (left) when viewed straight-on (i.e. with the axis arrow pointing towards your eye).



TUTORIAL 4 – Trigonometry Part C (building a 3D morphological model) Chris Richards, Modified by…

3D ANGLES:
This is a huge can of worms and there are many ways of representing this and multiple solutions. My preference is to use a mix of spherical polar coordinates and Euler angles (defined later) where I find most convenient.

TUTORIAL OUTLINE:
PART I: Starting with your 2D model from Tutorial 3, rename the joints, add a z coordinate and modify the plots.
Part II: Modify the x y z expressions to allow for a 3D angles.
PART III: Add a fictitious body to our limb model

Before we begin, you can always check your code with the example code included in this tutorial package. You will be writing a lot of code which builds on previous code. Be sure to have each code block working properly before moving on to the next section.

PART 1: 2D to 3D conversion

This is not super hard. Each joint point is a 2-element list, so we need to add a third element for z. If everything takes place in the XY plane, z always = 0.

NOTE: The general data structure we will use for most 3D operations is N x 3 (N rows, 3 columns). N = number of x, y, z points).

Step 1: Rename the joints "F, MP, W, E, S" to "0, 1, …" such that Fx becomes x0, Fy becomes y0, etc. Likewise, jF becomes j0, jMP becomes j1 and so on.

Step 2: Add a z coordinate. Start with z0 = 0. For each additional z coordinate, add 0 plus the previous z coordinate. E.g. z1 = z0 + 0, z2 = z1 + 0, etc.

Step 3: Assemble your joint xyz coordinates for n number of joints: j0 = {x0, y0, z0}… jn = {xn, yn, zn}. You should go from 0 to 4.

Step 4: Modify the xy range statement: rename it to XYZrange and add a min, max list so we have min&max x, min&max y, min&max z. XYZrange = {{-range, range}, … }.

Step 5: Modify jointPlot:

jointPlot = ListPointPlot3D[everything the same as before…];

Step 6: MM doesn't have an option to 'connect the dots' in 3D so we have to do it by brute force using Graphics3D[Line[...Nx3 data goes in here…]]. See Appendix 1 for how to plot and style in 3D.

Step 7: Evaluate your code. You should get the same exact limb from the previous tutorial except projected onto the XY plane. Since z components all = 0, the limb is a 2D object shown in 3D.

PART II: Allow 3D angles

A little math before we begin. We need a second angle (phi) to describe position along the z axis. For example if we're modeling a sprawling leg, the femur points medially (XY plane)

and protracts/retracts along angle theta. It also adducts/abducts (or whatever you call it) dorso-ventrally in the XZ plane along angle phi. Just as Cos(theta) and Sin(theta) describe the x and y components of position in the XY plane, Cos(phi) or Sin(phi) will describe the component of position with respect to z. Whether we use Cos(phi) or Sin(phi) depends on how we define phi. By convention, phi starts at the North pole and goes down. So, phi = 0 is pointing up along the z axis. Therefore, the z coordinate should be z = r*Cos(phi). This may or may not be intuitive. Regardless, the way to check is to know that the Cosine function starts at 1 (at 0°) and decreases to 0 (at Pi/2, i.e. 90°). Vice versa for the Sine function. If phi = 0 then the z coordinate should = r. If phi = Pi/2 we know z = 0 because the point would be in the XY plane. It's Cosine because Cos(phi) = 1 at 0 radians and 0 at Pi/2 radians. Let's implement that.

Step 1: Keeping with our earlier pattern, we'll call theta 'q' and phi 'p'. Leave j0 unchanged for now. For j1, make z1 = z0 + l1*Cos(p1) and change the other z2, z3, etc. accordingly. Don't forget to modify the Manipulate sliders with p1, p2, p3, p4.

Step 2: Try it. You can see that there is '3D-ness' to the model, but you'll notice something odd. If you haven't done so already, force the aspect ratio and image size of the plot by adding AspectRatio->1 inside ListPointPlot3D [] and by adding the following arguments inside the Show[] command:

BoxRatios->{1,1,1}
ImageSize->300 (or whatever)

Step 3: Now we're sure something is not right. The lengths of the segments are not constant (lengths MUST be constant because they are inputs in our model!). Use the following command to check distances between joints (e.g. j0 and j1):

EuclideanDistance[j0, j1]

This command uses the Pythagorean theorem (in 3D) to measure straight line distance between two points. You'll notice this changes as your angles change, meaning the model is wrong.

Step 4: IMPORTANT: We must also modify the XY coordinates to account for angle phi. We need to multiply each x and y term by *Sin(phi). If this is obvious to you, great – skip to step 5. If not, try this demonstration: put your pen on your desk (=XY plane), imagining that the base of the pen is the origin and the point of interest is at the tip (length of pen = r) which is exactly where the xy coordinates are. Since phi = Pi/2, no problem. But, as you lift your pen, its xy position moves closer to the origin (watch the shadow). The xy coordinates are projections (shadows) of r onto the XY plane. If phi = 0 there's no shadow and the XY must each = 0. Because the projection of r gets smaller (closer to the origin) as phi moves from Pi/2 we need a correction factor that is 1 when phi = Pi/2 and = 0 when phi = 0. This correction factor = Sin(phi).

Step 5: Modify each x and y coordinate by multiplying *Sin(phi). E.g. x1 = x0 + l1*Cos(phi)*Sin(phi).

Step 6: Let's modify our Manipulate sliders so that they have non-zero initial values giving our limb a default position in the XY plane (i.e. all phi angles p1, p2… = Pi/2).

Change the slider min max statements as follows: {p1, 0, Pi} becomes {{p1, Pi/2}, 0 Pi}. Leave the sliders for q1 through q4 unchanged. Now you'll see the limb works as you might expect. Verify that the segment lengths are constant as limb posture changes.

PART III: Add a fictitious body to our limb model

We will 'mount' the limb onto a rectangular planar body whose length, width and orientation we will manipulate. So, the body will be defined by four points in a plane whose xyz coordinates depend on constants (body length and width) and variables (pitch and yaw angles).

Step 1: Before moving on, I suggest to first clean up the code. Make a custom function (see appendix 2 you don't know how to do this). For example:

xyzJoint[previousJoint_, theta_, phi_, length_]:= some code.

The code will perform all of the math, i.e. previous x value + length*Cos(theta)*Sin(phi) etc. Inputs are xyz from the previous point, theta angle for that point, phi angle, and segment length. The output is a list of {x, y, z} coordinates.

The code inside the function outputs a list of x, y, z coordinates:

{previousJoint[[1]] + length*Cos(theta)*Sin(phi), previousJoint[[2]] etc….}

Step 2: Define your joint positions as single lines of code:

```
jo = {0,0,0};
j1 = xyzJoint[…]
…
j4 = …
```

Test the model to see if it works the same way as before.

Step 2: Build the right side of a rectangular body with length = lbod and width = wbod. Set pitchangle and yawangle = Pi/2 for now. The body will be defined by 4 points, rearR (right rear = origin of the limb ='hip joint'), frontR (right front), frontL, rearL. Remembering your Trig from above, start by defining the right side points based on the yawangle (ignoring pitch for now – i.e. z = 0). Define frontRx, frontRy and frontRz separately then put them together in a list to make an xyz point (remember, rearR is defined as o, o, o):

```
rearRx = …;
rearRy = …;
rearRz = …;
rearR = {rearRx, rearRy, rearRz};
```

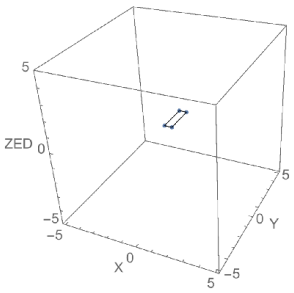
Also define the front, remembering to account for lbod and yawangle. Use the same Trig approach you learned in previous exercise.

```
frontRx = rearRx + …;
```

frontRy = rearRy + …;
frontRz = rearRz + …;
frontR = …;

Step 3: Build the left side of the body. This is a bit trickier. If there was no change in yaw, we could simply say frontLx = frontRx – "left displacement". The left displacement would be wbod. But this will not hold once the body rotates because the amount of left displacement would vary depending on yawangle. For example, if the body is turned 90 ° pointing sideways along the x axis, the "left displacement" would actually a displacement in y, not in x. So, we know the "left displacement" term has to be a function of yawangle. Here's how to think about it: The yaw angle rotates the body axis. Subsequently, we need to rotate another point by an additional 90 ° such that it is always perpendicular to the body axis:

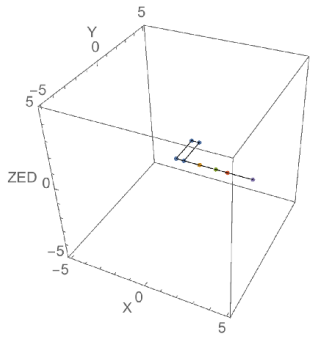
frontLx = frontRx + wbod*Cos[yawangle + Pi/2];
frontLy = frontRy + ….;
frontLz = …;
frontL = …



Step 4: Draw the body using the appropriate plotting commands. Put the entire code cell into a Manipulate[] command and interactively vary the yawangle.

Step 5: Now modify your xyz coordinates to account for pitchangle. Use what you learned in PART II above. You only need to modify the front points because the rear points are anchored.

Interactively play with pitchangle and yawangle. Their default orientations should be Pi/2 for each so the body lies along the y axis (cranio-caudal) in the XY plane. Again, verify point distances (e.g. distance between front and rear points should = lbody, etc.).



Step 6: 'Glue' the body to the limb by copying all of your limb code and combining it into a single Manipulate cell together with the body code. Make sure the limb origin is the same as the rearR point on the body. Make sure the constants (lbod, wbod, l1, l2, l3, l4, range, etc.) are evaluated above the Manipulate cell. To help organize the code (and set the model up for future manipulations) assemble the body points in a single variable:

pointsBODYg={rearL,rearR,frontR,frontL,rearL};
The 'g' stands for 'in the ground reference frame'.

Do the same for the leg points:
pointsLEGg = {j0, j1, j2, j3, j4}; We will refer to these variables in later tutorials.

Interactively play with the joint angles as well as the body orientation.

Step 7: Spend no more than 5 minutes wrestling the joint angles into positions that look vaguely like a sprawled limb (like a lizard foot or something). j0 would be the hip. You'll notice that this is not easy. It may help to make 2D plots of planar projections (e.g. the limb in the XZ plane for a rear view). To do this, just extract x and z points. Two tricks you will learn: "All" means take all elements of a particular dimension in a list. {m, n} means take elements m and n.

```
JOINTSxz = JOINTS[[All, All, {1,3}]];
graphXZ = ListPlot[JOINTSxz, Joined->True];
```

This means, take elements 1 and 3 (x and z coordinates) in the innermost dimension. Take all of the rest. You can combine 3D and 2D plots by naming them (e.g. graphXYZ and graphXZ) then combining them using GraphicsColumn[]:

```
GraphicsColumn[{graphXYZ, graphXZ}]
```

Even with multiple views, this is a non-awesome way of controlling the posture of a model limb. The better way is the following:

a. Rotate groups of points in bulk using rotation matrices (next tutorial)
b. Rotate groups of points with respect to 'local' anatomical axes or planes using reference frames (tutorial after next).

Step 8: To make your limb look a bit more like an animal limb, enter the following values for limb segment lengths:

```
l1=1;l2=1;l3=.7;l4=1.5;
```

Also, enter some initial joint angles – collect them in a list called Qi

```
Qi={0.9,2.3,-0.8,1.6,0.5,2.43,1.4,1.5};
```

Next, enter these initial values for each q slider at the end of the Manipulate command:

…{q1, 0, Pi} becomes …{{q1, Qi[[1]]}, 0, Pi} so that the q1 slider starts with the default value of our first initial angle. Modify the rest of the sliders accordingly.

Finally, for all of the theta (q) angles, make the min, max span between -Pi/2 and Pi/2. For all of the phi (p) angles, make the min, max span between 0 and Pi.

Now, try to play with the limb again. Still difficult to control, but at least it looks more life-like now. We will modify this final version of the model in a later tutorial.

Step 9: In preparation for future tutorials, store the xyz positions of the leg joints in pointsLEGg. Re-evaluate the Manipulate cell to restore the sliders to their default values then evaluate the following line at the bottom of your code in a new cell, leaving off the semi-colon:

```
pointsLEGg (*← no semi-colon*)
```

Save the notebook - we'll copy-paste the pointsLEGg values for use in later tutorials.

Appendix 1: 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. You can also set the ImageSize→some number within the Show[] command.

Appendix 2: 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.

Links

2D polar coordinates:

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

3D spherical polar coordinates:

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