TUTORIAL 3 Trigonometry B (building a 2D morphological model)
This tutorial continues from tutorial 2. Please complete exercise 5 (the final exercise) in tutorial 2.

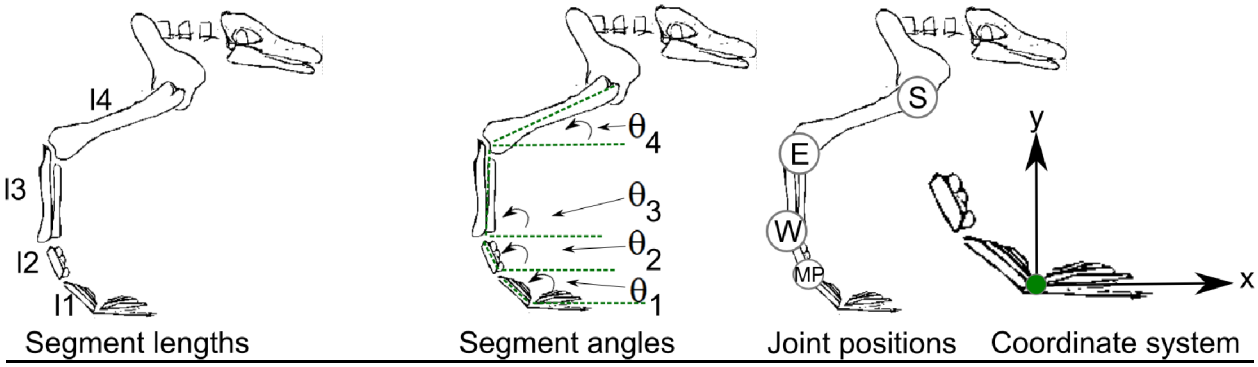
Why do we care? This might not be obvious until later tutorials, but learning to 'draw' anatomy using equations will open doors hopefully leading to biological insights (hypotheses to be tested). At the very least, simple modeling can build intuition about how moving parts influence one another. For example in a limb, how does an increase in knee angle influence hip height from the ground? It's complicated because this depends on the positions/actions of the other joints. So, the easiest thing is to make a model to 'play' with the joint positions, etc. This can be done with cardboard and paper clips, but I hope to convince you that it's just as easy to 'build' it mathematically and 'play' with it using clever interactive software. Then, we can change the morphology just by changing values of parameters in our model (rather than cutting out new pieces of cardboard). And we can expand it to 3D with trivial adjustments. As we'll see in later tutorials when we animate our model, mathematical visualization can be a quick way of 'sanity checking' your model output.

Aims of this tutorial:
*** Develop a limb morphological model 'stick figure'**

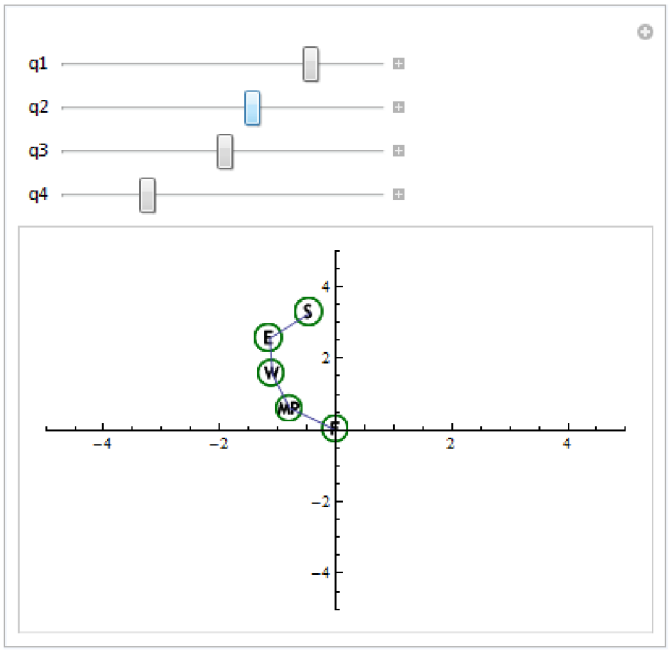
Skills required prior to this tutorial: Basic Mathematica workflow. Basic 2D plotting (ListPlot, etc.). Array/list structure and use of 'curly brackets'.

Skills gained from this tutorial:
*** Mathematica '*MM*' software:**
-Make interactive plots using the Manipulate[] command
-Learn how to style and customize graphs
-Import images into *MM* and use them in plots
-How to overlay plots using Show[]

As in the previous tutorial, we will consider the generic vertebrate limb drawn below:



The model we will 'build' in *MM* will have interactive sliders to 'play' with the posture an anatomy to generate hypotheses and build our intuition about geometry, eventually, anatomical constraints (below):



TUTORIAL OUTLINE:

PART I: Set up the basic mathematical model and plot the result
PART II: Beautify the plot so it looks vaguely like a limb (like the diagram above)
PART III: Transform the segment angles to joint angles

Later tutorials: - Apply anatomical and morphological constraints (hard); Animate the model

Before we begin, you can always check your code with the example code included in this tutorial package.

PART I: model setup

What we're aiming for is a list of x-y points (i.e. a list of lists where each element is {x, y}) telling the coordinates of the foot 'F', 'MP', wrist 'W', elbow 'E' and shoulder 'S' joints. We'll plot these as if they are regular data points.

Note: length segments will have the variable names l1, l2, etc. Segment angles will have the variable names q1, q2 (for θ1, θ2, etc to avoid messing with symbol fonts in *MM*)

CODE OUTLINE:

```
(*assign values of 1 for each length segment, for now*)
l1=1;l2=1;l3=1, etc.

(*PART A: define coordinate variables: equations for each x, y coordinate from Ex. 5 of the previous tutorial*)
Fx = … ;
Fy = … ;
MPx = … ;
MPy = … ;
…
…
```

(*PART B: define joint variables: assemble each x,y coordinate as a list with "j" standing for "joint"*)
jF = {Fx, Fy};
jMP = {MPx, MPy};
…
…
JOINTS = {{jF}, {jMP}, ….}

Note that JOINTS has seemingly unnecessary extra curly brackets around each list element. Why do this, given that each joint jF, jMP, etc., is already grouped in x-y pairs? We do this because *MM* will later interpret the extra outer { }'s as plot separators. In other words, *MM* will think each joint is a separate plot (i.e. group of points) enabling you to style each point independently. This will be useful later.

As you know, each point position depends on segment angles (none of which we've assigned values to) and lengths which are set = 1 for now. Consequently, *MM* will assemble our JOINTS list of points as a list of equations. This is useful, because we will later manipulate the values in the equations to 'play' with our model.

Step 1: Copy-paste your equations (or modify your existing notebook) from Ex. 5 of the previous tutorial. Include each equation as a list enclosed by one notebook cell (so we can evaluate it in bulk). See outline, part A

Step 2: Group each x, y pair. See outline, part B

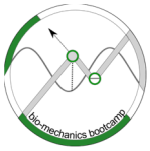
Step 3: Define the variable JOINTS and fill it with xy equations:
JOINTS = {{jF}, …}
Leave the semicolon off or now and evaluate the cell to look at the list of equations. Each element should be a pair of equations with sin/cos terms which gets increasingly ugly from distal to proximal joints.

Step 4: For the *MM* interactive magic we'll use the Manipulate[] command. If you're unfamiliar with this command, see Appendix 1. In a new cell, write the following structure:

Manipulate[(*begin manipulate*)
…
, {q1,0,Pi}, {q2,0,Pi}, {q3,0,Pi}, {q4,0,Pi}] (*end manipulate*)
Your code will go where '…' is.
For readability, you may want to comment the beginning and end brackets of the Manipulate command because you will:

Step 5: Copy-paste all of your code so far inside the manipulate brackets (where the '…' is). The first line should be the code defining the segment lengths and the last line should be the definition of JOINTS. Make sure JOINTS has a semicolon after it.

Step 6: Now, we want to plot the points. Since they are discrete points, we'll use ListPlot (see video tutorial at the end of this document if Plot vs ListPlot still mystifies you).
Define a variable called jointPlot to plot the joints as points in space. Write the following line directly after the JOINTS definition.



jointPlot = ListPlot[JOINTS];

And, we need to connect the joints to draw the joint segments. The way to do this is to use a separate plot (which can be styled and customized later) and overlay the two using the Show[] command. Annoyingly, we should first re-structure our JOINTS array to be a list of x-y pairs as opposed to a list of lists of xy pairs. I.e. option 1: {{x1,y1}, {x2, y2} …} vs. option 2: {{{x1,y1}}, {{x2, y2}} …} with the extra { }'s we had before. How do we flatten option 2 to become option 1? Use Flatten[]. This command is the 'un-list' which will remove levels of list strictures (i.e. reduce dimensions). E.g. Flatten[JOINTS], by default, will produce a long 1dimensional list of {x1,y1,x2,y2….} which is useless. But, if you add a 1, it will flatten by 1 dimension level:

SEGMENTS = Flatten[JOINTS, 1];(*this turns JOINTS into a list of lists*)

Recap, in case you're lost about when/when not to use extra curly brackets: JOINTS has extra curly brackets to eventually tell *MM* to plot each joint as separate plots. SEGMENTS lacks the extra curly brackets, meaning that *MM* will plot SEGMENTS as one continuous connected curve.

Since the segments need to be lines, use ListLinePlot[] to plot lines instead of points – it works the same way as ListPlot[].

segmentPlot = ListLinePlot[SEGMENTS];
Show[jointPlot, segmentPlot] (*THIS IS THE LAST LINE – no semi-colon needed!*)

Step 7: Save your work then execute the Manipulate cell. Hopefully, you'll see a rudimentary cartoon stick figure which you can wrestle into the same position as my drawing at the top of this tutorial.

This is cool, but fairly useless because of various *MM* annoyances (e.g., the scales of the plots change dynamically, so the 'shape' of the limb constantly distorts). And the markers look lame. Certainly not something you'd want to put into a presentation or lecture. In PART II of this tutorial, we'll beautify the graphics and give you the tools to customize it any way you like. In PART III, we'll modify the equations to make it more biologically useful.

PART II: 'beautification' and customization

Step 1: We want to prevent *MM* from autoscaling the axes. Do this by defining the plot range. Somewhere at the top of your code (before the Manipulate[]), define a variable called xyrange. It is a list of lists: {{xmin, xmax}, {ymin, ymax}}. For a morphological model, visual distortion will occur if the axes are different lengths, so let's keep xmin = ymin; xmax=ymax.

range = 5; (*for now*)
xyrange = {{-range, range}, {-range, range}};

This keeps the bounds between -5 +5 for x and y axes.
….

As an argument in each of your ListPlot commands, type 'PlotRange-> xyrange' to modify your code:

jointPlot = ListPlot[JOINTS, PlotRange -> xyrange]

Step 2: Your graph still will be distorted, so we have to force *MM* to maintain a fixed aspect ratio for the plot image: Add another argument to your ListPlot commands "AspectRatio -> 1" All arguments are separated by commas. The order of arguments doesn't matter:

jointPlot = ListPlot[JOINTS, PlotRange -> xyrange, AspectRatio -> 1]

Evaluate the cell and you'll see a stick figure that is starting to look like my drawing. You can adjust the segment angles as you wish.

Note, these arguments can also be specified for both plots simultaneously within the Show command.

Step 3: It would be nice to use our own joint markers. Given that we've defined JOINTS as a List of Lists of Lists, *MM* nicely will allow you input a list of joint markers that it will plot for each joint. E.g. try:

jointPlot = ListPlot[JOINTS, …, PlotMarkers -> {"foot", "mp", "W", "E", "S"}];

This literally writes the text at each joint, which looks ridiculous in this case (but can be powerfully useful occasionally). Each element in the PlotMarkers list can be whatever you want, including an image. I've provided bitmap images of joint markers in this tutorial folder (you can draw your own if you want, any image format is acceptable). Watch this: to import, just drag the foot marker file into an empty area of your notebook. The foot marker image will appear. Click the image and resize it so it's pretty small (roughly the size of 15pt text). Now copy that resized image into your PlotMarkers array (i.e. replace '"foot"' including "" by the image). Do this for all plot markers:

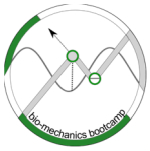
PlotMarkers → {F, MP, W, E, S},

Step 4: Now, on your own, play with modifying the line thickness and color of the joint segments. Go to *MM* help and type PlotStyle and see the wonderful ways that you can customize. Note: the lines will behave as a single plot (i.e. all segments will look the same) unless you partition SEGMENTS as JOINTS. You'll figure it out.

PART III: Transforming segment angles into joint angles

In biomechanics, limb angles are usually defined as the angle between adjacent segments, not by the angle of each segment with respect to horizontal. This may seem like a subtle difference. But if we want to model muscle shortening, muscles don't care about where the horizontal is or whether the femur is pointing up or down. Rather, muscle length changes are tied to the relative positions of the segments. Therefore, we now want to define *joint angles* as they are often reported in biomechanics papers: θfoot, θmp, θw, θe, θs. Again, let's use "q" to mean "θ". So, how do we convert q1… q4 to qfoot … qs?

Step 1: Copy-paste your previous code into a new cell or new notebook. **DELETE THE MANIPULATE FRAME** below your previous code (not the code, just the interactive plot image) – otherwise you'll have dueling manipulate frames trying to update each other and *MM* will crash.



Step 2: The foot joint is easy because it's always fixed to the ground. We'll leave it as is. For the other joints, we'll need to define which way they open. This is where our model obeys the laws of biology. For example, many vertebrate legs have a knee angle that opens 'backward' towards the tail. We'll define an 'anatomical sign' for each joint which dictates the whether the joint points towards the head (+, to the right) or tail (-, to the left). For the current exercise, all signs = 1 and we don't apply a sign to the foot angle.

sMP = 1; sW = 1; sE = 1;

Step 3: Rename the foot angle:
qfoot = q1;

Step 4: For each joint (other than the foot), we have to add the previous joint angle. Mathematically, a change in a distal joint will influence all of the joints proximal to that. This allows each segment to be relative to adjacent segments, as opposed to the horizontal plane. In other words, joint segments define the limb's posture and are independent of how the entire limb is oriented – if the limb is on its side or upside down, the joint angles will not change.

To convert a segment angle (q#) to a joint angle (qjoint), multiply by the anatomical sign and add the previous joint just distal to the current joint:

qmp = sMP*q2 + qfoot;
etc…

As a final adjustment, we would prefer the joint angles to be defined such that joint angle = 0 means that the joint is closed (i.e. the two adjacent segments are completely folded over on one another). To achieve this, simply subtract 180 degrees (actually Pi, because we're working in radians). Modify your previous equations:

qmp = sMP*(q2-Pi) + qfoot;
etc…

Step 5: If you had trouble, download the code (either as *MM* notebook or as a PDF). Play interactively not only with the joint angles, but also with the segment lengths. Appreciate how a single joint's action on the limb depends on the limb's overall posture. E.g. the wrist angle can 'push' the elbow dorso-ventrally or cranio-caudally, depending on whether the foot angle is flexed or extended.

This is perhaps not the most biologically inspiring model, but it is the starting point for future models that can be used to generate hypotheses. We can also 'map' muscles onto this skeletal model in future tutorials.

Appendix 1: *MM*'s Manipulate command

This is quite a can of worms, so we won't cover every nuance - you can explore it on your own. Basically, Manipulate is an interactive frame with sliders representing variables that you can change interactively. It is a 'Dynamic' object, meaning it updates itself automatically. Manipulate[A, B] means update code A according to user input B. Specifically, B can be any number of iterator structures {variable, min, max, increment}. For example,

Manipulate[Plot[a*x, {x, 0, 10}], {a, 3, 5, 0.1}] will produce a line whose slope can be interactively adjusted from 3 to 5 in increments of 0.1. Increment is optional, so it goes by super small increments by default if left unassigned. There can be as many variables as you want … $a*x^b + c$, {a, 3, 5}, {b, 0, 1}, {c, -10, 10} etc. A new interactive slider will appear each time a new iterator structure is added. Finally, argument 'A' which is $a*x^b + c$ in this case can actually be computed from any number of lines of code. E.g.

```
Manipulate[
y = Pi;
z = y*1.2;
h = 17*Sin[z];
d = h^3;
a*x^b + c + d
, {a, 3, 5}, {b, 0, 1}, {c, -10, 10}]
```

For a clean look, I prefer to put the iterators on their own line, as above.

IMPORTANT NOTES 1: With multiple-line structures like this within Manipulate, the last line should not include a semi-colon (otherwise the output will be blank). Even more important, every previous line MUST have a semi-colon, or *MM* will get very angry.

2 :**ALWAYS** save your work before executing a Manipulate command. Manipulate will dynamically update constantly. If there are multiple Manipulates open which use the same variables, they will try to update each other in a nasty loop and *MM* crashes. Easy fix: Delete the cell containing whatever Manipulate box you're not currently using (don't delete the code – just delete the cell). Harder fix (but better programming): Encase your code using 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.

```
Manipulate[
Module[{qfoot,qmp,qw,qe,qs,l1,l2,l3,l4},
all of the code
…
](*end module*)
, {q1,0,Pi},{q2,0,Pi},{q3,0,Pi},{q4,0,Pi}] (*end manipulate*)
```

Useful links:

hyperphysics trigonometry section (http://hyperphysics.phy-astr.gsu.edu/hbase/trig.html)

Mathematica Video Tutorials:

Welcome to Mathematica (6:38):
http://youtu.be/UTM_FDBPEjc

Mathematica: Plot vs. ListPlot: Two ways to plot curves (5:11)
https://www.youtube.com/watch?v=P6To6lx_bLc