TUTORIAL 5 - 2D and 3D geometry A (coordinate transformations, translations, rotations)

Why do we care? Long story short - if you study complex geometry or motion, you'll want to know about how to manipulate 3D data. Whether the goal is to model biological motion or interpret and analyze biological motion, it's often necessary to do bulk manipulations on 3D data (i.e. a cloud of x, y, z point coordinates). For example, 3D joint motions in ostrich legs were used to establish joint-based 'local' coordinate systems to quantify joint motion and orientation with respect to other joints and bone segments. This was found using mathematical transformations of 3D points to derive 'local' coordinate systems at each leg joint [1, 2]. This tutorial opens a vast can of worms and we'll give you the basic tools upon which you can later build sophisticated knowledge and facility. There are countless applications for knowing how to transform coordinates (i.e. translate and/or rotate). This tutorial will give you the necessary mathematica skills. There are also many web resources already available for understanding the arithmetic underlying geometrical transformations. For example, we'll skip over the mechanics of matrix multiplication because there are already many web tutorials for that and most technical software (Mathematica, Matlab, Python, R, Labview) have built-in functions that will perform matrix operations.

Aims of this tutorial: *Learn how to set up a matrix multiplications (i.e. how to organize geometry data for the math to work). *Learn 2D and 3D tools for translating and rotating geometric shapes

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[].

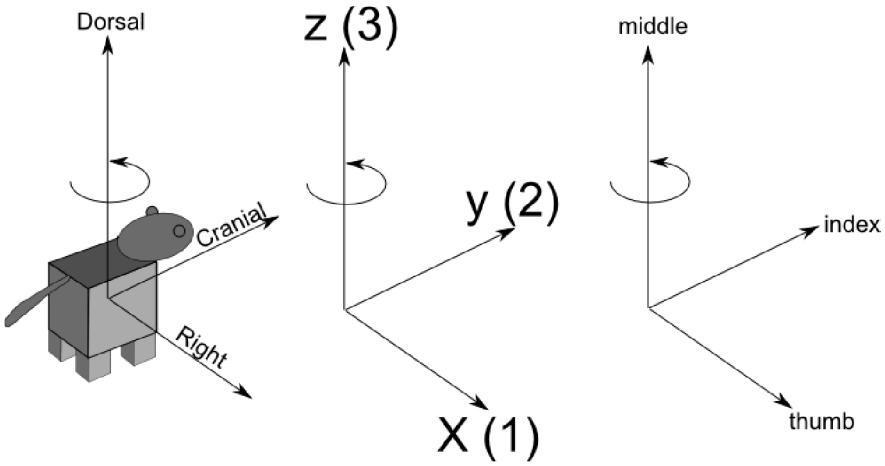
Skills gained from this tutorial: * Mathematica '*MM*' software:
-Use of RotationTransformation and TranslationTransformation for easily transforming a matrix of {x,y} or {x,y,z} points.

Background and theory: *MM* and other software will perform transformations for you without your having to know the actual math. In case you don't have access to *MM*, you can do all of the transformations you need, but you may have to write matrix equations yourself. This is not hard once you know the vocabulary which will then guide you to the appropriate web resources (of which there are many).

■ **Coordinate system:** Think of the axes of a Cartesian coordinate system as vectors pointing away from the origin in 3 different directions. As will be reiterated again and again: a vector contains information about direction using a list of numbers (2 numbers for 2D and 3 for 3D). If the origin of our coordinate system is {0, 0, 0}, then a vector {1, 0, 0} says "to arrive at point 1, 0, 0, move 1 unit along the x axis, but don't move along the other axes." By convention, unit vectors (vectors of magnitude = 1) are used to define the 3 axis as vectors.

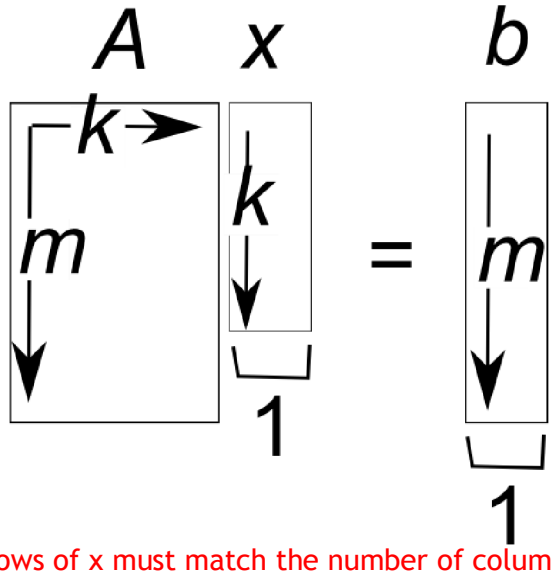
x = {1, 0, 0}
y = {0, 1, 0}
z = {0, 0, 1}

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.



■**Matrix multiplication:** If you have no idea what a matrix is, please google it now. This is a huge topic, but what you need to know is this: Think of an *m x k* matrix as a spreadsheet where *m* is the number of rows down and *k* is the number of columns across. Computers are ninjas at efficiently doing manipulations on matrices. Particularly, matrix.matrix multiplication or matrix.vector multiplication can be performed quickly on large data sets (matrices) in *MM*, Matlab, Python, Labview, whatever. Therefore, manipulating (e.g. 3D manipulating) large data sets is conventionally done using a matrix.vector multiplication where *A* is a matrix, *x* is a vector and the product *b* is a also a vector and '.' Denotes the dot product…

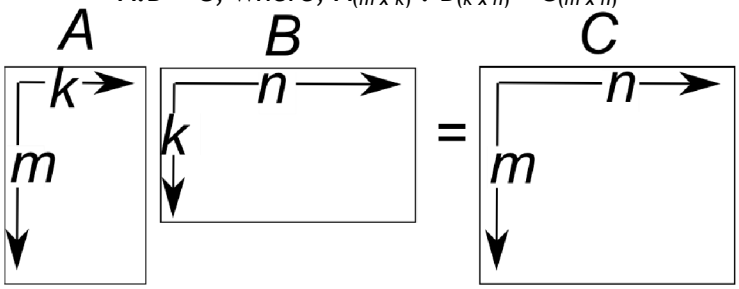
$$A_{(m \times k)} . x_{(k \times 1)} = b_{(m \times 1)}$$



Note that the number of rows of x must match the number of columns of A.

OR a matrix.matrix multiplication where *A* and *B* are matrices:

$$A.B = C, \text{ where, } A_{(m \times k)} \cdot B_{(k \times n)} = C_{(m \times n)}$$



Note that the number of rows of the final product *C* is determined by the number of rows of the first matrix (*A*) and the number of columns of *C* is determined by the number of columns of the second matrix (*B*). Therefore, order matters. A.B ≠ B.A.

■ **Rotation:** This is when you take a point in 2D {x, y} or 3D {x, y, z} and rotate it about some reference. In the most basic case, rotations are performed with reference to the coordinate system origin about one of the x, y, or z axes. For example, a 90 degree counterclockwise 2D rotation of point {2, 0, 0} about the z axis will swivel the point about the origin like hands on a clock moving backwards arriving at {0, 2, 0} (i.e. from 3 o'clock to noon). Or in another example, for a bird whose beak is pointed down the y axis, the rotations of the up/down wing flapping can be described by rotation about the y-axis. If the air flows along the y axis, the bird can change its wings' angle of attack (angle that the air meets the wing) by rotation about the x-axis (about the span of the wing. To turn, it swivels its body about the z-axis.

In 2D you can rotate points 'easily' using what you learned in previous Trig tutorials by solving for each coordinate independently. However, if we need to rotate many points by the same angle, it's much more computationally efficient to use a ***rotation matrix*** = *R* which is a 3 x 3 matrix of mathematical terms. The terms are a function of sine and cosine functions as well as the angle of rotation. **Don't fear**: Mathematica has built in rotation matrices for any kind of rotation we can dream of. So we don't have to deal with tricky linear algebra. Here is a brief description for those who may eventually need to write their own *R* matrices (i.e. if you don't have *MM*). The elements inside *R* are dependent on the axis of rotation and the point about which to rotate. Standard forms of *R* come in three main types (see the Rotation Matrix article on Wikipedia which is excellent). *R*x rotates a point around the x axis with respect to the origin. Ditto for *R*y and *R*z for the y and z axes. You can also rotate about an arbitrary axis (i.e. vector) going through some reference point that is not the origin. In this case, *R* is much more complicated (see Appendix 3 for details if you don't have Mathematica).

To rotate a point {x, y, z} about the origin of the coordinate system, perform the following matrix.vector dot product:

$$R.p = \{x', y', z'\}$$

where {x', y', z'} indicate that xyz have been transformed (in this case by rotation). This matrix.vector multiplication causes the appropriate terms to be multiplied, added and subtracted such that x, y, z get rotated with respect to a chosen axis. You can also do this for a matrix of *n* number of xyz points, $P_{(3 \times n)}$ = {{x1, y1, z1}, {x2, y2, z2}, {x3, y3, z3}… {xn, yn, zn}}. Note: If *P* has the wrong shape (n x 3) it won't work - you'll have to transpose *P*.

$$R.P = = \{\{x1', y1', z1'\}, \{x2', y2', z2'\}, \{x3', y3', z3'\}\ldots \{xn', yn', zn'\}\}$$

■ **Translation:** This is conceptually much simpler. A translation is just a linear straight line shift in a point (or set of points) without changing its orientation. This can be done easily by addition. I.e. for every xyz point in *P* (as defined above), you can add {xoffset, yoffset, zoffset} which will shift the entire point set by a fixed translation. Alternatively, you can multiply your point by the translation matrix *T*. Unlike *R*, *T* is a 4 x 4 matrix (long story) so you must append a 1 on the end of each x, y, z, point (then remove it later):

$$T_{(4x4)}.p = \{x', y', z', 1\}$$

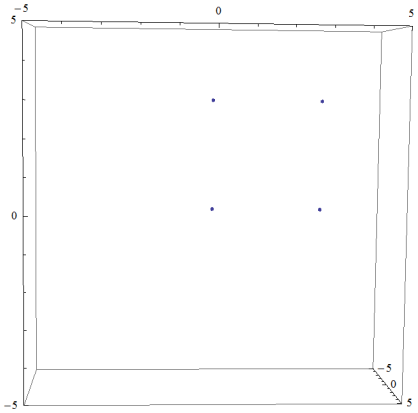
where {x', y', z', 1} indicate that xyz have been transformed (in this case by translation). In practice, for multiple x , y, z points, you have to write code to do the following:

-append 1 onto each x, y, z point
-Translate the point by 'dotting' it with *T*.
-Remove the 1
-Repeat for each additional data point

In *MM*, we don't have to do all of these steps – *MM* has an internal function which can transform a matrix of points similar to rotation above.

Exercise 1: 2D and 3D translation: In ***MM*,** Draw a 2D box and interactively translate it. Check to make sure our transformations didn't actually distort the object.

Step 1: Define a square shape (dimension = 3) as matrix of x, y, z points. Even though this is a 2D exercise, we'll use 3D points to make it easier to modify our code later. All z points = 0 for now, so think of all points lying in the plane z = 0.



```
P1 = {0, 0, 0};
P2 = {3, 0, 0};
P3 = {3, 3, 0};
P4 = {0, 3, 0};

SQUARE = {P1, P2, P3, P4}; (*arrange the points as a list of lists of xyz points*)
```

Step 2: Plot the square. We'll use the 3D equivalent of 'ListPlot' called ListPointPlot[]. Define the plot range so it remains fixed. (E.g. a xyz min/max = +/- 5). You can interactively and intuitively handle the plot in order to rotate the viewing angle:

Step 3: From now on, think of your points as an n x 3 matrix where *n* is the number of points and 3 is for x, y, z. For easy viewing *MM* will write a table as a matrix:
MatrixForm[SQUARE]
MatrixForm[] is only for convenience of display and visual checking. Keep your data always in table form with the curly brackets, etc. Don't try to do operations on table data in the MatrixForm.

Step 4: Let's define a translation function that we'll call 'transFun'. This will act as a function that will translate an entire matrix of points, magically offsetting the linear position

of your points by some arbitrary offset in x, y, or z directions. For example, a translation (offset) of {0, 1, 0} will translate your square 1 unit in the y direction. There is some hidden *MM* magic here, so follow the steps and I'll explain as we proceed.

```
transFun = TranslationTransform[{offsetx, offsety, offsetz}];
```

TranslationTransform is the magical *MM* command. We filled the command arguments as a vector of offsets. Remember, a vector is just a single row or a column of a matrix. We arbitrarily named the offset in the x direction 'offsetx', and likewise for y and z. You could name it anything you want.
If you like linear algebra, you can look at the transFun output, but otherwise, just leave the semicolon.

Step 5: To apply the translation, simply put a single point into it. To see how it works, assign values to offsetx, y and z. Then see what happens when we translate point {2, 2, 2} by an x offset of 1:

```
offsetx =1; offsety = 0; offsetz = 0;
transFun[{2, 2, 2}]
```

transFun will also work on a matrix of points. So, replace {2, 2, 2} with SQUARE points. Convert the output to MatrixForm if it helps. You'll see how it shifts ALL points, adding +1 to all x.

In a new cell, make a manipulate command and copy paste your definition of transFun. Make sure there's a semicolon. If you don't know or forgot about the Manipulate command, see Appendix 1. Then 3D plot the transformed square (no semicolon this time). I recommend naming your translated points SQUAREt.

```
Manipulate[
transFun = TranslationTransform[{offsetx, offsety, 0}];
SQUAREt = transFun[SQUARE];
ListPointPlot3D[SQUAREt]
,{offsetx, -1, 1},{offsety, -1, 1}]
```

Make sure you pre-define the plot range as before.

Step 6: Let's check to make sure the square is still a square. The 2D distance between any two points:

$$distance = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

Likewise, the 3D distance is:

$$distance = \sqrt{(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2}$$

Use the *MM* function EuclideanDistance[] to apply the above formula to check the distance between two points (e.g. the distance between P1 and P2 should be 3).

Step 7: Finally, extend this to 3D by making your z coordinate non-zero. Add another variable offsetz.

Exercise 2: 2D rotation about the origin: In ***MM*,** with your 2D box from above, learn how to rotate it about the coordinate system's origin. Check to make sure our transformations didn't actually distort the object.

Rotating a set of points about the coordinate system's origin is mathematically simple, but rotating about arbitrary axes becomes ugly. Fortunately, using *MM*, we don't have to worry about the complicated linear algebra. For a derivation of the math, see the link at the bottom.

Step 1: Analogous to the TranslationTransform, write a RotationTransform function and call it rotFun. For good times, highlight RotationTransform and press F1 to open the help file and you'll see the many ways in which this function can be used. For this exercise, we'll rotate about a point 'p' about the z axis (to keep it in the 2D plane). Here's the syntax:

RotationTransform[angle(radians), axis, point (2d or 3d)].

The angle argument is simple, but we need to define the axis of rotation and the center of rotation. In this case our point is the origin, so the third argument is {0, 0, 0}. 'Axis' is more tricky. This is a vector defining the direction of our axis. It will have two elements for the 2D case and 3 for the 3D case. For example, a vector {4, 3} means start at a point (any point), go right 4 and forward 3 (according to our coordinate system). The combination of a vector and a point means that *MM* has enough information to know the axis of rotation. In 3D, {4, 3, -2} means 4 right, 3 forward and 2 down from any point. In our simple case, we're rotating about our coordinate system – in this case, the z axis. To define this vector mathematically, we need to point any arbitrary distance along the z axis and only along the z axis. Thus, we know {0, 0, something} because there's no components pointing in either x or y. We'll pick '1' (but it could be any positive number –(by the way, what would happen if you use -1 instead of 1?). So our axis is {0, 0, 1}.

rotFun[q, {0, 0, 1}, {0, 0, 0}];

Our angle will be q, in radians.

Step 2: Rotate the SQUARE points.

SQUAREr = rotFun[q, {0, 0, 1}, {0, 0, 0}];

Copy your translation code (step 5, Ex. 1). Replace the transformation code with a rotation transform above and replace the xyz offsets with an angle slider that goes from {q, 0, 2Pi}.

Step 3: Try various things to test your intuition – i.e. change the center of rotation or change the axis. OR, turn it into a 3D rotation by altering the axis (e.g. a vector not parallel with any coordinate system axis).

Step 4: As before, make sure the transformations didn't affect the square shape. Here is a neat trick (assuming you've defined distanceFun):

```
check = distanceFun[SQUAREr[[1]], SQUAREr[[2]] ];(*check all the points this way*)
…
ListPointPlot3D[… , PlotLabel -> check]
```

Exercise 3: 3D rotations: Sequential rotations in two planar angles.

This is quite a can of worms. A point can arrive at a new location via a series of rotations and translations in multiple ways. Order of rotation matters. We'll not dive too deep in this exercise, but deep enough just to build the tools so you can explore yourself. Often, 3D biomechanics data is reported in two or more angles. For example, it's often useful to refer to an angle as a 3D construction of two angles with respect to two different anatomical planes [ref needed]. In this example, we'll work through how to constrain an angle to a plane and rotate the plane.

An analogy: Oars in a row boat must rotate forward→backward while in the water. In addition to forward/backward, there must be an up/down component to the rowing stroke as the oar blades must be lifted up/down to go out/in the water. The femur of a walking animal may also follow a 'rowing' motion as the femur protracts forward and retracts backward as it simultaneously rotates dorsal and ventral to lift the leg for swing vs. stance phases. In our coordinate system, the angle that controls dorso-ventral motions (for lifting/lowering the foot) will be constrained to rotate about the y axis (like a hinge). The forward backward angle will be constrained to rotate within the plane defined by the y axis and the femur.

Step 1: As before, 'draw' a 2D rectangle "RECT" as x, y, z coordinates. Make the rectangle 1 x 5 units long with the short side anchored to the y axis and constrained to the xy plane (i.e. it is horizontal). I.e., the rectangle (which is our femur) will be sticking horizontally out to the right. Define the posterior-proximal corner of the femur as the origin {0, 0, 0}.

Step 2: Make a rotation function 'rotFun1' for the first rotation. This is the dorso-ventral angle (angle1) about the y axis. This is similar to the previous exercise. Apply the rotation to form a set of rotated points RECTr1. If you plot this, you'll see that as you change angle1, the femur flaps up and down like a bird wing.

Step 3: The second rotation is constrained to an imaginary plane that rotates along with the femur. I.e. it is a plane anchored at the y-axis who's angle is defined by the angle1 of the femur. To define the plane, we need two 3D vectors that share the same plane. In our example, the femur rectangle itself defines a plane. We can take any three femur points (described as a pair of vectors connected at one end) to define the plane. As you know, a 3D vector tells how to get from one point to another. So, subtracting two points reveals the vector. Define 'femurPlaneVector1' by subtracting one point on the rectangle from another. Do the same for 'femurPlaneVector2', but choose another point (as long as the two vectors share a point). Make sure you define your plane based on RECTr1 which is already rotated from angle1. Remember, you want two *connected* vectors.

Step 4: The hard part is already done. RotationTransform[] can operate in many ways, depending on the input you give. Before, we input [angle1, axis of rotation, point]. This time, we'll use a different syntax: RotationTransform[angle2, {vector1, vector2}] where vector1 and vector2 are lists of x,y,z components of two connected vectors defining the new rotation plane. Use femurPlaneVector1 and femurPlaneVector2 as your vectors to define a new variable RECTr2 for the points that have undergone a second rotation. Put everything in a manipulate frame and you're done. Make sure to check to make sure your rectangle doesn't distort. Also rotate the interactive 3D plot to view various 2D projections (by aligning it squarely as if you're looking down one axis) to convince yourself that the femur is indeed constrained to a plane defined by angle1.

Exercise 4: One last challenge. Create your own custom function that rotates about an axis and center of rotation of your choice. If you don't know how to make a custom function, see Appendix 2 and practice first.

Try: Make a custom function that performs both a translation and a rotation in a prescribed order about a prescribed axis. E.g. given POINTSdata = {{x1, y1, z1}, {{x2, y2, z2}, …}, yourcustomTransform[angle, {xyz offsets}, POINTSdata] = {{x1', y1', z1'}, {{x2', y2', z2'}, …}.

Try: Make a custom function that performs a series of rotations in a certain order. Convince yourself that rotation order matters. Occasionally experimental data needs to be transformed in bulk for many trials. In this case, making a unique function to perform the same specific set of operations is helpful.

Try: Make a custom function encapsulating the sequential angle rotation in Ex. 3. twoAnglePlanarRot[angle1, angle2, POINTSdata]

Appendix 1: *MM*'s Manipulate command

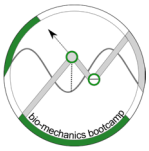
This is quite a can of worms, so we won't cover every nuance - you can explore it on your own. Basically, Manipulate is an interactive frame with sliders representing variables that you can change interactively. It is a 'Dynamic' object, meaning it updates itself automatically. Manipulate[A, B] means 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*)

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.

Appendix 3: Rotations about an arbitrary axis

If you have MM, you need not worry about this. But if you're working in some other software, it's possible you may have to use a formulation of R that allows you to input an arbitrary vector representing an arbitrary axis of rotation. It can be found in the Wikipedia Rotation Matrix article under the heading: Rotation Matrix from the Axis and the angle.

Also, you can use the general *R* matrix from this website (http://inside.mines.edu/fs_home/gmurray/ArbitraryAxisRotation/) allowing rotation about an arbitrary rotation axis going through an arbitrary point. Go to the final result at the end of 6.2 for the most general formulation of *R*.

References:

[1] Rubenson, Jonas, et al. "Running in ostriches (Struthio camelus): three-dimensional joint axes alignment and joint kinematics." *Journal of Experimental Biology* 210.14 (2007): 2548-2562.

[2] Kambic, Robert E., Thomas J. Roberts, and Stephen M. Gatesy. "Long-axis rotation: a missing degree of freedom in avian bipedal locomotion." *The Journal of experimental biology* (2014): jeb-101428.