**Math in Processing**

**Carson Foster**

While you can use all of the Java Math methods in Processing, the Processing library itself provides other useful math functions that you need to know.

Trigonometry

This isn’t a math class, so I’ll refrain from lecturing you about SOH CAH TOA. However, trig is very useful, so you should know what functions to use in Processing. First, we’ll look at the three basic trig functions. Can you guess what they are?



The three basic trig functions are as you would expect, and each function takes one argument: the angle. Like other functions in Processing, this angle must be in radians. If you ever need to convert between radians and degrees, use the following two functions:



Both of these functions return a float that represents the same angle as the argument, but in the system that matches the function name. Therefore, calling the degrees function will return a float that represents the same angle as the argument, but in the degrees system.

Next, let’s talk about inverse functions. The inverse of a trig function takes a value and returns the angle (in radians) for which that trig function equals the passed value. The inverse functions for trigonometry are also known as the arc variants (e.g. arcsine, arccosine, arctangent). This will be reflected in the naming convention for the functions:



However, there’s another inverse function that is closely related to atan. This function is:



This function is still an inverse of the tangent function, but instead of taking a value, it takes the vertical side length and the horizontal side length of the matching triangle. Stated in another way: if , then . This function is most often used to determine the direction between a point and the mouse cursor:



Randomness

Once more, you do have access to the randomness methods/classes in Java, like Math.random() and java.util.Random. However, there are three additional functions that I would like to discuss here, as they are fairly useful. First, we have the classic randomness function: random. Aptly named, it returns a pseudorandom float that is greater than or equal to a given low value, and less than a given high value.



If only one argument is passed, it treats that argument as the high value and zero as the low value. This is a simple, yet handy function that is simpler to use/access than either Java version of this method.

Next, Processing provides a function that provides a random value that conforms to the Gaussian distribution. This Gaussian distribution is also known as the normal curve or Bell curve, and will have a mean of zero and a standard deviation of one. Note that there are no maximum or minimum values; it is simply extremely unlikely to be returned a value that is very far away from zero. The function works as follows:



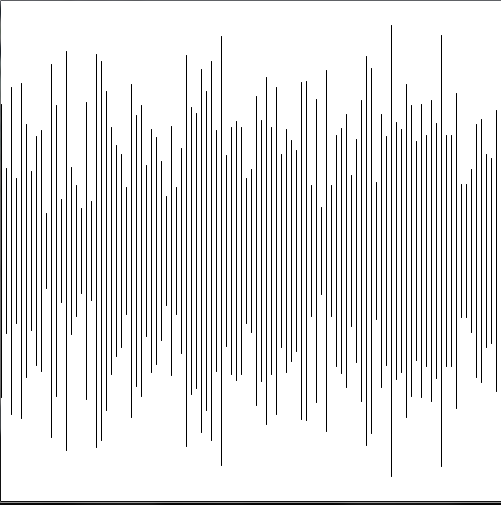
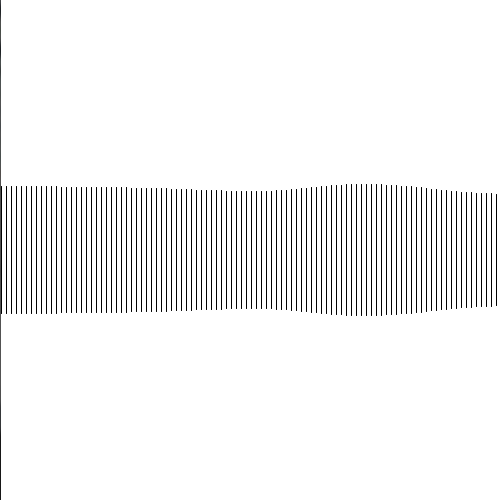
It’s such a simple command, yet such a powerful tool.

Our last Processing function that deals with randomness is called noise. It generates a specific type of noise called “classic Perlin noise”. I’m not too familiar with the concept, but essentially, Perlin noise is a more natural, organic type of pseudorandom number generator based on a coordinate grid. Each point on the 1D, 2D, or 3D grid (depending on how many arguments you pass) contains a random float value from 0 to 1. The smaller the differences between points, the smoother/less drastic the changes in random values will be. Here are the different versions of this function:



Fun fact: Perlin noise is often used for procedural generation! Pretty cool, right?

As an example of ‘smoothness’: look at the following visualizations of 1D Perlin noise for two different increment values:



The left visualization had an increment of 1; that is, each of the bars were generated by x-coordinates that increased by one each time. The right visualization, in contrast, had an increment of 0.005, which resulted in a very smooth, yet still random, pattern. The code that generated these is included below:



Vectors

Vectors are useful for a wide variety of applications, especially physics-based simulations. You can even use vectors to represent points in space. In Processing, there’s an entire class devoted to vectors: PVector. First, we must construct a vector:



These constructors tell us that we can create an empty vector, a 2D vector, or a 3D vector. They also tell us that Processing vectors use the component form of the vector, instead of specifying a heading/direction and a magnitude. Once a vector has been created, we can access, and change, its component fields directly:



As you might have guessed, these instance variables are called x, y, and z. However, if you want to use a method to set these components, Processing also provides a method for you to do so:



In the above and following sections of code, unless noted otherwise, p represents a PVector object. With this method, you can set the component values by providing them as arguments, providing another vector (v) to copy the components of, or providing a float array (arr) of length 2 or 3 (if the array has a length of 2, then the z component is set to zero).

Now that we can create, access, and change our vectors, we can move on to the different types of vector operations we can perform. First, let’s look at vector addition and subtraction:



All of these methods return the result, regardless of whether or not the static version is used. However, the instance methods all also directly modify the object they are called from. The last version of these methods with the target argument works like this: if target is an instantiated PVector, then set its value equal to the result of the operation between the other two vectors, otherwise, just return the result. Note that for vector subtraction (and normal subtraction) the convention is first object minus last object, so the last two methods subtract v2 from v1 (v1 – v2).

Next, we have multiplication and division. These two methods specifically deal with vector multiplication/division by a **scalar**.



The one argument that is required by all of these functions, n, is the scalar to multiply or divide by. Otherwise, these methods act in the same manner as the previous methods we looked at.

There are more interesting operations than these basic ones, which we shall get to now:



As the names suggest, dot calculates the dot product of two vectors, while cross calculates the cross product of two vectors. Note that the dot product of two vectors is a number, not another vector. These methods work in the same manner as those previously discussed, except for the second cross product method, which appears to be a hybrid of the two types of methods. For this method, the cross product is performed between p and the argument, but then instead of storing the result back in p, the result is stored into target.

Finally, we come to the miscellaneous useful methods. We shall start with those related to the magnitude of the vector.



The first method calculates the magnitude of the vector (the length). As this function requires a square root operation, Processing provides magSq to calculate the square of magnitude (which doesn’t require a square root, and is hence much quicker). In addition, you can set the magnitude of the vector, while keeping the direction/relationship between components the same. The methods with null are called as shown, passing null as the first argument. The null variant of setMag creates a new vector with the required magnitude and same direction as p and returns it. The normalization of a vector is reducing its magnitude to one, and the null version returns a new vector with magnitude one and the same direction as p. The final method sets the magnitude of the vector to the passed value if the current magnitude is above that passed value, and otherwise does nothing.



These last two methods deal with angles. The heading of a vector is the direction it points, which heading calculates and returns (in radians, of course). The last method does as its name suggests, and calculates the angle between two vectors. Both of these methods return floats.

Miscellaneous