A Little Trigonometry and a Bit of Randomness
Curves
Exponents, Roots

- The `sq()` function is used to square a number, multiply a number by itself.
- The result is always positive, because squaring a negative number yields a positive result.

```c
float a = sq(1);  // Assign 1 to a  (1 * 1)
float b = sq(-5); // Assign 25 to b  (-5 * -5)
float c = sq(9);  // Assign 81 to c  (9 * 9)
```
The \texttt{sqrt()} function is used to calculate the square root of a number.

The square root of a number is always positive, even though there may be a valid negative root.

```c
float a = sqrt(6561); // Assign 81 to a
float b = sqrt(625);  // Assign 25 to b
float c = sqrt(1);    // Assign 1 to c
```
Exponents, Roots

- The `pow()` function calculates a number raised to the power of another number.
- It has two parameters `pow(num, exponent)` where `exponent` is the number of times to multiply `num` by itself.

```c
float a = pow( 1, 3); // Assign 1.0 to a (1 * 1 * 1)
float b = pow( 3, 4); // Assign 81.0 to b (3 * 3 * 3 * 3)
float c = pow(-3, 3); // Assign -27.0 to c (-3 * -3 * -3)
float d = pow( 3, -2); // Assign 0.11 to d (1/3 * 3)
```
Normalising

- Numbers are often converted to the range 0.0 to 1.0 for making calculations.
- This can also be accomplished using the `norm()` function that takes the number to be normalised and the expected range for the number.

```c
float x = norm( 0.0, 0.0, 255.0); // Assign 0.0 to x
float y = norm( 102.0, 0.0, 255.0); // Assign 0.4 to y
float z = norm( 255.0, 0.0, 255.0); // Assign 1.0 to z

float a = norm(-144.0, 0.0, 180.0); // Assign -0.8 to a
float b = norm( 230.0, 0.0, 23.0);  // Assign 10.0 to b
float c = norm( 15.0, 10.0, 20.0);  // Assign 0.5 to c
```
Linear Interpolation

- After normalisation, a number can be converted to another range using `lerp()`.
- The name “lerp” is a contraction for “linear interpolation.” The function has three parameters: `lerp(value1, value2, amt)`

```cpp
float x = lerp(-20.0, 60.0, 0.0);  // Assign -20.0 to x
float y = lerp(-20.0, 60.0, 0.5);  // Assign  20.0 to y
float z = lerp(-20.0, 60.0, 1.0);  // Assign  60.0 to z
```
Mapping

- The `map()` function is useful to convert directly from one range of numbers to another. It has five parameters.
- `map(value, low1, high1, low2, high2)`

```plaintext
float x = map( 20, 0, 255, -1, 1);  // Assign -0.84 to x
float y = map(  0, 0, 255, -1, 1);  // Assign -1.0  to y
float z = map(255, 0, 255, -1, 1);  // Assign  1.0  to z
```
Simple Curves

- Exponential functions can be used to create simple curves.
- Using normalised values with the `pow()` function produces exponentially increasing or decreasing numbers between 0 and 1.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.2</td>
<td>0.0016</td>
</tr>
<tr>
<td>0.4</td>
<td>0.0256</td>
</tr>
<tr>
<td>0.6</td>
<td>0.1296</td>
</tr>
<tr>
<td>0.8</td>
<td>0.4096</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

\[ y = x^4 \]
Exponential Curve 1

Using pow() with a normalised value between 0 and 1

```c
for (int x = 0; x < 100; x++) {
    float n = norm(x, 0, 100);
    float y = pow(n, 4);
    y *= 100;
    point(x, y);
}
```
Exponential Curve 2

Using `pow()` with a normalised value between 0 and 1

```c
for (int x = 0; x < 100; x++) {
    float n = norm(x, 0, 100);
    float y = pow(n, 0.4);
    y *= 100;
    point(x, y);
}
```

\[ y = x^{0.4} \]
Exponential Gradients

Using `pow()` with a normalised value between 0 and 1

```cpp
for (int x = 0; x < 100; x++) {
    float n = norm(x, 0, 100);
    // Draw linear gradient
    float val = n * 255;
    stroke(val);
    line(x, 0, x, 50);
    // Draw exponential gradient
    float val2 = pow(n, 4) * 255;
    stroke(val2);
    line(x, 50, x, 100);
}
```
Trigonometry
Angles

Degree values

Radian values
Angles

• Use the pre-defined constant `PI` to represent $\pi$.

• Multiples of $\pi$ are can also be expressed with the constants `QUARTER_PI`, `HALF_PI`, and `TWO_PI`.

```plaintext
println(PI);                   // Prints "3.1415927"
println(HALF_PI);             // Prints "1.5707964"
```
Radians

- Processing has some handy functions for converting between radians and degrees.
  - radians(value) converts degrees to radians
  - degrees(value) converts radians to degrees

```java
float r1 = radians(90);
float r1 = radians(180);
println(r1);         // Prints "1.5707964"
println(r2);         // Prints "3.1415927"
float d1 = degrees(PI);
float d2 = degrees(TWO_PI);
println(d1);         // Prints "180.0"
println(d2);         // Prints "360.0"
```
Sine and Cosine

- The \( \text{sin()} \) and \( \text{cos()} \) functions are used to determine the sine and cosine value of any angle.
- Each of these functions requires one parameter:
  \[
  \text{sin}(\text{angle}) \ \text{cos}(\text{angle})
  \]
- The \text{angle} parameter is always specified as a \textit{radian} value. The values returned from these functions are always between the floating-point values of \(-1.0\) and \(1.0\).
Sine Waves

<table>
<thead>
<tr>
<th>Degrees</th>
<th>0</th>
<th>90</th>
<th>180</th>
<th>270</th>
<th>360</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radians</td>
<td>0</td>
<td>π/2</td>
<td>π</td>
<td>π+π/2</td>
<td>2π</td>
</tr>
<tr>
<td>Decimal radians</td>
<td>0</td>
<td>1.57</td>
<td>3.14</td>
<td>4.71</td>
<td>6.28</td>
</tr>
<tr>
<td>Constants</td>
<td>0</td>
<td>HALF_PI</td>
<td>PI</td>
<td>PI+HALF_PI</td>
<td>TWO_PI</td>
</tr>
</tbody>
</table>

Sine wave

Sine value
size(700, 100);
noStroke();
smooth();
fill(0);
float angle = 0;        // Angle to receive sine values from
float angleInc = PI/40; // Increment between the next angle
float yOffset = 50;     // Y offset
float scaleVal = 35;    // Scale value for the wave magnitude
for (int x = 0; x <= width; x += 5) {
    float y = yOffset + (sin(angle) * scaleVal);
    rect(x, y, 2, 4);
    angle += angleInc;
}
size(700, 100);
noStroke();
smooth();
fill(0);
float angle = 0;       // Angle to receive sine values from
float angleInc = 4.5;  // Increment between the next angle
float yOffset = 50;    // Y offset
float scaleVal = 35;   // Scale value for the wave magnitude
for (int x = 0; x <= width; x += 5) {
    float y = yOffset + (sin(radians(angle)) * scaleVal);
    rect(x, y, 2, 4);
    angle += angleInc;
}
The same sketch using angles expressed in degrees

```sketch
size(700, 100);
noStroke();
smooth();
fill(0);
float angle = 0;        // Angle to receive sine values from
float angleInc = 4.5;   // Increment between the next angle
float yOffset = 50;     // Y offset
float scaleVal = 35;    // Scale value for the wave magnitude
for (int x = 0; x <= width; x += 5) {
    float y = yOffset + (sin(radians(angle)) * scaleVal);
    rect(x, y, 2, 4);
    angle += angleInc;
}
```
The same sketch using angles expressed in degrees

```cpp
size(700, 100);
noStroke();
smooth();
fill(0);
float angle = 0;        // Angle to receive sine values from
float angleInc = 4.5;   // Increment between the next angle
float yOffset = 50;     // Y offset
float scaleVal = 35;    // Scale value for the wave magnitude
for (int x = 0; x <= width; x += 5) {
    float y = yOffset + (sin(radians(angle)) * scaleVal);
    rect(x, y, 2, 4);
    angle += angleInc;
}
```
size(700, 100);
float offset = 50;
float scaleVal = 30.0;
float angleInc = PI/56.0;
float angle = 0.0;
beginShape(TRIANGLE_STRIP);
for (int x = 4 ; x <= width+5; x += 5) {
    float y = sin(angle) * scaleVal;
    if ((x % 2) == 0) { // Every other time through the loop
        vertex(x, offset + y);
    } else {
        vertex(x, offset - y);
    }
}
angle += angleInc;
endShape();
Randomness
Random

- The `random()` function is used to create unpredictable values within a range:
  - `random(high)`
  - `random(low, high)`

```java
// Assign f a random float value from 0 to 5.2
float f = random(5.2);

// Try to assign a float to an int
int i = random(5.2); // ERROR!

// Assign j an int value from 0 to 5
int j = int(random(5.2));
```
Random Drawings 1

Using the random() function to draw

```
smooth();
strokeWeight(10);
stroke(0, 130);
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
```
Random Drawings 2

Using `random()` within a for structure is an easy way to generate lots of random numbers for drawing complex forms.

```java
background(0);
stroke(255,60);
for (int i = 0; i < 100; i++) {
    float r = random(10);
strokeWeight(r);
    float offset = r * 5.0;
    line(i-20, 100, i+offset, 0);
}
```
Random Drawings 3

To use random values to determine the flow of the program, you can place the `random()` function in a relational expression.

```java
int num = int(random(50)) + 1;
for (int i = 0; i < num; i++) {
    line(i * 2, 0, i * 2, 100);
}
```
Random Seed

▪ It’s sometimes desirable to include unpredictable numbers in your programs but to force the same sequence of numbers each time the program is run.

▪ The `randomSeed(value)` function is the key to producing such numbers. The value parameter must be an `int`.

▪ Use the same value parameter in a program each time it is run to force the same random numbers to be produced in the same order.
Drawing Using Random Seed

Using \texttt{randomSeed()} allows use to produce the same picture each time the program is run.

```cpp
int s = 6; // Seed value
background(0);
stroke(255,60);
randomSeed(s);
for (int i = 0; i < 100; i++) {
    float r = random(10);
strokeWeight(r);
float offset = r * 5;
    line(i-20, 100, i+offset, 0);
}
```
Noise

- Values produced using the `random()` function can be hard to control.

- The `noise()` function is a more controllable way to create unexpected values.
  
  - The `noise()` function uses the *Perlin Noise* technique, developed by Ken Perlin. Originally used for simulating natural textures through subtle irregularities, Perlin Noise is also used for generating shapes and realistic motion.
Noise

- The `noise()` function works by interpolating between random values to create smoother transitions than the numbers returned from `random()`.

- The `noise` function has between one and three parameters:
  - `noise(x)`
  - `noise(x, y)`
  - `noise(x, y, z)`
Different versions of the `noise()` function produce noise with 1, 2 or 3 dimensions:

- `noise(x)` creates random numbers that can be used for drawing lines, etc.
- `noise(x, y)` creates random number pairs that can be used for generating 2D textures.
- `noise(x, y, z)` creates random number triplets that can be used for generating 3D shapes, textures or animated 2D textures.
size(600, 100);
float v = 0.0;
float inc = 0.1;
noStroke();
fill(0);
noiseSeed(0);
for (int i = 0; i < width; i = i+4) {
    float n = noise(v) * 70.0;
    rect(i, 10 + n, 3, 20);
    v = v + inc;
}
float xnoise = 0.0;
float ynoise = 0.0;
float inc = 0.04;
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        float gray = noise(xnoise, ynoise) * 255;
        stroke(gray);
        point(x, y);
        xnoise = xnoise + inc;
    }
}
xnoise = 0;
ynoise = ynoise + inc;
Diverse textures can be created using `noise()` in collaboration with `sin()`.

The following example deforms a regular sequence of bars created with `sin()` into a texture reminiscent of those found in nature.

The `power` variable sets the amount the texture deforms from the lines and the density parameter `d` sets the granularity of the texture.
float power = 3; // Turbulence power
float d = 8; // Turbulence density
noStroke();
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        float total = 0.0;
        for (float i = d; i >= 1; i = i/2.0) {
            total += noise(x/d, y/d) * d;
        }
        float turbulence = 128.0 * total / d;
        float base = (x * 0.2) + (y * 0.12);
        float offset = base + (power * turbulence / 256.0);
        float gray = abs(sin(offset)) * 256.0;
        stroke(gray);
        point(x, y);
    }
}
d = 8          d = 32          d = 128

power = 3

power = 6
Lab Exercises
Lab Exercises

- Draw the curve $y=1-x^4$ to the display window.
- Use the data from the curve $y=x^8$ to drawing something unique.
- Compose a range of gradients created from curves.
- Create a composition with the data generated using $\sin()$. 
Lab Exercises

› Use three variables assigned to random values to create a composition that is different every time the program is run.

› Create a composition using a for structure and `random()` to make a different dense composition every time the program is run.

› Use `noise()` and `noiseSeed()` to create the same irregular shape every time a program is run.