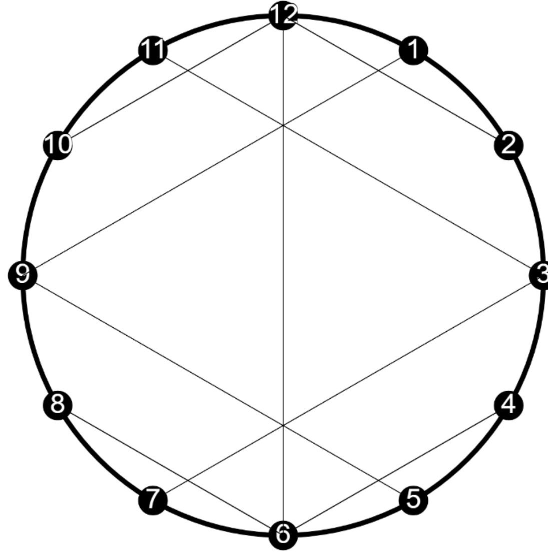
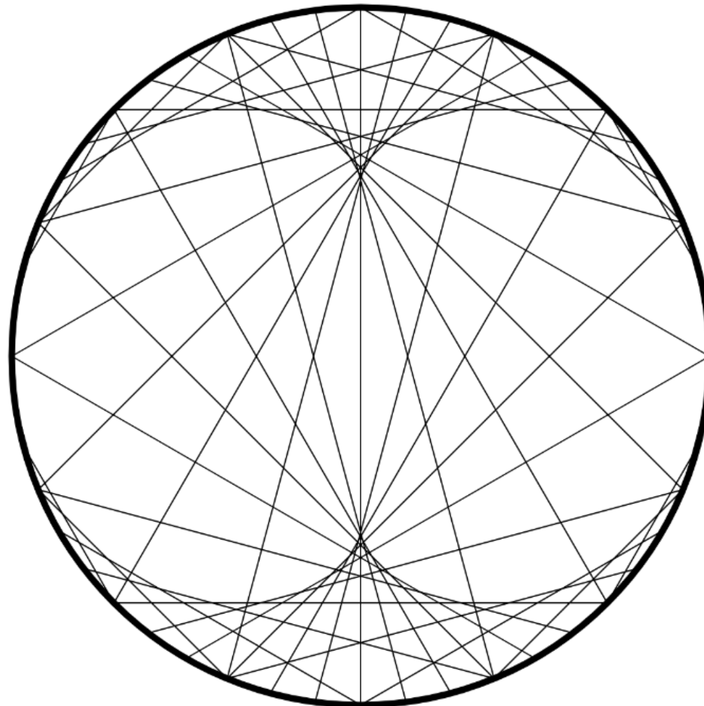
Multiplication Wheel

All sorts of fascinating mathematical shapes can be constructed from remarkably simple rules. If you join every number on a clock face to the number which is three times its size (wrapping around the clock as necessary, so that 5 maps to 3 in place of 15, etc), you get:



But if we moved over to a 48-hour clock, it would look something like this:



This curious shape is called a nephroid (because it's kidney-shaped, of course). Mathematically, you can also construct this shape by tracing the locus of a point on the circumference of a circle as it rolls around the outside of a circle with twice the radius.

In fact, this shape is just one of many interesting shapes that emerge when we follow this simple multiplication rule.

Let's use p5.js to create a dynamically animated version…

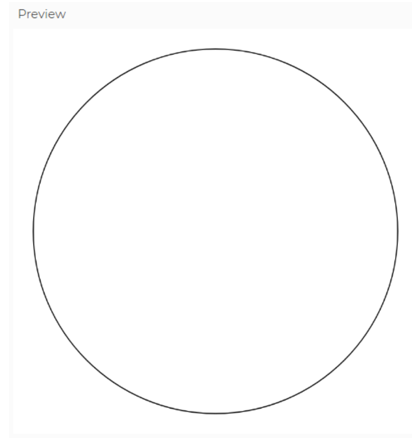
The outer ring

First, we'll need a circle around which we will place equally spaced points.
The radius of this circle will be fixed, so we can declare it at the very beginning, then set its value based on the sketch canvas width inside the *setup* function:

```
let r;

function setup() {
  createCanvas(400, 400);
  r = 0.9 * width / 2;
}

function draw() {
  background(255);
  circle(width/2, height/2, 2 * r);
}
```



I'm making my circle fill 90% of the width of the window, so that we won't lose any detail at the edges. Note that *width* is a p5 variable, so although we can make *r* a global variable by declaring it at the start, we have to give it its value in *setup*.

The *circle* function takes three arguments: x and y coordinates of the centre, and a diameter (not a radius!)

The points around
If we shift our viewpoint from Cartesian coordinates (across and up) to polar coordinates (how far and at what angle), this is straightforward. The conversion you may well have seen before: $x = r\cos\theta \quad y = r\sin\theta$

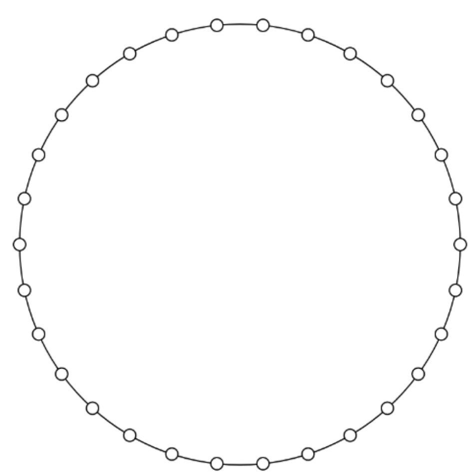
Setting $\theta = 0$ would give us $(r, 0)$ (that is, the right-most point), and $\theta = \frac{\pi}{2}$ (yes, computers work in radians by default) would give $(0, r)$ (that is, the bottom-most point… yes, computer graphics are top-down by default, not bottom-up like a Cartesian graph).
We don't much mind where we start and which way we go, though, so these details don't need to bother us here.

We need a loop so we can plot points for each value of θ we need between 0 and 2π.

```
let r;
let n = 30;

function setup() {
  createCanvas(400, 400);
  r = 0.9 * width / 2;
}

function draw() {
  background(255);
  circle(width/2, height/2, 2 * r);
  for (let i=0; i<n; i++){
    theta = i * 2 * PI / n;
    circle(width / 2 + r * cos(theta), height / 2 + r * sin(theta), 10);
  }
}
```



Note that I'm using *width/2* and *height/2* to offset the centre to that of the canvas.
Sidenote on loops in JavaScript

I've declared a new variable *n* to represent the number of points around the circle (I may well want to experiment with changing this later). Notice the loop notation in JavaScript:

```
for (let i=0; i<n; i++){
  console.log(i);
}
```

The keyword *for* is followed by what is effectively three lines of code (notice the semi-colons separating them).

These lines determine how the loop runs: a new variable *i* is declared and set to 0, and each time the code block runs, it is incremented by 1 (*i++* means the same as *i+=1*) provided the condition *i<n* is still met.

Finally, a set of curly braces follows, just like in an *if* statement or a function. The code within the curly braces executes each time the loop runs.

In Python, the equivalent to this code would be:

```
for i in range(n):
  print(i)
```

A helper function

In order to draw all our lines, we'll need to get the coordinates of the start and end points. The *line* function in p5 takes four arguments: a line from (x_1, y_1) to (x_2, y_2) is drawn using *line*(x_1, y_1, x_2, y_2). Since constructing the x and y values to draw our circles was a bit of a handful just now, it may be worth making a helper function. It'll take *i* as its input and return the corresponding x and y values:

```
1  let r;
2  let n = 30;
3
4  function coordinates(i){
5    let theta = i * 2 * PI / n;
6    return [width/2 + r*cos(theta), height/2 + r*sin(theta)]
7  }
8
9  function setup() {
10   createCanvas(400, 400);
11   r = 0.9 * width / 2;
12 }
13
14 function draw() {
15   background(255);
16   circle(width/2, height/2, 2 * r);
17   for (let i=0; i<n; i++){
18     let [x,y] = coordinates(i);
19     circle(x, y, 10);
20   }
21 }
```

Note that *r* and *n* are in the global scope, so my new helper function can use them even if they're not passed to it as variables.

Even though this code doesn't draw anything different, notice how the for loop code block has become simpler through the use of our helper function.

Recall that, in Python, a tuple (like an immutable list) can be unpacked using code like:

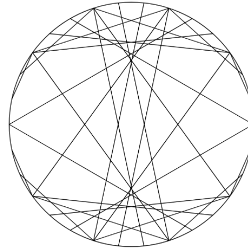
```
my_point = (2, 4)
x, y = my_point
```

In JavaScript, we can do a similar thing with arrays (the equivalent of a list), as is done here on line 18, where the array object *[x,y]* is defined to be the return value of *coordinates(i)*, which in turn is an array with two elements. The elements are now stored as *x* and *y*.

Drawing the lines

We can now quickly modify our code so that instead of drawing circles, we draw a line between our current position and the new one. Remember that if we're multiplying by 3, but also 'wrapping around', we'll need the modulo operator % (same as in Python). So if $5 \times 3 = 15$, on a 12 hour clock we'd say *(5*3)%12* and get an answer of *3*.

```
14 function draw() {
15   background(255);
16   circle(width/2, height/2, 2 * r);
17   for (let i=0; i<n; i++){
18     let [x1,y1] = coordinates(i);
19     //circle(x, y, 10);
20     let [x2,y2] = coordinates((i * 3) % n);
21     line(x1, y1, x2, y2);
22   }
23 }
```

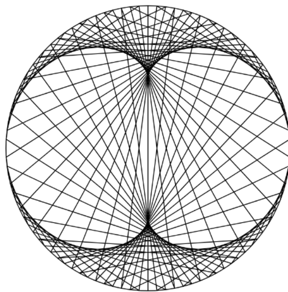


If it worked, you should get something like this. I've commented out the circles to make it look cleaner.

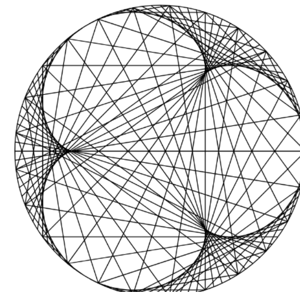
Make it move!

All that remains is to tweak some values, and add some animation.

First I'll crank up the number of points to 100 by changing the value of *n* at the start…



Then I'll see what happens if I change line 20 so I'm multiplying by 4 instead of 3…



And finally, instead of just changing the 3 to a 4 on line 20, what if we set it as, say, 2, right at the start, then increment it gradually each time the *draw* function loops?

```
1 let r;
2 let n = 100;
3 let factor = 2;
```

```
15 function draw() {
16   background(255);
17   factor += 0.01;
18   circle(width/2, height/2, 2 * r);
19   for (let i=0; i<n; i++){
20     let [x1,y1] = coordinates(i);
21     //circle(x, y, 10);
22     let [x2,y2] = coordinates((i * factor) % n);
23     line(x1, y1, x2, y2);
24   }
25 }
```

What's next?

Try adding some colour, or changing line thicknesses. You could even use the *map* function to colour lines depending on their position or their length for some really neat effects.

```
let dist = ((x2-x1)**2 + (y2-y1)**2)**0.5;
let r = map(dist, 0, width, 0, 255);
let g = 255 - r;
let b = 255;
stroke(r, g, b)
line(x1, y1, x2, y2);
```

Use text. No initialising of font required: this code shows you the current factor in the corner.

```
text(factor, 0, height - 20);
```

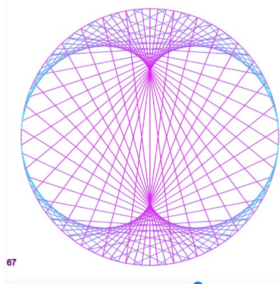
Try a slider. Declare a variable at the start:

```
let slider;
```

Then initialise it inside *setup*:

```
slider = createSlider(0, 100, 1);
slider.size(width);
```

Lastly, use *slider.value* in place of *factor*.



This makes it easier to pick out specific patterns. Looks like a nephroid appears again at 67. Can you find any other interesting ones?