Polar Plotter

p5.js

Polar curves are as mesmerising to watch as they are counter-intuitive to visualise for people who have grown up on good old-fashioned Cartesian coordinate systems.

If we want to be able to properly get to grips with them, a static diagram is not really sufficient – ideally we need an animation to show not only what curve is mapped out, but how, and for which values of theta.

In this project, we'll generate a *Plotter* class which will take care of showing on-screen the graphs of one or more polar curves, but also allow us to watch them being drawn. In order to easily construct curves for different functions we'll make use of JavaScript's own anonymous function notation, the equivalent of Python's *lambda* functions. Recall that this is a quick easy way to define a function without having to name it or define it separately.

The Plotter Class

```
class Plotter{
  constructor(f){
    this.f = f;
    this.points = [];
  }
  update(t){
    let r = this.f(t);
    this.points.push([r, t]);
  }
}
```

We'll start by defining our *Plotter* class, which takes as its input only one thing: a function. This is intended to be a function f such that $r = f(\theta)$, which is why in *update* we use $r = this.f(t)$. We'll worry about how to pass a function as an argument later.

I'm also making an array to store points the point has gone through so that we can join them to form the curve.

So far, this class will accept a function, then, when passed values of t (that is, θ), it computes and stores the relevant values in a list. Next, I want to display both the current position of the moving point and the curve so far, using the stored values in the *points* array.

```
display(){
  fill("white");
  noStroke();
  let [r, t] = this.points[this.points.length - 1];
  let x = width/2 + sf * r * cos(t);
  let y = height/2 - sf * r * sin(t);
  circle(x, y, 8);
}
```

This takes the most recent entry stored in *points*, computes x and y values using the conversion formulae $x = r\cos\theta$ and $y = r\sin\theta$, and then modifies the results so that the curve is centred in the middle of the screen and scaled according to some scale factor *sf* (so it'll be big enough to see).

Note that we also change the sign of the y ordinate so that as y increases we move up, not down.

It's good practice, if we start making use of variables in one part of our program that need to be defined elsewhere, to build them in as we go. Before we get too far ahead of ourselves, let's declare the variables we have so far assumed: *f*, *t* and *sf*:

```
let sf = 100;
let t = 0;
let curves = [];

function setup() {
  createCanvas(400, 400);
  curves.push(new Plotter(t=>cos(3*t)));
  curves.push(new Plotter(t=>1-sin(t)));
}
```

In the main scope of the program, we'll declare a scale factor. 100 seems reasonable: a curve like $r = \cos\theta$ will be a circle of diameter 100 pixels, for instance.
I'm making an array for my curves so I can potentially draw lots of curves at once.
In *setup* I push two *Plotter* objects into my *curves* array.

Notice that the argument passed to the *Plotter* object is an anonymous function…

Anonymous functions in JavaScript

In Python, we might write something like `lambda t: sin(t)` to define an anonymous function, using the keyword *lambda* to indicate the fact, then the input parameter, a colon and the output.
In JavaScript, the special symbol combo => (an equals sign followed by a greater-than sign) makes it even more efficient: **x=>x**2** does the same job in JavaScript as **lambda x: x**2** does in Python.

Drawing the moving points

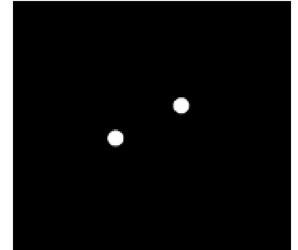
It's about time we made our moving markers appear on the screen (not least so we can error-check and fix any bugs or unexpected behaviour we discover).

```
function draw() {
  background(0);
  if (t > 2*PI){t = 0}
  t += 0.02
  for (let curve of curves){
    curve.update(t);
    curve.display();
  }
}
```

I'm making my background black so that I can see the moving white dots more clearly. The parameter *t* is reset to 0 every time it reaches its limit of 2π. Depending on how sophisticated you want to make this code, you could allow the user to define the range here.

Each curve is updated using the global variable *t*, and then displayed.

The result is a rather nice dance of spinning dots, but not – as yet – very enlightening when it comes to visualising the entire polar curve.



Ideally what I'd like is for the curve to be traced out by these dots as they move around. That, after all, is why we asked the *Plotter* object to store its historical positions in an array of points.

Drawing the curve

Remember that a curve rendered on a computer screen is nothing more than a series of straight lines between a large number of very close points. We don't need any sophisticated equipment for this: a simple *beginShape()* and *endShape()* sandwiching a loop through the points in the array will do the trick:

```
drawCurve(){
  noFill();
  stroke("white");
  strokeWeight(2);
  beginShape();
  for (let i=0; i<this.points.length; i++){
    let [r, t] = this.points[i];
    let x = width/2 + sf * r * cos(t);
    let y = height/2 - sf * r * sin(t);
    vertex(x, y);
  }
  endShape();
}
```

This forms another method within our *Plotter* class, along with *display*, *update* and *constructor*. If some lines look familiar it's because I copied and pasted from the *display* method. It might be worth making a helper function whose job it is to convert *r* and *t* values to pixel locations, but since I'm probably only doing this a couple of times, this seems easiest.

Remember to include *noFill()* if you want to avoid getting a solid shape (although if you want to investigate the area bounded by a polar curve, you can come back and edit this).

Making its shadow

I like the idea of the point tracing out the curve, but it may also be nice to see the whole curve in advance. I'm calling this the 'shadow' of the curve, and I'll display it in full, but in grey, even before the point makes its way around. Another *Plotter* method is called for here:

```
drawShadow(){
  noFill();
  stroke("grey");
  strokeWeight(1);
  beginShape();
  for (let t=0; t<2*PI; t+=0.02){
    let r = this.f(t);
    let x = width/2 + sf * r * cos(t);
    let y = height/2 - sf * r * sin(t);
    vertex(x, y);
  }
  endShape();
}
```

This one is similar to the previous method we made, but this time the values of *t* are purely local scope variables: they exist only within the *for* loop. The code runs through the entire loop generating points for the curve to draw the shape all in one go.

Now the curves should appear immediately in grey, and then be gradually filled in with white as its point makes its way around.

One more simple improvement would be to 'reset' the white tracing each time *t* is reset to zero. This is far from the best or most scalable solution, but it is the laziest! If *t* gets reset, it'll return to 0.02, and in that case the *Plotter*'s *update* method can wipe the *points* array.

```
update(t){
  let r = this.f(t);
  if (t < 0.03){this.points = []}
  this.points.push([r, t]);
}
```

Sweeping out the angle

One improvement I'd like to make is a ray line that sweeps out the angle θ as *t* increases. This would help me see, for instance, when the curve is being traced out by a negative value of r. This is easily done in the *draw* function:

```
function draw() {
  background(0);

  let [X, Y] = [width/2, height/2];
  stroke("grey");
  line(X, Y, X + width, Y);
  let [x, y] = [width * cos(t), height*sin(t)];
  stroke("green");
  line(X, Y, X + x, Y - y);
  stroke("red");
  line(X, Y, X - x, Y + y);

  if (t > 2*PI){t = 0}
  t += 0.02
  for (let curve of curves){
```

First, I set *X* and *Y* to be the coordinates of the centre of the screen: everything else is relative to that point. Next, I make the fixed 'initial line', stretching off the screen to the right and coloured grey.
The values chosen for lowercase *x* and *y* are simply a long way from the centre at the right angle, *t*.
I make a green line to represent positive r values, and then – by simply changing the sign of *x* and *y*, I make a red line in the exact opposite direction.

I can even change the colour of the moving point if I choose, based on whether the value of *r* at the time is positive or negative, by altering the relevant code in the *display* method of the *Plotter* class:

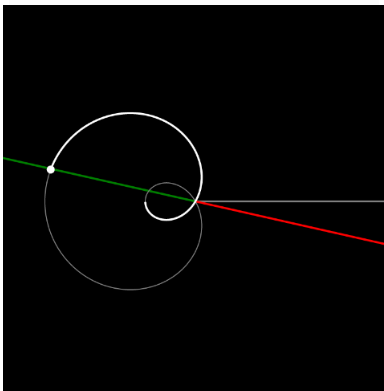
```
display(){
  fill("white");
  noStroke();
  let [r, t] = this.points[this.points.length - 1];
  let x = width/2 + sf * r * cos(t);
  let y = height/2 - sf * r * sin(t);
  if (r < 0){fill("red")}
  circle(x, y, 8);
}
```

We know the value of *r* in here, so we can simply change the colour in the even that *r* is negative.

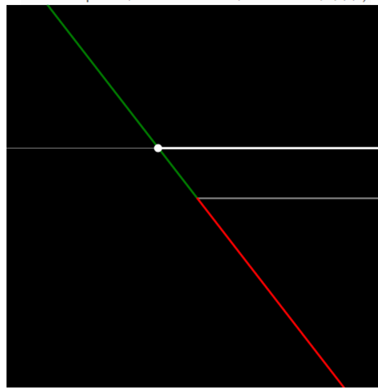
Challenge: can you make the path being drawn change colour depending on the sign of *r*?

Try out a few common (or less common) equations to see what the effect is. If need be, modify the scale factor to suit.

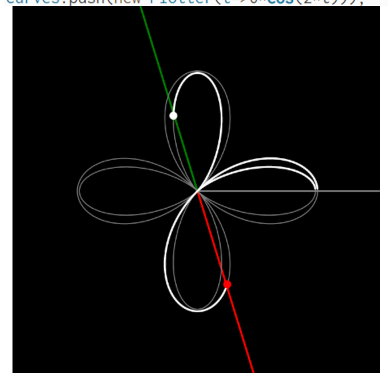
`curves.push(new Plotter(t=>1-2*cos(t)));`



`curves.push(new Plotter(t=>1/sin(t)));`



```
curves.push(new Plotter(t=>3+3*cos(4*t)));
curves.push(new Plotter(t=>6*cos(2*t)));
```



Even better if…

- You could copy and paste some button code from another project and let the user control whether to view just the moving dot, the traced out curve or the shadow.
- You could also use a slider to control all sorts of things, including the functions themselves. Can you make a slider that lets us view $r = \cos(n\theta)$ for a range of different values of n?
- Can you colour-code your curves, so they are easier to distinguish when there's more than one?
- Can you include a polar grid? Concentric circles and perhaps radial lines at intervals of $\frac{\pi}{6}$ around the centre?
- Can you use variables for some of the default values, to allow for a range greater than $0 \leq \theta < 2\pi$ more easily, or a step-length of, say, *dt* rather than always using 0.02?

```
curves.push(new Plotter(t=>10/t));
end = 8*PI;
```