The Mandelbrot Set

Think of a number. Square zero and add your number. Square the result, and add your number again. Square the result once more, and add your number once more. Chances are, the values quickly get huge. But does that always happen?

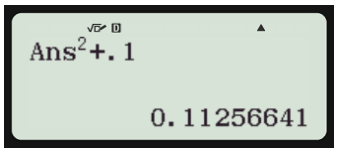
Example with starting value 2:

$$0^2 + 2 = 2$$
$$2^2 + 2 = 6$$
$$6^2 + 2 = 38$$
$$38^2 + 2 = 1446$$
$$1446^2 + 2 = 2090918$$

Example with starting value 0.1:

$$0^2 + 0.1 = 0.1$$
$$0.1^2 + 0.1 = 0.11$$
$$0.11^2 + 0.1 = 0.1121$$
$$0.1121^2 + 0.1 = 0.11256641$$
$$0.11256641^2 + 0.1 = 0.1126711967$$

Note: you can automate this process to some extent using your calculator's *ANS* function:



Under what circumstances does this recursive function converge, like for 0.1?
Well, if it converges to a limit, say L, then $L = L^2 + c$ where c is our starting value.

This is a quadratic in L which has a discriminant of $1 - 4c$. There will be no real values of c greater than $\frac{1}{4}$ that satisfy the equation. At $c = \frac{1}{4}$, the sequence converges to $\frac{1}{2}$, but it does so very very slowly.

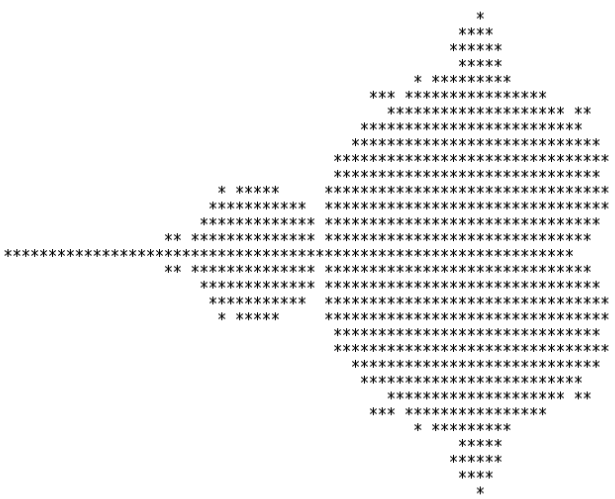
The lower limit at the negative end is somewhat further from zero: you can get as far down as -2 before values begin to diverge.

However the fun really starts when we bring complex numbers to the party. The iteration we describe is usually written in this form:

$$\boldsymbol{z_{n+1} = (z_n)^2 + c}$$

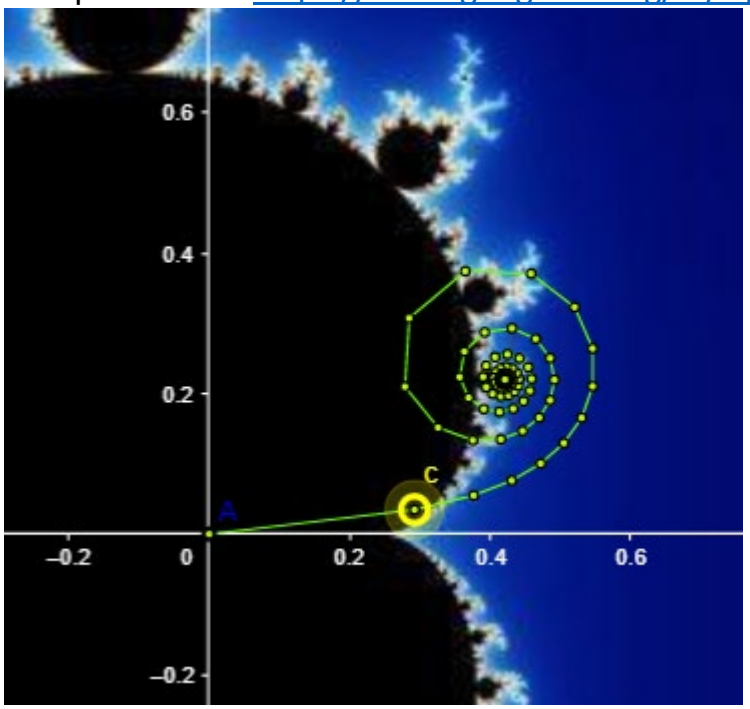
where $z_0 = 0 + 0i$ and c is a complex number.

The set of values of c for which this iterative process converges is known as the Mandelbrot Set, after Benoit Mandelbrot who found the first visualisation of the set by plotting the values of c for which the iteration converges, on an IBM machine in 1980. He had to run the program twice because a technician thought the printer was playing up the first time and threw out his now famous visualisation.



Investigating the Mandelbrot Set

You can view a nice GeoGebra demonstration of convergence for individual points made by Ben Sparks here: https://www.geogebra.org/m/Npd3kBKn



Notice how values close to 0.25, but complex, eventually converge to a point which here has been shaded in black.

The more closely we examine the boundary of this set, the more curious and complicated it appears to become. Mandelbrot's original diagram hints at the complexity, but it wasn't until modern computing that people were finally able to view the whole fractal beauty of this curious set.

Working with complex numbers in p5

It's not necessary to have a dedicated 'complex number' data type in order to perform complex number computations. In fact, since the only operations required for this project are squaring and adding, we can essentially store our complex numbers as two separate variables (a real part and an imaginary part), and tell p5 directly what the new complex number should be.

Let c be the complex number $a + bi$.
Let z be the complex number $x + yi$. *Note that, initially,* $z = 0 + 0i$*, but this will change as the iterative process proceeds.*

Using $c = a + bi$ and $z = x + yi$, calculate the following, giving your answers in rectangular form:

$z^2 + c =$

Answer:

$$i(q + xz2) + (v + {}_2y - {}_2x) = iq + v + {}_2y - xyi2 + {}_2x = (iq + v) + {}_2(yi + x) = c + {}_2z$$

The meaning of convergence for a computer

In general, it's non-trivial to determine whether a particular iterative process will converge (that is, approach, eventually, a fixed value). The mathematical definition requires that there is some point beyond which all values lie arbitrarily close to a particular number. Computationally, we will do the equivalent of mashing the = key on the calculator and seeing whether the values 'blow up' to some impossibly huge number.

```
7  let maxDepth = 50;
8
9  function depth(a, b){
10   let x = 0;
11   let y = 0;
12   for (let i=0; i<maxDepth; i++){
13     newX = x**2 - y**2 + a;
14     newY = 2*x*y + b;
15     x = newX;
16     y = newY;
17     if (x**2 + y**2 > 2**2){
18       return i;
19     }
20   }
21   return maxDepth;
22 }
```

This function takes the input values a and b, which will correspond to the real and imaginary parts of a given input value $c = a + bi$, and, starting from $z = 0 + 0i$ (that is, $x = 0, y = 0$), repeatedly applies the iteration $z = z^2 + c$ using the formulae we constructed.

The *maxDepth* is set to 50, meaning that if the value hasn't diverged after 50 iterations, we're going to treat it as being a member of the set. Note one other important detail, though: the function doesn't just return a 'yes' or 'no': it gives the 'depth', which indicates how quickly the value diverges (small numbers: very quickly, close to 50, took a while to get too big).

I'm using a radius of 2 here for my 'too big' condition (line 17): if the modulus of the complex number ever gets larger than 2, I'm going to stop the process and return the number of iterations up to that point.

Setting the scale and setting up the pixel array

The canvas will be a few hundred pixels across, and if we don't change the scaling for our complex numbers, we'll get a very small image, since numbers in the Mandelbrot set are never more than 2 units from the origin.

```
1  let xMin = -2;
2  let xMax = 2;
3  let yMin = -2;
4  let yMax = 2;
```

By setting minimum and maximum values for the x and y coordinate axes, I can convert from number of pixels across or down to a complex number within a relatively small area.

```
24 function setup() {
25   createCanvas(400, 400);
26   pixelDensity(1);
27   loadPixels();
28 }
```

Every single pixel in our display will be treated as a complex number, and coloured according to whether it is in the Mandelbrot set, and if not, how quickly it diverges. Set *pixelDensity* to 1 to avoid issues with HD displays.

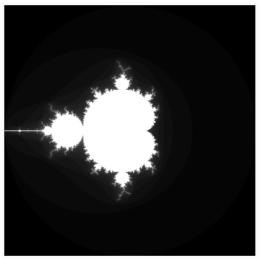
Iterating through the pixels

```
30 function draw() {
31   background(0);
32   for (let j=0; j<height; j++){
33     for (let i=0; i<width; i++){
34       let index = 4*(j*width + i);
35       let a = map(i, 0, width, xMin, xMax);
36       let b = map(j, 0, height, yMax, yMin);
37       let d = depth(a, b);
38       let col = map(d, 0, maxDepth, 0, 255);
39       pixels[index+0] = col;
40       pixels[index+1] = col;
41       pixels[index+2] = col;
42       pixels[index+3] = 255;
43     }
44     updatePixels();
45   }
46 }
```

Recall the nested j and i loops, and the *index* which converts from pixels across and down to a position within the pixel array. We use *map* to turn the distance across from a number of pixels to a value within our valid range, and the same for down. Next, *depth* is used to get information about convergence for the complex number in question, and finally the brightness is set by mapping depth onto a black-to-white scale. If the value doesn't diverge, it'll show up white (255), and the more quickly it diverges, the closer to black (0) it will be.

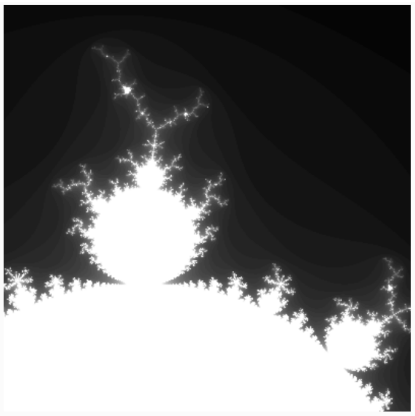
Zooming in…

The code we've made so far faithfully reproduces the familiar shape of the Mandelbrot set. If you look closely, you may detect some of the subtle features that Mandelbrot himself was unable to view on his (much lower resolution) image.



If we want to see these details more closely, we could simply change *xMin, xMax* etc to zoom in on a particular section:

```
let xMin = -0.4;
let xMax = 0.4;
let yMin = 0.4;
let yMax = 1.2;
```



I've carefully kept the width and height the same so that the image isn't stretched unequally vertically and horizontally. But it would be even better if we could dynamically zoom in on any part of the image that took our fancy.

Dynamic zoom

```
let zooming = 0;
```

Define a variable at the start which will keep track of whether we are zooming in (1), out(-1) or not zooming at all (0).

```
function draw() {
  applyZoom(1.1);
  background(0);
```

Add a line into the *draw* function, which should take care of updating the *xMin, xMax, yMin* and *yMax* values.

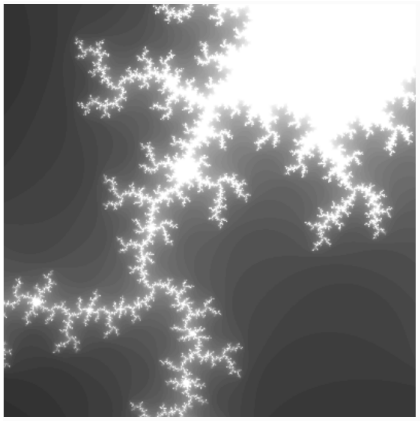
```
function applyZoom(sf){
  if (zooming == 0){return false};
  if (zooming == 1){sf = 1/sf}
  let x = map(mouseX, 0, width, xMin, xMax);
  let y = map(mouseY, 0, height, yMax, yMin);
  xMin = x - (x-xMin)*sf;
  xMax = x + (xMax-x)*sf;
  yMin = y - (y-yMin)*sf;
  yMax = y + (yMax-y)*sf;
}
```

The *applyZoom* function terminates if *zooming* is zero (ie, no zoom), and replaces the scale factor with its reciprocal if zooming in rather than out. Notice how the mouse position is used as the reference point, and boundaries updated by scaling down their distance from that reference point. See if you can make sense of these calculations.

```
function mousePressed(){
  zooming++;
  if (zooming > 1){
    zooming -=3;
  }
  return false;
}
```

Finally, the value of the *zooming* variable can be changed by clicking: a mouse click will increment its value, adjusting if it gets too high, so you can cycle through 'zoom in', 'zoom out' and 'stay put'.
Note that some browsers have issues if the *mousePressed* function doesn't return anything, so use *return false* to ensure things work.

Now, even though the size of our canvas may be limited by the efficiency of the pixel array updates, we can click to zoom in and investigate ever more tiny variations around the boundary of the Mandelbrot set.



What else?

There are many variations possible for this fractal pattern. If you can simplify $z^3 + c = (x + yi)^3 + (a + bi)$ you could explore some more variants on this set. For higher powers, you may find it helpful to build in some functions that convert rectangular to mod-arg form…