**Mandebrot Pseudocode**

Pseudocode to draw the Mandelbrot set using the 'escape time' algorithm, as adapted from [Wikipedia](https://en.wikipedia.org/wiki/Mandelbrot_set#Computer_drawings).

**Overview**

The main gist is that, for each pixel, you repeatedly apply an operation to the x,y numbers. If the result heads towards infinity, the x,y point is not withing the set. If it does not, the point is in the set. (So the numbers in the set are the black, beetle-shaped bit in the middle).

As the concept of 'heading towards infinity' is a bit tricky to encode, the way it's done here is to just do the operation a load of times (e.g. 100) and see if the number stays within a certain limit. If, after 100 iterations, the number is still within the limit, you count it as being in the set. If it exceeds the limit before then, it's not in the set.

**Scaling**

One tricky aspect is that you need to convert between your output resolution (e.g. 80x25 'pixels' for a console window) and the important area of the Mandelbrot scale, which is roughly between -2.5 and 1.0 on the X axis, and -1.0 and 1.0 on the Y Axis.

An example of this would be, for the pixel 40, 5:

40, on the source scale 0..80, becomes -0.75 on the target scale -2.5..1.0 5, on the source scale 0..25, becomes -0.6 on the target scale -1.0..1.0

**Pseudocode**

For each pixel (xPixel, yPixel) on the screen, do:

{

xScaled = xPixel scaled to lie in the Mandelbrot X scale (-2.5, 1)

yScaled = yPixel scaled to lie in the Mandelbrot Y scale (-1, 1)

x = 0.0

y = 0.0

iteration = 0

max\_iteration = 100

while ( x\*x + y\*y < 2\*2 AND iteration < max\_iteration )

{

xNext = x\*x - y\*y + xScaled

y = 2\*x\*y + yScaled

x = xNext

iteration = iteration + 1

}

colour = palette[iteration]

plot(xPixel, yPixel, colour)

}

To scale a number from a source scale to a target scale given sourceScaleMax, targetScaleMin, targetScaleMax:

{

positionInSourceScale = number / sourceScaleMax

scaledNumber = positionInSourceScale \* (targetScaleMax - targetScaleMin) + targetScaleMin

}

**Colouring**

To make it prettier, you can use different colours/characters/etc for the pixels which aren't in the set depending on which iteration they got to before the number exceeded the limit.

There are a few difference techniques which can be used.

The **easiest one** to start with is to

* Create an array of n colours
* To determine the colour of a pixel, you first calculate the number of iterations i; for that pixel.
* You then index into the array of colours using colours[i mod n]

**Improved Colours**

The more classic Mandelbrot colouring can be obtained by

* Calculate the number of iterations for a given pixel as before.
* Normalise this value into the range 1 to 360 to give you the “Hue”.

var hue = (360.0 / MAX\_ITERATION) \* iteration

* Pick the HSV colour, using 1 for the Saturation and 1 for the Value

[**Histogram**](https://en.wikipedia.org/wiki/Histogram)

A more accurate coloring method involves using a [histogram](https://en.wikipedia.org/wiki/Histogram), which keeps track of how many pixels reached each iteration number, from 1 to n. This method will equally distribute colors to the same overall area, and, importantly, is independent of the maximum number of iterations chosen.

First, create an array of size n. For each pixel, which took i iterations, find the ith element and increment it. This creates the histogram during computation of the image. Then, when finished, perform a second "rendering" pass over each pixel, utilizing the completed histogram. If you had a continuous color palette ranging from [0.0, 1.0], you could find the normalized color of each pixel as follows, using the variables from above.

total = 0

for (i = 0; i < max\_iterations; i += 1)

{

total += histogram[i]

}

hue = 0.0;

for (i = 0; i < iteration; i += 1)

{

hue += histogram[i] / total // Must be floating-point division.

}

color = palette[hue]

**Continuous (smooth) coloring**

The algorithm for this is described on Wikipedia, but looks a bit more complicated!

**Zooming**

Mandelbrot Sets start to become more impressive as your zoom in. I implemented this by changing the output resolution used when calculating the scale. You then just work with the portion of pixels which are visible on the screen.

**Other Ideas**

* **Implement using graphics instead of ASCII**

If you environment supports it, try creating a graphical version of the Mandelbrot set.

* **Ability to spin the image around.**
* **Add a virtual tour around the interesting points**
* **Implement the algorithm using a complex number type**