How To Write A Book
About The Mandelbrot Set

Claude Heiland-Allen

April 25, 2014

Contents

1 The Exterior **7**

1.1 A minimal C program. 7
1.2 Building a C program. 7
1.3 Ignoring generated files. 8
1.4 Input and output. 8
1.5 Printing all arguments: a “for” loop. 9
1.6 Device coordinates. 9
1.7 User coordinates. 10
1.8 Aspect ratio. 12
1.9 Complex numbers. 13
1.10 The Mandelbrot set. 14
1.11 Converting strings to numbers. 16
1.12 Image output. 18
1.13 Escaping a loop early: “break”. 21
1.14 Avoiding square roots. 21
1.15 Escape time colouring. 22
1.16 Refactoring. 23
1.17 Double precision. 25
1.18 Generic programming with preprocessor macros. 28
1.19 Colour images. 33
1.20 Decoupling rendering and colouring. 47
1.21 Decoupling rendering from image output. 53
1.22 Parallel rendering with OpenMP. 56
1.23 Final angle colouring. 57
1.24 Fixing bugs. 63
1.25 Defensive error handling. 65
1.26 Automatic number type selection. 68
1.27 Final radius colouring. 76
1.28 Smooth colouring with continuous escape time. 83
1.29 Generating a grid. 88
1.30 Improving the grid. 88
1.31 Distance estimation. 89
1.32 Removing distracting colour. 99
1.33 Refactoring image stream reading. 99
1.34 Annotating images with Cairo. 103

2 The Interior **107**
2.1 Diagnosing attractors. 107
2.2 Atom domains 112
2.3 Nucleus basins 115

3 Notebook **123**
3.1 Newton's Method: Nucleus 123
3.2 Newton's Method: Wucleus 123
3.3 Newton's Method: Bond 123
3.4 Newton's Method: Ray In 124
3.5 Newton's Method: Ray Out 124
3.6 Newton's Method: Preperiodic 124
3.7 Derivatives Calculation 125
3.8 External Angles: Bulb 125
3.9 External Angles: Hub 125
3.10 Farey Numbers 126
3.11 External Angles: Tuning 126
3.12 External Angles: Tips 126
3.13 Translating Hubs Towards Tips 126
3.14 Islands In The Spokes 127
3.15 Islands In The Hairs 127
3.16 Islands In Embedded Julias 127
3.17 Multiply Embedded Julias 127
3.18 Continuous Iteration Count 128
3.19 Interior Coordinates 128
3.20 Exterior Distance 128
3.21 Interior Distance 128
3.22 Perturbation 129
3.23 Finding Atoms 129
3.24 Child Size Estimates 129
3.25 Atom Shape Estimates 130
3.26 Atom Size Estimates 130
3.27 Tracing Equipotentials 130
3.28 Buddhabrot 130
3.29 Atom Domain Size Estimate 130
3.30 Stretching Cusps 130

4 Bonds. **133**
4.1 Prerequisites 133
4.2 Finding features 134
4.3 Recurrence relations 134
4.4 Implementation 135
4.5 Proofs 136

5 Perturbation. **141**
5.1 The Mandelbrot set 141
5.2 Perturbation techniques 141
5.3 Applying the technique 142
5.4 Series approximation 143

Listings

78b19af2c44dd3e379769b4c15c7bd926ca52482.diff 7
4ee49c5a4ca68969c155647637846d58ffd5505f.diff 7
19db4ccaf9ba37f2ce1c5304d6f2ae57bd74b6cb.diff 8
a0fe15d74f588c4d8ab6654c10ed0278a5c75b09.diff 8
c3aed969988c9261a66b23648fab8f8449cabff8.diff 9
1e7226121c5ef78a57b285686e7d1c52f4754b79.diff 10
952ffad52923291b390587dcc90c0c7d1aefbb3e.diff 11
80fce2bd7e0835cdbc137084da594b3fdb348b47.diff 12
912c346b11b8be2447c56e8152c0389385aee885.diff 14
bcba5f0deb472299e17b96e35fe514992496ce30.diff 15
024682465b9a56ec28bceb65e877d6e5636effcd.diff 17
c71fb8b6c4775c6087fa23173a1613abefcb197a.diff 20
73850281d75235cac5dc192a2f559f5fba36ec7d.diff 21
66567c93430e53142108cdde4b6551ebebd8c1cb.diff 22
c1667b27527c2eeb3f3fa12ead01b6276f63d627.diff 23
7b71abc5e0df24258b1bfe47369018995061a09f.diff 23
ecb18f012205e586c683a15c5829d71d06dcdb32.diff 26
4ecccb39c916dca81a60860029b23aeb62a2a1b7.diff 29
702198f081c208957d77323fac0b713c131ab422.diff 45
24e8b81a7b49818148e6a0c719e70ad53a8fca9c.diff 48
706785ce2684df3c7bae5caaa92885fcfef14300.diff 54
9dfc508d2c4af17bcd951c419487411ebca4b8d7.diff 56
99e73f934ddd79ebdbbb2f6d929e6ffd3562df9a.diff 58
64692c684bf50d22bed4f4cc4303dba070243788.diff 64
20ff08458c29dc339a363cb155fb2e1a77f5c302.diff 65
bd7aa445bfe2b02a2883dccdb561c58dcc620aeb.diff 75
bbd262f1df0b02d41c829800db414930c3f94e54.diff 80
fb98ed31d9265937092682c1f04d0c34aecff54f.diff 87
00d0fa5735a6b5f2ddeb0d8453a7403594a7baa0.diff 88
6255e2b5e432f65e1318d0704b9dbe3b1b0041a8.diff 89
e7ff7b7158c999b8fcc24e627c39226233181e38.diff 96
52a33c0fe498cf918bd6c6ce8324c2697fb44546.diff 99
6b5aebb7c5e6267a4c187bd38fc92151cab366c0.diff 100
643106f7ca1a9fcb260984b8ae53acd0fac07454.diff 103
a728ca1ae906061981ee1e7fd65eb46efc80c513.diff 111
fdc8ed916d0158b5a67ccffaa124a18c4be20226.diff 112
aaa5c9f64c859178b82e4758956dc3a965cc8312.diff 119

Chapter 1

The Exterior

1.1 A minimal C program.

Every program needs an entry point – in the programming language C this is a function called "main". When the program has finished running, it exits, returning a status code to the operating system to signal success (with 0) or failure (non-zero values).

```
index 0000000..4cce7f6
--- /dev/null
+++ b/mandelbrot.c
@@ -0,0 +1,3 @@
+int main() {
+  return 0;
+}
```

1.2 Building a C program.

A computer can't run a C program without translating it to machine code for the CPU, a task performed by a compiler. The GNU Compiler Collection includes the gcc compiler for C. We'll be using the C99 standard dialect of C. We'll enable every warning, so that we get hints about possible bugs. We'll set optimization to a high level, so that we don't have to wait so long for the program to run (it might take longer to compile).

A Makefile contains rules for building targets from prerequisites. Running "make" will check if the default target is up to date compared to its prerequisites, and if not it will run the commands listed in the rules. This happens recursively, so that if you change one file in a large project, only the affected path through the tree of dependencies will be rebuilt.

```
diff --git a/Makefile b/Makefile
new file mode 100644
index 0000000..9d426ea
--- /dev/null
+++ b/Makefile
@@ -0,0 +1,2 @@
+mandelbrot: mandelbrot.c
+	gcc -std=c99 -Wall -Wextra -pedantic -O3 -o mandelbrot mandelbrot.c
```

1.3 Ignoring generated files.

A version control system allows you to keep track of the change history of a collection of files over time. Git is one of the more popular tools. The “git status” command gives a brief summary of what has changed since the last “git commit”, for example, listing files that haven’t been added to the repository. Some of these files might be temporary or system-specific – for example, compiled programs. Adding them to the “.gitignore” file makes the status output less noisy, allowing us to focus on important changes.

```
diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..c0729ec
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1 @@
+mandelbrot
```

1.4 Input and output.

A program that returns a status code isn’t very interesting. To interact with the outside world needs input from the world and output to the world. The C standard library supports this within the “stdio.h” header file. One function declared in the header is “puts()”, which outputs a string (a sequence of characters in memory, terminated by the null character “ '\0' ”.

A program can also take input from its command line arguments, these are passed as parameters to “main”: the first parameter is the integer number of command line arguments, and the second contains the arguments themselves. The collection of arguments and the count includes the program name as the first argument.

The count has the type “int”, a small integer: the mathematical integers extend to infinity, but computers are based on small fixed-width numbers and C keeps at a fairly low level of abstraction above the hardware.

In C, “*” denotes a pointer type. Pointers are references to areas of memory containing data. The arguments are made up of characters stored in consecutive memory locations, and they are passed to the program as a pointer to consecutive memory locations, each containing another pointer to the first character in each argument. This gives the type “char **”, the names “argc” and “argv” are just a common convention..

Getting the data from the memory location referred to by a pointer is called “dereferencing”, and in C you can use “*” (as a unary operator this time) to get the location pointed to, or “[n]” to get the n’th item, with each item stored consecutively in memory. In C, counting starts from 0 for the first item.

The program now prints the name used to invoke it to its standard output stream.

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 4cce7f6..fc0453b 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -1,2 +1,5 @@
-int main() {
+#include <stdio.h>
+
+int main(int argc, char **argv) {
+  puts(argv[0]);
```

```
    return 0;
```

1.5 Printing all arguments: a “for” loop.

So far the program just prints its name, and ignores any other arguments. Suppose we want to print all of them. The idiomatic way to do this is to write a loop.

The “for” loop syntax has three clauses. In the first we declare and initialize a counter variable. This happens once, before any looping happens.

In the second clause we check that it’s within the valid range of argument indices. If the condition is false, then the control flow skips beyond the end of the loop body without executing it.

If the condition clause is true, then the loop body is executed. In our program, we print the i’th command line argument. When the control flow reaches the end of the loop body, the third clause is executed before going back to the start of the loop: in our program we increment the counter. This process repeats so long as the condition clause remains true.

An example invocation of our program might look like

```
$ ./mandelbrot one two three
./mandelbrot
one
two
three
$
```

where “$” is the command prompt.

```
diff --git a/mandelbrot.c b/mandelbrot.c
index fc0453b..5b99215 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -3,3 +3,5 @@
 int main(int argc, char **argv) {
-  puts(argv[0]);
+  for (int i = 0; i < argc; ++i) {
+    puts(argv[i]);
+  }
   return 0;
```

1.6 Device coordinates.

Western text is written left to right in rows from top to bottom. For text-based images, this gives an origin (0,0) at the top left, with the next character to the right at (1,0) and the first character in the line below at (0,1). We can generate a table of coordinates with two nested for loops: the outer one loops over rows, and the inner one loops over columns within each row.

We would print the coordinates in a table, but so far we only know how to print strings. The C standard library includes the function “printf()” for formatted output. Its first parameter is a format string, which contains conversion specifiers prefixed by “%”. You can read the full list in its reference manual by running “man 3 printf”.

We want to print our int coordinates in decimal, so we put “%d” in the format string, once for each coordinate. printf() is variadic: the number of parameters and the type of each parameter

depends on the format string. We want to have each row on a separate line, so we need to emit a newline character " '\n' " at the end of each row. The "putchar()" function prints a single character.

Our program now outputs a table of device coordinates:

```
$ ./mandelbrot
(0,0)(1,0)(2,0)(3,0)(4,0)(5,0)(6,0)(7,0)(8,0)(9,0)
(0,1)(1,1)(2,1)(3,1)(4,1)(5,1)(6,1)(7,1)(8,1)(9,1)
(0,2)(1,2)(2,2)(3,2)(4,2)(5,2)(6,2)(7,2)(8,2)(9,2)
(0,3)(1,3)(2,3)(3,3)(4,3)(5,3)(6,3)(7,3)(8,3)(9,3)
(0,4)(1,4)(2,4)(3,4)(4,4)(5,4)(6,4)(7,4)(8,4)(9,4)
(0,5)(1,5)(2,5)(3,5)(4,5)(5,5)(6,5)(7,5)(8,5)(9,5)
(0,6)(1,6)(2,6)(3,6)(4,6)(5,6)(6,6)(7,6)(8,6)(9,6)
(0,7)(1,7)(2,7)(3,7)(4,7)(5,7)(6,7)(7,7)(8,7)(9,7)
(0,8)(1,8)(2,8)(3,8)(4,8)(5,8)(6,8)(7,8)(8,8)(9,8)
(0,9)(1,9)(2,9)(3,9)(4,9)(5,9)(6,9)(7,9)(8,9)(9,9)
$
```

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 5b99215..63a6850 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -3,4 +3,9 @@
 int main(int argc, char **argv) {
-  for (int i = 0; i < argc; ++i) {
-    puts(argv[i]);
+  int width = 10;
+  int height = 10;
+  for (int j = 0; j < height; ++j) {
+    for (int i = 0; i < width; ++i) {
+      printf("(%d,%d)", i, j);
+    }
+    putchar('\n');
   }
```

1.7 User coordinates.

Device coordinates are integer pairs in $[0, width) \times [0, height)$. We now transform the device coordinates to "user coordinates" giving us a frame of reference more appropriate to our needs. Suppose we want to draw a filled circle defined by $x^2 + y^2 < r^2$. This circle is centered on (0,0) in user space, but the center of device space is (width/2, height/2). Let's pick $r < 1$, then the whole circle will be inside a box $[-1, 1] \times [-1, 1]$. We transform the device coordinates (i,j) to user coordinates (x,y), to make drawing the circle easier.

We need fractional numbers now, for which C has the "float" type. Floating point arithmetic stores a number of significant digits, and a scaling exponent (like scientific notation). We use floating point constants "2.0", otherwise "/" performs integer division, and we also exploit the fact that C "promotes" the other operands to floating point as necessary.

With this coordinate transformation, our circle looks like this:

```
$ ./mandelbrot
```

```
------------------------------------------------------------------------
------------------------------------------------------------------------
-----------------------xxxxxxxxxxxxxxxxxxxxxxxxxxx----------------------
-----------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------------
--------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-------------
-----------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------
---------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx--------
-------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx------
------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-----
-----xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----
----xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx---
----xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx---
----xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx---
-----xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----
------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-----
-------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx------
---------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx--------
-----------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------
--------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-------------
-----------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------------
-----------------------xxxxxxxxxxxxxxxxxxxxxxxxxxx----------------------
------------------------------------------------------------------------
$
```

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 63a6850..d31276f 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -3,7 +3,14 @@
 int main(int argc, char **argv) {
-  int width = 10;
-  int height = 10;
+  int width = 72;
+  int height = 22;
+  float r = 0.9;
   for (int j = 0; j < height; ++j) {
     for (int i = 0; i < width; ++i) {
-      printf("(%d,%d)", i, j);
+      float x = (i - width /2.0) / (width /2.0);
+      float y = (j - height/2.0) / (height/2.0);
+      if (x * x + y * y < r * r) {
+        putchar('x');
+      } else {
+        putchar('-');
+      }
     }
```

1.8 Aspect ratio.

The previous program's output failed to look like a circle for two reasons. First, the width and height are unequal, so scaling each device coordinate separately to [-1,1] will distort the shape. We correct this distortion by choosing one of the axes to scale to [-1,1], we'll choose the vertical direction becasue our image is wider than it is tall and we want to have the whole circle in the image. Dividing both coordinates by the same height/2.0 will make the y axis range [-1,1] and the x axis range [-k,k] with $k > 1$.

The second reason our circle doesn't look circular is that the characters displayed are not square. This does depend on the particular font used, but common terminal fonts have characters roughly half as wide as they are tall. This makes the spacing between characters is half as much in the horizontal direction than the vertical direction. We correct the spacing by multiplying by the aspect ratio.

Now our circle looks much more circular:

```
$ ./mandelbrot
------------------------------------------------------------------------
------------------------------------------------------------------------
----------------------------xxxxxxxxxxxxxxxxx---------------------------
-------------------------xxxxxxxxxxxxxxxxxxxxxxx------------------------
----------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxx---------------------
---------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx--------------------
-------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx------------------
------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-----------------
------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-----------------
-----------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------------
-----------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------------
-----------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------------
-----------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------------
-----------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx----------------
------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-----------------
------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-----------------
-------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx------------------
---------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx--------------------
----------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxx---------------------
-------------------------xxxxxxxxxxxxxxxxxxxxxxx------------------------
----------------------------xxxxxxxxxxxxxxxxx---------------------------
------------------------------------------------------------------------
$
```

```
diff --git a/mandelbrot.c b/mandelbrot.c
index d31276f..38f2a5f 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -5,2 +5,3 @@ int main(int argc, char **argv) {
   int height = 22;
+  float aspect = 0.5;
   float r = 0.9;
@@ -8,3 +9,3 @@ int main(int argc, char **argv) {
     for (int i = 0; i < width; ++i) {
```

```
-        float x = (i - width/2.0) / (width/2.0);
+        float x = (i - width/2.0) / (height/2.0) * aspect;
         float y = (j - height/2.0) / (height/2.0);
```

1.9 Complex numbers.

Different kinds of numbers can be thought of as a tower. Counting gives us the whole numbers, then the desire for an identity element for addition gives us zero:

$$0 + x = x + 0 = x$$

Inverting addition gives subtraction, which needs negative numbers:

$$x + (-x) = (-x) + x = 0$$

Repeated addition gives multiplication, with identity 1, and inverting multiplication gives division which needs fractions:

$$n \times (1/n) = (1/n) \times n = 1$$

Repeated multiplication gives powers, like $n^2 = n \times n$, and fractional powers give irrational numbers like $2^{\frac{1}{2}} = \sqrt{2} = 1.41421\ldots$ which can't be represented by a fraction.

Taking the square root of a negative number can't give any of the kinds of numbers we've seen so far, because it's easy to work out that a negative number multiplied by a negative number gives a positive number, and so does a positive number multiplied by a positive number. So we make up a new imaginary number i so that

$$i \times i = -1$$

and then we have complex numbers with two parts: the sum of a real part and an imaginary part. The rules for complex number arithmetic follow from the definition of i and the rules for regular arithmetic (which is associative, commutative, and distributive):

$$(a + b) + c = a + (b + c)(a \times b) \times c = a \times (b \times c)$$

$$a + b = b + aa \times b = b \times a$$

$$a \times (b + c) = (a \times b) + (a \times c)$$

Happily the C99 language has library support for complex numbers in "complex.h", so we can include it and then use complex floating point types, using `I` for i, and regular arithmetic operators. We now need to link with the C maths library ("libm") which contains the implementations of the functions declared in the header, so we add -lm to our Makefile rule. Now we can draw our circle $|c| < r$ using the "cabs()" function.

The whole numbers without zero are written $\mathbb{N}^+$, whole numbers with zero $\mathbb{N}$ (the N is for natural). The whole numbers with negative numbers (the integers) are written $\mathbb{Z}$ (the Z is for Zahlen, the German word for numbers). Fractions (the rational numbers) are written $\mathbb{Q}$ (the Q is for quotient, the result of division).

Adding roots of polynomials gives the algebraic numbers, and (a subset of) the complex numbers. It's possible to show that there are still more numbers, for example π and e (the base of natural logarithms) are transcendental: eventually you reach the continuum of the real number line written $\mathbb{R}$, and the complex plane written $\mathbb{C}$.

```
diff --git a/Makefile b/Makefile
index 9d426ea..8c548fd 100644
--- a/Makefile
+++ b/Makefile
@@ -1,2 +1,2 @@
 mandelbrot: mandelbrot.c
-	gcc -std=c99 -Wall -Wextra -pedantic -O3 -o mandelbrot mandelbrot.c
+	gcc -std=c99 -Wall -Wextra -pedantic -O3 -o mandelbrot mandelbrot.c -lm
diff --git a/mandelbrot.c b/mandelbrot.c
index 38f2a5f..5fed547 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -1 +1,2 @@
+#include <complex.h>
 #include <stdio.h>
@@ -11,3 +12,4 @@ int main(int argc, char **argv) {
       float y = (j - height/2.0) / (height/2.0);
-      if (x * x + y * y < r * r) {
+      complex float c = x + I * y;
+      if (cabs(c) < r) {
         putchar('x');
```

1.10 The Mandelbrot set.

Pick a complex number c. Starting from $z_0 = 0$, form a sequence of "iterates" by repeating $z_{n+1} = z_n^2 + c$. For some values of c, the iterates z_n get bigger and bigger, escaping towards infinity. These iterates are unbounded: for any size you pick, there'll be some later iterate that is bigger still. For other values of c, the iterates remain bounded. The Mandelbrot set is a map of the parameter plane, telling us which c values will give unbounded iterates, and which remain bounded. Mathematically, the Mandelbrot set consists of the c values that remain bounded. What does it look like?

It's possible to show that if an iterate gets bigger than 2, then it'll escape to infinity. So the whole Mandelbrot set must be inside a circle $|c| <= 2$. One way to check that a point is not in the Mandelbrot set is to keep iterating and see if it escapes. But for any particular c value, we don't yet know if it is in the Mandelbrot set or not – if it turns out that it is in the set, we would keep iterating forever. So that we don't have to wait beyond the end of time to see our picture of the Mandelbrot set, we choose to give up after a certain maximum iteration count. If the point still hasn't escaped, it might or might not be in the Mandelbrot set. But if the point escaped, we can be sure the point is not in the Mandelbrot set. We can draw these two cases with two different characters, giving us our first approximation to the Mandelbrot set.

```
$ ./mandelbrot
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
------------------------------------------------------------------------
----------------------------------xx------------------------------------
-------------------------------xxxxxxxxx--------------------------------
-----------------------------xxxxxxxxxxx--------------------------------
-----------------------xxxxxxxxxxxxxxxxxx-------------------------------
--------------xxxxxxxxxxxxxxxxxxxxxxxxx---------------------------------
-----------------------xxxxxxxxxxxxxxxxxx-------------------------------
-----------------------------xxxxxxxxxxx--------------------------------
-------------------------------xxxxxxxxx--------------------------------
----------------------------------xx------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
$
```

Mathematical variables are immutable. In maths, $z = z^2 + c$ is an equation, and you solve it by finding a value for z that makes the value on both sides the same. For example, for $c = 0$ there are two solutions: $z = 0$ and $z = 1$. In the C programming language, variables can change as the program runs. We already saw this with the loop counters, in our use of the increment operator `++i`. In C, `=` is an "assignment" operator, not mathematical equality (for which C uses `==`). Variables in C are like boxes that contain values. On the right hand side of `=`, values are taken out of the boxes and used to calculate the result of the expression. Then `=` takes that result and stores it into the box on the left hand side, overwriting the previously stored value. The use of z as an "l-value" on the left hand side of `z = z * z + c` has a different meaning to the usees of z as an "r-value" on the right hand side.

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 5fed547..0e9943d 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -7,3 +7,6 @@ int main(int argc, char **argv) {
   float aspect = 0.5;
-  float r = 0.9;
+  complex float center = 0;
+  float radius = 2;
+  int maximum_iterations = 256;
+  float escape_radius = 2;
   for (int j = 0; j < height; ++j) {
@@ -12,4 +15,8 @@ int main(int argc, char **argv) {
       float y = (j - height/2.0) / (height/2.0);
-      complex float c = x + I * y;
-      if (cabs(c) < r) {
+      complex float c = center + radius * (x + I * y);
+      complex float z = 0;
+      for (int n = 0; n < maximum_iterations; ++n) {
+        z = z * z + c;
+      }
+      if (cabs(z) <= escape_radius) {
         putchar('x');
```

1.11 Converting strings to numbers.

The region of interest in the image we have so far looks a bit small and to the left of the frame. We could edit the program to change the center and radius and recompile it, but that would tire quickly. Instead we'll use the command line arguments. The arguments are passed to main() as strings, but we need numbers. The C library header "stdlib.h" declares a function "atof()" to read a representation of a number in decimal format from a string and convert it to a floating point number.

However, we should first check that we have been given arguments before trying to use them. If there is no string where we try to read a string, we get "undefined behaviour": anything from a program crash (if we're lucky) to carrying on with incorrect data and generating meaningless output. We want three floating point numbers from three arguments, for the real and imaginary parts of the center, and the radius of the view; we get the program name as the first argument, so we need to be sure that we have more than three arguments. In case fewer arguments were provided, we leave the center and radius unchanged at the default values we set previously.

Now we can zoom in to see more of the detail of the Mandelbrot set:

```
$ ./mandelbrot -0.75 0.00 1.25
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
----------------------------------------------xxx-----------------------
----------------------------------------------xxx-----------------------
----------------------------------------xxxxxxxxxxxxx-xxx---------------
----------------------------------------xxxxxxxxxxxxxxx-----------------
--------------------------------------xxxxxxxxxxxxxxxxxxx---------------
-----------------------------xxxxx---xxxxxxxxxxxxxxxxxxxx---------------
----------------------------xxxxxxxx-xxxxxxxxxxxxxxxxxxx----------------
--------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx------------------
----------------------------xxxxxxxx-xxxxxxxxxxxxxxxxxxx----------------
-----------------------------xxxxx---xxxxxxxxxxxxxxxxxxxx---------------
--------------------------------------xxxxxxxxxxxxxxxxxxx---------------
----------------------------------------xxxxxxxxxxxxxxx-----------------
----------------------------------------xxxxxxxxxxxxx-xxx---------------
----------------------------------------------xxx-----------------------
----------------------------------------------xxx-----------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

$ ./mandelbrot -1.50 0.00 0.50
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
----------------------------------------------------------x------------x
---------------------------------------------------xxxxxxxxxxxxx-------x
--------------------------------------------------xxxxxxxxxxxxxxxxx---xx
```

```
-------------------------------------------------xxxxxxxxxxxxxxxxxxx--xx
------------------------------------------------xxxxxxxxxxxxxxxxxxxxx-xx
-------------------------------------------xxxx-xxxxxxxxxxxxxxxxxxxxx-xx
--------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-------------------------------------------xxxx-xxxxxxxxxxxxxxxxxxxxx-xx
------------------------------------------------xxxxxxxxxxxxxxxxxxxxx-xx
-------------------------------------------------xxxxxxxxxxxxxxxxxxx--xx
--------------------------------------------------xxxxxxxxxxxxxxxxx---xx
---------------------------------------------------xxxxxxxxxxxxx-------x
----------------------------------------------------------x------------x
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

$ ./mandelbrot -0.10 0.85 0.25
xxx---------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx---------------
------------xxx---xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-----xx---------------
-------------------x--x---x-xxxxxxxxxxxxxxx-x-x---x---------------------
-----------------------------xxxxxxxxxxx--------------------------------
---------------------------xxxxxxxxxxxxxx-------------------------------
--------------------------xxxxxxxxxxxxxxxxx-----------------------------
--------------------------xxxxxxxxxxxxxxxxx-----------------------------
-------------------------xxxxxxxxxxxxxxxxxx-----------------------------
--------------------------xxxxxxxxxxxxxxxxx-----------------------------
--------------------------xxxxxxxxxxxxxxx-------------------------------
------------------------------xxxxxxxxx---------------------------------
---------------------------------xxxx-----------------------------------
---------------------------------xxxx-----------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
-----------------------------------------x------------------------------
------------------------------------------------------------------------
-------------------------------x----------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 0e9943d..4cfbd54 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -2,2 +2,3 @@
 #include <stdio.h>
+#include <stdlib.h>

@@ -11,2 +12,6 @@ int main(int argc, char **argv) {
     float escape_radius = 2;
+    if (argc > 3) {
```

```
+     center = atof(argv[1]) + I * atof(argv[2]);
+     radius = atof(argv[3]);
+   }
    for (int j = 0; j < height; ++j) {
```

1.12 Image output.

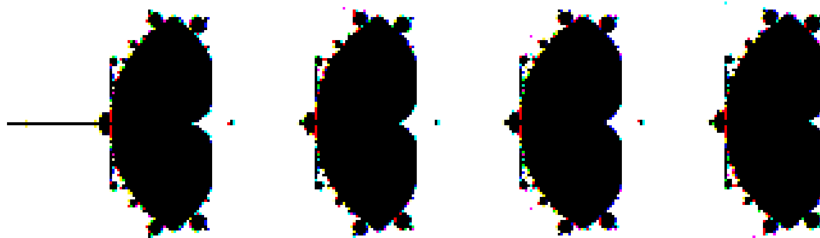
The last image had curious speckles near the bottom, but our text-based image resolution is too small to make them out properly. It's also upside down: mathematical convention dictates that complex numbers are visualized with the imaginary part increasing in the upwards direction. We can fix the flip by negating the imaginary part when transforming from device coordinates.

We want to create images with pixels instead of characters. There are lots of image formats out there, but one of the simplest is a family of related formats called PNM, the "portable anymap file format". We'll use the portable graymap format PGM. PGM files start with a text header, containing metadata about the image. First are the two characters P5 meaning "this is a PGM image", followed by a blank space (typically a newline character). Then come the width and height of the image, as decimal text, separated by spaces. The header ends with the "maxval" as decimal text, which is the value that will represent white (zero represents black), followed by a single newline character. The header is then followed by the image data, in rows from top to bottom with each row from left to right. With a maxval of 255, each pixel takes one byte.

We can increase the size to something a lot bigger than we had before, no longer limited by our command line terminal width. We no longer need to adjust the aspect ratio because PNM formats have square pixels. We can use printf() for the image header text. Because C treats bytes and characters interchangably we can still use putchar(), but now we use 0 and 255 for a black and white image. PNM formats have no separator between rows (there is no need because the width is fixed by the header), so we remove the one we output for text images.

Now our program writes image data, but our terminal expects text. We should "redirect" the output from our program to a file, so that we can open it with an image viewer (such as "display" from the ImageMagick suite). Command line shell redirection uses the ">" character to send the output to a file (overwriting any pre-existing file, so be careful). PGM files usually have a .pgm extension.

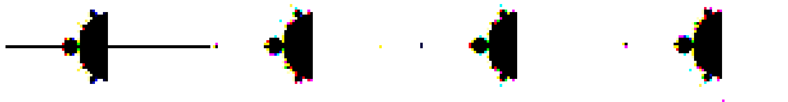
```
$ ./mandelbrot -0.75 0 1.25 > mandelbrot.pgm
```



What's that blob on the left?

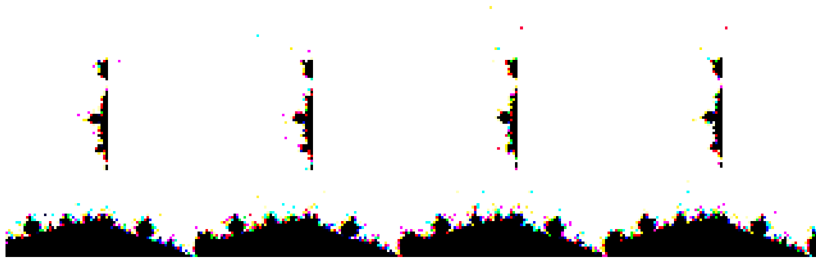
```
$ ./mandelbrot -1.75 0 0.125 > mandelbrot.pgm
```

```
$ ./mandelbrot -1.76 0 0.03 > mandelbrot.pgm
```



It looks like our original image of the whole Mandelbrot set. What about those speckles from before?

```
$ ./mandelbrot -0.1 0.85 0.25 > mandelbrot.pgm
```



They also look like (shrunken, rotated) copies of our original image of the whole Mandelbrot set.

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 4cfbd54..b47ff44 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -5,5 +5,4 @@
 int main(int argc, char **argv) {
-  int width = 72;
-  int height = 22;
-  float aspect = 0.5;
+  int width = 1280;
+  int height = 720;
   complex float center = 0;
@@ -16,7 +15,8 @@ int main(int argc, char **argv) {
   }
+  printf("P5\n%d %d\n255\n", width, height);
   for (int j = 0; j < height; ++j) {
     for (int i = 0; i < width; ++i) {
-      float x = (i - width /2.0) / (height/2.0) * aspect;
+      float x = (i - width /2.0) / (height/2.0);
       float y = (j - height/2.0) / (height/2.0);
-      complex float c = center + radius * (x + I * y);
+      complex float c = center + radius * (x - I * y);
       complex float z = 0;
@@ -26,8 +26,7 @@ int main(int argc, char **argv) {
       if (cabs(z) <= escape_radius) {
-        putchar('x');
+        putchar(0);
       } else {
-        putchar('-');
+        putchar(255);
       }
     }
-    putchar('\n');
   }
```

1.13 Escaping a loop early: “break”.

When we increased the resolution, our program took noticeably longer to render each image. We can speed it up. Our program currently keeps iterating to the maximum iterations limit for every pixel, only then checking if it didn't exceed the escape radius. But if any iterate along the way exceeded the escape radius, we know that it's already well on the way to infinity. If we check each time through the loop, then we can stop looping, with a “break” statement – it makes the control flow pass immediately to the statement following the loop body.

This simple optimization makes our program four times faster for the initial view:

```
$ time ./mandelbrot -0.75 0 1.25 > mandelbrot.pgm
real    0m6.276s
user    0m6.260s
sys     0m0.016s
$ time ./mandelbrot -0.75 0 1.25 > mandelbrot.pgm
real    0m1.593s
user    0m1.584s
sys     0m0.004s
$
```

```
diff --git a/mandelbrot.c b/mandelbrot.c
index b47ff44..57b0b7e 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -24,2 +24,5 @@ int main(int argc, char **argv) {
         z = z * z + c;
+        if (cabs(z) > escape_radius) {
+          break;
+        }
       }
```

1.14 Avoiding square roots.

Our program is still on the slow side. The cabs() function we call to find the absolute value of each iterate is performing a square root:

$$|x + iy| = \sqrt{x^2 + y^2}$$

This square root is unnecessary, because if we know both sides of a comparison are non-negative, we can square both sides and get the same result. Moreover, because the escape radius doesn't change, we can lift its squaring outside all the loops.

Because we'll use this square root avoidance technique more than once, we define a function once, then call it by name. Our cabs2() needs to extract the real and imaginary parts, which we can do using the functions creal() and cimag().

Now our program runs twice as fast as the previous version:

```
$ time ./mandelbrot -0.75 0 1.25 > mandelbrot.pgm
real    0m0.864s
user    0m0.864s
sys     0m0.000s
```

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 57b0b7e..a56e79a 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -4,2 +4,6 @@

+float cabs2(complex float z) {
+  return creal(z) * creal(z) + cimag(z) * cimag(z);
+}
+
 int main(int argc, char **argv) {
@@ -11,2 +15,3 @@ int main(int argc, char **argv) {
   float escape_radius = 2;
+  float escape_radius2 = escape_radius * escape_radius;
   if (argc > 3) {
@@ -24,3 +29,3 @@ int main(int argc, char **argv) {
         z = z * z + c;
-        if (cabs(z) > escape_radius) {
+        if (cabs2(z) > escape_radius2) {
           break;
@@ -28,3 +33,3 @@ int main(int argc, char **argv) {
       }
-      if (cabs(z) <= escape_radius) {
+      if (cabs2(z) <= escape_radius2) {
         putchar(0);
```

1.15 Escape time colouring.

So far we've been colouring in a binary fashion: points possibly maybe inside the Mandelbrot set are black, and points definitely outside are white. We're also breaking out of the loop early as soon as we know a point has escaped. We can add the information about "how many iterations it took before we knew the point escaped" to our images instead of throwing it away. This technique is called "escape time colouring".

First we need a variable to hold the final iteration count. We initialize it to zero, because this is really an impossible value: z_0 is always 0, and 0 is always less than 2. We correct our loop counter initialization: within our loop we perform the iteration step before we check whether the iterate escaped, so the n we need at that point starts from 1. If the point escaped we store the final n before breaking out of the loop. After the loop finishes, we can output the final n directly.

Now the images look even more interesting: the isolated speckles with copies of the whole seem to be connected to the main body by intricate branches and spirals:

```
$ ./mandelbrot -0.75    0       1.25     > mandelbrot.pgm
$ ./mandelbrot -1.76    0       0.05     > mandelbrot.pgm
$ ./mandelbrot -1.765   0.023   0.01     > mandelbrot.pgm
$ ./mandelbrot -1.76648 0.04173 0.0005   > mandelbrot.pgm
$ ./mandelbrot -0.8031  0.1780  0.1      > mandelbrot.pgm
$ ./mandelbrot -0.8031  0.1780  0.01     > mandelbrot.pgm
$ ./mandelbrot -0.8031  0.1780  0.001    > mandelbrot.pgm
$ ./mandelbrot -0.8031  0.1780  0.0001   > mandelbrot.pgm
$ ./mandelbrot -0.8031  0.1780  0.00001  > mandelbrot.pgm
$ ./mandelbrot -0.8031  0.1780  0.000001 > mandelbrot.pgm
```

```
diff --git a/mandelbrot.c b/mandelbrot.c
index a56e79a..611aa50 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -27,5 +27,7 @@ int main(int argc, char **argv) {
       complex float z = 0;
-      for (int n = 0; n < maximum_iterations; ++n) {
+      int final_n = 0;
+      for (int n = 1; n < maximum_iterations; ++n) {
         z = z * z + c;
         if (cabs2(z) > escape_radius2) {
+          final_n = n;
           break;
@@ -33,7 +35,3 @@ int main(int argc, char **argv) {
       }
-      if (cabs2(z) <= escape_radius2) {
-        putchar(0);
-      } else {
-        putchar(255);
-      }
+      putchar(final_n);
     }
```

1.16 Refactoring.

The last two images looked somehow off – grainy and pixelated. We'll investigate this soon, but first let's reorganize our program, breaking it down into smaller pieces. So far the work is all performed in the main() function, which loops over all the pixels, transforms the device coordinates, and calculates the escape time for the point. These three tasks are unrelated, so we split each into its own function, respectively render(), coordinate(), and calculate(). We define render() after the other two, because C needs to have declarations before use, and render() uses both coordinate() and calculate(). This refactoring should make it easier to adapt and extend our program in the future.

```
diff --git a/mandelbrot.c b/mandelbrot.c
index 611aa50..d28847d 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -8,2 +8,33 @@ float cabs2(complex float z) {

+complex float coordinate(int i, int j, int width, int height, complex float center, float radius) {
+  float x = (i - width /2.0) / (height/2.0);
+  float y = (j - height/2.0) / (height/2.0);
+  complex float c = center + radius * (x - I * y);
+  return c;
+}
+
+int calculate(complex float c, int maximum_iterations, float escape_radius2) {
+  complex float z = 0;
+  int final_n = 0;
```

```
+  for (int n = 1; n < maximum_iterations; ++n) {
+    z = z * z + c;
+    if (cabs2(z) > escape_radius2) {
+      final_n = n;
+      break;
+    }
+  }
+  return final_n;
+}
+
+void render(int width, int height, complex float center, float radius, int maximum_iterations, float escape_radius2) {
+  printf("P5\n%d %d\n255\n", width, height);
+  for (int j = 0; j < height; ++j) {
+    for (int i = 0; i < width; ++i) {
+      complex float c = coordinate(i, j, width, height, center, radius);
+      int final_n = calculate(c, maximum_iterations, escape_radius2);
+      putchar(final_n);
+    }
+  }
+}
+
 int main(int argc, char **argv) {
@@ -20,20 +51,3 @@ int main(int argc, char **argv) {
   }
-  printf("P5\n%d %d\n255\n", width, height);
-  for (int j = 0; j < height; ++j) {
-    for (int i = 0; i < width; ++i) {
-      float x = (i - width/2.0) / (height/2.0);
-      float y = (j - height/2.0) / (height/2.0);
-      complex float c = center + radius * (x - I * y);
-      complex float z = 0;
-      int final_n = 0;
-      for (int n = 1; n < maximum_iterations; ++n) {
-        z = z * z + c;
-        if (cabs2(z) > escape_radius2) {
-          final_n = n;
-          break;
-        }
-      }
-      putchar(final_n);
-    }
-  }
+  render(width, height, center, radius, maximum_iterations, escape_radius2);
   return 0;
```

1.17 Double precision.

We've been using C's floating point number type "float". Floating point numbers have a mantissa and an exponent, similar to scientific notation for decimal numbers like 1.234e-5. The mantissa has a fixed width, which means there are only so many significant figures that can be represented exactly. The "float" type is precise to about 7 decimal digits, having 24 bits (to convert from bits to digits we multiply by $log2/log10$, and the exponent says where the significant bits begin (leading 0s are not significant).

Recall our last pixellated image:

```
$ ./mandelbrot -0.8031  0.1780  0.000001 > mandelbrot.pgm
```

Using the device coordinate transformation, the distance between adjacent pixels in the image is 0.000001/(720/2) = 2.78e-9. Comparing this with the image center shows us why the image is pixelated: -0.8031 = -0.8031000 to 7 significant figures (working in decimal). The neighbouring representable numbers with 7 significant figures are -0.8031001 and -0.8030999. Each of these differs from the center coordinate by 0.0000001 = 1e-7, which is 36 times larger the pixel spacing we computed as 2.78e-9. This means that for every 36 pixels in the horizontal direction, there's only one "float" value that can be used: not surprisingly, if 36 pixels use the same input value, they'll give the same output value.

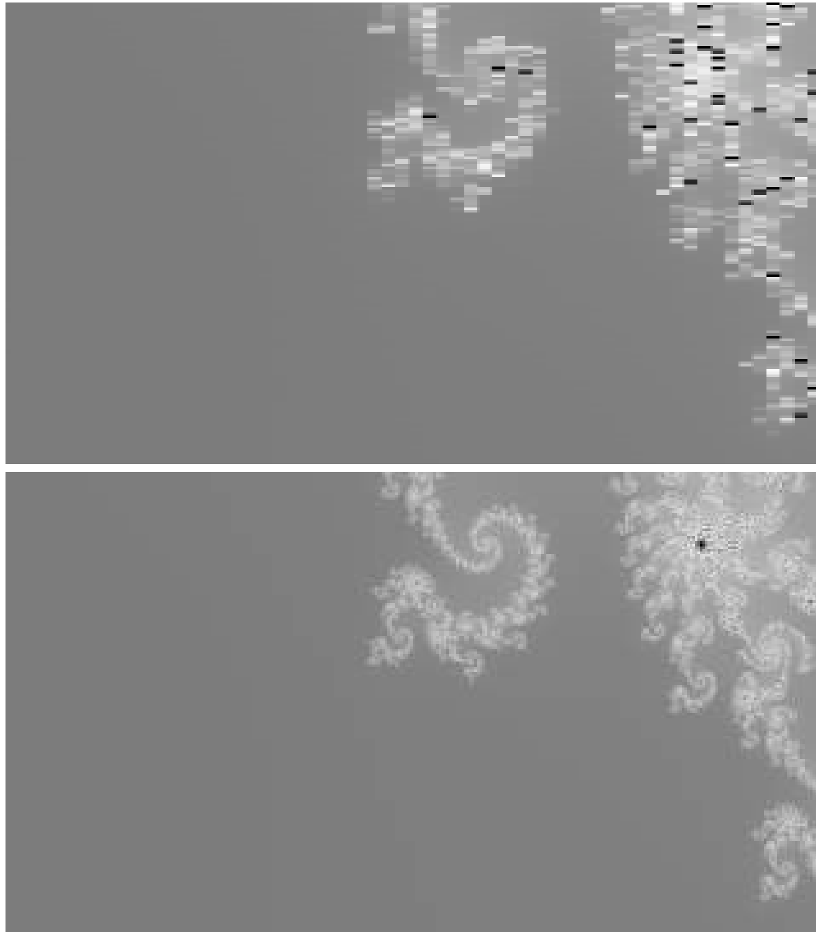
The situation is better in the vertical direction: 0.1780 is around 4.5 times smaller than -0.8031. The nature of binary floating point means representable values are more closely spaced the smaller the absolute value: if you halve the absolute value, the spacing halves too. For example with 7 significant figures in decimal, the unit of least precision for 1.000000 is 0.000001 = 1e-6, and the unit of least precision for 1000000 = 1e6 = 1.

Now we know why the pixelation occurs (we don't have enough precision, our mantissa isn't wide enough), and why it's so much worse in one direction than the other (floating point number precision is relative to the absolute size of the value).

C has more than one floating point number type, and the "float" we have been using is a "single precision" type. If we switch to the "double precision" type, imaginatively called "double", then our pixellation problem will be solved, at least for the moment. And sure enough, after replicating our functions to make them use double precision, we get a much clearer image:

```
$ ./mandelbrot -0.8031  0.1780  0.000001 0 > float.pgm
$ ./mandelbrot -0.8031  0.1780  0.000001 1 > double.pgm
$ display float.pgm double.pgm
```

We kept the single precision floating point version and renamed it with an "f" suffix, as is the idiom in C, and use atoi() to read the 4th command line argument as a flag to tell our program which implementation to use.



```
diff --git a/mandelbrot.c b/mandelbrot.c
index d28847d..c95350e 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -4,3 +4,7 @@

-float cabs2(complex float z) {
+float cabs2f(complex float z) {
+  return crealf(z) * crealf(z) + cimagf(z) * cimagf(z);
+}
+
+double cabs2(complex double z) {
   return creal(z) * creal(z) + cimag(z) * cimag(z);
@@ -8,5 +12,5 @@ float cabs2(complex float z) {

-complex float coordinate(int i, int j, int width, int height, complex float center, floa
-  float x = (i - width /2.0) / (height/2.0);
-  float y = (j - height/2.0) / (height/2.0);
+complex float coordinatef(int i, int j, int width, int height, ↙
    ↳ complex float center, float radius) {
+  float x = (i - width /2.0f) / (height/2.0f);
```

```
+   float y = (j - height/2.0f) / (height/2.0f);
    complex float c = center + radius * (x - I * y);
@@ -15,3 +19,10 @@ complex float coordinate(int i, int j, int ↙
    ↳ width, int height, complex float cent

-int calculate(complex float c, int maximum_iterations, float escape_radius2) {
+complex double coordinate(int i, int j, int width, int height, ↙
    ↳ complex double center, double radius) {
+   double x = (i - width /2.0) / (height/2.0);
+   double y = (j - height/2.0) / (height/2.0);
+   complex double c = center + radius * (x - I * y);
+   return c;
+}
+
+int calculatef(complex float c, int maximum_iterations, float ↙
    ↳ escape_radius2) {
    complex float z = 0;
@@ -20,2 +31,15 @@ int calculate(complex float c, int ↙
    ↳ maximum_iterations, float escape_radius2) {
      z = z * z + c;
+     if (cabs2f(z) > escape_radius2) {
+       final_n = n;
+       break;
+     }
+   }
+   return final_n;
+}
+
+int calculate(complex double c, int maximum_iterations, double ↙
    ↳ escape_radius2) {
+   complex double z = 0;
+   int final_n = 0;
+   for (int n = 1; n < maximum_iterations; ++n) {
+     z = z * z + c;
      if (cabs2(z) > escape_radius2) {
@@ -28,3 +52,14 @@ int calculate(complex float c, int ↙
    ↳ maximum_iterations, float escape_radius2) {

-void render(int width, int height, complex float center, float radius, int maximum_iterations,
+void renderf(int width, int height, complex float center, float ↙
    ↳ radius, int maximum_iterations, float escape_radius2) {
+   printf("P5\n%d %d\n255\n", width, height);
+   for (int j = 0; j < height; ++j) {
+     for (int i = 0; i < width; ++i) {
+       complex float c = coordinatef(i, j, width, height, center, ↙
    ↳ radius);
+       int final_n = calculatef(c, maximum_iterations, ↙
    ↳ escape_radius2);
+       putchar(final_n);
+     }
+   }
+}
+
```

```
+void render(int width, int height, complex double center, double radius, int maximum_iterations, double escape_radius2) {
   printf("P5\n%d %d\n255\n", width, height);
@@ -32,3 +67,3 @@ void render(int width, int height, complex float center, float radius, int maxim
     for (int i = 0; i < width; ++i) {
-      complex float c = coordinate(i, j, width, height, center, radius);
+      complex double c = coordinate(i, j, width, height, center, radius);
       int final_n = calculate(c, maximum_iterations, escape_radius2);
@@ -42,7 +77,8 @@ int main(int argc, char **argv) {
   int height = 720;
-  complex float center = 0;
-  float radius = 2;
+  complex double center = 0;
+  double radius = 2;
   int maximum_iterations = 256;
-  float escape_radius = 2;
-  float escape_radius2 = escape_radius * escape_radius;
+  double escape_radius = 2;
+  double escape_radius2 = escape_radius * escape_radius;
+  int use_double = 0;
   if (argc > 3) {
@@ -51,3 +87,10 @@ int main(int argc, char **argv) {
   }
-  render(width, height, center, radius, maximum_iterations, escape_radius2);
+  if (argc > 4) {
+    use_double = atoi(argv[4]);
+  }
+  if (use_double) {
+    render (width, height, center, radius, maximum_iterations, escape_radius2);
+  } else {
+    renderf(width, height, center, radius, maximum_iterations, escape_radius2);
+  }
   return 0;
```

1.18 Generic programming with preprocessor macros.

In supporting double precision to allow deeper zooming without pixelation, we duplicated a lot of code, just to change the number type used. Now we'll get rid of the code duplication, leaving us with one implementation of each algorithm. We move one copy of the implementation into mandelbrot_imp.c, using the name FTYPE wherever "float" or "double" is needed. We also wrap each function that has variants for different number types in FNAME().

Now, FTYPE and FNAME() are not defined by the C standard, so the code won't compile. We'll define them ourselves, in a header file mandelbrot.h. This begins and ends with an idiomatic header guard, to avoid errors if the file would be included more than once. Next it includes the

library headers that mandelbrot_imp.c needs. We to define FTYPE to be a floating point type, like float, whose variant function suffix is f. Then we include the implementation code. Now we can undefine FTYPE and FNAME(), and repeat this stanza for the other floating types available in C: double and long double.

This technique uses the C preprocessor, which the compiler runs early on in the compilation pipeline (first the preprocessor performs file inclusion and macro expansion, then the C compiler generates assembly source for the target architecture, which the assembler translates to machine code, which is then linked into a program). Macro expansion is similar to search and replace within a text editor, with macros able to take parameters. The ## operator in a macro joins two tokens together to form a new token.

Now our main program file mandelbrot.c includes our header file, and its main() parses the command line arguments and calls the appropriate function from our mini-library. We use strtold() now instead of atof() to avoid losing precision before we even begin. The switch() statement is an alternative to nested if statements. Finally we add the extra files to the prerequisites of our Makefile target, so that “make” will rebuild the program when we update any of them.

Now we can benchmark the implementation with different number types:

```
$ time ./mandelbrot 0 0 2 0 > float.pgm
real    0m0.296s
user    0m0.288s
sys     0m0.004s
$ time ./mandelbrot 0 0 2 1 > double.pgm
real    0m0.130s
user    0m0.128s
sys     0m0.000s
$ time ./mandelbrot 0 0 2 2 > long-double.pgm
real    0m0.525s
user    0m0.528s
sys     0m0.000s
$
```

These results are surprising: float is actually slower than double, despite having less precision.

```
diff --git a/Makefile b/Makefile
index 8c548fd..63e7211 100644
--- a/Makefile
+++ b/Makefile
@@ -1,2 +1,2 @@
-mandelbrot: mandelbrot.c
+mandelbrot: mandelbrot.c mandelbrot.h mandelbrot_imp.c
        gcc -std=c99 -Wall -Wextra -pedantic -O3 -o mandelbrot mandelbrot.c -lm
diff --git a/mandelbrot.c b/mandelbrot.c
index c95350e..4913865 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -4,71 +4,3 @@

-float cabs2f(complex float z) {
-  return crealf(z) * crealf(z) + cimagf(z) * cimagf(z);
-}
```

```
double cabs2(complex double z) {
    return creal(z) * creal(z) + cimag(z) * cimag(z);
}

complex float coordinatef(int i, int j, int width, int height, complex float center, flo
    float x = (i - width /2.0f) / (height/2.0f);
    float y = (j - height/2.0f) / (height/2.0f);
    complex float c = center + radius * (x - I * y);
    return c;
}

complex double coordinate(int i, int j, int width, int height, complex double center, d
    double x = (i - width /2.0) / (height/2.0);
    double y = (j - height/2.0) / (height/2.0);
    complex double c = center + radius * (x - I * y);
    return c;
}

int calculatef(complex float c, int maximum_iterations, float escape_radius2) {
    complex float z = 0;
    int final_n = 0;
    for (int n = 1; n < maximum_iterations; ++n) {
        z = z * z + c;
        if (cabs2f(z) > escape_radius2) {
            final_n = n;
            break;
        }
    }
    return final_n;
}

int calculate(complex double c, int maximum_iterations, double escape_radius2) {
    complex double z = 0;
    int final_n = 0;
    for (int n = 1; n < maximum_iterations; ++n) {
        z = z * z + c;
        if (cabs2(z) > escape_radius2) {
            final_n = n;
            break;
        }
    }
    return final_n;
}

void renderf(int width, int height, complex float center, float radius, int maximum_it
    printf("P5\n%d %d\n255\n", width, height);
    for (int j = 0; j < height; ++j) {
        for (int i = 0; i < width; ++i) {
            complex float c = coordinatef(i, j, width, height, center, radius);
            int final_n = calculatef(c, maximum_iterations, escape_radius2);
```

```
-        putchar(final_n);
-      }
-    }
-}
-
-void render(int width, int height, complex double center, double radius, int maximum_iteration
-    printf("P5\n%d %d\n255\n", width, height);
-    for (int j = 0; j < height; ++j) {
-      for (int i = 0; i < width; ++i) {
-        complex double c = coordinate(i, j, width, height, center, radius);
-        int final_n = calculate(c, maximum_iterations, escape_radius2);
-        putchar(final_n);
-      }
-    }
-}
+#include "mandelbrot.h"

@@ -77,19 +9,25 @@ int main(int argc, char **argv) {
   int height = 720;
-  complex double center = 0;
-  double radius = 2;
+  complex long double center = 0;
+  long double radius = 2;
   int maximum_iterations = 256;
-  double escape_radius = 2;
-  double escape_radius2 = escape_radius * escape_radius;
-  int use_double = 0;
+  long double escape_radius = 2;
+  long double escape_radius2 = escape_radius * escape_radius;
+  int float_type = 0;
   if (argc > 3) {
-    center = atof(argv[1]) + I * atof(argv[2]);
-    radius = atof(argv[3]);
+    center = strtold(argv[1], 0) + I * strtold(argv[2], 0);
+    radius = strtold(argv[3], 0);
   }
   if (argc > 4) {
-    use_double = atoi(argv[4]);
+    float_type = atoi(argv[4]);
   }
-  if (use_double) {
-    render (width, height, center, radius, maximum_iterations, escape_radius2);
-  } else {
-    renderf(width, height, center, radius, maximum_iterations, escape_radius2);
+  switch (float_type) {
+    case 0:
+      renderf(width, height, center, radius, maximum_iterations, escape_radius2);
+      break;
```

```
+     case 1:
+       render (width, height, center, radius, maximum_iterations, ↙
    ↳ escape_radius2);
+       break;
120 +     case 2:
+       renderl(width, height, center, radius, maximum_iterations, ↙
    ↳ escape_radius2);
+       break;
   }
 diff --git a/mandelbrot.h b/mandelbrot.h
125 new file mode 100644
index 0000000..8f482db
--- /dev/null
+++ b/mandelbrot.h
@@ -0,0 +1,25 @@
130 +#ifndef MANDELBROT_H
+#define MANDELBROT_H 1
+
+#include <complex.h>
+#include <stdio.h>
135 +
+#define FTYPE float
+#define FNAME(name) name ## f
+#include "mandelbrot_imp.c"
+#undef FTYPE
140 +#undef FNAME
+
+#define FTYPE double
+#define FNAME(name) name
+#include "mandelbrot_imp.c"
145 +#undef FTYPE
+#undef FNAME
+
+#define FTYPE long double
+#define FNAME(name) name ## l
150 +#include "mandelbrot_imp.c"
+#undef FTYPE
+#undef FNAME
+
+#endif
155  diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
new file mode 100644
index 0000000..e08bbcb
--- /dev/null
+++ b/mandelbrot_imp.c
160 @@ -0,0 +1,34 @@
+FTYPE FNAME(cabs2)(complex FTYPE z) {
+  return FNAME(creal)(z) * FNAME(creal)(z) + FNAME(cimag)(z) * ↙
    ↳ FNAME(cimag)(z);
+}
+
165 +complex FTYPE FNAME(coordinate)(int i, int j, int width, int ↙
    ↳ height, complex FTYPE center, FTYPE radius) {
```

```
+  FTYPE x = (i - width /FNAME(2.0)) / (height/FNAME(2.0));
+  FTYPE y = (j - height/FNAME(2.0)) / (height/FNAME(2.0));
+  complex FTYPE c = center + radius * (x - I * y);
+  return c;
170 +}
+
+int FNAME(calculate)(complex FTYPE c, int maximum_iterations, ↙
    ↳ FTYPE escape_radius2) {
+  complex FTYPE z = 0;
+  int final_n = 0;
175 +  for (int n = 1; n < maximum_iterations; ++n) {
+    z = z * z + c;
+    if (FNAME(cabs2)(z) > escape_radius2) {
+      final_n = n;
+      break;
180 +    }
+  }
+  return final_n;
+}
+
185 +void FNAME(render)(int width, int height, complex FTYPE center, ↙
    ↳ FTYPE radius, int maximum_iterations, FTYPE escape_radius2) ↙
    ↳ {
+  printf("P5\n%d␣%d\n255\n", width, height);
+  for (int j = 0; j < height; ++j) {
+    for (int i = 0; i < width; ++i) {
+      complex FTYPE c = FNAME(coordinate)(i, j, width, height, ↙
    ↳ center, radius);
190 +      int final_n = FNAME(calculate)(c, maximum_iterations, ↙
    ↳ escape_radius2);
+      putchar(final_n);
+    }
+  }
+}
```

1.19 Colour images.

So far our images have progressed from black and white to greyscale, with the grey level based on the escape time. We're writing PGM files, part of the PNM family of formats, and this family include PPM which supports colour. The text header for PPM is very similar to PGM, with the magic number P6 in place of PGM's P5. With the maxval of 255 that we are using, each pixel has 3 bytes, for the red, green and blue colour channels (in that order, tightly packed).

We rename our existing render to render_grey, and write a new render_colour to output the modified header and pixel data. Between calculating the escape time and outputting the pixel data, we need to turn the linear value into a point in 3D colour space. About the simplest way to do this is to vary the speed at which the output colour channels change with respect to the input value. We define a colour() function in the mandelbrot.h header because it doesn't depend on the floating point type used for calculations, and make the red channel change more quickly and the blue channel chnage more slowly.

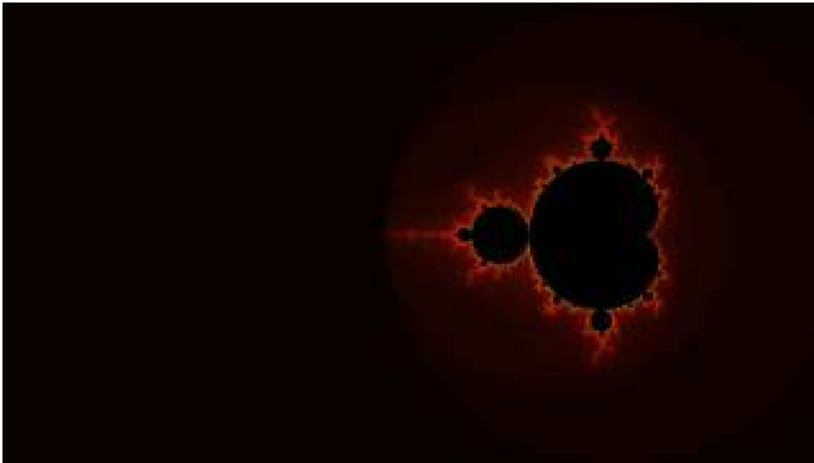
In the main program we handle another command line argument, to switch between greyscale rendering and colour rendering. With 3 number types and 2 colour types, we have ended up

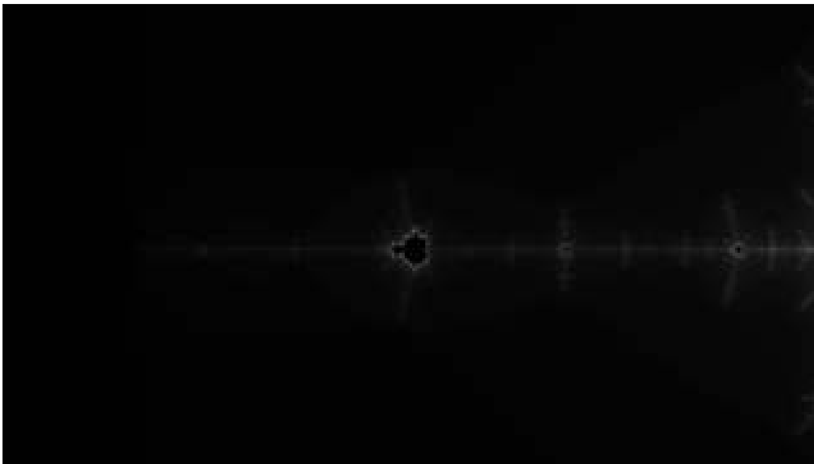
with 6 render functions to choose from in our now-nested switch statements. We increase the maximum iteration limit to show more detail near the main body of the Mandelbrot set, and switch the default number type to double because it is both faster and more precise than float (though we still need to choose the float type argument if we want to specify a colour type).

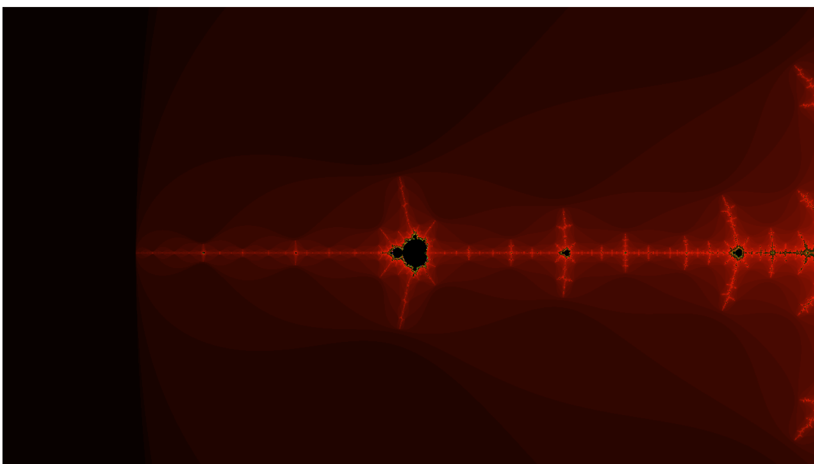
Now let's render a sequence of images zooming in towards a miniature copy within the Mandelbrot set, to compare colour with greyscale:

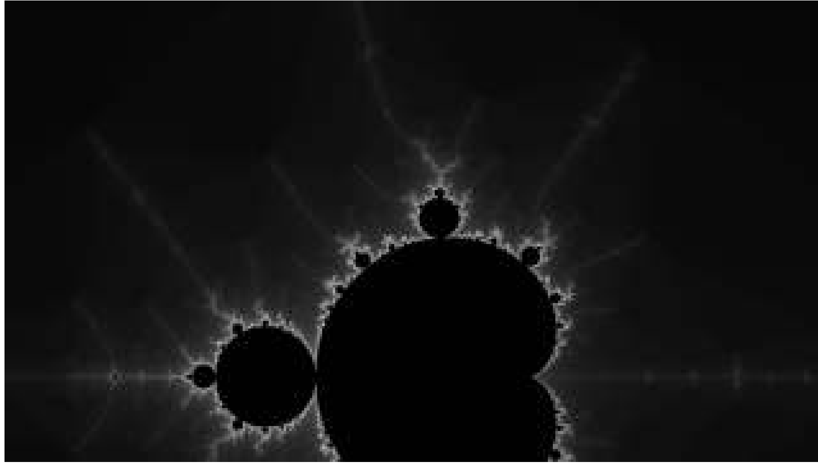
```
$ time for exp in $(seq -w 0 15)
  do
    ./mandelbrot \
      -1.75993729527142950509191675 7 0.012591790526496335594898047 \
      2e-${exp} 2 0 > ${exp}-grey.pgm
    ./mandelbrot \
      -1.75993729527142950509191675 7 0.012591790526496335594898047 \
      2e-${exp} 2 1 > ${exp}-colour.ppm
  done
real    7m41.621s
user    7m40.925s
sys     0m0.492s
```

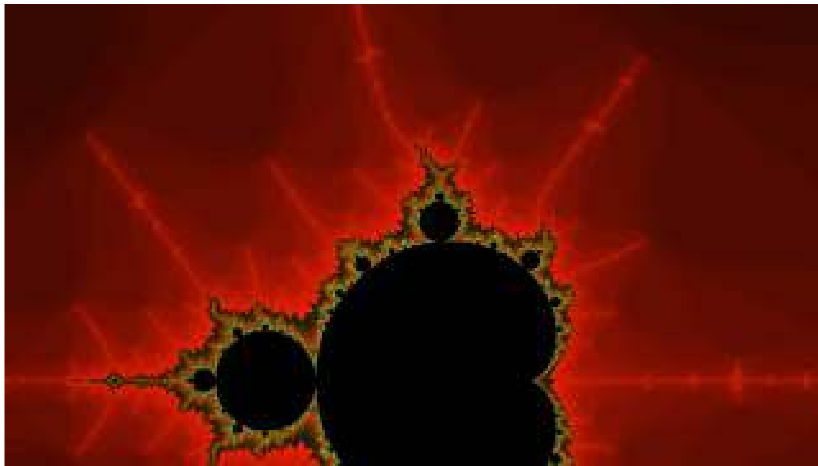
This is a bash shell script, which loops over the sequence of numbers from 0 to 15, and each pass through the loop it runs our program twice, using the number as radius exponent, once for greyscale and once for colour. We end up with 32 images to browse through.

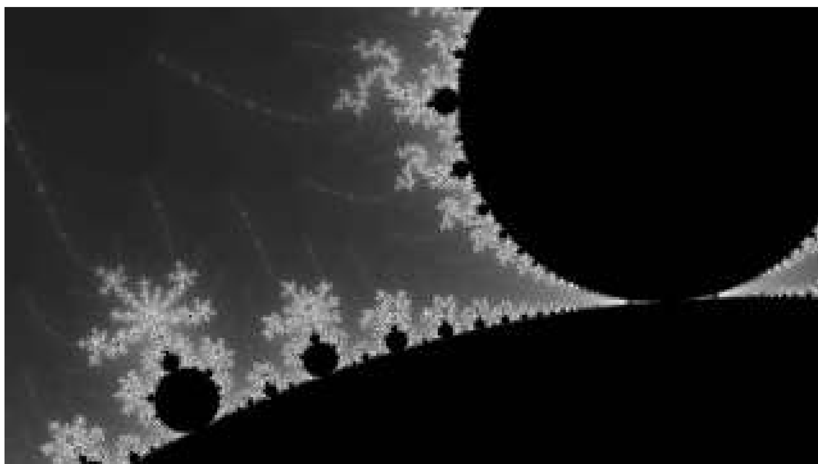


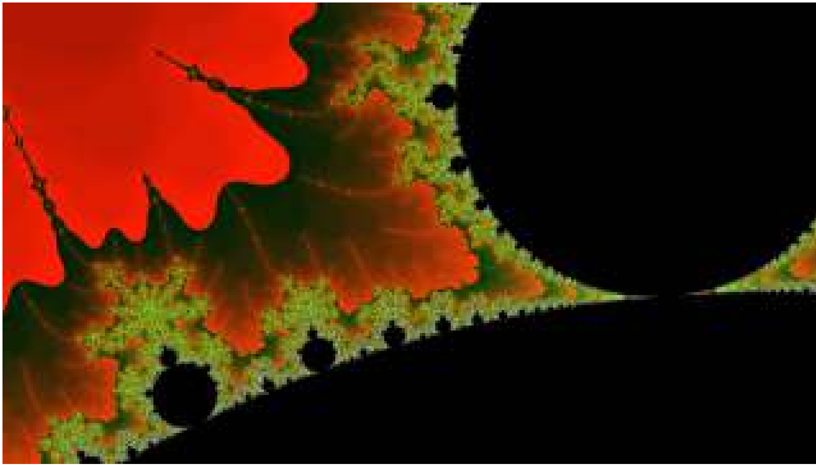


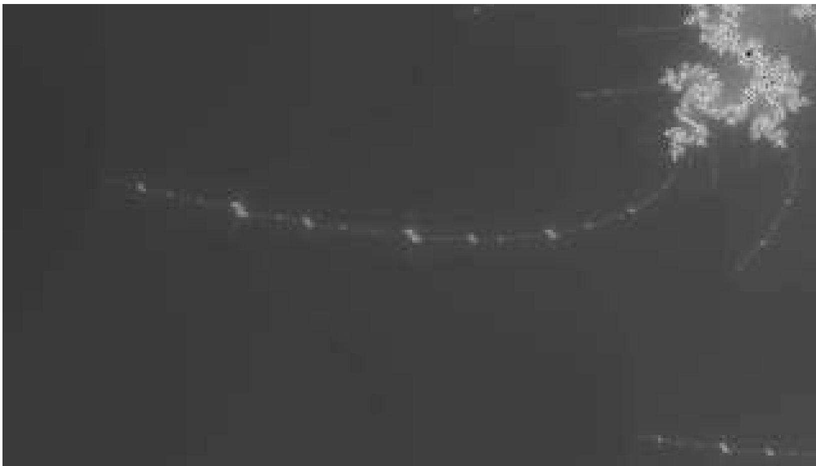


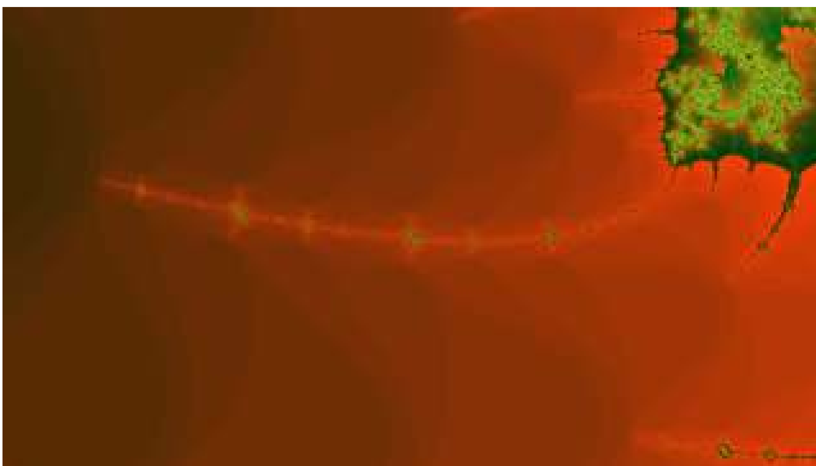


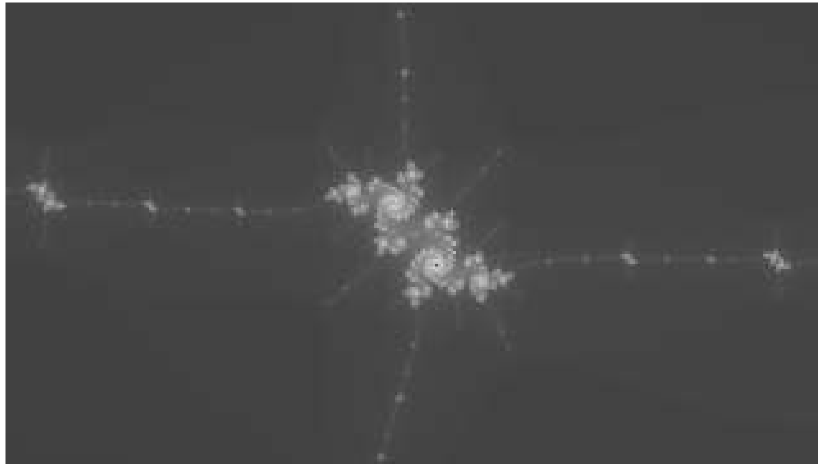


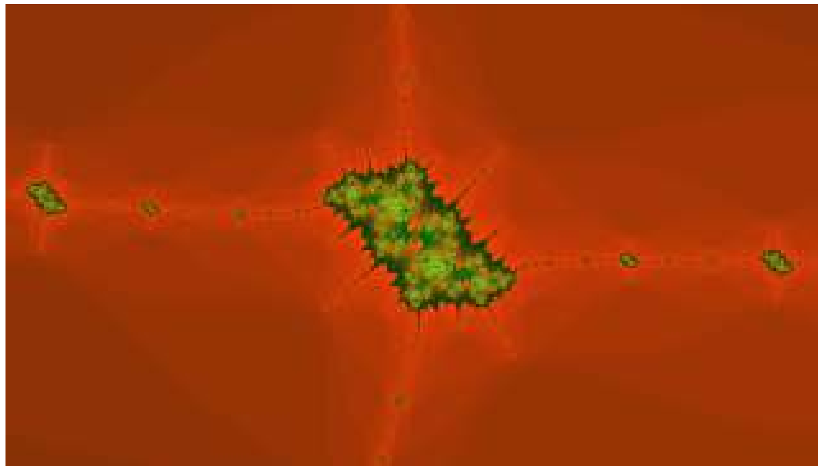


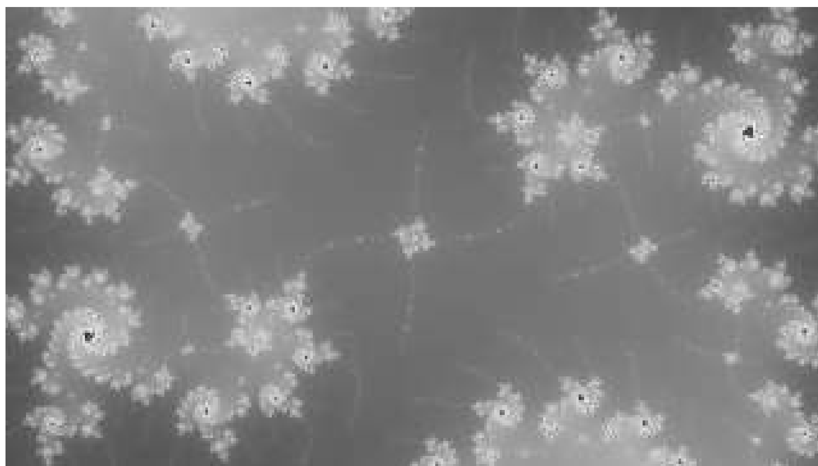


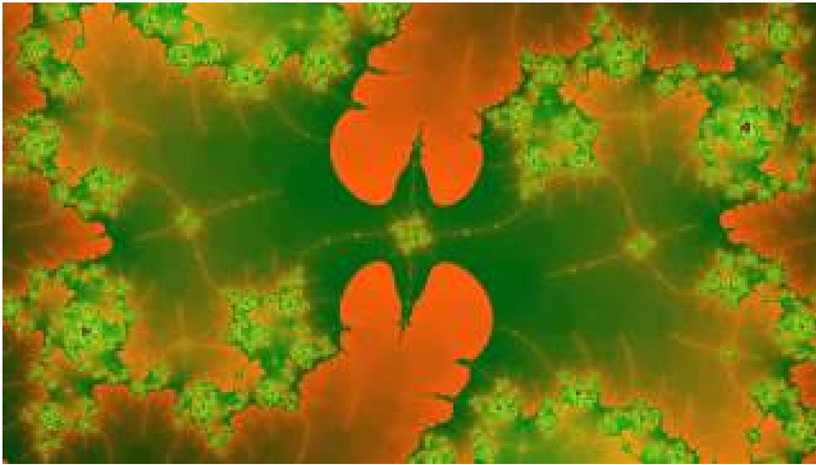


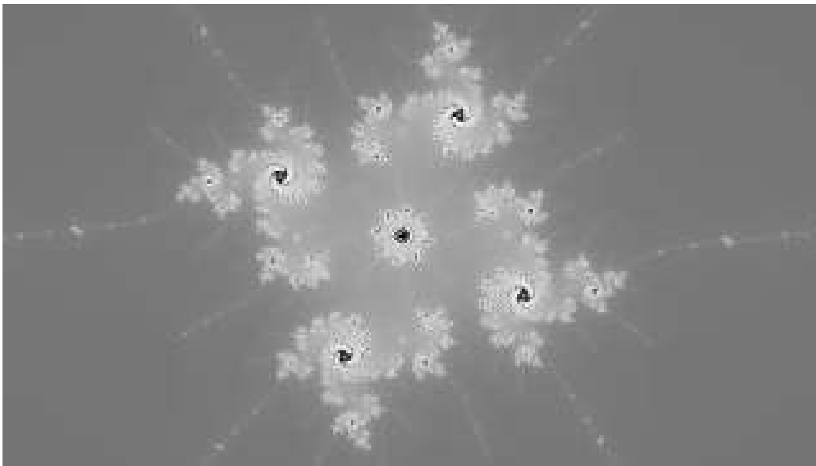


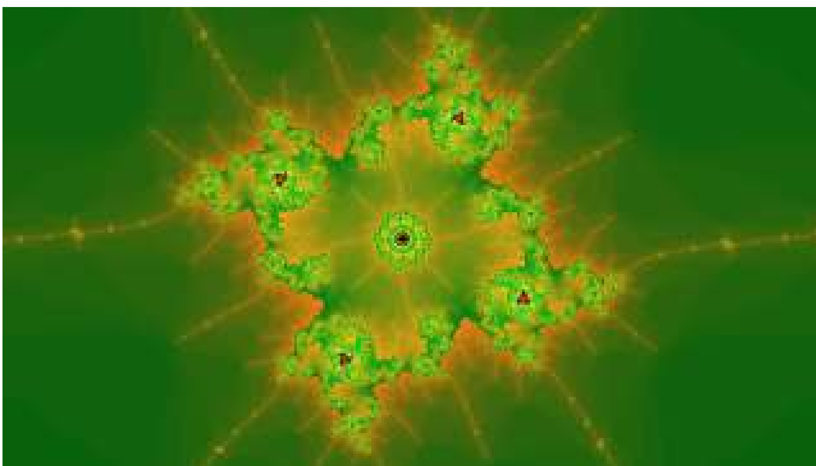


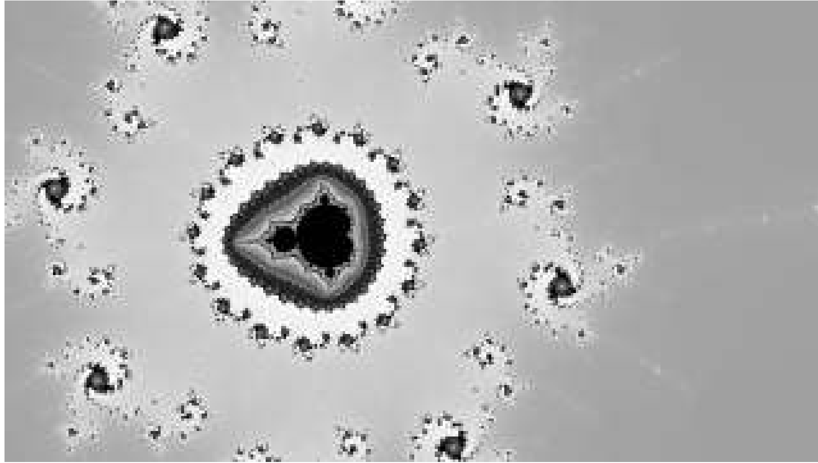


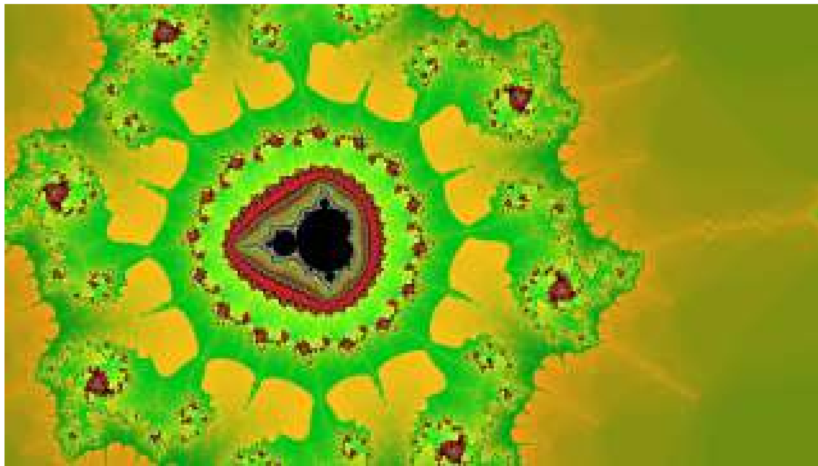


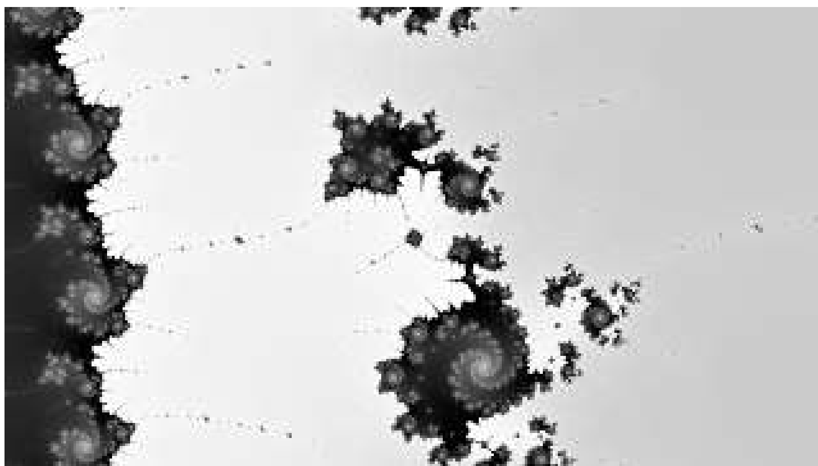


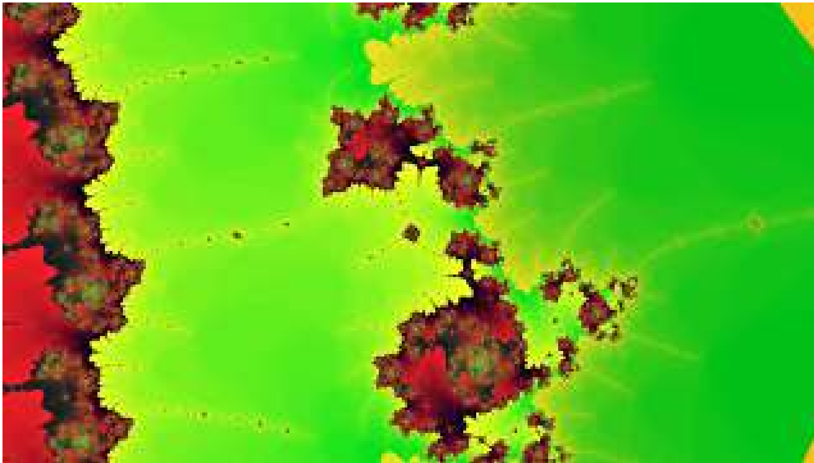


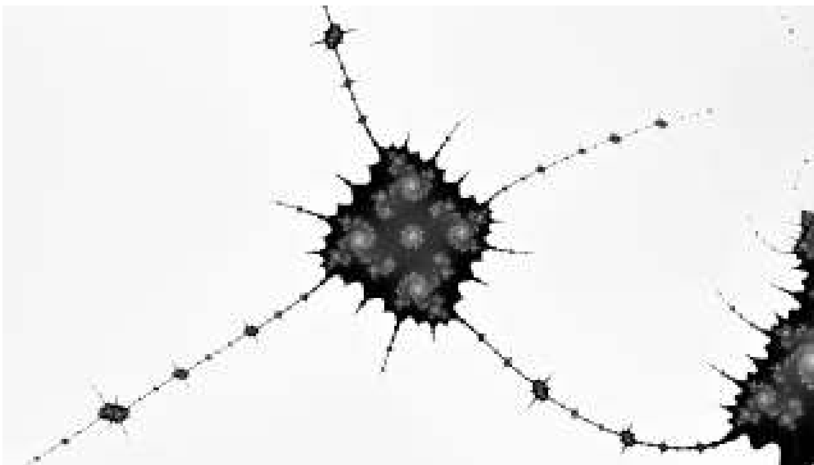


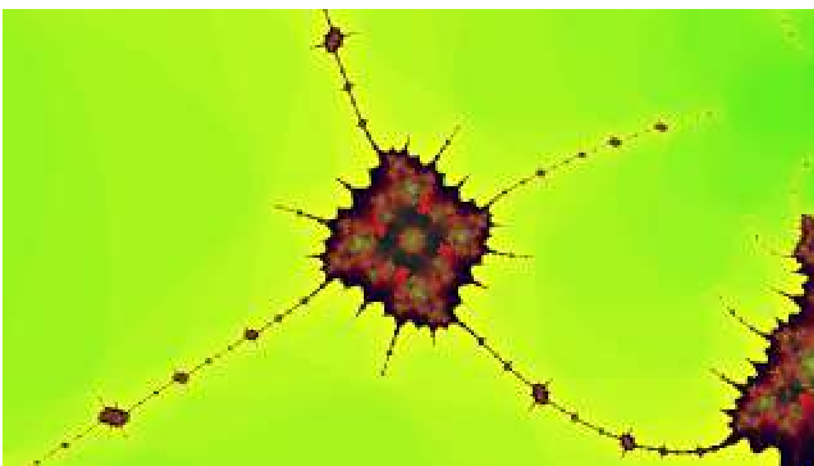


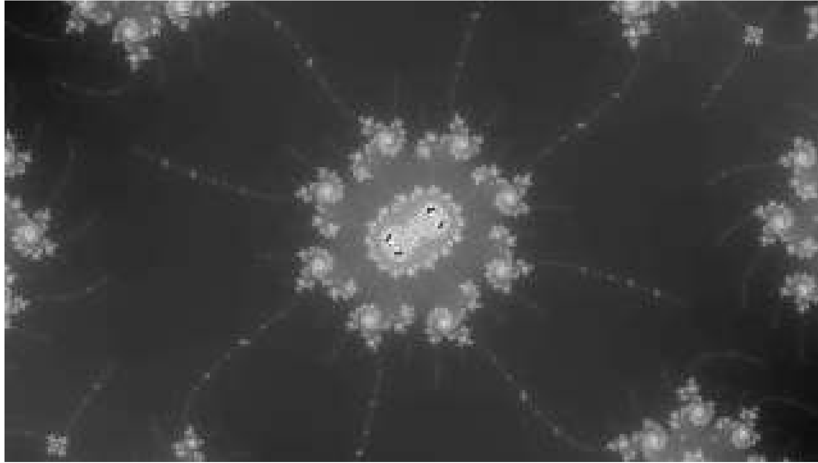


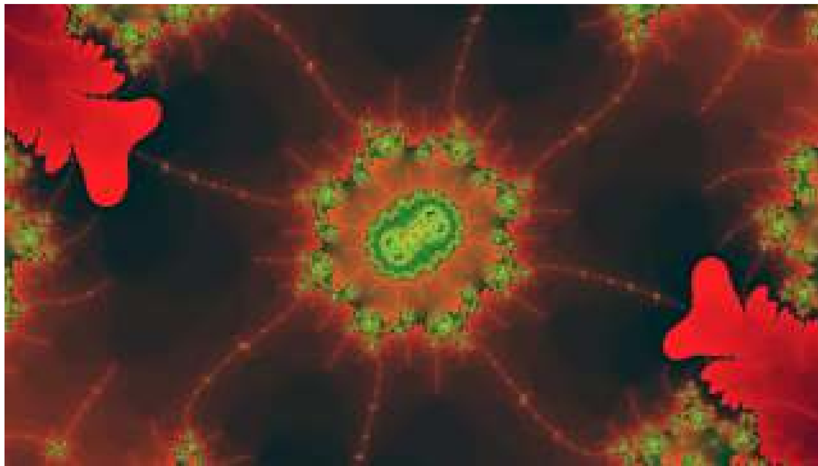


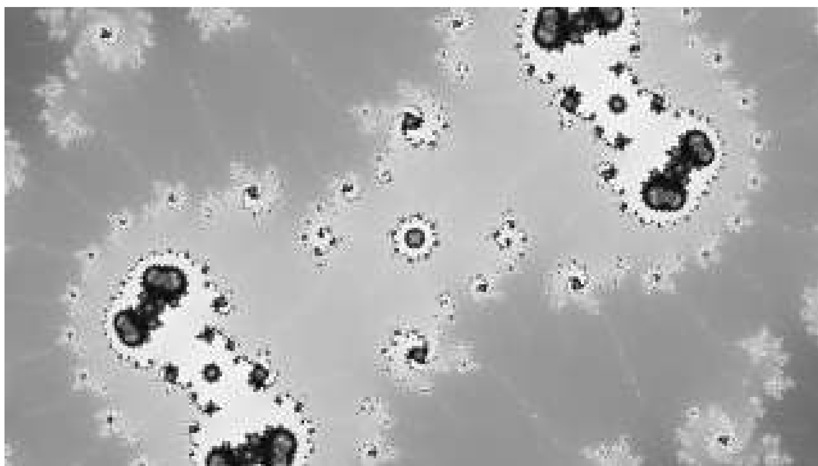


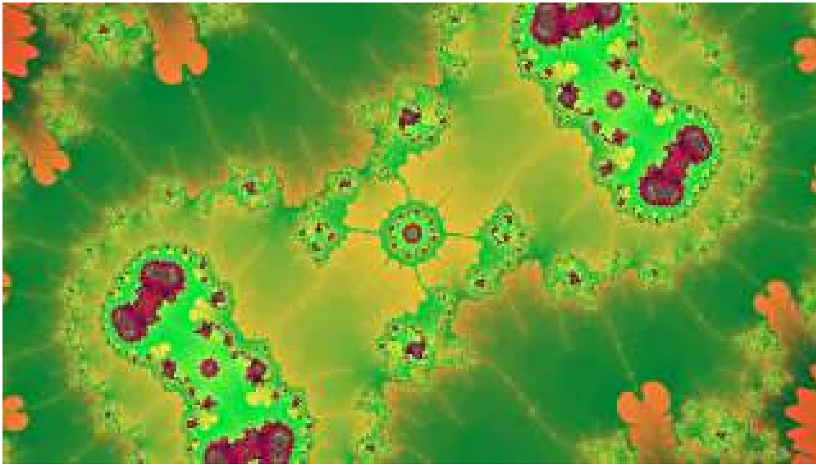


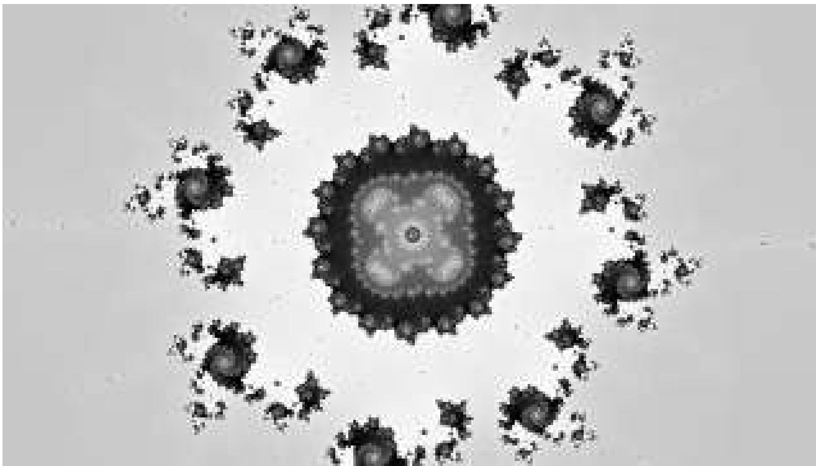


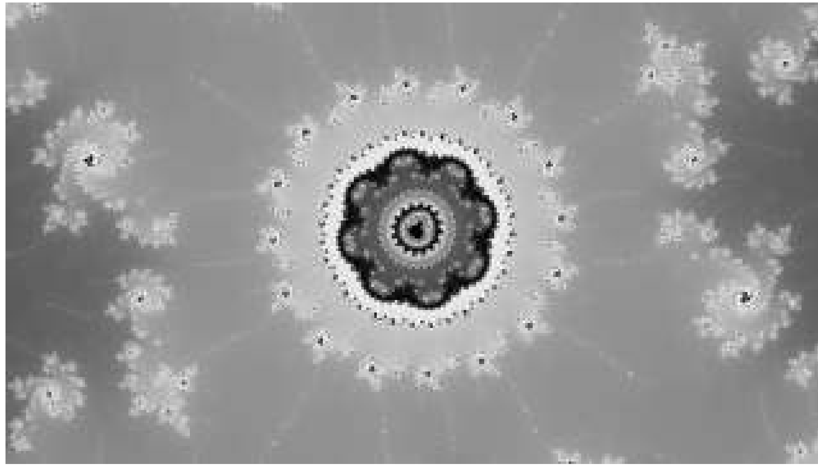


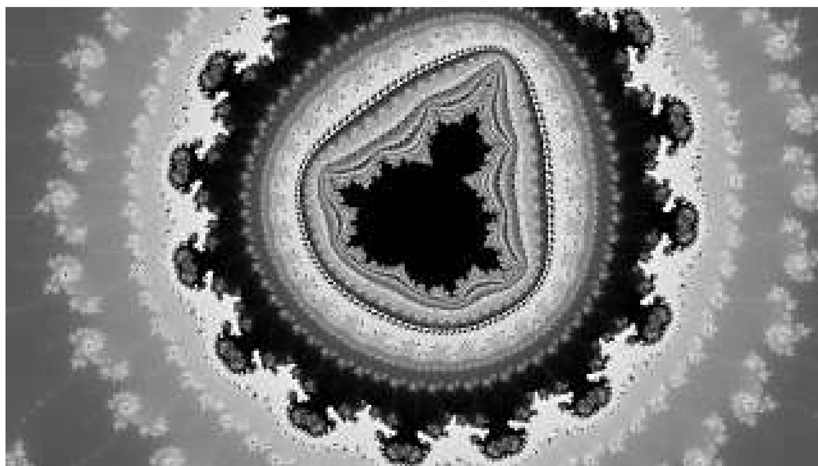


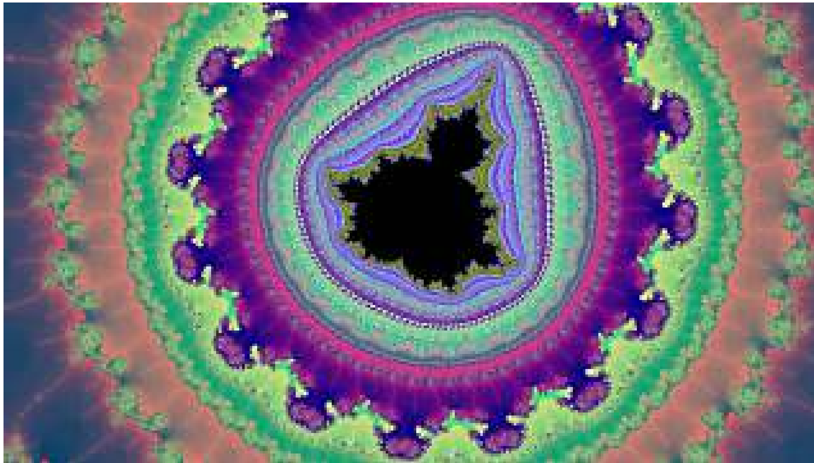












```
diff --git a/mandelbrot.c b/mandelbrot.c
index 4913865..361009b 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -11,6 +11,7 @@ int main(int argc, char **argv) {
   long double radius = 2;
-  int maximum_iterations = 256;
+  int maximum_iterations = 4096;
   long double escape_radius = 2;
   long double escape_radius2 = escape_radius * escape_radius;
-  int float_type = 0;
+  int float_type = 1;
+  int colour_type = 0;
   if (argc > 3) {
@@ -22,11 +23,31 @@ int main(int argc, char **argv) {
   }
-  switch (float_type) {
+  if (argc > 5) {
+    colour_type = atoi(argv[5]);
+  }
+  switch (colour_type) {
     case 0:
-
    ↳ renderf(width, height, center, radius, maximum_iterations, escape_radius2);
+      switch (float_type) {
+        case 0:
+          render_greyf(width, height, center, radius,
    ↳ maximum_iterations, escape_radius2);
+          break;
+        case 1:
+          render_grey (width, height, center, radius,
    ↳ maximum_iterations, escape_radius2);
+          break;
+        case 2:
+          render_greyl(width, height, center, radius,
    ↳ maximum_iterations, escape_radius2);
+          break;
```

```
+       }
        break;
      case 1:
-         render (width, height, center, radius, maximum_iterations, escape_radius2);
-         break;
-       case 2:
-         renderl(width, height, center, radius, maximum_iterations, escape_radius2);
+       switch (float_type) {
+         case 0:
+           render_colourf(width, height, center, radius, maximum_iterations, escape_radius2);
+           break;
+         case 1:
+           render_colour (width, height, center, radius, maximum_iterations, escape_radius2);
+           break;
+         case 2:
+           render_colourl(width, height, center, radius, maximum_iterations, escape_radius2);
+           break;
+       }
        break;
diff --git a/mandelbrot.h b/mandelbrot.h
index 8f482db..55c72e8 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -6,2 +6,8 @@

+void colour(int *r, int *g, int *b, int final_n) {
+  *r = final_n * 8;
+  *g = final_n;
+  *b = final_n / 8;
+}
+
 #define FTYPE float
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index e08bbcb..1fb27c7 100644
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -24,3 +24,3 @@ int FNAME(calculate)(complex FTYPE c, int maximum_iterations, FTYPE escape_radiu

-void FNAME(render)(int width, int height, complex FTYPE center, FTYPE radius, int maxim
+void FNAME(render_grey)(int width, int height, complex FTYPE center, FTYPE radius, int maximum_iterations, FTYPE escape_radius2) {
   printf("P5\n%d %d\n255\n", width, height);
@@ -34 +34,16 @@ void FNAME(render)(int width, int height, complex FTYPE center, FTYPE radius, in
 }
+
```

```
+void FNAME(render_colour)(int width, int height, complex FTYPE ↙
    ↳ center, FTYPE radius, int maximum_iterations, FTYPE ↙
    ↳ escape_radius2) {
+  printf("P6\n%d␣%d\n255\n", width, height);
+  for (int j = 0; j < height; ++j) {
+    for (int i = 0; i < width; ++i) {
+      complex FTYPE c = FNAME(coordinate)(i, j, width, height, ↙
    ↳ center, radius);
+      int final_n = FNAME(calculate)(c, maximum_iterations, ↙
    ↳ escape_radius2);
+      int r, g, b;
+      colour(&r, &g, &b, final_n);
+      putchar(r);
+      putchar(g);
+      putchar(b);
+    }
+  }
+}
```

1.20 Decoupling rendering and colouring.

Unfortunately the colour scheme we chose looked quite bad, and it took a few minutes to render all the images. To reduce waiting time, we can save the raw data from rendering to a file, and colour it later.

We rename our mandelbrot.c program to render.c, and strip out the colour type handling. We delete the render_grey() from mandelbrot_imp.c, and rename render_colour() back to render(), using a colour mapping that preserves information. With 3 bytes, each with 8 bits, we can represent 24bit integers, enough range to represent over 16 million iterations. The » operator is bitwise shift right, here we use it to extract the individual bytes from the escape time. We rely on putchar() truncating to the least significant byte in the resulting integer.

We write a new program to remap the colours in the PPM output from the render program. Our colour.c reads the image header from its standard input using scanf(), which works a bit like printf() in reverse. We check that we read the header successfully by confirming that we parsed two items, the width and height, and that the next character after the maxval of 255 is a newline. Then we loop over all the pixels, repacking the RGB values into a single integer.

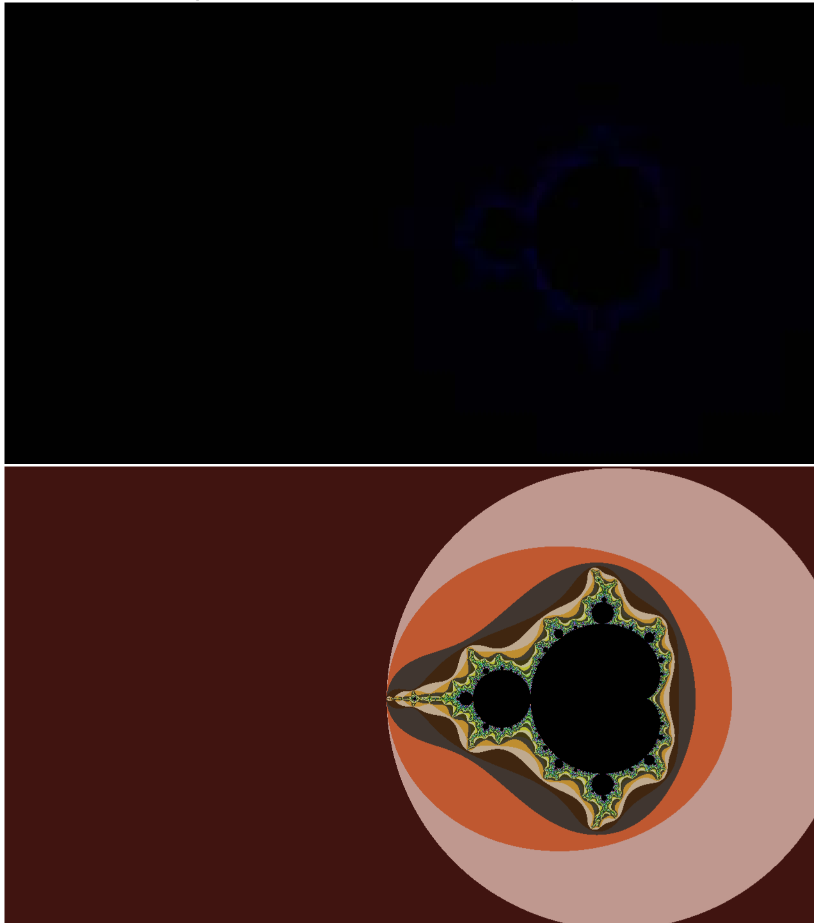
We now use the HSV colour space, allowing us to specify colours by hue, saturation and value. Hue is circular, from red at 0, with increasing values running through the rainbow yellow green blue violet, and then back through purple and pink to red again at 1, where the cycle repeats. Saturation blends colour intensity from greyscale at 0 to full blast at 1, and value controls brightness: black at 0 and full strength at 1. We use single precision float for the calculations: 24bits is more than enough, given each colour channel has only 8bits. We need to scale the output values from hsv2rgb() from [0..1] to [0..255], and we clamp them to the output range to make sure no overflow glitches occur.

We add the new rules to the Makefile, and the new binaries to the .gitignore file, and rebuild to check everything works:

```
$ make
$ time for exp in $(seq -w 0 15)
  do
    ./render \
```

```
           -1.759937295271429505091916757 \
            0.012591790526496335594898047 \
            2e-${exp} 2 > ${exp}-render.ppm
      done
    real    3m51.110s
    user    3m50.706s
    sys     0m0.288s
    $ time for i in ??-render.ppm
      do
        ./colour < ${i} > ${i}-colour.ppm
      done
    real    0m2.190s
    user    0m2.080s
    sys     0m0.072s
    $
```

As we suspected, rendering takes a lot longer than colouring, and having separated the two processes we no longer have to rerender when we adjust the colour scheme.



```
diff --git a/.gitignore b/.gitignore
index c0729ec..2911ab0 100644
```

```
--- a/.gitignore
+++ b/.gitignore
@@ -1 +1,2 @@
-mandelbrot
+render
+colour
diff --git a/Makefile b/Makefile
index 63e7211..6ef9ffe 100644
--- a/Makefile
+++ b/Makefile
@@ -1,2 +1,7 @@
-mandelbrot: mandelbrot.c mandelbrot.h mandelbrot_imp.c
-	gcc -std=c99 -Wall -Wextra -pedantic -O3 -o mandelbrot mandelbrot.c -lm
+all: render colour
+
+colour: colour.c
+	gcc -std=c99 -Wall -Wextra -pedantic -O3 -o colour colour.c -lm
+
+render: render.c mandelbrot.h mandelbrot_imp.c
+	gcc -std=c99 -Wall -Wextra -pedantic -O3 -o render render.c -lm
diff --git a/colour.c b/colour.c
new file mode 100644
index 0000000..cba7d71
--- /dev/null
+++ b/colour.c
@@ -0,0 +1,64 @@
+#include <math.h>
+#include <stdio.h>
+
+int channel(float c) {
+  return fminf(fmaxf(roundf(255 * c), 0), 255);
+}
+
+void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {
+  float i, f, p, q, t, r, g, b;
+  if (s == 0) { r = g = b = v; } else {
+    h = 6 * (h - floorf(h));
+    int ii = i = floorf(h);
+    f = h - i;
+    p = v * (1 - s);
+    q = v * (1 - (s * f));
+    t = v * (1 - (s * (1 - f)));
+    switch(ii) {
+      case 0: r = v; g = t; b = p; break;
+      case 1: r = q; g = v; b = p; break;
+      case 2: r = p; g = v; b = t; break;
+      case 3: r = p; g = q; b = v; break;
+      case 4: r = t; g = p; b = v; break;
+      default:r = v; g = p; b = q; break;
```

```
+    }
+  }
+  *red = r;
+  *grn = g;
+  *blu = b;
+}
+
+void colour(int *r, int *g, int *b, int final_n) {
+  float hue = final_n / 64.0f;
+  float sat = (final_n & 1) ? 0.75f : 0.25f;
+  float val = (final_n == 0) ? 0.0f :
+              (final_n & 2) ? 0.75f : 0.25f;
+  float red, grn, blu;
+  hsv2rgb(hue, sat, val, &red, &grn, &blu);
+  *r = channel(red);
+  *g = channel(grn);
+  *b = channel(blu);
+}
+
+int main() {
+  int width = 0;
+  int height = 0;
+  if (2 == scanf("P6 %d %d 255", &width, &height)) {
+    if (getchar() == '\n') {
+      printf("P6\n%d %d\n255\n", width, height);
+      for (int j = 0; j < height; ++j) {
+        for (int i = 0; i < width; ++i) {
+          int r = getchar();
+          int g = getchar();
+          int b = getchar();
+          int final_n = (r << 16) | (g << 8) | b;
+          colour(&r, &g, &b, final_n);
+          putchar(r);
+          putchar(g);
+          putchar(b);
+        }
+      }
+    }
+  }
+  return 0;
+}
diff --git a/mandelbrot.c b/mandelbrot.c
deleted file mode 100644
index 361009b..0000000
--- a/mandelbrot.c
+++ /dev/null
@@ -1,56 +0,0 @@
-#include <complex.h>
-#include <stdio.h>
-#include <stdlib.h>
-
-#include "mandelbrot.h"
-
```

```
int main(int argc, char **argv) {
  int width = 1280;
  int height = 720;
  complex long double center = 0;
  long double radius = 2;
  int maximum_iterations = 4096;
  long double escape_radius = 2;
  long double escape_radius2 = escape_radius * escape_radius;
  int float_type = 1;
  int colour_type = 0;
  if (argc > 3) {
    center = strtold(argv[1], 0) + I * strtold(argv[2], 0);
    radius = strtold(argv[3], 0);
  }
  if (argc > 4) {
    float_type = atoi(argv[4]);
  }
  if (argc > 5) {
    colour_type = atoi(argv[5]);
  }
  switch (colour_type) {
    case 0:
      switch (float_type) {
        case 0:
          render_greyf(width, height, center, radius, maximum_iterations, escape_radius2);
          break;
        case 1:
          render_grey (width, height, center, radius, maximum_iterations, escape_radius2);
          break;
        case 2:
          render_greyl(width, height, center, radius, maximum_iterations, escape_radius2);
          break;
      }
      break;
    case 1:
      switch (float_type) {
        case 0:
          render_colourf(width, height, center, radius, maximum_iterations, escape_radius2);
          break;
        case 1:
          render_colour (width, height, center, radius, maximum_iterations, escape_radius2);
          break;
        case 2:
          render_colourl(width, height, center, radius, maximum_iterations, escape_radius2);
          break;
      }
      break;
```

```
-    }
-    return 0;
-}
diff --git a/mandelbrot.h b/mandelbrot.h
index 55c72e8..ea5a50b 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -7,5 +7,5 @@
 void colour(int *r, int *g, int *b, int final_n) {
-    *r = final_n * 8;
-    *g = final_n;
-    *b = final_n / 8;
+  *r = final_n >> 16;
+  *g = final_n >> 8;
+  *b = final_n;
 }
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index 1fb27c7..0d576f9 100644
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -24,14 +24,3 @@ int FNAME(calculate)(complex FTYPE c, int ↙
    ↳ maximum_iterations, FTYPE escape_radiu

-void FNAME(render_grey)(int width, int height, complex FTYPE center, FTYPE radius, int
-    printf("P5\n%d %d\n255\n", width, height);
-    for (int j = 0; j < height; ++j) {
-        for (int i = 0; i < width; ++i) {
-            ↙
    ↳ complex FTYPE c = FNAME(coordinate)(i, j, width, height, center, radius);
-            ↙
    ↳ int final_n = FNAME(calculate)(c, maximum_iterations, escape_radius2);
-            putchar(final_n);
-        }
-    }
-}
-
-void FNAME(render_colour)(int width, int height, complex FTYPE center, FTYPE radius, in
+void FNAME(render)(int width, int height, complex FTYPE center, ↙
    ↳ FTYPE radius, int maximum_iterations, FTYPE escape_radius2) ↙
    ↳ {
    printf("P6\n%d %d\n255\n", width, height);
diff --git a/render.c b/render.c
new file mode 100644
index 0000000..ba89d16
--- /dev/null
+++ b/render.c
@@ -0,0 +1,35 @@
+#include <complex.h>
+#include <stdio.h>
+#include <stdlib.h>
+
+#include "mandelbrot.h"
+
```

```
200 +int main(int argc, char **argv) {
    +  int width = 1280;
    +  int height = 720;
    +  complex long double center = 0;
    +  long double radius = 2;
205 +  int maximum_iterations = 4096;
    +  long double escape_radius = 2;
    +  long double escape_radius2 = escape_radius * escape_radius;
    +  int float_type = 1;
    +  if (argc > 3) {
210 +    center = strtold(argv[1], 0) + I * strtold(argv[2], 0);
    +    radius = strtold(argv[3], 0);
    +  }
    +  if (argc > 4) {
    +    float_type = atoi(argv[4]);
215 +  }
    +  switch (float_type) {
    +    case 0:
    +      renderf(width, height, center, radius, maximum_iterations, ↙
       ↳ escape_radius2);
    +      break;
220 +    case 1:
    +      render (width, height, center, radius, maximum_iterations, ↙
       ↳ escape_radius2);
    +      break;
    +    case 2:
    +      renderl(width, height, center, radius, maximum_iterations, ↙
       ↳ escape_radius2);
225 +      break;
    +  }
    +  return 0;
    +}
```

1.21 Decoupling rendering from image output.

So far we have been rendering pixels and outputting them as we go. For future flexibility, we'll collect all the pixel data in memory in the same format they were calculated. This will simplify our render function.

We'll define a new type for our pixels, a struct containing an int. A struct is a collection of fields, accessed with the member selection operator ".", or via a pointer to a struct with the "->" operator. An image is a collection of pixels with a width and a height, so we define another struct for that. We use a pointer to pixels, because we might want to create images of different sizes. Because the size isn't fixed, we need to use dynamic memory allocation. We allocate memory with calloc(), which takes the number of items and the size of each item. The sizeof() operator can take a type, but it's more convenient to get the size from the pointer for which we are allocating memory. Memory allocated should be deallocated when it is no longer needed, and the matching pair to calloc() is free().

We need to write a function to output the image. We print the header and loop over the pixels, packing them into bytes (we inline our minimal colour() function because this is the only use site). Next we modify our render() function to copy the calculated data into our image.

Finally we modify the main() in render.c to create a new image, render into it, write it out, and free it before exiting.

```
diff --git a/mandelbrot.h b/mandelbrot.h
index ea5a50b..faf47f2 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -6,6 +6,35 @@

-void colour(int *r, int *g, int *b, int final_n) {
-  *r = final_n >> 16;
-  *g = final_n >> 8;
-  *b = final_n;
+struct pixel {
+  int final_n;
+};
+
+struct image {
+  int width;
+  int height;
+  struct pixel *pixels;
+};
+
+struct image *image_new(int width, int height) {
+  struct image *img = calloc(1, sizeof(*img));
+  img->width = width;
+  img->height = height;
+  img->pixels = calloc(width * height, sizeof(*img->pixels));
+  return img;
+}
+
+void image_free(struct image *img) {
+  free(img->pixels);
+  free(img);
+}
+
+void image_write(struct image *img) {
+  printf("P6\n%d %d\n255\n", img->width, img->height);
+  for (int j = 0; j < img->height; ++j) {
+    for (int i = 0; i < img->width; ++i) {
+      int n = img->pixels[j * img->width + i].final_n;
+      putchar(n >> 16);
+      putchar(n >>  8);
+      putchar(n      );
+    }
+  }
 }
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index 0d576f9..8c80a61 100644
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -24,13 +24,8 @@ int FNAME(calculate)(complex FTYPE c, int maximum_iterations, FTYPE escape_radiu

```

```
-void FNAME(render)(int width, int height, complex FTYPE center, FTYPE radius, int maximum_ite
-  printf("P6\n%d %d\n255\n", width, height);
-  for (int j = 0; j < height; ++j) {
-    for (int i = 0; i < width; ++i) {
-
    ↳ complex FTYPE c = FNAME(coordinate)(i, j, width, height, center, radius);
+void FNAME(render)(struct image *img, complex FTYPE center, FTYPE
    ↳ radius, int maximum_iterations, FTYPE escape_radius2) {
+  for (int j = 0; j < img->height; ++j) {
+    for (int i = 0; i < img->width; ++i) {
+      complex FTYPE c = FNAME(coordinate)(i, j, img->width, img->
    ↳ height, center, radius);
       int final_n = FNAME(calculate)(c, maximum_iterations,
           ↳ escape_radius2);
-      int r, g, b;
-      colour(&r, &g, &b, final_n);
-      putchar(r);
-      putchar(g);
-      putchar(b);
+      img->pixels[j * img->width + i].final_n = final_n;
     }
diff --git a/render.c b/render.c
index ba89d16..58f4d0e 100644
--- a/render.c
+++ b/render.c
@@ -22,13 +22,16 @@ int main(int argc, char **argv) {
   }
+  struct image *img = image_new(width, height);
   switch (float_type) {
     case 0:
-
    ↳ renderf(width, height, center, radius, maximum_iterations, escape_radius2);
+      renderf(img, center, radius, maximum_iterations,
    ↳ escape_radius2);
       break;
     case 1:
-
    ↳ render (width, height, center, radius, maximum_iterations, escape_radius2);
+      render (img, center, radius, maximum_iterations,
    ↳ escape_radius2);
       break;
     case 2:
-
    ↳ renderl(width, height, center, radius, maximum_iterations, escape_radius2);
+      renderl(img, center, radius, maximum_iterations,
    ↳ escape_radius2);
       break;
   }
+  image_write(img);
+  image_free(img);
   return 0;
```

1.22 Parallel rendering with OpenMP.

Modern computers have increasing numbers of CPU cores. So far our program only uses one. The OpenMP system uses compiler "pragmas" to annotate the program, letting the compiler know which parts can be parallelized safely. We couldn't use this before when we were outputting pixels as they were being calculated, because the order would be jumbled: some pixels might take longer to compute because they needed more iterations to escape. By collecting all the results in memory and only then writing the image, we can now add a pragma in our render() function and the required compiler flag to the Makefile.

We parallelize the outermost loop, because parallism adds some overhead for synchronisation, and the inner loop might not perform enough real work to offset the increased overheads. Benchmarking the resulting program is quite pleasing:

```
$ time for exp in $(seq -w 0 15)
  do
    ./render \
      -1.759937295271429505091916757 \
       0.012591790526496335594898047 \
      2e-${exp} 2 > ${exp}-render.ppm
  done
real    3m51.435s
user    3m50.862s
sys     0m0.452s
$ make
$ time for exp in $(seq -w 0 15)
  do
    ./render \
      -1.759937295271429505091916757 \
       0.012591790526496335594898047 \
      2e-${exp} 2 > ${exp}-render.ppm
  done
real    1m12.379s
user    3m25.829s
sys     0m0.308s
$
```

On a quad-core machine, this tiny change makes the rendering finish in a quarter of the real (wall clock) time.

```
diff --git a/Makefile b/Makefile
index 6ef9ffe..7cd9766 100644
--- a/Makefile
+++ b/Makefile
@@ -6,2 +6,2 @@ colour: colour.c
 render: render.c mandelbrot.h mandelbrot_imp.c
-	gcc -std=c99 -Wall -Wextra -pedantic -O3 -o render render.c -lm
+	gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -o render render.c -lm
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index 8c80a61..e160302 100644
```

```
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -25,2 +25,3 @@ int FNAME(calculate)(complex FTYPE c, int ↙
    ↪ maximum_iterations, FTYPE escape_radiu
 void FNAME(render)(struct image *img, complex FTYPE center, FTYPE↙
     ↪ radius, int maximum_iterations, FTYPE escape_radius2) {
+  #pragma omp parallel for
   for (int j = 0; j < img->height; ++j) {
```

1.23 Final angle colouring.

First we coloured according to whether the point escaped or not. Then we augmented with extra information that we had previously discarded: the escape time. There's more information that we are discarding: maybe we can use the final value of the iterate z in some way. We modify our pixel type to include this, choosing a single precision type because our aim is colouring and colours have limited precision. We modify our calculate() function to write to a pixel passed as a pointer, as functions have at most one return value. We no longer need to return anything, so we give a void return type. We change our render function to pass the address (the "&" operator) of the pixel.

When saving the image, we now have more information to write. We're already abusing PPM to store 24bit values packed into 3 8bit colour channels, perhaps we can use the same technique for the final z. We know that the magnitude of the final z is greater than our escape radius 2, but we don't have an upper bound (suppose we have a very zoomed out image). Any rescaling to a 24bit integer that we can pack into bytes would have an arbitrary bound.

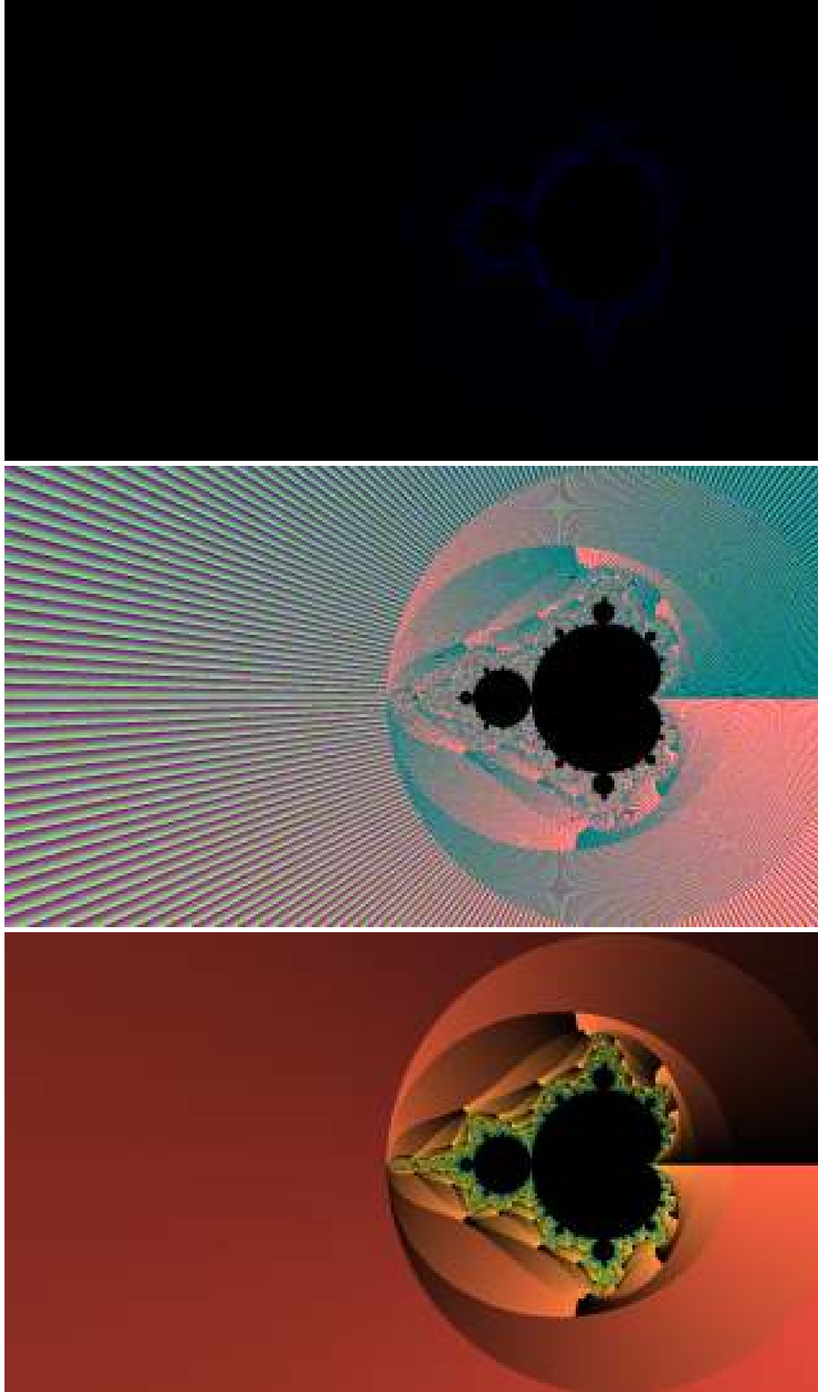
Complex numbers expressed as a sum of real and imaginary parts are in rectangular (Cartesian) form. We have the magnitude by Pythagoras Theorem, $|x + iy|^2 = x^2 + y^2$, and considering the angle that this triangle makes with the positive real axis gives the phase (also called argument): $arg(x + iy) = atan(y/x)$. Angles are unique up to multiples of a whole turn (2 pi), so maths conventionalized the prinicipal argument to be in [-pi,pi]. Given a fixed range, we can rescale it to the fixed range for our PPM output: first we normalize to [-0.5,0.5], we shift it by 1 to [0.5,1.5], and then we use the fmod() function to take the fractional part, giving us a range of [0,1]. Multiplying by the number of values in our 24bit range gives us an integer that we can pack into bytes as before.

So far we've been writing to the standard output stream and redirecting it to a file with the shell. Now we have two images to write. Our render program now takes a stem string on the command line, which we use to generate filenames in our image_save() function (modified from image_write()). To form the filenames, we use the snprintf() function − similar to printf but stores the output in a string. We need some memory to store the output, so we use calloc(), and we need to know how much space to allocate, so we use the length function strlen() defined in string.h. We need more space than the length of the stem, because we'll add a suffix to distinguish our two images.

We open the files for writing binary data using fopen(), and print the image header with fprintf(), similar to printf() but directing output to a particular file handle. Similarly we use fputc() instead of putchar(). We must fclose() the files when we have finished writing.

In our colour program, we generate filenames from a stem in the same way. We open them for reading binary data, and use fscanf() and fgetc() to read from the file handles instead of the standard input stream. We check that both images have the same dimensions. Then the same loop as before, only this time we read both files, and rescale the angle data to [0,1]. Our colour

function uses both sourcesinformation now: hue coming from the escape time, and value from the final angle.



```
diff --git a/colour.c b/colour.c
index cba7d71..9f44825 100644
--- a/colour.c
```

```
+++ b/colour.c
@@ -2,2 +2,4 @@
 #include <stdio.h>
+#include <stdlib.h>
+#include <string.h>

@@ -30,7 +32,6 @@ void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {

-void colour(int *r, int *g, int *b, int final_n) {
+void colour(int *r, int *g, int *b, int final_n, float final_z_arg) {
   float hue = final_n / 64.0f;
-  float sat = (final_n & 1) ? 0.75f : 0.25f;
-  float val = (final_n == 0) ? 0.0f :
-              (final_n & 2) ? 0.75f : 0.25f;
+  float sat = 0.75f;
+  float val = final_z_arg;
   float red, grn, blu;
@@ -42,22 +43,51 @@ void colour(int *r, int *g, int *b, int final_n) {

-int main() {
-  int width = 0;
-  int height = 0;
-  if (2 == scanf("P6 %d %d 255", &width, &height)) {
-    if (getchar() == '\n') {
-      printf("P6\n%d %d\n255\n", width, height);
-      for (int j = 0; j < height; ++j) {
-        for (int i = 0; i < width; ++i) {
-          int r = getchar();
-          int g = getchar();
-          int b = getchar();
-          int final_n = (r << 16) | (g << 8) | b;
-          colour(&r, &g, &b, final_n);
-          putchar(r);
-          putchar(g);
-          putchar(b);
-        }
-      }
+int main(int argc, char **argv) {
+  char *stem = "out";
+  if (argc > 1) {
+    stem = argv[1];
+  }
+  int length = strlen(stem) + 100;
+
+  char *fname_n = calloc(length, 1);
+  snprintf(fname_n, length, "%s_n.ppm", stem);
+  FILE *f_n = fopen(fname_n, "rb");
+  free(fname_n);
+  int width_n = 0;
+  int height_n = 0;
```

```
+  if (2 != fscanf(f_n, "P6␣%d␣%d␣255", &width_n, &height_n)) { return 1; }
+  if (fgetc(f_n) != '\n') { return 1; }
+
+  char *fname_a = calloc(length, 1);
+  snprintf(fname_a, length, "%s_z_arg.ppm", stem);
+  FILE *f_a = fopen(fname_a, "rb");
+  free(fname_a);
+  int width_a = 0;
+  int height_a = 0;
+  if (2 != fscanf(f_a, "P6␣%d␣%d␣255", &width_a, &height_a)) { return 1; }
+  if (fgetc(f_a) != '\n') { return 1; }
+
+  if (width_n != width_a) { return 1; }
+  if (height_n != height_a) { return 1; }
+  int width = width_n;
+  int height = height_n;
+
+  printf("P6\n%d␣%d\n255\n", width, height);
+  for (int j = 0; j < height; ++j) {
+    for (int i = 0; i < width; ++i) {
+      int r_n = fgetc(f_n);
+      int g_n = fgetc(f_n);
+      int b_n = fgetc(f_n);
+      int r_a = fgetc(f_a);
+      int g_a = fgetc(f_a);
+      int b_a = fgetc(f_a);
+      int   n =  (r_n << 16) | (g_n << 8) | b_n;
+      float a = ((r_a << 16) | (g_a << 8) | b_a) / (float) (1 << 24);
+      int r, g, b;
+      colour(&r, &g, &b, n, a);
+      putchar(r);
+      putchar(g);
+      putchar(b);
     }
   }
+
   return 0;
diff --git a/mandelbrot.h b/mandelbrot.h
index faf47f2..cd6a6a9 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -4,3 +4,6 @@
 #include <complex.h>
+#include <math.h>
 #include <stdio.h>
+#include <stdlib.h>
+#include <string.h>

@@ -8,2 +11,3 @@ struct pixel {
   int final_n;
```

```
+  complex float final_z;
 };
@@ -16,27 +20,2 @@ struct image {

-struct image *image_new(int width, int height) {
-  struct image *img = calloc(1, sizeof(*img));
-  img->width = width;
-  img->height = height;
-  img->pixels = calloc(width * height, sizeof(*img->pixels));
-  return img;
-}
-
-void image_free(struct image *img) {
-  free(img->pixels);
-  free(img);
-}
-
-void image_write(struct image *img) {
-  printf("P6\n%d %d\n255\n", img->width, img->height);
-  for (int j = 0; j < img->height; ++j) {
-    for (int i = 0; i < img->width; ++i) {
-      int n = img->pixels[j * img->width + i].final_n;
-      putchar(n >> 16);
-      putchar(n >>  8);
-      putchar(n      );
-    }
-  }
-}
-
 #define FTYPE float
@@ -59,2 +38,51 @@ void image_write(struct image *img) {

+struct image *image_new(int width, int height) {
+  struct image *img = calloc(1, sizeof(*img));
+  img->width = width;
+  img->height = height;
+  img->pixels = calloc(width * height, sizeof(*img->pixels));
+  return img;
+}
+
+void image_free(struct image *img) {
+  free(img->pixels);
+  free(img);
+}
+
+void image_save(struct image *img, char *filestem) {
+  int length = strlen(filestem) + 100;
+  char *filename = calloc(length, 1);
+  {
+    snprintf(filename, length, "%s_n.ppm", filestem);
+    FILE *f = fopen(filename, "wb");
+    fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
+    for (int j = 0; j < img->height; ++j) {
```

```
+      for (int i = 0; i < img->width; ++i) {
+        int n = img->pixels[j * img->width + i].final_n;
+        fputc(n >> 16, f);
+        fputc(n >>  8, f);
+        fputc(n      , f);
+      }
+    }
+    fclose(f);
+  }
+  {
+    snprintf(filename, length, "%s_z_arg.ppm", filestem);
+    FILE *f = fopen(filename, "wb");
+    fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
+    for (int j = 0; j < img->height; ++j) {
+      for (int i = 0; i < img->width; ++i) {
+        complex float z = img->pixels[j * img->width + i].final_z;
+        float t = fmodf(carg(z) / (2.0f * pif) + 1.0f, 1.0f);
+        int n = (1 << 24) * t;
+        fputc(n >> 16, f);
+        fputc(n >>  8, f);
+        fputc(n      , f);
+      }
+    }
+    fclose(f);
+  }
+  free(filename);
+}
+
 #endif
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index e160302..594fc32 100644
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -1 +1,3 @@
+FTYPE FNAME(pi) = FNAME(3.14159265358979323846264338327950288419716939937510);
+
 FTYPE FNAME(cabs2)(complex FTYPE z) {
@@ -11,5 +13,6 @@ complex FTYPE FNAME(coordinate)(int i, int j, int width, int height, complex FTY

-int FNAME(calculate)(complex FTYPE c, int maximum_iterations, FTYPE escape_radius2) {
+void FNAME(calculate)(struct pixel *out, complex FTYPE c, int maximum_iterations, FTYPE escape_radius2) {
+  out->final_n = 0;
+  out->final_z = 0;
   complex FTYPE z = 0;
-  int final_n = 0;
   for (int n = 1; n < maximum_iterations; ++n) {
@@ -17,3 +20,4 @@ int FNAME(calculate)(complex FTYPE c, int maximum_iterations, FTYPE escape_radiu
     if (FNAME(cabs2)(z) > escape_radius2) {
```

```
-        final_n = n;
+        out->final_n = n;
+        out->final_z = z;
         break;
@@ -21,3 +25,2 @@ int FNAME(calculate)(complex FTYPE c, int maximum_iterations, FTYPE escape_radiu
     }
-    return final_n;
 }
@@ -29,4 +32,3 @@ void FNAME(render)(struct image *img, complex FTYPE center, FTYPE radius, int ma
         complex FTYPE c = FNAME(coordinate)(i, j, img->width, img->height, center, radius);
-        int final_n = FNAME(calculate)(c, maximum_iterations, escape_radius2);
-        img->pixels[j * img->width + i].final_n = final_n;
+        FNAME(calculate)(&img->pixels[j * img->width + i], c, maximum_iterations, escape_radius2);
       }
diff --git a/render.c b/render.c
index 58f4d0e..b5828c9 100644
--- a/render.c
+++ b/render.c
@@ -22,2 +22,6 @@ int main(int argc, char **argv) {
   }
+  char *stem = "out";
+  if (argc > 5) {
+    stem = argv[5];
+  }
   struct image *img = image_new(width, height);
@@ -34,3 +38,3 @@ int main(int argc, char **argv) {
   }
-  image_write(img);
+  image_save(img, stem);
   image_free(img);
```

1.24 Fixing bugs.

Our programs crash if we try to access files that don't exist (reading from a file that isn't there, or trying to save a file into a directory that isn't there):

```
$ ./colour nonexistant >out.ppm
Segmentation fault
$ ./render 0 0 2 1 directory/nonexistant
Segmentation fault
$
```

We forgot to check the result of fopen(): it returns a null pointer (one that points nowhere) if the file couldn't be opened. When reading or writing to a file, the C library code dereferences the file pointer, and dereferencing a null pointer causes a segfault crash. We fix the crash in colour.c by exiting with an error if the file pointer is null, and also fix a lesser bug by remembering to

close the files we've opened (the C library will close all the open files and free all memory when the program ends, but for longer-running programs it's important to clean up as you go).

We have the same bug in the image_save() function used by the render program, to fix the crash we do nothing if the file couldn't be opened for writing, while noting that this is far from ideal: we might lose a lot of calculation effort without noticing that the files weren't saved correctly.

```
diff --git a/colour.c b/colour.c
index 9f44825..d6e0acf 100644
--- a/colour.c
+++ b/colour.c
@@ -53,2 +53,3 @@ int main(int argc, char **argv) {
   FILE *f_n = fopen(fname_n, "rb");
+  if (! f_n) { return 1; }
   free(fname_n);
@@ -62,2 +63,3 @@ int main(int argc, char **argv) {
   FILE *f_a = fopen(fname_a, "rb");
+  if (! f_a) { return 1; }
   free(fname_a);
@@ -92,2 +94,5 @@ int main(int argc, char **argv) {

+  fclose(f_n);
+  fclose(f_a);
+
   return 0;
diff --git a/mandelbrot.h b/mandelbrot.h
index cd6a6a9..3bf4350 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -54,5 +54,6 @@ void image_save(struct image *img, char *filestem) {
   char *filename = calloc(length, 1);
-  {
-    snprintf(filename, length, "%s_n.ppm", filestem);
-    FILE *f = fopen(filename, "wb");
+
+  snprintf(filename, length, "%s_n.ppm", filestem);
+  FILE *f = fopen(filename, "wb");
+  if (f) {
     fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
@@ -68,5 +69,6 @@ void image_save(struct image *img, char *filestem) {
   }
-  {
-    snprintf(filename, length, "%s_z_arg.ppm", filestem);
-    FILE *f = fopen(filename, "wb");
+
+  snprintf(filename, length, "%s_z_arg.ppm", filestem);
+  f = fopen(filename, "wb");
+  if (f) {
     fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
@@ -84,2 +86,3 @@ void image_save(struct image *img, char *filestem) {
   }
```

```
+
  free(filename);
```

1.25 Defensive error handling.

It's possible for calloc() to fail, if we run out of memory. So we check the result for null pointers. We return a null pointer from our image_new() to indicate failure. In our image_save() which previously failed silently, we check for errors and return an int to indicate success or failure. In the main render program we check for errors and print messages to the standard error stream to let us know what went wrong, and return a status code to the shell:

```
$ ./render 0 0 2 1 directory/nonexistant
ERROR image save failed
$
```

```
diff --git a/mandelbrot.h b/mandelbrot.h
index 3bf4350..33c6c84 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -40,2 +40,3 @@ struct image *image_new(int width, int height) {
  struct image *img = calloc(1, sizeof(*img));
+ if (! img) { return 0; }
  img->width = width;
@@ -43,2 +44,3 @@ struct image *image_new(int width, int height) {
  img->pixels = calloc(width * height, sizeof(*img->pixels));
+ if (! img->pixels) { free(img); return 0; }
  return img;
@@ -51,39 +53,51 @@ void image_free(struct image *img) {

-void image_save(struct image *img, char *filestem) {
+int image_save(struct image *img, char *filestem) {
+ int retval = 0;
+
  int length = strlen(filestem) + 100;
  char *filename = calloc(length, 1);
+ if (filename) {

- snprintf(filename, length, "%s_n.ppm", filestem);
- FILE *f = fopen(filename, "wb");
- if (f) {
-   fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
-   for (int j = 0; j < img->height; ++j) {
-     for (int i = 0; i < img->width; ++i) {
-       int n = img->pixels[j * img->width + i].final_n;
-       fputc(n >> 16, f);
-       fputc(n >>  8, f);
-       fputc(n      , f);
+   snprintf(filename, length, "%s_n.ppm", filestem);
+   FILE *f = fopen(filename, "wb");
+   if (f) {
+     fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
```

```
+      for (int j = 0; j < img->height; ++j) {
+        for (int i = 0; i < img->width; ++i) {
+          int n = img->pixels[j * img->width + i].final_n;
+          fputc(n >> 16, f);
+          fputc(n >>  8, f);
+          fputc(n      , f);
+        }
       }
+      fclose(f);
+    } else {
+      retval = 1;
     }
-      fclose(f);
-  }

-    snprintf(filename, length, "%s_z_arg.ppm", filestem);
-    f = fopen(filename, "wb");
-    if (f) {
-      fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
-      for (int j = 0; j < img->height; ++j) {
-        for (int i = 0; i < img->width; ++i) {
-          complex float z = img->pixels[j * img->width + i].final_z;
-          float t = fmodf(carg(z) / (2.0f * pif) + 1.0f, 1.0f);
-          int n = (1 << 24) * t;
-          fputc(n >> 16, f);
-          fputc(n >>  8, f);
-          fputc(n      , f);
+    snprintf(filename, length, "%s_z_arg.ppm", filestem);
+    f = fopen(filename, "wb");
+    if (f) {
+      fprintf(f, "P6\n%d␣%d\n255\n", img->width, img->height);
+      for (int j = 0; j < img->height; ++j) {
+        for (int i = 0; i < img->width; ++i) {
+          complex float z = img->pixels[j * img->width + i].↙
    ↳ final_z;
+          float t = fmodf(carg(z) / (2.0f * pif) + 1.0f, 1.0f);
+          int n = (1 << 24) * t;
+          fputc(n >> 16, f);
+          fputc(n >>  8, f);
+          fputc(n      , f);
+        }
       }
+      fclose(f);
+    } else {
+      retval = 1;
     }
-      fclose(f);
+
+    free(filename);
+  } else {
+    retval = 1;
   }
```

```
-   free(filename);
+   return retval;
 }
diff --git a/render.c b/render.c
index b5828c9..a003d5e 100644
--- a/render.c
+++ b/render.c
@@ -7,2 +7,3 @@
 int main(int argc, char **argv) {
+   int retval = 1;
    int width = 1280;
@@ -27,16 +28,25 @@ int main(int argc, char **argv) {
    struct image *img = image_new(width, height);
-   switch (float_type) {
-     case 0:
-       ↙
    ↳ renderf(img, center, radius, maximum_iterations, escape_radius2);
-       break;
-     case 1:
-       ↙
    ↳ render (img, center, radius, maximum_iterations, escape_radius2);
-       break;
-     case 2:
-       ↙
    ↳ renderl(img, center, radius, maximum_iterations, escape_radius2);
-       break;
+   if (img) {
+     switch (float_type) {
+       case 0:
+         renderf(img, center, radius, maximum_iterations, ↙
    ↳ escape_radius2);
+         break;
+       case 1:
+         render (img, center, radius, maximum_iterations, ↙
    ↳ escape_radius2);
+         break;
+       case 2:
+         renderl(img, center, radius, maximum_iterations, ↙
    ↳ escape_radius2);
+         break;
+     }
+     int err = image_save(img, stem);
+     image_free(img);
+     if (! err) {
+       retval = 0;
+     } else {
+       fprintf(stderr, "ERROR image save failed\n");
+     }
+   } else {
+     fprintf(stderr, "ERROR image new failed\n");
    }
-   image_save(img, stem);
-   image_free(img);
```

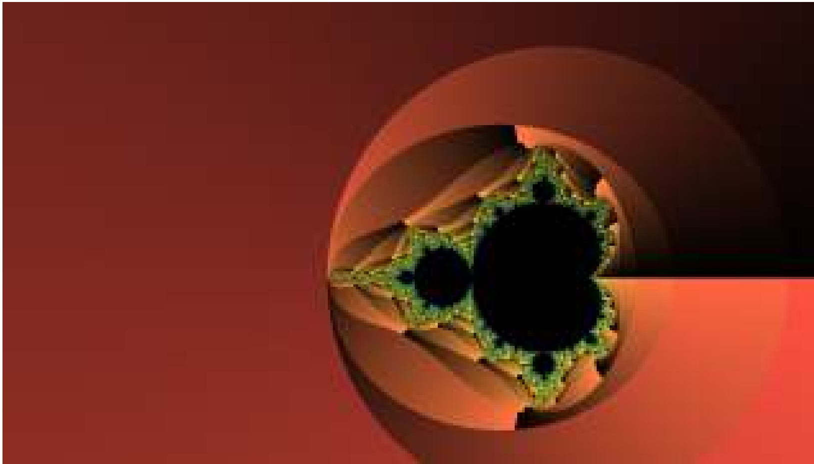
```
-  return 0;
+  return retval;
 }
```

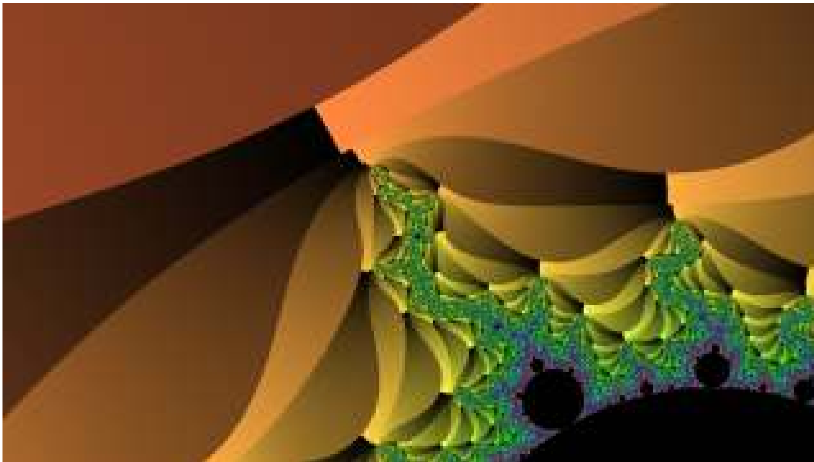
1.26 Automatic number type selection.

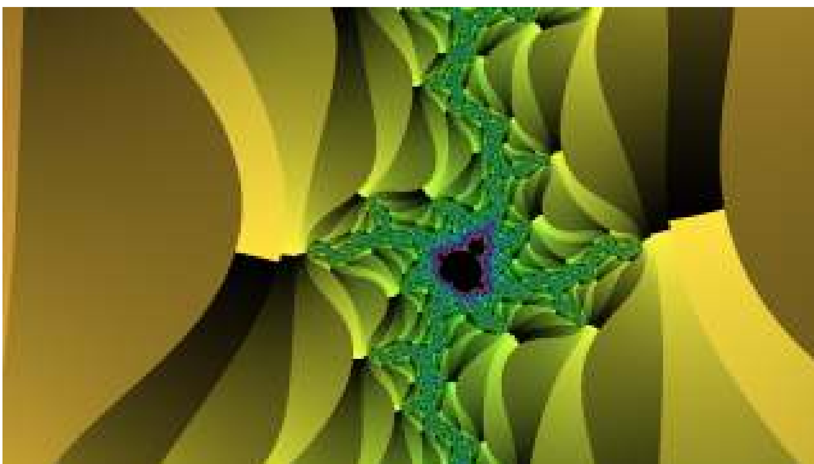
We're telling our render program which number type to use. We can improve this by calculating the best number type to use, using the spacing between pixels. We compute the number of bits of precision needed to distinguish points with a magnitude around 1 using a base-2 logarithm. We default to double precision because it's the fastest, but if we need more precision we upgrade to long double. If long double still insuffices, we print a warning message on the standard error stream. We also print information messages so we know which number type our program is using.

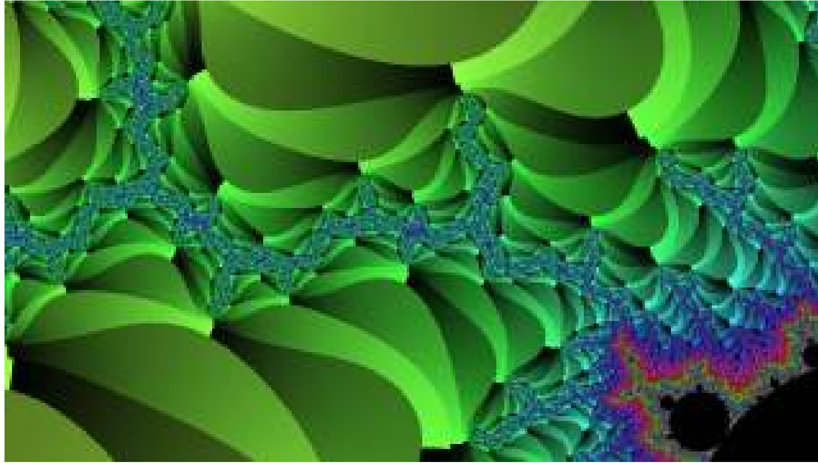
```
$ for exp in $(seq -w 0 20)
  do
    ./render %\
      -1.260318889628377388064386083141 84 %\
       0.383472140271138377248417651135 96 %\
      2e-${exp} ${exp} \&\&
    ./colour ${exp} > ${exp}.ppm
  done
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using double
INFO using long double
INFO using long double
INFO using long double
WARNING insufficient precision (need 63 bits)
INFO using long double
WARNING insufficient precision (need 67 bits)
INFO using long double
WARNING insufficient precision (need 70 bits)
INFO using long double
WARNING insufficient precision (need 73 bits)
INFO using long double
$
```

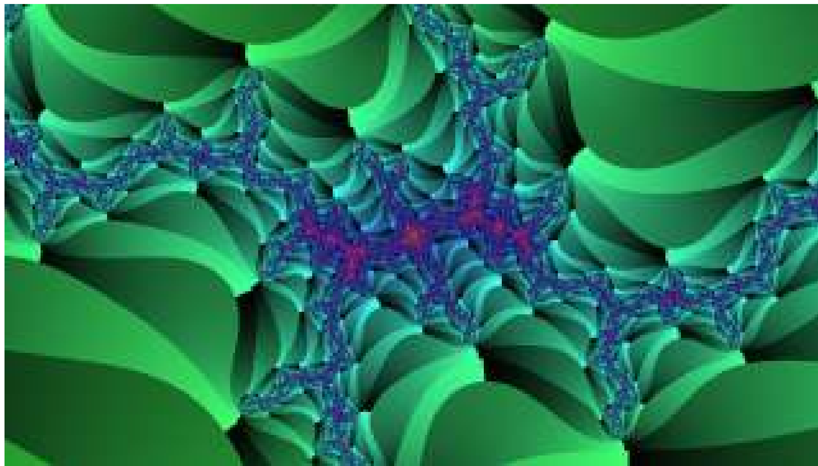
The last 4 images are indeed increasingly pixelated.

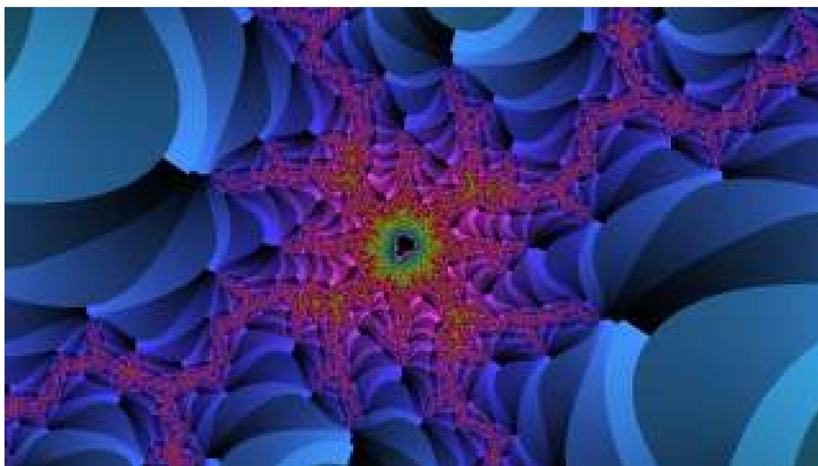


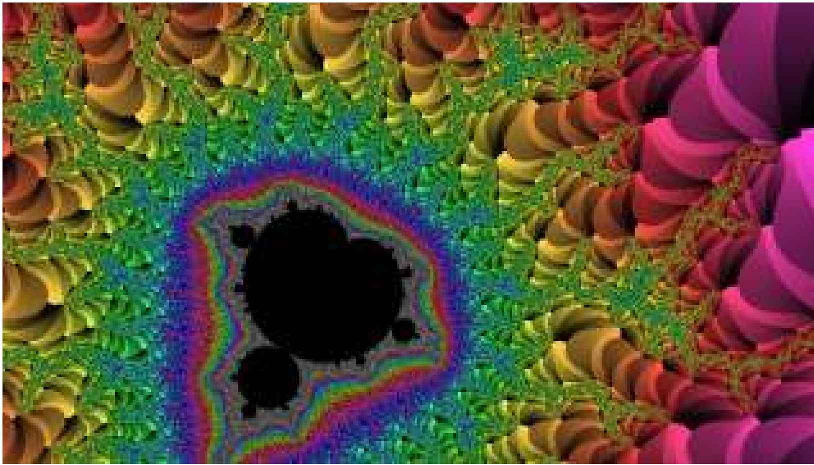


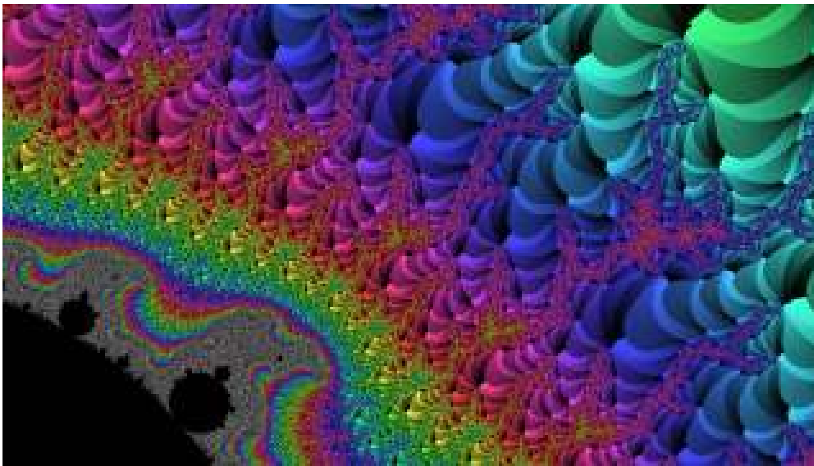


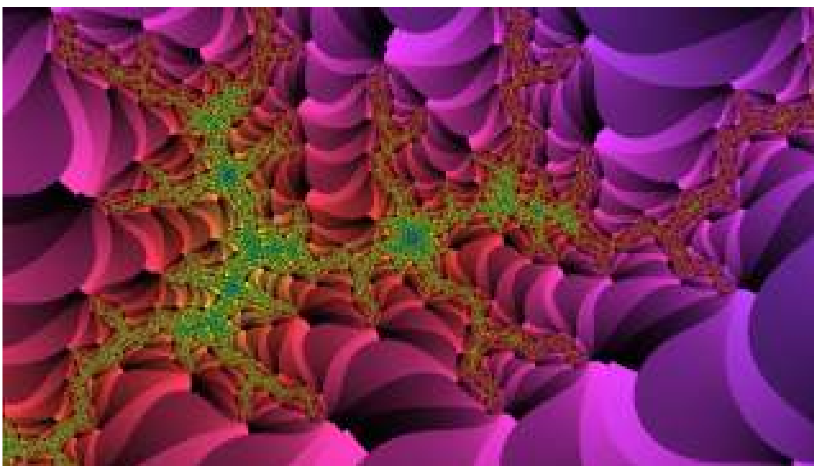


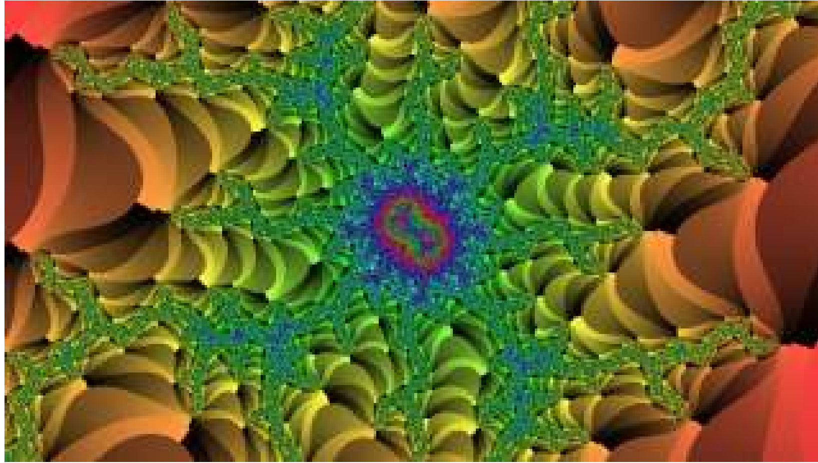


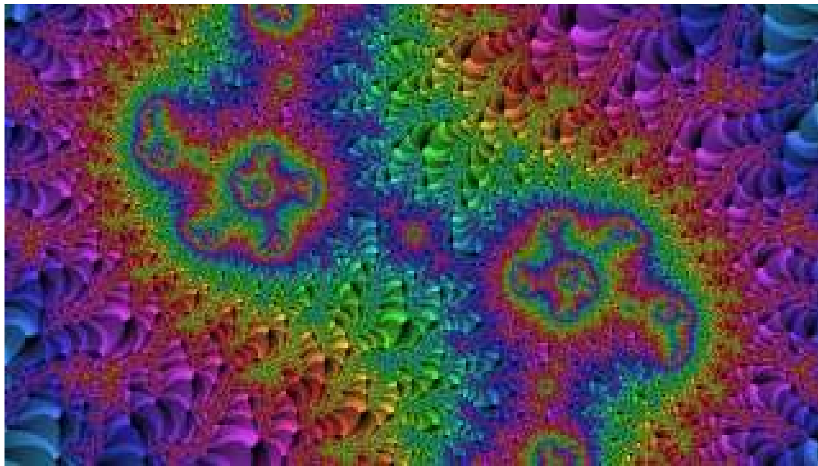


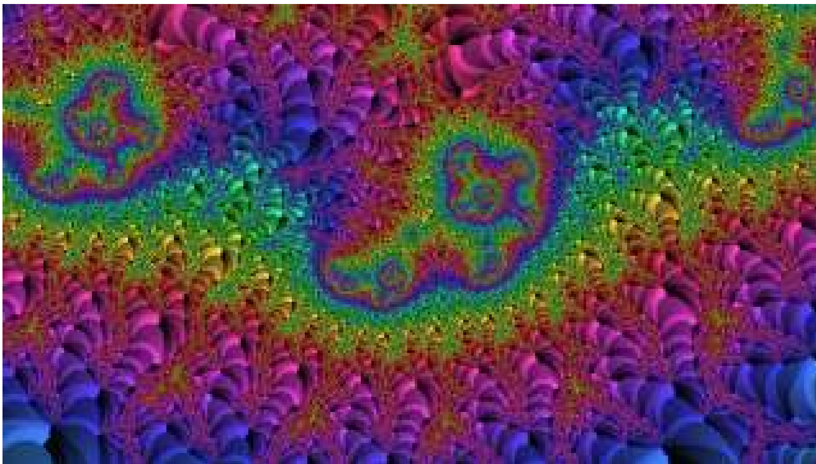


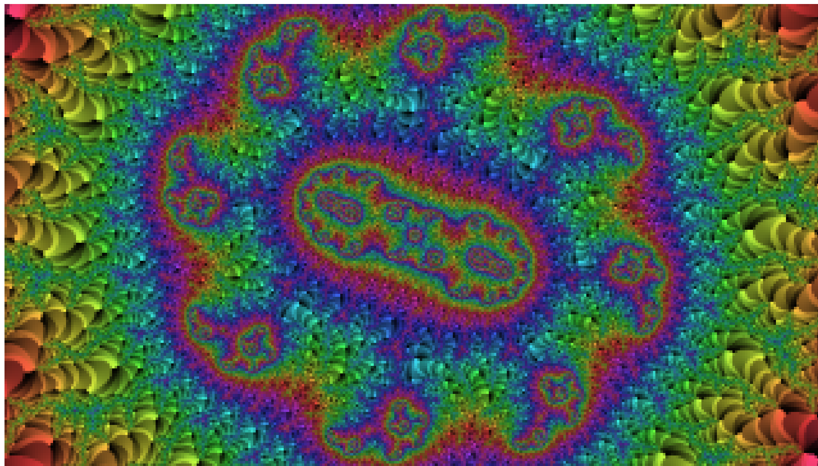


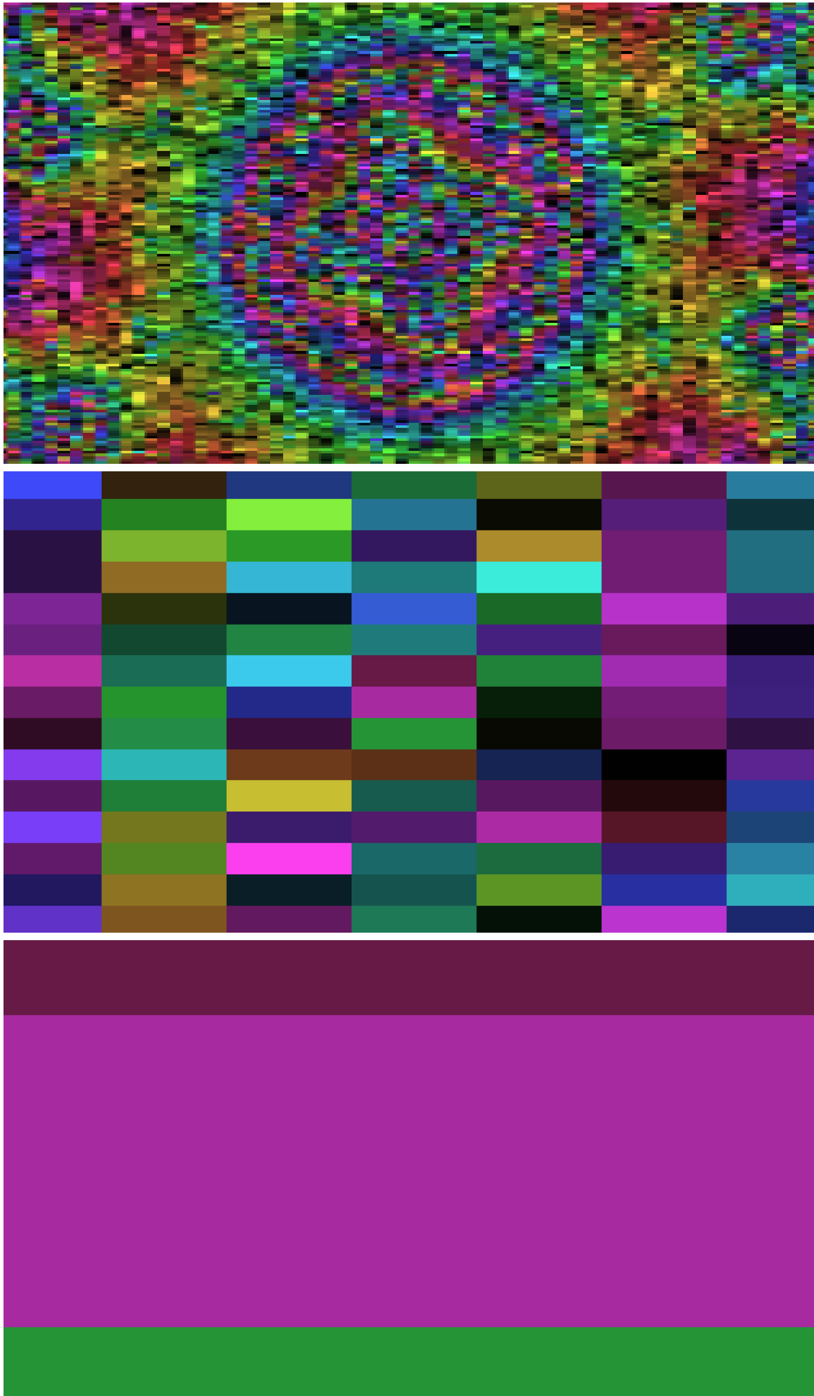












```
diff --git a/render.c b/render.c
index a003d5e..1ed52aa 100644
--- a/render.c
+++ b/render.c
@@ -15,3 +15,2 @@ int main(int argc, char **argv) {
```

```
   long double escape_radius2 = escape_radius * escape_radius;
-  int float_type = 1;
   if (argc > 3) {
@@ -20,9 +19,16 @@ int main(int argc, char **argv) {
   }
+  char *stem = "out";
   if (argc > 4) {
-    float_type = atoi(argv[4]);
+    stem = argv[4];
   }
-  char *stem = "out";
-  if (argc > 5) {
-    stem = argv[5];
+  int float_type = 1;
+  long double pixel_spacing = radius / (height / 2.0);
+  int pixel_spacing_bits = -log2l(pixel_spacing);
+  if (pixel_spacing_bits > 50) {
+    float_type = 2;
+  }
+  if (pixel_spacing_bits > 60) {
+    fprintf(stderr, "WARNING insufficient precision (need %d bits
   ↳ )\n", pixel_spacing_bits);
   }
+
   struct image *img = image_new(width, height);
@@ -31,2 +37,3 @@ int main(int argc, char **argv) {
       case 0:
+        fprintf(stderr, "INFO using float\n");
        renderf(img, center, radius, maximum_iterations,
            ↳ escape_radius2);
@@ -34,2 +41,3 @@ int main(int argc, char **argv) {
      case 1:
+        fprintf(stderr, "INFO using double\n");
        render (img, center, radius, maximum_iterations,
            ↳ escape_radius2);
@@ -37,2 +45,3 @@ int main(int argc, char **argv) {
      case 2:
+        fprintf(stderr, "INFO using long double\n");
        renderl(img, center, radius, maximum_iterations,
            ↳ escape_radius2);
```

1.27 Final radius colouring.

We've got the final z iterate, but we're only using the phase because we weren't sure how to scale the magnitude. We know the magnitude is greater than the escape radius: $|z_n| > R$. We also know that the previous iterate's magnitude must have been less than R, otherwise we would have broken out of the loop earlier: $|z_{n-1}| <= R$. Combining with the definition of the iteration, $z_n = z_{n-1}^2 + c$, and using the triangle inequality $|a + b| <= |a| + |b|$, we get:

$$R < |z_n| <= |z_{n-1}|^2 + |c| <= R^2 + |c|$$

If we make R bigger, so that $|c|$ fades into insignificance, we get the approximation $R < |z_n| <=$

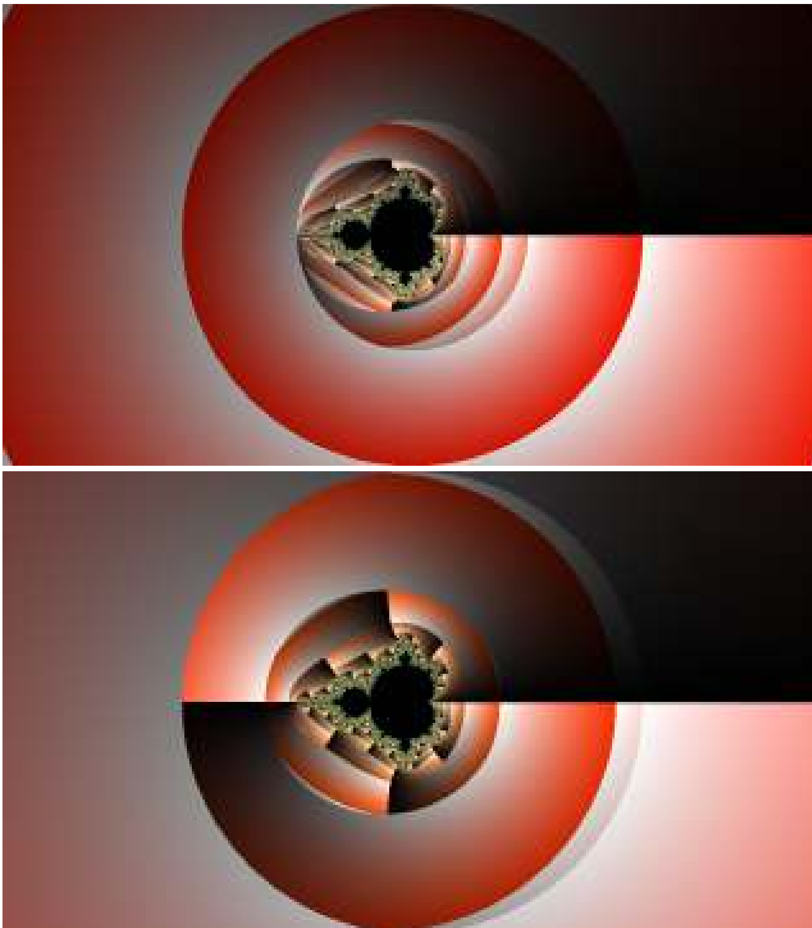
R^2 , and taking base-R logarithms we get

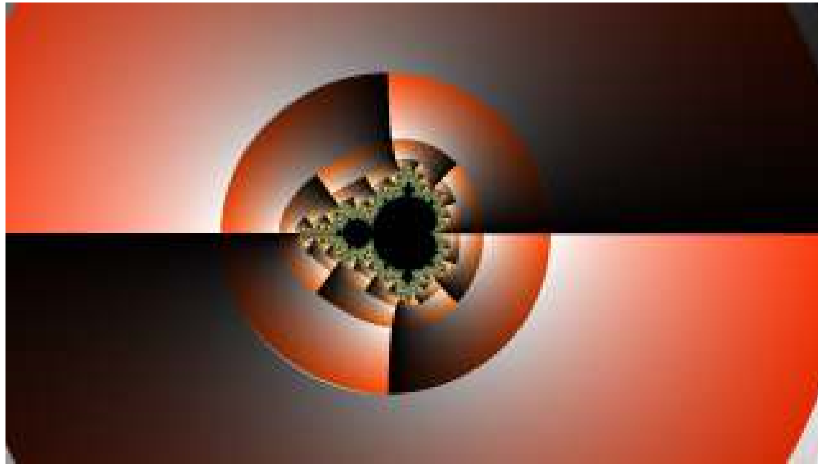
$$1 < log|z_n|/logR <= 2$$

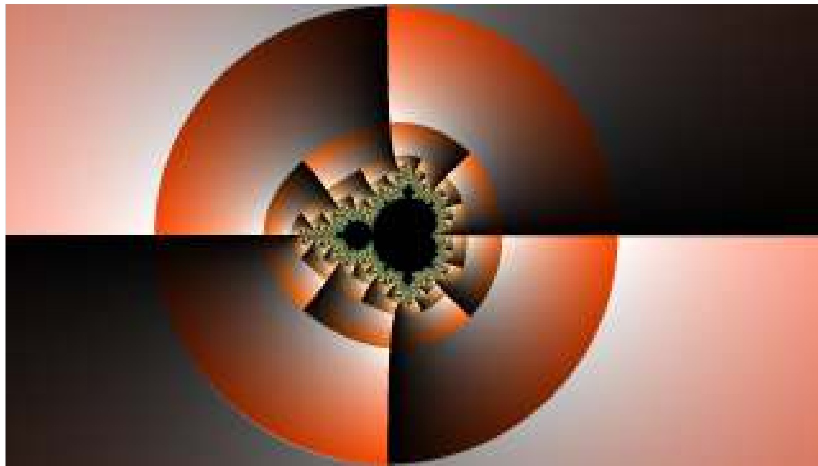
Now we can subtract 1 to get a value in [0,1], which we know how to pack into a 24bit PPM file. We can use this for colouring, and we can vary the escape radius to see how big it needs to be for the approximation to give acceptable results:

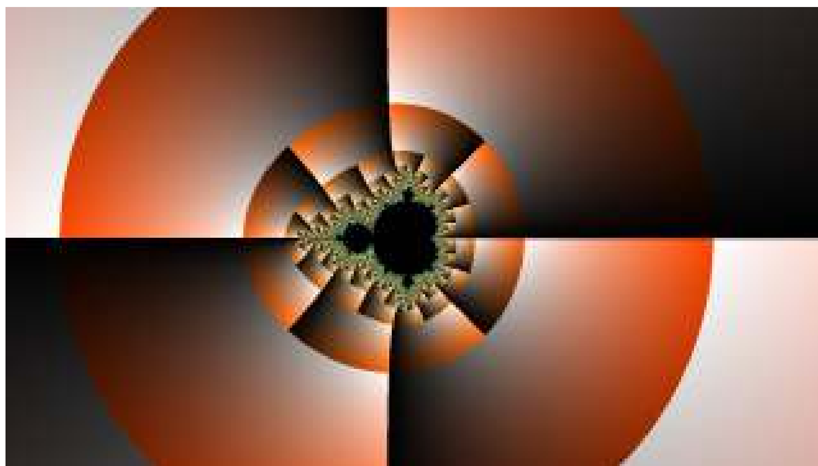
```
$ for er in 2 4 8 16 32 64 128 256 512 1024
  do
    ./render 0 0 4 ${er} ${er} &&
    ./colour ${er} > ${er}.ppm
  done
$
```

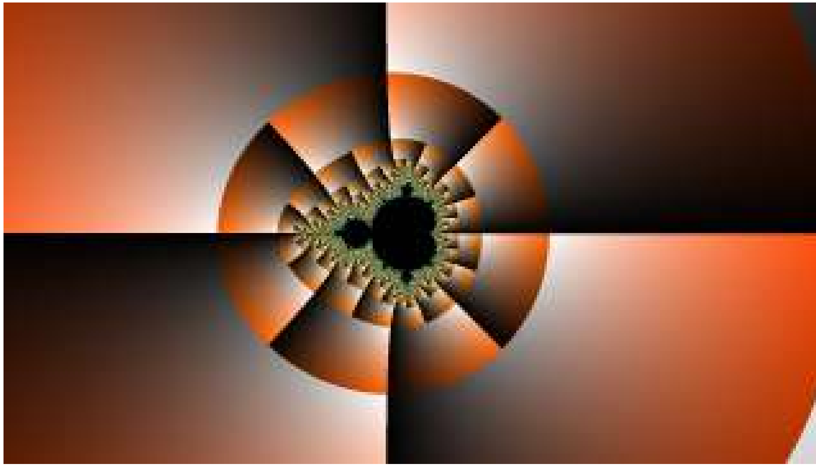
Looking at the output images, it's clear that the approximation behaves badly for small R, with glitches at the edges of the bands. We also see that the shapes change, with the cells formed by the angle colouring becoming longer and thinner. With the escape radius at 512, the cells look almost square, and there are no visible gltiches.

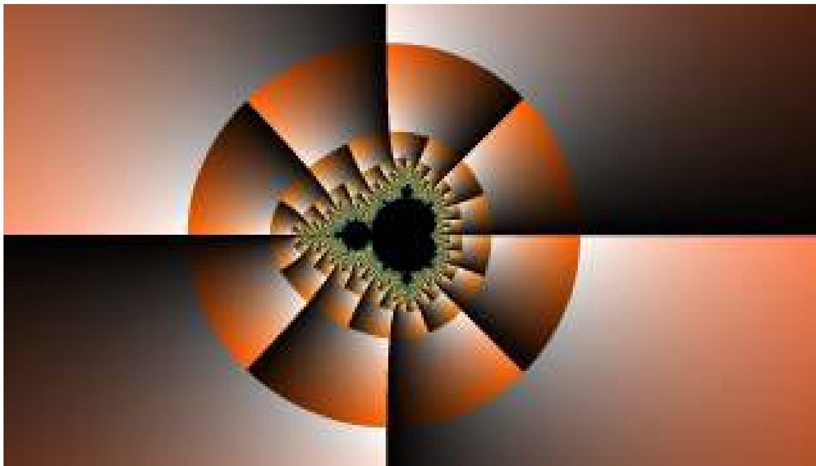


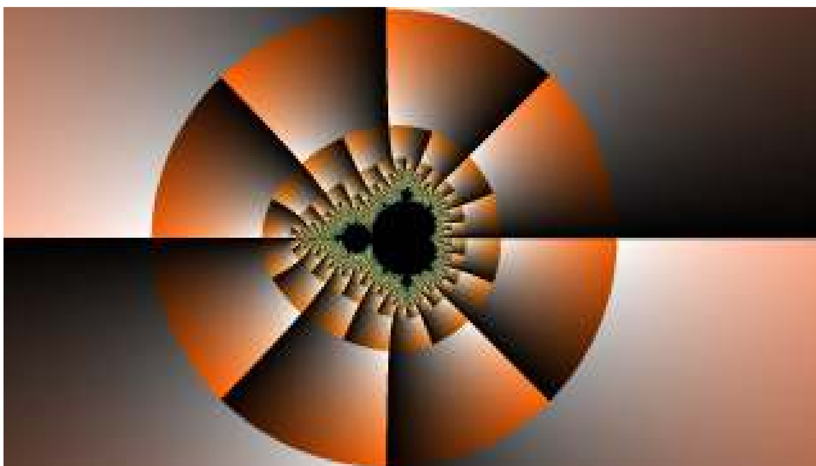


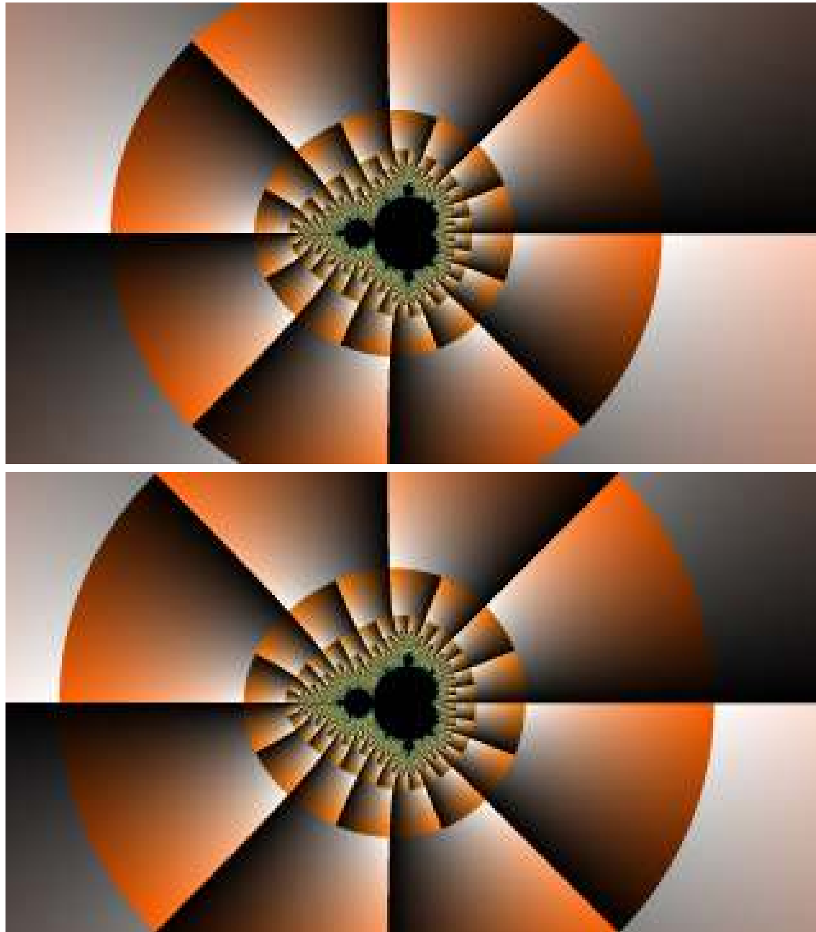












```
diff --git a/colour.c b/colour.c
index d6e0acf..9d3acdf 100644
--- a/colour.c
+++ b/colour.c
@@ -32,5 +32,5 @@ void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {

-void colour(int *r, int *g, int *b, int final_n, float final_z_arg) {
+void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_z_arg) {
   float hue = final_n / 64.0f;
-  float sat = 0.75f;
+  float sat = final_z_abs;
   float val = final_z_arg;
@@ -60,2 +60,12 @@ int main(int argc, char **argv) {

+  char *fname_r = calloc(length, 1);
+  snprintf(fname_r, length, "%s_z_abs.ppm", stem);
+  FILE *f_r = fopen(fname_r, "rb");
+  if (! f_r) { return 1; }
+  free(fname_r);
```

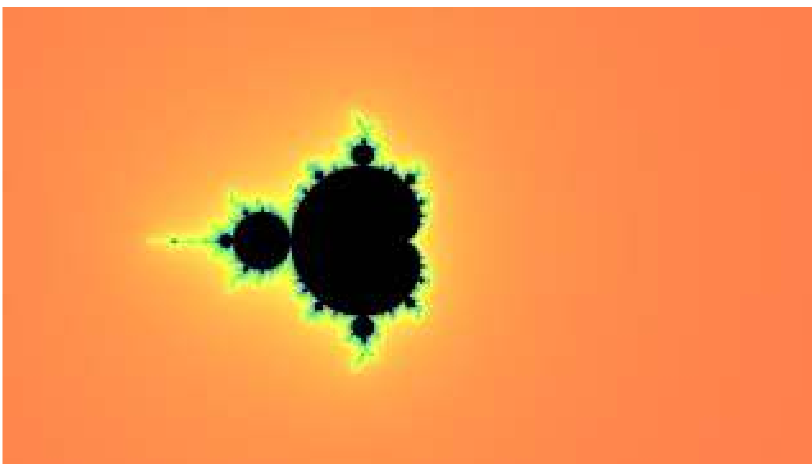
```
20 +   int width_r = 0;
   +   int height_r = 0;
   +   if (2 != fscanf(f_r, "P6 %d %d 255", &width_r, &height_r)) { ↙
       ↳ return 1; }
   +   if (fgetc(f_r) != '\n') { return 1; }
   +
25     char *fname_a = calloc(length, 1);
   @@ -70,3 +80,5 @@ int main(int argc, char **argv) {

   +   if (width_n != width_r) { return 1; }
       if (width_n != width_a) { return 1; }
30 +   if (height_n != height_r) { return 1; }
       if (height_n != height_a) { return 1; }
   @@ -81,2 +93,5 @@ int main(int argc, char **argv) {
           int b_n = fgetc(f_n);
   +       int r_r = fgetc(f_r);
35 +       int g_r = fgetc(f_r);
   +       int b_r = fgetc(f_r);
           int r_a = fgetc(f_a);
   @@ -85,8 +100,9 @@ int main(int argc, char **argv) {
           int   n =  (r_n << 16) | (g_n << 8) | b_n;
40 +       float r = ((r_r << 16) | (g_r << 8) | b_r) / (float) (1 << ↙
       ↳ 24);
           float a = ((r_a << 16) | (g_a << 8) | b_a) / (float) (1 << ↙
               ↳ 24);
   -       int r, g, b;
   -       colour(&r, &g, &b, n, a);
   -       putchar(r);
45 -       putchar(g);
   -       putchar(b);
   +       int red, grn, blu;
   +       colour(&red, &grn, &blu, n, r, a);
   +       putchar(red);
50 +       putchar(grn);
   +       putchar(blu);
         }
   diff --git a/mandelbrot.h b/mandelbrot.h
   index 33c6c84..a8be2fa 100644
55 --- a/mandelbrot.h
   +++ b/mandelbrot.h
   @@ -53,3 +53,3 @@ void image_free(struct image *img) {

   -int image_save(struct image *img, char *filestem) {
60 +int image_save(struct image *img, char *filestem, float ↙
       ↳ escape_radius) {
       int retval = 0;
   @@ -77,2 +77,21 @@ int image_save(struct image *img, char *↙
       ↳ filestem) {

   +     snprintf(filename, length, "%s_z_abs.ppm", filestem);
65 +     f = fopen(filename, "wb");
   +     if (f) {
   +       fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
```

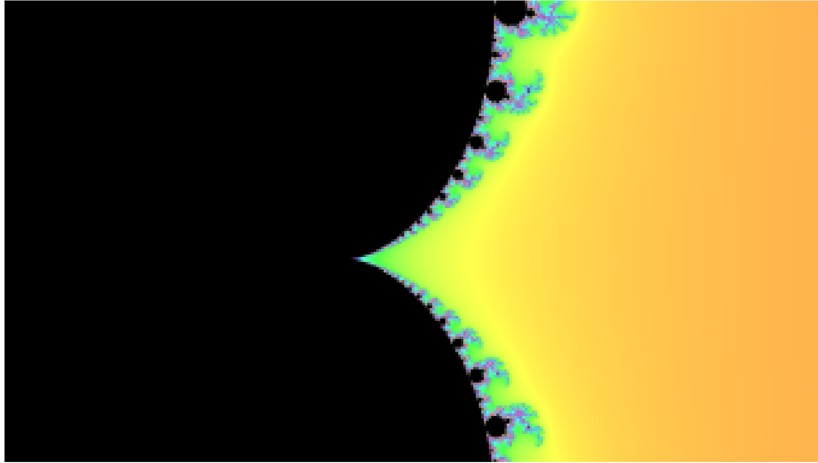
```
+       for (int j = 0; j < img->height; ++j) {
+         for (int i = 0; i < img->width; ++i) {
70 +          complex float z = img->pixels[j * img->width + i].↙
     ↳ final_z;
+           float r = logf(cabsf(z)) / logf(escape_radius) - 1.0f;
+           int n = (1 << 24) * r;
+           fputc(n >> 16, f);
+           fputc(n >>  8, f);
75 +           fputc(n      , f);
+         }
+       }
+       fclose(f);
+     } else {
80 +       retval = 1;
+     }
+
      snprintf(filename, length, "%s_z_arg.ppm", filestem);
@@ -84,3 +103,3 @@ int image_save(struct image *img, char *↙
     ↳ filestem) {
85            complex float z = img->pixels[j * img->width + i].↙
                ↳ final_z;
-           float t = fmodf(carg(z) / (2.0f * pif) + 1.0f, 1.0f);
+           float t = fmodf(cargf(z) / (2.0f * pif) + 1.0f, 1.0f);
            int n = (1 << 24) * t;
diff --git a/render.c b/render.c
90 index 1ed52aa..039253e 100644
--- a/render.c
+++ b/render.c
@@ -14,3 +14,2 @@ int main(int argc, char **argv) {
    long double escape_radius = 2;
95 -   long double escape_radius2 = escape_radius * escape_radius;
    if (argc > 3) {
@@ -19,5 +18,9 @@ int main(int argc, char **argv) {
    }
-   char *stem = "out";
100    if (argc > 4) {
-     stem = argv[4];
+     escape_radius = strtold(argv[4], 0);
+   }
+   long double escape_radius2 = escape_radius * escape_radius;
105 +   char *stem = "out";
+   if (argc > 5) {
+     stem = argv[5];
    }
@@ -49,3 +52,3 @@ int main(int argc, char **argv) {
110      }
-     int err = image_save(img, stem);
+     int err = image_save(img, stem, escape_radius);
      image_free(img);
```

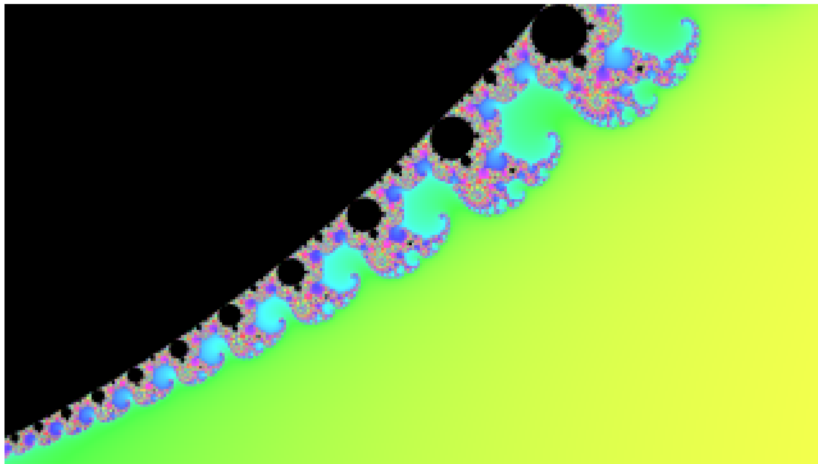
1.28 Smooth colouring with continuous escape time.

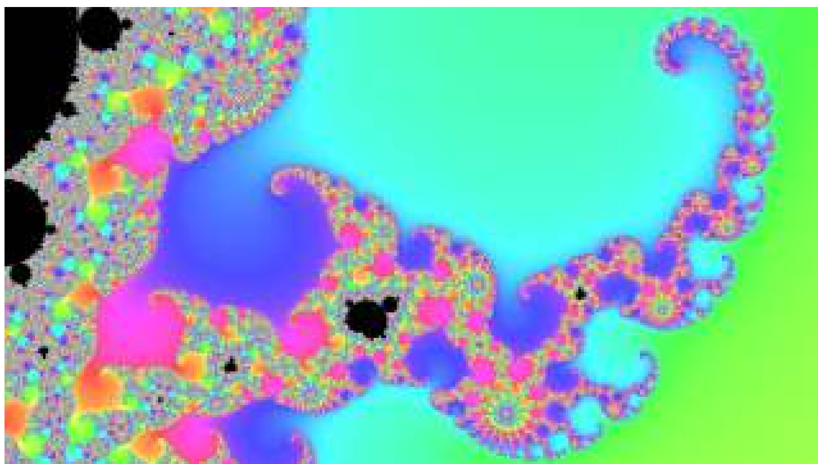
The escape time bands around the Mandelbrot set increase stepwise by 1. We now have the final iterate radius, which we can use to smooth the colouring. If the final radius is large, then the previous iterate was close to escaping. If the final radius is small, then the previous iterate was further from escaping. Here we assume the escape radius is large enough for the approximation ignoring |c| to hold. Squaring gives a curve, which we can linearize using logarithms. We have a value in [0,1], but log 0 is undefined (it heads towards negative infinity), moreover we still want to end up with a value in [0,1]. Adding 1 gives a value in [1,2], and taking the base-2 logarithm gives the required range, as $log_B 1 = 0$ and $log_B B = 1$ for all B. We subtract this value from the integer escape time, giving a fractional escape time with continuity at the transitions between bands and smoothly increasing size.

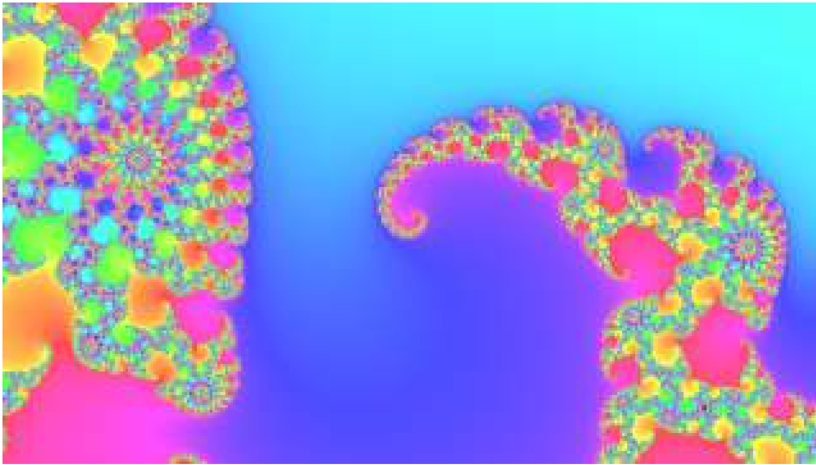
```
$ for exp in $(seq -w 0 11)
  do
    ./render %\
      0.306095924635699074287З \
      0.0237427672737983365906 \
      2e-${exp} 512 ${exp} &&
    ./colour ${exp} > ${exp}.ppm
  done
$
```

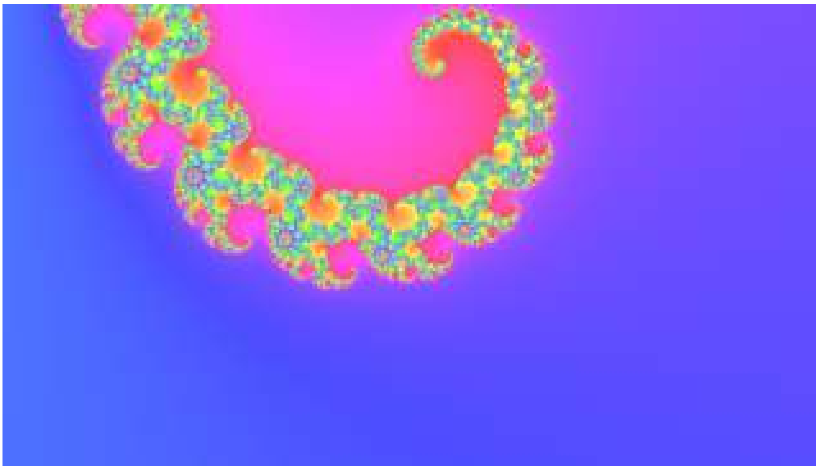


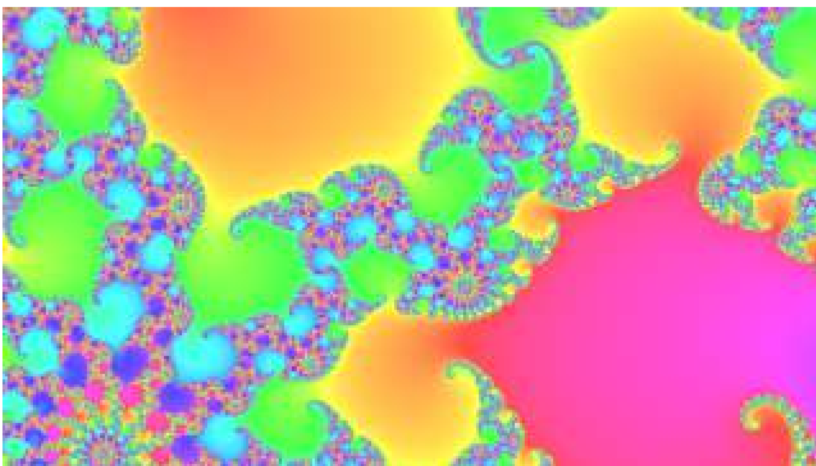


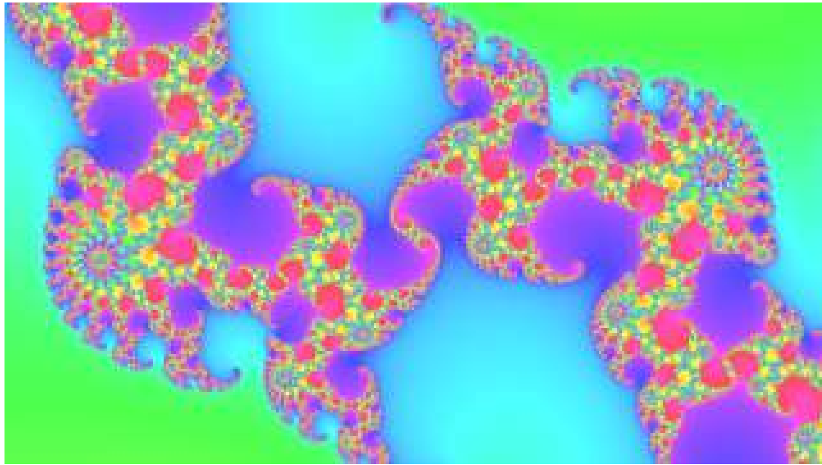


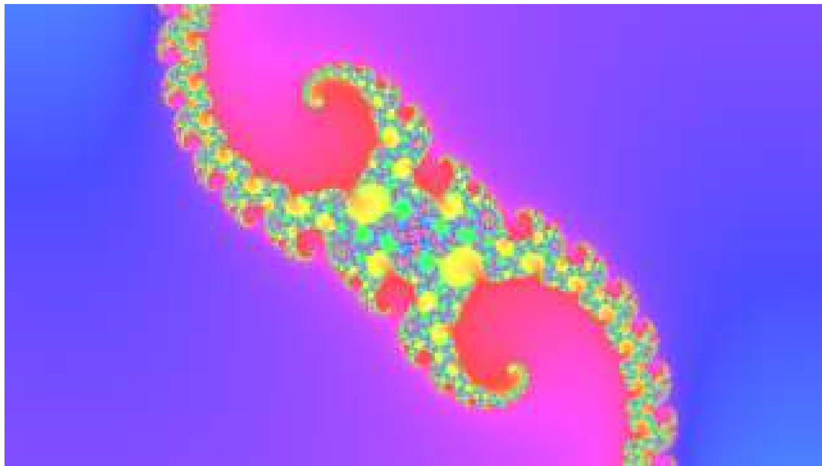


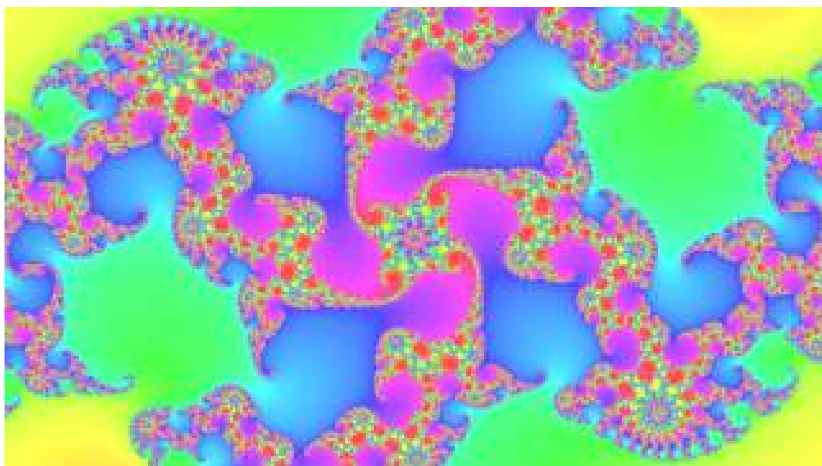


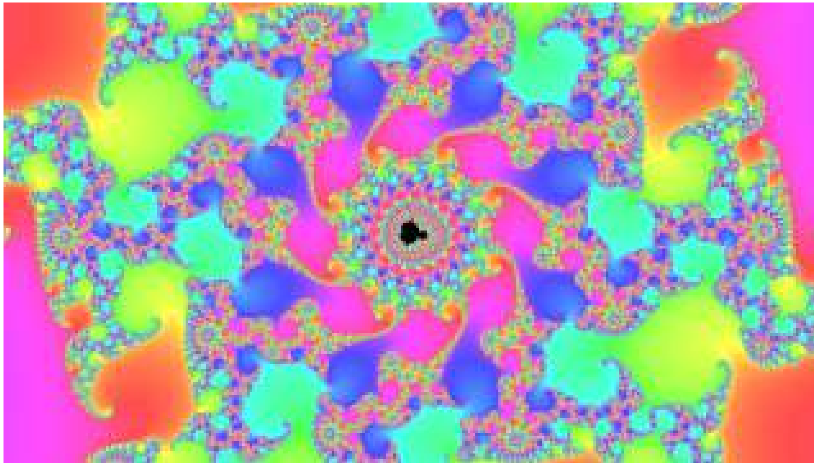


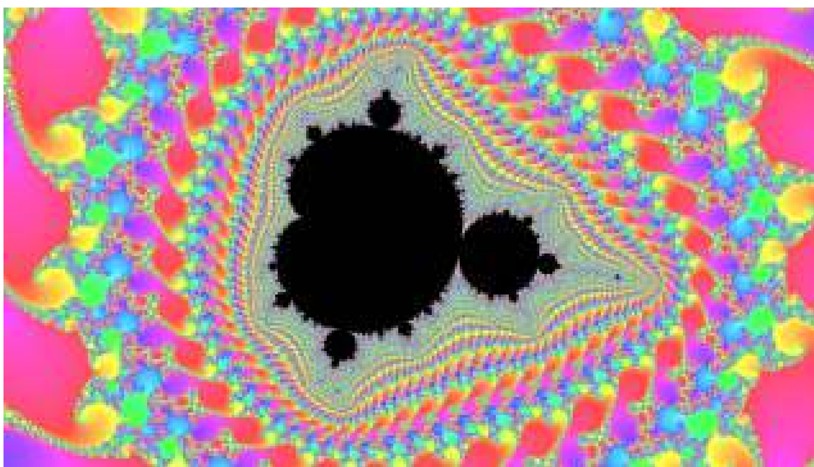








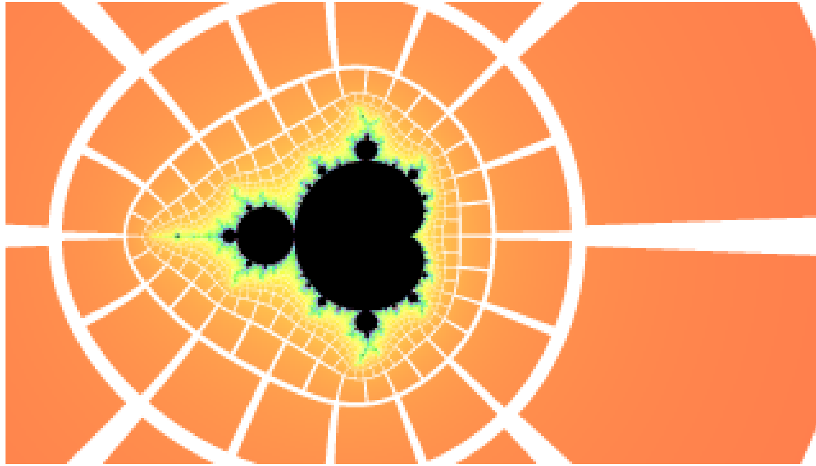




```
diff --git a/colour.c b/colour.c
index 9d3acdf..ce5db40 100644
--- a/colour.c
+++ b/colour.c
@@ -33,5 +33,9 @@ void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {
 void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_z_arg) {
-  float hue = final_n / 64.0f;
-  float sat = final_z_abs;
-  float val = final_z_arg;
+  float hue = (final_n - log2f(final_z_abs + 1.0f)) / 64.0f;
+  float sat = 0.7f;
+  float val = 1.0f;
+  if (final_n == 0) {
+    sat = 0.0f;
+    val = 0.0f;
+  }
   float red, grn, blu;
```

1.29 Generating a grid.

With smooth colouring, we're no longer displaying information about the escape time bands or final angle cells. In each cell, the angle increases from 0 to 1, and in each band, the final radius increases likewise. So within each cell we have local coordinates [0,1]x[0,1]. We can use these coordinates, in much the same way as are using the device coordinates for our images. The cells are quite small, so we'll use a simple image: we keep the colour saturated in the central region of each cell, and desaturate it to white near the edges of each cell. This gives us a clear grid, showing how the escape time bands get closer together towards the boundary of the Mandelbrot set, and also how the angle cells double-up within each successive band.

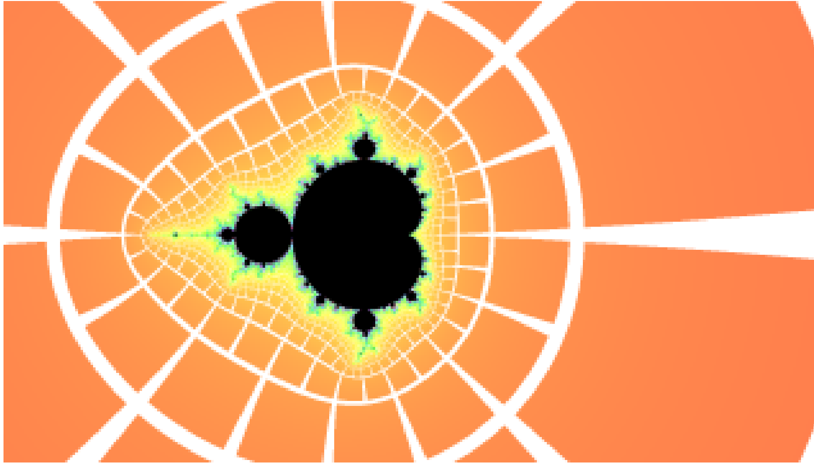


```
diff --git a/colour.c b/colour.c
index ce5db40..362c150 100644
--- a/colour.c
+++ b/colour.c
@@ -33,4 +33,8 @@ void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {
 void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_z_arg) {
-  float hue = (final_n - log2f(final_z_abs + 1.0f)) / 64.0f;
-  float sat = 0.7f;
+  float continuous_escape_time = final_n - log2f(final_z_abs + 1.0f);
+  int grid =
+    0.05 < final_z_abs && final_z_abs < 0.95 &&
+    0.05 < final_z_arg && final_z_arg < 0.95;
+  float hue = continuous_escape_time / 64.0f;
+  float sat = grid * 0.7f;
   float val = 1.0f;
```

1.30 Improving the grid.

The grid has noticeable step changes in the width of the rays heading towards the Mandelbrot set, because each successive escape time band has double the number of cells, so at the boundary between bands there is a discontinuity where the width halves. We can compensate for this by making the rays wider (in local coordinate terms) further out and thinner further in. The inside

end should be twice as narrow as the outside end, to match up with the halving of the width when crossing band boundaries. We keep the middle the same width as before by ensuring the width factor is 1 there, which means the power should be 0 at the fractional radius 0.5. Now the rays of the grid seem to continue along a smooth path towards the Mandelbrot set.



```
diff --git a/colour.c b/colour.c
index 362c150..01327b3 100644
--- a/colour.c
+++ b/colour.c
@@ -34,5 +34,7 @@ void colour(int *r, int *g, int *b, int final_n,
    float final_z_abs, float final_
   float continuous_escape_time = final_n - log2f(final_z_abs +
     1.0f);
+  float k = powf(0.5f, 0.5f - final_z_abs);
+  float grid_weight = 0.05f;
   int grid =
-    0.05 < final_z_abs && final_z_abs < 0.95 &&
-    0.05 < final_z_arg && final_z_arg < 0.95;
+    grid_weight     < final_z_abs && final_z_abs < 1.0f -
    grid_weight &&
+    grid_weight * k < final_z_arg && final_z_arg < 1.0f -
    grid_weight * k;
   float hue = continuous_escape_time / 64.0f;
```

1.31 Distance estimation.

Our images look noisy and grainy near the boundary of the Mandelbrot set. The escape time bands get closer and closer, while the pixel spacing is fixed. The pixel grid samples isolated points of a mathematically abstract image defined on the continuous plane. The Nyquist-Shannon sampling theorem shows that sampling isolated points from a continuum is a valid approximation only so long as the values don't change too quickly between the points. Aliasing occurs when the values do change too quickly compared to the sampling rate, with the grainy noisy visual effects as we have seen. Because the escape time bands increase in number without bound as we approach the boundary of the Mandelbrot set, no sampling rate can be high enough.

We can measure the rate of change using differential calculus. If we have a function $f(z)$, we can measure how the output of f changes when we change its input. The ratio $(f(z+h)-f(z))/h$

with small h tells us if f expands or contracts the region around z. This is made mathematically precise by taking the limit of this ratio as h tends to 0. For example, the differential of z^2 with respect to z can be calculated like this:

$$((z+h)^2 - z^2)/h = (z^2 + 2hz + h^2 - z^2)/h = 2z + h -> 2z$$

We iterate a function of two variables $F(z,c) = z^2 + c$, so there are two possible derivatives: one with respect to z, and one with respect to c. We want to differentiate with respect to c, to see how F expands or contract near each point in our image. Our iteration is defined by

$$F^{n+1} = F(F^n(z,c),c) = F^n(z,c)^2 + c$$

We can use mathematical induction: suppose we know the derivative dc_n for F^n. By the chain rule, the derivative of a composition of functions is the product of the derivatives at each step. We're composing F^n with squaring, and we know the derivatives for both, moreover the derivative of c with respect to c is 1, so we end up with

$$dc_{n+1} = 2F^n(z,c)dc_n + 1$$

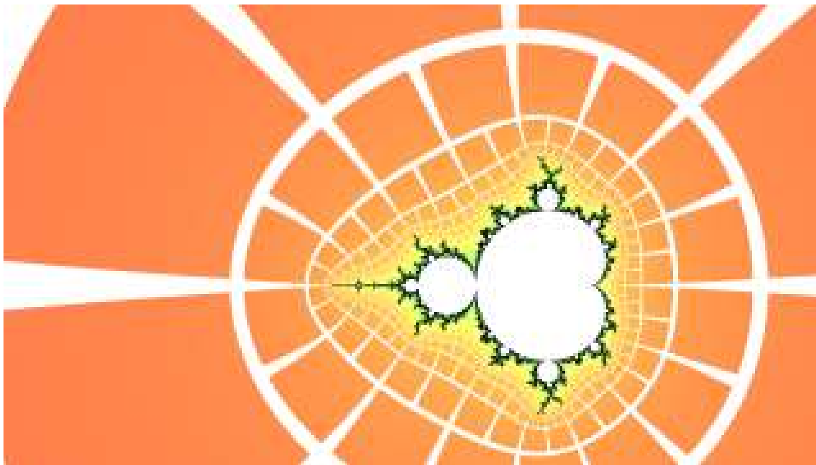
$F^n(z,c)$ is just the iterate z_n, so we can calculate the derivative as we go along. We now just need a base case for our induction, and we start with $z_0 = 0$ which doesn't involve c at all, so $dc_0 = 0$ too. We store the final dc value in our augmented pixel struct.

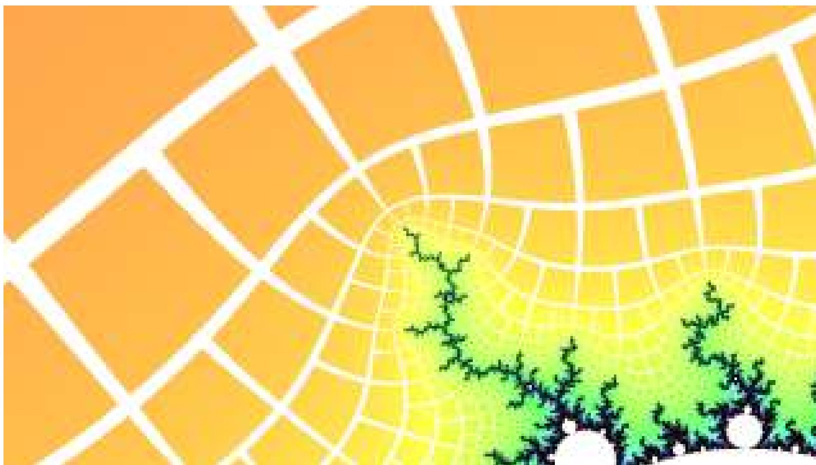
When it comes to saving the image, we have the same problem as before: how to scale the value. We know the escape time bands get closer together, and we know this causes problems related to the pixel spacing. So, multiplying the derivative by the pixel spacing is a first step (the pixel spacing is in fact the derivative with respect to our image device coordinates). We know the problems occur when the bands are closer, meaning the variation is more rapid, so we divide something by this product. What we divide by will remain somewhat magical for now, involving the final z iterate, but our final result is a distance estimate for the Mandelbrot set scaled to image device coordinates.

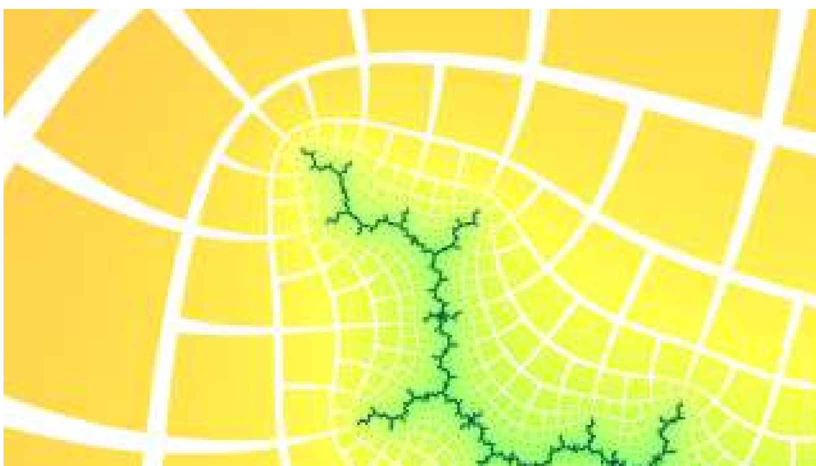
This still is far from the [0,1] output range we need for PPM, but we can choose how many bits of precision we want to keep for small values (12 in our case), which gives us a fixed-point number (essentially an integer type with a scaling factor that remains constant). We clamp the value to prevent wrapping around from putchar() truncation which would now give us meaningless values for overflow.

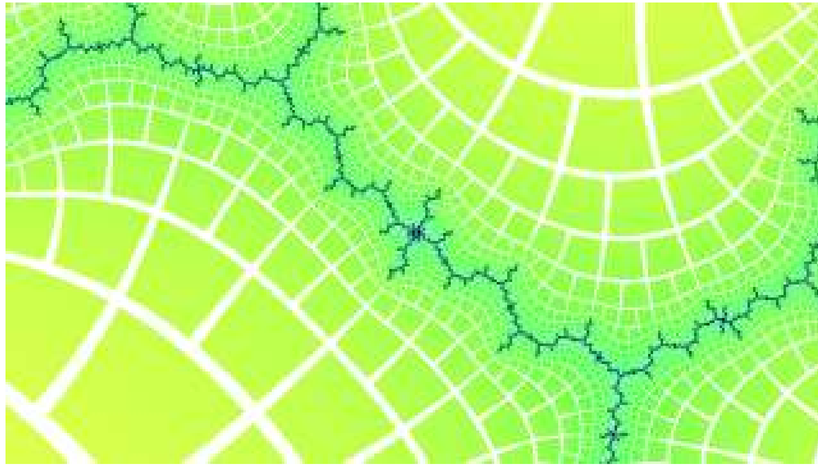
The distance estimate has the useful property (proved in the Koebe 1/4 theorem) that it tells us roughly how far our point is from the boundary of the Mandelbrot set. We can use it to mask the grainy noise from aliasing, with the pleasing side-effect of highlighting the intricate shape of the boundary of the Mandelbrot set with all its filaments and shrunken copies. To contrast with the dark boundary, we fill the interior with white.

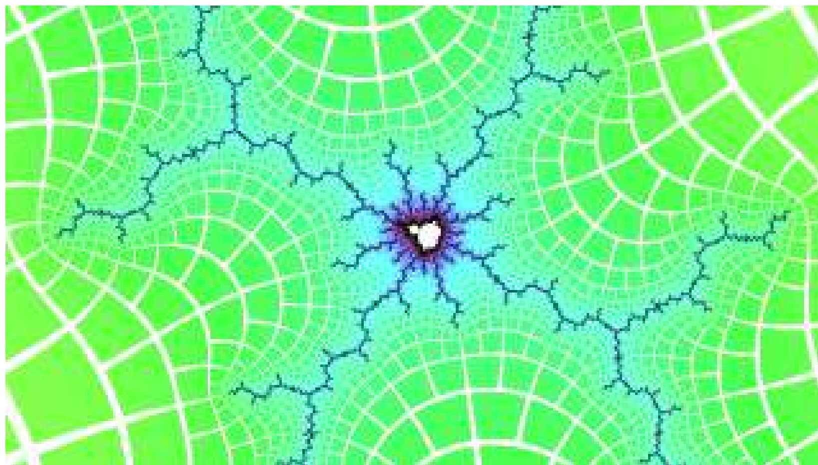
```
$ for exp in $(seq -w 0 16)
  do
    ./render \
      -1.2864956446267794623797546990264761111 \
       0.4339437613619272464268001250115015311 \
      2e-${exp} 512 ${exp} &&
    ./colour ${exp} > ${exp}.ppm
  done
```

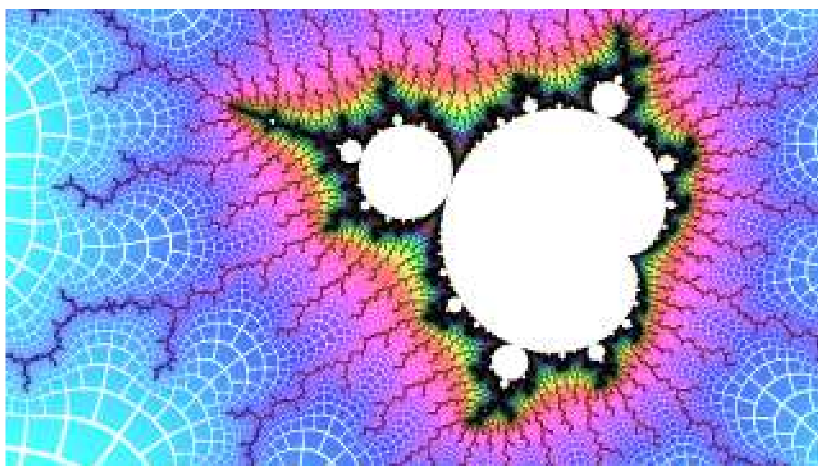


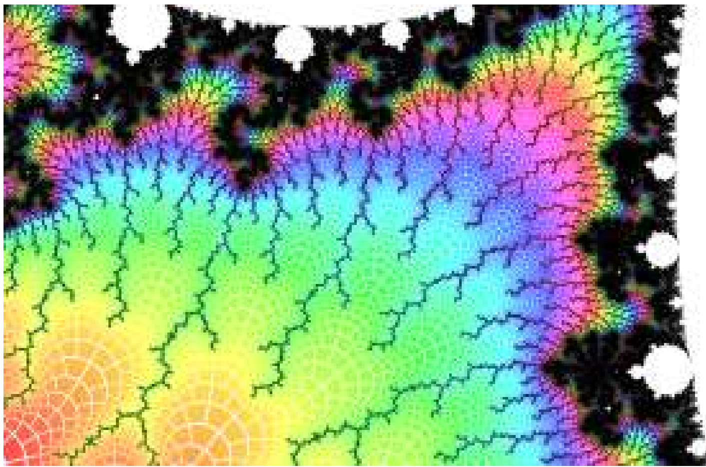


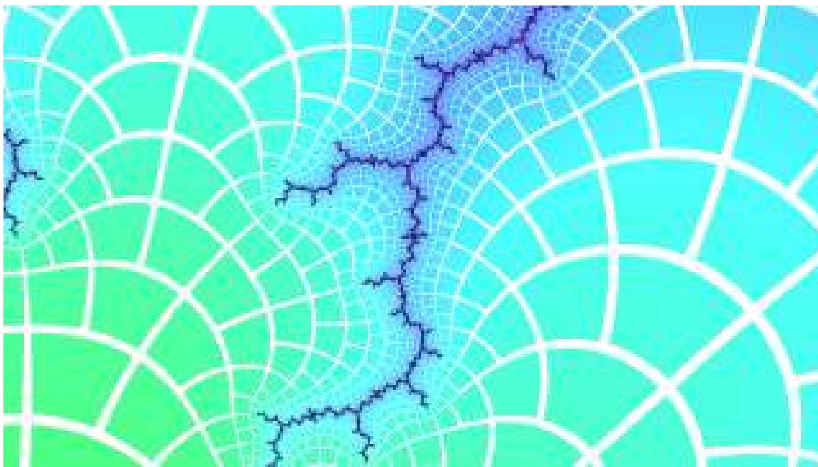


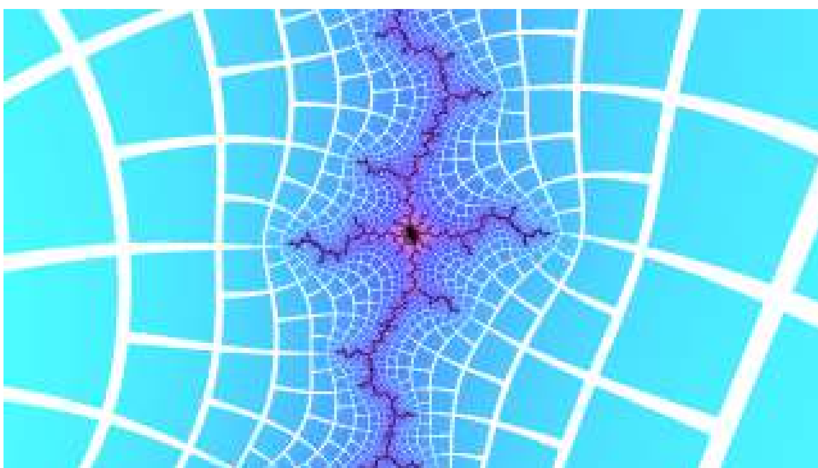


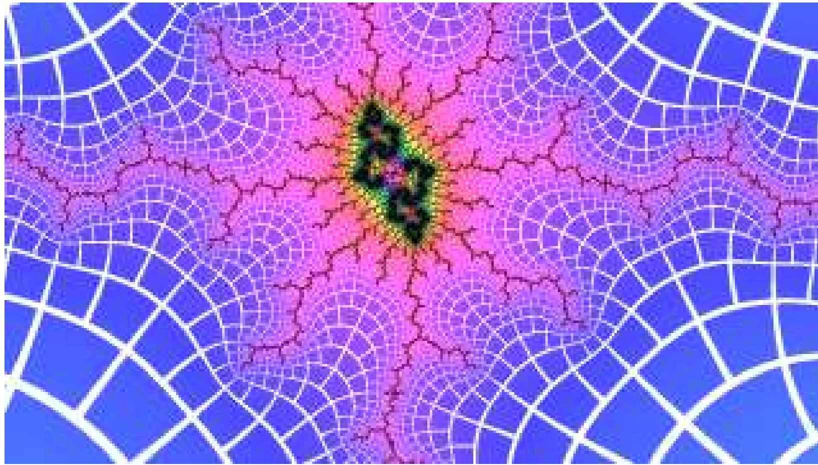


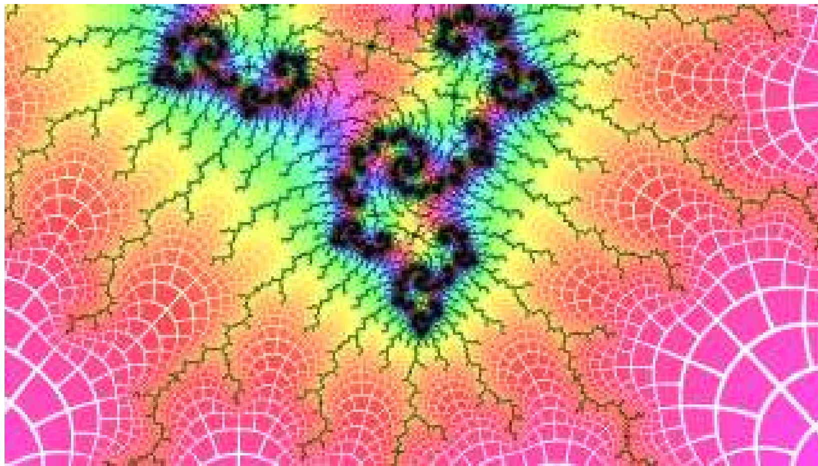


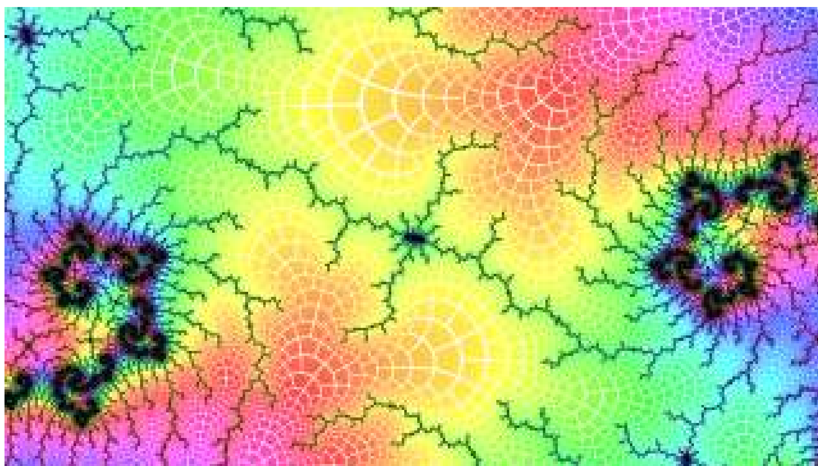


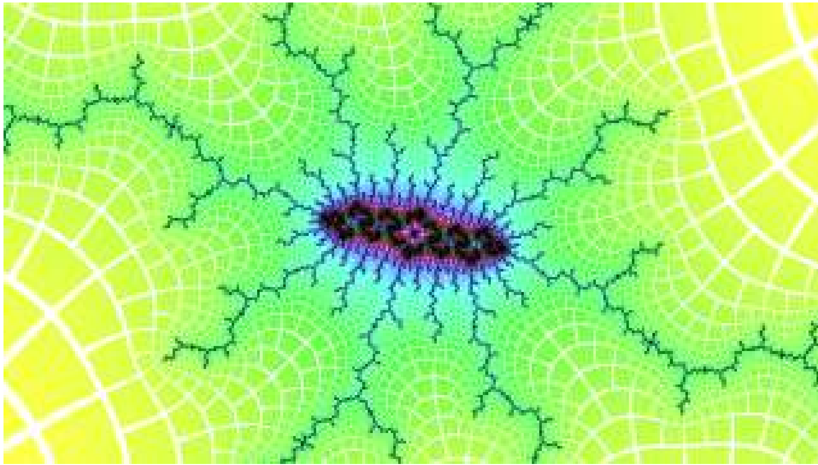


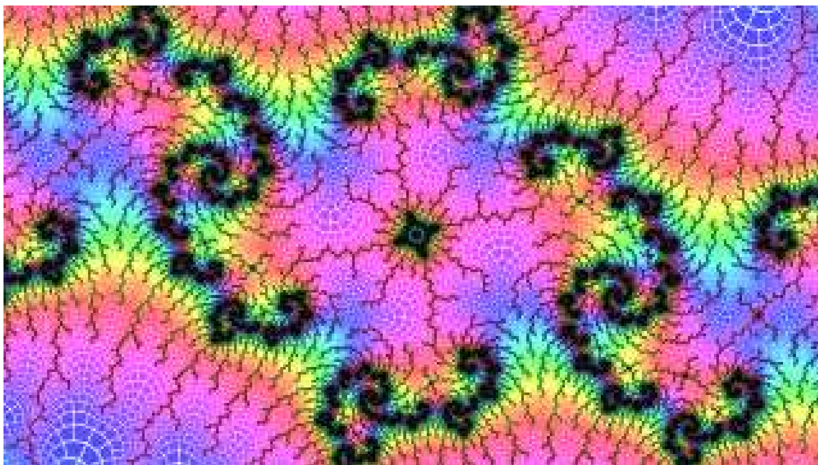


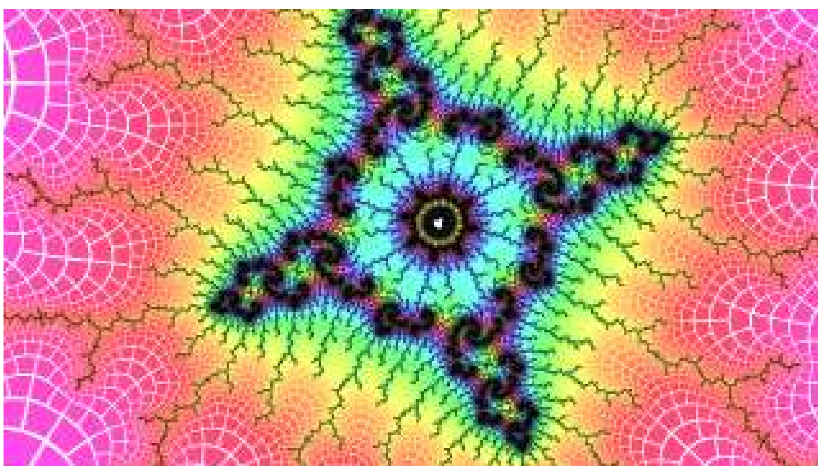


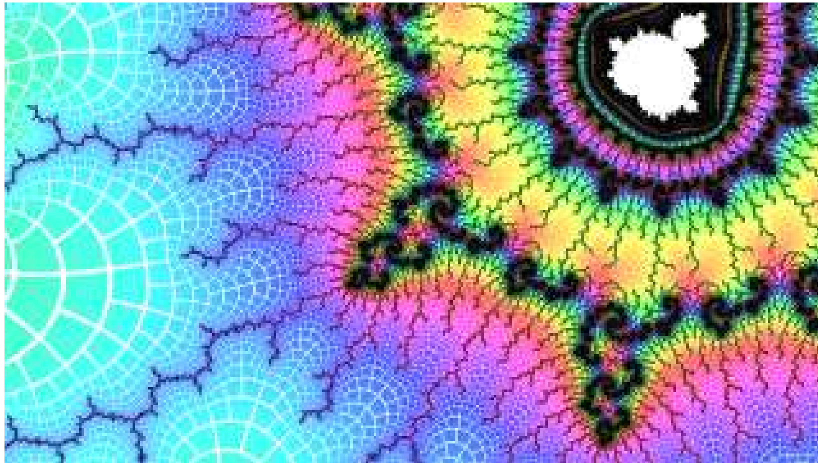


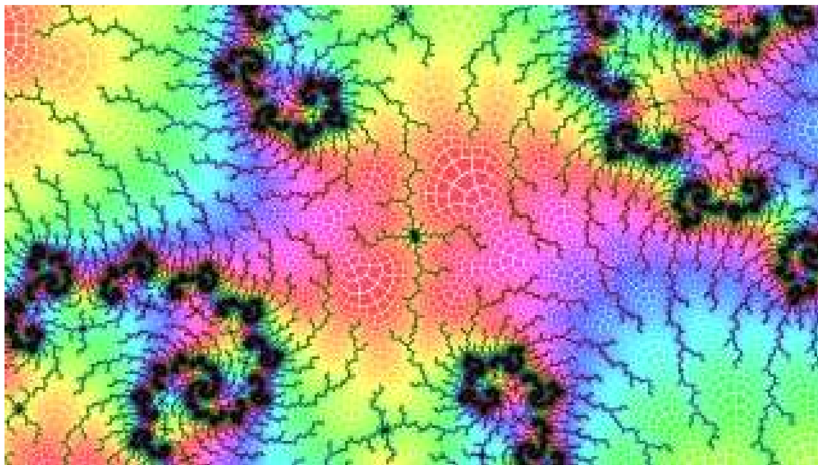












```
diff --git a/colour.c b/colour.c
index 01327b3..28e0e27 100644
--- a/colour.c
+++ b/colour.c
@@ -32,3 +32,3 @@ void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {

-void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_z_arg)
+void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_z_arg, float de) {
   float continuous_escape_time = final_n - log2f(final_z_abs + 1.0f);
@@ -41,6 +41,6 @@ void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_
   float sat = grid * 0.7f;
-  float val = 1.0f;
+  float val = tanh(fmin(fmax((1 << 11) * de, 0.0), 4.0));
   if (final_n == 0) {
     sat = 0.0f;
-    val = 0.0f;
+    val = 1.0f;
```

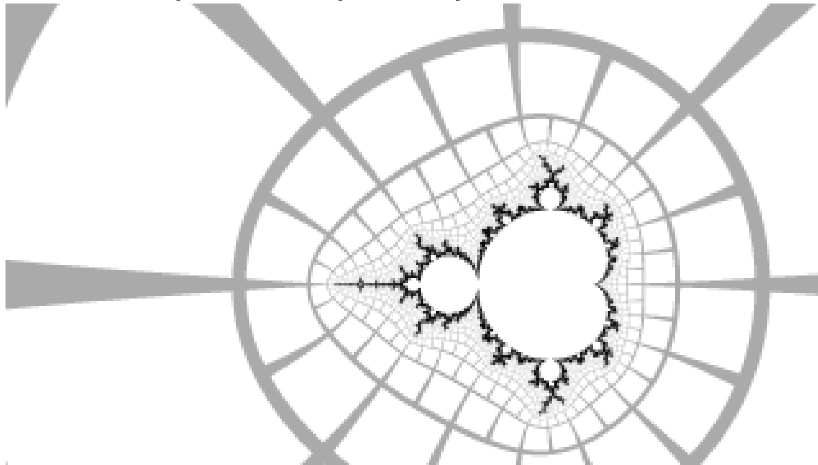
```
   }
@@ -90,6 +90,18 @@ int main(int argc, char **argv) {

+  char *fname_d = calloc(length, 1);
+  snprintf(fname_d, length, "%s_de.ppm", stem);
+  FILE *f_d = fopen(fname_d, "rb");
+  if (! f_d) { return 1; }
+  free(fname_d);
+  int width_d = 0;
+  int height_d = 0;
+  if (2 != fscanf(f_d, "P6 %d %d 255", &width_d, &height_d)) { return 1; }
+  if (fgetc(f_d) != '\n') { return 1; }
+
   if (width_n != width_r) { return 1; }
   if (width_n != width_a) { return 1; }
+  if (width_n != width_d) { return 1; }
   if (height_n != height_r) { return 1; }
   if (height_n != height_a) { return 1; }
+  if (height_n != height_d) { return 1; }
   int width = width_n;
@@ -109,2 +121,5 @@ int main(int argc, char **argv) {
       int b_a = fgetc(f_a);
+      int r_d = fgetc(f_d);
+      int g_d = fgetc(f_d);
+      int b_d = fgetc(f_d);
       int   n =  (r_n << 16) | (g_n << 8) | b_n;
@@ -112,4 +127,5 @@ int main(int argc, char **argv) {
       float a = ((r_a << 16) | (g_a << 8) | b_a) / (float) (1 << 24);
+      float d = ((r_d << 16) | (g_d << 8) | b_d) / (float) (1 << 24);
       int red, grn, blu;
-      colour(&red, &grn, &blu, n, r, a);
+      colour(&red, &grn, &blu, n, r, a, d);
       putchar(red);
diff --git a/mandelbrot.h b/mandelbrot.h
index a8be2fa..1b4ad9f 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -12,2 +12,3 @@ struct pixel {
   complex float final_z;
+  complex float final_dc;
 };
@@ -53,3 +54,3 @@ void image_free(struct image *img) {

-int image_save(struct image *img, char *filestem, float escape_radius) {
+int image_save(struct image *img, char *filestem, float escape_radius, double pixel_spacing) {
   int retval = 0;
@@ -115,2 +116,23 @@ int image_save(struct image *img, char *filestem, float escape_radius) {

```

```
+    snprintf(filename, length, "%s_de.ppm", filestem);
+    f = fopen(filename, "wb");
+    if (f) {
+      fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
+      for (int j = 0; j < img->height; ++j) {
+        for (int i = 0; i < img->width; ++i) {
+          complex double z  = img->pixels[j * img->width + i].final_z;
+          complex double dc = img->pixels[j * img->width + i].final_dc;
+          float de = 2 * cabs(z) * log(cabs(z)) / (cabs(dc) * pixel_spacing);
+          int n = ((1 << 24) - 1) * fminf(fmaxf(de / (1 << 12), 0.0f), 1.0f);
+          fputc(n >> 16, f);
+          fputc(n >>  8, f);
+          fputc(n      , f);
+        }
+      }
+      fclose(f);
+    } else {
+      retval = 1;
+    }
+
+
     free(filename);
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index 594fc32..7d83325 100644
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -16,4 +16,7 @@ void FNAME(calculate)(struct pixel *out, complex FTYPE c, int maximum_iterations
   out->final_z = 0;
+  out->final_dc = 0;
   complex FTYPE z = 0;
+  complex FTYPE dc = 0;
   for (int n = 1; n < maximum_iterations; ++n) {
+    dc = 2 * z * dc + 1;
     z = z * z + c;
@@ -22,2 +25,3 @@ void FNAME(calculate)(struct pixel *out, complex FTYPE c, int maximum_iterations
       out->final_z = z;
+      out->final_dc = dc;
       break;
diff --git a/render.c b/render.c
index 039253e..1584e2a 100644
--- a/render.c
+++ b/render.c
@@ -52,3 +52,3 @@ int main(int argc, char **argv) {
     }
-    int err = image_save(img, stem, escape_radius);
+    int err = image_save(img, stem, escape_radius, pixel_spacing);
```

```
    image_free(img);
```

1.32 Removing distracting colour.

The bold colours are now a distraction: they don't show us anything we can't see from the grid or the distance estimation. Desaturating the image and making the exterior grid a light grey gives us a clean style that is easy on the eyes.



```
diff --git a/colour.c b/colour.c
index 28e0e27..3ac0445 100644
--- a/colour.c
+++ b/colour.c
@@ -40,6 +40,5 @@ void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_
   float hue = continuous_escape_time / 64.0f;
-  float sat = grid * 0.7f;
-  float val = tanh(fmin(fmax((1 << 11) * de, 0.0), 4.0));
+  float sat = 0.0f;
+  float val = fmin(tanh(fmin(fmax((1 << 11) * de, 0.0), 4.0)), 0.8 + 0.2 * grid);
   if (final_n == 0) {
-    sat = 0.0f;
     val = 1.0f;
```

1.33 Refactoring image stream reading.

Our colour.c program has a lot of repetition. We factor it out into smaller functions, with a struct stream to hold the necessary data for each input image. We ensure that stream_new() returns a non-null pointer only for valid images. Reading the channel bytes and packing them into an int is performed in stream_read(). We write a wrapper to form an appropriate filename from a stem and a suffix. Now the main() is a lot shorter, with more robust error handling. We proceed to the image processing loop only if all the input files were opened and are compatible, and signal a successful return value. We close all the streams that were opened successfully before exit.

```
diff --git a/colour.c b/colour.c
index 3ac0445..c837116 100644
--- a/colour.c
+++ b/colour.c
@@ -52,3 +52,58 @@ void colour(int *r, int *g, int *b, int final_n
    , float final_z_abs, float final_

+struct stream {
+  FILE *file;
+  int width;
+  int height;
+};
+
+struct stream *stream_new(char *name) {
+  struct stream *s = calloc(1, sizeof(*s));
+  if (! s) {
+    return 0;
+  }
+  s->file = fopen(name, "rb");
+  if (! s->file) {
+    free(s);
+    return 0;
+  }
+  if (2 != fscanf(s->file, "P6 %d %d 255", &s->width, &s->height)
    ) {
+    fclose(s->file);
+    free(s);
+    return 0;
+  }
+  if (fgetc(s->file) != '\n') {
+    fclose(s->file);
+    free(s);
+    return 0;
+  }
+  return s;
+}
+
+void stream_free(struct stream *s) {
+  fclose(s->file);
+  free(s);
+}
+
+int stream_compatible(struct stream *s, struct stream *t) {
+  return s->width == t->width && s->height == t->height;
+}
+
+int stream_read(struct stream *s) {
+  int r = fgetc(s->file);
+  int g = fgetc(s->file);
+  int b = fgetc(s->file);
+  return (r << 16) | (g << 8) | b;
+}
+
```

```
+struct stream *stream_new_for_stem(char *stem, char *suffix) {
+  int length = strlen(stem) + strlen(suffix) + 100;
+  char *name = calloc(length, 1);
+  snprintf(name, length, "%s_%s.ppm", stem, suffix);
+  struct stream *s = stream_new(name);
+  free(name);
+  return s;
+}
+
 int main(int argc, char **argv) {
+  int retval = 1;
   char *stem = "out";
@@ -57,84 +112,38 @@ int main(int argc, char **argv) {
   }
-  int length = strlen(stem) + 100;

-  char *fname_n = calloc(length, 1);
-  snprintf(fname_n, length, "%s_n.ppm", stem);
-  FILE *f_n = fopen(fname_n, "rb");
-  if (! f_n) { return 1; }
-  free(fname_n);
-  int width_n = 0;
-  int height_n = 0;
-  if (2 != fscanf(f_n, "P6 %d %d 255", &width_n, &height_n)) { return 1; }
-  if (fgetc(f_n) != '\n') { return 1; }
+  struct stream *s_n = stream_new_for_stem(stem, "n");
+  struct stream *s_r = stream_new_for_stem(stem, "z_abs");
+  struct stream *s_a = stream_new_for_stem(stem, "z_arg");
+  struct stream *s_d = stream_new_for_stem(stem, "de");
+  if (s_n && s_r && s_a && s_d) {
+    if (stream_compatible(s_n, s_r) &&
+        stream_compatible(s_n, s_a) &&
+        stream_compatible(s_n, s_d)) {

-  char *fname_r = calloc(length, 1);
-  snprintf(fname_r, length, "%s_z_abs.ppm", stem);
-  FILE *f_r = fopen(fname_r, "rb");
-  if (! f_r) { return 1; }
-  free(fname_r);
-  int width_r = 0;
-  int height_r = 0;
-  if (2 != fscanf(f_r, "P6 %d %d 255", &width_r, &height_r)) { return 1; }
-  if (fgetc(f_r) != '\n') { return 1; }
+      int width = s_n->width;
+      int height = s_n->height;
+      printf("P6\n%d %d\n255\n", width, height);
+      for (int j = 0; j < height; ++j) {
+        for (int i = 0; i < width; ++i) {
+          int   n = stream_read(s_n);
+          float r = stream_read(s_r) / (float) (1 << 24);
+          float a = stream_read(s_a) / (float) (1 << 24);
```

```
    +            float d = stream_read(s_d) / (float) (1 << 24);
    +            int red, grn, blu;
105 +            colour(&red, &grn, &blu, n, r, a, d);
    +            putchar(red);
    +            putchar(grn);
    +            putchar(blu);
    +        }
110 +      }
    +      retval = 0;

    -   char *fname_a = calloc(length, 1);
    -   snprintf(fname_a, length, "%s_z_arg.ppm", stem);
115 -   FILE *f_a = fopen(fname_a, "rb");
    -   if (! f_a) { return 1; }
    -   free(fname_a);
    -   int width_a = 0;
    -   int height_a = 0;
120 -   ↙
      ↳ if (2 != fscanf(f_a, "P6 %d %d 255", &width_a, &height_a)) { return 1; }
    -   if (fgetc(f_a) != '\n') { return 1; }
    -
    -   char *fname_d = calloc(length, 1);
    -   snprintf(fname_d, length, "%s_de.ppm", stem);
125 -   FILE *f_d = fopen(fname_d, "rb");
    -   if (! f_d) { return 1; }
    -   free(fname_d);
    -   int width_d = 0;
    -   int height_d = 0;
130 -   ↙
      ↳ if (2 != fscanf(f_d, "P6 %d %d 255", &width_d, &height_d)) { return 1; }
    -   if (fgetc(f_d) != '\n') { return 1; }
    -
    -   if (width_n != width_r) { return 1; }
    -   if (width_n != width_a) { return 1; }
135 -   if (width_n != width_d) { return 1; }
    -   if (height_n != height_r) { return 1; }
    -   if (height_n != height_a) { return 1; }
    -   if (height_n != height_d) { return 1; }
    -   int width = width_n;
140 -   int height = height_n;
    -
    -   printf("P6\n%d %d\n255\n", width, height);
    -   for (int j = 0; j < height; ++j) {
    -     for (int i = 0; i < width; ++i) {
145 -       int r_n = fgetc(f_n);
    -       int g_n = fgetc(f_n);
    -       int b_n = fgetc(f_n);
    -       int r_r = fgetc(f_r);
    -       int g_r = fgetc(f_r);
150 -       int b_r = fgetc(f_r);
    -       int r_a = fgetc(f_a);
    -       int g_a = fgetc(f_a);
    -       int b_a = fgetc(f_a);
```

```
-        int r_d = fgetc(f_d);
-        int g_d = fgetc(f_d);
-        int b_d = fgetc(f_d);
-        int    n =  (r_n << 16) | (g_n << 8) | b_n;
-        float r = ((r_r << 16) | (g_r << 8) | b_r) / (float) (1 << 24);
-        float a = ((r_a << 16) | (g_a << 8) | b_a) / (float) (1 << 24);
-        float d = ((r_d << 16) | (g_d << 8) | b_d) / (float) (1 << 24);
-        int red, grn, blu;
-        colour(&red, &grn, &blu, n, r, a, d);
-        putchar(red);
-        putchar(grn);
-        putchar(blu);
       }
     }
+  if (s_n) { stream_free(s_n); }
+  if (s_r) { stream_free(s_r); }
+  if (s_a) { stream_free(s_a); }
+  if (s_d) { stream_free(s_d); }

-  fclose(f_n);
-  fclose(f_a);
-
-  return 0;
+  return retval;
 }
```

1.34 Annotating images with Cairo.

We'd like to add labels and other annotations to our Mandelbrot set images. Cairo is a library for vector graphics, with support for rasterized image surfaces. We factor out the stream reading code to a separate file, and use it in both our ./colour program and our new ./annotate program which so far just reads the coloured image and saves it to compressed PNG (Portable Network Graphics) format.

```
diff --git a/Makefile b/Makefile
index 7cd9766..248d755 100644
--- a/Makefile
+++ b/Makefile
@@ -1,4 +1,7 @@
-all: render colour
+all: render colour annotate

-colour: colour.c
+annotate: annotate.c stream.h
+        gcc -std=c99 -Wall -Wextra -pedantic -O3 -o annotate annotate.c -lm -lcairo
+
+colour: colour.c stream.h
```

```
        gcc -std=c99 -Wall -Wextra -pedantic -O3 -o colour colour.
            c -lm
15 diff --git a/annotate.c b/annotate.c
   new file mode 100644
   index 0000000..71f9aba
   --- /dev/null
   +++ b/annotate.c
20 @@ -0,0 +1,35 @@
   +#include <stdint.h>
   +
   +#include <cairo/cairo.h>
   +
25 +#include "stream.h"
   +
   +int main(int argc, char **argv) {
   +  int retval = 1;
   +  char *in = "in.ppm";
30 +  char *out = "out.png";
   +  if (argc > 1) { in  = argv[1]; }
   +  if (argc > 2) { out = argv[2]; }
   +  struct stream *s = stream_new(in);
   +  if (s) {
35 +    cairo_surface_t *c = cairo_image_surface_create(
       CAIRO_FORMAT_ARGB32, s->width, s->height);
   +    if (c) {
   +      uint32_t *data = (uint32_t *) cairo_image_surface_get_data(
       c);
   +      int channels = 4;
   +      int stride = cairo_image_surface_get_stride(c) / channels;
40 +      for (int j = 0; j < s->height; ++j) {
   +        for (int i = 0; i < s->width; ++i) {
   +          int p = stream_read(s);
   +          int k = j * stride + i;
   +          data[k] = (0xFF << 24) | p;
45 +        }
   +      }
   +      cairo_surface_mark_dirty(c);
   +      cairo_surface_write_to_png(c, out);
   +      retval = 0;
50 +      cairo_surface_destroy(c);
   +    }
   +    stream_free(s);
   +  }
   +  return retval;
55 +}
   diff --git a/colour.c b/colour.c
   index c837116..4d144a2 100644
   --- a/colour.c
   +++ b/colour.c
60 @@ -5,2 +5,4 @@

   +#include "stream.h"
   +
```

```
 int channel(float c) {
@@ -52,36 +54,2 @@ void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_

-struct stream {
-    FILE *file;
-    int width;
-    int height;
-};
-
-struct stream *stream_new(char *name) {
-    struct stream *s = calloc(1, sizeof(*s));
-    if (! s) {
-        return 0;
-    }
-    s->file = fopen(name, "rb");
-    if (! s->file) {
-        free(s);
-        return 0;
-    }
-    if (2 != fscanf(s->file, "P6 %d %d 255", &s->width, &s->height)) {
-        fclose(s->file);
-        free(s);
-        return 0;
-    }
-    if (fgetc(s->file) != '\n') {
-        fclose(s->file);
-        free(s);
-        return 0;
-    }
-    return s;
-}
-
-void stream_free(struct stream *s) {
-    fclose(s->file);
-    free(s);
-}
-
 int stream_compatible(struct stream *s, struct stream *t) {
@@ -90,9 +58,2 @@ int stream_compatible(struct stream *s, struct stream *t) {

-int stream_read(struct stream *s) {
-    int r = fgetc(s->file);
-    int g = fgetc(s->file);
-    int b = fgetc(s->file);
-    return (r << 16) | (g << 8) | b;
-}
-
 struct stream *stream_new_for_stem(char *stem, char *suffix) {
diff --git a/stream.h b/stream.h
new file mode 100644
```

```
    index 0000000..f0b886b
115 --- /dev/null
    +++ b/stream.h
    @@ -0,0 +1,48 @@
    +#ifndef STREAM_H
    +#define STREAM_H 1
120 +
    +#include <stdio.h>
    +#include <stdlib.h>
    +
    +struct stream {
125 +  FILE *file;
    +  int width;
    +  int height;
    +};
    +
130 +struct stream *stream_new(char *name) {
    +  struct stream *s = calloc(1, sizeof(*s));
    +  if (! s) {
    +    return 0;
    +  }
135 +  s->file = fopen(name, "rb");
    +  if (! s->file) {
    +    free(s);
    +    return 0;
    +  }
140 +  if (2 != fscanf(s->file, "P6 %d %d 255", &s->width, &s->height)
      ) {
    +    fclose(s->file);
    +    free(s);
    +    return 0;
    +  }
145 +  if (fgetc(s->file) != '\n') {
    +    fclose(s->file);
    +    free(s);
    +    return 0;
    +  }
150 +  return s;
    +}
    +
    +void stream_free(struct stream *s) {
    +  fclose(s->file);
155 +  free(s);
    +}
    +
    +int stream_read(struct stream *s) {
    +  int r = fgetc(s->file);
160 +  int g = fgetc(s->file);
    +  int b = fgetc(s->file);
    +  return (r << 16) | (g << 8) | b;
    +}
    +
165 +#endif
```

Chapter 2

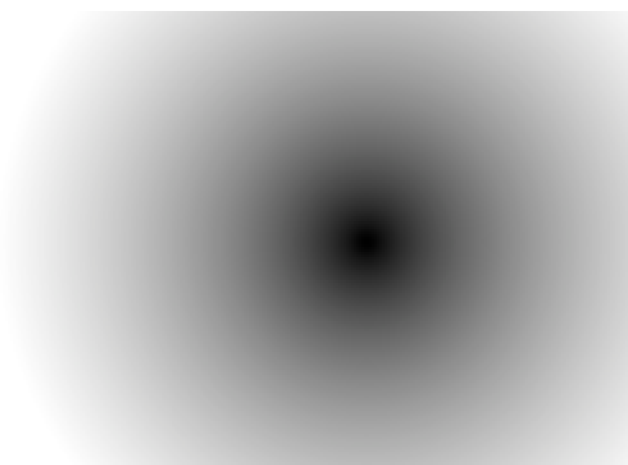
The Interior

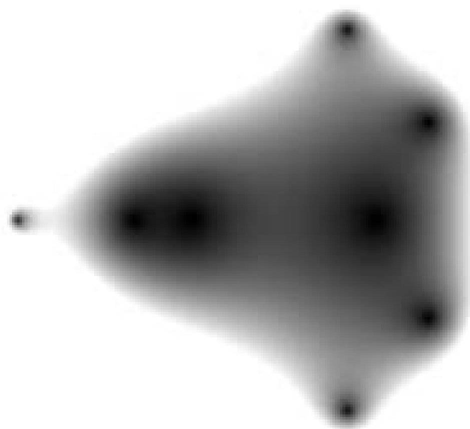
2.1 Diagnosing attractors.

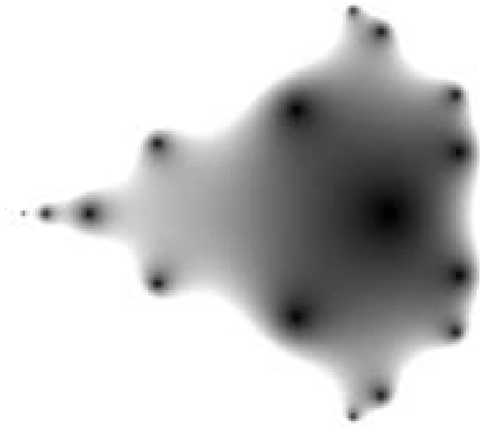
So far we have only explored the exterior, where points escape to infinity. Plotting the magnitude of the iterates provides a glimpse at the rich structure within it. After one iteration we see a fuzzy circle, where the iterates nearer zero are coloured darker. After two, another dark blob is visible to the left. At the third iterations, the left blob disappears and three new blobs appear. At the fourth iteration the blob of the second iteration reappears. At later iterations the familiar shape of the Mandelbrot set emerges.

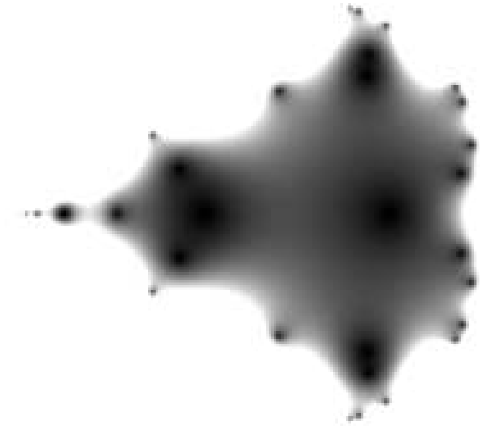
The initial blob is ever-present: if the c value is close enough to zero then so will all its iterates remain. In fact the iterates are attracted to a fixed point. The disappearing reappearing trick of the other blobs gives a hint towards periodic attractors - the iterate returns close to zero regularly.

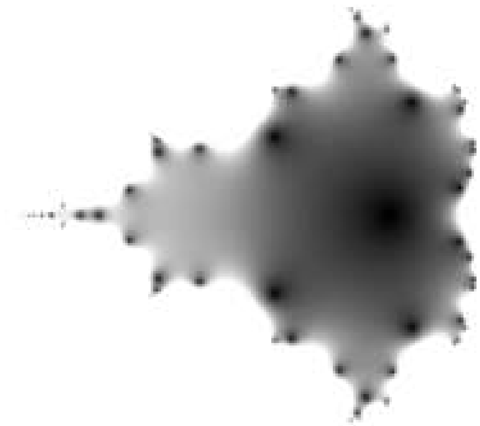
The centers of the blobs are roots of the polynomial formed by repeating a fixed number of iterations: at a nucleus of period p, $F^p(z,c) = 0$.

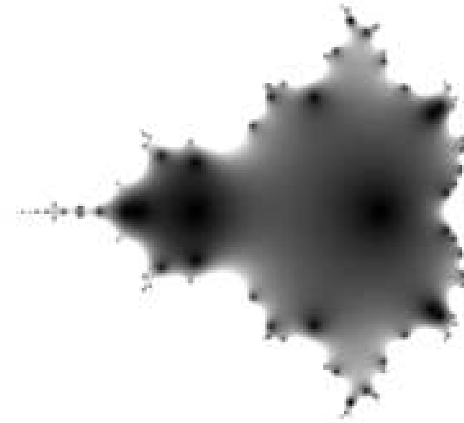


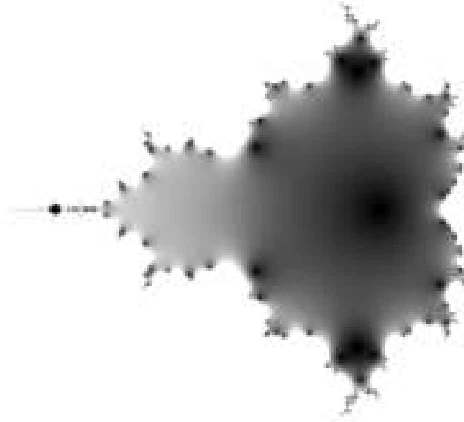


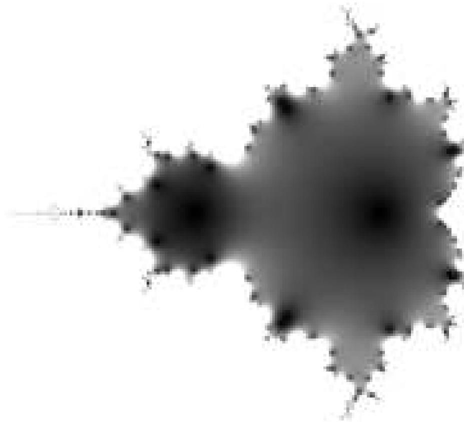


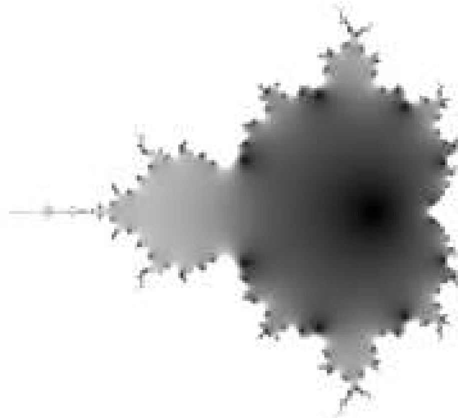


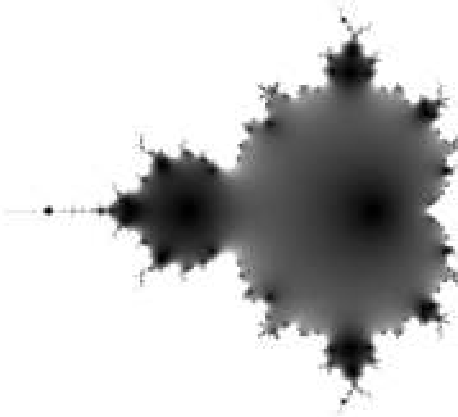










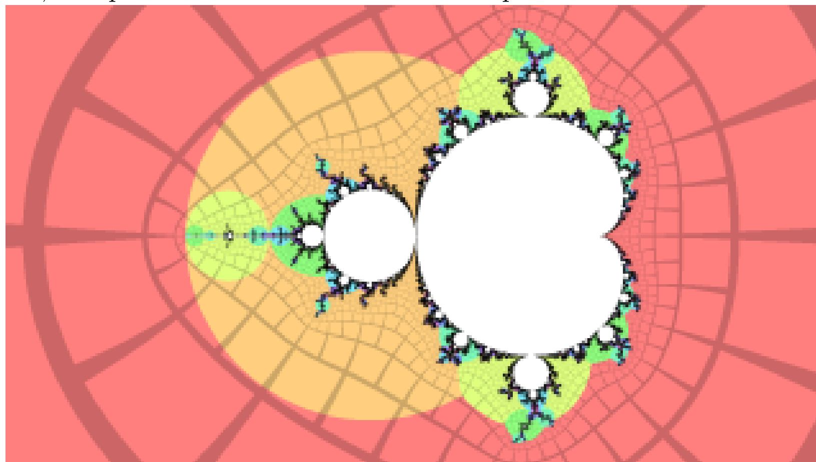


```
diff --git a/code/interior.c b/code/interior.c
new file mode 100644
index 0000000..8298141
--- /dev/null
+++ b/code/interior.c
@@ -0,0 +1,34 @@
+#include "mandelbrot.h"
+
+#define width 1280
+#define height 720
+#define maximum_iterations 12
+
+complex float z[height][width];
+
+int channel(float c) {
+  return fminf(fmaxf(roundf(255 * c), 0), 255);
+}
+
+int main() {
+  memset(z, 0, sizeof(z[0][0]) * height * width);
+  for (int n = 0; n <= maximum_iterations; ++n) {
```

```
+     fprintf(stdout, "P5\n%d %d\n255\n", width, height);
+     for (int j = 0; j < height; ++j) {
+       for (int i = 0; i < width; ++i) {
+         int k = channel(log2f(cabsf(z[j][i])/2.0 + 1.0f));
+         fputc(k, stdout);
+       }
+     }
+     if (n < maximum_iterations) {
+       #pragma omp parallel for
+       for (int j = 0; j < height; ++j) {
+         for (int i = 0; i < width; ++i) {
+           complex float c = coordinatef(i, j, width, height,
    -0.75f, 1.25f);
+           z[j][i] = z[j][i] * z[j][i] + c;
+         }
+       }
+     }
+   }
+   return 0;
+}
```

2.2 Atom domains

Points in the exterior eventually escape to infinity. This means that the minimum iterate magnitude is well defined, and we can use the iteration count to define the atom domain as the iteration count when this minimum is reached. Colouring according to atom domain reveals shadows of the blobs in the previous section - we can read off the period of the blob with red being 1, orange 2, yellow-green 3, and so on. We can also see that the period 3 bulbs have antenna hubs with 3 spokes, and period 4 bulbs have hubs with 4 spokes.



```
diff --git a/.gitignore b/.gitignore
index 357c6fd..aeb29f7 100644
--- a/.gitignore
+++ b/.gitignore
@@ -10,3 +10,6 @@ book/*.lot
 book/*.out
```

```
 book/*.pdf
 book/*.toc
+*.pgm
+*.ppm
+*.png
diff --git a/code/colour.c b/code/colour.c
index 4d144a2..46ee814 100644
--- a/code/colour.c
+++ b/code/colour.c
@@ -32,15 +32,15 @@ void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {
   *blu = b;
 }

-void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_z_arg, float de) {
+void colour(int *r, int *g, int *b, int final_n, float final_z_abs, float final_z_arg, float de, int final_p) {
   float continuous_escape_time = final_n - log2f(final_z_abs + 1.0f);
   float k = powf(0.5f, 0.5f - final_z_abs);
   float grid_weight = 0.05f;
   int grid =
     grid_weight     < final_z_abs && final_z_abs < 1.0f - grid_weight &&
     grid_weight * k < final_z_arg && final_z_arg < 1.0f - grid_weight * k;
-  float hue = continuous_escape_time / 64.0f;
-  float sat = 0.0f;
+  float hue = (final_p - 1.0f) / 9.618033988749895f;
+  float sat = final_p > 0.0f ? 0.5f : 0.0f;
   float val = fmin(tanh(fmin(fmax((1 << 11) * de, 0.0), 4.0)), 0.8 + 0.2 * grid);
   if (final_n == 0) {
     val = 1.0f;
@@ -76,10 +76,12 @@ int main(int argc, char **argv) {
   struct stream *s_r = stream_new_for_stem(stem, "z_abs");
   struct stream *s_a = stream_new_for_stem(stem, "z_arg");
   struct stream *s_d = stream_new_for_stem(stem, "de");
-  if (s_n && s_r && s_a && s_d) {
+  struct stream *s_p = stream_new_for_stem(stem, "p");
+  if (s_n && s_r && s_a && s_d && s_p) {
     if (stream_compatible(s_n, s_r) &&
         stream_compatible(s_n, s_a) &&
-        stream_compatible(s_n, s_d)) {
+        stream_compatible(s_n, s_d) &&
+        stream_compatible(s_n, s_p)) {

       int width = s_n->width;
       int height = s_n->height;
@@ -90,8 +92,9 @@ int main(int argc, char **argv) {
           float r = stream_read(s_r) / (float) (1 << 24);
           float a = stream_read(s_a) / (float) (1 << 24);
```

```
            float d = stream_read(s_d) / (float) (1 << 24);
+           int   p = stream_read(s_p);
55          int red, grn, blu;
-           colour(&red, &grn, &blu, n, r, a, d);
+           colour(&red, &grn, &blu, n, r, a, d, p);
            putchar(red);
            putchar(grn);
60          putchar(blu);
@@ -105,6 +108,7 @@ int main(int argc, char **argv) {
    if (s_r) { stream_free(s_r); }
    if (s_a) { stream_free(s_a); }
    if (s_d) { stream_free(s_d); }
65 +  if (s_p) { stream_free(s_p); }

    return retval;
  }
  diff --git a/code/mandelbrot.h b/code/mandelbrot.h
70 index 1b4ad9f..d9e3050 100644
--- a/code/mandelbrot.h
+++ b/code/mandelbrot.h
@@ -11,6 +11,7 @@ struct pixel {
    int final_n;
75  complex float final_z;
    complex float final_dc;
+   int final_p;
  };

80 struct image {
@@ -134,6 +135,22 @@ int image_save(struct image *img, char *↙
    ↳ filestem, float escape_radius, double pi
        retval = 1;
      }

85 +    snprintf(filename, length, "%s_p.ppm", filestem);
+     f = fopen(filename, "wb");
+     if (f) {
+       fprintf(f, "P6\n%d %d\n255\n", img->width, img->height);
+       for (int j = 0; j < img->height; ++j) {
90 +         for (int i = 0; i < img->width; ++i) {
+           int n = img->pixels[j * img->width + i].final_p;
+           fputc(n >> 16, f);
+           fputc(n >>  8, f);
+           fputc(n      , f);
95 +         }
+       }
+       fclose(f);
+     } else {
+       retval = 1;
100 +   }

      free(filename);
    } else {
  diff --git a/code/mandelbrot_imp.c b/code/mandelbrot_imp.c
```

```
index 7d83325..7d7a960 100644
--- a/code/mandelbrot_imp.c
+++ b/code/mandelbrot_imp.c
@@ -1,4 +1,4 @@
-FTYPE FNAME(pi) = FNAME(3.14159265358979323846264338327950288419716939937510);
+const FTYPE FNAME(pi) = FNAME(3.14159265358979323846264338327950288419716939937510);

 FTYPE FNAME(cabs2)(complex FTYPE z) {
   return FNAME(creal)(z) * FNAME(creal)(z) + FNAME(cimag)(z) * FNAME(cimag)(z);
@@ -15,15 +15,24 @@ void FNAME(calculate)(struct pixel *out, complex FTYPE c, int maximum_iterations
   out->final_n = 0;
   out->final_z = 0;
   out->final_dc = 0;
+  out->final_p = 0;
   complex FTYPE z = 0;
   complex FTYPE dc = 0;
+  FTYPE mz2 = FNAME(1.0) / FNAME(0.0);
+  int p = 0;
   for (int n = 1; n < maximum_iterations; ++n) {
     dc = 2 * z * dc + 1;
     z = z * z + c;
-    if (FNAME(cabs2)(z) > escape_radius2) {
+    FTYPE z2 = FNAME(cabs2)(z);
+    if (z2 < mz2) {
+      mz2 = z2;
+      p = n;
+    }
+    if (z2 > escape_radius2) {
       out->final_n = n;
       out->final_z = z;
       out->final_dc = dc;
+      out->final_p = p;
       break;
     }
   }
```

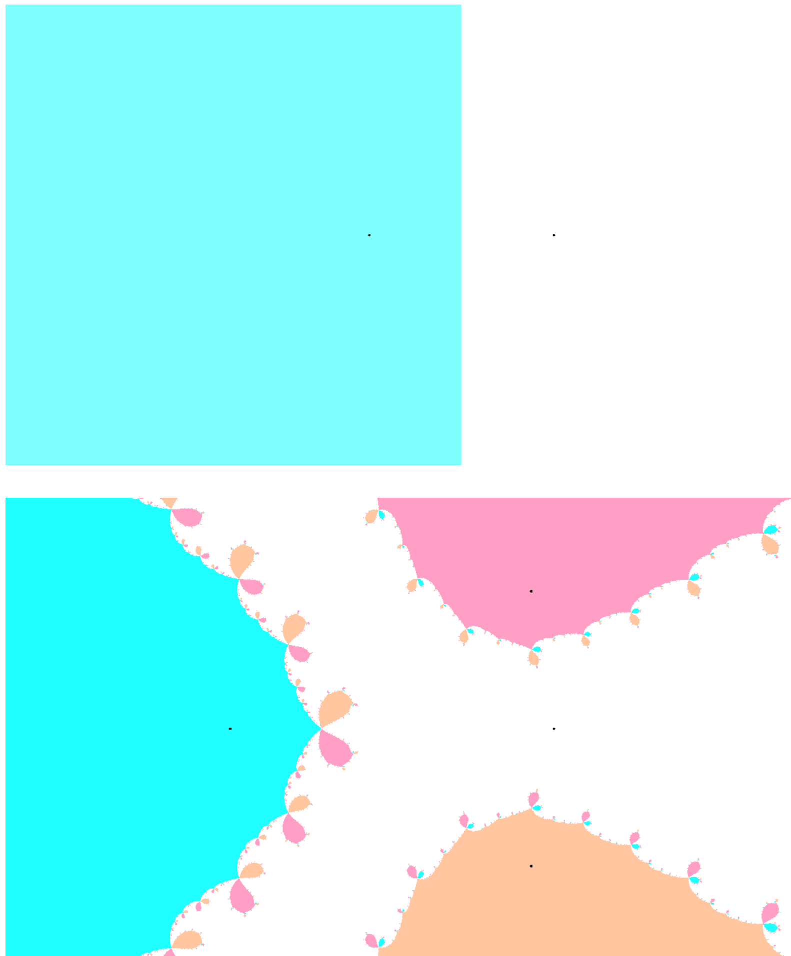
2.3 Nucleus basins

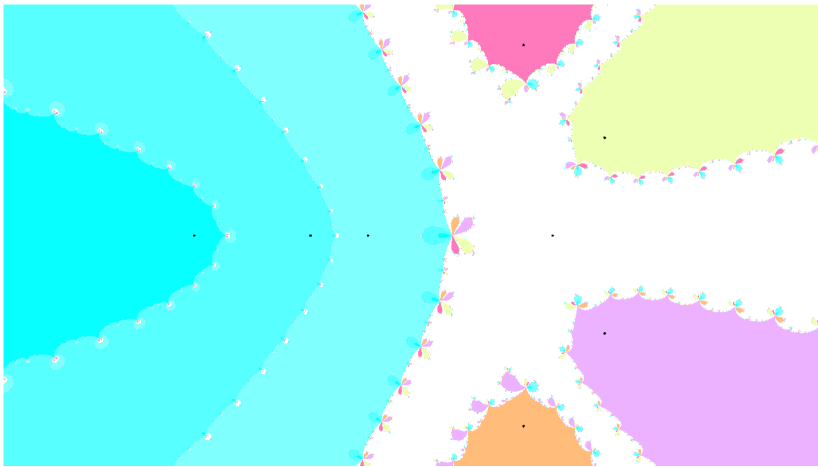
As mentioned previously, the centers of the blobs are roots of the polynomial formed by repeating a fixed number of iterations: at a nucleus of period p, $F^p(0,c) = 0$. Newton's Method can be used to find the roots numerically. Starting from an initial guess c_0, iterate:

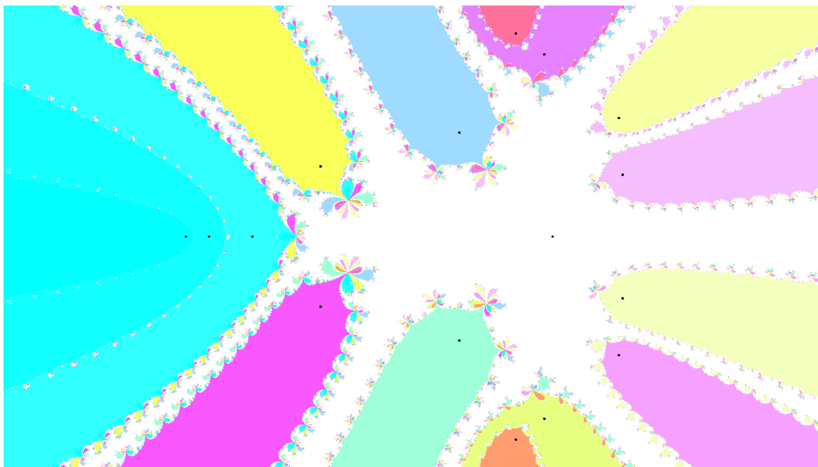
$$c_{m+1} = c_m - \frac{F^p(0,c_m)}{\frac{\partial}{\partial c}F^p(0,c_m)}$$

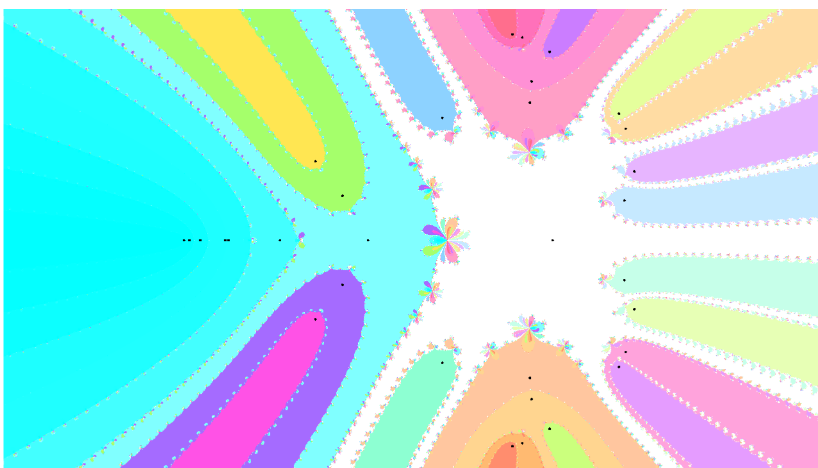
We can stop once the subtracted term is insignificant and the iteration has converged to a root. We can plot the basins of attraction by colouring according to the coordinates of the root reached, and we can plot the roots themselves by checking the distance between the coordinates

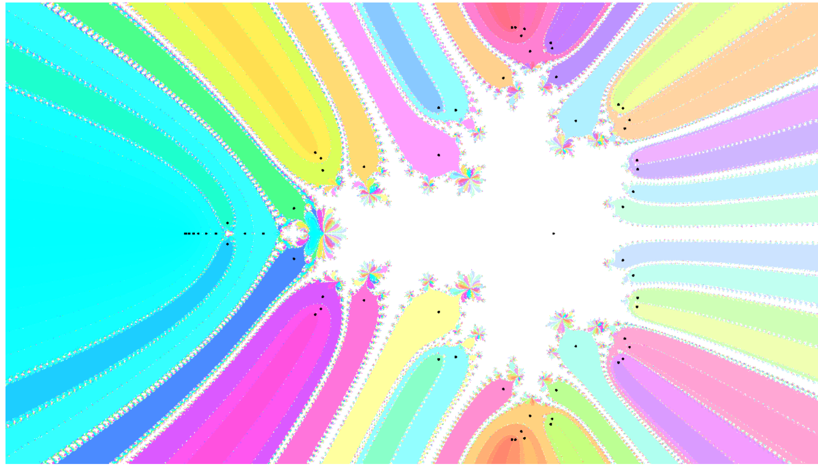
of the root and our initial guess (the coordinates of each pixel). Similarly to the blobs appearing and disappearing, roots can be reached that have a true period that is a factor of the period in the Newton iteration.

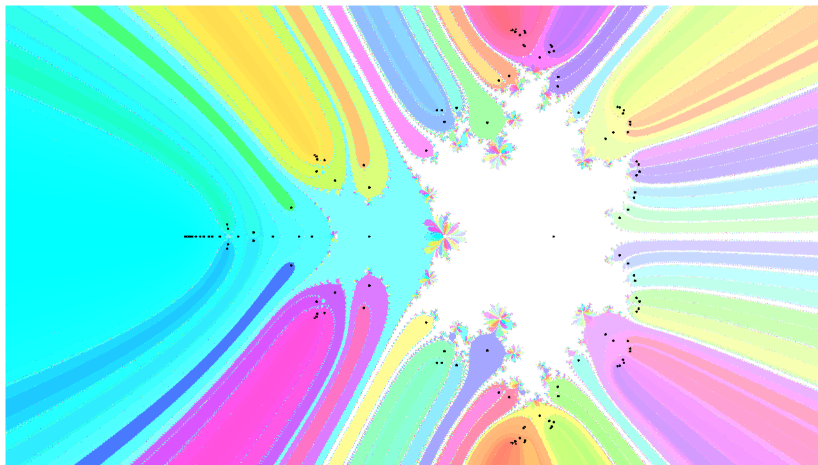


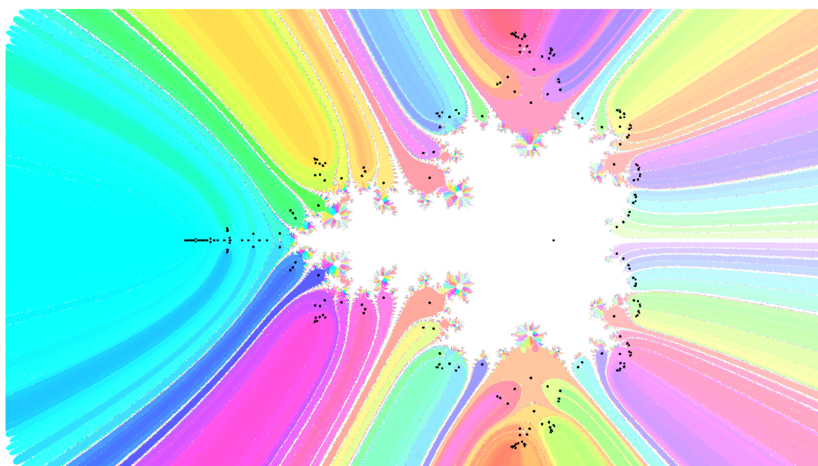


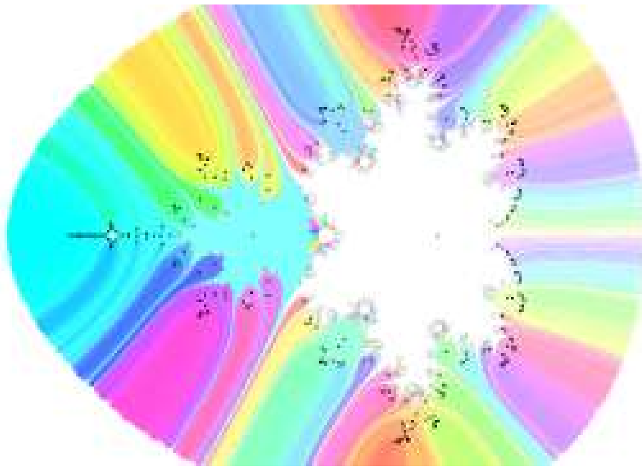


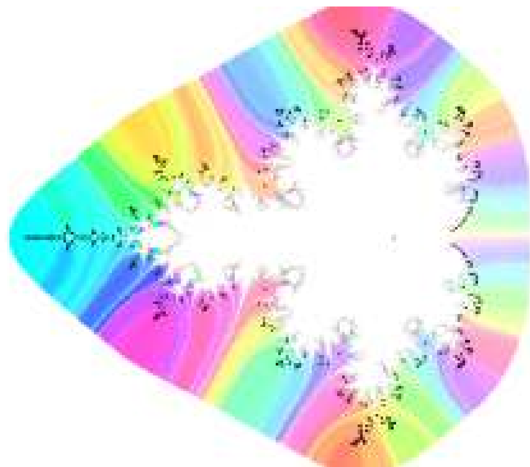


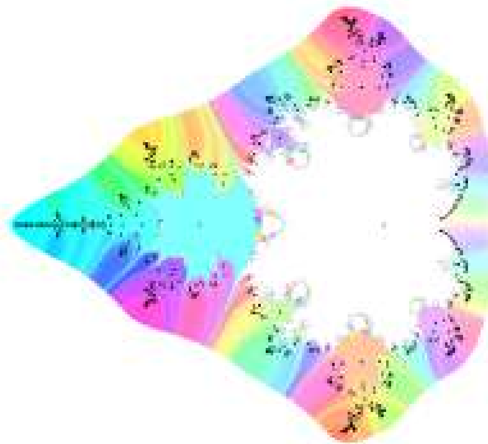












```
diff --git a/.gitignore b/.gitignore
index 61a35d5..238d6d1 100644
--- a/.gitignore
+++ b/.gitignore
@@ -2,6 +2,7 @@ code/annotate
```

```
 code/render
 code/colour
 code/interior
+code/newton
 code/*.o
 book/*.aux
 book/*.loa
diff --git a/code/Makefile b/code/Makefile
index 3292d62..e55e435 100644
--- a/code/Makefile
+++ b/code/Makefile
@@ -1,7 +1,7 @@
 COMPILE = gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp
 LINK = gcc -fopenmp

-all: render colour annotate interior
+all: render colour annotate interior newton

 clean:
         rm -f annotate colour render interior annotate.o colour.o ↙
             ↳ hsv2rgb.o image.o interior.o mandelbrot_double.o ↙
             ↳ mandelbrot_float.o mandelbrot_long_double.o pi.o ↙
             ↳ render.o stream.o
@@ -10,6 +10,7 @@ annotate: annotate.o stream.o
 colour: colour.o stream.o hsv2rgb.o
 render: render.o mandelbrot_float.o mandelbrot_double.o ↙
     ↳ mandelbrot_long_double.o image.o pi.o
 interior: interior.o mandelbrot_float.o pi.o
+newton: newton.o mandelbrot_long_double.o pi.o hsv2rgb.o

 .SUFFIXES:

diff --git a/code/newton.c b/code/newton.c
new file mode 100644
index 0000000..09c8f36
--- /dev/null
+++ b/code/newton.c
@@ -0,0 +1,55 @@
+#include "mandelbrot.h"
+#include "hsv2rgb.h"
+#include "pi.h"
+
+#define width 1280
+#define height 720
+#define maximum_iterations 12
+
+complex long double n[height][width];
+
+complex long double nucleusl(complex long double c0, int period, ↙
    ↳ int max_iters) {
+  complex long double c = c0;
+  for (int j = 0; j < max_iters; ++j) {
+    complex long double z = 0;
```

```
+     complex long double dc = 0;
+     for (int i = 0; i < period; ++i) {
+       dc = 2.0 * z * dc + 1.0;
+       z = z * z + c;
+     }
+     complex long double cc = c - z / dc;
+     if (cabs2l(c - cc) < 1.0e-16) {
+       return cc;
+     }
+     c = cc;
+   }
+   return 0.0;
+}
+
+int main() {
+   memset(n, 0, sizeof(n[0][0]) * height * width);
+   for (int k = 0; k <= maximum_iterations; ++k) {
+     #pragma omp parallel for
+     for (int j = 0; j < height; ++j) {
+       for (int i = 0; i < width; ++i) {
+         complex long double c = coordinatel(i, j, width, height, -0.75f, 1.25f);
+         n[j][i] = nucleusl(c, k, 256);
+       }
+     }
+     fprintf(stdout, "P6\n%d %d\n255\n", width, height);
+     for (int j = 0; j < height; ++j) {
+       for (int i = 0; i < width; ++i) {
+         complex long double c = coordinatel(i, j, width, height, -0.75f, 1.25f);
+         float hue = 7.0f * carg(n[j][i]) / (2.0f * pif);
+         float sat = cabs(n[j][i]) / 2.0;
+         float val = cabs(n[j][i] - c) < 4 * 1.25/height ? 0.0f : 1.0f;
+         float red, grn, blu;
+         hsv2rgb(hue, sat, val, &red, &grn, &blu);
+         fputc(channel(red), stdout);
+         fputc(channel(grn), stdout);
+         fputc(channel(blu), stdout);
+       }
+     }
+   }
+   return 0;
+}
```

Chapter 3

Notebook

3.1 Newton's Method: Nucleus

The nucleus n of a hyperbolic component of period p satisfies

$$F^p(0, n) = 0$$

Applying Newton's method in one complex variable gives

$$n_{m+1} = n_m - \frac{F^p(0, n_m)}{\frac{\partial}{\partial c} F^p(0, n_m)}$$

3.2 Newton's Method: Wucleus

A wucleus w of a point c within a hyperbolic component of period p satisfies

$$F^p(w, c) = w$$

Applying Newton's method in one complex variable gives

$$w_{m+1} = w_m - \frac{F^p(w_m, c) - w_m}{\frac{\partial}{\partial z} F^p(w_m, c) - 1}$$

This iteration is unstable because there are p distinct w_i. Perhaps use p-dimensional Newton's method to find all w_i at once, or trace rays from the orbit of the nucleus to the orbit containing the wuclei.

3.3 Newton's Method: Bond

The bond b between a hyperbolic component of period p and nucleus n at internal angle θ measured in turns satisfies

$$F^p(w, b) = w$$

$$\frac{\partial}{\partial z} F^p(w, b) = e^{2\pi i \theta}$$

Applying Newton's method in two complex variables gives

$$\begin{pmatrix} \frac{\partial}{\partial z} F^p(w_m, b_m) - 1 & \frac{\partial}{\partial c} F^p(w_m, b_m) \\ \frac{\partial}{\partial z} \frac{\partial}{\partial z} F^p(w_m, b_m) & \frac{\partial}{\partial c} \frac{\partial}{\partial z} F^p(w_m, b_m) \end{pmatrix} \begin{pmatrix} w_{m+1} - w_m \\ b_{m+1} - b_m \end{pmatrix} = - \begin{pmatrix} F^p(w_m, b_m) - w_m \\ \frac{\partial}{\partial z} F^p(w_m, b_m) - e^{2\pi i \theta} \end{pmatrix}$$

3.4 Newton's Method: Ray In

The next point r along an external ray with current doubled angle θ, current depth p and current radius R satisfies

$$F^p(0, r) = \lambda R e^{2\pi i\theta}$$

where $\lambda < 1$ controls the sharpness of the ray. Applying Newton's method in one complex variable gives

$$r_{m+1} = r_m - \frac{F^p(0, r_m) - \lambda R e^{2\pi i\theta}}{\frac{\partial}{\partial c} F^p(0, r_m)}$$

When crossing dwell bands, double θ and increment p, resetting the radius R. Stop tracing when close to the target (for example when within the basin of attraction for Newton's method for nucleus).

3.5 Newton's Method: Ray Out

When crossing dwell bands, compute the doubling preimages θ_-, θ_+ of θ

$$\theta_- = \frac{\theta}{2}$$
$$\theta_+ = \frac{\theta + 1}{2}$$

Then compute r_- and r_+ using Newton's method, and choose the nearest to r_m to be r_{m+1}. Collect a list of which choices were made to determine the final θ in high precision without relying on the bits of θ itself. $\lambda > 1$ controls the sharpness of the ray. Stop traccing when r reaches the escape radius.

3.6 Newton's Method: Preperiodic

A preperiodic point u with period p and preperiod k satisfies

$$F^{p+k}(0, u) = F^k(0, u)$$

Applying Newton's method in one complex variable gives

$$u_{m+1} = u_m - \frac{F^{p+k}(0, u_m) - F^k(0, u_m)}{\frac{\partial}{\partial c} F^{p+k}(0, u_m) - \frac{\partial}{\partial c} F^k(0, u_m)}$$

However this may converge to a u' with lower preperiod. Verify by finding the lowest k' such that Newton's method with preperiod k' starting from u' still converges to u' and check that $k' = k$.

3.7 Derivatives Calculation

The quadratic polynomial $F(z,c) = z^2 + c$ and its derivatives under iteration

$$\begin{aligned}
F^{m+1}(z,c) &= F(F^m(z,c),c) \\
\frac{\partial}{\partial z} F^{m+1} &= 2F^m \frac{\partial}{\partial z} F^m \\
\frac{\partial}{\partial z}\frac{\partial}{\partial z} F^{m+1} &= 2F^m \frac{\partial}{\partial z}\frac{\partial}{\partial z} F^m + 2\left(\frac{\partial}{\partial z} F^m\right)^2 \\
\frac{\partial}{\partial c} F^{m+1} &= 2F^m \frac{\partial}{\partial c} F^m + 1 \\
\frac{\partial}{\partial c}\frac{\partial}{\partial z} F^{m+1} &= 2F^m \frac{\partial}{\partial c}\frac{\partial}{\partial z} F^m + 2\frac{\partial}{\partial c} F^m \frac{\partial}{\partial z} F^m
\end{aligned}$$

with initial values

$$\begin{aligned}
F^0(z,c) &= z \\
\frac{\partial}{\partial z} F^0 &= 1 \\
\frac{\partial}{\partial z}\frac{\partial}{\partial z} F^0 &= 0 \\
\frac{\partial}{\partial c} F^0 &= 0 \\
\frac{\partial}{\partial c}\frac{\partial}{\partial z} F^0 &= 0
\end{aligned}$$

3.8 External Angles: Bulb

The $\frac{p}{q}$ bulb of the period 1 cardioid has external angles

$$.(b_0 b_1 \ldots b_{q-3} 01).(b_0 b_1 \ldots b_{q-3} 10)$$

where

$$[b_0 b_1 \ldots] = \text{map}(\in (1 - \frac{p}{q}, 1))(\text{iterate}(+\frac{p}{q})\frac{p}{q})$$

3.9 External Angles: Hub

The $\frac{p}{q}$ bulb has external angles $.(s_-)$ and $.(s_+)$. The junction point of its hub has external angles in increasing order

$$\begin{aligned}
&.s_-(s_+) \\
&.s_-(\sigma^\beta s_+) \\
&\vdots \\
&.s_-(\sigma^{(q-p-1)\beta} s_+) \\
&.s_+(\sigma^{(q-p)\beta} s_+) \\
&\vdots \\
&.s_+(s_-)
\end{aligned}$$

where $\frac{p}{q}$ has Farey parents

$$\frac{\alpha}{\beta} < \frac{\gamma}{\delta}$$

and σ is the shift operator

$$\sigma^k(b_0 b_1 \ldots) = b_k b_{k+1} \ldots$$

3.10 Farey Numbers

Given $\frac{p}{q}$ in lowest terms, it has Farey parents $\frac{\alpha}{\beta} < \frac{\gamma}{\delta}$ where

$$\begin{aligned} \frac{p}{q} &= \frac{\alpha+\gamma}{\beta+\delta} \\ 1 &= p\beta - q\alpha \\ -1 &= p\delta - q\gamma \end{aligned}$$

Parents can be found by recursively searching from $\frac{0}{1}$ and $\frac{1}{1}$.

3.11 External Angles: Tuning

Given an external angle pair corresponding to a hyperbolic component

$$\begin{aligned} s_- &= .(a_0 a_1 \ldots) \\ s_+ &= .(b_0 b_1 \ldots) \end{aligned}$$

and an external angle

$$t = .c_0 c_1 \ldots (d_0 d_1 \ldots)$$

then t tuned by s is formed by replacing every 0 in t by s_- and every 1 in t by s_+ as finite blocks being the repeated part of each.

3.12 External Angles: Tips

Given hub angles $a_0, a_1, \ldots a_{m-1}$ the tip of the spoke between a_i and a_{i+1} has external angle with binary representation the longest matching prefix of a_i and a_{i+1} with 1 appended. The widest spoke has the shortest binary representation.

3.13 Translating Hubs Towards Tips

Translating a hub towards the tip of the widest spoke repeats the last preperiodic digit. Translating towards the tip of any other spoke replaces the preperiodic part with a modified tip: the final 1 becomes 01 for angles below the widest tip and the final 1 becomes 10 for angles above the widest tip.

3.14 Islands In The Spokes

The largest (lowest period) island after a hub is in the widest spoke. Its external angles are the repetitition of the first p (its period) digits of the external angles of the widest spoke.

Numbering spokes in decreasing order of width with the widest spoke numbered 1, the angled internal address for the path

$$\frac{p}{q} * s_0 s_1 \ldots s_m$$

is

$$1 \frac{p}{q} (q + \sum_{i=0}^{0} s_i)(q + \sum_{i=0}^{1} s_i) \ldots (q + \sum_{i=0}^{m} s_i)$$

3.15 Islands In The Hairs

An island with period $p > 1$ is surrounded by hairs. There are 2^n lowest period islands in the 2^n hairs at depth n and offset k, each with period

$$q = np + k$$

Their external angles are formed by counting in binary using the external angles of the parent period p island as digits

$$e = .(\pm_0 \pm_1 \pm_2 \ldots \pm_{n-1} 0) \text{ for } k = 1$$

where the external angles of p are

$$+ = .(+_0 +_1 \ldots +_{p-1})$$
$$- = .(-_0 -_1 \ldots -_{p-1})$$

3.16 Islands In Embedded Julias

An island of period q in the hairs of an island of period p is surrounded by an embedded Julia set. If the external angles of p are $.(+)$ and $.(-)$ and an external angle of q is $.(\ldots 0)$ then the external angles heading outwards along the hair within the embedded Julia set are

$$.(\ldots 0[+])$$

and heading inwards along the hair are

$$.(\ldots 0[-])$$

Mixing inwards and outwards motions gives

TODO

3.17 Multiply Embedded Julias

TODO diagram

3.18 Continuous Iteration Count

If n is the lowest such that

$$|z_n| > R >> 2$$

then

$$v = n - \log_2 \frac{\log |z_n|}{\log R}$$

where the value subtracted is in $[0, 1)$. When combined with the final angle $\theta = \arg z_n = \tan^{-1} \frac{\Im(z_n)}{\Re(z_n)}$ these can be used to tile the exterior with hyperbolic squares.

TODO diagram

The local coordinates of x are

$$(v \mod 1, \frac{\theta}{2\pi} \mod 1)$$

3.19 Interior Coordinates

The interior coordinate b for a point c within a hyperbolic component of period p can be found by applying Newton's method in one complex variable to find a wucleus w for c satisfying

$$F^p(w, c) = w$$

then

$$b = \frac{\partial}{\partial z} F^p(w, c)$$

The interior coordinate satisfies

$$|b| < 1$$

and can be used to map a disc into the interior of a component.

3.20 Exterior Distance

Given c outside the Mandelbrot set, the exterior distance estimate

$$d = \lim_{n \to \infty} 2 \frac{|F^n(0, c)| \log |F^n(0, c)|}{|\frac{\partial}{\partial c} F^n(0, c)|}$$

satisfies by the Koebe $\frac{1}{4}$ Theorem

$$\forall c'.|c - c'| < \frac{d}{4} \implies c' \notin M$$

Shading using $t = \tanh \frac{d}{\text{pixel size}}$ works well because $t \in [0, 1)$ and $\tanh 4 \approx 1$.

3.21 Interior Distance

Given c inside a hyperbolic component of period p, with a wucleus w, then the interior distance estimate

$$d = \frac{1 - |\frac{\partial}{\partial z} F^p(w, c)|^2}{\left| \frac{\partial}{\partial c} \frac{\partial}{\partial z} F^p(w, c) + \frac{\frac{\partial}{\partial z} \frac{\partial}{\partial z} F^p(w,c) \frac{\partial}{\partial c} F^p(w,c)}{1 - \frac{\partial}{\partial z} F^p(w,c)} \right|}$$

satisfies by the Koebe $\frac{1}{4}$ Theorem

$$\forall c'.|c - c'| < \frac{d}{4} \implies c' \in M$$

Shading using $t = \tanh \frac{d}{\text{pixel size}}$ works well because $t \in [0, 1)$ and $\tanh 4 \approx 1$.

3.22 Perturbation

TODO

3.23 Finding Atoms

Given an angled internal address, the nucleus and size of an atom can be found by:

- splitting the address to island and children
- finding the external angles of the island
- tracing the external ray towards the island
- using Newton's method to find the nucleus of the island

and then iteratively for the children

- find the bond at the internal angle of the child
- using the size estimate and normal at the bond point to estimate the nucleus
- using Newton's method to find the nucleus of the child

3.24 Child Size Estimates

For the $\frac{q}{p}$ child of a cardioid of size R:

$$r \approx \frac{R}{p^2} \sin\left(2\pi\frac{q}{p}\right)$$

For the $\frac{q}{p}$ child of a disc of size R:

$$r \approx \frac{R}{p^2}$$

The size of a cardioid is approximated by the distance from its cusp to its $\frac{1}{2}$ bond, and for a disc its radius. The nucleus n of the child at internal angle θ with bond b is approximated by

$$n \approx b - ir\frac{\partial\theta}{\partial b}$$

For the main period 1 cardioid

$$b = u(1 - u)$$

where

$$u = \frac{e^{2\pi i \frac{q}{p}}}{2}$$

3.25 Atom Shape Estimates

Given the nucleus c and period p of a hyperbolic component, the shape discriminant ϵ_p satisfies

$$\epsilon_p = -\frac{1}{(\frac{\partial}{\partial c})(\frac{\partial}{\partial z})}\left(\frac{\frac{\partial}{\partial c}\frac{\partial}{\partial c}}{2\frac{\partial}{\partial c}} + \frac{\frac{\partial}{\partial c}\frac{\partial}{\partial z}}{\frac{\partial}{\partial z}}\right)$$

where the derivatives are evaluated at $F^p(c,c)$. Then $\epsilon_p \approx 0$ for cardioids and $\epsilon_p \approx 1$ for discs.

3.26 Atom Size Estimates

Given nucleus c and period p, calculate γ by

$$\begin{aligned} z_0 &= 0 \\ z_{m+1} &= z_m^2 + c \\ \lambda &= \prod_{i=0}^{p-1} 2z_i \\ \beta &= \sum_{i=0}^{p-1} \frac{1}{\prod_{j=1}^{i} 2z_i} \\ \gamma &= \frac{1}{\beta\lambda^2} \end{aligned}$$

then the size estimate is $|\gamma|$ and the orientation estimate is $\arg\gamma$.

3.27 Tracing Equipotentials

Proceed as external ray tracing, but keep dwell and radius fixed and increment the angle by $\frac{2\pi}{\text{sharpness}}$.

3.28 Buddhabrot

For each exterior c, colour hue according to dwell and plot the z iterates into an accumulation buffer. Post-process this high dynamic range image to map brightness to point density.

3.29 Atom Domain Size Estimate

Given nucleus c with period p, find $1 \le q < p$ such that $|F^q(0,c)|$ is minimized. Then the atom domain size estimate is

$$R = \frac{F^q(0,c)}{\frac{\partial}{\partial c}F^p(0,c)}$$

3.30 Stretching Cusps

Moebius transformation:

$$\begin{aligned} f(z) &= \frac{az+b}{cz+d} \\ f^{-1}(z) &= \frac{dz-b}{-cz+a} \end{aligned}$$

Unwrapping circle to line:

$$g(z) = \frac{(z - p_0)(p_1 - p_\infty)}{(z - p_\infty)(p_1 - p_0)}$$

Mapping cardioids to circles:

$$h(z) = \sqrt{1 - 4z} - 1$$
$$h^{-1}(z) = \frac{1 - (z + 1)^2}{4}$$

Derivatives for distance estimation:

$$\frac{\partial}{\partial z} \frac{az + b}{cz + d} = \frac{ad - bc}{(cz + d)^2}$$
$$\frac{\partial}{\partial z} \frac{1 - (z + 1)^2}{4} = -\frac{z + 1}{2}$$

Chapter 4

Bonds.

The Mandelbrot set's mu-atoms each have a nucleus. Each mu-atom has adjacent children, connected by bond points. Given an estimate of the nucleus, and its period, one can compute the nucleus using Newton's method in one variable. Given the nucleus and its period, one can compute bond points using Newton's method in two variables. The required derivatives can be computed efficiently using recurrence relations.

4.1 Prerequisites

Definition 1 (Roots of functions)**.** *A root of a function f is an x such that $f(x) = 0$.*

Definition 2 (Jacobian matrix)**.** *The Jacobian matrix is the matrix of all first-order partial derivatives of a vector- or scalar-valued function with respect to another vector:*

$$\boldsymbol{y} = f(\boldsymbol{x}) \qquad J_f = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Definition 3 (Newton's method in one variable)**.** *Given a well-behaved function f and a first guess x_0 for a root of f, better approximations can be found by:*

$$x_{m+1} = x_m - \frac{f(x_m)}{\frac{\partial}{\partial x} f(x_m)}$$

Definition 4 (Newton's method in more than one variable)**.** *Given a well-behaved function f and a first guess $\boldsymbol{v}_0$ for a root of f, better approximations can be found by:*

$$\boldsymbol{v}_{m+1} = \boldsymbol{v}_m - J_f(\boldsymbol{v}_m)^{-1} f(\boldsymbol{v}_m)$$

or equivalently by solving:

$$J_f(\boldsymbol{v}_m)(\boldsymbol{v}_{m+1} - \boldsymbol{v}_m) = -f(\boldsymbol{v}_m)$$

4.2 Finding features

Definition 5 (Quadratic function).

$$f(z,c) = z^2 + c$$

Definition 6 (Iterated function).

$$f^0(z,c) = z$$
$$f^{m+1}(z,c) = f(f^m(z,c),c)$$

Theorem 1 (Finding nucleus). *A nucleus n of period p satisfies $f^p(0,n) = 0$. Applying Newton's method in one variable:*

$$n_{m+1} = n_m - \frac{f^p(0,n_m)}{\frac{\partial}{\partial c} f^p(0,n_m)}$$

Theorem 2 (Finding bond points). *Given a parent nucleus n of period p, the bond point b at angle a (measured in turns) satisfies:*

$$f^p(w,b) - w = 0$$
$$\frac{\partial}{\partial z} f^p(w,b) - e^{2\pi i a} = 0$$

Defining:

$$g\begin{pmatrix} z \\ c \end{pmatrix} = \begin{pmatrix} f^p(z,c) - z \\ \frac{\partial}{\partial z} f^p(z,c) - e^{2\pi i a} \end{pmatrix}$$

and applying Newton's method in two variables:

$$v_0 = \begin{pmatrix} n \\ n \end{pmatrix}$$
$$J_g(\boldsymbol{v}_m)(\boldsymbol{v}_{m+1} - \boldsymbol{v}_m) = -g(\boldsymbol{v}_m)$$

where

$$J_g = \begin{pmatrix} \frac{\partial}{\partial z}(f^p(z,c) - z) & \frac{\partial}{\partial c}(f^p(z,c) - z) \\ \frac{\partial}{\partial z}(\frac{\partial}{\partial z} f^p(z,c) - e^{2\pi i a}) & \frac{\partial}{\partial c}(\frac{\partial}{\partial z} f^p(z,c) - e^{2\pi i a}) \end{pmatrix}$$

4.3 Recurrence relations

Definition 7 (Naming derivatives). *Picking names arbitrarily for the required derivatives:*

$$A_m = f^m$$
$$B_m = \frac{\partial}{\partial z} f^m$$
$$C_m = \frac{\partial}{\partial z}\frac{\partial}{\partial z} f^m$$
$$D_m = \frac{\partial}{\partial c} f^m$$
$$E_m = \frac{\partial}{\partial c}\frac{\partial}{\partial z} f^m$$

Theorem 3 (Nucleus recurrence)**.** *Theorem 1 can now be rewritten as:*

$$n_{m+1} = n_m - \frac{A_p(0, n_m)}{D_p(0, n_m)}$$

Theorem 4 (Bond point recurrence)**.** *Theorem 2 can now be rewritten as:*

$$\begin{pmatrix} w_0 \\ b_0 \end{pmatrix} = \begin{pmatrix} n \\ n \end{pmatrix}$$

$$\begin{pmatrix} B_p - 1 & D_p \\ C_p & E_p \end{pmatrix} \begin{pmatrix} w_{m+1} - w_m \\ b_{m+1} - b_m \end{pmatrix} = - \begin{pmatrix} A_p - w_m \\ B_p - e^{2\pi i a} \end{pmatrix}$$

where $\{A, B, C, D, E\}_p$ *are evaluated at* (w_m, b_m)*.*

Theorem 5 (Initial values)**.**

$$\begin{aligned} A_0 &= z \\ B_0 &= 1 \\ C_0 &= 0 \\ D_0 &= 0 \\ E_0 &= 0 \end{aligned}$$

Theorem 6 (Recurrence relation)**.**

$$\begin{aligned} A_{m+1} &= A_m^2 + c \\ B_{m+1} &= 2A_m B_m \\ C_{m+1} &= 2(B_m^2 + A_m C_m) \\ D_{m+1} &= 2A_m D_m + 1 \\ E_{m+1} &= 2(A_m E_m + B_m D_m) \end{aligned}$$

4.4 Implementation

Example implementation pseudo-code in Haskell:

```
-- compute successive approximations to nucleus
nucleus :: N -> C -> [C]
nucleus p n0 = iterate go n0
  where
    go nm =
      let (a, _, _, d, _) = derivatives p 0 nm
      in  nm - a / d

-- compute successive approximations to bond point
bond :: N -> C -> R -> [C]
bond p n0 a0 = map snd (iterate go (n0, n0))
  where
    t = cis (2 * pi * a0)
    go (wm, bm) =
```

```
      let (a, b, c, d, e) = derivatives p wm bm
          jm = toMatrix ((b - 1, d), (c, e))
          xm = toVector (wm - a, t - b)
          dxm = solve jm xm
          xm1 = xm + dxm
          (wm1, bm1) = fromVector xm1
      in  (wm1, bm1)

-- compute derivatives via recurrence relations
derivatives :: N -> C -> C -> (C, C, C, C, C)
derivatives p z0 c0 =
    iterate go (z0, 1, 0, 0, 0) `genericIndex` p
  where
    go (a, b, c, d, e) =
      ( a^2 + c0
      , 2 * a * b
      , 2 * (b^2 + a * c)
      , 2 * a * d + 1
      , 2 * (a * e + b * d)
      )
```

4.5 Proofs

Derivation of A_0.

$$A_0$$

$= \quad \{ \text{ definition of } A \text{ } \}$

$$f^0(z,c)$$

$= \quad \{ \text{ definition of } f \text{ } \}$

$$z$$

□

Derivation of B_0.

$$B_0$$

$= \quad \{ \text{ definition of } B \text{ } \}$

$$\frac{\partial}{\partial z} f^0(z,c)$$

$= \quad \{ \text{ definition of } f \text{ } \}$

$$\frac{\partial}{\partial z} z$$

$= \quad \{ \text{ derivative } \}$

$$1$$

□

Derivation of C_0.

$$
\begin{aligned}
& C_0 \\
= & \qquad \{ \text{ definition of } C \text{ } \} \\
& \frac{\partial}{\partial z}\frac{\partial}{\partial z} f^0(z,c) \\
= & \qquad \{ \text{ definition of } f \text{ } \} \\
& \frac{\partial}{\partial z}\frac{\partial}{\partial z} z \\
= & \qquad \{ \text{ derivative } \} \\
& \frac{\partial}{\partial z} 1 \\
= & \qquad \{ \text{ derivative } \} \\
& 0
\end{aligned}
$$

□

Derivation of D_0.

$$
\begin{aligned}
& D_0 \\
= & \qquad \{ \text{ definition of } D \text{ } \} \\
& \frac{\partial}{\partial c} f^0(z,c) \\
= & \qquad \{ \text{ definition of } f \text{ } \} \\
& \frac{\partial}{\partial c} z \\
= & \qquad \{ \text{ derivative } \} \\
& 0
\end{aligned}
$$

□

Derivation of E_0.

$$
\begin{aligned}
& E_0 \\
= & \qquad \{ \text{ definition of } E \text{ } \} \\
& \frac{\partial}{\partial c}\frac{\partial}{\partial z} f^0(z,c) \\
= & \qquad \{ \text{ definition of } f \text{ } \} \\
& \frac{\partial}{\partial c}\frac{\partial}{\partial z} z \\
= & \qquad \{ \text{ derivative } \} \\
& \frac{\partial}{\partial c} 1 \\
= & \qquad \{ \text{ derivative } \} \\
& 0
\end{aligned}
$$

□

Derivation of A_{m+1}.

$$A_{m+1}$$
$=$ { definition of A }
$$f^{m+1}(z,c)$$
$=$ { definition of f }
$$f(f^m(z,c),c)$$
$=$ { definition of A }
$$f(A_m,c)$$
$=$ { definition of f }
$$A_m^2 + c$$

□

Derivation of B_{m+1}.

$$B_{m+1}$$
$=$ { definition of B }
$$\frac{\partial}{\partial z}A_{m+1}$$
$=$ { definition of A }
$$\frac{\partial}{\partial z}(A_m^2 + c)$$
$=$ { distributivity }
$$\frac{\partial}{\partial z}A_m^2 + \frac{\partial}{\partial z}c$$
$=$ { constant derivative }
$$\frac{\partial}{\partial z}A_m^2 + 0$$
$=$ { zero }
$$\frac{\partial}{\partial z}A_m^2$$
$=$ { chain rule }
$$2A_m(\frac{\partial}{\partial z}A_m)$$
$=$ { definition of B }
$$2A_mB_m$$

□

Derivation of C_{m+1}.

$$
\begin{aligned}
& C_{m+1} \\
= & \qquad \{ \text{ definition of } C \text{ } \} \\
& \frac{\partial}{\partial z} B_{m+1} \\
= & \qquad \{ \text{ definition of } B \text{ } \} \\
& \frac{\partial}{\partial z}(2A_m B_m) \\
= & \qquad \{ \text{ linearity } \} \\
& 2\frac{\partial}{\partial z}(A_m B_m) \\
= & \qquad \{ \text{ product rule } \} \\
& 2((\frac{\partial}{\partial z}A_m)B_m + A_m(\frac{\partial}{\partial z}B_m)) \\
= & \qquad \{ \text{ definition of } B \text{ } \} \\
& 2(B_m B_m + A_m(\frac{\partial}{\partial z}B_m)) \\
= & \qquad \{ \text{ algebra } \} \\
& 2(B_m^2 + A_m(\frac{\partial}{\partial z}B_m)) \\
= & \qquad \{ \text{ definition of } C \text{ } \} \\
& 2(B_m^2 + A_m C_m)
\end{aligned}
$$

□

Derivation of D_{m+1}.

$$
\begin{aligned}
& D_{m+1} \\
= & \quad \{ \text{ definition of } D \text{ } \} \\
& \frac{\partial}{\partial c} A_{m+1} \\
= & \quad \{ \text{ definition of } A \text{ } \} \\
& \frac{\partial}{\partial c}(A_m^2 + c) \\
= & \quad \{ \text{ distributivity } \} \\
& \frac{\partial}{\partial c} A_m^2 + \frac{\partial}{\partial c} c \\
= & \quad \{ \text{ derivative } \} \\
& \frac{\partial}{\partial c} A_m^2 + 1 \\
= & \quad \{ \text{ chain rule } \} \\
& 2A_m(\frac{\partial}{\partial c} A_m) + 1 \\
= & \quad \{ \text{ definition of } D \text{ } \} \\
& 2A_m D_m + 1
\end{aligned}
$$

□

Derivation of E_{m+1}.

$$
\begin{aligned}
& E_{m+1} \\
= & \quad \{ \text{ definition of } E \text{ } \} \\
& \frac{\partial}{\partial c} B_{m+1} \\
= & \quad \{ \text{ definition of } B \text{ } \} \\
& \frac{\partial}{\partial c}(2A_m B_m) \\
= & \quad \{ \text{ linearity } \} \\
& 2\frac{\partial}{\partial c}(A_m B_m) \\
= & \quad \{ \text{ product rule } \} \\
& 2(A_m(\frac{\partial}{\partial c} B_m) + (\frac{\partial}{\partial c} A_m) B_m) \\
= & \quad \{ \text{ definition of } E \text{ } \} \\
& 2(A_m E_m + (\frac{\partial}{\partial c} A_m) B_m) \\
= & \quad \{ \text{ definition of } D \text{ } \} \\
& 2(A_m E_m + D_m B_m)
\end{aligned}
$$

□

Chapter 5

Perturbation.

5.1 The Mandelbrot set

Consider iterations of a quadratic polynomial F over complex numbers:

$$\begin{aligned} F(z,c) &= z^2 + c \\ F^{n+1}(z,c) &= F(F^n(z,c),c) \end{aligned} \tag{5.1}$$

The Mandelbrot set M is the set of c values for which the iterates of $z = 0$ remain bounded:

$$M = \{c \in \mathbb{C} : F^n(0,c) \nrightarrow \infty \text{ as } n \to \infty\} \tag{5.2}$$

The shape of M is exceedingly intricate, with variety increasing the further we zoom in. Zooming in requires more precise numerical methods: we need at least enough information to distinguish nearby points in the region we want to visualize. Using this much information for each point is certainly good enough, but as the precision increases it gets slower.

5.2 Perturbation techniques

The idea[1] is simple: take a high precision orbit for one point as a reference, and assuming a well-behaved function, the orbits for points near to the reference will for the most part be near to the reference orbit. Instead of computing nearby orbits at high precision, save time and effort by computing only the difference from the reference orbit.

This works out because if you have two high precision numbers close together, their difference has less meaningful precision. For example

$$\begin{aligned} A &= 123456798 \\ B &= 123456789 \\ A - B &= 9 \end{aligned} \tag{5.3}$$

even with A and B known to 9 significant figures, we can only determine their difference to 1 significant figure. Most commonly computers use binary floating point arithmetic, and the

[1] http://superfractalthing.co.nf/sft_maths.pdf

precision lost e when subtracting nearby values x and y can be measured[2] in bits:

$$e = -\log_2 \left(1 - \frac{\min(|x|, |y|)}{\max(|x|, |y|)} \right) \tag{5.4}$$

Measuring the error that accumulates when subtracting (or adding numbers with opposite sign) while calculating the perturbed orbit can inform us when the reference orbit isn't good enough (possibly giving us a badly glitched blobby image), and even provide hints for choosing a better reference location.

5.3 Applying the technique

Choose a reference c and iterate z (and its derivatives if needed) with high precision:

$$\begin{aligned}
z_0 &= z & z_{n+1} &= z_n^2 + c \\
\frac{\partial}{\partial z} z_0 &= 1 & \frac{\partial}{\partial z} z_{n+1} &= 2 z_n \frac{\partial}{\partial z} z_n \\
\frac{\partial}{\partial c} z_0 &= 0 & \frac{\partial}{\partial c} z_{n+1} &= 2 z_n \frac{\partial}{\partial c} z_n + 1 \\
\frac{\partial}{\partial z}\frac{\partial}{\partial z} z_0 &= 0 & \frac{\partial}{\partial z}\frac{\partial}{\partial z} z_{n+1} &= 2 z_n \frac{\partial}{\partial z}\frac{\partial}{\partial z} z_n + 2 {\frac{\partial}{\partial z} z_n}^2 \\
\frac{\partial}{\partial c}\frac{\partial}{\partial z} z_0 &= 0 & \frac{\partial}{\partial c}\frac{\partial}{\partial z} z_{n+1} &= 2 z_n \frac{\partial}{\partial c}\frac{\partial}{\partial z} z_n + 2 \frac{\partial}{\partial c} z_n \frac{\partial}{\partial z} z_n
\end{aligned} \tag{5.5}$$

Define the deltas $\langle\!\langle c \rangle\!\rangle, \langle\!\langle z \rangle\!\rangle, \ldots$ for a nearby orbit $C, Z, \ldots$ which can be computed with lower precision:

$$\begin{aligned}
C &= c + \langle\!\langle c \rangle\!\rangle & Z_n &= z_n + \langle\!\langle z_n \rangle\!\rangle \\
\frac{\partial}{\partial z} Z_n &= \frac{\partial}{\partial z} z_n + \langle\!\langle \frac{\partial}{\partial z} z_n \rangle\!\rangle & \frac{\partial}{\partial c} Z_n &= \frac{\partial}{\partial c} z_n + \langle\!\langle \frac{\partial}{\partial c} z_n \rangle\!\rangle \\
\frac{\partial}{\partial z}\frac{\partial}{\partial z} Z_n &= \frac{\partial}{\partial z}\frac{\partial}{\partial z} z_n + \langle\!\langle \frac{\partial}{\partial z}\frac{\partial}{\partial z} z_n \rangle\!\rangle & \frac{\partial}{\partial c}\frac{\partial}{\partial z} Z_n &= \frac{\partial}{\partial c}\frac{\partial}{\partial z} z_n + \langle\!\langle \frac{\partial}{\partial c}\frac{\partial}{\partial z} z_n \rangle\!\rangle
\end{aligned} \tag{5.6}$$

Some boring algebraic manipulation gives the iterations for the deltas:

$$\begin{aligned}
\langle\!\langle z_{n+1} \rangle\!\rangle &= 2 z_n \langle\!\langle z_n \rangle\!\rangle + \langle\!\langle z_n \rangle\!\rangle^2 + \langle\!\langle c \rangle\!\rangle \\
\langle\!\langle \frac{\partial}{\partial z} z_{n+1} \rangle\!\rangle &= 2 \left(\frac{\partial}{\partial z} z_n \langle\!\langle z_n \rangle\!\rangle + z_n \langle\!\langle \frac{\partial}{\partial z} z_n \rangle\!\rangle + \langle\!\langle z_n \rangle\!\rangle \langle\!\langle \frac{\partial}{\partial z} z_n \rangle\!\rangle \right) \\
\langle\!\langle \frac{\partial}{\partial c} z_{n+1} \rangle\!\rangle &= 2 \left(\frac{\partial}{\partial c} z_n \langle\!\langle z_n \rangle\!\rangle + z_n \langle\!\langle \frac{\partial}{\partial c} z_n \rangle\!\rangle + \langle\!\langle z_n \rangle\!\rangle \langle\!\langle \frac{\partial}{\partial c} z_n \rangle\!\rangle \right) \\
\langle\!\langle \frac{\partial}{\partial z}\frac{\partial}{\partial z} z_{n+1} \rangle\!\rangle &= 2 \left(\frac{\partial}{\partial z}\frac{\partial}{\partial z} z_n \langle\!\langle z_n \rangle\!\rangle + z_n \langle\!\langle \frac{\partial}{\partial z}\frac{\partial}{\partial z} z_n \rangle\!\rangle + 2 z_n \langle\!\langle \frac{\partial}{\partial z} z_n \rangle\!\rangle \right. \\
&\quad \left. + \langle\!\langle z_n \rangle\!\rangle \langle\!\langle \frac{\partial}{\partial z}\frac{\partial}{\partial z} z_n \rangle\!\rangle + \langle\!\langle \frac{\partial}{\partial z} z_n \rangle\!\rangle^2 \right) \\
\langle\!\langle \frac{\partial}{\partial c}\frac{\partial}{\partial z} z_{n+1} \rangle\!\rangle &= 2 \left(\frac{\partial}{\partial c}\frac{\partial}{\partial z} z_n \langle\!\langle z_n \rangle\!\rangle + z_n \langle\!\langle \frac{\partial}{\partial c}\frac{\partial}{\partial z} z_n \rangle\!\rangle + \langle\!\langle z_n \rangle\!\rangle \langle\!\langle \frac{\partial}{\partial c}\frac{\partial}{\partial z} z_n \rangle\!\rangle \right. \\
&\quad \left. + \frac{\partial}{\partial c} z_n \langle\!\langle \frac{\partial}{\partial z} z_n \rangle\!\rangle + \langle\!\langle \frac{\partial}{\partial c} z_n \rangle\!\rangle \frac{\partial}{\partial z} z_n + \langle\!\langle \frac{\partial}{\partial c} z_n \rangle\!\rangle \langle\!\langle \frac{\partial}{\partial z} z_n \rangle\!\rangle \right)
\end{aligned} \tag{5.7}$$

[2] http://en.wikipedia.org/wiki/Loss_of_significance#Loss_of_significant_bits

For interior coordinates[3] and interior distance estimation[4], we need to solve $Z_p = Z_0$, but when $z_p = z_0$ we can apply the perturbation technique to Newton's method for root finding:

$$\begin{aligned} Z_0^{(m+1)} &= Z_0^{(m)} - \frac{Z_p^{(m)} - Z_0^{(m)}}{\frac{\partial}{\partial z} Z_p^{(m)} - 1} \\ \langle\langle z_0 \rangle\rangle^{(m+1)} &= \langle\langle z_0 \rangle\rangle^{(m)} - \frac{\langle\langle z_p \rangle\rangle^{(m)} - \langle\langle z_0 \rangle\rangle^{(m)}}{\frac{\partial}{\partial z} z_p + \langle\langle \frac{\partial}{\partial z} z_p \rangle\rangle^{(m)} - 1} \end{aligned} \tag{5.8}$$

The precondition isn't too onerous, as it's often sensible to choose a periodic point as a reference, and it's enough if p is a multiple of the period of the reference.

5.4 Series approximation

The deltas are polynomial series in $\langle\langle c \rangle\rangle$. Define the coeffecients $[\![z_n]\!]_m, [\![\frac{\partial}{\partial c} z_n]\!]_m$ of the polynomials for each delta:

$$\langle\langle z_n \rangle\rangle = \sum [\![z_n]\!]_m \langle\langle c \rangle\rangle^m \quad \langle\langle \frac{\partial}{\partial c} z_n \rangle\rangle = \sum [\![\frac{\partial}{\partial c} z_n]\!]_m \langle\langle c \rangle\rangle^m \tag{5.9}$$

Some boring algebraic manipulation gives the iterations for the first few coefficients of $\langle\langle z_n \rangle\rangle$:

$$\begin{aligned} [\![z_{n+1}]\!]_1 &= 2 z_n [\![z_n]\!]_1 + 1 \\ [\![z_{n+1}]\!]_2 &= 2 z_n [\![z_n]\!]_2 + [\![z_n]\!]_1^2 \\ [\![z_{n+1}]\!]_3 &= 2 z_n [\![z_n]\!]_3 + 2 [\![z_n]\!]_1 [\![z_n]\!]_2 \end{aligned} \tag{5.10}$$

and similarly for the coefficients of $\langle\langle \frac{\partial}{\partial c} z_n \rangle\rangle$:

$$\begin{aligned} [\![\frac{\partial}{\partial c} z_{n+1}]\!]_1 &= 2\left(\frac{\partial}{\partial c} z_n [\![z_n]\!]_1 + z_n [\![\frac{\partial}{\partial c} z_n]\!]_1\right) \\ [\![\frac{\partial}{\partial c} z_{n+1}]\!]_2 &= 2\left(\frac{\partial}{\partial c} z_n [\![z_n]\!]_2 + z_n [\![\frac{\partial}{\partial c} z_n]\!]_2 + [\![z_n]\!]_1 [\![\frac{\partial}{\partial c} z_n]\!]_1\right) \\ [\![\frac{\partial}{\partial c} z_{n+1}]\!]_3 &= 2\left(\frac{\partial}{\partial c} z_n [\![z_n]\!]_3 + z_n [\![\frac{\partial}{\partial c} z_n]\!]_3 + [\![z_n]\!]_1 [\![\frac{\partial}{\partial c} z_n]\!]_2 + [\![z_n]\!]_2 [\![\frac{\partial}{\partial c} z_n]\!]_1\right) \end{aligned} \tag{5.11}$$

The coefficients are independent of $\langle\langle c \rangle\rangle$ so the same coefficients can be used for many points in an image, and when $|\langle\langle c \rangle\rangle|$ is small the sum can be approximated by truncating to the first few terms. However, the coefficients grow quickly as n increases, which limits how long the per-reference approximation remains valid, after which we have to switch back to per-point delta iteration.

[3] http://mathr.co.uk/blog/2013-04-01_interior_coordinates_in_the_mandelbrot_set.html
[4] http://en.wikipedia.org/wiki/Mandelbrot_set#Interior_distance_estimation