

How To Write A Book  
About The Mandelbrot Set

Claude Heiland-Allen

December 24, 2013



# Contents

<b>I</b>	<b>Imaging <math>M</math></b>	<b>7</b>
<b>1</b>	<b>The Exterior</b>	<b>9</b>
1.1	A minimal C program. . . . .	9
1.2	Building a C program. . . . .	9
1.3	Ignoring generated files. . . . .	10
1.4	Input and output. . . . .	10
1.5	Printing all arguments: a “for” loop. . . . .	11
1.6	Device coordinates. . . . .	11
1.7	User coordinates. . . . .	12
1.8	Aspect ratio. . . . .	14
1.9	Complex numbers. . . . .	15
1.10	The Mandelbrot set. . . . .	16
1.11	Converting strings to numbers. . . . .	18
1.12	Image output. . . . .	20
1.13	Escaping a loop early: “break”. . . . .	21
1.14	Avoiding square roots. . . . .	22
1.15	Escape time colouring. . . . .	23
1.16	Refactoring. . . . .	24
1.17	Double precision. . . . .	25
1.18	Generic programming with preprocessor macros. . . . .	28
1.19	Colour images. . . . .	33
1.20	Decoupling rendering and colouring. . . . .	36
1.21	Decoupling rendering from image output. . . . .	42
1.22	Parallel rendering with OpenMP. . . . .	44
1.23	Final angle colouring. . . . .	45
1.24	Fixing bugs. . . . .	51
1.25	Defensive error handling. . . . .	52
1.26	Automatic number type selection. . . . .	55
1.27	Final radius colouring. . . . .	57
1.28	Smooth colouring with continuous escape time. . . . .	60
1.29	Generating a grid. . . . .	61
1.30	Improving the grid. . . . .	61
1.31	Distance estimation. . . . .	62
1.32	Removing distracting colour. . . . .	66
1.33	Refactoring image stream reading. . . . .	66



# Listings

78b19af2c44dd3e379769b4c15c7bd926ca52482.diff . . . . .	9
4ee49c5a4ca68969c155647637846d58ffd5505f.diff . . . . .	9
19db4ccaf9ba37f2ce1c5304d6f2ae57bd74b6cb.diff . . . . .	10
a0fe15d74f588c4d8ab6654c10ed0278a5c75b09.diff . . . . .	10
c3aed969988c9261a66b23648fab8f8449cabff8.diff . . . . .	11
1e7226121c5ef78a57b285686e7d1c52f4754b79.diff . . . . .	12
952ffad52923291b390587dcc90c0c7d1aefbb3e.diff . . . . .	13
80fce2bd7e0835cdbbc137084da594b3fdb348b47.diff . . . . .	14
912c346b11b8be2447c56e8152c0389385aee885.diff . . . . .	16
bcba5f0deb472299e17b96e35fe514992496ce30.diff . . . . .	17
024682465b9a56ec28bceb65e877d6e5636effcd.diff . . . . .	19
c71fb8b6c4775c6087fa23173a1613abefcb197a.diff . . . . .	21
73850281d75235cac5dc192a2f559f5fba36ec7d.diff . . . . .	22
66567c93430e53142108cdde4b6551ebebdc8c1cb.diff . . . . .	22
c1667b27527c2eeb3f3fa12ead01b6276f63d627.diff . . . . .	23
7b71abc5e0df24258b1bfe47369018995061a09f.diff . . . . .	24
ecb18f012205e586c683a15c5829d71d06dcdb32.diff . . . . .	26
4ecccb39c916dca81a60860029b23aeb62a2a1b7.diff . . . . .	29
702198f081c208957d77323fac0b713c131ab422.diff . . . . .	34
24e8b81a7b49818148e6a0c719e70ad53a8fca9c.diff . . . . .	37
706785ce2684df3c7bae5caaa92885fcfef14300.diff . . . . .	42
9dfc508d2c4af17bcd951c419487411ebca4b8d7.diff . . . . .	45
99e73f934ddd79ebdbbb2f6d929e6ffd3562df9a.diff . . . . .	46
64692c684bf50d22bed4f4cc4303dba070243788.diff . . . . .	51
20ff08458c29dc339a363cb155fb2e1a77f5c302.diff . . . . .	52
bd7aa445bfe2b02a2883dccb561c58dcc620aeb.diff . . . . .	56
bbd262f1df0b02d41c829800db414930c3f94e54.diff . . . . .	58
fb98ed31d9265937092682c1f04d0c34aecff54f.diff . . . . .	60
00d0fa5735a6b5f2ddeb0d8453a7403594a7baa0.diff . . . . .	61
6255e2b5e432f65e1318d0704b9dbe3b1b0041a8.diff . . . . .	62
e7ff7b7158c999b8fcc24e627c39226233181e38.diff . . . . .	63
52a33c0fe498cf918bd6c6ce8324c2697fb44546.diff . . . . .	66
6b5aebb7c5e6267a4c187bd38fc92151cab366c0.diff . . . . .	66



Part I  
Imaging  $M$





# 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 @@
5 +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
5 +++ 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
5 +++ 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
5 +++ b/mandelbrot.c
@@ -1,2 +1,5 @@
-int main() {
+int main() {
+int main(int argc, char **argv) {
+puts(argv[0]);
10 + 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
5 | @@ -3,3 +3,5 @@
| int main(int argc, char **argv) {
| --- puts(argv[0]);
| + for (int i = 0; i < argc; ++i) {
| + puts(argv[i]);
10 | + }
| 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
5 @@ -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;
10 + for (int j = 0; j < height; ++j) {
+   for (int i = 0; i < width; ++i) {
+     printf("(%d,%d)", i, j);
+   }
15 +   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
```





```

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  $2^{\frac{1}{2}} = \sqrt{2} = 1.41421\dots$  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):

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

$$a + b = b + a \quad a \times b = b \times a$$

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

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  $\mathbb{I}$  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  $\mathbb{N}$  is for natural). The whole numbers with negative numbers (the integers) are written  $\mathbb{Z}$  (the  $\mathbb{Z}$  is for Zahlen, the German word for numbers). Fractions (the rational numbers) are written  $\mathbb{Q}$  (the  $\mathbb{Q}$  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  $\pi$  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
5 @@ -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
10 index 38f2a5f..5fed547 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -1 +1,2 @@
+#include <complex.h>
15 #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;
20 +   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  $|c| \leq 2$ . 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
```

```

-----
-----
-----
-----
-----
-----
-----

```









```

+   center = atof(argv[1]) + I * atof(argv[2]);
+   radius = atof(argv[3]);
+ }
15 | 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 interchangeably 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
$ display mandelbrot.pgm

```

What’s that blob on the left?

```

$ ./mandelbrot -1.75 0 0.125 > mandelbrot.pgm
$ display mandelbrot.pgm
$ ./mandelbrot -1.76 0 0.03 > mandelbrot.pgm
$ display 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
$ display mandelbrot.pgm

```

They also look like (shrunk, 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 +5,4 @@
   int main(int argc, char **argv) {
   int width = 72;
   int height = 22;
   float aspect = 0.5;
10 + int width = 1280;
   + int height = 720;
   complex float center = 0;
   @@ -16,7 +15,8 @@ int main(int argc, char **argv) {
   }
15 + 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);
20   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('');
30 + 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
5 @@ -24,2 +24,5 @@ int main(int argc, char **argv) {
+     z = z * z + c;
+     if (cabs(z) > escape_radius) {
+         break;
+     }
10 }
```

## 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
5 @@ -4,2 +4,6 @@
+float cabs2(complex float z) {
+ return creal(z) * creal(z) + cimag(z) * cimag(z);
+}
10 +
+ 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;
15  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) {
20         break;
@@ -28,3 +33,3 @@ int main(int argc, char **argv) {
    }
----- if (cabs(z) <= escape_radius) {
+     if (cabs2(z) <= escape_radius2) {
25         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
5 @@ -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) {
10         z = z * z + c;
        if (cabs2(z) > escape_radius2) {

```

```

+         final_n = n;
+         break;
@@ -33,7 +35,3 @@ int main(int argc, char **argv) {
15     }
+     if (cabs2(z) <= escape_radius2) {
+         putchar(0);
+     } else {
+         putchar(255);
20     }
+     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
5 @@ -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);
10 + complex float c = center + radius * (x - I * y);
+ return c;
+ }
+
+int calculate(complex float c, int maximum_iterations, float ↵
+    ↵ escape_radius2) {
15 + 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) {
20 +         final_n = n;
+         break;
+     }
+ }
+ return final_n;
25 + }
+

```



```

+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) {
30 +   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);
+   }
35 + }
+}
+
+ int main(int argc, char **argv) {
@@ -20,20 +51,3 @@ int main(int argc, char **argv) {
40 }
printf("P5\n%d_%d\n255\n", width, height);
for (int j = 0; j < height; ++j) {
  for (int i = 0; i < width; ++i) {
45     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) {
50       z = z * z + c;
      if (cabs2(z) > escape_radius2) {
        final_n = n;
        break;
      }
55     }
    putchar(final_n);
  }
}
+ render(width, height, center, radius, maximum_iterations, ↵
    ↵ escape_radius2);
60 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  $\log_2/\log_{10}$ , 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 pixelation 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
5 @@ -4,3 +4,7 @@

-float cabs2(complex float z) {
+float cabs2f(complex float z) {
+ return crealf(z) * crealf(z) + cimagf(z) * cimagf(z);
10 +}
+
+double cabs2(complex double z) {
+ return creal(z) * creal(z) + cimag(z) * cimag(z);
15 @@ -8,5 +12,5 @@ float cabs2(complex float z) {

-complex float coordinate(int i, int j, int width, int height, complex float center, float
- 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) {
20 + 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
```

```

25 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);
30 + return c;
+}
+
+int calculatef(complex float c, int maximum_iterations, float ↵
    ↵ escape_radius2) {
    complex float z = 0;
35 @@ -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;
40 +     }
+ }
+ 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;
50     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) {
55 + 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);
60 +         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);
65 @@ -32,3 +67,3 @@ void render(int width, int height, complex float ↵

```

```

    ↪ center, float radius, int maxim
    for (int i = 0; i < width; ++i) {
70 + ↪ 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;
75 float radius = 2;
    +   complex double center = 0;
    +   double radius = 2;
    +   int maximum_iterations = 256;
float escape_radius = 2;
80 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) {
85 @@ -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]);
90 +   }
    +   if (use_double) {
    +     render (width, height, center, radius, maximum_iterations, ↪
    ↪ escape_radius2);
    +   } else {
    +     renderf(width, height, center, radius, maximum_iterations, ↪
    ↪ escape_radius2);
95 +   }
    +   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 `strtol()` 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
5 @@ -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
10 index c95350e..4913865 100644
--- a/mandelbrot.c
+++ b/mandelbrot.c
@@ -4,71 +4,3 @@
15 -float cabs2f(complex float z) {
-   return crealf(z) * crealf(z) + cimagf(z) * cimagf(z);
-}
-
-double cabs2(complex double z) {
20 -   return creal(z) * creal(z) + cimag(z) * cimag(z);
-}
```

```

-
- complex float coordinatef(int i, int j, int width, int height, complex float center, float
- float x = (i - width / 2.0f) / (height / 2.0f);
25 - 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, double
- 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;
35 - }
-
- int calculatef(complex float c, int maximum_iterations, float escape_radius2) {
- complex float z = 0;
- int final_n = 0;
40 - for (int n = 1; n < maximum_iterations; ++n) {
- z = z * z + c;
- if (cabs2f(z) > escape_radius2) {
- final_n = n;
- break;
45 - }
- }
- 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;
55 - if (cabs2(z) > escape_radius2) {
- final_n = n;
- break;
- }
- }
- return final_n;
60 - }
-
- void renderf(int width, int height, complex float center, float radius, int maximum_it
- printf("P5\n%d %d\n255\n", width, height);
65 - 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);
70 - }
- }
- }

```

```

-
75 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);
80 putchar(final_n);
}
}
}
+#include "mandelbrot.h"
85
@@ -77,19 +9,25 @@ int main(int argc, char **argv) {
int height = 720;
complex double center = 0;
double radius = 2;
90 + complex long double center = 0;
+ long double radius = 2;
int maximum_iterations = 256;
double escape_radius = 2;
double escape_radius2 = escape_radius * escape_radius;
95 int use_double = 0;
+ long double escape_radius = 2;
+ long double escape_radius2 = escape_radius * escape_radius;
+ int float_type = 0;
if (argc > 3) {
100 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);
}
105 if (argc > 4) {
use_double = atoi(argv[4]);
+ float_type = atoi(argv[4]);
}
if (use_double) {
110 ↳ render(width, height, center, radius, maximum_iterations, escape_radius2);
} else {
↳ renderf(width, height, center, radius, maximum_iterations, escape_radius2);
+ switch (float_type) {
+ case 0:
115 + 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
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
155 new file mode 100644
index 0000000..e08bbcb
--- /dev/null
+++ b/mandelbrot_imp.c
@@ -0,0 +1,34 @@
160 #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 change 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 \
    -1.759937295271429505091916757 0.012591790526496335594898047 \
    2e- $\{exp\}$  2 0 >  $\{exp\}$ -grey.pgm
  ./mandelbrot \
    -1.759937295271429505091916757 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
5 @@ -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;
10  long double escape_radius2 = escape_radius * escape_radius;
   int float_type = 0;
+  int float_type = 1;
+  int colour_type = 0;
   if (argc > 3) {
15 @@ -22,11 +23,31 @@ int main(int argc, char **argv) {
   }
   switch (float_type) {
+  if (argc > 5) {
+    colour_type = atoi(argv[5]);
20 +  }
+  switch (colour_type) {
     case 0:
       renderf(width, height, center, radius, maximum_iterations, escape_radius2);
25 +   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);
30 +   break;
```

```

+         case 2:
+             render_greyl(width, height, center, radius, ↵
↳ maximum_iterations, escape_radius2);
+             break;
+         }
35         break;
+         case 1:
+             ↵
↳ render (width, height, center, radius, maximum_iterations, escape_radius2);
+             break;
+         case 2:
+             ↵
40         ↵ 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;
45         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);
50         break;
+         }
+         break;
diff --git a/mandelbrot.h b/mandelbrot.h
index 8f482db..55c72e8 100644
55 --- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -6,2 +6,8 @@

+void colour(int *r, int *g, int *b, int final_n) {
60 + *r = final_n * 8;
+ *g = final_n;
+ *b = final_n / 8;
+ }
+
65 #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
70 @@ -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 maximum_ite
+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);

```

```

75 @@ -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);
80 + 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;
85 +     colour(&r, &g, &b, final_n);
+     putchar(r);
+     putchar(g);
+     putchar(b);
+   }
90 + }
+}

```

## 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 re-render when we adjust the colour scheme.

```

diff --git a/.gitignore b/.gitignore
index c0729ec..2911ab0 100644
--- a/.gitignore
+++ b/.gitignore
5 @@ -1,2 @@
-mandelbrot
+render
+colour
diff --git a/Makefile b/Makefile
10 index 63e7211..6ef9ffe 100644
--- a/Makefile
+++ b/Makefile
@@ -1,2, +1,7 @@
-mandelbrot: mandelbrot.c mandelbrot.h mandelbrot_imp.c
15 -----
    ↪ 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 ↵
    ↪ c -lm
20 +
+render: render.c mandelbrot.h mandelbrot_imp.c
+    ↪ gcc -std=c99 -Wall -Wextra -pedantic -O3 -o render render.c ↵
    ↪ c -lm
diff --git a/colour.c b/colour.c
25 index 0000000..cba7d71
--- /dev/null
+++ b/colour.c

```

```

@@ -0,0 +1,64 @@
#include <math.h>
30 #include <stdio.h>
+
+int channel(float c) {
+ return fminf(fmaxf(roundf(255 * c), 0), 255);
+}
35 +
+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));
40 + int ii = i = floorf(h);
+ f = h - i;
+ p = v * (1 - s);
+ q = v * (1 - (s * f));
+ t = v * (1 - (s * (1 - f)));
45 + 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;
50 + case 4: r = t; g = p; b = v; break;
+ default:r = v; g = p; b = q; break;
+ }
+ }
+ *red = r;
55 + *grn = g;
+ *blu = b;
+}
+
+void colour(int *r, int *g, int *b, int final_n) {
60 + 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;
65 + hsv2rgb(hue, sat, val, &red, &grn, &blu);
+ *r = channel(red);
+ *g = channel(grn);
+ *b = channel(blu);
+}
70 +
+int main() {
+ int width = 0;
+ int height = 0;
+ if (2 == scanf("P6%d_%d_255", &width, &height)) {
75 + 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();

```

```

80 +         int g = getchar();
+         int b = getchar();
+         int final_n = (r << 16) | (g << 8) | b;
+         colour(&r, &g, &b, final_n);
+         putchar(r);
85 +         putchar(g);
+         putchar(b);
+     }
+ }
+ }
90 + }
+ return 0;
+}
diff --git a/mandelbrot.c b/mandelbrot.c
deleted file mode 100644
95 index 361009b..0000000
--- a/mandelbrot.c
+++ /dev/null
@@ -1,56 +0,0 @@
-#include <complex.h>
100 -#include <stdio.h>
-#include <stdlib.h>
-
-#include "mandelbrot.h"
-
105 -int main(int argc, char **argv) {
-    int width = 1280;
-    int height = 720;
-    complex long double center = 0;
-    long double radius = 2;
110 -    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;
115 -    if (argc > 3) {
-        center = strtold(argv[1], 0) + I * strtold(argv[2], 0);
-        radius = strtold(argv[3], 0);
-    }
-    if (argc > 4) {
120 -        float_type = atoi(argv[4]);
-    }
-    if (argc > 5) {
-        colour_type = atoi(argv[5]);
-    }
125 -    switch (colour_type) {
-        case 0:
-            switch (float_type) {
-                case 0:
-                    ↙
-                    ↘ render_greyf(width, height, center, radius, maximum_iterations, escape_radius2);
130 -                    break;
-                case 1:

```

```

    ↘ render_grey(width, height, center, radius, maximum_iterations, escape_radius2);
    break;
    case 2:
135 ↘ render_greyl(width, height, center, radius, maximum_iterations, escape_radius2);
    break;
    }
    break;
    case 1:
140 switch (float_type) {
    case 0:
    ↘ render_colourf(width, height, center, radius, maximum_iterations, escape_radius2);
    break;
    case 1:
145 ↘ render_colour(width, height, center, radius, maximum_iterations, escape_radius2);
    break;
    case 2:
    ↘ render_colourl(width, height, center, radius, maximum_iterations, escape_radius2);
    break;
150 }
    break;
}
return 0;
}
155 diff --git a/mandelbrot.h b/mandelbrot.h
index 55c72e8..ea5a50b 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
@@ -7,5 +7,5 @@
160 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;
165 + *g = final_n >> 8;
+ *b = final_n;
    }
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index 1fb27c7..0d576f9 100644
170 --- 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 ↘
175 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);
    ↪
180 ↪ int final_n = FNAME(calculate)(c, maximum_iterations, escape_radius2);
    ↪ putchar(final_n);
    ↪
    ↪
    ↪
    ↪
185 void FNAME(render_colour)(int width, int height, complex FTYPE center, FTYPE radius, int maximum_iterations);
+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
190 index 0000000..ba89d16
--- /dev/null
+++ b/render.c
@@ -0,0 +1,35 @@
195 #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) {
215 +     float_type = atoi(argv[4]);
+ }
+ switch (float_type) {
+     case 0:
+         renderf(width, height, center, radius, maximum_iterations, ↪
    ↪     escape_radius2);
+         break;
220 +     case 1:
+         renderl(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
5 @@ -6,6 +6,35 @@

-void colour(int *r, int *g, int *b, int final_n) {
- *r = final_n >> 16;
- *g = final_n >> 8;
10 - *b = final_n;
+struct pixel {
+ int final_n;
+};
+
15 +struct image {
+ int width;
+ int height;
+ struct pixel *pixels;
+};
20 +
+struct image *image_new(int width, int height) {
+ struct image *img = calloc(1, sizeof(*img));
+ img->width = width;
+ img->height = height;
25 + img->pixels = calloc(width * height, sizeof(*img->pixels));
+ return img;

```

```

+}
+
+void image_free(struct image *img) {
30 + free(img->pixels);
+ free(img);
+}
+
+void image_write(struct image *img) {
35 + 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);
40 + putchar(n >> 8);
+ putchar(n );
+ }
+ }
}
}
45 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
50
-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) {
55 ↵
↵ 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);
60 int final_n = FNAME( calculate )( c, maximum_iterations, ↵
↵ escape_radius2 );
--- int r, g, b;
--- colour(&r, &g, &b, final_n);
--- putchar(r);
--- putchar(g);
65 --- 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);

```

```

75  switch (float_type) {
      case 0:
          ↵
+   ↵ renderf(width, height, center, radius, maximum_iterations, escape_radius2);
          ↵ renderf(img, center, radius, maximum_iterations, ↵
          ↵ escape_radius2);
          break;
80  case 1:
          ↵
+   ↵ render(width, height, center, radius, maximum_iterations, escape_radius2);
          ↵ render (img, center, radius, maximum_iterations, ↵
          ↵ escape_radius2);
          break;
85  case 2:
          ↵
+   ↵ renderl(width, height, center, radius, maximum_iterations, escape_radius2);
          ↵ renderl(img, center, radius, maximum_iterations, ↵
          ↵ escape_radius2);
          break;
      }
+   image_write(img);
90  +   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 parallelism 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- $\{\text{exp}\}$  2 >  $\{\text{exp}\}$ -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
5 @@ -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  $\mathcal{Z}$ 
   ↘ render render.c -lm
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
10 index 8c80a61..e160302 100644
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -25,2 +25,3 @@ int FNAME( calculate )( complex FTYPE c, int  $\mathcal{Z}$ 
   ↘ maximum_iterations, FTYPE escape_radiu
   void FNAME( render )( struct image *img, complex FTYPE center, FTYPE $\mathcal{Z}$ 
   ↘ radius, int maximum_iterations, FTYPE escape_radius2 ) {
15 + #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):  $\arg(x + iy) = \text{atan}(y/x)$ . Angles are unique up to multiples of a whole turn ( $2\pi$ ), so maths conventionalized the principal 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 sources information 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
5 @@ -2,2 +2,4 @@
   #include <stdio.h>
+ #include <stdlib.h>
+ #include <string.h>

10 @@ -30,7 +32,6 @@ void hsv2rgb(float h, float s, float v, float *r
    ↪ 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 z
    ↪ final_z_arg) {
    float hue = final_n / 64.0f;
15 - 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;
20 float red, grn, blu;
@@ -42,22 +43,51 @@ void colour(int *r, int *g, int *b, int z
    ↪ final_n) {

- int main() {
- int width = 0;
```

```

25 | 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) {
30 | for (int i = 0; i < width; ++i) {
int r = getchar();
int g = getchar();
int b = getchar();
int final_n = (r << 16) | (g << 8) | b;
35 | colour(&r, &g, &b, final_n);
putchar(r);
putchar(g);
putchar(b);
}
40 | }
+int main(int argc, char **argv) {
+ char *stem = "out";
+ if (argc > 1) {
+ stem = argv[1];
45 | + }
+ int length = strlen(stem) + 100;
+
+ char *fname_n = calloc(length, 1);
+ snprintf(fname_n, length, "%s_n.ppm", stem);
50 | + 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; }
55 | + 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");
60 | + 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; }
65 | +
+ if (width_n != width_a) { return 1; }
+ if (height_n != height_a) { return 1; }
+ int width = width_n;
+ int height = height_n;
70 | +
+ 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);
75 | + 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);
80 +     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);
85 +     putchar(g);
+     putchar(b);
    }
  }
+
90  return 0;
diff --git a/mandelbrot.h b/mandelbrot.h
index faf47f2..cd6a6a9 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
95 @@ -4,3 +4,6 @@
    #include <complex.h>
+ #include <math.h>
    #include <stdio.h>
+ #include <stdlib.h>
100 + #include <string.h>

@@ -8,2 +11,3 @@ struct pixel {
    int final_n;
+ complex float final_z;
105 };
@@ -16,27 +20,2 @@ struct image {

- struct image *image_new(int width, int height) {
-     struct image *img = calloc(1, sizeof(*img));
110 -     img->width = width;
-     img->height = height;
-     img->pixels = calloc(width * height, sizeof(*img->pixels));
-     return img;
- }
115 -
- void image_free(struct image *img) {
-     free(img->pixels);
-     free(img);
- }
120 -
- 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) {
125 -             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) {
135 +struct image *image_new(int width, int height) {
+ struct image *img = calloc(1, sizeof(*img));
+ img->width = width;
+ img->height = height;
140 + img->pixels = calloc(width * height, sizeof(*img->pixels));
+ return img;
+}
+
+void image_free(struct image *img) {
145 + free(img->pixels);
+ free(img);
+}
+
+void image_save(struct image *img, char *filestem) {
150 + int length = strlen(filestem) + 100;
+ char *filename = calloc(length, 1);
+ {
+   snprintf(filename, length, "%s_n.ppm", filestem);
+   FILE *f = fopen(filename, "wb");
155 +   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);
160 +       fputc(n >> 8, f);
+       fputc(n, f);
+     }
+   }
+   fclose(f);
165 + }
+ {
+   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);
170 +   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;
175 +       fputc(n >> 16, f);
+       fputc(n >> 8, f);
+       fputc(n, f);
+     }
+   }
+ }

```

```

180 +   fclose(f);
+   }
+   free(filename);
+}
+
185 #endif
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index e160302..594fc32 100644
--- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
190 @@ -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
195 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;
200   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) {
205 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
210 }
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);
215 ↵
+   ↵ 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
220 index 58f4d0e..b5828c9 100644
--- a/render.c
+++ b/render.c
@@ -22,2 +22,6 @@ int main(int argc, char **argv) {

```

```

    }
225 + char *stem = "out";
+   if (argc > 5) {
+     stem = argv[5];
+   }
    struct image *img = image_new(width, height);
230 @@ -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 nonexistent >out.ppm
Segmentation fault
$ ./render 0 0 2 1 directory/nonexistent
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
5 @@ -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) {
10 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) {
15 +   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 *z
    ↪ filestem) {
    char *filename = calloc(length, 1);
25  ---{
    ---snprintf(filename, length, "%s_n.ppm", filestem);
    ---FILE *f = fopen(filename, "wb");
    +
    + snprintf(filename, length, "%s_n.ppm", filestem);
30  + 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 *z
    ↪ filestem) {
    }
35  ---{
    ---snprintf(filename, length, "%s_z_arg.ppm", filestem);
    ---FILE *f = fopen(filename, "wb");
    +
    + snprintf(filename, length, "%s_z_arg.ppm", filestem);
40  + 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 *z
    ↪ filestem) {
    }
45  +
    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/nonexistent
ERROR image save failed
$

```

```

diff --git a/mandelbrot.h b/mandelbrot.h
index 3bf4350..33c6c84 100644
--- a/mandelbrot.h
+++ b/mandelbrot.h
5  @@ -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) {
10  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) {

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

    snprintf(filename, length, "%s_n.ppm", filestem);
    FILE *f = fopen(filename, "wb");
25  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;
30              fputc(n >> 16, f);
            fputc(n >> 8, f);
            fputc(n
+        snprintf(filename, length, "%s_n.ppm", filestem);
+        FILE *f = fopen(filename, "wb");
35  +        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;
40  +                    fputc(n >> 16, f);
+                    fputc(n >> 8, f);
+                    fputc(n
+                }
+            }
45  +        fclose(f);
+    } else {
+        retval = 1;
+    }
    fclose(f);
50  }

    snprintf(filename, length, "%s_z_arg.ppm", filestem);
    f = fopen(filename, "wb");
    if (f) {
55          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);

```

```

60 | int n = (1 << 24) * t;
fputc(n >> 16, f);
fputc(n >> 8, f);
fputc(n, f);
+   snprintf(filename, length, "%s_z_arg.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].z
    ↪ 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);
75 | +       fputc(n, f);
+     }
+   }
+   fclose(f);
+ } else {
80 | +   retval = 1;
+ }
fclose(f);
+   free(filename);
85 | + } else {
+   retval = 1;
+ }
free(filename);
90 | + return retval;
+ }
diff --git a/render.c b/render.c
index b5828c9..a003d5e 100644
a/render.c
95 | +++ b/render.c
@@ -7,2 +7,3 @@
+ int main(int argc, char **argv) {
+ int retval = 1;
+ int width = 1280;
100 | @@ -27,16 +28,25 @@ int main(int argc, char **argv) {
+ struct image *img = image_new(width, height);
switch (float_type) {
case 0:
↪
105 | ↪ renderf(img, center, radius, maximum_iterations, escape_radius2);
break;
case 1:
↪
↪ render(img, center, radius, maximum_iterations, escape_radius2);
break;
case 2:

```

```

110 | renderl(img, center, radius, maximum_iterations, escape_radius2);
      | break;
+   | if (img) {
+   |     switch (float_type) {
+   |       case 0:
115 |         renderf(img, center, radius, maximum_iterations,
      |         escape_radius2);
+   |         break;
+   |       case 1:
+   |         render (img, center, radius, maximum_iterations,
      |         escape_radius2);
+   |         break;
120 |       case 2:
+   |         renderl(img, center, radius, maximum_iterations,
      |         escape_radius2);
+   |         break;
+   |     }
+   |     int err = image_save(img, stem);
125 |     image_free(img);
+   |     if (! err) {
+   |       retval = 0;
+   |     } else {
+   |       fprintf(stderr, "ERROR_image_save_failed\n");
130 |     }
+   |   } else {
+   |     fprintf(stderr, "ERROR_image_new_failed\n");
+   |   }
image_save(img, stem);
135 | 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 suffices, 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 %\
    -1.26031888962837738806438608314184 %\
    0.38347214027113837724841765113596 %\
    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
5 @@ -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) {
10 }
+ char *stem = "out";
    if (argc > 4) {
--- float_type = atoi(argv[4]);
+ stem = argv[4];
15 }
--- char *stem = "out";
--- if (argc > 5) {
--- stem = argv[5];
+ int float_type = 1;
20 + long double pixel_spacing = radius / (height / 2.0);
+ int pixel_spacing_bits = -log2l(pixel_spacing);
+ if (pixel_spacing_bits > 50) {

```



```

+   float_type = 2;
+   }
25 +   if (pixel_spacing_bits > 60) {
+       fprintf(stderr, "WARNING: insufficient precision (need %d bits ↵
        ↵ )\n", pixel_spacing_bits);
+   }
+
+   struct image *img = image_new(width, height);
30 @@ -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) {
35 +       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) {
40 +       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:  $|z_{n-1}| \leq R$ . Combining with the definition of the iteration,  $z_n = z_{n-1}^2 + c$ , and using the triangle inequality  $|a + b| \leq |a| + |b|$ , we get:

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

If we make  $R$  bigger, so that  $|c|$  fades into insignificance, we get the approximation  $R < |z_n| \leq R^2$ , and taking base- $R$  logarithms we get

$$1 < \log|z_n| / \log R \leq 2$$

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 glitches.

```

diff --git a/colour.c b/colour.c
index d6e0acf..9d3acdf 100644
--- a/colour.c
+++ b/colour.c
5 @@ -32,5 +32,5 @@ void hsv2rgb(float h, float s, float v, float *v
    ↪ 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;
10 --- float sat = 0.75f;
+ float sat = final_z_abs;
+ float val = final_z_arg;
@@ -60,2 +60,12 @@ int main(int argc, char **argv) {

15 + 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%d255", &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\n%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(cargf(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
index 1ed52aa..039253e 100644
90 --- 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 %\
    0.3060959246356990742873 \
    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
5 | @@ -33,5 +33,9 @@ void hsv2rgb(float h, float s, float v, float *r
    | ↪ red, float *grn, float *blu) {
    | void colour(int *r, int *g, int *b, int final_n, float z
    | ↪ final_z_abs, float final_z_arg) {

```

```

float hue = final_n / 64.0f;
float sat = final_z_abs;
float val = final_z_arg;
10 + 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;
15 +     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  $[0,1] \times [0,1]$ . 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
5 @@ -33,4 +33,8 @@ void hsv2rgb(float h, float s, float v, float *r
    ↪ red, float *grn, float *blu) {
    void colour(int *r, int *g, int *b, int final_n, float v
    ↪ 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 + v
    ↪ 1.0f);
10 + 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;
15 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
5 @@ -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 =
10 --- 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:

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

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
5 @@ -32,3 +32,3 @@ void hsv2rgb(float h, float s, float v, float *v
    ↪ 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 de
    ↪ final_z_abs, float final_z_arg, float de) {
    float continuous_escape_time = final_n - log2f(final_z_abs + v
    ↪ 1.0f);
10 @@ -41,6 +41,6 @@ void colour(int *r, int *g, int *b, int final_n, v
    ↪ 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) {
15         sat = 0.0f;
        val = 0.0f;
+       val = 1.0f;
    }
@@ -90,6 +90,18 @@ int main(int argc, char **argv) {
20
+   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; }
25 +   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; }
30 +
+   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; }
35 +   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);
40 +   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) {
45 +   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);
50 +   putchar(red);
diff --git a/mandelbrot.h b/mandelbrot.h
index a8be2fa..1b4ad9f 100644
a/mandelbrot.h
+++ b/mandelbrot.h
55 @@ -12,2 +12,3 @@ struct pixel {
+   complex float final_z;
+   complex float final_dc;
    };
@@ -53,3 +54,3 @@ void image_free(struct image *img) {
60
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 *z
    ↪ filestem, float escape_radius) {
65 +     snprintf(filename, length, "%s_de.ppm", filestem);
+     f = fopen(filename, "wb");
+     if (f) {
+         fprintf(f, "P6\n%d_\n255\n", img->width, img->height);
70 +         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].z
    ↪ final_z;
+                 complex double dc = img->pixels[j * img->width + i].z
    ↪ final_dc;
+                 float de = 2 * cabs(z) * log(cabs(z)) / (cabs(dc) * z
    ↪ pixel_spacing);
75 +                 int n = ((1 << 24) - 1) * fminf(fmaxf(de / (1 << 12), z
    ↪ 0.0f), 1.0f);
+                 fputc(n >> 16, f);
+                 fputc(n >> 8, f);
+                 fputc(n, f);
+             }
80 +         }
+         fclose(f);
+     } else {
+         retval = 1;
+     }
85 +
+
+     free(filename);
diff --git a/mandelbrot_imp.c b/mandelbrot_imp.c
index 594fc32..7d83325 100644
90 --- a/mandelbrot_imp.c
+++ b/mandelbrot_imp.c
@@ -16,4 +16,7 @@ void FNAME(calculate)(struct pixel *out, complexz
    ↪ FTYPE c, int maximum_iterations
    out->final_z = 0;
+ out->final_dc = 0;
95 + 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;
100 @@ -22,2 +25,3 @@ void FNAME(calculate)(struct pixel *out, complexz
    ↪ FTYPE c, int maximum_iterations
    out->final_z = z;
+     out->final_dc = dc;
+     break;
diff --git a/render.c b/render.c
105 index 039253e..1584e2a 100644
--- a/render.c
+++ b/render.c
@@ -52,3 +52,3 @@ int main(int argc, char **argv) {

```

```

    }
110 | 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
5 @@ -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;
10 + 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
5 @@ -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;
10 +};

```

```

+
+struct stream *stream_new(char *name) {
+ struct stream *s = calloc(1, sizeof(*s));
15 + if (! s) {
+     return 0;
+ }
+ s->file = fopen(name, "rb");
+ if (! s->file) {
20 +     free(s);
+     return 0;
+ }
+ if (2 != fscanf(s->file, "P6%d%d255", &s->width, &s->height)
+     ) {
+     fclose(s->file);
25 +     free(s);
+     return 0;
+ }
+ if (fgetc(s->file) != '\n') {
+     fclose(s->file);
30 +     free(s);
+     return 0;
+ }
+ return s;
+}
35 +
+void stream_free(struct stream *s) {
+     fclose(s->file);
+     free(s);
+}
40 +
+int stream_compatible(struct stream *s, struct stream *t) {
+     return s->width == t->width && s->height == t->height;
+}
+
45 +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;
50 +}
+
+struct stream *stream_new_for_stem(char *stem, char *suffix) {
+     int length = strlen(stem) + strlen(suffix) + 100;
+     char *name = calloc(length, 1);
55 +     snprintf(name, length, "%s_%s.ppm", stem, suffix);
+     struct stream *s = stream_new(name);
+     free(name);
+     return s;
+}
60 +
+int main(int argc, char **argv) {
+     int retval = 1;
+     char *stem = "out";

```

```

@@ -57,84 +112,38 @@ int main(int argc, char **argv) {
65     }
    int length = strlen(stem) + 100;

    char *fname_n = calloc(length, 1);
    snprintf(fname_n, length, "%s_n.ppm", stem);
70     FILE *f_n = fopen(fname_n, "rb");
    if (!f_n) { return 1; }
    free(fname_n);
    int width_n = 0;
    int height_n = 0;
75     ↵
    ↵ 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");
80 + 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)) {
85     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; }
90     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; }
95     + int width = s_n->width;
    + int height = s_n->height;
    + printf("P6\n%d\n%d\n255\n", width, height);
    + for (int j = 0; j < height; ++j) {
    +     for (int i = 0; i < width; ++i) {
100     +         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(blue);
    +     }
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);
155 |     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);
160 |     ↵
|     ↵ 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);

```

```
165 | putchar(red);  
    | putchar(grn);  
    | putchar(blu);  
    |     }  
    |     }  
+   | if (s_n) { stream_free(s_n); }  
+   | if (s_r) { stream_free(s_r); }  
170 | if (s_a) { stream_free(s_a); }  
+   | if (s_d) { stream_free(s_d); }  
  
    | fclose(f_n);  
    | fclose(f_a);  
175 | return 0;  
+   | return retval;  
    | }
```