ASGR Assignment 1 Solutions

Aaron Robotham

Load things:

```
library(magicaxis)
library(data.table)
library(foreach)
library(arrow)
```

```
##
## Attaching package: 'arrow'

## The following object is masked from 'package:utils':
##
##     timestamp
```

---

1) [5 marks total]

Here we are testing basic and advanced **R** skills. Do the following:

a) [1] Create a function **Rfunc1** that takes the vector inputs a and b and returns $\sqrt{a^{2b}}$. Show the output when a=21:30 and b=(21:20)/200.

```
##  [1] 1.376680 1.362204 1.389893 1.374109 1.402115 1.385152 1.413491 1.395455
##  [9] 1.424137 1.405116
```

b) [1] Create an infix function **skip** that can skip different numbers of elements without throwing an error, e.g. for 1:9 %skip% 3 we should get 1,4,7 without any warnings. Using this function show the results of:

```
## [1] 13 26 59 99
```

c) [3] Write a base **R** function **RMatMult** that does proper matrix multiplication for general matrices (not just square, but generic [n,m]x[m,n]). This should not use %*% and instead explicit loops. Write the same function now called **RcppMatMult** using **Rcpp** code.

Show the result of $A \times B$, where A =matrix(17:24,4,2) and B =matrix(15:8,2,4) using both functions.

Comment of the code speed using **RMatMult**, **RcppMatMult** and %*%.

```
##      [,1] [,2]
## [1,]   17   21
## [2,]   18   22
## [3,]   19   23
## [4,]   20   24

##      [,1] [,2] [,3] [,4]
## [1,]   15   13   11    9
## [2,]   14   12   10    8

##      [,1] [,2] [,3] [,4]
## [1,]  549  473  397  321
## [2,]  578  498  418  338
## [3,]  607  523  439  355
```

```
## [4,]  636  548  460  372
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  549  473  397  321
## [2,]  578  498  418  338
## [3,]  607  523  439  355
## [4,]  636  548  460  372
```

Explicit Rcpp version (using the namespace to tidy things up):

Using more sugar tidies things up even more.

```
##      [,1] [,2] [,3] [,4]
## [1,]  549  473  397  321
## [2,]  578  498  418  338
## [3,]  607  523  439  355
## [4,]  636  548  460  372
```

```
##    user  system elapsed
##   0.230   0.002   0.233
```

```
##    user  system elapsed
##   0.020   0.000   0.021
```

```
##    user  system elapsed
##   0.028   0.000   0.028
```

```
##    user  system elapsed
##   0.006   0.000   0.006
```

More elegantly we can use the **microbenchmark** package:

```
## Unit: nanoseconds
##                      expr   min    lq       mean median    uq     max neval
##      RMatMult(mat1, mat2) 17778 19584 23192.9153  20438 21448 4479568 10000
##  RcppMatMult1(mat1, mat2)  1565  1745  1900.1875   1823  1917   21818 10000
##  RcppMatMult2(mat1, mat2)  2390  2619  2808.7303   2710  2837   21747 10000
##             mat1 %*% mat2   386   453   523.1018    493   530   16806 10000
```

Seems the **R**'s built in **%*% **is fastest, followed by our** Rcpp **functions** RcppMatMult1 / RcppMatMult2**, then quite a bit slower is out native** R **function** RMatMult**.

2) [10 marks total]

Load the parquet format geo_heal_10.parquet data and read the associated PDF describing the data.

a) [5] What is the 25th most populous country as per the geo_heal_10.parquet data, and what is the area of this country in square kilometres? How does that compare to expected area, and why might it differ?

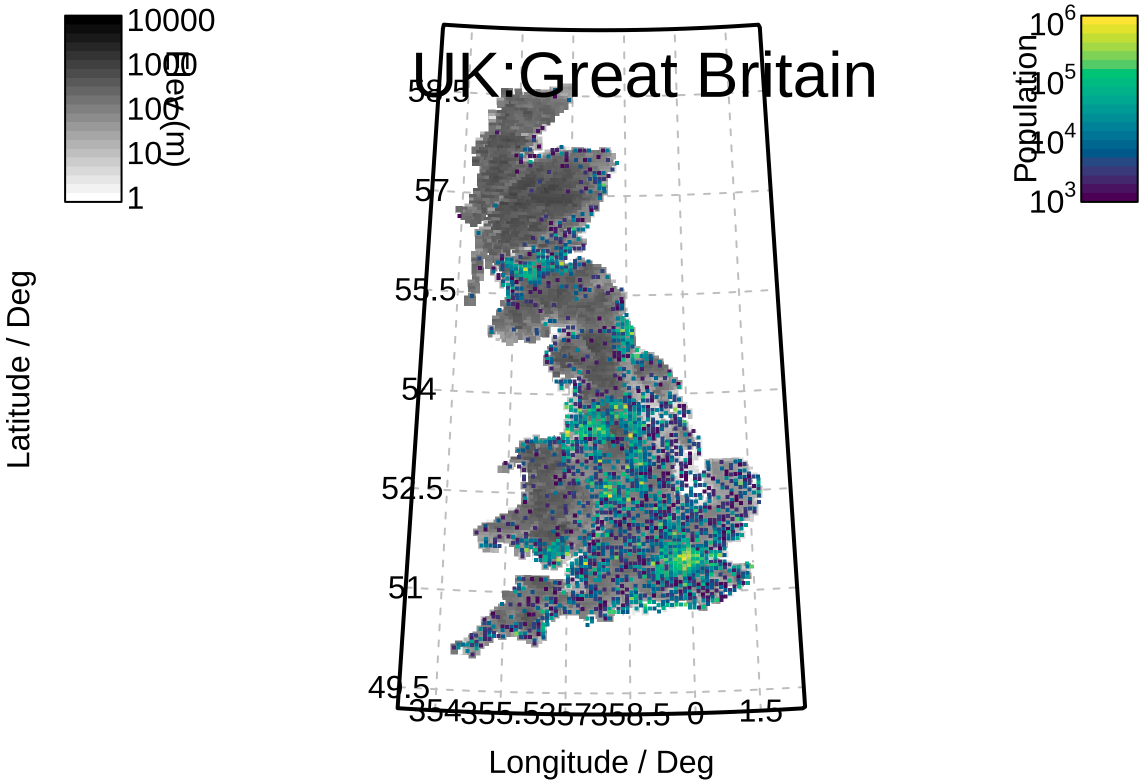
```
## Warning in `[.data.table`(geo_heal_10, , `:=`(country2, sub(":.*", "",
## country))): A shallow copy of this data.table was taken so that := can add or
## remove 1 columns by reference. At an earlier point, this data.table was copied
## by R (or was created manually using structure() or similar). Avoid names<- and
## attr<- which in R currently (and oddly) may copy the whole data.table. Use set*
## syntax instead to avoid copying: ?set, ?setnames and ?setattr. It's also not
## unusual for data.table-agnostic packages to produce tables affected by this
## issue. If this message doesn't help, please report your use case to the
## data.table issue tracker so the root cause can be fixed or this message
## improved.
```

```
##    country2       V1
##      <char>    <int>
## 1:   Canada 32386991
```

```
## [1] 9874030
```

Note in the above variants of country names (like ‘China:Hong Kong’) have been merged together. You will lose half a mark if you do not do this (e.g. ‘China’ has population of 629,853,393 because sub-regions are missed).

b) [5] Create a sensible map showing the terrain height of Great Britain (i.e. the main island of the UK). You can choose the z-scale, but sensible options might be grey or viridis.



Notes on map. Half will be lost for lack of units (degrees, metres), lack of z-scale bar, and linear projection (should be some sort of spherical projection ideally).

In the map shown here I’ve also added cells coloured by viridis for the population, but that is not necessary to get full marks.

---

3) [10 marks total]

You decide you want to start playing Poker, and by far the most popular game is Texas Hold ’Em. This uses a standard deck containing value cards 2-10JQKA and suits Clubs (c) Diamonds (d) Hearts (h) Spades (s) (note, in card notation “10” is written as “T” thus Ts is the 10 of spades and 8d is the eight of diamonds etc). Whilst the odds are easy to find online, we want to construct our own computer simulations to calculate the probability of different outcomes. To get marks for this exercise you must write your own simulation code to produce the results requested below yourself- simply stating the correct answer will get you **zero** marks.

In this variant of the game you try to form the best 5 card hand (look online for details of this, but e.g. https://en.wikipedia.org/wiki/Texas_hold_%27em#Hand_values) out of 2 private cards that only you can use, and 5 shared board cards that you and your opponents can use. Given this set-up, what is the probability (to 0.1% absolute accuracy, which will probably require of order 10^6 simulated games) that after seeing all 7 cards your best 5 card hand is:

a) [3] Any Flush (5 cards of the same suit, including Straight Flushes and Royal Flushes, e,g,: 2s4s7s9sJs)?

b) [3] Any Straight (cards forming a gap-less run of 5, including Straight Flushes and Royal Flushes, e.g.: 3s4h5d6c7s)? [Careful to treat Aces as both high and low since Ac2d3s4h5d is also a legal straight].

c) [4] A Full House (a three-of-a-kind and a pair, e.g.: 5d5h5sQdQc)? [Careful to check you have not made a better four-of-a-kind].

To prepare for this question we are going to encode a matrix of values of suits for easy processing:

```
##     val_Alo val_Ahi valname  suit suitname valsuitname
##       <num>   <int>  <char> <int>   <char>      <char>
##  1:       2       2       2     1        c          2c
##  2:       2       2       2     2        d          2d
##  3:       2       2       2     3        h          2h
##  4:       2       2       2     4        s          2s
##  5:       3       3       3     1        c          3c
##  6:       3       3       3     2        d          3d
##  7:       3       3       3     3        h          3h
##  8:       3       3       3     4        s          3s
##  9:       4       4       4     1        c          4c
## 10:       4       4       4     2        d          4d
```

a) To get the raw flush results:

To get the probability:

```
## [1] 0.0328
```

We expect the answer to be very near to 3.06%. Reasonable effort is [2] marks, plus or minus 1% is [2.5] marks, plus of minus 0.1% is [3] marks.

b) Straights are a bit trickier, and we need to be careful to find aces too. Here we will do it the more obvious way of sorting our unique value cards and looking for runs of 5 or greater.

To get the raw results:

To get the probability:

```
## [1] 0.0502
```

We expect the answer to be close to 4.65% (all straights, including straight flush and royal flush). Reasonable effort is [2] marks, plus or minus 1% is [2.5] marks, plus of minus 0.1% is [3] marks.

Other useful, compact and efficient functions for tackling this question are *rle* and also *tabulate*.

The *rle* approach gives the same answer:

```
## [1] 0.0502
```

There are clever ways to detect straights that do not involve sorting- try to think of one!

c) Here we just need to be careful to look for the better 4-of-a-kind.

To get the raw results:

To get the probability:

```
## [1] 0.0241
```

We expect the answer to be close to 2.60%. Reasonable effort is [3] marks, plus or minus 1% is [3.5] marks, plus of minus 0.1% is [4] marks.

---

4) [10 marks total]

Here you will need to code up efficient **Rcpp** routines that can check if a number is prime. Your **Rcpp** code *must* be included to get the solutions, since it is trivial to find the answers online (or using various **R** packages).

a) [3] Which of 40,001,407; 40,001,447; 40,001,467; 40,001,473 are prime? (you will lose a mark for each wrong answer, so do not guess!)

b) [5] How many primes have a value less than 200,000,000?

c) [2] What is the 5,000,000$^{\text{th}}$ prime number?

As a hint, if you code a sensible strategy using **Rcpp**, you should be able to answer all these questions in well under a minute (at least in terms of computer time).

a) For this we make an efficient **Rcpp** routine to check whether a specific number is prime:

```
## [1] 1
```

```
## [1] 1
```

```
## [1] 0
```

```
## [1] 1
```

b) For this we make an efficient **Rcpp** routine to check which numbers less than $Nlim$ are prime:

You need to be careful that you do not treat 1 as prime, which we see as a special case above.

Check the length for the desired solution:

```
## [1] 11078937
```

c) Assuming you saved the above sieve output, this is trivial:

```
## [1] 86028121
```

---

5) [15 marks total]

This question follows on from the introduction to the Mandelbrot set from the course material. To look at this in more detail we should re-write the core algorithm with **Rcpp** since using pure **R** will be far too slow. There are a few tricks to help you on your way:

- We can describe a complex number x as $x = A + Bi$ where A is the modulus of the real part and B is the modulus of the imaginary part, so $x^2 = (A^2 - B^2) + i(2AB)$. This means we do not need to explicitly use complex numbers computationall to find the result, we just have to make use of these identities.
- When doing these iterations, it turns out that if at any point $|x_n| > 2$ (or equally $A^2 + B^2 > 4$) then the solution will certainly diverge. These means if we hit this criterion we can stop the iterations and report that value of C as having a divergent solution.

a) [5] With this knowledge you should code a **Rcpp** function call *bounded* that takes the input A (real part of C) B (imaginary part of C) and N (number of iterations to make). The result should be NA if our distance from the complex origin never exceeds 2, and otherwise it should return the iteration number at which this stopping criterion is reached. To help you check the validity of your code, here are some expected results:

```
## [1] 6
```

```
## [1] 10
```

```
## [1] 38
```

```
## [1] 125
```

```
## [1] NA
```

```
## [1] 24
```

```
## [1] 11
```

```
## [1] NA
```

```
## [1] 36
```

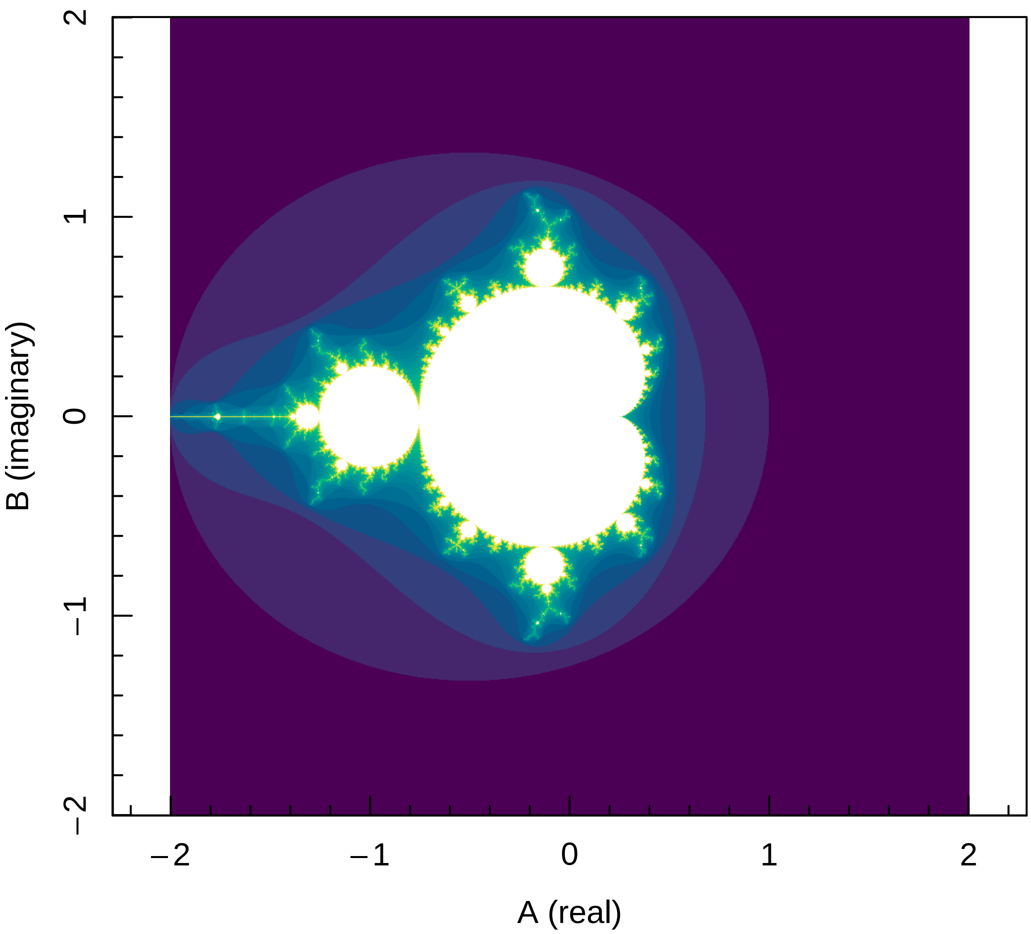
```
## [1] 91
```

b) [5] Getting more sophisticated, the next step is the encode a function where we compute convergence solutions on an arbitrary grid. This should be called *mandel_grid*, and should take inputs *real_lim* (two elements; lower and upper limits for the real component A of the complex number), *imag_lim* (two elements; lower and upper limits for imaginary component B of the complex number), *res* (how many samples to make between limits) and N (number of iterations to make).

Since speed was mentioned in the question, should be clear that full marks requires writing the above in **Rcpp**. Lose 2 marks if written in base **R** (so max 3).

To view the results we will create a simple plotting function that brings out some of the attractive features of the Mandelbrot set:

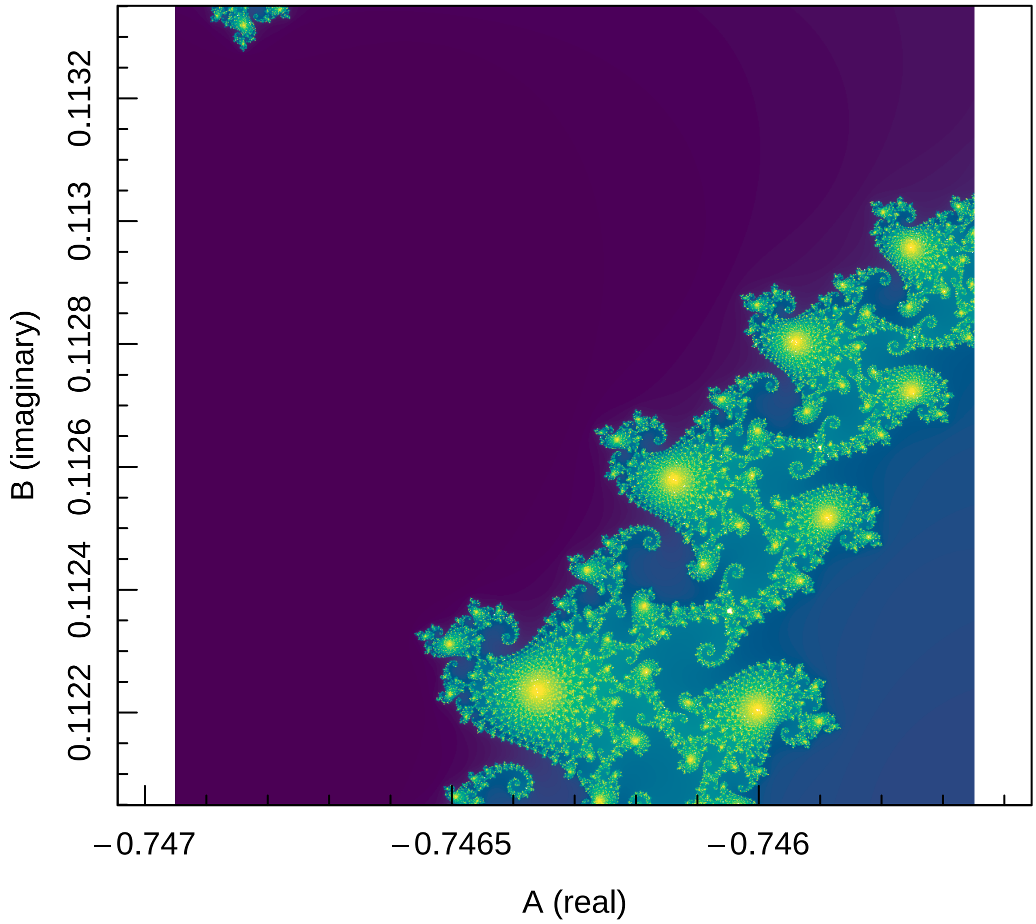
For reference here is what we should see if we now generate a default resolution matrix of solutions where the colour represents the iteration number, where white is NA (i.e. it has not met our divergent criterion in 1,000 iterations):



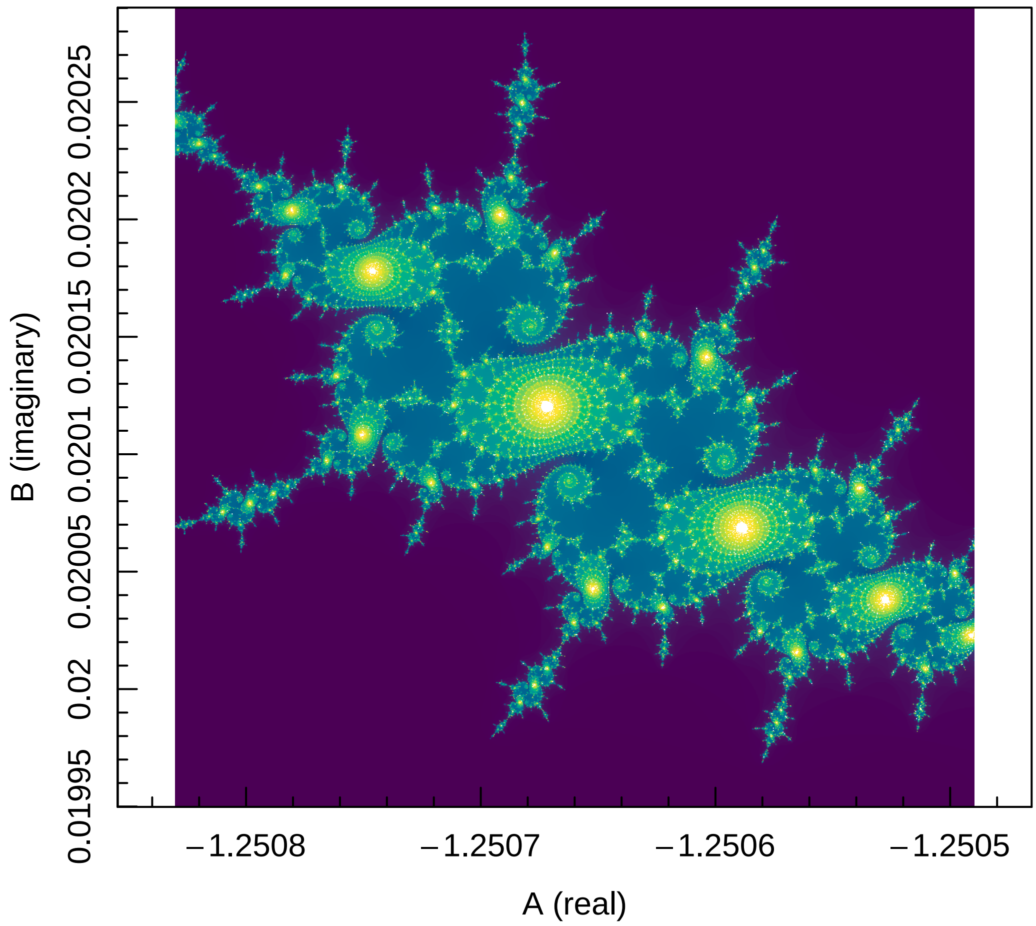
To give a guide of what sort of performance you should be expecting, the above only took 0.3 seconds to run on a single core. For the really adventurous, try to get your code working even faster than this, perhaps by using multiple threads etc. Good luck!

c) [5] If you have a look online you should be able to find list of attractive regions with suggested zoom levels. Re-create 5 of these yourself with the above code you have developed. These should be distinct in terms of the patterns shown.

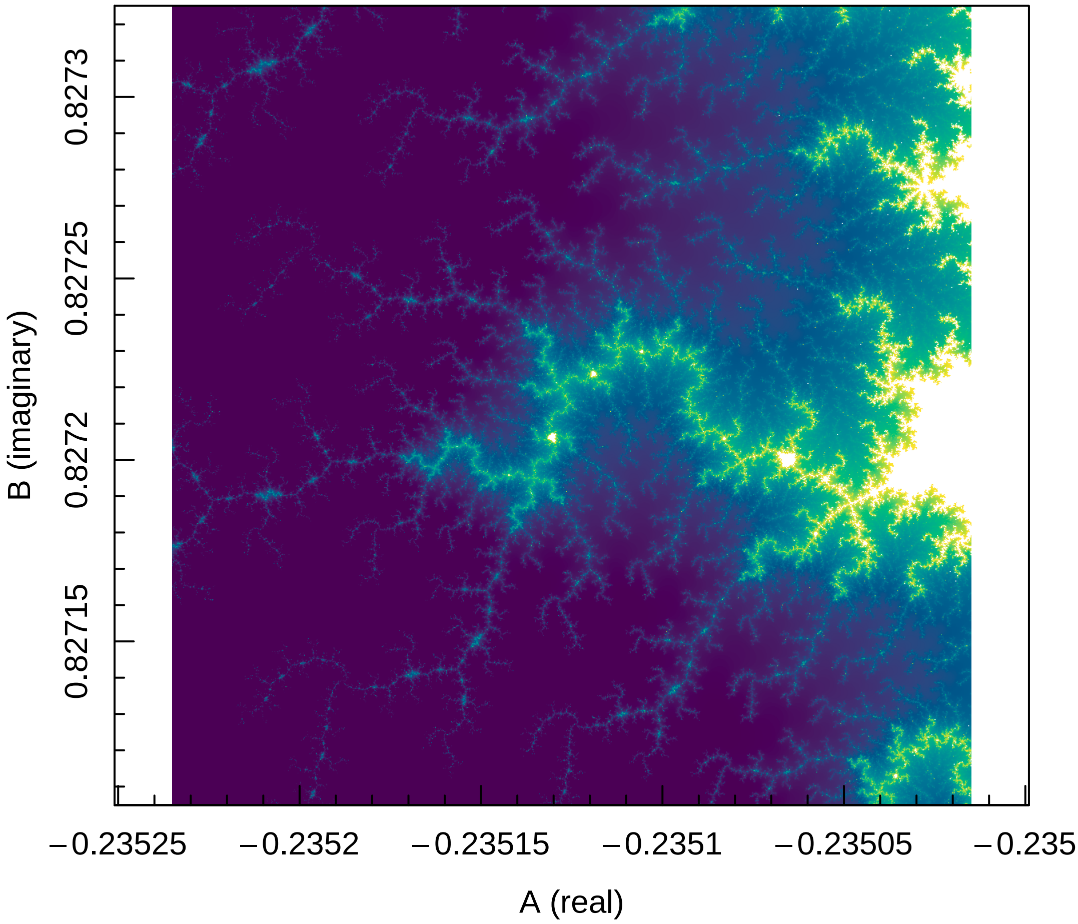
From http://www.cuug.ab.ca/dewara/mandelbrot/Mandelbrowser.html:

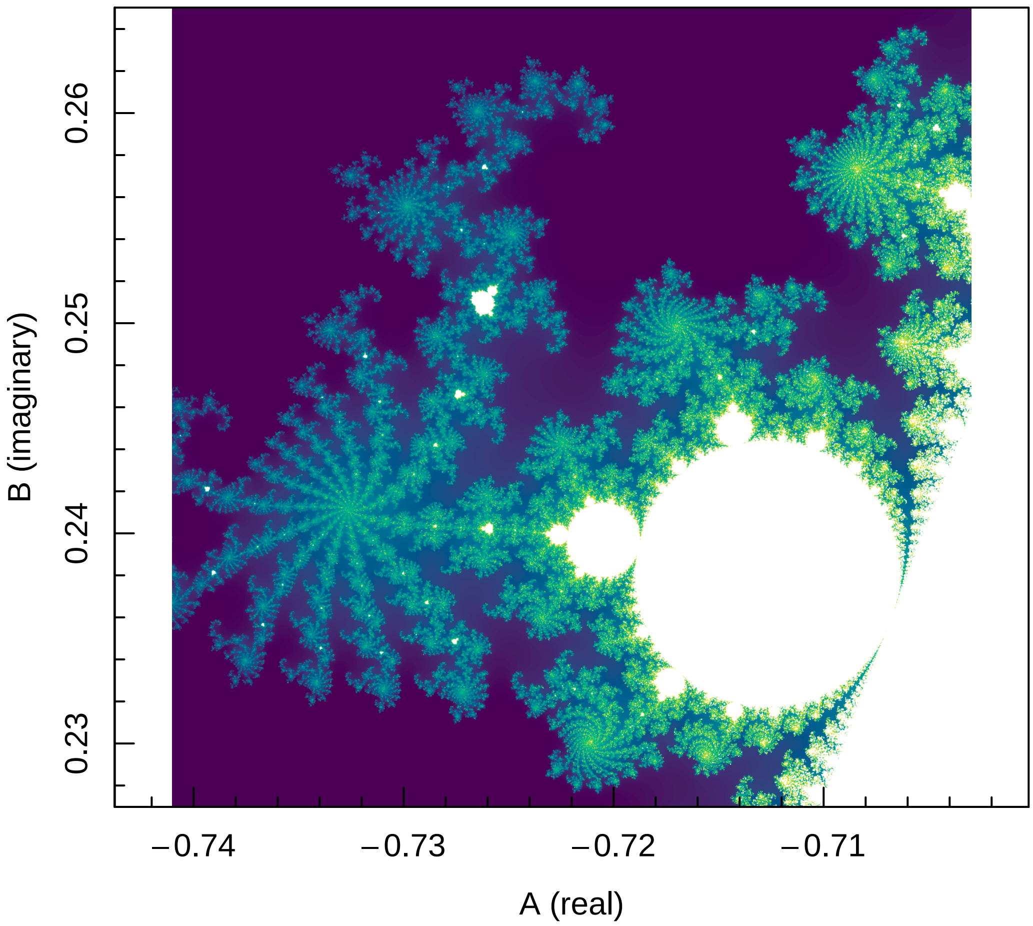


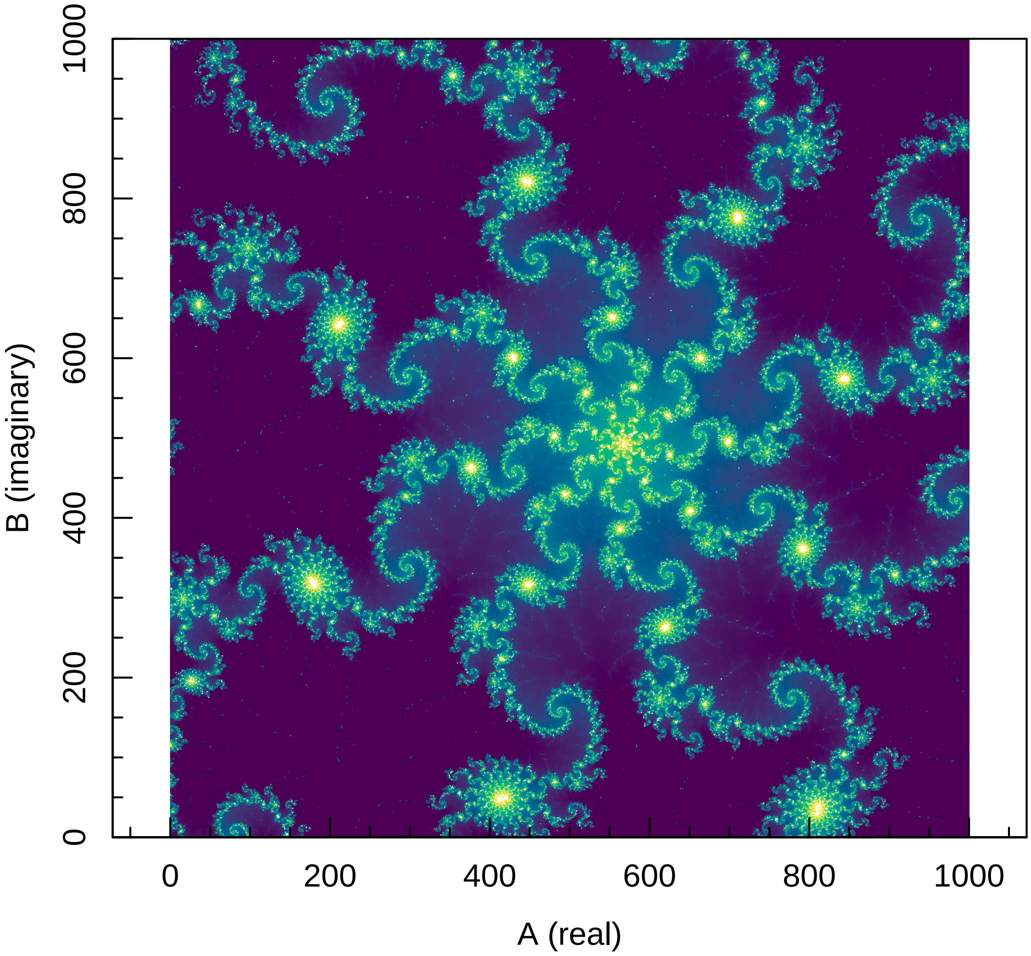
B (imaginary)
0.1122
0.1124
0.1126
0.1128
0.113
0.1132
−0.747
−0.7465
−0.746
A (real)



B (imaginary)
0.01995
0.02
0.02005
0.0201
0.02015
0.0202
0.02025
−1.2508
−1.2507
−1.2506
−1.2505
A (real)


0.8273
0.82725
0.8272
0.82715
B (imaginary)
−0.23525
−0.2352
−0.23515
−0.2351
−0.23505
−0.235
A (real)


0.26
0.25
0.24
0.23
B (imaginary)
−0.74
−0.73
−0.72
−0.71
A (real)



B (imaginary)
A (real)
0
200
400
600
800
1000
0
200
400
600
800
1000