Repa And A Tutorial

Marcus Lönnberg
marlon@student.chalmers.se

Karl Schmidt
karsch@student.chalmers.se

May 4, 2012

Note that this tutorial is for Repa 2.1.1.5 and Repa-io 2.2.0.1.

1 Repa and an introduction

Repa, which stands for REgular PArallel arrays, is a Haskell library that gives parallelised combinators for array operators. That sounds quite boring, but the gist of it is that Repa lets the user manipulate array data with free parallelism. This tutorial aims to give a (mercifully short) introduction to the library and then get into the meat of things with an extensive example.

Before we get to aforemention meat we need to cover some basics on arrays, in general and in Repa.

2 Repa and arrays, shapes & indexes

Matrices in the mathematical sense generally have some underlying dimension: $3 \times 5 \times 6$ for a 3-dimensional matrix of 90 cells with width 3, height 5 and depth 6, for instance. It would be quite practical if, when working with a matrix in a programming language, this information was expressed in the type of a matrix.

The shape type in Repa is an attempt to just that. The extent of and positions in a matrix are both represented by a type class Shape. The usual instance of Shape in Repa is a snoc list – the cons list's slightly backward cousin – where, for instance, a 3×2 matrix will have the shape `(Z :. 3 :. 2)`. When generating a matrix one provides its shape, its initial elements and generally some type information as not all types can be stored in a Repa array. For instance we might have a program that starts

```
import Data.Array.Repa as R
main = do
  let xs = fromList (Z :. 2 :. 2 :: DIM2) [0..(3::Int)]
```

Here we define `xs` to be a `2x2` array with `xs` = $\begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix}$

Apart from `fromList` there are also the generators `fromFunction` and `fromVector`.

Doing a lookup on an array in Repa looks very much like doing a lookup on a normal Data.Array. `xs!pos` gives you the element at `pos` for both types. In Data.Array you lookup an index and `pos` is of the `Ix` class used in the array. In Repa pos is instead an instance of the `Shape` type used in the array. We can expand the previous example with

```
import Data.Array.Repa as R
main = do
  let xs = fromList (Z :. 2 :. 2 :: DIM2) [0..(3::Int)]
  putStrLn $ show $ xs ! (Z :. 1 :. 1)
```

This prints `3`. Note that arrays in Repa start from 0; looking up `(Z :. 2 :. 2)` would give you a runtime error stating that the index is out of bounds. We can also try to run

```
import Data.Array.Repa as R
main = do
  let xs = fromList (Z :. 2 :. 2 :: DIM2) [0..(3::Int)]
  putStrLn $ show $ xs ! (Z :. -1 :. 5)
```

This prints `3` again! How? Repa's arrays are internally just 1-dimensional `Data.Vector`:s. Lookups are simply converted to the equivalent 1d vector lookup and as long as you're still within the bounds of the backing vector they're perfectly allowed. Be careful!

This all doesn't look terribly useful, but the meat of Repa lies in its automatically parallel traversal and combinator operations. These are a bit complicated to grok separately, so we'll write a decently complicated sample application to demonstrate.

3 Repa and a sequence of complex numbers

For any complex number c we can define the following sequence, which is probably familiar to some people

$$z_0 = 0$$
$$z_n = {z_{n-1}}^2 + c$$

This sequence will be unique for each c and its properties vary. For instance, with $c = 0$ we get a sequence that is always 0. With $c = 2$ we get $z_0 = 0, z_1 = 2, z_2 = 6, z_3 = 38, z_4 = 1446, \ldots$. With $c = i$ we get $z_0 = 0, z_1 = i, z_2 = -1 + i, z_3 = -i, z_4 = -1 + i, \ldots$. One sequence was constant, one grew very quickly and one quickly started to repeat. We'd quite like to see how these sequences behave for a few million different values of c.

This problem is naturally parallel: if $c_1 \neq c_2$ then calculations on the sequence for c_1 are not useful for and do not affect calculations for c_2 in any way. We can calculate them completely independently, which Repa is good at.

Another thing is that a million is a decently large number. We'll need is some way of visualizing our data. Repa kindly provides several ways of outputting array data as images, which is another thing we'll be looking into.

3.1 Some preliminaries

Before we can get to Repa we'll need some complex value arithmetic. Haskell has the `Data.Complex` module with everything we need – but, unfortunately, that type isn't storable in Repa arrays. Tuples, on the other hand, are. So, let's define basic complex value arithmetic using 2-tuples:

```
--Cannot use Data.Complex in Repa, so encode as tuple (Real, Imag)
type Complex = (Double,Double)

-- Multiplication
{-# INLINE mul #-}
mul :: Complex → Complex → Complex
mul (r1,i1) (r2,i2) = ( real, imag )
  where real = r1*r2 - i1*i2
        imag = i1*r2 + i2*r1

-- Addition
{-# INLINE add #-}
add :: Complex → Complex → Complex
add (r1,i1) (r2,i2) = ( r1+r2, i1+i2 )

-- Square
{-# INLINE sq #-}
sq :: Complex → Complex
sq (r,i) = ( r*r - i*i, 2*r*i )

-- Square of the 2-norm
{-# INLINE normSq #-}
normSq :: Complex → Double
normSq (r,i) = ( r*r + i*i )

-- 2-norm
{-# INLINE norm #-}
norm :: Complex → Double
norm = sqrt ∘ normSq
```

The inline pragmas are GHC compiler directives that force GHC to emit the code for the function where it is being called. Normally this is done

automatically through a code size heuristic that tries to balance the benefit of inlining against program size, however with Repa it is critical for performance that array element operations are always inlined and using the pragma is a good habit to have.

3.2 Generating and displaying data

Next, we'd like to get some of these complex numbers into a Repa array. A good, simple idea seems to be generating a grid of values on some axis-aligned rectangular region. In Repa this would take the form of an `Array DIM2 Complex`, that is an array with a 2D shape storing instances of our type `Complex`.

A function that does so is

```
-- Generate a region of the complex plane as a Repa array.
-- Takes array dimension and bounding values for the region.
complexPlaneSegment :: DIM2 → (Double,Double) → (Double,Double) →
                       Array DIM2 Complex
complexPlaneSegment (Z :. x :. y) (rmin,rmax) (imin,imax) =
  fromList (Z :. x :. y) ris
  where rstep = (rmax-rmin) / (fromIntegral x - 1)
        istep = (imax-imin) / (fromIntegral y - 1)
        reals = take x $ iterate (+rstep) rmin
        imags = take y $ iterate (+istep) imin
        ris   = [(r,i) | r ← reals, i ← imags]
```

Here `complexPlaneSegment` takes 3 arguments: a 2D Shape specifying the extent of the grid, for instance `(Z :. 640 :. 480)` for 640 points in the x direction and 480 in y. The other two are bounding values for the real and imaginary parts, respectively. Generating the array data is now a matter of generating the set of discrete `reals` and `imags` that are on the grid lines and columns and combining them with a list comprehension to pairs – our `Complex` type.

To make sure this worked alright we'd like to see what our data looks like, and not as a printout of numbers in the terminal. A very handy function here is `writeMatrixToGreyscaleBMP :: FilePath -> Array DIM2 a -> IO ()` from the module `Data.Array.Repa.IO.BMP`. This takes a file path and any Repa 2D array of a (fractional) numeric type and outputs a greyscale BMP. The data is automatically scaled so that 0 (and lower) is black and the largest number in the array is white. Unfortunately, we don't have a fractional numeric type – we have the `Complex` tuple.

However, we do have the `norm :: Complex -> Double` function to convert our array values to something useful! To do this sort of batch conversion Repa provides a `map :: (a -> b) -> Array sh a -> Array sh b` function for arrays that simply runs its first argument on every element of its second argument. This function, like many others, is automatically parallel

– perhaps not very useful here but there are more of these coming. Let's try to combine what we have.

```
import Prelude as P
import Data.Array.Repa        as R
import Data.Array.Repa.IO.BMP as R

-- Put complexPlaneSegment and the complex math stuff here!

main = do
  let cs    = complexPlaneSegment (Z :. 100 :. 100) (-2,2) (-2,2)
  let norms = R.map norm cs
  writeMatrixToGreyscaleBMP ("repa_and_my_first_image.bmp") norms
```

Look in your working folder and you should have a lovely little .bmp that looks just like fig. 1! Bask in how the distance from the origin actually increases with the distance from the origin!

Note also that the Repa map-function is *not* the same as that from the prelude. We needed to specify which map we meant to use. This is the case for several functions in Repa and why the package is usually imported under some handy abbreviated name or the conflicting functions from the Prelude hidden.

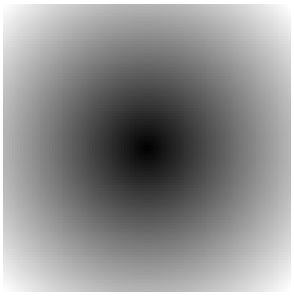


Figure 1: Your bitmap

3.3 Generating and visualizing the sequence

Alright, we have a bunch of data representing complex numbers. Time to generate our numeric sequence! Recall that the update rule was

$$z_0 = 0$$
$$z_n = z_{n-1}^2 + c$$

meaning that to get the next step we need the preceding step and the constant c. In Repa terms, to generate the array representing the next step we need the array for the previous step and an array of c:s. For the latter we have a pretty good starting point – an axis-aligned grid from `complexPlaneSegment`. We can also note that since $z_0^2 = 0^2 = 0$ it is always the case that $z_1 = c$. Since the z_0s aren't very interesting we can use our grid as the first element of the sequence and the constant parameter.

So, how do we generate the next array? We would like to essentially say `z_nplus1s = nextIteration cs z_ns` for suitable arrays `z_ns` and `cs`, which certainly looks promising, but to do this we need some operation that

combines two arrays into one new array. Repa actually provides a number of these and we can make do with the simplest one: `zipWith`. The Repa version has type `zipWith :: (a -> b -> c) -> Array sh a -> Array sh b -> Array sh c` – it takes a function that combines elements and uses it to combine two arrays. It is very similar to the one for lists but it is *not* the same function and you need to specify which one you mean. Here our update function will simply be the update rule for our sequence, so lets encode that.

```
-- The update rule for our Complex sequence
-- Again, we want this to be used in parallel so force inlining!
{-# INLINE update #-}
update :: Complex → Complex → Complex
update c z_n = sq z_n `add` c

-- Generating a new iteration
nextIteration :: Array DIM2 Complex →
                 Array DIM2 Complex →
                 Array DIM2 Complex
nextIteration cs z_ns = force2 $ R.zipWith update cs z_ns
```

Admirably straight-forward, however we added this strange `force2` function in there. `force2`, which is a version of it's more general brother `force` for 2-dimensional arrays, is essentially the command that tells Repa to chunk and evaluate this thing now. Before you put in force, nothing parallel happens. Note that this is *not* forcing strictness – that's another matter and we'll get to it later – but simply a that when the `force` expression is evaluated there is to be parallelism. Internally this is handled as a coercion from a "Delayed" array type to a "Manifest" array type, which is the sort of detail you shouldn't need to worry about: feel free to treat `force` as a magic keyword to make zipWith, map and other full-array operations go faster.

Now then, let's try to iterate a couple of times and see what we get

```
import Prelude as P
import Data.Array.Repa        as R
import Data.Array.Repa.IO.BMP as R

-- All our helper functions go here!

main = do
  -- Create an initial grid
  let cs    = complexPlaneSegment (Z :. 200 :. 200) (-2,2) (-2,2)
  -- Iteratively generate a bunch of number sequence arrays
  -- Take the last one
  let iter  = 4
  let zs    = last $ take iter $ iterate (nextIteration cs) cs
  let norms = force2 $ R.map norm zs
```

```
  -- Write to file
  writeMatrixToGreyscaleBMP
    ("iteration_" P.++ show iter P.++ ".bmp")
    norms
```

And the end result is ...

Well, not very informative. What happened? Well, as noted initially our sequence will grow very, very fast for some numbers. Detail anywhere else just goes poof; indeed most results will become `NaN` in short order. We want to clamp the display to some maximum threshold on the norm. As we cheat with foreknowledge, we know that any sequence that hits a member with a norm larger than 2 will eventually diverge to infinity, so let's go with that for our clamp.

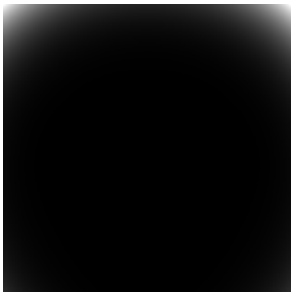
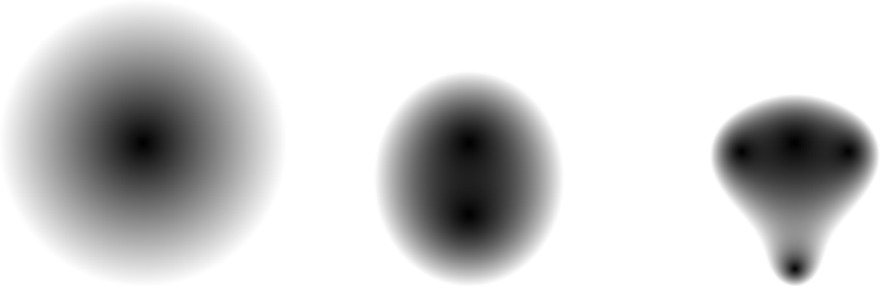


Figure 2: Bitmap #2!

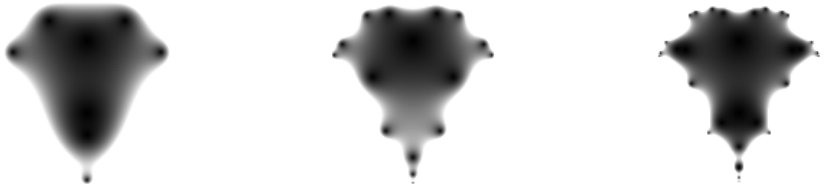
Also, generating one image at a time was annoying, so let's try to batch that.

```
import Prelude as P
import Data.Array.Repa        as R
import Data.Array.Repa.IO.BMP as R
import Control.Monad

-- Clamp a value to an upper bound
{-# INLINE clamp #-}
clamp :: Double → Double → Double
clamp thresh x = if abs x < thresh then x else thresh

main = do
  -- Create an initial grid with the same old functions
  let cs    = complexPlaneSegment (Z :. 300 :. 200) (-3,3) (-2,2)

  -- Iteratively generate number sequence arrays as before
  let iter   = 12
  let thresh = 2
  let zss    = take iter $ iterate (nextIteration cs) cs

  -- Normalize _and clamp_
  let normss = P.map ( force2 ∘ R.map (clamp thresh ∘ norm) ) zss

  -- Write all to file
  forM_ [0..length normss-1] $
    λi → writeMatrixToGreyscaleBMP
      ("iteration_" P.++ show i P.++ ".bmp")
      (normss!!i)
```

This program will do a sequence of 12 iterations and we get the images in figure 3

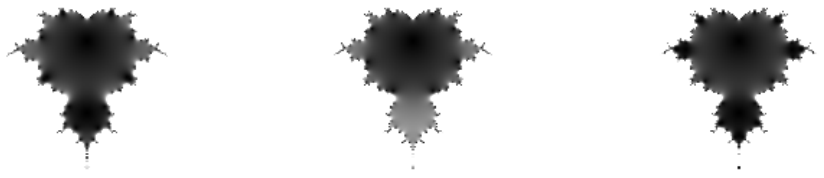


(a) iteration 0 (b) iteration 1 (c) iteration 2



(d) iteration 3 (e) iteration 4 (f) iteration 5

(g) iteration 6 (h) iteration 7 (i) iteration 8



(j) iteration 9 (k) iteration 10 (l) iteration 11

Figure 3: Norms in the first 12 iterations, whiteout threshold of 2

This is, and we do not use this word lightly, pretty boss. We note that there is a (or possibly several – whether or not this is a connected set is an open question) region of starting values that doesn't diverge to infinity and that region looks ... really weird. The region is a fractal, called the Mandelbrot Set after Benoît Mandelbrot who was the first to find it (and render it, in the late 70s. He did not use Repa).

It's also oriented oddly, if you know what this set usually looks like. The real axis is vertically aligned here and the imaginary is horizontal. Whoops, turns out `writeMatrixToGreyscaleBMP` doesn't agree with our matrix orientation. Fortunately there is a `transpose` function in Repa for 2D matrices, so we can fix this. We could do so at any point in the pipeline from generating the data on up, but one approach is to keep the data model as is and just change the write to file code from

```
  -- Write all to file
  forM_ [0..length normss-1] $
    λi → writeMatrixToGreyscaleBMP
      ("iteration_" P.++ show i P.++ ".bmp")
      (normss!!i)
```

to

```
  -- Write to file
  forM_ [0..length normss-1] $
    λi → writeMatrixToGreyscaleBMP
      ("iteration_" P.++ show i P.++ ".bmp")
      (transpose (normss!!i) )
```

and now the axes are swapped on drawing. Crisis averted.

Let's also take the opportunity to do some code cleanup. We've coded the image size and region as magic constants, but of course we have access to the usual Haskell tools for taking command line arguments. A slightly more flexible version of the main function might be

```
import System.Environment

main = do
  -- get command line argument list
  args ← getArgs

  -- pic size, iters
  let [xsize, ysize, iter]       = P.map read $ take 3 args

  -- threshold, rendering area and zoom factor
  let [thresh, rmid, imid, zoom] = P.map read $ drop 3 args

  -- Sim parameters
  let aspectRatio        =
        fromIntegral xsize / fromIntegral ysize
```

```
let dim                 =
      (Z :. xsize :. ysize :: DIM2)
let rRange              =
      (-aspectRatio/zoom + rmid, aspectRatio/zoom + rmid)
let iRange              =
      (-1/zoom + imid, 1/zoom + imid)

-- Create an initial grid
let cs    = complexPlaneSegment dim rRange iRange

-- Iteratively generate a bunch number sequence arrays
let zss    = take iter $ iterate (nextIteration cs) cs
let normss = P.map ( force2 ∘ R.map (clamp thresh ∘ norm) ) zss

-- Write to file
forM_ [0..length normss-1] $
  λi → writeMatrixToGreyscaleBMP
    ("iteration_" P.++ show i P.++ ".bmp")
    (transpose (normss!!i) )
```

With a good choice of parameters this can preduce in a figure like fig. 4.

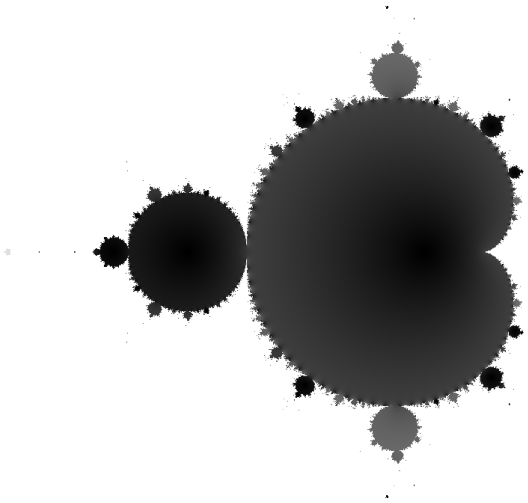


Figure 4: Result of call "`./mandelbrot 1280 960 100 2 -0.5 0 0.66 +RTS -N`". 100th iteration

3.4 The standard visualization

Often when you see an image of the Mandelbrot Set it tends to not have any detail inside the set, just on the outside region. Why? Well, while the interior sequence has plenty of interesting detail one might also wish to visualize just how quickly the exterior points diverge to infinity. In order to do this the usual measure is simply how many iterations it takes for the exterior values to reach our threshold – this is called the *escape time* algorithm. How do we implement it in Repa?

It is not sufficient to just look at the norm at iteration i. We can tell if a sequence has escaped by then but not when. To get the full detail we need to iterate and accumulate the results from all the arrays we've generated. Fortunately we don't need to do anything more complicated than zip the escape time result from the previous iteration and the current z_ns using a new update function.

```
-- The escape time function for a certain threshold and iteration
{-# INLINE escapeTime #-}
escapeTime :: Double → Int → Double → Complex → Double
escapeTime thresh iter i z =
  -- If already escaped, change nothing
  if i /= -1 then i
  -- Else if escaped now, store escape time
  else if normSq z > thresh*thresh then fromIntegral n
  -- Else not escaped
  else -1

-- Generating a new iteration of escape times from
-- previous iteration and current orbits
nextEscape :: Double → Int →
              Array DIM2 Complex →
              Array DIM2 Double →
              Array DIM2 Double
nextEscape thresh iter z_ns is = deepSeqArray is $
  force2 $ R.zipWith (escapeTime thresh iter) z_ns is
```

So, here we've added a scary `deepSeqArray` function. This is how we cure lazyness. The function simply forces the first argument to be evaluated immediately. Here this is of limited use since we tend to keep all iterations in memory, but in a sensible application this would make a world of difference in memory use. Much like `force`, get in the habit of calling `deepSeqArray` when you know you need an array to become evaluated to continue.

We also store the iterations as Doubles which may seem silly, but we need a fractional format to render them anyhow. Also, this will turn out useful for other things later. Next we need to calculate these in the main function, which is simply folding over the orbits we've already calculated. This can be done in the main function which now becomes.

```
main = do
  -- Read arguments from commandline as before

  -- Create an initial grid
  let cs    = complexPlaneSegment dim rRange iRange

  -- Create initial iterations of the same size as the grid
  let is     = force2 $ R.map (λ_ → -1) cs

  -- Iteratively generate a bunch number sequence arrays
  let zss    = take iter $ iterate (nextOrbit cs) cs
  let normss = P.map ( force2 ∘ R.map (clamp thresh ∘ norm) ) zss

  -- Generate escape time values from orbits
  let iss    =
    scanl
    (λz_ns i → nextEscape thresh i (zss!!i) z_ns)
    is
    [0..iter-1]

  -- Write escape time to file
  writeMatrixToGreyscaleBMP
    ("iteration_" P.++ show iter P.++ "_escape_time.bmp")
    (transpose (last iss))
```

Note that we arbitrarily opted to get rid of the `forM_` batch writing at this point as file spam was getting a wee bit ridiculous. It's not just to confuse you.

The main change is that we use our new function for generating new escape times and iterate over the list. Generating the 100th iteration now gives ... an out of memory error.

So, this was touched on earlier – one should not keep several multi-megabyte arrays in memory, which is exactly what we're doing right now with our old orbits. We only need the most recent orbit and escape time information to make new ones.

We can rewrite our `nextIteration` function to calculate both the next escape time result and the next orbit.

```
nextIteration ::
  Double → Int →
  Array DIM2 Complex →
  Array DIM2 Complex →
  Array DIM2 Double →
  (Array DIM2 Complex, Array DIM2 Double)
nextIteration thresh iter cs z_ns is =
  deepSeqArray is $ deepSeqArray z_ns $ (z_ns',is')
  where is'   = force2 $ R.zipWith (escapeTime thresh iter) z_ns is
        z_ns' = force2 $ R.zipWith update cs z_ns
```

When calling in main we're still folding, just slightly differently

```
main = do
  -- Same initialization and...

  -- Fold to generate arrays.
  let (orbits,esctime) = foldl
      (λ(zs,is) i → nextIteration thresh i cs zs is)
      (cs,is)
      [0..iter-1]

  -- Normalize orbits
  let norms = force2 ∘ R.map (clamp thresh ∘ norm) $ orbits

  -- Write escape time to file
  writeMatrixToGreyscaleBMP
    ("iteration_" P.++ show iter P.++ "_escape_time.bmp")
    (transpose esctime)
```

This will correctly generate the escape time image seen in figure 5. In fact, it'll cheerfully go up to a few thousand iterations if you're patient.

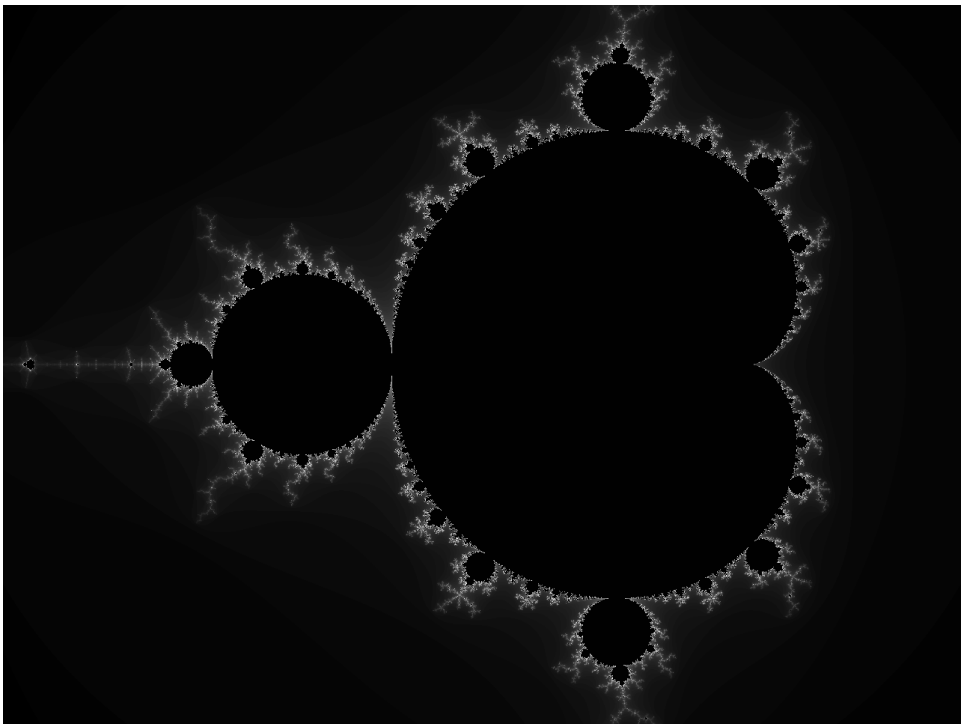


Figure 5: Escape time result from "`./mandelbrot 1280 960 100 2 -0.5 0 0.66 +RTS -N`". 100th iteration

3.5 Making everything pretty

There are two issues with the escape time image in figure 5, which may or may not be visible. First, it's in greyscale! There is no detail. Second, there is a banding effect as things either escape on iteration i or they do not, rather than smooth interpolation. Let's see what we can do.

First, color. Repa, of course, has a function to write color bitmaps: `writeImageToBMP :: FilePath -> Array DIM3 Word8 -> IO ()`. Here the function takes a `DIM3` matrix of `Word8` values, where the inner dimension must be 4 and each 8-byte word one of the RGBA components. We would like some way of generating colors from our Double values. One idea is to try to map each value to an index in a list `[Color]` where `type Color = (Word8,Word8,Word8)`. Non-integer values, if there are any, are then interpreted as an interpolation between the adjacent colors. An implementation of this is

```
-- A color is an RGB tuple
type Color = (Word8,Word8,Word8)

-- With components thus
red   (r,_,_) = r
green (_,g,_) = g
blue  (_,_,b) = b

-- Generate a write-ready color array from any coloring function
-- and array of Doubles
colorArray :: (Double → Color) → Array DIM2 Double →
          Array DIM3 Word8
colorArray color xs = unsafeTraverse xs resize update
  where resize (Z :. x :. y) = (Z :. x :. y :. 4)
        update lookup (Z :. x :. y :. c) =
          let pos = (Z :. x :. y)
              col = color $ lookup pos in
          case c of
            0 → red   col -- R
            1 → green col -- G
            2 → blue  col -- B
            3 → 0         -- A

-- The by list coloring function
colorByList :: [Color] → Double → Color
colorByList cs@(c1:c2:_) x
  | x > 1     = colorByList (drop i cs) (x - fromIntegral i)
  | x < 0     = (0,0,0) -- Negative → black
  | otherwise = interpolate x c1 c2
    where i = floor x

-- Color interpolation
interpolate :: Double → Color → Color → Color
```

```
interpolate x (r1,g1,b1) (r2,g2,b2) =
  (mix r1 r2, mix g1 g2, mix b1 b2)
  where mix v1 v2 = round ( x    *fromIntegral v2 ) +
                    round ( (1-x)*fromIntegral v1 )
```

New here is the `unsafeTraverse` function, which is a special, unchecked-but-fast version of the major Repa combinator `traverse`. Traversals in Repa are used to create new arrays from old ones, where the new arrays can have a different shape. This was not the case with zips and maps, you may recall – those operations were instead *shape preserving*.

The type is `traverse :: Array sh a -> (sh -> sh') -> ((sh -> a) -> sh' -> b)` . This should be read as follows: the first argument is a source array, here typically our array of escape times. The second argument is how the new array's extent is produced from the old one: here we add an inner dimension of 4 to fit the the RGBA `word8`:s. The third argument is a function defining how each element of the new array is produced. It is passed a lookup function that allows you to look at values in the source array and a position in the new array to populate. Here this produces one color component by looking at the corresponding Double in the source array and the color tuple it maps to.

The main thing that is needed from `main` is a preferably infinite list of colors for `colorByList`, an easy way of producing this is by doing.

```
main = do
  -- Init and generate data as before, then write using

  -- rgb color scheme
  let rgbcolors = cycle $ [(255,0,0),(0,255,0),(0,0,255)]

  -- Map image to color scheme, write RGBA
  writeImageToBMP
    ("iteration_" P.++ show iter P.++ "_norm_iter_color.bmp") $
    colorArray (colorByList rgbcolors) (transpose esctime)
```

Running this results in figure 6.

Here the banding is very clear, but there is fortunately an algorithm that fixes this. We can replace our escape time measure by a separate function called *Normalized Iteration Count* that takes into account by how much an orbit escapes the threshold. It only works for large thresholds, so we'll need to abandon our trusty two. We can also get rid of the noise near the set by simply applying a logarithm. This results in the final code change

```
-- Normalized iteration count: escape time with smoothing
{-# INLINE normIterCount #-}
normIterCount :: Double → Int → Complex → Double → Double
normIterCount thresh n z i =
  if i /= -1 then i
  else if normSq z > thresh*thresh then ni
```

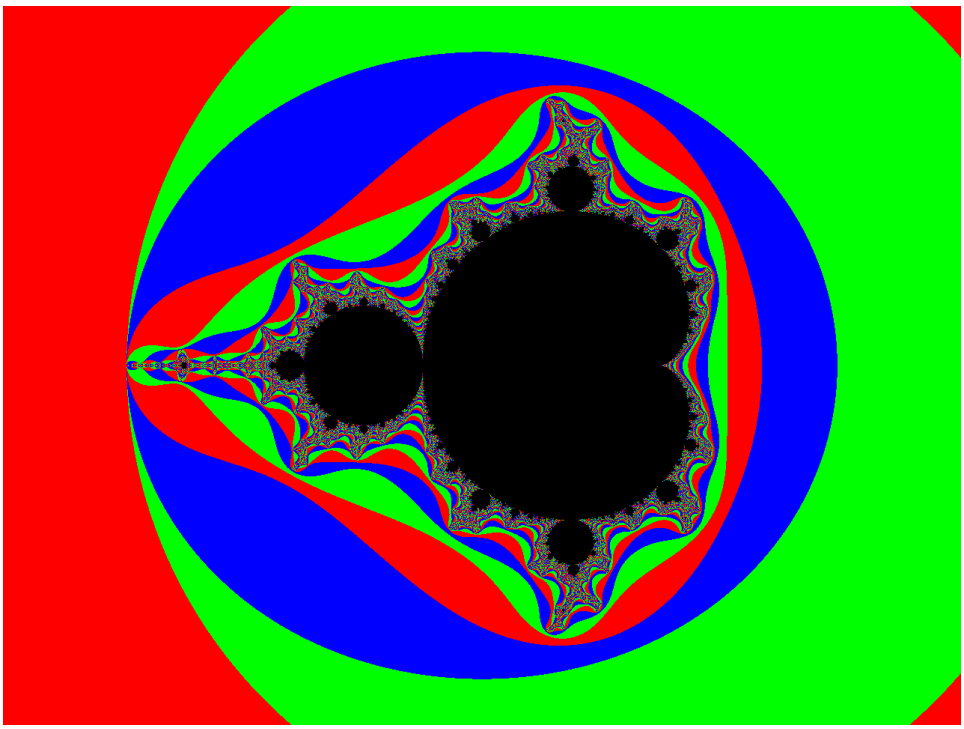


Figure 6: Escape time result from "`./mandelbrot 1280 960 100 2 -0.5 0 0.66 +RTS -N`". 100th iteration, colored

```
  else -1
  where ni = fromIntegral n -
             logBase 2 (log (normSq z) / (2*log thresh) )

-- Generate the next iteration of both escape times and
-- orbits from coordinates and previous iterations of the outputs
nextIteration ::
  Double → Int →
  Array DIM2 Complex →
  Array DIM2 Complex →
  Array DIM2 Double →
  (Array DIM2 Complex, Array DIM2 Double)
nextIteration thresh iter cs zs is =
  deepSeqArray is $ deepSeqArray zs $ (zs',is')
  where is' = force2 $ R.zipWith (normIterCount thresh iter) z_ns is
        zs' = force2 $ R.zipWith update cs zs

main :: IO ()
main = do
  -- Initialize as before

  -- Fold to generate arrays.
```

```
let (orbits,esctime) = foldl
    (λ(zs,is) i → nextIteration thresh i cs zs is)
    (cs,is)
    [0..iter-1]
let norms      = force2 ∘ R.map (clamp thresh ∘ norm) $ orbits
let logEsctime = force2 ∘ R.map (logBase 2)           $ esctime

-- rgb color scheme
let rgbcolors = cycle $ [(255,0,0),(0,255,0),(0,0,255)]

-- Map image to color scheme, write RGBA
writeImageToBMP
  ("iteration_" P.++ show iter P.++ "_norm_iter_color.bmp") $
  colorArray (colorByList rgbcolors) (transpose logEsctime)
```

We can produce some pretty neat images of the Mandelbrot Set while taking the full benefit of multicore processors. Repa is very well suited for image processing tasks like this, as the rendering is very easy to define as a parallel task on an array. The final result is smooth, fast fractal rendering in around 150 lines of code. To conclude the tutorial, a couple of different images generated by the final version of the program:

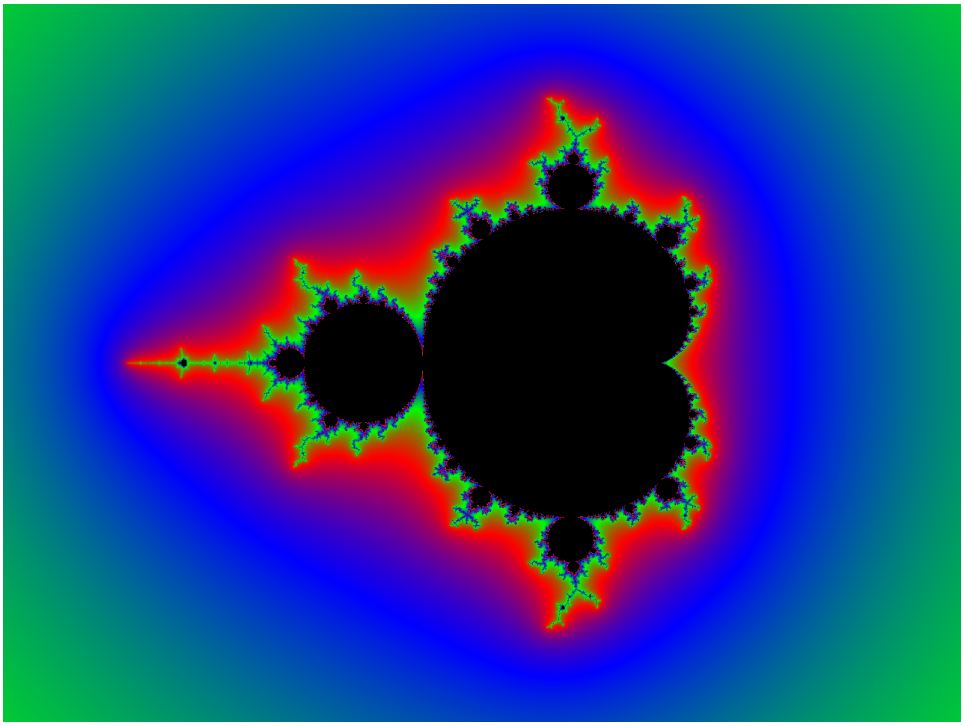


Figure 7: "`./mandelbrot 1280 960 100 100 -0.5 0 0.66 +RTS -N`" Final drawing algorithm: Normalized Iteration Count with logarithmic scaling

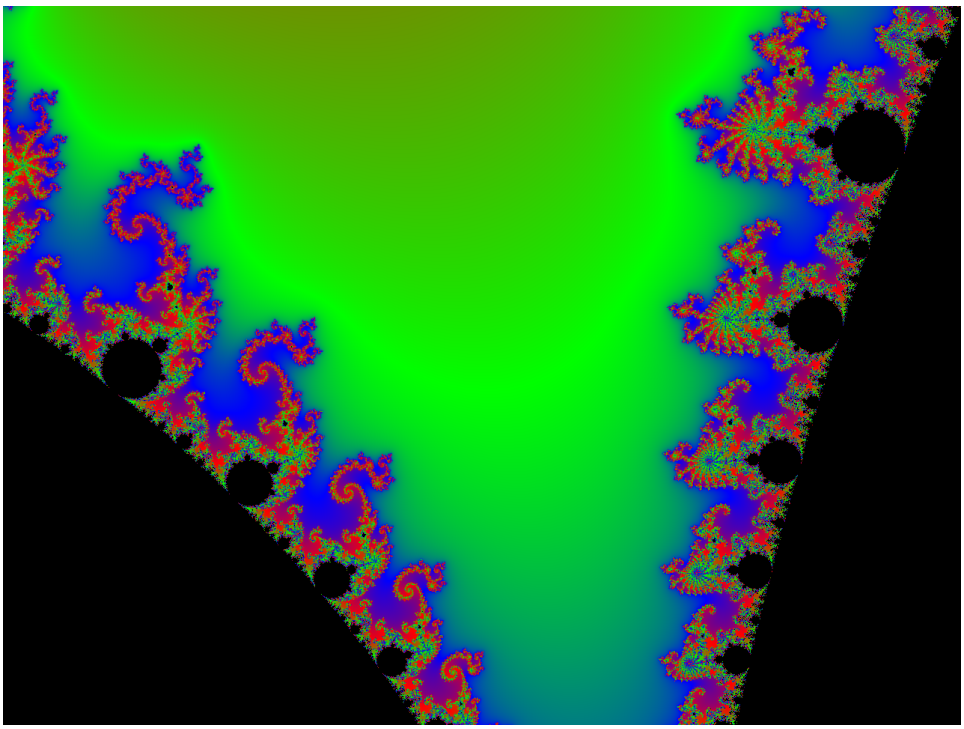


Figure 8: "`./mandelbrot 1280 960 1000 100 -0.78 0.2 16 +RTS -N`" A more detailed zoom on coordinates $-0.78 + 0.2i$

3.6 Possible extensions, optimizations and improvements

If you followed along so far, you should have a pretty good grasp of what Repa can do. If you're not yet bored you may be wondering: what more can you do with this? Adding things like a GUI and generally making the renderer usable by humans springs to mind, however such things are generally quite boring. There are more interesting ideas around, and a non-exhaustive list is:

- Optimizations! The escape time algorithm especially is quite slow, simply because a lot of orbits are being calculated that do not need to be. First, the central black cardioid and the perfectly circular first order bulb can both be excluded by checking their bounds. Second and most importantly, once an element has been determined to have escaped one does not need to iterate further on it. A lot of wasted time is currently spent there. Extensive computation is only required in a band near the boundary of the set, especially for renders that cover the entire thing. These optimizations can be achieved by using the *Slice* functionality in Repa, which has not been covered in this tutorial.

- Other algorithms! There are fractals that are related to the Mandelbrot but use slightly different update functions. There exists a Julia Set for each point in the complex plane, for instance. One can also vary the basic sequence generating function for the Mandelbrot Set by changing to different polynomial, which will result in different fractals. There are also more computationally intense related fractals like the Buddhabrot which could be rendered using Repa.
- Other coloring schemes! We have colored using escape time and orbit norm, but one might also color the orbits using the real and imaginary components, or use completely different color schemes like coloring an each exterior point by the distance to the closest point on the Mandelbrot Set itself.
- Beating Nyquist! A common problem in our renderings is that details near the boundary looks extremely messy, since the iteration count changes much more rapidly than once/pixel. The logarithmic scale was a rather hackish way of dealing with this while preserving information. A better solution is to take several samples per pixel and average them to produce a more accurate escape time value for the entire pixel.
- Make a raytracer! A Mandelbrot renderer is really a special case of ray tracing – our rays are orbits in the complex plane and they collide with a great big threshold cylinder and return its color. The exact same techniques used here to trace those orbits can be used to trace any rays, for instance those of photons, and collide them with anything, such as geometrical objects. Repa is very well suited for implementing a general purpose offline 3D renderer, if this strikes your fancy.

A Code

The full program we've written is reproduced below

```
{------------------------
-- Mandelbrot Renderer --
--     using Repa      --
--                     --
--   Marcus Lonnberg   --
--   marlon@student    --
--                     --
--   Karl Schmidt      --
--   karsch@student    --
------------------------}

module Main where
```

```
import Prelude as P
import Data.Array.Repa        as R
import Data.Array.Repa.IO.BMP as R

import Control.Monad

import System.Environment

import Data.Word

  ------------------------
  -- Complex Arithmetic --
  ------------------------

--Cannot use Data.Complex in Repa, so encode as tuple (Real, Imag)
type Complex = (Double,Double)

-- Multiplication
{-# INLINE mul #-}
mul :: Complex -> Complex -> Complex
mul (r1,i1) (r2,i2) = ( real, imag )
  where real = r1*r2 - i1*i2
        imag = i1*r2 + i2*r1

-- Addition
{-# INLINE add #-}
add :: Complex -> Complex -> Complex
add (r1,i1) (r2,i2) = ( r1+r2, i1+i2 )

-- Square
{-# INLINE sq #-}
sq :: Complex -> Complex
sq (r,i) = ( r*r - i*i, 2*r*i )

-- Square of the 2-norm
{-# INLINE normSq #-}
normSq :: Complex -> Double
normSq (r,i) = ( r*r + i*i )

-- 2-norm
{-# INLINE norm #-}
norm :: Complex -> Double
norm = sqrt . normSq
```

```
   ------------------
   -- Repa Helpers --
   ------------------

-- Generate a region of the complex plane as a Repa array.
-- Takes array dimension and bounding values for the region.
complexPlaneSegment :: DIM2 -> (Double,Double) -> (Double,Double) ->
                       Array DIM2 Complex
complexPlaneSegment (Z :. x :. y) (rmin,rmax) (imin,imax) =
  fromList (Z :. x :. y) ris
  where rstep = (rmax-rmin) / (fromIntegral x - 1)
        istep = (imax-imin) / (fromIntegral y - 1)
        reals = take x $ iterate (+rstep) rmin
        imags = take y $ iterate (+istep) imin
        ris   = [(r,i) | r <- reals, i <- imags]

  -------------------------------------
  -- Update functions when iterating --
  -------------------------------------

-- Clamp a value to an upper bound
{-# INLINE clamp #-}
clamp :: Double -> Double -> Double
clamp thresh x = if abs x < thresh then x else thresh

-- The update rule for our Complex sequence
-- Again, we want this to be used in parallel so force inlining!
{-# INLINE update #-}
update :: Complex -> Complex -> Complex
update c z_n = sq z_n `add` c

-- Escape time: just check when we escape
{-# INLINE escapeTime #-}
escapeTime :: Double -> Int -> Complex -> Double -> Double
escapeTime thresh n z i =
  if i /= -1 then i
  else if normSq z > thresh*thresh then fromIntegral n
  else -1

-- Normalized iteration count: escape time with smoothing
{-# INLINE normIterCount #-}
normIterCount :: Double -> Int -> Complex -> Double -> Double
```

```
normIterCount thresh n z i =
  if i /= -1 then i
  else if normSq z > thresh*thresh then ni
  else -1
  where ni = (fromIntegral n) - logBase 2 (log (normSq z) / (2*log thresh) )

-- Generating a new orbit
nextOrbit :: Array DIM2 Complex ->
             Array DIM2 Complex ->
             Array DIM2 Complex
nextOrbit cs z_ns = deepSeqArray z_ns $
  force2 $ R.zipWith update cs z_ns

-- Generating a new iteration of escape times from
-- previous iteration and current orbits
nextEscape :: Double -> Int ->
              Array DIM2 Complex ->
              Array DIM2 Double ->
              Array DIM2 Double
nextEscape thresh iter z_ns is = deepSeqArray is $
  force2 $ R.zipWith (normIterCount thresh iter) z_ns is

-- Generate the next iteration of both escape times and orbits from a
-- previous iteration
nextIteration ::
  Double -> Int ->
  Array DIM2 Complex ->
  Array DIM2 Complex ->
  Array DIM2 Double ->
  (Array DIM2 Complex, Array DIM2 Double)
nextIteration thresh iter cs z_ns is =
  deepSeqArray is $ deepSeqArray z_ns $ (z_ns',is')
  where is'   = force2 $ R.zipWith (normIterCount thresh iter) z_ns is
        z_ns' = force2 $ R.zipWith update cs z_ns

-- Colors are just RGB tuples
type Color = (Word8,Word8,Word8)

-- With components thus
red   (r,_,_) = r
green (_,g,_) = g
blue  (_,_,b) = b

-- Generate a coloring of an array from a coloring function
```

```
colorArray :: (Double -> Color) -> Array DIM2 Double ->
          Array DIM3 Word8
colorArray color xs = unsafeTraverse xs resize update
  where resize (Z :. x :. y) = (Z :. x :. y :. 4)
        update lookup (Z :. x :. y :. c) =
          let pos = (Z :. x :. y)
              col = color $ lookup pos in
          case c of
            0 -> red   col -- R
            1 -> green col -- G
            2 -> blue  col -- B
            3 -> 0         -- A

-- Generate a coloring function from a list, where the 2nd argument is
-- treated as an "index" with interpolation
colorByList :: [Color] -> Double -> Color
colorByList cs@(c1:c2:_) x
  | x > 1     = colorByList (drop i cs) (x - fromIntegral i)
  | x < 0     = (0,0,0) -- Negative -> black
  | otherwise = interpolate x c1 c2
    where i = floor x

-- Interpolate between two color values. First argument assumed in [0,1]
interpolate :: Double -> Color -> Color -> Color
interpolate x (r1,g1,b1) (r2,g2,b2) = (mix r1 r2, mix g1 g2, mix b1 b2)
  where mix v1 v2 = round (x*fromIntegral v2) + round ( (1-x)*fromIntegral v1 )



-- Main function. Get arguments, run simulation, write pictures.
main :: IO ()
main = do
  -- get command line argument list
  args <- getArgs

  -- pic size, iters
  let [xsize, ysize, iter]       = P.map read $ take 3 args

  -- threshold, rendering area and zoom factor
  let [thresh, rmid, imid, zoom] = P.map read $ drop 3 args

  -- Sim parameters
  let aspectRatio         = fromIntegral xsize / fromIntegral ysize
  let dim                 = (Z :. xsize :. ysize :: DIM2)
```

```
let rRange              = (-aspectRatio/zoom + rmid, aspectRatio/zoom + rmid)
let iRange              = (-1/zoom + imid, 1/zoom + imid)

-- Create an initial grid
let cs    = complexPlaneSegment dim rRange iRange

-- Create initial escape iterations
let is     = force2 $ R.map (\_ -> -1) cs

-- Fold to generate arrays.
let (orbits,esctime) = foldl (\(zs,is) i -> nextIteration thresh i cs zs is) (cs,i
let norms      = force2 . R.map (clamp thresh . norm) $ orbits
let logEsctime = force2 . R.map (logBase 2)           $ esctime

-- rgb color scheme
let rgbcolors = cycle $ [(255,0,0),(0,255,0),(0,0,255)]

-- Map image to color scheme, write RGBA
writeImageToBMP
  ("iteration_" P.++ show iter P.++ "_norm_iter_color.bmp") $
  colorArray (colorByList rgbcolors) (transpose logEsctime)
```