# notebook1

March 1, 2018

## Contents



# 1 Overview of the Blurring filter

1. What is a Blurring filter?

   - A blurring filter in this case is just a function that works on the neighborhood of a pixel
     - this is also known as a linear filter
     - can be expressed mathematically as $g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$
       * where $g$ represents the new image

$f$ is the original imageblurGausX (Source r b, Fractional b) => Array r DIM2 b -> Array D DIM2 b

blurGausY :: (Source r b, Fractional b) => Array r DIM2 b -> Array D DIM2 b

   - and $h$ is the is the weight/kernel
   - This operation is also considered f being convolved by h

2. Types of Kernels

   - Not all Kernels produce a blurring filter, for example the Sobel kernel (shown below) is more useful for edge detection than for blurring
     - the following two are the x and y Sobel kernel respectively $\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{pmatrix}$, $\begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix}$
   - Though there are a few filters that can work for blurring, I used the Gaussian filter which can be defined as such below
     - Gaussian $= \frac{1}{256} \times \begin{pmatrix} 1 & 4 & 6 & 4 & 1 \\ 2 & 8 & 12 & 8 & 2 \\ 6 & 24 & 36 & 24 & 6 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}$

   (a) Issues and Optimizations to the Kernel

   - So the issue with the Gaussian filter above is that applying the convolution would take $O(K^2)$ for every pixel of the image $f$, which could turn out very slow.
   - Thankfully there is a way to get this operation to $O(2K)$, however the kernel matrix must be separable.
   - So for the Gaussian filter above we can define it like this

- $v = \frac{1}{16} \times \begin{pmatrix} 1 & 4 & 6 & 4 & 1 \end{pmatrix}$
- $h = v$
- Gausian $= vh^T$

3. Dealing With Edge Cases

- So there is one more issue with the linear filter method, and that deals with the edge of the image matrix, as there may not be an up,left,right or down for any particular pixel.
- So there are a few schemas available, and I will outline the ones I tried
  - My first instinct was to fill the surrounding matrix with zeros, and then apply the transformation above. This is also known as the zero padding
    * for a very short while I even tried reducing how much each edge was divided by, as 0's don't add anything to the total, so I made a formulation that divided by the correct numbered multiplied.
    * This formulation worked kind of well until I realized that GausBlurX $\circ$ GausBlurY $\neq$ GausBlurY $\circ$ GausBlurX. and thus the transformation really isn't separable
  - After reading the textbook for a bit I ended up deciding to go with clamp padding, which basically just repeats the edges on either side, so the calculation would more accurately represent the edges. I will talk more about how this is implemented in each section in the implementation section

# 2 Implementing the Blurring filter

- I did this exercise twice actually

1. With a Matrix Library
   - Pros
     * this was quite easy to work with, as it abstracted the 1D vector
     * Can use Purely
     * has O(1) slices by just changing the internal bounds
   - Cons
     * it did not give me proper access to create the type, so I had to rely on fusion to convert my image to the data type
     * turned out to be slow when I promoted the Word8 (8-bit unsigned $\mathbb{Z}$) to Word16 (16-bit unsigned $\mathbb{Z}$). It went from a .11 second computation to never stopping believe it or not! (I figure it has something to do with caching rather than anything with 8-bit vs 16 bit computation)
       · I ended up spending hours trying to investigate this problem and got nowhere
2. With the REPA Library
   - Pros
     * Very fast, as they are automatically parallelized and use a lot of fusion
     * Represented with pure data operations outside of forcing parallelism
     * Abstracted out making the stencil and boundary clamp part of the problem
     * Has an extension to work natively with the library that reads images (JuicyPixels)
   - Cons
     * Very hard to understand at first
       · Took me hours to get what I was even doing
       · Instead of a normal representation of nested vectors, it abstract out all details via a backwards list structure known as a shape

· Also all computation causes the vector to not exist until some action forces it, that way intermediate data structures are just left as functions and are fused out

- and there was an even cooler way to implement this then the two listed above, and it deals with comonads and cellular automata, but that is something to be investigated at a later date

1. Matrix Representation

   - So using a Matrix representation for this problem was my first thought, and thankfully there is a library that has just that in Haskell.

   (a) MatrixHelper

   - before even bothering with converting an image, I wanted to make sure I was able to move a cursor throughout the entire image, so I decided to make the file "MatrixHelper.hs" which contains some useful tools
   - The first tool that I created was

```
extractWindows :: Int → Int → Matrix a → Matrix (Matrix a)
extractWindows row col mat = matrix (nrows mat - row + 1) (ncols mat - col + 1) f
  where
    f (i,j) = submatrix i (i + row - 1) j (j + col - 1) mat
```

     – The function takes a row, column and a matrix, and returns a Matrix containing sub matrices of the size of the row and column
     – the size of the sub matrices are row x col and start from the top left moving to the right then finally down by 1 pixel each time
     – taking sub matrices is thankfully $O(1)$, as we never really create a new matrix, so really this operation is $O((n - row) \times (m - col))$ where n is the size of the rows and m is size of the columns and row and col are defined as they are in the function above

   - The other tools I created in this file aren't really useful for image blurring, but instead might prove useful for the letter detection. They aren't too useful as they don't pad the image, below is the code, however I won't explain

```
linearFilter :: (RealFrac a, Integral b) ⇒ Matrix a → Matrix a → Matrix b
linearFilter filt = fmap (round . sum . elementwise (*) filt) . extractWindows row col
  where row = nrows filt
        col = ncols filt

(⊕) :: (RealFrac a, Integral b) ⇒ Matrix a → Matrix a → Matrix b
(⊕) = flip linearFilter
```

   (b) ImageHelper

   - now that I am able to move a cursor through the matrix, I now have to figure out how to read an image, and convert an image to a matrix representation. I ended up breaking this work into "ImageHelper.hs"
     – Reading and converting an image

```
testImage :: IO (Image PixelRGB8)
testImage = do
  img <- readImage "../data/test-old.png"
  case img of
    Right (ImageRGB8 img) → return img
    Left err → error ("can't load image: " <> err)
    Right _ → error "unsupported format"
```

     ∗ I ended up deciding to go with JuicyPixels for all my image reading needs, here I read a RGB8 image, this isn't the most robust reader, since I later decided to do the matrix representation as all greyscale, I could probably read more formats and convert them here.

```
imageToGreyMatrix :: LumaPlaneExtractable a ⇒ Image a → Matrix (PixelBaseComponent a)
imageToGreyMatrix img = matrix (imageWidth img) (imageHeight img) f
where
newImg  = extractLumaPlane img           -- turns the image into greyscale
f (x,y) = pixelAt newImg (x - 1) (y - 1) -- matrix is 1 indexed not 0

-- fusion does not happen, so this is slower than the non ' version
imageToGreyMatrix' :: LumaPlaneExtractable a ⇒ Image a → Matrix (PixelBaseComponent (P
imageToGreyMatrix' img = fromList (imageWidth img) (imageHeight img) newVec
where
newVec  = VS.toList . imageData . extractLumaPlane $ img
```

* the code above is two ways to turn an image into a matrix
* the first is rather straight forward as I use the `matrix` function to read create the matrix the same size as the image and just grab the pixel at each point after running a quick conversion to greyscale
* the second version was me being a bit cheeky, since I knew that the Matrix internally used a matrix representation, I thought I could just do VS.toList and fromList it, and have that fuse into 0 pass throughs, sadly it seemed it didn't work as intended and is slower (probably because I didn't inline the funcion for better analysis)

- now that I read in the image, it's time to actually make the image blur and blur the image, I kept all this computation in ImageHelper.hs for some reason
  - Blurring the image

```
gausianConst :: Num a ⇒ [a]
gausianConst = [1,4,6,4,1]

blurSepX :: Matrix Word16 → Matrix Word16
blurSepX mat =  withWord16 (* gausblur) <$> extracted
  where
    clampL    = colVector $ getCol 1           mat -- this gives us the
    clampR    = colVector $ getCol (ncols mat) mat -- clamp border effect
    buffered  = (clampL <|> clampL) <|> mat <|> (clampR <|> clampR)
    extracted = extractWindows 1 5 buffered
    gausblur  = fromList 5 1 gausianConst

blurSepY :: Matrix Word16 → Matrix Word16
blurSepY mat = withWord16 (gausblur *)  <$> extracted
  where
    clampU    = rowVector $ getRow 1           mat
    clampD    = rowVector $ getRow (nrows mat) mat
    buffered  = (clampU ↔ clampU) ↔ mat ↔ (clampD ↔ clampD)
    extracted = extractWindows 5 1 buffered
    gausblur  = fromUist 1 5 gausianConst

withWord16 :: (Matrix Word16 → Matrix Word16) → Matrix Word16 → Word16
withWord16 f mat = (`div` 16) . sum $ f mat16
  where
    mat16 = fromIntegral <$> mat :: Matrix Word16

blur :: Matrix Word16 → Matrix Word8
blur = fmap fromIntegral . blurSepY . blurSepX
```

  * GausianConst
    - so this just mimics the h definition in part 1, I end up converting this to a matrix in the computation below

    * blurSepX
        · this is filter that blurs the image with the $v^T$ filter from part1. this is called blurX instead of blurY as we are getting 5 by 1 slices of our matrix as seen in `extracted = extractWindows 5 1 buffered`. now buffered isn't our original matrix, instead it's our matrix but padded on the left and right with the leftmost and rightmost elements respectively. as discussed in part1 these are just clamps, originally I just had zeros on the edges, but I did not like the results
        · I will discuss what `withWord16` does in the section below, as there is interesting optimizations happening there. However I will say that I do send the partial application of the matrix multiplication to `withWord16` as this is where the pixel value is calculated
    * blurSepY
        · This is basically the same as blurSepX except we are working on vertical slices instead of horizontal slices (hence y and not x)
    * withWord16
        · So this function is particular interesting, as this is where the matrix representation of the code breaks down. If Ichanged all the bindings to `Word8` and get rid of the `mat16` line, then I would get rounding errors, as 88 * 6 = 16, and thus after calculating a matrix multiplication, we get a number between 0-255 that gets divided by 16 after it leaving us with a range of 0-16 after rounding.
        · However quickly converting an image with `Word8` math was really fast and I could even convert big images rather quickly, however when I converted that one section of computation to `Word16` the program stopped, and I could only convert small images
    * blur
        · This function is rather simple, it's just the composition of the two other blurs, and this is what I will use to generate the images that will be under the next section
- After Getting the blurring up, I had to convert the data type back to an image, which was once again quite simple

```
matrixToGreyImg :: Pixel a => Matrix a -> Image a
matrixToGreyImg mat = generateImage f (ncols mat) (nrows mat)
  where f i j = mat ! (i + 1, j + 1)
```

  – this code really speaks for itself, ! is an index operation, and the matrix is 1 indexed not 0, so I had to add the (+1)'s
- Other Issues
  – Another issue besides the time of this interpretation is the amount of memory it consumes.
  – this way of dealing with the image would load the entire image into memory, which is far from ideal, and we'll see in the REPA representation the memory usage stays low all throughout
- Now that we finally have our Image → Matrix → Image code up we can finally convert some images!
  – shown below is the code I used to run the process

```
mainMatrix :: IO ()
mainMatrix = do
  x <- testImage
  let new  = blur $ fmap fromIntegral (imageToGreyMatrix x)
  let new' = matrixToGreyImg new
  savePngImage "./test-2.png" (ImageY8 (matrixToGreyImg (imageToGreyMatrix x)))
  savePngImage "./test.png" (ImageY8 new')
```

    * we just run and save the image before blur and after blur
        · the far left one is the original fully colored, we'll see more of that one later
        · the middle is the unaltered greyscale version of the image

· and finally the far right is the blurred version of this 150x150px image;

2. Repa Representation

- This representation comes from the fact that I wanted the code to run on the full 1500x1500 version of the images above, and I spent hours trying to debug why `Word16` slowed down the program so much.
- Repa also gives me tools to do this work rather easily, so I'll break this section into 3 parts
  (a) What is REPA and why did it take me to understand what I was doing
  (b) Working just on Grey images
  (c) Working on both grey and colored images

(a) What is REPA and why did it take me to understand what I was doing

- so REPA is a library for high performance regular multi-dimensional parallel arrays.
- This means a few things
  i. we don't have to say a word about parallelism and our code will still be run in parallel (I ended up getting 100% on all 8 of my cores!)
  ii. REPA is rather fast and memory efficient
  iii. REPA due to its "multi-dimensional" nature has rather complex type signatures and makes grokking it rather hard at first
- So REPA achieves its speed in a rather interesting way, whenever a function is invoked, REPA doesn't actually make an array

```
a = fromListUnboxed (Z :. 4 :. 4) [1..16] :: Array U DIM2 Int
R.map (+ 1) a :: Array D DIM2 Int
```

  - so here we make an unboxed array (that's what U means) of dimension 4 by 4 (that's what Z :. ... means) with type Int inside.
  - when we run map over the entire array instead of getting another Unboxed type U back, we instead get the type `Array D DIM2 Int` back, where D means that this array is really just functions from indices to elements. So the array never really exists in memory
  - this is rather useful, as this map can be fused out and the intermediate arrays never exist
- Another note is the shape, the (Z :. 4 :. 4) notation denotes the shape of the array, and this data structure is best to be thought of as a reverse list
- even with understanding both of these points, it took me more than just a few hours to fully understand how to use the library, and in the following two sections I'll try to explain the logic of what is happening.

(b) Working just on Grey images

- so like the matrix representation I decided to once again only work on grey images at first
- the code for both these sections are in RepaHelper.hs
- The first step was trying to figure out how to turn an Image into a Repa array.

```
-- only going to be working on 2D images for now, trying to figure out slices is too much
imageToGreyRepa :: LumaPlaneExtractable a ⇒ Image a → Array D DIM2 (PixelBaseComponent a)
imageToGreyRepa img@(Image w h _) = R.fromFunction (Z :. w :. h) f
  where f (Z :. i :. j) = pixelAt newImg i j
        newImg          = extractLumaPlane img
```

- so I end up representing a grey image as a 2D array (DIM2 stands for dimension 2).
- I do this by making a function that takes an image (an image consists of the width, height, and data) which we call img with width w and height h, and returning our array
- this array never really gets materialized, as we just make the array from a function that just queries the greyed version of the image

- Now that we have the image in the data that we can work with, we must now make our Gaussian once again

```
gausianStencilX :: Num a ⇒ Stencil DIM2 a
gausianStencilY :: Num a ⇒ Stencil DIM2 a
gausianStencilX = [stencil2| 1 4 6 4 1 |]
gausianStencilY = [stencil2| 1
                             4
                             6
                             4
                             1 |]
```

- this code is a bit special. So Repa has a stencil library that was made to basically apply any arbitrary kernel as long as it's smaller than 7x7
- so these two represent the Gaussian and give other data to our function below

```
blurGausX :: (Source r b, Fractional b) ⇒ Array r DIM2 b → Array D DIM2 b
blurGausY :: (Source r b, Fractional b) ⇒ Array r DIM2 b → Array D DIM2 b
blurGausX = R.map (/ 16) . mapStencil2 BoundClamp gausianStencilX
blurGausY = R.map (/ 16) . mapStencil2 BoundClamp gausianStencilY

blur :: (Source r b, Fractional b) ⇒ Array r DIM2 b → Array D DIM2 b
blur = blurGausX . blurGausY
```

- The library is kind enough to give us a mapStencil over a 2D array, so really we just map the Gaussian and then divide by 16, and compose both of them to get the blur
- The stencil probably runs in parallel, though I'm not quite sure where the parallelization is coming from (might be even before this point!)

- So now that we have the blur filter working, we just have to make a conversion function then run it

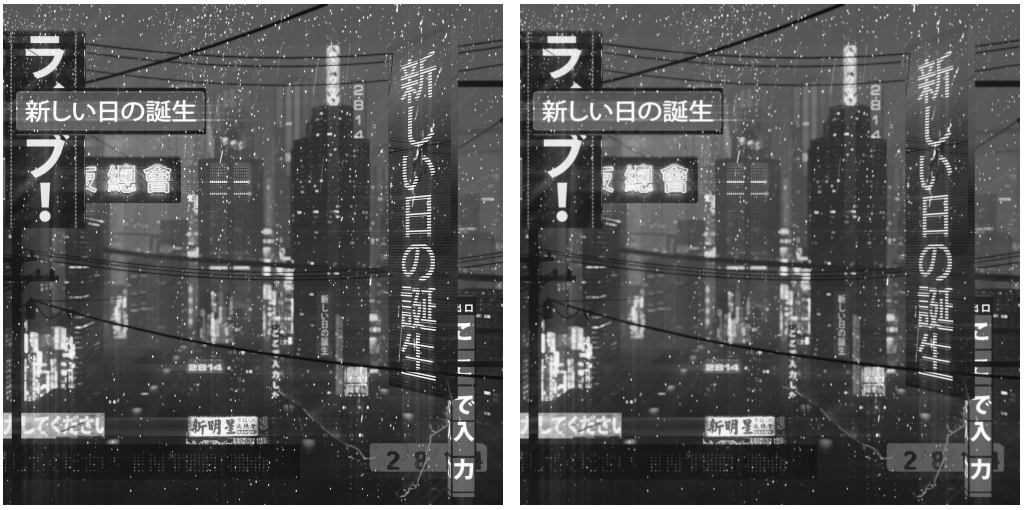
```
repaToGreyImage :: (RealFrac a, Source r a) ⇒ Array r DIM2 a → Image Word8
repaToGreyImage xs = generateImage create width height
  where Z :. width :. height = R.extent xs
        create i j           = round (xs ! (Z :. i :. j)) :: Word8
```

- once again we use generateImage and grab the proper coordinates

- So now lets run the image blur again but on a bigger version of the same image!

```
mainRepaGrey = do
  x <- testImage
  let y = R.imageToGreyRepa x
  let z = R.blur $ R.map fromIntegral y
  savePngImage "./repa-test-real.png" (ImageY8 (R.repaToGreyImage z))
```

- this code basically grabs the image and runs our functions
- and once again we save the pngs which can be viewed below
- I've had to scale the image down for the PDF, so do use my directories and look at "repa-big-test" and "test-big"




  - So the image on the left below is the original
  - And the image on the right is the blurred version
  - I ended up scaling the images to .4, as the blurring is more subtle on bigger image
  -

(c) Working on both grey and colored images

- this section in particular took a lot of effort as it took me quite a while to grok how slices work in this library and how to properly use the tools
- Reading an image to a REPA array is already done for me as there is a `readImageRGB` function provided in JuicyPixels-repa, so I don't have to worry about that part
- to get a grip on the tools, I thought I would make repaExtractWindows which really isn't used

```
repaExtractWindows :: (Source r a) => Int -> Int -> Array r DIM3 a -> Array D DIM3 (Array D
repaExtractWindows row col arr = R.fromFunction (Z :. i - row :. j - col :. k) grabsubs
  where Z :. i :. j :. k = R.extent arr
        grabsubs sh      = R.extract sh (Z :. row :. col :. 1) arr
```

  - I made this as I was confused on how to get subsection of the array properly. At first `R.fromFunction` used to be `R.traverse` that took an array and did some calculations, but it turned out to not be needed
  - there is a nifty function called extract which I can give it a shape of where to start and how big it is.

- Now that Iwas understanding what I was doing, we can now make blurCol

```
data MyImage a = RGB a a a | RGBA a a a a | Grey a

fromList :: [a] -> MyImage a
fromList [a,b,c]   = RGB a b c
fromList [a,b,c,d] = RGBA a b c d
fromList [a]       = Grey a
fromList _         = error "not a valid image"

blurCol :: (Fractional e, Source r e) => Array r DIM3 e -> Array D DIM3 e
blurCol = flip reshape . f . fromList . fmap blur . slices <*> R.extent
  where f (RGBA a b c d) = interleave4 a b c d
        f (RGB a b c)    = interleave3 a b c
        f (Grey a)       = a
```

```
slices :: Source r e => Array r DIM3 e -> [Array D DIM2 e]
slices arr = f <$> [0..(k-1)]
  where
    (Z :. _ :. _ :. k) = R.extent arr
    f a                = slice arr (Z :. All :. All :. (a :: Int))
```

- I broke this function up into 3 discrete pieces, my custom data type, blurCol, and slices
  - * Slices
    - slices takes an array and stuffs the 3rd dimension of the array into its own list while keeping all x and y coordinates of the array
    - this leaves us with a list of 2D arrays, which means the old blur filter can work
  - * MyImage
    - this data type was mostly a response to the interleave functions. due to how strict Haskell is with its types I can't just check the size of the list I get from slices and decide on which one I want
    - so I created this data type just to facilitate the interleave functionality
    - interleave just interleaves all elements in the arrays given to it
    - Also note at first I used R.++ instead, but that ended up splitting they image into 3 versions of the original image
  - * blurCol
    - blurCol is where the magic happens, now that we have 2D slices, we can now just call blur on each slice and combine it with interleave. Finally we get a 2D array back, so we just reshape the array into a 3rd dimensional one

- All that is left is converting the array into a 3D one. there is actually a function called `imgToImage :: Img a → DynamicImage` but oddly enough it segfaults (it uses some weird foreign pointer magic), so I made my own once again

```
repaToRGBImage :: (RealFrac a, Source r a) ⇒ Array r DIM3 a → Image PixelRGB8
repaToRGBImage arr = generateImage create height width -- may have mixed up the width and h
  where
    Z :. width :. height :. _ = R.extent arr
    create i j                = PixelRGB8 (grab 0) (grab 1) (grab 2)
      where grab k = round $ arr ! (Z :. j :. i :. k) :: Word8
```

  - So this code looks a lot like the grey image converter, however there is one weird difference, and that is in the generate image I give it height then width, and I index my array with j then i instead of i then j. I think something might have been flipped at once point, I'm not too sure, but this works like a charm.
  - A previous version used to use foldl1 with the PixelRGB8 and keeping the computation in the list, but since Iused the wrong fold, I ended up mixing up R and B on the final image which I will show with the rest.

- So now lets run and see what we get!

```
main = do
  x <- C.readImageRGB "./data/Color-test.png"
  let y = case x of Left _ -> error "image not found"; Right z -> z
  let z = R.blurCol (R.map fromIntegral (imgData y))
  z' <- R.computeUnboxedP z :: IO(R.Array R.U R.DIM3 Double)
  let z'' = R.repaToRGBImage z'
  savePngImage "./Color-save.png" (ImageRGB8 z'')
```

  - the computeUnboxedP is the only parallel code I've written, which just tells the array to be computed in parallel, idk if this speeds up the computation however.
  - but lets see the fruits of our labor. I would suggest looking in the data director and look at "Color-test" "Color-save-proper-colors" and "Color-save-proper-colors" for the full size images


- the left is the original
- the right was the mistaken R B flip and blurred
- and the bottom is the blurred version of the first