Notebook2

March 7, 2018

Contents

1 Implementing Low and High Pass Image Filters

- Instead of doing my initial spiel about high and low pass filters, I will instead discuss the relevant information after I implement `lowPass` and `highPass` and before I talk about the results found.

1. Setup

   - Since we need to read images, I bring some functions to convert images into a proper REPA Array (REPA are just very quick and parallel arrays I use), and make two additional functions

     ```
     repaRGBToGrey :: (Integral b, Source r b) ⇒ Array r DIM3 b → Array D DIM2 b
     repaRGBToGrey arr = R.traverse arr (\_ → ix2 i j) average
       where (Z :. i :. j :. _)    = R.extent arr
             getAll3 (Z :. i :. j) = (\k -> Z :. i :. j :. k) <$> [0..2]
             average f sh          = round $ sum (fromIntegral ∘ f <$> getAll3 sh) / 3

     saveRepaGrey :: (RealFrac a, Source r a) ⇒ FilePath → Array r DIM2 a → IO ()
     saveRepaGrey loc = savePngImage loc ∘ ImageY8 ∘ repaToGreyImage
     ```

     – here `repaRGBToGrey` just turns a 3d array into a grey one by adding all values over the third dimension then dividing by 3, giving us a new array

     – saveRepaGrey is just a useful function that converts our 2D array into an Image which I then save, so I don't have to call all the transformations all the time.

2. Implementations

   - I ended up trying two different discrete cosine transformers, as when I test dct composed with idct I did not get the same vector back. I eventually managed to get the first one working, by noticing that the value returned from `(dct · idct) vec` = $\frac{vec}{\text{length(vec)}*2}$
   - with that said, I will now talk about the successful attempt.
   - So I notice an import in the "statistics" package has a discrete cosine transformer and its inverse, so I decided to use that

     ```
     repaVecComp :: Shape sh => (V.Vector e1 → V.Vector e2) → Array V sh e1 → Array V sh e2
     repaVecComp f arr = fromVector (R.extent arr) (f (toVector arr))

     repaDct :: Shape sh => Array V sh Double → Array V sh Double
     repaDct = repaVecComp dct

     -- For some reason repaDct ∘ repaIDct doesn't give me the identity
     ```

```
-- I have to divide it by twice the length
repaIDct :: Shape sh => Array V sh Double → Array V sh Double
repaIDct = repaVecComp ((\v → fmap (/ (fromIntegral (length v) * 2)) v) ∘ idct)
```

  - to get the dct and idct function to work, I had to convert our array to a vector (this is O(1) as it just unpacks!), so I make a generic function called repaVeccomp that does some computation in the vector plane before giving me back the array. With this defined I just pass `dct` for `repaDct` and
    `((\v → fmap (/ (fromIntegral (length v) * 2)) v) ∘ idct)` for idct (offset described above!).

- I can now check if these functions work as expected by the following function

```
testsame :: Bool
testsame = (round <$> toList (id vec)) == [1,2,3,4]
  where id  = repaIDct ∘ repaDct
        vec = fromVector (ix2 2 2) (V.fromList [1,2,3,4])
```

  - This function returns true, and is really just confirms that `repaIDct ∘ repaDct` is really the identity.

- Now we are ready to implement the pass Filters.

```
-- type signature slightly simplied for pdf
genPass :: (Int → Int → Bool) → Int → Array r DIM2 b → Array D DIM2 b
genPass (<>) n arr = R.traverse arr id shrink
  where shrink f sh@(Z :. i :. j)
          | i <> n ∧ j <> n = f sh
          | otherwise       = 0


lowPass :: (Source r b, Num b) ⇒ Int → Array r DIM2 b → Array D DIM2 b
lowPass = genPass (≤)

highPass :: (Source r b, Num b) ⇒ Int → Array r DIM2 b → Array D DIM2 b
highPass = genPass (≥)
```

  - What genPass does is zero out a region if the predicate isn't true, so for the lowPass, only the top left is kept in tact since $i \geq n \wedge j \geq n$ must be true. and for highPass only the bottom right is kept in tact as $i \leq n \wedge j \leq n$ must be true else the region is 0.
    - One mistake that I did earlier was using $\vee$ instead of $\wedge$ which changed some results, which I'll compare and contrast some of it, since notebook1 gives us the intuition to do so.
  - Now let's discuss what frequencies these high and low pass filters filter out. So the low pass filter lets the low frequencies through and zeros out the high frequencies, and if you were to compare our operations to the diagram in notebook1 you'll notice that this corresponds to the region where there is a very slow shift of values. Where as the high pass removes this region and keeps all the fine details.

- Finally we are ready to get some images out of this

```
computeAbsDiff file passedName filter num = do
  x ← readIntoRepa file
  y ← computeVectorP $ R.map (fromIntegral ∘ (+ (- 128)) (repaRGBToGrey x)

  let cosY = repaDct y
  let fileName = passedName <> "-" <> show num <> ".png"

  passThrough ← computeVectorP (filter num cosY) >>= computeUnboxedP ∘ R.map (+ 128)
                                                                     ∘ repaIDct
```

```
difference  ← R.computeUnboxedP $ R.map (abs ∘ (+ 128)) (y -^ passThrough)

saveRepaGrey fileName passThrough
saveRepaGrey ("abs-diff-" <> fileName) difference
```

- So here we read the file, then convert it to grey in parallel. Note how we subtract 128, this is because the dct is best done on values from -128 to 127
- after we set up the data we compute the dct
- We also create the file name here as I will save the output
- Now we call the passthrough we passed in through `filter` then call the inverse to get back the image in its original representation
- From here I compute the difference by subtracting the original image from the filtered image
- I then save the two outputs which we will many of below

3. Image Output and dissection

- all images here and more can be found in the data/lowHighPass folder

(a) Drawn Image

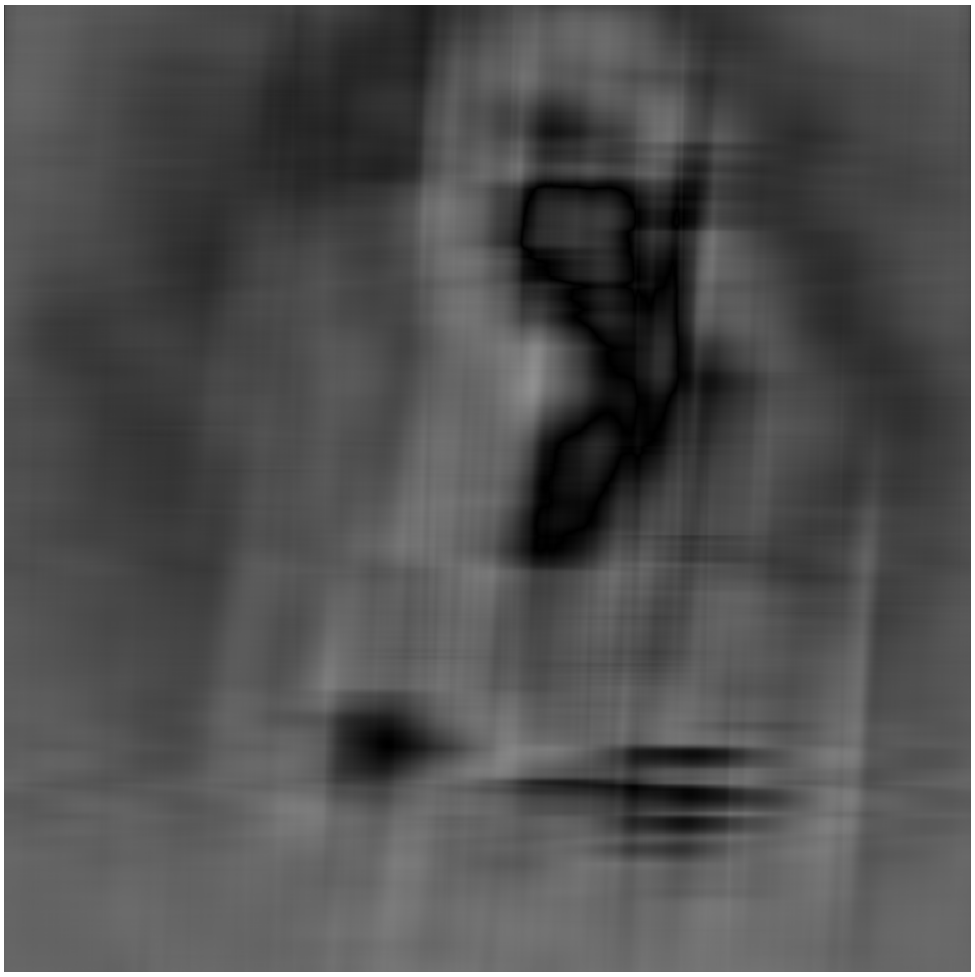
- 
  - here is the original image
- `computeAbsDiff "data/army.jpg" "girllow" lowPass 10`

- so this is the low pass filter with n = 10
- Notice how we keep the overall outline of the girl
- overall not that great on this object

  - here is the absolute difference between the original image and the low pass filter
  - Notice all the detail that was left out of the lowPass

- `computeAbsDiff "data/army.jpg" "girlhigh" highPass 10`

– here I run the high pass algorithm with zeroing out everything passed 10
– Quite a lot of the detail is captured

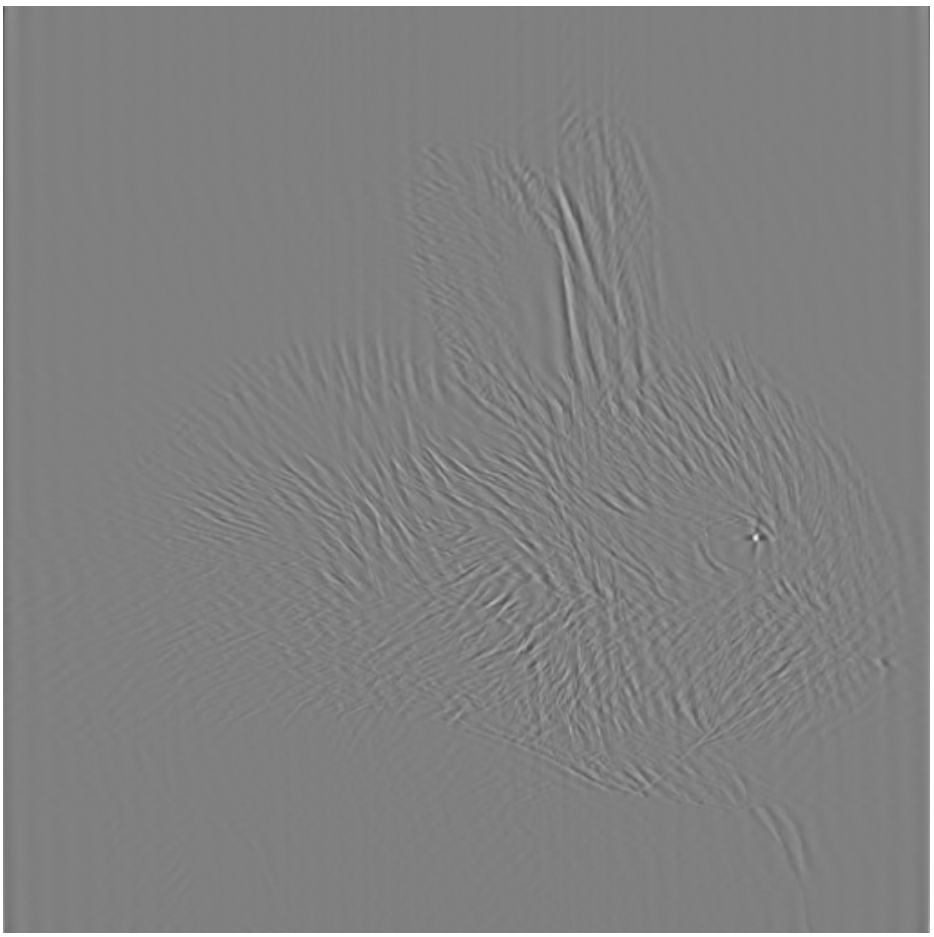


  - This is the difference between the two images, and as can be seen, the overall structure of her hair is formed in the difference demonstrating that bit of information is missed

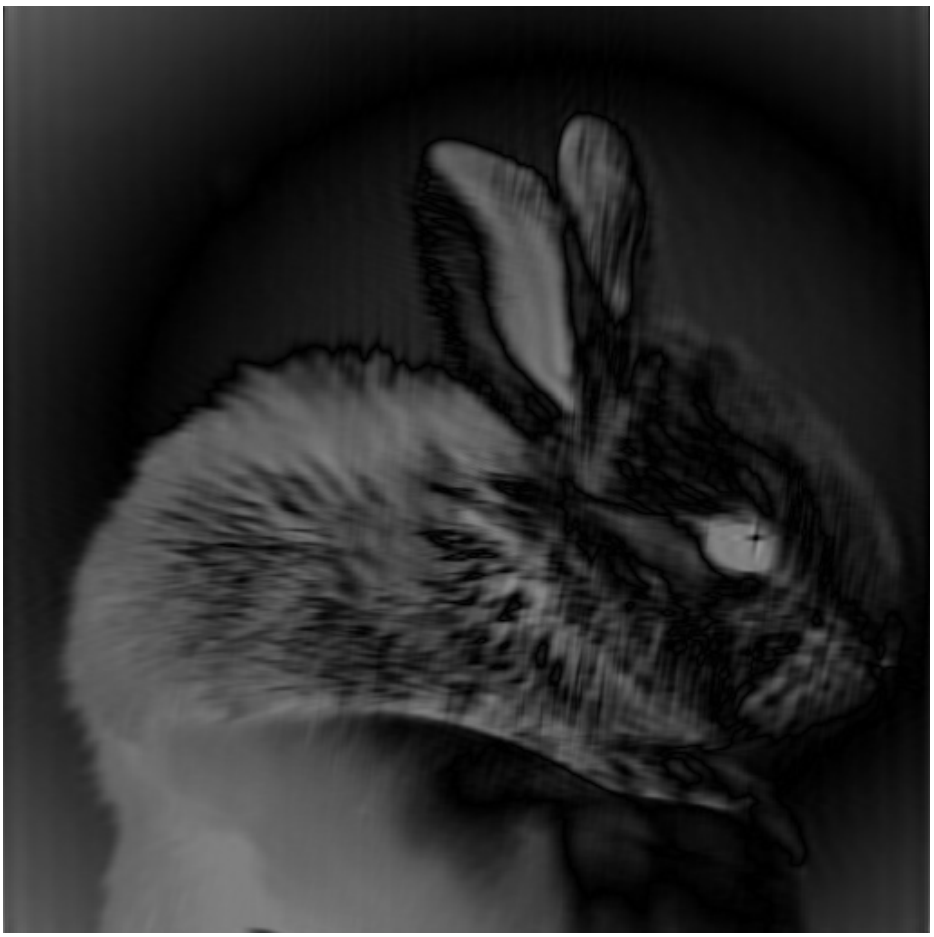
(b) Bunny Image

- Another image I did a lot of tests on was this bunny image, I will post the results of high and low on 20 and 70 respectively.

  - here is the original image
- 70 tests

– 

  * Here we can see the high pass filter on the bunny, it keeps the nice fine details but overall we lose the shape outside of the sharp details.

– 

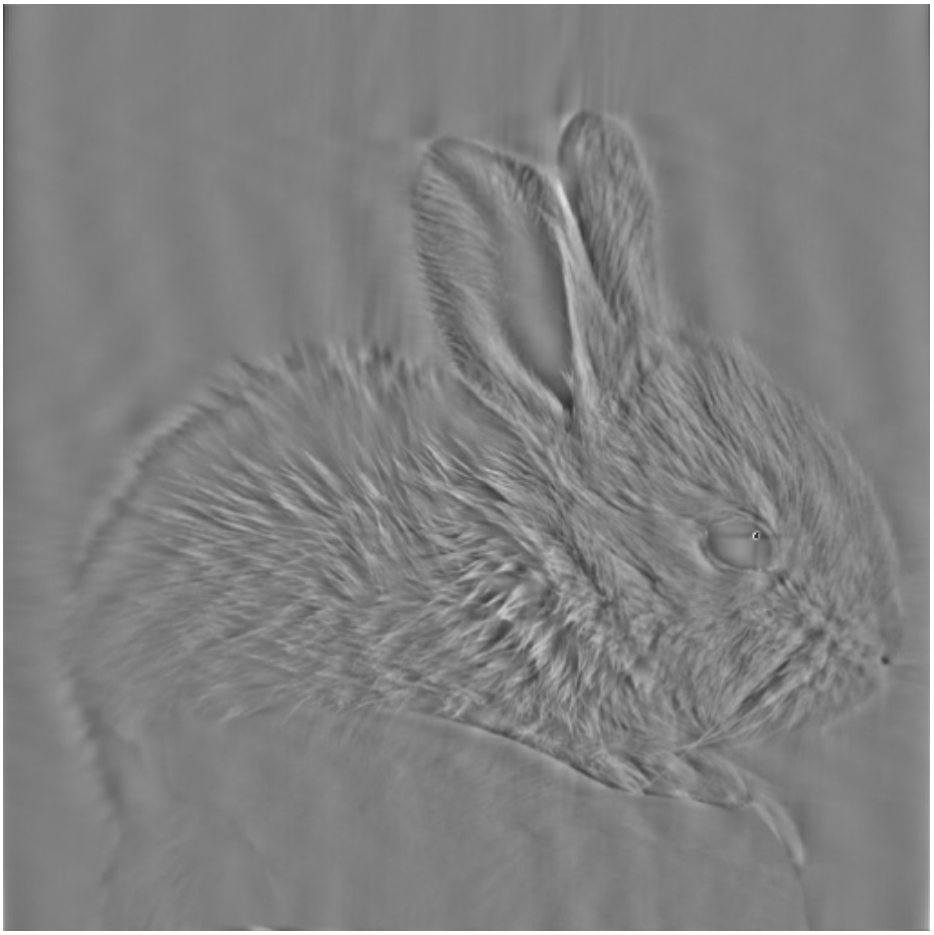
∗ The absolute difference confirms that pretty much most information is lost

– 

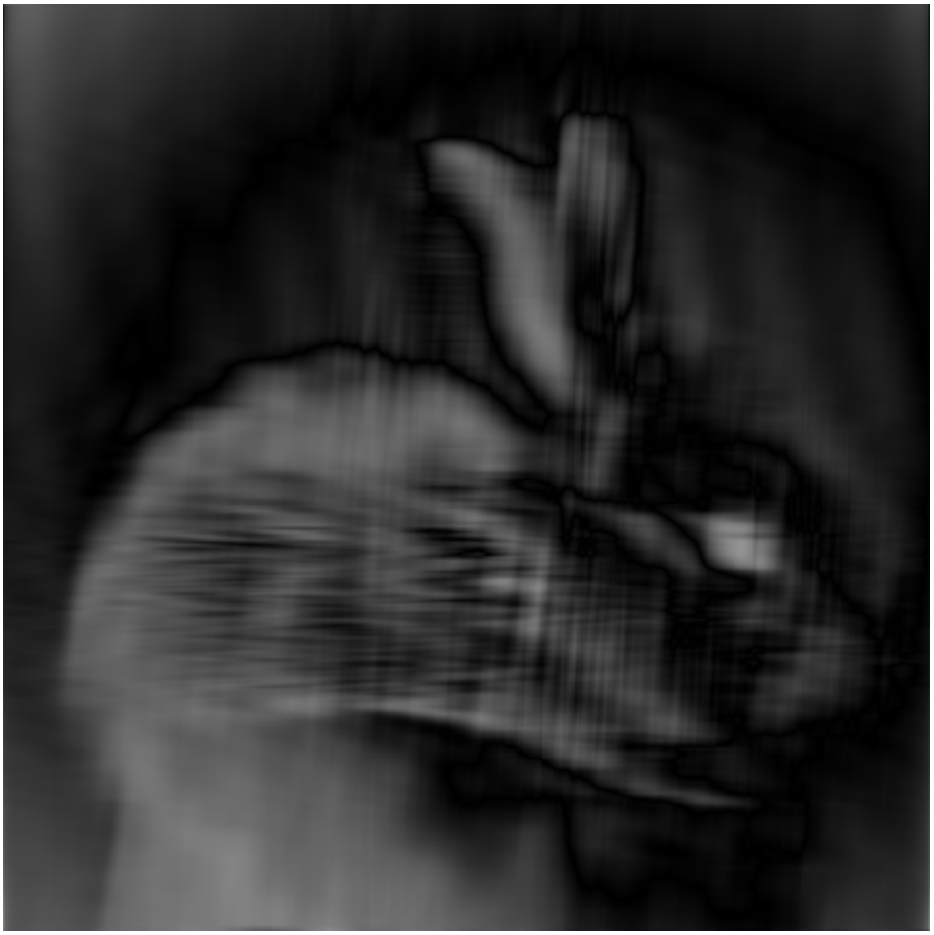
* This is the lowPass Filter on the bunny with 20
* as we can see the overall shape of the bunny is kinda reserved, but all fine detials are missing

- 
    - The Absolute difference confirms this as, as the absolute difference contains a lot of detail
- 20 tests

–



  * this is the high pass filter on the bunny, but instead with a size of 20, compared to 70 more details can be seen and the bunny is starting to look a lot better outlined than with 20

– 

  * This is the absolute difference, compared to the 70 this is a lot more blurry and disjoint, with only parts of the general shape being missing

– 

  * This is the low pass filter with 20, compared to 70 pretty much all detail is gone and only the general shape being intact

–

∗ This is the absolute difference, as can be noticed, the detail of the bunny is much greater.