Notebook1

March 17, 2018

Contents

1 Mathematical Description of the Discrete Cosine Transform 1

2 Implementation and Graphing of the Discrete Cosine Transform 2

1 Mathematical Description of the Discrete Cosine Transform

1. Brief Overview

   - The Discrete cosine transform can represent an image as a sum of sinusoids with frequencies and magnitudes that differ.
   - The Cosine transform has the property that most of the important bits of information with an image (or even an audio wave) is concentrated (Notebook 2 will expand on this idea).
   - Due to this property, the DCT (or the Modified discrete cosine transofrm) is used to MP3 compression and for JPG compression.

2. The Equation

   - The Two dimensional DCT of an M-by-N matrix can be expressed as follows.
     $\mathrm{C_{pq}} = \alpha_\mathrm{p}\alpha_\mathrm{q} \sum_{m=0}^{M-n} \sum_{n=0}^{N-1} A_{mn} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}$
     where
     $0 \leq \mathrm{p} \leq \mathrm{M} - 1$
     $0 \leq \mathrm{q} \leq \mathrm{N} - 1$
     $$\alpha_\mathrm{p} = \begin{cases} 1/\sqrt{M} & p = 0 \\ \sqrt{2/M} & 1 \leq p \leq M-1 \end{cases}$$
     $$\alpha_\mathrm{q} = \begin{cases} 1/\sqrt{N} & p = 0 \\ \sqrt{2/N} & 1 \leq q \leq N-1 \end{cases}$$
   - We can note that if we assign p = n and q = n, they both fit the domain of p and q, and I will be using this for the program

2 Implementation and Graphing of the Discrete Cosine Transform

- So first we must implement the mathematical equation above
  - The following code can be found in Cosine.hs

```
import Data.List.Split

dcBasis :: (Floating p1, Eq p1) => p1 -> p1 -> Int -> Int -> [[p1]]
dcBasis p q m n = chunksOf n $ (*) . (αpq *) <$> ti <*> tj
  where compAlpha x bound
          | x == 0    = 1 / sqrt bound
          | otherwise = sqrt (2 / bound)
        mfloat           = fromIntegral m
        nfloat           = fromIntegral n
        αpq              = compAlpha p mfloat * compAlpha q nfloat
        f offset bound x = cos $ (pi * offset * (2 * x + 1)) / (2 * bound) -- x is
        ti               = f p mfloat . fromIntegral <$> [0..(n-1)] -- an element
        tj               = f q nfloat . fromIntegral <$> [0..(m-1)] -- of a vector
```

    - Lets break down this function line by line, the first function is `compAlpha` which basically abstracts out the logic for α_p and α_q seen in the mathematical equation, we can see the application of this in `apq` which just multiplies a_p by a_q basically. As described in the math we just set p and q to m and n respectively.
    - `mfloat` and `nfloat` basically just turn m and n into a float respectively.
    - `f` is once again another abstraction that abstracts out the $\cos \frac{\pi(2m+1)p}{2M}$ and the n variant seen in the mathematics.
    - thus `ti` and `tj` computes this abstraction for all values from 0 to N-1 for the one dealing with n's (while also giving the correct p value) and 0 to M-1 for the one dealing with m's (while also giving the correct q value).
    - Now all we have to do is combine our computations which comes in `chunksOf n $ (*) . (αpq *) <$> ti <*> tj`.
      the `(*) . (αpq *)` is really a function that takes 2 numbers before becoming a value so, `(*) . (αpq *) <$> ti` has the type signature [Num a ⇒ a → a] (as `<$>` is just map). Now, `<*>` we can think of as the generalized cross product in which the order can be seen in the example below. Noting this, we can now notice that the nested summation in reality for creating indices is exactly the same as using the applicative (`<*>`) with the function containg the m points and the right argument containg the n points for the computation of a nxm long matrix. The final part of `chunmksOf n` simply just formats the data into a list that is really nxm instead of flat

· `[(1 +), (-) 99] <*> [1,2,3] ≡ [2,3,4,98,97,96]`

- Now that we have the formula up, we can now easily create any nxn matrix asked (the prompt asks for 16x16, but I'll make it generic)

```
nbyN :: (Floating a, Eq a, Enum a) => Int -> [[[[a]]]]
nbyN n = chunksOf n $ (\x y -> dcBasis x y n n) <$> [0..(fromIntegral n-1)]
                                                <*> [0..(fromIntegral n-1)]
eightByEight :: [[[[Double]]]]
eightByEight = nbyN 8
```

  - here we create the NxN matrix by simply calling dcBasis with all permutations of x and y, we do this by applying the same trick as above (refer to the above explanation for the `f <$>` x_1 `<*>` x_2 pattern).
  - Notice that our structure is not a 2d vector instead a 2d list containing 2d lists.

- Now that we have the data, we must normalize it, because if we look at the last row in the last 2d list, we see their values really are too small to effectively Grey scale them.

  - `eightByEight !! 7 !! 7 !! 7` ≡ [-9.515058436089173e-3,2.7096593915592444e-2,-4.055291860268229e-2,4.7835429045636306e-2,-4.78354290456363e-2,4.055291860268227e-2,-2.7096593915592413e-2,9.515058436089185e-3]
  - so in the file `Normalize.hs` I make a normalization type and a function to compute such a type

```
import Misc
-- Really only going to use ZeroC for now, but can expand later
data Normalize = Zeroc
               | Same


normalizeValues :: (Functor f1, Fractional b, Foldable f1, Ord b) => Normalize -> f
normalizeValues Same  xss = undefined
normalizeValues Zeroc xss = (/ greatest) <$$> xss
  where flattened = concat xss
        greatest  = maximum (abs <$> flattened)
```

    * so I only have two cases for the Normalize type, either it's normalized at zero or it's the same everywhere, for simplicity I only defined the normalizeValues for centering at zero. It's a really simple formula, I just do element wise division (`<$$>` is a helper function that just composes 2 maps so I can get the elements in a doubly nested structure) from the greatest value after taking the absolute value.

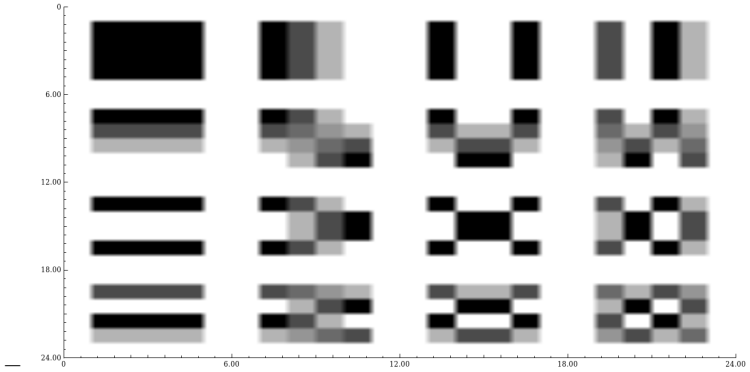
    - Notice that this only works on a 2d list, and not our 4d list, we'll see how I handle this in the next section

- Finally we are ready to plot our Cosine transform, it took me a while to actually make this as I had to search for the correct plotting function.

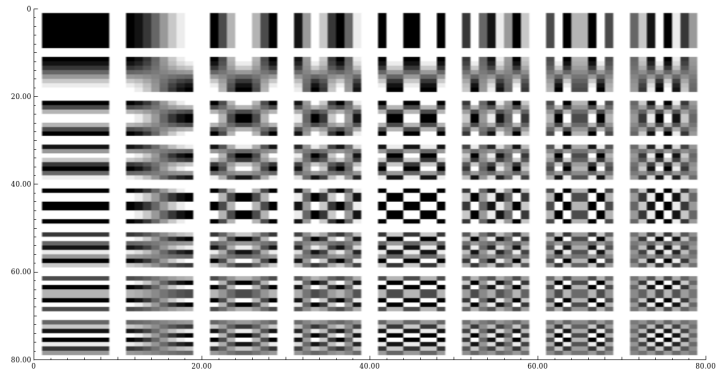
```
-- the fmap ((-1) :) . (<> [-1]) is for padding the top and bottom
-- this pads the left and right (padList <>) . (<> padList)
plotDCT :: Int -> IO ()
plotDCT n = imshow (fromBlocks (fromLists <$$> padded))
  where padded  = fmap (((-1) :) . (<> [-1])) . (padList <>) . (<> padList) <$$> norm
        padList = [replicate n (-1)]
        norm    = normalizeValues Zeroc <$$> nbyN n -- name shortened for pdf
```

    - Here we create norm, which is just the normalized vector, we double map through the 4d list so we can treat each section as a 2d list
    - from here I pad the list by double mapping this norm by the strategy described in the comments.
    - Now I double map into this padded array to turn each section into a matrix with the double type, and then the fromBlocks function handles converting the doubly nested list of matricies into a single matrix which finally allows imshow to graph it.
    - Note that this algorithm can take an `n`, so we can map any NxN discrete matrix

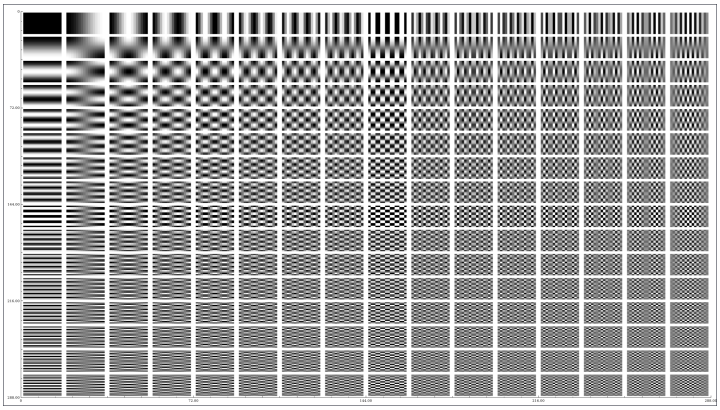
- Time to show off the graphs! Note that white on other people's versions is black here



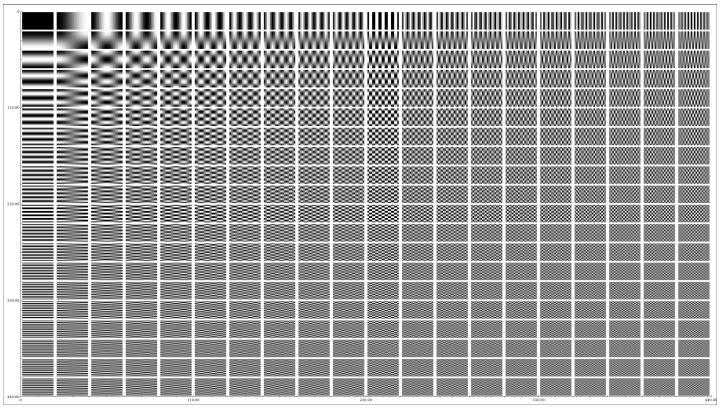
    - So this one here is the 4x4 Discrete Cosine Transform, really this one is quite low res and thus doesn't carry too much detail, but we can see the checker board shape starting to emerge on the bottom right

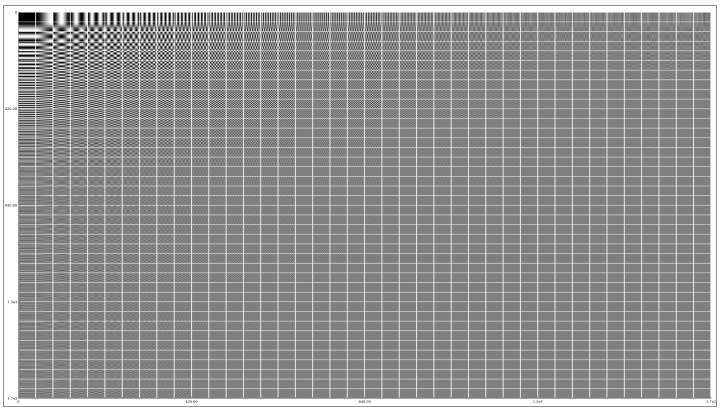


* This is the 8x8 Discrete Matrix Cosine Transform, notice that 4,4 is the most clear. This pattern will come up in the rest.



* This is the 16x16 which was requested by the assignment, as can be seen, as you go right and down the checker board gets more and more fine. Also notice that the 8,8 one is the most clear matrix, and that on the top row and left column, there are only vertical and horizontal lines respectively





* I also plotted the 40x40 and the 20x20 for fun, it might be hard to see in the pdf, but the 20,20 and the 10,10 transforms are the most clear