notebook1

February 11, 2018

Contents

1 Overview of the Blurring filter **1**

2 Implementing the Blurring filter **2**

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 image
       - 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 my 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 backwords 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

1.