Heeger-Bergen Wavelets

Liping Yin Albert Chua

December 13, 2021

1 Introduction

The point of this paper is discussing the implementation of wavelets described by Heeger and Bergen in [1]. Our implementation will be based on [2]. We will first implement a serial version, generalize it to use OPENMP in computationally expensive sections, and then attempt to use MPI and OPENMP together in the final version. A link to the Github is provided at the project Github. Contributions were made based on commits in the Github.

2 Filter construction

In this section we describe the mathematical formulation of the filter bank. All filters are defined in frequency with support in $[-\pi,\pi]^2 = [-\pi,\pi]\times[-\pi,\pi]$ using polar coordinates (r,θ).

Define the following functions:

$$\forall\, r \geq 0, \quad \hat{L}(r) := \begin{cases} 1 & r \leq \pi/2 \\ \cos\left(\frac{\pi}{2}\log_2\left(\frac{2r}{\pi}\right)\right) & \pi/2 < r \leq \pi \\ 0 & r \geq \pi \end{cases},$$

$$\forall\, r \geq 0, \quad \hat{H}(r) := \begin{cases} 0 & r \leq \pi/2 \\ \cos\left(\frac{\pi}{2}\log_2\left(\frac{r}{\pi}\right)\right) & \pi/2 < r \leq \pi \\ 1 & r \geq \pi \end{cases}.$$

The first function is a low pass filter that is only nonzero in a disk around the origin. The second function is a high pass filter that is nonzero outside of this disk.

We also have dilations of the functions given above as:

$$\forall\, j \in \mathbb{Z}, \quad \hat{L}_j(r) := \hat{L}(2^j r) = \begin{cases} 1 & r \leq \pi/2^{j+1} \\ \cos\left(\frac{\pi}{2}\log_2\left(\frac{2\cdot 2^j r}{\pi}\right)\right) & \pi/2^{j+1} < r \leq \pi/2^j \\ 0 & r \geq \pi/2^j \end{cases},$$

and

$$\forall\, j \in \mathbb{Z}, \quad \hat{H}_j(r) := \hat{H}(2^j r) = \begin{cases} 0 & r \leq \pi/2^{j+1} \\ \cos\left(\frac{\pi}{2}\log_2\left(\frac{2^j r}{\pi}\right)\right) & \pi/2^{j+1} < r \leq \pi/2^j \\ 1 & r \geq \pi/2^j \end{cases}.$$

Now we create a set of directional cones. Let Q be the number cones. Then define

$$\hat{G}_q(\theta) := \alpha_Q \left(\cos\left(\theta - \frac{\pi q}{Q}\right)^{Q-1} \mathbb{1}_{|\theta - \pi q/Q| \leq \pi/2} + \cos\left(\theta - \frac{\pi(q-Q)}{Q}\right)^{Q-1} \mathbb{1}_{|\theta - \pi(q-Q)/Q| \leq \pi/2} \right),$$

where $0 \leq q < Q$ and $\alpha_Q := 2^{Q-1}\frac{(Q-1)!}{\sqrt{Q(2(Q-1))!}}$.

Now we can make filters. Define

$$\widehat{\psi_{j,q}}(r,\theta) = \hat{L}_j(r)\hat{H}_{j+1}(r)\hat{G}_q(\theta),$$

which are defined in frequency. Define ℓ_J, h_J, and $\psi_{j,q}$ to be the space representations of the filters defined above. These filters are found in space by taking the inverse Fourier Transforms of $\hat{\ell}_J$, $\hat{h}_J$, and $\hat{\psi}_{j,q}$. Then consider the set

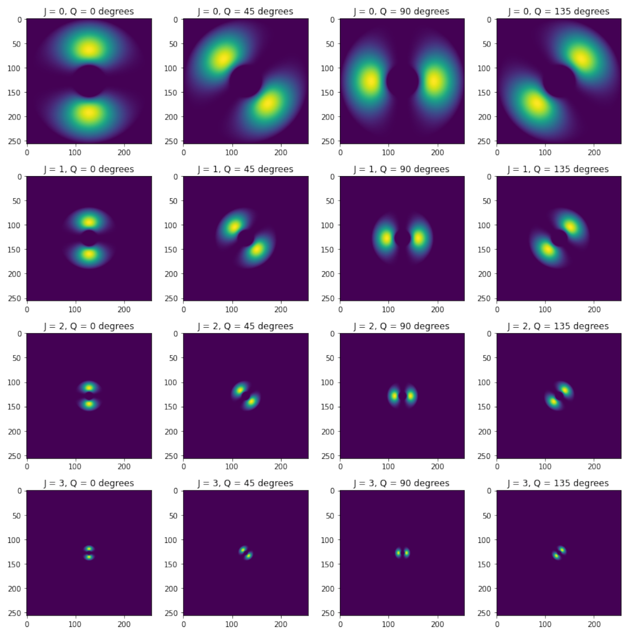
$$\hat{F}_{J,Q} = \{\hat{h}_J, \hat{\ell}_J, \hat{\psi}_{j,q} : 0 \leq j < J, 0 \leq q < Q\}.$$

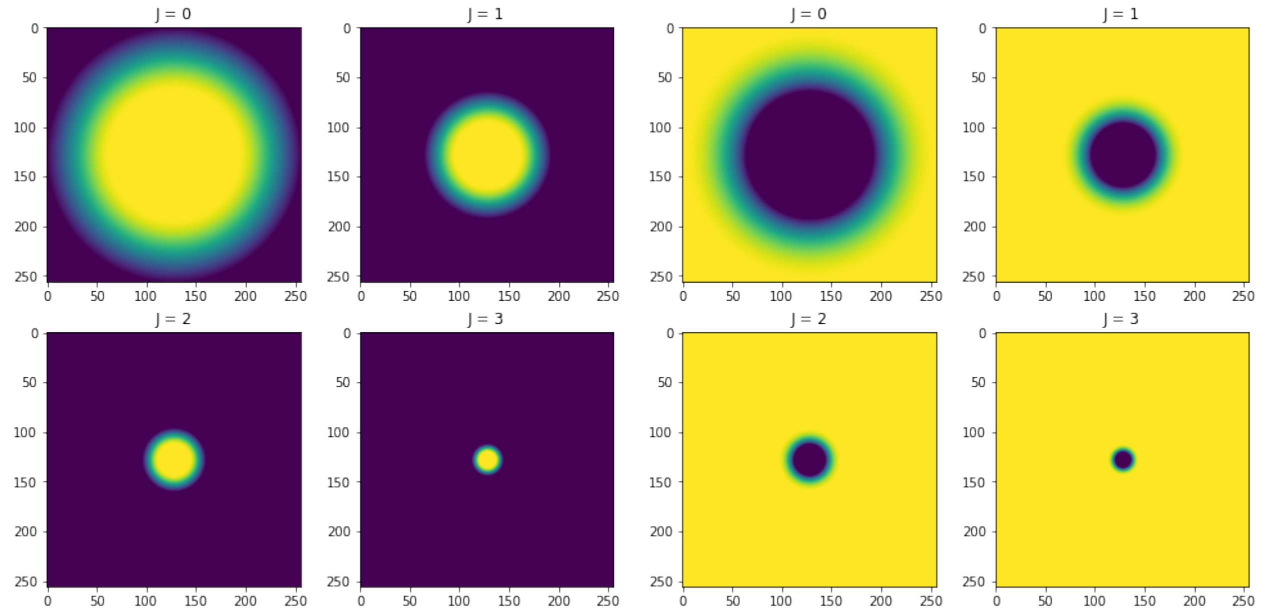
The wavelet transform of an image f is given by

$$W_{J,Q}f = \{f * \hat{h}_J, f * \hat{\ell}_J, f * \hat{\psi}_{j,q} : 0 \leq j < J, 0 \leq q < Q\}.$$

Define $\mathcal{F}$ as the Fourier Transform of a function and $*$ as convolution. The general idea for the implementation is to use the convolution theorem, which is the following: given f and g, under some stupid mathematical restrictions, we have $\mathcal{F}(f*g) = \mathcal{F}(f)\cdot\mathcal{F}(\})$. From here, we simply invert the Fourier Transform to get $f * g = \mathcal{F}^{-1}(\mathcal{F}(f)\cdot\mathcal{F}(\}))$, where $\mathcal{F}^{-1}$ denote the Inverse Fourier Transform. The reason we don't do the convolution using tiny filters is that it is generally more computationally intensive since a discretized Fourier Transform is at worst $O(N^3)$ (if N is the width of the square image) even without using a Fast Fourier Transform.

Here is a picture of the directional filters, lowpass filters, and high pass filters (respectively) with the scale and direction indicated:





In these pictures, blue represents zero and nonblue represents a positive number. The brighter the color, the larger the pixel value. These were taken from Albert's github from another implementation.

Let's describe the general implementation of the algorithm now:

```
procedure Heeger-Bergen Wavelet Transform
    Take in image $f$
    Generate Filter Bank
    Calculate $f * \ell_J$
    Calculate $f * h_J$
    for $j = 0 : (P - 1)$ do
        for $q = 0 : (Q - 1)$ do
            Calculate $f * \psi_{j,q}$
```

Note that it is possible to speed this up by calculating the fourier transform of the image and save it, but we didn't do that for this implementation since we didn't notice that this wasn't done. This will only provide marginal speed up in our tests though. Given the algorithm above, let's describe how this would actually work on a computer.

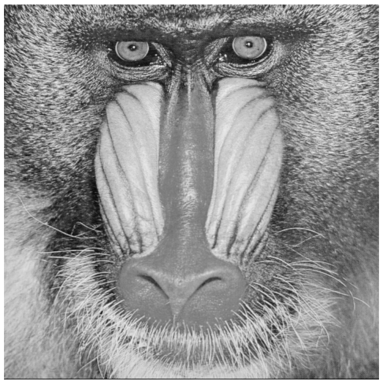
Take an image f. We can discretize this as an $N \times N$ array. Implement a discretization of $\hat{\psi}_{j,q}(u)$ as $N \times N$ filters. Then take the Discrete Fourier Transform of f and call it $\hat{f}$. Now take the hadamard product of $\hat{f}$ with each filter in $\hat{F}_{J,Q} = \{\hat{h}, \hat{\ell}, \hat{\psi}_{j,q} : 0 \leq j < J, 0 \leq q < Q\}$. Invert the Fourier transform and take the real part of the result. This gives us the convolution of f with the filter.

3 Serial Implementation

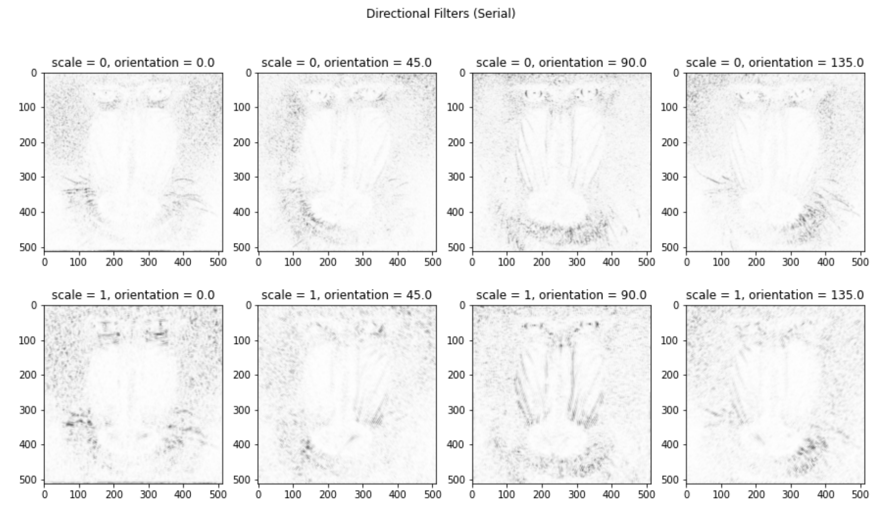
Our implementation is completely from scratch. We implement the convolution, Discrete Fourier Transform, Inverse Discrete Fourier Transform, generation of the filters, and all the matrix operations with naive C++

implementations. All of these processes are also parallelized later on. Please have mercy. There was almost 700 lines to implement all the matrix operations just for the serial code.

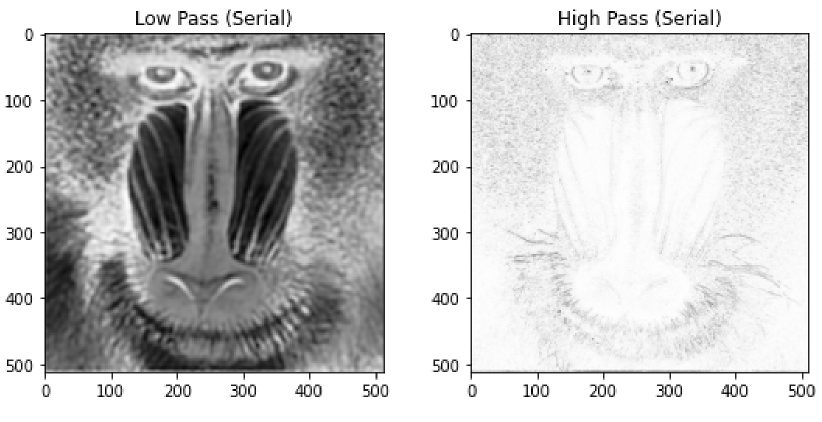
The speed will not be a fast as when we used packages. The point of not using packages was to try and learn how to effectively use C++ in scientific computing. Our first implementation is a serial implementation. We test on a 512 by 512 image of a baboon given below with $J = 2$ and $Q = 4$. In total, there are 10 images from the transform. They are saved into a text file and then matplotlib is used to actually print the images.



Here is the convolution with the directional filters using our serial implementation:

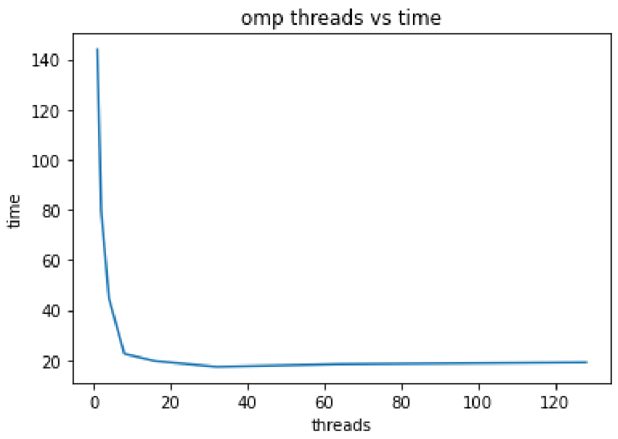


The low and high pass filters are also given below:



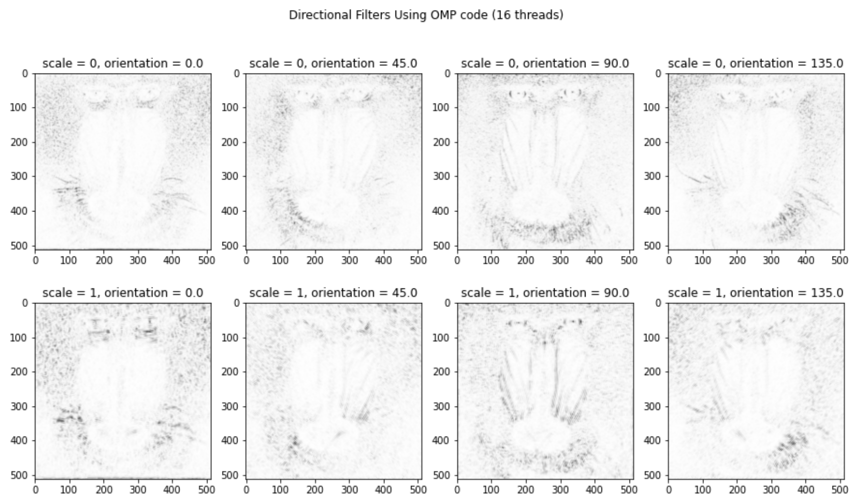
4 OMP Implementation

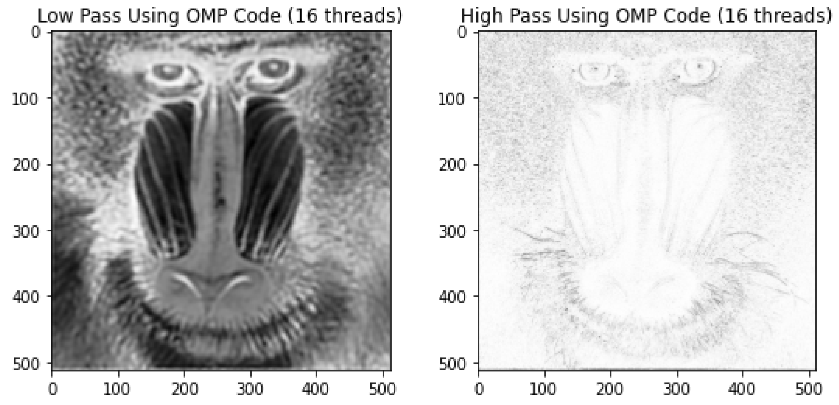
Next, we attempt on OPENMP implementation. The idea for making our algorithm go faster is to use OPENMP to speed up all the loops. In particular, when convolve the image with all the filters, we add a parallel region to do them simultaneously. Also, there is parallelization added when generating the filters and doing the matrix operations using parallel regions. The result with serial code is approximately 140 seconds over multiple runs. Here is a graph of thread number versus run time.



There is a bottleneck since the filters are not generated completely in parallel. We tried to use a loop to generate them, but there was a race condition in the implementation of the filters that we did not manage to fix. There is probably a simple way to fix it, but we ran out time. Our goal of making our code "go fast" was achieved since the run time is 7 times faster after using approximately 10 threads.

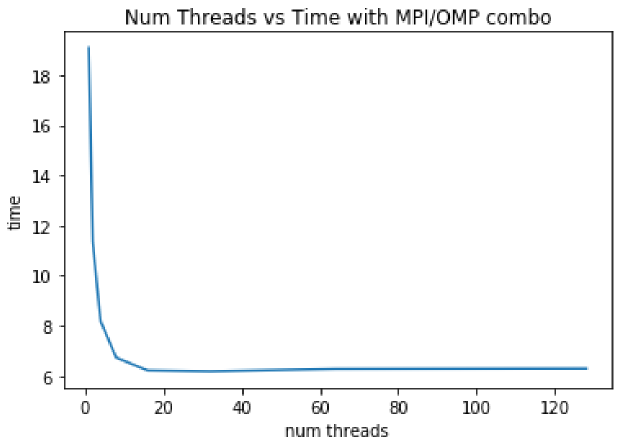
Here are some pictures of our results using 16 threads to verify that the results match the serial code. There is also a supplement called "testFinalProject.ipynb" with our results.


Directional Filters Using OMP code (16 threads)
scale = 0, orientation = 0.0
scale = 0, orientation = 45.0
scale = 0, orientation = 90.0
scale = 0, orientation = 135.0
scale = 1, orientation = 0.0
scale = 1, orientation = 45.0
scale = 1, orientation = 90.0
scale = 1, orientation = 135.0

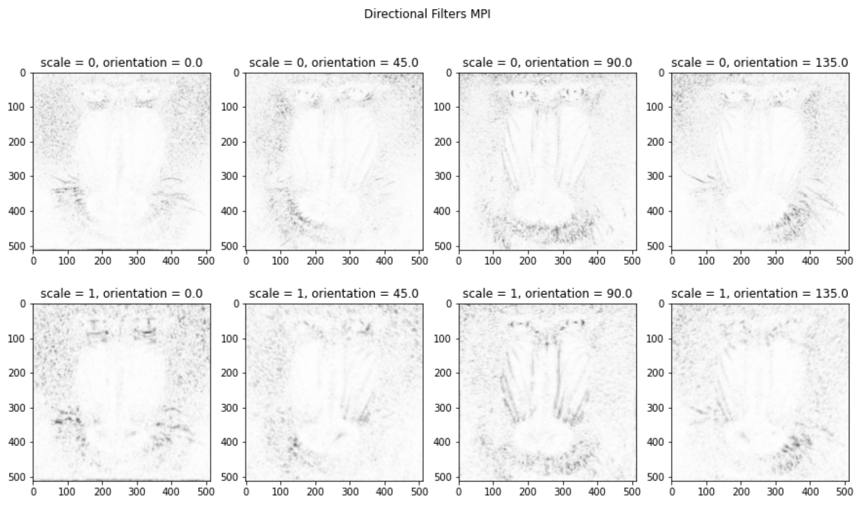

Low Pass Using OMP Code (16 threads)
High Pass Using OMP Code (16 threads)

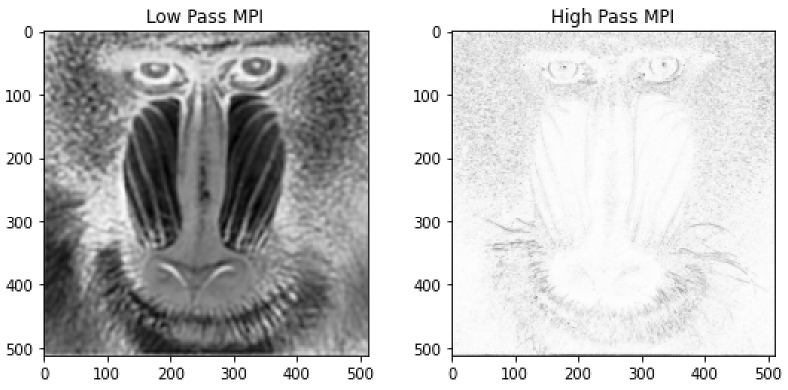
5 MPI Implementation

Lastly, we combined OPENMP and MPI in one implementation. The idea for our implementation is to make $PQ + 2$ ranks, which is the number of convolutions we need to do. We make each rank do a convolution. In each of the ranks, we use OPENMP for threading the matrix operations. The idea is to unify distributed and shared memory. This made our code go even faster based on the graph given below:



To check our results, here are some pictures for comparison too:

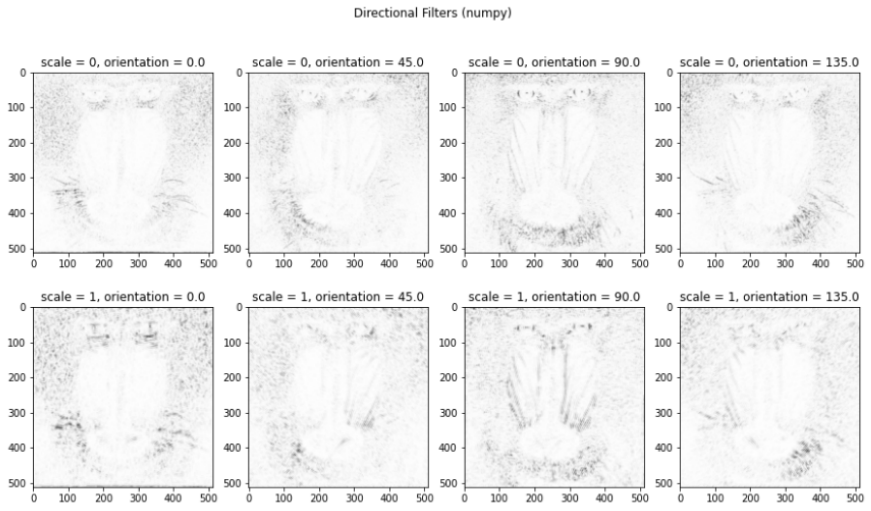

Directional Filters MPI
scale = 0, orientation = 0.0
scale = 0, orientation = 45.0
scale = 0, orientation = 90.0
scale = 0, orientation = 135.0
scale = 1, orientation = 0.0
scale = 1, orientation = 45.0
scale = 1, orientation = 90.0
scale = 1, orientation = 135.0

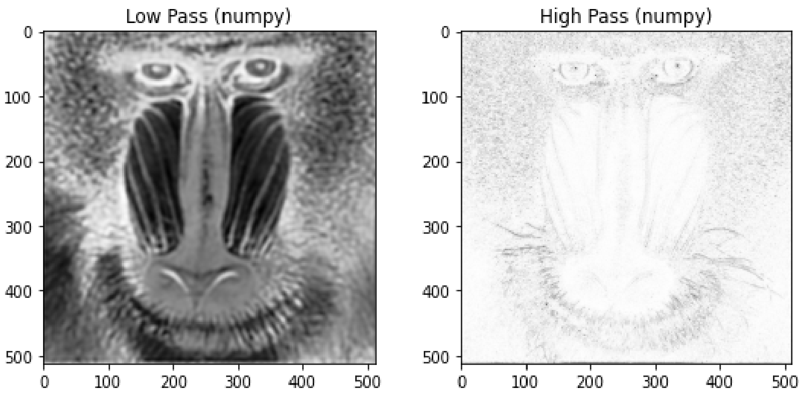


There is a slight flaw in our program since we actually load the original image $PQ + 2$ times instead of broadcasting. This was done to save time. As we can see, in all three implementations, the results are the same.

6 Numpy Results

Lastly, we take some code from Albert's github with a python implementaiton using numpy. Here are the results:


Directional Filters (numpy)
scale = 0, orientation = 0.0
scale = 0, orientation = 45.0
scale = 0, orientation = 90.0
scale = 0, orientation = 135.0
scale = 1, orientation = 0.0
scale = 1, orientation = 45.0
scale = 1, orientation = 90.0
scale = 1, orientation = 135.0



These were run on google colab with a single processor, and the time taken was about 16 seconds. Based on this result, we still have some work to be done. The main problem with our code is that the matrix operations and discrete fourier transform are too slow. If we were to use C++ implementations, our code would probably be much faster.

7 Conclusions and Ideas for Improvement

In all our parallel code, we manage to "go faster" than a serial implementation or even a numpy implementation. However, our code doesn't use memory efficiently and doesn't take advantage of computing power efficiently. If we had more time, we would try to speed up our matrix multiplication/hadamard product/other matrix operations. We would also use a Fast Fourier Transform implementation instead of the Discrete Fourier Transform since the runtime is $O(N^2 \log^2 N)$ instead of $O(N^3)$. Also, we would like to extend our implementation to images that aren't square. Please have mercy on us. We tried hard and did make working parallel code on a semi-original project.

References

[1] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 229–238, 1995.

[2] Thibaud Briand, Jonathan Vacher, Bruno Galerne, and Julien Rabin. The Heeger-Bergen pyramid-based texture synthesis algorithm. *Image Processing On Line*, 4:279–299, 2014.