Michael Christensen
November 11, 2013
CS 465

Homework #6 – Filtering

Written Exercises:

1. *Compute the Fourier transform of f(x,y) = sin(x) sin(y). Note that there's a hard way and a couple of easy ways to do this – see if you can find one of the easy ways instead of actually solving integrals.*

- *f(x,y) = sin(x) sin(y)*, where x = 2*pi*s*x\`, y = 2*pi*t*y\`, for some x\`, y\`
- By the Convolution Theorem (and since linearly separable), g = fh → G = F*H.
- Therefore, if we let g = f(x,y), f = sin(x), and h = sin(y), G = *Fourier(g)* = *Fourier*(sin(x))**Fourier*(sin(y)).
- H = Fourier(sin(x)) = ½i[δ(u+t) – δ(u-t)]
- F*H = (½[δ(u+s) - δ(u-s)])*(½*i*[δ(u+t) – δ(u-t)])
- **F*H = ¼*i*[δ(u+s)*δ(u+t) – δ(u+s)*δ(u-t) - δ(u-s)*δ(u+t) + δ(u-t)*δ(u-t)] (not sure if you wanted this in a more simplified form. Speaking with the TA I also feel that he said either way was fine)**

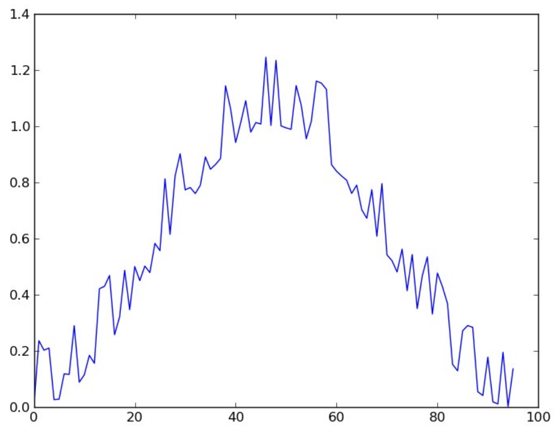
2. *What is the Fourier transform of f(t) = cos(16*pi*t) cos(64*pi*t)?*

- By the Convolution Theorem, g = f*h → G = FH; f = cos(2*pi*8*t), h = cos(2*pi*32*t)
- G = *Fourier*(g) = *Fourier(*f)*Fourier*(h)
- *Fourier*(f) = ½[δ(u+8) + δ(u-8)]
- *Fourier*(g) = ½[δ(u+32) + δ(u-32)]
- G = ½[δ(u+8) + δ(u-8)]*½*i*[δ(u+32) + δ(u-32)] = **¼*i*[δ(u+8)*δ(u+32) + δ(u+8)*δ(u-32) + δ(u-8)*δ(u+32) + δ(u-8)*δ(u-32)]**

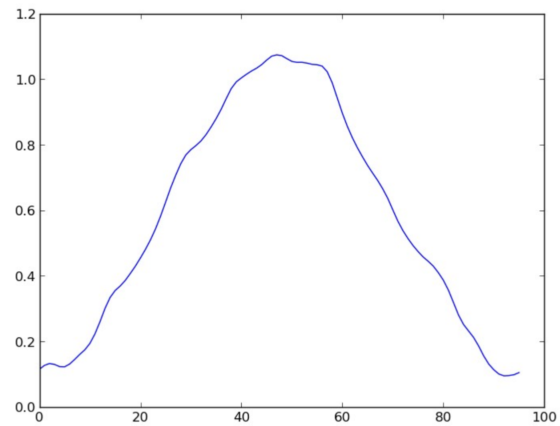
Programming Exercises:

Part A: 1-D Filtering:

Original Signal:



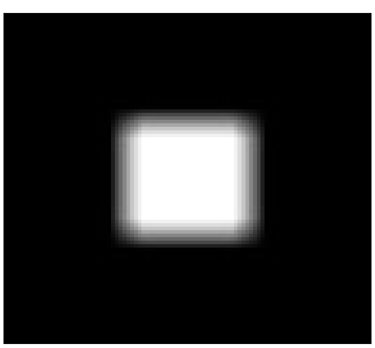
Filtered Signal:



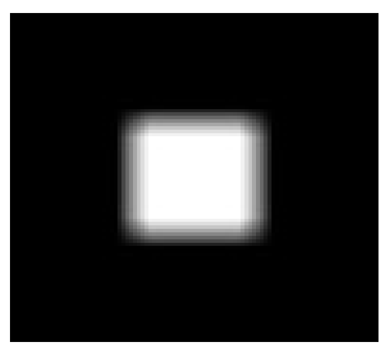
Explanation: I used a filter based on the Butterworth Lowpass Filter described in section 4.8.2 of the textbook. The main difference between what I implemented and that described in the book is that since this was a 1-dimensional signal, I only needed to use 1-dimension in my filter. The book mentions a radius *D0* (pages 269, 273), the circle within with the “filter passes without attenuation all frequencies”; the 1-dimensional equivalent is just a 1-d linear distance, which I set to 10. However, I noticed that changing this variable did not result in any noticeable differences in the output image. Basically, this filter is a transfer function that cuts off frequencies *D0* distance from the origin, thereby removing the sharp spikes seen in the original image because those spikes are composed of high-frequency sinusoids and therefore farther away from the origin in the frequency/Fourier domain. The overall smoothness of the image is visible because the sinusoids making up the 1-d smooth signal are within that *D0* distance of the origin.

Part B: 2-D Filtering/Convolution Theorem:

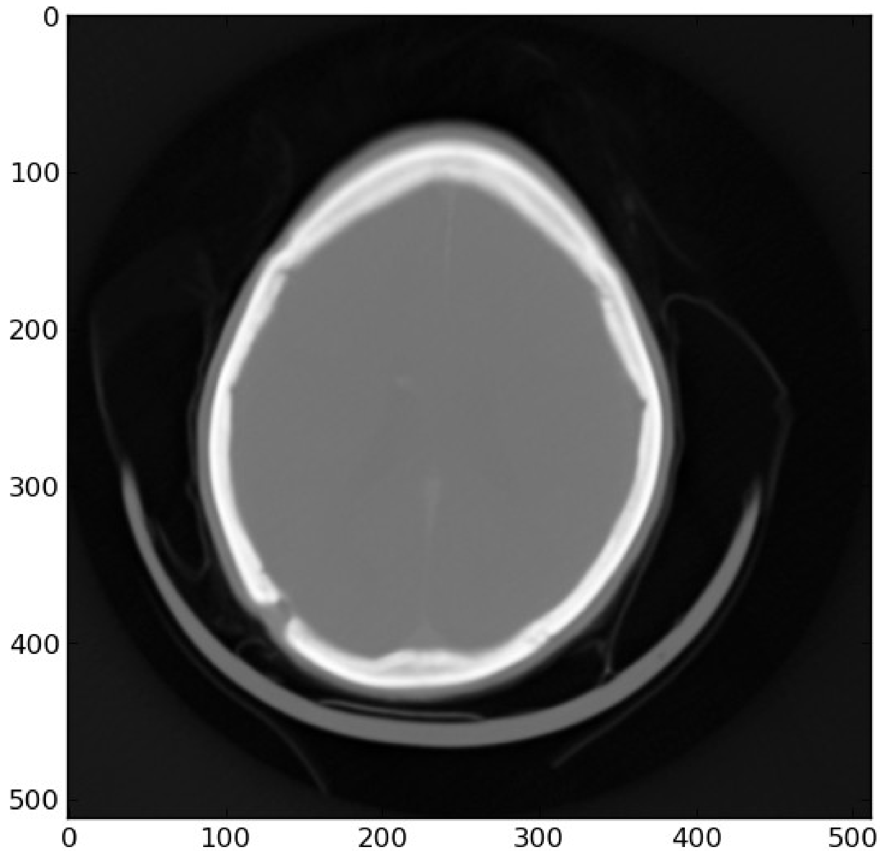
WhiteBox after spatial filtering from <u>homework 3</u>:



WhiteBox after filtering in frequency domain:



Part C: Interference Pattern:



Challenges: Part C was the most difficult part for me and took the majority of my time. I struggled finding the most out of place frequency, mainly due to my lack of a good heuristic and therefore not having a good place to start. The written exercises weren't difficult, assuming I did them correctly and I didn't misinterpret exactly how far we needed to get in the calculation.

Time:
Written: 2 hours to read, think, do, redo, read, etc.
Programming: 7 hours (lots of time figuring out and debugging Part C)

Code:
```
import numpy as np
import matplotlib.pyplot as pl
import matplotlib.cm as cm
import pickle as pk
import scipy.ndimage.interpolation as sy

# Design 1-D low-pass filter to smooth 'HW6_PA.pkl'
def parta():
    # Load image/signal
    #
    signals = pk.load(open('HW6_PA.pkl', 'r'))
    pl.plot(signals)
    pl.show()

    # FFT, shift to be zero-centered
    #
    Fu = np.fft.fft(signals)
    Fushifted = np.fft.fftshift(Fu)

    # Do frequency filtering, where Hu is the transfer function
    # Using a Butterworth Lowpass Filter:
    # H(u,v) = sqrt[1 / (1 + [D(u,v)/D_0]^2n)], where
    # D(u,v) = [(u - P/2)^2 + (v - Q/2)^2], where
    # u = 0, 1, 2, ... P - 1 and v = 0, 1, 2, ... Q - 1, and
    # D_0 is the distance from the origin of the cutoff frequency
    # Arbitrarily choose n = 2, radii = ?
    #
    Hu = np.zeros(Fu.shape)
    P = Fu.shape[0]
    n = 2
    D_0 = 10.0
    for u in range(0, Hu.shape[0]):
        du = np.power((u - P/2), 2)
        Hu[u] = np.sqrt(1/np.power((1 + du/96.0), 2*n))
    Gu = np.multiply(Hu, Fushifted)

    # Inverse FFT to convert back to spatial domain
    #
    Guunshifted = np.fft.ifftshift(Gu)
    gt = np.fft.ifft(Guunshifted)

    # Save/show it
    #
    pl.plot(gt)
    pl.show()

# Use Convolution Theorem to implement a 9x9 filter
# (uniform spatial averaging filter) in the Frequency
# Domain.a?
def partb():
    # By the Convolution Theorem, f*g = Fourier(f)Fourier(g).
    # So we if f is original signal, and g is the 9x9 filter,
    # by convoluting f*g, we are really multiplying their Fourier transforms.
    # Then just take the inverse to get it back in the spatial domain.

    # Spatial domain
    wb3d = pl.imread('whitebox.png')                    #f
```

```
    #There is most assuredly an easier way to do this, but time is of the essence
and I don't want to study man pages right now
    wb = np.zeros((wb3d.shape[0], wb3d.shape[1]), dtype=np.double)
    for a in range(wb.shape[0]):
        for b in range(wb.shape[1]):
            wb[a][b] = wb3d[a][b][0]

    WBU = np.fft.fft2(wb)
    #WBUshifted = np.fft.fftshift(WBU)
    #pl.imshow(np.abs(WBUshifted), cmap=cm.Greys_r)
    #pl.show()

    # spatialfilter = np.ones((9,9))
    # spatialfilter = np.pad(spatialfilter, (wb.shape[0]-9)/2, 'constant')
    # After speaking with Ty, he said that this was the correct way to create the
filter.
    # Before, I was creating a 9x9 matrix of ones, padding it on all sides with
zeros. This
    # way, I put the 9x9 matrix split into the four corners
    spatialfilter = np.zeros(wb.shape)
    spatialfilter[0:5,0:5] = np.ones((5,5))
    spatialfilter[spatialfilter.shape[0]-4:spatialfilter.shape[0], 0:5] =
np.ones((4,5))
    spatialfilter[0:5,spatialfilter.shape[1]-4:spatialfilter.shape[1]] =
np.ones((5,4))
    spatialfilter[spatialfilter.shape[0]-4:spatialfilter.shape[0],
spatialfilter.shape[1]-4:spatialfilter.shape[1]] = np.ones((4,4))
    GBU = np.fft.fft2(spatialfilter)
    # No need to shift either the filter nor the original signal's fourier because
I'm not displaying it
    # GBUshifted = np.fft.fftshift(GBU)
    # pl.imshow(np.abs(GBUshifted), cmap=cm.Greys_r)
    # pl.show()

    ResultU = np.multiply(WBU, GBU)

    resultt = np.fft.ifft2(ResultU)

    pl.imsave('Part B', resultt, cmap=cm.Greys_r)

def calculate_normal(img, kernel, row, col):
    avg = 0
    a = int((kernel.shape[0] - 1) / 2)
    b = int((kernel.shape[1] - 1) / 2)
    for s in range(-a, a+1):
        for t in range(-b, b+1):
            if (row+s < 0) or (row+s >= img.shape[0]) \
                or (col+t < 0) or (col+t >= img.shape[1]):
                    avg += 0
            else:
                avg += img[row + s, col + t] * kernel[a + s, b + t]
    return avg / kernel.size

def partc():
    ifimg3d = pl.imread('interfere.png')
    ifimg = np.zeros((ifimg3d.shape[0], ifimg3d.shape[1]), dtype=np.double)
    for a in range(ifimg.shape[0]):
        for b in range(ifimg.shape[1]):
```

```
            ifimg[a][b] = ifimg3d[a][b][0]
    # pl.imshow(ifimg, cmap=cm.Greys_r)
    # pl.show()

    iffour = np.fft.fft2(ifimg)
    #iffour = iffour * (1.0/(iffour.max() - iffour.min())) # scale it?
    iffourshifted = np.fft.fftshift(iffour)
    #pl.imshow(np.abs(iffourshifted), cmap=cm.Greys_r)
    #pl.show()

    # Find frequency unlike the others, make it like the others
    #
    # Cycle through each element, get the average of it and its surrounding 5
neighbors,
    # take that average and subtract it from the element to see how much it defers
(Absolute value)
    # Then find the frequencies with the largest difference.
    kernel = np.ones((5,5))
    diffavg = np.zeros(iffourshifted.shape)
    magImg = np.abs(iffourshifted)
    for r in range(1,diffavg.shape[0]):
        for c in range(1,diffavg.shape[1]):
                diffavg[r, c] = magImg[r,c] / calculate_normal(magImg, kernel, r,
c)

    highest, ir, ic = 0, 0, 0
    for r in range(diffavg.shape[0]):
        for c in range(diffavg.shape[1]):
            if diffavg[r,c] > highest:
                highest = diffavg[r,c]
                ir = r
                ic = c
    print ir
    print ic
    iffourshifted[ir, ic] = iffourshifted[ir-1, ic-1] # assuming the nearby values
are normal, instead of taking the average of everything around
    diffavg[ir, ic] = diffavg[ir-1, ic-1]

    highest, ir, ic = 0, 0, 0
    for r in range(diffavg.shape[0]):
        for c in range(diffavg.shape[1]):
            if diffavg[r,c] > highest:
                highest = diffavg[r,c]
                ir = r
                ic = c
    print ir
    print ic
    iffourshifted[ir, ic] = iffourshifted[ir-1, ic-1]

    unshifted = np.fft.ifftshift(iffourshifted)
    spdom = np.fft.ifft2(unshifted)
    pl.imshow(np.abs(spdom), cmap=cm.Greys_r)
    pl.show()


def main():
    #parta()
    #partb()
```

```
    partc()

if __name__ == "__main__":
    main()
```