University of Nevada, Reno

CS 474 — Image Processing

Assignment #3

Joshua Gleason

Instructor:
Dr. George Bebis

September 22, 2013

Contents

List of Figures

1 Introduction

The Fourier Transform (FT) is a function that maps a function in the spacial domain, to the frequency domain. This is a very useful technique in image processing, because many types of the noise encountered in digital images is periodic. This means that if the frequency of the noise ($\frac{1}{\text{period}}$) is known then it can simply be removed in the frequency domain. Working in the frequency domain also has the benefit of easily filtering out other undesired frequencies. High frequencies, such as edges can be softened by filtering out higher frequencies, similarly the edges can be sharped by filtering out the lower frequencies. Another important property of the FT is that it can be inverted. This means that the original equation can be derived by applying the inverse FT to the the FT of the original function.

The 2D continuous Fourier Transform is defined in Eq. 1 and 2, these equation can be adapted to the discrete case as Eq. 3 and 4 shows, this is called the discrete Fourier Transform (DFT). While the continuous Fourier Transform has uses in many other fields of signal processing, it is not relevant, except for deriving the DFT in this work because digital images exist only as discrete functions. In Eq. 3 and 4, M and N stand for the height and width of the image. It is important to notice that if these values are equal the only difference in between the forward and reverse DFT is the sign of the exponent. This is a property that simplifies implementation of the function because the function only needs to be implemented once with a flag to denote the sign of the exponent.

Using these definitions it can also be shown that the multiplication of two functions in the frequency domain is equivalent to convolution in the spacial domain. Although applying this property using Eq. 3 and 4 to compute the DFT, the complexity is no better than convolution in the spacial domain, however using a technique known as the Fast Fourier Transform (FFT) this operation becomes much more efficient.

$$\Im(f(x,y)) = F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-j2\pi(ux+vy)}dxdy \tag{1}$$

$$\Im^{-1}(F(u,v)) = f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{-j2\pi(ux+vy)}dudv \tag{2}$$

$$\Im(f(x,y)) = F(u,v) = \frac{1}{M}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi(\frac{ux}{M}+\frac{vy}{N})} \tag{3}$$

$$\Im^{-1}(F(u,v)) = f(x,y) = \frac{1}{N}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{j2\pi(\frac{ux}{M}+\frac{vy}{N})} \tag{4}$$

2 Implementation

For this project, the FFT function from the Numerical Recipes book was used. By using the separability of the 2D DFT, this was achieved by simply applying the FFT on all the rows of an image, followed by applying on all the columns. For viewing and editing purposes, the image was shifted by negating every other pixel in the image before applying this transformation. This effectively shifted the image by half, making it easier to view and also easier to modify.

The first experiment required that the 2D FFT function be tested on a 32x32 pixel rectangle located in the center of a 512x512 pixel image. Once the FFT was calculated for the image, it was normalized to a logarithmic scale between 0 and 255 to be displayed.

The second experiment was a bit more complex, as it required that the FFT of the image be found and then modified. After calculating the FFT of the noisy boy image, the four strongest pixels needed to be replaced with the average of their neighbors. This was done by searching the image and building a list of four pixels with the highest magnitude. Any pixel within 5% of the center of the image was ignored to prevent center pixels from being removed (the reason for this is explained in the Results section). After finding these pixels and replacing with the average of the neighbors the inverse FFT was found and saved as the resulting image.

In an attempt to remove the noise in the spacial domain, various Gaussian kernel were constructed and filtered across the image. The kernel was calculated using a variance of approximately 1. Then the values in the kernel were divided by the sum of values to make sure all the values summed to 1. This ensured that the image would not be dimmed or brightened by the filter. A total of four filters were tested on the image, those being a 5x5, 7x7, 15x15, and 25x25.

3 Results

The results of taking the FFT of the rectangle image verified that the FFT worked as predicted and returned a *sinc* function. This could be shown by taking the 2D FFT of a step function with a finite range.

By removing the four pixels with the greatest magnitude the four most prominent frequencies were removed from the image. By looking at the image the most prominent frequencies appear to obviously be the cosine noise that has been added to the image. The reason that the center pixels must be ignored is because the very low frequencies such as flat surfaces actually take up the majority of the image which means they are the most prominent frequencies, if they were removed the image would be sharpened, and many of the important features would be lost.

The results of the Gaussian filtering in the spacial domain did not work nearly as well as directly modifying the frequency domain. By looking at the results in the Images section, it seems that the noise is not even removed after using a 25x25 filter, at which point so much of the image quality has been lost that it would be pointless to blur any more. This shows that working in the frequency domain does have some great benefits over the spacial domain because it would be nearly impossible to remove the noise there, while it was very easy to do so in the frequency domain (modifying only four pixels!).

4 Source Code

4.1 funcs.h

```
#ifndef JDG_FUNCTIONS
#define JDG_FUNCTIONS

#include "image.h"

namespace numrec
{

#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fft(float data[], unsigned long nn, int isign)
{
  unsigned long n,mmax,m,j,istep,i;
  double wtemp,wr,wpr,wpi,wi,theta;
  float tempr,tempi;

  n=nn << 1;
  j=1;
  for (i=1;i<n;i+=2) {
    if (j > i) {
      SWAP(data[j],data[i]);
      SWAP(data[j+1],data[i+1]);
    }
    m=n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    }
    j += m;
  }
  mmax=2;
  while (n > mmax) {
    istep=mmax << 1;
    theta=isign*(6.28318530717959/mmax);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m=1;m<mmax;m+=2) {
      for (i=m;i<=n;i+=istep) {
        j=i+mmax;
        tempr=wr*data[j]-wi*data[j+1];
        tempi=wr*data[j+1]+wi*data[j];
        data[j]=data[i]-tempr;
        data[j+1]=data[i+1]-tempi;
        data[i] += tempr;
        data[i+1] += tempi;
      }
      wr=(wtemp=wr)*wpr-wi*wpi+wr;
      wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
  }
```

```
}
#undef SWAP

} // namespace numrec

namespace jdg
{

enum FilterType{ IDEAL=0, GAUSSIAN=1, BUTTERWORTH=2 };

template <class pType>
void buildLowPass( jdg::Image<pType>& filter, FilterType type, float
    cutoff1,
  float cutoff2=0.0, bool freq_domain=false );

template <class pType>
void fft( Image<std::complex<pType> >& f, int val=1 );

template <class pType>
void convolve( Image<std::complex<pType> >& img,
  const Image<std::complex<pType> >& kernel, const PadWith=NEAREST );

template <class pType>
void fft( Image<std::complex<pType> >& f, int val )
{
  // resize to a power of 2
  int height = std::pow(2, std::ceil(log(f.getHeight())/log(2)));
  int width = std::pow(2, std::ceil(log(f.getWidth())/log(2)));

  // pad the image with zeros
  if ( height != f.getHeight() || width != f.getWidth() )
    f.pad( width, height, NEAREST );

  // large enough to hold rows or columns
  float* ary_vals = new float[std::max(width,height)*2];

  // perform 1D fft on all rows
  for ( int row = height-1; row >= 0; --row )
  {
    // build a row array
    for ( int i = width-1; i >= 0; --i )
    {
      // build the array for a row
      ary_vals[2*i] = static_cast<float>(f(i,row).real());
      ary_vals[2*i+1] = static_cast<float>(f(i,row).imag());

      // multiply by -1^(x+y)
      if ( (i+row)%2 != 0 && val >= 0 ) // odd
      {
        ary_vals[2*i] *= -1;
        ary_vals[2*i+1] *= -1;
      }
    }

    // find the fft of the row
    numrec::fft( ary_vals - 1, width, val );

    // put value back into image and multiply by 1/height
```

```
    for ( int i = width-1; i >= 0; --i )
    {
      f(i,row) = std::complex<pType>(
        static_cast<pType>(ary_vals[2*i]),     // real part
        static_cast<pType>(ary_vals[2*i+1]));  // imaginary part

      if ( val < 0 )
        f(i,row) *= 1.0/(height*width);
    }
  }

  // perform 1D fft on all columns
  for ( int col = width-1; col >= 0; --col )
  {
    for ( int i = height-1; i >= 0; --i )
    {
      ary_vals[2*i] = static_cast<float>(f(col,i).real());
      ary_vals[2*i+1] = static_cast<float>(f(col,i).imag());
    }

    numrec::fft( ary_vals - 1, height, val );

    for ( int i = height-1; i >= 0; --i )
      f(col,i) = std::complex<pType>(
        static_cast<pType>(ary_vals[2*i]),
        static_cast<pType>(ary_vals[2*i+1]));
  }

  delete [] ary_vals;
}

template <class pType>
void convolve( Image<std::complex<pType> >& img,
  const Image<std::complex<pType> >& kernel, const PadWith pad )
{
  Image<std::complex<pType> > kern = kernel;

  int origW = img.getWidth(), origH = img.getHeight();
  int dims =
    std::max( img.getWidth(), img.getHeight() ) +
    std::max( kern.getWidth(), kern.getHeight() );

  int shiftX = std::min(img.getWidth(), kernel.getWidth())/2;
  int shiftY = std::min(img.getHeight(), kernel.getHeight())/2;

  // pad images
  img.pad( dims, dims, pad, shiftX, shiftY );
  kern.pad( dims, dims );

  // fourier transform
  fft(img);
  fft(kern);

  // multiplication
  img *= kern;

  // invert fourier
  fft(img,-1);
```

```
  // unpad the image back to original size ZEROS because it's efficient
  img.pad( origW, origH, jdg::ZEROS, -2*shiftX, -2*shiftY );
}

template <class pType>
void buildLowPass( jdg::Image<pType>& filter, FilterType type, float param1
   ,
  float param2, bool freq_domain )
{
  //filter.resizeCanvas(512,512);
  int width = filter.getWidth();
  int height = filter.getHeight();

  float startX = -(width-1) / 2.0,
      startY = -(height-1) / 2.0,
      stopX = -startX,
      stopY = -startY;

  param1 = param1 * 0.5*sqrt(width*width+height*height);

  float param1_sqr = param1*param1;

  for ( float y = startY; y <= stopY; y+=1.0 )
    for ( float x = startX; x <= stopX; x+=1.0 )
      if ( type==IDEAL )
      {
        if ( sqrt(x*x+y*y) > param1 )
          filter(x-startX,y-startY) = 0;
        else
          filter(x-startX,y-startY) = 1;
      }
      else if ( type==GAUSSIAN )
      {
        filter(x-startX,y-startY) = exp(-(0.5*x*x+0.5*y*y)/(param1_sqr));
      }
      else if ( type==BUTTERWORTH )
      {
        filter(x-startX,y-startY) = 1.0/(1.0+pow((x*x+y*y)/param1_sqr,
           param2));
      }
  if ( !freq_domain )
    jdg::fft( filter, -1 );
}

}

#endif
```

4.2 main.cc

```
#include "funcs.h"

using namespace std;

template <class pType>
struct Point
{
```

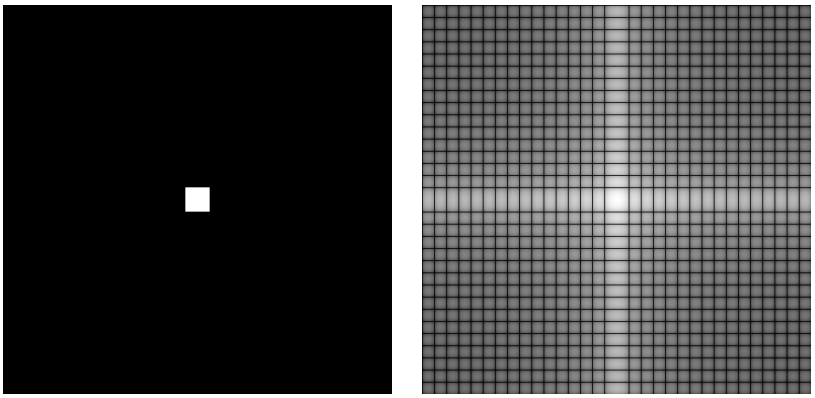
```
  Point() {}
  Point(pType _val, int _x, int _y) :
    val(_val), x(_x), y(_y) {}
  pType val;
  int x;
  int y;
};

int main(int argc, char *argv[])
{

  jdg::Image<float> show_img;
  jdg::Image<complex<float> > img;
  jdg::Image<complex<float> > img2;

  // experiment 1
    img.load("./images/rect.pgm");
    jdg::fft(img,1);
    show_img = img;
    show_img.normalize(jdg::MINMAX_LOG,0.0,255.0);
    show_img.save("./images/rect_fft.pgm");
    show_img.show();

  // experiment 2
    img.load("./images/boy_noisy.pgm");
    jdg::fft(img,1);

    show_img = img;
    show_img.normalize(jdg::MINMAX_LOG,0.0,255.0);
    show_img.show();
    show_img.save("./images/boy_noisy_fft_orig.pgm");

    // find top 4 values and remove
      Point<complex<float> >* top4 = new Point<complex<float> >[4];
      Point<complex<float> > temp, temp2;
      for ( int i = 0; i < 4; i++ )
      {
        temp.val = img(i,0);
        temp.x = i;
        temp.y = 0;
        top4[i] = temp;
      }

      Point<int> center;
      center.x = img.getHeight()/2;
      center.y = img.getHeight()/2;

      // minimum distance from center is 1/20 of image max(height,width)
      float min_dist = max(img.getHeight(),img.getWidth())/20.0;

      #define DIST(p1,p2) sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2
          .y))

      for ( int i = img.getHeight()-1; i>=0; --i )
        for ( int j = img.getWidth()-1; j>=0; --j )
        {
          temp.x = j;
          temp.y = i;
```

```
      if ( DIST(temp, center) > min_dist )
      {
        temp.val = img(j,i);
        for ( int k = 0; k < 4; k++ )
        {
          if ( abs(top4[k].val) < abs(img(j,i)) )
          {
            temp2 = temp;
            temp = top4[k];
            top4[k] = temp2;
          }
        }
      }
    }

  #undef DIST

  // average with values around it
  for ( int i = 0; i < 4; i++ )
  {
    img(top4[i].x,top4[i].y) = 0;
    for ( int x = -1; x <= 1; x++ )
      for ( int y = -1; y <= 1; y++ )
        if (!( x == 0 && y == 0 ))
        {
          img(top4[i].x,top4[i].y)+=img(top4[i].x+x,top4[i].y+y);
        }
    img(top4[i].x,top4[i].y) /= 8.0;
  }

  delete [] top4;

// save fixed fft
show_img = img;
show_img.normalize(jdg::MINMAX_LOG,0.0,255.0);
show_img.show();
show_img.save("./images/boy_noisy_fft_fixed.pgm");

// inverse fft
jdg::fft(img,-1);

// save fixed image
show_img = img;
show_img.show();
show_img.save("./images/boy_noisy_fixed.pgm");

// try with gausianm 5x5 and 7x7
img.load("./images/boy_noisy.pgm");

// 5x5
img2.resizeCanvas(5,5);
buildLowPass( img2, jdg::GAUSSIAN, 0.307307, 0, true );

float sum = 0.0;
// normalize the gaussian to sum to 1
for ( int i = 0; i < 5; i++ )
  for ( int j = 0; j < 5; j++ )
    sum += abs(img2(j,i));
```

```
img2 /= sum;

jdg::Image<complex<float> > temp_img(img);

jdg::convolve( temp_img, img2 );
show_img = temp_img;
show_img.show();
show_img.save("./images/boy_noisy_5x5_gaussian.pgm");

jdg::convolve( temp_img, img );

// 7x7
img2.resizeCanvas(7,7);
buildLowPass( img2, jdg::GAUSSIAN, 0.307307, 0, true );

sum = 0.0;
// normalize the gaussian to sum to 1
for ( int i = 0; i < 7; i++ )
  for ( int j = 0; j < 7; j++ )
    sum += abs(img2(j,i));
img2 /= sum;

temp_img = img;

jdg::convolve( temp_img, img2 );
show_img = temp_img;
show_img.show();
show_img.save("./images/boy_noisy_7x7_gaussian.pgm");

// 15x15
img2.resizeCanvas(15,15);
buildLowPass( img2, jdg::GAUSSIAN, 0.307307, 0, true );

sum = 0.0;
// normalize the gaussian to sum to 1
for ( int i = 0; i < 15; i++ )
  for ( int j = 0; j < 15; j++ )
    sum += abs(img2(j,i));
img2 /= sum;

temp_img = img;

jdg::convolve( temp_img, img2 );
show_img = temp_img;
show_img.show();
show_img.save("./images/boy_noisy_15x15_gaussian.pgm");

// 25x25
img2.resizeCanvas(25,25);
buildLowPass( img2, jdg::GAUSSIAN, 0.307307, 0, true );

sum = 0.0;
// normalize the gaussian to sum to 1
for ( int i = 0; i < 25; i++ )
  for ( int j = 0; j < 25; j++ )
    sum += abs(img2(j,i));
img2 /= sum;
```

```
    temp_img = img;

    jdg::convolve( temp_img, img2 );
    show_img = temp_img;
    show_img.show();
    show_img.save("./images/boy_noisy_25x25_gaussian.pgm");

  // return
  return 0;
}
```

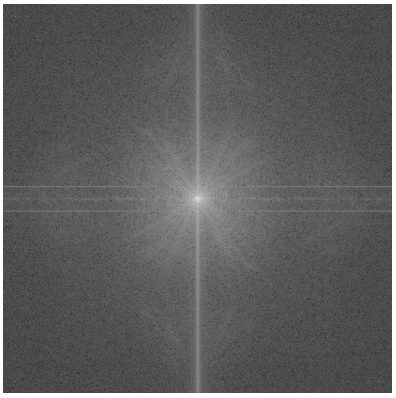
5 Images



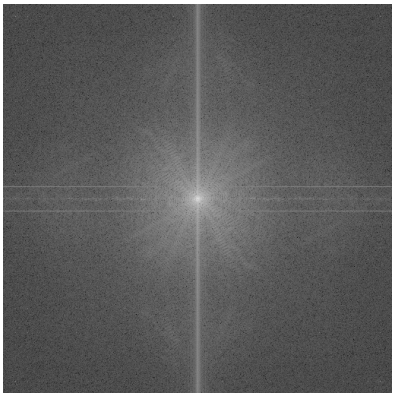
(a) Rectangle (b) Rectangle FFT

Figure 1: 32x32 Rectangle with FFT.

(a) Original Image

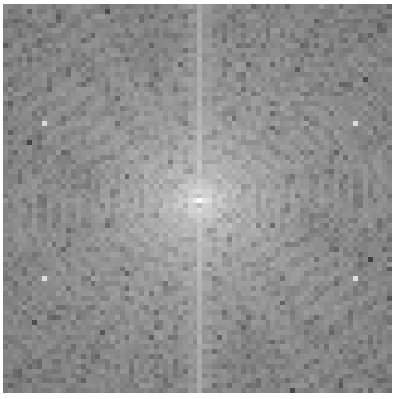


(b) FFT

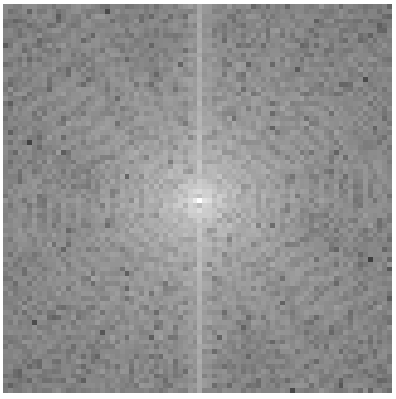


(c) Corrected FFT

(d) Corrected Image



(e) FFT Zoomed



(f) Corrected FFT Zoomed

Figure 2: Noisy Image corrected in frequency domain.

(a) Original Image

(b) 5x5

(c) 7x7

(d) 15x15

(e) 25x25

Figure 3: Attempting to remove noise using Gaussian Filters.