University of Nevada, Reno

CS 474 — Image Processing

Assignment #4

Joshua Gleason

Instructor:
Dr. George Bebis

November 2, 2010

Contents

List of Figures

1 Introduction

2 Implementation

3 Results

4 Source Code

4.1 funcs.h

```
#ifndef JDG_FUNCTIONS
#define JDG_FUNCTIONS

#include "image.h"
#include <cmath>

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
{

// 2D Fast Fourier Transform

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

// convolution using convolution theorem (done in freq domain)
template <class pType>
void convolve( Image<std::complex<pType> >& img,
  const Image<std::complex<pType> >& kernel, const PadWith=NEAREST );

// convolution using spacial domain (slow)
template <class pType>
void convolve_spacial( Image<pType>& img,
  const Image<pType>& kernel, const PadWith=NEAREST );

// the value for n should be positive for a highpass and positive for
   lowpass
// D0    : cuttoff frequency (1pixel=1Hz)
// n     : order             (higher means faster convergence)
// gammaL: lower horizontal asymptote
// gammaH: upper horizontal asymptote
template <class pType>
void butterworth( jdg::Image<pType>& filter, float D0, float n=1.0,
  float gammaL=0.0, float gammaH=1.0 );

// build a gaussian filter type > 0 for highpass and < 0 for lowpass
// D0    : variance          (in pixels)
// type  : high or low pass
// gammaL: lower horizontal asymptote
// gammaH: upper horizontal asymptote
template <class pType>
void gaussian( jdg::Image<pType>& filter, float D0, int type=1, float
   gammaL=0.0,
  float gammaH=1.0 );

// build an ideal filter type > 0 for highpass and < 0 for lowpass
// D0    : cuttoff frequency (1pixel=1Hz)
// type  : high or low pass
// gammaL: lower horizontal asymptote
// gammaH: upper horizontal asymptote
template <class pType>
void idealfilter( jdg::Image<pType>& filter, float D0, int type=1, float
   gammaL=0.0,
  float gammaH=1.0 );

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);

  // normalize back to value (this was divided out twice in the fft)
  img *= img.getWidth() * img.getHeight();

  // 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 convolve_spacial( Image<pType>& img,
  const Image<pType>& kernel, const PadWith )
{
  int width = img.getWidth(),
      height = img.getHeight(),
      kernW = kernel.getWidth(),
      kernH = kernel.getHeight();

  int kernW_h = kernW/2,
      kernH_h = kernH/2,
      realX, realY;

  Image<pType> retImg(width, height);

  for ( int x = 0; x < width; x++ )
  for ( int y = 0; y < height; y++ )
```

```
  {
    retImg(x,y) = 0;
    for ( int kernX = 0; kernX < kernW; kernX++ )
    for ( int kernY = 0; kernY < kernH; kernY++ )
    {
      realX = x-kernW_h+kernX;
      realY = y-kernH_h+kernY;
      if ( realX >= 0 && realY >= 0 && realX < width && realY < height )
        retImg(x,y) = retImg(x,y) +
          img( x-kernW_h+kernX, y-kernH_h+kernY ) *
          kernel( kernW-kernX-1, kernH-kernY-1 );
    }
  }

  img = retImg;
}

template <class pType>
void butterworth( jdg::Image<pType>& filter, float D0, float n,
  float gammaL, float gammaH )
{
  int width = filter.getWidth();
  int height = filter.getHeight();

  float startX = -(width-1) / 2.0,
        startY = -(height-1) / 2.0,
        stopX = -startX,
        stopY = -startY,
        D0_sqr = D0*D0,
        diff = gammaH-gammaL,
        yy;

  for ( float y = startY; y <= stopY; y+=1.0 )
  {
    yy = y*y;
    for ( float x = startX; x <= stopX; x+=1.0 )
      if ( x != 0 && y != 0 )
        filter(x-startX,y-startY) = gammaL + diff/(1+pow(D0_sqr/(x*x+yy),n)
          );
      else if (n>0) // lowpass
        filter(x-startX,y-startY) = gammaH;
      else          // highpass
        filter(x-startX,y-startY) = gammaL;
  }
}

template <class pType>
void gaussian( jdg::Image<pType>& filter, float D0, int type, float gammaL,
  float gammaH )
{
  int width = filter.getWidth();
  int height = filter.getHeight();

  float startX = -(width-1) / 2.0,
        startY = -(height-1) / 2.0,
        stopX = -startX,
        stopY = -startY,
        D0_sqr2 = 2*D0*D0,
```

```
        diff = gammaH-gammaL,
        yy;

  for ( float y = startY; y <= stopY; y+=1.0 )
  {
    yy = y*y;
    for ( float x = startX; x <= stopX; x+=1.0 )
      if ( type >= 0 ) // highpass
        filter(x-startX,y-startY) = diff*(1-exp(-(x*x+yy)/D0_sqr2))+gammaL;
      else             // lowpass
        filter(x-startX,y-startY) = diff*exp(-(x*x+yy)/D0_sqr2)+gammaL;
  }
}

template <class pType>
void idealfilter( jdg::Image<pType>& filter, float D0, int type, float
   gammaL,
  float gammaH )
{
  int width = filter.getWidth();
  int height = filter.getHeight();

  float startX = -(width-1) / 2.0,
        startY = -(height-1) / 2.0,
        stopX = -startX,
        stopY = -startY,
        first = (type < 0 ? gammaH : gammaL),
        second = (type < 0 ? gammaL : gammaH),
        yy;


  for ( float y = startY; y <= stopY; y+=1.0 )
  {
    yy = y*y;
    for ( float x = startX; x <= stopX; x+=1.0 )
      if ( sqrt(x*x+y*y) <= D0 )
        filter(x-startX,y-startY) = first;
      else
        filter(x-startX,y-startY) = second;

  }
}

}

#endif
```

4.2 main.cc

```
#include <iostream>
#include "funcs.h"
#include <sstream>
#include <iomanip>

using namespace std;


complex<double> natlog( complex<double> val )
```

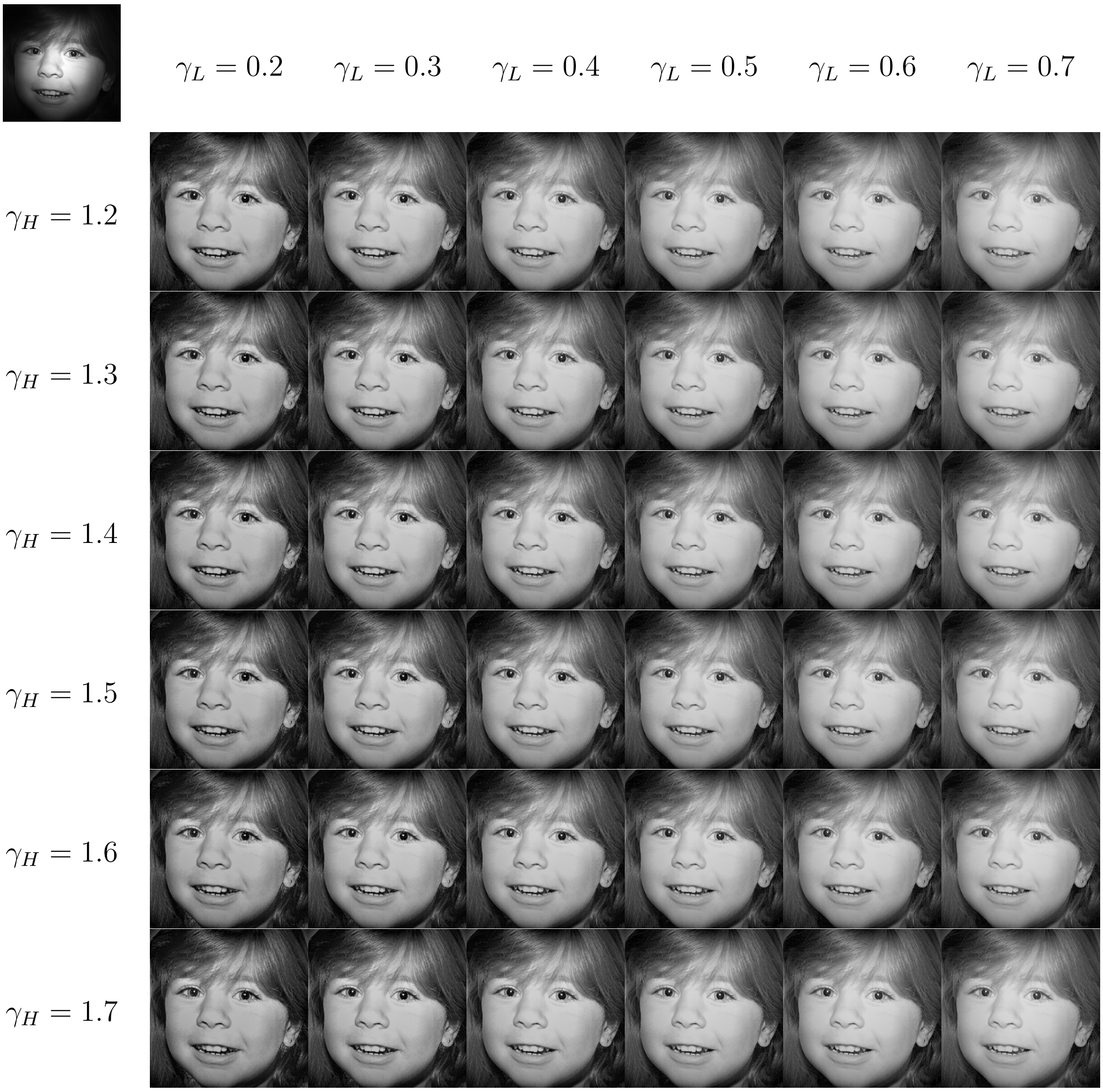
```
{
  return log(abs(val));
}

complex<double> exponential( complex<double> val )
{
  return exp(abs(val));
}

int main(int argc, char* argv[])
{
  // part 1
  jdg::Image<complex<double> > img;

  //jdg::Image<complex<double> > boat("images/boat.pgm");
  jdg::Image<complex<double> > gaussian15(15,15);
  jdg::Image<complex<double> > gaussian7(7,7);
  jdg::Image<complex<double> > result_freq;
  jdg::Image<complex<double> > result_space;
  jdg::Image<double> display;
  double error;

  std::string imagename;

  int mask15[] =
    {2 ,2 ,3 ,4 ,5 ,5 ,6 ,6 ,6 ,5 ,5 ,4 ,3 ,2 ,2,
     2 ,3 ,4 ,5 ,7 ,7 ,8 ,8 ,8 ,7 ,7 ,5 ,4 ,3 ,2,
     3 ,4 ,6 ,7 ,9 ,10,10,11,10,10,9 ,7 ,6 ,4 ,3,
     4 ,5 ,7 ,9 ,10,12,13,13,13,12,10,9 ,7 ,5 ,4,
     5 ,7 ,9 ,11,13,14,15,16,15,14,13,11,9 ,7 ,5,
     5 ,7 ,10,12,14,16,17,18,17,16,14,12,10,7 ,5,
     6 ,8 ,10,13,15,17,19,19,19,17,15,13,10,8 ,6,
     6 ,8 ,11,13,16,18,19,20,19,18,16,13,11,8 ,6,
     6 ,8 ,10,13,15,17,19,19,19,17,15,13,10,8 ,6,
     5 ,7 ,10,12,14,16,17,18,17,16,14,12,10,7 ,5,
     5 ,7 ,9 ,11,13,14,15,16,15,14,13,11,9 ,7 ,5,
     4 ,5 ,7 ,9 ,10,12,13,13,13,12,10,9 ,7 ,5 ,4,
     3 ,4 ,6 ,7 ,9 ,10,10,11,10,10,9 ,7 ,6 ,4 ,3,
     2 ,3 ,4 ,5 ,7 ,7 ,8 ,8 ,8 ,7 ,7 ,5 ,4 ,3 ,2,
     2 ,2 ,3 ,4 ,5 ,5 ,6 ,6 ,6 ,5 ,5 ,4 ,3 ,2 ,2};

  int mask7[] =
    {1 ,1 ,2 ,2 ,2 ,1 ,1,
     1 ,2 ,2 ,4 ,2 ,2 ,1,
     2 ,2 ,4 ,8 ,4 ,2 ,2,
     2 ,4 ,8 ,16,8 ,4 ,2,
     2 ,2 ,4 ,8 ,4 ,2 ,2,
     1 ,2 ,2 ,4 ,2 ,2 ,1,
     1 ,1 ,2 ,2 ,2 ,1 ,1};

  // convert arrays to images
  for ( int i = 0; i < 15; i++ )
    for ( int j = 0; j < 15; j++ )
      gaussian15(i,j) = mask15[i*15+j];

  for ( int i = 0; i < 7; i++ )
    for ( int j = 0; j < 7; j++ )
      gaussian7(i,j) = mask7[i*7+j];
```

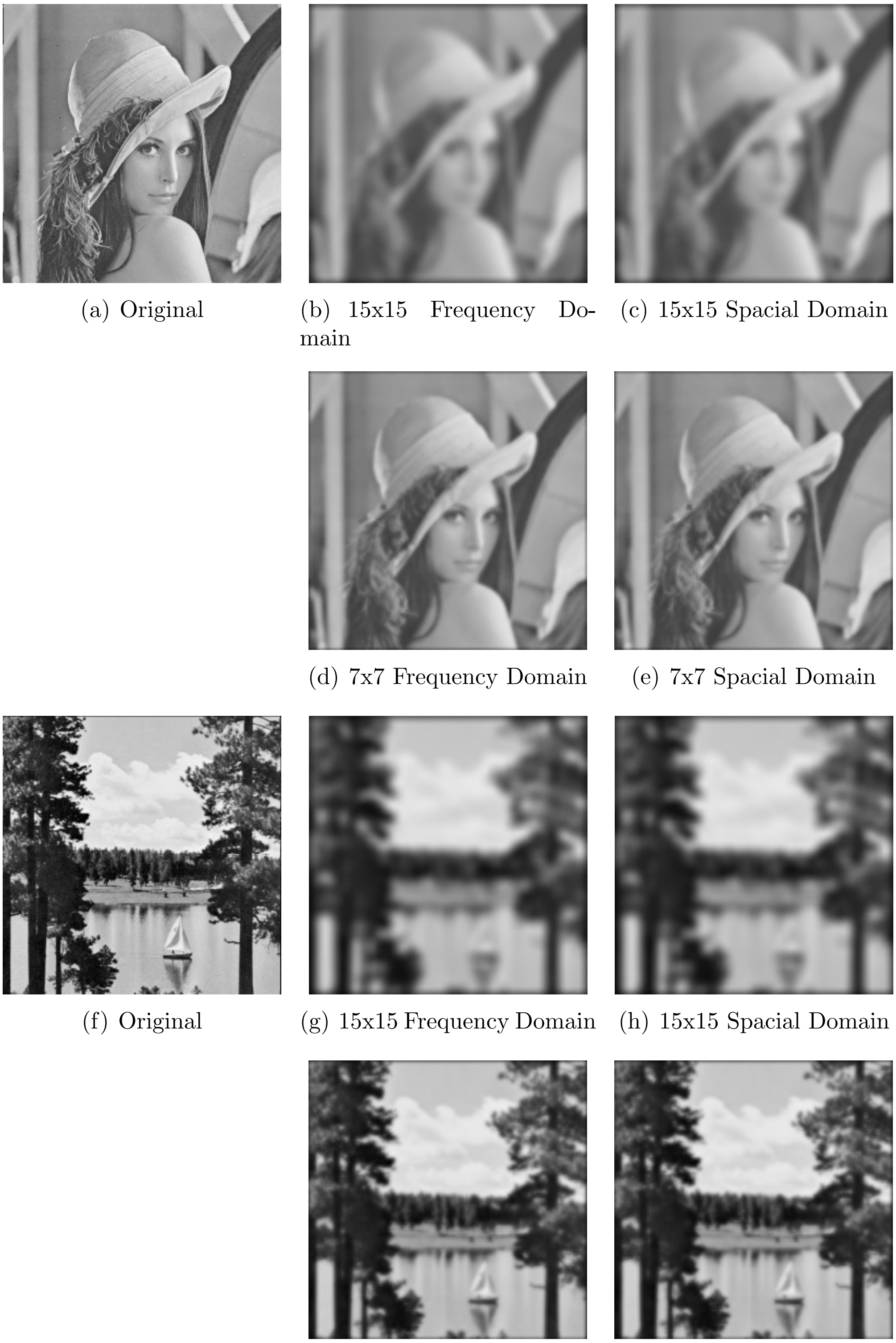
```
// normalize masks to sum to 1
gaussian15.normalize( jdg::L1, 1.0 );
gaussian7.normalize( jdg::L1, 1.0 );

int count = 0;
while (count < 2)
{
  if ( count == 0 )
    imagename = "images/lenna";
  else
    imagename = "images/boat";

  img.load( imagename+".pgm" );

  count++;
  // Filter img with 15x15

  result_freq = img;
  result_space = img;

  // convolve using frequency domain
  jdg::convolve( result_freq, gaussian15, jdg::ZEROS );
  jdg::convolve_spacial( result_space, gaussian15, jdg::ZEROS );

  display = result_freq;
  display.save(imagename+"_15_freq.pgm");
  cout << imagename << "_15_freq.pgm Saved!" << endl;

  display = result_space;
  display.save(imagename+"_15_space.pgm");
  cout << imagename << "_15_space.pgm Saved!" << endl;

  // calculate error
  display -= result_freq; // difference
  display *= display;     // squared

  // sum
  error = 0.0;
  for ( int i = 0; i < display.getWidth(); i++ )
    for ( int j = 0; j < display.getHeight(); j++ )
      error += display(i,j);

  // times 1/MN
  error *= 1.0/(display.getWidth()*display.getHeight());

  cout << imagename << " 15x15 Difference : "
       << error << endl;

  // Filter img with 7x7
  result_freq = img;
  result_space = img;

  // convolve using frequency domain
  jdg::convolve( result_freq, gaussian7, jdg::ZEROS );
  jdg::convolve_spacial( result_space, gaussian7, jdg::ZEROS );

  display = result_freq;
```

```
  display.save(imagename+"_7_freq.pgm");
  cout << imagename << "_7_freq.pgm Saved!" << endl;

  display = result_space;
  display.save(imagename+"_7_space.pgm");
  cout << imagename << "_7_space.pgm Saved!" << endl;

  // calculate error
  display -= result_freq; // difference
  display *= display;     // squared

  // sum
  error = 0.0;
  for ( int i = 0; i < display.getWidth(); i++ )
    for ( int j = 0; j < display.getHeight(); j++ )
      error += display(i,j);

  // times 1/MN
  error *= 1.0/(display.getWidth()*display.getHeight());

  cout << imagename << " 7x7 Difference : "
       << error << endl << endl;
}

// part 2

for ( float YL = 0.2; YL <= 0.8; YL+=0.1 )
for ( float YH = 1.2; YH <= 1.8; YH+=0.1 )
{
  jdg::Image<complex<double> > a("images/girl.pgm");
  jdg::Image<complex<double> > filter(a.getWidth(),a.getHeight());
  jdg::Image<double> show;

  // step 1
  a.callFunc( &natlog );

  // step 2
  jdg::fft(a);

  // step 3
  // 0.3 and 1.3 ? seem to be good

  jdg::butterworth( filter, 1.8, 1.0, YL, YH );
  //jdg::gaussian( filter, 1.8, 1, YL, YH );
  //jdg::idealfilter( filter, 1.8, 1, YL, YH );

  a = a*filter;

  // step 4
  jdg::fft(a,-1);

  // step 5
  a.callFunc( &exponential );

  ostringstream sout;
  sout << "./images/girl_" << YL*10 << "_" << YH*10 << ".pgm";

  show = a;
```

```
    show.normalize( jdg::MINMAX_LOG, 0, 225 );
    show.save(sout.str().c_str());
    cout << sout.str() << " Saved!" << endl;
  }
  return 0;
}
```

5 Images


$\gamma_L = 0.2$
$\gamma_L = 0.3$
$\gamma_L = 0.4$
$\gamma_L = 0.5$
$\gamma_L = 0.6$
$\gamma_L = 0.7$
$\gamma_H = 1.2$
$\gamma_H = 1.3$
$\gamma_H = 1.4$
$\gamma_H = 1.5$
$\gamma_H = 1.6$
$\gamma_H = 1.7$



(a) Original (b) 15x15 Frequency Domain (c) 15x15 Spacial Domain

(d) 7x7 Frequency Domain (e) 7x7 Spacial Domain

(f) Original (g) 15x15 Frequency Domain (h) 15x15 Spacial Domain

(i) 7x7 Frequency Domain (j) 7x7 Spacial Domain

Figure 1: Two images convolved with 15x15 and 7x7 Gaussian. Compairison of convolution using the convolution theorem and convolution in the spacial domain.