University of Nevada, Reno

CS 474 — Image Processing

Assignment #5

Joshua Gleason

Instructor:
Dr. George Bebis

December 15, 2010

Contents

1 Introduction

The Discrete Fourier Transform(DFT) is very useful in image and signal processing. It transforms a signal into a sum of cosines with varying magnitude and phase. This allows the different frequencies which compose a signal to be analyzed, filtered, and even modified. However, because cosines have an infinite duration, the location of the different frequencies in the signal will remain unknown.

One method of determining the locations of the frequencies is to use what is called a Discrete Short-Time Fourier Transform. This method entails applying the DFT to smaller portions of the signal, then repeating the process at some, preferably overlapping intervals across the entire signal. This allows the locations of various frequencies to become more well known. However, a major disadvantage of this method is that if more precision is desired, the windows must become smaller, but in retrospect the ability to detect lower frequencies decreases at the same time. This is known as the Heisenberg Principle which states that one cannot simultaneously know the exact time and frequency of a component of a signal. The more that is known about the frequency, the less is known about its location, and the more that is known about the location, the less is known about the frequencies at that location. This means that a compromise must be made. The major issue with this method is that the parameters of the windowing function can greatly effect the amount and type of information obtained.

This brings us to the Discrete Wavelet Transform(DWT) which overcomes the problem of determining the width of a window by analyzing the signal at multiple resolutions simultaneously. A wavelet function, unlike a cosine, has a finite duration which means it can be scaled to analyze only a portion of a signal. By using wavelet functions, a signal can be decomposed into lower resolution versions of itself. If this is done repeatedly, an image or signal can be reduced to a single value along with a large number of detail coefficients. Because these coefficients are acquired from multiple resolutions of the image, they can be used to examine these resolutions simultaneously. The DWT gives a more complete view of the details which compose a signal than the Short-Time Fourier Transform because a single window size does not need to be chosen.

Another very useful property of some wavelets, such as the Haar and Daubechies, is that they generate a somewhat sparse set of coefficients. This means that the majority of the details are expressed using only a small percentage of the coefficients. This is one of the reasons wavelets are being used for image compression.

Like the Fourier Transform, the DWT is separable. Because of this, implementing the 2D DWT can be done by applying the DWT on all the rows of an image and then all the columns. However in practice this is usually done by reducing the resolution by one step on all the columns, then the rows. Then repeating until the entire image is reduced to one value and a set of coefficients. This is known as the non-standard decomposition, and is used in all the following experiments.

2 Experiments

The first experiment was to implement the 2D DWT and its inverse. Once both functions were implemented, an image was transformed into the wavelet domain, and then back into the spacial domain. Because of the properties of the DWT, the resulting image should be the same as the original image; other than perhaps some minor rounding errors.

The second experiment was to report the number of non-zero coefficients in both the

DFT and the DWT. Because the DWT is considerably more sparse than the DFT, there should be considerably more non-zero coefficients in the DFT.

The third, and most interesting experiment is to remove all but the largest coefficients in the wavelet transformed image, then apply the inverse DWT and compare with the original image. Many of the coefficients should be able to be removed because of the sparsity of the DWT. By reconstructing using different amounts of coefficients, a type of lossy image compression similar(but less complex) to JPEG can be achieved.

Once all three of these are complete, the Daubechies D4 wavelet was implemented and run through the same experiments. The D4 wavelet filter is composed of four values, making the implementation a bit more tricky. However, once a generic wavelet implementation was created, I took the liberty to implement Daubechies wavelet filters of sizes 4 through 22 using values obtained on the Internet.

I also took the last experiment a step further and created an algorithm that continuously added and removed coefficients until the error between the reconstructed image and the original converged at a desired value. By using this algorithm, the minimum number of coefficients required to express an image at a desired error was computed for the Haar and Daubechies wavelets at errors of 5 through 40 at intervals of 5. This experiment was conducted in the hope that it would give an understanding about how using different wavelets can change the ability to compress and image.

3 Implementation

The DWT Transform was implemented in the non-standard fashion by applying the filter once through each row, then through each column. Each time the filter was applied, the particular array was re-ordered so that the coefficients were stored at the end, and the averaging values were stored at the front. This was because only the averaging coefficients needed to be transformed again, the coefficient values are not changed once they are obtained. For the Daubechies Wavelets, the filters needed to wrap around the ends of the function in order to reduce the error in reconstruction.

The inverse DWT used an inverse of the original filter to recover an image already in the wavelet domain. For implementation, this filter's center was located one index before its end, as opposed to the first index like in the forward DWT. Once this was realized, implementation of the inverse DWT was straightforward.

Calculation of the error was done using the Mean Square Error formula expressed in Eq. 1.

$$MSE = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[f_{original}(x,y) - f_{reconstructed}(x,y) \right]^2 \tag{1}$$

The algorithm that searched for the largest coefficients was perhaps the greatest implementation obstacle. Because I wanted to implement a searching algorithm that converged on the fewest number of coefficients for a given amount of error, it had to be much faster than simply sorting all the coefficients in the image.

Instead I used a combination of a binary search, along with insertion sort. The reason insertion sort was chosen, is that it can easily obtain the top values of a given set without sorting all of them. For example, if the top 10 values wanted to be found, the first step would be to place the first 10 values into the list, sorting simultaneously using insertion sort. Once these values are in the list the sorting algorithm tries to put the next value into the list. If

the value is greater than the smallest value in the list(the value in the 10th index) then that value is simply passed over. If the value is greater than the smallest value, then it is placed into the list using the standard insertion sort method. The only differences are that when the 11th term is inserted into the list, the last value is discarded, and rather than traversing the entire list to determine where to place the value, binary search is used to speed up the process.

For obtaining the minimum number of coefficients, binary search needed to be used, otherwise the DWT would need to be calculated every time another coefficient was kept until the desired error was reached. Instead I implemented a modified version of binary search that started by keeping the top 17% of the coefficients, rather than the top 50%. This was done because sorting half the values in an image takes a long time, and often times only a fraction of those coefficients are required. This continues until the perfect number of coefficients is found. In practice, finding the different errors ranging from 5 to 40 using all 11 wavelets for a 256x256 image took only about 20 seconds of processing time on my 64-bit Intel i5 2.5GHz (4 core) machine with 8 GB of memory. When making use of the multiple cores, I was able to write a script that completed this task in just over 3 seconds.

4 Results

For the first experiment, the results were just as expected where I received an error of less than 10e-20. I expect that the value was this small because of the great deal of precision used in the calculation of the wavelet values.

For the second experiment, the number of values that are zero in the Fourier Transform for lenna.pgm was only 284 and for boat.pgm was only 156. In the Haar Wavelet there were 5618 zero value coefficients for lenna.pgm and 4730 for boat.pgm. This result matches what was expected because the Haar Wavelet should be much more sparse than the Fourier. Using the Daubechies D4 Wavelet, there were 7172 zero value coefficients for lenna.pgm and 5999 for boat.pgm. The Daubechies Wavelet had less non-zero coefficients than both Haar and Fourier. This gives reason to believe that for lenna.pgm and boat.pgm, the D4 Wavelet is best for compression. The results for different sized Daubechies Wavelets and other images can be seen in the Images section below.

For reconstruction using a percentage of the coefficients, the algorithm described in the previous section was used. For lenna.pgm, using the Haar Wavelet the following results were aquired. The number of coefficients here is the minimum necessary for reconstruction for the given error. It seems that the more complex an image is, the better suited more lenghty Daubechies Wavelets are for compression.

- Using Haar
  - Lenna
    - Error 5: 31.75% or 20809 coefficients were needed.
    - Error 50: 8.78% or 5752 coefficients were needed.
    - Error 400: 0.63% or 413 coefficients were needed.
  - Boat
    - Error 5: 42.4% or 27786 coefficients were needed.
    - Error 50: 18.47% or 12103 coefficients were needed.
    - Error 400: 3.74% or 2448 coefficients were needed.

- Using Daubechies D4
  - Lenna
    - Error 5: 28.38% or 18602 coefficients were needed.
    - Error 50: 7.21% or 4723 coefficients were needed.
    - Error 400: 0.48% or 315 coefficients were needed.
  - Boat
    - Error 5: 42.17% or 27639 coefficients were needed.
    - Error 50: 17.95% or 11761 coefficients were needed.
    - Error 400: 3.11% or 2035 coefficients were needed.

Looking at these results, it seems that the number of non-zero values is somehow proportional to how well the image will compress. This makes sense because the number of non-zero values may give as an approximation of the average amount of information contained by each coefficient. The graphs in the following section also compare the compression quality of the other Daubechies Wavelets on multiple different image. Looking at these bar graphs, it seems that the best Daubechies wavelet for compressing is actually the D6, with the D8 and D4 usually about even. This seems to be true for all the photographs with sufficient detail. For the images with large areas of low detail, the Haar actually works much better. For example in wheel.pgm and tools.pgm, the Haar wavelet outperforms the Daubechies. However, by examining konye.pgm(which I found on the Internet), it is noticed that the D8 wavelet outperformed the D6. The image also contained fewer zero value coefficients, and there is a need for substantially more coefficients in order to reconstruct the image with the same error. This make sense as more edges in an image mean more high frequencies. These high frequencies show up more at a higher resolutions which means more coefficients.

5 Conclusion

In these experiments I learned a great deal about wavelets and especially how they can be used for compression. By using wavelets, combined with other techniques such as encoding and quantization techniques a very powerful compression algorithm can be created. We learned that fingerprint compression actually uses wavelets for compression and also that the jpeg2000 format also uses wavelets. This experiment also stressed how important understanding different uses and implemention of wavelets can be in image processesing.

6 Images

(a) Haar @ 50 Error (b) Haar @ 100 Error (c) Haar @ 400 Error

(d) D-4 @ 50 Error (e) D-4 @ 100 Error (f) D-4 @ 400 Error

(g) D-20 @ 50 Error (h) D-20 @ 100 Error (i) D-20 @ 400 Error

Figure 1: Results of different wavelets with different amounts of reconstruction error. The mean square error forumla was used to calculate error.

Figure 2: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

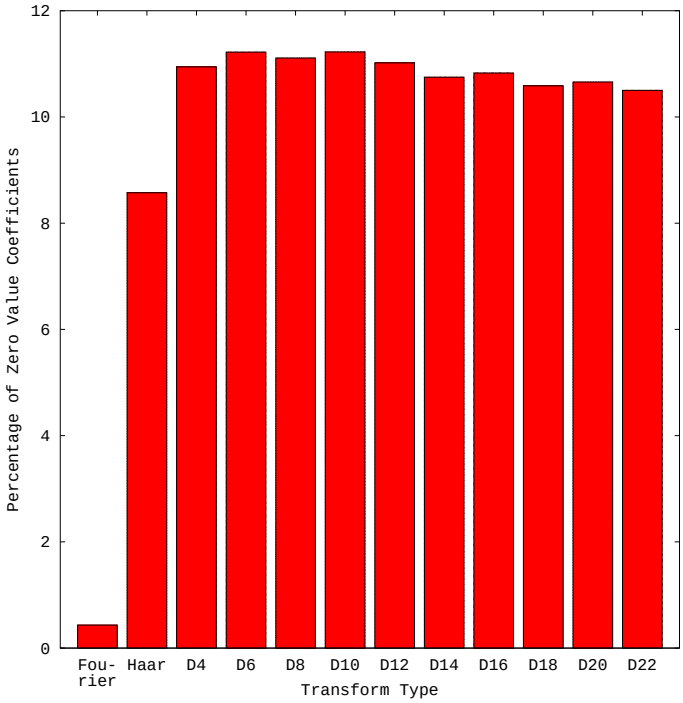

Figure 3: Percentage of coefficients in **lenna** with a value of zero. Uses multiple lengths of the Daubechies Wavelet.

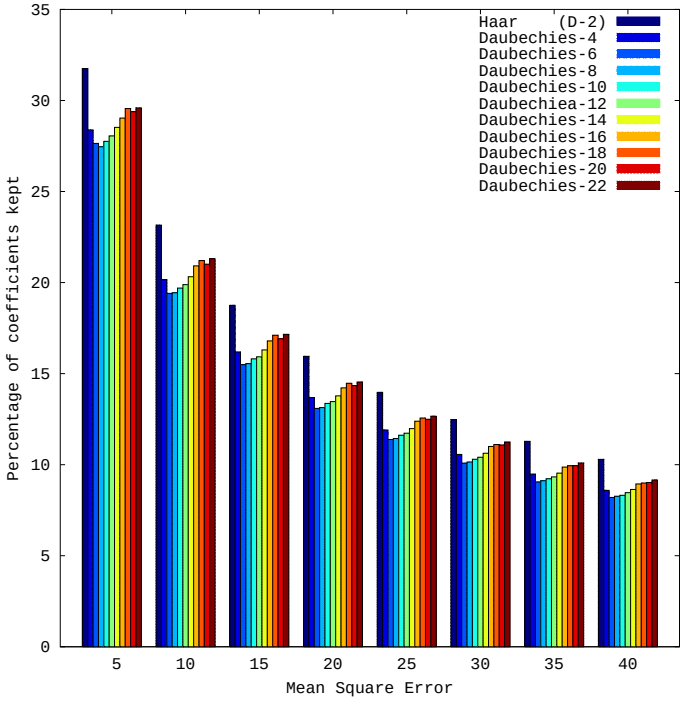

Figure 4: Minimum coefficients in **lenna** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

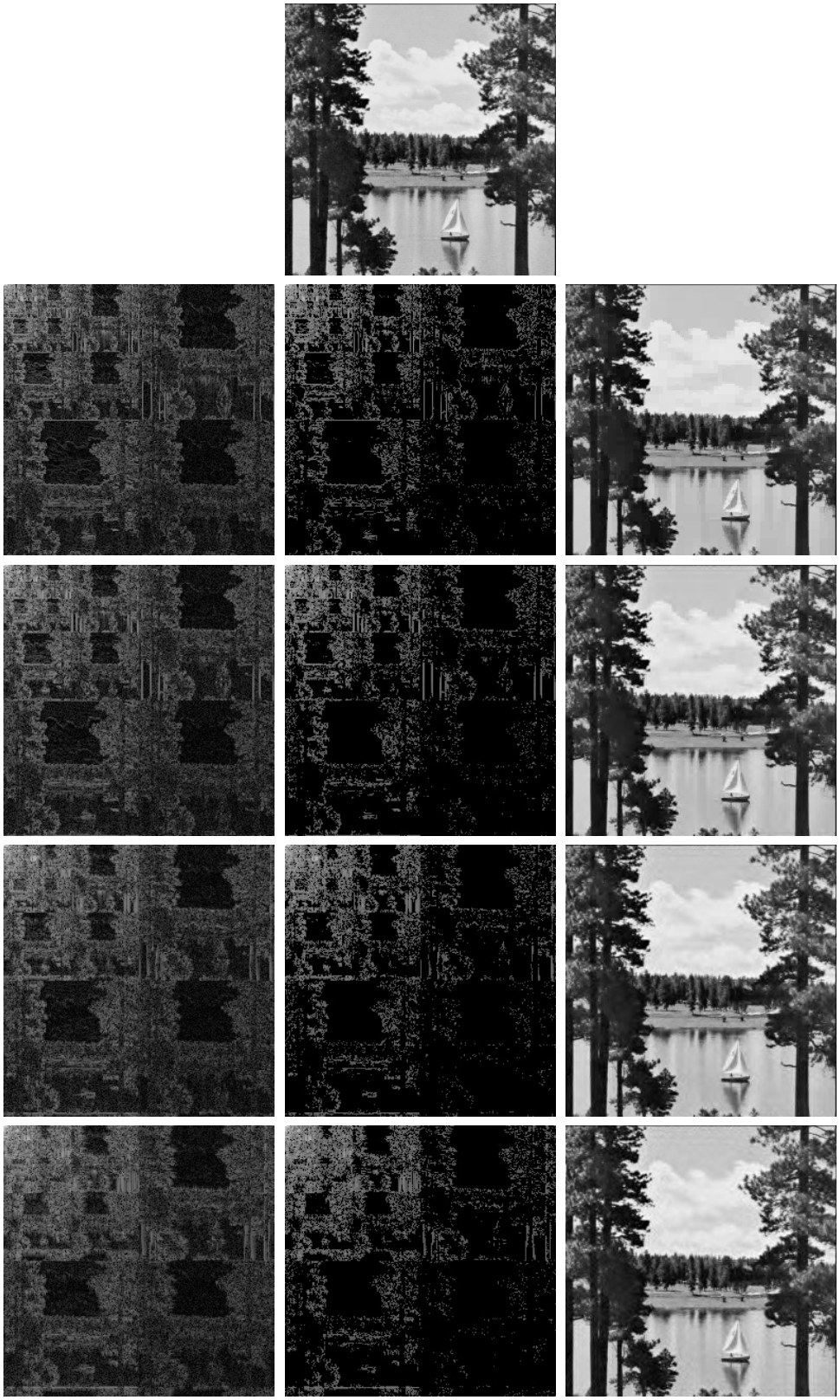

Figure 5: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

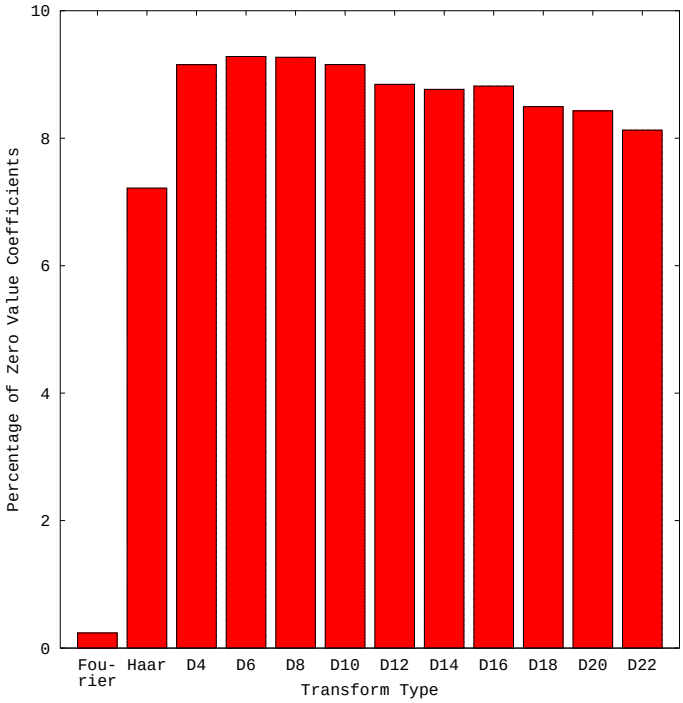

Figure 6: Percentage of coefficients in **boat** with a value of zero. Uses multiple lengths of the Daubechies Wavelet.

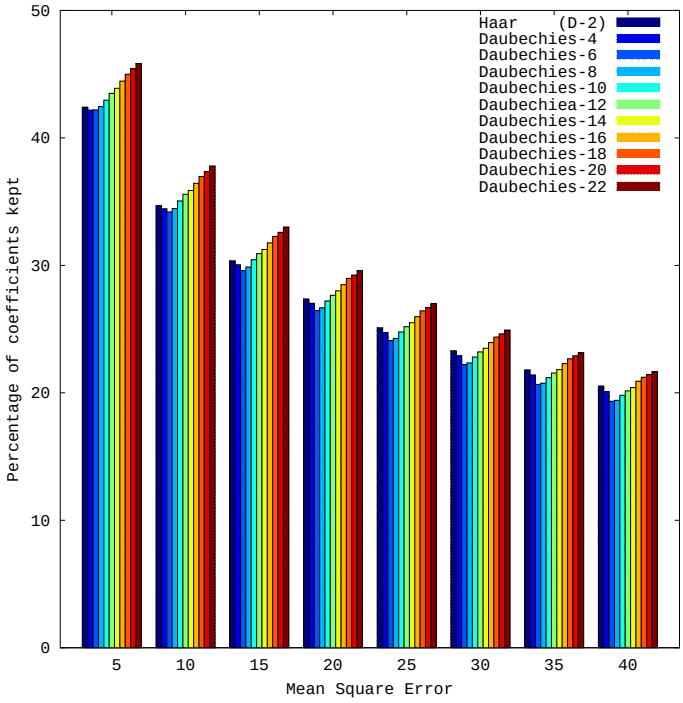

Figure 7: Minimum coefficients in **boat** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

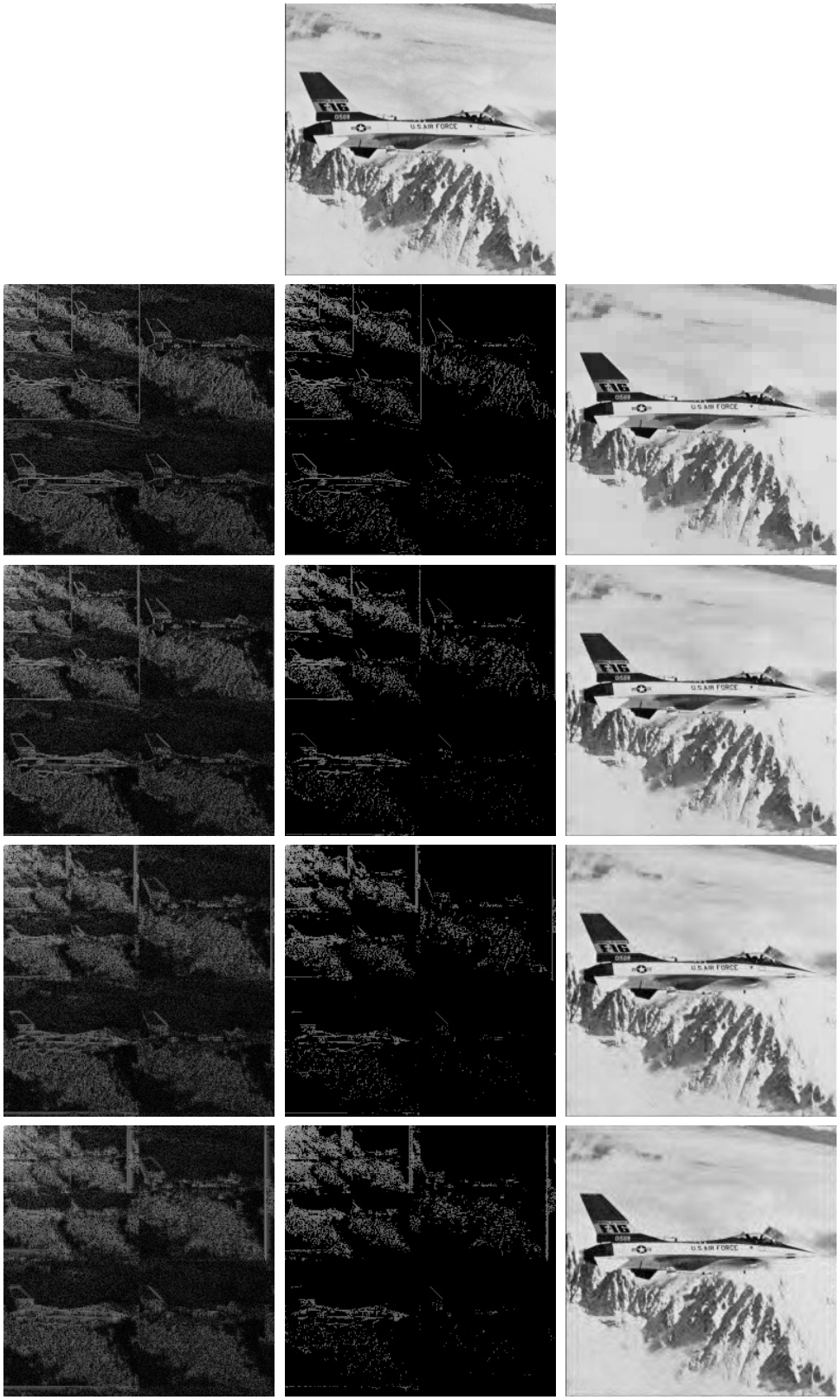

Figure 8: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

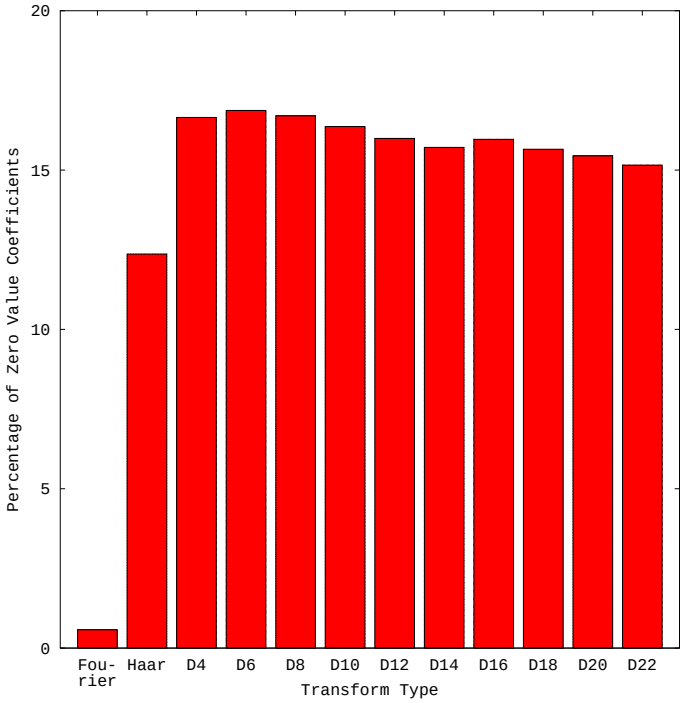

Figure 9: Percentage of coefficients in **f_16** with a value of zero. Uses multiple lengths of the Daubechies Wavelet.

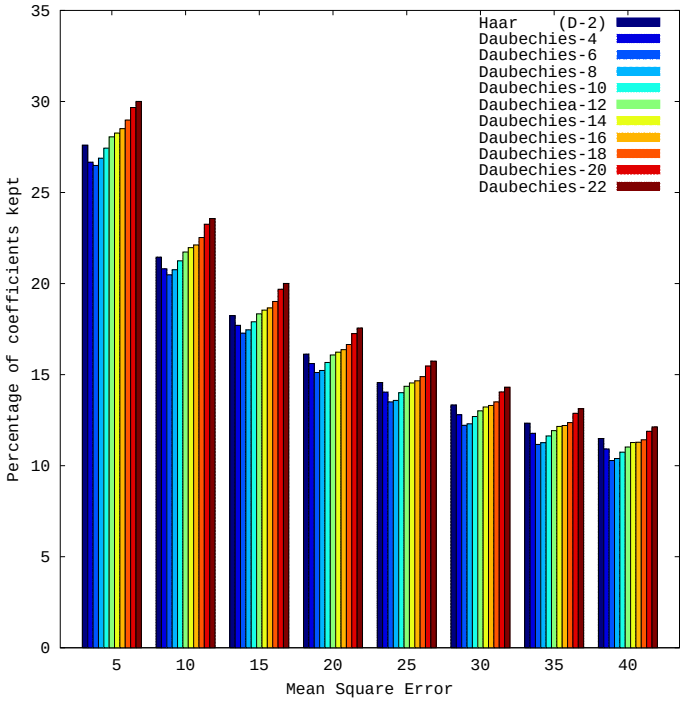

Figure 10: Minimum coefficients in **f_16** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

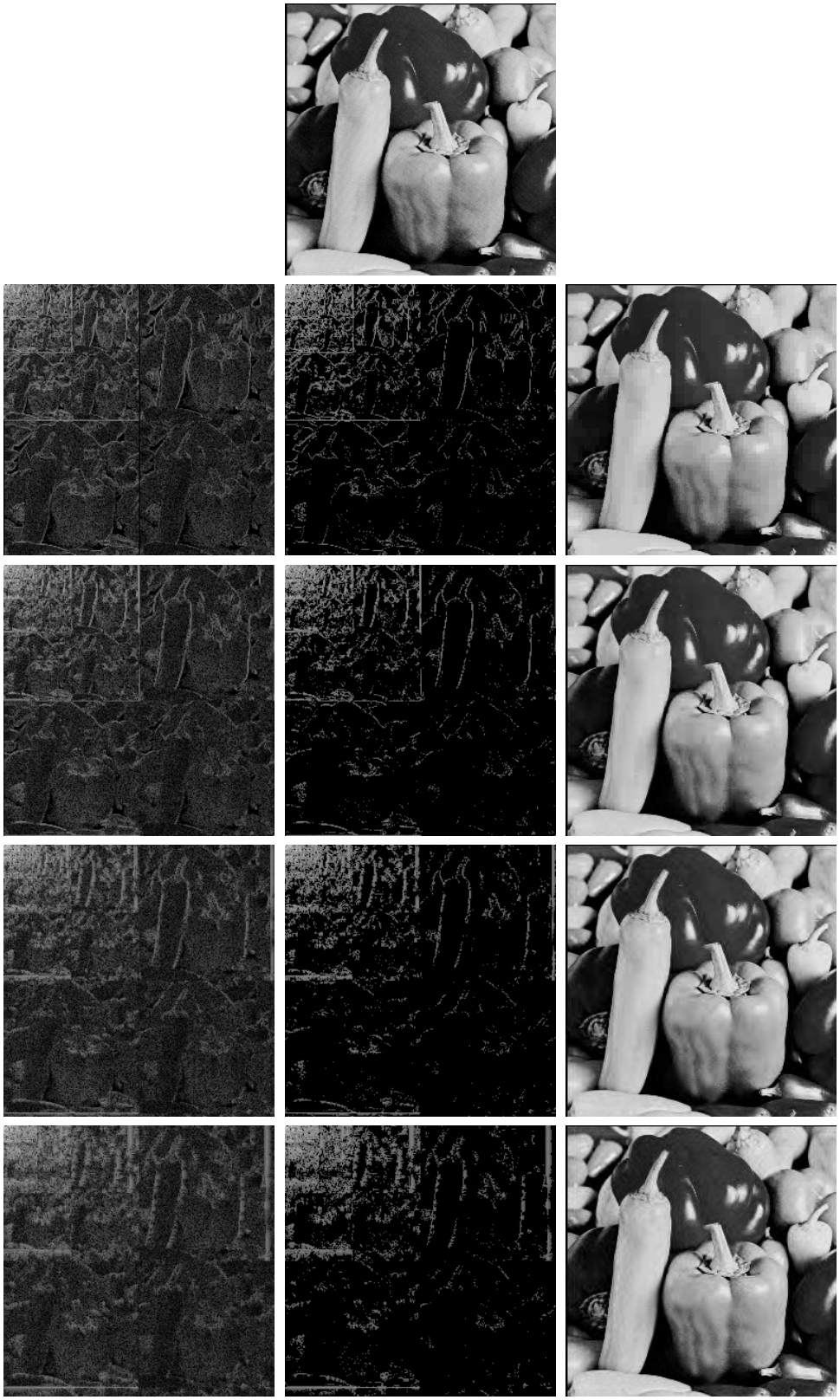

Figure 11: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

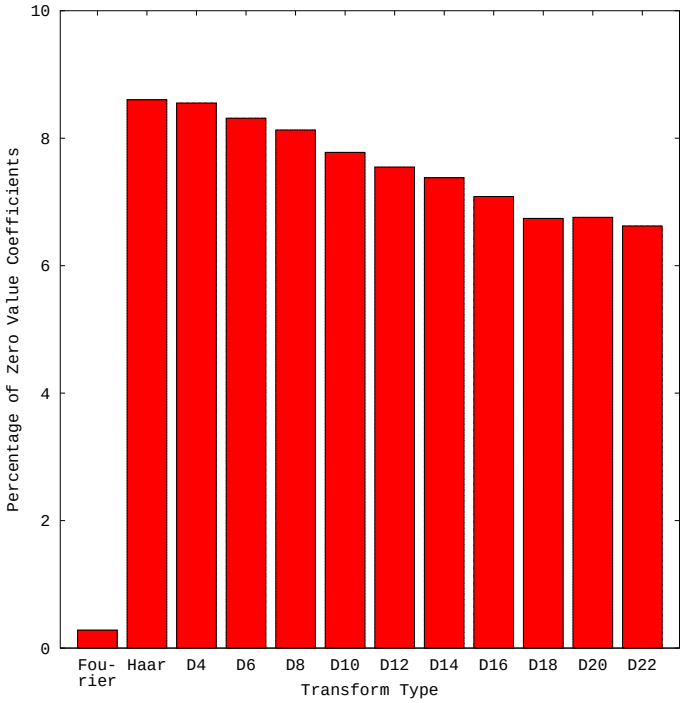

Figure 12: Percentage of coefficients in **peppers** with a value of zero. Uses multiple lengths of the Daubechies Wavelet.

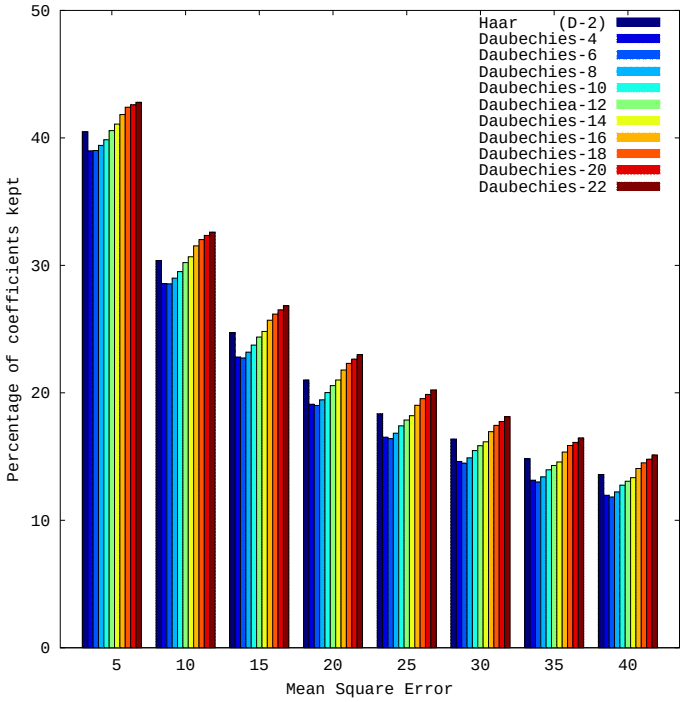

Figure 13: Minimum coefficients in **peppers** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

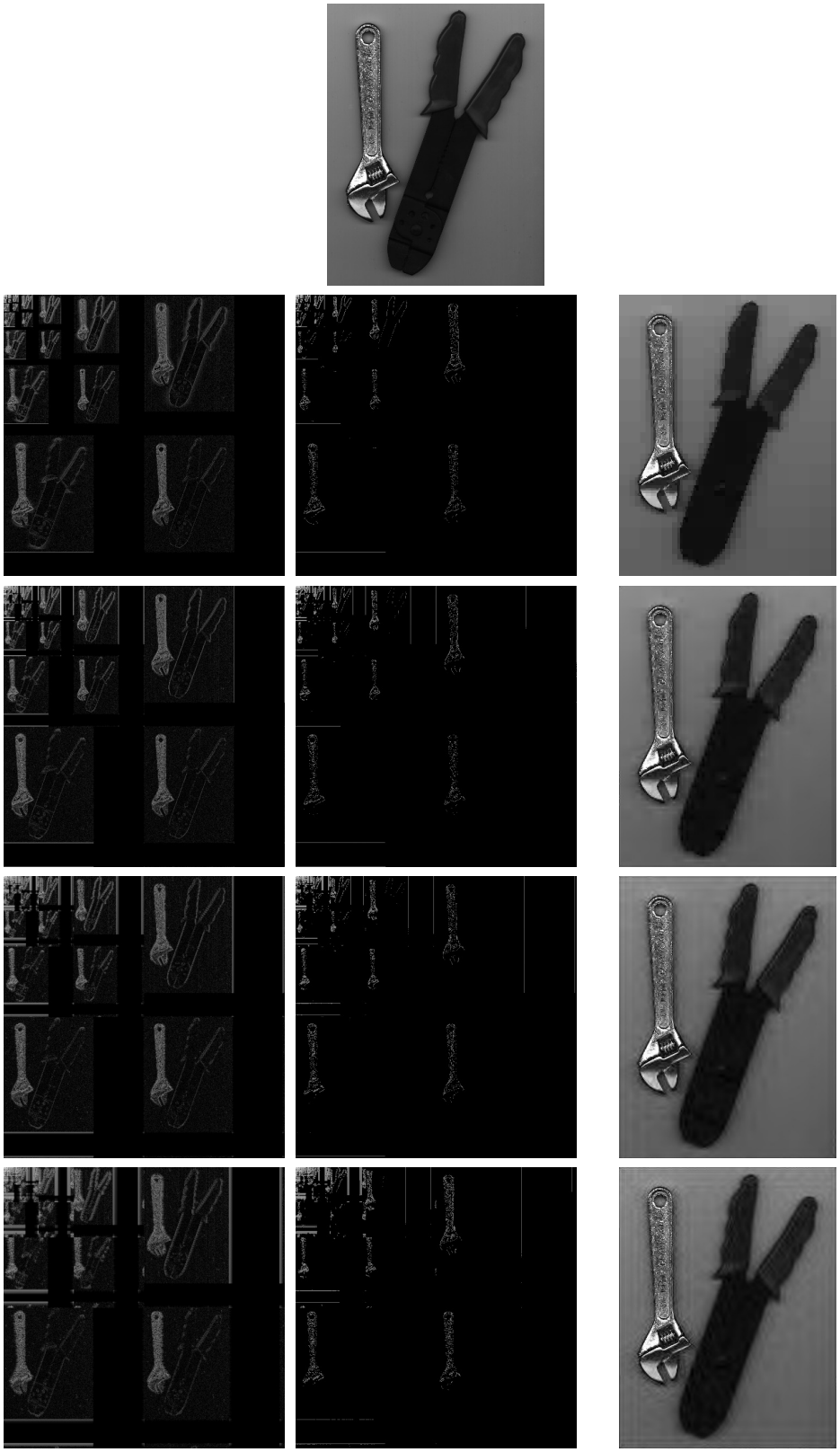

Figure 14: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

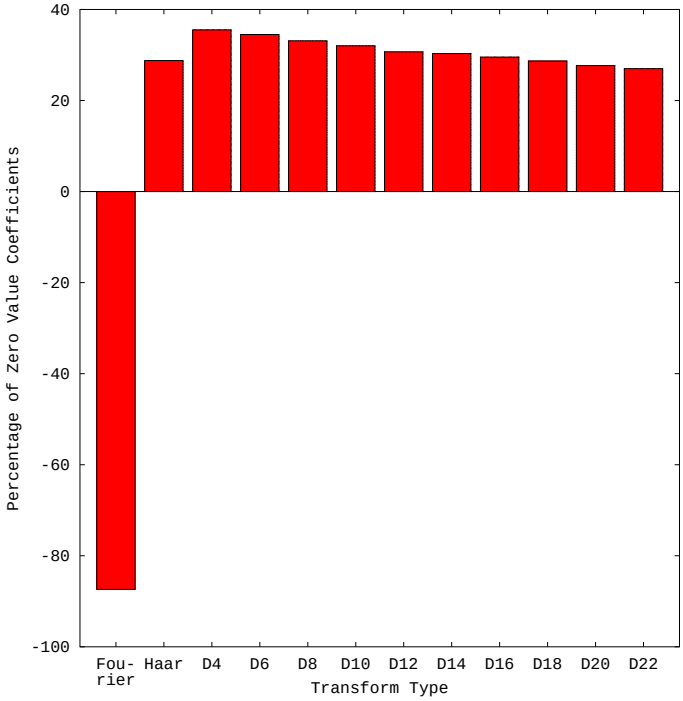

Figure 15: Percentage of coefficients in **tools** with a value of zero. Uses multiple lengths of the Daubechies Wavelet. Please note that this percentage is based on the number of pixels in the original image. Because the FFT requires the image be padded with zeros to a power of 2, more non-zero coefficients exist than pixels in the original image.

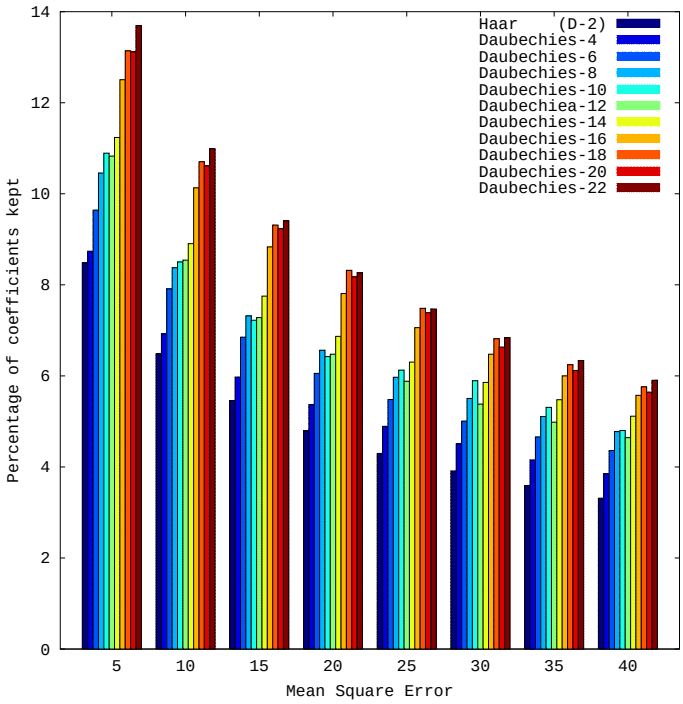

Figure 16: Minimum coefficients in **tools** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

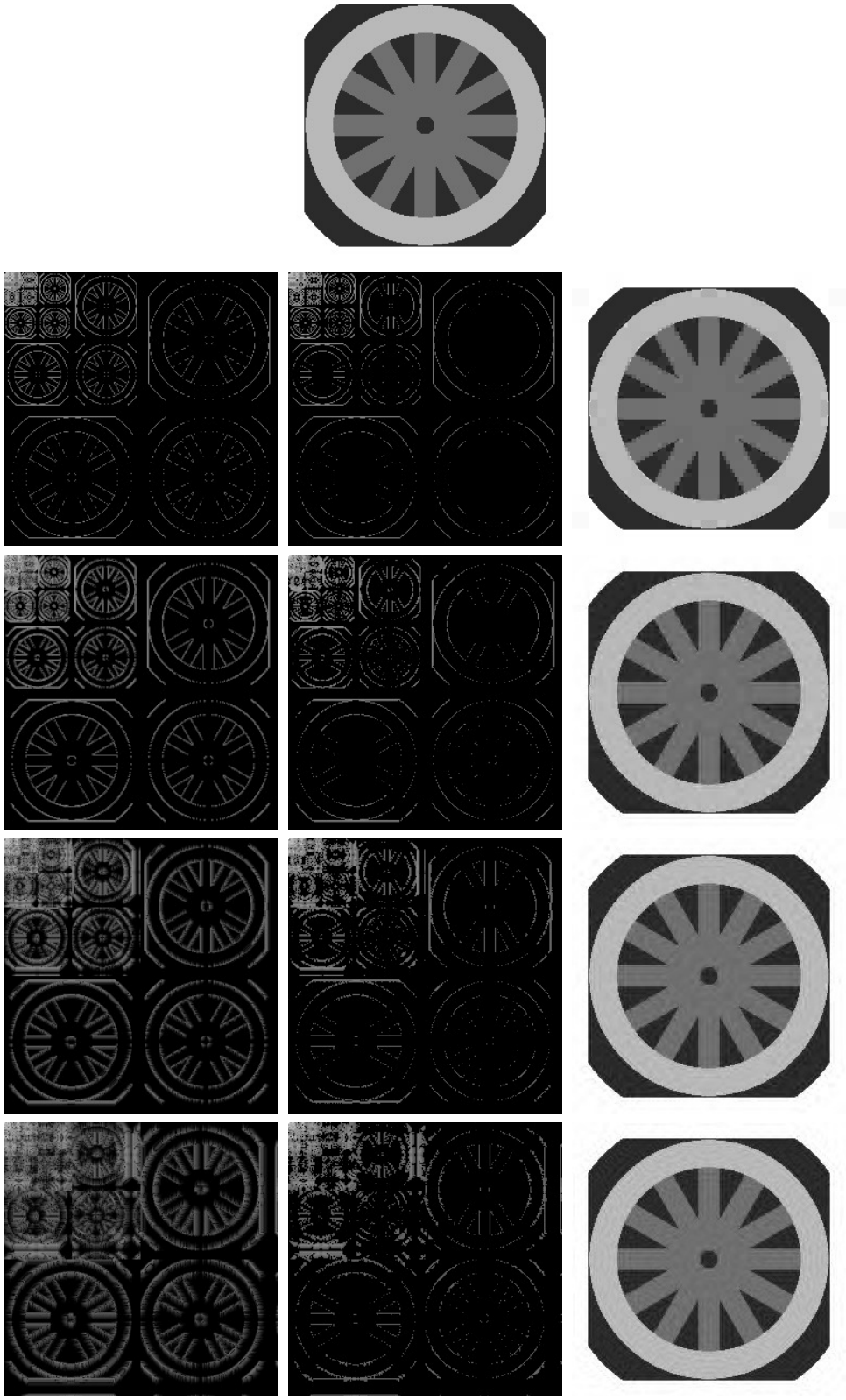

Figure 17: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

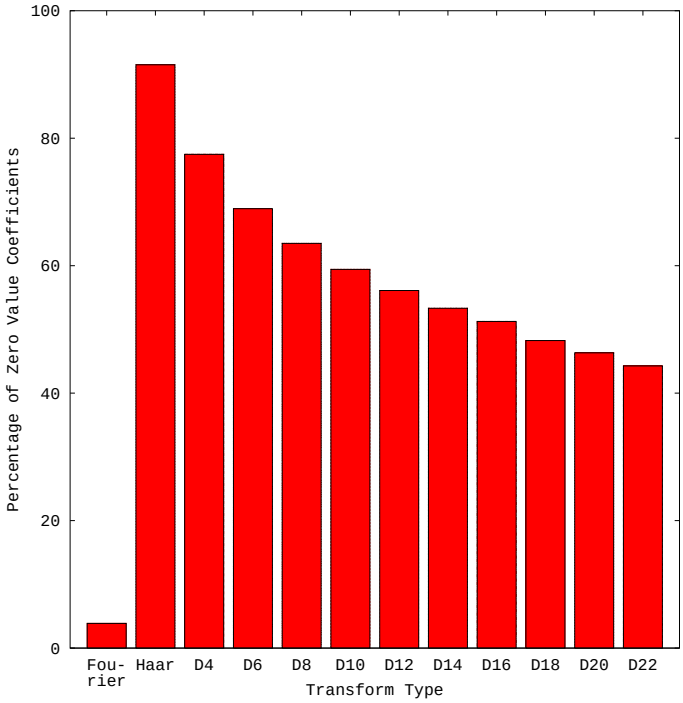

Figure 18: Percentage of coefficients in **wheel** with a value of zero. Uses multiple lengths of the Daubechies Wavelet.

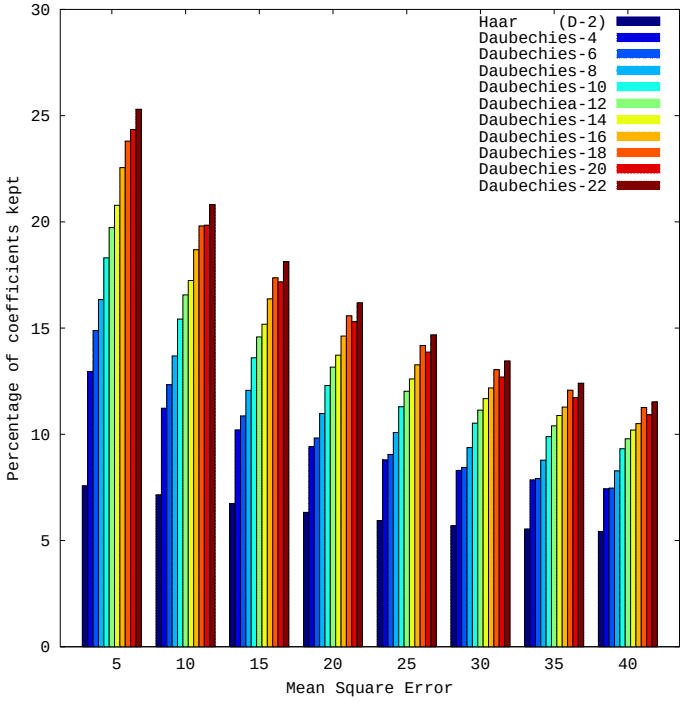

Figure 19: Minimum coefficients in **wheel** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

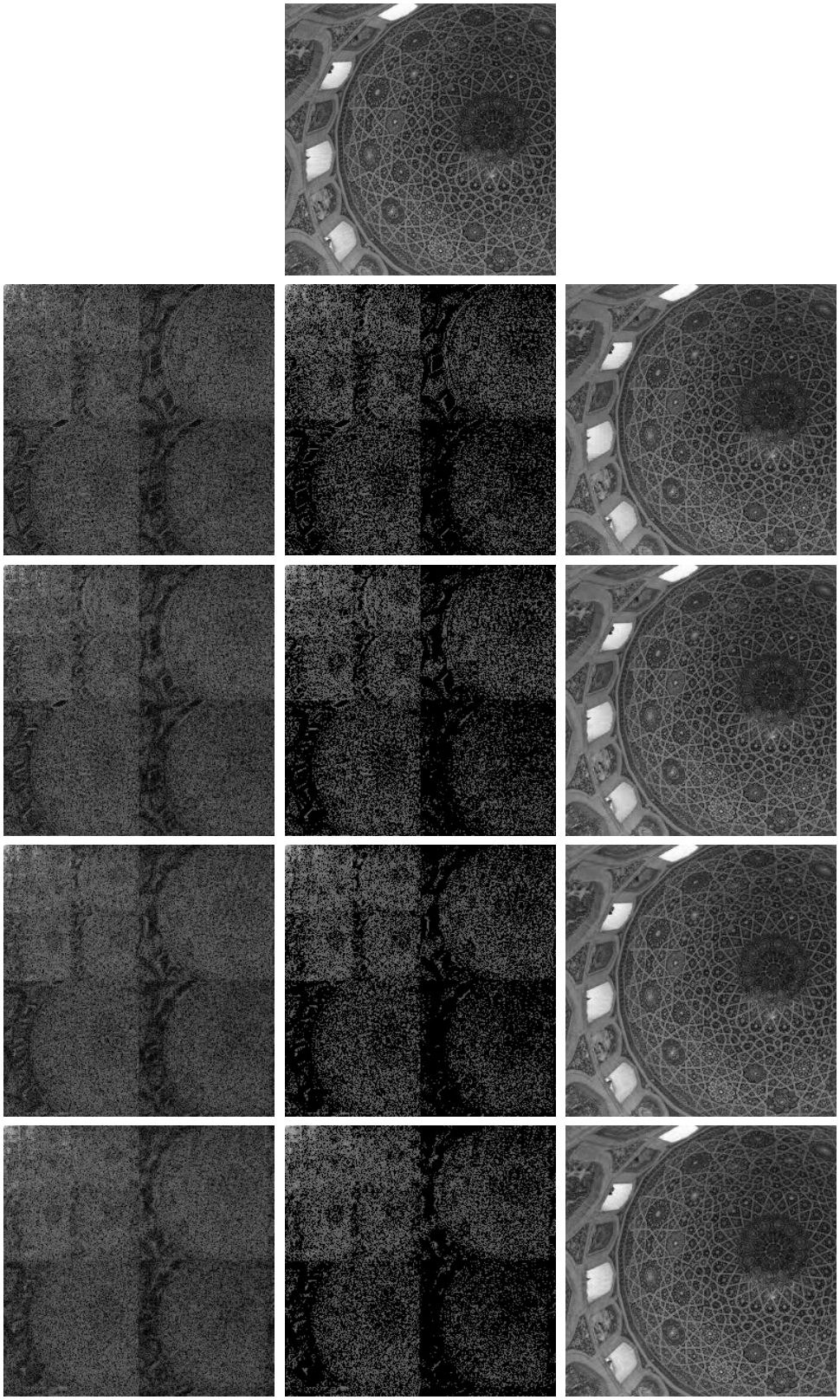

Figure 20: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

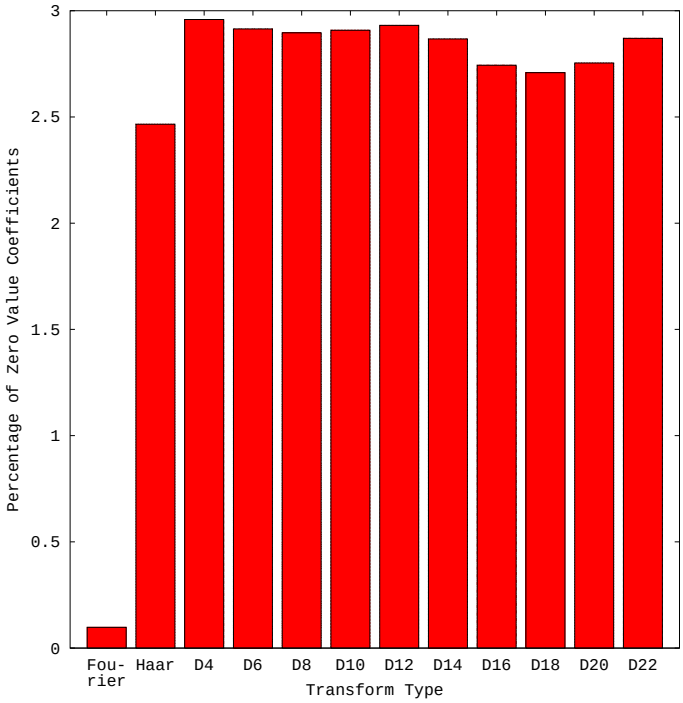

Figure 21: Percentage of coefficients in **konye** with a value of zero. Uses multiple lengths of the Daubechies Wavelet.

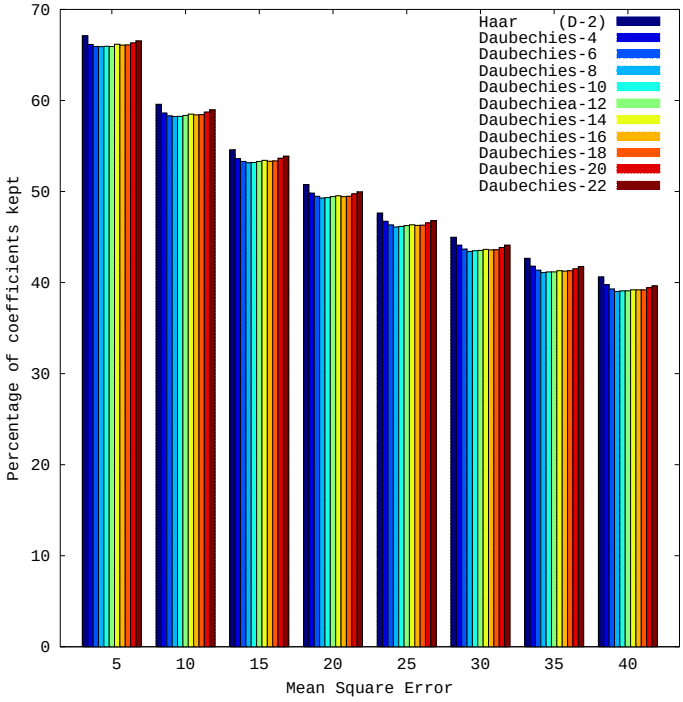

Figure 22: Minimum coefficients in **konye** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

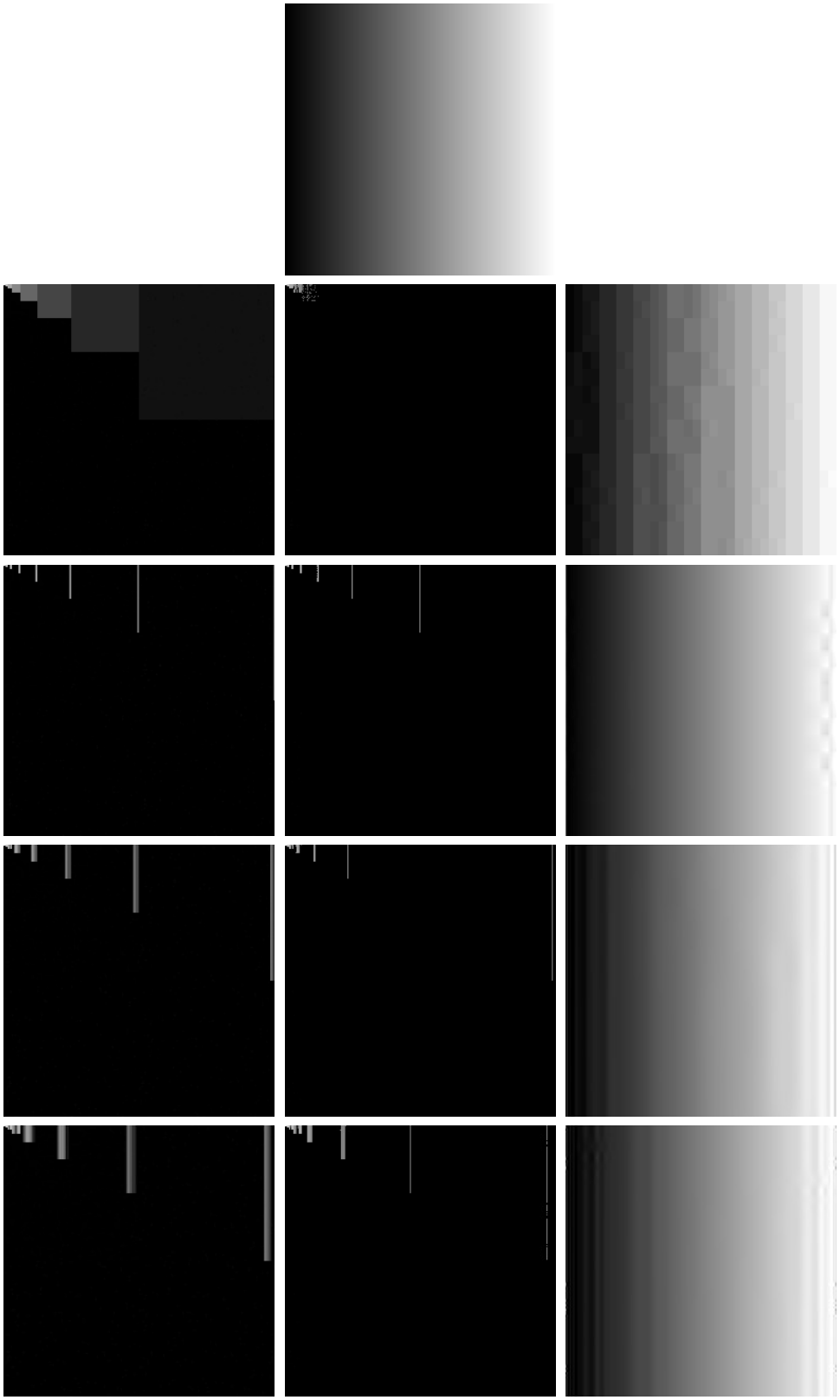

Figure 23: By row the Haar, Daubechies-4, Daubechies-10, and Daubechies-22 wavelets were used respectively. The first column shows the un-modified coefficients. The second column shows the minimum number of coefficients required to maintain a Mean Square Error of 40. The last column shows the reconstructed(lossy) image.

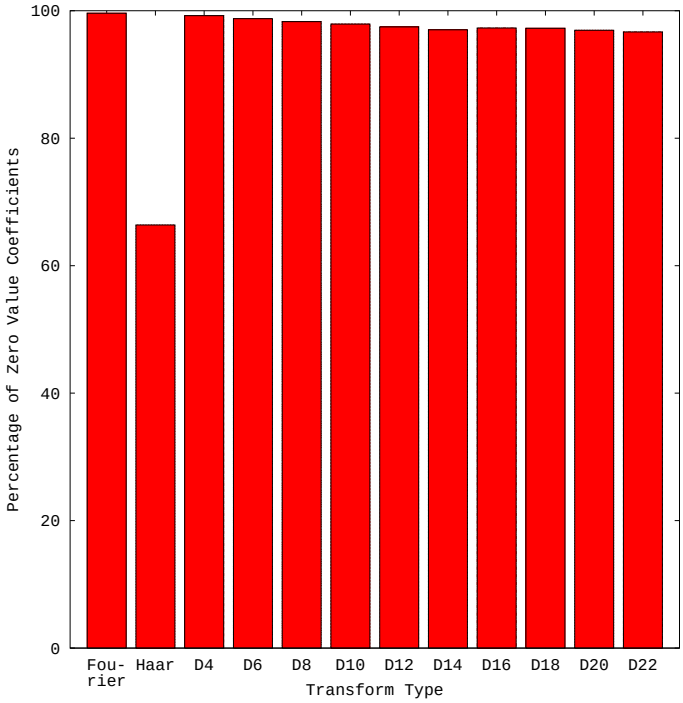

Figure 24: Percentage of coefficients in **gradient** with a value of zero. Uses multiple lengths of the Daubechies Wavelet.

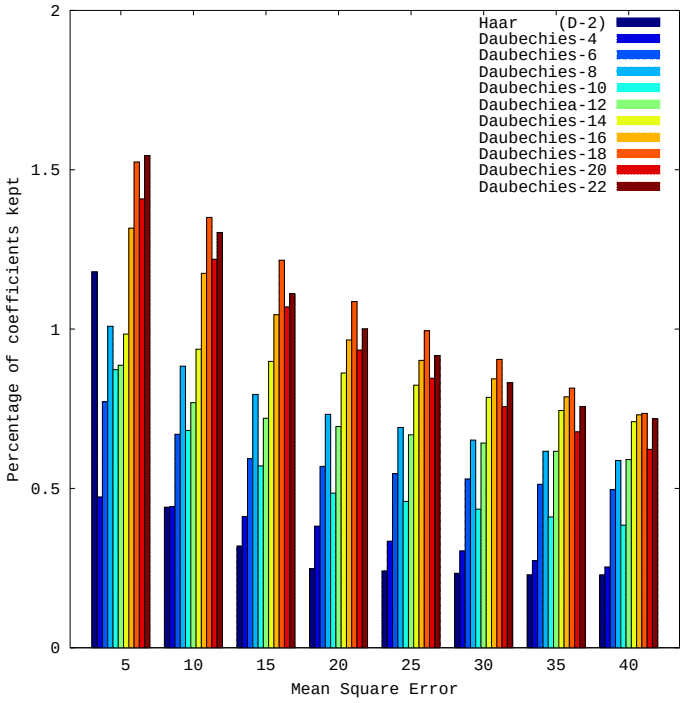

Figure 25: Minimum coefficients in **gradient** required to reconstruct with the desired Mean Square Error. Uses multiple lengths of the Daubechies Wavelet.

7 Source Code

7.1 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
{

// 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 );

// wavelet and inverse wavelet transforms, works given any forward lowpass filter
// the filters length MUST BE EVEN

template <class pType>
void waveletTrans( const Image<pType>& img, Image<pType>& coefs,
  const std::vector<pType>& lowpass );

template <class pType>
void iwaveletTrans( const Image<pType>& img, Image<pType>& coefs,
  const std::vector<pType>& lowpass );

// Function implementation

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);
      else
        f(i,row) *= 1.0/(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 =
    max( img.getWidth(), img.getHeight() ) +
    max( kern.getWidth(), kern.getHeight() );

  int shiftX = min(img.getWidth(), kernel.getWidth())/2;
  int shiftY = 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=NEAREST )
{
  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;

  }
}

// dim is the length of the high and low pass filters
template <class pType>
void waveletTrans( const Image<pType>& img, Image<pType>& coefs,
  const std::vector<pType>& lowpass )
{
  int height = std::pow(2, std::ceil(log(img.getHeight())/log(2)));
  int width = std::pow(2, std::ceil(log(img.getWidth())/log(2)));

  int length = std::max(height,width);

  int dim = static_cast<int>(lowpass.size());

  coefs = img;

  if ( coefs.getHeight() != length || coefs.getWidth() != length )
    coefs.pad(length, length);  // pad image with zeros to make power of 2

  // temporary row/col for calculations
  pType* temp = new pType[length];

  // IDEA: mod index by the length each time for wrapping

  int i, j, row, col, half;
```

```
  // define highpass from lowpass

  std::vector<pType> highpass(dim);

  j = 1;
  for ( i = 0; i < dim; i++ )
  {
    highpass[i] = lowpass[dim-i-1]*j;
    j *= -1;
  }

  while ( length >= 2 )
  {
    half = length / 2;

    // rows
    for ( row = 0; row < length; row++ )
    {
      for ( col = 0; col < half; col++ )
      {
        temp[col] = lowpass[0]*coefs(col*2,row);
        temp[col+half] = highpass[0]*coefs(col*2,row);
        for ( i = 1; i < dim; i++ )
        {
          temp[col] += lowpass[i]*coefs((col*2+i)%length,row);
          temp[col+half] += highpass[i]*coefs((col*2+i)%length,row);
        }
      }

      // copy values back
      for ( col = 0; col < length; col++ )
        coefs(col,row) = temp[col];
    }

    // columns
    for ( col = 0; col < length; col++ )
    {
      for ( row = 0; row < half; row++ )
      {
        temp[row] = lowpass[0]*coefs(col,row*2);
        temp[row+half] = highpass[0]*coefs(col,row*2);
        for ( i = 1; i < dim; i++ )
        {
          temp[row] += lowpass[i]*coefs(col,(row*2+i)%length);
          temp[row+half] += highpass[i]*coefs(col,(row*2+i)%length);
        }
      }

      // copy values back
      for ( row = 0; row < length; row++ )
        coefs(col,row) = temp[row];
    }

    length /= 2;
  }

  delete [] temp;
}

template <class pType>
void iwaveletTrans( const Image<pType>& img, Image<pType>& coefs,
  const std::vector<pType>& lowpass )
{
  int height = std::pow(2, std::ceil(log(img.getHeight())/log(2)));
  int width = std::pow(2, std::ceil(log(img.getWidth())/log(2)));

  int size = std::max(height,width);

  int dim = static_cast<int>(lowpass.size());

  coefs = img;
```

```
if ( coefs.getHeight() != size || coefs.getWidth() != size )
  coefs.pad(size, size);  // pad image with zeros to make power of 2

// temporary row/col for calculations
pType* temp = new pType[size];

int i, j, row, col, half, length = 2;

std::vector<pType> highpass(dim), ilowpass(dim), ihighpass(dim);

// build inverse filters
j = 1;
for ( i = 0; i < dim; i++ )
{
  highpass[i] = lowpass[dim-i-1]*j;
  j *= -1;
}

for ( i = 0; i < dim; i+=2 )
{
  ilowpass[i] = lowpass[dim-i-2];
  ilowpass[i+1] = highpass[dim-i-2];
}

j = 1;
for ( i = 0; i < dim; i++ )
{
  ihighpass[i] = ilowpass[dim-i-1]*j;
  j *= -1;
}

while ( length <= size )
{
  half = length / 2;

  // rows
  for ( row = 0; row < length; row++ )
  {
    for ( col = 0; col < half; col++ )
    {
      // index of array is [i]
      temp[col*2] = 0;
      temp[col*2+1] = 0;

      for ( i = 0; i < dim/2; i++ )
      {
        j = col + i - dim/2 + 1;
        while ( j < 0 ) // adjust back up;
          j += half;

        temp[col*2] += coefs(j, row) * ilowpass[i*2]+coefs(j+half, row)*ilowpass[i*2+1];
        temp[col*2+1] += coefs(j, row) * ihighpass[i*2]+coefs(j+half, row)*ihighpass[i*2+1];
      }
    }

    for ( col = 0; col < length; col++ )
      coefs(col,row) = temp[col];
  }

  // columns
  for ( col = 0; col < length; col++ )
  {
    for ( row = 0; row < half; row++ )
    {
      // start at zero
      temp[row*2] = 0;
      temp[row*2+1] = 0;

      for ( i = 0; i < dim/2; i++ )
      {
        j = row + i - dim/2 + 1;
        while ( j < 0 ) // adjust back up;
          j += half;
```

```
          temp[row*2] += coefs(col, j) * ilowpass[i*2]+coefs(col, j+half)*ilowpass[i*2+1];
          temp[row*2+1] += coefs(col, j) * ihighpass[i*2]+coefs(col, j+half)*ihighpass[i*2+1];
        }

      }

      for ( row = 0; row < length; row++ )
        coefs(col,row) = temp[row];
    }

    length *= 2;
  }
}

}
#endif
```

7.2 filters.h

```
#ifndef DAUBECHIES_FILTERS
#define DAUBECHIES_FILTERS

#include <vector>

using namespace std;

enum Wavelet { D2, D4, D6, D8, D10, D12, D14, D16, D18, D20, D22 };

template <class pType>
class Filters
{
public:
  Filters() : _D2(2), _D4(4), _D6(6), _D8(8), _D10(10),
    _D12(12), _D14(14), _D16(16), _D18(18), _D20(20), _D22(22)
  {
    _D2[0] = 7.07106781186547524400844362104849039284835937688474036588339 8e-01;
    _D2[1] = 7.07106781186547524400844362104849039284835937688474036588339 8e-01;

    _D4[0] = 4.82962913144534143374871599864448683816952419504202275201171 5e-01;
    _D4[1] = 8.36516303737807905575293780916873203459370388348439293495341 4e-01;
    _D4[2] = 2.24143868042013381025972762240400355467883518184271761387168 3e-01;
    _D4[3] = -1.29409522551260381174449418812024164174534450659965256907001 6e-01;

    _D6[0] = 3.32670552950082615998511589139005630012923399245068359708470 5e-01;
    _D6[1] = 8.06891509311092576494493604088713490519297394994823618165092 0e-01;
    _D6[2] = 4.59877502118491570095151942147616720808110177431492306643386 7e-01;
    _D6[3] = -1.35011020010254588696389906699374480562219845223781191975686 2e-01;
    _D6[4] = -8.54412738820266616928191691817733115361976389880866297635174 8e-02;
    _D6[5] = 3.52262918857095366027406647155100293277583879174316103989340 6e-02;

    _D8[0] = 2.30377813308896500863291183044070850001615248248309297791096 8e-01;
    _D8[1] = 7.14846570552915647089921955273992603707608401099308175845011 0e-01;
    _D8[2] = 6.30880767929858907881716338300615220203222922677195117405747 3e-01;
    _D8[3] = -2.79837694168598542114137471800753854119873202244917528400335 8e-02;
    _D8[4] = -1.87034811719093084079570672789081419584544174374580091205777 0e-01;
    _D8[5] = 3.08413818355607636272193625349590501703148217200340334182121 9e-02;
    _D8[6] = 3.28830116668851997354075135492443886645419411375497125972727 8e-02;
    _D8[7] = -1.05974017850690321048832085240272291810999649063764198348497 4e-02;

    _D10[0] = 1.60102397974192914480723748020420733650544124625057832772569 9e-01;
    _D10[1] = 6.03829269797189670540119306525062107507422163101698698796928 3e-01;
    _D10[2] = 7.24308528437772927728071244102218640768756218232007372576733 5e-01;
    _D10[3] = 1.38428145901320731505397146339024697314105791173956102269465 2e-01;
    _D10[4] = -2.42294887066382031862571379474616361991490808062618598391372 6e-01;
    _D10[5] = -3.22448695846383746484797550621349283135649841637984722543426 8e-02;
    _D10[6] = 7.75714938400457135231304893886018198062309945201252798321014 6e-02;
    _D10[7] = -6.24149021279827427419051911292019297076355716568760732341743 5e-03;
    _D10[8] = -1.25807519990819994685097399317757929492045916260978502016923 2e-02;
    _D10[9] = 3.33572528547377127799818341581735574763652474230531509970642 8e-03;

    _D12[0] = 1.11540743350109463621323917240923439042539591984421675908236 0e-01;
```

```
_D12[1] = 4.946238903984530856772041768778555886377863828962743623531834e-01;
_D12[2] = 7.511339080210953506789344984397316855802547833382612009730420e-01;
_D12[3] = 3.152503517091976290859896548109263966495199235172945244404163e-01;
_D12[4] = -2.262646939654398200763145006609034656705401539728969940143487e-01;
_D12[5] = -1.297668675672619355622896058765854608452337492235814701599310e-01;
_D12[6] = 9.750160558732304910234355253812534233983074749525514279893193e-02;
_D12[7] = 2.752286553030572862554083950419321365738758783043454321494202e-02;
_D12[8] = -3.158203931748602956507908069984866905747953237314842337511464e-02;
_D12[9] = 5.538422011614961392519183980465012206110262773864964295476524e-04;
_D12[10] = 4.777257510945510639635975246820707050230501216581434297593254e-03;
_D12[11] = -1.077301085308479564852621609587200035235233609334419689818580e-03;

_D14[0] = 7.785205408500917901996352195789374837918305292795568438702937e-02;
_D14[1] = 3.965393194819173065390003909368428563587151149333287401110499e-01;
_D14[2] = 7.291320908462351199169430703392820517179660611901363782697715e-01;
_D14[3] = 4.697822874051931224715911609744517386817913056787359532392529e-01;
_D14[4] = -1.439060039285649754050683622130460017952735705499084834401753e-01;
_D14[5] = -2.240361849938749826381404202332509644757830896773246552665095e-01;
_D14[6] = 7.130921926683026475087657050112904822711327451412314659575113e-02;
_D14[7] = 8.061260915108307191292248035938190585823820965629489058139218e-02;
_D14[8] = -3.802993693501441357959206160185803585446196938467869898283122e-02;
_D14[9] = -1.657454163066688065410767489170265479204504394820713705239272e-02;
_D14[10] = 1.255099855609984061298988603418777957289474046048710038411818e-02;
_D14[11] = 4.295779729213665211321291228197322228235350396942409742946366e-04;
_D14[12] = -1.801640704047490915268262912739550962585651469641090625323864e-03;
_D14[13] = 3.537137999745202484462958363064254310959060059520040012524275e-04;

_D16[0] = 5.441584224310400995500940520299935503599554294733050397729280e-02;
_D16[1] = 3.128715909142999706591623755057177219497319740370229185698712e-01;
_D16[2] = 6.756307362972898068078007670471831499869115906336364227766759e-01;
_D16[3] = 5.853546836542067127712655200450981944303266678053369055707175e-01;
_D16[4] = -1.582910525634930566738054787646630415774471154502826559735335e-02;
_D16[5] = -2.840155429615469265162031323741647324684350124871451793599204e-01;
_D16[6] = 4.724845739132827703605900098258949861948011288770074644084096e-04;
_D16[7] = 1.287474266204784588570292875097083843022601575556488795577000e-01;
_D16[8] = -1.736930100180754616961614886809598311413086529488394316977315e-02;
_D16[9] = -4.408825393079475150676372323896350189751839190110996472750391e-02;
_D16[10] = 1.398102791739828164872293057263345144239559532934347169146368e-02;
_D16[11] = 8.746094047405776716382743246475640180402147081140676742686747e-03;
_D16[12] = -4.870352993451574310422181557109824016634978512157003764736208e-03;
_D16[13] = -3.917403733769470462980803573237726762752293500738904937244926 94e-04;
_D16[14] = 6.754494064505693663695475738792991218489630013558432103617077e-04;
_D16[15] = -1.174767841247695337306282316988909444086693950311503927620013e-04;

_D18[0] = 3.807794736387834658869765887955118448771714496278417476647192e-02;
_D18[1] = 2.438346746125903537320415816492844155263611085609231361429088e-01;
_D18[2] = 6.048231236901111119030768674342361708959562711896117565333713e-01;
_D18[3] = 6.572880780513005380782126390451732140305858669245918854436034e-01;
_D18[4] = 1.331973858250075761909549458997955536921780768433661136154346e-01;
_D18[5] = -2.932737832791749088064031952421987310438961628589906825725112e-01;
_D18[6] = -9.684078322297646051350813353769660224825458104599099679471267e-02;
_D18[7] = 1.485407493381063801350727175060423024791258577280603060771649e-01;
_D18[8] = 3.072568147933337921231740072037882714105805024670744781503060e-02;
_D18[9] = -6.763282906132997367564227482971901592578790871353739900748331e-02;
_D18[10] = 2.509471148314519575871897499885543315176271993709633321834164e-04;
_D18[11] = 2.236166212367909720537378270269095241855646688308853754721816e-02;
_D18[12] = -4.723204757751397277925707848242465405729514912627938018758526e-03;
_D18[13] = -4.281503682463429834496795002314531876481181811463288374860455e-03;
_D18[14] = 1.847646883056226476619129491125677051121081359600318160732515e-03;
_D18[15] = 2.303857635231959672052163928245421692940662052463711972260006e-04;
_D18[16] = -2.519631889427101369749886842878606607282181543478028214134265e-04;
_D18[17] = 3.934732031627159948068988306589150707782477055517013507359938e-05;

_D20[0] = 2.667005790055555358661744877130858277192498290851289932779975e-02;
_D20[1] = 1.881768000776914890208929736790939942702546758640393484348595e-01;
_D20[2] = 5.272011889317255864817448279595081924981402680840223445318549e-01;
_D20[3] = 6.884590394536035657418717825492358539771364042407339537279681e-01;
_D20[4] = 2.811723436605774607487269984455892876243888859026150413831543e-01;
_D20[5] = -2.498464243273153794161018979207791000564669737132073715013121e-01;
_D20[6] = -1.959462743773770435042992543190981318766776476382778474396781e-01;
_D20[7] = 1.273693403357932600826772332014009770786177480422245995563097e-01;
_D20[8] = 9.305736460357235116035228983545273226942917998946925868063974e-02;
_D20[9] = -7.139414716639708714533609307605064767292611983702150917523756e-02;
```

```
    _D20[10] = -2.945753682187581285828323760141839199388200516064948779769654e-02;
    _D20[11] = 3.321267405934100173976365318215912897978337413267096043323351e-02;
    _D20[12] = 3.606553566956169655423291417133403299517350518618994762730612e-03;
    _D20[13] = -1.073317548333057504431811410651364448111548781143923213370333e-02;
    _D20[14] = 1.395351747052901165789318447957707567660542855688552426721117e-03;
    _D20[15] = 1.992405295185056117158742242640643211762555365514105280067936e-03;
    _D20[16] = -6.858566949597116265613709819265714196625043336786920516211903e-04;
    _D20[17] = -1.164668551292854509514809710258991891527461854347597362819235e-04;
    _D20[18] = 9.358867032006959133405013034222854399688456215297276443521873e-05;
    _D20[19] = -1.326420289452124481243667531226683305749240960605829756400674e-05;

    _D22[0] = 1.869429776147108402543572939561975728967774455921958543286692e-02;
    _D22[1] = 1.440670211506245127951915849361001143023718967556239604318852e-01;
    _D22[2] = 4.498997643560453347688940373853603677806895378648933474599655e-01;
    _D22[3] = 6.856867749162005111209386316963097935940204964567703495051589e-01;
    _D22[4] = 4.119643689479074629259396485710667307430400410187845315697242e-01;
    _D22[5] = -1.622752450274903622405827269985511540744264324212130209649667e-01;
    _D22[6] = -2.742308468179469612021009452835266628648089521775178221905778e-01;
    _D22[7] = 6.604358819668319190061457888126302656753142168940791541113457e-02;
    _D22[8] = 1.498120124663784964066562617044193298588272420267484653796909e-01;
    _D22[9] = -4.647995511668418727161722589023744577223260966848260747450320e-02;
    _D22[10] = -6.643878569502520527899215536971203191819566896079739622858574e-02;
    _D22[11] = 3.133509021904607603094798408303144536358105680880031964936445e-02;
    _D22[12] = 2.084090436018106302294811255656491015157761832734715691126692e-02;
    _D22[13] = -1.536482090620159942619811609958822744014326495773000120205848e-02;
    _D22[14] = -3.340858873014445606090808617982406101930658359499190845656731e-03;
    _D22[15] = 4.928417656059041123170739741708273690285547729915802418397458e-03;
    _D22[16] = -3.085928588151431651754590726278953307180216605078488581921562e-04;
    _D22[17] = -8.930232506662646133900824622648653989879519878620728793133358e-04;
    _D22[18] = 2.491525235528234988712216872666801088221199302855425381971392e-04;
    _D22[19] = 5.443907469936847167357856879576832191936678525600793978043688e-05;
    _D22[20] = -3.463498418698499554128085159974043214506488048233458035943601e-05;
    _D22[21] = 4.494274277236510095415648282310130916410497987383753460571741e-06;
  }

  const vector<pType>& operator()( const Wavelet type ) const
  {
    if ( type == D2 )
      return _D2;
    else if ( type == D4 )
      return _D4;
    else if ( type == D6 )
      return _D6;
    else if ( type == D8 )
      return _D8;
    else if ( type == D10 )
      return _D10;
    else if ( type == D12 )
      return _D12;
    else if ( type == D14 )
      return _D14;
    else if ( type == D16 )
      return _D16;
    else if ( type == D18 )
      return _D18;
    else if ( type == D20 )
      return _D20;
    else // if ( type == D22 )
      return _D22;
  }

private:
  vector<pType> _D2,_D4,_D6,_D8,_D10,_D12,_D14,_D16,_D18,_D20,_D22;
};

#endif // DAUBECHIES_FILTERS
```

7.3 main.cc

```
#include "funcs.h"
#include "filters.h"
#include <iostream>
```

```
#include <sstream>
#include <iomanip>

using namespace std;

template <class pType>
struct Point
{
  Point() {}
  Point(int _x, int _y, pType _val) : x(_x), y(_y), val(_val) {}
  int x, y;
  pType val;
};

struct Settings
{
  string input, output, output_before, output_after;
  double minErr;
  int pixels;
  double percentage;
  Wavelet type;
  bool showImages;
  bool showProgress;
  bool showResults;
};

const int DIVIDER = 17;

// define the filters
const Filters<double> FILTERS;

// Function Prototypes

void readSettings( int argc, char* argv[], Settings& settings );

// Finds the largest values in the image and places them in lst, once found the values
// are SET TO ZERO in img.  If more space is needed for lst then it is resized, lst assumes
// that any values are already sorted inside of it so the first pass it should be passed
// with size = 0
template <class pType>
void findLargest( jdg::Image<pType>& img, vector<Point<pType> >& lst, int size );

// returns the mean square error between the two images
template <class pType>
double mse( const jdg::Image<pType>& img1, const jdg::Image<pType>& img2 );

// removes the most coefficients possible to maintain the mean square error less than maxErr
// coefs will contain the image in the wavelet domain (before inverse)
template <class pType>
int removeCoefs( const jdg::Image<pType>& img, jdg::Image<pType>& coefs, double maxErr,
  Wavelet type, bool showProgress );

// simply removes 'pixels' number of coeficients from the wavelet transform of img
// coefs will contain the image in the wavelet domain (before inverse)
template <class pType>
void removeExactCoefs( const jdg::Image<pType>& img, jdg::Image<pType>& coefs, int pixels,
  Wavelet type );

// Main

int main(int argc, char* argv[])
{
  // read settings
  Settings settings;
  readSettings( argc, argv, settings );

  // initialize variables
  jdg::Image<double> img(settings.input);
  jdg::Image<double> coefs, recons;

  // show original image
  if ( settings.showImages )
    img.show();
```

```
  // save coefficients before image
  if ( settings.output_before != "" )
  {
    jdg::waveletTrans(img, coefs, FILTERS(settings.type));
    coefs.normalize(jdg::MINMAX_LOG, 0.0, 255.0 );
    coefs.save(settings.output_before);
  }

  // remove coefficients
  if ( settings.pixels > 0 )
    removeExactCoefs( img, coefs, settings.pixels, settings.type );
  else if ( settings.percentage > 0.0 )
  {
    settings.pixels = settings.percentage*(img.getHeight()*img.getWidth()-1.0);
    removeExactCoefs( img, coefs, settings.pixels, settings.type );
  }
  else // remove just enough to get the minimum desired error
    settings.pixels =
      removeCoefs( img, coefs, settings.minErr, settings.type, settings.showProgress );

  // reconstruct
  jdg::iwaveletTrans(coefs,recons, FILTERS(settings.type) );

  // save coefficients after image
  if ( settings.output_after != "" )
  {
    coefs.normalize( jdg::MINMAX_LOG, 0.0, 255.0 );
    coefs.save(settings.output_after);
  }

  // pad back to original size
  if ( recons.getWidth() != img.getWidth() || recons.getHeight() != img.getHeight() )
    recons.pad(img.getWidth(), img.getHeight());

  // save results
  if ( settings.output != "" )
    recons.save(settings.output);

  // display statistical results
  if ( settings.showResults )
    cout << "\nFinal Results" << endl
         << "MSE: " << mse(img, recons)
         << "\tUsing " << (100.0*settings.pixels)/(img.getWidth()*img.getHeight()-1.0) << '%'
         << "\t(" << settings.pixels << " coefficients)" << endl;

  // show output image
  if ( settings.showImages )
  {
    recons.normalize(jdg::THRESHOLD, 0.0, 255.0);
    recons.show();
  }

  return 0;
}

// Function implementations

void readSettings( int argc, char* argv[], Settings& settings )
{
  // defaults
  settings.input = "images/lenna.pgm";
  settings.output = "";
  settings.output_before = "";
  settings.output_after = "";
  settings.type = D4;
  settings.minErr = 25.0;
  settings.pixels = -1;
  settings.percentage = -1.0;
  settings.showImages = true;
  settings.showProgress = true;
  settings.showResults = true;
```

```
for ( int i = 0; i < argc; i++ )
{
  if ( strcmp(argv[i],"-d2") == 0 )
    settings.type = D2;
  else if ( strcmp(argv[i],"-d4") == 0 )
    settings.type = D4;
  else if ( strcmp(argv[i],"-d6") == 0 )
    settings.type = D6;
  else if ( strcmp(argv[i],"-d8") == 0 )
    settings.type = D8;
  else if ( strcmp(argv[i],"-d10") == 0 )
    settings.type = D10;
  else if ( strcmp(argv[i],"-d12") == 0 )
    settings.type = D12;
  else if ( strcmp(argv[i],"-d14") == 0 )
    settings.type = D14;
  else if ( strcmp(argv[i],"-d16") == 0 )
    settings.type = D16;
  else if ( strcmp(argv[i],"-d18") == 0 )
    settings.type = D18;
  else if ( strcmp(argv[i],"-d20") == 0 )
    settings.type = D20;
  else if ( strcmp(argv[i],"-d22") == 0 )
    settings.type = D22;
  else if ( strcmp(argv[i],"-hideimages") == 0 )
    settings.showImages = false;
  else if ( strcmp(argv[i],"-hideprogress") == 0 )
    settings.showProgress = false;
  else if ( strcmp(argv[i],"-hideresults") == 0 )
    settings.showResults = false;
  else if ( strcmp(argv[i],"-hideall") == 0 )
    settings.showImages =
    settings.showProgress =
    settings.showResults = false;
  else if ( strcmp(argv[i],"-ocoefbefore") == 0 )
  {
    if ( argc > i )
      settings.output_before = argv[++i];
  }
  else if ( strcmp(argv[i],"-ocoefafter") == 0 )
  {
    if ( argc > i )
      settings.output_after = argv[++i];
  }
  else if ( strcmp(argv[i],"-o") == 0 )
  {
    if ( argc > i )
      settings.output = argv[++i];
  }
  else if ( strcmp(argv[i],"-i") == 0 )
  {
    if ( argc > i )
      settings.input = argv[++i];
  }
  else if ( strcmp(argv[i],"-error") == 0 )
  {
    if ( argc > i )
      settings.minErr = atof(argv[++i]);
  }
  else if ( strcmp(argv[i],"-pixels") == 0 )
  {
    if ( argc > i )
      settings.pixels = atoi(argv[++i]);
  }
  else if ( strcmp(argv[i],"-percentage") == 0 )
  {
    if ( argc > i )
      settings.percentage = atof(argv[++i])/100.0;
  }
  else if ( strcmp(argv[i],"-help") == 0 )
  {
    cout << "Commands are ..." << endl << endl
         << "\t-hideimages           don't display images    (shown by default)" << endl
```

```
          << "\t-hideprogress         hide progress messages  (shown by default)" << endl
          << "\t-hideresults          hide results statistics (shown by default)" << endl
          << "\t-hideall              hide images,progress,results" << endl
          << "\t-o FILENAME           save output file into file name" << endl
          << "\t-i FILENAME           load image to compress  (default images/lenna.pgm)\n"
          << "\nUse only one of the following (default -error 25.0)" << endl << endl
          << "\t-error FLOAT          specify max M.S.Error" << endl
          << "\t-pixels INT           specify exact number of coeficients" << endl
          << "\t-percentage FLOAT     specify exact percentage of pixel coeficients" << endl
          << "\nUse only one of the following (default -d4)" << endl << endl
          << "\t-d2,-d4,-d6,...,-d22  number of coeficients in daubechies kernel"
             "(d2 is haar)" << endl
          << endl;
      exit(0);
    }
  }
}

template <class pType>
double mse( const jdg::Image<pType>& img1, const jdg::Image<pType>& img2 )
{
  // assumes same width and height
  int width = img1.getWidth();
  int height = img1.getHeight();

  double meanerr = 0.0;
  pType diff;
  for ( int x = 0; x < width; x++ )
    for ( int y = 0; y < height; y++ )
    {
      diff = img1(x,y)-img2(x,y);
      meanerr += diff*diff;
    }
  meanerr /= height*width;
  return meanerr;
}

template <class pType>
void removeExactCoefs( const jdg::Image<pType>& img, jdg::Image<pType>& coefs, int pixels,
  Wavelet type )
{
  int height = img.getHeight(),
      width = img.getWidth();

  // create list of points
  vector<Point<pType> > lst;

  // holds all the coeficients, then all the ones that are put into lst are removed
  jdg::Image<pType> smallest(width,height);

  // resize coefs if required
  if ( coefs.getWidth() != width || coefs.getHeight() != height )
    coefs.pad(width,height);

  // wavelet transform
  jdg::waveletTrans( img, coefs, FILTERS(type) );

  if ( pixels >= height*width-1 )
    return; // don't need to remove any :p

  // remove the top left because it is not a coefficient
  pType uLeft = coefs(0,0);
  coefs(0,0) = 0;

  // find the largest pixels
  findLargest( coefs, lst, pixels );

  // clear the image and put only the largest pixels in
  for ( int x = 0; x < width; x++ )
    for ( int y = 0; y < height; y++ )
      coefs(x,y) = 0.0;

  for ( int i = 0; i < pixels; i++ )
```

```
    coefs(lst[i].x, lst[i].y) = lst[i].val;

  // don't forget this one!
  coefs(0,0) = uLeft;
}

template <class pType>
int removeCoefs( const jdg::Image<pType>& img, jdg::Image<pType>& coefs, double maxErr,
  Wavelet type, bool showProgress )
{
  int origHeight = img.getHeight(),
      origWidth = img.getWidth();

  pType uLeft;

  double meanErr = 0.0;

  // create list of points
  vector<Point<pType> > lst;

  // holds all the coeficients, then all the ones that are put into lst are removed
  jdg::Image<pType> smallest(origWidth,origHeight), test(origWidth,origHeight);

  // wavelet transform
  jdg::waveletTrans( img, smallest, FILTERS(type) );

  int height = smallest.getHeight(),
      width = smallest.getHeight();
  int min = 0,
      max = height * width - 1;
  int mid = 0;

  // resize coefs if required
  if ( coefs.getWidth() != width || coefs.getHeight() != height )
    coefs.pad(width,height);

  uLeft = smallest(0,0);  // this needs to be in every image
  smallest(0,0) = 0.0;    // make it so that it's not counted when looking for max

  bool less = false;

  while ( max >= min )
  {
    if ( !less )
      mid = min+(max-min)/DIVIDER;  // growing slowley because it's faster :P
    else
      mid = (min+max)/2;

    findLargest( smallest, lst, mid );

    for ( int x = 0; x < width; x++ )
      for ( int y = 0; y < height; y++ )
        coefs(x,y) = 0.0;

    for ( int i = 0; i < mid; i++ )
      coefs(lst[i].x, lst[i].y) = lst[i].val;

    coefs(0,0) = uLeft;

    jdg::iwaveletTrans( coefs, test, FILTERS(type) );

    if ( test.getWidth() != origWidth || test.getHeight() != origHeight )
      test.pad(origWidth, origHeight);

    meanErr = mse(test, img);

    if ( showProgress )
      cout << "MSE: " << meanErr
           << "\tUsing " << (100.0*mid)/(width*height-1.0) << '%'
           << "\t(" << mid << " pixels)" << endl;

    if ( meanErr > maxErr )
      min = mid+1;  // use more coeffs
```

```
    else
    {
      max = mid-1;  // use less coeffs
      less = true;
    }
  }

  // in case it converges one to low
  if ( meanErr > maxErr )
  {
    mid++;
    if ( mid > (int)lst.size() )
    {
      findLargest( smallest, lst, mid );

      for ( int x = 0; x < width; x++ )
        for ( int y = 0; y < height; y++ )
          coefs(x,y) = 0.0;

      for ( int i = 0; i < mid; i++ )
        coefs(lst[i].x, lst[i].y) = lst[i].val;

      coefs(0,0) = uLeft;

    }
    else  // more likely
      coefs(lst[mid-1].x, lst[mid-1].y) = lst[mid-1].val;
  }

  return mid;
}

template <class pType>
void findLargest( jdg::Image<pType>& img, vector<Point<pType> >& lst, int size )
{
  int len;
  if ( (int)lst.size() < size )
  {
    len = (int)lst.size();
    lst.resize(size);
  }
  else  // already sorted
    return;


  Point<pType> current, temp;
  int height = img.getHeight(),
      width = img.getWidth();

  for ( int x = 0; x < width; x++ )
    for ( int y = 0; y < height; y++ )
      if ( img(x,y) != 0 )
      {
        current.x = x;
        current.y = y;
        current.val = img(x,y);

        // no point in binary search
        if ( len < 20 )
        {
          int i = 0;
          while ( i < len )
          {
            if ( abs(lst[i].val) < abs(current.val) )
            {
              temp = lst[i];
              lst[i] = current;
              current = temp;
            }
            ++i;
          }
          if ( len < size )
          {
```

```
            lst[len] = current;
            len++;
          }
        }
        else if ( abs(lst[len-1].val) >= abs(current.val) ) // special case
        {
          if ( len < size )
          {
            lst[len] = current;
            len++;
          }
        }
        else  // do binary search
        {
          int min,max,mid;
          min = 0;
          max = len-1;
          mid = (max+min)/2;

          // find location
          while ( abs(lst[mid].val) != abs(current.val) && min <= max )
          {
            mid = (max+min)/2;
            if ( abs(current.val) < abs(lst[mid].val) )
              min = mid+1;
            else
              max = mid-1;
          }

          if ( mid + 1 < size )
          {
            if ( len < size )
              len++;

            // need to place into location mid+1
            for ( int i = mid+1; i < len; i++ )
            {
              temp = lst[i];
              lst[i] = current;
              current = temp;
            }
          }
        }
      } // end loop
  // remove values from img
  for ( int i = 0; i < size; i++ )
    img(lst[i].x, lst[i].y) = 0;
}
```

7.4 countzeros.cc

```
#include "funcs.h"
#include "filters.h"
#include <iostream>
#include <fstream>

using namespace std;
using namespace jdg;

// count the number of coefficients in different transforms and output to file
int main( int argc, char* argv[] )
{
  if ( argc < 2 )
    return 0;

  Image<double> orig(argv[1]);
  Image<complex<double> > fourier = orig;
  Image<double> show, wavelet[11];
  Filters<double> filts;
  int wzeros[11], fzeros;

  ofstream fout;
```

```
  if ( argc > 2 )
  {
    fout.open(argv[2]);
    cout << "Outputing to file " << argv[2] << endl;
  }

  // fourier
  fft( fourier );

  fzeros =
    orig.getWidth() * orig.getHeight() - fourier.getWidth() * fourier.getHeight();


  for ( int x = 0; x < fourier.getWidth(); x++ )
    for ( int y = 0; y < fourier.getHeight(); y++ )
      if ( abs(fourier(x,y)) < 0.5 )
        fzeros++;

  cout << "Fourier: " << fzeros << endl;

  if ( argc > 2 )
    fout << 100.0*fzeros/(orig.getWidth() * orig.getHeight()) << ' ';

  // wavelets
  for ( int i = 0; i < 11; i++ )
  {
    waveletTrans( orig, wavelet[i], filts((Wavelet)i) );
    wzeros[i] =
      orig.getWidth() * orig.getHeight() - wavelet[i].getWidth() * wavelet[i].getHeight();
    for ( int x = 0; x < fourier.getWidth(); x++ )
      for ( int y = 0; y < fourier.getHeight(); y++ )
        if ( abs(wavelet[i](x,y)) < 0.5 )
          wzeros[i]++;

    cout << "D" << (i+1)*2 << ": " << wzeros[i] << endl;

    if ( argc > 2 )
      fout << 100.0*wzeros[i]/(orig.getWidth() * orig.getHeight()) << ' ';
  }

  if ( argc > 2 )
  {
    fout << endl;
    fout.close();
  }

  return 0;
}
```