Image Processing Project 2012
Tel Aviv University

Print-Scan Resilient Watermarking in the Wavelet Domain

Tomer Filiba
037013075

Motivation

The original intent of this work was to implement an invisible watermarking technique that could reliably survive the process of printing and scanning. The motivation was to find an **alternative to QR codes**, which are used today in many billboard ads to provide a machine-readable link to the product's web site, targeting the growing mobile-phone market.

Instead of including a large, visible, monochrome barcode within the image (which also consumes precious advertising real-estate), I wanted to embed the URL (the *payload*) into the image using an **invisible watermark**, in hopes that the resulting image could then be scanned by a dedicated mobile-phone application, which could extract the URL out and point the browser at the product's site.

Recovering a printed-and-scanned watermark from an image turned out more complicated than I had thought, but nonetheless, I believe this approach is feasible and has real value the advertising business.

Theory

Quick Response (QR) codes are two-dimensional (*matrix*) barcodes that encode machine-readable data as small, monochrome images. Unlike traditional (one-dimensional) barcodes, QR codes provide a much higher data capacity, and were designed for quick decoding by low-end devices.

The encoding was developed by the Japanese corporation Denso Wave in 1994, and was first used in the automotive industry for tracking vehicle parts during manufacturing. In the year 2000, QR codes became an ISO standard, and Denso Corporation has committed not to practice any patent-rights it owns, making them free to use for everyone.

With the growth of the smartphone market, advertisers looked for ways to engage their potential customers in the ads they were creating. Because of the limitations of smartphones and their keyboards, instead of referring users to a human-readable URL, advertisers began to incorporate a QR code on the ad, with the link to the firm's site. They usually even offer discounts to consumers

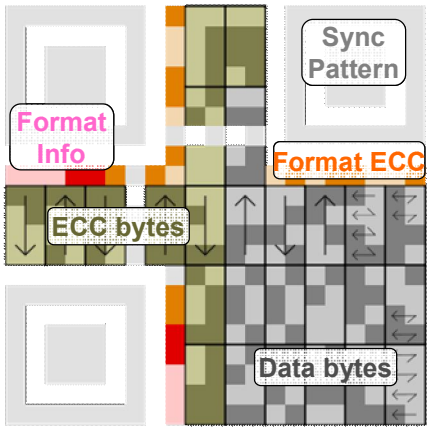
who scan their ads, because it allows them to measure conversion rates and the reach of their campaigns, while increasing turnover. In quite a short while, scanning ads quickly had become popular with both consumers and advertisers.

QR codes combine the payload (the data your wish to convey) with Reed-Solomon Error Correcting Code (ECC), making decoding possible with up to 30% image corruption. This is very important in the printed world, as lighting condition and camera quality range widely (which affects gray-level quantization), and physical damage is common.

However, QR codes are quite unappealing to the naked eye (they look like random black-and-white dots), and because they are meant to be scanned from a distance, they usually make up a substantial portion of the ad's area (up to 50%). Apart from their size, the standard mandates sufficiently-large blank margins around the code, as well as noticeable synchronization blocks. This makes them even less appealing in the context of the ad.

My goal was to embed machine-readable data in the picture, which is invisible to the naked eye. For this I investigated invisible watermarking techniques as described in several papers (listed in the references), and ultimately settled for spread-spectrum watermarking in the wavelet domain. Before I get to describing that, there's some background to cover first, as well as a survey the different watermarking techniques that I had studied.

Introduction to QR Codes



Since I aimed at replacing QR codes, I began by studying their properties and structure. The standard itself is quite long, so I will cover only a portion of it.

The most prominent visual feature of QR codes is the three large synchronization blocks at the top-left, top-right and bottom-left corners. These, along with a smaller forth one near the bottom-right corner are used to align the image and rescale its *modules* (black or white dots, which represent bits). Depending on its version (1-40) and error correction level (L, M, Q, H), it can store between 9 to 2,953 bytes. Error correction level L can correct up to 7% errors, while H reaches up to 30%.

The data can be encoded as one of four encodings: bytes (octets), numbers (3 digits in 10 bits), alphanumeric (2 characters in 11 bits) or Kanji (13 bits per character).

Introduction to Reed Solomon ECC

Reed Solomon (RS) is a well known and vastly used ECC that was devised in the 1960's for satellite communication, but because of its computational properties and excellent error correction capabilities, it has found its way to virtually all transmission technologies, DVDs, data storage (RAID6) and even most two-dimensional barcode systems, including QR codes!

Reed-Solomon is member of a larger family of ECC known as **BCH codes**, which range in their complexity (Reed-Solomon being a more complex one). Unlike many ECC algorithms which work at the bit-level, RS code works at the byte (octet) level, meaning several bit errors within the same octet count as a single error.

The input string is treated as a polynomial whose coefficients come from an 8-bit finite (Galois) field, with multiplication and division defined as is customary for polynomials. The properties of this finite field allow us to multiply and divide polynomials in logarithmic time using lookup tables, so the more complex process of long multiplication and division is not necessary.

For a given number of ECC bytes (n), we define a **generator polynomial** $g(x) = (x-a^n)(x-\alpha^{n-1})\ldots(x-\alpha)(x-\alpha^0)$, where α is normally taken as 2. We then take the input polynomial, padded by n zeros to the right, and it divide by the generator polynomial to obtain the remainder. For instance, if our input data is [12 34 56] and n is 4, we divide [12 34 56 00 00 00 00] by [01 0F 36 78 40], which is the generator polynomial for n=4. The remainder is [37 E6 78 D9], so the result is [12 34 56 37 E6 78 D9] (we care only for the polynomial's coefficients).

Decoding of a RS-encoded message begins with calculating the message's **syndrome**, which is a polynomial defined by treating the message itself as a polynomial and evaluating it at α^0, α^1, α^2, …, α^n. Since these are the generator polynomial's zeros, if the message is in tact, the syndrome will be zero. Otherwise, it means the message is damaged, and we may be able to correct it.

If the location of the errors is known, a single byte of ECC can be used to counter the error. Otherwise, it takes two bytes of ECC to locate and counter the error. Forney's algorithm is used to correct errors at known locations, and the Berlekamp-Massey algorithm is used to locate errors. The algorithmic implementation is much simpler than the mathematical background required to fully understand it, so this introduction will suffice for our purposes.

A Survey of Invisible Watermarking Techniques

The purpose of watermarking schemes is to embed additional information (payload) into an image, such that it could be later on retrieved from the image. Watermarking may be visible (such as text overlaid on the image), where they usually serve to show the source of the image, or invisible. Invisible watermarking may be **fragile**, which means any modification to the image destroys the watermark. This can be used like digital signatures, to prove the authenticity of an image. Another kind of invisible watermarks is designed to be **robust**, meaning it should survive various *attacks*, such as cropping, rotation, addition of noise, smoothing, and other image-processing filters. This kind of watermarks is usually used for Digital Rights Management (DRM), to embed copyright information or a trackable, unique identifier into the media.

Spatial Domain

The simplest forms of invisible watermarking are done in the spatial domain, meaning, directly into the color/intensity channels of the image. The most widely

known watermark in this family is called **LSB-watermarking**, where the least-significant bit of each pixel is used to encode the payload. LSB watermakring alters the intensity levels of the image by up to one unit, making it completely indiscernible to the naked eye. Additionally, this scheme has a very high data capacity: each pixel encodes one bit; if we use color images, then each pixel can encode three bits (one in each color channel).

On the other hand, this scheme is very **unreliable**: it will only work on bitmaps (or other lossless formats); any attempt to compress the image (using JPEG, for instance) will obliterate the payload. More so, it is highly susceptible to noise, and a **malicious attacker** can very easily wipe the payload out of the image (while preserving the image's fidelity) simply by setting each pixel's LSB to zero. What concerns us more, however, is the fact we're working in the spatial-domain of intensities: any change in lighting conditions will garble up the payload. This makes LSB watermarking unsuitable for the print-scan world.

Another approach is to use embed information as a series of pseudo-random (PR) sequences: for a message of n bits, we choose n PR sequences (s_1, … s_n), each of the size of the host image. Then, if the i^{th} bit is 1, we add $k \cdot s_i$ to the image (as additive noise), and do nothing if the bit is 0. K is the constant known as the *gain factor*, which controls the “scale” of the noise added – the larger k is, the more the noise will be noticeable.

In order to extract the payload, we’d generate the same sequences again, by using the same seed for the pseudo random number generator (PRNG). For each sequence s_i, we test the correlation of s_i with the image. If the correlation coefficient is greater than some threshold value, we take b_i to be 1; otherwise b_i is taken as 0.

Instead of using a fixed threshold, we can use 2 sets of PR sequences: s_1, …, s_n and t_1, …, t_n. During encoding, if b_i is 1, we'd add $k \cdot s_i$; otherwise, we'll add $k \cdot t_i$. During decoding, we'd test the correlation of the image with s_i and t_i; $b_i = 1$ if $corr_{si} > corr_{ti}$ and 0 otherwise.

This is known as Code-Division Multiple Access (**CDMA**), a spread-spectrum technique in which all transmitters (the different bits of the message, in our case) use the same spectrum (the host image) simultaneously, resulting basically in white noise. However, using correlation techniques, we are able to discern the different sequences and decode the data.

CDMA is more robust than LSB watermarking, as breaking the correlation of long sequences of bits is much harder than simply masking out the LSB. In fact, we can think of it as **encryption scheme** where the key is the seed for PRNG. Trying to extract or remove the payload without the knowing the seed is near impossible, as it's basically additive noise.

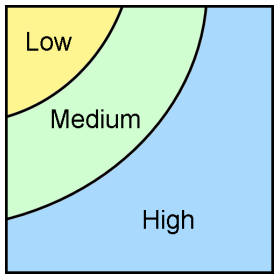
On the other hand, CDMA offers lower data capacity: the longer the payload, the noisier the image will be, and a larger gain factor will be required to successfully correlate the sequences. More so, embedding the payload into the **spatial domain** means the noise is directly visible, and it also makes us lighting conditions and lossy compressions.

Frequency Domain

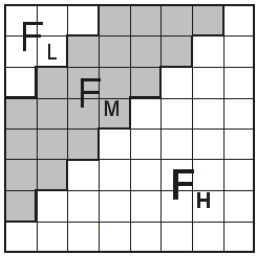
Instead of embedding the payload into the spatial domain, it would be wise to move to the **frequency domain**, using transformations like the Discrete Fourier Transform (**DFT**), which generates complex values, or the Discrete Cosine Transform (**DCT**), which generates only real values. It is easier to exploit the properties of the human eye in the frequency domain, as the eye mostly notices low and medium frequencies; the perceptibility of high frequencies is very low.

In fact, JPEG compression works exactly this way: it splits the image into blocks of 8×8 pixels, performs a DCT on each and assigns weights to each frequency, known as the **quantization matrix**. These weights start small for low frequencies and grow as we reach higher ones. The transformed 8×8 matrix is then point-wise divided by the quantization values, thus discarding frequencies our eye can hardly notice.

In order to achieve even better compression and higher fidelity, JPEG first transforms the RGB color-space to YCbCr (where Y represents intensity). JPEG preserves the Y channel's full range, as our eye is very sensitive to intensity, but it may compress the Cb and Cr channels, as the sensitivity to hue and saturation is lower.



We can approach frequency-domain watermarking in several ways. For instance, we might transform the image using DFT/DCT, apply CDMA techniques on the resulting matrix, and use the inverse transform to return to the spatial domain. This way, the payload would be spread over all pixels, making it less perceptible and more robust. However, most of the area in the DFT matrix falls in the high frequency range (the blue area in the diagram to the right); information in this band is less likely to survive compressions like JPEG, or even simple low-pass filters like the Gaussian. On the other hand, embedding noise into the low band would result in large amounts of visible noise, as the eye is very sensitive to this band. Therefore, the only sensible band to use is the medium one, which is quite narrow (about 25% of the matrix), resulting in low data capacity.



Another approach, suggested in [Shoemaker 2002], is to exploit the way JPEG works in order to make the embedding JPEG-resilient. As in JPEG, we split the image to 8×8 blocks, perform DCT on each, and embed a single bit into the middle band (F_M in the diagram to the right), using any suitable embedding technique (e.g., CDMA). The benefit here is, we **separately** embed one bit per 64 pixels, thus keeping the visible noise level low and overcoming the limitations of JPEG compression (as the middle band is less likely to lose information). However, this scheme is tailored to the way JPEG works, and may not survive other compression schemes. Moreover, it is too sensitive to alignment issues: misalignment of a single pixel would render the payload useless, which makes it is unsuitable for the print-scan world.

Wavelet Domain

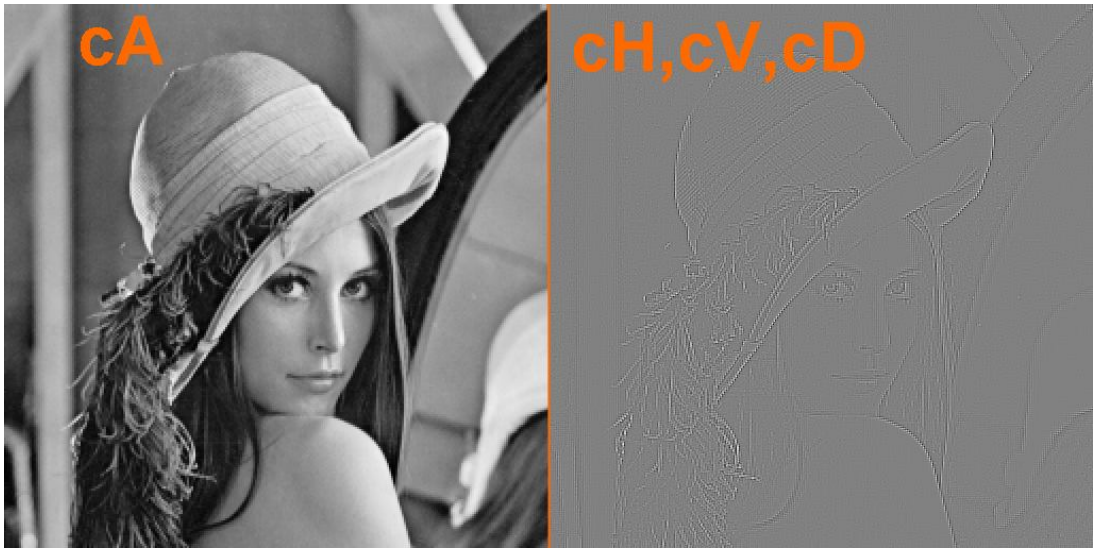
The Discrete Wavelet Transform (DWT) is similar in concept to DFT, but instead of using a basis of periodic functions (sines and cosines), it relies on **wavelets**. In the continuous case, it decomposes a signal into a series of square-integrable functions over a complete (or over-complete) orthonormal basis, generated by the **mother-wavelet**. Wavelets themselves are functions that start out as zero, oscillate in some way for a certain duration, and then go back to zero.

In the two-dimensional discrete case, DWT decomposes a given matrix into four sub-matrices (each ¼ of the size of the original), called cA, cH, cV and cD – for approximation, horizontal detail, vertical detail and diagonal detail, respectively. The approximation matrix contains the low frequencies of the image, while cH, cV and cD contain the frequencies in their respective direction. Below are the image of Lena (left) and the result of DWT on that image (right), using Haar as the mother wavelet (shown to scale):

Unlike the Fourier transform, which discards all spatial information, the wavelet transform has the property of capturing **both frequency and temporal** (location) information – as demonstrated by the matrices above.

Wavelets are used in the image processing world mainly for compression, as they tend to separate the image into more compressible subsets (used by JPEG-2000), and the fact they can be recursively applied lends them to many applications, such as *multiresolution analysis*.

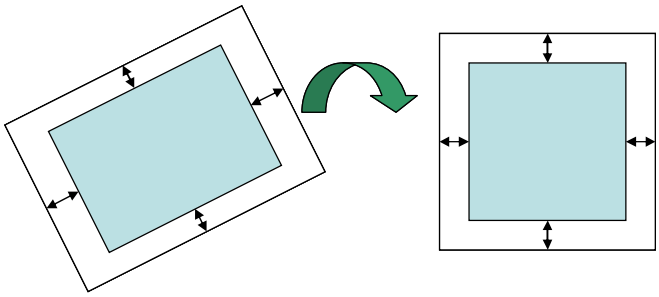
As we are more concerned with the visual properties of the wavelet transform than with its mathematical background, it's important to note that the approximation matrix (containing the lower frequencies) is the **most perceptible** one. The other matrices make up the higher frequencies (finer detail) and are thus less discernible by the human eye. The image below demonstrates the tremendous perceptual difference between the approximation (made by inverse DWT of **only** cA) and the finer detail (the inverse DWT of cH, cV and cD, **without** cA):



Therefore, when we come to embed a payload into the wavelet domain, it's important to take into consideration the visual properties of the four matrices. For instance, embedding the payload into cA would make it directly noticeable, while embedding it into the finer detail matrices would increase the image's fidelity.

Other than low perceptibility, watermarking in the wavelet domain has several desired properties. For example, because we're embedding in a frequency-like domain and because JPEG analyzes frequencies, the result survives JPEG compression better. For the same reason, the result is also less sensitive to additive noise (which takes place in the spatial domain). And lastly, the DWT transform is quite insensitive to lighting conditions and color in general – making it suitable for the print-scan world. All of these properties will be explored in the analysis.

On the other hand, because the wavelet transform encodes location information as well as frequencies, it is **very sensitive to scale and rotation**. These, however, can be corrected by visual cues, like adding a white margin around the image, of a known size, or marking the four corners, so the algorithm could first align the image and resize it. However, alignment is beyond the scope of this project.



Implementation

My initial thought was to encode a message (a URL) as a QR code, which is a binary image, and then embed this payload image into the host image. As QR codes already contain error correction and alignment features, I hoped I would be able to utilize them for my purposes, thus begin from the actual watermarking. However, it quickly became apparent that the QR encoding of the message is too long for maintaining a low noise level and allowing effective extraction. Moreover, the alignment features of QR could not be used to align the host image, so it turned out futile.

It is worth noting that most watermarking schemes usually constrain themselves to embedding payload **images** into host images, as the end user is normally a human. This means that as long as the noise level in the extracted image is sufficiently low, the human inspector can use it. In our case, we're interested in embedding a binary message that would be **machine-readable**; therefore, we cannot rely on the reader to cope with noise on its own – we must be able to correct errors mechanically (up to some threshold). For this, we first take the *message* (a binary string) and add Reed-Solomon ECC to it, yielding the final *payload* that would go into the image.

Once we have the payload ready, the **embedding** algorithm proceeds as follows: we begin by performing a DWT decomposition of the host image into cA, cH, cV, and cD matrices. Next, we take cH, cV and cD and interleave them together, obtaining a long vector of coefficients. We then split this vector into n chunks of equal size, where n is the number of bits of the payload. A single bit of the payload is then embedded into each chunk, using CDMA techniques, and the vector is then deinterleaved into cH', cV' and cD', which, combined with the original cA, are transformed back into the spatial domain by the inverse DWT. If the image contains color channels, we repeat the process for each channel separately. In pseudo-code:

```
FUNCTION embed(img, payload, seed, k, mother):
    cA, cH, cV, cD ← dwt2(img, mother)
    vec ← interleave(cH, cV, cD)
    chunk_size ← ⌊length(vec) / length_in_bits(payload)⌋
    seq0, seq1 ← generate_sequences(seed, chunk_size)
    for each bit in payload:
        bitseq ← if (bit = 1) then seq1 else seq0
        vec[i*chunk_size : (i+1)*chunk_size] += k * bitseq
    cH', cV', cD' ← deinterleave(vec)
    outimg ← idwt2(cA, cH', cV', cD', mother)
    return outimg
```

Note that by using all three finer-detail matrices for embedding, which are (nearly) orthogonal to each other, we **utilize 75%** of the "area" of the image to carry payload information. This enables our algorithm to have a relatively high data capacity.

The process of **extraction** is similar: we begin by transforming the input image using DWT, interleaving the fine-detail matrices, splitting it to n chunks and checking each chunk's correlation coefficient with the two pseudo-random sequences, seq_0 and seq_1. We then choose logical 1 if the correlation with seq_1 is greater than the correlation with seq_0, and logical 0 otherwise.

```
FUNCTION extract(img, numbits, seed, mother):
    cA, cH, cV, cD ← dwt2(img, mother)
    vec ← interleave(cH, cV, cD)
    chunk_size ← ⌊length(vec) / numbits⌋
    seq0, seq1 ← generate_sequences(seed, chunk_size)
    bits ← []
    for i ← 0 to numbits-1:
        chunk ← vec[i*chunk_size : (i+1)*chunk_size]
        corr0 ← correlate(chunk, seq0)
        corr1 ← correlate(chunk, seq1)
        b ← if (corr1 > corr0) then 1 else 0
        append(bits, b)
    return convert_to_bytes(bits)
```

Note that we need to know in advance the (maximal) number of bits in the payload, as it determines the size of each chunk. As with embedding, when the image contains color channels, we try to extract the payload from each channel separately; if all attempts fail, we calculate the mean value of the different channels at each pixel (thus converting the image to a grayscale) and try again.

The `extract` function itself (as given above) cannot tell if the extraction was successful or not; it will always return a sequence of payload bytes "extracted" from the image. In order to tell success from failure, we rely on the Reed-Solomon decoder. The ECC can either correct the errors or fail to correct them; in the latter case, it will fail at a very high probability (much like verifying a checksum). If we choose n message bytes with n ECC bytes, we can **detect** all errors at probability 1, but ever for shorter ECC, the probability of false positives is very low.

Parameters

The watermarking process depends on several parameters, which control the performance of the algorithm to a great extent:

- **Maximal message length** (in bytes) – defines the data capacity of the watermark; values usually range in between 6-30.
- **ECC length** (in bytes) – the number of Reed-Solomon ECC bytes to add; the value ranges between 0 up to the length of the message.

Together these two parameters define the **payload length** (in bytes), i.e., the amount of data that's stored in the host image. Note that the longer the payload the less robust the watermark, as shorter chunks are used. This increases the odds of misinterpreting the correlation during extraction.

- **Seed** – the seed value used to initialize the pseudo-random number generator (PRNG) that generates the sequences. Using a uniformly-distributed PRNG, any seed is expected to work. An important property of the spread-spectrum approach is that by using different seeds, we can layer multiple watermarks one on top of the other, with little chance of cross-interference.
- **Mother wavelet** – the mother wavelet to use for DWT/inverse DWT; this is one of a long list of possible wavelets, such as 'haar', 'db2', 'sym4', 'bior3.3', etc. The full list can be found in the appendices.
- **Gain factor (k)** – a constant used only in the embedding process, which defines the amount of noise (amplitude) that would be added to the host image. The pseudo-random sequence that's added to the image is fist multiplied by this factor. The value chosen for the gain factor depends on the mother wavelet in use, as well as on the noise-level of the host image. Low values (around 2) make the noise to be hardly perceptible at the expense of fragility, while higher ones (5-15) tend to make the image rather noisy, but are very robust to various attacks.

Challenges

I initially hoped to develop a watermarking scheme that would be invariant to rotation and scale, but as explained earlier, the wavelet transform captures spatial location in the result. This makes it highly sensitive to rotation and rescaling, as testing for the correlation of a rotated image with one of our PR sequences would be meaningless. The same goes for scale: resizing the image or adding margins around it, will, again, render the correlation meaningless. This is also due to the fact we split the interleaved vector into chunks, as any change to the vector's length would misalign the chunks and the sequences used.

I've spent almost two weeks trying to make the algorithm invariant to rotation and scale, and have read some material about it [3, 4, 5, 6], but it called for much more complex transforms (uniform log-polar mapping) which I didn't fully understand, or embedding multiple synchronization sequences in different domains (spatial, Fourier and wavelets) in order to counter rotation and scale. Due to time constraints, I had to give up on this, and leave it as an open question. The concluding chapter offers some rudimentary solutions to this problem.

Analysis

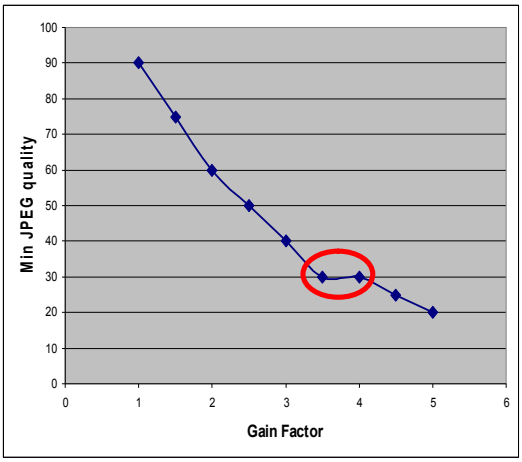
Fidelity

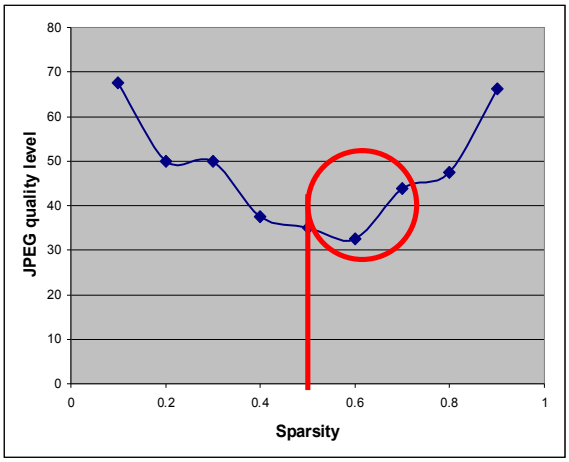
Like all watermarking schemes, we ultimately add noise to the host image, so the first question we need to answer is how well we preserve the image's fidelity. In other words, how much visible noise do we add to the image?

Two parameters control the noise level of the image: the gain factor (k) and the sparsity the of CDMA sequences. Obviously, the larger the gain factor, the more apparent the noise, as k **times** the PR sequence is added to the image. The sparsity, or the distribution of 1's and 0's in sequence itself, is also important: the denser the sequence (more 1's), the more noisy will the image be.

On the other hand, larger k values make the correlation more robust to external noise, as it will require higher levels of noise to mask it out. Our goal is therefore to find the minimal gain factor that still retains high enough correlation, under other all sorts of attacks. The same goes for sparsity: the optimal value (in terms of robustness) is around 0.5, where there are 50% 1's and 50% zeros. However, we want to find the highest sparsity level (meaning, the minimal percentage of 1's) that will satisfy our needs, as this adds the minimal amount of noise.

To measure the robustness of the watermark, we will use the **minimal JPEG quality** (**MJQ**) at which the payload is still extractible – thus, lower MJQ values are better. The following charts show the MJQ of the image of Lena, under the two parameters: the one to the left shows the decrease of MJQ as the gain factor increases; the one to the right shows that the optimal MJQ is achieved with a sparsity of around 0.6. In both cases, the "sweet spots" are marked in red.





The first graph was produced with a constant sparsity level of 0.7, and the second was produced with a constant gain factor of 3. Going below an MJQ of 30 is quite useless – lower JPEG qualities are just unacceptable by human observers. Nonetheless, by choosing a large enough gain factor, the algorithm was shown to reach an MJQ of 5 – although image quality at this level. From the charts above, we can expect the gain factor range between 2 and 4, depending on the amount of noise in the host image. If not stated otherwise, a gain factor of 4 was used throughout the analysis.

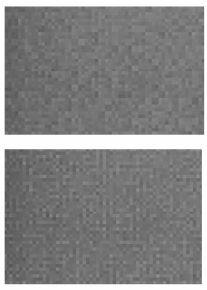
The sparsity graph exhibits symmetry, and reasonable values (in terms of MJQ) range between 0.3 and 0.7. However, going below 0.5 doesn't make sense, as it means we're adding more noise while not gaining additional robustness – recall that 1's in the CDMA sequence translate to noise, while 0's do not affect the image. Therefore, the sparsity should range between 0.5 and 0.7, and throughout the analysis it was fixed at 0.7.

Payload Length

A nice property of the algorithm is that the payload length does not affect the noise level of the image. This is because the interleaved vector is first split into chunks, and a single bit of payload is embedded in each. With longer payloads, there will be more chunks (each of a smaller size), which decreases the watermark's robustness, but has no affect on the overall noise.

Choice of Wavelet

Another interesting property that affects the visual quality of the image is the choice of an **orthogonal vs. biorthogonal** wavelet family. For instance, Haar and Daubechies are orthogonal, resulting in what seems to be strictly vertical or horizontal noise. Biothrogonal wavelets, on the other hand, tend to produce **smoother-looking noise** that doesn't seem to have a defined direction. This property of "direction of noise" is exemplified in the images to the right (cropped from the background of Lena).



Another advantage of biorthogonal wavelets is they require a lower gain factor than orthogonal ones, in order to reach the same MJQ. The following table lists the best 9 wavelet mothers in terms of MJQ (on the image of Lena), under a constant gain factor: the biorthogonal family clearly wins, with orthogonal ones largely clustered around an MJQ of 70.

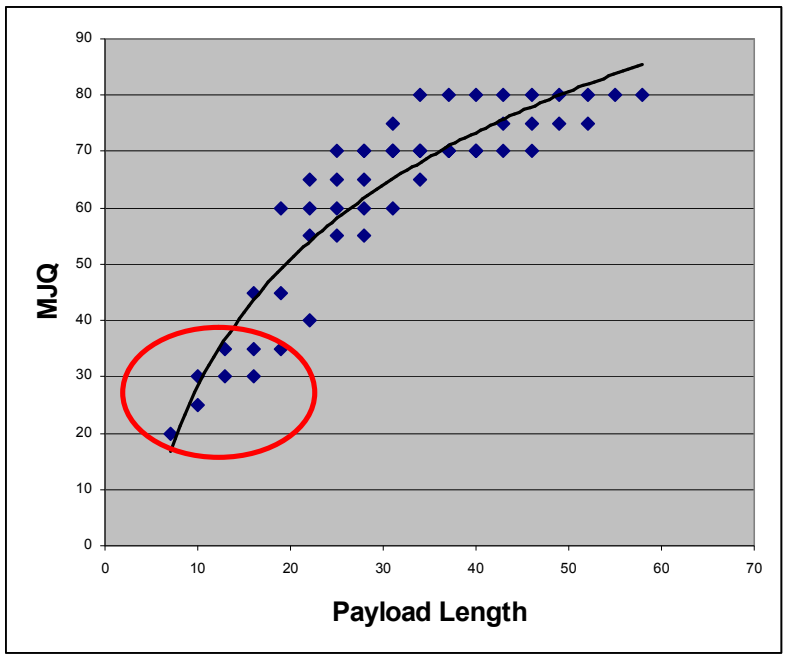
Wavelet	MJQ	Wavelet	MJQ	Wavelet	MJQ
bior3.3	25	bior3.7	35	db19	45
bior3.1	30	bior3.9	35	db6	45
bior3.5	35	bior2.2	45	bior2.4	50

Therefore the wavelet of choice is either **bior3.1** or **bior3.3**; in the analysis, I used bior3.1.

Capacity and ECC

As explained previously, the longer the payload, the more sensitive it is to noise. This happens as each chunk gets smaller to accommodate the longer payload, so a shorter sequence is used for correlation with the chunk. This increases the chances of false-positives during correlation, where logical 0's and 1's will be confused.

The chart below shows the relation between MJQ and payload length (message and ECC combined) in the Lena image:



This chart is a bit misleading, as 20 message bytes and 20 ECC bytes would allow lower MJQ than 39 message bytes and 1 ECC byte – the data is three-dimensional, compressed to two by summation. However, the trend-line is apparent either way.

Since we aim at embedding URLs, 6 bytes of data suffice for **URL shorteners** (like http://goo.gl or http://is.gd, for example, http://is.gd/TWMBG2). In this case, we can settle for one URL shortener site so we only need to store the suffix of the URL, i.e., the last 6 bytes (actually, they use base64-like encoding, so we only need $6 \cdot 6 = 36$ bits instead of 48, but 6 full bytes is a reasonable demand).

A reasonable ECC length for this would be 4 bytes, which allows us to correct between 2 and 4 byte-wise errors. For the rest of the analysis, we use **6 message bytes** and **4 ECC bytes**, yielding 10 bytes of payload.

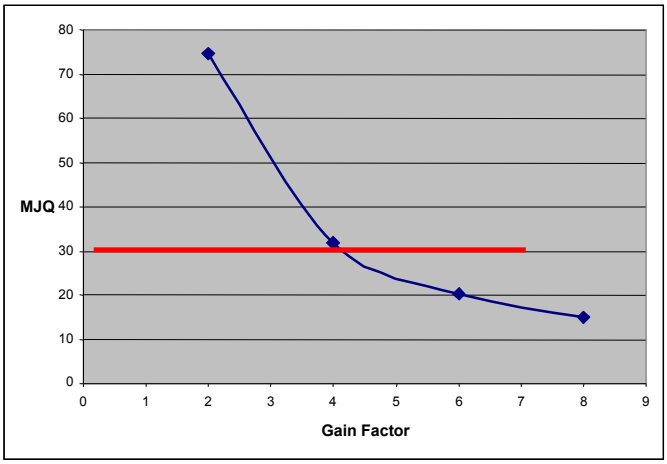
It should be noted, of course, that larger images can host longer payloads, as the chunk size would increase with the image's dimensions.

Attacks

The following analysis examines the durability of the watermark to various filters, compression schemes and added noise, called *attacks on the watermark*. As already explained, any geometrical deformation will render the watermark useless, so all the following attacks preserve the image's geometry. If not stated otherwise, the analysis shows the average values over 12 sample images, 3 of which are monochrome and rest are in color.

JPEG

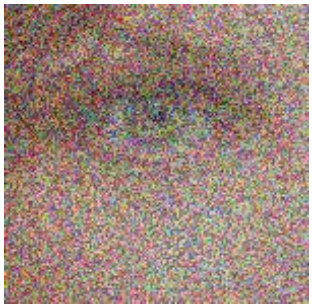
Minimal JPEG Quality (MJQ) was used throughout the analysis as a way to measure the robustness of the watermark under various parameters. Going below a quality level of 30 or 20 usually renders the images unacceptable, so there's no interest in going below that level. However, the watermark is very robust to JPEG compression, and the MJQ can be perfectly tuned by choosing the appropriate gain factor. The chart below shows the MJQ and gain factor relationship, averaged over 12 sample images:

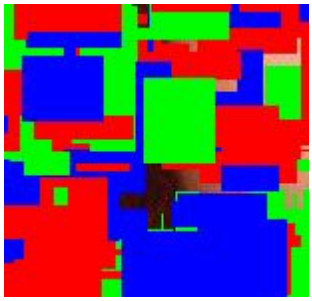


As can be seen clearly, a gain factor of 4 should suffice for most practical applications, which also keeps the noise level low.

Noise

The algorithm was tested under two kinds of noise: **Gaussian noise** added at the pixel level (to each color channel separately), or **block-coverage** noise, where a patches of color are randomly placed on the image. Here are two examples of Gaussian noise (top) and block coverage noise (bottom). The average noise levels over 12 sample images were:





Gain Level	Gaussian noise level	Block coverage ratio
4	32%	63%
6	65%	82%

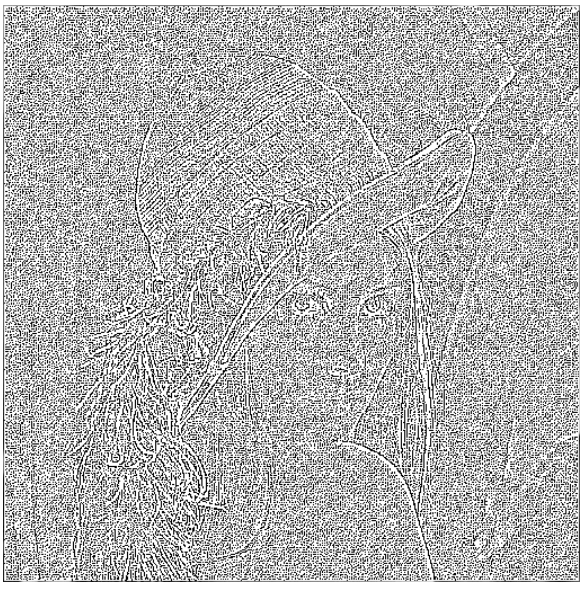
The more interesting noise in our print-scan world is block-coverage, where part of the image is covered or damaged. It should be noted that some images have reached over 95% block coverage, and the payload was still extractible. The table above shows the average amount of noise added, but the variance is rather wide (±20%).

Sharpening

The algorithm embeds the watermark in the finer-detail (higher frequency) parts of the image, so we can expect any sort of sharpening or edge enhancement to preserve the payload.

Indeed, in the test suite, all images were resilient to such high-pass filters, even when the resulting image no longer resembled the original one. The tests were conducted by recursively applying the filter to the result of the previous application, 30 times. The image to the right shows Lena after 30 recursive applications of the contour filter – the payload is still extractible.

Filter	Max Iterations
contour	unbounded
detail	unbounded
edge-enhance	unbounded
find-edges	unbounded
sharpen	unbounded
emboss	4.25



Blurring Filters

For the same reason the algorithm is resilient to sharpening, it is sensitive to low-pass/blurring filters, as they remove the fine details. The average maximal number of iterations of the *smoothen filter* was 4.6, and the average maximal value of sigma in the Gaussian filter was 1.7.

As a side note, the average maximal sigma value for Laplacian of Gaussian (LoG) was 2.2. This is expected to the higher than the Gaussian alone, as LoG first blurs and then "sharpens", so more higher-frequencies are present in the final image.

Total-Variation (TV) Denoising

One of the great surprises of the algorithm was its resilience to TV-denoising. TV denoising is a computationally intensive filter that removes noise in the image's flat areas, while preserving finer details, like edges. It works by reducing the total variation in the different regions of the image.

TV denoising takes a weight parameter which controls the extent of noise removal. With a gain factor of 4, the average weight ranges between 150 and 250; while with a gain factor of 6, the range is 200-400.

It should be noted that at these levels of denoising, the image looses much of its fidelity and becomes blurry. But the fact that the watermarking endures denoising to such extents means we can always run TV denoising with a low weight (for instance, 5) on the output of the algorithm, making the image more visually appealing while still retaining fidelity and robustness.

Lomo Filters

The last kind of tests I put the algorithm under was Instagram (or Lomo) filters. Instagram uses various filters to give the image an "old-style" feeling (as if it were taken by an old film camera), as well as improving contrast and color balance, to make the image look better.

I used http://lomo.helloburin.com to apply various Lomo filters onto the results of the algorithm, and tested the number of recursive application that it could sustain. The results were rather overwhelming: black-and-white images could sustain an average of 10 applications, even when the image lost all resemblance to the original one. Color images could sustain around 3 applications. I think this has to do with the Lomo filters themselves and not with the watermark. To the right is the image of Lena after 6 different Lomo filters.

Further Analysis

Different Interleaving Methods

During embedding and extraction, the three finer-detail matrices are interleaved to form a long vector. The interleaving process used simply takes the first value from each and flattens it, then the second value, and so on. For instance, the following three matrices

1	2
3	4

10	20
30	40

100	200
300	400

will be interleaved as

1	10	100	2	20	200	3	30	300	4	40	400

When this vector is partitioned into chunks, each chunk corresponds to a physical location in the image – because DWT preserves location. For example, partitioning the vector into two chunks would result in one covering chunk the top row and the other covering the bottom row.

I though this makes the watermark vulnerable to "burst noise", as in the case of block-coverage, but any attempts to use different interleaving schemes (like performing a random permutation of the vector) have proven to have worst MJQ, so I settled for the linear interleaving shown above. I cannot explain why this sort of interleaving performs best, but it does.

Choice of CDMA Sequences

When generating the CDMA sequences, I first generated two random sequences of 0's and 1's of the required size. In order to increase the probability of successful detection, I changed the generation process to choose only uncorrelated sequences (i.e., $|\text{corr}(seq_0, seq_1)| < \varepsilon$) via trial and error. This indeed helped me achieve lower MJQ values, but I ultimately settled for a much simpler solution, that exploits the properties of Pearson correlation: the correlation of any sequence with its reverse is -1. Therefore, I now simply generate the sequence for seq_0, and then reverse it for seq_1. This scheme utilizes the whole range of the correlation coefficient ([-1, 1]), so the odds of misinterpreting the correlation are even lower.

Instead of using 0's and 1's in the correlation sequence, I tried using (-1)'s and 1's, but the results of this were not superior to 0's and 1's, while in fact they only increased the noise level in the resulting image, as 0's do not alter the image.

Different Color Spaces

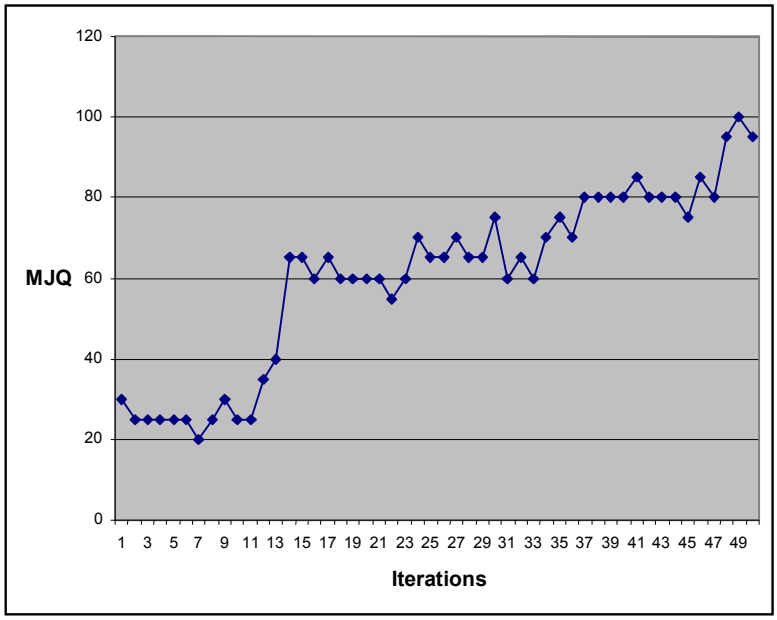
The algorithm embeds the payload into each color channel (RGB) separately. I hoped to achieve better results in different color spaces, e.g., HSL or YC_BC_R, but the results were not promising. The only reliable channel was the intensity (L in HSL or Y in YC_BC_R), which is basically a **weighted average** of the RGB channels. Therefore, embedding the watermark in each channel has the same effect; moving to different color spaces did not prove helpful.

Sensitivity to Color

The watermark is quite insensitive to color/intensity changes, like gamma-correction, brightness/contrast or hue/saturation changes, etc. I did notice it was sensitive to color-inversion; however, it turned out that color inversion merely swaps the bits – 0 becomes 1 and vice versa, deterministically. This probably has to do with the DWT and the properties of correlation. Since this process is deterministic, in case decoding the payload fails (using a Reed-Solomon decoder), we simply invert the bits and try again.

Multiple watermarks

As mentioned earlier, it is possible to embed several watermarks in the same host image, by using separate seeds for the PRNG. Every iteration damages previous ones, of course, as it overlays new noise on the image – but the interference is rather minimal, as can be seen in the chart below:



In this test, I embedded different messages 50 times into the image of Lena, one on top of the other, and measured the MJQ of the **first embedding** each time – the one that suffered the greatest loss in quality. It is quite astonishing that 10 consecutive iterations still keep the MJQ level around 30, and the next 25 iterations keep it around 65.

The more astonishing fact is, the image quality almost did not deteriorate in the process – the differences were marginal. This test demonstrates the power of spread-spectrum embedding techniques: we can increase the data capacity a tenfold by using different seeds, at almost no incurred cost!

By using a CDMA sequence of -1/+1, more recent noise cancels out previously added noise (in the expected case), which results in the same overall image quality: successive embedding would maintain a constant level of noise.

Print-Scan Results

I have successfully printed and scanned two pictures, one black-and-white (Lena) and the other in color. It took some effort to manually align and rescale the scanned images, but once I've realigned them, the results were perfect: the payload was successfully extracted from both images.

In fact, the Lena image scan had an MJQ of 55; the face scan did not do so well and required an MJQ of 100. I think this has to do with the quality of my printer and scanner, as the colors seems a quite distorted compared to the original image.

Complexity

The algorithm performs quite well, and I believe it would be adapted to mobile phones. The project has been written in Python, an interpreted, dynamic language that doesn't place performance as its top goal. Writing the code in C would surely run faster, and I believe a compiled version could run well mobile phones.

The embedding process is generally cheaper than extraction, as embedding happens once for black-and-white images and three times for color images, but extraction is attempted for up to 4 times: for every color channel separately, then for their average (grayscale).

This is a bit of a problem, as embedding would normally happen on stronger machines while extraction should be able to run on low-end devices.

The DWT transforms seem very cheap from a computational perspective, at least for the images I inspected (512x512 to 900x600) on my computer (which is 4 years old). The factor that mostly controls the time taken by the algorithm is the size of the image, with smaller, black-and-white images taking under a second, and larger, color images taking around two-four seconds.

Conclusions

In this project, I set off trying to embed QR codes into host images, in a manner that would be print-scan friendly. This quickly proved problematic, and the focus of the project has shifted to the implementation of a very resilient, invisible watermarking algorithm, which is applicable in the print-scan world.

The watermark offers quite a large data capacity on its own, relying on Reed Solomon ECC to correct errors and detect corrupted messages. This payload can be made even longer by using multiple seeds and layering subsets of the payload one on top of the other, exploiting the properties of spread-spectrum encoding. It can easily increase the payload length by tenfold.

As we've seen in the analysis, the resulting watermark is highly resilient to JPEG compression, can sustain large amounts of noise (both at pixel resolution and block resolution), all sharpening filters, as well as various blurring/denoising filters. It even survives the application of Lomo filters, which have become very popular nowadays, so we may be able to extract the payload from a picture taken via applications like Instagram.

Because the watermark is so robust, it can be used in Digital Rights Management, to embed a copyright notice that can be later extracted to track the source of the image. Because of the spread-spectrum nature of the watermark, it is nearly impossible to remove it while still retaining acceptable image quality.

On the other hand, the watermark is very sensitive to geometric transformations, like scaling and rotation, due to the nature of the discrete wavelet transform. However, it is believed that some preprocessing can be used to counter and realign geometrically transformed images, but this calls for further investigation.

For example, we can employ a search-space strategy that attempts to lock-in on the right rotation angle of the signal: suppose we embed 10 different synchronization patterns at different rotation angles (every 36 degrees), using multiple seeds. When we scan an image, we will successively rotate it and look for any of the patterns. If a match is found, we'd know how to counter the rotation of the image. Since a sync-pattern is embedded every 36 degrees (and assuming uniform distribution of image rotations) the expected number angles to try is 18 (expectancy of uniform distribution).

Taking into account that people hold cameras mostly horizontally (in the range of ±5 degrees), we can embed denser sync patterns at lower angles and sparser patterns in higher ones. But this, of course, is just an idea, and it goes beyond the scope of this work.

References

The following two sites were my entry point to the world of invisible watermarking and Reed Solomon codes.

1. Chris Shoemaker. ***Hidden Bits: A Survey of Techniques for Digital Watermarking***. 2002. http://web.vu.union.edu/~shoemakc/watermarking/watermarking.html
2. Wikiversity. ***Reed Solomon codes for coders.*** http://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders

The following papers explore watermarking methods that are resilient to printing and scanning, using various methods. They tend to be very mathematical with little practical side, so they were of little value to me. However, I borrowed ideas from them, which I used in my attempts to write a

3. Dajun & Qibin. ***A Practical Print-Scan Resilient Watermarking Scheme***. Institute of Infocomm Research, Singapore. 2005.
4. Kang, Huang & Zeng. ***Efficient General Print-Scanning Resilient Data Hiding Based on Uniform Log-Polar Mapping***. IEEE Transactions on Information Forensics and Security, Vol 5 No. 1. 2010.
5. Pramila, Keskinarkaus & Seppanen. ***Multiple Domain Watermarking for Print-Scan and JPEG Resilient Data Hiding***. Media Team, University of Oulu, Finland. 2007.
6. Lin, Wu, Bloom, Cox Miller & Lui. ***Rotation, Scale and Translation Resilient Watermarking for Images***. IEEE Transactions of Image Processing, Vol 10 No. 5. 2001.

I collected the papers around the internet, and they are provided in the Dropbox folder. For a nice review of "QR codes gone bad", see

http://www.holesinthenet.co.il/archives/40063

Appendix

Installation on Windows

The project is written in Python, and requires several packages in order to run. First, you'll need Python 2.7 (either x86 or x64, depending on your computer), and the following packages (install in the order they appear below:

1. PIL
2. Numpy
3. Scipy
4. Scikits-image
5. PyWavelet
6. ReedSolo

These packages can be obtained online (including Python), but they are also provided in the Dropbox folder, under `install/`. If something is missing or doesn't work, please contact me.

Source Code

The Python source code is provided in the Dropbox folder, under `code/`. It comprises of `watermarker.py` – the watermarking library, and two command-line tools: `embed.py` and `extract.py`. You can run them like so:

```
C:\DropboxFoler\imgproj\code> python embed.py imagefile.jpg 123456
Created imagefile-123456.png
C:\DropboxFoler\imgproj\code> python extract.py imagefile-123456.png
123456
```

Use `--help` to see all of the command line options (like -k for gain factor, -s for seed, -t for TV denoising weight, -m for wavelet mother, etc.). The file `example.txt` includes a sample execution of these tools from the command line (on Linux)

List of Mother Wavelets

- **Biorthogonal family**: bior1.1, bior1.3, bior1.5, bior2.2, bior2.4, bior2.6, bior2.8, bior3.1, bior3.3, bior3.5, bior3.7, bior3.9, bior4.4, bior5.5, bior6.8
- **Coiflets family**: coif1, coif2, coif3, coif4, coif5
- **Daubechies family**: db1, db2, db3, db4, db5, db6, db7, db8, db9, db10, db11, db12, db13, db14, db15, db16, db17, db18, db19, db20
- **Meyer**: dmey
- **Haar**: haar
- **Reverse biorthogonal family**: rbio1.1, rbio1.3, rbio1.5, rbio2.2, rbio2.4, rbio2.6, rbio2.8, rbio3.1, rbio3.3, rbio3.5, rbio3.7, rbio3.9, rbio4.4, rbio5.5, rbio6.8

- **Symlets**: sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12, sym13, sym14, sym15, sym16, sym17, sym18, sym19, sym20

A very useful site is the **Wavelet Browser**, which includes information on each wavelet. For example: http://wavelets.pybytes.com/wavelet/bior3.1/