Improved Steganographic Embedding Using Feature Restoration

Undergraduate dissertation of Ventsislav K. Chonev

Supervised by Andrew D. Ker

Chapter 1

Introduction

This project is concerned with feature restoration, a problem within the fields of steganography and steganalysis. This chapter introduces both fields briefly, then proceeds to define the problem and outline the structure of the work we have done to solve it.

1.1 Steganography

Cryptography is a science concerned with hiding information. In cryptography, one devises schemes to encrypt messages in such a way that no one apart from the intended recipient may decrypt them. However, a general weakness of cryptography is that, while its methods preserve the secrecy of the exchanged information, they do little to hide the *presence* of communication. Even if an unbreakable cryptographic cipher is used, a passive intruder would still see an unreadable message and become aware of secret communication between the two parties, despite being unable to read it.

The related field of steganography goes further by attempting to conceal the presence of communication altogether. This is done by hiding the message inside a cover - for example, Johannes Trithemius's work *Steganographia* (published in 1606) is ostensibly about magic and the occult but is in fact only a covertext for a treatise of cryptography and steganography.

In the context of modern Computer Science, steganography is often expressed as the *prisoners' problem.* Alice and Bob are in prison, living in separate cells. They each have access to a computer, and may use a communication channel existing between the two computers. This channel is the only way they can exchange information. Alice and Bob each have access to a variety of file formats - text, pictures, video, sound, etc. - and may exchange such files over the communication channel. They are also assumed to share a secret key which they agreed on before being imprisoned. Finally, each of them has access to a random number generator (RNG). The warden Wendy monitors all communication along the channel, and if she becomes suspicious, she will close it for good. It is Alice and Bob's goal to devise an escape plan without arousing Wendy's suspicion. A practical way to do so is to embed messages into cover files using the shared key for encryption and decryption. Steganography is the field concerned with devising such embedding schemes.

1.2 Least Significant Bit Matching

In this project, we will focus exclusively on one steganographic scheme for embedding messages into greyscale images[1], called Least Significant Bit (LSB) matching.

Encoding a message into a cover image is done as follows:

1. Seed the RNG with the steganographic key.
2. Use the RNG to generate a permutation L of the pixels of the cover image.
3. Convert the plaintext into a bitstream B in some standard way (e.g., by concatenating the binary representations of the ASCII codes of the characters). Assume $|B| \leqq |L|$.
4. Simultaneously traverse B and L, embedding one bit of B into each pixel in L. To embed a bit b into a pixel p, examine $LSB(p)$ - the parity of p. If $LSB(p) = b$, leave the pixel as it is. Otherwise: if $p = 0$, increment p by 1, if $p = 255$, decrement p by 1, and if $0 < p < 255$, randomly choose to increment or decrement p by 1 with equal probability.

The hidden information B is called the *payload,* and the ratio $\frac{|B|}{|L|}$ is called the *payload percentage.* For example, a payload of 7373 bytes = 58984 bits embedded into a 256×256 image is a 90%-payload. The pixels which hold the bit string B in their LSBs are called *payload pixels*.

Decoding a message from a stego-image is done similarly:

1. Seed the RNG with the steganographic key.
2. Use the RNG to generate a permutation L of the pixels of the stego-image. If the same key was used for the encoding, this permutation will be the same as the one used to create the stego-image.
3. Traverse L. At each pixel p in L, calculate $b = LSB(p)$ and append b at the end of a bit-string B.
4. Convert B into plaintext. If the message length is known, take the prefix of B of that length.

1.3 Steganalysis

Steganalysis is the complement field of steganography; it is concerned with detecting the presence of a hidden payload in a cover file. In the context of the prisoners' problem, steganalysis is Wendy's job: she wants to detect any covert communication between Alice and Bob, without accusing them

[1]A grayscale image is just a matrix of 8-bit values. Throughout this project, we used the pgm file format, which is precisely such a matrix, preceded by a short header.

falsely. A common assumption in steganalysis, like in cryptanalysis, is Kerckhoffs's principle: Wendy knows the exact steganographic algorithm used by Alice and Bob, but not their shared key.

Most of modern steganalysis is based on *statistical features*. The idea of feature-based steganalysis is introduced in [4]; since then various feature sets and techniques have been proposed ([5]-[10]). In all cases, the central idea is that covers have a high degree of coherence, which makes their content predictable, unlike the embedded stego signal, which is essentially additive noise. Therefore, feature-based steganalysis performs the opposite to denoising: it strips away the content of the cover to leave only the noise, and then measures chosen statistical characteristics (features) of it. A good feature set should be sensitive to embedding noise, but insensitive to the image content.

Once a feature set is chosen and an extractor is built for it, machine learning techniques are used to train a classifier to distinguish between the features of stego-covers and innocent covers. Classifiers considered in literature include the Fisher Linear Discriminator ([2]), Support Vector Machines and Neural Networks ([3]).

1.4 Feature Restoration

Feature restoration is a technique for reducing the suspiciousness of a stego-image, thereby decreasing the likelihood of detection. When a payload is embedded into a cover image, its features are likely to become highly abnormal and indicative of hidden information. Feature restoration attempts to selectively modify the non-payload pixels of the stego-image in order to bring the features back to normal, and avoid detection by a steganalyzer. This technique is only sensible for high payloads (i.e., above 50%), because efficient techniques for concealing low payloads are known and well-studied. The concept of feature restoration was suggested in [2], but no work has been done to investigate algorithms or to benchmark them against real steganalyzers.

1.5 This Project

This project attempts to fill the void in the literature by devising some algorithms for feature restoration and comparing their performance. Throughout, we will focus solely on the steganographic scheme LSB-matching applied to grayscale images because it is both one of the simplest and least detectable steganographic schemes known. Chapter 2 presents the best known features for detecting LSB-matching, called WAM. Chapter 3 investigates an important property of WAM. Chapter 4 defines a distance metric to measure the suspiciousness of feature vectors. Chapter 5 investigates algorithms for feature restoration.

It should be pointed out that feature restoration is a very difficult problem to solve optimally. Formalising it produces a Binary Integer Quadratic Problem, which is known to be an NP-hard optimisation problem - see Chapter 5 for details. Therefore, our investigation of algorithms for feature restoration will focus on heuristics and approximations rather than ambitiously attempting to solve the problem optimally.

Nearly all of the programming for this project was done in C++. We implemented LSB-matching (see Appendix A for the well-commented code), a feature extractor for WAM (see Appendix B after reading Chapter 2), and the feature restoration algorithms that we devised (see Appendix C after reading Chapter 5). We also wrote much miscellaneous code, such as a reader for *pgm* images, to facilitate the work of these main modules. Finally, everything was glued together using bash scripts to run experiments and Matlab to convert their results into charts.

Chapter 2

Wavelet Absolute Moments

The focus of this project is a set of image features, called Wavelet Absolute Moments (WAM). Their calculation is a multi-stage process involving a wavelet transform. We begin by providing some minimalistic background in wavelets and wavelet transforms. Then we proceed to define the WAM features, and to describe the building of a feature extractor, pointing out some important efficiency considerations.

2.1 Motivation for Wavelet Theory

Wavelet Theory is a field which evolved from Fourier Theory. At the heart of Fourier Theory is the Fourier Transform, which is used to represent a given function in a convenient form. The function $f(x)$ is expressed as a (possibly infinite) sum of sine and cosine waves of varying frequencies and amplitudes. The Fourier series of a periodic $f(x)$ with period L is given by:

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n cos\left(\frac{\pi n x}{L}\right) + \sum_{n=1}^{\infty} b_n sin\left(\frac{\pi n x}{L}\right)$$

$$a_n = \frac{1}{L}\int_{-L}^{L} f(x)\, cos\left(\frac{\pi n x}{L}\right) dx$$

$$b_n = \frac{1}{L}\int_{-L}^{L} f(x)\, sin\left(\frac{\pi n x}{L}\right) dx$$

The Fourier Transform of any $f(x)$ is obtained by extending the series to complex coefficients, replacing the discrete summation with integration, and letting $L \longrightarrow \infty$. The transform $F(\nu)$ gives the amplitudes of the sine and cosine waves of frequency ν appearing in the summation of f. Thus, there are two ways to look at a function: its *spatial domain* representation $f(x)$, which allows one to evaluate the function at a specific point, and its *frequency domain* representation $F(\nu)$, which allows one to examine its frequency content.

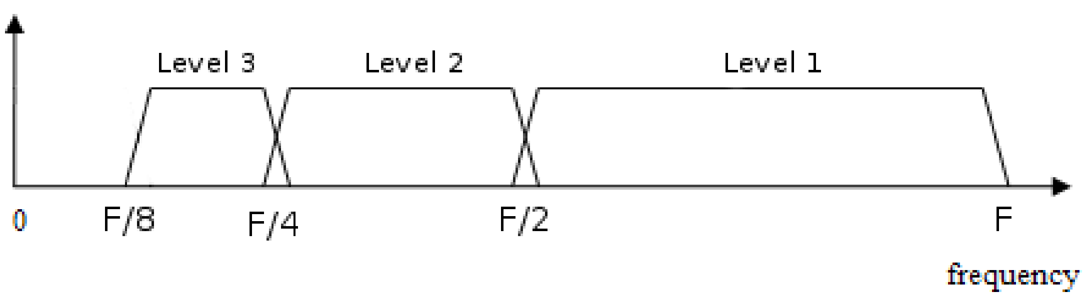
A drawback of Fourier analysis is that sine and cosine waves are global. Hence, it is difficult to examine an interesting region of a function in isolation because all the waves contribute to it. Instead, one might like to zoom in on the interesting section by knowing which summands are relevant to it. This has prompted the emergence of Wavelet Theory.

2.2 The Discrete Wavelet Transform

Wavelet Theory provides a transformation similar to the Fourier Transform, but using a basis of functions localised in space - wavelets. This transformation will allow us to speak of a *wavelet domain*, just like the Fourier Transform produced a frequency domain. The basis consists of dilated, compressed and translated versions of a *mother wavelet*, which is just a specially chosen function satisfying certain requirements that formalise the notion of being a localised wave. In this context, translation and scaling are only allowed in the x direction.

Allowing unconstrained translation and scaling of the mother wavelet would produce an uncountably infinite basis, with a high degree of redundancy. In order to make the wavelet basis countable, we allow only translations by a fixed step $a = 1$ and dilation and compression by a fixed factor $b = 2$. As we are only interested in wavelets that have some overlap with the analysed signal, we can obtain an upper and a lower bound on the translation parameter.

To bound the scaling parameter, consider the Fourier spectrum of the daughter wavelets and the signal. The qualifying requirements for a mother wavelet imply that its spectrum is a band. Fourier Theory shows that compressing such a function by a factor of 2 doubles each frequency present in the spectrum, effectively doubling the spectrum's width and shifting it up. Dilating the mother wavelet has the opposite effect. If we choose the mother wavelet carefully, the spectra of wavelets at consecutive scales will touch each other, or overlap slightly:



Typically, we are interested in bandlimited signals $f(x)$. Such signals have a maximum frequency, so compressing the mother wavelet multiple times will eventually produce wavelets whose frequency spectrum does not intersect the spectrum of $f(x)$, providing a lower bound on the scaling parameter s. However, an upper bound does not exist. Adding an extra scale level to the wavelet basis halves the width of the spectrum left to cover down to 0, so a finite basis of daughter wavelets cannot be enough to represent $f(x)$. The solution is to add an extra function (*father wavelet*) to the basis. Its spectrum is low-pass, so it is a placeholder in the basis for an infinite number of daughter wavelets. Now the basis is finite.

The Discrete Wavelet Transform (DWT) is a mechanism to obtain the wavelet domain representation of a discrete signal for a given wavelet basis. The algorithm is inductive: if the signal is expressed as a weighted sum of daughter wavelets up to a certain scale s and the father wavelet for that scale, an inductive step may be made to include the next scale $s+1$ into the basis. This incorporates more dilated wavelets into the basis and reveals finer detail about the signal. The formal details may be found in [12].

In practice, however, the DWT is computed using signal filtering techniques. A Quadrature Mirror Filter (QMF) is designed, specific to the chosen wavelet basis. A QMF is a pair of filters (g,h), such that g is low-pass and admits only frequencies up to some threshold, whereas h is high-pass and admits only frequencies above that threshold. At each stage, the signal is fed into both filters using a discrete convolution. The operation produces low-frequency output and high-frequency output:

$x = inputSignal$

$filterLength = length[g] = length[h]$

$yLow[i] = \sum_{j=0}^{filterLength} g[filterLength - j - 1] * x[i+j]$

$yHigh[i] = \sum_{j=0}^{filterLength} h[filterLength - j - 1] * x[i+j]$

The high-pass output $yHigh$ yields the wavelet coefficients for that scale. The low-pass output *yLow* corresponds to the father wavelet coefficients, and is a coarse approximation of the signal. Then *yLow* is *downsampled* - its every other entry is discarded, halving its length. The downsampled *yLow* is given as input to the next stage. This is Mallat's algorithm for computing the DWT.

The algorithm generalises to 2D. Given a 2D signal x and a QMF (g,h), we can filter each row of x using one of (g,h), and then each column of the result using one of (g,h). Thus, 2D filtering may be performed in four ways:

- Using g both for row filtering and column filtering gives the output matrix LL.
- Using g for rows and h for columns gives LH.
- Using h for rows and g for columns gives HL.
- Using h for rows and columns gives HH.

All four output matrices are downsampled by 2, discarding every other row and column. LL is a coarse approximation of the input signal. After downsampling, it is used as input to the next level. LH is typically referred to as the horizontal output band **H**, HL - as the vertical band **V**, and HH - as the diagonal band **D**.

2.3 Definition of WAM Features

Now that we have some knowledge of what a wavelet domain is, we are in a position to define WAM. Given a 2D signal S, its WAM features are defined as follows (following the exposition in [2, 3]):

1. Compute a one-level 2D DWT of S using the Daubechies QMF of length 8, *without* down-sampling. Discard the low-frequency output LL. Denote the three output bands H, V, and D, respectively.

2. Let σ_0^2 denote the noise variance of the signal; for LSB-matching we have $\sigma_0^2 = 0.5$. Estimate the local variance of each element in H, V, and D, using windows of sizes 3, 5, 7 and 9:

$$\sigma_H^2(x,y) = max\left(0, min\left(h_3, h_5, h_7, h_9\right) - \sigma_0^2\right) = max\left(0, min\left(h_3, h_5, h_7, h_9\right) - 0.5\right)$$

   where $h_i = \frac{1}{|I|}\sum_{(a,b)\in I}\left(H(a,b)\right)^2$ is the average squared wavelet coefficient in the $i \times i$ neighbourhood of (x,y) in H. Similarly for $\sigma_V^2(x,y)$ and $\sigma_D^2(x,y)$.

3. Apply a Wiener filter to each of H, V, and D to obtain the denoised wavelet coefficients, then subtract them from the coefficients to leave only the noise residuals:

$$R_H(x,y) = H(x,y) - H_{den}(x,y) = H(x,y) - \frac{\sigma_H^2(x,y)}{\sigma_0^2 + \sigma_H^2(x,y)} H(x,y) = \frac{0.5}{0.5 + \sigma_H^2(x,y)} H(x,y)$$

   and similarly for $R_V(x,y)$ and $R_D(x,y)$.

4. Compute the average noise residual in each band: $\overline{R_H}$, $\overline{R_V}$ and $\overline{R_D}$.

5. Finally, compute the first nine central absolute moments of the residuals in each band:

$$M_H^k = \frac{1}{|I|}\sum_{(x,y)\in I} |R_H(x,y) - \overline{R_H}|^k$$

   where I is the index set of H, and k takes values 1...9. Similarly for M_V^k and M_D^k.

6. The WAM features of S are the 27-dimensional vector

$$\left[M_H^1, \ldots, M_H^9, M_V^1, \ldots, M_V^9, M_D^1, \ldots, M_D^9\right]^T$$

It is worth clarifying the steps in the pipeline. Step 1 transforms the image into the wavelet domain. This is done because it has been shown that features taken from the wavelet domain are more sensitive than ones taken from the spatial domain. Steps 2 and 3 strip away the image content, in order to leave only the noise: the wavelet coefficients are denoised using a quasi-Wiener filter[1], then the denoised coefficients are subtracted from the coefficients themselves, leaving only the noise content. The Wiener filter requires an estimation of local variance, which necessitates step 2. Finally, absolute central moments of the noise residuals are known to provide a good characterisation of the noise of the image ([3]), so they are used as features.

[1] The formula given in step 3 is exactly that of the Wiener filter, except we are applying it in the wavelet domain, rather than the frequency domain, hence the 'quasi'.

2.4 WAM Feature Extractor

For the purposes of this project, we built a feature extractor for WAM from the ground up. This included implementing directly the 2D DWT, the estimation of local variance, and the calculation of noise residuals and moments. This tool will be at the heart of our feature restoration algorithms. With it, we can gauge how the features change when some pixels are perturbed in a particular way. Using such queries, we will selectively modify specific non-payload pixels of the stego image in order to reduce the distance[2] to a target innocent feature vector.

Because we will be relying heavily on the WAM oracle in feature restoration, it is crucial to ensure its efficiency. A very significant part of the effort spent on this project was aimed at building the feature extractor and optimising it. This section gives a brief overview of the challenges involved in creating this tool and how we overcame them. The full code with detailed comments appears in the appendix.

Firstly, we implemented the 2D DWT; this is straightforward from the definition given above. The variance estimation performed in step 2 of the WAM algorithm is more challenging. For each output band $w \in \{LH, HL, HH\}$, for each element $w(x,y)$, we are required to compute the averages of the squared 3×3, 5×5, 7×7 and 9×9 regions around (x,y) in w. Directly traversing each of these regions for every (x,y) is wasteful, and would imply visiting each position 81 times. Instead, we used the transformation: $w \to v$, where $v(x,y) = \sum_{i\le x, j\le y} (w(i,j))^2$. This transformation may be computed much more efficiently from w:

```
PrefixedSquaresForm(matrix w of size N by M)
{
for i in [0..N) do
  for j in [0..M) do
    v(i,j) = square(w(i,j));

for i in [0..N) do
  for j in [1..M) do
    v(i,j) = v(i,j-1) + v(i,j);

for j in [0..M) do
  for i in [1..N) do
    v(i,j) = v(i-1,j) + v(i,j);
return v;
}
```

The benefit of this form is that it allows us to compute the sum of the squared values in any rectangular region of w using at most 3 additions, making the efficient calculation of local variance trivial:

$$sumOfSquaresInRectangle(0,0,X_1,Y_1) = v(X_1,Y_1)$$

$$sumOfSquaresInRectangle(0,Y_0,X_1,Y_1) = v(X_1,Y_1) - v(X_1,Y_0-1)$$

$$sumOfSquaresInRectangle(X_0,0,X_1,Y_1) = v(X_1,Y_1) - v(X_0-1,Y_1)$$

[2] For a notion of distance to be defined later.

$$sumOfSquaresInRectangle\left(X_0,Y_0,X_1,Y_1\right)=v\left(X_1,Y_1\right)-v\left(X_0-1,Y_1\right)-v\left(X_1,Y_0-1\right)+v\left(X_0-1,Y_0-1\right)$$

Calculating the residuals and the moments is straightforward from the definition. We should point out that the moments are the most expensive part of the pipeline.

At this point, we have an oracle capable of computing the WAM features of any given image. For a WAM calculation on a **fresh** image, this pipeline is very efficient. However, most of the time we will be interested in the effect on the features of perturbing some pixels of an image whose features we have already calculated. The best we can do with the oracle outlined above is to perform the perturbation, calculate the WAM features of the resulting image from scratch, and then undo the perturbation. This wastes a lot of computation power, especially considering the local nature of most of the steps involved in the calculation. The following improvements are aimed at lazily recalculating the features of a perturbed image based on cached results of the WAM pipeline prior to the perturbation.

Firstly, consider how the DWT coefficients change when a **single pixel** (X,Y) is perturbed. As 2D filtering is local, most of the wavelet coefficients will remain the same. In fact, changes will be present only in the 8×8 neighbourhood $[X-7...X]\times[Y-7...Y]$. Therefore, if we cache the results LL, LH, HL, HH, then we can redo the filtering only on the 8 rows and 8 columns in question.

A similar results holds for local variance and the residuals. As the variance around a DWT coefficient depends only on the 9×9 neighbourhood around it, changing a single pixel (X,Y) of the image affects only the variances in the region $[X-11...X+4]\times[Y-11...Y+4]$. The recalculation region for residuals is the same as for variances. If we cache the computed variances, perturbing a single pixel would require us to recompute a very small number of them. Unfortunately, there is no good way to update the prefixed squares structure outlined above, so instead we recalculate the dirty variances directly from the definition - by traversing the 9×9 neighbourhood around each changed DWT coefficient - and then update the residuals in the same region.

Unfortunately, the 27 moments need to be recomputed fully, because they are global. Nevertheless, performing the above lazy steps has significantly speeded up the WAM oracle. If we would like to know how the features change when **one** particular pixel is perturbed, we can perform a lazy query. It does the perturbation, updates the local quantities lazily as above, calculates the moments to obtain the feature vector of the perturbed image, and then performs the opposite perturbation in order to return to the original.

Often we will be interested in the effect on the features of **multiple** perturbations, rather than single-pixel ones. To query the effect of such a group change efficiently, we will perform lazy updates on the local quantities around each pixel involved, then calculate the moments only *once* at the end, then undo the changes. A final point on improving the feature extractor is that if too many pixels are involved in a perturbation, it might be cheaper to calculate *all* the variances using the prefixed squares form as above, instead of traversing the 9×9 region around each perturbed DWT coefficient. The exact number of pixels at the threshold will depend on the size of the image; our tests show that for 256×256 images it is about 3.

For testing purposes, we compared output with M. Goljan's original WAM code. In doing so, we uncovered two bugs in the original code. Firstly, in the local variance estimation of 3×3, ..., 9×9 regions around the point of interest, a fixed denominator of 9, 25, 49 or 81 is used, even near the edges, where the elements contributing to the summation are fewer. Secondly, the filtering routine underlying the DWT performs a convolution with the periodicity extension of the signal being filtered, rather than the signal itself. Both of these errors are relevant only near the edges (if at all), so we reimplemented them in order to be able to compare output with the original implementation of WAM. We believe they are inconsequential to the problem of feature restoration.

Chapter 3

Linearity of the WAM Features

This chapter investigates the linearity of the WAM features.

3.1 Definition of Linearity

Let f be a vector containing certain computable features of an image A. Consider two single-pixel changes on A: C_1 and C_2. Applying C_1 on its own produces an image with feature vector v_1. Likewise, C_2 on its own produces v_2. Performing both changes yields v_3. Let $\delta_1 = v_1 - f$, $\delta_2 = v_2 - f$, $\delta_3 = v_3 - f$. The feature set is said to be linear if $\delta_1 + \delta_2 = \delta_3$ for all pairs of changes C_1, C_2 on all images A.

3.2 Motivation

The aims of this chapter are to determine whether the WAM features are linear in the sense defined above, and, if they are not, to find out under what conditions two changes exhibit some degree of linearity. The motivation is that it would generally be helpful to have an intuitive grasp of how changing an image affects its WAM features. This intuition will be useful in designing algorithms for feature restoration.

From the definition of the WAM features, we can expect that they are not linear. Some nonlinearity is introduced by the calculation of local variances, and much more by the calculation of the nine moments for each subband. Therefore, it is reasonable to expect that $\delta_1 + \delta_2 = \delta_3$ will almost never hold. Moreover, due to the local nature of the variance estimation, we expect nonlinearities to be more evident for changes of pixels which are close to each other.

3.3 Experiments and Conclusions

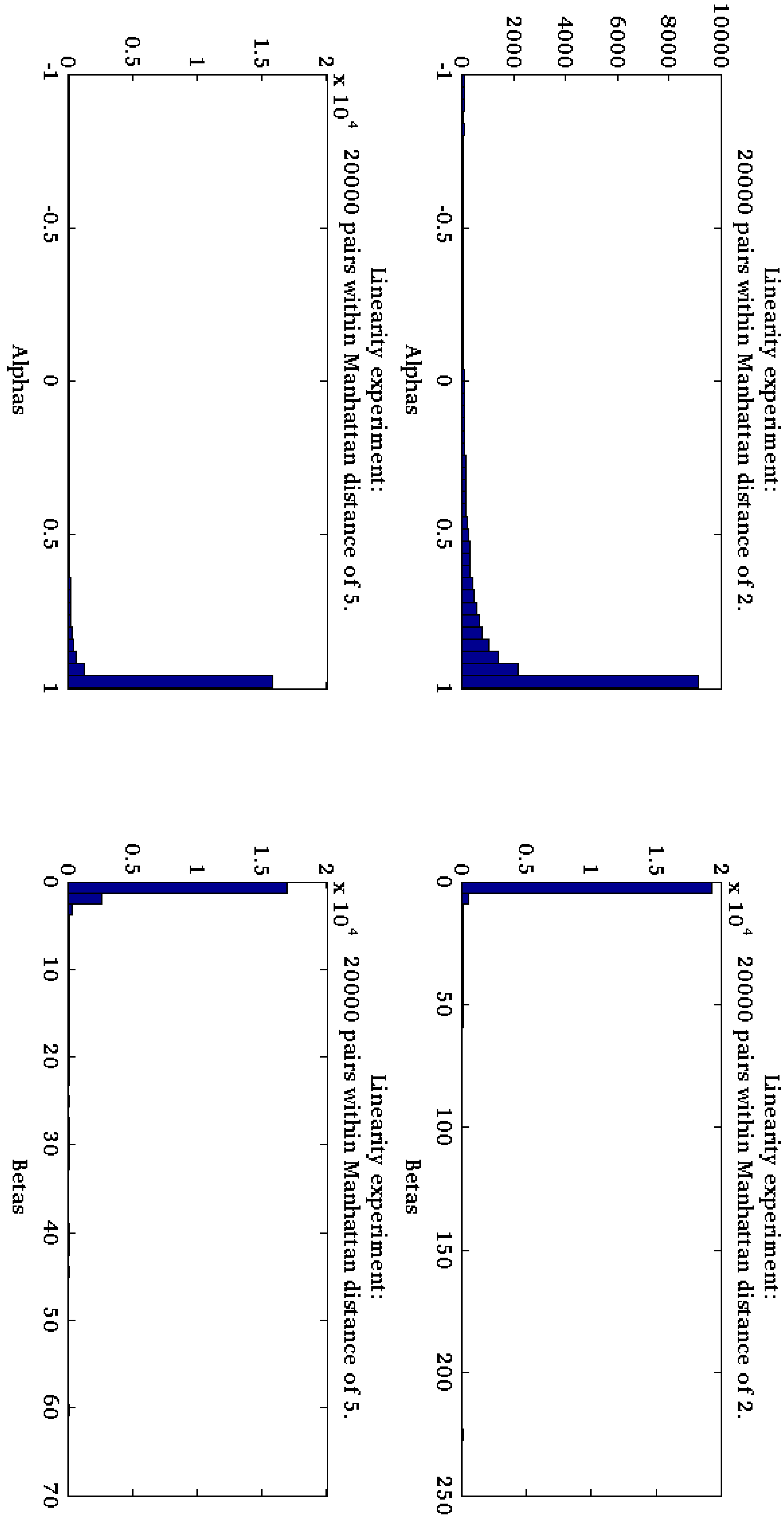
To measure linearity, we introduce two metrics. Firstly, $\alpha = \frac{\delta_3 \cdot (\delta_1 + \delta_2)}{\|\delta_3\| \|\delta_1 + \delta_2\|}$ is the cosine of the angle between $\delta_1 + \delta_2$ and δ_3. Secondly, $\beta = \frac{\|\delta_1 + \delta_2\|}{\|\delta_3\|}$ is the ratio of their sizes. For linear features, both metrics must be 1 for any pair of single-pixel changes.

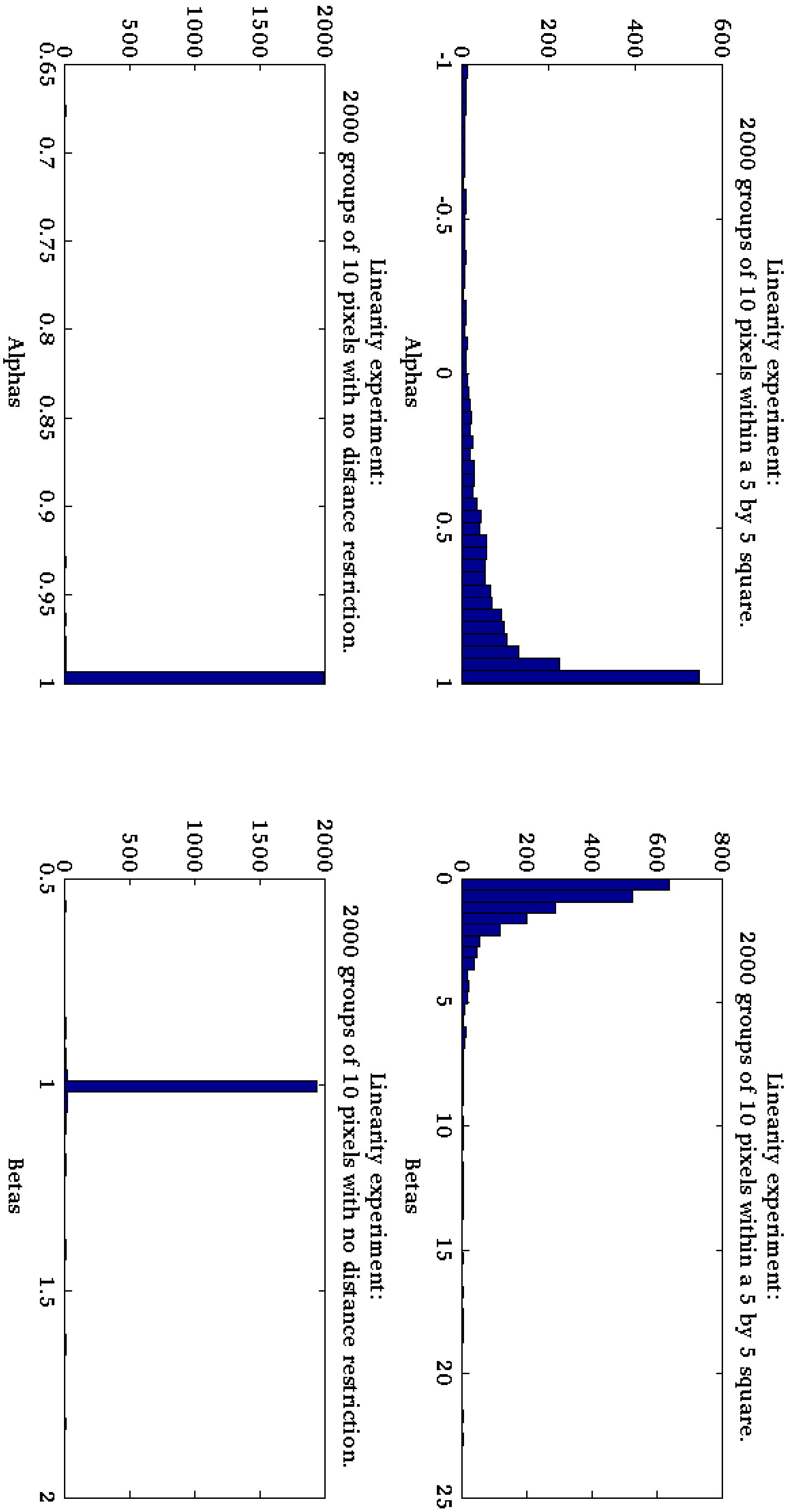
To investigate the linearity of the WAM features, we ran the following experiments. On a fixed image, randomly choose a pair of pixels with Manhattan distance[1] within Y. Randomly choose to perturb their values by 1 or -1. Query the WAM oracle to obtain δ_1, δ_2, and δ_3 - the effect of performing the first change, the second change or both changes, respectively. Then calculate the two metrics and record them. Repeat X times. The experiment was performed with $X = 20000$, $Y = 2, 5, 15, 30, \infty$. Below we show histograms of α and β for $X = 20000$, $Y = 2, 5$.

To test linearity on arbitrarily large groups of changes, we also ran a generalisation of the experiment: Pick X groups of Z changes, contained within a $(a \times b)$ neighbourhood of pixels. For each group, query the oracle to obtain $\sum_{i=1}^{Z} \delta_i$ (the sum of the individual effects) and δ_{Z+1} (the effect of all the changes together), and calculate the metrics. Histograms for $(X, Z, a, b) = (2000, 10, 5, 5)$ and $(2000, 10, \infty, \infty)$ are shown below.

Contrary to our expectations, pairs of pixel changes which exhibit linearity are very common, particularly when the pixels affected are far apart. The nonlinear nature of the WAM features manifests itself only for pairs of changes which are *very* close together. The result holds for larger groups of changes as well. Knowledge of this pseudo-linearity will be a helpful tool ahead.

[1]The Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$.


Linearity experiment:
20000 pairs within Manhattan distance of 2.
Alphas
Linearity experiment:
20000 pairs within Manhattan distance of 2.
Betas
Linearity experiment:
20000 pairs within Manhattan distance of 5.
Alphas
Linearity experiment:
20000 pairs within Manhattan distance of 5.
Betas


Linearity experiment:
2000 groups of 10 pixels within a 5 by 5 square.
Alphas
Linearity experiment:
2000 groups of 10 pixels within a 5 by 5 square.
Betas
Linearity experiment:
2000 groups of 10 pixels with no distance restriction.
Alphas
Linearity experiment:
2000 groups of 10 pixels with no distance restriction.
Betas

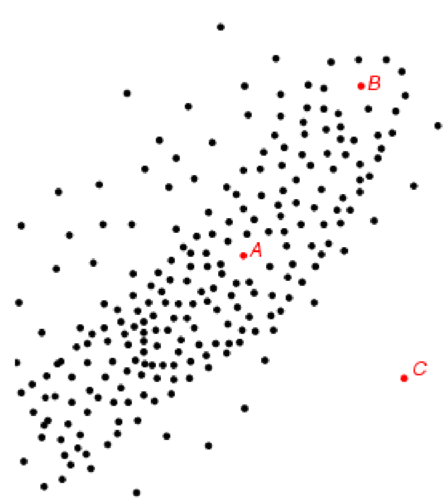
Chapter 4

Feature Distance Metrics

In feature restoration the goal is to make a stego-image less suspicious by changing its non-payload pixels so that the distance between the image's feature vector and a particular target feature vector is minimised. Therefore, a *notion of distance* is needed. This chapter shows why Euclidean distance is a poor choice for feature restoration and defines a distance measure more suited to the task.

4.1 Shortcomings of Euclidean Distance

The main disadvantage of Euclidean distance is its assumption that vector components are uncorrelated. When this assumption fails, the existing correlations are not taken into account by the norm. This could lead to difficulty if we are trying to gauge whether a particular feature vector may pass for innocent, based on an estimation of the cluster of all natural covers. To appreciate this, consider the figure below:



The example is a simplified one as it is in two dimensions, but it serves to illustrate the point. The cluster shown in the figure represents the features of all innocent-looking images. When a payload is embedded into image A, the resulting stego-image is likely to be well outside the cluster. B and C are two candidate restored images. B is inside the cluster and looks less suspicious than C. However, their Euclidean distances to the original A are the same, so they are equally good solutions under Euclidean distance. The WAM features are indeed correlated, because they are 3 sets of 9 successive moments. This indicates the need to use a non-Euclidean notion of distance on WAM feature vectors.

4.2 The Mahalanobis Norm

4.2.1 Notation

The expectation of a random variable x_i is $E(x_i)$. The covariance of two random variables x_i, x_j with $E(x_i) = \mu_i$ and $E(x_j) = \mu_j$ is $Cov(x_i, x_j) = E((x_i - \mu_i)(x_j - \mu_j))$. Extend this notation to random vectors. For a multivariate, random vector $x = (x_1, x_2, ..., x_n)^T$ with $E(x_i) = \mu_i$, the expected value is $E(x) = (\mu_1, \mu_2, ..., \mu_n)^T = \mu$, and the covariance matrix is $S = E((x-\mu)(x-\mu)^T)$, so that $S_{ij} = E((x_i - \mu_i)(x_j - \mu_j)) = Cov(x_i, x_j)$.

Note that the covariance matrix of a real-valued random vector is symmetric, therefore, by the finite-dimensional spectral theorem, it is always diagonalizable by an orthogonal matrix Q: $S = Q\Lambda Q^T$ where $Q^T Q = QQ^T = I$ and Λ is a diagonal matrix containing the eigenvalues of S. Recall also that any invertible covariance matrix is positive definite, which makes its eigenvalues strictly positive[1]. This implies the existence of $\Lambda^{-\frac{1}{2}}$, obtained by replacing every diagonal entry of Λ by the root of its reciprocal.

4.2.2 Whitening

Let the covariance matrix of the random vector x be S, diagonalized as $Q\Lambda Q^T$, and let the mean of x be μ. Define the linear transformation $w(x) = \Lambda^{-\frac{1}{2}} Q^T (x - \mu)$. We can make two observations about w.

Firstly, $E(w)$ is the zero vector. This is immediate from linearity of expectation:

$$E(w) = E(\Lambda^{-\frac{1}{2}} Q^T (x-\mu)) = \Lambda^{-\frac{1}{2}} Q^T (\mu - \mu) = 0$$

Secondly, the covariance matrix of w is the identity matrix I:

$$E((w - E(w))(w - E(w))^T) = E(ww^T) = E(\Lambda^{-\frac{1}{2}} Q^T S Q \Lambda^{-\frac{1}{2}}) =$$

$$= E(\Lambda^{-\frac{1}{2}} Q^T Q\Lambda Q^T Q \Lambda^{-\frac{1}{2}}) = E(\Lambda^{-\frac{1}{2}} I \Lambda I \Lambda^{-\frac{1}{2}}) = I$$

[1] In general, covariance matrices are positive semi-definite, which allows 0 to be an eigenvalue, making the matrix singular. This is the case exactly when one of the random variables is a linear combination of the others, which would mean the random vector was not entirely random to begin with. In this discussion, we ignore such pathological cases.

Now, consider applying the transformation w to the whole image of the random vector x. The first observation says that the resulting cluster will be centred around the origin. The second says that any two distinct components of the transformed vector will have zero covariance, so there will be **no correlation** among the vector components. The transformation squashes the cluster into a multi-dimensional sphere around the origin. This transformation is called *whitening*, and the resulting vectors are called *white*.

4.2.3 Mahalanobis's Distance

For a whitened the cluster, the issues discussed above are no longer present. A natural idea in defining a norm that accounts for correlations is to whiten the cluster using w, and then use Euclidean distance on the result. The distance between the centre of the cluster μ and a vector x is:

$$\|w(x)-w(\mu)\|_E = \|w(x)\|_E = \|\Lambda^{-\frac{1}{2}}Q^T(x-\mu)\|_E = \sqrt{((x-\mu)^T Q(\Lambda^{-\frac{1}{2}})^T)(\Lambda^{-\frac{1}{2}}Q^T(x-\mu))} =$$

$$= \sqrt{(x-\mu)^T Q\Lambda^{-1}Q^T(x-\mu)} = \sqrt{(x-\mu)^T S^{-1}(x-\mu)}$$

More generally, a measure of distance between two arbitrary points x and y is:

$$\|w(x)-w(y)\|_E = \sqrt{w(x)^T w(x) - 2w(x)^T w(y) + w(y)^T w(y)} =$$

$$= \sqrt{(x-\mu)^T S^{-1}(x-\mu) - 2(x-\mu)^T S^{-1}(y-\mu) + (y-\mu)^T S^{-1}(y-\mu)}$$

$$= \sqrt{x^T S^{-1}x + y^T S^{-1}y - 2x^T S^{-1}y} = \sqrt{(x-y)^T S^{-1}(x-y)}$$

Thus, the Mahalanobis norm of x with mean μ and covariance matrix S is defined by $\|x\|_M = \sqrt{(x-\mu)^T S^{-1}(x-\mu)}$, and the Mahalanobis distance of x and y is $Dist_M(x,y) = \sqrt{(x-y)^T S^{-1}(x-y)}$. The Mahalanobis distance is a measure of dissimilarity. A small distance means x and y are equally likely to belong to the cluster. Conversely, a large distance means one is well within the cluster, and the other is far from it. This notion of distance was originally introduced in [11].

In order to use the Mahalanobis distance, we need two estimations. Firstly, the feature vector μ - the average of the WAM vectors of all natural cover images. Here, by 'natural' we mean ones that carry no steganographic payload. Secondly, the 27 by 27 covariance matrix S of the WAM vector, along with its inverse. Computing these exactly is impossible, as it would mean calculating the WAM features of all natural images.

For the purposes of estimating μ and S, we used 17500 natural images, IMG_1 to IMG_{17500}, each with dimensions 256×256, cropped from 500 images with dimensions 1500×2000. The original $3Mpx$ images were taken with a Minolta DIMAGE A1 camera, and have never been subjected to compression. The estimation of the mean vector was:

$$\mu = \frac{1}{17500}\sum_{i=1}^{17500} WAM\,(IMG_i)$$

We obtained the unbiased estimate of the covariance matrix:

$$Cov(x_i, x_j) = \frac{1}{17499} \sum_{t=1}^{17500} (WAM(IMG_t)_i - \mu_i) \left(WAM(IMG_t)_j - \mu_j \right)$$

Then we inverted the covariance matrix using Matlab. The choice was motivated by the fact that Matlab's implementation of the Gauss-Jordan algorithm is known to have a good degree of numerical stability, hopefully reducing precision-related problems to a minimum.

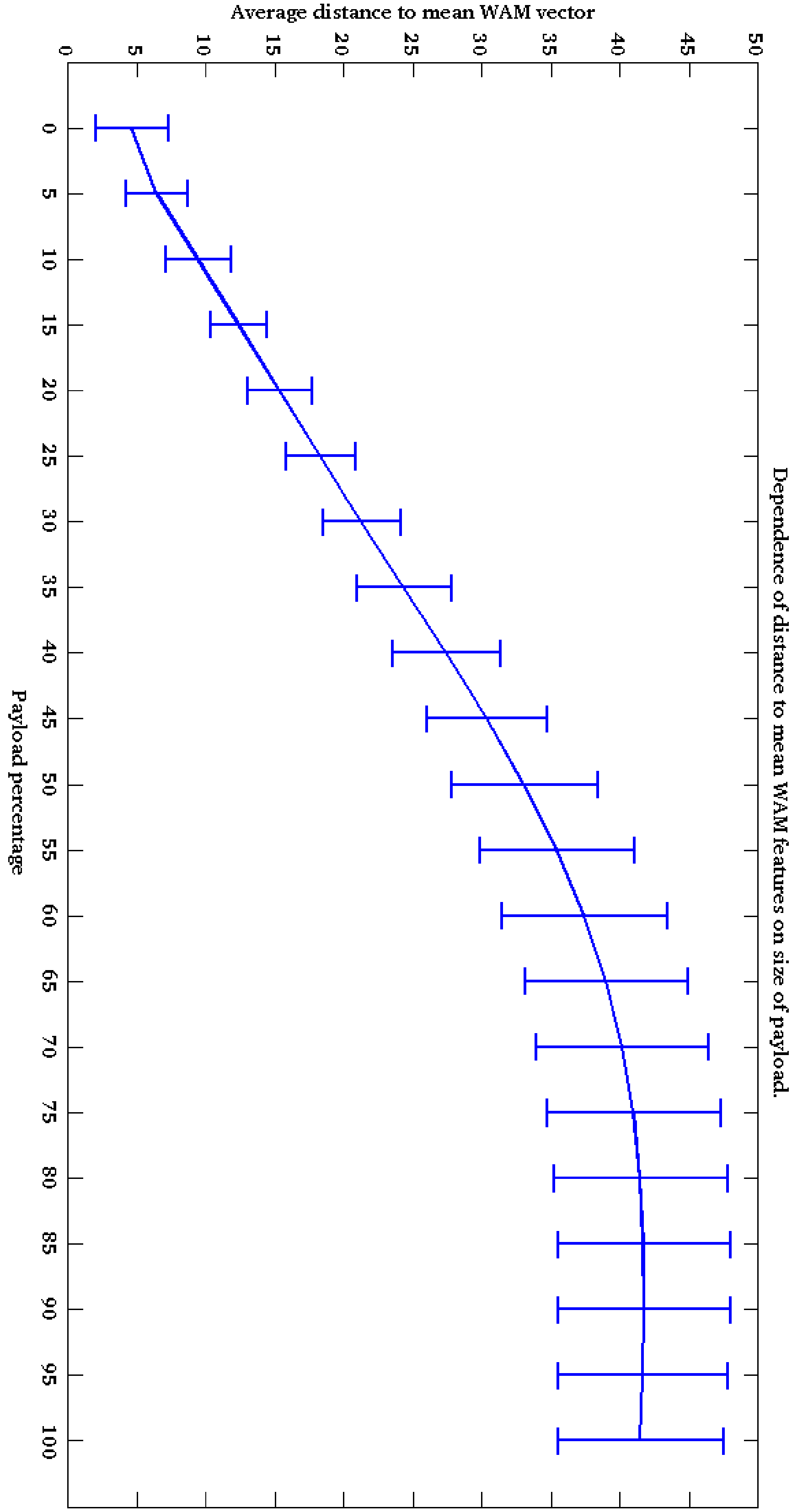
4.3 Contextualising the Distance

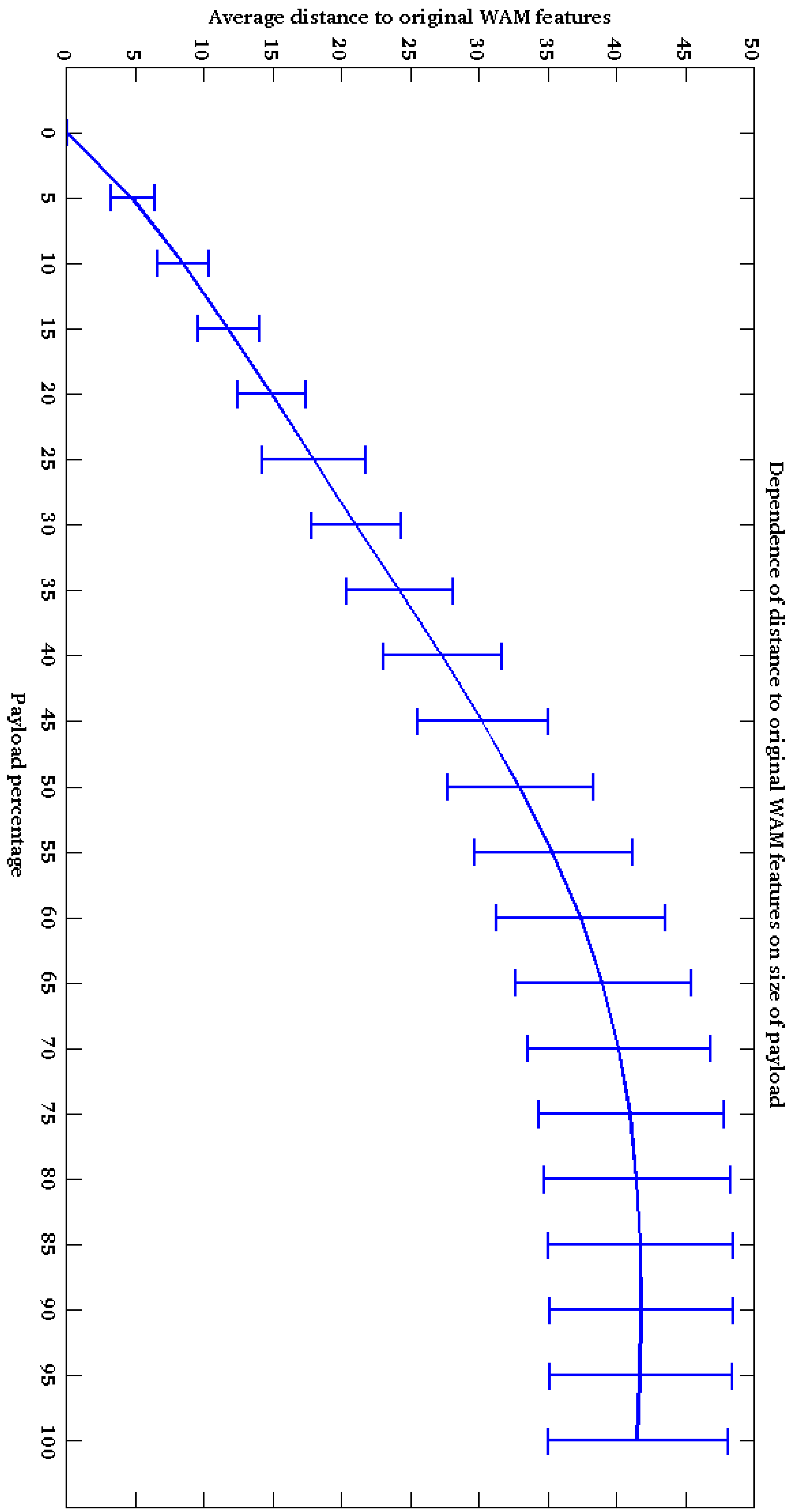
Let g and f be the feature vectors of the original image and the stego-image, respectively. We may choose from two distance measures to minimise in feature restoration:

- $Dist(g, f)_M$, the Mahalanobis distance to the **original** image's WAM vector
- $Dist(\mu, f)_M$, the Mahalanobis distance to the estimated **mean** WAM vector

Since the WAM features are sensitive to noise, and a larger payload introduces more noise, it is to be expected that the distance measures depend on the payload percentage. This section provides context for the numbers, and gives an intuition of how embedding various payloads changes the distance measures.

We estimated the average value of the distances for a number of fixed payloads. For each payload percentage, we embedded a message of the appropriate length into the covers IMG_1 to IMG_{17500}, calculated their WAM features, and computed the distances. Finally, the average and the standard deviation of the distances at that payload were computed. The results appear in the two figures below. They look almost exactly the same, so at this point we make the decision that we will only use $Dist(\mu, f)_M$ to measure the performance of feature restoration algorithms. With this choice of target, we will actually be making the features plausible, instead of restoring them.


Dependence of distance to mean WAM features on size of payload.
Average distance to mean WAM vector
0
5
10
15
20
25
30
35
40
45
50
Payload percentage
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100


Dependence of distance to original WAM features on size of payload
Average distance to original WAM features
0
5
10
15
20
25
30
35
40
45
50
Payload percentage
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100

Chapter 5

Feature Restoration

Now we are in a position to attack the problem of feature restoration. We have chosen a steganographic scheme for encoding and decoding messages into and from greyscale images (LSB-matching) and a particular set of features for steganalysis (WAM), and have defined a notion of an image's suspiciousness (the Mahalanobis distance between its WAM vector and the estimated mean WAM vector μ). Feature restoration is the problem of deliberately altering the non-payload pixels of a stego-image in order to reduce its suspiciousness. At our disposal, we have two black-box tools: an efficient oracle able to calculate the features of any image and foresee the effect of any perturbation, and a distance calculator able to measure the suspiciousness of any feature vector. We are also aware that the features are pseudo-linear.

In this chapter, we present a number of algorithms we have devised for feature restoration. We have implemented each of these algorithms and benchmarked its performance. Each section will describe an algorithm, along with the ideas motivating it, and give results from our benchmarking. We conclude the chapter with a comparison and a pointer to further work that should be done on the topic.

To benchmark each algorithm A, we used a set of 100 images of size 256×256. For each payload size of interest, we embedded a payload of that size into each of the 100 images, obtaining a stego set for that payload. Then we ran A on each of the 100 stego images, giving it an allowance of 50000 WAM oracle queries per image. We kept track of the distance to μ at each query count, and averaged this distance over the 100 stego images, obtaining a curve of the average distance to μ at each query count. The decision to measure distance reduction versus queries performed instead of clock time was based on the idea that query usage is a performance measure independent of implementation details and hardware[1]. The payloads we considered in this way were 50% (a low payload), 90% (a high payload but nonetheless allowing some degree of freedom), and 99% (an extremely high and constraining payload).

Some of our algorithms have a parameter space. In these cases, we use smaller-scale benchmarking in order to explore the parameter space and converge on a good set of parameters. This uses only

[1] Just to give the reader an intuition of how queries used translate to time: 1000 queries take about 11 seconds when each query refers to a single-pixel change, and about 30 seconds when each query refers to 25 changes. This is on an Intel(R) Core(TM) I5 CPU running at 2.27 GHz with 4 GB of RAM.

20 stego images, at only one payload percentage (90%). Then the algorithm with the chosen parameters is benchmarked fully.

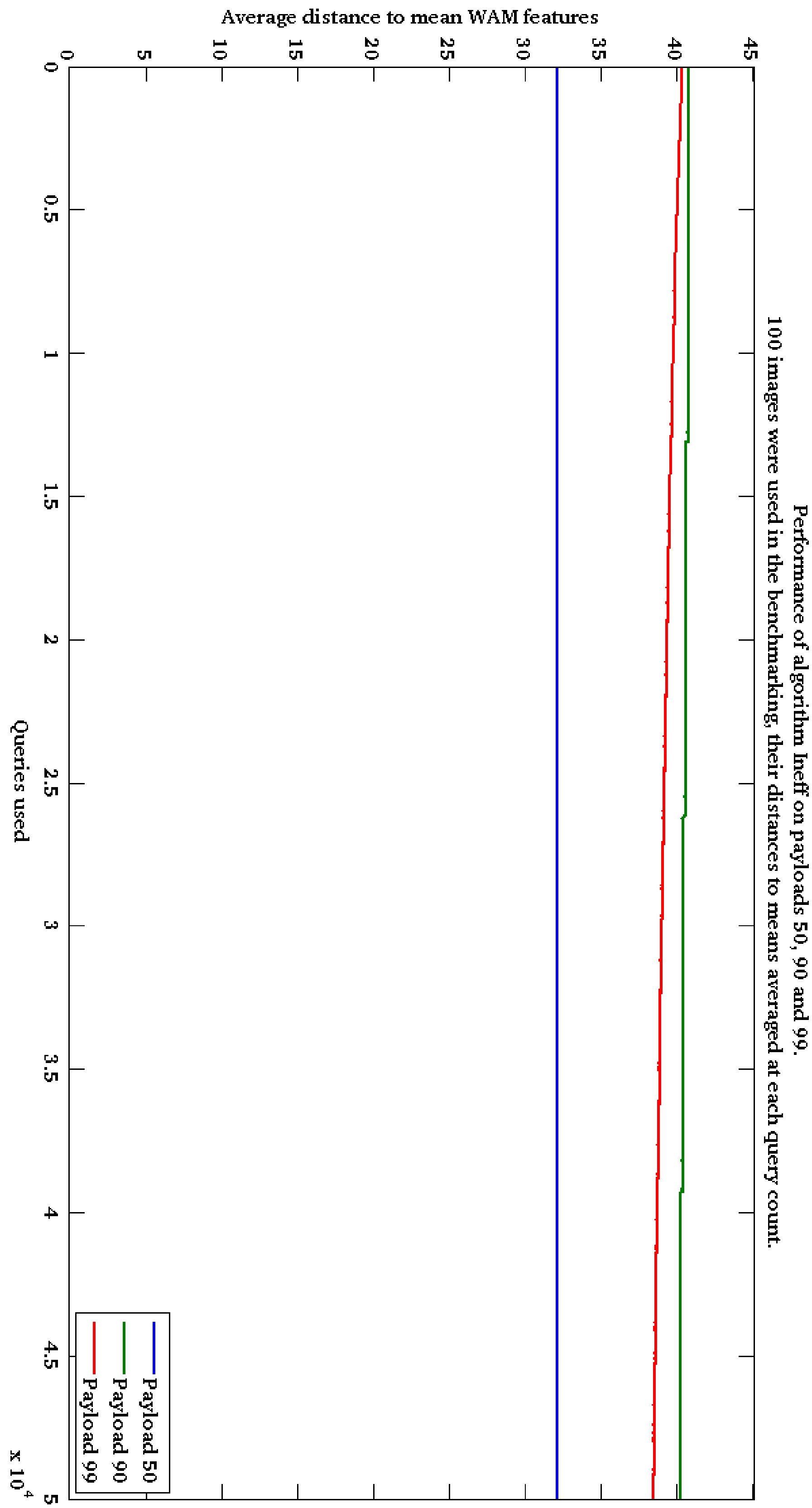
A final point concerns the maximal value by which a restoration algorithm may change each non-payload pixel. It is clear that such a maximum must be set and adhered to in order to prevent distortions of the image which are visible to the naked eye. However, setting such a limit is not easy because the question of which distortions are visible is subjective. The matter is further complicated by the fact that cameras often have imperfections which result in artifacts appearing in natural images. We have fixed this maximum change cap at 10. This limit, though perhaps somewhat generous, was chosen for two reasons. Firstly, 10 appears to be the threshold where distortions of the output image are invisible to the casual observer, but barely noticeable upon very close inspection (zooming in). Secondly, we were interested in finding out the extent to which the nature of WAM allows the distance to μ to be reduced; therefore, we elected not to constrain the feature restoration algorithms too much with subjective notions of how noticeable a distortion is and whether it may be plausibly blamed on the camera. If one is indeed concerned about masking one's attempt at feature restoration against a human observer, one may use a smaller limit in the interest of remaining undetected. The limit was implemented and adhered to in our algorithms and in their benchmarking.

5.1 Greedy and Inefficient Algorithm

We begin with the only algorithm for feature restoration appearing in literature. It was suggested in [2], more to introduce the concept of feature restoration than as a serious attempt at solving the problem.

The algorithm performs a number of iterations. On each iteration, it considers each non-payload pixel in turn, and queries the WAM oracle twice - once for the effect of perturbing the pixel by 1, and once for -1. Throughout the iteration, the algorithm keeps track of the perturbation which brings the feature vector closest to μ. At the end of the iteration, the algorithm realises the best change found. Iterations are performed for as long as there are queries available. If an iteration finds no distance-reducing change, the algorithm terminates.

Unfortunately, this algorithm is very inefficient. At each iteration it uses a very large number of queries - $2(1-p)NM$, where p is the payload percentage and (N,M) are the dimensions of the image - but only performs one actual change. The smaller the payload, the less feasible this algorithm becomes, as each iteration requires a larger number of oracle queries. The care taken in selecting the best available change at each iteration would be justified if this change entailed a very significant distance reduction compared to runners-up. However, our experiments showed that at each iteration, a very large number of the changes considered entail a distance reduction virtually as good as that of the best change, so the queries spent on them are essentially wasted. The performance graphs of the algorithm are flat:

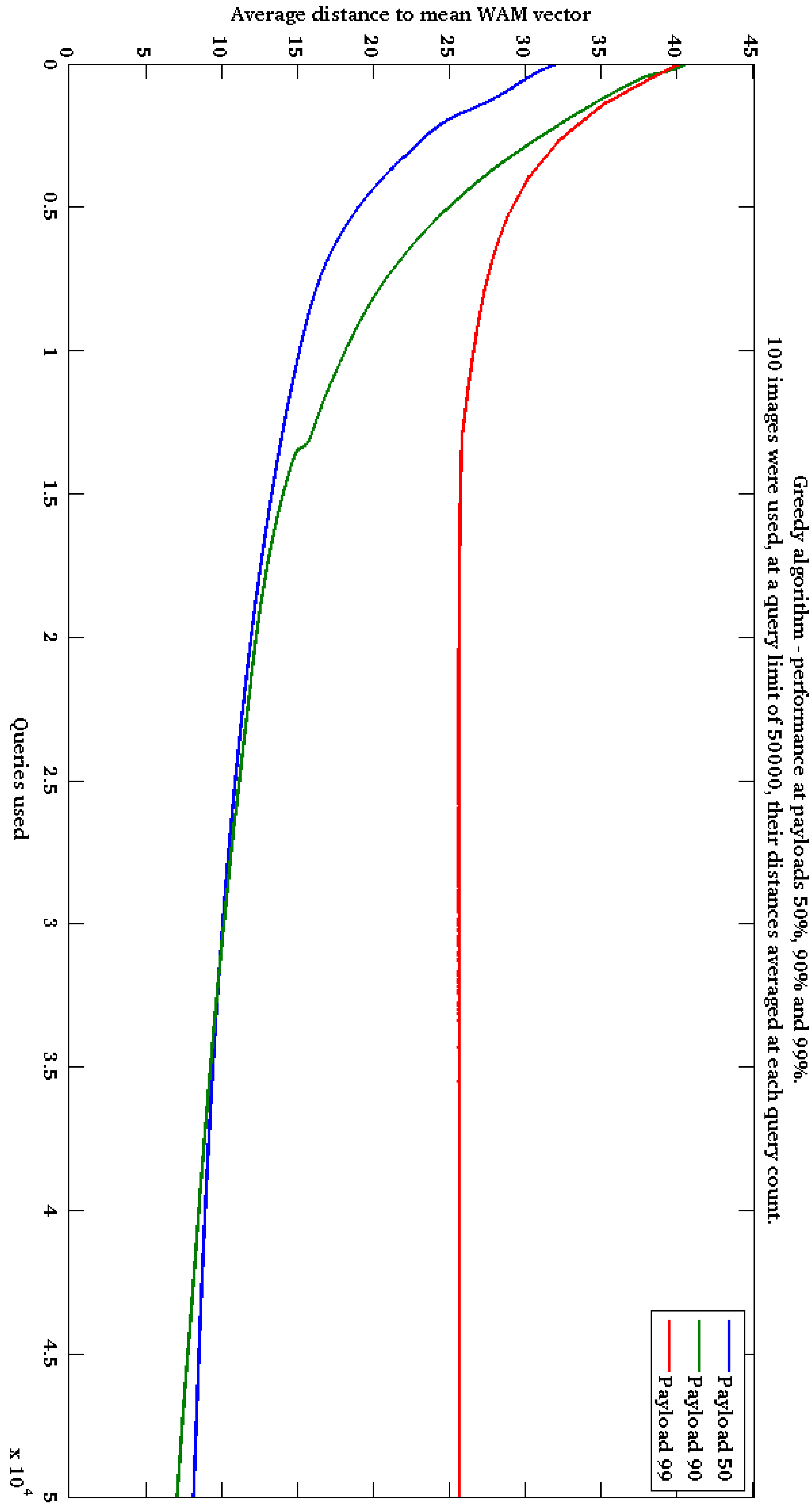

Performance of algorithm Ineff on payloads 50, 90 and 99.
100 images were used in the benchmarking, their distances to means averaged at each query count.
Average distance to mean WAM features
Queries used
x 10^4
0
5
10
15
20
25
30
35
40
45
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Payload 50
Payload 90
Payload 99

5.2 Greedy Algorithm

While the algorithm in the previous section is infeasible except for very large payloads, it offers a good starting point for refinement. As we mentioned earlier, the inefficient algorithm discards many reducing changes in its search for the best one, thereby wasting far too many queries. Many of these discarded changes in fact entail a large distance reduction. Moreover, having seen the pseudo-linearity of the WAM features, we can expect that if a change C_1 is distance-reducing *before* change C_2 is performed, then C_1 is likely to still be distance-reducing *after* C_2 is performed, especially if the pixels involved in C_1 and C_2 are far apart. These observations suggest a much more efficient greedy algorithm.

Traverse the non-payload pixels of the image in order. For each pixel (i, j), query the WAM oracle for the result of perturbing (i, j) by 1 and -1. If neither change is reducing, move on. If one of the two changes is reducing, perform it and move on. If both are reducing, perform the one which entails a greater distance reduction and move on. If all non-payload pixels are traversed in this way, wrap around and traverse them again. Repeat for as long as oracle queries are available.

The performance charts for this greedy algorithm are found below. An interesting point to note in the chart is that the curve corresponding to our experiments at 90% payload eventually overtakes the curve for 50%. We attribute this to the difference in the number of non-payload pixels. At 50%, a single traversal of all the pixels takes much more queries than at 90%. Consequently, at 90%, each individual pixel is visited by the algorithm much more frequently. The better performance of the algorithm at 90% payload suggests that there are often pixels which should receive concetrated attention and be perturbed multiple times for maximum distance reduction. This insight leads to the algorithm in the next section.


Greedy algorithm - performance at payloads 50%, 90% and 99%.
100 images were used, at a query limit of 50000, their distances averaged at each query count.
Average distance to mean WAM vector
Queries used
x 10^4
0
5
10
15
20
25
30
35
40
45
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Payload 50
Payload 90
Payload 99

5.3 Variance Sort Algorithm

A possible weakness of the Greedy algorithm is that it spends an equal proportion of its query allowance on each non-payload pixel, when it might be better to traverse the pixels in a carefully chosen order and, when visiting a pixel, attempt to do as much work on it as possible, instead of simply perturbing it by +1 or -1 and moving on. This sets two questions: firstly, how to decide how much work is needed at each pixel, and secondly, what order to traverse them in?

A good answer to the first question is to begin by querying the oracle for the effects of perturbing the pixel (i, j) by +1 and -1 in order to determine a *direction of change.* If both changes increase the distance to μ, ignore this pixel. If one change increases the distance, but the other decreases it, choose the latter. If both changes decrease the distance, choose the one which entails a greater distance reduction. After a direction $d = \pm 1$ is chosen, repeatedly perturb the pixel by d, querying the oracle for the effect of the change (i, j, d) at each step, for as long as this change reduces the distance to μ.[2]

To answer the second question, we remind ourselves that the WAM features are a measure of noise. Noisy regions of the image contribute most to the distance to μ, so we should focus on them first. Therefore, we decided to estimate the variance in the *spatial* domain at each pixel, and to consider pixels in the way outlined above in order of decreasing variance. In this way, we hope to systematically restore the noisiest bits of the stego-image first. The algorithm is:

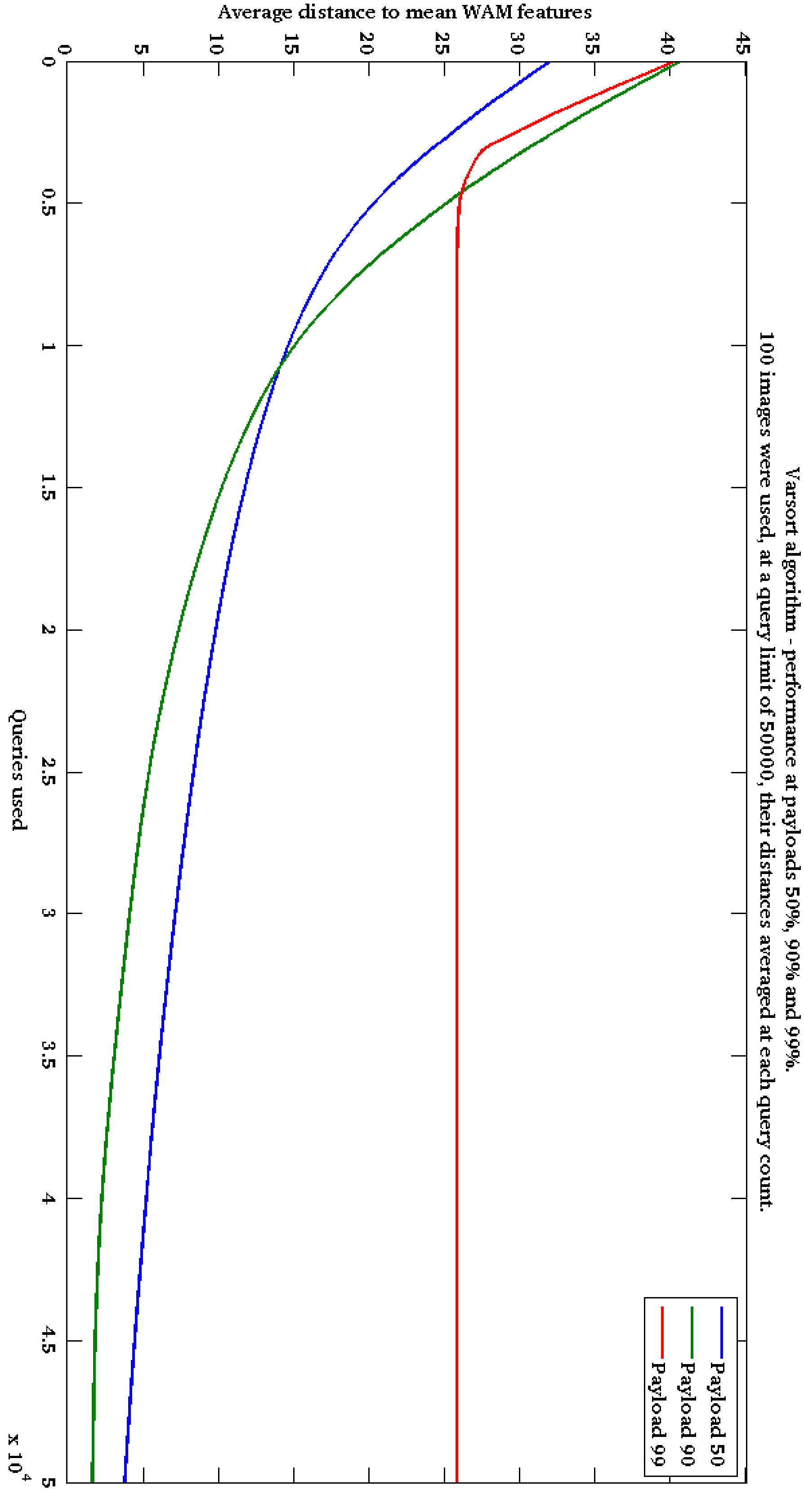
1. Estimate the variance in the spatial domain around each non-payload pixel (i, j). To do this, consider the 5×5 square of pixel values with (i, j) at its centre; let these be x_1 to x_n (most of the time n will be 25, near the edges it will be less). Their mean is $\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$, and their variance is the unbiased estimator $\widehat{Var(x)} = \frac{1}{n-1}\sum_{k=1}^{n}(x_k - \bar{x})^2$. Do this for all non-payload pixels, and sort them in order of decreasing variance to obtain the sorted list of pixels *L*.

2. Traverse *L*. For each pixel (i, j) in *L*, perform step 3. If you reach the end of *L*, wrap around to its beginning. If you run out of oracle queries, terminate.

3. At pixel (i, j), query the WAM oracle twice - with (i, j, 1) and (i, j, -1). If both changes increase the distance to μ, move on. Otherwise, select $v = \pm 1$ such that (i, j, v) entails a greater reduction of the distance to μ. Then repeatedly perform change (i, j, v) and query the oracle with (i, j, v); if the oracle predicts an increase in the distance then stop and move on to another pixel. Also, (i, j) is allowed to differ by at most 10 from its value at the beginning of the restoration algorithm. If this limit is reached, move on to another pixel.

The performance chart for the algorithm appears below. The Varsort algorithm is a definite improvement over the Greedy one. At payloads 50% and 90%, it completely conceals the payload within the query allowance, whereas at payload 99% it brings the distance down to the same level as the Greedy algorithm, but in fewer queries.

[2]And, of course, for as long as the cumulative change at pixel (i, j) is within the limit of 10 that we set at the beginning.


Varsort algorithm - performance at payloads 50%, 90% and 99%.
100 images were used, at a query limit of 50000, their distances averaged at each query count.
Average distance to mean WAM features
Queries used
x 10^4
Payload 50
Payload 90
Payload 99
0
5
10
15
20
25
30
35
40
45
0.5
1
1.5
2
2.5
3
3.5
4
4.5

5.4 Genetic Algorithm

The next algorithm attempts to combine randomness as a source of candidate perturbations with a deliberate process of selecting the most beneficial ones. It was inspired by genetic algorithms.

Recall that a *change* or *singleton change* is just a triple (x, y, v) where x and y refer to the position of a pixel, and v is the value by which it is perturbed. Recall also that a "group change" is a set of changes. On a given image, an *individual* is defined as a pair $(gc, distRed)$ where gc is a group change and $distRed$ is a real number - the amount by which the distance between the WAM features of the image and μ would be reduced if gc were performed. A *population* is defined to be a set of individuals.

Our genetic algorithm will maintain a population P which changes over time. The following operators are defined on it:

- *Inward Migration.* This is the mechanism by which the population grows. A singleton change is randomly chosen, with its value component equal to 1 or -1. Then the WAM oracle is queried with the change to obtain the distance reduction it entails. If this reduction is negative (that is, the change *increases* the distance to μ), the change is discarded and another attempt is made. When a singleton distance-reducing change is found, it is wrapped with its distance reduction label to form an individual, and is inserted into P.
- *Merge.* This is a mechanism to hopefully obtain good distance-reducing changes from existing ones in the population and to eliminate bad ones. All the individuals in P are sorted in decreasing order of their distance reduction labels. The worst few individuals are discarded; the exact number or proportion may be specified as an argument to the operation. Then the remaining individuals are paired up: the best with the 2nd best, the 3rd with the 4th, and so on. In case of an odd number of individuals, the worst is discarded. Then the two individuals in each pair are *merged*: the group change of the result is the union of the two group changes[3], and the distance label of the result is obtained by querying the WAM oracle.
- *Outward Migration.* This is the mechanism which alters the image by applying individuals to it. The individual with the greatest distance reduction label in P is found and removed from P. If its distance reduction label is positive, its group change is applied to the image. If not, it is discarded. It is also discarded if performing the group change would make one or more pixels differ from their original values by more than our limit of 10.

Note that distance reduction labels may become outdated. An individual's distance reduction label is correct when the individual is introduced into P with an inward migration (because the migration explicitly queries the WAM oracle), but afterwards the image may be modified by outward migrations, making distance reduction labels inaccurate. However, the pseudo-linearity of WAM implies that the inaccuracy is likely to be small. Thus, a distance reduction label in P should be seen as very untrustworthy only if a large number of outward migrations have been performed since the individual was introduced into the population.

[3] If the two group changes being unioned both refer to some pixel (x, y) with perturbation values u and v, respectively, then the result of the union refers to (x, y) with value u+v.

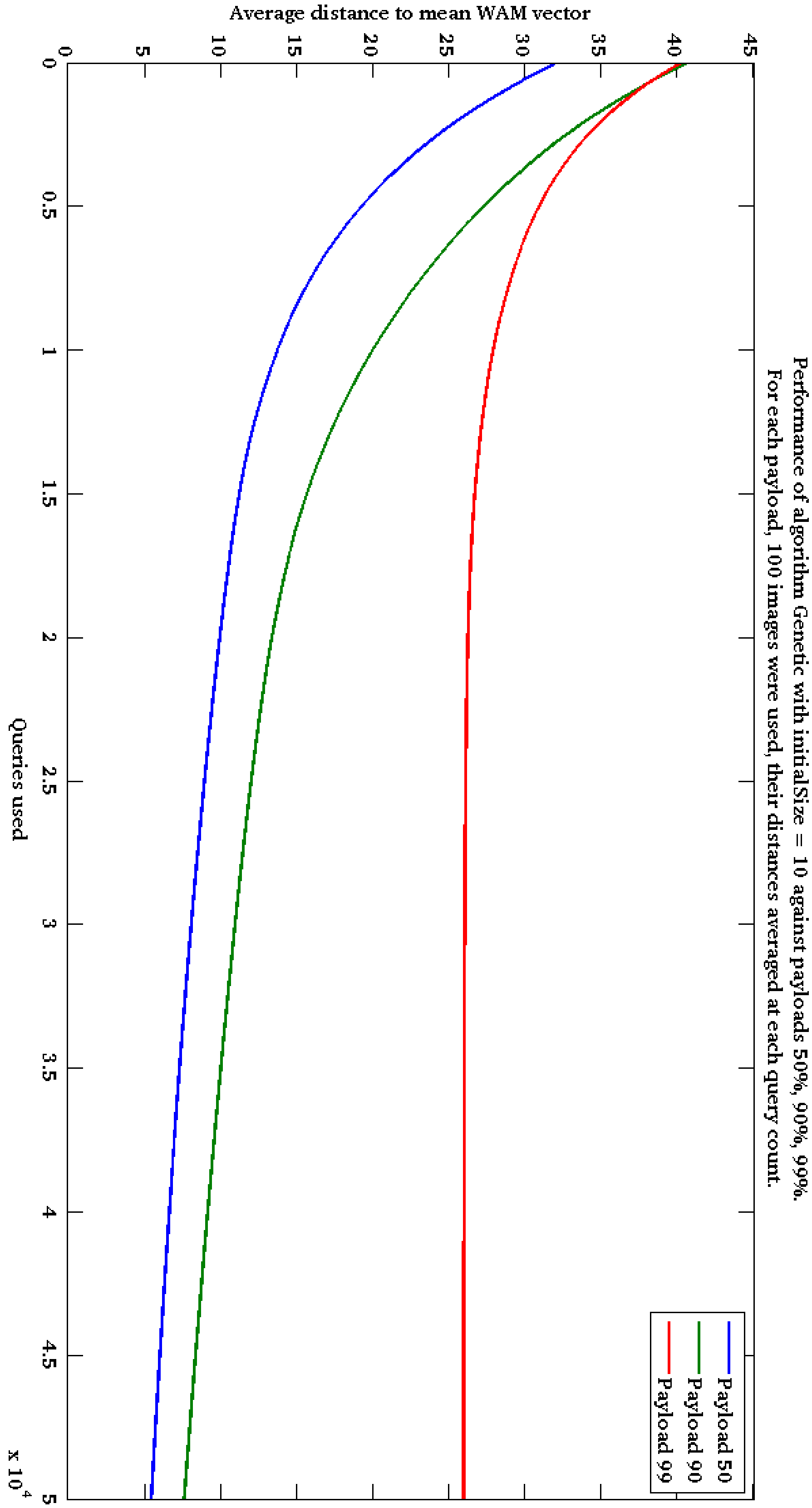
The mechanism of merging serves three purposes. Firstly, it eliminates the worst elements of the population, in the spirit of genetic algorithms. Secondly, it attempts to obtain better candidate group changes by combining good ones appearing in the population, as is common in genetic algorithms. More subtly, however, it refreshes the distance-reduction labels of the whole population: recall that the label of the union of two individuals is obtained by querying the WAM oracle directly. Thus, all the labels appearing in P are accurate immediately after a Merge, so applying this operation occasionally should serve to increase their reliability.

Having defined the operations above, we only need to decide how to combine them in order to obtain a full algorithm. Our algorithm has a parameter *initialSize*. Throughout, P will vary in size from *initialSize* to 2.1**initialSize*. Our algorithm is:

1. Initialise P by performing an Inward Migration *initialSize* times.
2. Iterate for as long as oracle queries are available:
3. On each iteration, perform two Inward Migrations and one Outward Migration. Thus, on each iteration, the size of P grows by 1.
4. If this size reaches 2.1**initialSize,* perform a Merge which begins by discarding 0.1**initial-Size* individuals. Then the merging process halves the size of the population, bringing it down to *initialSize.*
5. When the query allowance is exhausted, perform Outward Migrations until P becomes empty, then terminate.

In our implementation, the population is a priority queue, with higher priority given to individuals with greater distance reduction, to speed up the operations defined above. As a minor implementation issue, we point out that care was needed to ensure that the algorithm does not use more queries than it is allowed to, considering that both Merge and Inward Migration use more than one query at a time. Trivial modifications were made to allow these routines to stop partway through and avoid exceeding the query limit.

To obtain a good value for *initialSize*, we performed some partial benchmarking with a number of possible values (see Appendix D). Our results showed that a small *initialSize* is preferable. Therefore, we proceeded to fully benchmark the algorithm with *initialSize* = 10. The results are in the chart below. The algorithm achieves a good distance reduction overall, but is outclassed by the Greedy and Varsort algorithms.


Performance of algorithm Genetic with initialSize = 10 against payloads 50%, 90%, 99%.
For each payload, 100 images were used, their distances averaged at each query count.
Average distance to mean WAM vector
Queries used
x 10^4
Payload 50
Payload 90
Payload 99

5.5 Random Algorithm

Now that we have seen an algorithm which partly uses randomness, it is interesting to explore one which is purely random. It is a very simple concept, but the results from it yielded two important insights into feature restoration.

The algorithm has two parameters, *lots* and *changesPerLot*, and works as follows:

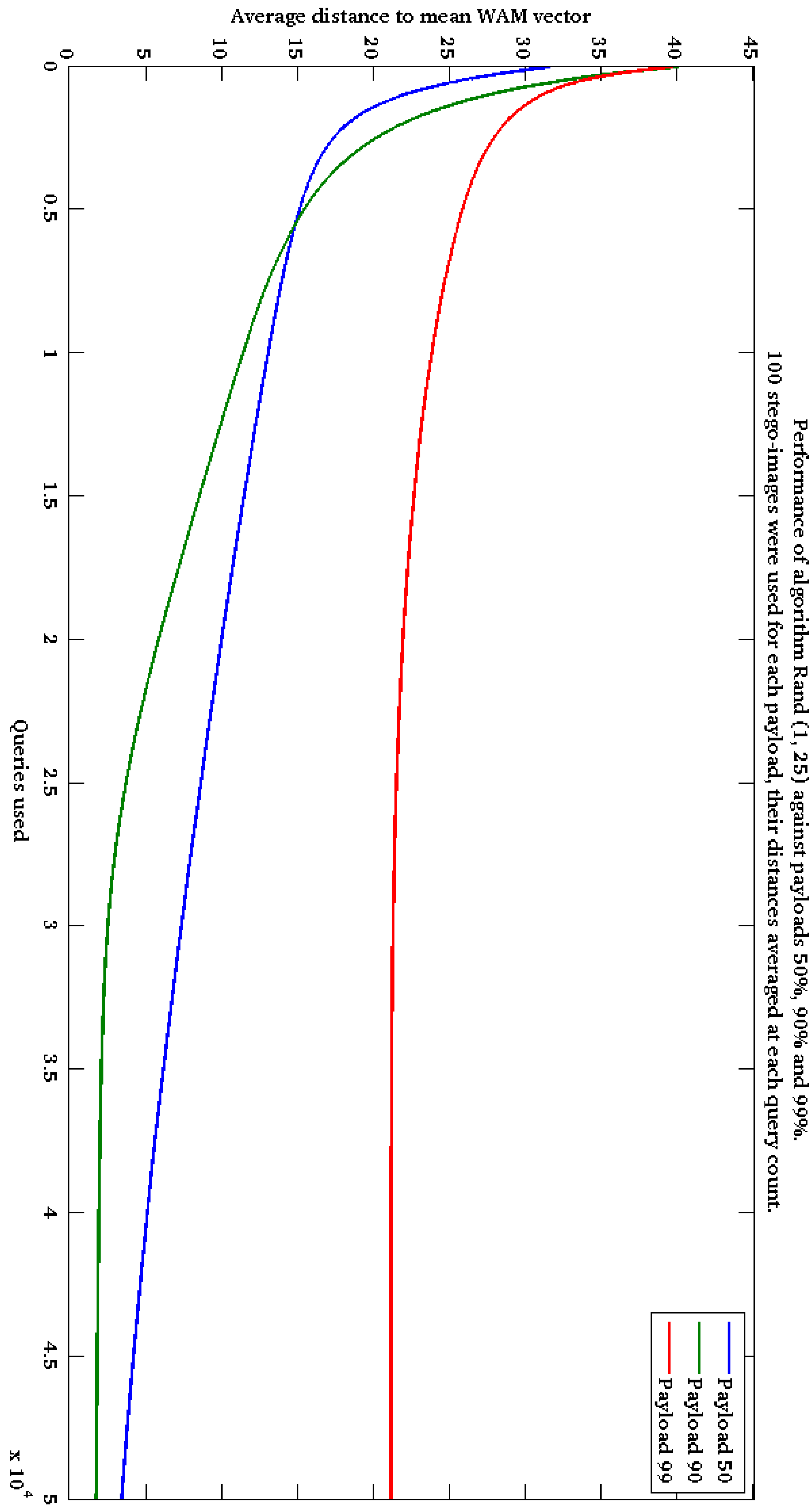
1. Iterate for as long as there are queries available:
2. On each iteration, randomly choose *lots* group changes. Each group change must contain *changesPerLot* singleton changes. Each singleton change must have value equal to -1, 0 or 1; at least one singleton change must have a non-zero value. (Equivalently, each group change contains **upto** *changesPerLot* singleton changes, each with value ± 1.) Query the WAM oracle for the effect of each of the *lots* group changes. Perform the one which reduces the distance to μ most, discard the rest. If all the group changes increase the distance, perform none.
3. Whilst iterating, keep track of how many consecutive void iterations (ones which perform no change) have been done. If this number reaches 100, reduce *changesPerLot* by 25%, to a minimum of 1.

Increasing the parameter *lots* makes the algorithm less greedy and more cautious - at each iteration, more work is done and more queries are spent, in the hope that this will yield a better distance-reducing change. Increasing the parameter *changesPerLot* has the effect of making the algorithm conserve its queries by including more singleton changes in each group change and querying the oracle only once for their cumulative effect.

The motivation for step 3 is that if the algorithm is consistently unable to find *large* distance-reducing group changes, then perhaps there are very few of them. Perturbing many pixels of the image creates noise more often than removing it, so it is probably worth switching to a finer distance reduction tool, namely, smaller group changes, hence the adaptive behaviour in the step.

To obtain a good set of parameters, we partially benchmarked the algorithm with $(lots, changesPerLot) \in [1...4] \times [1...4]$ (see Appendix E). The general shape of the 16 curves is the same. They reveal two insights. Firstly, for a fixed parameter *lots*, better distance reduction is given by increasing *changesPerLot*. That is, it is beneficial to conserve queries by attempting to do more work on each perturbation. Secondly, for a fixed parameter *changesPerLot*, better distance reduction is given by reducing *lots*. Thus, spending many queries on each iteration in order to find a very good distance-reducing change is wasteful. The message is that greediness is better than caution.

Having seen how the parameters affect the distance reduction, we chose $(lots, changesPerLot) = (1, 25)$ as a suitable contender against the remaining algorithms. The results of its full benchmarking appear below. They look quite promising and improve on the Varsort algorithm.


Performance of algorithm Rand (1, 25) against payloads 50%, 90% and 99%.
100 stego-images were used for each payload, their distances averaged at each query count.
Average distance to mean WAM vector
Queries used
x 10^4
0
5
10
15
20
25
30
35
40
45
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Payload 50
Payload 90
Payload 99

5.6 Quadratic Programming Algorithm

The final approach we present for the problem of feature restoration is the least heuristic one. The idea is to use Quadratic Programming.

A Quadratic Problem (QP) is the problem of assigning values to a vector x of unknowns so that $x^T Hx + f^T x$ is minimised, and x satisfies $lb \leqq x \leqq ub$, $Ax \leqq b$, $A_1 x = b_1$, where H, A, A_1 are known matrices and f, lb, ub, b, b_1 are known vectors. An Integer Quadratic Problem (IQP) has the same form, with the additional restriction that x must be an integer vector. Finally, a Binary Integer Quadratic Problem (BIQP) is an IQP where the constraint $lb \leqq x \leqq ub$ is replaced by the restriction that x must be a binary vector.

We begin by showing how to express feature restoration as a BIQP. Throughout, we assume linearity of the features - the effect of performing many changes is the sum of the effects of performing each change on its own. Let the WAM features of the stego-image are *stego*. Suppose we have chosen a set of single-pixel changes C_1 to C_n, the set of all admissible single-pixel changes on the image. Change C_i alters the feature vector by δ_i. Let also x be a binary vector of length n, so that x_i specifies whether change C_i is performed or not. Then the features of the restored image are:

$restored = stego + \sum_{i=1}^{n} x_i \delta_i = stego + Vx$, where

$V = \left[\delta_1 \middle| \cdots \middle| \delta_n\right]$ is the $27 \times n$ matrix with columns δ_1 to δ_n. If S is the 27×27 covariance matrix of the WAM features, the Mahalanobis distance between the mean WAM vector μ and *restored* is:

$\| restored \|_M^2 = \| stego + Vx \|_M^2 =$

$(stego + Vx - \mu)^T S^{-1} (stego + Vx - \mu) =$

$(stego - \mu)^T S^{-1} (stego - \mu) + (Vx)^T S^{-1} (Vx) - 2 (stego - \mu)^T S^{-1} (Vx) =$

$x^T Hx + f^T x + c$, where

$H = V^T S^{-1} V$, $f = 2V^T S^{-1} (stego - \mu)$, and $c = (stego - \mu)^T S^{-1} (stego - \mu)$

In feature restoration, the goal is to choose the values of x, subject to $x_i \in \{0,1\}$, so that $\|$ *restored* $\|_M$ is minimised. Additionally, we wish to specify that some changes should not be performed together. For example, if a pixel (x,y) may be perturbed by -2, -1, 1 or 2, our set of changes will include $C_i = (x,y,-2)$, $C_j = (x,y,-1)$, $C_k = (x,y,1)$, $C_t = (x,y,2)$, but at most one of these changes may be performed. This is easily specified with linear constraints: $x_i + x_j + x_k + x_t \leqq 1$. The resulting problem is a BIQP.

It must be stressed that BIQP is very difficult to solve, completely infeasible for large-scale instances with lots of variables and linear constraints. Expressing feature restoration as a BIQP serves to give us a sense of its difficulty, and motivates focusing on heuristics and approximations, instead of ambitiously attempting to solve the problem optimally. We also reiterate that the problem is cast as a BIQP *assuming full linearity*. Without this simplifying assumption, it is even harder to solve optimally.

Nonetheless, we are now aware that reducing the distance to the mean feature vector is, in effect, minimising a quadratic objective. While it is infeasible to solve a BIQP obtained as above, a natural idea is to repeatedly obtain smaller instances of QPs, using fewer pixels and fewer restrictions. At each step, the feature restoration algorithm will produce a QP and give it to a solver, then recover the solution, and repeat. The difficulty lies is in deciding how many of the restrictions of BIQP to relax. If we create too difficult quadratic programming problems, the solver will not be able to feasibly calculate solutions to them. On the other hand, if we relax too many constraints, we could be losing too much optimality.

The first constraint that we dispose of is integrality of x_i. A number of reasons motivate this decision. Firstly, although there are plenty of freely available Integer Programming packages which minimise a *linear* objective, it is very difficult to find ones capable of minimising a *quadratic* objective. In fact, we only found one package in the public domain which advertised this ability. However, we were disappointed to discover that it did not work correctly. On the other hand, quadratic programming *without* integer restrictions has been well-studied, and there are many, reasonably efficient QP solvers available. In particular, Matlab offers a routine *quadprog* which solves precisely QPs as defined at the start of this section. Therefore, we decided to relax the integrality constraints on the x_i, and replace them with $0 \leqq x_i \leqq 1$. When our feature restoration recovers the output from the QP solver, each x_i will be rounded.

We also decided to omit the linear constraints which specify which changes must not be performed together. This decision was arrived at purely by experimenting with Matlab's *quadprog* routine: adding linear constraints, even small ones such as $x_i + x_j \leqq 1$, drastically increases the time the routine takes to terminate. Therefore, to build each QP, we will select a set of n pixels, then obtain $2n$ changes from them: $C_{2i} = (x_i, y_i, 1)$, $C_{2i+1} = (x_i, y_i, -1)$, but we will **not** add constraints $x_{2i} + x_{2i+1} \leqq 1$. Thus, our algorithm is:

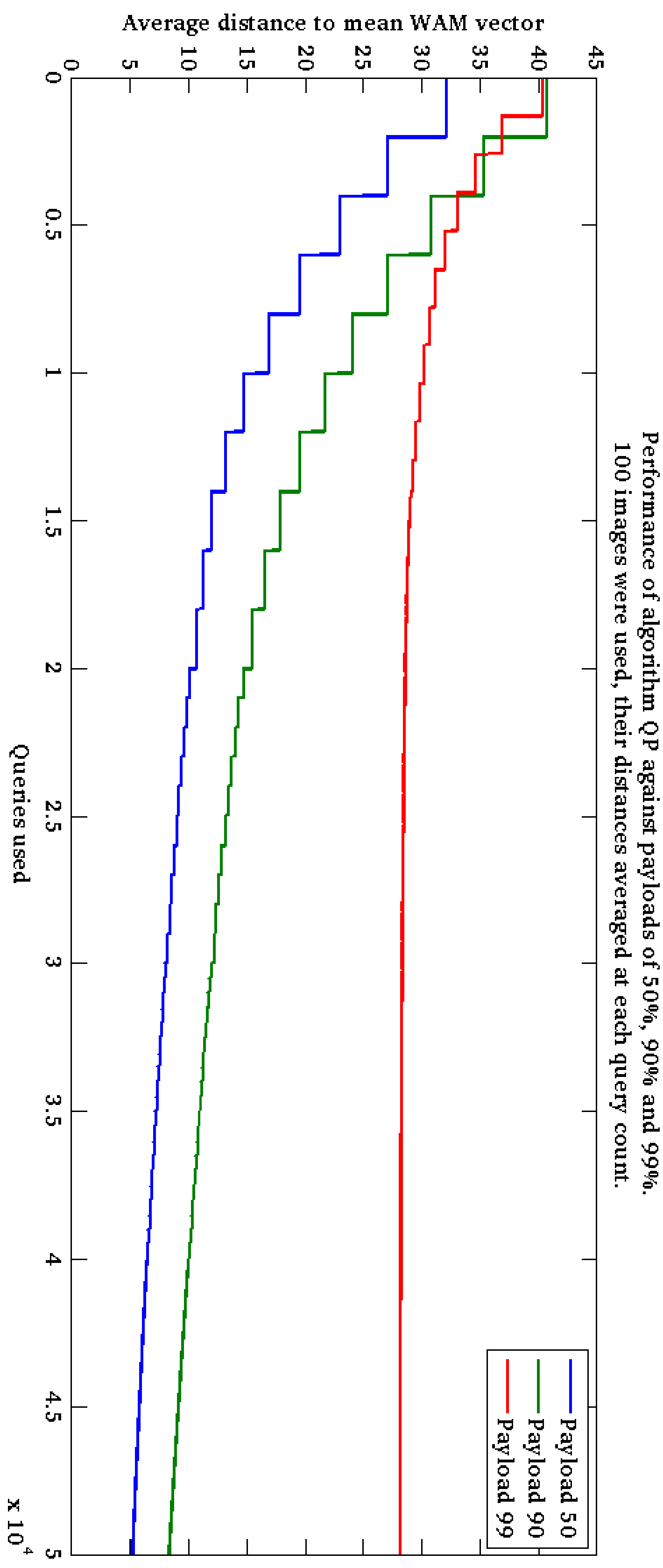
1. Repeat steps 2-8 for as long as queries are available:
2. Choose a value of n. (See below.)
3. Select n non-payload pixels, and the set of $2n$ changes C_1 to C_{2n} - one for perturbing each pixel by 1, and one for -1.
4. Use the WAM oracle to calculate δ_1 to δ_{2n}, and set them as the columns of a matrix V.
5. Obtain the current wam features of the image: $features$.
6. Create the QP: minimise $x^T H x + f^T x$, where $H = V^T S^{-1} V$ and $f = 2V^T S^{-1}(features - \mu)$.
7. Give the QP to a quadratic programming solver, wait for it to finish, and recover its output x.
8. For each $x_i \geqq 0.5$, perform change C_i.

The only remaining question is how to specify n: the number of pixels used to create each QP. We experimented with some values. Past 1000, the solver needs too much time to solve each instance. Moreover, for a fixed value of n, we found that the QPs at the start of the algorithm are solved very quickly, whereas later they begin to take a disproportionately large amount of time. Therefore, we used $n = 1000$ during the first 40% of the queries, then $n = 500$ for the next 20%, $n = 250$ for the next 20%, and $n = 25$ for the final 20% of the queries. Of course, if fewer pixels are available for the restoration, as is the case during the first iterations of the algorithm against payloads of 99%, we just use all of them.

We needed to be careful about measuring the performance of this algorithm, because it includes a time-expensive component (namely, the QP solving) which could potentially dominate the cost of querying the oracle. With the values we have chosen, the algorithm's timescale is similar to that of the Random algorithm with parameters (1, 25). Therefore, it is fair to benchmark the QP algorithm using queries as a unit of time, especially considering that for larger image sizes the cost of queries will dominate the cost of running the QP solver.

A programming challenge was to link the feature restoration code with Matlab's quadratic problem routine. To do so, we wrote a Matlab script which repeatedly checks for the presence of a file, signalling that it should read some input, solve a QP and output. After outputting the solution to the QP, it prints another flag file to signal the feature restoration to read Matlab's output and continue its work. Thus, we effectively used the presence of flag files as a semaphore in order to automate the feature restoration algorithm.

The chart for this algorithm appears below.



Performance of algorithm QP against payloads of 50%, 90% and 99%.
100 images were used, their distances averaged at each query count.
Average distance to mean WAM vector
Queries used
x 10^4
Payload 50
Payload 90
Payload 99

5.7 Comparison and Conclusion

In order to compare our feature restoration algorithms, we overlaid their performance charts for each payload percentage. This produced the **three final comparison charts** - one for 50% payload, one for 90% and one for 99%, all shown below.

At 50% payload, the two best are the Random algorithm with parameters $(1,25)$, and the Variance Sort algorithm. In 50000 queries, they reduce the distance to μ to a point corresponding to no payload at all. However, the Random algorithm's chart is steeper at the start, so it should be preferred if the query allowance is smaller - say, 10000 to 15000.

At 90% payload, the situation is identical. The Random and Variance Sort algorithms outclass the rest, achieving an apparent payload of 0%, with the Random algorithm doing so in fewer queries.

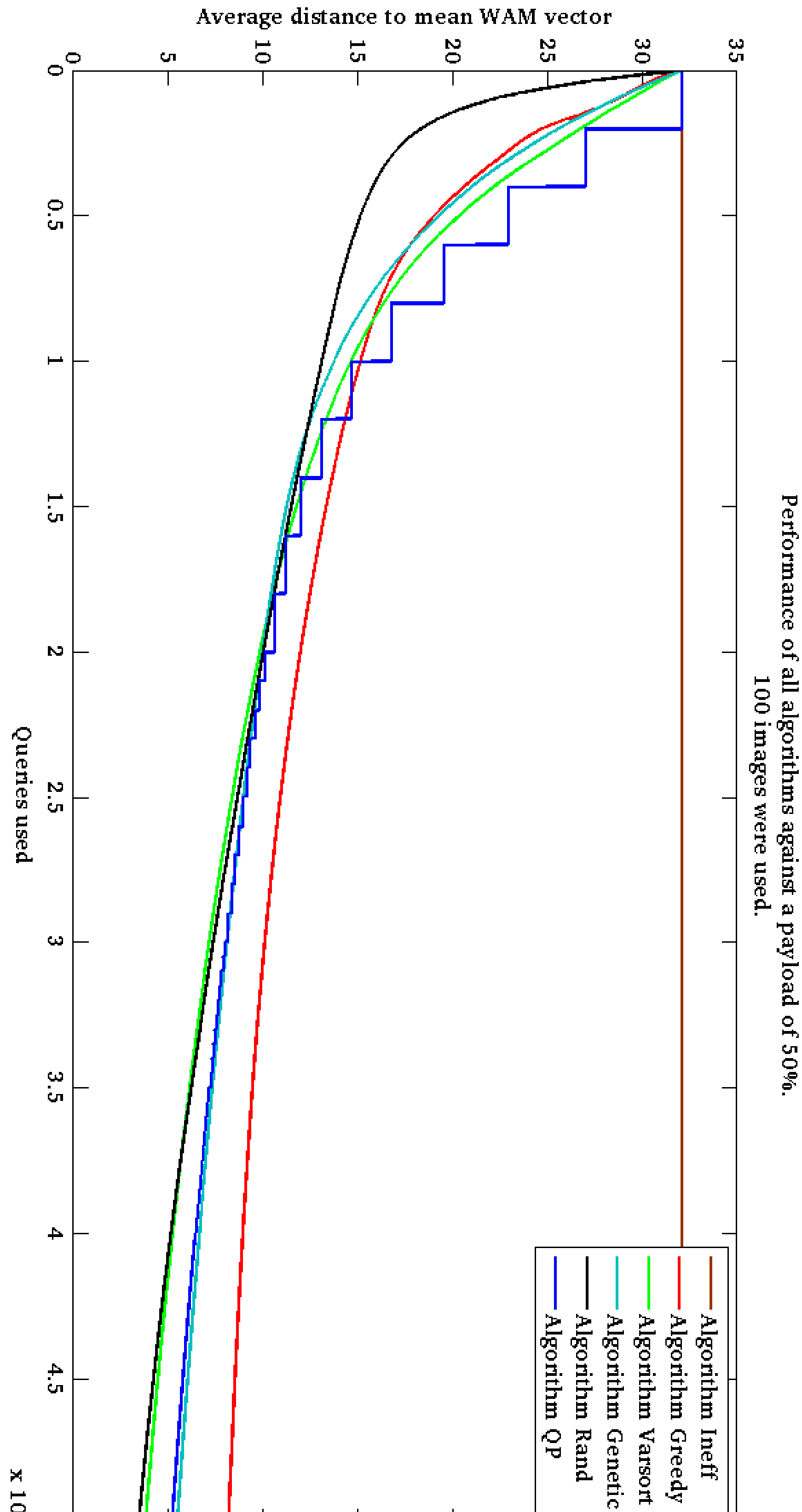
At 99% payload, the Random algorithm performs significantly better than the rest, achieving a greater distance reduction both in the short term and long term. In 50000 queries, it reaches an apparent payload of about 30%. This result is definitely more than we expected to achieve at the start of this project. The Variance Sort, Greedy and Genetic algorithms all achieve an apparent payload of about 40%. Surprisingly, the QP algorithm does not perform particularly well.

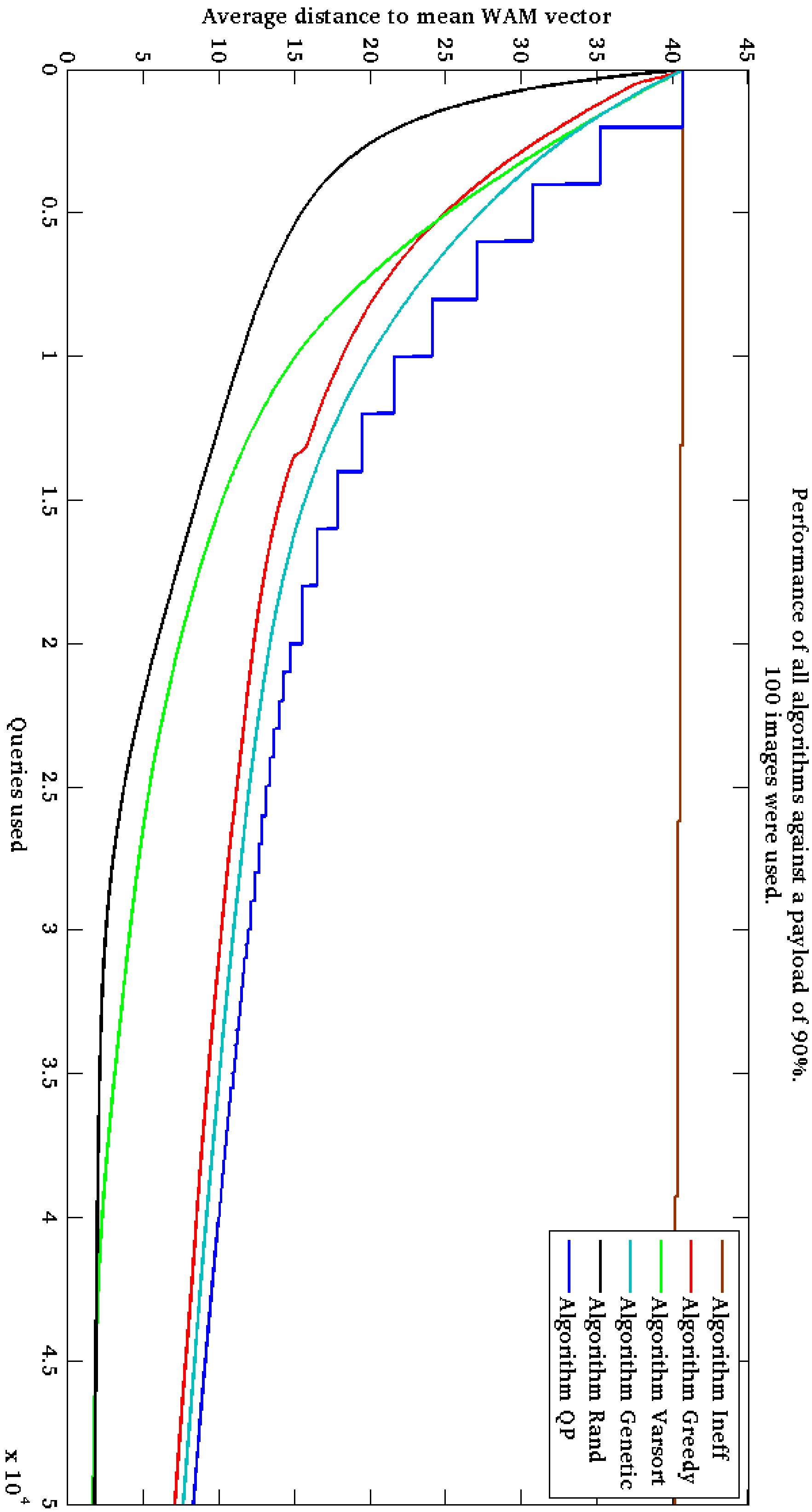
It must be stressed that the Random algorithm's good performance is due to the amount of work it does per query used. For each oracle query, it perturbs up to 25 pixels, whereas the other algorithms typically use queries on single-pixel changes. Initially, we suspected this behaviour might turn out to be very suboptimal. The results have proven us wrong; the project has discovered that the keys to efficient feature restoration are:

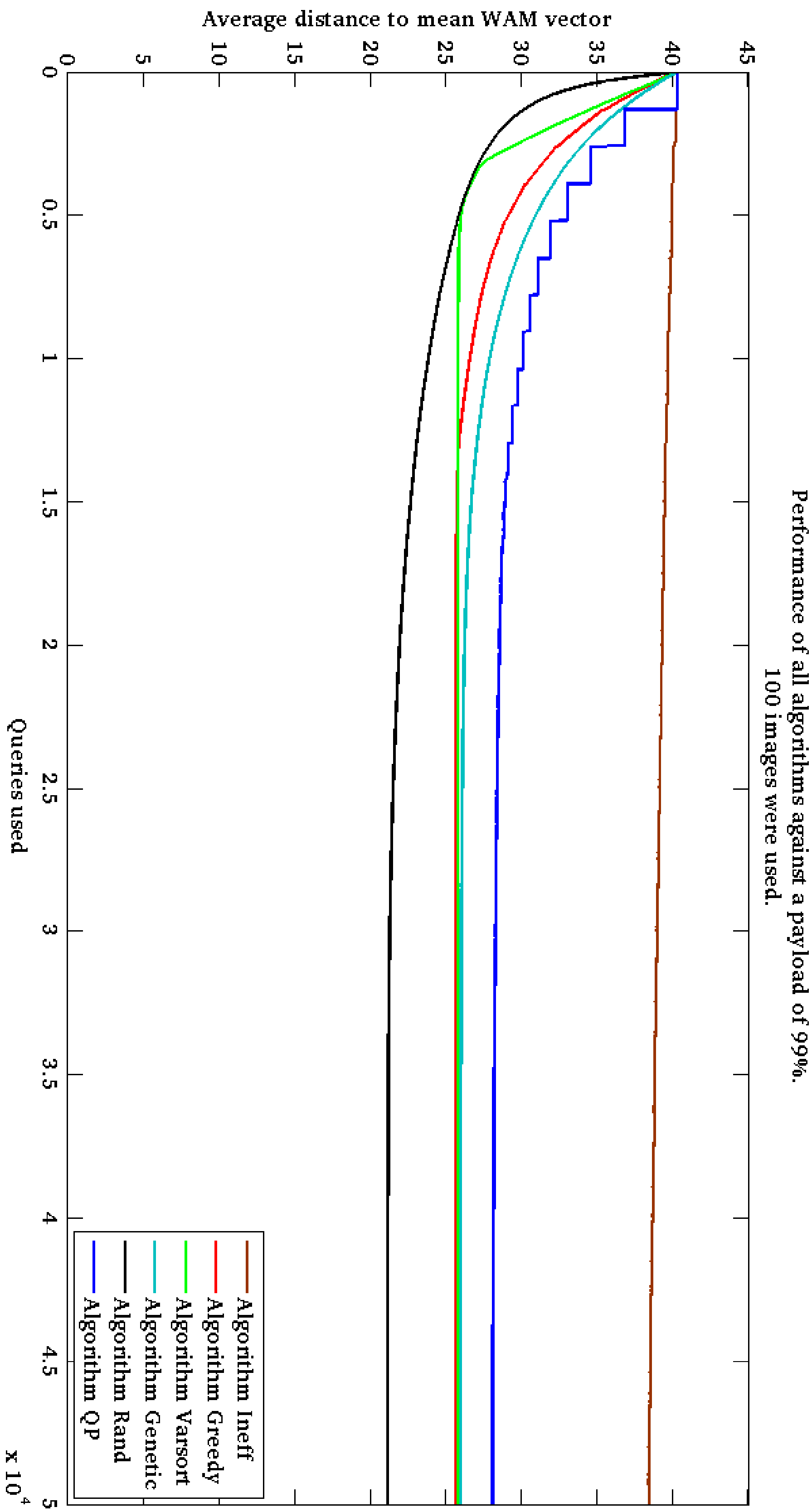
1. *Greediness*. When you spot a beneficial change, perform it, instead of being cautious and looking for even better ones.
2. *Conserving queries*. Attempt to maximise the ratio of changes performed per query used.

Also, our results show that smaller payloads like 50% are more difficult to feature restore than payloads of around 90%. Whilst perhaps counter-intuitive, this is because at 90% the search space of available perturbations is much smaller.

We believe this project was a successful investigation because it brought these principles to light. We implemented an efficient WAM oracle, devised a number of algorithms for feature restoration and obtained encouraging results from them, along with the above insights into the problem, setting the basis for further work.


Performance of all algorithms against a payload of 50%.
100 images were used.
Average distance to mean WAM vector
Queries used
x 10^4
Algorithm Ineff
Algorithm Greedy
Algorithm Varsort
Algorithm Genetic
Algorithm Rand
Algorithm QP


Performance of all algorithms against a payload of 90%.
100 images were used.
Average distance to mean WAM vector
Queries used
x 10^4
0
5
10
15
20
25
30
35
40
45
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Algorithm Ineff
Algorithm Greedy
Algorithm Varsort
Algorithm Genetic
Algorithm Rand
Algorithm QP


Performance of all algorithms against a payload of 99%.
100 images were used.
Average distance to mean WAM vector
Queries used
x 10^4
0
5
10
15
20
25
30
35
40
45
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Algorithm Ineff
Algorithm Greedy
Algorithm Varsort
Algorithm Genetic
Algorithm Rand
Algorithm QP

5.8 Further Work

In order to fit this investigation into the timescale of a 3rd-year project, we had to exclude a number of important questions from its scope. Further work should focus on:

1. Devising feature restoration algorithms which conserve their queries similarly to the Random algorithm with a large second parameter.
2. Testing feature restoration against real steganalyzers to see if it truly makes the stego images less detectable.
3. Experimenting with various payload percentages in order to determine the best proportion of pixels to reserve for feature restoration. We suspect the optimal payload size is close to 90%.
4. Experimenting with various image sizes (this project only worked with 256×256), and image sets with different characteristics (source camera, use of compression, etc.).
5. Using feature restoration on different distance metrics and feature sets: we focused solely on WAM and Mahalanobis's distance, but our feature restoration algorithms treat them as black boxes, making them applicable to other sets of features and measures of distance.

Fortunately, we will be addressing these questions over the summer of 2010 under an EPSRC project titled "Large-Scale Benchmarking of Machine Learning Steganalysis", and hope to answer some of them by September.

Bibliography

[1] Kodovsky, J., and Fridrich, J., "On completeness of feature spaces in blind steganalysis," in *[Proceedings of the 10th ACM Multimedia & Security Workshop]* (2008)

[2] Goljan, M., Fridrich, J., and Holotyak, T., "New blind steganalysis and its implications," in *[Security, Steganography and Watermarking of Multimedia Contents VIII], Proc. SPIE* **6072**, 0101–0113 (2006).

[3] Ker, A., and Lubenko, I., "Feature reduction and payload location with WAM steganalysis," in *[Electronic Imaging, Media Forensics and Security XI], Proc. SPIE* **6072**, 0A01–0A13 (2009).

[4] Avcıbaş, I., Memon, N., and Sankur, B., "Steganalysis using image quality metrics," in *[Electronic Imaging, Security and Watermarking of Multimedia Contents II], Proc. SPIE* **4314**, 523–531, (2001).

[5] Farid, H., and Lyu, S.,: "Detecting hidden messages using higher-order statistics and support vector machines," in *[5th International Workshop on Information Hiding], LNCS* **2578**, 340–354, (2002).

[6] Lyu, S. and Farid, H., "Steganalysis using color wavelet statistics and one-class support vector machines," in *[Security, Steganography, and Watermarking of Multimedia Contents VI], Proc. SPIE* **5306**, 35–45 (2004).

[7] Harmsen, J., and Pearlman, W., "Steganalysis of additive noise modelable information hiding," in *[Proc. SPIE Electronic Imaging, Security, Steganography, and Watermarking of Multimedia Contents V]*, 131–142 (2003).

[8] Ker, A., "Steganalysis of LSB matching in grayscale images," in *[IEEE Signal Processing Letters]* **12**(6), 441–444 (2005).

[9] Xuan, G., Shi, Y., Gao, J., Zou, D., Yang, C., Zhang, Z., Chai, P., Chen, C., and Chen, W., "Steganalysis based on multiple features formed by statistical moments of wavelet characteristic functions," in *[Proc. 7th Information Hiding Workshop], Springer LNCS* **3727**, 262–277 (2005).

[10] Holotyak, T., Fridrich, J., and Voloshynovskiy, S., "Blind statistical steganalysis of additive steganography using wavelet higher order statistics," in *[9th IFIP TC-6 TC-11 Conference on Communications and Multimedia Security], Springer LNCS* **3677**, 273–274 (2005).

[11] Mahalanobis, P., "On the generalised distance in statistics," in *[Proc. National Institute of Sciences of India], Vol.* **2**(1) (1936).

[12] Mallat, S., "A theory for multiresolution signal decomposition: the wavelet representation," in *[IEEE Transactions on Pattern Analysis and Machine Intelligence Vol* ***2****]* (1989).

Appendix A

Code for LSB-matching

```
// LSBEncoder.cpp
// A class which takes care of encoding.
#include "./perms.cpp"
#include "./bitconversion.cpp"
class LSBEncoder
{
public:
LSBEncoder() { }
// Public routine. It embeds a given message into an image specified by its name using a given key.
// The parameters also specify the name of the stego image and the name of the file to be used as a
// log of the encoding.
void messageIntoPgm(string message, string imageName, string newName, string logFile, int key, matrixInt &usedPixels, matrixInt &changesMade)
  {
  matrixDouble image;
  pgmToMatrix(imageName, image);  // Read the image.
  messageIntoMatrix(message, image, key, usedPixels, changesMade);  // Put a message into it.
  matrixToPgm(image, newName);  // Write the image back.
  // The log contains information about which pixels contain and the payload, and how much
  // each pixel has been changed: -1 or +1 for payload pixels, 0 for non-payload.
  FILE *fout = fopen(logFile.c_str(), "w");
  FECHO(fout, "%s\n", imageName.c_str());
  FECHO(fout, "%s\n", newName.c_str());
  writeMatrix(fout, usedPixels);
  writeMatrix(fout, changesMade);
  fclose(fout);
  }
private:
// Encoding procedure. Converts the message into a bit string, initialises the RNG with the given key
// and uses Knuth shuffle to generate a random permutation of the pixels of the image. Then it
// traverses the pixels in the order specified by the permutation, and embeds each bit of the message
// into a pixel as specified by the algorithm for LSB matching.
void messageIntoMatrix(string message, matrixDouble &image, int key, matrixInt &usedPixels, matrixInt &changesMade)
  {
  int N, M;
  int i, j;
```

```
  vecBool bitMessage;
  vecInt perm;
  bitMessage = messageToBits(message);  // Obtain a bitstring.
  N = image.size();
  M = image[0].size();
  // Prepare matrices.
  usedPixels.clear();
  changesMade.clear();
  usedPixels.resize(N);
  changesMade.resize(N);
  for(i = 0; i < N; i++) { usedPixels[i].resize(M); changesMade[i].resize(M); }
  for(i = 0; i < N; i++)
    for(j = 0; j < M; j++)
      {
      usedPixels[i][j] = 0;
      changesMade[i][j] = 0;
      }
  srand(key); // Initialise the RNG with key.
  randomPermutation(N*M, perm); // Get a random permutation.
  // Do the encoding.
  for(i = 0; i < bitMessage.size(); i++)
    {
    int pixel = perm[i];
    int pixelX = pixel / M;
    int pixelY = pixel % M;
    int pixelValue = (int) image[pixelX][pixelY];
    int bit = bitMessage[i];
    int change = 0;
    usedPixels[pixelX][pixelY] = 1;
    if(LSB(pixelValue) == bit) continue;
    else
      {
      if(pixelValue == 255) change = -1;
      else if(pixelValue == 0) change = 1;
      else if(rand() % 2 == 0) change = 1;
      else change = -1;
      }
    image[pixelX][pixelY] = pixelValue + change;
    changesMade[pixelX][pixelY] = change;
    }
  }
};
// LSBDecoder.cpp
// A class which performs decoding.
#include "./perms.cpp"
#include "./bitconversion.cpp"
class LSBDecoder
{
public:
LSBDecoder() { }
// A public routine with two versions. Given the name of a pgm file and a key, extract
    the message
// embedded into it using LSB matching. Optionally, a message length may be provided, if
    not all
// pixels hold payload.
string messageFromPgm(string fileName, int key, int expectedBitLength)
  {
  matrixDouble image;
  pgmToMatrix(fileName, image);
  return messageFromMatrix(image, key, expectedBitLength);
  }
string messageFromPgm(string fileName, int key)
  {
  matrixDouble image;
  pgmToMatrix(fileName, image);
  return messageFromMatrix(image, key);
```

```
  }
private:
// Decoding procedure. Initialises the RNG with key, generates a permutation of the pixels,
// traverses them in the order specified by the permutation, and collects the LSBs of the
// traversed pixels. Then it lumps them into groups of 8, thus getting a byte message.
// expectedLength is there to tell us when to stop the traversal. Obviously there will be
// pixels into which no information has been embedded, so there's no sense in looking
// at their LSBs. If this parameter is missing, then keep going until all pixels are traversed.
string messageFromMatrix(matrixDouble &image, int key, int expectedBitLength)
  {
  int N, M;
  int i, j;
  vecBool bitString;
  vecInt perm;
  N = image.size();
  M = image[0].size();
  if(expectedBitLength > N*M) expectedBitLength = N*M;
  srand(key);
  randomPermutation(N*M, perm);
  for(i = 0; i < expectedBitLength; i++)
    {
    int pixel = perm[i];
    int pixelX = pixel / M;
    int pixelY = pixel % M;
    int pixelValue = (int) image[pixelX][pixelY];
    bitString.push_back(LSB(pixelValue));
    }
  return bitsToMessage(bitString);
  }
string messageFromMatrix(matrixDouble &image, int key)
  { return messageFromMatrix(image, key, image.size()*image[0].size()); }
};
// perms.cpp
// An invokation of an STL routine for obtaining a random permutation.
void randomPermutation(int N, vecInt &ans)
{
int i, j;
ans.clear();
ans.resize(N);
for(i = 0; i < N; i++) ans[i] = i;
random_shuffle(ans.begin(), ans.end());
}
// bitconversion.cpp
// Two routines to convert between bit strings and byte strings.
// Take a bit-string and convert it to a regular string by grouping bits into
// groups of 8. <bits> is assumed to have a length which is divisible by 8.
string bitsToMessage(vecBool bits)
{
int letters = bits.size() / 8;
int i, j;
string ans = "";
for(i = 0; i < letters; i++)
  {
  int c = 0;
  for(j = 0; j < 8; j++) c += bits[8*i+j]*(1 << (7-j));
  c -= 128;
  ans.append(1, (char)c);
  }
return ans;
}
// Take a string and break each character into its bit representation.
vecBool messageToBits(string message)
{
vecBool ans;
```

```
int i, j;
for(i = 0; i < message.length(); i++)
    {
    int c = message[i];
  c += 128;
    vecBool tmp;
    for(j = 0; j < 8; j++)
        {
        tmp.push_back(c % 2);
        c /= 2;
        }
    // tmp is the reversed bit-representation of message[i]. Append it in reverse to ans.
    for(j = 7; j >= 0; j--) ans.push_back(tmp[j]);
    }
return ans;
}
```

Appendix B

Code for WAM

```
// DWTCalculation.cpp
// A class offering routines for the 2D DWT.

#define maxDim 2048
#define maxFilterLength 16

class DWTCalculation
{
/*
extractRow(MX, i) means x becomes row i of MX.
extractColumn(MX, j) means x becomes column j of MX.
putRow(MX, i) means row i of MX becomes x.
putColumn(MX, j) means column j of MX becomes x.
*/
#define extractRow(MX, i) { for(j = 0; j < M; j++) x[j] = (MX)[(i)][j]; xLength = M; }
#define extractColumn(MX, j) { for(i = 0; i < N; i++) x[i] = (MX)[i][(j)]; xLength = N; }
#define putRow(v, MX, i) { for(j = 0; j < M; j++) (MX)[(i)][j] = (v)[j]; }
#define putColumn(v, MX, j) { for(i = 0; i < N; i++) (MX)[i][(j)] = (v)[i]; }

// These arrays are used for convenience, as temporary storage of the input and output
// of the filtering routine. x is the input to doFiltering(). yLow and yHigh are its
// output. lowFilter and highFilter are the two filters.
double yLow[maxDim], yHigh[maxDim];
double lowFilter[maxFilterLength], highFilter[maxFilterLength];
double x[maxDim];
int xLength;
int filterLength;

public:
DWTCalculation()
  {
  // Set the two filters to be the Daubechies 8-tap QMF.
  int i;
  vecDouble filter;
  filter.resize(8);
  filter[0] = 0.230377813309;
  filter[1] = 0.714846570553;
  filter[2] = 0.630880767930;
  filter[3] = -0.027983769417;
  filter[4] = -0.187034811719;
  filter[5] = 0.030841381836;
  filter[6] = 0.032883011667;
  filter[7] = -0.010597401785;
  filterLength = filter.size();
```

```
  for(i = 0; i < filterLength; i++) lowFilter[i] = filter[filterLength-i-1];
  for(i = 0; i < filterLength; i++) highFilter[i] = filter[i];
  for(i = 0; i < filterLength; i += 2) highFilter[i] = -highFilter[i];
  }

// DWT procedure. Its input is the image. The filter is fixed to be 8-tap Daubechies.
// The matrices L and H store the intermediate results of the DWT calculation - the
// results of row-wise filtering. These are then filtered column-wise to
// produce the four output matrices LL, LH, HL and HH. The procedure's signature is
// STL-oriented, in sync with the rest of my code. However, the inner workings favour
// arrays over vectors. This is intentional and aims to improve efficiency, as array
// access is somewhat faster than vector access.
public:
void oneLevelOf2DDWT(matrixDouble &image, matrixDouble &L, matrixDouble &H, matrixDouble &LL, matrixDouble &LH, matrixDouble &HL, matrixDouble &HH, bool downsample)
    {
    register int i, j;

    int N, M;   // Dimensions of the image.
    N = image.size();
    M = image[0].size();
    // Specify dimensions of LL, LH, HL, HH to be NxM. Later we'll downsample,
    // reducing each dimension by half.
    resizeMatrix(HH, N, M)
    resizeMatrix(HL, N, M)
    resizeMatrix(LH, N, M)
    resizeMatrix(LL, N, M)
    resizeMatrix(L, N, M)
    resizeMatrix(H, N, M)
    // Filter each row. Store the low-frequency output in LL, and the high-frequency output in HH.
    for(i = 0; i < N; i++)
      {
      extractRow(image, i)
      doFiltering();
      putRow(yLow, L, i)
      putRow(yHigh, H, i)
      }
    for(j = 0; j < M; j++)
      {
      extractColumn(L, j)
      doFiltering();
      putColumn(yLow, LL, j)
      putColumn(yHigh, LH, j)
      extractColumn(H, j)
      doFiltering();
      putColumn(yLow, HL, j)
      putColumn(yHigh, HH, j)
      }
    if(downsample)
      {
      // Now downsample rows by discarding every other.
      N /= 2;
      for(i = 0; i < N; i++)
        {
        LL[i] = LL[i+i];
        LH[i] = LH[i+i];
        HL[i] = HL[i+i];
        HH[i] = HH[i+i];
        }
      // Now downsample columns by discarding every other.
      M /= 2;
      for(i = 0; i < N; i++)
        for(j = 0; j < M; j++)
          {
          LL[i][j] = LL[i][j+j];
```

```
            LH[i][j] = LH[i][j+j];
            HL[i][j] = HL[i][j+j];
            HH[i][j] = HH[i][j+j];
            }
        // Now set dimensions of LL, LH, HL and HH to be (M/2)x(N/2).
        resizeMatrix(HH, N, M)
        resizeMatrix(HL, N, M)
        resizeMatrix(LH, N, M)
        resizeMatrix(LL, N, M)
        }
  }

// A lazy recalculation of the DWT. If the image is changed at (X, Y), and L, H, LL,
// LH, HL, HH are the DWT results prior to the change, do the least amount of work
// to update the results to reflect the changed image.
public:
void recalculateDWT(
matrixDouble &image,
matrixDouble &L,
matrixDouble &H,
matrixDouble &LL,
matrixDouble &LH,
matrixDouble &HL,
matrixDouble &HH,
int X, int Y)
  {
  int c, i, j;
  int N, M;
  N = image.size();
  M = image[0].size();
  extractRow(image, X)
  doFiltering();
  putRow(yLow, L, X)
  putRow(yHigh, H, X)
  for(j = Y - 7; j <= Y; j++)
    {
    if(j >= 0) c = j; else c = j + M;
    extractColumn(L, c)
    doFiltering();
    putColumn(yLow, LL, c)
    putColumn(yHigh, LH, c)
    extractColumn(H, c)
    doFiltering();
    putColumn(yLow, HL, c)
    putColumn(yHigh, HH, c)
    }
  }
// The filtering routine.
private:
void doFiltering()
  {
  register int i, j, k;
  // Convolve.
  double sum0, sum1;
  for(i = 0; i < xLength; i++)
    {
    sum0 = sum1 = 0;
    for(j = 0; j < filterLength; j++)
      {
      k = i + j; if(k >= xLength) k -= xLength; // Note the periodicity extension.
      sum0 += lowFilter[filterLength - j - 1]*x[k];
      sum1 += highFilter[filterLength - j - 1]*x[k];
      }
    yLow[i] = sum0;
    yHigh[i] = sum1;
    }
```

```
  }
};
#undef putRow
#undef putColumn
#undef extractRow
#undef extractColumn
#undef maxDim
#undef maxFilterLength


// WAMCalculation.cpp.
// A stateless class capturing the calculation of the WAM features.

class WAMCalculation
{
private:
matrixDouble prefixedSquares;

public:
WAMCalculation() { }

vecDouble calculateWAM(
  matrixDouble &image,
  matrixDouble &L,
  matrixDouble &H,
  matrixDouble &LL,
  matrixDouble &LH,
  matrixDouble &HL,
  matrixDouble &HH,
  matrixDouble &variancesH,
  matrixDouble &variancesV,
  matrixDouble &variancesD,
  matrixDouble &residualsH,
  matrixDouble &residualsV,
  matrixDouble &residualsD,
  vecDouble &momentsH,
  vecDouble &momentsV,
  vecDouble &momentsD
    )
  {
  vecDouble features;

  // Dimensions of the image.
  int N = image.size();
  int M = image[0].size();

  // Calculate the DWT.
  DWTCalculation DWT = DWTCalculation();
  DWT.oneLevelOf2DDWT(image, L, H, LL, LH, HL, HH, false);

  // Calculate the 9 moments from H.
    computeVariances(LH, variancesH);
    computeResiduals(LH, variancesH, residualsH);
    computeMoments(residualsH, momentsH);

  // Calculate the 9 moments from V.
    computeVariances(HL, variancesV);
    computeResiduals(HL, variancesV, residualsV);
    computeMoments(residualsV, momentsV);

  // Calculate the 9 moments from D.
    computeVariances(HH, variancesD);
    computeResiduals(HH, variancesD, residualsD);
  computeMoments(residualsD, momentsD);

  // Assemble the 27-d feature vector.
```

```
  features.insert(features.end(), momentsH.begin(), momentsH.end());
  features.insert(features.end(), momentsV.begin(), momentsV.end());
  features.insert(features.end(), momentsD.begin(), momentsD.end());

  // Return it.
  return features;
  }

// The calculation of local variances. Given a matrix of dwt coefficients,
// estimate the local variance of the submatrix (fromX, fromY) - (toX, toY).
void computeVariances(matrixDouble &dwt, matrixDouble &var, int fromX, int toX, int fromY, int toY)
  {
  int N = dwt.size();
  int M = dwt[0].size();
  int i, j, r, c;
  for(i = fromX; i < toX; i++)
    for(j = fromY; j < toY; j++)
      {
      r = (i + N) % N;
      c = (j + M) % M;
      var[r][c] = computeOneVariance(r, c);
      }
  }
// Routine to compute the variances of the whole dwt matrix.
void computeVariances(matrixDouble &dwt, matrixDouble &var)
  {
  int N = dwt.size();
  int M = dwt[0].size();
  resizeMatrix(var, N, M);
    transform(dwt);
  computeVariances(dwt, var, 0, N, 0, M);
  }

// Each coefficient's local variance is estimated separately, based on four
// windows centred around it - of sizes 3, 5, 7 and 9.
double computeOneVariance(int x, int y)
  {
  double v[6] = {0, 0, 0, 0, 0};
  int d;
  int X, Y, F, G;
  int N = prefixedSquares.size();
  int M = prefixedSquares[0].size();
  v[1] = v[2] = v[3] = v[4] = 0;


  for(d = 1; d <= 4; d++) // For each window size 2*d+1:
    {
    // The window is (X, Y) - (F, G).
    X = x - d;
    Y = y - d;
    F = x + d;
    G = y + d;

    // Crop window at edges.
    if(X < 0) X = 0;
    if(Y < 0) Y = 0;
    if(F >= N) F = N-1;
    if(G >= M) G = M-1;

    // Use prefixedSquares to get the numerator.
    if(!X && !Y) v[d] = prefixedSquares[F][G];
    else if(!X) v[d] = prefixedSquares[F][G] - prefixedSquares[F][Y-1];
    else if(!Y) v[d] = prefixedSquares[F][G] - prefixedSquares[X-1][G];
    else v[d] = prefixedSquares[F][G] - prefixedSquares[F][Y-1] - prefixedSquares[X-1][G] + prefixedSquares[X-1][Y-1];
```

```

    #ifdef WAM_fix_denominator
    // Use a fixed denominator. We used this.
    v[d] /= (2*d + 1)*(2*d + 1);
    #else
    // Use a non-fixed denominator.
    v[d] /= (F-X+1)*(G-Y+1);
    #endif
    }

  // Find the least v[d].
  double best = v[1];
  for(d = 2; d <= 4; d++) best = MIN(best, v[d]);

  // Subtract a half, clamp to 0.
  best -= 0.5;
  if(best < 0) best = 0;

  // Return.
  return best;
  }

// The calculation of all residuals, given the dwt and variance matrices. For a
// coefficient x with local variance estimate v, the residual is 0.5*x / (0.5+v).
void computeResiduals(matrixDouble &dwt, matrixDouble &var, matrixDouble &res)
  {
  int N = dwt.size();
  int M = dwt[0].size();
  resizeMatrix(res, N, M);
  computeResiduals(dwt, var, res, 0, N, 0, M);
  }

// The calculation of only some residuals.
// Those appearing in the window (fromX, fromY) - (toX, toY).
void computeResiduals(matrixDouble &dwt, matrixDouble &var, matrixDouble &res, int fromX,
     int toX, int fromY, int toY)
  {
    int i, j, r, c;
  int N = dwt.size();
  int M = dwt[0].size();
  for(i = fromX; i < toX; i++)
    for(j = fromY; j < toY; j++)
      {
      r = (i + N) % N;
      c = (j + M) % M;
      res[r][c] = 0.5*dwt[r][c] / (0.5 + var[r][c]);
      }
  }

// The calculation of moments, given the residuals. The m-th moment is defined by
// (sum(i, j) of |residuals[i][j] - average|^m) / (M*N), where average is the
// mean value of the matrix residuals. We're interested in the first nine moments.
void computeMoments(matrixDouble &res, vecDouble &moments)
  {
  int N = res.size();
  int M = res[0].size();
  int i, j, m;
  double average = 0;

  // Initialise.
  moments.resize(9);
  for(m = 0; m < 9; m++) moments[m] = 0;

  // Find average residual.
  for(i = 0; i < N; i++)
    for(j = 0; j < M; j++)
```

```
        average += res[i][j];
    average /= (N*M);

    // Do the moments.
    for(i = 0; i < N; i++)
      for(j = 0; j < M; j++)
        {
        double R = res[i][j] - average;
        R = ABS(R);

        double v = R;
        for(m = 0; m < 9; m++)
          {
          moments[m] += v;
          v *= R;
          }
        }
    for(m = 0; m < 9; m++) moments[m] /= (M*N);
    }

// Routine which computes the prefixed squares form of mx and
// puts it into prefixedSquares. Useful for variances.
void transform(matrixDouble &mx)
    {
    int i, j;
    int N = mx.size();
    int M = mx[0].size();
    resizeMatrix(prefixedSquares, N, M);

      for(i = 0; i < N; i++)
          for(j = 0; j < M; j++)
              prefixedSquares[i][j] = mx[i][j]*mx[i][j];

      for(i = 0; i < N; i++)
          for(j = 1; j < M; j++)
              prefixedSquares[i][j] += prefixedSquares[i][j-1];

      for(j = 0; j < M; j++)
          for(i = 1; i < N; i++)
              prefixedSquares[i][j] += prefixedSquares[i-1][j];
      }
};


// WAMUpdater.cpp
// This is the most important class of the WAM pipeline. It holds an image, and
// all the information from its WAM calculation. It provides an interface to modify
// the image whilst keeping the WAM information in sync with it. It also offers
// an interface for hypothetical queries: "I don't want to change this image
// in this way yet, but if I did, what would the effect be?"

class WAMUpdater
{

private:

matrixDouble image; // The image.
matrixDouble L, H, LL, LH, HL, HH; // The results from the DWT calculation.
matrixDouble variancesH, variancesV, variancesD; // The variance matrices.
matrixDouble residualsH, residualsV, residualsD; // The residuals matrices.
vecDouble momentsH, momentsV, momentsD; // The three sets of moments.
int queriesUsed; // A counter of many queries have been answered so far.

// References to the two above classes.
WAMCalculation W;
DWTCalculation DWT;
```

```
public:
// Obvious access methods.
int getN() { return image.size(); }
int getM() { return image[0].size(); }
int getQueriesUsed() { return queriesUsed; }
int getPixelValueAt(int x, int y) { return (int) image[x][y]; }

// To get the WAM vector, concatenate the 3 sets of moments.
vecDouble getWAM()
  {
  vecDouble features;
    features.insert(features.end(), momentsH.begin(), momentsH.end());
    features.insert(features.end(), momentsV.begin(), momentsV.end());
    features.insert(features.end(), momentsD.begin(), momentsD.end());
  return features;
  }

// Null constructor.
WAMUpdater() { }

/*
Initialisation. Given a file containing an image,
do a first-time WAM calculation to initialise all
the info.
*/
WAMUpdater(string imageFile)
    {
  // Read in.
  pgmToMatrix(imageFile, image);
  int N, M;
  N = image.size();
  M = image[0].size();

  // Initialise query counter.
  queriesUsed = 0;

  // Set up sizes.
  resizeMatrix(variancesH, N, M)
  resizeMatrix(variancesV, N, M)
  resizeMatrix(variancesD, N, M)
  resizeMatrix(residualsH, N, M)
  resizeMatrix(residualsV, N, M)
  resizeMatrix(residualsD, N, M)
  momentsH.resize(9);
  momentsV.resize(9);
  momentsD.resize(9);

  // Calculate everything.
  W.calculateWAM(
  image,
  L, H, LL, LH, HL, HH,
  variancesH,
  variancesV,
  variancesD,
  residualsH,
  residualsV,
  residualsD,
  momentsH,
  momentsV,
  momentsD);
    }

// Check method:
// Can I perturb pixel (x, y) by v?
bool canMakeOneChange(int x, int y, int v)
```

```
  {
  return (x >= 0 &&
  x < image.size() &&
  y >= 0 &&
  y < image[0].size() &&
  (int)image[x][y] + v >= 0 &&
  (int)image[x][y] + v <= 255);
  }
bool canMakeOneChange(Change c)
{ return canMakeOneChange(c.x, c.y, c.v); }

/*
Check method for group changes.
We're relying on no duplicate pixels
appearing in the group change. In the rest
of our code, we arrange this to be so.
*/
bool canMakeGroupChange(vector<Change> v)
  {
  for(int i = 0; i < v.size(); i++) if(!canMakeOneChange(v[i])) return false;
  return true;
  }

// Query routine.
vecDouble QueryOneChange(Change C)
  {
  // Count the query.
  queriesUsed++;
  // Make the change.
  makeOneChange(C);
  // Remember the features.
  vecDouble features = getWAM();
  // Undo the change.
  undoOneChange(C);
  // Return.
  return features;
  }

// Same but for group queries.
vecDouble QueryGroupChange(vector<Change> &v)
  {
  queriesUsed++;
  makeGroupChange(v);
  vecDouble features = getWAM();
  undoGroupChange(v);
  return features;
  }

/*
The interface to change the image.
Given a singleton change, cast it
to a group change and apply it.
*/
void makeOneChange(Change C)
  {
  vector<Change> v;
  v.push_back(C);
  makeGroupChange(v);
  }

// Routine to apply a group change.
void makeGroupChange(vector<Change> &v)
  {
  int i;
  // Firstly change the image.
  for(i = 0; i < v.size(); i++)
```

```
      {
      image[v[i].x][v[i].y] += v[i].v;
      image[v[i].x][v[i].y] = MIN(image[v[i].x][v[i].y], 255);
      image[v[i].x][v[i].y] = MAX(image[v[i].x][v[i].y], 0);
      // Recalculate the DWT stuff around each changed pixel.
      if(v[i].v != 0)
        recalculateDWT(v[i].x, v[i].y);
      }
    // Recalculate the variances and residuals around the changes v.
    recalculateVariances(v);
    recalculateResiduals(v);
    // Recalculate the moments.
    recalculateMoments();
    }

// Mechanism for undoing changes.
// Yes, I'm aware that it's
// code duplication.
void undoOneChange(Change C)
    {
    vector<Change> v;
    v.push_back(C);
    undoGroupChange(v);
    }

void undoGroupChange(vector<Change> &v)
    {
    int i;
    for(i = 0; i < v.size(); i++)
      {
      image[v[i].x][v[i].y] -= v[i].v;
      image[v[i].x][v[i].y] = MIN(image[v[i].x][v[i].y], 255);
      image[v[i].x][v[i].y] = MAX(image[v[i].x][v[i].y], 0);
      recalculateDWT(v[i].x, v[i].y);
      }
    recalculateVariances(v);
    recalculateResiduals(v);
    recalculateMoments();
    }

private:
// To recalculate the DWT stuff around (x, y),
// just invoke the relevant method in the DWT class.
void recalculateDWT(int x, int y)
    {
    DWT.recalculateDWT(image, L, H, LL, LH, HL, HH, x, y);
    }

// Direct recalculation of moments.
void recalculateMoments()
    {
    W.computeMoments(residualsH, momentsH);
    W.computeMoments(residualsV, momentsV);
    W.computeMoments(residualsD, momentsD);
    }

// Recalculation of variances around group change v.
void recalculateVariances(vector<Change> &v)
    {
    int i, x, y;
    // If big group, recalculate variances using prefixedSquares.
    if(v.size() > 2) recalculateVariancesByTransform(v);
    // If small, just recalculate the dirty variances by walking over the 9x9 region around
        them.
    else recalculateVariancesFromDefinition(v);
    }
```

```

// OK(i, j) means (i, j) is in range.
#define OK(i, j) ( (i) >= 0 && (i) < N && (j) >= 0 && (j) < M )

void recalculateVariancesFromDefinition(vector<Change> v)
  {
  int x0, y0, x1, y1, x, y;
  int i, j, r, c;
  int N = image.size();
  int M = image[0].size();

  for(int ind = 0; ind < v.size(); ind++)
    {
    if(v[ind].v == 0) continue;
    // (x, y) is the changed pixel.
    x = v[ind].x;
    y = v[ind].y;
    // [x0..x1]x[y0..y1] are the affected variances.
    x0 = x - 11;
    y0 = y - 11;
    x1 = x + 4;
    y1 = y + 4;

    // For each dirty variance,
    for(i = x0; i <= x1; i++)
      for(j = y0; j <= y1; j++)
        {
        // recalculate it.
        // Note that because the DWT filtering uses a periodicity
        // extension, we need to wrap around in both directions,
        // because the dirtiness of the DWT coefficients wraps around
        // the image. Behold, we're actually taking pains to emulate
        // the bug in the original WAM code.
        r = (i + 2*N) % N;
        c = (j + 2*M) % M;
        variancesH[r][c] = calculateOneVarianceFromDefinition(LH, r, c);
        variancesV[r][c] = calculateOneVarianceFromDefinition(HL, r, c);
        variancesD[r][c] = calculateOneVarianceFromDefinition(HH, r, c);
        }
    }
  }

// To calculate one dirty variance,
double calculateOneVarianceFromDefinition(matrixDouble &dwt,
int x, int y)
  {
  #ifndef WAM_fix_denominator
  double den[6];
  #endif
  double v[6] = {0, 0, 0, 0, 0};
  int d, dx, dy;
  int N = image.size();
  int M = image[0].size();
  // walk around (x, y) summing things up as necessary.
  for(dx = -4; dx <= 4; dx++)
    for(dy = -4; dy <= 4; dy++)
      if(OK(x + dx, y + dy))
        for(d = 1; d <= 4; d++)
          if(ABS(dx) <= d && ABS(dy) <= d)
            {
            v[d] += SQUARE(dwt[x + dx][y + dy]);
            #ifndef WAM_fix_denominator
            den[d]++;
            #endif
            }
  #ifdef WAM_fix_denominator
```

```
  // Divide by a fixed denominator.
  v[1] /= 9;
  v[2] /= 25;
  v[3] /= 49;
  v[4] /= 81;
  #else
  // Or a variable one. We used fixed.
  for(d = 1; d <= 4; d++) v[d] /= den[d];
  #endif
  // Find the smallest.
  double best, best1, best2;
  best1 = MIN(v[1], v[2]);
  best2 = MIN(v[3], v[4]);
  // Subtract a half and clamp to zero.
  best = MIN(best1, best2) - 0.5;
  if(best < 0) best = 0;
  // Return.
  return best;
  }
#undef OK

// If there are too many variances to update,
void recalculateVariancesByTransform(vector<Change> v)
  {
  int i, x, y;
  // Calculate the prefixed squares form.
  W.transform(LH);
  // Then update the dirty variances.
  for(i = 0; i < v.size(); i++)
    {
    if(v[i].v == 0) continue;
    x = v[i].x; y = v[i].y;
    W.computeVariances(LH, variancesH, x - 11, x + 5, y - 11, y + 5);
    }
  // Same.
  W.transform(HL);
  for(i = 0; i < v.size(); i++)
    {
    if(v[i].v == 0) continue;
    x = v[i].x; y = v[i].y;
    W.computeVariances(HL, variancesV, x - 11, x + 5, y - 11, y + 5);
    }
  // Same.
  W.transform(HH);
  for(i = 0; i < v.size(); i++)
    {
    if(v[i].v == 0) continue;
    x = v[i].x; y = v[i].y;
    W.computeVariances(HH, variancesD, x - 11, x + 5, y - 11, y + 5);
    }
  }

// To recalculate the residuals around a group change v,
void recalculateResiduals(vector<Change> &v)
  {
  int i, x, y;
  for(i = 0; i < v.size(); i++)
    {
    if(v[i].v == 0) continue;
    // Take each x, y in v.
    x = v[i].x;
    y = v[i].y;
    // And redo the residuals in [x-11..x+4]x[y-11..y+4].
    W.computeResiduals(LH, variancesH, residualsH, x - 11, x + 5, y - 11, y + 5);
    W.computeResiduals(HL, variancesV, residualsV, x - 11, x + 5, y - 11, y + 5);
    W.computeResiduals(HH, variancesD, residualsD, x - 11, x + 5, y - 11, y + 5);
```

```
    }
  }

// Output methods.
public:
void outputImage(string fileName)
  { matrixToPgm(image, fileName); }

void outputWAM(string fileName)
  {
  vecDouble features = getWAM();
  writeVector(fileName, features);
  }

void outputAll(string fileName)
    {
    writeMatrix(fileName+".dwt.H", LH);
    writeMatrix(fileName+".dwt.V", HL);
    writeMatrix(fileName+".dwt.D", HH);

    writeMatrix(fileName+".var.H", variancesH);
    writeMatrix(fileName+".var.V", variancesV);
    writeMatrix(fileName+".var.D", variancesD);

    writeMatrix(fileName+".res.H", residualsH);
    writeMatrix(fileName+".res.V", residualsV);
    writeMatrix(fileName+".res.D", residualsD);

    writeVector(fileName+".mom.H", momentsH);
    writeVector(fileName+".mom.V", momentsV);
    writeVector(fileName+".mom.D", momentsD);
    }
};
```

Appendix C

Code for Feature Restoration

```
// Restorer.cpp
// A superclass for our restorers. It promises the apply() routine, which is implemented
    by subclasses.

class Restorer
{
protected:
WAMUpdater W;
vecDouble goal;
matrixInt usedPixels;
matrixInt changesMade;


string originalFile;
string stegoFile;

// A list of all the non-payload pixels available for restoration.
vector<Pixel> availablePixels;

public:

virtual void apply() = 0;

Restorer(string encodingLogFile)
  {
  char x[128];
  int i, j;
  FILE *fin = fopen(encodingLogFile.c_str(), "rb");

  fscanf(fin, "%s", x);
  originalFile = string(x);

  fscanf(fin, "%s", x);
  stegoFile = string(x);


  loadMatrix(fin, usedPixels);
  loadMatrix(fin, changesMade);

  W = WAMUpdater(stegoFile);
  fclose(fin);

  prepareMahalanobis();
  setGoal("means");
```

```

  int N, M; N = W.getN(); M = W.getM();
  // Find all the non-payload pixels, and put them into a vector. This will allow quick
      generation of valid group changes.

  for(i = 0; i < N; i++)
    for(j = 0; j < M; j++)
      if(!usedPixels[i][j])
        availablePixels.push_back(newPixel(i, j));
  availablePixels.resize(availablePixels.size());
  }


/*
Routine to set the goal. The choice is between aiming for the original features, and
    aiming for the mean features. We are going to use the mean features.
*/
void setGoal(string instruction)
  {
  if(instruction == "means")
    { goal = means; }
  else if(instruction == "original")
    {
    WAMUpdater F = WAMUpdater(originalFile);
    goal = F.getWAM();
    }
  else
    { exit(0); }
  }

// RANGE: [a..b)
int randomNumberInRange(int a, int b)
  { return a + (rand() % (b-a)); }

protected:
// Returns a combination of k elements, drawn from [0..m), where m is the number of
    available pixels.

vecInt randomSample(int k)
{
int i, ind;
bool hasDups;
int m = availablePixels.size();

vecInt combination;
combination.resize(k);

do
  {
  for(i = 0; i < k; i++)
    {
    ind = randomNumberInRange(0, m);
    combination[i] = ind;
    }
  sort(combination.begin(), combination.end());
  hasDups = false;
  for(i = 1; i < k && !hasDups; i++)
    if(combination[i] == combination[i-1]) hasDups = true;
  } while(hasDups);


return combination;
}


vector<Change> randomNodupGroupChange(int size, int vmin, int vmax)
```

```
{
int i, ind, v, x, y;
bool hasDups;
vector<Change> groupChange;
vecInt indices;

groupChange.resize(size);
indices.resize(size);

vmax += 1;

do
  {
  indices = randomSample(size);
  for(i = 0; i < size; i++)
    {
    ind = indices[i];
    x = availablePixels[ind].x;
    y = availablePixels[ind].y;
    do
      { v = randomNumberInRange(vmin, vmax); }
    while(ABS(changesMade[x][y] + v) > PIXEL_CHANGE_CAP);

    groupChange[i] = newChange(x, y, v);
    }
  } while (!W.canMakeGroupChange(groupChange));

return groupChange;
}

};

// RestorerIneff.cpp
// The restorer for the inefficient algorithm.

class RestorerIneff : public Restorer
{
private:
int iterations;

public:
RestorerIneff(string encodingLog, int ITERATIONS) : Restorer(encodingLog)
  {
  iterations = ITERATIONS;
  }


void apply()
  {
  int N, M;
  int it;
  int i, j, v;
  Change C, bestChange;
  double dist, bestdist;

  N = W.getN();
  M = W.getM();

  vecDouble cur = W.getWAM();
  bestdist = Distance(goal, cur);
  for(it = 1; it <= iterations; it++)
    {
    ECHO("%d %.12lf\n", W.getQueriesUsed(), bestdist);

    bestChange = newChange(-1, -1, -1);
    for(i = 0; i < N; i++)
```

```
      for(j = 0; j < M; j++)
        if(!usedPixels[i][j])
          for(v = -1; v <= 1; v += 2) // v in {-1, 1}
            {
            C = newChange(i, j, v);

            if(W.canMakeOneChange(C) && ABS(changesMade[i][j] + v) <= PIXEL_CHANGE_CAP)
              {
              vecDouble w = W.QueryOneChange(C);
              dist = Distance(w, goal);
              if(dist < bestdist)
                { bestChange = C; bestdist = dist; }
              }
            }
    if(bestChange.x == -1) break;
    else
      {
      W.makeOneChange(bestChange);
      changesMade[bestChange.x][bestChange.y] += bestChange.v;
      }

    cur = W.getWAM();
    bestdist = Distance(goal, cur);
    }
  ECHO("%d %.12lf\n", W.getQueriesUsed(), bestdist);
  W.outputImage("restored.pgm");
  }
};


// RestorerGreedy.cpp
// The restorer class for the Greedy algorithm.

class RestorerGreedy : public Restorer
{
private:
int queryLimit;

public:
RestorerGreedy(string encodingLog, int QUERIES) : Restorer(encodingLog)
  {
  queryLimit = QUERIES;
  }

void apply()
  {
  int N, M; // Dimensions of the image.
  int m;  // Number of pixels available for restoration.

  // Temporary variables used below.
  int x, y, pixelValue, v;
  double hypDistPlus, hypDistMinus, bestHypDist, curDist;
  vecDouble hypWAMplus, hypWAMminus, bestHypWAM, curWAM;
  int i, j;

  // Dimensions.
  N = W.getN();
  M = W.getM();

  // WAM features of the image and their distance to the goal features.
  curWAM = W.getWAM();
  curDist = Distance(curWAM, goal);

  // Create a list of the pixels available for restoration.
  vector<Pixel> availablePixels;
  for(i = 0; i < N; i++)
```

```
    for(j = 0; j < M; j++)
      if(!usedPixels[i][j])
        availablePixels.push_back(newPixel(i, j));
  m = availablePixels.size();
  availablePixels.resize(m);

  // Iterate while there are queries available.
  while(W.getQueriesUsed() < queryLimit)
    for(i = 0; i < m && W.getQueriesUsed() < queryLimit; i++)
      {
      ECHO("%.12lf␣%d\n", curDist, W.getQueriesUsed());
      // For each non-payload pixel (x, y),
      x = availablePixels[i].x;
      y = availablePixels[i].y;
      pixelValue = W.getPixelValueAt(x, y);

      // Calculate the WAM of the image obtained by perturbing the pixel by 1.
      // Avoid possibility of incrementing pixel outside 8-bit range.
      if(pixelValue != 255 && ABS(changesMade[x][y] + 1) <= PIXEL_CHANGE_CAP)
        {
        hypWAMplus = W.QueryOneChange(newChange(x, y, 1));
        hypDistPlus = Distance(hypWAMplus, goal);
        }
      else hypDistPlus = 1000000;

      // Same for -1.
      // Calculate the hypothetical distances of the two possibilities.
      if(pixelValue != 0 && ABS(changesMade[x][y] - 1) <= PIXEL_CHANGE_CAP)
        {
        hypWAMminus = W.QueryOneChange(newChange(x, y, -1));
        hypDistMinus = Distance(hypWAMminus, goal);
        }
      else hypDistMinus = 1000000;

      // Determine which of these two changes leads to a greater distance reduction.
      if(hypDistPlus < hypDistMinus)
        {
        bestHypDist = hypDistPlus;
        bestHypWAM = hypWAMplus;
        v = 1;
        }
      else
        {
        bestHypDist = hypDistMinus;
        bestHypWAM = hypWAMminus;
        v = -1;
        }

      // Finally, check if this distance reduction is positive. If so, perform the change.
      if(bestHypDist < curDist)
        {
        W.makeOneChange(newChange(x, y, v));
        changesMade[x][y] += v;
        curDist = bestHypDist;
        curWAM = bestHypWAM;
        }
      }
    ECHO("%.12lf␣%d\n", curDist, W.getQueriesUsed());
    W.outputImage("restored.pgm");
    }
};

// RestorerGenetic.cpp
// The restorer for our Genetic algorithm

```

```
struct PopulationIndividual
  {
  GroupChange vc;
  double distanceReduction;
  };

bool operator <(PopulationIndividual a, PopulationIndividual b)
  {
  if(a.distanceReduction < b.distanceReduction) return true;
  else return false;
  }

PopulationIndividual newIndividual(GroupChange gc, double distRed)
  {
  PopulationIndividual ans;
  ans.vc = gc;
  ans.distanceReduction = distRed;
  return ans;
  }

typedef PopulationIndividual Individual;
typedef priority_queue<PopulationIndividual> Population;

class RestorerGenetic : public Restorer
{
private:
Population P; // The population for the genetic algorithm.
int initialSize;  // Parameters of the genetic algorithm.
int queryLimit; // What it says on the tin.

public:
// Constructor.
RestorerGenetic(string encodingLogFile, int initSize, int queries) : Restorer(
    encodingLogFile)
  {
  // Set state space of the genetic algorithm restorer.
  initialSize = initSize;
  queryLimit = queries;
  }

void apply()
  {
  initialiseGeneticAlgorithm();
  iterate();
  dwindle();
  W.outputImage("restored.pgm");
  }

private:
void initialiseGeneticAlgorithm()
  {
  int i, j;
  int N, M;
  N = W.getN(); M = W.getM();

  vecDouble curWAM;
  double curDist;
  curWAM = W.getWAM();
  curDist = Distance(curWAM, goal);
  ECHO("%.12lf %d\n", curDist, W.getQueriesUsed());

  for(i = 0; i < initialSize; i++)
    migrateIn();
  }

void iterate()
```

```
  {
  vecDouble curWAM;
  double curDist;

  curWAM = W.getWAM();
  curDist = Distance(curWAM, goal);
  ECHO("%.12lf␣%d\n", curDist, W.getQueriesUsed());

  while(1)
    {
    if(W.getQueriesUsed() >= queryLimit) break;
    migrateIn();
    if(W.getQueriesUsed() >= queryLimit) break;
    migrateIn();
    if(W.getQueriesUsed() >= queryLimit) break;

    migrateOut();
    curWAM = W.getWAM();
    curDist = Distance(curWAM, goal);
    ECHO("%.12lf␣%d\n", curDist, W.getQueriesUsed());

    if(P.size() == 2*initialSize + initialSize / 10) merge();
    }
  }

void migrateIn()
  {
  vector<Change> groupChange;
  vecDouble hypWAM, curWAM;
  double curDist, hypDist, distRed;

  curWAM = W.getWAM();
  curDist = Distance(curWAM, goal);
  while(1)
    {
    if(W.getQueriesUsed() >= queryLimit) return;
    do
      { groupChange = randomNodupGroupChange(1, −1, 1); }
    while(groupChange[0].v == 0);
    hypWAM = W.QueryGroupChange(groupChange);
    hypDist = Distance(hypWAM, goal);
    distRed = curDist − hypDist;
    if(distRed > 0.000001) break;
    }
  P.push(newIndividual(groupChange, distRed));
  }

void migrateOut()
  {
  int i;
  int x, y, v;
  Individual best;

  best = P.top(); P.pop(); // Extract the top element from P.

  // Check that applying it does not violate the pixel change cap.
  for(i = 0; i < best.vc.size(); i++)
    {
    x = best.vc[i].x;
    y = best.vc[i].y;
    v = best.vc[i].v;
    // If it does, break early. If not, finish normally.
    if(ABS(changesMade[x][y] + v) > PIXEL_CHANGE_CAP) break;
    }
  // If haven't finished normally, then applying best violates pixel change cap. Discard it.
```

```
    if(i != best.vc.size()) return;

    // Check that applying best will not take a pixel outside [0..255]. Also, that the distance
    // reduction label is positive.
    if(W.canMakeGroupChange(best.vc) && best.distanceReduction > 0)
      {
      // If so, make the change.
      W.makeGroupChange(best.vc);
      // Record that pixels have been touched.
      for(i = 0; i < best.vc.size(); i++)
        changesMade[best.vc[i].x][best.vc[i].y] += best.vc[i].v;
      }
    }

  void merge()
    {
    Individual first, second;
    Individual merged;
    vecDouble hypWAM, curWAM;
    double curDist, hypDist, distRed;
    int i;

    vector<Individual> mergeResults;
    for(i = 0; i < initialSize; i++)
      {
      first = P.top(); P.pop();
      second = P.top(); P.pop();
      if(W.getQueriesUsed() < queryLimit)
        {
        merged.vc = mergeTwoGroupChanges(first.vc, second.vc);

        hypWAM = W.QueryGroupChange(merged.vc);
        curWAM = W.getWAM();
        curDist = Distance(curWAM, goal);
        hypDist = Distance(hypWAM, goal);
        merged.distanceReduction = curDist - hypDist;
        mergeResults.push_back(merged);
        }
      else
        {
        mergeResults.push_back(first);
        mergeResults.push_back(second);
        }
      }

    while(!P.empty()) P.pop();

    for(i = 0; i < mergeResults.size(); i++)
      P.push(mergeResults[i]);
    }

  void dwindle()
    {
    while(!P.empty())
      migrateOut();

    vecDouble curWAM = W.getWAM();
    double curDist = Distance(curWAM, goal);
    ECHO("%.12lf␣%d\n", curDist, W.getQueriesUsed());
    }


};

// RestorerRand.cpp
```

```
// The restorer for the Random algorithm.
class RestorerRand : public Restorer
{
private:
int LOTS, CHANGES;
int queryLimit;

public:
RestorerRand(string encodingLog, int lots, int changes, int queries) : Restorer(encodingLog)
  {
  LOTS = lots;
  CHANGES = changes;
  queryLimit = queries;
  }

void apply()
  {
  int i, j;
  int iterationsWithoutReduction = 0;
  int N = W.getN();
  int M = W.getM();

  double hypDist, bestDist;
  vector<Change> groupChange, bestGroupChange;
  vecDouble curWAM, hypWAM;

  while(W.getQueriesUsed() < queryLimit)
    {
    curWAM = W.getWAM();
    bestDist = Distance(goal, curWAM);
    bestGroupChange.clear();

    ECHO("%.12lf %d\n", bestDist, W.getQueriesUsed());

    if(iterationsWithoutReduction == 100)
      {
      iterationsWithoutReduction = 0;
      CHANGES = MAX(1, (CHANGES*3)/4);
      }

    for(i = 0; i < LOTS; i++) // Randomly choose LOTS groups of pixels to change.
      {
      // Each group has CHANGES pixels in it. Each is perturbed by 1, 0 or -1.
      // Effectively, the group has upto CHANGES perturbations, each of value 1 or -1.
      groupChange = randomNodupGroupChange(CHANGES, -1, 1);

      // For large values of CHANGES it is virtually impossible for the group change to contain all
      // zeroes. However, for a small value of CHANGES, it is worth checking. If so, reselect.
      if(CHANGES <= 10)
        {
        for(j = 0; j < groupChange.size(); j++)
          if(groupChange[j].v != 0)
            break;
        if(j == groupChange.size())
          { i--; continue; }
        }

      // Change is not void. Query the oracle for its effects.
      // If better than best, record it.
      hypWAM = W.QueryGroupChange(groupChange);
      hypDist = Distance(goal, hypWAM);
      if(hypDist < bestDist)
        {
```

```
          bestDist = hypDist;
          bestGroupChange = groupChange;
          }

        }
      if(bestGroupChange.empty())
        { iterationsWithoutReduction++; }
      else
        {
        iterationsWithoutReduction = 0;
        W.makeGroupChange(bestGroupChange);
        for(i = 0; i < bestGroupChange.size(); i++)
          changesMade[bestGroupChange[i].x][bestGroupChange[i].y] += bestGroupChange[i].v;
        }
      }
    curWAM = W.getWAM();
    bestDist = Distance(goal, curWAM);
    ECHO("%.12lf %d\n", bestDist, W.getQueriesUsed());
    W.outputImage("restored.pgm");
    }
};

// RestorerQP.cpp
// The restorer for the quadratic programming algorithm.
class RestorerQP : public Restorer
{
private:
int queryLimit;
int COUNT;

vector<Change> changes;

matrixDouble A;
matrixDouble V;
vecDouble b;

public:
RestorerQP(string encodingLog, int queries) : Restorer(encodingLog)
  {
  queryLimit = queries;
  COUNT = 2000;
  }

void apply()
  {
  clock();
  srand(time(0));

  vecDouble curWAM = W.getWAM();
  double curDist = Distance(curWAM, goal);
  ECHO("%.12lf %d\n", curDist, W.getQueriesUsed());

  while(W.getQueriesUsed() < queryLimit)
    {
    if(W.getQueriesUsed() >= 20000) COUNT = 1000;
    if(W.getQueriesUsed() >= 30000) COUNT = 500;
    if(W.getQueriesUsed() >= 40000) COUNT = 50;

    pickChanges();
    calculateModel();
    writeMatlabInput();
    signalMatlab();
    idleUntilMatlabIsReady();
    performChanges();
    }
  W.outputImage("restored.pgm");
```

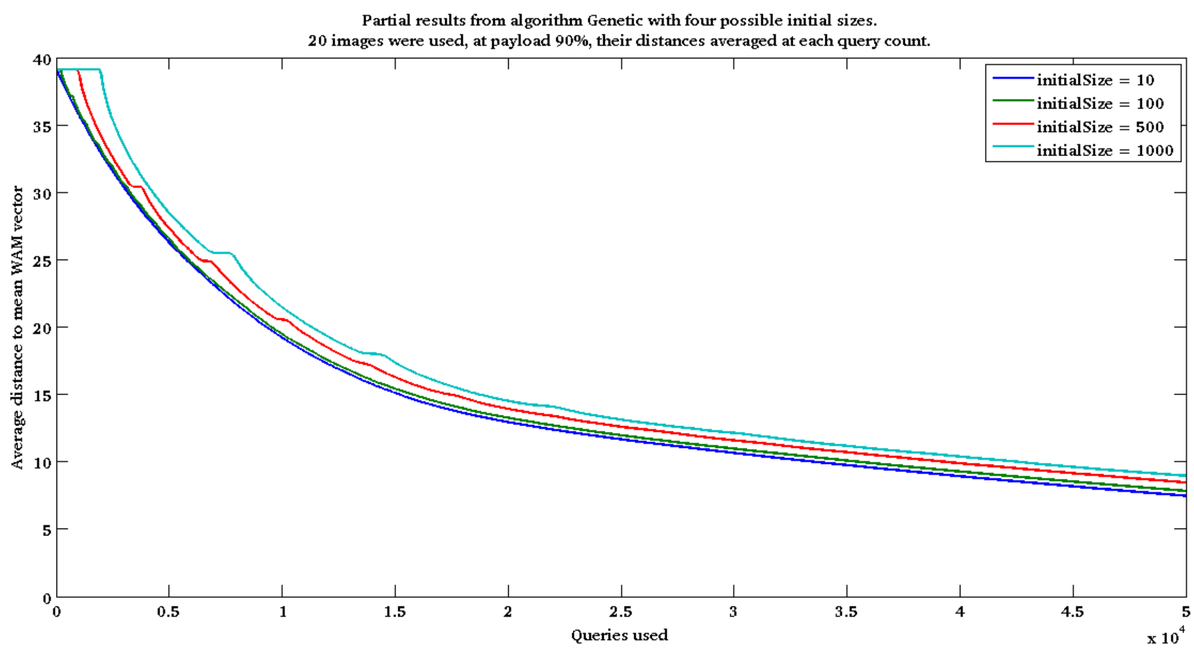
```
  }

void pickChanges()
  {
  int i, x, y, v;

  random_shuffle(availablePixels.begin(), availablePixels.end());
  changes.clear();
  for(i = 0; i < availablePixels.size() && changes.size() < COUNT && changes.size() + W.getQueriesUsed() < queryLimit; i++)
    {
    x = availablePixels[i].x;
    y = availablePixels[i].y;
    if(ABS(changesMade[x][y]) == PIXEL_CHANGE_CAP) continue;
    Change c = newChange(x, y, -1);
    changes.push_back(c);
    c = newChange(x, y, 1);
    changes.push_back(c);
    }
  sort(changes.begin(), changes.end());
  }

void calculateModel()
  {
  int i, j;
  vecDouble f, v, delta;
  int n = changes.size();

  A.clear();
  V.clear();
  b.clear();
  f = W.getWAM();
  V.resize(27);
  for(i = 0; i < 27; i++) V[i].resize(n);
  for(i = 0; i < n; i++)
    {
    Change c = changes[i];
    v = W.QueryOneChange(c);
    delta = v - f;
    for(j = 0; j < 27; j++)
      V[j][i] = delta[j];
    }
  // A = V'*covInv*V;
  matrixDouble Vprimed = transpose(V);
  matrixDouble a = Vprimed*covInv;
  A = a*V;
  // b = 2*V'*covInv*(f - means);
  vecDouble c = f - means;
  for(i = 0; i < 27; i++) c[i] *= 2;
  b = a*c;
  }

void writeMatlabInput()
  {
  int i, j;
  int n = changes.size();
  // Describe the quadratic objective. Note that Matlab's
  // input is 1/2x'Hx, so I need to double my A before putting it in.
  FILE *fout = fopen("H.dat", "wb");
  for(i = 0; i < n; i++)
    {
    for(j = 0; j < n; j++)
      FECHO(fout, "%.12lf ", 2*A[i][j]);
    FECHO(fout, "\n");
    }
  fclose(fout);
```

```

  // Describe the linear objective. Matlab's f is my b.
  fout = fopen("f.dat", "wb");
  for(i = 0; i < n; i++)
    FECHO(fout, "%.12lf␣", b[i]);
    FECHO(fout, "\n");
  fclose(fout);
  // Linear constraint matrix.
  // A constraint for each i, i + 1. We did not use these! The Matlab script ignores them.
  fout = fopen("A.dat", "wb");
  for(i = 0; i < n; i += 2)
    {
    for(j = 0; j < n; j++)
      if(j == i || j == i + 1)
        FECHO(fout, "1␣");
      else
        FECHO(fout, "0␣");
    FECHO(fout, "\n");
    }
  fclose(fout);
  // Right-hand side of the linear constraints.
  fout = fopen("b.dat", "wb");
  for(i = 0; i < n; i += 2)
      FECHO(fout, "1␣");
  FECHO(fout, "\n");
  fclose(fout);
  }

void signalMatlab()
  {
  FILE *fout;
  fout = fopen("signalToMatlab", "wb");
  FECHO(fout, "112233\n");
  fclose(fout);
  }

void idleUntilMatlabIsReady()
  {
  int x;
  FILE *fin;
  while(1)
    {
    fin = fopen("signalFromMatlab", "rb");
    if(fin != NULL)
      {
      fscanf(fin, "%d", &x);
      if(x == 998877) break;
      }
    }
  fclose(fin);
  remove("signalFromMatlab");
  }

void performChanges()
  {
  int n = changes.size();
  int i;
  vector<Change> gc;
  double x;
  FILE *fin = fopen("matlabOutput", "rb");
  for(i = 0; i < n; i++)
    {
    fscanf(fin, "%lf", &x);
    if(x >= 0.5) { gc.push_back(changes[i]); }
    }
```

```
    fclose(fin);
   W.makeGroupChange(gc);
    vecDouble curWAM = W.getWAM();
    double curDist = Distance(curWAM, goal);
    for(i = 0; i < gc.size(); i++)
      {
      int pixelx = gc[i].x;
      int pixely = gc[i].y;
      int changev = gc[i].v;
      changesMade[pixelx][pixely] += changev;
      }
   ECHO("%.12lf␣%d\n", curDist, W.getQueriesUsed());
    }
 };

% The Matlab script we used to automate the QP algorithm.
while true
    clear;
      fid = fopen('signalToMatlab', 'r');
      if(fid == -1) continue; end;
      x = fscanf(fid, '%d', 1);
      fclose(fid);
      if x ~= 112233 continue; end;
    load -ascii H.dat;
    load -ascii f.dat;
    n = 2000;
    lb = ones(n, 1) - ones(n, 1);
    ub = ones(n, 1);
    tic
    xs = quadprog(H, f, [], [], [], [], lb, ub);
    toc
    save -ascii -double 'matlabOutput' xs;
      delete('signalToMatlab');
    delete('H.dat');
    delete('f.dat');
    fid = fopen('signalFromMatlab', 'w');
    fprintf(fid, '998877');
    fclose(fid);
end
```

Appendix D

Partial Benchmarking of Genetic Algorithm



Appendix E

Partial Benchmarking of Random Algorithm