MMathematics and Computer Science

Part C

Project Report

Video Steganography

Andreas Neufeld

21st May 2012

Steganography is the process of hiding a secret message in innocently looking cover objects, which are h.264 videos in this project. We simulate the embedding of hidden data in the compressed video stream using methods that are known from image steganography. In addition to the embedding method we vary the coefficients that are used for embedding. We propose a set of features that is useful for steganalysis, that is to detect if a video is manipulated. The features are based on histograms and co-occurrences of certain coefficients in the video stream. The simulation and feature extraction is performed on a large set of videos and explored visually. A benchmark based on the *Kullback-Leibler Divergence (KL-D)* which we found to be uncomputable in practice is discussed. A computable alternative, the *Maximum Mean Discrepancy (MMD)*, is used to compare the detectability of the different embedding methods in our proposed set of features. We are interested in the least detectable embedding method as well as the coefficients that are best suited for embedding. We will use both luma and chroma components of the videos and compare which one is safer for embedding. The MMD calculations are GPU accelerated.

1 Acknowledgements

I would like to thank my supervisor, Dr. Andrew D. Ker, for his constant guidance and advice which has only made this project possible, and for being an excellent class tutor.

I would also like to thank Dr. Tomáš Pevný, for his feedback and inspiration that has significantly shaped this project.

I would like to extend my gratitude to my parents, for their support of this project and so many other things.

Contents

1 Acknowledgements **2**

2 Introduction **5**
2.1 Steganography vs. Cryptography and Watermarking 5
2.2 Steganographic Media 5
2.3 enCOding and DECoding 6
2.4 Practical Approach 7

3 The H.264 Advanced Video Coding (AVC) Standard **7**
3.1 Colour Spaces 7
3.1.1 RGB 7
3.1.2 YCrCb 7
3.1.3 Sampling Patterns 8
3.2 Prediction 9
3.3 Transformation, Quantization and Entropy Coding 10

4 Steganographic Security **12**
4.1 Steganographic Channels 12
4.2 Embedding Methods 13
4.2.1 Naive hiding: LSB embedding 14
4.2.2 ± 1 embedding 14
4.2.3 F5 embedding 15
4.3 Practical embedding 15
4.3.1 Matrix Embedding 17
4.3.2 Non-shared selection channel 17
4.4 Proposed Features 17
4.5 Proposed Distortion Measurements 19
4.5.1 Kullback-Leibler Divergence 19
4.5.2 Maximum Mean Discrepancy 20
4.6 Questions about Embedding 21

5 Implementation **21**
5.1 Stegosaurus GUI 22
5.1.1 The xml backend 24
5.1.2 The .fv file format 25
5.1.3 Normalization 26
5.2 The Video Collection 27
5.3 H.264 Syntax 28
5.4 Feature Extraction 28
5.5 Embedding simulation 30
5.6 KL-D 32
5.7 MMD 34

6 Experimental Results 35
6.1 Visual Distortion 35
6.2 Feature Analysis 35
6.2.1 Which channel is best suited for embedding? 41
6.2.2 Which embedding method is the least detectable? 43
6.2.3 Which coefficients are best suited for embedding? 43
6.3 Limitations 45

7 Conclusion 45
7.1 Future Work 46
7.2 Personal Report 47

8 Appendix: Source Code 49
8.1 ffmpeg-extract 49
8.2 Stegosaurus 52
8.2.1 structures and classes 52
8.2.2 GUI interaction 54
8.2.3 Normalization 55
8.2.4 MMD Calculation 59

2 Introduction

2.1 Steganography vs. Cryptography and Watermarking

Steganography is the process of hiding data in cover objects, the name comes from the Greek words “steganos” which means “covered” and “graphia” which means “writing” [1]. The aim of steganography is to hide the fact that communication happened at all which is different from the aim of cryptography. If communication happens encrypted the fact of communication is not hidden.

Watermarking uses similar techniques as steganography but with a different intention, a watermarked video could contain a hidden key which identifies the special copy of the video file. This would allow us to trace copies of the video. But in watermarking the absence of knowledge that a watermark is being used is not crucial, more importantly the watermark shall be robust against attacks like re-encoding and video filters. A steganographic system is broken as soon as the fact that communication took place is revealed, no matter if the messages are deciphered or not.

Steganography is a very old idea, in ancient times one method of data hiding was to write a message on the bald head of a slave and then to send him to the receiver once the hair has grown again. There are more recent methods especially in the area of online multimedia content which is becoming more present. Cisco Systems predicted in its VNI Forecast (February 2012) that by 2015 62% of all online traffic will be video, increasing from 40% in 2010[1]. German authorities have detected and successfully extracted 141 documents containing detailed plans for future attacks embedded into a pornographic video found on an suspected al-Qaeda operative who was arrested in May 2011 in Berlin[2][3]. It is important to investigate if video can be used for steganography and how steganography in videos can be detected.

2.2 Steganographic Media

In this project we try to assist Alice and Bob who are imprisoned and trying to escape. They can exchange videos but all their communication is controlled by warden Eve who observes their communication.

There are three different approaches to digital steganography for Alice and Bob to use [1]: Steganography by cover ...

selection There is a fixed database of cover objects each assigned with a secret meaning. Alice needs to pick the correct cover to transmit in order to convey a message.

synthesis Alice creates a cover with special properties to transmit a hidden meaning. This can be colours of clothes in a picture or the length of a text rather than the content.

[1]http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html
[2]http://www.zeit.de/2012/12/Al-Kaida-Deutschland
[3]http://edition.cnn.com/2012/04/30/world/al-qaeda-documents-future/?hpt=hp_c1

modification Given a cover object Alice modifies it slightly so that the change is not at all or at least hardly noticable. For example Alice could apply different video filters until the modified video contains the desired message. Or Alice can change the syntax elements inside the video file slightly so that Bob can extract a message from them.

Steganography by cover selection is not interesting to Alice and Bob, they would both have to synchronize a large collection of video files and agree on a meaning of each individual video. Cover synthesis is hard to do on video since it demands to re-shoot the same video multiple times. Therefore Alice and Bob will use cover modification. In this project we will discuss different approaches to how elements within a video bitstream can be tweacked to secretely convey a message and also investigate how detectable these modifications are.

Before we can start we need to introduce the h.264 video codec which is commonly used online at the time of writing and serves as cover object for our experiments.

2.3 enCOding and DECoding

A video is a collection of images called frames each taken at a constant time interval. A CODEC is a pair of an encoder and a decoder which allow to convert a raw video into a bitstream (encoding) and also recover the raw video from the bitstream (decoding). As for many other codecs the h.264 standard only specifies the decoding process [2], any program that creates a valid h.264 bitstream can be called an encoder. Different encoders may yield different bitstreams but the decoded video is unique.

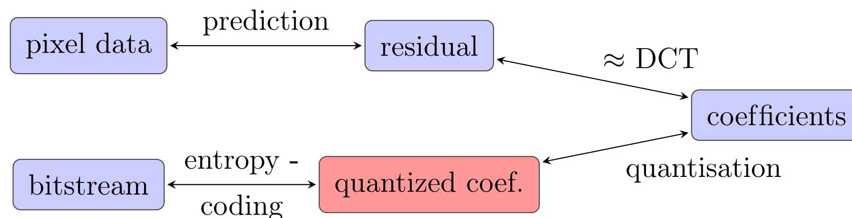


Figure 1: Video coding workflow. We modify only the quantized coefficients in the compressed video stream when simulating embedding.

When encoding a frame using h.264 first a prediction is made, this can either be an intra- or an inter prediction. Intra prediction uses data from the same frame that already has been encoded and inter prediction uses past and/or future frame(s). The prediction is subtracted from the pixel data, the residual gets transformed using an integer approximation to the Discrete Cosine Transform (DCT) and quantized. The quantized coefficients then get entropy-coded into the final bitstream.

2.4 Practical Approach

In this project we will simulate steganography by modification of quantised coefficients in an h.264 video stream and compare different approaches with repect to their detectability. The embedding simulation will happen on a set of dvds that are transcoded to h.264 with the x264[4] encoder and using ffmpeg[5] to read the resulting video bitstreams.

A set of features will be proposed to represent a section of video as a high dimensional feature vector. The detectability of different embedding schemes with different parameters will be measured as distortion in these features using the *Kullback-Leibler Diverence (KL-D)* and *Maximum Mean Discrepancy (MMD)*. The distortion measures are described in detail in section 4.5.

3 The H.264 Advanced Video Coding (AVC) Standard

H.264 is a block-based video codec, the pipeline mentioned in section 2.3 is traversed for each 16x16 pixel macroblock in a video frame. Within each macroblock there are different channels representing luma und chroma components which are described in the following section.

3.1 Colour Spaces

3.1.1 RGB

RGB image or video data stores the relative proportion of red, green and blue colour of each pixel. It can be captured by arrays of different sensors that only capture one colour and displayed by illuminating the red, green and blue part of each pixel in the display accordingly. Figure 2 shows the RGB-decomposition of a Foreman[6]-snapshot.

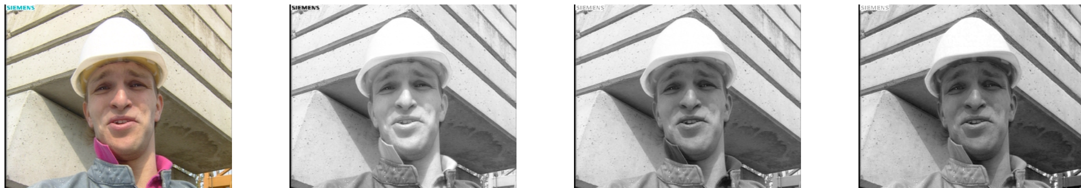


Figure 2: Original frame and RGB components

3.1.2 YCrCb

The human eye is more sensitive to luminance rather than to colour. That's the reason for most video codecs to convert the input video data into a different representation,

[4]http://www.videolan.org/developers/x264.html
[5]http://www.ffmpeg.org/
[6]A video commonly used for encoder testing (http://media.xiph.org/video/derf/y4m/foreman_cif.y4m).

YCrCb. The aim is to separate luminance and colour so that colour information can be encoded in lower resolution.

The luminance component can be calculated as follows:

$$Y = k_r R + k_g G + k_b B \tag{1}$$

The constants k_r, k_g, k_b are given by the h.264 specification [3].

Now only the difference from Y gets stored in the colour information:

$$\begin{aligned} Cr &= R - Y \\ Cb &= B - Y \\ Cg &= G - Y \end{aligned} \tag{2}$$

Cg can be expressed as a linear sum of the other variables [2], so only Y, Cr and Cb need to be stored. Figure 3 shows the decomposition of the same snapshot as above.

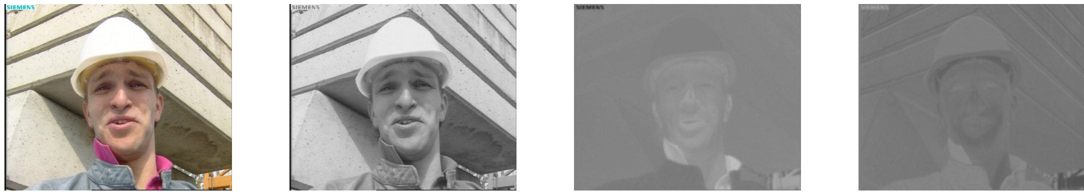
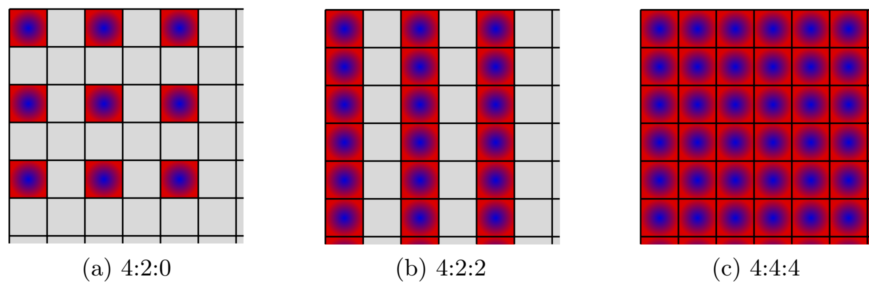


Figure 3: Original frame and YCrCb components

3.1.3 Sampling Patterns

There are three sampling patterns available in H.264 that allow to reduce the amount of stored chroma information: 4:4:4, 4:2:2 and 4:2:0. They are illustrated in Figure 4. 4:2:0 is most common since the human eye usually does not notice any difference to the originial image even though the number of chroma samples is reduced to a quarter.



(a) 4:2:0 (b) 4:2:2 (c) 4:4:4

Figure 4: Different sampling patterns. Each cell represents a luma sample and coloured cells represent chroma samples.

There are two chroma channels, thus in 4:2:0 sampling half as many chroma as luma samples are being stored.

3.2 Prediction

A frame is divided into slices and each slice is divided into 16x16 pixel macroblocks. A macroblock can be further partitioned down to block size 4x4.

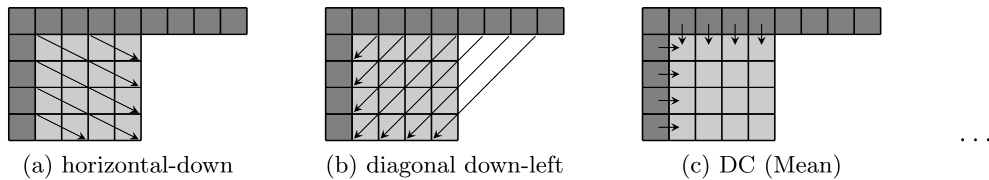


Figure 5: A few different Intra Prediction Modes. Dark grey Pixels are already decoded, the light grey ones need to be predicted from them.

There are three different kinds of frames, I, P and B frames, similarly there are I, P and B macroblocks. An I macroblock makes a prediction based on the pixels to its left and top in the same frame (intra prediction), we can regard an I Frame as a picture with a compression technique that is similar to JPEG. P macroblocks use a portion of a past frame as prediction where the so-called motion vector usually has half pixel precision to capture fracture pixel movements. B macroblocks come with two motion vectors that can point in both future and past frames in order to make a more accurate prediction.

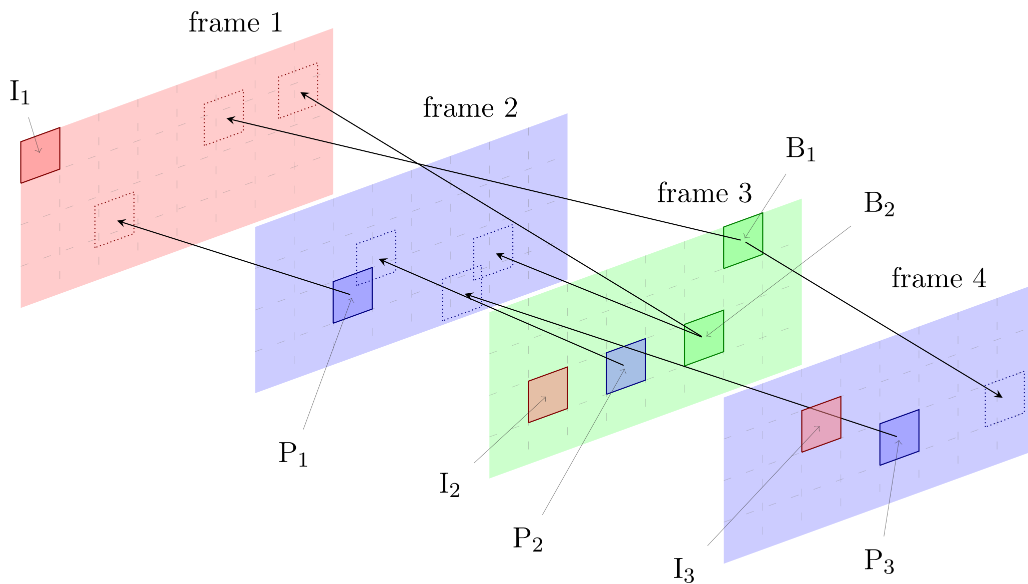


Figure 6: Inter Prediction. Showing I-, P- and B-Macroblocks.

The first frame in a video has to be an I frame since there are no possible reference frames present. There are two kinds of I frames in h.264, regular I frames ind IDR (Instantaneous Decode Refresh) frames. IDR frames contain only I macroblocks as well and also they don't allow references by future P and B frames behind them. IDR frames partition the video into sections that can be decoded independently allowing the user to jump to a future position in the video stream without having to decode all frames in between. The first frame is an IDR frame.

The P macroblocks in figure 6 use frame 1 and frame 2 as reference frames. Frame 3 has a future reference on frame 4, hence frame 4 has to be decoded before frame 3. The macroblock P_3 is not allowed to reference frame 3 even though frame 3 is displayed before frame 4. P macroblocks in later frames are allowed to reference frame 3, there is no distinction between reference frame types.

P macroblocks use the point they reference to as prediction where coordinates have half- or quarter-pixel precision. B macroblocks combine both reference frames to a prediction using a weighted average or other combination methods. Prediction is a key part of each video codec.

Frames can also be partitioned into horizontal slices, there are I, P and B slices. Slices are separate units that can be decoded independently allowing the decoder to process a frame on parallel hardware. Blu rays for example have to contain at least 4 slices to ensure playability by relatively cheap hardware in blu ray players.

In order to visualize how an encoder might use the different frame types and how much they differ in size, Figure 7 shows the output of x264 when encoding foreman with QP = 20. I frames are red, P frames blue and B frames green.

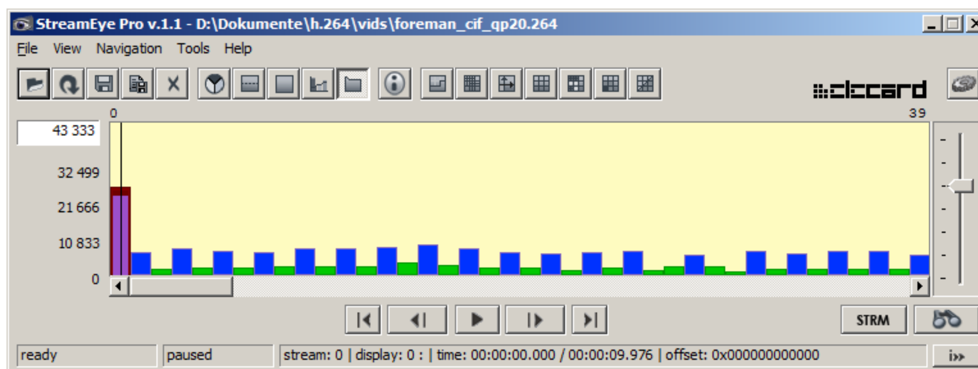


Figure 7: Example x264 output stream

3.3 Transformation, Quantization and Entropy Coding

H.264 supports two different luma transforms: the default and Intra 16x16 (AC/DC) transforms which are illustrated in figure 8 and figure 9. Chroma uses only the AC/DC transform, see figure 10.

The Quantisation Parameter (QP) is constant for each 4x4 block and the actual step size s is exponential in QP, it doubles as QP increases by 6 [2]. On input coefficient x we

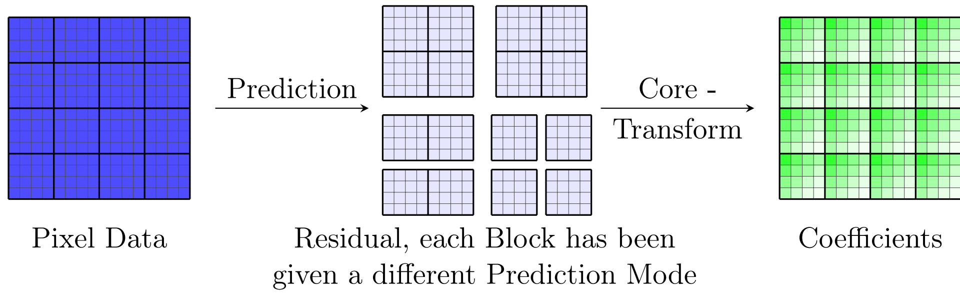


Figure 8: Luma default Transform. Lower intensity indicates lower magnitude of coefficients. Quantisation is omitted here.

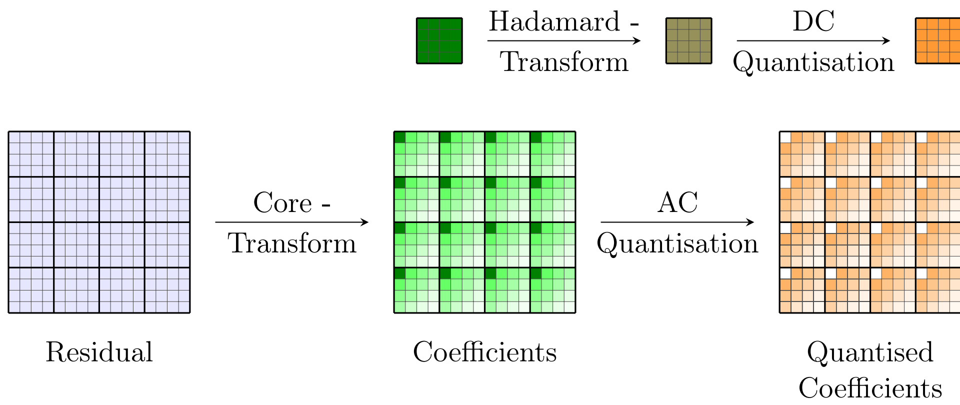


Figure 9: Luma Intra 16x16 Transform. Partitions are not allowed here.

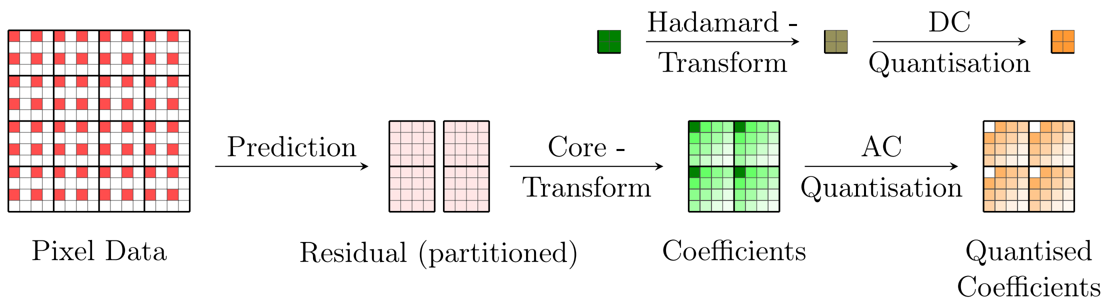


Figure 10: Chroma Transform.

compute quantized coefficient y by $y = \text{round}(\frac{x}{s})$. When decoding the video x will be approximated as $x' = y \cdot s$. Pixel values are integers, h.264 uses an integer approximation to the DCT as core transform, the Hadamard transform is an integer transform as well, quantization in h.264 takes integers as input and produces integers as output.

Each video file has a *profile* defining which encoding techniques need to be supported by the decoder to play the video. Higher profiles support more advanced features such as the 8x8 DCT transform or re-ordering of macroblocks within a frame. In this project we will use the main profile only since it supports all functionality that we require.

There are two different entropy coding modes, CAVLC (Context Adaptive Variable-Length Coding) and CABAC (Context Adaptive Binary Arithmetic Coding). Entropy coding is a lossless process whereas the quantisation is lossy, the decoded frame will differ from the original one. In CAVLC, the coefficients are scanned in the zig-zag pattern which is illustrated in figure 11

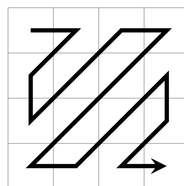


Figure 11: Zig-zag scan order

4 Steganographic Security

4.1 Steganographic Channels

A steganographic channel (or steganographic system / stegosystem) fully describes how two parties called Alice and Bob communicate secretly. It consists basically of the following elements:

- cover source
- embedding and extraction methods
- source of stego keys (similar to cryptography)
- source of messages
- data exchange channel, observed by warden Eve

The part of Figure 12 can be defined formally [1]:

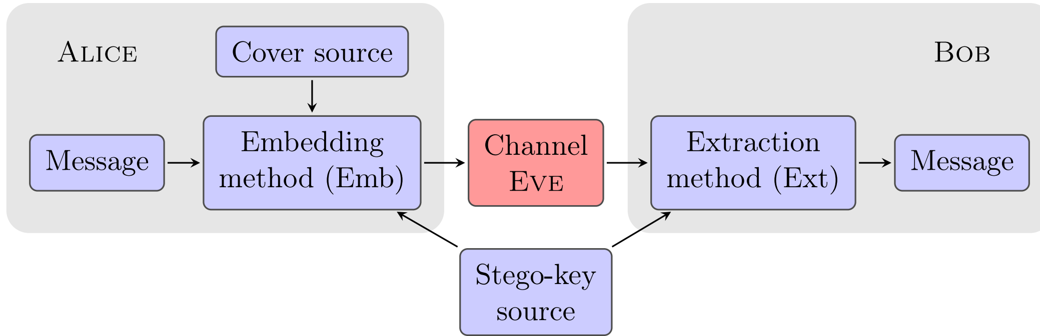


Figure 12: Steganographic channel

$$\begin{aligned}
C &\quad \text{set of cover objects } x \in C, && (3)\\
K(x) &\quad \text{set of stego-keys for } x, && (4)\\
M(x) &\quad \text{set of messages for } x, && (5)\\
Emb: &\quad C \times K \times M \longrightarrow C, && (6)\\
Ext: &\quad C \times K \longrightarrow M, && (7)\\
\text{such that} &\quad Ext(Emb(x,k,m),k) = m. && (8)
\end{aligned}$$

The exchanged data is controlled by warden Eve who we assume to be a passive observer. Alice and Bob require a stegosystem that achieves maximum payload within a certain detectability. We will now present different embedding methods and approaches to estimate their performance.

4.2 Embedding Methods

All embedding methods discussed in this project share the extraction method which is to look at the least significant bits (LSB) in a certain subset of the encoded quantized transform coefficients. There are different approaches to give the LSB the desired value which will be described in the next sections.

In the examples below we will assume that the hiding probability is 1 and that every coefficient is used for hiding. In practical steganography we would use a much smaller hiding probability and avoid hiding in zeros and ones since zeros receive special treatment in the entropy coding process, changing the number of zeros may have impact on the video file size and cause a desynchronization of codes. In CAVLC the number of non-zero coefficients in surrounding blocks determines the bitstring table that is used to store a coefficient, changing the number of non-zero coefficients may result in the decoder selecting the wrong table.

4.2.1 Naive hiding: LSB embedding

An obvious thing to do is to just set the LSB to the desired value. We may assume that in half of the cases the LSB already contains the correct value. The following diagram illustrates the hiding procedure:

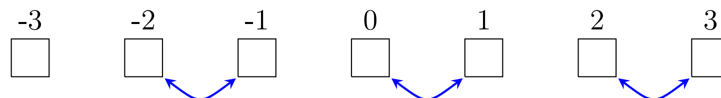


Figure 13: LSB Embedding.

Flipping the LSB changes a coefficient to a fixed target and is self-inverse. Therefore the coefficients group into pairs swapping values. If we use LSB embedding on each coefficient an example histogram of coefficient occurrences will change in the following way:

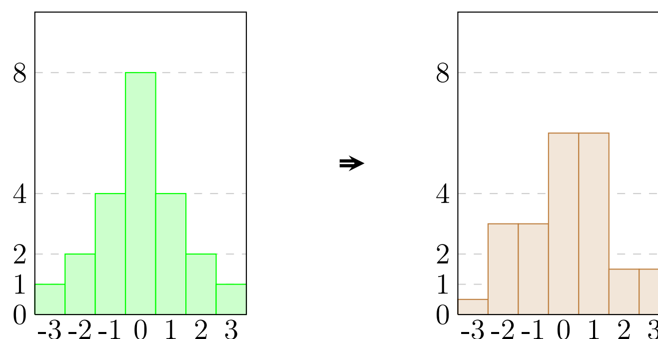


Figure 14: Impact of LSB Embedding on the coefficient histogram. Let l and r be two coefficient values that differ only in their LSB. Then half of the coefficients with value l will change into r and vice versa, the histogram will contain the average value in position l and r.

4.2.2 ± 1 embedding

A problem of the LSB embedding is its asymmetry, the coefficients group into unnatural pairs. We can add symmetry by randomly increasing or decreasing a coefficient instead of flipping LSB, this means that more than one bit may need to be changed.

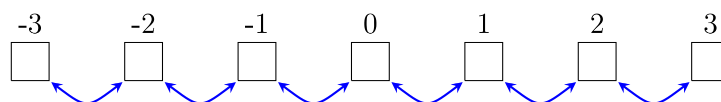


Figure 15: ± 1 Embedding.

We can perform ± 1 embedding on the example histogram above, assuming that the number of increments is exactly the number of decrements for any coefficient value:

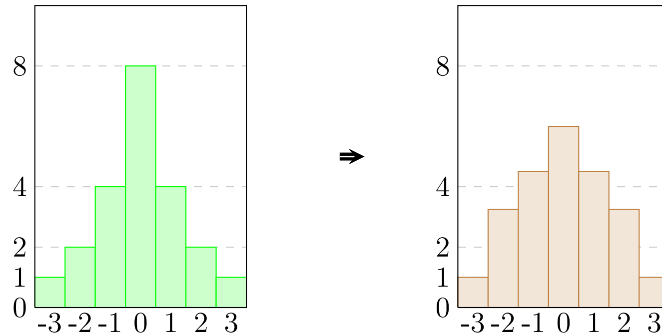


Figure 16: Impact of ±1 Embedding on the coefficient histogram. The stego histogram is symmetric as the input histogram is as well, but the shape has changed. Now it is not two values that average out exactly but the entire histogram smoothens.

If we do not want to change the number of zeros on embedding but still wish to use the value 1 coefficients we have to always increase the absolute value of value 1 and -1 coefficients. This is the hiding scheme with threshold:

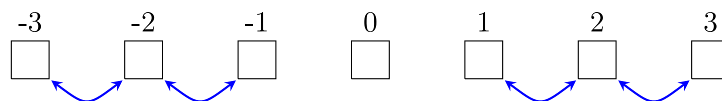


Figure 17: ±1 Embedding with threshold 1. Coefficients with values 1 or -1 always increment their absolute value.

The outcome on our example histogram is shown in Figure 18.

4.2.3 F5 embedding

If we wish to maintain the histogram shape another possibility is to decrease the absolute value when embedding a bit into a coefficient, as shown in figure 19.

The oucome on our example histogram is shown in figure 20.

A threshold can be imposed similarly as in the case of ±1 embedding, this time the histogram shape is preserved.

4.3 Practical embedding

We have seen different methods to properly set the LSB of a particular coefficient in section 4.2 but the question remains which coefficients need to be changed to transmit the hidden message. The naive way is to set a random path through all coefficients in the video file and read their LSBs in sequence. We would expect Alice to change half the coefficients on the way to get the message right. Higher payloads can be achieved by using Matrix Embedding.

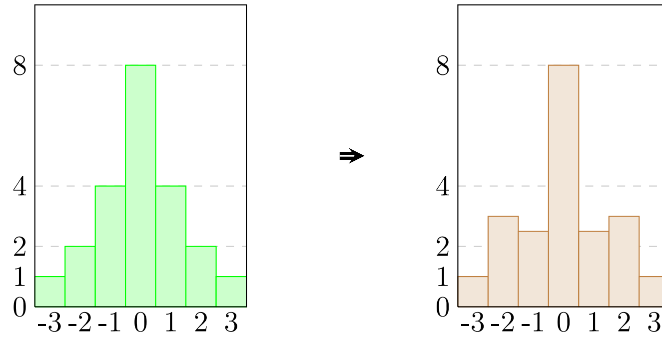


Figure 18: Impact of ± 1 embedding with threshold 1 on the coefficient histogram. Half of the value 1 coefficients change to value 2 and no coefficient changes from 0 to 1, this leads to there being more 2's than 1's in the output histogram. We lose the typical histogram shape.

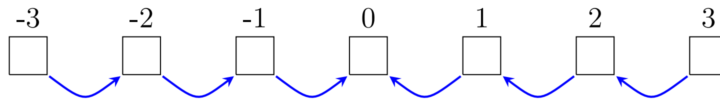


Figure 19: F5 Embedding.

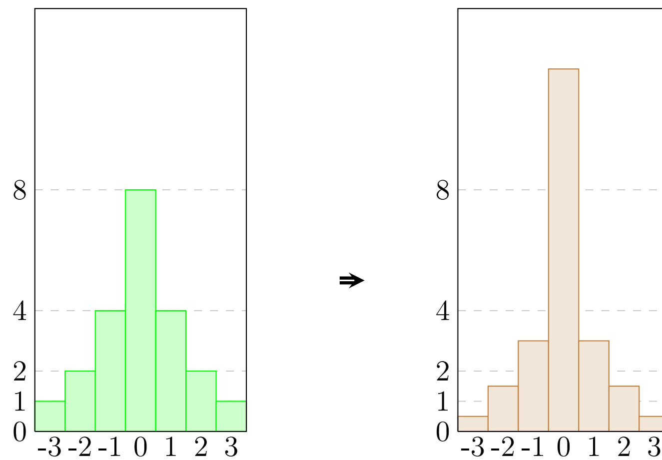


Figure 20: Impact of F5 Embedding on the coefficient histogram. The coefficients are pushed to zero yielding a higher peak in the middle of the histogram.

4.3.1 Matrix Embedding

Hamming codes are error correcting block codes based on the *parity check matrix* H containing all non-zero binary vectors in a fixed dimension [4]. In the case of 3 dimensional vectors H is the following matrix:

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

We can use the fact that all coefficients have their LSB set to some value. Given a group of seven coefficients we have a binary vector v of seven LSBs and the product $H \cdot v = u$ is well-defined. If Alice wants to communicate three dimensional vector x to Bob and $x \neq u$ she needs to find $\tilde{u} = u - x$ where all operations happen modulo 2 so that $\tilde{u}$ again is a binary vector. Since $x \neq u$, $\tilde{u}$ is non-zero and therefore it is a column in H, say column number j. Then the LSB in coefficient number j needs to be flipped so that $H \cdot v' = x$.

Three dimensional Hamming codes achieve three communicated bits by changing one coefficient in worst case whereas the naive approach requires to change $\frac{3}{2}$ bits on average to transmit 3 bits.

4.3.2 Non-shared selection channel

Alice having the clean cover object has strictly more information available that Bob does. For example Bob does not know if a particular coefficient was changed or not. For instance if Alice and Bob agree to use coefficients with absolute value $\geqslant 2$ only for embedding to keep a distance to value 0 coefficients Bob cannot know if a coefficient with value 1 had value 2 before Alice changed it or if it is an original 1. Thus Alice always has to increment ones.

Alice might even have the original uncompressed video and only wants to use coefficients for embedding that are highly affected by quantisation. For example if the step size is 20 and the DCT coefficient is 89, quantized to 4, Alice would want to change it to 90 which is quantized to 5. If the DCT output is 80 Alice will not want to use that coeffcient at all. This information is unknown to Bob, Alice and Bob need wet paper codes [1].

A possible solution to the problem is Alice and Bob to share a large binary Matrix D which will map vector v of LSB values from the video to the desired message u. Alice needs to solve $D \cdot v = u$ with the restriction that only dry coefficients are changed. This can be achieved using any solving method, Gaussian elimination for example. Bob will simply evaluate the product without knowing which parts of v were manipulated by Alice. The matrix D can efficiently be defined using a common random generator seed.

4.4 Proposed Features

We assume the outcome of the DCT approximation that is used by h.264 to be an array where coefficients are scanned in zig-zag order.

We have seen in the last section that all of the mentioned hiding methods have a noticable impact on the coefficients histogram, thus we certainly want to include the histogram in our set of features. Our histograms include zeros but only zeros before the last non-zero coeffcient. Histograms alone are not good enough because Alice can change coefficients that are not containing any hidden information read by Bob simply to restore histograms. *Statistical restoration* makes additional unnecessary changes in the video which may introduce additional visual distortion or distortion in other sets of features. In most cases statistical restoration reduces the security of a steganographic system [1].

The magnitude of coefficients tends to decrease as we move down the zigzag scan list, therefore we will drop coefficients at the end to reduce dimensionality. Also we want to use individual ranges for each coefficient since ranges differ, these ranges can be determined by data visualisation in the Stegosaurus GUI program. Section 5.4 contains details about the chosen ranges.

In addition to histograms we will use co-occurrences as features. An array of 10 coefficients will allow to extract 9 co-occurrence pairs. Restoration of co-occurrence statistics is a much more complicated task and is likely to require a reduction of embedded payload. The DCT coefficients are highly decorrelated but h.264 uses a rough approximation to the DCT. The definition of the DCT is:

$$Y = A \cdot X \cdot A^T$$

$$\text{where } A = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & -b & -c \end{bmatrix}, \quad \begin{array}{rcccl} a & = & 1/2 & & \\ b & = & \sqrt{1/2} \cdot cos(\pi/8) & \approx & 0.6532 \\ c & = & \sqrt{1/2} \cdot cos(3\pi/8) & \approx & 0.2706 \end{array}$$

And h.264 uses the following matrix [2]:

$$C_{f4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & -2 & -1 \end{bmatrix}$$

There are likely to be correslations between coefficients specific to h.264 that will get lost when embedding data.

Each macroblock has one luma channel L but two chroma channels U and V. We may use the correlation between both chroma channels as features as well. Again we take histograms of 2 dimensional vectors, the first component being a coefficient in the U channel and the second component being the corresponding coefficient in V. We keep seperate histograms for each coefficient, that is up to 16 if no coefficient is dropped.

All features mentioned above, that is (i) histograms, (ii) co-occurrences and (iii) UvsV vectors are concatenated to the final feature vector. One vector is extracted per IDR section in a video. Histograms and co-occurrences of the Cr and Cb components are added up so that we have a luma, a chroma DC and a chroma AC part.

P and B slices are treated separately since we are interested in which one of the two is capable of carrying more hidden information. An IDR section may only contain P slices which is why there can be more P vectors than B vectors. B slices rely on P slices as reference points, hence there is a corresponding P feature vector for each B feature vector.

There are targeted attacks agains LSB embedding using custom features capturing the similarity of the histogram with expected histogram after LSB embedding [5], estimating the embedded payload as well [6]. These features can be derived from the coefficients histogram and therefore they do not need to be included in our feature set. These method specific features allow us to check for specific methods using very low-dimensional features but we attempt to use high dimensional features in order to detect any embedding method. Our features will be tested on the three different embedding methods mentioned above, (i) LSB embedding, (ii) ± 1 embedding and (iii) F5 embedding.

4.5 Proposed Distortion Measurements

There are two distortion measures that we will use, the *Kullback-Leibler Divergence (KL-D)* and the *Maximum Mean Discrepancy (MMD)*. The KL-D is an information theoretic concept and therefore any results shown using the KL-D are universally true for all possible detectors. The MMD takes a more geometric approach and is related to the performance of *support vector machines (SVMs)*.

We are particularly interested in small KL-D estimates since these impose a bound on the performance on any detector. Small MMD estimates only tell us that SVMs will not perform well as detectors, but other detectors might do.

Both measures will be briefly introduced now.

4.5.1 Kullback-Leibler Divergence

In information theory there is a notion of a *type* or *empicircal probability distribution* P_x which is the fractional occurrence of each codeword [4]. We assume that clean videos are of a certain type and different embedding methods will produce videos of a different type. Information therory also tells us that there is at most a polynomial number of types of sequences of length n while the number of possible sequences is exponential. There are only few types each containing many sequences.

There is more noise in videos than just steganographic embedding, different cameras, light settings or video filters may leave a trace in the video and form different types as well, we use similar video sources and identical encoding settings to reduce this problem.

Detecting if a video is manipulated is testing whether it follows the clean type P_{clean} or a steganographic type, for example P_{F5}, there are information theoretic bounds on the performance of hypothesis testing on different types, the probability of missed detection is approximately [4]:

$$P_{\text{MD}} \approx 2^{-D_{\text{KL}}(P_{\text{stego}} \| P_{\text{clean}})}$$

We want to maximize this value since we want the warden to miss embedded data. Therefore we need to minimize the KL-D, which is defined to be [4]:

$$D_{\mathrm{KL}}(P\|Q) = \sum_{x\in X} P(x)\cdot \log_2 \frac{P(x)}{Q(x)}$$

The KL-D is the expected number of extra bits needed to encode a symbol that follows distribution P when a code for distribution Q is used. If $P = Q$, $D_{\mathrm{KL}}(P\|Q) = 0$, the distributions are identical, therefore they are not distinguishable.

We model a type as a multivariate normal distribution or Gaussian since the central limit theorem states that Gaussians are appropriate to model independent and identically distributed (iid) random variables. The KL-D of two Gaussians $\mathcal{N}_0(\mu_0, \Sigma_0)$ and $\mathcal{N}_1(\mu_1, \Sigma_1)$ in dimension k is:

$$D_{\mathrm{KL}}(\mathcal{N}_0\|\mathcal{N}_1) = \frac{1}{2}\left(tr(\Sigma_1^{-1}\Sigma_0) + (\mu_1-\mu_0)^T\Sigma_1^{-1}(\mu_1-\mu_0) - \ln\left(\frac{\det\Sigma_0}{\det\Sigma_1}\right) - k\right)$$

This expression makes the KL-D computable, the computationally most intense part is the inversion of Σ_1, this can be achieved using the Housholder QR-factorisation, details follow in section 5.6.

4.5.2 Maximum Mean Discrepancy

The MMD of two distributions p and q and function class $\mathcal{F}$ is defined to be [7]:

$$\mathrm{MMD}(\mathcal{F}, p, q) = \sup_{f\in\mathcal{F}} \left(\mathbb{E}_{x\sim p} f(x) - \mathbb{E}_{x\sim q} f(x)\right)$$

In case of a finite sample sets X and Y of size N this is equivalent to:

$$\mathrm{MMD}(\mathcal{F}, X, Y) = \sup_{f\in\mathcal{F}}\left(\frac{1}{N}\sum_{i=1}^{N} f(x_i) - \frac{1}{N}\sum_{i=1}^{N} f(y_i)\right)$$

We have to make strong assumptions on the class of function $\mathcal{F}$ to make the MMD computable, Gaussian kernels have been shown to perform well when used in SVMs [8]. On this class of kernel functions an MMD estimate is given by:

$$\mathrm{MMD}(\mathcal{F}, X, Y) = \sqrt{\frac{1}{N(N-1)}\sum_{i\neq j} k(x_i, x_j) - 2k(x_i, y_j) + k(y_i, y_j)}$$

$$\text{where}\qquad k(x, y) = \exp(-\gamma\|x-y\|^2)$$

We choose γ to be η^{-2} where η is the median of L_2 distances of all pairs of feature vectors. The features need to be normalized before calculating the MMDs, more on this in section 5.1.

The MMD checks for each pair of feature vectors how well they are separable by applying a Gaussian kernel which is directly related to the performance of an SVM on each pair of input vectors.

4.6 Questions about Embedding

We are interested in the following questions:

1. Which embedding method is least detectable?

2. Which channel (Luma, Chroma DC, Chroma AC) is best suited for embedding?

3. Which coefficients are best suited for embedding? Are smaller coefficients preferable to larger ones? Do we want to include the DC coefficient when embedding?

Other important questions that are not covered in this project:

1. Are low bitrate videos preferable to high quality encodings?

2. What effect do video filters have on steganographic capacity?

3. Is embedding in the compressed stream workable or do we have to embed while encoding?

To answer these we will simulate embedding with different methods and two parameters, the threshold and a flag indicating if the DC coefficient shall be used as well. The threshold is a lower limit on the coefficient value that is used for hiding, it is interpreted differently for each embedding method.

We are interested in the detectability of a certain payload that is embedded in a video, before we can interpret the distance measures in a meaningful way we need to measure the impact. In this project we will use bits per non-zero coefficient (bpnc) since the number of non-zero coefficients is a good indicator for the amount of data in a specific part of a video file. For each method we will have a nearly proportional map from embedding probability to bpnc.

For each embedding scheme we will extract features with embedding probabilities 0.001, 0.002, . . . , 0.009 and plot the bpnc of p on the x-axis and detectability of p on the y axis. The square root law of steganographic capacity [9] tells us that larger covers require smaller embedding probabilities to ensure steganographic security. This explains the very small embedding probabilities we use in our experiments.

The KL-D has shown to be not computable in practice, see section 5.6 for details.

5 Implementation

There is more than one step involved in the practical portion of this project:

- create a video collection

- exploratory data analysis

- simulate hiding, extract features

- compare different embedding methods

Extraction is performed using a modification of ffmpeg called `ffmpeg-extract`. Before we can extract features we need to determine appropriate ranges for the histograms, this is achieved by visually plotting sets of features in the Stegosaurus GUI program. The GUI is also interface to manage feature sets and launch any calculations. First we will describe the video collection used for our experiments.

5.1 Stegosaurus GUI

The name is rooted in the etymological connection with steganography, figure 21 shows a screenashot. It is base on Qt[7] and gpu accelerated by several CUDA[8] kernels and the cuBLAS[9] (cuda Basic Linear Algebra Subroutines) library.

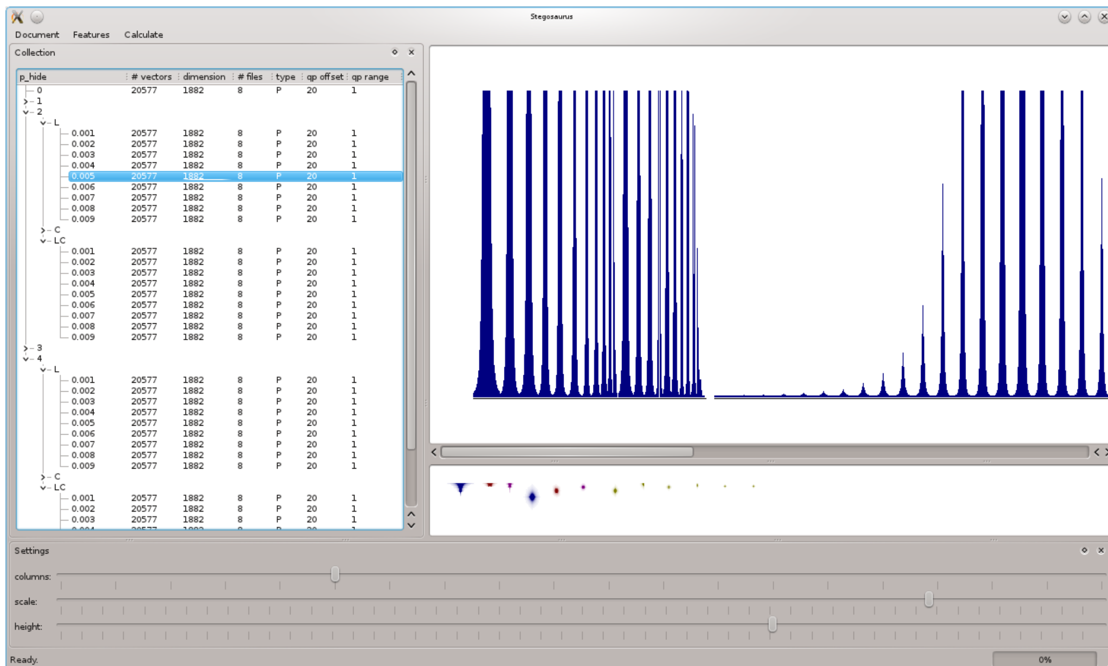


Figure 21: Stegosaurus GUI snapshot. Figure 22 shows the available menu options.

On first level in the `Collection` tree we have a code for the embedding scheme, codes are listed in table 1.

The blue histograms and multicolour shapes below represent the same information, one uses bar height to indicate a certain value and the other uses opacity. Each pixel in the bottom figure has a corresponding bar and its opacity is the square root of the bar height divided by the maximum bar height. The square root gives the picture a smoother look. Figure 23 gives a more detailed view.

[7] http://qt.nokia.com/

[8] http://www.nvidia.co.uk/object/cuda_home_new_uk.html

[9] http://developer.nvidia.com/cublas

	AC+DC			AC		
threshold	1	2	4	1	2	4
± 1	1	2	3	4	5	6
F5	11	12	13	14	15	16
LSB	21	22	23	24	25	26

Table 1: method codes

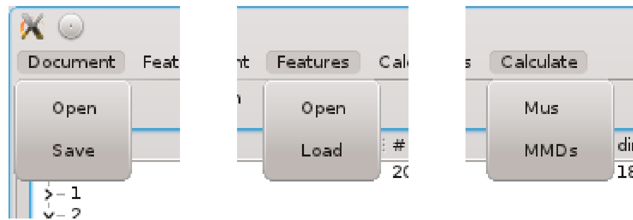


Figure 22: GUI menus

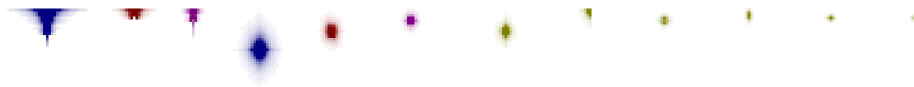


Figure 23: The first three units represent histogram features: L, U and V read from left to right. Row i contains the histogram of the i'th coefficient. Afterwards we have the co-occurence features, again for L, U and V. The left coefficient is on the y axis, that is the reason for the stretch in y direction. The range for the first coefficient is larger than the range for the second one. The remaining dots are UvsV features, one per coefficient with non-zero range.

Initially the feature vector contained multiple QP values, the `columns` slider inside the `Settings` dock define how many QP chunks are shown in each row in the central widgets. The `scale` option defines a linear scaling parameter on the histograms and the `height` defines the maximum shown height of the histograms without changing the scaling. Increasing the height makes the program display more information.

5.1.1 The xml backend

The GUI uses an xml document to store all files containing feature vectors that are available. The document is hardcoded to be `stegodoc.xml` in the current directory, other documents cannot be opened. The xml parsing is done using Qt's DOM API. We can see that the collection is represented by a tree structure to the user which makes the tree-like DOM api a convenient backend. This part of a `stegodoc.xml` file:

Listing 1: stegodoc.xml

```
<stegosaurus>
  <sets>
    <featureSet method="0" qp_range="1" qp_offset="20" type="P">
      <file>/mnt/stego/features/seta/0/p_clean_0_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/setb/0/p_clean_0_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/setc/0/p_clean_0_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/setd/0/p_clean_0_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/sete/0/p_clean_0_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/setf/0/p_clean_0_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/setg/0/p_clean_0_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/seth/0/p_clean_0_minc_1_qp_20.fv</file>
    </featureSet>
    <featureSet method="1" qp_range="1" qp_offset="20" type="P">
      <file>/mnt/stego/features/seta/1/qp_20/p_C_70_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/seta/1/qp_20/p_C_80_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/seta/1/qp_20/p_C_90_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/seta/1/qp_20/p_L_10_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/seta/1/qp_20/p_L_20_minc_1_qp_20.fv</file>
      <file>/mnt/stego/features/seta/1/qp_20/p_L_30_minc_1_qp_20.fv</file>
      ...
    </featureSet>
  </sets>
</stegosaurus>
```

There are 120gb of features divided into 7994 files for each qp offset value, the xml document allows to quickly browse load without having to re-open them every time. Files are inserted into the system by Features → Open which opens a `QFileDialog` that can select and open multiple `.fv` files. They will be inserted into the DOM tree automatically and can be stored to disk via Document → Save. Duplicates are ignored. Features → Load opens a dialog asking which feature sets are to be loaded, see figure 24.

The open feature sets are managed by the `StegoModel`. The `StegoModel` is the backend of the GUI controlling all information displayed including the progress bar. See the Appendix for class definitions.

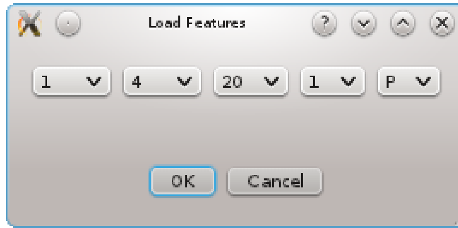


Figure 24: A snapshot of the feature loading dialog. The comboboxes from left to right represent minimum method, maximum method, qp offset, qp range and type (P or B). The corresponding set of clean features is loaded as well.

5.1.2 The .fv file format

Features are stored as unsigned 32 bit integer counts in the feature files. An advantage is that 32 bit integers require only half as much disk space as 64 bit doubles do.

We need a header for Stegosaurus to read feature files properly, these are the header elements and their types:

slice_type [char] Type 0 represent P slices, 1 represents B slices.

method [char] Method code, as described above.

probability [double] Embedding probability, only stored if method $\neq 0$.

accept [char] Channels used for embedding, again only stored if method $\neq 0$. The first bit indicates if luma is being used, the second bit chroma DC and the third bit chroma AC. Therefore we have 1 representing luma only, 6 representing chroma only and 7 representing both.

qp_offset [char] We use 20 as qp offset in our experiments.

qp_range [char] We use single qp, i.e. range 1, in our experiments. Increasing the range not only increases the dimensionality but also the number of vectors that are extracted since it gets less likely to have a zero count vector in an IDR section.

ranges [unsigned char] The histogram ranges for each individual coefficient.

The dimension is not stored explicitly but all information is given that is required to compute the features dimensionality. Also the number of vectors is not included, this is because we want the file to be extendable by new vectors without having to change the header. The file needs to be scanned once to determine the number of contained feature vectors.

Figure 25 shows the ranges that we use in our experiments, they have been set by visually exploring mean vectors in the Stegosaurus program. We can calculate the dimension of each part of the feature vector:

histograms We need to sum up the ranges, multiply by two to get positive and negative part and add 1 for the zeros in each interval: $2 \cdot (15+11+10+3 \cdot 8+2 \cdot 5+3 \cdot 3)+16 = 168$ for luma, $2 \cdot (8+2 \cdot 6+4)+4 = 52$ for chroma DC and $2 \cdot (2 \cdot 4+3 \cdot 3)+15 = 49$ for chroma AC, all adding up to 269.

co-occurrences We need to multiply the interval length of the first and second coefficient for each channel, we have $31 \cdot 23 = 713$ for luma, $17 \cdot 13 = 221$ for chroma DC and $9 \cdot 9 = 81$ for chroma AC, adding up to 1015.

UvsV This calculation is similar to the co-occurrence case, but we proceed chroma coefficient- and not channel-wise. We have $\underbrace{(2 \cdot 8+1)^2}_{DC}$ and $\underbrace{2 \cdot (2 \cdot 4+1)^2 + 3 \cdot (2 \cdot 3+1)^2}_{AC}$, summing up to 598.

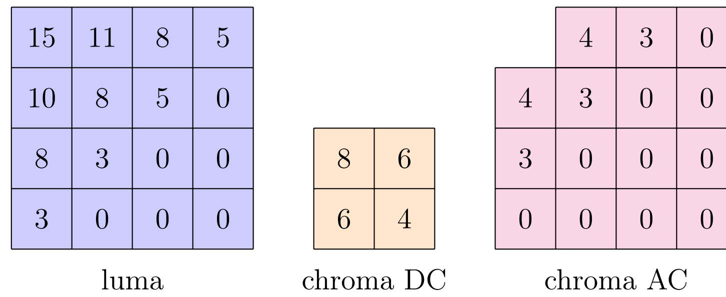


Figure 25: The ranges we used in our experiments. Value r represents range $[-r, r]$. Values get stored in zig-zag scanorder.

The total dimension is 1882.

5.1.3 Normalization

Prior to any normalization technique we need to scale counts appropriately. One vector of counts might be a multiple of another count vector, we consider such vectors to be identical. We scale counts so that the sum of all components equals the dimension, each component is one on average. This allows us to change the dimension without having to adjust the scaling in the GUI each time. We have implemented two normalization techniques:

Rescaling The minimum component, which is typically zero in our case, is mapped to zero, the maximum component is mapped to one. Values in between are linearly interpolated. For component c we gat rescaled value $c' = (c - min)/(max - min)$.

Standardizing We first find the mean of all raw feature vectors. On an input vector we subtract the mean and multiply each component afterwards so that we have variance one in every component. The mean of standardized vectors is zero.

All parameters, these are minimum / maximum values, mean vector and so on, that are needed for normalization are found on basis of the clean set only. It is important that we use the same parameters across both clean and stego sets, otherwise we might lose a steganographic impact.

The purpose of normalization is to treat every component of the feature vector equally. We have very non-uniform data, the snapshot in figure 21 shows non-normalized data. Changes in large coefficients that occur very rarely will not have a large impact on the shape of the vector, but they should not be treated with less significance than changes in smaller coefficients.

Both normalization techniques result in similar results, all results reported were computed with rescaled features.

Normalization is performed on the GPU. All reading methods read into GPU memory only. The system memory requirement of Stegosaurus is very low.

5.2 The Video Collection

The videos available are a transcoded set of 61 DVDs, encoded in different bitrates and with/without denoising filters applied to them. They were transcoded using the following script:

Listing 2: encode.sh (raw)

```
#!/bin/sh
# Usage: encode.sh <bitrate> <file>
output="../../raw/$1/dvd_$2_cavlc_$1.avi"

ffmpeg -y -pass 1 -i $2 -vcodec libx264 -x264opts cabac=0
  -b:v $1 -deinterlace -aspect 16:9 -an -threads 2 $output
ffmpeg -y -pass 2 -i $2 -vcodec libx264 -x264opts cabac=0
  -b:v $1 -deinterlace -aspect 16:9 -an -threads 2 $output
```

Listing 3: encode.sh (denoising)

```
#!/bin/sh
# Usage: encode.sh <bitrate> <file>
output="../../denoised/$1/dvd_$2_cavlc_$1.avi"

ffmpeg -y -pass 1 -i $2 -vcodec libx264 -x264opts cabac=0
  -b:v $1 -deinterlace -vf scale=1024:576,hqdn3d,unsharp
  -aspect 16:9 -an -threads 2 $output
ffmpeg -y -pass 2 -i $2 -vcodec libx264 -x264opts cabac=0
  -b:v $1 -deinterlace -vf scale=1024:576,hqdn3d,unsharp
  -aspect 16:9 -an -threads 2 $output
```

Of these 61 videos then 16 were picked so that the number of contained feature vectors was roughly 10 times the dimension to aid stable results, these were the videos containing fewest vectors, we work with a set of 20577 vectors. Extractions were made with bitrates 1000k, 2000k and 3000k for both raw and denoised video but we only had enough time to process one set of videos, the 3000k raw encodings.

Before we come to the feature extraction we will explain the syntax of an h.264 video.

5.3 H.264 Syntax

Formally a video bitstream is not stored in a file but transmitted over a Network Abstraction Layer (NAL) which contains different NAL units. A frame is divided into slices, this allows the decoder to decode parts of the Frame on different processors. An IDR (Instantaneous Decoder Refresh) slice signals the decoder to refresh itself, Motion Vectors cannot point behind a Frame that contains an IDR-Slice.

The details of the parameters or the slice Header are not important here, but we will explain the macroblock header elements:

MB Type Defines type (I, P or B) and partition of the macroblock.

Prediction Contains prediction mode(s) in case of an I-macroblock and motion vectors otherwise. Sub-partitions are defined here.

Coded Block Pattern Often complete 4x4 blocks can be omitted because they do not contain any non-zero coefficients, these blocks are listed here.

ΔQP Each macroblock is allowed to change its QP.

Figure 26 illustrates the syntax elements.

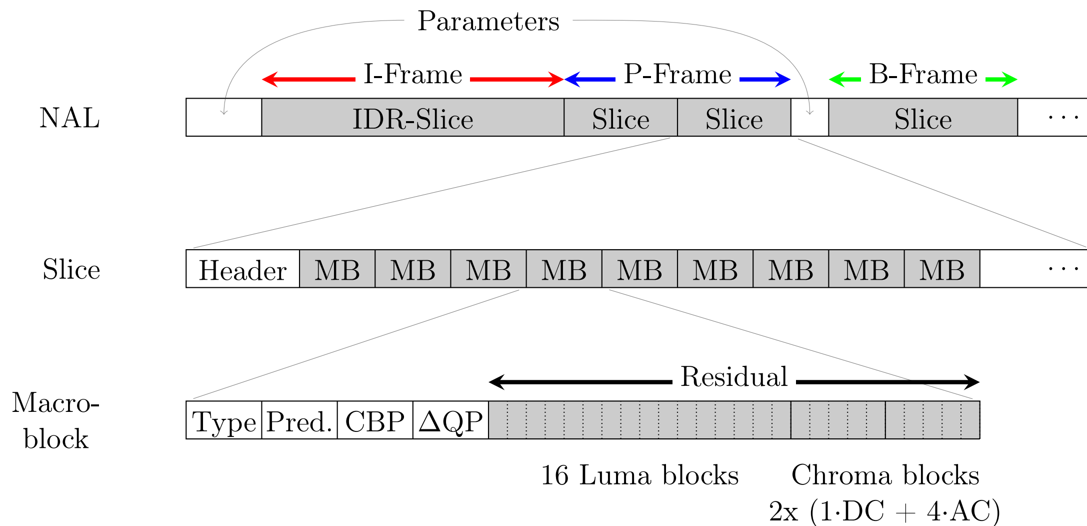


Figure 26: Syntax of the H.264 Bitstream.

5.4 Feature Extraction

The extraction takes place in the `decode_residual` method in the *h264_cavlc.c* file in ffmpegs `libavcodec` library. Extraction always happens when an h.264 video is decoded, we transcode the input videos into a computationally cheaper codec (MPEG-1) to launch the extraction. This is the extraction script:

Listing 4: extract.sh

```
#!/bin/sh
#extract.sh <file>
/home/andy/ffmpeg-extract/ffmpeg -y -i $1 -vcodec mpeg1video -threads 1 output.mpg
```

All parameters are hardcoded, we need to change code and re-compile if we want to change embedding methods or any other parameters.

In addition to the above the extraction program outputs video statistics as well, that is the number of vectors extracted from each file and the average bpnc. The different logfiles can easily be concatenated by using bash functionality. This is part of an example concatenated logfile:

Listing 5: logs_qp_20 (for the first two DVDs using method 1)

```
[qp_20, C, p = 0.001] average p_bpnc: 0.000181641 --- 1549 vectors.
[qp_20, C, p = 0.001] average b_bpnc: 0.000262209 --- 1213 vectors.

[qp_20, C, p = 0.001] average p_bpnc: 0.000211074 --- 1402 vectors.
[qp_20, C, p = 0.001] average b_bpnc: 0.000379492 --- 1202 vectors.

[qp_20, C, p = 0.002] average p_bpnc: 0.000361263 --- 1549 vectors.
[qp_20, C, p = 0.002] average b_bpnc: 0.000514996 --- 1213 vectors.

[qp_20, C, p = 0.002] average p_bpnc: 0.000488388 --- 1402 vectors.
[qp_20, C, p = 0.002] average b_bpnc: 0.000780432 --- 1202 vectors.

[qp_20, C, p = 0.003] average p_bpnc: 0.000547533 --- 1549 vectors.
[qp_20, C, p = 0.003] average b_bpnc: 0.000790526 --- 1213 vectors.

...

[qp_20, C, p = 0.009] average p_bpnc: 0.00243165 --- 1402 vectors.
[qp_20, C, p = 0.009] average b_bpnc: 0.00366531 --- 1202 vectors.

[qp_20, LC, p = 0.001] average p_bpnc: 0.000931314 --- 1549 vectors.
[qp_20, LC, p = 0.001] average b_bpnc: 0.00081836 --- 1213 vectors.

[qp_20, LC, p = 0.001] average p_bpnc: 0.000904551 --- 1402 vectors.
[qp_20, LC, p = 0.001] average b_bpnc: 0.000697173 --- 1202 vectors.

[qp_20, LC, p = 0.002] average p_bpnc: 0.00199814 --- 1549 vectors.
[qp_20, LC, p = 0.002] average b_bpnc: 0.00197204 --- 1213 vectors.

...

[qp_20, L, p = 0.008] average p_bpnc: 0.00603218 --- 1402 vectors.
[qp_20, L, p = 0.008] average b_bpnc: 0.00461397 --- 1202 vectors.

[qp_20, L, p = 0.009] average p_bpnc: 0.00737204 --- 1549 vectors.
[qp_20, L, p = 0.009] average b_bpnc: 0.00655301 --- 1213 vectors.

[qp_20, L, p = 0.009] average p_bpnc: 0.00677329 --- 1402 vectors.
[qp_20, L, p = 0.009] average b_bpnc: 0.00555186 --- 1202 vectors.
```

We used the weighted average of these bpnc values, weighted according to the number of vectors they represent, to construct the bpnc → MMD maps. We only computed the weighted average for $p = 0.009$ and interpolated the remaining values since the bpnc has shown to be practically linear for the numbers of vectors we worked with. It was not

feasible in the time given to compute the weighted average for every probability value.

5.5 Embedding simulation

We now look at the implementation of the embedding simulation, this is the code implementing ±1 embedding:

Listing 6: ±1 embedding simulation

```
void simulate_hiding_plusminus(H264FeatureContext* fc, int blocknum, int thresh) {
  int i;
  int *coefs = fc->tape;
  int min;
  double r;
  int sl = fc->slice_type;

  // check if we are on correct channel
  if (!(fc->accept_blocks & (1 << blocknum))) {
    return;
  }

  // work out the correct starting point in array
  switch (blocknum) {
    case 0:
      min = MIN_COEF-1;
      break;
    case 1:
      if (MIN_COEF > 1)
        return; // we only have first coefficients here
      min = 0;
      break;
    case 2:
      min = max(MIN_COEF-2, 0);
      break;
  }

  for (i = min; i < num_coefs[blocknum]; i++) {
    // check range
    if (coefs[i]<thresh && coefs[i]>-thresh) continue;
    r = (((double) rand()) / ((double) RAND_MAX));
    if (r < fc->p_hide) {
      // statistics used for bpnc calculation
      switch (sl) {
      case TYPE_P_SLICE:
        fc->hidden_bits_p++;
        break;
      case TYPE_B_SLICE:
        fc->hidden_bits_b++;
        break;
      }
      r = (((double) rand()) / ((double) RAND_MAX));
      if (r < 0.5) continue; // half of the coefficients don't need to be changed
      if (coefs[i] == thresh) {
        coefs[i]++;
      } else if (coefs[i] == -thresh) {
        coefs[i]--;
      } else{
        r = (((double) rand()) / ((double) RAND_MAX));
        if (r < PROB_INCREASE) { // if changing, increase or decrease?
          coefs[i] += 1;
        } else {
```

```
          coefs[i] -= 1;
        }
      }
    }
  }
}
```

The implementations of the different embedding methods only differ at the very end where the actual embedding is performed and in the range checking, we will therefore give the for loop only for these functions:

Listing 7: F5 embedding simulation

```
void simulate_hiding_f5(H264FeatureContext* fc, int blocknum, int thresh) {
  ...
  for (i = min; i < num_coefs[blocknum]; i++) {
    if (coefs[i]<=thresh && coefs[i]>=-thresh) continue;
    r = (((double) rand()) / ((double) RAND_MAX));
    if (r < fc->p_hide) {
      switch (sl) {
      ...
      }
      r = (((double) rand()) / ((double) RAND_MAX));
      if (r < 0.5) continue; // half of the coefficients don't need to be changed
      if (coefs[i] < 0) { // if changing, increase or decrease?
        coefs[i] += 1;
      } else {
        coefs[i] -= 1;
      }
    }
  }
}
```

Listing 8: LSB embedding simulation

```
void simulate_hiding_lsb(H264FeatureContext* fc, int blocknum, int thresh) {
  ...
  for (i = min; i < num_coefs[blocknum]; i++) {
    if (coefs[i]<2*thresh && coefs[i]>=-2*thresh+2) continue;
    r = (((double) rand()) / ((double) RAND_MAX));
    if (r < fc->p_hide) {
      switch (sl) {
      ...
      }
      r = (((double) rand()) / ((double) RAND_MAX));
      if (r < 0.5) continue; // half of the coefficients don't need to be changed
      coefs[i] = coefs[i] ^ 1;
    }
  }
}
```

The threshold is interpreted differently for each embedding method, table 2 shows intervals of coefficients that are not used for embedding.

The `H264FeatureContext` contains the current feature counts as well as the statistics needed to calculate bpnc values, details are given in the appendix.

threshold	± 1	F5	LSB
1	$\{0\}$	$[-1, 1]$	$[0, 1]$
2	$[-1, 1]$	$[-2, 2]$	$[-2, 3]$
3	$[-2, 2]$	$[-3, 3]$	$[-4, 5]$
4	$[-3, 3]$	$[-4, 4]$	$[-6, 7]$
5	$[-4, 4]$	$[-5, 5]$	$[-8, 9]$

Table 2: Unused coefficients for each embedding method depending on threshold. We see that the interval size for LSB embedding is asymmetric and increasing most quickly.

5.6 KL-D

We first need to estimate the mean μ and the covariance matrix Σ from the extracted feature vectors X. Let N be the number of vectors, then we have:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$
$$\Sigma = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)(x_i - \mu)^T$$

We need to invert Σ to calculate a KL-D estimate, as described in section 4.5.1. We want to work as precisely as possible, an approximated solution is not satisfactory, therefore we decided to use the Housholder QR factorization to find the inverse. A QR factgorization of matrix A is a decomposition of A into the product $Q \cdot R$ where Q is orthogonoal and R is upper-triangular. A Householder matrix is of the following form:

$$H = I - 2\frac{ww^T}{w^T w}$$

On input vector v the part of v in direction of w will change sign, the rest will be unchanged. We can immediately see that there are two eigenvalues, 1 and -1, -1 having only w as eigenvector. Also H is symmetric. If we use w_0 such that $w_0^T w_0 = 1$ we get an orthonormal instead of just an orthogonal matrix H, since:

$$H \cdot H^T = H \cdot H = (I - 2w_0 w_0^T)(I - 2w_0 w_0^T)$$
$$= I - 4w_0 w_0^T + 4w_0 w_0^T = I$$

Starting with matrix Σ in the first step we want to transform the first column vector to a multiple of the first unit vector using a Householder transformation. The corresponding

w can easily be found, as illustrated in figure 27. The transformation then has to be applied to the entire Σ matrix. In the next step the dimension is reduced by one, the Householder matrix will have the following shape:

$$H_2 = \begin{pmatrix} 1 & \\ & \widetilde{H}_2 \end{pmatrix}$$

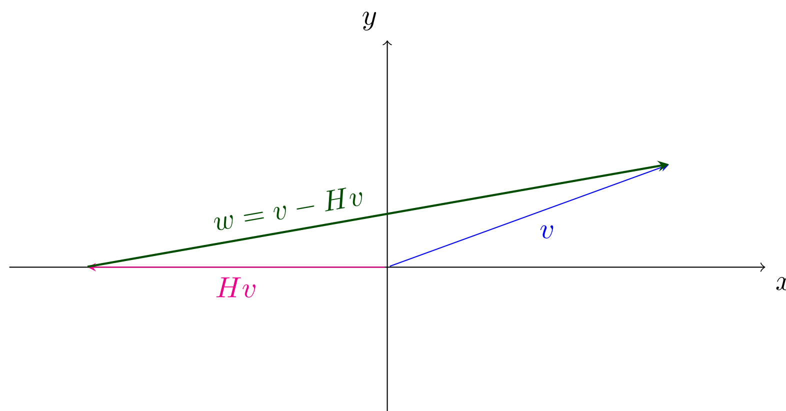


Figure 27: Construction of w such that input vector v is mirrored on a multiple of a unit vector. Householder transformations do not change the norm of the input vector, thus the outcome is known to be $-\|v\| \cdot e_x$. We will always want the sign of component x to change for more stable results. If v was very close to the x-axis we could encounter problems with numerical precision.

We next apply H_2 to the entire matrix and so on. In the end we have $Q = H_D H_{D-1} \ldots H_1$ where D is the dimension of each feature vector. We do not construct Q explicitly, instead we replace the zeros in the upper triangular matrix R with the corresponding w_0 vectors. The diagonal elements have to be removed from the matrix and are stored in a separate array. This way QR factorization can be computed in place.

The determinant of a Householder matrix is -1 since the determinant is the product of eigenvalues. We have an even number of dimensions, therefore the determinant of Σ is the product of R's diagonal entries which we have stored in a separate array. It turns out that especially towards the end the entries get very small, leading to a practically zero product, Σ is not invertible, therefore we cannot compute the KL-D this way. This holds for all normalization techniques we have implemented.

If we reduced the feature set by removing all features that are always or nearly always zero and linearly dependent features this approach might work. There is an entirely different approach to estimate the KL-D of two sets of points based on k nearest neighbours (kNN) [7] but these are shown to be very unstable.

5.7 MMD

GPUs provide massively parallel computing units that achieve more FLOPS compared to CPUs in general, but peak performance on GPUs is only achieved under certain conditions. Threads on the GPU are organized into three dimensional blocks and blocks are organized in a three dimensional grid. A number of threads is executed simultaneously, but they all have to perform the same operations at any given time. If there is a branch, for example if the first half of the threads satisfies an `if` condition and the second half does not, the first half will be executed simultaneously and afterwards the second half as well [10]. For best performance we want to minimize branching.

The performance of GPU global memory depends on the way it is accessed, reading consecutive blocks is fast but performance decreases to a small fraction if random locations in GPU global ar accessed [11]. There are different levels of memory in a GPU, there is the global memory which is the largest and slowest, then there is shared memory which is shared between threads within the same block and local memory accessable by individual threads only. There are other types available, but we do not use them in this project, see [10, 11, 12] for details.

Coming back to our implementation task, this is the equation we need to calculate as seen in section 4.5.2:

$$\mathrm{MMD}(\mathcal{F}, X, Y) = \sqrt{\frac{1}{N(N-1)} \sum_{i \neq j} k(x_i, x_j) - 2k(x_i, y_j) + k(y_i, y_j)}$$

$$\text{where} \qquad k(x, y) = \exp(-\gamma \|x - y\|^2)$$

We could use cuBLAS to accelerate the kernel computation, but we do not achieve best performance that way. If we were to implement the vector sum on the GPU we would launch one thread per element for the first half of the vector and compute the sum with another element in the second half. This can be repeated until there is only one element left which is the result [10]. We see that this implementation does not achieve peak performance since the number of busy threads decreases over time. We assume that cuBLAS uses a similar implementation. The dimension of the feature vector is relatively small, each CUDA kernel launch comes with an overhead, decreasing performance further for the cuBLAS-based implementation.

Instead we want to launch a thread for each pair of vectors (x, y) and let it compute $\|x - y\|^2$ by summing up sqaures of $x_i - y_i$ sequentially as we would do on the CPU. We have to partition the feature vector into partitions since there is a timeout on kernel executions. This computation is the first step in both the estimation of γ as well as the calculation of the MMD itself. For the MMD calculation we need to multiply with $-\gamma$ and calculate the exponential afterwards. Since GPU global memory is very limited we need to work block-wise and launch the kernel multiple times in order to cover any vector pair. γ is estimated on the clean set only, $\|x - y\|^2 = \|y - x\|^2$, we only need to look at half of the pairs there.

We need to cover three combinations, clean x clean, clean x stego and stego x stego. We compute the differences of the three directly to increase numerical stability, if we

were to sum up all results for clean x clean and the other two cases separately and find the difference afterwards we may lose precision. Doubles are represented by mantissa and exponent ans lose accuracy in absolute terms as the absolute value of a number increases [12].

Figure 28 illustrates the computation of the MMD estimate.

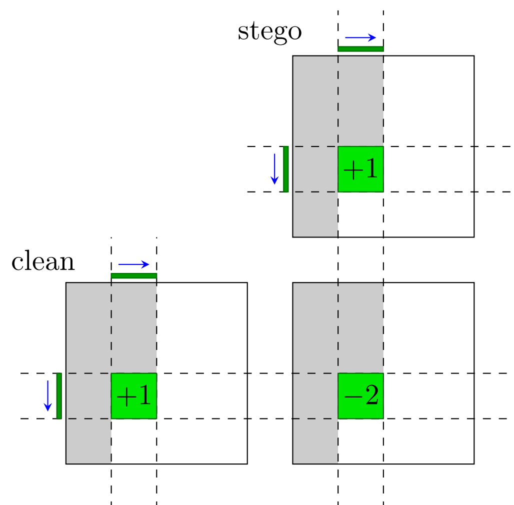


Figure 28: Our MMD implementation. The green parts are buffers on the GPU where each dark green unit represents an array of features vector while the light green squares are two dimensional arrays of doubles. We can imagine this picture to have depth D where D is the feature dimension and the light green parts scan through the vectors towards or away from us. Both right and down moving dark green arrays are always in the same position.

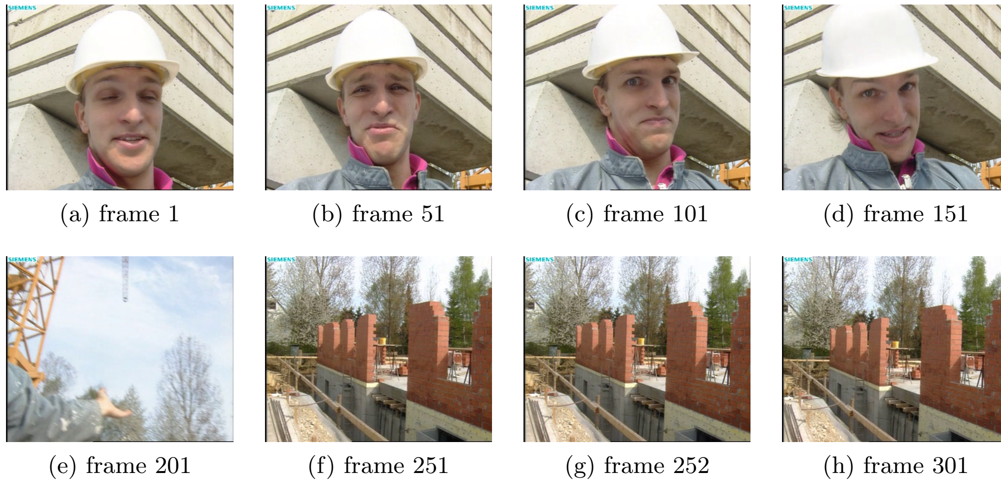
6 Experimental Results

6.1 Visual Distortion

For visible effects we need to maximize impact, set the threshold and minimum coefficient as well as the hiding probability to one. Now we present a series of snapshots of the qp 20 Foreman encoding done by `x264`. Frame number 252 is an IDR frame, thus we see a discontinuity there. All types of slices and macroblocks are used for embedding here, when extracting features we consider P and B slices only. The main focus of this project is on feature analysis and not on the examination of visual distortion.

6.2 Feature Analysis

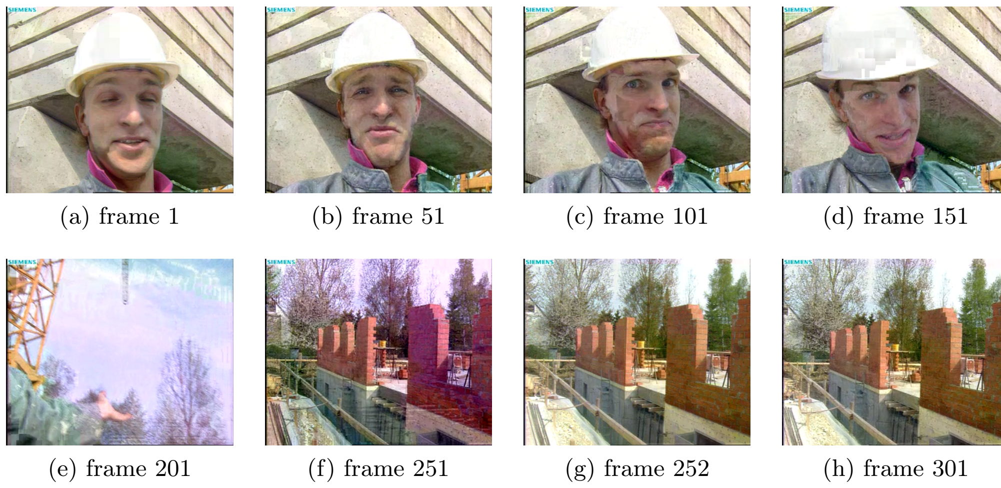
We have a set of features for each (i) embedding method, (ii) embedding probability, (iii) slice type, (iv) channel, (v) threshold value and (vi) use of AC only or AC and DC coefficients. Each point in the graphs below represents the MMD of one such feature set



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

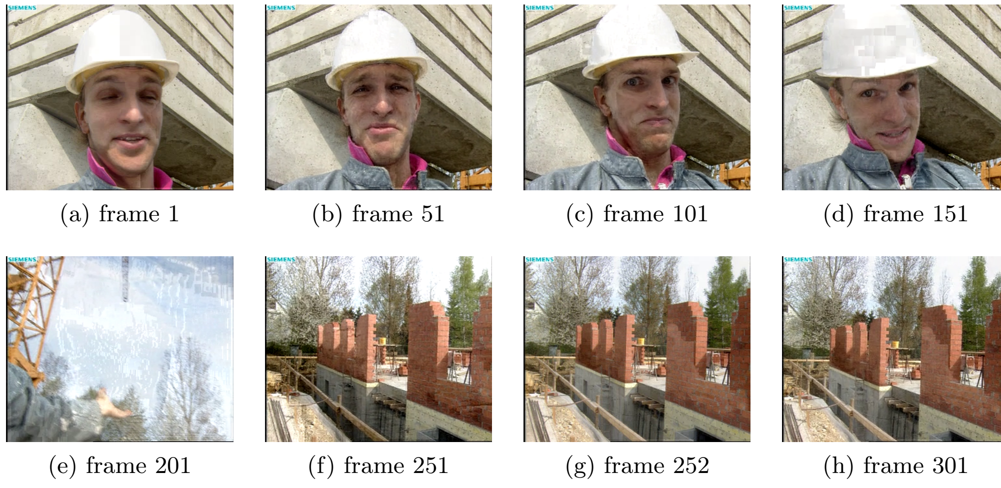
Figure 29: Unmodified Foreman snapshots



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

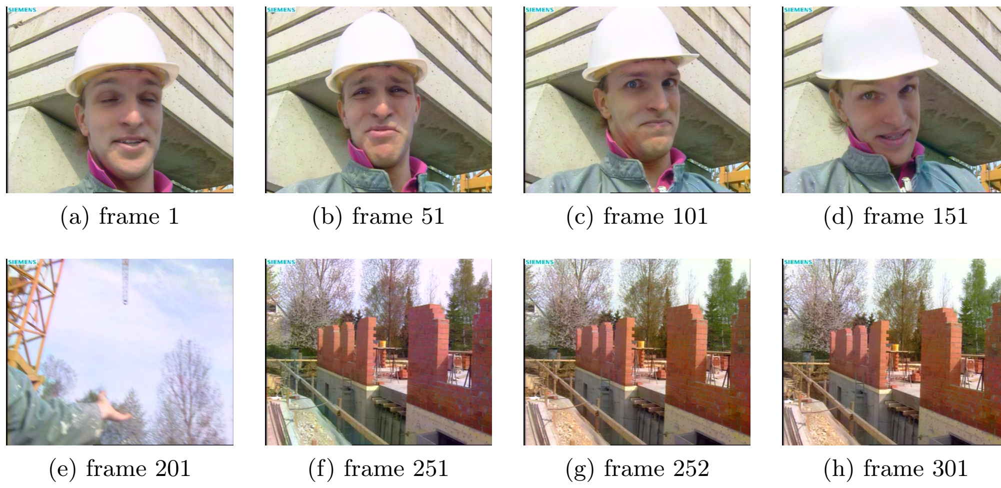
Figure 30: ± 1 embedding in Luma and Chroma



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

Figure 31: ±1 embedding in Luma



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

Figure 32: ±1 embedding in Chroma

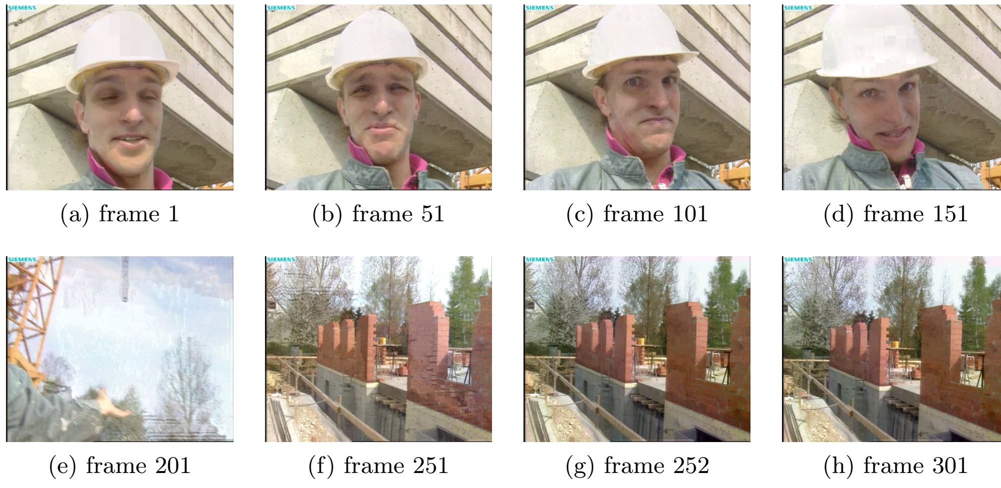

(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151
(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

Figure 33: F5 embedding in Luma and Chroma

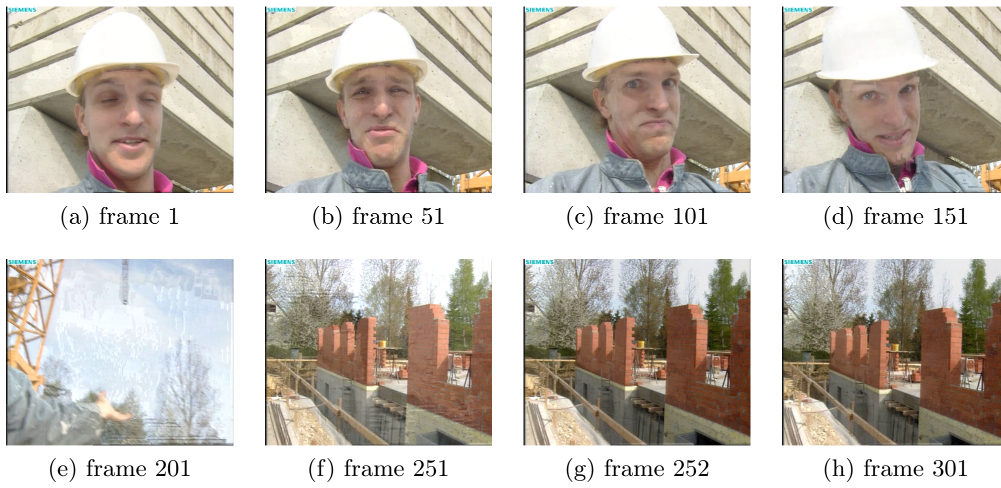
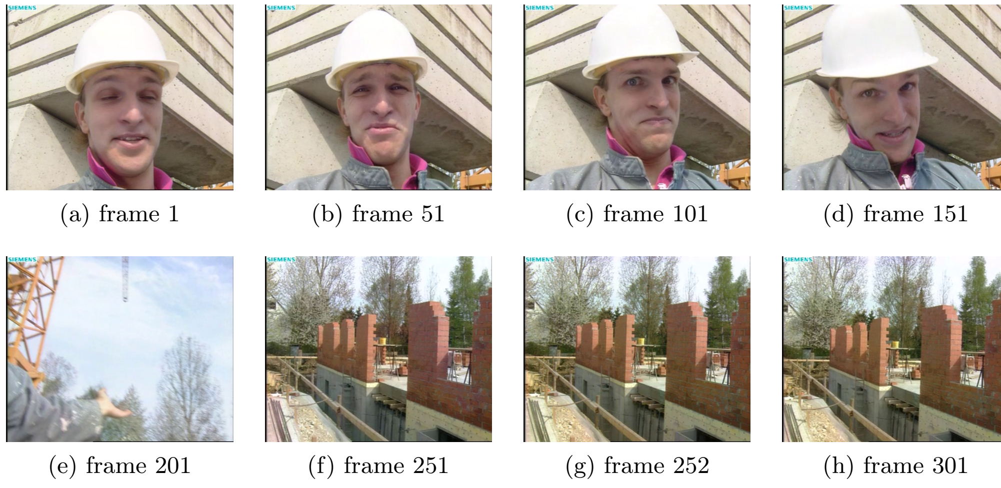

(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151
(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

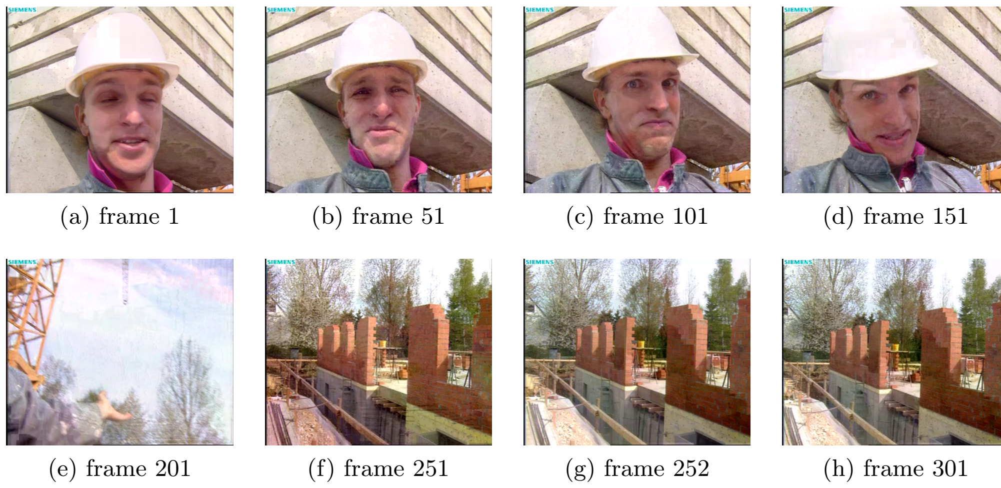
Figure 34: F5 embedding in Luma



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

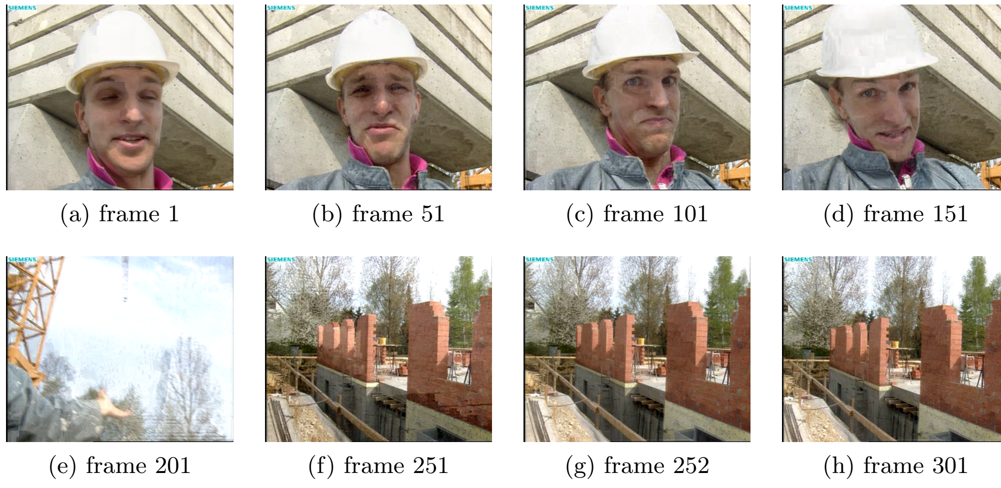
Figure 35: F5 embedding in Chroma



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

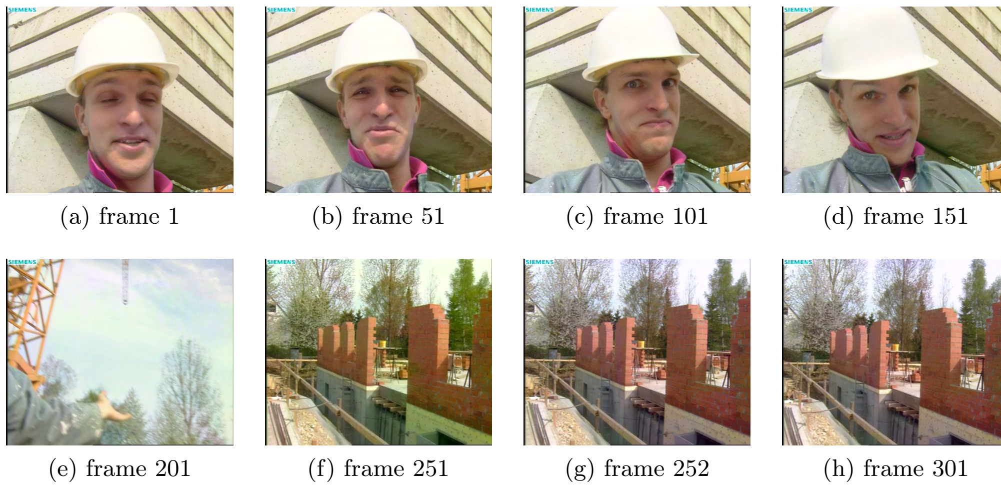
Figure 36: LSB embedding in Luma and Chroma



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

Figure 37: LSB embedding in Luma



(a) frame 1 (b) frame 51 (c) frame 101 (d) frame 151

(e) frame 201 (f) frame 251 (g) frame 252 (h) frame 301

Figure 38: LSB embedding in Chroma

and the corresponding set of clean features. Each feature set has a corresponding average bpnc (bits per non-zero coefficient) value, we investigate the MMD in terms of the bpnc to answer our questions about embedding. Figure 39 shows the MMD as a function of bpnc where the embedding probability ranges from 0.1 to 1. Each point is the result of an MMD calculation on two sets of 20577 vectors with dimension 1882, extracted from 16 DVDs.

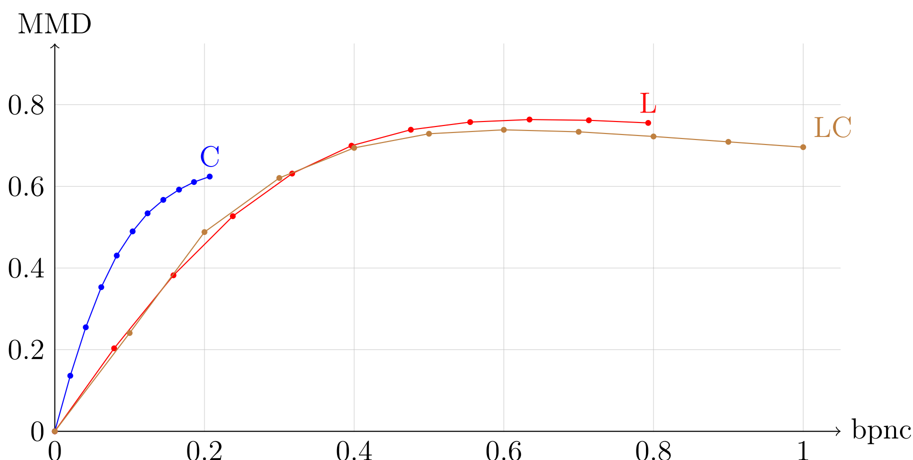


Figure 39: bpnc → MMD for ±1 embedding in AC and DC using threshold 1 (P slices). The embedding probability ranges from 0.1 to 1.

The MMD is locally linear near zero [13], therefore it can be accurately represented by a function of type $y = a \cdot x$ for a small embedding probability. We present the parameter a of the least squares approximation for all graphs we computed in table 3 and we will give full graphs on a few examples to answer our questions about embedding. The embeddin probability in the following experiments ranges from 0.001 to 0.009, in uniform steps of size 0.001.

6.2.1 Which channel is best suited for embedding?

We will look at the detectability three different channel embeddings (luma only, both chroma channels and luma and chroma) ±1 embedding with threshold 1 since it uses all non-zero coefficients for embedding. These are the corresponding MMD values in terms of bpnc for P slices:

We can see that the MMD estimate is very stable and that chroma is least suited for embedding. The UvsV features that are available for chroma only increase detectability in chroma. But we can also see that a combination of luma and chroma is slightly better than using luma alone.

Figure 41 shows the same graph for B slices. We can see that the values are less stable and that using luma alone achieves lowest detectability here from the second sample onwards. We also see that B slices carry more chroma information than P slices do, since we reach a larger bpnc value for chroma. This is why embedding both luma and chroma

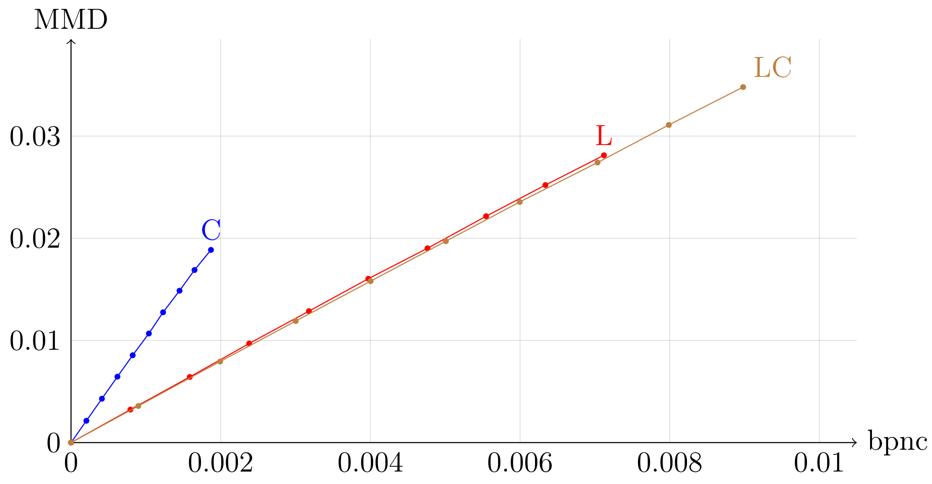


Figure 40: bpnc → MMD for ±1 embedding in AC and DC using threshold 1 (P slices).

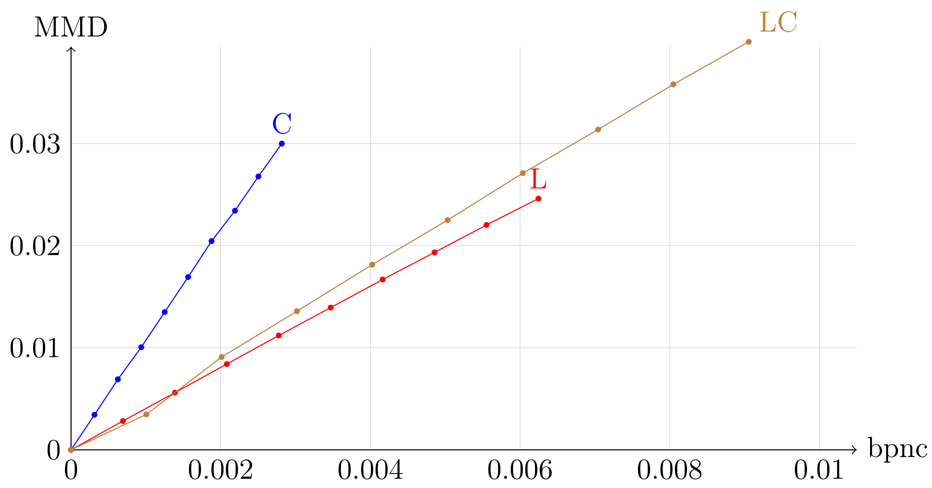


Figure 41: bpnc → MMD for ±1 embedding in AC and DC using threshold 1 (B slices).

is more detectable than in luma alone in this case. If we were to use a lower probability on chroma embedding we could possibly beat luma-alone embedding here as well.

6.2.2 Which embedding method is the least detectable?

We can only compare ±1 embedding with threshold 2 and F5 embedding with threshold 1 directly since they use the same coefficients for embedding, see table 2 for precise range definitions. These are the corresponding graphs:

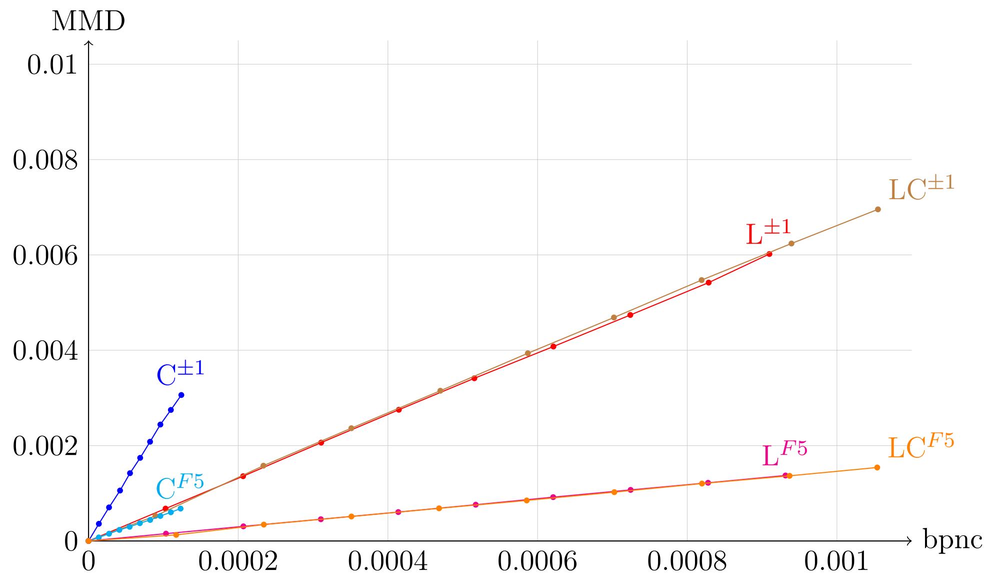


Figure 42: All channels of ±1 embedding with threshold 2 and F5 embedding with threshold 1. The different threshold values are needed for both methods to embed in the same coefficients.

F5 is better performing, chroma embeddings using F5 embedding are less detectable than luma embeddings using ±1 embedding.

We cannot compare the LSB embedding as directly since it is asymmetrical and never uses the same coefficients for embedding as the other methods do. But we can infer from table 3 that it doe not achieve lower growth rates than F5 embedding for thresholds 1 and 2. With threshold 4 the LSB embedding does not yield meaningful results in chroma since there is hardly any embedding at all. F5 is the least detectable among the embedding methods tested. Comparing values at the same threshold LSB embedding out-performes ±1 embedding.

6.2.3 Which coefficients are best suited for embedding?

We can see in table 3 that across all embedding methods we get minimum detectability at threshold 1 and when using both AC and DC coefficients. This tells us that we want to hide in coefficients that occur frequently.

		t.	P L	P C	P LC	B L	B C	B LC
±1	AC+DC	1	3.988	10.230	3.903	3.979	10.729	4.450
		2	6.582	24.795	6.652	7.046	37.170	8.736
		4	11.122	47.643	11.185	15.596	40.352	14.109
	AC	1	7.282	11.747	7.090	8.805	15.977	8.631
		2	11.560	28.002	11.315	15.016	22.831	13.759
		4	19.661	85.349	19.482	25.311	89.874	22.514
F5	AC+DC	1	1.475	5.463	1.460	1.417	4.250	1.446
		2	2.970	10.809	2.885	3.044	10.428	3.243
		4	5.708	24.750	5.505	8.405	22.682	9.033
	AC	1	2.655	7.295	2.526	3.077	6.649	2.975
		2	5.164	12.762	4.937	5.447	14.178	5.572
		4	8.939	15.455	8.872	10.426	59.427	10.974
LSB	AC+DC	1	3.529	6.125	3.416	3.388	8.679	3.643
		2	5.087	20.562	4.952	7.522	21.525	7.353
		4	7.514	29.008	7.242	16.536	39.191	17.574
	AC	1	6.162	12.088	5.841	7.635	14.906	7.385
		2	8.681	21.611	8.483	12.885	21.639	10.454
		4	9.679	0.614	9.061	43.411	0.000	36.668

Table 3: MMD growth rates. The threshold is interpreted differently for each method, see table 2 for precise ranges. A very small number of NaN MMD estimates was ignored in the computation of the least squares approximation. We get a NaN result if the sum under the square root of the MMD estimate is negative.

We observe that for threshold 1 and embedding in the DC coefficient as well P and B slices show similar behaviour, but as the threshold increases or the DC coefficient is not used detectability in B slices increases faster than it does in P slices. We can deduce that coefficients in B slices are generally smaller than coefficients in P slices and higher concentrated in early places in the scan order. Since smaller values and earlier places are preferable for hiding both slice types are equally well suited for embedding.

6.3 Limitations

Our video collection contains very uniform data, the same encoder was used to generate all videos in it with identical parameters, the resolution is constant at 1024x576 (the standard DVD resolution) and DVDs are a high qualitiy source. All these conditions do not apply to real videos that are transmitted online, the noise might make the detection of steganographic embedding more difficult.

We use the CAVLC entropy coding to avoid interference when changing the quantized coefficients. In CABAC the modification of the values might already cause interference, CABAC maintains a probabilistic model of the coefficients to produce codes that are more efficient than CAVLC codes.

We look at the quantized DCT coefficients only but other elements in the macroblock can be used as well, for example the motion vectors or ΔQP.

We simulate the embedding in the compressed video stream, instead we could embed while encoding the video. This would avoid interference with any entropy coder and also minimize visual impact since steganographic distortion is used for predictions made by the encoder as well.

7 Conclusion

We have implemented several embedding mechanisms for h.264 video files, proposed a set of features useful to detect if a video is modified and evaluated a benchmark on a large video collection. We started with ffmpeg version 0.8 ("Love") as codebase to extract the proposed features and used a GUI program named Stegosaurus for exploratory data visualization and analysis. Stegosaurus manages feature vectors located in different files and can apply different normalization techniques to the input data which is an integer vector of occurence counts. All Calculations are launched graphically and are GPU accelerated. ffmpeg was used to create the video collection, taking x264 to transcode a set of DVDs.

P and B slices were treated separately to which of both is better suited for embedding. We have used two different methods for benchmarking, the (i) KL-D which is an information-theoretic measure of similarity of two probability distributions and the (ii) MMD which is related to the performance of an SVM on the two input data sets. The KL-D estimate was not computable in practice but the MMD estimate has shown to very stable and linear as expected [7]. We have implemented three different embedding methods, the (i) ± 1, (ii) F5 and LSB embeddings. For each method we have extracted features with different lower thresholds on the coefficient values as well as with using

DC and AC coefficients or AC coefficients only for embedding. Also each extraction was done on chroma only, luma only or both.

Across all embedding methods we have observed similar behaviour when varying the parameters or channels, we make the following conclusions:

1. The F5 method is the least detectable among those tested. The LSB embedding is slightly less detectable than the ± 1 embedding.
2. Embedding in the chroma channel is most detectable. Using luma alone or luma and chroma leads to similar detectability where generally P slices are better suited for luma and chroma embedding while luma only embeddings are less detectable in B slices. Including the DC coefficient decreases detectability, especially in B slices since these store a larger part of their data in DC coefficients. P and B slices behave similarly at low threshold and if both AC and DC coefficients are used for embedding.

To our knowledge nobody has used the KL-D or MMD as detectability measures in video steganography before, most papers work on the basis of visual distortion. We have proposed, implemented and tested a new feature set which uses both luma and chroma information. In most papers about video or image steganography only the luma channel is being used, our experiments have shown that embedding in only luma does not achieve minimum detectability in our features. We have used h.264 for our tests which is a very popular video format at the time of writing, it is used on blu ray discs widely spread on the internet, it is used by YouTube[10] and other video sharing websites.

Our project uses very similar data, all videos are dvd movies, transcoded with the same encoder and same parameters. Real data retrieved online will contain a lot more noise introduced by different encoders, qualitiy settings, cameras, video filters and so on. We use the CAVLC entropy coder which is generally out-performed by CABAC, therefore most online media and blu rays use CABAC. We use CAVLC to avoid interference with the CABAC entropy coder.

7.1 Future Work

There is a lot of scope for future work in the implementation, the functionality of the GUI is rather limited, we used other software as well to evaluate the experiments, especially a spreadsheet to compute the correct bpnc values for each extraction. We would want Stegosaurus to calculate bpnc's automatically and let the user browse through the results of previous computations. This would also require a more structured backend, such as a database storing all important information about the feature files or possibly the features directly which would make much larger sets of features tractable.

It would be interesting to see if an actual detector confirms the distance meaurements we have found. This detector can be an SVM or a simpler classifier such as an averaged perceptron.

[10]http://www.youtube.com/

7.2 Personal Report

We had never heard of steganography before we read the project description, neither had we ever worked in the field of video coding before. However the idea of hiding data in video sounded stupendous so that we started to read books and online media about video compression and the h.264 codec. We attempted to implement 4:2:2 colour space support into x264[11] to get familiar with the algorithms' details as well as the code layout, x264 was the largest program we had modified at that time. It was a new experience to set long term-goals over months for a project that would take over one and a half years.

It was interesting to read about information theory and machine learning, especially as we had a collection of applications in mind. The experiments carried out in this project are on a different scale than experiments we have done before, in terms of computational demand and data load in both input and output. We use two machines for the experiments, one having a strong CPU (AMD 8 core) for the feature extraction and the other one having a strong GPU (Nvidia Geforce GTX 590[12]) and large amount of disk space (9TB RAID) for the MMD calculations. Both were running constantly for dozens of hours doing calculations or copying data. Planning the hardware required and setting up calculations on this scale was new to us. Now we look forward to write a scientific publication about the same topic.

References

[1] Jessica Fridrich. *Steganography in Digital Media*. Cambridge University Press, 2010. ISBN: 978-0-521-19019-0.

[2] Iain E. Richardson. *The H.264 Advanced Video Compression Standard*. John Wiley and Sons Ltd, 2010. ISBN: 978-0-470-51692-8.

[3] Telecommunication and Standardization Sector of ITU. *H.264*. 2009.

[4] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. 2nd ed. John Wiley and Sons Ltd, 2006. ISBN: 978-0-471-24195-9.

[5] Andreas Westfeld and Andreas Pfitzmann. 'Attacks on Steganographic Systems'. In: *Information Hiding*. Ed. by Andreas Pfitzmann. Vol. 1768. Lecture Notes in Computer Science. 10.1007/10719724_5. Springer Berlin / Heidelberg, 2000, pp. 61–76. ISBN: 978-3-540-67182-4. URL: `http://dx.doi.org/10.1007/10719724_5`.

[6] Jessica Fridrich et al. *Quantative steganalysis of digital images: estimating the secret message length*. 2003.

[7] T. Pevný and J. Fridrich. 'Benchmarking for Steganography'. In: *Information Hiding, 10th International Workshop*. Ed. by K. Solanki. Lecture Notes in Computer Science. Santa Barbara, CA: Springer-Verlag, New York, 2008.

[8] Tomáš Pevný. 'Kernel Methods in Steganalysis'. PhD thesis. Binghamton University, SUNY, 2008.

[11] Which has been accomplished by the developers at the time we finish this report.
[12] http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-590

[9] Andrew D. Ker et al. 'The Square Root Law of Steganographic Capacity'. In: *10th ACM Workshop on Multimedia and Security*. 2008, pp. 107–116.

[10] Jason Sanders and Edward Kandrot. *CUDA by Example*. Morgan Kaufmann Publishers, 2011.

[11] Rob Farber. *CUDA Application Design Development*. Morgan Kaufmann Publishers, 2011.

[12] David B Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann Publishers, 2010.

[13] Tomáš Pevný. *MMD in the Context of Ker's Benchmark*. Personal Communication.

8 Appendix: Source Code

We now present more source code snippets that implement certain functionalities. First we introduce the structs and classes that organize the backend behind the GUI and afterwards we will show the code implementing the featrue normalization and MMD calculation.

8.1 ffmpeg-extract

This is the header file introducing all structures that are used for feature extraction. Extraction is written in plain C.

Listing 9: libavcodec/ffmpeg_extract.h

```
#ifndef AVCODEC_H264_EXTRACT
#define AVCODEC_H264_EXTRACT

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define METHOD                  1 // 0 = clean, 1=pm
#define QP_RANGE                1
#define QP_OFFSET               16
#define QP_DELTA                4
#define QP_JUMPS                4
#define TYPE_I_SLICE            2
#define TYPE_P_SLICE            0
#define TYPE_B_SLICE            1
#define PROB_INCREASE           0.5
#define ACCEPT_L                1
#define ACCEPT_LC               7
#define ACCEPT_C                6
#define PROB_DELTA              0.001
#define STEGF                   9
#define THRESHOLD               1
#define MIN_COEF                1   // this counts from 1, coef 1 is DC coef

#define max( a, b ) ( ((a) > (b)) ? (a) : (b) )

typedef int feature_elem;
typedef uint32_t store_elem;

static const char *blockstrings[8]
  = {"clean", "L", "C_dc", "LC_dc", "C_ac", "LC_ac", "C", "LC"};
static const char num_coefs[3]
  =  {16, 4, 15}; // Luma, Cr DC, Cb DC, Cr AC, Cb AC
static const unsigned char ranges[3][16]
  =  {{15,  11, 10,  8, 8, 8,  5, 5, 3, 3,  0, 0, 0,  0, 0,  0},
     {8,  6, 6,  4,   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
     {4,  4,  3, 3, 3,  0, 0, 0, 0,  0, 0, 0,  0, 0,  0,  0}};

typedef struct H264FeatureVector {
  int vector_num;
  store_elem *****histograms;  // [slice_type][qp][block][coef][element]
  store_elem *****pairs;  // [slice_type][qp][block][element_left][element_right]
  store_elem *****uvsv;   // [slice_type][qp][coef][element_u][element_v]
                          // coef=0 is DC, coef in [1..16) AC
```

```
  int vector_histograms_dim;
  int vector_pairs_dim;
  int vector_uvsv_dim;
  store_elem *vector_histograms;
  store_elem *vector_pairs;
  store_elem *vector_uvsv;
} H264FeatureVector;

typedef struct H264FeatureContext {
  int qp;
  int *tape;
  int slice_type;
  int refreshed;
  uint64_t hidden_bits_b;
  uint64_t hidden_bits_p;
  uint64_t num_coefs_b;
  uint64_t num_coefs_p;
  uint64_t num_vectors_b;
  uint64_t num_vectors_p;
  double bpnc_b;
  double bpnc_p;
  char *logName;

  // simulation parameters:
  int current_qp;
  int *proper_coefs;
  int accept_blocks;
  double p_hide;

  int *seenUs; // [0] = DC, otheriwse = AC, ranges from 0 to 4
  int **lastUs;
  int ux, uy, x, y;
  FILE *logfile;
  FILE **files_hist;  // [SLICE_TYPE]
  H264FeatureVector *vec;
} H264FeatureContext;

void myprint(char *text);
void myRandom(double* r);

H264FeatureContext* init_features(char* method_name,
                                    int accept_blocks, double p_hide, int qp);
void close_features(H264FeatureContext *fc);
void writeHeader(FILE *file, char pair, char slice_type,
                   char method, double prob, char accept, int qp);
int get_block_index(int n);
void simulate_hiding_plusminus(H264FeatureContext* fc, int blocknum, int thresh);
void simulate_hiding_f5(H264FeatureContext* fc, int blocknum, int thresh);
void simulate_hiding_lsb(H264FeatureContext* fc, int blocknum, int thresh);
void constructProperCoefArray(int* result, int* level, int* run_before,
                                int total_coeff, int totalZeros);
void addCounts(H264FeatureContext* fc, int qp, int n, int len);
void storeFeatureVectors(H264FeatureContext *feature_context);
void refreshFeatures(H264FeatureContext *feature_context);

#endif /* AVCODEC_H264_EXTRACT */
```

The following function updates the current count vector on an 4x4 coefficient block:

Listing 10: addCounts (in libavcodec/ffmpeg_extract.h)

```
void addCounts(H264FeatureContext *fc, int qp, int n, int len) {
```

```
int i, l, r;
int coef_index;
int *tape = fc->tape;
int blocknum = get_block_index(n);
int qp_index = qp - fc->qp;
int sl = fc->slice_type;

if (blocknum == -1 || qp_index < 0 || qp_index >= QP_RANGE)
  return;

if (sl == TYPE_I_SLICE) return;

simulate_hiding_plusminus(fc, blocknum, THRESHOLD);

// histograms
for (i = 0; i < len; i++) { // num_coefs[blocknum]
  coef_index = tape[i];
  if (coef_index != 0) {
    if (sl == TYPE_P_SLICE)
      fc->num_coefs_p++;
    else if (sl == TYPE_B_SLICE)
      fc->num_coefs_b++;
  }
  coef_index = coef_index + ranges[blocknum][i];
  if (coef_index < 0) continue;
  if (coef_index > 2*ranges[blocknum][i]) continue; // -1
  fc->vec->histograms[sl][qp_index][blocknum][i][coef_index]++;
}
//pairs
for (i = 0; i < len-1; i++) { // num_coefs[blocknum]
    // we are allowed to exceed the local range here
    // space is limited by ranges of first two coefs
  l = tape[i] + ranges[blocknum][0];
  r = tape[i+1] + ranges[blocknum][1];
  if (l < 0 || l > 2*ranges[blocknum][0]) continue;
  if (r < 0 || r > 2*ranges[blocknum][1]) continue;
  fc->vec->pairs[sl][qp_index][blocknum][l][r]++;
}
// UvsV
if (n == 49) {
  memcpy(fc->lastUs[0], tape, num_coefs[1]*sizeof(int));
    // it could happen that we skip one frame and align perfectly,
    // VERY unlikely though. Comparing x,y with ux,uy should be enough
  fc->seenUs[0] = 1;
}
else if (n == 50 && fc->seenUs[0] && fc->ux == fc->x && fc->uy == fc->y) {
  for (i = 0; i < num_coefs[1]; i++) {
    l = fc->lastUs[0][i] + ranges[1][0];
    r = tape[i+1] + ranges[1][0];
    if (l < 0 || l > 2*ranges[1][0]) continue;
    if (r < 0 || r > 2*ranges[1][0]) continue;
    fc->vec->uvsv[sl][qp_index][0][l][r]++;
  }
  fc->seenUs[0] = 0;
}
if (n >= 16 && n <= 19) {
  memcpy(fc->lastUs[n-15], tape, num_coefs[2]*sizeof(int));
  fc->seenUs[n-15] = 1;
} else if ((n >= 32 && n <= 35) && fc->seenUs[n-31] &&
            fc->ux == fc->x && fc->uy == fc->y) {
  for (i = 0; i < num_coefs[2]; i++) {
    l = fc->lastUs[n-31][i] + ranges[2][i];
```

```
      r = tape[i+1] + ranges[2][i];
      if (l < 0 || l > 2*ranges[2][i]) continue;
      if (r < 0 || r > 2*ranges[2][i]) continue;
      fc->vec->uvsv[sl][qp_index][i+1][l][r]++;
    }
    fc->seenUs[n-31] = 0;
  }
}
```

8.2 Stegosaurus

8.2.1 structures and classes

The basic structures and classes are defined in the `core/Stegosaurus.h` file. These are the structures representing a feature set:

Listing 11: structures representing a feature set

```
#define SQUARE(x) (x)*(x)

#define CUBLAS_CALL(x) switch (x) {               \
    case CUBLAS_STATUS_SUCCESS:                   \
      break;                                      \
    case CUBLAS_STATUS_NOT_INITIALIZED:           \
      printf("cublas: not init \n");              \
      break;                                      \
    case CUBLAS_STATUS_INVALID_VALUE:             \
      printf("cublas: invalid value \n");         \
      break;                                      \
    case CUBLAS_STATUS_MAPPING_ERROR:             \
      printf("cublas: Mapping error \n");         \
      break;                                      \
    default:                                      \
      printf("cublas: Something else \n");        \
      break;                                      \
  }

#define CUDA_CALL(x) switch (x) {                           \
    case cudaSuccess:                                       \
      break;                                                \
    case cudaErrorInvalidValue:                             \
      printf("cuda: invalid value! \n");                    \
      break;                                                \
    case cudaErrorInvalidDevicePointer:                     \
      printf("cuda: invalid dev pointer! \n");              \
      break;                                                \
    case cudaErrorInvalidMemcpyDirection:                   \
      printf("cuda: invalid memcopy direction! \n");        \
      break;                                                \
    default:                                                \
      printf("cuda: Something else. \n");                   \
      break;                                                \


typedef struct myGaussian {
  int dim;
  double *mu;
  double *sigma;
  double *sigma_inverse;
  double *qr_diag;
```

```
  double *qr;
} myGaussian;

typedef struct featureHeader {
  int video_bitrate;
  char slice_type;
  char method;
  double prob;
  char accept;
  char qp_offset;
  char qp_range;
  unsigned char ranges[3][16];
} featureHeader;

typedef struct featureSet {
  featureHeader *header;
  int dim;
  int hist_dim;          // each for single qp
  int pair_dim;
  int uvsv_dim;
  int masked_dim;
  uint64_t M;            // number of vectors
  int id;                // helps to find a set
  int gpu_matrix_width; // max number of matrix columns in gpu memory
  int num_files;
  int current_file;
  uint64_t *vsPerFile;
  FILE **files;
  char **paths;
  char *name;
  long dataOffset;

  store_elem *counts;   // want to make this long as soon as compression works
  double *vec;
  double *vec_g;
  double *ones_g;
  double *max_g;
  double *min_g;        // usually is identically zero, but better be sure
  double *mu_g;
  double *mu_vec;
  double *var_g;
  uint64_t *mask_counts;
  int *mask_vec;
  double *max_vec;     // vector of maximum entries
  double *min_vec;
  double *qp_vec;
  double prob;
  double divM;
  double mmd;
  myGaussian* gauss;
} featureSet;
```

The following structures are used to handel GPU access:

Listing 12: structures representing a feature set

```
typedef struct gpuContext {
  int threads_per_block;
  int num_streams;
  long doublesOnGPU;
  cublasHandle_t handle;
} gpuContext;
```

```
typedef struct stegoContext {
  gpuContext *gpu_c;
  featureSet *features;
  long doublesInRAM;
} stegoContext;
```

8.2.2 GUI interaction

The Stegosaurus GUI interacts manages the feature sets through the `StegoModel`, this is the definition of the `StegoModel` and related classes:

Listing 13: class definitions of the `StegoView`, `FeatureCollection` and `StegoModel`

```
class StegoView {
public:
  virtual void updateView() { };
  virtual void updateProgress(double p) { };
  virtual void updateCollection() { };
};

class FeatureCollection {
public:
  FeatureCollection(featureHeader *h);
  ~FeatureCollection();

  class Iterator {
  public:
    Iterator(FeatureCollection *f);
    bool hasNext();
    featureSet* next();
  protected:
    FeatureCollection *fc;
    map< int, featureSet* >::iterator iter;
  };

  int getNumSets();
  int addFeatureFile(const char *path, featureHeader *header,
                       stegoContext *steg, featureSet *cleanSet);
  featureSet* getFeatureSet(int index);
  FeatureCollection::Iterator* iterator();
protected:
  map < int, featureSet* > collection;
  featureHeader header;
  int num_sets;
  int current_set;
  int selected;
};

class StegoModel {
public:
  StegoModel();
  ~StegoModel();

  class Iterator {
  public:
    Iterator(StegoModel* m);
    bool hasNext();
    pair< int, int > getX();
    FeatureCollection* next();
  protected:
```

```
    StegoModel *model;
    pair< int , int > x;
    map< pair< int , int >, FeatureCollection* >::iterator iter;
  };

  void addView(StegoView *view);
  void openDirectory(const char *path);
    int openFile(const char* path, int i, int num_sets, featureHeader& header);
  void estimateMus();
  void doMMDs();
  double doMMD(featureSet* clean, featureSet* stego);
  void setFeatures(featureSet *set);
  featureSet* getCleanSet();

  int getDimension();
  int getHistDim();
  int getPairDim();
  int getUvsVDim();
  int getSigmaDim();
  int getQPRange();
  double* getMaxVector();          // Vector of maximum elements
  double* getMuVector();           // can be a particular feature vector or mu
  double* getQPHist();
  double* getSigma();
  double *getDiag();
  int** getRanges();
  StegoModel::Iterator* iterator();

  void collectionChanged(); // rebuilds the collection tree in the GUI
protected:
  int **ranges;
  list< StegoView* > views;
  map< pair< int , int >, FeatureCollection* > collections;
  set<string> *seenPaths;
  stegoContext *steg;
  featureSet *cleanSet;
  mmdContext *mc;
  void modelChanged();                // asks all views to update themselves
  void progressChanged(double p);     // aks all views to update their progress
};
```

8.2.3 Normalization

These are the declarations of all functions and CUDA kernels that are used for feature normalisation:

Listing 14: Normalization (declarations in `core/Stegosaurus.h`)

```
__global__ void initDArrayKernel(double *m, int dim, double val);
__global__ void finishMax(int dim, double* min, double* max);
__global__ void compareMax(int dim, double *current_max, double *new_features);
__global__ void compareMin(int dim, double *current_min, double *new_features);
__global__ void rescaleKernel(int dim, double *vec_g,
                                double *min_g, double *max_g);
__global__ void varianceKernel(double divM, double *vec_g,
                                 double *mu_g, double *var_g, int dim);
__global__ void normalizeKernel(double* vec_g, double* mu_g,
                                  double* var_g, int dim);

extern "C" {
```

```
 ...
  int estimateScalingParameters(stegoContext* steg, featureSet* set);
  int readCounts(featureSet* set);
  int readVectorRescaled(stegoContext *steg, featureSet *set, double *vec_g);
  int readVectorNormalized(stegoContext *steg, featureSet *set, double *vec_g);
  int readVectorL1D(stegoContext *steg, featureSet *set, double *vec_g);
  int readVectorL2(stegoContext *steg, featureSet *set, double *vec_g);
  void scaleL1D(stegoContext* steg, int dim, double* vec,
                double* vec_g, double* ones_g);
 ...
}
```

The corresponding kernel implementations:

Listing 15: Normalization kernel implementations (in `core/Stegosaurus.cu`)

```
__global__ void initDArrayKernel(double *m, int dim, double val) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx < dim)
    m[idx] = val;
}

__global__ void finishMax(int dim, double *min, double *max) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx < dim) {
    max[idx] = max[idx] - min[idx];
    if (max[idx] < 0.0000001)
      max[idx] = 1.;
  }
}

__global__ void compareMax(int dim, double *current_max, double *new_features) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx < dim) {
    if (current_max[idx] < new_features[idx])
      current_max[idx] = new_features[idx];
  }
}

__global__ void compareMin(int dim, double *current_min, double *new_features) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx < dim) {
    if (current_min[idx] > new_features[idx])
      current_min[idx] = new_features[idx];
  }
}

__global__ void rescaleKernel(int dim, double *vec_g,
                              double *min_g, double *max_g) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx < dim && max_g[idx] > 0.) {
    vec_g[idx] = (vec_g[idx]-min_g[idx]) / (max_g[idx]-min_g[idx]);
  }
}

__global__ void varianceKernel(double divM, double* vec_g,
                               double* mu_g, double* var_g, int dim) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;
```

```
  double delta = mu_g[idx] - vec_g[idx];

  if (idx > dim)
    var_g[idx] += delta * delta * divM;
}

__global__ void normalizeKernel(double *vec_g, double *mu_g,
                                  double *var_g, int dim) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx < dim)
    vec_g[idx] = (vec_g[idx] - mu_g[idx]) * var_g[idx];
}
```

And the corresponding function implementations as well:

Listing 16: Normalization function implementations (in `core/Stegosaurus.cu`)

```
int estimateScalingParameters(stegoContext *steg, featureSet *set) {
  uint64_t i, j;
  uint64_t M = set->M;
  uint64_t dim = set->dim;
  uint64_t max_elem = 0ul;
  int tpb = steg->gpu_c->threads_per_block;

  initDArray(set->max_g, dim, tpb, 0.);
  initDArray(set->min_g, dim, tpb, INFINITY);
  initDArray(set->mu_g, dim, tpb, 0.);
  initDArray(set->var_g, dim, tpb, 0.);
  for (j = 0ull; j < set->dim; j++) {
    set->mask_counts[j] = 0ul;
    set->mask_vec[j] = 0;
  }

  for (i = 0ull; i < M; i++) {
    readVectorL1D(steg, set, set->vec_g);
    for (j = 0ull; j < set->dim; j++) {
      if (set->counts[j] > 0ul)
        set->mask_counts[j]++;
      if (set->counts[j] > max_elem)
        max_elem = set->counts[j];
    }
    compareMax<<<BLOCKS(dim,tpb),tpb>>>(dim, set->max_g, set->vec_g);
    compareMin<<<BLOCKS(dim,tpb),tpb>>>(dim, set->min_g, set->vec_g);
    cublasDaxpy(steg->gpu_c->handle, dim, &(set->divM),
                set->vec_g, 1, set->mu_g, 1);
  }
  finishMax<<<BLOCKS(dim,tpb),tpb>>>(dim, set->min_g, set->max_g);
  for (j = 0ull; j < set->dim; j++) {
    if (set->mask_vec[j] > set->M/100ull) set->mask_vec[j] = 1;
  }
  printf("max_elem: %u \n", max_elem);
  stegoRewind(set);
  for (i = 0ull; i < M; i++) {
    readVectorL1D(steg, set, set->vec_g);
    varianceKernel<<<BLOCKS(dim,tpb),tpb>>>(set->divM,
                       set->vec_g, set->mu_g, set->var_g, dim);
  }
  stegoRewind(set);
  CUBLAS_CALL( cublasGetVector(dim, sizeof(double),
                                 set->max_g, 1, set->max_vec, 1));
  CUBLAS_CALL( cublasGetVector(dim, sizeof(double),
                                 set->min_g, 1, set->min_vec, 1));
```

```
  CUBLAS_CALL( cublasGetVector(dim, sizeof(double),
                                   set->mu_g, 1, set->mu_vec, 1));
  // maybe this can be done in a kernel!!
  CUBLAS_CALL( cublasGetVector(dim, sizeof(double), set->var_g, 1, set->vec, 1));
  for (i = 0ull; i < dim; i++) {
    if (set->vec[i] > 0.)
      set->vec[i] = 1./sqrt(set->vec[i]);
    else {
      set->vec[i] = 1.;
    }
  }
  CUBLAS_CALL( cublasSetVector(dim, sizeof(double), set->vec, 1, set->var_g, 1));

  return 0;
}

// changes current file of set if necessary
int readCounts(featureSet *set) {
  int i;
  int read = 0;

  read = fread(set->counts, sizeof(store_elem),
                 set->dim, set->files[set->current_file]);
  if (read == 0) {
    fseek(set->files[set->current_file], set->dataOffset, SEEK_SET);
    set->current_file++;
    if (set->current_file == set->num_files)
      return -1;
    fseek(set->files[set->current_file], set->dataOffset, SEEK_SET);
    return readCounts(set);
  } else if (read != set->dim) {
    return -1;
  }

  for (i = 0; i < set->dim; i++) {
    set->vec[i] = (double) set->counts[i];
  }

  return read;
}

// reads directly into gpu memory
int readVectorL2(stegoContext *steg, featureSet *set, double *vec_g) {
  int read;
  double norm;

  read = readCounts(set);
  CUBLAS_CALL( cublasSetVector(set->dim, sizeof(double), set->vec, 1, vec_g, 1));
  CUBLAS_CALL( cublasDnrm2(steg->gpu_c->handle, set->dim, vec_g, 1, &norm));
  norm = 1./norm;
  CUBLAS_CALL( cublasDscal(steg->gpu_c->handle, set->dim, &norm, vec_g, 1));

  return read;
}

int readVectorL1D(stegoContext *steg, featureSet *set, double *vec_g) {
  int read;

  read = readCounts(set);
  scaleL1D(steg, set->dim, set->vec, vec_g, set->ones_g);

  if (read != set->dim) printf("read something wrong! \n");
```

```
  return read;
}

int readVectorRescaled(stegoContext *steg, featureSet *set, double *vec_g) {
  int read = readVectorL1D(steg, set, vec_g);
  int tpb = steg->gpu_c->threads_per_block;

  rescaleKernel<<<BLOCKS(set->dim, tpb), tpb>>>(set->dim,
                    vec_g, set->min_g, set->max_g);

  return read;
}

int readVectorNormalized(stegoContext *steg, featureSet *set, double *vec_g) {
  int read = readVectorL1D(steg, set, vec_g);
  int tpb = steg->gpu_c->threads_per_block;

  normalizeKernel<<<BLOCKS(set->dim, tpb), tpb>>>(vec_g,
                    set->mu_g, set->var_g, set->dim);

  return read;
}

void scaleL1D(stegoContext *steg, int dim,
                double *vec, double *vec_g, double *ones_g) {
  double norm;

  CUBLAS_CALL( cublasSetVector(dim, sizeof(double), vec, 1, vec_g, 1));
  CUBLAS_CALL( cublasDdot(steg->gpu_c->handle, dim, vec_g, 1, ones_g, 1, &norm));
  norm = (double) dim/norm;
  CUBLAS_CALL( cublasDscal(steg->gpu_c->handle, dim, &norm, vec_g, 1));
}
```

8.2.4 MMD Calculation

The `mmdContext` contains all variables that are needed for an MMD calculation. `bw_x` and `bw_y` represent the blockwidths, that is the number of vectors that are loaded on the GPU in both x and y directions.

Listing 17: structures and function declarations (in `core/Stegosaurus.h`)

```
typedef struct mmdContext {
  uint64_t n;
  uint64_t cache;
  int kernel_blockwidth;
  int kernel_gridwidth;
  double gamma;
  double mmd;
  featureSet *clean;
  featureSet *stego;
  double *clean_vectors_down_g;
  double *clean_vectors_right_g;
  double *stego_vectors_down_g;
  double *stego_vectors_right_g;
  double *results_c_vs_c_g;
  double *results_c_vs_s_g;
  double *results_s_vs_s_g;
  double *results;
  double *v_g;
  double *temp_g;
```

```
  double *vectors_g;
} mmdContext;

__global__ void gammaKernel(int dim, uint64_t cache, int offset,
                              int steps, uint64_t bw_x, uint64_t bw_y,
                              double* down_g, double* right_g, double* results);
__global__ void mmdKernel(double minus_gamma, double *cvc_g,
                            double *cvs_g, double *svs_g);

  void initMMD(stegoContext* steg, mmdContext& mc);
  void closeMMD(mmdContext& mc);
  void estimateGamma(stegoContext* steg, mmdContext& mc);
  void launchGammaKernel(mmdContext& mc, int dim, uint64_t bw_x, uint64_t bw_y,
                           double* down_g, double* right_g, double* results_g);
  void estimateMMD(stegoContext* steg, mmdContext& mc);
```

The MMD calculation is launched through the `StegoModel`.

Listing 18: launching MMD calculation (in `core/Stegosaurus.cpp`)

```
void StegoModel::doMMDs() {
  map< pair< int, int >, FeatureCollection* >::iterator fiter;
  FeatureCollection::Iterator *citer;

  mc = (mmdContext*) malloc(sizeof(mmdContext));
  mc->clean = cleanSet;
  startAction(mc->clean);
  initMMD(steg, *mc);
  estimateGamma(steg, *mc);

  for (fiter = collections.begin(); fiter != collections.end(); fiter++) {
    if (fiter->second != 0) {
      printf("<%i, %i> \n", fiter->first.first, fiter->first.second);
      citer = fiter->second->iterator();
      while (citer->hasNext()) {
        mc->stego = citer->next();
        printf("doing set %g \n", mc->stego->header->prob);
        startAction(mc->stego);
        estimateMMD(steg, *mc);
        mc->stego->mmd = mc->mmd;
        endAction(mc->stego);
      }
    }
  }
  endAction(mc->clean);
  closeMMD(*mc);
}
```

Listing 19: `core/Stegosaurus_MMD.cu`

```
#include "Stegosaurus.h"


__global__ void gammaKernel(int dim, uint64_t cache, int offset, int steps,
                              uint64_t bw_x, uint64_t bw_y, double *down_g,
                              double *right_g, double *results) {
  int i;
  int idx_x = threadIdx.x + blockIdx.x*blockDim.x;
  int idx_y = threadIdx.y + blockIdx.y*blockDim.y;
  int dx = idx_x*dim + offset;
  int dy = idx_y*dim + offset;
  double temp;
```

```
  double current_sum = 0.;

  if (idx_x < bw_x && idx_y < bw_y) {
    for (i = 0; i < steps; i++) {
      temp = down_g[dy + i] - right_g[dx + i];
      current_sum += temp * temp;
    }
    results[idx_y + cache*idx_x] += current_sum;
  }
}

__global__ void mmdKernel(double minus_gamma, double *cvc_g,
                          double *cvs_g, double *svs_g) {
  int idx = threadIdx.x + blockIdx.x*blockDim.x;
  double cvc = exp(minus_gamma * cvc_g[idx]);
  double cvs = exp(minus_gamma * cvs_g[idx]);
  double svs = exp(minus_gamma * svs_g[idx]);

  cvc_g[idx] = cvc + svs - 2*cvs;
}

void initMMD(stegoContext *steg, mmdContext& mc) {
  int dim = mc.clean->dim;

  mc.n = mc.clean->M;
  mc.kernel_blockwidth = (int) sqrt(steg->gpu_c->threads_per_block);
  mc.cache   = min(mc.n, (uint64_t) (sqrt(steg->gpu_c->doublesOnGPU / 3l +
                  (long) SQUARE(dim) * 4l / 9l) - (long)dim * 2l / 3l));
  mc.kernel_gridwidth = (mc.cache + mc.kernel_blockwidth-1)/mc.kernel_blockwidth;
  printf("cache: %d, kbw: %i \n", mc.cache, mc.kernel_blockwidth);

  CUDA_CALL( cudaMalloc(&mc.clean_vectors_down_g, dim*mc.cache*sizeof(double)));
  CUDA_CALL( cudaMalloc(&mc.clean_vectors_right_g, dim*mc.cache*sizeof(double)));
  CUDA_CALL( cudaMalloc(&mc.stego_vectors_down_g, dim*mc.cache*sizeof(double)));
  CUDA_CALL( cudaMalloc(&mc.stego_vectors_right_g, dim*mc.cache*sizeof(double)));
  CUDA_CALL( cudaMalloc(&mc.results_c_vs_c_g, mc.cache*mc.cache*sizeof(double)));
  CUDA_CALL( cudaMalloc(&mc.results_c_vs_s_g, mc.cache*mc.cache*sizeof(double)));
  CUDA_CALL( cudaMalloc(&mc.results_s_vs_s_g, mc.cache*mc.cache*sizeof(double)));
  CUDA_CALL( cudaHostAlloc(&mc.results, mc.cache*mc.cache*sizeof(double),
                           cudaHostAllocDefault));
}

void closeMMD(mmdContext& mc) {
  CUDA_CALL( cudaFree(mc.clean_vectors_down_g));
  CUDA_CALL( cudaFree(mc.clean_vectors_right_g));
  CUDA_CALL( cudaFree(mc.stego_vectors_down_g));
  CUDA_CALL( cudaFree(mc.stego_vectors_right_g));
  CUDA_CALL( cudaFree(mc.results_c_vs_c_g));
  CUDA_CALL( cudaFree(mc.results_c_vs_s_g));
  CUDA_CALL( cudaFree(mc.results_s_vs_s_g));
  CUDA_CALL( cudaFreeHost(mc.results));
}

// we need to execute the kernel step-wise since there is a timeout on kernel
// executions that can be deactivated on Tesla- but not on Geforce-cards.
void launchGammaKernel(mmdContext& mc, int dim, uint64_t bw_x, uint64_t bw_y,
                       double* down_g, double* right_g, double* results_g) {
  int i;
  int step_size = 128;
  dim3 grid, block;

  grid = dim3(BLOCKS(bw_x, mc.kernel_blockwidth),
```

```
                BLOCKS(bw_y, mc.kernel_blockwidth));
  block = dim3(mc.kernel_blockwidth, mc.kernel_blockwidth);
  for (i = 0; i < dim; i += step_size) {
    gammaKernel<<<grid,block>>>(dim, mc.cache, i, min(step_size, dim - i),
                    bw_x, bw_y, down_g, right_g, results_g);
    cudaThreadSynchronize();
  }
}

void estimateGamma(stegoContext *steg, mmdContext& mc) {
  uint64_t i, j;
  uint64_t bw_x, bw_y, pos_x, pos_y;
  int tpb = steg->gpu_c->threads_per_block;
  featureSet *cleanSet = mc.clean;
  uint64_t M = mc.n;
  uint64_t l;
  int dim = cleanSet->dim;
  priority_queue< double > q;

  for (pos_x = 0ull; pos_x < M; pos_x += mc.cache) {
    bw_x = min(mc.cache, M-pos_x);
    jumpToVector(mc.clean, pos_x);
    for (i = 0ull; i < bw_x; i++) {
      readVectorRescaled(steg, mc.clean, mc.clean_vectors_right_g + i*dim);
    }
    for (pos_y = pos_x; pos_y < M; pos_y += mc.cache) {
      bw_y = min(mc.cache, M-pos_y);
      jumpToVector(mc.clean, pos_y);
      for (i = 0ull; i < bw_y; i++) {
        readVectorRescaled(steg, mc.clean, mc.clean_vectors_down_g + i*dim);
      }
      initDArray(mc.results_c_vs_c_g, SQUARE(mc.cache), tpb, 0.);
      launchGammaKernel(mc, dim, bw_x, bw_y, mc.clean_vectors_down_g,
                        mc.clean_vectors_right_g, mc.results_c_vs_c_g);
      CUDA_CALL( cudaMemcpy(mc.results, mc.results_c_vs_c_g,
                            SQUARE(mc.cache)*sizeof(double),
                            cudaMemcpyDeviceToHost));
      cudaThreadSynchronize();
      for (i = 0ull; i < bw_x; i++) {
        for (j = 0ull; j < bw_y; j++) {
          if (pos_x + i < pos_y + j)  {
            q.push(mc.results[j + i*mc.cache]);
          }
        }
      }
    }
  }
  stegoRewind(mc.clean);

  printf("queue size: %i, M = %i, expected size: %i \n",
         q.size(), M, M*(M-1ull)/2ull);
  for (l = 0; l < M*(M-1ull)/4ull; l++) {
    q.pop();
  }
  printf("median: %g => gamma = %g , queue size: %i \n",
         q.top(), 1./q.top(), q.size());
  mc.gamma = 1./q.top();
}

void estimateMMD(stegoContext *steg, mmdContext& mc) {
  uint64_t i, j, k;
  uint64_t bw_x, bw_y;
```

```
  uint64_t pos_x, pos_y;
  int tpb = steg->gpu_c->threads_per_block;
  int dim = mc.clean->dim;
  uint64_t M = mc.n;
  double mmd = 0.;
  time_t start = time(NULL);

  for (pos_x = 0ull; pos_x < M; pos_x += mc.cache) {
    bw_x = min(mc.cache, M-pos_x);
    jumpToVector(mc.clean, pos_x);
    for (i = 0; i < bw_x; i++) {
      readVectorRescaled(steg, mc.clean, mc.clean_vectors_right_g + i*dim);
    }
    jumpToVector(mc.stego, pos_x);
    for (i = 0; i < bw_x; i++) {
      readVectorRescaled(steg, mc.stego, mc.stego_vectors_right_g + i*dim);
    }
    for (pos_y = 0ull; pos_y < M; pos_y += mc.cache) {
      bw_y = min(mc.cache, M-pos_y);
      jumpToVector(mc.clean, pos_y);
      for (i = 0; i < bw_y; i++) {
        readVectorRescaled(steg, mc.clean, mc.clean_vectors_down_g + i*dim);
      }
      jumpToVector(mc.stego, pos_y);
      for (i = 0; i < bw_y; i++) {
        readVectorRescaled(steg, mc.stego, mc.stego_vectors_down_g + i*dim);
      }
      initDArray(mc.results_c_vs_c_g, SQUARE(mc.cache), tpb, 0.);
      initDArray(mc.results_c_vs_s_g, SQUARE(mc.cache), tpb, 0.);
      initDArray(mc.results_s_vs_s_g, SQUARE(mc.cache), tpb, 0.);
      launchGammaKernel(mc, dim, bw_x, bw_y, mc.clean_vectors_down_g,
                        mc.clean_vectors_right_g, mc.results_c_vs_c_g);
      launchGammaKernel(mc, dim, bw_x, bw_y, mc.clean_vectors_down_g,
                        mc.stego_vectors_right_g, mc.results_c_vs_s_g);
      launchGammaKernel(mc, dim, bw_x, bw_y, mc.stego_vectors_down_g,
                        mc.stego_vectors_right_g, mc.results_s_vs_s_g);
      mmdKernel<<<BLOCKS( mc.cache*mc.cache, tpb), tpb>>>(-1.*mc.gamma,
                          mc.results_c_vs_c_g, mc.results_c_vs_s_g,
                          mc.results_s_vs_s_g);
      cudaThreadSynchronize();
      CUBLAS_CALL( cublasGetVector(SQUARE(mc.cache), sizeof(double),
                                   mc.results_c_vs_c_g, 1, mc.results, 1));
      for (i = 0ull; i < bw_x; i++) {
        for (j = 0ull; j < bw_y; j++) {
          if (pos_x + i == pos_y + j) continue;
          mmd += mc.results[j + i*mc.cache];
        }
      }
    }
  }
  stegoRewind(mc.clean);
  stegoRewind(mc.stego);
  mmd /= (double) (M * (M-1));
  mmd = sqrt(mmd);
  printf("have some mmd: %g [%is]\n", mmd, time(NULL)-start);
  mc.mmd = mmd;
}
```