Temporal Compression of Digital Holograms Using the ffmpeg Standard

Chris Mullen

Final Year Project – 2014

B.Sc. Single Honours in Computer Science

and Software Engineering

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare

Ireland

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

A thesis submitted in partial fulfillment of the requirements for the B.Sc. Single Honours in Computer Science and Software Engineering.

Supervisor: Mr. Thomas J. Naughton

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of B.Sc. Single Honours in Computer Science and Software Engineering, is entirely my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Chris Mullen   Date: 24/03/14

Abstract

Digital Holograms are an effective way of representing real world objects in three dimensions. The data structures required to store them however, results in a lot of noisy data being written. We know noise data is very hard to compress using conventional methods, the consequence being very large file sizes, in particular for digital hologram video. While a lot of work has been done on compression of individual digital holograms, currently, no efficient algorithm exists that is capable of compressing digital hologram video at an acceptable rate.

Previous work in the area of inter-frame compression (as used to reduce temporal redundancy in video compression) has shown that utilizing *ffmpeg's* motion estimation algorithms as a means of compressing hologram data has great potential (under ideal conditions). My thesis mainly focuses on verifying these results and attempting to apply them to a more practical setting, especially in terms of recognizing redundancies in noise data.

Through rigorous testing of *ffmpeg's* compression algorithms, I have come across a number of weaknesses in how motion-estimation is implemented; The first being its inability to recognize similar pixel blocks between noise-filled frames when even very small pixel shifts are applied. The improvement I created here was a *GNU-Octave* wrapper which preprocessed frames so shifted pixel blocks are returned to their original position for encoding, and then shifted back for decoding.

The other major flaw I found in *ffmpeg's* motion estimation algorithm is its poor performance when errors are introduced to the data, so when combined with lossy compression methods (e.g. Speckle Reduction), we need to be able cope with some differences in our similar data blocks. This is another area where *ffmpeg* can be improved.

Acknowledgements

I would like to thank my project supervisor Thomas J. Naughton for his assistance and direction over the course of doing this project. His expertise and enthusiasm for Digital Holography was a great help both for completing this project and expanding my knowledge in this area.
I would like to give thanks to my family, friends and colleagues for their continued support during the completion of this project.
I am also grateful to the Computer Science Department for their resources and technical support when required.

Table of Contents

Chapter 1 Introduction

In this chapter I will give an overview of what I hope to achieve during the course of this project, the goals I wish to achieve, the motivation behind researching this area, and the general method used to go from researching the topic of inter-frame compression to building on and improving an existing compression algorithm.

1.1 Goals

During the course of undertaking this project I hope to investigate a number of temporal compression algorithms (redundancies as a result of time, i.e. adjacent/proximal video frames) and examine their usefulness when applied to digital holography.
I hope to improve the compression of digital holograms either by optimizing an existing temporal compression algorithm or by developing my own codec that can be applied to digital holograms.
I also hope to create a good solution to the temporal compression of digital holograms and a high quality report based on my observations and conclusions.

1.2 Motivation

Currently, 3-Dimentional Holographic displays exist only as laboratory prototypes, their components being too expensive for the current market. In the future however, 3-Dimentional Holographic Displays will be quite common, so it is important to develop tools for them at this stage, such as video compression formats.
In their current state, digital holograms are very difficult to compress, due to their complex pixel value and large amount of noise data.
One unaltered digital hologram with dimensions ~2000x2000 pixels can occupy over 60mb of disk space currently, compared to the 10kb of disk space required for a jpeg of similar dimensions; and so, more efficient data compression is required for digital holography to become a viable form of media.

1.3 Method

The project largely consisted of research into conventional image/video compression techniques and how well they performed when applied to digital holography. I then attempted to break these compression techniques using valid inputs which applied more to digital holography. Initially most of the information I gathered was from previous theses in this area, through the *GNU Octave* and *ffmpeg* Documentations [16],[17], and a number of sources relating to high-level video compression algorithms e.g. GOP (Group of Pictures - used in most temporal video compression algorithms).

After extensive research and experimentation, I was then able to look at areas of *ffmpeg* that could be improved in relation to digital holography and implemented a number of improved solutions to this area.

1.4 Report Overview

The remainder of this thesis is arranged accordingly:

Chapter 2 examines the problem I am attempting to solve, and summarizes relevant publications which were examined as background information in the areas of digital holography, lossy hologram compression techniques, inter-frame compression methods, and it's applications to digital holography. This background research will be the basis for **Chapter 3**, where we will see the initial plan, where the project was divided up in terms of project deadlines/time required for each task. We will also look at problems encountered and how they affected the course of the project, contrasting the initial project plan with the actual time taken.

C**hapter 4** will document the extensive experimentation of the limits of *ffmpeg* (using valid image inputs generated in *GNU Octave* code) in terms of both inter-frame coding and motion estimation, looking at these results we will be able to see at what point *ffmpeg* begins to struggle with these inputs, and where motion estimation fails completely. Each experiment has a Discussion/Conclusion section where the results are interpreted and weaknesses are discussed.

Chapter 5 analyzes the conclusions taken from my experiments, along with looking more detail into *ffmpeg's* motion-estimation algorithm to discuss possible solutions for improving the existing inter-frame compression algorithm to work more effectively with digital holograms.

Chapter 6 discusses the design aspect of my solution; it will examine existing solutions to the problem and how my design will contrast/interact with them. It will also debate how well my solution works to solve the problem by reviewing the goals of my thesis.

Chapter 7 will cover the physical implementation of my design, the tools used, how the solution was tested and verified, etc. It will also cover specific parts of my code in more detail, which were engineered after the design stage.

Chapter 8 subjects my solution to the same experiments used to test *ffmpeg's* existing motion estimation algorithm. The results of this experimentation will then be compared with the results from **Chapter 4**.

Chapter 9 will then discuss the fitness of my solution compared with *ffmpeg's* standard motion estimation algorithm, it will also review the skills learned during this project, and possible scope for future work in this area.

Chapter 2 Background and Problem Statement

2.1 Introduction

In this chapter I will study previous work in the areas of digital hologram recording and reconstruction, spatial and temporal compression of hologram phase images, and some exploration into several video codecs. Using this research, I will have a better understanding of what areas of existing temporal compression solutions can be improved. This will take the form of a Problem Statement. There has been a great deal of research into the areas of Digital holography [1], [2], [3] and lossy compression of single hologram images [4], [5], [6], [7], [8], [9], [10], [11].
Applying video compression techniques to Digital Holography is a relatively new idea whereby standard inter-frame compression algorithms are applied to Digital Hologram Phase diagrams in an attempt to detect temporal redundancies in groups of video frames [12], [13], [14]. ***Fig. 2.1.1***

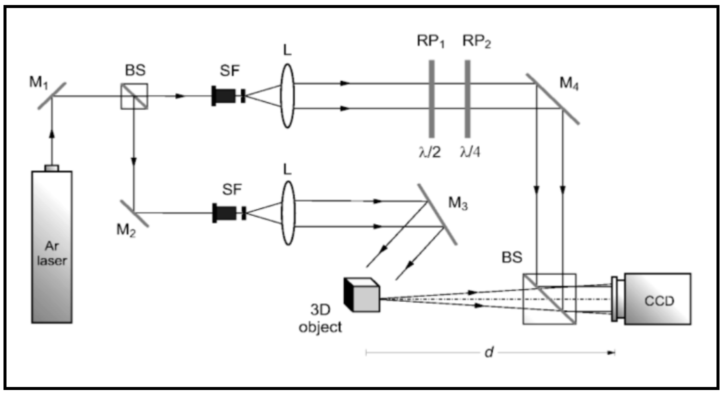


Fig. 2.1.1 [4]

Fig. 2.1.1 *shows the experimental setup for the recording of digital holograms utilizing Phase-shift Interferometry [4]*. ***Ar Laser=Argon Ion Laser; M=Mirror; BS=Beam Splitter; SF=Spatial Filter; L=Lens; RP=Retardation Plate; CCD=Charge Coupled Device***

2.2 Background

In order to gain a comprehensive knowledge of the areas required to carry out this project, the following publications were examined in greater detail:

2.2.1 - Compression of Digital Hologram Sequences Using MPEG-4 [12]

2.2.2 - Visually Lossless Compression of Digital Hologram Sequences [13]

2.2.3 - Hologram Video Codec by *ffmpeg* [14]

2.2.4 - Overview of the H.264/AVC Video Coding Standard [15]

2.2.1 Compression of Digital Hologram Sequences Using MPEG-4 [12]

This publication looks at utilizing existing inter-frame coding algorithms to compress digital hologram phase diagrams in an attempt to reduce the disk space required to store the voluminous digital hologram files. MPEG-4 Part 2 (MPEG-4 Visual) is used specifically to investigate the benefit of inter-frame coding on digital holograms. It utilizes block-based motion compensation to detect redundant pixel blocks in proximate frames. Compression ratios of ~20:1 were realized with lossless results under ideal conditions. This shows there is potential in adapting inter-frame coding techniques to digital holography.

2.2.2 Visually Lossless Compression of Digital Hologram Sequences [13]

This paper focuses on visually measuring the reconstructed quality of digital holograms as opposed to using a numerical metric. Measuring quality reduction numerically can be misleading because lossy compression methods introduce errors in the reconstruction, and because of their speckled nature, the speckle pattern is very sensitive to data changes. A certain level of lossy compression can be achieved while still retaining a visually lossless hologram reconstruction. Compression ratios of between 4:1 and 7.5:1 were achieved while showing no perceptible changes in the appearance of video sequences made up of reconstructed hologram images. See below: ***Fig. 2.2.2.1*** and ***Fig. 2.2.2.2*** shows (*Left*) an unconstructed digital hologram and (*Right*) its reconstruction.

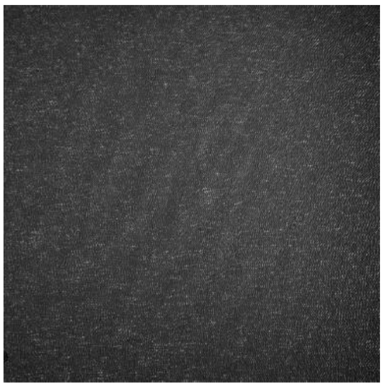


Fig. 2.2.2.1 [13]

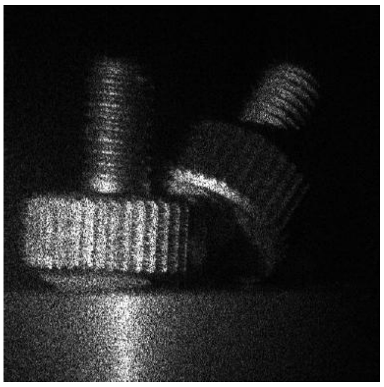


Fig. 2.2.2.2 [13]

2.2.3 Hologram Video Codec by *ffmpeg* [14]

This thesis aims to investigate the effect of quantizing data hologram data at various levels to observe both the file size reduction, and the reconstructed quality of the hologram image. It also verifies that motion estimation can be used to compress noisy hologram data under ideal conditions, and experiments with various video formats to obtain the most suitable codec to apply to digital holography.

2.2.4 Overview of the *H.264/AVC Video Coding Standard* [15]

This publication provides a greater understanding of how Inter-frame compression algorithms operate, specifically Motion Detection/Estimation and Inter-frame coding. It describes H.264/AVC (MPEG-4 Part 10) explicitly, but the ideas of block-based inter-frame coding are common in modern video codecs.

Inter-frame encoding attempts to eliminate temporal redundancies between two or more images by recognizing similar data present in both frames and removing duplicate data. An inter-coded frame, (or inline I-frame) is first divided up into smaller pieces of data (macro-blocks), then instead of storing the raw pixel data of each block, a block-matching algorithm is run on previously encoded raw frames (I-frames), to attempt to find similar macro-blocks.

Note: I-frames are strictly intra-coded, so they can be decoded without additional information, and so, can be used as a reference frame.

Instead of storing the raw pixel data for the macro-block, a motion vector to the matched block can be stored much more efficiently. The block found is likely to not be an exact match so the differences between the macro-blocks (prediction error) also need to be stored.

Using this method, both forward and backward inter-frame prediction is possible, this requires two passes, one to record the raw I-frames (out of order to allow for forward-prediction), and the second pass to record the P-frames (predicted frames, constructed based on I-frame redundancy).

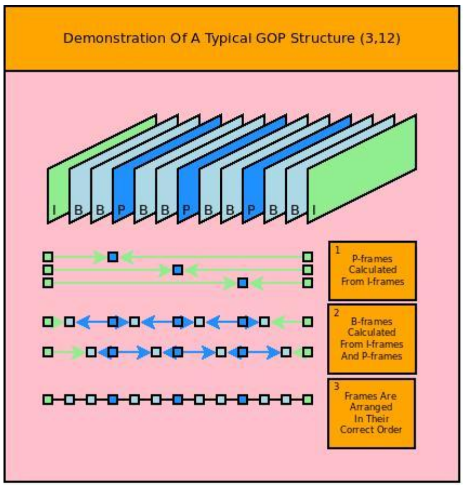
B-frames or Bi-directional Predicted Frames are also sometimes used, they are similar in function to P-frames but instead of being constructed from raw I-frames, they are created using both I-frames and P-frames. This can reduce the quality of the video file because prediction errors are propagated down through subsequent B-frames.

Video codecs usually combine the above frame types in a fixed pattern called a GOP (Group of Pictures). A GOP is usually described with two figures: (M, N).

M= the difference between two anchor frames (I-frames/P-frames).

N= the total number of frames between each raw-encoded frame (I-frames).

Below, is an example of a simple GOP structure with M=3, N=12 (***Fig. 2.4.1***)



Demonstration Of A Typical GOP Structure (3,12)
I
B
B
P
B
B
P
B
B
P
B
B
I
1 P-frames Calculated From I-frames
2 B-frames Calculated From I-frames And P-frames
3 Frames Are Arranged In Their Correct Order

Fig. 2.4.1

Step1- *I-frames are broken up into macro-blocks and cross-correlated with previously encoded I-frames. From this, motion vectors are constructed between similar blocks of data. Depending on the M value, the block is shifted to a fraction of the distance for each P-frame. Any unique data between the two I-frames make up the remainder of each P-frame. I-frames are recorded separately and stored out of order to make forward and backward predictions possible.*
Step2- *B-frames are constructed using a similar method to P-frames, only being made up of redundancies between both I-frames and P-frames.*
Step3- *After processing, the frames are rearranged in their correct order and stored as a single video file.*

I used *ffmpeg* exclusively for testing and implementing inter-frame encoding. A full list of parameters I used, examples, etc. can be found in **Appendix A**.

2.4 Problem Statement

Digital Holograms occupy a lot of disk space compared to other forms of media, due to their complex-valued representation. each pixel in a digital hologram is represented (in two bytes) by both a phase and amplitude value, which not only requires additional disk space but also makes compression more difficult due to the fact that most compression algorithms are designed for real-valued inputs.
Previous work in this area has been successful with this issue by removing the amplitude values and processing the phase-only image [1], [2], [3]. Other work has shown that when similar digital-hologram frames are compressed to video, redundancies exist between adjacent frames, resulting in more significant compression than traditional methods [12].
Since current video codecs are not designed for noisy digital-hologram phase images, there is great potential in compression using temporal redundancy in digital hologram images.

Chapter 3 Project Management

In this chapter I will detail the plan and overall approach taken to complete this project from the outset to finish. I will also examine problems encountered and how this affected my initial plan. I will then contrast my planned sequence of events, with the actual timeline from start to finish.

3.1 Approach

Initially, the project was divided up evenly between the significant tasks that needed to be carried out over my given timeframe research, planning, coding, testing, etc. Firstly, I began conducting background research into the areas of motion estimation and digital holography to gain indispensible knowledge in these fields. Then with a good comprehensive understanding of inter-frame compression techniques and digital holography, I began to thoroughly test *ffmpeg's* temporal coding algorithms to breaking point.

After constructing a set of reasonable tests, the next step was to analyze my experimentation results, and review motion estimation in more detail to derive and design a solution that improves the performance of *ffmpeg's* motion-estimation algorithm. After establishing some high-level and detailed designs for my solution, I then began implementing each element function to be used in the final product. Once my implementation was complete I began unit testing my solution and finally the entire system. I was then able to replicate my previous experiments to examine the fitness of my solution in relation to the current solution for motion estimation.

3.2 Initial Project Plan

My initial plan is depicted in **Fig. B.1 Appendix B** It shows my initial approximation of planned steps including all major stages of development from background research to final solution. Physically adhering to this plan proved impossible (detailed in ***Section 3.3***). The actual timeline differs greatly from my initial project plan as a result (shown in ***Section 3.4***).

3.3 Problems and Changes to Plan

One of the main obstacles I encountered was the strict deadlines I set myself; the main cause for delay was during the experimentation stage which greatly altered my initial project plan.
Several revisions of these experiments had to be created in order to tease out as much sources of error. Since motion estimation is an automatic process; the inputs had to be chosen very carefully to prevent unwanted redundancies to be detected. This stage slowed my progress more than any other as (originally) a lot of my results were inconsistent due to unforeseen factors, and experiments had to be re-designed as a result. Another element of this stage that stalled my progress somewhat was actually utilizing some of *ffmpeg's* specialist tools (i.e. motion estimation, achieving truly lossless intra-frame compression, etc.), which due to the large open-source nature of *ffmpeg*, is poorly documented in some cases, and in others, not at all.

3.4 Final Project Record

My final project record, presented in **Fig. B.2 Appendix B**, shows the actual timeline of the course of this project. This contrasts with my initial plan due to a number of unforeseen circumstances regarding the creation of distinct test conditions for experiments, producing accurate test results, and researching advanced *ffmpeg* features.

Chapter 4 Assessment of *ffmpeg's* Video Compression Algorithms

In this chapter I will evaluate the strengths/limitations of *ffmpeg's* inter-frame compression algorithms through a series of experiments testing such aspects as motion detection with simple objects, motion detection with noise data under ideal conditions, assessment of motion-detection in separating repeated noise data from surrounding noise data, and examining how well motion detection performs with corruptions added to repeated data.

Experiment 1 - Introduction

One of the goals of my fourth year project is to look into compressing holograms using *ffmpeg*. The *ffmpeg* standard utilizes a number of compression algorithms by default, both intra-frame (where each frame is compressed independently of any other frame, e.g. run-length encoding) and inter-frame (where similar frames are compressed and stored together e.g. motion estimation).
My project covers inter-frame compression specifically. To gain a better understanding off *ffmpeg*, I created a number of experiments to isolate certain parts of the compression algorithm.

Experiment 1.A

My first experiment is created to verify that motion estimation has a significant effect on compression of simple frames, for which it was designed. It consists of ten frames (lossless .png) showing a simple animation (This is the control), and a second set of frames, identical to the first only every second frame is rotated 180˚. The images are lossless to remove intra-frame compression from the equation. This is to determine if motion estimation is being used by the *ffmpeg* video-compression algorithm. Both of these sets of frames are then compressed by *ffmpeg* to create two separate videos (lossless *H.264 .mp4*). The purpose of flipping the frames is to attempt to disable, or at least degenerate the affects of motion estimation. There are two test cases in this experiment (each will have a control and test group of frames):
test1- (motion estimation disabled by setting all frames to be stored as I-frames (***Fig 1.1.1***)).
test2- (motion estimation maximized (***Fig 1.1.2***)). (Frames are labelled F1-F10).
Parameters used to generate frames and compress to video:

gen(mkvid=true,play=true);

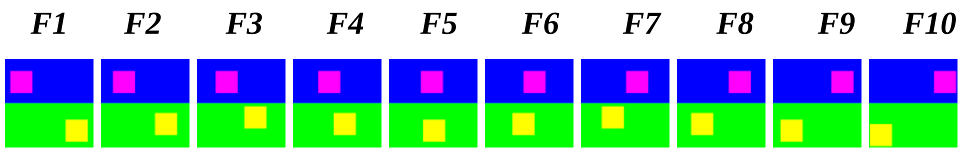


Fig. 1.1.1

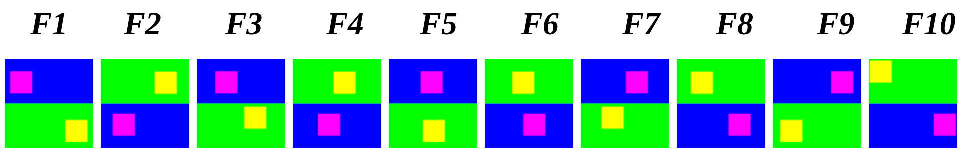


Fig. 1.1.2

Results:

(All frames were generated in *GNU octave* to produce consistent frames, each 14.6kb).

Test Case	Description	File Size (kilobytes)
test1a.mp4	motion estimation disabled	5.4
test1b.mp4	test1a.mp4 with flipped frames	5.4
test2a.mp4	motion estimation maximized	3.8
test2b.mp4	test2a.mp4 with flipped frames	4.4

Discussion/Conclusions:

Both of test1‘s test cases are the same size, which means that motion estimation does have an effect on the final video size. For test2 there is definitely a reduction in size, which when applied to noisy hologram images, should compress them considerably. The results aren't as significant as I first thought (only 1.6kb difference), but I think this is due to the simplicity of the frames, only a small portion of each frame is moved (the coloured blocks) so there isn't a huge amount of data to be picked up by motion estimation, and most of the compression is caused by intra-frame correlation (large areas of repeated data can be compressed as one block rather than individual pixels).

I would have expected that test2a.mp4 would be the same size as test1a.mp4 and test2a.mp4 due to the frames being flipped, but due to how *ffmpeg's* motion estimation works, (block-based), for the areas where the blue meets the green, when flipped, the blocks don't have to move very far, and so is picked up by motion estimation. Test2‘s test cases are different because, while motion estimation is made more difficult, the margin between the blue and green backgrounds is still detected. These issues should not matter when applied to noisy images due to their complex nature, but I have determined that motion estimation does have an effect on even simple video compression.

Experiment 1.B

The second experiment I performed had similar goals to the first, but attempted to apply motion estimation to a noise pattern rather than simple geometric shapes, since this will be more relevant to noisy hologram images. The noise pattern is used as a worst case scenario for motion estimation as it contains totally random values.

The experiment consists of a noise image created with *GNU Octave's rand()* function on a black background.

There are ten frames as before (generated in *GNU Octave*, each 63.4kb), with the noise pattern moved to ten different positions in the grid for each frame. Two lossless videos are created using the same frames, one with motion estimation maximized and one with motion estimation disabled (***Fig. 1.2.1***). The Background is kept as small as possible to minimize intra-frame correlation, which can reduce the significance of my results (as in **Experiment 1**).

(Frames are labelled F1-F10).

The noise block is 250x250 pixels, and the background is 300x300 pixels.

Parameters used to generate frames and compress to video:

noise(mkvid=true,play=true,framesout=true);

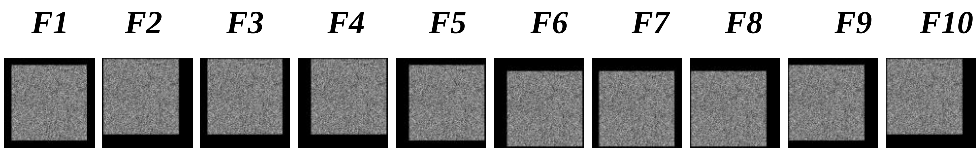


Fig. 1.2.1

Results:

Test Case	Description	File Size (kilobytes)
test1.mp4	motion estimation maximized	39.1kb
test2.mp4	motion estimation disabled	255.7kb

Discussion/Conclusions:

When applied to noisy data, motion estimation performs very well, with motion estimation disabled the video is over 6.5 times the size of the same data compressed using motion estimation.

All of the data dealt with so far has been using lossless formats (*.png, H.264 .mp4*).

With the conclusions I have found so far, I can now apply these results to lossy image/video compression, since holograms contain a lot of noise.

Note: When decompressing the frames from each video test file, they are reprocessed in octave, because octave creates .png images more efficiently (by default) than ffmpeg and so, to get a fair comparison we compare octave-generated .png images with octave-generated .png images.

Aside: this experiment was repeated with a large un-reconstructed phase-only hologram image in place of the noise block above showing slightly diminished (but still a significant ~6.2:1 ratio) compression rates. This shows that motion-estimation can be scaled effectively.

Experiment 2 - Introduction

In order to investigate the limits of *ffmpeg's* motion estimation algorithm (when applied to noisy data), I first must verify that there is a positive file size reduction using inter-frame coding without motion estimation i.e. the same data in the same location.

Experiment 2.A

This is a short test to verify that inter-frame coding has an effect on noisy data under ideal conditions (in terms of movement in redundant blocks). The test consists of two sets of two frames, my control and test case (both have dimensions 320x320 pixels). Both test cases have the same randomly generated noise frames. (***Frames 1*** and ***2*** are generated separately so they have nothing in common). The average difference per pixel is calculated to ensure they are totally dissimilar. Average difference per pixel is calculated as follows: *avg_diff=sum(sum(abs(A-B)))/prod(size(A));*

A new randomly generated block of pixels is created and added to each frame at position (1, 1), (the top-left corner), this time it is calculated once so all frames now share this block. for ***Frame 2*** of my test case, instead of using the repeated block as in all frames before, an entirely new pixel block is created and placed in the same location (***Fig. 2.1.1***) *(A border has been drawn around the block to make it easier to track).*
(Frames are labelled F1-F2).
Now ***Frame1*** in both my test cases are identical, and ***frame2*** differ only by the pixel values of the newly created block.
Each set of frames is compressed to lossless video *(.mp4 part 10 (H.264)*).
now if inter-frame coding is having an effect, I expect to see a significant reduction in control.mp4 versus ***test.mp4***, since in the control, the repeated 64x64 block should be stored once for both frames, while the test case's frames will have nothing in common, and so all pixels are stored as raw data.
Parameters used to generate frames and compress to video:

err(mkblock='a',back='r',block='r',blocksize=64,backdim=5,erramount='n',startpos=1,pxshift=0,mkvid=true,lossless=true,play=true,framesout=false,correlation=false,drawfigs=false);

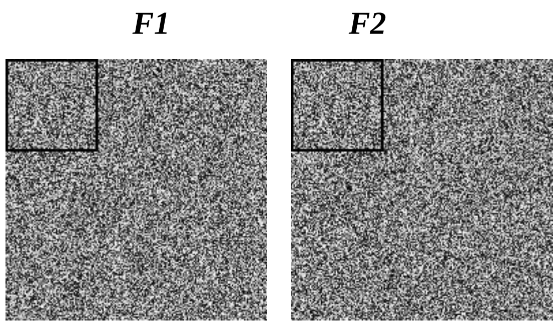


Fig. 2.1.1

Results:

Test Case	Description	File Size (bytes)
control.mp4	Block is repeated (in Same Position)	275732
test.mp4	Block is not repeated (totally random image)	291457

Discussion/Conclusions

The results verify that inter-frame encoding is having an effect. Because the repeated block of data is being recognised by *ffmpeg's* inter-frame encoding algorithm, its raw pixel data is stored once and a pointer is used for any more instances of that block. Because of the random data we are using, a small allowance can be made for the file size difference but what we see here is a much more significant change. (The experiment was run a number of times to verify this, and results only varied a negligible ~200 bytes). There is also a small overhead for the motion vector (pointer).

Experiment 2.B

The previous experiment looked at inter-frame coding without movement. Now we look at *ffmpeg's* motion estimation capabilities.

The experiment is set up exactly the same as before, only now we vary the position of the repeated block (or unrepeated block as in my test case). I expect to see similar results as in **Experiment 2.A** if the repeated block is being recognised, despite being moved. If the repeated block is not being noticed, both video files should be roughly the same file size as they will both be entirely made up of raw data. Various pixel shifts will be used to get a good approximation of where motion estimation fails (***Fig. 2.1.1***) *(A border has been drawn around the block to make it easier to track).* (Frames are labelled F1-F2).

Parameters used to generate frames and compress to video:

err(mkblock='a',back='r',block='r',blocksize=64,backdim=5,erramount='n',startpos=1,pxshift=2,mkvid=true,lossless=true,play=true,framesout=false,correlation=false,drawfigs=false);

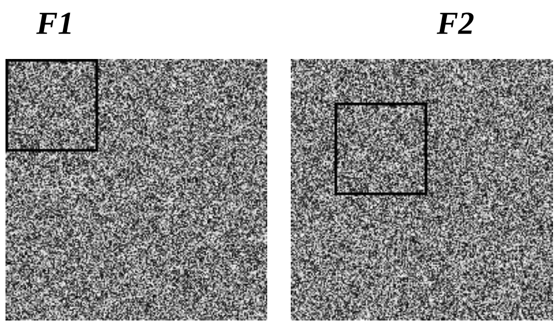


Fig. 2.2.1

Results: (See Appendix E for Results breakdown)

Fig. 2.2.2 shows file size (Y-axis) of my control and test versus pixel shift (X-axis)

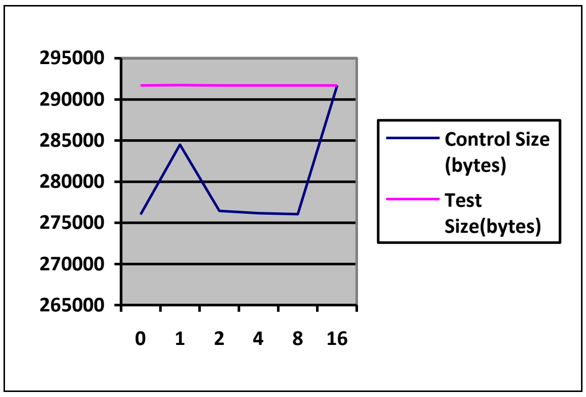


Fig. 2.2.2

Discussion/Conclusions:

After the block is shifted by just one pixel, there is an immediate decline in the performance of *ffmpeg's* motion estimation algorithm. It then recovers after a shift of two pixels and performs well again as the pixel shift increases, up until it fails completely between a shift of 8 and 16 pixels (I found that it breaks down after a shift of 10 pixels, but due to the random nature of the data I am dealing with, this can't be an exact figure for all cases).

Note: To account for the sudden degeneration of motion estimation after a pixel shift of one, I repeated this experiment a number of times and found that this particular random selection of pixels (the block and the frame) just happen to compress badly together in this configuration. Motion estimation has clearly broken down here after a very small pixel shift, so in order for inter-frame compression to be viable, in terms of digital hologram compression, there are fundamental improvements to be made.

Experiment 3 - Introduction

Now I will look at how *ffmpeg's* inter-frame coding algorithm (in particular; motion estimation) deals with errors added to repeated data. Since digital holograms contain a lot of unwanted noise data, it is likely that through lossy compression, two similar digital holograms will have moderately different phase-diagrams. So in order to achieve better compression, a certain amount of flexibility is needed within motion estimation algorithms if they are to be used to effectively compress digital hologram data.

Experiment 3.A

This experiment tests how objects are recognised with motion estimation after various levels of error is applied. The setup of the experiment is similar to **Experiment 2**, two sets of two frames, my test and control.
My control consists of a randomly generated block (64x64 pixels) on a black background (320x320 pixels, all zeros). With a consistent pixel shift of 32 pixels (both X and Y axis) applied to the block in one of the frames.
My test case is identical, only varying degrees of error (random pixels are changed to random values) are applied to the shifted block. This is to see at what point the moved block is no longer recognised by *ffmpeg*. Due to the simplicity of the images, the overall file size will be a lot smaller and the margin of error will also be reduced considerably.
Note: This experiment was first attempted by shifting a block containing all ones (on a black background). Due to the simplicity of the objects, and large amount of redundancy between the two frames, frames were dropped when even one pixel was changed. (Experimentation with frame rates (in/out) yielded no solution). This means that with simple images, small amounts of corrupt data is ignored and removed. This is why a random noise block is shifted instead.

Here we are testing two things, how motion estimation deals with errors added to data (*erramount*) (***Fig. 3.1.1***), and also when this corrupt data is shifted slightly (*pxshift*) (***Fig. 3.1.2***). As the motion estimation algorithm's performance declines, we should see the difference in file size between "motion-estimation enabled" and "motion-estimation disabled" to become less and less. A wide range of *erramount* values will be tested from 0 (the repeated block is totally undamaged) to 4096 (the repeated block is now a totally different one, with nothing in common with the original).
All data is written to file before testing for consistency. (Frames are labelled F1-F2).
Parameters used to generate frames and compress to video:

err(mkblock='a',back='z',block='r',blocksize=64,backdim=5,erramount=0,startpos=1,pxshift=8,mkvid=true ,lossless=true,play=true,framesout=true,correlation=false,drawfigs=false);

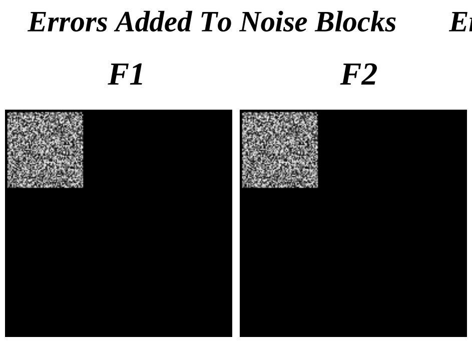


Fig. 3.1.1

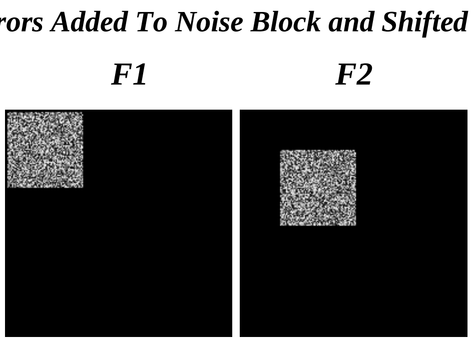


Fig. 3.1.2

Results:

(control.mp4 is equivalent to test.mp4 when *erramount*=0)

“ME”=T is shorthand for Motion Estimation Enabled (True).

pxshift (pixels)	Test.mp4(bytes)”ME=T”	Test.mp4(bytes)”ME=F”	*erramount* (pixels)
0	7335	13251	0
0	7701	13255	128
0	7971	13227	256
0	8449	13259	512
0	9400	13248	1024
0	11086	13266	2048
0	12880	13264	4096

8	7879	13266	0
8	8261	13267	128
8	8528	13262	256
8	9073	13247	512
8	9957	13285	1024
8	11565	13268	2048
8	13093	13242	4096

Discussion/Conclusions

Initially, there is a large file size difference between “ME=T” and “ME=F”, meaning that the repeated block is clearly being recognised in both frames, but as more and more pixel errors are introduced, there is a steady degradation in performance until it stops functioning completely (between 2048 and 4096 pixel errors). As mentioned before, some leeway must be given due to the random nature of this data.

Also, when a small pixel shift is applied, there is again, a reduction in performance, although motion estimation is clearly having an effect. The reduction in performance is proportional to the test with *pxshift*=0 and fails completely around the same point.

Experiment 3.B

This experiment attempts to improve on the previous test of *ffmpeg's* motion estimation/detection algorithm by moving the repeated noise blocks around a noisy background. In the previous test we saw motion estimation under ideal conditions, with a consistent black background, noise patterns are easy to separate from the rest of the image. Now however, I am attempting to separate noise from noise, and I expect severely reduced compression in all test cases.
The experiment is set up exactly as before, only with two randomly generated backgrounds are used for each frame (so no motion-estimation can take place between them) instead of all zeros.

As before, we are testing for two things, how motion estimation deals with errors added to data (*erramount*) (in noisy data this time) (***Fig. 3.2.1***), and also when this data is shifted slightly (*pxshift*) (***Fig. 3.2.2***).
(Frames are labelled F1-F2).
The file sizes should be a lot larger due to the larger amount of noise data, but we are testing for relative size between "*motion-estimation enabled*" and "*motion-estimation disabled*".
Parameters used to generate frames and compress to video:

err(mkblock='a',back='r',block='r',blocksize=64,backdim=5,erramount=0,startpos=1,pxshift=8,mkvid=true ,lossless=true,play=true,framesout=true,correlation=false,drawfigs=false);

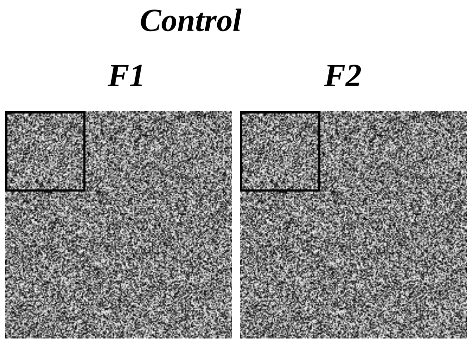


Fig. 3.2.1

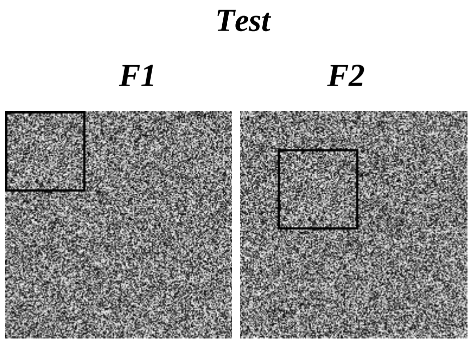


Fig. 3.2.2

Results: (See Appendix E for Results breakdown)

(*control.mp4* is equivalent to *test.mp4* when *erramount*=0)

"ME"=T is shorthand for Motion Estimation Enabled (True).

Fig. 3.3.3 and ***Fig. 3.3.4*** show File sizes with motion estimation enabled/disabled (X-axis) and pixel errors (Y-axis).

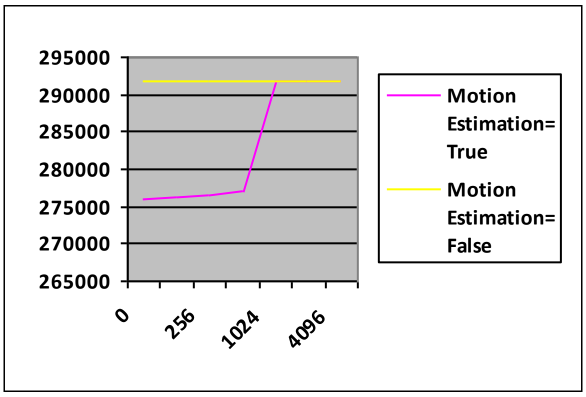


Fig. 3.3.3 (pxshift=0)

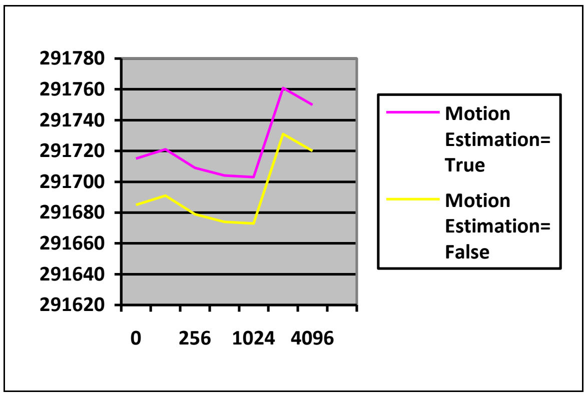


Fig. 3.3.4 (pxshift=8)

Discussion/Conclusions

With the inclusion of additional noise data, *ffmpeg* can deal with errors up to a point, there are a number of fundamental flaws shown in these results however. Firstly, there is a steady decline in compression up until total failure as before. This happens much earlier with the noise background (512-1024 pixel errors, as opposed to 1024-2048 pixel errors in the previous experiment).

Also, when a small pixel shift is applied to the repeated block, motion estimation breaks down completely, even with no pixel errors added. This shows a complete incompatibility with shifted noise data. In order for inter-frame compression to be effective in compressing digital holograms, this must be overcome.

Chapter 5 Analysis and Solution

Now I will take the conclusions from my experiments and determine how the *ffmpeg* standard can be improved to work more effectively with noisy hologram data. We will look at the specific issues this thesis will be tackling, and present a detailed solution to them based on this.

5.1 Problem Modelling

There are two issues I have found with *ffmpeg's* motion estimation algorithm: first is its difficulty in detecting identical noise blocks in noisy data (shifted in adjacent frames), and secondly, it's inability to detect similar noisy data after a small number of pixel errors are introduced.

One aspect of *ffmpeg's* temporal compression algorithms I can make use of, is its inter-frame coding algorithm (without motion estimation), that is keeping similar blocks of data in the same position in adjacent frames, as it seems to perform well when no exhaustive macro-block search is required.

5.2 Solution

My solution consists of a pre/post processing wrapper, written in *GNU Octave*, which will detect the repeated data in both frames by cross-correlating both frames, to produce a vector, and block dimensions. These are written to file so the process can be reversed during decoding.

Then the repeated block will be returned to its original location using the movement vector (the block will be swapped with whatever pixel block is currently occupying that space).

The frames are then passed into *ffmpeg's* inter-frame coding algorithm (without motion detection/estimation) and compressed as normal.

To decompress, the movement vector/block dimensions are read from file and the same elements of the wrapper function are re-used to shift the redundant data back to its correct location.

By definition, lossy compression introduces errors in hologram reconstruction [13] so an extension of my above solution is to implement object recognition despite corruption, since *ffmpeg* also has difficulty in recognizing corruptions (random values changed) in noisy pixel data. This will take the form of an additional image containing the subtracted values of both redundant blocks to be stored alongside the frames, and the redundant block duplicated verbatim. Since they will mostly be identical; this image will mostly contain mainly zero values, and so will be compressed very efficiently. This block will then be summed point by point with the redundant block during decoding.

Chapter 6 Software Design

This chapter covers both high-level and detailed descriptions of the main aspects of my solution and how it interacts with the existing *ffmpeg* system I will also be verifying my solution fit-for-purpose and comparing and contrasting the two systems.

6.1 Introduction

This chapter presents a very high-level description of *ffmpeg's* current inter-frame configuration versus the same system with my design implemented. This is to show the position of my wrapper function in the *ffmpeg* chain and how they will interact with each other

6.2 High Level Design

Currently, motion estimation is carried out by separating one of the frames into sections (macro-blocks) and searches a previously coded raw image for similar data. If redundancies are found (highlighted in green), the duplicate data is stored once, plus a movement vector. Then the frames are compiled to a single video file using *ffmpeg* (***Fig. 6.2.1***).

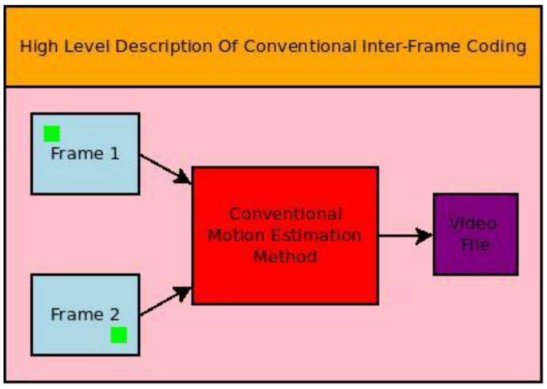

High Level Description Of Conventional Inter-Frame Coding
Frame 1
Frame 2
Conventional Motion Estimation Method
Video File

Fig. 6.2.1

My solution consists of a pre-processing stage for adjacent frames where redundant blocks are detected, and shifted so they are in the same position in each frame, this is to take advantage of *ffmpeg's* ability to compress adjacent frames using inter-frame coding and improves on its weakness of detecting redundancies in noise data (***Fig. 6.2.2***).

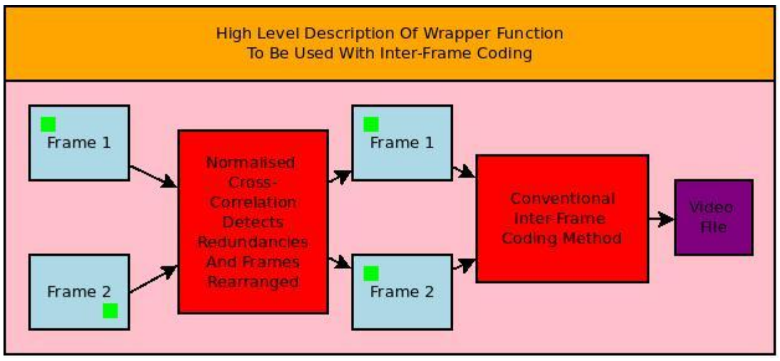

High Level Description Of Wrapper Function To Be Used With Inter-Frame Coding
Frame 1
Frame 2
Normalised Cross-Correlation Detects Redundancies And Frames Rearranged
Frame 1
Frame 2
Conventional Inter-Frame Coding Method
Video File

Fig. 6.2.2

6.3 Detailed Design Options

In order to find redundancies in proximal noise frames, I have come up with a number of block-searching solutions (detailed below). Once the location and dimensions of the block is found, the remainder of the solution is the same; Both of my block-searching algorithms return two points of reference, to the redundant block's position in each frame Then, using these points of reference, the redundant block is shifted until both points line up. The block that used to occupy this area (placeholder) is swapped with the redundant block's original position and rotated 180˚
(see **Section 7.2**).

Now we have two frames with the same (or at least similar, a certain amount of corrupt pixels can be accepted) block in the same position in two adjacent frames. These altered frames are then input to ffmpeg's inter-frame coding algorithm to be compressed as normal (***Fig. 6.3.1***).

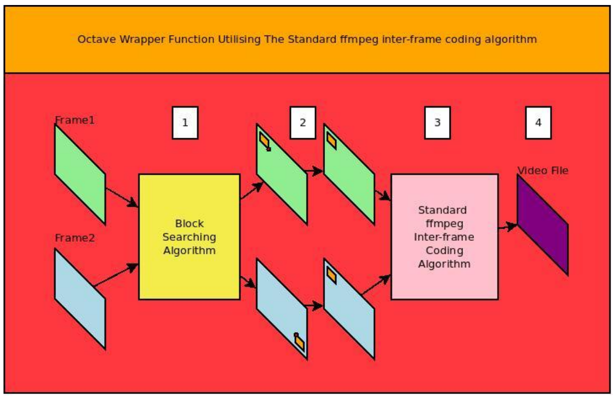

Octave Wrapper Function Utilising The Standard ffmpeg inter-frame coding algorithm
1
2
3
4
Frame1
Frame2
Block Searching Algorithm
Standard ffmpeg Inter-frame Coding Algorithm
Video File

Fig. 6.3.1

Step1 – The redundant pixel block is found using one of the two options discussed below. And reference points are returned
Step2 – Using said reference points, the redundant blocks are shifted so they are in the same position in each frame.
Step3 – Taking advantage of ffmpeg's inter-frame coding algorithm, the two frames are stored without any duplicate data.
Step4 – The video frames are compressed to a single video file.

Note: Decompression just reverses these steps, the video file is decompressed to its composing frames, using the two reference points, we shift the block back to its original position and rotate the placeholder block 180^o to obtain the original frames.

6.4 Design Option 1

With current inter-frame compression techniques, frames are divided up into chunks. Then an exhaustive search is carried out with each of these blocks and the previous frame(s) to test the cross-correlation [19] of each. Through my own research, conventional inter-frame compression methods have performed poorly with this noisy data. This is because the image is only divided in a specific way, and if a repeated block is not a perfect match with a similar block in the other frame, the difference also has to be recorded. Normally this is fine because in standard images the differences are very small, but with noisy data, the disk space required to store these differences can exceed the requirements of the raw data, in which case, the raw data is stored instead, and the inter-frame coding algorithm breaks down (***Fig. 6.4.1***).

Through my own observations, this method has performed poorly in separating noise from noise to detect redundancies, although there is scope to improve on this method by re-implementing block-based motion estimation and experimenting with macro-block sizes and varying quantization levels of noise data.

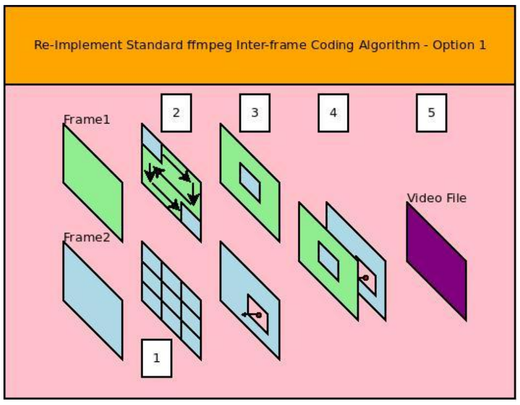

Re-Implement Standard ffmpeg Inter-frame Coding Algorithm - Option 1
Frame1
Frame2
2
3
4
5
1
Video File

Fig. 6.4.1

(This shows future frame estimation, the same process is used for past frame estimation (substitute Frame1 with Frame2 and vice versa))
Step1- One frame is divided into macro-blocks.
Step2- Each block is cross-correlated with the previous frame.
Step3- Duplicate block is removed and pointer stored in place of it (plus block differences).
Step4- Frames are reassembled in their correct order.
Step5- Frames are compressed as a single video file.

6.5 Design Option 2

My second idea moves away from traditional motion estimation algorithms slightly as it doesn't divide the frames into macro-blocks, and exhaustively search each one to find similar regions in a nearby frame. Block-based redundancy detection has been shown to perform poorly with noisy pixel values.

Instead it will utilize normalized cross-correlation [19] on two nearby frames to find the areas most mutually related, to get two approximations of the redundant block. These approximations are then cross-correlated recursively until their dimensions match exactly; this is the redundant block's location (***Fig. 6.5.1***).

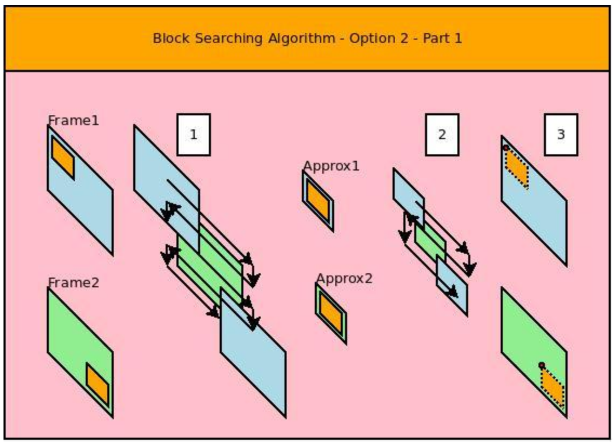


Fig. 6.5.1

Step1 *– Both frames are cross-correlated to obtain an approximation of the redundant pixel block in each frame.*
Step2 *– Both approximations are then cross-correlated as in Step1. This continues recursively until the approximation's dimensions match, then either the redundant block is direct center in both approximations, (in which case, the block dimensions are found by checking each approximation against each other (**Fig. 6.5.2**)) or each approximation is made up entirely of the redundant block.*
Step3 *– the redundant block's location in both frames, and its dimensions are returned for the preprocessing stage.*

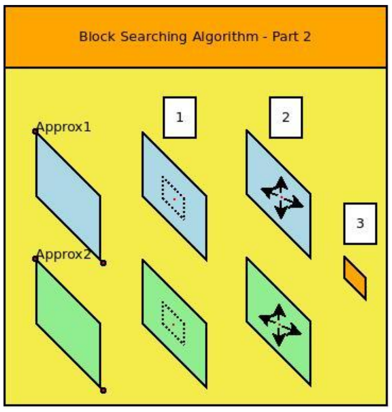


Fig. 6.5.2

Step1 *– The centre point is found in each approximation (this will be the centre point of the redundant block also).*
Step2 *– The redundancy block's dimensions are found by building increasing sized blocks out from the centre point and cross-referencing each, this continues until the blocks are no longer similar.*
Step3 *– After finding the redundant pixel block, its position is recorded in both frames and returned for pre-processing.*

6.6 Option Comparison

Both of my design ideas work with *ffmpeg's* strengths in relation to temporal compression, i.e. inter-frame coding without motion detection; and overcome its difficulty in recognizing noise patterns in noisy data.

Option 1 can still potentially have difficulty in recognizing small details due to its block-based implementation (***Fig. 6.6.1***); it is also likely that the time taken to detect redundancies will be much lower than **Option 2** since it only requires one pass through the image. **Option 2** may require several iterations until the redundant block is found, although **Option 2** will provide better detection of small details, since the entire frames are cross-correlated. **Option 2** will not be able to detect multiple objects in different directions, but it won't be limited to any particular block-size as in **Option 1**.

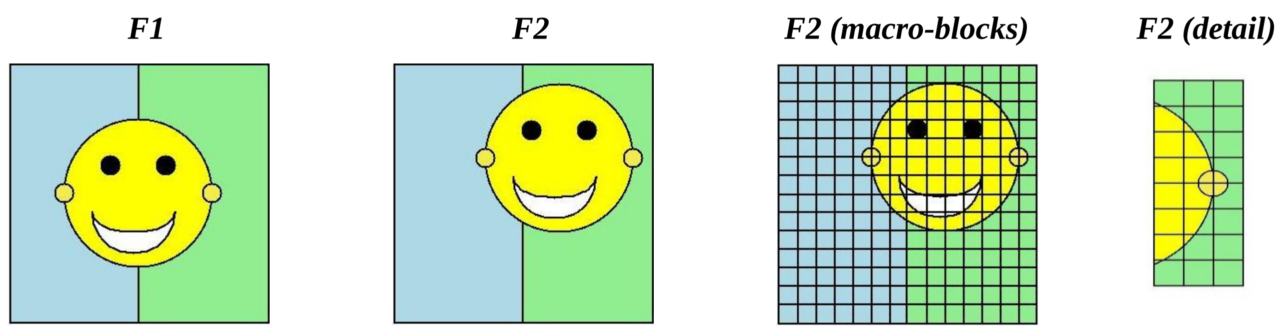


Fig. 6.6.1

This figure shows a simple example of the lack of precision that can occur in block-based motion estimation. We have two adjacent frames (F1, F2).
F2 is divided into segments (macro-blocks) and each one is cross-correlated with F2 to find redundancies. Due to the simplicity of these frames, most of the redundancy will easily be detected and removed, some small details however e.g. the ear, falls between four different macro-blocks, when each of these is cross-correlated with the previous frame, the impact of the correlation of the ear will make very little difference to the overall correlation of the whole block.
The trade-off here is block size versus number of unique blocks that need to be stored. Usually the block size is 16x16 pixels, which provides a good balance between detail and run-time.
Note: the right ear will be detected due to the remainder of the macro-block being identical, the left ear however, will not; because even though the left and right ear are identical, the surrounding background pixels are totally different. With noise images, any area of the image could be called "detailed".

Since the block searching algorithms only have to be used once, (after that, the motion vectors are already stored for reconstruction), **Option 2** could have better application to Production Quality Digital Hologram Videos, and Option 1 for everyday use.

6.5 Design Verification

My chosen solution was verified by confirming that it fulfilled my project goals. It takes advantage of temporal redundancies found in video, by utilizing an existing system (i.e. *ffmpeg*) as a tool to compress Digital hologram data significantly. It overcomes a number of existing weaknesses in *ffmpeg* (object recognition in noisy data) and works with its strengths (inter-frame coding (without motion-estimation)) to achieve adequate compression ratios. This solution is a compromise of two of my goals which was to either optimize an existing temporal compression algorithm to function with digital holography, or develop my own codec for achieving better compression with adjacent digital holograms in a video file.

Chapter 7 Implementation

Now we will look at the software tools and languages used in the past, and what I used to implement my design. We will also examine certain critical design aspects in detail (not covered in my design chapter) and verify that my solution for inter-frame compression performs better than the current method used in the *ffmpeg* standard.

7.1 Introduction

Previous work in this area has used *MATLAB* mostly. I have decided to use *GNU Octave* for most of the coding during the course of this project as it is an open source high level language that is syntactically similar to *MATLAB*.
The version of *GNU Octave* I have been using is *3.6.4*. (Documentation here [17])
ffmpeg is another tool used frequently in this project. *ffmpeg* is an open source, cross-platform multimedia framework allowing users to encode, decode, transcode (convert from one codec to another), mux (multiplex, combine multiple streams e.g. audio, video into one), demux (demultiplex), stream, filter and play almost any form of digital media.
The *ffmpeg* revision I am using is: *git-2014-02-10-4958628*.
I am primarily using *Ubuntu 13.10* to compile and test all of the code in this project.
Although all of the tools used are cross-platform and open-source, I have not tested on other platforms.

7.2 Coding

Here I will detail some of the critical design aspects of my solutions.
Firstly, to improve on one of the weaknesses in current inter-frame compression algorithms (i.e. cross-correlating macro-blocks in detailed data); I decided to cross-correlate the entire image to obtain increasingly precise approximations of redundancy locations (***Fig. 7.2.1,*** Frames Named ***F1*** and ***F2***). The redundant block is clearly shown. This is to remove the possibility of redundant data not lining up perfectly with the macro-block. See **Appendix D Section 1** for code snippets and more detailed examples.

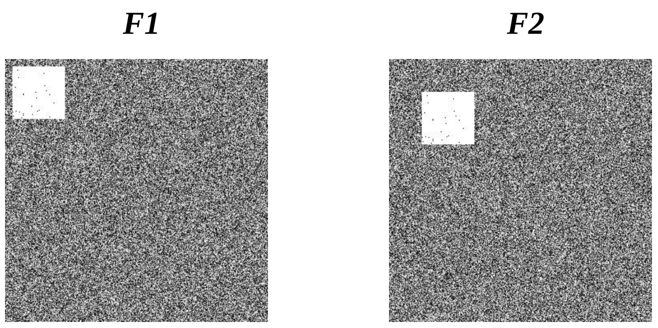


Fig. 7.2.1

Another aspect of my design was to swap blocks of pixels within the same image as part of the pre-processing stage. This was to remove to object detection from *ffmpeg's* motion estimation algorithm (as it has performed poorly with noisy data). In the case of overlapping blocks (i.e. the pixel shift is smaller than the dimensions of the block), simply swapping the data results in duplicated pixels (where the blocks overlap) and loss of surrounding data. This is why the place holding block (block in the position where our redundant block will be moved) is rotated 180 degrees (***Fig. 7.2.2*** and ***Fig. 7.2.3***). See **Appendix D Section 2** for code snippets.

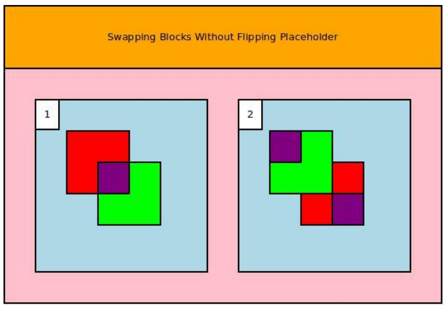


Fig. 7.2.2

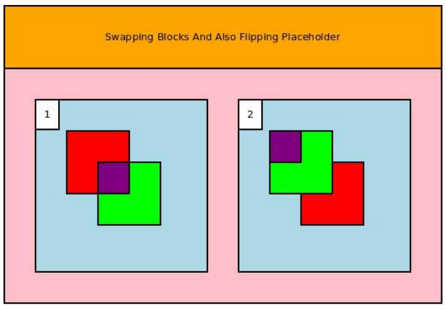


Fig. 7.3.3

*In **Fig. 7.2.2** we see two overlapping blocks (red and green) with the overlapping area highlighted in purple. Step1 shows the blocks before they are swapped, Step 2 shows them after. The individual blocks need to be copied first and then shifted, resulting in duplicate data where the blocks overlap. After the shift, each block is written into the image, but one overwrites the other. Rotating one of the blocks fixes this. We flip the placeholder block so inter-frame compression can still recognize the temporally redundant block (**Fig. 7.3.3**).*

7.3 Verification

My code was tested using a combination of White Box (Unit) and Black Box (System) testing.
In my test suite during the experiment phase, the emphasis of the program was on flexibility. It was designed to be as general-purpose as possible so it could be adapted to satisfy a wide range of tests, e.g. everything was customizable, from block position to pixel shift applied to certain regions, values contained in generated data, etc.
Because of this the limits of each element of my test suite and final solution were tested by inputting extreme values and verifying the results are valid, as a result of white box testing with my test suite, certain handling functions were implemented to prevent invalid outputs e.g. if *pxshift>backgroundsize-blocksize-startpos* the block will move out of bounds and produce an invalid image.(*startpos*= starting position of the pixel block (x,y), *blocksize*=dimensions of movement block, *pxshift*=number of pixels to shift the block by (x,y), *backgroundsize*=dimensions of entire frame). The *randblock* function was ran a number of times to ensure that the block's location was different each time.
My wrapper functions were also tested in the same way; my block-searching algorithms were tested by placing redundant blocks in a variety of extreme and non-extreme locations in each frame, to see if they are detected. The same was done with the block swapping algorithm, additionally; "overlapping" blocks were tested to ensure there was no data loss/duplicate data. These functions are also used for decompression, so additional care was given when testing.
After this, the entire systems were tested (Black-Box). The experiments were re-run to verify valid results, and each wrapper solution was used to compress valid motion estimated video files and decompress/reassemble the original input frames.

7.4 Validation

Validation of my solution was conducted by re-running the experiments from **Chapter 4** (used to test *ffmpeg's* motion estimation algorithms to assess the limits of their effectiveness) on my wrapper solutions. The results were then compared with *ffmpeg's* current inter-frame compression algorithm.

Chapter 8 Evaluation

This chapter subjects my solution to the same experiments as was used to test *ffmpeg's* limitations initially. This is to get a good comparison between the two solutions, and assess how my method of inter-frame compression improves on the existing solution, when applied to noisy Hologram data.

8.1 Metrics

The metrics I used to measure the effectiveness of my solution was, most importantly: resulting video file size (bytes). Run-time was also observed to assess efficiency.

8.2 Experimental Setup

For full details on the experimental setup, see **Chapter 4.** Both **Experiment 2.B** and **Experiment 3.B** were repeated (with my wrapper included) to evaluate the performance difference in terms of disk space used. This is because they found the limitations of *ffmpeg's* motion estimation algorithm, while **2A** and **3A** document ideal conditions where the standard motion-estimation algorithm thrives. These would not apply to my wrapper function because if it works effectively, the results from **3.A/3.B** will be comparable to **2.A/3.A**

8.3 Results (See Appendix F for Results breakdown)

Experiment 2.B

This experiment tests how well each motion-estimation solution deals with pixel shifts in noisy data. ***Fig. 8.3.1*** shows *filesizes* (Y-axis) versus *pixelshift* (X-axis). For full details see **Chapter 4**.

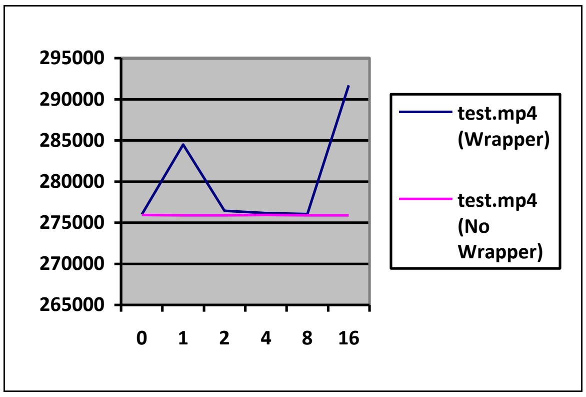


Fig. 8.3.1

Discussion/Conclusions

Since the two test cases were done using random data at different times, we can't compare them value-for-value. The purpose of this experiment is to compare the relative change in values as more and more pixel shifts are introduced. The control then in each test case is when
pxshift (Pixel Shift) = 0;

The results from *ffmpeg's* standard inter-frame compression algorithm (No Wrapper) begin to break down between a pixel shift of 8 and 16. With the wrapper introduced, this does not occur.
pxshift was increased further until the frame limits were reached, showing no signs of failure.

Note: values only vary slightly due to the random nature of noise data, some configurations compress better than others.

Experiment 3.B (See Appendix F for Results breakdown)

This experiment tests how well each motion-estimation solution deals with pixel errors added to noisy data. ***Fig. 8.3.2*** and ***Fig. 8.3.3*** shows *filesizes* (Y-axis) versus *erramount* (X-axis). For full details see **Chapter 4**.

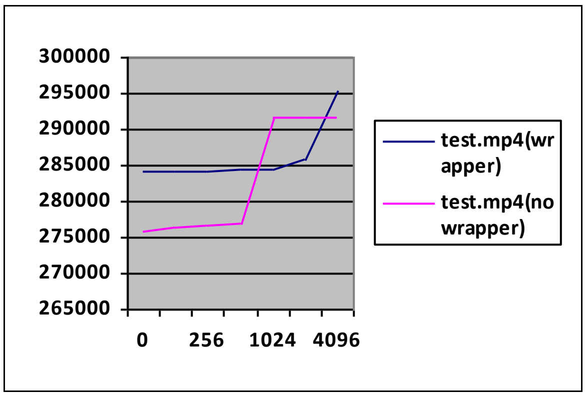


Fig. 8.3.2(Pxshift=0)

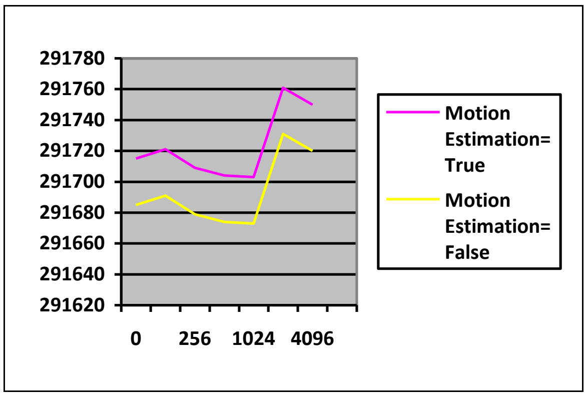


Fig. 8.3.3(pxshift=8)

Discussion/Conclusions

Without the wrapper function, the standard *ffmpeg* motion estimation algorithm's performance slowly decays as more and more pixel error is added, until it fails completely. When a pixel shift is added there is no motion estimation whatsoever (see **Chapter 4 Experiment 3.B Results**). My solution begins to suffer around the 1024-2048 pixel error mark, where half of the redundancy block is totally new data. Despite a pixel shift being added, the results are much the same, meaning motion-estimation is dealt with much better than error detection.

8.4 Validity

Due to the large amount of noisy data (created randomly every time each frame is generated), small discrepancies are likely between test cases. If motion detection is having an effect however, the size difference is much greater. As we can see, there is a significant difference in file size which suggests that the wrapper function has had a positive effect on compression. There is also a sizeable (though not quite as significant) improvement in the area of error detection in redundant data blocks as part of motion estimation.

Chapter 9 Discussion and Conclusion

This chapter will discuss my solution's effectiveness; identify certain skills I acquired during the course of doing this project. It will also conclude the thesis by reviewing the project, and describe future work in the area of inter-frame compression as applied to Digital Holography.

9.1 Solution Review

My solution improved on an existing system by providing a work-around for its weaknesses (*ffmpeg's* motion detection in noisy data) and utilizing its strengths (inter-frame coding). The results showed a significant reduction in disk space compared with *ffmpeg's* stock solution, with comparable run-time.

9.2 Project Review

This project set out to analyze *ffmpeg's* limitations as applied to Digital Holography and attempt to improve on the existing standard, specifically in the area of motion estimation. When the limits were found, a number of possible solutions were examined and implemented to out-perform the existing motion estimation algorithm. This work can someday be used as a basis for a video codec for digital holograms

9.3 Key Skills

I acquired several skills over the duration of this project. I became proficient in using *ffmpeg* in a practical setting, especially in utilizing some of its advanced elements e.g. various motion estimation techniques. I advanced my knowledge significantly in both the areas of digital holography, and video codec techniques. I greatly improved my programming skills, notably in the *GNU Octave* language. My time management and planning skills were also enhanced during the project management stage to prioritize certain duties to efficiently carry out this project.
I also gained a better ability in researching publications and in devising and carrying out high quality experiments. All of these skills I think will benefit me in future projects.

9.4 Conclusion

This thesis is concerned with the application of a conventional video codec (*ffmpeg*) to a novel form of multimedia called digital hologram video. A number of improvements were made to the existing *ffmpeg* standard in the areas of motion-detection of noisy data, and detection of corrupt pixel data in redundant pixel regions.

By combining a pre/post-processing wrapper function with the existing inter-frame coding algorithm contained in the *ffmpeg* standard, an impressive compression ratio can be achieved with noisy images. This was not possible before as detection of redundant pixel data in a noisy environment is very difficult. Also implemented was an error detection algorithm, where a highly similar pixel block is recognized and its differences stored. This was another area I found *ffmpeg* had difficulty in detecting redundancies.

9.5 Future Work

There is huge scope to expand in the area of Digital Holography, in particular the area that this thesis works towards, i.e. A Digital Hologram Video codec standard. One aspect is improving and optimizing my chosen video codec solution by combining this inter-frame compression algorithm with lossy intra-frame compression solutions discussed previously.

An added feature for adjustable macro-block sizes could be beneficial for optimizing performance, since it could reduce the computation required for large repeated blocks of data.

My error detection algorithm could be applied to two different areas; redundant pixel blocks can be detected despite having small errors, in which case the errors are discarded, and similar data blocks can be recognized and stored together to achieve higher compression ratios, in which case, the corruptions are stored.

My second design idea for inter-frame redundancy detection, was too time-constraining for practical real-time video compression use, and lacked multi-directional redundancy recognition, but due to its ability to detect arbitrary sized redundancy blocks entirely, and increased detail recognition, if run-time was not as great a priority; could be applied to the area of pattern recognition. This could also be combined with my error detection algorithm to recognize larger regions of similar pixel data in greater detail.

References

[1] Risto Näsänen, Tristan Colomb, Yves Emery, Thomas J. Naughton: “Enhancement of three-dimensional perception of numerical hologram reconstructions of real-world objects by motion and stereo”, Optics Express Vol. 19, Issue 17, pp. 16075-16086, August 2011

[2] Tomi Pitkäaho and Thomas J. Naughton: “Calculating depth maps from digital holograms using stereo disparity”, Optics Letters, Vol. 36, Issue 11, pp. 2035-2037, May 2011

[3] Melania Paturzo, Pasquale Memmolo, Andrea Finizio, Risto Näsänen, Thomas J. Naughton, Pietro Ferraro: “Holographic Display of synthetic 3D dynamic scene”, 3*D Research*, vol. 1, no. 2, pp. 31-35, June 2010

[4] Thomas J. Naughton, Yann Frauel, Bahram Javidi, and Enrique Tajahuerce: “Compression of digital holograms for 3D imaging” Optics in Information Systems Vol. 13, No. 1, May 2002

[5] Thomas J. Naughton, Yann Frauel, Bahram Javidi, and Enrique Tajahuerce: “Compression of digital holograms for three-dimensional object reconstruction and recognition”, Applied Optics Vol. 41, no. 20, pp.4124-4132, July 2002

[6] Thomas J. Naughton, John B. McDonald, and Bahram Javidi: “Efficient compression of Fresnel fields for Internet transmission of three-dimensional images”, Applies Optics Vol. 42, No. 23, August 2003

[7] Thomas J. Naughton and Bahram Javidi: “Compression of encrypted three-dimensional objects using digital holography”, Opt. Eng. Vol. 43, Issue 10, pp. 2233-2238, October 2004

[8] Alison E. Shortt, Thomas J. Naughton, and Bahram Javidi: “Compression of digital holograms of three-dimensional objects using wavelets”, Optics Express Vol. 14, Issue 7, pp. 2625-2630, April 2006

[9] Alison E. Shortt, Thomas J. Naughton, and Bahram Javidi: “Compression of Optically Encrypted Digital Holograms Using Artificial Neural Networks”, Journal of Display Technology Vol. 2, no. 4, December 2006

[10] Emmanouil Darakis, Thomas J. Naughton, John J. Soraghan: “Compression defects in different reconstructions from phase-shifting digital holographic data”, Applied Optics Vol. 46, no. 21, pp. 4579-4586, March 2007

[11] Alison E. Shortt, Thomas J. Naughton, and Bahram Javidi: “Histogram approaches for lossy compression of digital holograms of three-dimensional objects”, IEEE Trans. Image Processing Vol. 16, Issue 6, pp. 1548-1556, June 2007

[12] Emmanouil Darakis, Thomas J. Naughton: “Compression Of Digital Hologram Sequences Using MPEG-4”, Department Of Computer Science, National University Of Ireland – Maynooth, County Kildare, Ireland; RFMedia Laboratory, University of Oulu, Oulu Southern Institute, 2009

[13] Emmanouil Darakis, Marcin Kowiel, Risto Näsänen, Thomas J. Naughton: “Visually Lossless Compression of Digital Hologram Sequences”, RFMedia Laboratory, University of Oulu, Oulu Southern Institute, 84100 Ylivieska, Finland; Department Of Computer Science, National University of Ireland – Maynooth, County Kildare, Ireland, 2010.

[14] Hanwei Jin: “Hologram Video Codec by ffmpeg”, Final Year Project, Department Of Computer Science, National University of Ireland – Maynooth, 2013

[15] T.Wiegand, G.J. Sullivan, G. Bjøntegaard, A.Luthra: “Overview of the H.264/AVC Video Coding Standard”, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, No. 7, July 2003

[16] ffmpeg Official Documentation – http:// www.ffmpeg.org/documentation.html

[17] *GNU Octave* Manual – http://www.gnu.org/software/octave/octave.pdf

[18] Windows FAQ - http://windows.microsoft.com/en-ie/windows/codecs-frequently-asked-questions#codecs-frequently-asked-questions=windows-7

[19] Cross-correlation Tutorial http://mathworld.wolfram.com/Cross-Correlation.html

Appendix A

ffmpeg/ffplay Documentation Summary

This document summarises some relevant *ffmpeg/ffplay* parameters (from the official *ffmpeg* documentation [16]) with a breakdown of their function and a number of examples describing their use. These examples are all used mainly in the experimentation section of this thesis and physical implementation of the wrapper function.

Ffmpeg/ffplay usage examples

“ffmpeg -f image2 -y -i frame%d.png -g 0 -loglevel verbose test.mp4”
“ffmpeg -i test.mp4 -y -q:v 1 -loglevel verbose frame%d.png”
“ffmpeg -f image2 -y -i frame%d.png -me_method full -loglevel verbose test.mp4”
“ffmpeg -f image2 -y -i frame%d.png -c:v libx264 -preset veryslow -crf 0 -loglevel verbose test.mp4”
“ffplay -x 400 -y 400 -window_title 'CONTROL' -i control.mp4 -loop 0”

ffmpeg parameter breakdown

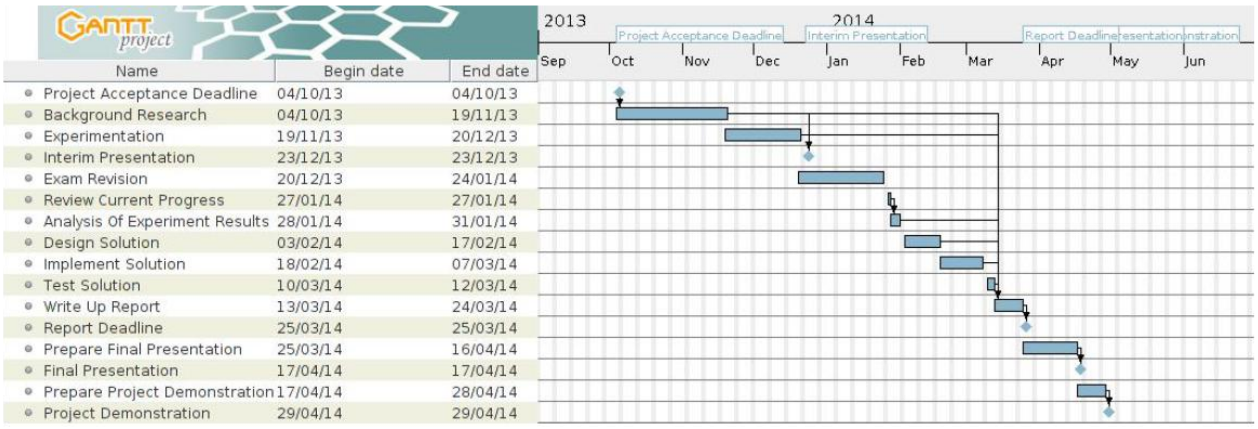
-f <string>= “Force Format”, Force input/output format, detects patterns in the input file(s) to decide the input file format. Setting this to “image2” indicates that this is an image. This can be applied to the output format but is usually guessed from the extension of output file(s).
-r <int>=set framerate of input/output file (depending on its location in input string)
-y = Overwrite output files with the same filename, without prompt.
-n = Used in place of “-y”, Halts all activity and exits if ffmpeg attempts to write a file that already exists
-i <string> = “Input File Name”, specify file location of input file(s) here.
-q<:stream_specifier> <int>=“Fixed Quality Scale”, Used to manage data loss through lossy compression methods. “stream_specifier” is a char, specifying the type of data being dealt with. 'v' indicates the data is a video, another option is 'a', indicationg audio. -q also takes in an integer (between 1 and 31) detailing the quality of the output file, with 1 being the highest achievable quality (lossless).
-c<:stream_specifier> <string> = “Codec”, can be placed either before or after an output file, indicating the encoder/decoder depending on its position relative to the output file. “stream_specifier” has the same function as detailed above. -c also takes in a string which selects the encoder/decoder to use, e.g. “libx264” for lossless H.264 video.

-preset <string> = part of the H.264 video format. It is a collection of options that will provide a certain encoding/decoding speed to compression ratio. e.g. “ultrafast”, providing the fastest and also least compressed output. “veryslow”, providing the slowest and also best compressed output, “medium”, (default) has the best compromise between the two.
-crf <int> = “Constant Rate Factor”, Part of the H.264 video format. Takes in an integer (between 0 and 51), which determines the range of the quantizer scale. where 0 is lossless, 23 is the default, and 51 is the highest data loss. The range is exponential, so increasing the crf by 6 is roughly half the bitrate, while reducing by 6 is roughly double the bitrate. 18 is understood to be the visually lossless level, but for the purposes of my research i required true lossless compression.
-g <int> =“i-frame”, used to set the GOP (group of pixels) size, setting this to zero essentially disables motion estimation.
-me_method <string> = ”Motion Estimation Method”, details the algorithm for carrying out motion estimation on video frames. It is dependant on the video codec being used. For the H.264 video format (in descending order of quality): “tesa” (transformed exhaustive search algorithm), “full” (exhaustive search algorithm), “umh” (uneven multi-hexagon search), “hex” (hexagon based search),”epzs” (default, equivalent to a diamond-based search),”zero” (“disables motion-estimation”, setting the GOP to zero, more effective).
-loglevel <string> = sets the logging level used by the library, used for debugging, default is “info”, the other two i used was “quiet” to disable logging, and more often “verbose”, which as the name suggests is the same as “info” only more detailed.
<string> = (last string in sequence), this will specify the output filename and format. The file extension triggers the “-f” flag detailed above.

ffplay Parameter Breakdown (mainly used for debugging)
-x <int> = Width of playback window
-y <int> = Height of playback window
(useful for zooming in and viewing small details in images)
-window_title <string> = displays a title above the playback window
-i <string> = input file name (either video or image file)
-loop <int> = sets number of times to play back video, 0 means loop indefinitely

Appendix B - Project Plan/Actual Timeline

Initial Plan

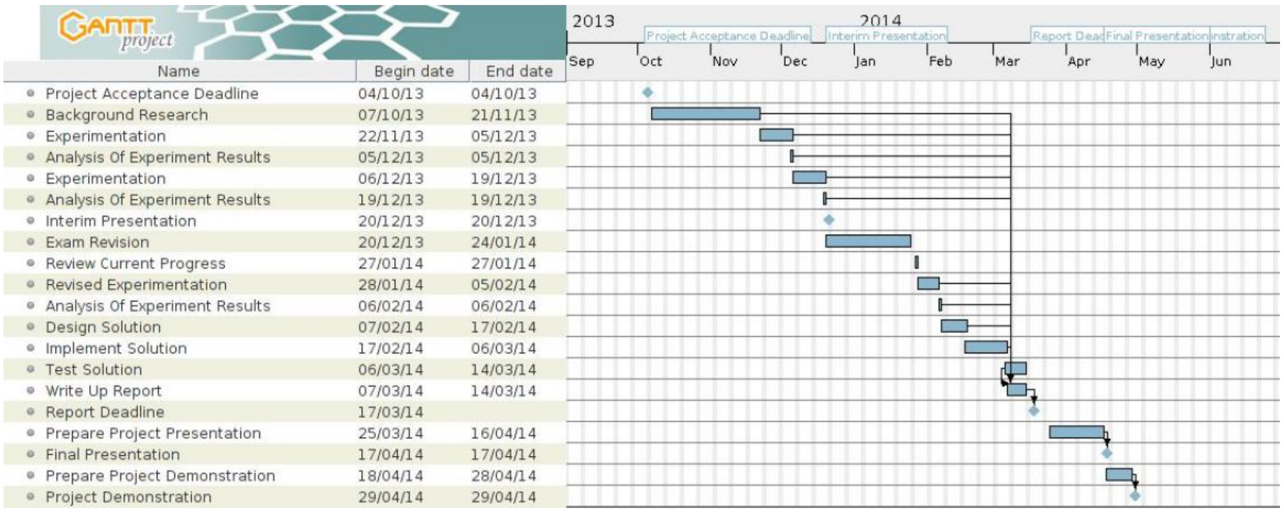


Name	Begin date	End date
Project Acceptance Deadline	04/10/13	04/10/13
Background Research	04/10/13	19/11/13
Experimentation	19/11/13	20/12/13
Interim Presentation	23/12/13	23/12/13
Exam Revision	20/12/13	24/01/14
Review Current Progress	27/01/14	27/01/14
Analysis Of Experiment Results	28/01/14	31/01/14
Design Solution	03/02/14	17/02/14
Implement Solution	18/02/14	07/03/14
Test Solution	10/03/14	12/03/14
Write Up Report	13/03/14	24/03/14
Report Deadline	25/03/14	25/03/14
Prepare Final Presentation	25/03/14	16/04/14
Final Presentation	17/04/14	17/04/14
Prepare Project Demonstration	17/04/14	28/04/14
Project Demonstration	29/04/14	29/04/14

Fig. B.1

This shows the original plan of my final year project. It differs greatly from my actual timeline due to extended time required in the experiment phase of my project.

Actual Timeline



Name	Begin date	End date
Project Acceptance Deadline	04/10/13	04/10/13
Background Research	07/10/13	21/11/13
Experimentation	22/11/13	05/12/13
Analysis Of Experiment Results	05/12/13	05/12/13
Experimentation	06/12/13	19/12/13
Analysis Of Experiment Results	19/12/13	19/12/13
Interim Presentation	20/12/13	20/12/13
Exam Revision	20/12/13	24/01/14
Review Current Progress	27/01/14	27/01/14
Revised Experimentation	28/01/14	05/02/14
Analysis Of Experiment Results	06/02/14	06/02/14
Design Solution	07/02/14	17/02/14
Implement Solution	17/02/14	06/03/14
Test Solution	06/03/14	14/03/14
Write Up Report	07/03/14	14/03/14
Report Deadline	17/03/14	
Prepare Project Presentation	25/03/14	16/04/14
Final Presentation	17/04/14	17/04/14
Prepare Project Demonstration	18/04/14	28/04/14
Project Demonstration	29/04/14	29/04/14

Fig. B.2

This shows the actual timeline of my final year project. It includes problems and changes to the initial plan detailed in chapter 3.

Appendix C - Terminology

***Lossy* –** lossy compression is a data encoding method that compresses data by discarding some less important parts of it.

***Lossless* –** (opposite of lossy) lossless compression allows the original data to be perfectly reconstructed from the compressed data.

***Intra-Frame* –** intra-frame coding refers to the fact that the various lossless and lossy compression techniques are performed relative to information that is contained only within the current frame

***Inter-Frame* –** Inter-frame coding tries to take advantage from temporal redundancy between neighboring frames allowing higher compression rates.

***Quantization* –** is a lossy compression technique achieved by compressing a range of values to a single quantum value

***Macro-Block* –** A macro-block is a processing unit in image and video formats. It has uses in lossy intra-frame compression techniques (e.g. Discrete Cosine Transform) and inter-frame techniques (e.g. motion estimation).

I-frame – Inline Frame, an I-frame is a single frame that is compressed independently of the frames that precede and follow it and stores all of the data needed to reconstruct that frame.

P-frame – Predicted Frame, P-frames are constructed from I-frames and contain only the data that have changed from the preceding I-frame

B-frame – Bi-directional frames, B-frames are similar to P-frames but are constructed from the I-frames and P-frames preceding and following them.

***GOP* –** Group of Pictures, represented by (M, N). It is used in inter-frame compression to specify the distance between each I-frame (M), and the distance between each anchor (P-frame/I-frame) frame (N).

***Codec* –** A codec is a device or computer program capable of encoding or decoding a digital data stream or signal [18]

Appendix D – Implementation Examples

Section 1

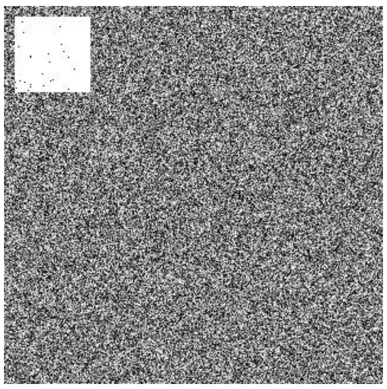
This section shows the process of approximating redundant pixel locations using several iterations of my recursive correlation function. Here we see our two frames ***F1*** and ***F2***:

Normalised cross-correlation is achieved in *GNU Octave* with the following code:

C=xcorr2(a,b,"coeff");

"coeff" - Scales the normalized cross-correlation on the range of [0 1] so that a value of 1 corresponds to a correlation coefficient of 1 [17], [19].
A check is then implemented which equates the dimensions of each approximation. If they don't match, the program iterates again; If they match, either each approximation is made up entirely of the redundant block;
or it is dead centre in each, in which case, an increasing block is constructed in each and cross referenced until they no longer similar, at this point is the dimensions of the redundant block.

F1



F2

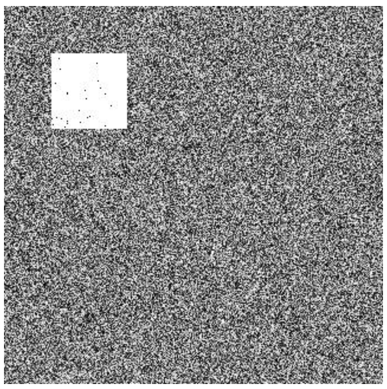
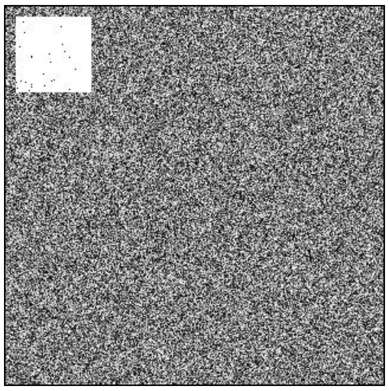
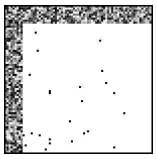


Fig. D.1

The first approximation of the redundant block's location is found by using normalised cross-correlation [19] on the entire images. Shown here are ***Approx1a*** and ***Approx 1b (A border has been drawn around them to see their edges)***

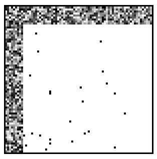


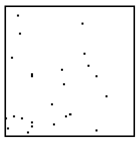
Approx1a



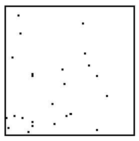
Approx 1b

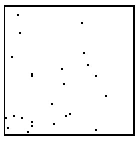
These approximations are then cross-correlated again, to obtain a more accurate set of approximations:


Approx2a


Approx 2b

The approximations are getting closer and closer to the redundant block (approx2b is now exactly made up of the redundancy).


Approx3a


Approx 3b

Approx3a and approx3b are now exactly the same dimensions and contain the redundant block exactly.

Section 2 – Block swapping algorithm

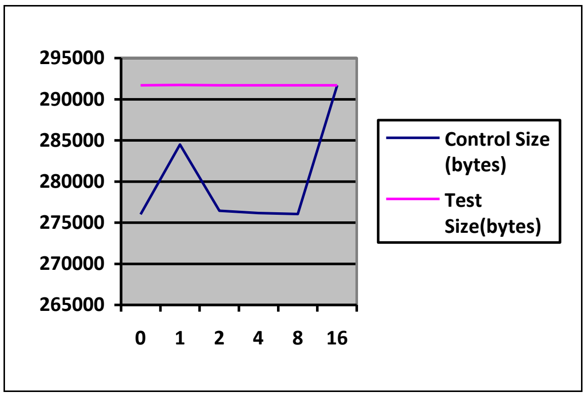
This section contains an extract from the wrapper function corresponding to **Section 7.2.** It details the code used to swap certain pixel blocks given a *frame*, two points of reference (*P1*,*P2*) and block dimensions (*blockrows*, *blockcols*). One of the swapped blocks is rotated 180˚ to prevent loss of data (rot90(rot90(block))).

```
function swapblock(frame,p1,p2,blockrows,blockcols)
        tmpframe=frame;
 frame(p1{1}:p1{1}+blockrows1,p1{2}:p1{2}+blockcols1)=rot90(rot90(tmpframe(p2{1}:p2{1}+bl
ockrows-1,p2{2}:p2{2}+blockcols-1)));
    frame(p2{1}:p2{1}+blockrows-1,p2{2}:p2{2}+blockcols-1)=tmpframe(p1{1}:p1{1}+blockrows-
1,p1{2}:p1{2}+blockcols-1);
    imwrite(frame,"frame2.png");
    swapblockback(frame,p2,p1,blockrows,blockcols);
```

Appendix E – Assessment of ffmpeg's Motion Estimation Algorithms

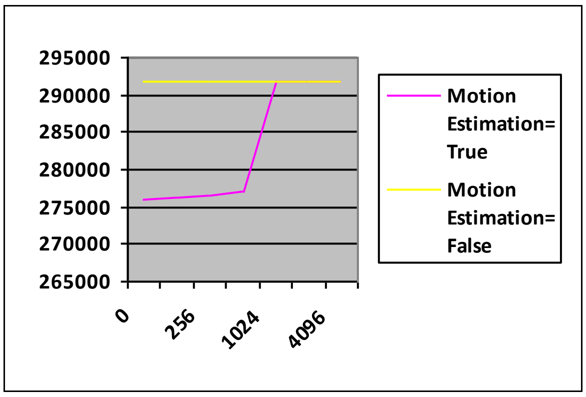
Experiment 2.B

Control Size (bytes)	Test Size (bytes)	Pixel Shift (No. Of Pixels)
276012	291690	0
284489	291722	1
276442	291706	2
276148	291699	4
276066	291682	8
291695	291689	16

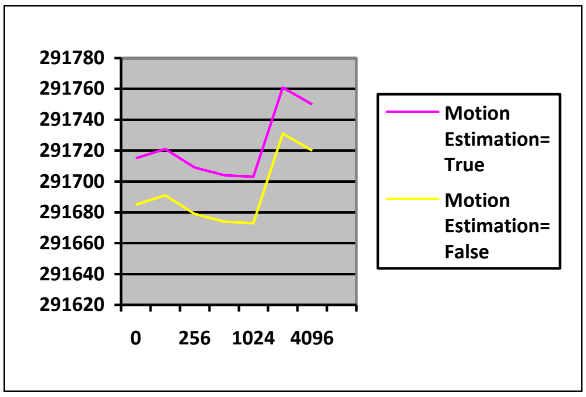


Experiment 3.B

pxshift (pixels)	Test.mp4(bytes)"ME=T"	Test.mp4(bytes)"ME=F"	*erramount* (pixels)
0	275892	291692	0
0	276304	291680	128
0	276559	291671	256
0	277065	291691	512
0	291722	291692	1024
0	291748	291718	2048
0	291733	291703	4096



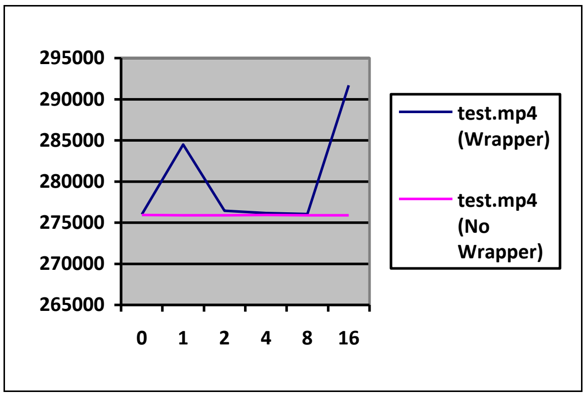
pxshift (pixels)	Test.mp4(bytes)"ME=T"	Test.mp4(bytes)"ME=F"	*erramount* (pixels)
8	291715	291685	0
8	291721	291691	128
8	291709	291679	256
8	291704	291674	512
8	291703	291673	1024
8	291761	291731	2048
8	291750	291720	4096



Appendix F - Assessment of My Solution

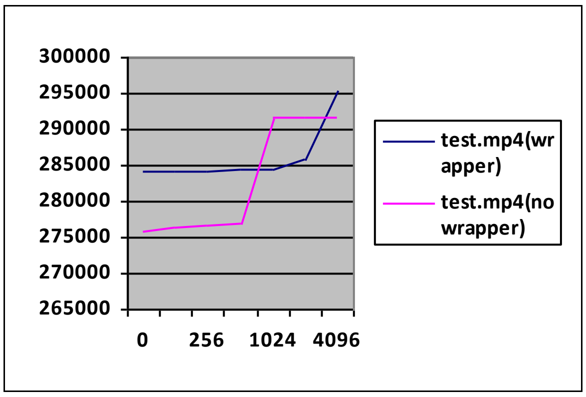
Experiment 2.B

Pixel Shift (No. Of Pixels)	Test.mp4 (bytes) (No Wrapper)	Test.mp4 (bytes) (Wrapper)
0	276012	275925
1	284489	275901
2	276442	275903
4	276148	275942
8	276066	275892
16	291695	275910

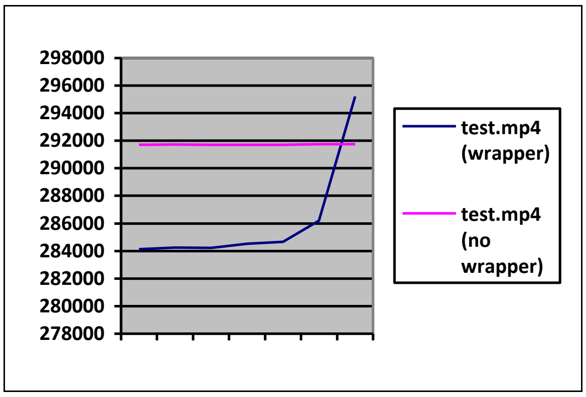


Experiment 3.B

Pxshift(pixels)	*erramount* (pixels)	Test.mp4(bytes) (No Wrapper)	Test.mp4(bytes) (Wrapper)
0	0	275892	284166
0	128	276304	284216
0	256	276559	284263
0	512	277065	284465
0	1024	291722	284563
0	2048	291748	285825
0	4096	291733	295602



Pxshift(pixels)	*erramount* (pixels)	Test.mp4(bytes) (No Wrapper)	Test.mp4(bytes) (Wrapper)
8	0	291715	284124
8	128	291721	284257
8	256	291709	284224
8	512	291704	284523
8	1024	291703	284667
8	2048	291761	286195
8	4096	291750	295217


test.mp4 (wrapper)
test.mp4 (no wrapper)