University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master of Science in Computer Science	Spring semester, 2015 Open access
Writer: Thomas Hinna	…… (Thomas Hinna)
Faculty supervisor: Tom Ryen, Kjersti Engan External supervisor(s):	
Thesis title: Chest compression frequency measurement from smartphone video	
Credits (ECTS):	
Key words: Motion analysis, Android, CPR, video analysis, FFT,	Pages: 81 + enclosure: Source code + Android test results + DVD Stavanger, 15.06.2015 Date/year

Chest compression frequency measurement from smartphone video

Thomas Hinna

June 15, 2015

Abstract

Dispatcher assisted telephone CPR plays an important role in increasing survival rate after cardiac arrest. Providing the medical emergency dispatcher with real time chest compression frequency measurements can improve this even further. The dispatcher would be able to give more detailed instructions and reduce unnecessary hands-off time by getting reports that the compressions has stopped.

The main goal in this thesis was to create an algorithm that can measure the frequency of chest compressions based on video from a smartphone, in real time. Several prototype algorithms has been implemented and tested in MATLAB in order to find the best method. We used video annotations to compare the results to the reference frequency in several test videos.

Our proposed algorithm uses a live camera feed from a smartphone and a combination of differential motion analysis and discrete Fourier transform, to measure the chest compression frequency in real time. The algorithm utilizes a region of interest strategy to avoid interference from other people in the frame. The final algorithm has been implemented as an Android application, where all the calculations are done on the phone.

The final algorithm generally shows very good results. It handles both continuous compressions and CPR 30:2, as well as variations in compression rate. There are one major issue however, if the person doing CPR has long and loose hair the measured frequency is often half of the true frequency. Future work will be to fix this issue, as well as making a Android library based on the algorithm, so that other applications can easily use it.

Acknowledgements

This thesis, together with the prototypes and Android application, is the results of a Master thesis project assigned by the Faculty of Science and Technology at the University of Stavanger.

I would like to thank my supervisors Tom Ryen and Kjersti Engan for invaluable support and feedback throughout the entire thesis. They have been an inspiration with their great understanding and enthusiasm to the field.

I would also like to thank Laerdal Medical for giving me access to their facilities and their employees who helped me with the final tests as well as information needed during this thesis. A special thanks to Tonje Søraas Birkenes for her guidance in the world of CPR.

Stavanger, June 2015

Thomas Hinna

Contents

1 Introduction

When someone suffers from cardiac arrest, they become unresponsive and their breathing stops or becomes abnormal [1]. In these cases, it is vital to initiate cardiopulmonary resuscitation (CPR) as quickly as possible and to perform the chest compressions at a recommended rate [2]. CPR guidelines recommend a rate above 100 compressions per minute (cpm), and research suggests that a rate between 100 and 120 cpm gives a better chance of survival [3][4].

Dispatcher assisted telephone CPR already plays an important role in increasing survival rate after cardiac arrest [5], and one can easily envision taking this one step further. Providing the medical emergency dispatcher with real time information on the compression rate will allow him or her to give instructions that are more detailed. This could ensure that the chest compressions happen at the recommended rate, and at the same time reduce hands-off time by reporting when the compressions stop.

There already exists an Android application that can measure compression frequency in real time (ZOLL PocketCPR, only meant for training and practice purposes, [6]), which uses the accelerometer to measure the frequency of the compressions. The user needs to hold the smartphone when doing CPR. This seems to be a cumbersome solution that could easily lead to problems; the user will need to hold the smartphone correctly all the

time, the microphone or loudspeaker could be covered up and it could easily introduce more stress in an already stressful situation. Also, when doing CPR 30:2, the user would need to drop and pick up the smartphone when performing breaths, which would waste unnecessary time.

Our idea is to use the camera on the smartphone instead. A person finding someone in need of CPR would start an application on the smartphone and then place the smartphone on the ground next to the person in cardiac arrest. Figure 1.1 shows how this scenario could look like. The application will dial the correct medical emergency number, based on the location, and measure the frequency in real time using the feed from the smartphone camera. The results will be sent to the medical emergency dispatcher and optionally displayed on the smartphone screen. The dispatcher can then use the frequency measurement to give better instructions.

Figure 1.1: Example scenario using smartphone to measure frequency.

1.1 Goals and objectives

The main goal of this work is to create an algorithm that can measure the frequency of chest compressions based on video from a smartphone. The algorithm needs to work in real time, and all computations will happen on a smartphone, meaning that the algorithm cannot be to computationally expensive. Several methods should be tested to find the one that is most suitable, and the final algorithm should be implemented as an Android application. Finally, the Android application needs to be properly tested in all possible situations.

1.2 Contributions

- An algorithm to measure CPR frequency in real time based on smartphone video.
- An Android application implementing the algorithm.
- A MATLAB video annotation program, used to find reference frequency from CPR videos.
- Documented tests on the new algorithm.

1.3 Thesis outline

This thesis will be structured as follows

Chapter 2 General descriptions and background material is given on topics such as Android, MATLAB, motion analysis and discrete Fourier transform. These topics are essential for the readers understanding of the work done in the thesis.

Chapter 3 Explains the algorithm prototypes developed and tested in this thesis.

Chapter 4 This chapter provides a more detailed description of the chosen algorithm and its implementation in Android.

Chapter 5 A presentation and analysis of the results from the tests done on the different prototypes and from the Android implementation.

Chapter 6 Conclusion and future work.

2 Background

This chapter will introduce the technologies and software used during the work on this thesis. The first section introduce the software used. The subsequent sections covers technologies and algorithms used.

2.1 Software used

This section gives the reader a brief introduction to Android, Android Studio and MATLAB.

2.1.1 Android

Android is an open source platform and operating system [7]. It is installed on more than 1 billion smartphones and tablets worldwide, giving it the largest installed base of any mobile platform [8] [9]. Although Android is mostly designed to run on smartphones and tablets, there are also versions that are designed to run on smartwatches, TVs and in cars [10]. For this thesis we will only consider smartphones/tablets. Android applications are mainly developed using Java.

2.1.2 Android Studio

Android Studio is an integrated development environment (IDE) used to develop Android applications [11]. It has recently become the official IDE for Android and it is based on IntelliJ IDEA. Android Studio has all the features needed to create an Android application, including a customizable build system, a rich layout editor, good debugging capabilities and useful code templates.

2.1.3 MATLAB

MATLAB is a development environment and a high-level programming language developed by Mathworks [12]. It is an abbreviation for matrix laboratory, and it is designed to operate mostly on matrices and vectors [13]. It has numerous built-in functions and excellent debugging tools, making it to a very good tool for fast and effortless prototyping. MATLAB also provides easy ways to visualise and plot data, which is very useful when working with image processing. The plots and figure windows are also easily configured with custom GUI-elements, adding interactivity in an easy way.

Toolboxes

MATLAB supports the use of toolboxes. A toolbox is a collection of functions/algorithms and applications that belong together because they do common things. There is a wide variety of toolboxes available, both official ones developed by Mathworks and custom ones developed by third parties. The toolboxes relevant for this thesis are:

Computer Vision System Toolbox This toolbox provides algorithms and functions that can be useful when working with video [14]. It provides functions to read and display videos, camera calibration, graphics drawing, motion detection and more.

Image Processing Toolbox A useful toolbox when working with images, which, among others, contain functions for image segmentation, noise reduction and image enhancements [15].

Signal Processing Toolbox This toolbox provides useful functions and apps when working with various signals [16]. It includes algorithms

for resampling and smoothing signals, filter design, finding peaks, measuring bandwidth and more.

2.2 Motion analysis

With higher processing capabilities and advances in motion analysis methodology, the interest in motion processing has increased in recent years [17]. Motion analysis are used in many applications, such as detection and tracking of human faces, autonomous vehicles, robot navigation, smart room tracking and to detect 3D shape from motion. Usually, motion analysis systems work on an image sequence. It can work on as little as two or three consecutive images, or it can use significantly longer image sequences. Motion analysis is the analysis of these images, to extract the information we are looking for. From a practical point of view, the problem can be divided into three groups:

Motion detection

This is considered the simples of the three, where the goal simply is to register any detected motion in the scene [17]. Motion detection is often used for security, for example using a camera to detect motion. A crude way of detecting motion could for example be to analyse the histogram of the images when we see a change larger than some threshold, we probably have some motion in the scene.

Moving object detection and location

These problems are considered to be much more difficult than simple motion detection [17]. Motion detection usually employs a static camera, where moving object detection can have a static camera and moving objects in the scene, or static objects in the scene and a moving camera. The most complex methods can even work when both the camera and the objects in the scene are moving. This group of problems can involve detecting a moving object, detecting the trajectory of its motion and predicting the future trajectory.

To solve these tasks, image object-matching techniques are often used, such as matching of object features or matching of specific object points (for example corners). These features or object points can then be tracked through the image sequence, thus detecting the motion of the object.

Moving object detection can be used in cloud tracking from satellite images, city traffic analysis and motion analysis for autonomous road vehicles.

Derivation of 3D object properties

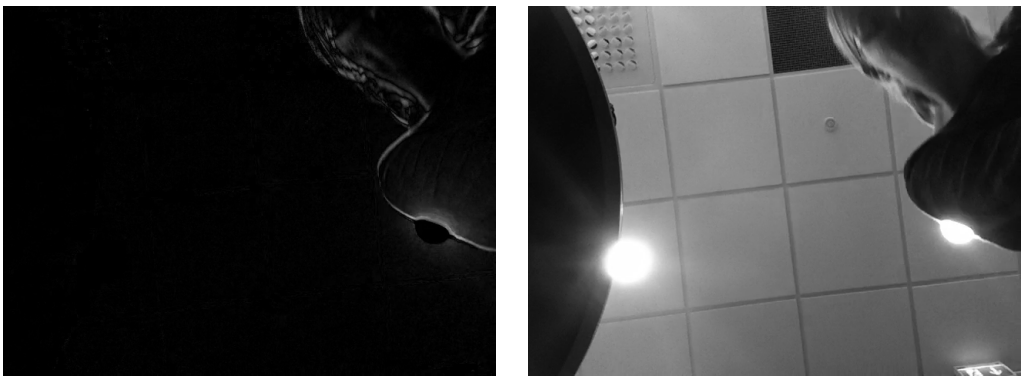
This group of problems involve extracting 3D object properties from a set of 2D projections attained from an image sequence, covering different instants of the object motion [17]. These problems can be very complex, and can for example involve extracting the 3D shape of an object in a scene.

2.2.1 Differential motion analysis

Subtraction of two consecutive images in a sequence (assuming static camera and unchanged illumination) can be the basis for motion detection [17]. Doing so will create a difference image, where pixel intensity at position (i, j) can be represented as $d(i, j)$, where i is horizontal index and j is vertical index of the pixel. Non-zero pixels represent motion, and zero-valued pixels represent no motion. If we have two consecutive images, f_1 and f_2, we get:

$$f(n) = \begin{cases} 0 & \text{if } |f_1(i,j) - f_2(i,j)| <= \varepsilon \\ 1 & \text{otherwise} \end{cases} \tag{2.1}$$

where ε is some threshold. Usually the threshold is a small positive number, used to avoid seeing noise as motion. The difference image can either be a binary image (which will be the result of equation 2.1) or we can use the absolute difference to indicate motion, thus saving not only which pixels that contain motion, but also how big the change is. This can be useful, as it preserves more motion information. Figure 2.1a shows an example of a difference image and figure 2.1b shows one of the two original images it was created from.



(a) Difference image. (b) Original image.

Figure 2.1: Difference image compared to one of the original images.

There are four reasons why a pixel could have changed value between two consecutive image frames, f_1 and f_2 (assuming unchanged illumination and stationary camera):

1. $f_1(i, j)$ is a pixel on a moving object and $f_2(i, j)$ is a pixel on a static background.

2. $f_1(i, j)$ is a pixel on one moving object and $f_2(i, j)$ is a pixel on a different moving object.

3. $f_1(i, j)$ is a pixel on a moving object and $f_2(i, j)$ is a pixel on the same moving object, but on a different part of the object.

4. Noise.

If there are a lot of noise in the difference image, it can lead to inaccurate motion data, but mostly the problem can be avoided (or at least minimized) by choosing a good threshold and by disregarding small (or single pixel) regions of change.

Figure 2.2 shows one of the problems with differential motion analysis; the aperture problem, caused by ambiguous motion. The black area represents an object boundary. Based on the difference image in a situation like that, it is impossible to detect the detailed direction in which any of the points on the object boundary are moving. The arrows represent some possible motion directions for one point at the boundary, where all may yield the same final

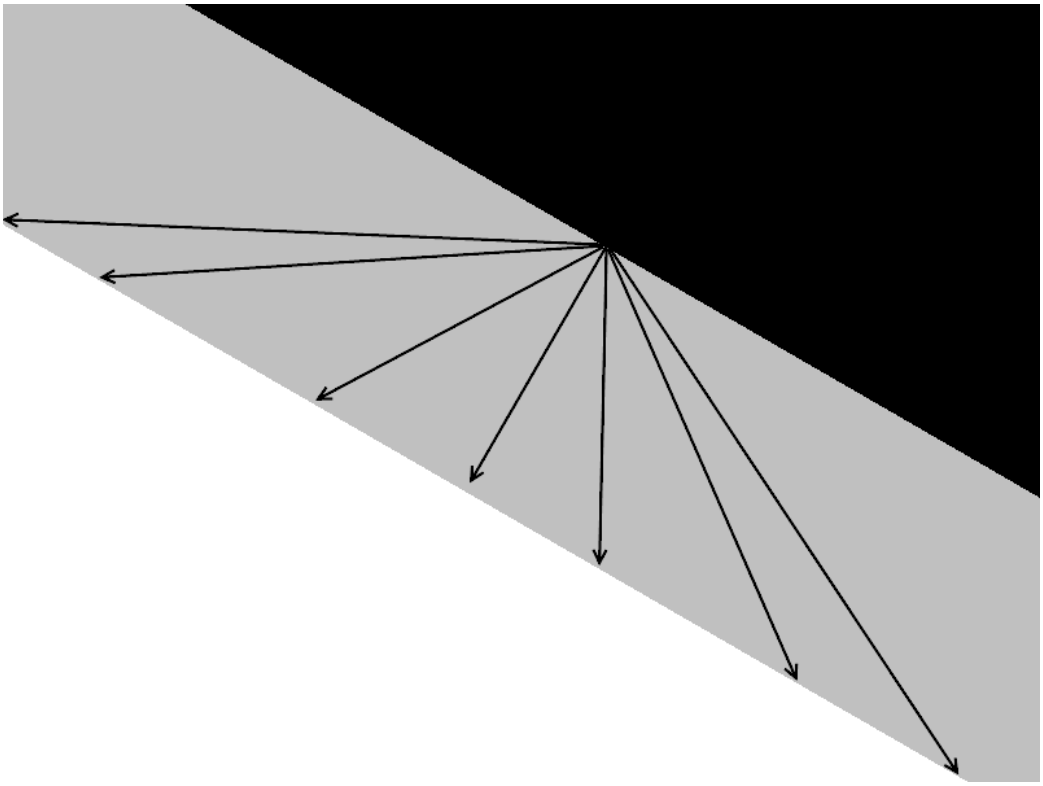


Figure 2.2: Differential motion analysis aperture problem example.

position of the object boundary in the image. This might be a critical point, depending on the goal of the motion analysis. Difference images can also be used to detect and visualize what has changed between two images, in which case this problem becomes insignificant.

2.3 Optical flow

Optical flow is the apparent motion in a scene, caused by the relative motion between the camera (an observer) and the scene [18]. With a static camera, it is a way to show the movement of objects between consecutive images. Given a sequence of images *I(x,y,t)*, estimating the optical flow means to find the velocity *(Vx, Vy)* of pixel *(x,y)* between image t and *t+1*. This can be done in many ways, for example by using block matching. Block matching works by matching a block of pixels in image t with an equally sized block

of pixels in image *t+1* by moving the block over a search region.

2.4 The YUV color space

YUV is a color space that uses one luminance component (Y) and two chrominance components (UV) to represent an image [19]. There are several variations of this format, defined by how the image is sampled. The format relevant for this thesis is YUV420, because this is the default format used by cameras in Android [20]. An example of how a four pixels by four pixels image would look like in this format is seen in figure 2.3. Here we see that each two-by-two pixels are represented by four luminance values and one of each of the two chrominance values. Representing this image as a byte array would look like figure 2.4; first all Y values, then all U values and finally, all V values.

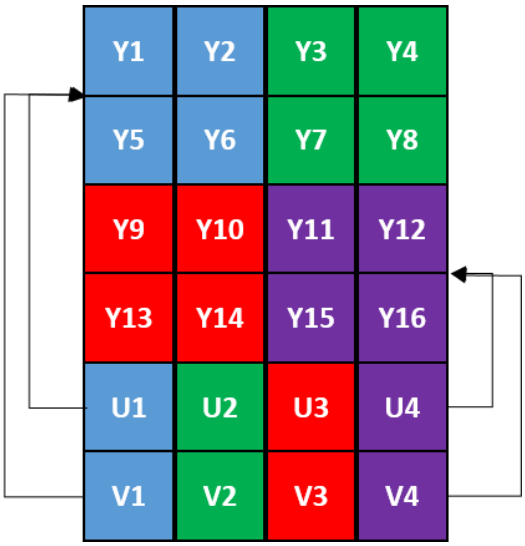


Figure 2.3: YUV image example.

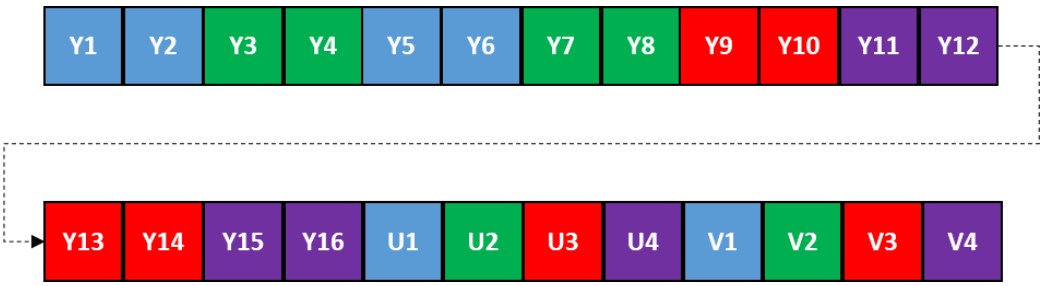


Figure 2.4: YUV image storage example.

2.5 Weighted moving average filter

A weighted moving average (WMA) filter is a type of finite impulse response filter (FIR) [21]. The filter applies a set of multiplying factors to give different weights to data in the sample window. The weights can have different distributions, to create different results. The filter can for example be used to smooth abrupt changes in a data series or to mask erroneous spikes/drops in the data. The formula for applying a WMA filter is seen in equation 2.2.

$$y(n) = \sum_{i=0}^{N-1} W_i \times x_{n-1} \tag{2.2}$$

where $x(n)$ and $y(n)$ is the n'th input and output sample in a signal respectively, and W_i is coefficient i in the weight vector W, where $\sum_{i=0}^{N-1} W_i = 1$. If the sum of the weight vector is not equal to one, equation 2.2 has to be divided by the sum of weights to be correct. N is the window length. Usually, the most recent input sample gets the largest weight.

2.6 Discrete Fourier transform

The Fourier transform was developed by the French mathematician Joseph Fourier [22]. He discovered that any piecewise smooth signal can be represented as a sum of sinusoids (or complex exponentials). These were called Fourier series.

Discrete Fourier transform (DFT) is used to transform a sequence of numbers from its original domain (for example time domain), into a frequency domain representation. The sequence of numbers usually represents the re-

sults from sampling a continuous signal $f(n)$ at fixed intervals T. Using N samples, DFT is defined as:

$$F(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n) e^{-2\pi i \frac{nk}{N}} \tag{2.3}$$

The integer values of k are called the frequency bin numbers. The frequency bins represents the discrete frequencies the signal can be fitted into. DFT can be used to extract the frequency components of a signal, and thereby to find the strongest frequency component in that signal. In figure 2.5 we can see a plot of a portion of a signal comprised of one 50 Hz sinusoid with an amplitude of 0.7 and a 120 Hz sinusoid with an amplitude of 1.

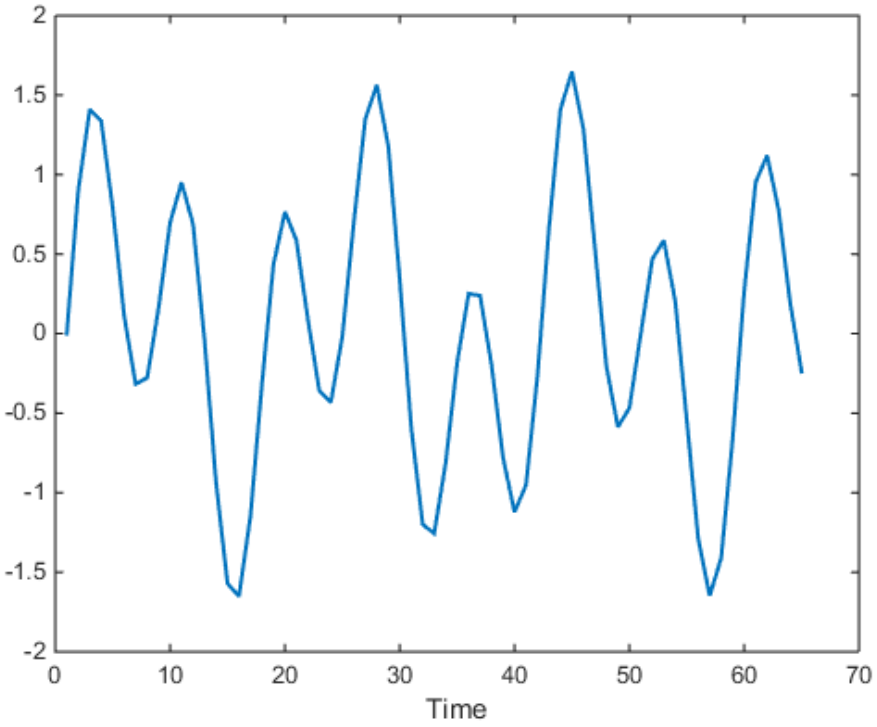


Figure 2.5: Sinusoid plot example.

In figure 2.6 we see the result from calculating the DFT on the same signal. We see clear peaks at 50 and 120 Hz, which is as expected. The peak

at 120 Hz is a bit higher than the one at 50, since that signal component has an amplitude of 1, compared to 0.7 for the 50 Hz sinusoid. If the components of the original signal were unknown, we see that it is possible to extract the frequency and relative amplitude information based on the results from the DFT.

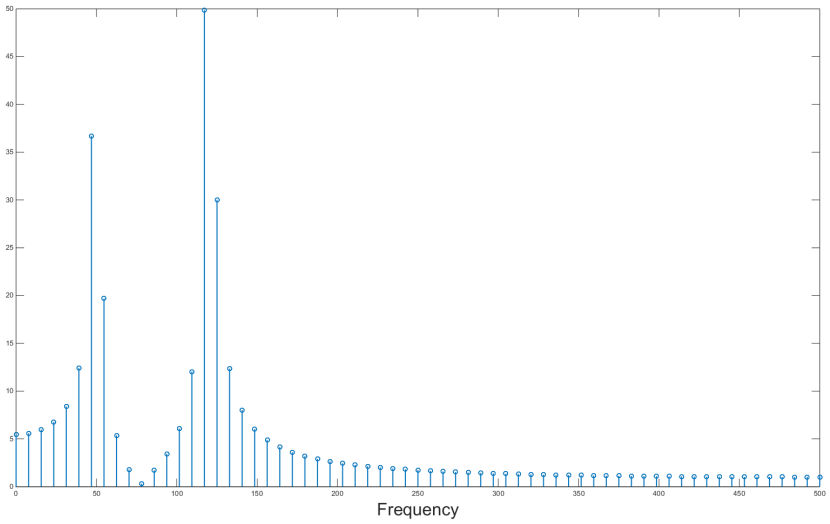


Figure 2.6: DFT output example.

2.6.1 Windowing

DFT operates under the assumption that its input is a periodic signal (a signal which repeats itself) [23]. However, this might not always be correct. If the signal is noisy, or if the length of the signal analysed differs from one period-length, the signal might not be seen as periodic. This effect is called leakage, and can cause errors in the result of DFT. To circumvent this potential problem, we can use windowing. Windowing means to apply a window function to the input before calculating DFT, to force the data to be periodic. There are several different windows that can be used, but they are mostly zero (or close to zero) in the start and end of the block, with some shape in in the middle.

A common window is the Hanning window, which is defined as:

$$w(n) = 0.5(1 - \cos(\frac{2\pi n}{N-1})) \tag{2.4}$$

, where N is the length of the window. The function creates a window which is zero in both ends, with a rounded lobe in-between, as seen in figure 2.7. This makes the input periodic, and emphasises the centre of the input.

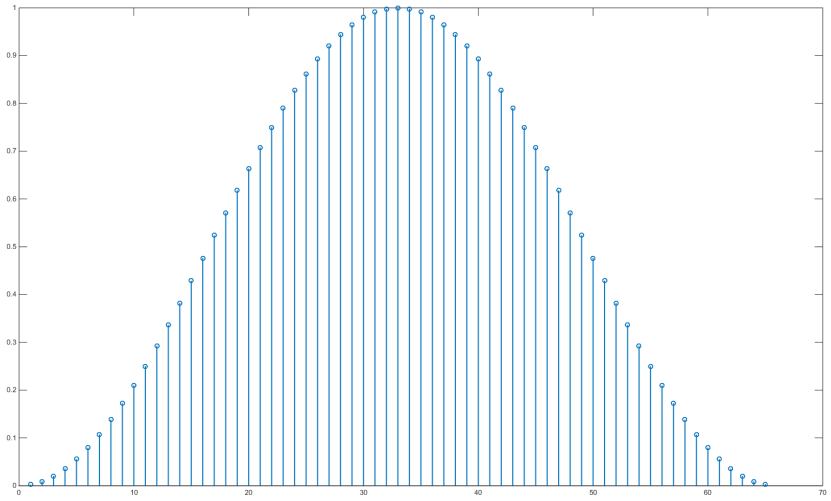


Figure 2.7: Hanning window.

2.6.2 Fast Fourier transform

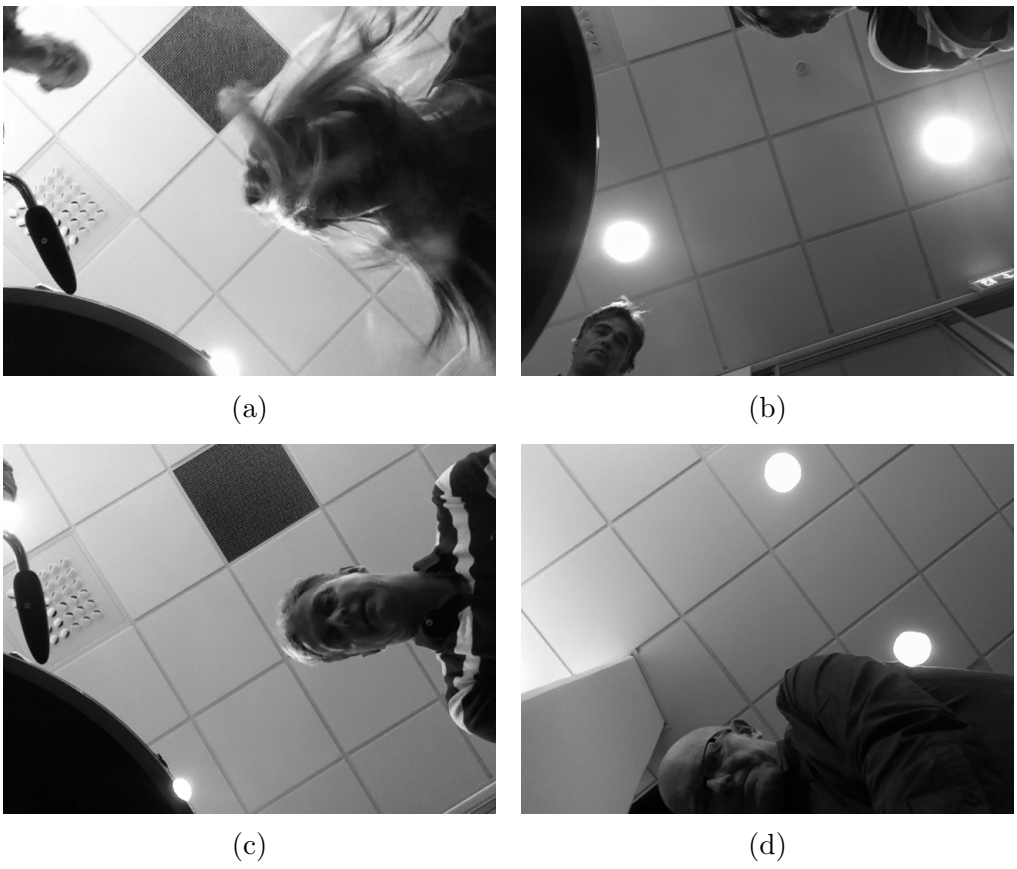
Direct calculation of the DFT using equation 2.3 has a computational complexity of $O(N^2)$ [22]. Fast Fourier transform is an algorithm used to obtain the DFT. It is a recursive algorithm that is based on factorization of matrix multiplications and the removal of redundant calculations. It significantly reduces the computational effort by reducing the computational complexity to $O(\mathrm{N}\ log_2(N))$, hence the name.

3 MATLAB prototypes

This chapter will cover different algorithms considered to extract the CPR frequency from the mobile video. The prototype algorithms are implemented in MATLAB.

3.1 Test videos

During the work with these prototypes, several test videos were used. These videos featured different people performing CPR on a training manikin. They simulated picking up the camera/phone to talk on it, stopping the compressions or changing their position in the frame. Some also included slow or abrupt change of compression rate. All the recorded videos were originally in color, but converted to grayscale when they were loaded into MATLAB. We will get back to the reason for this when we get to the Android implementation. Testing in MATLAB showed that the difference from analyzing grayscale video compared to color video are negligible, and in most cases the result will be exactly the same. Figure 5.4 shows a frame of four different test videos.



(a) (b)

(c) (d)

Figure 3.1: Screenshot from some of the test videos.

3.2 Sum of changes

This method exploits the nature of the motion involved in CPR. The motion has a natural acceleration and deceleration phase, in which each top and bottom (the motion direction turning points) of the motion will have less movements than the middle part (the acceleration phase, where the person performing CPR is moving upwards or downwards). This method uses differential motion analysis, described in section 2.2.1.

This method works by first calculating the difference image for all consecutive images in the video we are analyzing. In figure 3.2 we can see an example of a difference image from one of the test videos used. What we see here is a man doing CPR. Some contours in his face and on his clothes are

visible, indicating that he is moving. The background is completely black, which means that there are no movement there. In this particular frame there are virtually no noise. To be sure to eliminate most of the noise that might appear in the difference images, a threshold is used. This will remove the pixels which has only changed slightly (mostly noise) and still leave the pixels that has changed a lot intact. It might remove some of the motion information, but since there are still a lot of information left (due to relatively clear contours), this has minimal impact on the results.

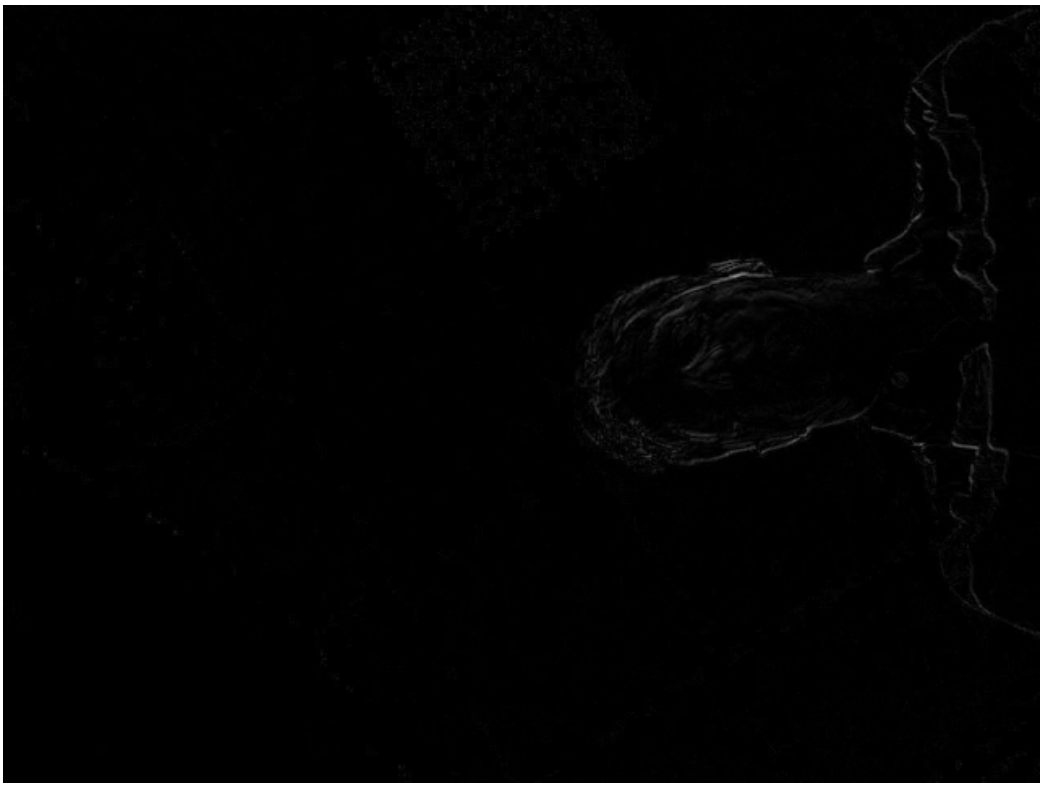


Figure 3.2: Difference image example.

By adding all pixel values in a difference image, we get an indicator on the level of movements between two consecutive images. Doing this for all difference images in a three second window of a video, results in a graph as seen in figure 3.3.

Here, we can see the acceleration and deceleration phase of the CPR motion. A rising value indicates acceleration and declining value indicates

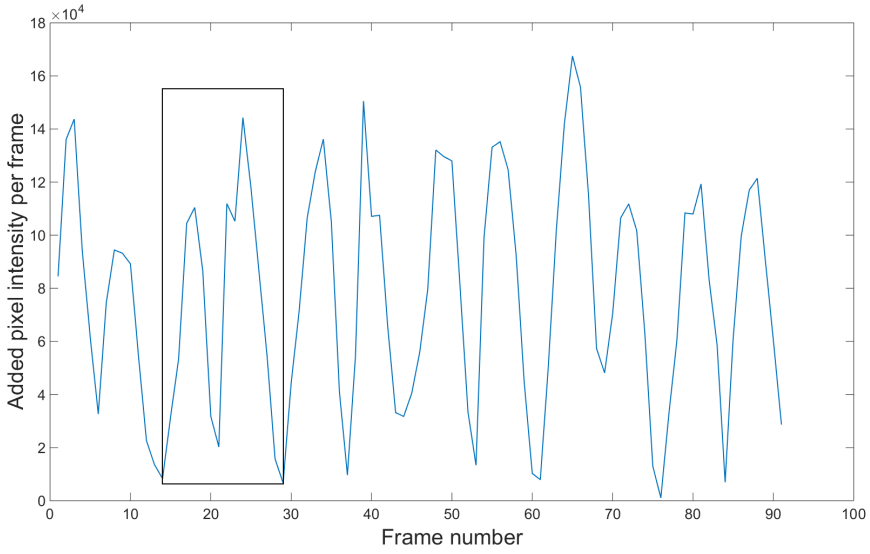


Figure 3.3: Sum of changes from a set of difference images.

deceleration. Each bottom represents a turning point in the compression (either the top or the bottom of the compression), and each top represents a period of maximum movement in the video. This means that one compression period corresponds to two periods in the figure (illustrated by the black square).

3.2.1 Estimating the frequency

Estimating the frequency from the data seen in figure 3.3 can be done in several ways. One way is to count the number of local maxima and minima. Since we need four turning points (maxima or minima) to represent one compression period, we divide the number of extrema by four. See equation 3.1. Using equation 3.2, we get the number of compressions per minute. Window size is the number of frames in the window we are examining.

$$\text{Number of compressions} = \frac{\text{Number of directional changes}}{4} \tag{3.1}$$

$$\text{Compressions per minute} = \text{Compressions} \times \frac{\text{Windows size}}{\text{Frame rate in video}} \times 60 \tag{3.2}$$

Another method of estimating the frequency is to first calculate the average of all the values, and then count how many average-crossings there are. This has an advantage over the first one, in which that finding average-crossings are easier than counting tops and bottoms. The potential problem with counting tops and bottoms arises when the data is jagged; it can be hard to detect all the tops/bottoms, without including false positives.

The third method of estimating frequency is to use FFT to calculate the dominant frequency in the data series (sum of changes). One advantage of using FFT is that is responds faster to changing frequencies, meaning that the algorithm will faster report the correct frequency if the compression rate rises or declines (compared to counting tops/bottoms or average-crossings).

3.3 Sum of changes using region of interest

This method is the same as the one in section 3.2, except that it does not calculate the sum of changes over the entire image. Instead we try to find a region of interest (ROI), and only estimate the frequency inside that region. The reason for doing this can be seen in figure 3.4. Here, we see the same person as seen in figure 3.2, but he does not represent the only movement in the frame. What we see to the left is a woman waving her hand in front of the camera. This might not be a realistic scenario, but it illustrates an aspect which might be a problem when using the entire image; the movement of the woman can interfere with the compressions that the man is performing, resulting in inaccurate or wrong frequency estimates.

One solution to this potential problem is to use an ROI strategy. To do so, we first divide the frames into blocks (at for example 50×50 pixels) and then establish an ROI consisting of a group of these blocks. If we had a situation like the one in figure 3.4 in the beginning of the video, we would need to find which region is the proper ROI. This can be done by isolating the two areas and checking which is the biggest (and therefore the most probable to be the region we actually want) and has the most probable motion frequency (meaning a realistic CPR rate). When the ROI is established, we only let it expand or contract slowly, in order to not include other nearby interfering movements.

In theory, using a threshold can remove some of the information in difference images. During testing, we tried estimating the frequency using all the original information (the information that was there before we applied the

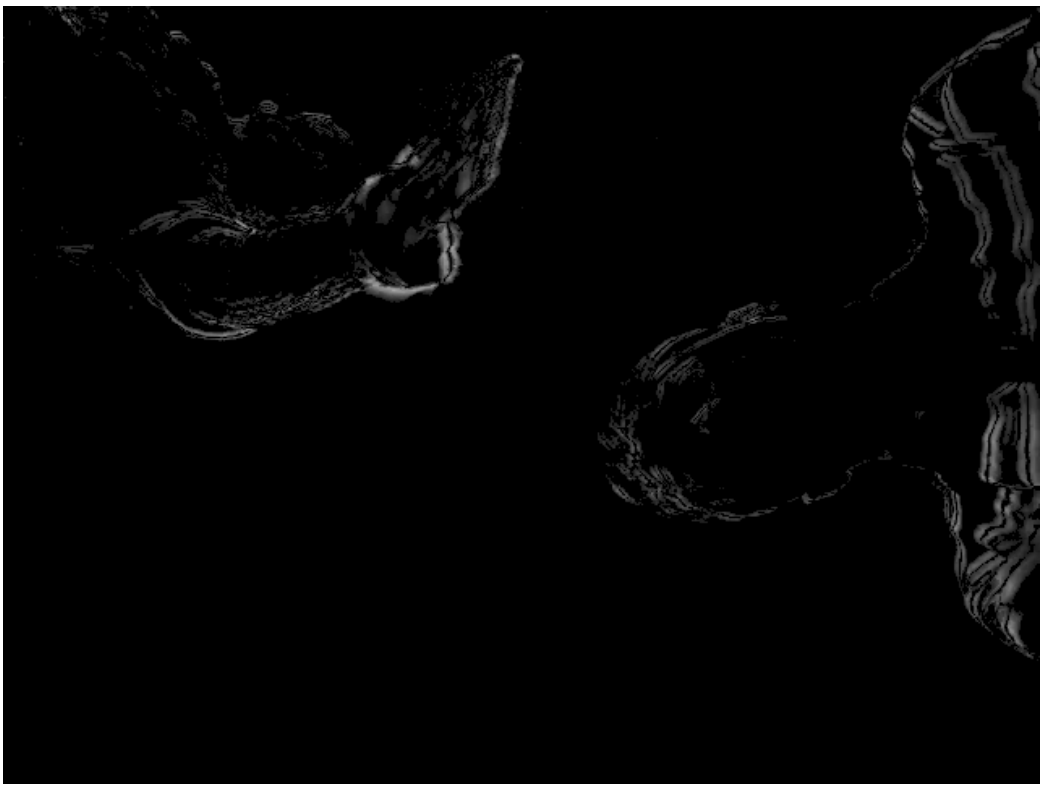


Figure 3.4: Difference image with two moving persons involved.

threshold) and the information that was left after the threshold was applied. The results were the same.

3.3.1 Alternative region of interest

When working with the difference images split into blocks, several ways of including blocks into the ROI were investigated:

Big changes One option was to calculate the sum of changes for each block and only use the blocks which had a very high sum of changes, for example higher than the average, or higher than some other large threshold.

Reasonable frequencies Another option was to estimate the frequency inside each block, and only use the blocks which had a reasonable frequency. This would rule out the blocks which had an unrealistically low or high frequency.

Signal strength When estimating the frequency inside each block using FFT, we can calculate how strong the signal of the detected frequency is compared to the rest of the signal (the noise in this case). In figure 3.5 an example of the output from FFT is shown. To calculate the signal strength, we first square all values, to accentuate the peaks. Then we divide the value represented by the highest peak (the strongest frequency component) with the sum of the rest to get an indication of how strong that frequency is, compared to the rest. We then choose to only include the blocks with a high enough signal strength in the ROI.

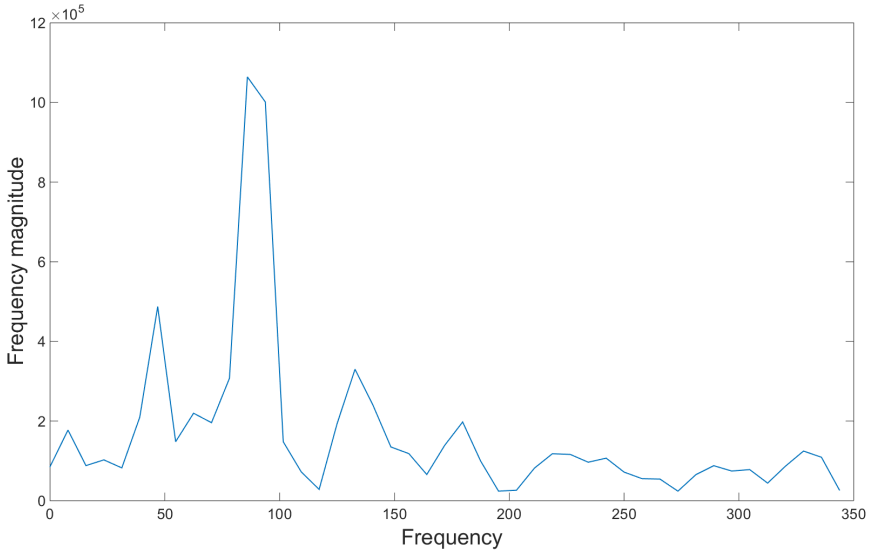


Figure 3.5: Example of an output from FFT.

A combination Naturally, each possible combination of the above were also tested.

3.4 Optical flow

Optical flow can be used to calculate motion vectors between consecutive frames in a video. In theory those motion vectors could be analyzed to find whenever the general motion in the video has changed direction. A change in direction would mean either a top or a bottom in the compression motion,

and that could therefore be used to calculate the compression rate. In figure 3.6 we can see a frame from a video, with the motion vectors overlaid.

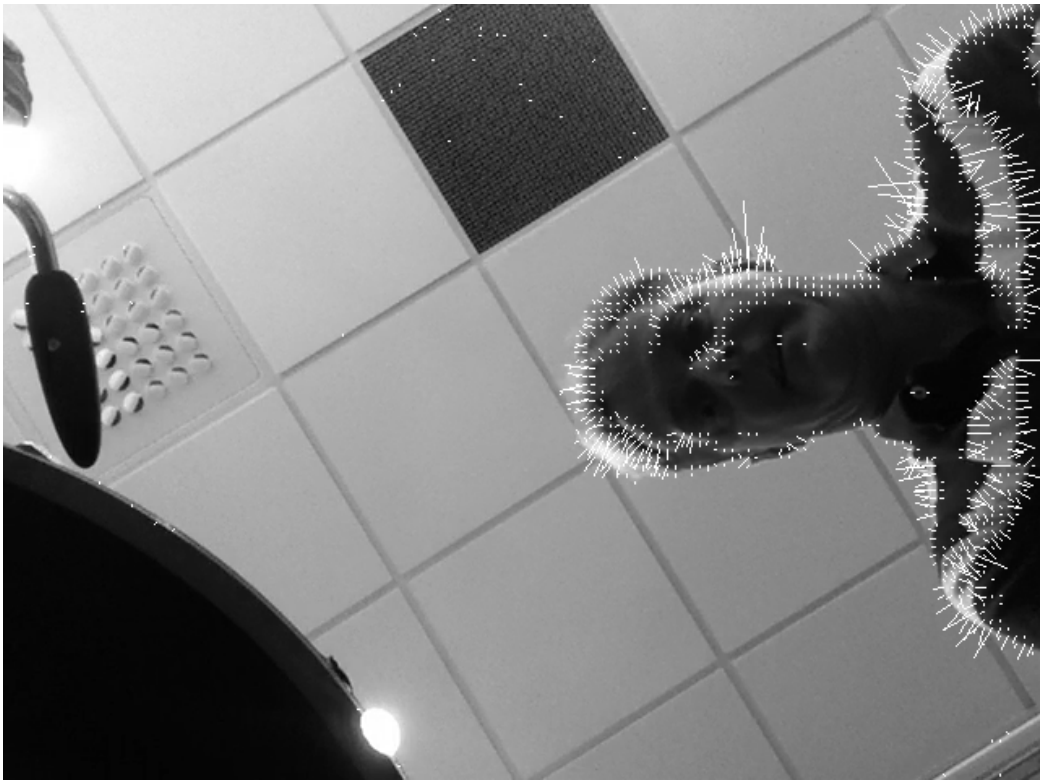


Figure 3.6: Optical flow motion vectors.

Since the motion vectors have a direction, we could determine that we have a change in motion direction when enough of the individual vectors have changed direction. In the image we can also see that optical flow is prone to noise in the frame, for instance some noise on the table and on the lamp to the left has been falsely detected as motion. Those vectors are very short, so we choose just to ignore vectors below a certain length.

3.5 Detecting vertical and horizontal directional changes

This method also uses differential motion analysis, but in another way than the sum of changes-methods. It has some similarities to optical flow, in that it also tries to detect changes in motion direction. Here, as well, we interpret a change in motion direction as a top or bottom in a compression.

This method works by analyzing each difference image. It scans all rows and columns inside the region of interest (ROI) to find the first edge in each respective row/column, as illustrated in figure 3.7. ROI is marked with red. We use ROI (similar to section 3.3) to avoid searching the entire image frame, as well as to avoid analyzing wrong movement such as noise and other people in the frame.

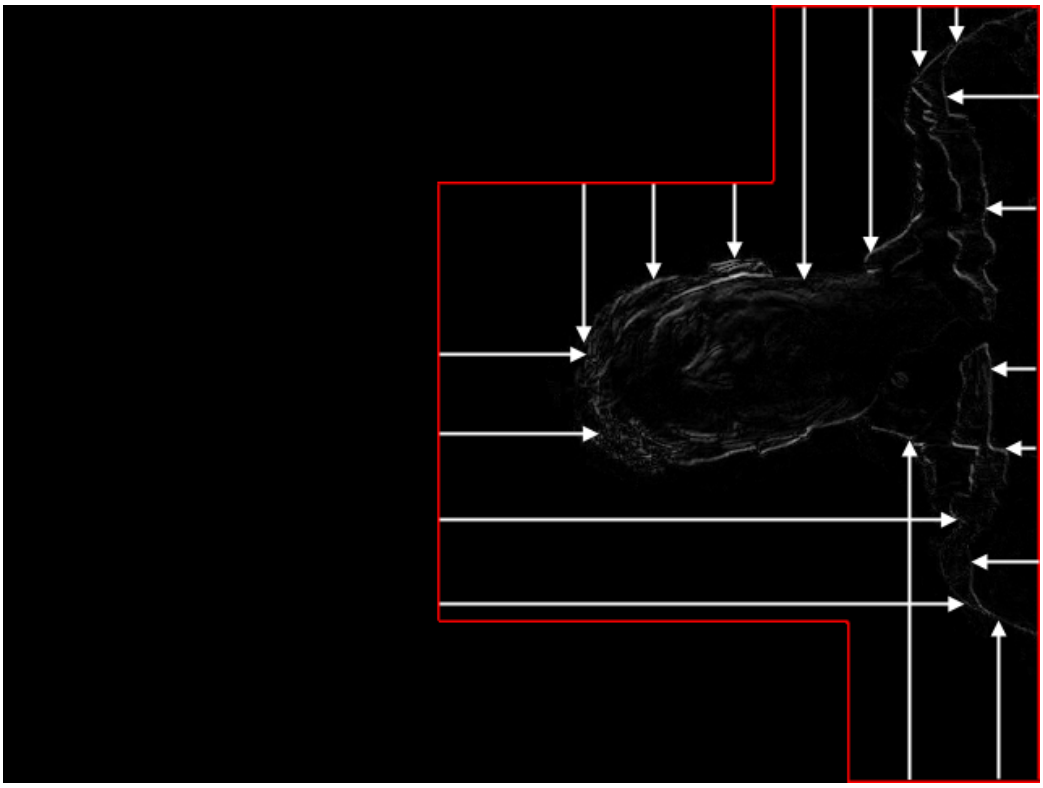


Figure 3.7: Illustration of edge searching in difference image, searching from all four sides.

For each row and column, we record the edge location. When there is a new frame available, it is determined whether the edge has continued in its previous motion direction or if the motion has changed direction. To find the edge points (an edge point is a pixel along an edge) in the next frame, two methods have been tested:

Entire region of interest Scanning the entire ROI for edge pixels for all frames. Similar to what is illustrated in figure 3.7.

Based on previous edges In each row and column we start at the previous edge point position and start the search from there. Here, we also use a threshold; if no edge points are found within a certain length from the previous point, we assume there are no edges in that row or column. When searching a row or column with no previous edges, we search from the edge of the region of interest without using a threshold on how long the search can go (as long as it is inside the region of interest).

After the edges have been found, there are several ways to detect when the general motion direction in the video has changed:

Left and upper edges Examining only the left and upper edges of the motion, we tried to determine separately when each edge had changed direction. When both had changed, we marked it as a change in motion direction.

Left, upper, right and bottom Same as above, just with all edges being examined. Here, only three (out of four) edges needs to change direction before the frame is marked as a change in motion direction.

Individual edge points When more than half of all the edge points found have changed direction (looking at both left and upper edges and all four edge sides), we determine that the motion direction has changed.

When all the direction changes has been found in the video (or in the part of the video we are analyzing), we can easily find the frequency using the same formula as in section 3.2.1, except that the number of compressions is found by:

$$\text{Number of compressions} = \frac{\text{Number of directional changes}}{2} \tag{3.3}$$

An alternative to this is to find the last three motion direction changing points, and use the time between the first and last of them (which represents an entire compression motion) to calculate the frequency. This is done using formula 3.4, and the unit of measurement is compressions per minute (cpm).

$$Frequency = \frac{\text{Frames per second}}{\text{Frames between direction changing points}} * 60 \tag{3.4}$$

3.6 Using contours

This method also utilizes the difference images. It works by searching each row and column of the difference image for edges. Here, an edge is defined as a pixel in the image with a value over a given threshold. Figure 3.8 shows an image of these edges in one of the frames from a video. The results is the contours of a man doing CPR.

Every time it finds an edge, it skips some pixels in the row/column it is searching, to avoid thick lines as contours. After the contour lines have been found, we calculate the distance to the center of the image for each contour point. The sum of these distances is then analyzed in the same way as the sum of changes-methods to extract the compression rate.

As we can see, this method share some similarity with the previous method, but it uses the contours/edge points in a different way when estimating the frequency. This method also finds more of the contour, whereas the previous method stops its search along a row/column when it reaches an edge.

Figure 3.8: Example of contours in video frame.

3.7 Prototype summary

Testing the prototypes (section 5.2) makes it clear that the sum of changes method using region of interest performs best. The measured frequency are closer to the real frequency and the maximum error are less than the other prototypes. Regarding the ROI, we decide that an approach similar to what is described in section 3.3 are better than the alternatives described in section 3.3.1. The idea behind all the alternative methods are to use only some of the blocks inside the ROI, based on different criteria. Initial testing showed that the sum of changes method needs all the motion information to function properly, which we get from the original ROI strategy (only use ROI to avoid interfering movements, not divide the ROI any further). A simple overview of the chosen algorithm is seen in the flowchart in figure 3.9.

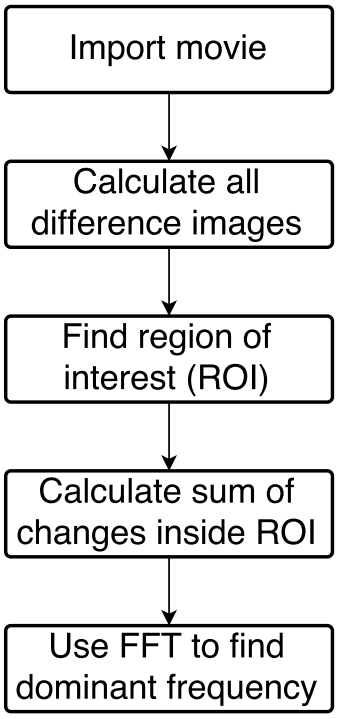


Figure 3.9: Chosen prototype algorithm flowchart.

4 Android implementation

This chapter gives details on the implementation of the chosen algorithm as an Android application. The first section is an overview of the algorithm along with a flowchart, the next is an overview of the Android application. The last section contains all the implementation details.

4.1 Algorithm overview

A flowchart describing the algorithm can be seen in figure 4.1. Every time a new frame from the camera is available, the difference image between the new frame and the previous frame is calculated. When 15 difference images (every 0.5 seconds with 30 fps) has been calculated, an asynchronous task is started to calculate the frequency. This task uses the 15 new difference images provided, as well as up to 75 of the previously used (making it a total of up to 90 images, equal to 3 seconds with 30 fps). First, the task tries to establish an ROI. If it is successful, we calculate the sum of changes inside the ROI, using up to 90 of the previous images. We do the same using the entire frames if the ROI is not established yet. Sum of changes here means summing all pixel values in the difference images. In the end, we use FFT to find the dominant frequency in the window we are examining. A weighted

moving average filter is applied to the frequency before it is displayed.

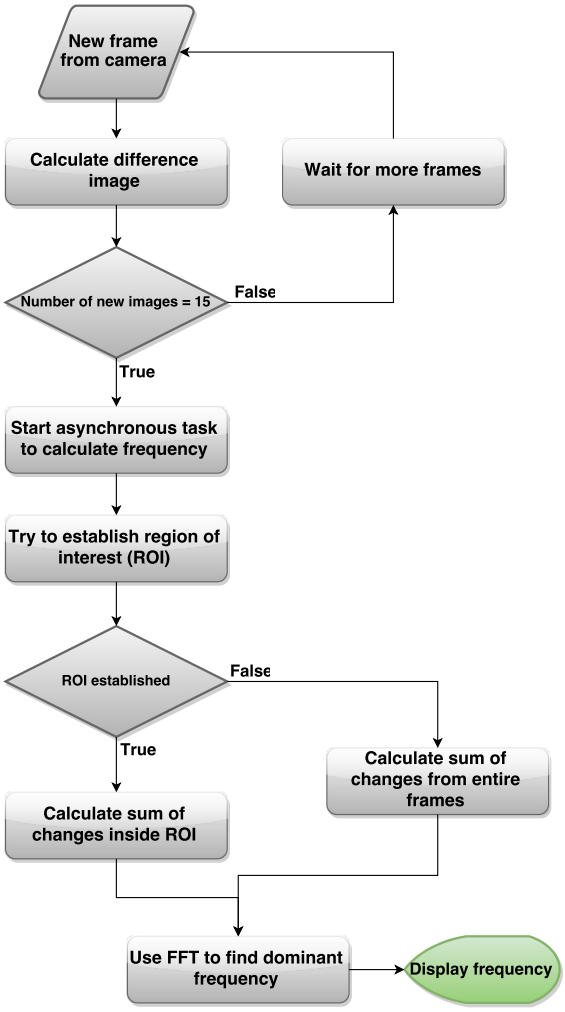


Figure 4.1: Algorithm flowchart.

4.2 Application overview

The application is developed for Android and it has been tested on a LG Nexus 5. It has been tested using Android version 5.1, but it is built to support Android 4.1 (SDK version 16) and newer. The user interface is rather simple, given the focus on the algorithm implementation (figure 4.2). There are three buttons, all starting the frequency estimation. The first button simply starts the algorithm and shows the frequency output. The middle one also records the video and save the frequency output to the internal memory on the phone, while the last one just saves the frequency output. When the algorithm is running, the frequency is shown at the top (see figure 4.3), the view from the camera is seen to the left, and a visual representation of the ROI is seen to the right. More information on development of Android applications can be found at [24].

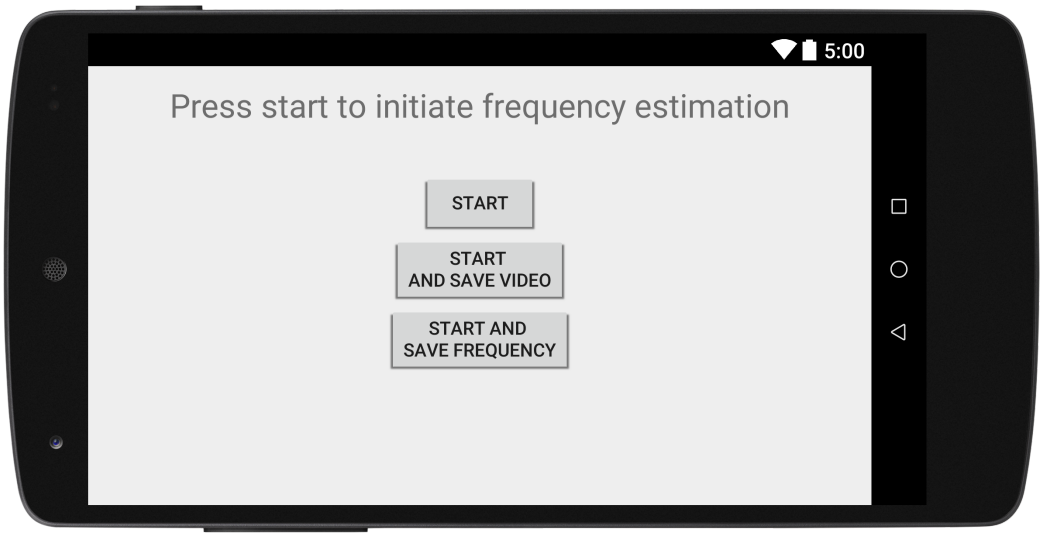


Figure 4.2: Android application GUI 1.

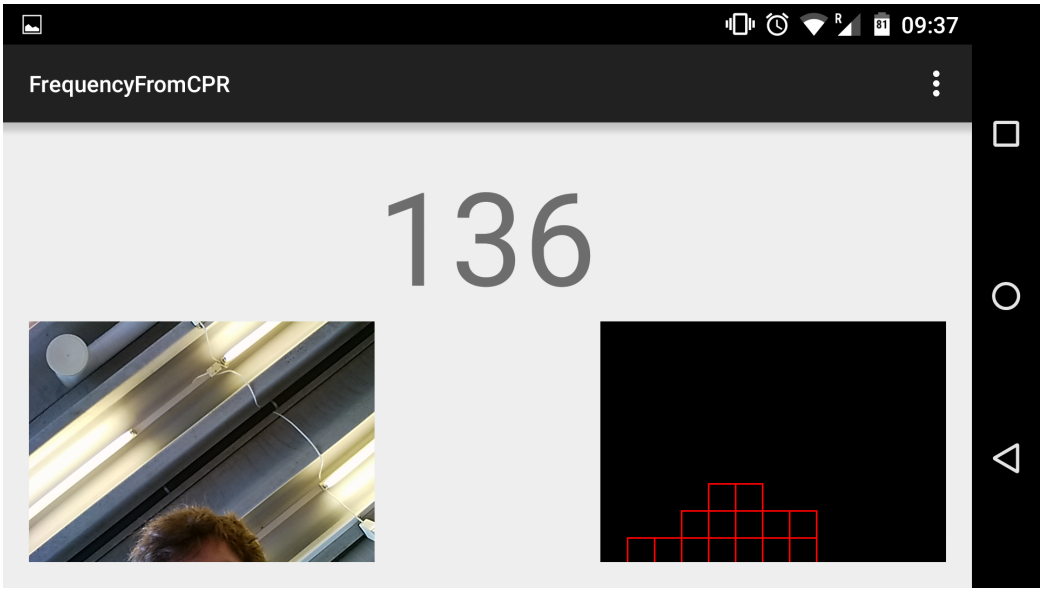


Figure 4.3: Android application GUI 2.

Below is a brief description of the classes in the application:

MainActivity This is the main class of the application, the one that starts everything. It handles all the button presses, and initiates the frequency estimation accordingly. It also handles the application pausing, by overriding the onPause method. This method is used to stop the camera and to flush and close any file writers that might be used to save video or frequency output. More information on the Activity class can be found at [25].

CalculateFrequencyAsync This is the asynchronous task which is used to calculate the frequency. It extends AsyncTask, which makes it possible to have it running in the background, thus keeping the user interface responsive. More information on AsyncTask can be found at the developer guide [26].

CameraCallback This class handles the camera callback, via the onPreviewFrame method. Every time a new frame is available from the camera, this method is called. There is also a method to calculate the difference image here.

CameraHelper The camera helper class it used to get a camera instance, and to setup the camera parameters. This includes setting the fps and image size closest to the desired ones.

CameraPreviewSurface The preview surface is used to display the camera feed in the user interface.

DrawableSurface A drawable surface is used to display the current ROI in the user interface.

FileHandler The file handler handles the saving of files to the internal phone memory. Used to save the frequency output as well as the video.

FrequencyFinder This class is used to find the frequency from an array of numbers (sum of changes). It uses both FFT and count of maxima/minima and average-crossings.

Imageblock An image block represents a block in an image. A block does not contain the actual pixel values. Instead, it has a set of ranges that represent a block in an image. More on this is described in section 4.3.9.

ImageblockCollection This is a collection of image blocks. The image blocks are stored in a two dimensional array.

RegionOfInterest The region of interest is stored as a two dimensional integer array, with methods to establish and maintain the ROI.

RegionOfInterestHelper A helper class, used to make sure there is only one ROI area.

Utils A collection of utility methods, such as finding maximum and average in an array, and creating a Hanning window.

WeightedMovingAverageFilter Used to filter the frequency output before displaying it in the user interface, to smoothen the output some. When saving the frequency using the file handler, both the filtered and unfiltered frequency is saved.

4.3 Implementation details

This section gives more detailed information on the implementation of the algorithm and Android application.

4.3.1 Setting up the camera

To ensure compatibility with older versions of Android, we have chosen not to use the newest camera APIs available (the newest are supported by Android version 5 and newer). The original APIs have all the functionality we need and works well for the work done here. More information on the camera APIs can be found here [27][28].

Setup of the camera involves setting the resolution and number of frames per second (fps). This is done by getting a list of supported resolutions and framerates from the camera and selecting the closest one to the one we want. We have chosen a resolution of 640 x 480 pixels and a framerate of 30 fps. These should be supported by most smartphones available, and the test videos used for the prototype testing had this resolution and framerate.

4.3.2 Getting the camera frames

The camera frames are available via the camera preview. The preview is the live camera preview usually shown on the screen. By attaching a preview callback to the camera instance, we get a callback every time a new preview frame is available. The frame is delivered as a byte array. That array is the data for one frame, in YUV format. Since we do not need the color information in the image, we only use the first 307200 bytes in the array (equals image width $\times$ image height, using 640×480 resolution). We convert these bytes into integers (luminance values), by removing all but the least significant byte.

4.3.3 Calculating difference image

Calculating a difference image is a done as described in section 2.2.1. Here it is done using a for loop and by always comparing against the threshold value.

4.3.4 Asynchronous task

Every time 15 new difference images has been calculated (every 0.5 seconds when using the desired 30 FPS), an asynchronous task is started. This task runs in the background, so that it does not keep the user interface from updating (which would be perceived as lag) and prevents frame drops. This task is where the biggest workload of the algorithm happens. The task starts by calculating the sum of changes sum inside each image block, before moving on to the ROI.

4.3.5 Region of interest

The ROI is calculated every time the asynchronous task is run. Before the ROI is established, the entire frames are used for the next step. When the ROI is found, only the image blocks contained inside the ROI are used. Finding the ROI can be divided into two parts:

- Before the ROI is established, we have to establish it.
- When the ROI is formed, we have to update it.

Creating the ROI takes two seconds (four asynchronous task is run every 0.5 seconds). In the three first rounds, all image blocks are checked to see which ones have a sum of changes (the sum of all changes inside a block for the last 90 frames) above average. These blocks are marked, and in the fourth round all blocks that have been marked at least three times are considered to be inside the ROI. When this is done there might be some gaps in the ROI (there might be blocks that only had movements the last round, i.e.), so we fill the gaps by:

- Checking all rows and columns for blocks marked with 1 (1 equals inside ROI, 0 not).
- When we find a blocked marked as 1, we also find the last block marked as 1 in that row/column.
- Mark all blocks in between the two as being inside the ROI.

Filling the gaps like this might include some blocks that should not be marked (blocks without any movement), so we remove all blocks with zero movement in the last round. The last step in establishing the ROI is to make sure there are only one ROI area by:

1. Selecting a random block from the ROI and add it to a stack.
2. Select the first element in the stack.
3. Add the neighbors of this block which is also inside ROI to the stack.
4. Mark the block as being part of ROI area number one.
5. Repeat from step 2 until the stack is empty.
6. Repeat from step 1 until all blocks inside the ROI have been checked (now marking the blocks as being inside ROI area number two and so on).

Next, the algorithm simply define the largest ROI area as being the current ROI. When updating the ROI, the algorithm first checks how many blocks are inside the current ROI. If there is none, the ROI is reset and established again, from scratch. Otherwise, the existing one is updated by:

- Searching each row and column.
- When a block is found that has a different mark than the next block (in the row/column we are searching in), the algorithm has found an edge of the ROI.
- Then it is checked whether the block already included in the ROI should be excluded (if there are no movement there) or if the block not in the ROI should be included (if there are any movement there). The blocks are marked accordingly.
- If a block is marked as "should be included" three times, the block is included in the ROI. It is possible for a block to be marked as "should be included" twice in one round, if the block is in a corner.
- If a block is marked as "should be excluded" two times, it is removed from the ROI.

The last two points are done to make the updating of the ROI a bit slower, in an effort to reduce the chance that movements from other persons would interfere. The last part of updating the ROI is to make sure there are still only one ROI area (which is done as previously described).

4.3.6 Calculating the sum of changes

If the ROI is not established, the sum of changes is calculated using the entire frames. The algorithm do a for loop over all image blocks in the image block collection and calculates the total sum of changes for all frames. The approach if the ROI is established is the same, except now the algorithm checks whether a block is included in the ROI before including it in the sum of changes. The result from this calculation is an integer array, which is analyzed to find the frequency.

4.3.7 Finding frequency using FFT

The algorithm responsible for analyzing the integer array found above, can be summarized in this bullet list:

- First, the mean of the input is subtracted from all values in the input (sum of changes array), to remove some noise.
- The input is filtered with a Hanning window, to force the input to be periodic.
- The discrete Fourier transform is calculated using FFT. This is done via JTransforms, an open source FFT library [29].
- Convert the complex output from FFT into frequency magnitudes by calculating the absolute values.
- Find index of maximum frequency magnitude (dominant frequency) and convert to frequency (cpm) by multiplying the index with a factor found by the following equation:

  $$\frac{\text{frames per second}}{\text{Number of samples in sum of changes}} \times \text{frames per second}$$

- Count the number of local maxima/minima and average-crossings in the original input. Use maximum of these two to find an alternative frequency value by first dividing by four (to get number of compressions) and then applying formula 3.2.
- Then inspect the frequency magnitudes to see if one of the non-maximum peaks represent a frequency closer to the alternative frequency than the

frequency first found (directly from FFT). If there is one, the associated frequency is reported as the measured frequency. Only using FFT would often cause the measured frequency to be much lower than correct, due to low frequency noise in the input signal. Combining these two methods seems to produce good results.

4.3.8 Output filter

The final step in the algorithm is to apply a weighted moving average filter to the output. This filter use the last six frequencies found to create a weighted average. The weights used are $\frac{6}{21}$,$\frac{5}{21}$,$\frac{4}{21}$,$\frac{3}{21}$,$\frac{2}{21}$ and $\frac{1}{21}$. Or 6, 5, 4, 3, 2 and 1, divided by the sum of weights, as explained in section 2.5. This is done to smooth the output, to improve the user experience.

4.3.9 Additional implementation details

- To represent an image block, only its indices are stored (for example 0-49, 640-689 and so on). These range of indices are calculated for each block when the image block collection is instantiated. When we want to calculate the sum of changes inside one block, we simply pass in the entire image and use the ranges to find the sum inside that particular block.

- When we first tried to save the camera video, we used the MediaRecorder class (an Android class used to save video and still images). However, this stopped the camera preview callback, which in turn made it impossible for us to use. It appears that the MediaRecorder and camera preview are not meant to be combined. The simple solution to save the video was to use a buffered output stream and save all bytes received from the preview callback, and interpret the bytes in MATLAB later.

5 Results and analysis

This chapter covers the test results for the algorithms developed in MATLAB, as well as the Android implementation. All results are discussed and the reason behind all choices will be revealed. First, there is a section on video annotations, and then the results and analysis for the MATLAB prototypes is presented. The last section covers the results and accompanying analysis of the Android implementation.

5.1 Video annotation for benchmarking

When analyzing the videos using the different prototypes, as well as videos saved using the Android application, it is important to know what the real observable frequency is. To find that frequency we use annotations, in which we step through the videos (frame by frame) and mark the frames which represent a top or bottom in a compression movement, as well as the frames which indicate that the compressions has started or ended (this could represent a pause in the compressions). The results from these annotations are then used to find the real frequency in the video.

Although the initial idea was to compare the results from the tests for both the MATLAB prototypes and the Android implementation using these

video annotations, all the final test results for the Android implementation were created without it (more on this in section 5.3). However, some initial tests on the Android application were also compared to the correct frequency using annotations, to confirm that the implementation worked as planned.

The MATLAB prototypes were tested on several videos, but the presented results will focus on two of them (video duration measured in seconds):

Name	Description	Duration
A	Man with no hair performing CPR with steady rate. One pause.	20
B	Woman with long, loose hair performing CPR. Variations in rate. Includes minor changes in position and two pauses. Man walking around in background during parts of video.	57

Table 5.1: Test videos used to test MATLAB prototypes.

For more information on the test videos, see Appendix C.

Mean error and standard deviation

Most of the results consist of graphs comparing measured frequency and reference frequency (found either from video annotations or data from test mannequin). The mean error is the average difference between the two signals, corrected for all lag. It is found using the following formula:

$$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i.$$

The standard deviation (SD) shows how much variation there is in the signal, compared to the mean:

$$SD = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \overline{x})^2}.$$

5.2 Prototypes in MATLAB

This section covers the results found using the MATLAB prototypes. There are no separate results for the sum of changes (without ROI) method, only for the sum of changes using region of interest method. The preliminary testing showed the need for a ROI strategy, as explained in section 3.3. With no movements in the frame, except the person doing CPR, the results will be equal for both methods.

5.2.1 Sum of changes using region of interest

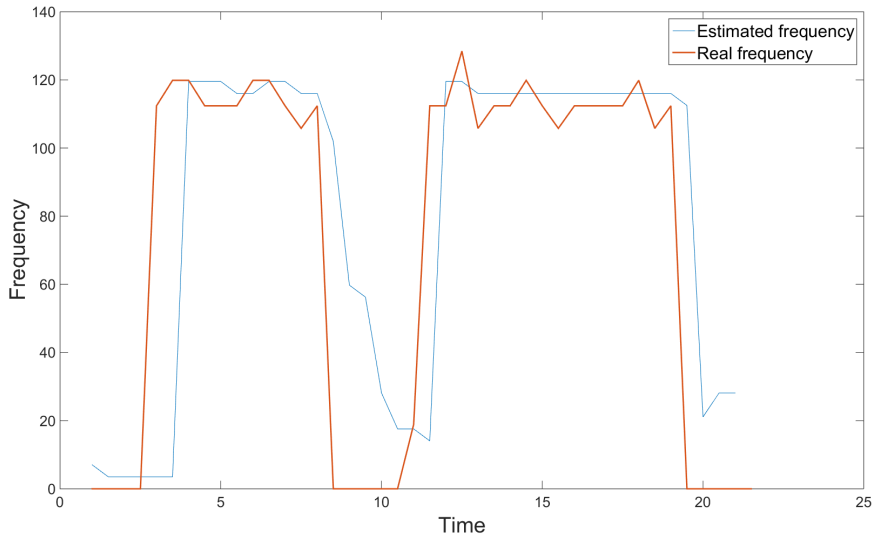


Figure 5.1: Estimated frequency versus real frequency using sum of changes with ROI for video A.

Figure 5.1 shows very promising results for this method. There is a very small delay (0.5 – 1 second) between the real frequency and what the algorithm measures. This behavior is expected as there will always be some kind of delay when working with a sliding window like we do here. In use, this delay is not critical. In order to measure the difference between benchmark and estimated frequency, a correction for this delay is made in figure 5.2. The average difference between these two signals (corrected for all lag) is 6.6 (SD equals 6.7).

The results from test video B (figure 5.3) are different. The average difference between the benchmark and the true frequency is 24.95 (SD 20.28). The first 20 seconds looks promising, except that it takes some time to find the proper frequency. The large peak in the center is a bit too low, and the last part is jagged and faulty. Further inspection of the video and results reveal a possible reason for this. The person performing CPR has long and loose hair. In the beginning of each CPR period (either at the start of the video or after a break) or after a change in frequency, the hair moves very inconsistent. After a few seconds the movement of the hair change to a more

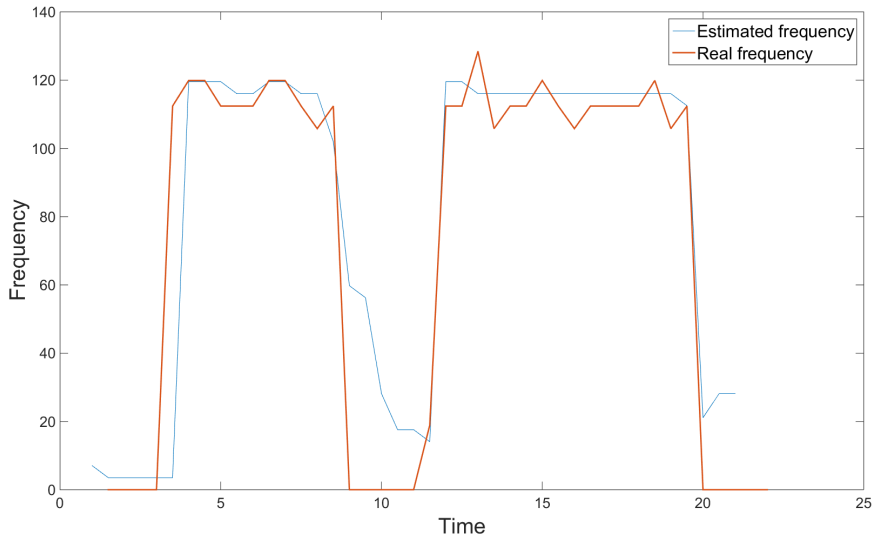


Figure 5.2: Estimated frequency aligned with real frequency, video A

steady rhythmic motion. When the hair moves very inconsistent, some of the movements of the hair seem to interfere with the movements of the head and shoulders. They somehow cancel each other out, resulting in a sum of changes plot as seen in figure 5.4a (taken from a 3 second window in the beginning of the video, when the frequency is reported to be 60 compressions per minute (cpm), but actually is close to 120 cpm). The graph appears to be correct, with distinct peaks. The problem lays between the distinct peaks, in the barely visible and indistinct peaks (around frame 20, 51 and 82). The FFT plot for this graph is seen in figure 5.4b. There we see a peak at 60 cpm, as expected.

Examining the plot for an equal window a few seconds later in the video, reveals quite a different plot (figure 5.4c). By now the hair has had time to settle in to a more rhythmic movement pattern. Here, the peaks that previously were indistinct now are very clear and visible. This allows the algorithm to find the correct frequency. The FFT plot seen in figure 5.4d shows a distinct peak at 120 cpm. There is also a peak at 60 cpm, but lower than the peak representing 120 cpm. The problem displayed here shows that this algorithm might struggle if the person performing CPR has long and loose hair, but it should find the correct frequency eventually. Combining

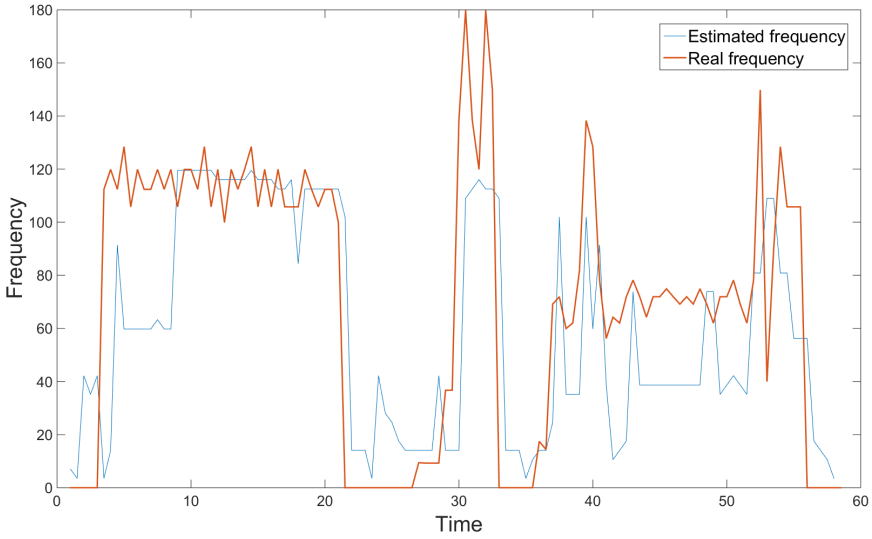


Figure 5.3: Estimated frequency versus real frequency using sum of changes with ROI for video B.

long hair with varying compression rate or lots of changes in position can also be an issue, as this will cause the hair to move very inconsistent.

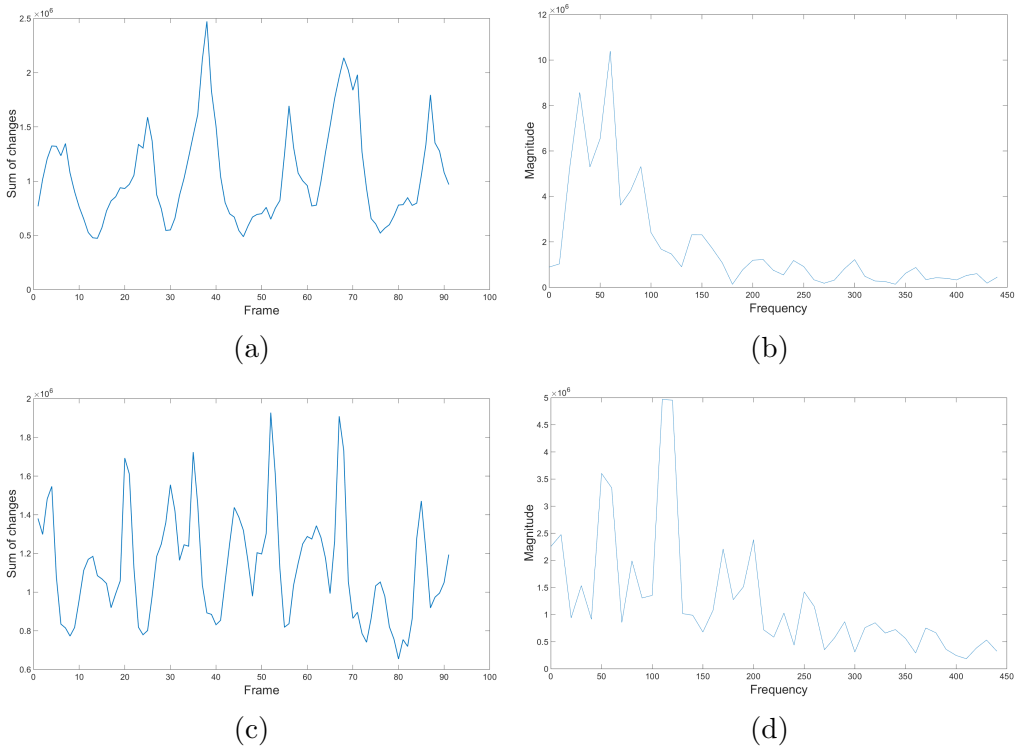


Figure 5.4: Sum of changes with accompanying FFT plots.

5.2.2 Optical flow

Optical flow (section 3.4) could in theory be an excellent way of detecting motion and motion directional changes in the video. It was however not tested to great extent, because it quickly became clear that is was to computationally expensive. During testing in MATLAB it was not able to analyze a video in real time, thus it was swiftly abandoned as a possible solution. Calculating the optical flow was too slow on a desktop computer, so finding the optical flow and then analyzing the result to find when the motion direction had changed, on a smartphone, would not work in real time.

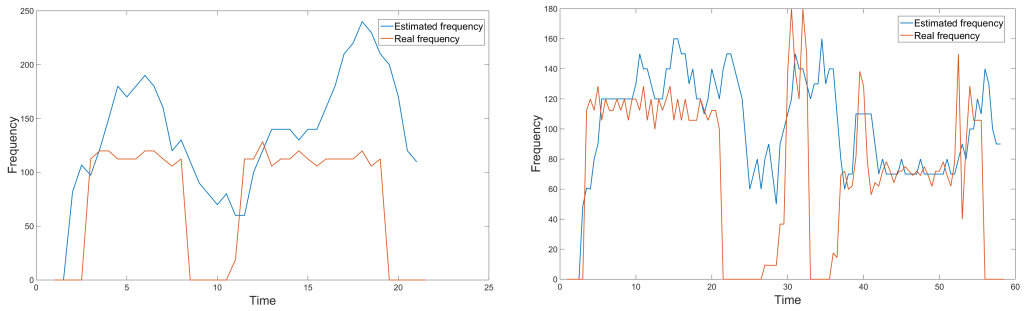
5.2.3 Detecting vertical and horizontal directional changes

This method has been tested in several variations, as described in section 3.5. One of the variations was to find the edges by searching the entire region of interest in each frame. This variant was quickly abandoned in favor of finding

edges in new frames based on the edges found in the previous frame. It is logical that when you are trying to find in which direction the edge is moving, you should consider the same edge in all frames, or at least use the previous edge location as a starting point when searching for the edge point in the next frame. Three different approaches to detect when the motion direction had changed were tested:

Individual edge points

As figure 5.5 shows, this method has promising results for test video B (although a bit high a few places to the left in the graph), but yields to high frequencies for test video A. The results for video B has an average error of 39.9 (SD 45.5) and for video A the average error is 64.0 (SD 44.7) compared to the true frequency in the video. This also happens with other videos tested; the results are simply not consistent enough. This is probably caused by how the edge points are found. The method is too simple and not robust enough, causing the inconsistencies.



(a) Test video A.

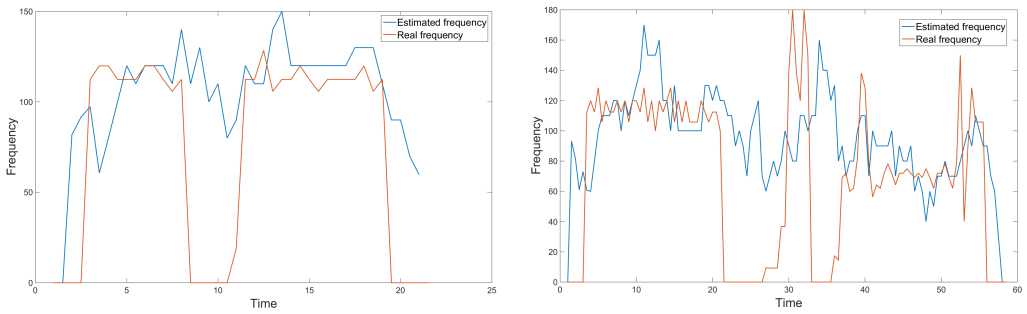
(b) Test video B.

Figure 5.5: Frequency using individual edge points versus real frequency.

Left, upper, right and bottom edge

In figure 5.6 we see the results from test video A and B. Blue represents the estimated frequency using this algorithm and red is the real frequency in the video. As we see the results are not that far from being correct, apart from the areas where the frequency drops to zero or near-zero. This can be explained partly because of the windowing-effect (the frequency is averaged

over a window), but mostly because of the way this method works. The edges can change location rapidly, and the algorithm will still search for the next edge as usual. If it cannot find the edge within the search range, it will search the entire width/height of the frame, thus quickly finding even a rapid moving edge. Since the edge location may change very fast (depending on the movement in the video), it is necessary to do it this way. The results for test video A has an average error of 36.9 (SD 39.0) and the results for test video B has an average error of 31.1 (SD 36.1) compared to the true frequency.



(a) Test video A.

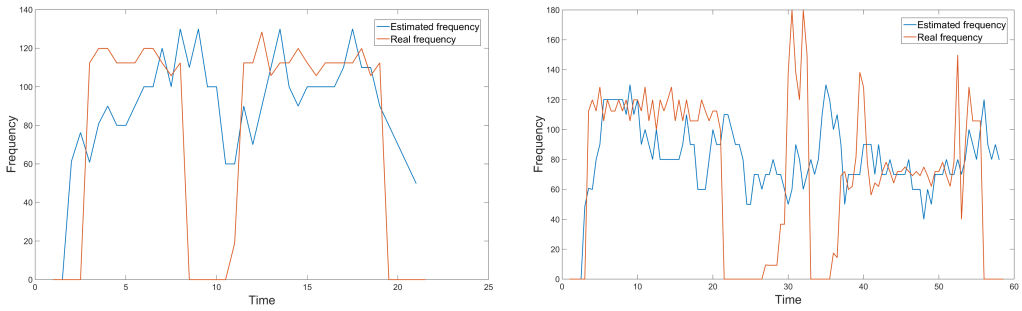
(b) Test video B.

Figure 5.6: Frequency using all edges versus real frequency.

Left and upper edge

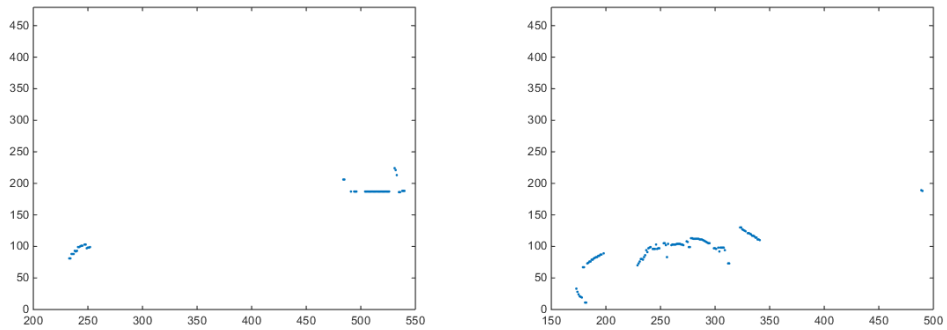
In figure 5.7 we see the result from analyzing the same videos as in figure 5.6, but here using only the left and top edge of the motion to decide when the direction has changed. As we can see, these results are quite similar to the previous ones. This means that finding the turning points using all four sides are comparable to only using two sides. This seems logical, considering the motion involved; when one side of the person changes direction, the opposite side should also change direction.

Analyzing the results from using the left and upper edge, as well as all four edges, closer, reveals the downside of this method; there seems to be a bit too high degree of coincidences involved. For example, in figure 5.8 we see the plot of the top edge for two consecutive frames. As we can see, they are far from being similar.



(a) Test video A. (b) Test video B.

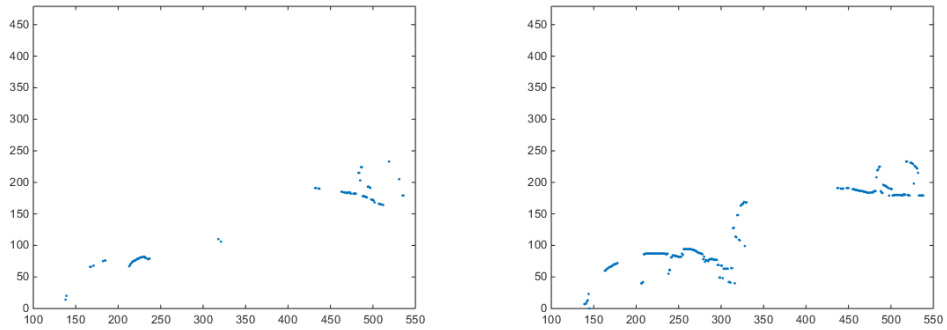
Figure 5.7: Frequency using left and top edges versus real frequency.



(a) (b)

Figure 5.8: Example of top edges found of two consecutive difference frames.

Later in the same video, we get the edges (from the top) as seen in figure 5.9 These are slightly better than the previous one, but still far from being acceptable. There are frames where the edge points found are consistent across frames, but all in all this method shows a severe lack of consistency.



(a) (b)

Figure 5.9: Example of top edges found of two consecutive difference frames 2.

5.2.4 Using contours

This method initially showed promising results. Figure 5.10 shows the frequency found in test video A, compared to the real frequency found in the video. As we can see, the blue line (estimated frequency) is fairly close to the red line (real frequency). They both display two distinct periods of compressions and they have roughly the same frequency found in the peaks.

The estimated frequency is too slow to detect drops in the frequency. This happens because the frequency is estimated inside a sliding window (here it is 3 seconds 90 frames); it is averaged inside that window. Still, the results for this video looks very good. Reducing the window size will make the gap between the two peaks in the graph be more correct, but it will make the peaks less accurate. This can be seen in figure 5.11. Here, we see a more distinct gap between the peaks (closer to the real frequency), but the peaks themselves are more jagged and inaccurate.

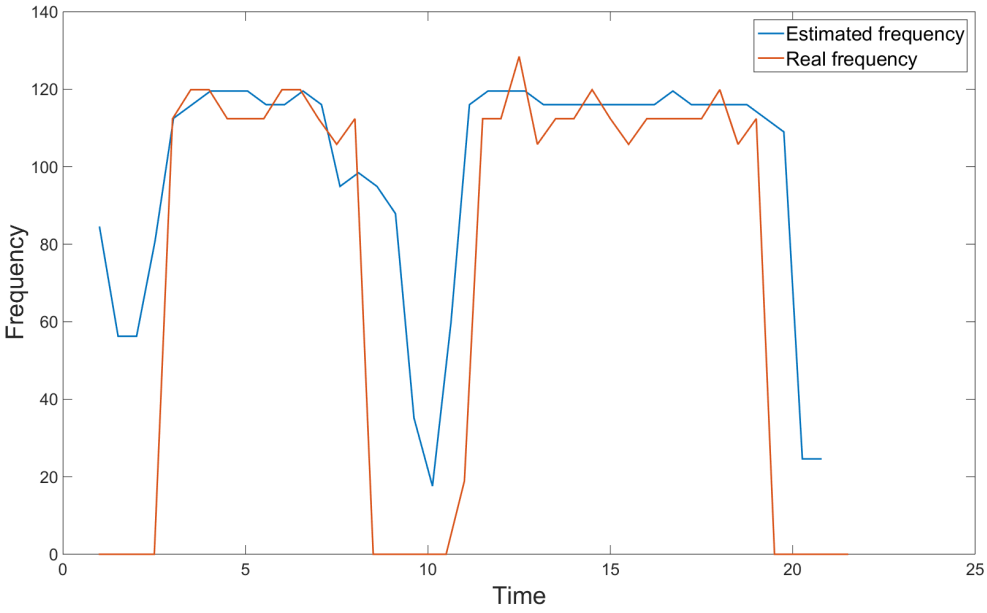


Figure 5.10: Estimated frequency versus real frequency using contours, video A.

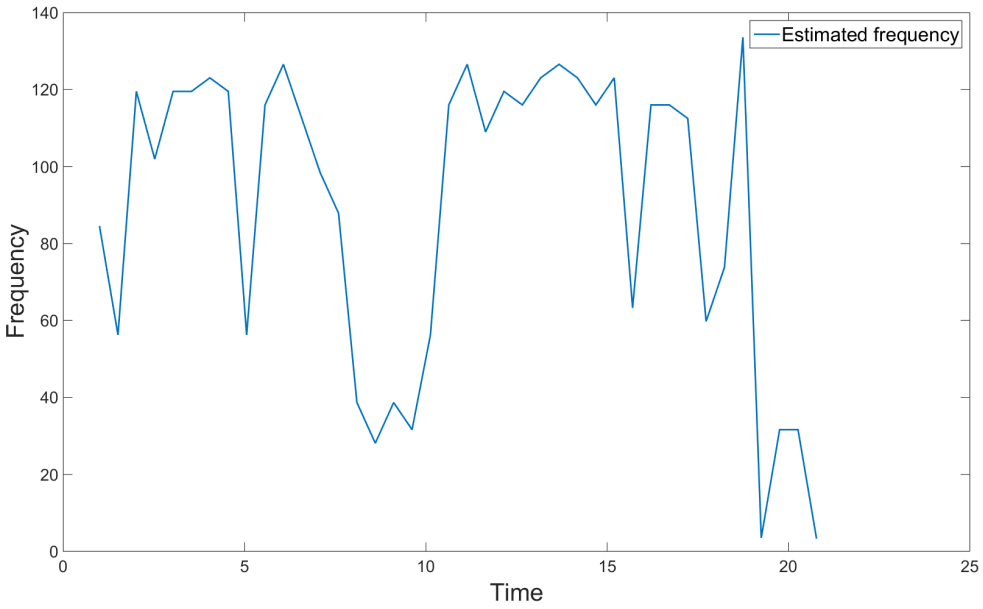


Figure 5.11: Estimated frequency using smaller window, video A.

Analyzing test video B gives quite different results, as seen in figure 5.12. Looking at the results we see that it has correct results some places, but all in all it fails.

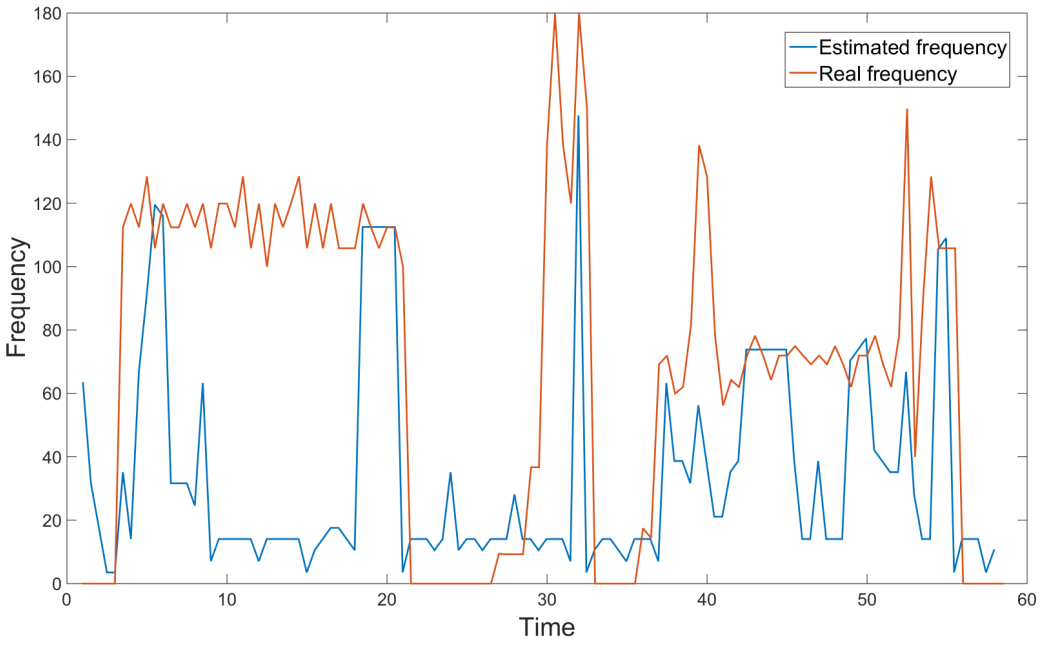
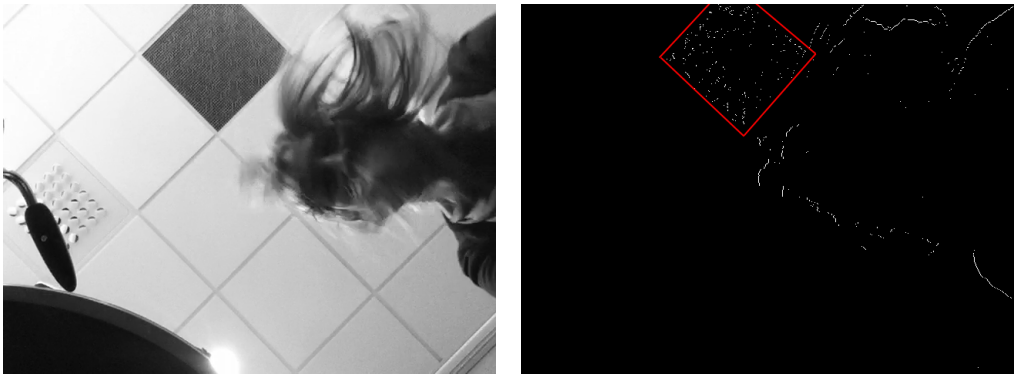


Figure 5.12: Estimated frequency versus real frequency using contours, video B.

Investigating the video and results further reveals why. There are a lot of noise in the difference images, as seen in figure 5.13b. The red square marks an area consisting entirely of noise. Figure 5.13a shows what the video looks like at the same time. Here we can see that the noise is caused by a moirè-pattern in the video. Moirè is not unusual in videos, but either way it shows that this method is heavily affected by noise in the difference images. Using a region of interest could remove some of the noise, but the trouble would be to find the proper region of interest. As seen in this particular frame, the noisy area is directly adjacent to the movements of the person, thus making it hard to exclude if from the ROI.



(a) (b)

Figure 5.13: Frame from video B and associated difference image.

5.3 Android implementation

A series of tests is done to check the performance of the implemented Android algorithm. The tests were done by doing CPR on a test mannequin while recording the measured frequency on the mobile phone. This was done at Laerdal Medical. The facilities at Laerdal can record the chest compression frequency from the test mannequin, and this is used as the reference frequency. The results from the smartphone is then compared with the reference frequency. The tests were performed by different people, quickly described in table 5.2, and under various conditions. There will always be some lag between the real and measured frequency. The results have been corrected for this lag, to avoid skewing the results too much. All results are seen in table 5.3 with explanations and analysis below. All graphs show the filtered and unfiltered frequency from the algorithm and the reference frequency from the test mannequin. To help analyze the results, all test were recorded by a secondary camera from the same position as the smartphone. More information on these videos, as well as test results not displayed here, can be found in Appendix C.

Test person	Description
1	Woman with short hair
2	Woman with long hair
3	Woman with sholder length hair
4	Man with short hair
5	Woman with long hair
6	Man with short hair

Table 5.2: Test persons doing CPR.

Number	Short description	Test person	Mean error	SD	Max error	Duration
1	Only compressions, steady rate	1	2,7	1,9	10,0	68
2	Some sideways movement	1	6,0	16,3	109,7	73
3	Variation in rate, other people in frame	1	4,0	4,0	29,5	70
4	CPR 30:2	1	10,4	18,6	100,0	228
5	Loose hair	2	46,8	18,6	65,4	60
6	Hair tied up	2	11,4	23,8	113,6	118
7	CPR 30:2, hair tied up	2	21,5	40,7	190,0	77
8	Lots of non-compression movements	2	17,2	22,2	100,0	206
9	Only compressions, loose hair	3	55,7	4,1	70,4	57
10	Hair behind ears	3	16,7	24,0	117,2	86
11	Only compressions, steady rate	4	4,0	2,3	10,0	33
12	Includes a pause	4	3,0	3,4	28,0	49
13	Only compressions, steady rate	4	8,0	8,9	54,9	25
14	Only compressions, steady rate	4	3,7	2,2	10,0	58
15	Low compression rate, $\sim$ 50 cpm	4	10,2	9,0	30,3	35
16	High compression rate, $\sim$ 160 cpm	4	5,0	3,6	17,5	21
17	Variation in rate	4	3,9	4,4	36,7	67
18	Bad camera placement	4	13,0	17,5	83,3	32
19	Other people in ROI	4	4,3	4,6	41,4	40
20	Loose hair	5	40,9	25,9	112,9	74
21	Hair tied up	5	3,3	3,2	28,0	90
22	Loose hair, with changing hair position	5	34,6	23,2	76,4	65
23	Partially tied up and partially loose hair	5	19,1	22,8	101,0	157
24	CPR 30:2, hair tied up	5	6,6	16,3	113,6	178
25	Variation in rate	6	6,7	13,8	111,1	285
26	CPR 30:2	6	11,4	21,8	180,0	190

Table 5.3: Android test results.

Test 1

Very good results are shown in this test. No big errors, and generally very close to true frequency. The test were comprised of standard compressions with a steady and correct rate.

Test 2

Good results displayed here as well (figure 5.14). Two major drops in frequency is reported by the algorithm. Analyzing the video shows that the first happens because the test person shifted position slightly, and the second was caused by sideways movement. This causes the person to have some of its movements outside the ROI, thus resulting in wrong frequency measurement. Some of the movements is simply ignored.

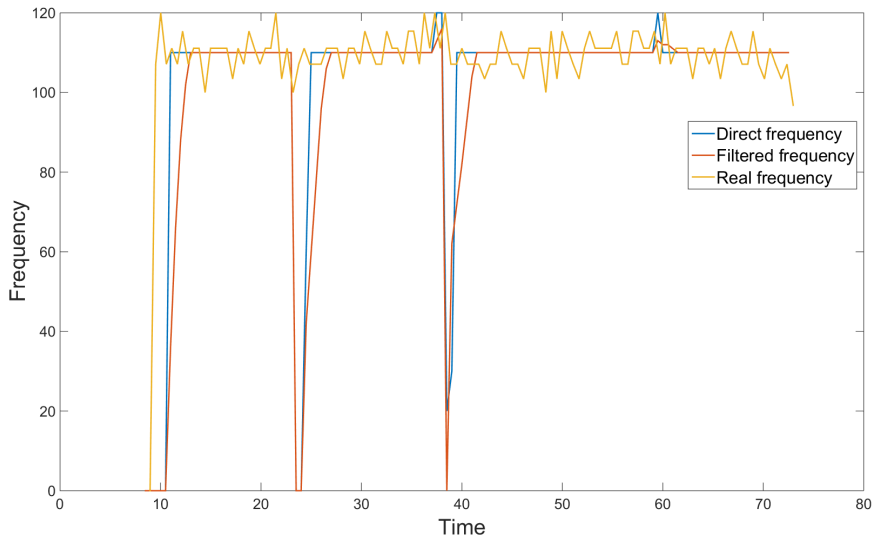


Figure 5.14: Android test results from test 2: some sideways movement.

Test 3

Figure 5.15 shows that the algorithm copes well with variations in speed. The video reveals that there were other people going in and out of the frame; because of the ROI this does not affect the results.

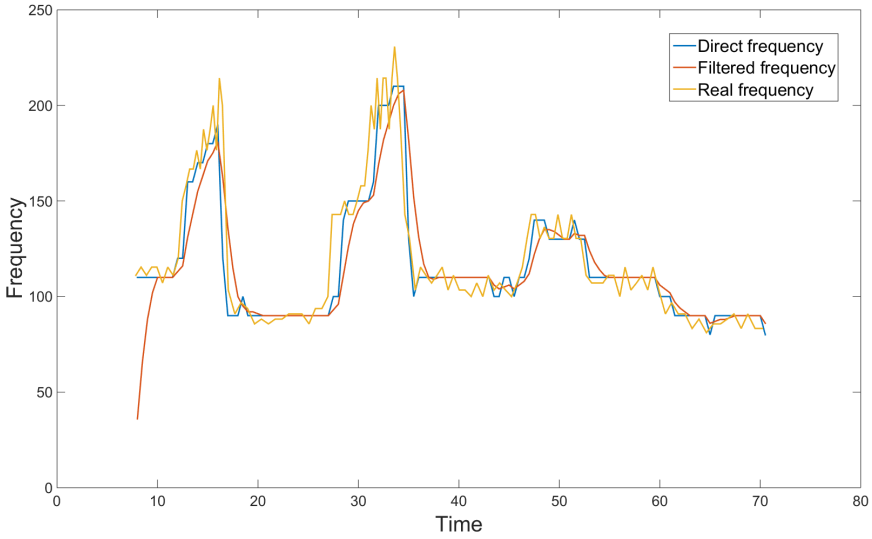


Figure 5.15: Android test results from test 3: variation in rate, other people in frame.

Test 4

CPR 30:2 (30 compressions followed by 2 breaths) were performed in this test. The results are very good, except some spikes in between the compression periods. There are also some variation in speed, which it handles very well. The spikes are probably caused by two things, either some of the test person is visible in the frame during breaths or some noise is misinterpreted as movement.

Test 5

Test 5 consists of normal compressions, but the test person has long and loose hair, causing many errors. See figure 5.16. The algorithm often measures just half of the real frequency. This is probably caused by the inconsistent movements of the hair interfering with the movements of the head, where some of the movements cancel each other out (to some extent).

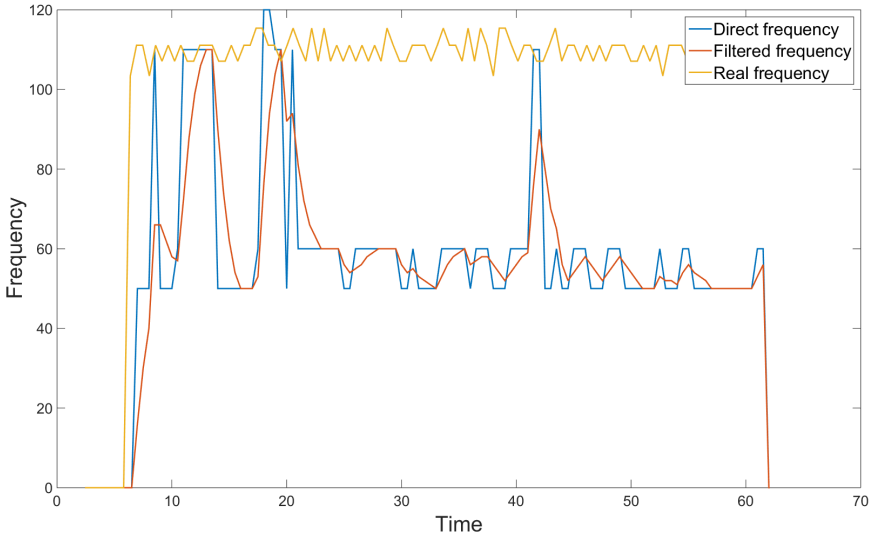


Figure 5.16: Android test results from test 5: loose hair.

Test 6

This test gives much better results than the previous one. It is the same test person, but now with her hair up. There are still some drops, but they are caused by sideways movements (exiting the ROI).

Test 7

Same as test 6, but now performing CPR 30:2. Similarly to test 4, there are some spikes in between compression periods, here probably caused by noise in the video.

Test 8

This test is not realistic and there are few actual compressions. It is included to test how the algorithm responds to a person just moving around in the frame, instead of performing CPR. Figure 5.17 shows many spikes, caused by this seemingly random movement.

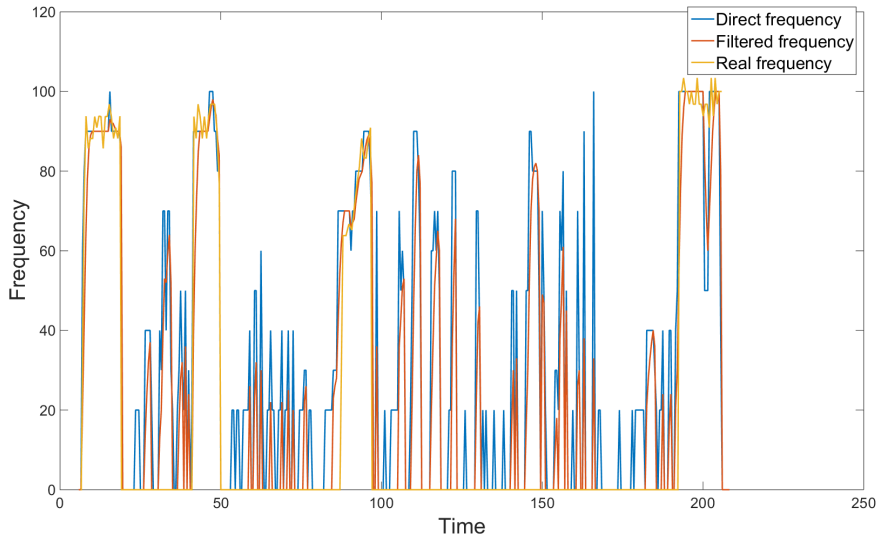


Figure 5.17: Android test results from test 8: lots of non-compression movements.

Test 9

This test results are comparable to test 5. The test person has shorter hair (around shoulder length) but the results are still roughly half of the real frequency. Clearly, this algorithm struggles with loose hair.

Test 10

When the test person has her hair behind her ears, the results are much better (figure 5.18). The errors in the last half is caused partially by some hair becoming loose and partially because of the changing lights. In this test the lights in the roof were turned on and off again a couple of times in the last half. They were initially off, were turned on at 54 seconds, off again at 72 seconds and finally on again at 83. When the light changes quickly it causes the entire video frame to change, because the auto exposure compensates for the change in illumination. Also, when the lights are on, the test persons face becomes darker, to avoid blown out highlights. In turn, this makes it so that the hair contributes relatively more to the movement (since there are less details in the face). Given that the hair causes problems, this is bad.

When the lights are turned off, the frequency is measured correctly.

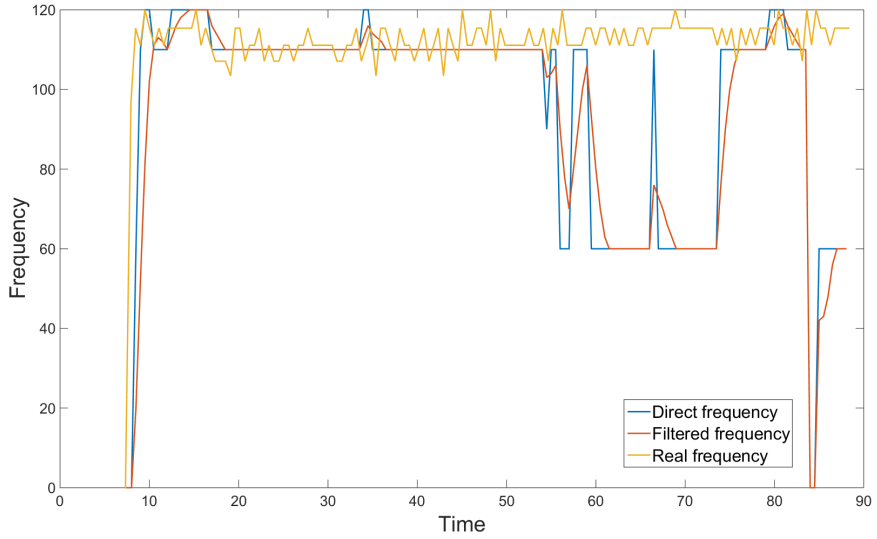


Figure 5.18: Android test results from test 10: hair behind ears.

Test 11

This test consisted of ordinary compressions and a steady rate. The results are very good, and it shows how good the algorithm performs under optimal conditions.

Test 12

Similar to test 11, but includes a pause. The results are still very good.

Test 13

This test is similar to test 11. The results show a small drop in the end, which is caused by the test person turning his head. The frequency quickly recovers to the correct level afterwards.

Test 14

Similar to test 11, still very good results.

Test 15

In this test lower compression rate (around 50 compressions per minute) were tested. We see (figure 5.19) that the algorithm struggles a bit here. Analyzing the video for this test, reveals that the compression movements are done in a staccato manner. The test person pauses some between compressions, to keep the rate this low. This movement patterns seems to trick the algorithm.

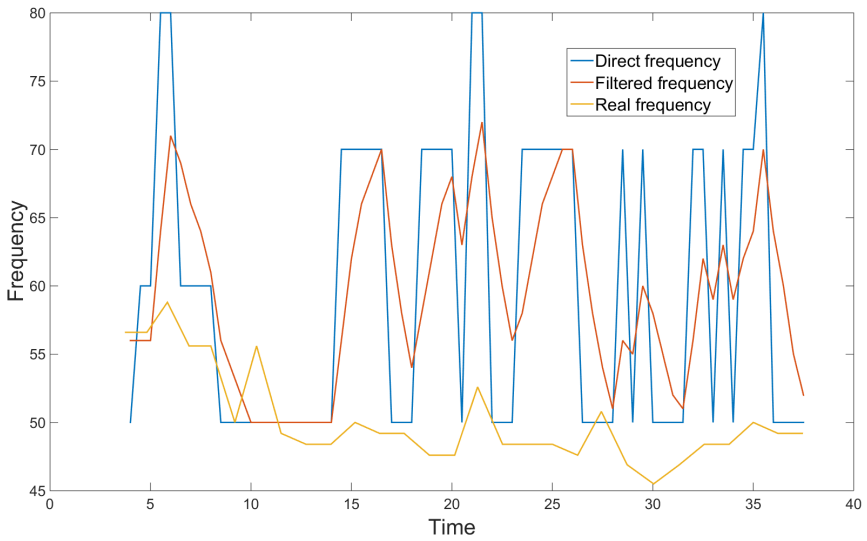


Figure 5.19: Android test results from test 15: low compression rate, $\sim$ 50 cpm.

Test 16

In this test compressions with a higher than recommended rate were used (around 160 compressions per minute). The results are very good; the algorithm handles high frequencies good (at least reasonably high).

Test 17

Figure 5.20 shows the results of test 17. This test included variations in speed, which the algorithm handled very well. There are two small spikes in the beginning, which as we can see is mostly corrected by the filter.

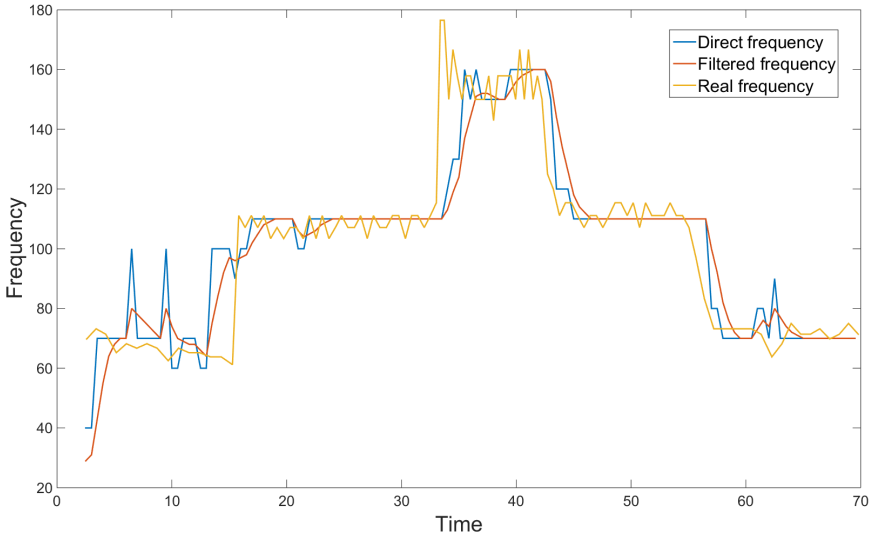


Figure 5.20: Android test results from test 17: variation in rate.

Test 18

In this test the test person were only just visible in the frame (only three quarters of the head were visible). The results are still good (figure 5.21). There is a drop around 12 seconds in, caused by the test person turning his head. The spikes in the end are probably caused by some movement happening outside the ROI (the ROI has not updated fast enough), because the test person slightly shifted position. When the movement only takes up a fraction of the frame, it has a bigger effect when some of the movement is outside the ROI (since the portion of the movement happening outside the ROI can be a relatively big part of the motion).

Test 19

This test included someone walking into the frame, while the test person were performing CPR. The other person did not interfere with the frequency measurement (because of ROI) until the end (seen in figure 5.22). In the end the person was so close to the test person, that he entered and became a part of the ROI. This lasted very briefly, but caused a drop in frequency. Since the drop were small, it was in this case mostly corrected by the filter.

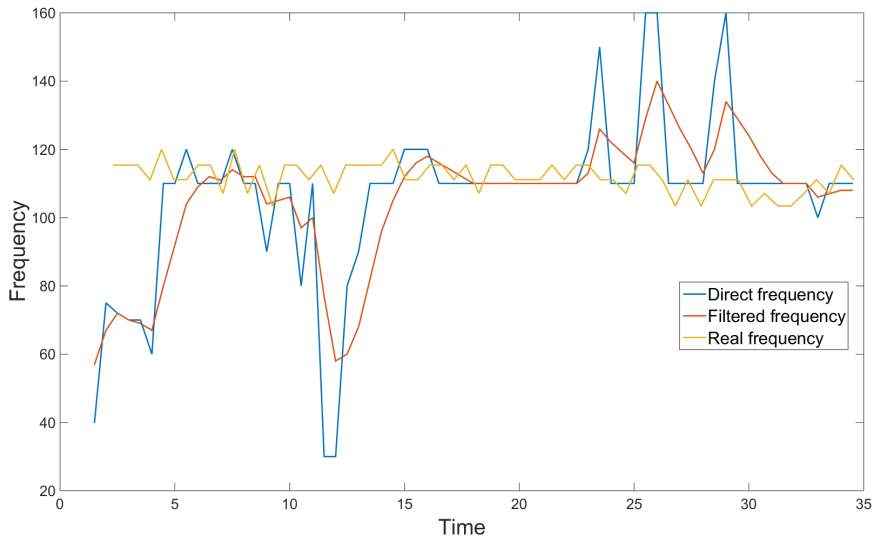


Figure 5.21: Android test results from test 18: bad camera placement.

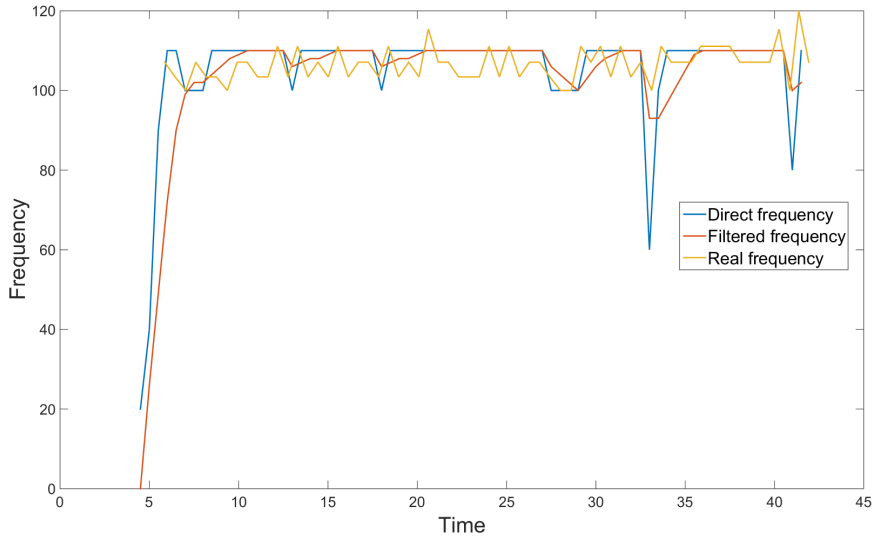


Figure 5.22: Android test results from test 19: other people in ROI.

Test 20

The algorithm struggles with loose hair, as the test person had here. As previously with long loose hair, the results are often half of the correct. There are also parts in between were it manages to find the correct frequency.

Test 21

Similar as test 20, but now with the test person's hair up. This yields much better results. The test also included change in compression rate, which once again the algorithm handled very well.

Test 22

This test starts similar to test 20, but around halfway through the test person moves her hair over to the other side of her head. This causes the hair to move more up and down, and less side to side. This gives better results (figure 5.23) compared to the first half, but still not correct all the time. Here we also see a negative aspect of the filter; it reduces the height of the correct peaks right after the pause (between 32 and 40 seconds).

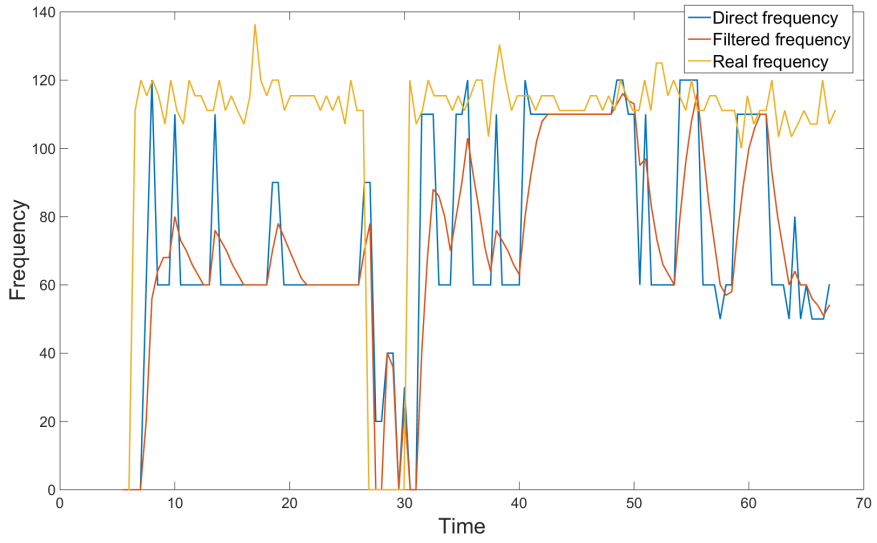


Figure 5.23: Android test results from test 22: loose hair, with changing hair position.

Test 23

Figure 5.24 shows the results for this test. The test person had her hair up. Around the middle we see a large error. This happened because the hair had gotten partially loose again. The test person stopped and put her hair up again, and the results after that were pretty accurate.

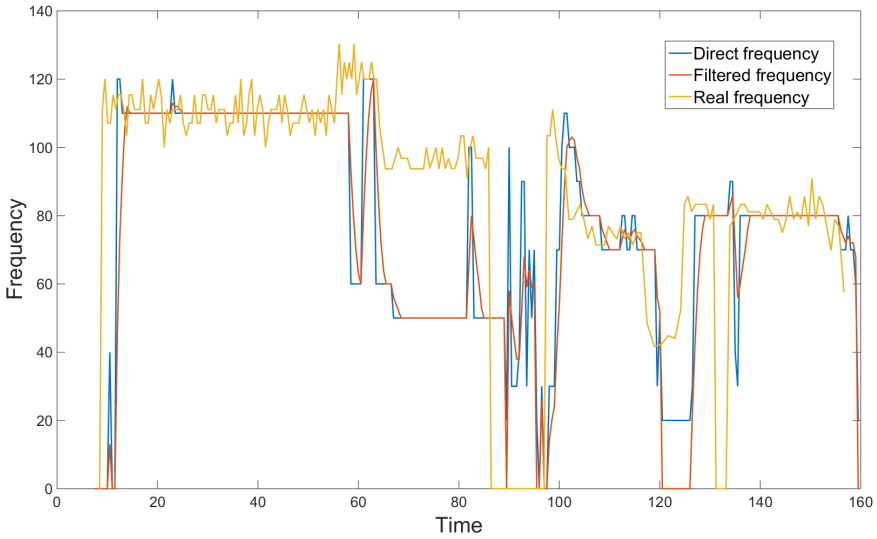


Figure 5.24: Android test results from test 23: partially tied up and partially loose hair.

Test 24

CPR 30:2 were performed in this test. The test person had her hair up and the results are very good. There is a peak right after one of the pauses (breaths) which most likely is caused by some sideways movement (in addition to the compression movement) or slow ROI.

Test 25

This test included large variations in frequency rate and the measured frequency is very accurate. Figure 5.25 shows that it struggles some when the frequency are very low, but it mostly handles the rest very good. The first

drop is caused by a large sideways movement. There are some small peaks in the middle, caused by another person entering the ROI. Here we see that the filter mostly cancels out these peaks. The drops in the end are caused by the lights in the ceiling that were switched on and off. It quickly finds the correct frequency after the drops, and the smaller drops are mostly corrected by the filter anyway.

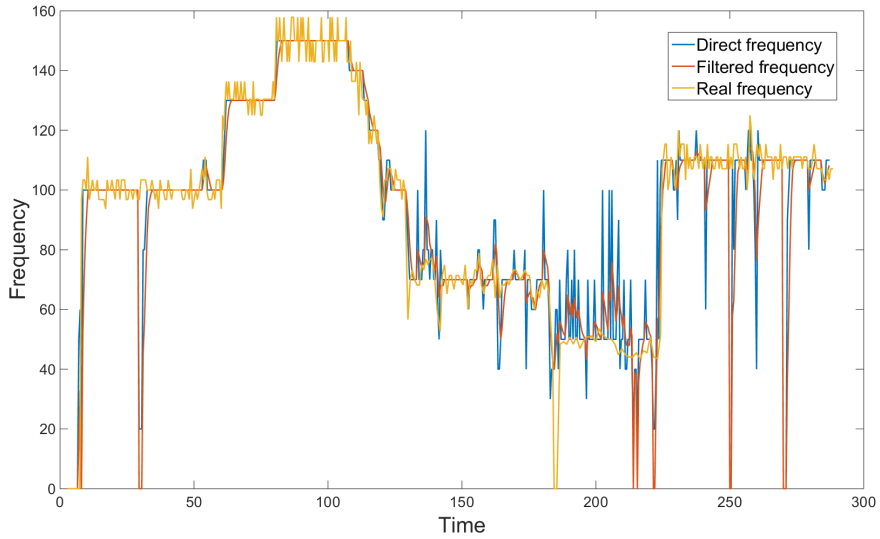


Figure 5.25: Android test results from test 25: variation in rate.

Test 26

In this test, CPR 30:2 were performed. The results are overall very good. As previously seen in CPR 30:2 tests, there are some spikes in between the compressions periods. In this test they are caused by a partially visible shoulder in the frame (during breaths). There is also a drop during one of the compression periods, which is caused by the light being turned on (and changing the illumination in the entire frame).

5.3.1 Test summary

Overall, the algorithm implemented in the Android application produces mostly accurate frequency measurements. As shown in the tests, it strug-

gles severely when the person performing CPR has long and loose hair, but it works well when the test person has their hair up or has short hair. Of course, there is still room for improvements:

- There are often spikes between CPR periods when performing CPR 30:2. This seems to be caused by either a visible shoulder in the frame, or by noise. It is hard to distinguish between a moving shoulder and actual compressions. This could be solved by introducing pattern recognition to recognize the pauses in compressions. We know approximately how long each compression period lasts (since we know the frequency and number of compressions in each period), so it should be possible to learn when there is a pause and when there are actual compressions happening. Most of the noise in the images are already removed by thresholding. Most of the noise in the test videos are white noise caused by a ventilation grid in the ceiling, which is not normal in most private houses, and it only caused problems in some of the test videos because if was situated directly above the test persons.

- When the test person turns his/her head, it often causes a short drop in measured frequency. This is probably caused by the extra head movement canceling out some of the other movements. This should not be a big problem however, since the frequency is quickly corrected afterwards. A more slowly reacting filter could help, but it would also lead to slower reaction times when the frequency actually is changing.

- Sideways movements (roughly perpendicular to the compression motion) seems to cause drops as well, if the movement is too fast. This is probably caused by the ROI not updating quickly enough, causing some of the movements to happen outside the ROI, effectively being ignored. The ROI updating has intentionally been made a bit slow (it requires two rounds with movement to be included in the ROI), to reduce the chance that movements from other people would interfere with the frequency measurement. Sideways movements do not happen that much when doing real CPR, most cases in the testing was done intentionally to test how the algorithm responded to it. Therefore it should not be a big problem. Anyway, the algorithm quickly found the correct frequency when the sideways movements stopped.

- When the test person doing CPR performed a very low compression frequency (around 50 compressions per minute), there was some spikes

in the measured frequency. The algorithm implementation favors the higher frequencies and will therefore select the highest when there are two plausible measurements. This should however not be a big problem in this case, since the spikes are small. The person analyzing the frequency output would see that the frequency is lower than the recommended CPR frequency and inform accordingly.

- There are drops in measured frequency when the lights in the ceiling are turned on or off. The auto exposure compensates for the change in illumination and all pixel values change. If this change happen too fast, the thresholding will not be enough and it will interfere with the measurements. This should not be a severe problem however, since we can expect the lights do not change that often in realistic situations. The correct frequency is quickly recovered afterwards, as well.

- Other people entering the ROI causes interference in some cases (depending on their movement). This is hard to avoid. Reducing the block size might help some, but cannot remove the issue. Reducing block sizes will also increase the computational expenses, and will (with the current implementation) make the ROI reacting slower to change in movement positions.

There are several cases where our implemented algorithm works very well:

- Ordinary compressions from persons with short hair works very good. The frequency measurement are accurate and there are no drops in frequency (other than described above).

- People with long hair that has their hair up or behind their ears also works very well and has similarly accurate results.

- It quickly detects changes in compression rate, both rising and falling rate changes.

- It handles high compression rates good. It has been tested with (and accurately detected) frequencies up to 160 compressions per minute.

- Use of ROI to avoid interference from other people. As long as the other persons do not enter the ROI (they can become part of the ROI if they move close enough), they can move around and not interfere at all with the measurements.

- Detect the frequency even when the person performing CPR is barely visible in the frame. There are some inaccuracies in this case, but still the results are quite good.
- It works well at different light conditions. Tested with lights on and off, and variations in light intensity.

Analysing the results from table 5.3, we see that many of the tests have very small mean error, coupled with very low standard deviation. This means the measured frequency is very close to the reference frequency, and the variations are small. Test 1, 2, 3, 11, 12, 13, 14, 16, 17, 19, 21, 24 and 25 all have mean errors of less than 10, which is very good. Some of these tests have a high maximum error, mostly caused by spikes or drops in the measured frequency. However, this does not affect the results very much, as evident by the low mean error. Test 1 shows the best results, with a mean error of only 2.7 (SD 1.9). This test happened under optimal conditions, and shows how good the algorithm can perform.

Tests done by people with long and loose hair (test number 5, 9, 20 and 22) have the worst results when considering mean error. However, the standard deviation is in relative terms not that high, meaning that the algorithm fails consistently (to a certain degree) in these scenarios. Still, the algorithm fails.

The algorithm handles CPR 30:2 good, seen by the results of test 4, 7, 24 and 26. These tests have high maximum errors, caused by spikes in measured frequency during compressions pauses (breaths). These spikes are caused by a visible shoulder or noise in the frame.

Apart from the tests involving persons with long and loose hair, the mean errors and accompanying standard deviations are generally good. There are some high maximum errors on some of the tests, but they are often caused by short spikes or drops in measured frequency.

6 Conclusion and future work

This chapter will conclude the work done in this thesis and present suggestions for future work. The first section is the conclusion and the last is future work.

6.1 Conclusion

Performing CPR using the recommended frequency of 100-120 compressions per minute can potentially help saving lives. In this thesis, we have proposed an algorithm that use a combination of differential motion analysis and discrete Fourier transform to measure the compressions frequency. The algorithm is implemented in an Android application, and all the calculations are done on the smartphone in real time.

The algorithm is proven to work well under most circumstances, with one major exception. When the person doing CPR has long and loose hair, the algorithm will most of the time report half of the correct frequency. This is because the movements of the loose hair interfere with the movements of the head, to some extent canceling it out. When the person has the hair up, or has short hair, the results are very good. The algorithm is able to detect the correct frequency in most situations. It has been tested with

frequencies as low as 50 cpm (resulting in some spikes) and up to 160 cpm (no issues). Variations in compression rate is quickly detected, both rising and falling rate changes. Since it handles rising and falling rates so well, it also works well if the person is doing CPR 30:2. When doing CPR 30:2 there are some occasional spikes between compression periods, but this should not be a problem in real life use.

The algorithm is very sensitive; it works even when only a portion of the rescue person's head is visible in the frame. Tests shows that the ROI strategy works as intended. However, if other people get so close to the person doing CPR that they become a part of the ROI, the results are unpredictable. The ROI updating has intentionally been made slower, to reduce the chance that other people would interfere with the frequency measurement. This has a negative aspect; when the person doing CPR turns their head or moves sideways (perpendicular to the compression movements), some of the movement will for a short period be outside the ROI. This causes the measured frequency to drop. The correct frequency is quickly recovered however (when the ROI is updated to include all the movement), so this should not be a big issue.

6.2 Future work

Suggestions for future research work is presented below.

Improvements on output filter

The weighted moving average filter applied to the frequency measurement output is rather simple. Making this filter smarter could improve the perceived performance of the algorithm. It could for example react slower, which would cover up a lot more of the frequency spikes/drops. Experiments should be done to determine what the optimal reaction time should be, to cover up most of the spikes/drops but still react quickly to real changes in frequency rate. It could also be improved by having it react better to the large variations in reported frequency when the compressions start or stop.

Improvements on ROI handling

There are room for improvements on the definition of ROI. Right now, the block size for image blocks (and therefore ROI blocks) is 50×50 pixels.

Experiments can be done to determine if this is the optimal size or if it should be altered. Reducing the block size will reduce the chance that other persons interfere with the person doing CPR, but it might put additional computational strain on the smartphone. One can even do experiments where the ROI blocks are divided even further, when checking if new blocks should be included in the ROI (trying to determine if it is the motion inside the ROI expanding or if it is outside movement entering the ROI).

If there are more than one ROI area, we select the largest one. This can be improved by estimating the frequency inside each area and selecting the area with the most probable frequency, or select based on some other condition. A combination of several factors should be examined.

Pattern recognition to recognize CPR 30:2

During CPR 30:2 there can be a visible shoulder (or some other body part) in the frame, causing errors. Introducing pattern recognition to first detect if the person is performing CPR 30:2 or just compressions, and then learn when there are pauses and when there are compressions, could improve the results greatly in these cases.

Improved analysis of sum of changes graph

Right now, the sum of changes is analyzed using a combination of discrete Fourier transform and by counting the number of extrema and average-crossings. Experiments should be done to test different approaches, to see if there is a method that works better.

Solving the hair problem

This is the biggest issue with the current algorithm; it struggles with long and loose hair. Different approaches to measure the frequency should be evaluated, to see if there are a better method than the one currently used.

Create Android library

The algorithm should be converted into an Android library, so that other applications can easily implement it. Before doing so, more testing on different smartphones and different versions of Android needs to be done, to ensure compatibility.

A

MATLAB and Android source code

All source code developed during the work with this thesis is attached to this document as a zip file. The file is named "Source code", and is structured as follows:

MATLAB prototypes This folder contains all the prototypes developed in MATLAB. For further instructions on how to run this code, please see the readme file in the folder.

Android application This folder contains the code for the Android implementation. This code was developed in Android Studio, and the easiest way to run it is to import the entire folder/project and run it from Android Studio. Alternatively the apk can be manually installed on a Android phone by copying the file to the phone and installing it manually. The apk can be found in the "...\app\build\outputs\apk" folder. Fore more information on how to install an unsigned apk, please see the user manual for your phone.

MATLAB tools This folder contains the functions and scripts developed to help with the creation of the results seen in this thesis. More information can be found in the readme file in the folder.

B
Android test results

All results from the tests on the Android application can be found in the attached file "Android test results". The folder is divided into sub folders, named from 1 to 26, representing the 26 tests analysed in the results and analysis chapter, in table 5.3. Inside each folder is the plot comparing the algorithm results with the reference frequency, the algorithm result files saved from the application, the mannequin data and a mat file used to produce the resulting graph and error measurements in MATLAB.

C

Contents on applied DVD

The applied DVD contains all files attached to this document, as well as some additions:

Algorithm test videos All the videos recorded during the testing of the Android algorithm implementation. The videos is found inside each test folder (1-26).

Prototype videos This folder contains the videos used in the MATLAB prototype development. They are named according to the description given in the MATLAB prototypes chapter.

Bibliography

[1] Tonje Søraas Birkenes, More and better CPR to victims of sudden cardiac arrest, Thesis for the degree PhD, 2013.

[2] Van Hoeyweghen RJ, Bossaert LL, Mullie A, et al. Quality and efficiency of bystander CPR. Belgian Cerebral Resuscitation Study Group. Resuscitation 1993;26:47-52.

[3] Jerry P. Nolana,, Jasmeet Soar b, David A. Zideman, et al., European Resuscitation Council Guidelines for Resuscitation 2010 Section 1. Executive summary

[4] Idris AH, Guffey D, Pepe PE, Brown SP, et al. Chest compression rates and survival following out-of-hospital cardiac arrest. Critical Care Medicine: April 2015 - Volume 43 - Issue 4 - p 840848

[5] Rea TD, Eisenberg MS, Culley LL, et al. Dispatcher-assisted cardiopulmonary resuscitation and survival in cardiac arrest. Circulation 2001;104:2513-6.

[6] ZOLL PocketCPR,Android application, `https://play.google.com/store/apps/details?id=com.pocketcpr.poccpr`

[7] J. A. Jr., L. Darcey and S. Conder, Introduction To Android Application Development, Pearson Education, 2014

[8] Android history `https://www.android.com/history/` [Accessed 18 February 2015].

[9] About Android `https://developer.android.com/about/index.html` [Accessed 18 February 2015

[10] Android Main Site `https://www.android.com/` [Accessed 18 February 2015].

[11] Android Studio `https://developer.android.com/sdk/index.html` [Accessed 18 February 2015].

[12] About MATLAB `http://se.mathworks.com/products/matlab/` [Accessed 16 February 2015].

[13] MATLAB matrices `http://se.mathworks.com/help/matlab/learn_matlab/matrices-and-arrays.html` [Accessed 16 February 2015].

[14] About Computer Vision System Toolbox `http://se.mathworks.com/products/computer-vision/` [Accessed 17 April 2015].

[15] About Image Processing Toolbox `http://se.mathworks.com/products/image/` [Accessed 17 April 2015].

[16] About Signal Processing Toolbox `http://se.mathworks.com/products/signal/` [Accessed 17 April 2015].

[17] M. Sonka, V. Hlavac and R. Boyle, Image Processing, Analysis, and Machine Vision, Cengage Learning, 2008.

[18] A. Radguia et al., Optical flow estimation from multichannel spherical image decomposition, Computer Vision and Image Understanding, Volume 115, Issue 9, September 2011, Pages 12631272.

[19] Hiroshi Sugita,Akira Taguchi, Chrominance Signal Interpolation of YUV 4:2:0 Format Color Images, Electronics and Communications in Japan, Part 3, Vol. 89, No. 9, 2006.

[20] Android Camera PreviewCallback `http://developer.android.com/reference/android/hardware/Camera.PreviewCallback.html` [Accessed 24 February 2015].

[21] John G. Proakis, Dimitris G. Manolakis, Digital Signal Processing, Principles, Algorithms, and Applications, Third Edition, Prentice Hall-International, 1996.

[22] F. Y. Shih, Image Processing and Pattern Recognition: Fundamentals and Techniques, John Wiley & Sons, 2010.

[23] F. J. Harris, On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform, Proc. IEEE, vol. 66, pp. 51-83, 1978.

[24] Android for developers `http://developer.android.com/develop/index.html` [Accessed 23 February 2015].

[25] Android Activity developer guide `http://developer.android.com/reference/android/app/Activity.html` [Accessed 23 February 2015].

[26] Android AsyncTask developer guide `http://developer.android.com/reference/android/os/AsyncTask.html` [Accessed 24 February 2015].

[27] Android Camera developer guide `http://developer.android.com/guide/topics/media/camera.html#saving-media` [Accessed 24 February 2015].

[28] Android Camera2 API `http://developer.android.com/reference/android/hardware/camera2/package-summary.html` [Accessed 24 February 2015].

[29] JTransforms open source FFT library, `https://sites.google.com/site/piotrwendykier/software/jtransforms`, [Accessed 24 February 2015].