NTNU – Trondheim
Norwegian University of
Science and Technology

Tablet based real-time Doppler spectrum processing

Bjørn Rudi Dahl

Electronics System Design and Innovation
Submission date: June 2014
Supervisor: Ilangko Balasingham, IET
Co-supervisor: Hans Torp, ISB
Gabriel Kiss, ISB

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem description

Reducing the mortality rate among unborns, young children and pregnant women is one of the Millennium Developments Goals (MDG 4 and 5) of the United Nations (UN). Diagnostic ultrasound is the only imaging method to be used in pregnancy and widely offered to the general population in developed countries. The Umoja project, ultrasound for midwives in rural areas, aims to develop an extremely low cost, robust and portable ultrasound imaging system (the Umoja ultrasound system) for obstetric imaging, specifically designed for operation in challenging rural areas of developing countries. The project is a joint effort between three main partners: the National Center for Fetal Medicine (NCFM) at St. Olavs Hospital and NTNU, the Department of Circulation and Medical Imaging (ISB) at NTNU, and GE Vingmed Ultrasound AS.

When building a low cost system, there is an inherent trade-off between the Doppler imaging quality and the overall cost of the portable system. The hardware limitations that come with the low cost requirements will limit the available computational power. Therefore it is desirable to implement and execute as much of the signal processing pipeline as possible directly on the tablet device, which includes both CPU and GPU units at a low cost. The thesis will investigate the following topics:

1. - implementation of a real-time spectrum processor on the tablet device
2. - optimization of the Fourier transform and the additional signal processing algorithms on the tablet device
3. - based on data received from the scanner generate the Doppler sound and investigate the use of Android's AudioTrack class for playback purposes directly on the tablet
4. - evaluation of the influence of different parameters on the quality of the Doppler spectrum vs. computation time
5. - optimization of the Doppler triplex mode with live color Doppler support.

Preface

This diploma has been written at the Department of Electronics and Telecommunications at NTNU, Trondheim, and carried out under the Department of Circulation and Medical Imaging at St. Olavs Hospital.

My supervisors during the masters thesis has been Prof. Hans G. Torp and Gabriel Kiss, who both have contributed with good ideas and invaluable feedback during the semester. I want to thank Gabriel Kiss in particular for sharing his knowledge on computer programming and helping me optimize my code.

I also want to thank the rest of the Umoja-team; Morten Dreier, Sturla Eik-Nes, Naiad Kahn, Yücel Karabıyık, Eva Tegnander and Agnes Heyer for making my year at ISB such a great experience.

Lastly I would like to thank Ilangko Balasingham for being my formal advisor.

Trondheim, June 2014
Bjørn Rudi Dahl

Abstract

By utilizing mobile, low-cost Doppler ultrasound technology in developing countries, the mortality rate of unborn children, small children and pregnant women might be reduced. Two main Doppler techniques which are used in ultrasound are the Doppler spectrum and the Doppler audio. The Doppler spectrum is used for quantitative analysis of the blood-flow while the Doppler audio is used for guidance during recording

This thesis presents the implementation of a real-time Doppler spectrum generator and a real-time Doppler stereo audio generator, which is implemented on Android mobile devices; along with real-time ratings and optimizations. The Doppler spectrum implementation was compared to a Matlab implementation, while the Doppler audio was compared to a synthetic test signal. The frequency leakage, in the Doppler stereo audio generation, from positive- to negative Doppler shifts and vice versa was at about $-20dB$ with the synthetic test signal.

The performance of a mid- to high-end Android tablet showed high framerates for Fast Fourier-Transform lengths up to 128 with both 75 and 87.5 percent overlap and audio-playback frequencies of $8kHz$ and $16kHz$. The low- to mid-end tablet displayed low framerates with the current implementation.

The optimization of the Fast Fourier-transform by utilizing FFTW has shown promising results to increase the framerate on the low- to mid-end tablet. Other ways of improving the framerate on low-end tablets is to turn off the audio or turn off the stereo processing along with reducing the settings on the spectrum generator.

Sammendrag

Ved å tilgjengeliggjøre Doppler ultralydteknologi til en lav pris i utviklingsland, kan dødeligheten blandt ufødte, spedbarn og gravide kvinner kanskje bli redusert. To av hovedteknikkene anvendt i Doppler ultralyd er Dopplerspektreret og Doppler lyd. Dopplerspektreret er brukt i kvantitativ analyse av blodstrømsmålinger mens Doppler-lyden er brukt til å veilede brukeren under skanning.

Denne avhandlingen presenterer implementasjonen av en sanntids Dopplerspektral analysator og en sanntids Dopplerlyd-generator, som har blitt implementert på bærbare Android-enheter. En evaluering av sanntidsytelsen til systemet og mulige optimiseringer er i tillegg presentert. Implementasjonen av Dopplerspektrum-generatoren ble sammenlignet med en Matlab-implementasjon, mens Dopplerlyden ble testet med et syntetisk test-signal. Det er en frekvenslekasje mellom lydkanalene som ligger ca. $20dB$ under det ønskede frekvensinnholdet.

En høy-ytelses Android tablet hadde høye bilderater for Fast Fourier Transformasjonslengder opp til 128 med både 75 og 87.5 prosent overlapp og $8 - 16kHz$ lydfrekvenser. Lavytelses-tableten hadde generelt lave bilderater for alle instillinger.

Ved å optimisere Fast Fourier Transformasjonen ved å bruke FFTW-biblioteket, ble det vist at høye bilderater er oppnåelig på lavytelses-tableten. Andre mulige optimiseringer for å øke bilderaten er å skru av lydgenereringen, skru av stereo-genereringen og/eller redusere instillingene på spektrumgeneratoren.

Contents

List of Figures

List of Tables

Nomenclature

API	Application Programming Interface
CW-Doppler	Continuous Wave Doppler
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPS	Frames Per Second
GPL	General Public License
NDK	Native Development Kit
PRF	Pulse Repetition Frequency
PW-Doppler	Pulsed Wave Doppler
RF	Radio Frequency
SIMD	Single Instruction Multiple Data
SNR	Signal to Noise Ratio
VES	VTK OpenGL ES
VTK	Visualization ToolKit

1 Introduction

The thesis presents the realtime implementation of the processing of Doppler signals with live spectrum display and audio on Android platforms.

1.1 Motivation

Reducing the mortality rate among unborns, young children and pregnant women is one of the Millennium Developments Goals (MDG 4 and 5) of the United Nations (UN). Diagnostic ultrasound is the only imaging method used in pregnancy and which is widely offered to the general population in developed countries. The Umoja-project, ultrasound for midwives in rural areas, aims to develop an extremely low cost, robust and portable ultrasound imaging system (the Umoja ultrasound system) for obstetric imaging, spesifically designed for operation in challenging rural areas of developing countries. The project is a joint effor between three main partners: the National Center for Fetal Medicine (NCFM) at St. Olavs Hospital and NTNU, the Department of Circulation and Medical Imaging (ISB) at NTNU, and GE Vingmed Ultrasound AS.

When building a low cost system, there is an inherent trade-off between the Doppler imaging quality and the overall cost of the portable system. The hardware limitations that come with the low cost requirements will limit the available computational power. Therefore it is desirable to implement and execute as much of the signal processing pipeline as possible directly on the tablet device, which includes both CPU and GPU units at low cost[1].

1.2 State of research

A similar implementation as the Doppler spectrum generator has been reported by a research team in 2012 [3]. Their implementation featured a custom made PW-Doppler system and direct transfer of IQ-data through the audio-jack. It is unclear from the paper how well the system performed, and if it was good enough for real-time usage. Their implementation did not feature Doppler audio-playback, and I have not found any scientific papers describing Doppler audio-generation on mobile devices.

Another company called signostics[1] has launched a ultrasound-device featuring PW-Doppler capabilities which is implemented on the Windows mobile device, but I could not find any references to Doppler audio in their product.

[1] www.signosticsmedical.com

1.3 Objectives

For the further development of the Umoja-system, this thesis will focus on the implementation of a well-performing real-time Doppler spectrum generator and real-time stereo audio-generator on mobile devices running the Android operating system.

The spectrum and the audio must be validated to ascertain that the implementation produces a high-quality spectrum and good audio.

Since the processing power differs between different mobile devices, optimizing bottlenecks in the implementation is needed and as such the cost of the system can be lowered. By optimizing the Fast Fourier-Transform, low- to mid-end devices might be able to handle the workload of both the Doppler spectrum- and audio-processing.

Since the spectrum can be processed with different parameters, such as overlap percentage and FFT-size, finding the optimal setting which produces a good result and at the same time gives a good framerate for the spectrum update is crucial to enable usage of the implementation on low-cost systems.

The Doppler spectrum generator is based on my fifth-year autumn project[7], but it has been almost totally rewritten for this thesis.

1.4 Thesis outline

The theory section covers the basic theory relevant for understanding the algorithms used to implement the spectrum- and audio-generator, aswell as the optimization of the Fast Fourier-Transform.

The methods section covers the software and hardware which has been used in the thesis. It also includes a detailed description of how the spectrum and audio-generator algorithms are implemented in the Umoja-system. Lastly a rating scheme for the achievable framerates are presented.

Results are divided into six main parts. The first two parts are reserved for timing of the spectrum and audio-genration on both tablets investigated, aswell as a full-delay timing for the tablets. The third and fourth parts of the results section is the achievable framerates with the two tested Fast Fourier-Transform algorithms, and these values are derived from the timing results in the first two parts of the results section. Part five of the results is the validation of the spectrum and audio, and lastly part six depics different smoothing methods and the spectrum quality with different generation-parameters.

After the results section, there is a discussion which is based on the results; before conclusions are drawn in the last section.

A final section, named recommendations, is added to the end. This section is a list of the most important improvements which can be made to the algorithms and the system concerning both spectrum and audio-generation.

2 Theory

The theory part is split into three main parts: The first part explains basic ultrasound principles and other principles needed to understand the second and third parts. The second part is dedicated to the generation of the Doppler spectrum in ultrasound, while the third part explains how stereo audio is generated from raw ultrasound data.

2.1 Basic Ultrasound

2.1.1 Basics

Ultrasound is based on the propagation, reflection and scattering of high frequency sound waves in a scattering medium. These high frequency sound waves are produced by an apparatus called a transducer, which in modern ultrasound systems consists of several pieozoelectric crystals which expand when a voltage is applied to them. These piezoelectric crystals vibrate, by applying an oscillating current, to create the ultrasonic waves which then again travel through the interrogated medium and return to the transducer again. The piezoelectric crystals will also produce an electric current when presure is applied to them, and this electric signal which is generated can be further analyzed to produce an ultrasound image [21].

When several such crystals are composed to form a surface, the voltage to each crystal can be controlled individually and the ability to create a focal point at which each wave from each transducer-element constructively interfere can be created. These focal points can both be controlled in the azimuthal- and the longitudinal -direction depending on how the elements are arranged and how the delays to each element is are applied. A general grayscale 2D image or B-mode image can be created by emitting ultrasonic waves towards several focal points and controlling the focal point on recieve by the means of dynamic focusing. Dynamic focusing is achieved by controlling the delay of the current from each element.

Other modes used in ultrasound are e.g. ColorFlow imaging and spectral analysis, which both are dependant on Pulsed-Wave ultrasound discussed in section 2.2.1.

2.1.2 Doppler effect

The Doppler shift can be defined for a source transmitting at a frequency f_0 towards a target approaching the source with radial velocity v_t [11]. The target will percieve the incoming signal f_t to be shifted so that

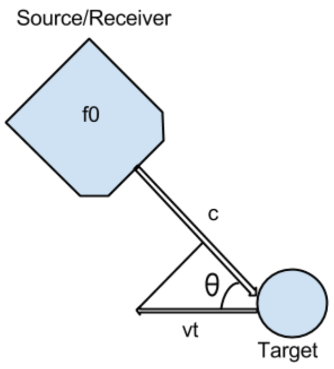


Figure 2.1: Doppler shift adjusted for the angle between the target and the source/receiver.

$$f_t = f_0 \left(\frac{v_t + c}{c} \right) \tag{2.1}$$

where c is the speed of sound in the given medium. When the reflected pulse returns to the transmitter, the system will percieve the recieved frequency f_e to be

$$f_e = f_t \left(\frac{c}{c - v_t} \right) \tag{2.2}$$

The Doppler/frequency-shift f_d is the difference between the transmitted and shifted recieved signal frequency

$$f_d = f_e - f_0 = f_0 \left[\frac{c + v_t}{c - v_t} - 1 \right] \approx 2 f_0 \frac{v_t}{c} \tag{2.3}$$

The last approximation assumes that the target's velocity v_t is much smaller than the sound speed in the medium: $v_t \ll c$.

The Doppler shift can also be adjusted for a target that does not move directly towards or from the source by multiplying equation 2.3 with the cosine of the angle between the source and the target [21]

$$f_d = 2 f_0 \frac{v_t cos\theta}{c} \tag{2.4}$$

illustrated in figure 2.1.

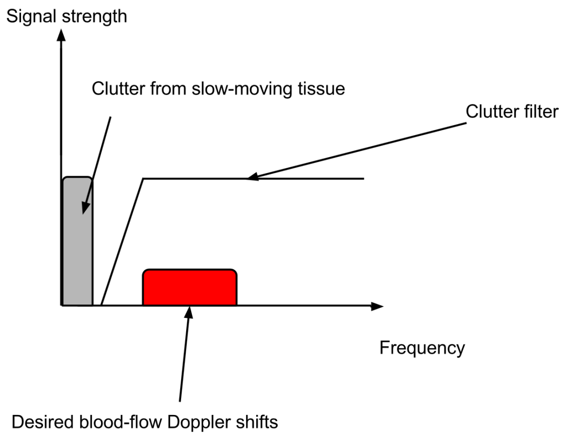

Signal strength
Clutter from slow-moving tissue
Clutter filter
Frequency
Desired blood-flow Doppler shifts

Figure 2.2: Ideal clutter filtering.

2.1.3 Clutter filter

When using ultrasound to measure the Doppler shift of moving blood inside veins or arteries, there will also be strong reflections from slow-moving vessel walls and surrounding tissue. Since the measured Doppler shift also contains frequencies from slow moving tissue, as depicted in figure 2.2 we need to high-pass filter the recieved Doppler-signal to separate the clutter signal from the moving blood cells [17].

Figure 2.2 depicts an ideal situation, where the clutter- and blood-signals are separated. This is never the case, as the signal from blood and tissue will overlap and we have a situation more like the one in figure 2.3. Without sufficient clutter rejection, the low velocity blood flow cannot be estimated and the high velocity blood flow will have a high bias.

2.1.4 Spectrum

The Doppler power spectrum is used to visualize motion of patricles within an area of interest in a scattering medium. For ultrasound this constitutes visualization of blood-flow within vessels or the heart [21]. There are mainly two ways of aqcuiring the Doppler signal from moving particles in ultrasonics; PW-Doppler and CW-Doppler. Only PW-Doppler will be discussed here, as it is the only method used in this thesis.

PW-Doppler is based on the Doppler-effect explained in 2.1.2. It requires a

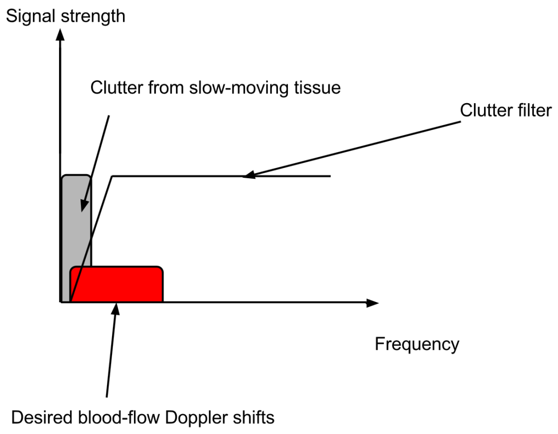

Signal strength
Clutter from slow-moving tissue
Clutter filter
Frequency
Desired blood-flow Doppler shifts

Figure 2.3: Clutter filtering.

transducer with the ability to fire short ultrasound pulses at a rapid rate, called the Pulse Repetition Frequency (PRF). The pulses travels through the interrogated medium before returning/scattering back to the system which in turn samples the pulse in a short time-span to obtain a range cell of interest. This method of sampling enables the user to select an area of interest to measure blood-flow. By deploying the Doppler spectrum generation method in section 2.2, a frequency-time periodogram of the blood-flow can be visualized as in figure 2.4.

The maximum measurable Doppler-shift f_{dmax} is evaluated as

$$f_{dmax} = 2f_0 \frac{v_{max} cos\theta}{c} \tag{2.5}$$

The maximum PRF is limited by the range of interest in the following manner, where the factor 2 in the denominator results from the fact that the pulse has to travel to- and from an object.

$$PRF_{max} = \frac{c}{2r_{max}} \tag{2.6}$$

Due to the limitations of the Nyquist-Shannon theorem [21], the maximum measurable Doppler-shift is also dependent on the PRF by the following equation

$$f_{dmax} = \frac{PRF_{max}}{2} \tag{2.7}$$

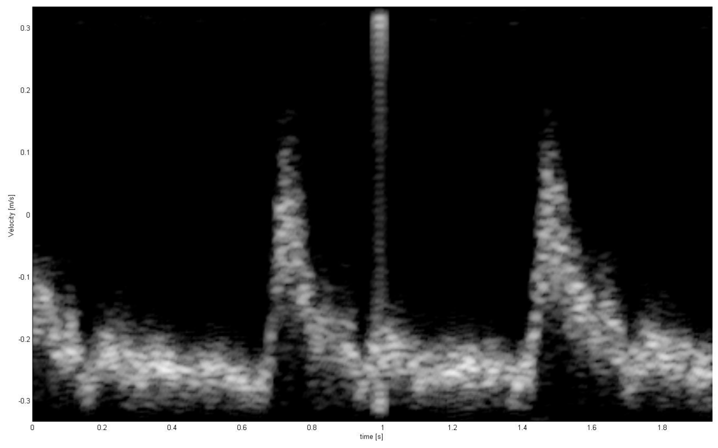


Figure 2.4: A Doppler spectrum generated from a carotid artery.

which results in, by inserting equation 2.7 into equation 2.5

$$\frac{PRF_{max}}{2} = 2f_0 \frac{v_{max} cos\theta}{c} \Rightarrow v_{max} = \frac{PRF_{max} c}{4 f_0 cos\theta} \tag{2.8}$$

2.1.5 Time-sharing

Modern ultrasound systems have real-time B-mode images, colorflow and PW-Doppler spectral measurements at the same time, called triplex-mode. Since the modalities needs to share time to use the transducer, the Doppler spectrum will have synthetic spectral signal in the time-slots where the greyscale- and colorflow-image is updated [16]. This causes the spectrum to be slightly obscured when triplex-mode is active. Deactivating the B-mode and colorflow overlay will allow the PW-Doppler spectrum modality to update the spectrum without filling the gaps with synthetic signals.

2.2 Doppler spectrum generation

2.2.1 Power spectrum

The Doppler power spectrum is generated from a PW-Doppler capable system shown in figure 2.5 which emits short pulses of ultrasound [21] and

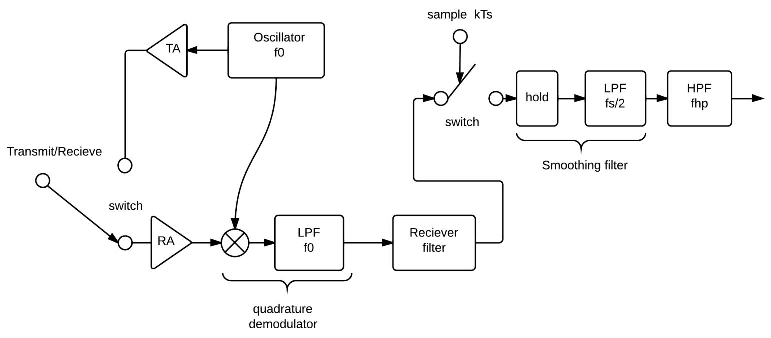


Figure 2.5: The operation of a PW-Doppler system. Taken from [21]

samples the return signal at an adjustable time-delay after emmission. This enables us to choose a range gate of interest along the ultrasound beam.

The system emits a pulse with a center frequency f_0 with bandwith B, at a rate of PRF Hz. The pulse travels through the interrogated medium before returning to the system again which samples the pulse at at least twice the highest frequency of the bandwidth of the emitted pulse after quadrature demodulation.

Before sampling, the signal is amplified by a Reciever Amplifier (RA) and demodulated down to baseband by a quadrature demodulator, which also produces the real and imaginary parts of the complex envelope of the incoming RF-signal. The demodulator also removes high frequency products of the mixing by applying a low pass filter with about 100kHz bandwidth. The pulse is then filtered by a Reciever filter (matched filter) which maximizes the SNR.

After sampling, the signal goes through a smoothing filter to remove transients from the sampling before a clutter filter (see section 2.1.3) is applied to remove signal from slow-moving tissue [21]. The resulting signal is hereby referred to as "IQ-data", a term which is used for spectrum and audio processing further on.

2.2.2 FFT Butterfly algorithm

The Fast Fourier Transform (FFT) is an algorithm to compute the Discrete Fourier Transform (DFT) and its inverse. The DFT of a complex discrete sequence $x_0,...,x_{N-1}$ is calculated by the following formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{(-i2\pi k \frac{n}{N})} \qquad k = 0,...,N-1 \tag{2.9}$$

which has a complexity of $O(n^2)$. Two well known versions of the FFT is the Cooley-Tukey radix-2 algorithm and mixed-radix algorithm. The radix-2 algorithm [6] or the "Butterfly"-algorithm splits an ordinary $N = 2^p$ point DFT into two $\frac{N}{2}$ point DFTs, one which calculates the even numbered indices x_{2m} and one for the odd numbered indices x_{2m+1} with $m = 0,...,N/2-1$ as shown in figure 2.6. The algorithm is only applicable to radix-2 size inputs. Splitting the DFT into a sum of the even numbered indices and odd numbered indices yields

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk} = E_k + e^{-\frac{2\pi i}{N}k} O_k \tag{2.10}$$

where $e^{-\frac{2\pi i}{N}k} = W_N^k$ is the so-called twiddle factor. Due to the periodicity of the DFT and the exponential, we know that

$$E_{k+\frac{N}{2}} = E_k \tag{2.11}$$

and

$$O_{k+\frac{N}{2}} = O_k \tag{2.12}$$

and

$$e^{-\frac{2\pi i}{N}(k+N/2)} = -e^{-\frac{2\pi ik}{N}} \tag{2.13}$$

This periodicity allows us to calculate the complete set of outputs for $0 \leq k \leq \frac{N}{2}$ as

$$X_k = E_k + W_N^k O_k \tag{2.14}$$

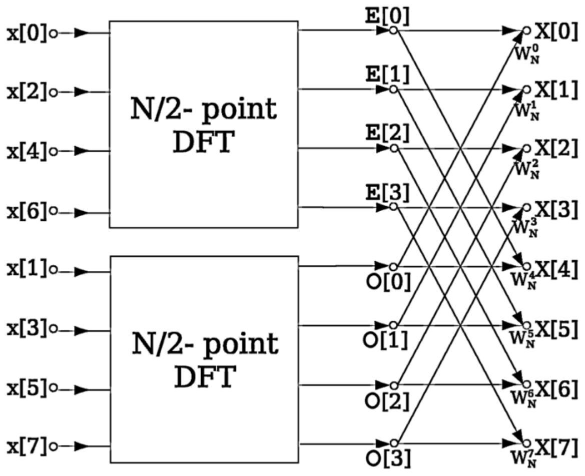


Figure 2.6: Radix-2 Cooley-Tukey fast Fourier transform. $x[n]$ denotes the complex input, $X[k]$ is the complex output. $E[k]$ is the even indiced DFT of the input and $O[k]$ is the odd indiced DFT of the input. W_N^k are the so-called twiddle-factors. The image is retrieved from [4]

$$X_{k+\frac{N}{2}} = E_k - W_N^k O_k \tag{2.15}$$

Equations 2.14 and 2.15 are the butterflies of the FFT. This method of computing the complex DFT reduces the complexity to $O(Nlog_2N)$ which is a substantial reduction on the number of arithmetic operations compared to the standard DFT algorithm. The Cooley-Tukey algorithm can also be generalized to re-express a DFT of a composite size $N = N_1N_2$, also called mixed-radix cases.

2.2.3 FFT - Power spectrum

The power spectrum is visualized on a device as in figure 2.4 by transforming the M-point IQ-data to the frequency domain by the means of a N-point FFT and logarithmically compressing the resulting data for visualization purposes. A diagram of the processing is shown i figure 2.7.

The input IQ-data is divided into overlapping segments to increase the time-resolution and reduce the modified periodogram's (a periodogram which has been windowed by e.g. a Hamming window) variance [22]. The modified periodogram is averaged after taking the absolute value and squaring the

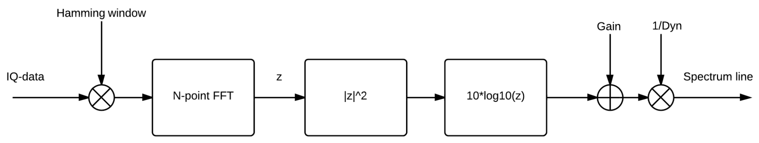


Figure 2.7: Spectrum processing flowchart.

result, that is before the logarithmic compression, to decrease the standard deviation as much as possible.

2.3 Doppler audio generation

The IQ-data which is used for spectral processing can also be used to generate audio, which also can be split into two channels; one for positive Doppler-shifts and one for negative shifts.

The Doppler audio is generated in three steps:

1. Interpolate the signal from the sampling frequency (PRF) to an appropriate audio-playback frequency F_s.
2. Filter the interpolated signal to remove transients.
3. Filter the audio to split the positive and negative frequencies into two channels.

The steps are described in further detail in the rest of this section.

2.3.1 Resampling by interpolation

Linear interpolation is a method of curve fitting between two points. It is defined by

$$\frac{y-y_0}{x-x_0} = \frac{y_1-y_0}{x_1-x_0} \tag{2.16}$$

where (x_0, x_1) are the known values of x and (y_0, y_1) are the know values of y. x and y is the unknown point of interest [5]. Equation 2.16 can be rewritten to solve for y

$$y = y_0 + (y_1 - y_0)\,\frac{x - x_0}{x_1 - x_0} \tag{2.17}$$

which can be represented graphically as in figure 2.8. By defining the original step size $x_1 - x_0$ of the original sampling frequency (PRF) as 1, the new step sizes for an arbitrary resampling frequency F_s can be defined as

$$step = \frac{PRF}{F_s} \tag{2.18}$$

which is the new relative distance between each interpolated point.

If we let $X(f)$ be the Fourier transform of any function, $x(t)$ which is sampled at some interval T to produce the sequence $x\,[n]$. The discrete-time Fourier transform of the sequence $x\,[n]$ is defined as [14]

$$\sum_{n=-\infty}^{\infty} x\,(nT)\,e^{-i2\pi f nT} = \frac{1}{T}\sum_{k=-\infty}^{\infty} X\,(f - {}^{k}\!/\!{}_{T}) \tag{2.19}$$

where the discrete-time Fourier transform is rewritten as a periodic summation of $X(f)$. Here the sampling interval $T = \frac{1}{PRF}$. If we now resample at a frequency $F_s = \frac{1}{L}$, we increase the periodicity by a factor of L

$$\frac{L}{T}\sum_{k=-\infty}^{\infty} X\left(f - k\frac{L}{T}\right) = \frac{PRF}{F_s}\sum_{k=-\infty}^{\infty} X\left(f - k\frac{PRF}{F_s}\right) \tag{2.20}$$

which implies that if we resample a signal, we will have replicas of the spectrum $X(f)$ around multiples of the old sampling frequency PRF as shown in figure 2.9.

2.3.2 Adaptive low-pass filter, Parks-McClellan filter coefficients

To filter the unwanted replicas of the signal spectra, a lowpass-filter with cutoff-frequency at $\frac{PRF}{F_s}$ needs to be applied to the interpolated signal. This is since we sample the IQ-data at a rate of PRF and the new sampling frequency after interpolation is F_s. Since the PRF can vary relatively arbitrarily, the filter needs to be adaptive as shown in figure 2.10. The filter design is based on the Parks-McClellan algorithm for finding optimal Chebyshev FIR filter coefficients [20].

2.3.3 Split filter

To split the positive and negative frequencies of the audio signals, two split-filters can be applied to the lowpass-filtered and interpolated audio signal.

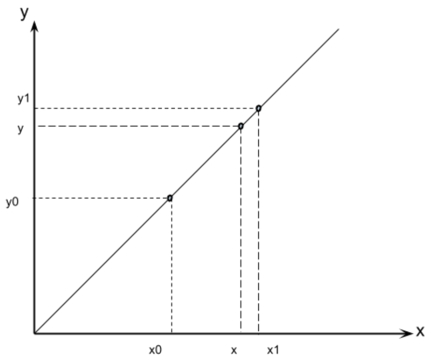


Figure 2.8: Linear interpolation scheme

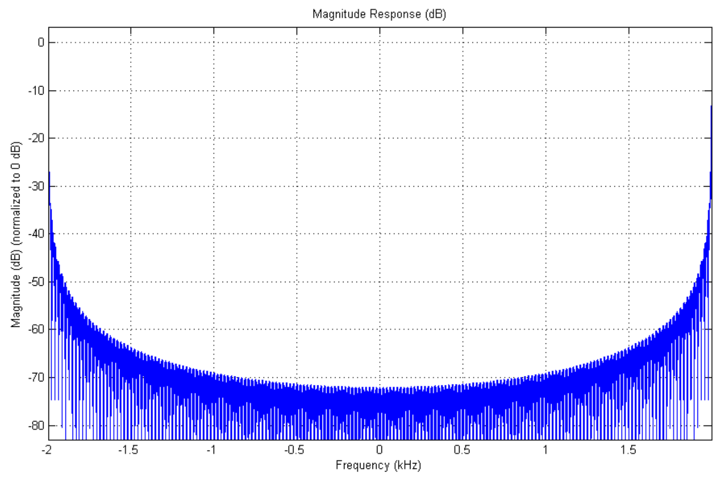
The split filter is first designed as a lowpass-filter with a cut-off frequency of $\frac{F_s/2}{2}$,as illustrated in figure 2.11 , before each frequency component is shifted by $\frac{\pi}{2}$ to get a band-pass filter from $0 \rightarrow \frac{F_s}{2}$ by multiplying each filter coefficient $h\,[0...N-1]$with a complex exponential

$$h_{shifted}\,[n] = h\,[n]\exp\left(i\frac{\pi}{2}n\right) \qquad n = 0...N-1 \tag{2.21}$$

which gives us the split filter, for the retrieval of positive frequencies, illustrated as the top figure in figure 2.12. The bottom split filter, which retrieves negative frequencies, is attained by complex conjugating the split-filter coefficients

$$h_{-shifted}\,[n] = h^*_{shifted}\,[n] \qquad n = 0...N-1 \tag{2.22}$$

where $*$ denotes complex the complex conjugate.



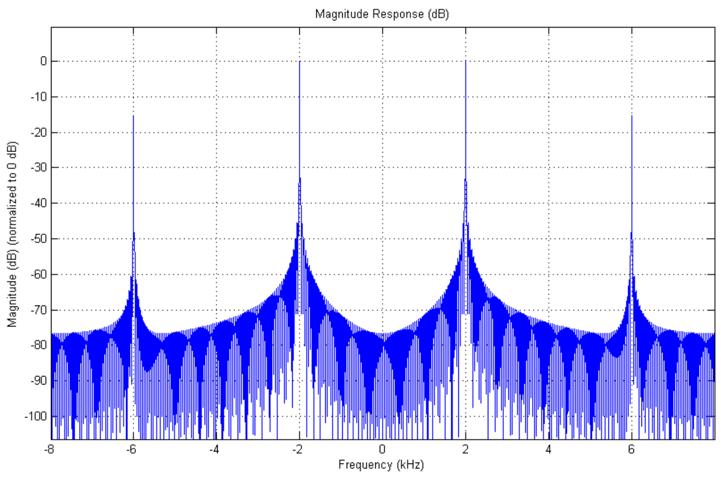


Figure 2.9: Effects of interpolation. The top figure is a pure sine-wave at the Nyquist frequency for a sampling frequency of $4kHz$. The bottom figure is the same spectrum after linear interpolation to a new sampling frequency of $16kHz$.

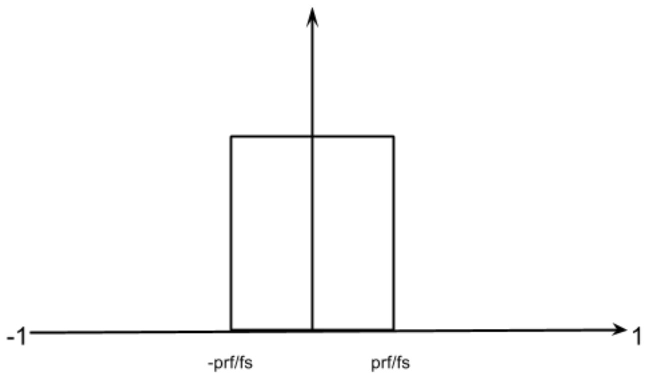


Figure 2.10: Ideal low pass filter to filter transients caused by interpolation.

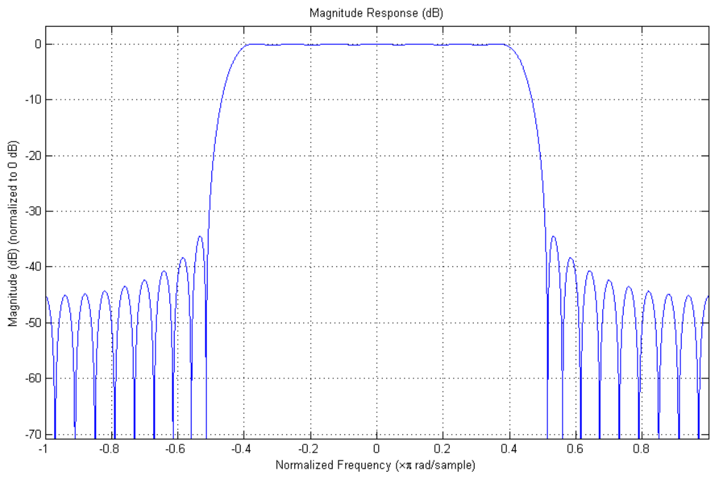


Figure 2.11: Split filter before up-shifting

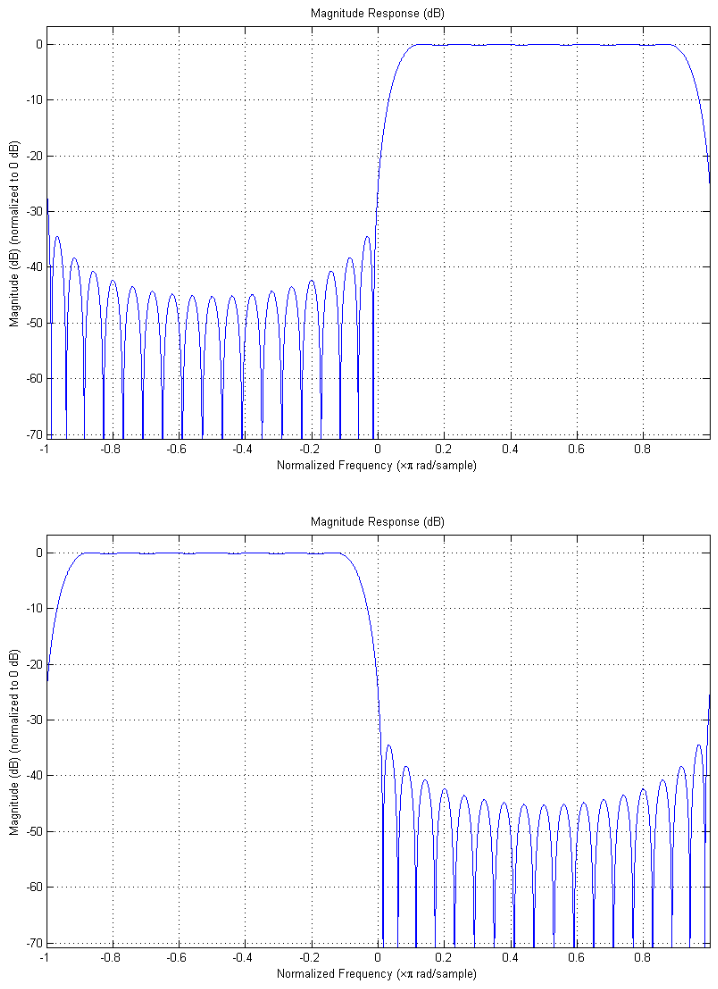


Figure 2.12: Split filter and its conjugate. These two filters split the positive from the negative frequencies.

3 Methods

The methods part is divided into four parts needed to understand the implementation of the algorithms used for real-time spectrum- and audio-generation and the real-time requirements of the system. The first part is a brief overview of the software used to implement the algorithms on the tablet, while the second part briefly describes the hardware used for data-acquisition and implementation. The third part explains in detail how the spectrum- and audio-generation is implemented on the hardware, while the last part is dedicated to a brief overview of the real-time requirements of the system, and how the framerate for the spectrum generation is rated.

3.1 Software

The thesis has relied on a number of software components which enables the live streaming and processing of IQ-data from the ultrasound-scanner (Vivid-I/Q), and other libraries for visualization, interaction and audio playback on an Android device.

VTK

VTK is an open source software system for computer graphics, modeling and image processing [12]. It is implemented as a C++ toolkit, enabling users to create complex visualizations of models or datasets. The toolkit is created and continually extended by *Kitware* and it is cross-platform as it runs on Linux, Windows, Mac and mobile platforms; such as Android or IOS.

VES

VES is the VTK OpenGL ES Rendering toolkit created by *Kitware* enabling users to render OpengGL ES 2.0 applications on mobile devices.

Kiwi

Kiwi ties VTK and VES together to enable users to write applications for mobile devices utilizing both the VTK and VES libraries, depicted in figure 3.1. Kiwi also enables multitouch interaction with Android or IOS and is created by *Kitware.*

OpenSL ES

OpenSL ES is a cross-platform audio API that is created for embedded systems. It enables programmers to write audio-applications for embedded

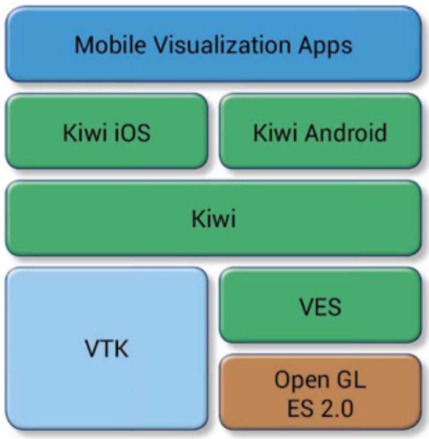


Figure 3.1: Software structure layout for VTK, VES and Kiwi layout. The picture is taken from [8].

devices such as smartphones.

Android NDK

The Android NDK, or Native Development Kit, is a toolset that allows developers to write applications for Android using native-code such as C or C++.

Data streaming

The GEStreamer code enables live-streaming of raw data from the Vivid-I/Q. This streaming client is developed at the Department of Circulation and Medical Imaging which is a part of the Medical Faculty at NTNU. The streaming client has the capability to establish a connection to the ultrasound scanner and recieve data over IP and convert and sort it into a VTK format; data which is accessible through functions provided by the streaming client libraries. The streaming client also enables sending commands to the ultrasound scanner via IP for remote control of modality, PRF, gate selection etc.

Figure 3.2: The GE Vivid-Q laptop ultrasound scanner. The image is retrieved from http://www3.gehealthcare.com/

3.2 Hardware

Vivid-Q

The Vivid-Q is a high performance laptop ultrasound scanner which can run on internal battery as an ordinary laptop. It has a programmable system architecture and is the source of the raw-data used in the thesis. The GEStreamer code streams live ultrasound data from the Vivid-Q via an ethernet cable connected to the scanners back side. The Vivid-Q also recieves commands from the GEStreamer code via the same ethernet cable.

Google Nexus 10

The Google Nexus 10 is used as the high-performance reference and it is a ten-inch tablet with a 2560-by-1600 pixel display produced by Samsung. It is powered by a dual-core ARM Cortex-A15 CPU operating at 1.0GHz to 2.5GHz and a Quad Core ARM Mali TS04 GPU supporting OpenGL ES 1.1-3.0.

ASUS Transformer Pad T300

The ASUS T300 is used as a low-medium performance reference and it is a ten-inch tablet with a 1280-by-800 pixel display produced by ASUS. It is powered by a quad-core NVIDIA Tegra 3 T30L CPU operating at 1.2GHz and a GeForce 12-core GPU supporting OpenGL ES 2.0 but not OpenCL.

Samsung Galaxy S4 Active

The Samsung Galaxy S4 Active is used as the smartphone test-device and it is a five-inch smartphone with a 1080-by-1920 pixel display produced by Samsung. It is powered by a quad-core ARM Krait 300 CPU operating at 1.9GHz and a Quallcomm Adreno 320 quad-core GPU supporting OpenGL ES 3.0.

3.3 Implementation

The data from the ultrasound scanner comes in packets which contain timestamped *IQ*-data. These data-packets are parsed in the function *AddDopplerData* in the class *vtkIQStreamData* where they are fed into the spectrum-generator and audio-generator packet-by-packet as shown in figure 3.3.

vtkIQStreamData feeds the two generation classes with *IQ*-data . It enables synchronization between the spectrum and audio since they recieve the same *IQ*-packet at the same time, as well as easy access to change parameters and access data within the two sub-classes: *vtkDopplerGenerateSpectrum* and *vtkDopplerGenerateAudio*. All aspects of my implementation are written in C++ and run in real-time on Android. All developments have been tested on a Google Nexus10, an ASUS T300 and a Samsung Galaxy Nexus 4 Active.

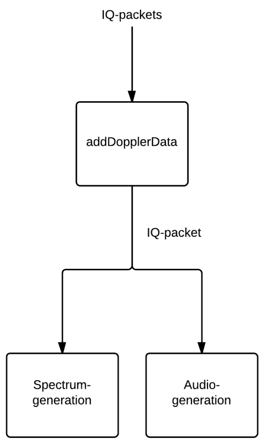


Figure 3.3: IQ-data processing flowchart. The source code for the flowchart is given in appendix C.

3.3.1 Doppler spectrum

The class *vtkDopplerGenerateSpectrum* is responsible for generating spectrum lines and new timestamps given a packet of IQ-data as shown in figure 3.4. The spectrum-line data and new timestamps are stored in a linear buffer and can be retrieved through the *vtkIQStreamData*-class. The spectrum lines are generated with an overlap between IQ-data to increase the time-resolution.

Firstly the IQ-data packet is pushed into the back of a linear buffer in *vtkIQStreamdata*, which can contain at most a given length in time of IQ-samples. The IQ-data is then parsed to *vtkDopplerGenerateSpectrum* by a function-call to $ProcessIQPacket$.

A set of M samples is chosen from the last overlap position n if the sum of them do not exceed the number of IQ-samples available kN, where k is the packet number and N is the number of IQ-samples in this packet. If there are enough samples available we move further down the chain, if not; a new packet is added if there are more packets left this run. The overlapping is shown in figure 3.5.

The next step is applying a Hamming window described in section 2.2.3 to reduce side-lobe levels and spectral leakage between overlapping sections of IQ-data. Zeropadding to a length L is performed on the temporary M -point IQ-set if $M < L$, before the L-point FFT is excecuted. If the data window size is larger than the preset FFT-size, then the FFT-size is set to the window-size. After applying the FFT, we need to shift the zero frequency to the middle of the spectrum line for visualization purposes. This results in a spectrum where the zerofrequency-line is in the middle of the image, the positive frequencies are represented at the top half of the image and the negative frequencies at the lower part.

The Fourier-transformed IQ-data is then processed further on by finding the power, then the data is put through a moving average filter of a given amount of miliseconds to reduce the variance of the estimated spectrum, if we have enough spectrum lines. Finally logarithmic compression is applied to scale and compress the data to intensity values from $0 \rightarrow 255$.

After this processing is done, a timestamp is generated for the produced spectrumline and the spectrumline is saved to a linear buffer. The start position of the window is incremented by s miliseconds and another round of processing commences.

Both VTKs built-in FFT algorithm and FFTW was implemented in the dataStreamClient, but only VTKs built-in FFT was implemented on the Android devices.

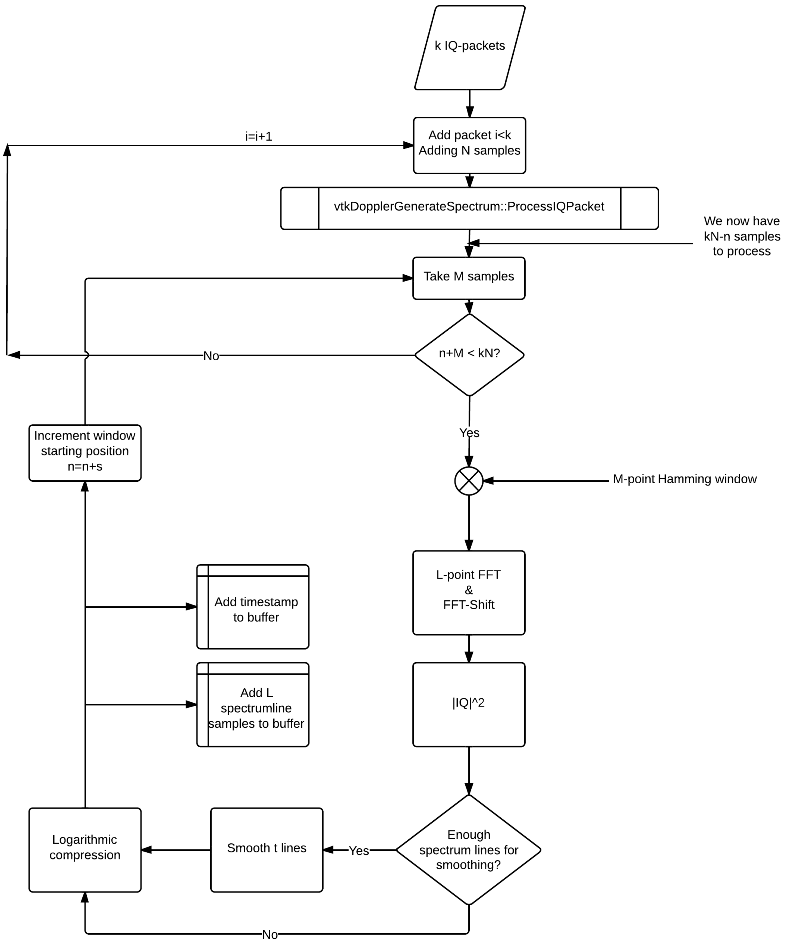

k IQ-packets
Add packet i<k Adding N samples
i=i+1
vtkDopplerGenerateSpectrum::ProcessIQPacket
We now have kN-n samples to process
Take M samples
n+M < kN?
No
Yes
Increment window starting position n=n+s
M-point Hamming window
L-point FFT & FFT-Shift
Add timestamp to buffer
|IQ|^2
Add L spectrumline samples to buffer
Enough spectrum lines for smoothing?
Yes
Smooth t lines
Logarithmic compression
No

Figure 3.4: Doppler spectrum processing flowchart. The spectrum generation code is given in appendix D.

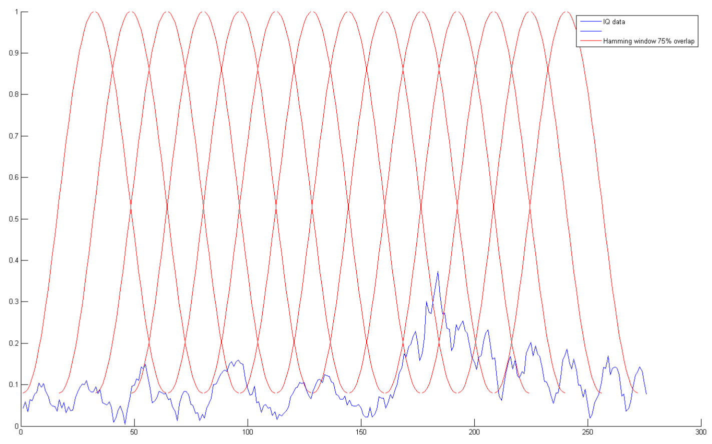


Figure 3.5: Overlapping the IQ-data with a Hamming window, 75% overlap.

3.3.2 FFT optimization

VTK FFT

The VTK FFT-implementation has been analyzed manually by analyzing the source-files of VTK's implementation. This may cause the conclusion to what kind of implementation VTK uses to be incorrect, and the reader is advised as such.

VTK seems to implement a version of Rader's FFT algorithm [18] alongside with a radix-2 FFT. It splits the input length into prime factors and uses Rader's FFT algorithm for prime factors other than 2, for which it uses the standard butterfly radix-2 FFT method. This means that for radix-2 FFT-sizes VTK will use radix-2 FFT to process all the data for a complexity of $O(Nlog_2N)$while its complexity will approach $O(N^2)$ for prime factors approaching the FFT-size, or for FFT-sizes which are primes.

FFTW

FFTW is a C subroutine-library implementation of the discrete Fourier transform that adapts to the hardware in order to maximize performance[9]. FFTW uses a *planner* which searches for the fastest solution of the DFT of a given DFT-length for the spesific harware. It searches through different FFT-algorithms for the given length and times the algorithms to decide which is the fastest. This *plan* for a given FFT-length can be saved for later use on the same machine, and it is called *wisdom*. FFTW creates *codelets*, specific lines of code designed for fast computation of a sub-problem, which are hardware specific SIMD-instructions on supported hardware. If these codelets can be written as SIMD-instructions, the performance is greatly increased.

FFTW includes settings for how wide the search for the optimal plan will be, and thus how much time will be spent searching through algorithms to find the optimal plan. Since this plan can be saved as wisdom for later use, the search only needs to be done once for each machine and then the wisdom can be loaded when it is needed. FFTW also supports multithreading.

The library is released under the GNU General Public License (GPL), but the GPL protection can be voided by purchasing a non-free license from MIT. This license is a one-time fee, which gives the user access to use the software commercially without having to publish their source-code.

3.3.3 Audio

The audio generation setup is much the same as the spectrum generation. The class *vtkDopplerGenerateAudio* also has a function called *ProcessIQPacket* which is called from *vtkIQStreamData*. *vtkIQStreamData* sends *IQ*-data in packets to the *ProcessIQPacket*-function. The packet sent to both the *vtk-DopplerGenerateSpectrum*-class and *vtkDopplerGenerateAudio*-class is the same, and it is sent at the same time, as this ensures some synchronization between the shown spectrum and the audio.

The audio generation class is responsible for generating stereo audio, one channel for positive frequencies and one for negative, and sending the resulting data at a certain rate F_s, which is the playback frequency, to an audioplayer implemented by Gabriel Kiss using OpenSL ES' AudioTrack class. *vtkDopplerGenerateAudio* contains several steps to make good audible stereo sound from the raw *IQ*-data.

First there is a check to determine if the *PRF*, sampling frequency F_s or any other important parameter has changed, or if we are at a new processing startup, e.g. when we have a mode change from ColorFlow to PW-Doppler. This check determines wether we have to design a new split-filter or not as seen in figure 3.7. If we have to design a new split-filter, then this is done and the new design is saved until a new setup is required.

The split filter is a combined lowpass-filter, as in figure 2.10 ,and split-filter, as in figure 2.12 ,which reduces the overall filtering complexity with a factor of two. So instead of lowpass-filtering and then applying the split-filter; the lowpass-filter's cut-off frequency is halved and the filter is shifted up the new cut-off frequency, as illustraded in figure 3.8. The split filter is designed using the Parks-McClellan FIR-filter design algorithm, and an external library computes the filter coefficients by a simple function call to the function *parksMcClellan* which contains the FIR-filter coefficients after the call.

```
//Designing the split filter as a lowpass-filter at prf/fs/2
ParksMcClellan parksMcClellan(AudioCoefficientsNumber,
      InterpolationIncrement/2, 0, 0.03, LPF);
```

The problem with this filter-design is that the side-lobe levels are very high, at about $-16dB$. The split-filter is multiplied by a Hann-window before shifting to reduce these sidelobe levels, and the resulting two split filters can be seen in figure 3.9. The mainlobe of the filter has widened, but the sidelobes have decreased significantly which gives us a good differentiation between positive and negative frequencies.

The frequency up-mixing, Hann-window multiplication and conjugation is performed by the following code in the program

```
//Mixing the different coefficients up, so we now have a lowpass-
    filter from 0->prf/fs;
//The complex conjugate is a lowpass-filter from -prf/fs->0.
//Also multiplying with the Hann window, to reduce sidelobe levels.
for(int i=AudioCoefficientsNumber-1; i >= 0; --i) {
        val.Real = parksMcClellan.FirCoeff[i]*cos(i*M_PI*
            InterpolationIncrement/2)*Hann[i];
        val.Imag = parksMcClellan.FirCoeff[i]*sin(i*M_PI*
            InterpolationIncrement/2)*Hann[i];
        AudioSplitFilter.push_back(val);
        val.Imag = -val.Imag;
        AudioSplitFilterConjugate.push_back(val);
}
```

The variable $InterpolationIncrement$ is $\frac{PRF}{F_S}$, the up-shifting is done by multiplying the real and imaginary parts of the filter-coefficients by the cosine and sine of the up-shifting argument respectively. The filter-coefficients are also multiplied by a Hann window and complex conjugated by changing the sign of the imaginary part of the coefficients. The filter length is set to $AudioCoefficientsNumber$. A reverse-run for-loop is used to insert the filter-coefficients in reversed order since the filtering is a convolution between the IQ-samples and the filter-coefficients.

After the initial setup of the filter, or if nothing has changed since the last run, the data is upsampled from a samplerate of PRF to F_s by the means of linear interpolation.

After the interpolation the audio-samples are filtered with the split-filter, which results in two audio-channels. One channel for the negative frequencies and one for the positive. These two channels, which are concatenated into one array (L,R,L,R), are then sent to an audioplayer which plays the generated samples in real-time. The audio-player has an adjustable buffer, which is set for $25ms$, of samples to compensate for possible interruption in the processing.

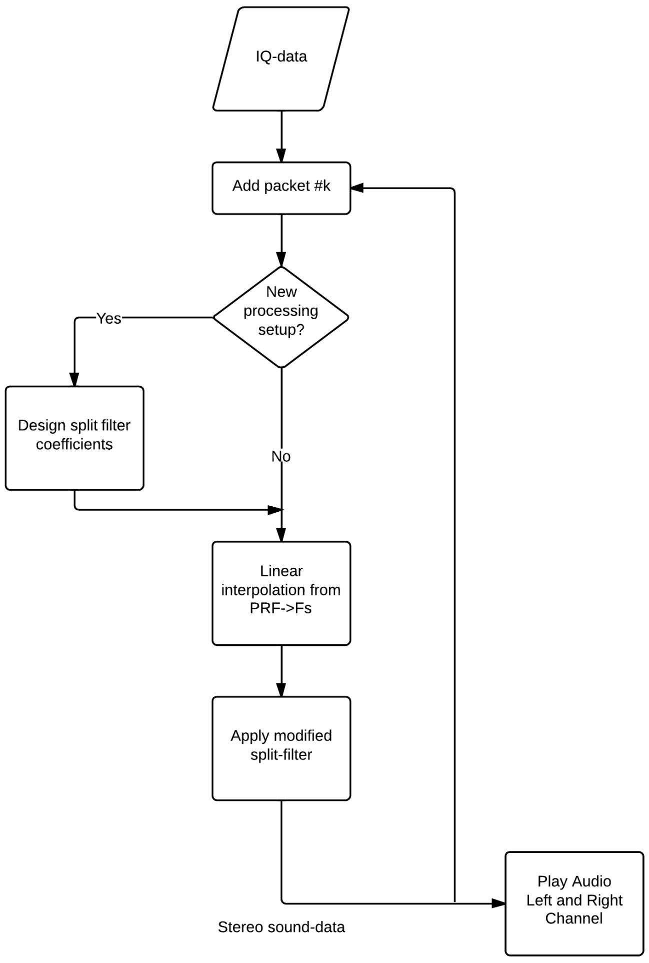

IQ-data
Add packet #k
New processing setup?
Yes
No
Design split filter coefficients
Linear interpolation from PRF->Fs
Apply modified split-filter
Stereo sound-data
Play Audio Left and Right Channel

Figure 3.6: Audio processing flowchart. The source code for the audio-processing is given in appendix E.

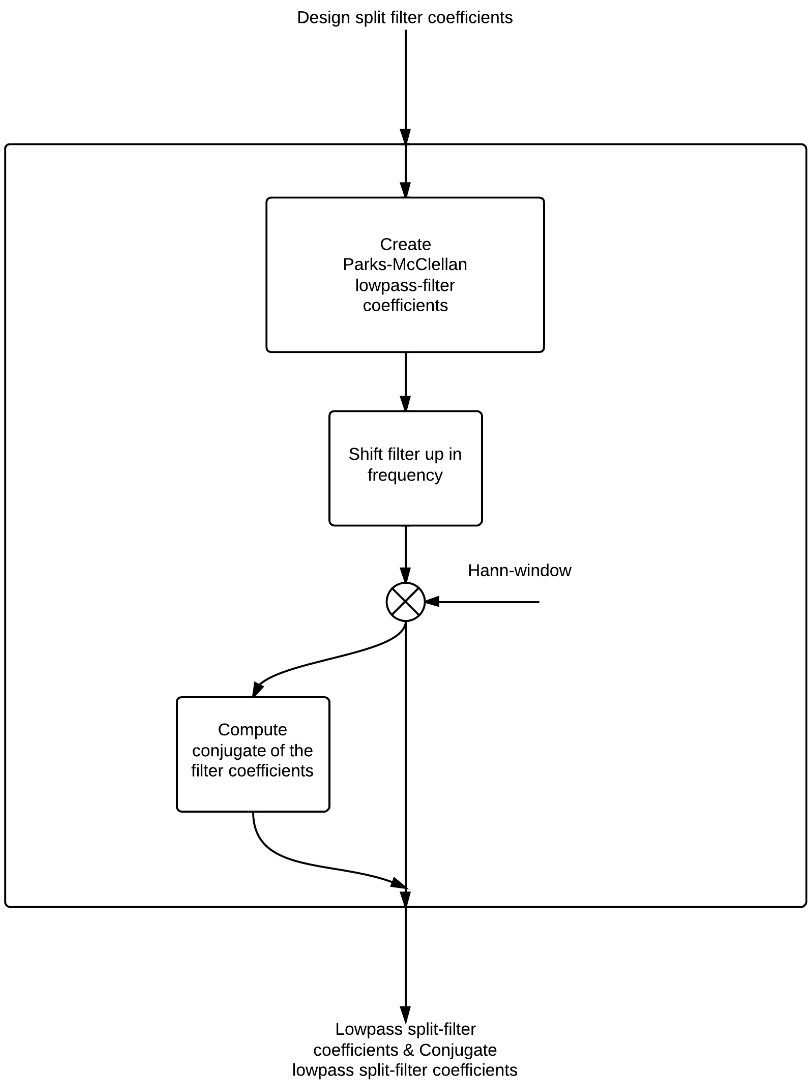
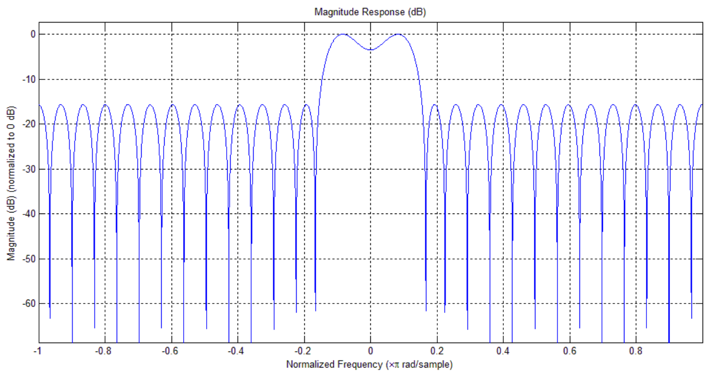

Design split filter coefficients
Create Parks-McClellan lowpass-filter coefficients
Shift filter up in frequency
Hann-window
Compute conjugate of the filter coefficients
Lowpass split-filter coefficients & Conjugate lowpass split-filter coefficients

Figure 3.7: Parks-McClellan split filter design. The filter coefficients are shifted up $\frac{PRF}{Fs/2}$.



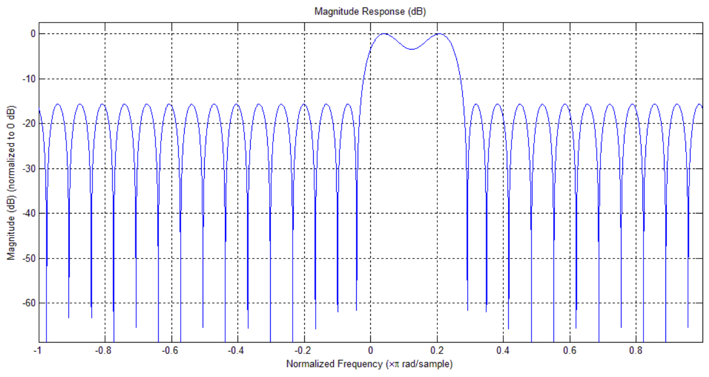
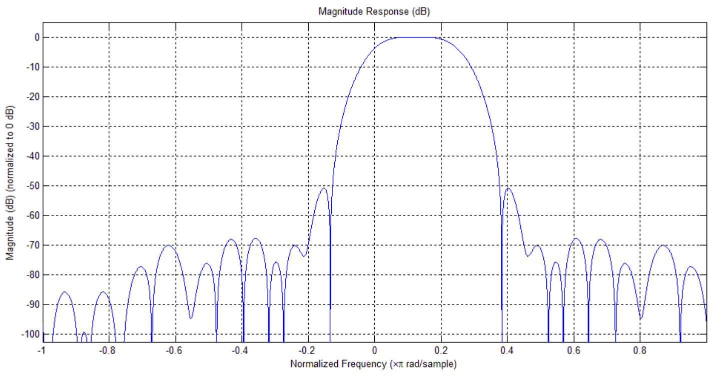


Figure 3.8: Combined adaptive lowpass- and split-filter design. The top figure is before shifting and the bottom one is after frequency-shifting. This filter was designed for a PRF of $4kHz$ and resampling frequency $F_s = 16kHz$. This should give a final cut-off frequency after shifting at $\frac{PRF}{F_s} = 0.25$ which can be verified from the bottom figure. The x-axis represents the normalized frequency $(\times\pi rad/sample)$ after interpolation.



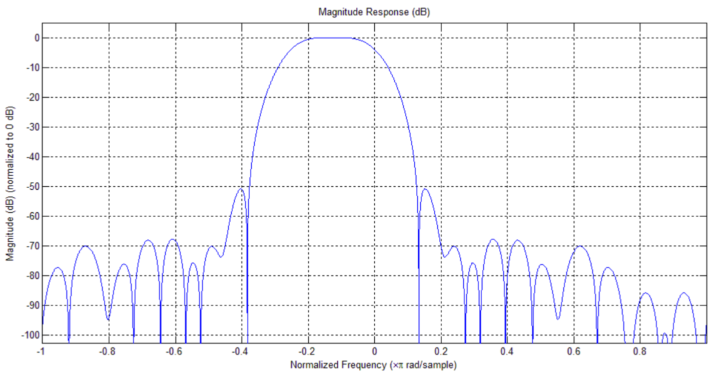


Figure 3.9: Combined adaptive lowpass- and split-filter design after Hann-Window multiplication. The mainlobe has widened, but the sidelobes have been considerably lowered. This filter was designed for a PRF of $4kHz$ and resampling frequency $F_s = 16kHz$. The x-axis represents the normalized frequency $(\times\pi rad/sample)$ after interpolation.

3.4 Real-Time requirements

A signal processing system, or any other system for that matter, is only considered real-time when it can process and output data faster, or at least as fast as the input. This means that if a system spends 5 seconds processing and outputting 4 seconds worth of data, it is not deemed real-time and the data will stack up as time progresses.

The implication is that both the spectral processing and the audio processing times combined will have to be less than the data-aqcuisition time; since they are run on the same thread. It is still a good idea to keep the processing time as low as possible, to allow for some delay in other parts of the system.

The number of frames per second which is deemed fluent is a tricky subject, since it can vary between modalities. A table which *nootebookcheck.net* uses to rate graphic cards' performance in video games, depicted in figure 3.10, *can* be a good indication of fluency, and it is used as a pseudo-objective benchmark for the spectrum generation.

The table in figure 3.10 gives a score of *fluent* when the FPS reaches about $25 \rightarrow 35$. The measured run-times for the 10 seconds of IQ-data can be converted to FPS by the following equation

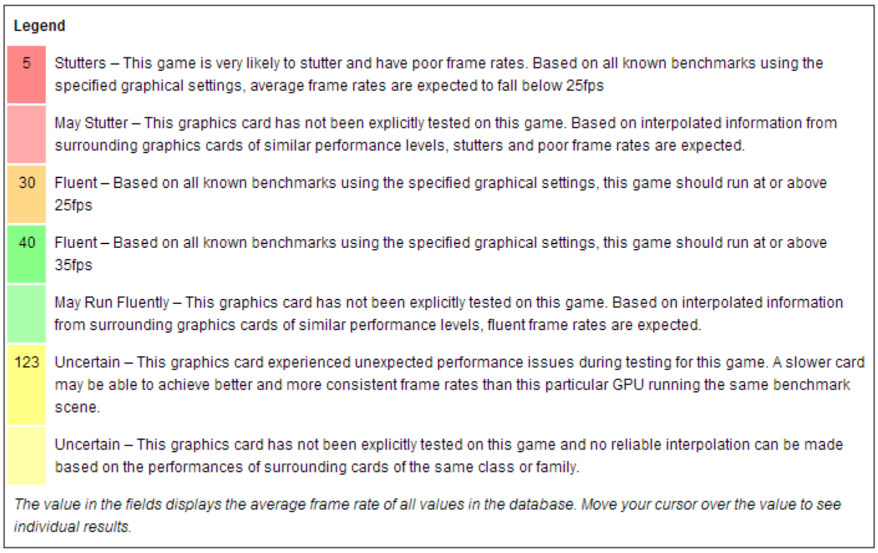
$$FPS = \frac{1}{Runtime/10} = \frac{10}{Runtime} \tag{3.1}$$

where the variable *Runtime* is the total measured run-time in seconds. This formula can also be rewritten to find the maximum run-time which achieves a FPS of i.e. 25

$$Runtime_{max} = \frac{10}{FPS} \tag{3.2}$$

which gives a maximum run-time of $\frac{10}{25} = 400ms$. To achieve 35 FPS the maximum run-time can at most be $\frac{10}{35} = 285.7ms$.

The maximum achievable FPS is also dependant on how fast the system can parse the data to the spectrum generator and refresh the screen, but this is not considered here.



Legend

5	Stutters – This game is very likely to stutter and have poor frame rates. Based on all known benchmarks using the specified graphical settings, average frame rates are expected to fall below 25fps
	May Stutter – This graphics card has not been explicitly tested on this game. Based on interpolated information from surrounding graphics cards of similar performance levels, stutters and poor frame rates are expected.
30	Fluent – Based on all known benchmarks using the specified graphical settings, this game should run at or above 25fps
40	Fluent – Based on all known benchmarks using the specified graphical settings, this game should run at or above 35fps
	May Run Fluently – This graphics card has not been explicitly tested on this game. Based on interpolated information from surrounding graphics cards of similar performance levels, fluent frame rates are expected.
123	Uncertain – This graphics card experienced unexpected performance issues during testing for this game. A slower card may be able to achieve better and more consistent frame rates than this particular GPU running the same benchmark scene.
	Uncertain – This graphics card has not been explicitly tested on this game and no reliable interpolation can be made based on the performances of surrounding cards of the same class or family.

The value in the fields displays the average frame rate of all values in the database. Move your cursor over the value to see individual results.

Figure 3.10: Fluency ratings for graphics cards from nootebookcheck.net

4 Results

4.1 Spectrum generation timing

The Doppler spectrum generation was timed on both a Google Nexus 10 3.2 and an ASUS T300 Transformer Pad 3.2. A cumulative timer for 10 seconds of IQ-data was measured and the measurements was performed and averaged 5 times to eliminate the influence of background programs and reduce the variance of the measurements. All timers were measured in real-time with the tablets connected to the Vivid-Q. The probe was given a coat of gel, but it was not in contact with anything.

Both the Nexus 10 and the T300 were measured using two different PRF's; 2890Hz and 6410Hz. These values for the PRF were chosen because they are preset PRF's used in the Umoja-prototype, and the size of the IQ-packet recieved per timepoint to the program varies with different PRF's. A higher PRF results in an IQ-packet larger than a low PRF IQ-packet.

Alongside with a full cumulative timer of the Doppler spectrum generation, a cumulative FFT-timer was also measured in the same manner as the Doppler spectrum timer to measure how much of the processing that went into executing the FFT.

All raw-data used to produce the figures in the coming sections can be found in appendix H.

Doppler spectrum generation cumulative run-times was also measured for both VTK's built in FFT-algorithm and an implementation of FFTW's implementation using FFTW's *Exhaustive* algorithm search. The cumulative run-timing was conducted in the exact same fashion as for the timers on the Android devices except that this timing was performed on a laptop computer[1]. From these timings, an anticipated new cumulative run-time for the Android devices is presented. The FFTW-measurements on the laptop was performed with SIMD-support, but no multithreading was used due to the small problem size.

[1]Dell Precision | M4700. Intel i7-3540M CPU at 3.00GHz. Nvidia Quadro K2100M GPU and 16GByte RAM. Using Visual Studio 2012 Professional debug build.

4.1.1 Nexus 10

The measurements on the Nexus 10 are given for overlapping windows of both 75% and 87.5%, as shown in figures 4.1 and 4.2 respectively. The window-length for the respective overlap percentages was set to a static $20ms$. The full run-times are displayed as the full bars in the figures while the green parts of the bars represent the FFT run-time only.

As can be seen from figure 4.1 the cumulated run-times for the Doppler spectrum generator ranged from 80 ms to 250 ms for both PRF's. There is a significant difference at a FFT-size of 200 for an overlap of 75%, where the cumulative timer for a PRF of 6410 is almost double compared to a PRF of 2890. This discrepancy is intuitive since a higher PRF would result in more processing before the FFT, but the difference is also extreme and thus the measurement of the size-200 FFT at $2890Hz$ should be disregarded.

Table 4.1 shows that the fraction of the cumulative timer for the FFT versus the whole run-time of the spectrum generation, ranges from 35.53% to 47.51%. The highest relative run-time percentages are found for a FFT-size of 100.

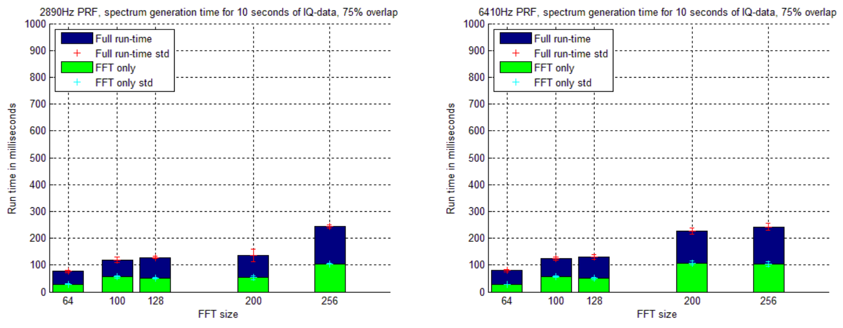


Figure 4.1: Spectrum timing for the Nexus 10 with 75% overlap. PRF of 2890 to the left and 6410 to the right. The full column represents the cumulated run-time for 10 seconds of IQ-data while the green bar represents the FFT only. The data is averaged over 5 measurements.

Figure 4.2 shows that for an overlap of 87.5% the cumulative run-times for the whole spectrum processing ranges from 130 to 460 ms, and it also shows that there is almost no difference between a PRF of 2890 and a PRF of 6410.

As can be seen from table 4.2 the fraction of the cumulative FFT run-time versus the whole spectrum processing run-time ranges from 37.58% to 48.11%. The highest ratio can be found where the FFT-size is 200.

PRF\FFT size	64	100	128	200	256
2890Hz	38.00%	46.95%	39.76%	39.01%	43.08%
6410Hz	35.53%	46.52%	38.55%	47.51%	42.40%

Table 4.1: FFT vs full run-time fractions with 75% overlap on the Google Nexus 10. The table represents how much of the processing is spent performing the FFT versus the full run-time of the spectrum generation.

PRF\FFT size	64	100	128	200	256
2890Hz	38.51%	47.28%	40.34%	48.08%	42.05%
6410Hz	37.58%	47.11%	39.38%	48.11%	41.68%

Table 4.2: FFT vs full run-time fractions with 75% overlap on the Google Nexus 10. The table represents how much of the processing is spent performing the FFT versus the full run-time of the spectrum generation.

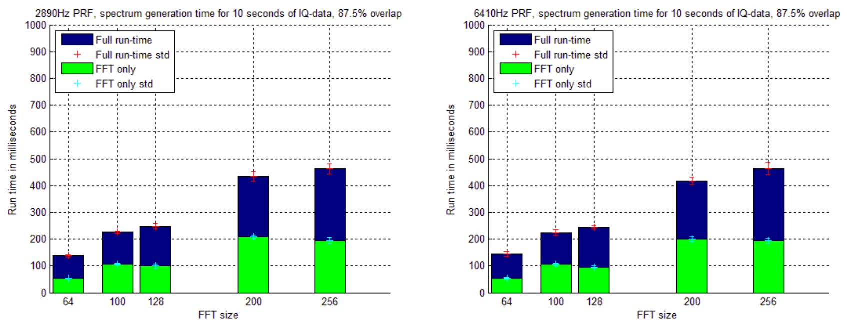


Figure 4.2: Spectrum timing for the Nexus 10 with 87.5% overlap. PRF of 2890 to the left and 6410 to the right. The full column represents the cumulated run-time for 10 seconds of IQ-data while the green bar represents the FFT only. The data is averaged over 5 measurements.

The gathered plot for the cumulative run-times for the Nexus 10 can be seen in figure 4.3. The figure depicts the full cumulative run-times for both 75% and 87.5% overlap alongside with the cumulative run times for the associated FFT-sizes.

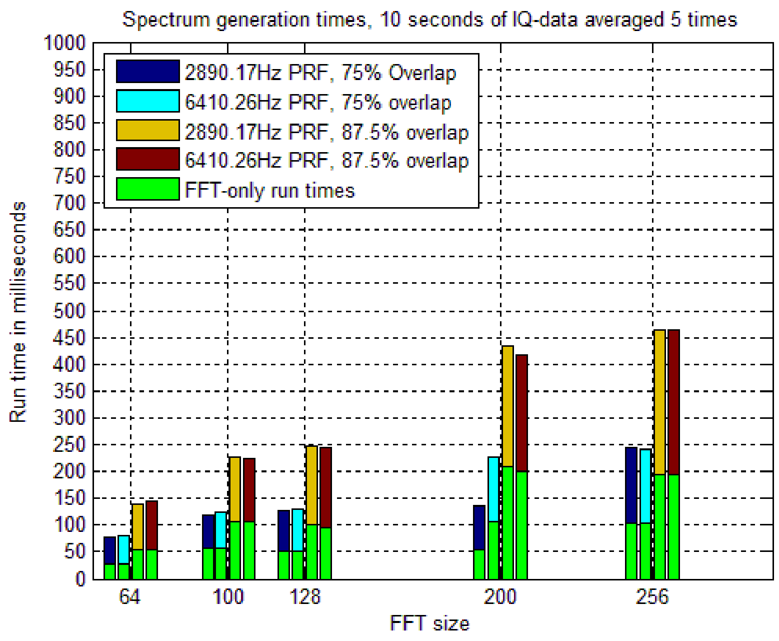


Figure 4.3: The spectrum timings for the Nexus 10 gathered in one plot.

PRF\FFT size	64	100	128	200	256
2890Hz	47.93%	55.82%	47.68%	57.74%	49.60%
6410Hz	42.72%	54.93%	48.15%	57.99%	49.00%

Table 4.3: FFT vs full run-time fractions with 75% overlap on the ASUS T300. The table represents how much of the processing is spent performing the FFT versus the full run-time of the spectrum generation.

4.1.2 ASUS T300

The measurements for the ASUS T300 Transformer pad were conducted in the exact same fashion as for the Google Nexus 10 and the results are presented in the same manner.

For an overlap of 75% figure 4.4 shows that the cumulative spectrum generation timer ranged from 120 to 450 ms, and that there was no significant difference in run-times between a PRF of 2890 and a PRF of 6410.

As can be seen from table 4.3 the fractions of the FFT vs the full spectrum generation run-times varied from 42.72% to 57.99% and the highest fractions can be found at an FFT-size of 200.

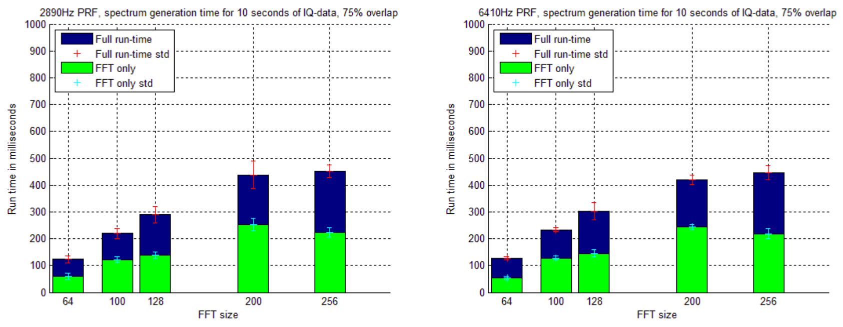


Figure 4.4: Spectrum timing for the ASUS T300 10 with 75% overlap. PRF of 2890 to the left and 6410 to the right. The full column represents the cumulated run-time for 10 seconds of IQ-data while the green bar represents the FFT only. The data is averaged over 5 measurements.

An overlap of 87.5% has cumulative run-times for the spectrum generation ranging from 220 to 900 ms as can be seen from figure 4.5. The most significant differences between a PRF of 2890 a PRF of 6410 can be see at FFT-size of 200 and 256.

The fractions of the FFT cumulative run-times and the whole cumulative

PRF\FFT size	64	100	128	200	256
2890Hz	42.83%	57.68%	46.97%	57.91%	50.17%
6410Hz	54.29%	49.94%	46.71%	57.53%	49.12%

Table 4.4: FFT vs full run-time fractions with 75% overlap on the ASUS T300. The table represents how much of the processing is spent performing the FFT versus the full run-time of the spectrum generation.

run-time for the spectrum generation ranges from 42.83% to 57.91%, as can be seen from table 4.4. Significant differences between the two different PRF 's can be seen at FFT-sizes of 64 and 100. The highest fractions without difference between the FFT-sizes can be found for an FFT-size of 200.

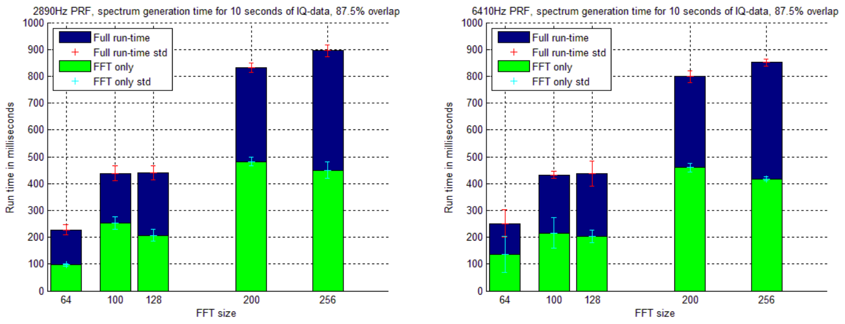


Figure 4.5: Spectrum timing for the ASUS T300 10 with 87.5% overlap. PRF of 2890 to the left and 6410 to the right. The full column represents the cumulated run-time for 10 seconds of IQ-data while the green bar represents the FFT only. The data is averaged over 5 measurements.

The gathered plot for the cumulative run-times for the ASUS T300 can be seen in figure 4.6. The figure depicts the full cumulative run-times for both 75% and 87.5% overlap alongside with the cumulative run times for the associated FFT-sizes.

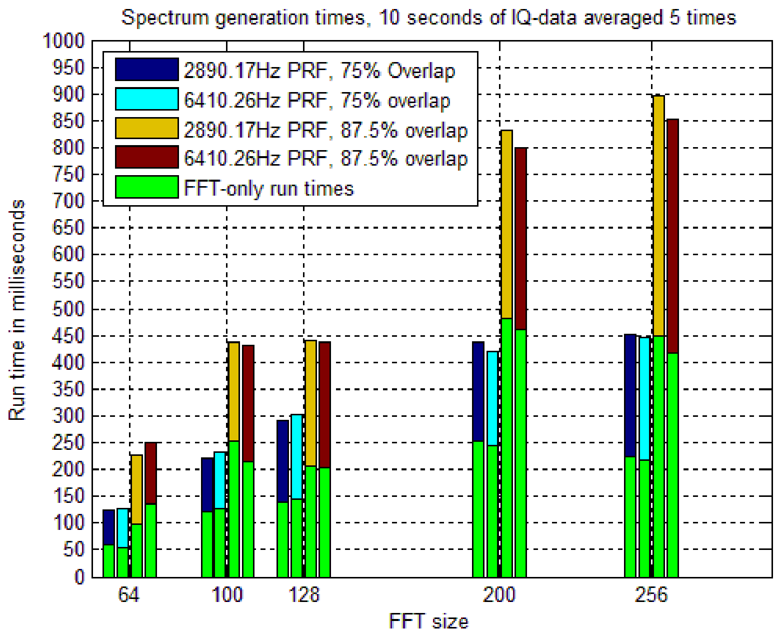


Figure 4.6: The spectrum timings for the ASUS T300 gathered in one plot.

% Overlap \ FFT-size	64	100	128	200	256
75	34.87%	15.16%	13.38%	12.44%	11.42%
87.5	33.92%	17.53%	20.10%	10.28%	13.49%

Table 4.5: Run-time differences represented as fractions for both 75% and 87.5% overlap. The fractions are calculated as $\frac{FFTWruntime}{VTKruntime}$.

4.1.3 VTKs built-in FFT vs FFTW

The two different implementations of the FFT, VTKs built-in FFT and FFTW with *Exhaustive* settings, was timed on a laptop in the same manner as on the Android devices. A cumulative timer for the execution of the FFT for 10 seconds of IQ-data was performed 5 times and averaged. The timing was done for both 75% and 87.5% overlap as can be seen from figure 4.7. The blue bar displays VTKs built-in FFT cumulative timer and the red bar represents FFTWs cumulative timer.

Table 4.5 depicts figure 4.7 in terms of how much time FFTW's FFT-algorithm spends on 10 seconds of IQ-data compared to VTKs FFT-algorithm. The table shows cumulative run-times for the FFTW algorithm to be from $3-10$ times faster than the VTK FFT-algorithm.

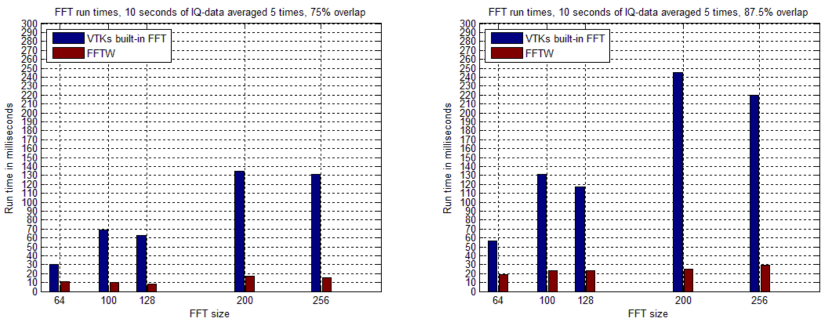


Figure 4.7: VTKs built-in FFT vs FFTW. The run times are cumulated over 10 seconds of IQ-data and averaged over 5 measurements. 75% Overlap to the left and 87.5% overlap to the right.

4.1.4 Anticipated run-times with the use of FFTW

New run-times for the Google Nexus 10 and the ASUS T300 is computed from figures 4.3 and 4.6, respectively. The original run-times for the FFTs were scaled according to table 4.5 and the results are displayed for the Nexus 10 on the left hand side in figure 4.8 and in the left hand side in figure 4.9 for the ASUS T300.

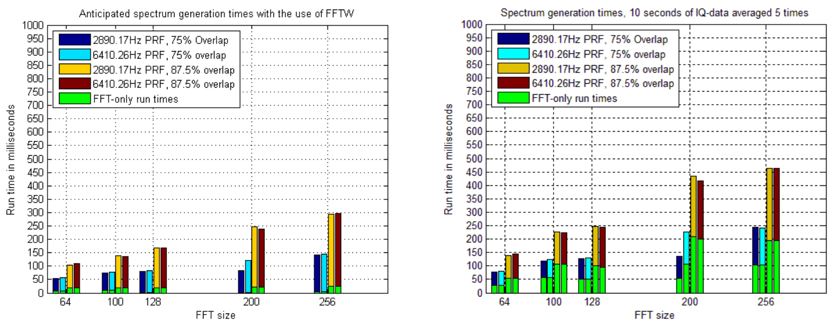


Figure 4.8: The anticipated run time for spectrum generation by using FFTW on the Nexus 10 on the left vs the measured times with VTK on the right.

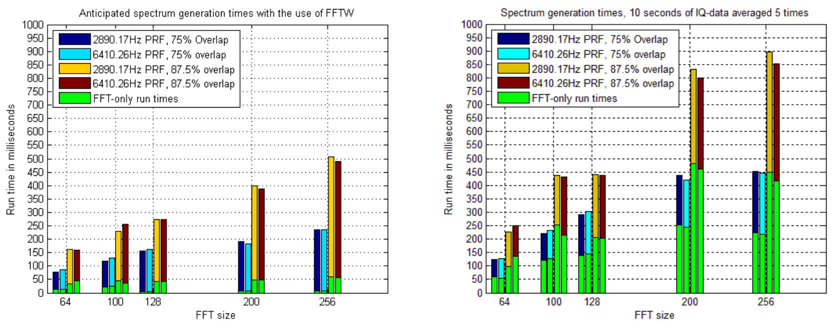


Figure 4.9: The anticipated run time for spectrum generation by using FFTW on the ASUS T300 on the left vs the measured times with VTK on the right.

4.2 Audio Generation timing and system delay

The computational efficiency of the audio generation is shown in figure 4.10, where the audio generation code was timed for 10 seconds of IQ-data 5 times, and averaged. These measurements were conducted for two different PRF's and two different playback frequencies. Both the ASUS T300 and the Nexus 10 were timed. The timing setup for the audio was the same as for the spectrum.

For the Nexus 10, the cumulated computational delay accounts for $75ms$ run-time for a playback frequency of $8kHz$ and between $115 - 130ms$ for a playback frequency of $16kHz$. The Asus T300 displayed run times at approximately $115ms$ for a playback frequency of $8kHz$ while for a playback frequency of $16kHz$ the cumulative timer was close to $190ms$.

An "impluse" test was also conducted, where the Nexus 10 and the ASUS T300 was laid flat upon the Vivid-Q. Both systems were running, and the probe was tapped against the palm of the hand to produce an "impulse" sound. The lag between when the sound was played from the ultrasound machine and the tablet was timed by recording several taps and measuring the difference in time between start-playoff from the Vivid-Q and start-playoff from the tablet. The recording device was a Samsung Galaxy S4 which was held by hand at a distance of approximately $50cm$. The audio-recordings were imported to a laptop where they were analyzed in Audacity[2] to measure the lag. The Results of this test is shown in figure 4.11.

[2]http://audacity.sourceforge.net/

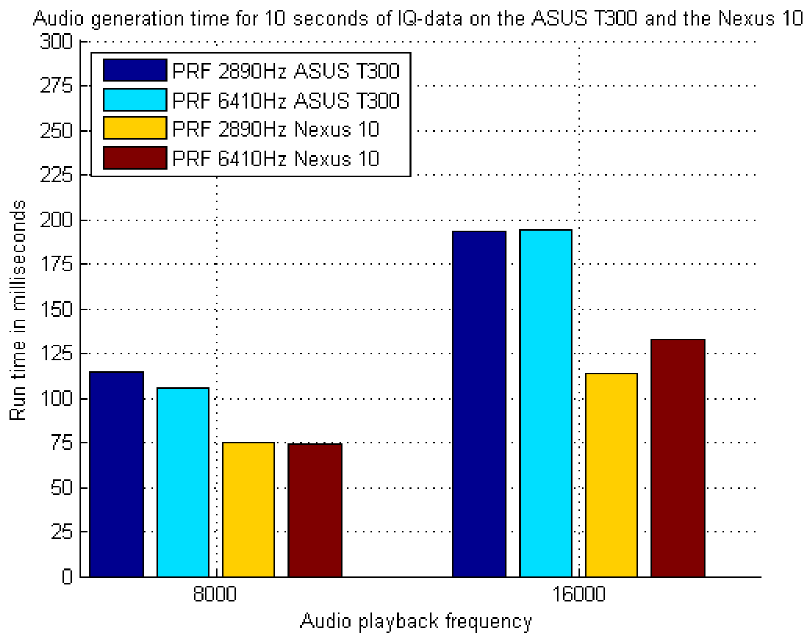


Figure 4.10: Audio generation times on the ASUS T300 and Nexus 10, based on the average of 5 runs with 10 seconds of IQ-data.

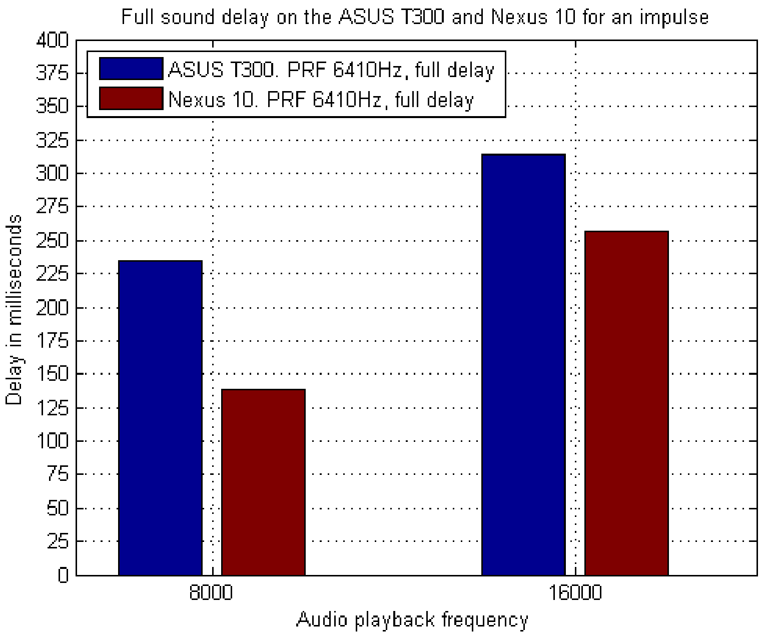


Figure 4.11: Measured lag between the Vivid-Q and the two different tablets. This is a measure of the full delay between the transfer of IQ-data; audio processing and audio playback through the system.

4.3 Achievable framerates with VTK's FFT

Based on equation 3.1, the data in appendix H and the audio-timing in figure 4.10 the framerate of the spectrum generation is computed and listed in tables 4.6→4.11.The performance is evaluated based on figure 3.10 with ratings from green-orange-red where green is best and red is worst. Due to a huge discrepancy in the results for the timing of the Nexus 10, the cell at 75% overlap, 2890Hz PRF and and FFT-size of 200 should be disregarded.

4.3.1 Spectrum generation only

Tables 4.6→4.7 are the achievable framerates if we only take into account the spectrum generation. Table 4.6 shows that the Nexus 10 performs well in most cases, except FFT-sizes of 200 → 256 with an overlap of 87.5%, while table 4.7 shows that the ASUS T300 performs well for low FFT-sizes (64 → 128) with an overlap of 75%, but the only adequate performance for 87.5% overlap is with a FFT-size of 64.

FFT-size\PRF	2890	6410
64	131	128
100	84	81
128	80	76
200	74	44
256	41	41

FFT-size\PRF	2890	6410
64	73	69
100	44	45
128	40	41
200	23	24
256	22	22

Table 4.6: Average framerates for the Nexus 10. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H and equation 3.1.

FFT-size\PRF	2890	6410
64	81	79
100	46	43
128	35	33
200	23	24
256	22	22

FFT-size\PRF	2890	6410
64	44	40
100	23	23
128	23	23
200	12	13
256	11	12

Table 4.7: Average framerates for the ASUS T300. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H and equation 3.1

4.3.2 Spectrum and audio generation

Tables 4.8→4.11 depict the achievable framerates if both the spectrum generation and the audio generation are taken into account.

The Nexus 10 performs well for all FFT-sizes and PRF's at 75% overlap and an audio-playback frequency of $8kHz$. The Nexus 10 encounters issues at FFT-sizes of 200 and 256 at 87.5% overlap.

For an audio-playback frequency of $16kHz$ the Nexus 10 performs well at 75% overlap for all FFT-sizes, while issues appears for 87.5% overlap for FFT-sizes at or above 200. The framerate between $8kHz$ and $16kHz$ drops significantly aswell.

The ASUS T300 displays low performance for almost all FFT-sizes, overlap percentages and audio-playback frequencies. The exceptions are FFT-sizes of 64 and 100 at an audio-playback frequency of $8kHz$, FFT-size of 64 with 87.5% overlap and a FFT-size of 64 with 75% overlap at an audio-playback frequency of $16kHz$.

FFT-size\PRF	2890	6410
64	66	65
100	51	50
128	49	49
200	48	33
256	31	32

FFT-size\PRF	2890	6410
64	47	45
100	33	33
128	31	31
200	20	20
256	19	19

Table 4.8: Average framerates for the Nexus 10 with audio generation at $8kHz$. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H , audio run-times in figure 4.10 and equation 3.1.

FFT-size\PRF	2890	6410
64	53	48
100	43	40
128	42	38
200	40	28
256	28	27

FFT-size\PRF	2890	6410
64	40	36
100	30	28
128	28	27
200	18	18
256	17	17

Table 4.9: Average framerates for the Nexus 10 with audio generation at $16kHz$. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H , audio run-times in figure 4.10 and equation 3.1.

FFT-size\PRF	2890	6410
64	41	42
100	29	29
128	24	24
200	18	19
256	17	18

FFT-size\PRF	2890	6410
64	29	28
100	18	18
128	18	18
200	11	11
256	10	10

Table 4.10: Average framerates for the ASUS T300 with audio generation at 8kHz. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H , audio run-times in figure 4.10 and equation 3.1.

FFT-size\PRF	2890	6410
64	32	32
100	24	24
128	21	20
200	16	16
256	16	16

FFT-size\PRF	2890	6410
64	24	23
100	16	16
128	16	16
200	10	10
256	9	10

Table 4.11: Average framerates for the ASUS T300 with audio generation at 16kHz. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H, audio run-times in figure 4.10 and equation 3.1.

4.4 Achievable framerates with FFTW

Based on equation 3.1, the data in appendix H, the VTK vs FFTW fractions in table 4.5 and the audio-timing in figure 4.10 the anticipated framerate, with the use of FFTW, of the spectrum generation is computed and listed in tables 4.12→4.17.The performance is evaluated based on figure 3.10. Due to a huge discrepancy in the results for the timing of the Nexus 10, the cell at 75% overlap, 2890Hz PRF and and FFT-size of 200 should be disregarded.

4.4.1 Spectrum generation only

Table 4.12 show that the Nexus 10 will perform very well for all tested FFT-sizes and overlap percentages, with the lowest framerate at 36Hz found at 87.5% overlap and an FFT-size of 256. The ASUS T300 will also perform very well, as table 4.13 shows, for almost all settings. The tablet will only encounter problems for high FFT-sizes (256) and 87.5% overlap.

FFT-size\PRF	2890	6410
64	186	174
100	135	131
128	126	122
200	120	82
256	70	70

FFT-size\PRF	2890	6410
64	103	97
100	72	72
128	66	66
200	43	45
256	36	36

Table 4.12: Anticipated framerates for the Nexus 10. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H, table 4.5 and equation 3.1.

FFT-size\PRF	2890	6410
64	128	117
100	84	78
128	64	62
200	52	55
256	42	42

FFT-size\PRF	2890	6410
64	65	69
100	43	39
128	42	42
200	28	28
256	22	22

Table 4.13: Anticipated framerates for the ASUS T300. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H, table 4.5 and equation 3.1.

4.4.2 Spectrum and audio generation

Tables 4.14→4.17 depict the achievable framerates if both the spectrum generation and the audio generation is taken into account, alongside with an implementation of the FFTW-library.

The Nexus 10 displays good performance for both $8kHz$ and $16kHz$ playback-frequency, all FFT-sizes and both overlap percentages, as shown in figures 4.14 and 4.15. For an overlap of 87.5% and FFT-sizes of 200 and 256 the Nexus 10 dips below a FPS of 35 but stays above 24.

Tables 4.16 and 4.17 depicts a relatively good performance for the ASUS T300 at an audio-playback frequency of $8kHz$ but it has trouble processing the workload at FFT-sizes of 200 and 256 at an overlap of 87.5%. For an audio-playback frequency of $16kHz$ the ASUS T300 has fluent framerates at 75% overlap until the FFT-size reaches 256. The only fluent framerate at a overlap of 87.5% is at a FFT-size of 64, the rest of the FFT-sizes at 87.5% depicts low performance for the ASUS T300.

FFT-size\PRF	2890	6410
64	78	76
100	67	66
128	65	64
200	63	51
256	46	46

FFT-size\PRF	2890	6410
64	58	56
100	48	47
128	44	44
200	33	34
256	29	28

Table 4.14: Anticipated framerates for the Nexus 10 with audio generation at $8kHz$. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H , audio run-times in figure 4.10 , table 4.5 and equation 3.1.

FFT-size\PRF	2890	6410
64	60	53
100	54	48
128	52	47
200	51	40
256	39	37

FFT-size\PRF	2890	6410
64	49	43
100	40	37
128	38	36
200	29	28
256	26	25

Table 4.15: Average framerates for the Nexus 10 with audio generation at $16kHz$. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H , audio run-times in figure 4.10 , table 4.5 and equation 3.1.

FFT-size\PRF	2890	6410
64	50	51
100	42	42
128	36	37
200	32	34
256	28	30

FFT-size\PRF	2890	6410
64	37	39
100	28	27
128	28	29
200	21	22
256	17	18

Table 4.16: Average framerates for the ASUS T300 with audio generation at $8kHz$. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H , audio run-times in figure 4.10 , table 4.5 and equation 3.1.

FFT-size\PRF	2890	6410
64	37	36
100	32	31
128	29	28
200	26	27
256	23	24

FFT-size\PRF	2890	6410
64	29	30
100	24	22
128	23	23
200	18	18
256	15	16

Table 4.17: Anticipated framerates for the ASUS T300 with audio generation at $16kHz$. The left table is for 75% overlap and the right table is for 87.5% overlap. The numbers are based on the raw-data in appendix H, audio run-times in figure 4.10 , table 4.5 and equation 3.1.

4.5 Spectrum and Audio validation

The spectrum generator is validated using a synthetic signal created by Hans Torp using Matlab. The synthetic signal is generated as a $140Hz$ sine-wave in the frequency domain with additive white noise and the synthetic IQ-data is sampled at a rate of $4kHz$. The synthetic signal is depicted with white noise on the right side of figure 4.12 It can be recreated using the Matlab script in appendix A.

The Audio is validated using a another synthetic signal which is composed of two stationary band-pass signals without noise. The two stationary band-pass signals are centered around $1.7kHz$ and $-0.7kHz$ with a bandwith of about $0.2kHz$ as depicted at the top in figure 4.15. The audio-validation signal can be recreated using the Matlab script in appendix B.

The validation is performed by comparing the output of both the spectrum generation- and sound generation-algorithms in Matlab and C++.

4.5.1 Spectrum Validation

Figure 4.12 depicts the spectrums generated from the synthetic signal both in Matlab and C++. The comparison between the two different images depicted in figure 4.13 was created by using the Matlab built-in function $imabsdiff$ with the two different pictures as input and$imagesc$ to visualize. There is a slight difference in intensity in the input images. Since $imagesc$ was used to display the image, the flat gray areas are where the spectrums were exactly the same. The Black region is where the images were saturated. Some shot-noise can be seen in the image.

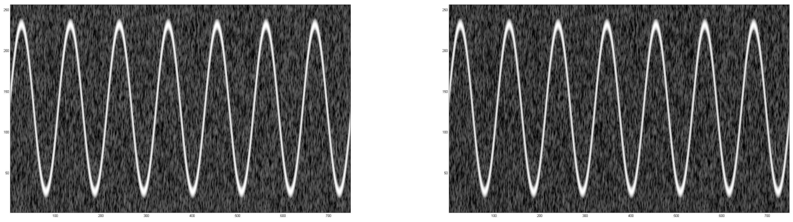


Figure 4.12: Spectrum generated from noisy synthetic IQ-data, representing a frequency sine-wave of 140Hz. The spectrum on the left is generated with the C++ -code, while the spectrum on the right is from Matlab using a standard spectrum-generation algorithm. The two methods use an overlap of 75% with a FFT-size of 256. Both spectrums are visualized using Matlab.

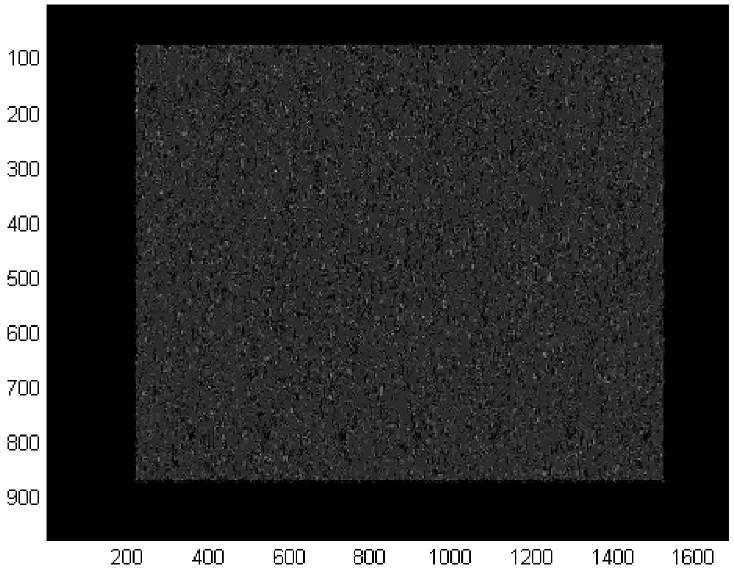


Figure 4.13: The difference between the Matlab-produced spectrum and the C++-generated spectrum.

4.5.2 Audio Validation

The audio is validated using a synthetic signal composed of two stationary band-pass signals and comparing the Matlab-algorithm with the C++ -algorithm - output. The power spectrum of the band pass signals can be observed in figure 4.14 and was created using a FFT-size of 256 and FFT-shifted to bring the zero'th frequency to the middle of the spectrum. The negative frequency signal is found below bin-number 128 and the positive frequencies are found above that bin-number. The original sampling frequency is $4kHz$ and the new sampling frequency after interpolation is $16kHz$.

Figure 4.17 is the result of computing the negative frequencies-channel output, using both Matlab, as depicted in the top figure, and C++ as depicted in the bottom figure, and displaying the frequency spectrum using Matlab. Frequency leakage from the positive band-pass signal peaking at about $-20dB$ can be seen at approximately $2.2 \rightarrow 2.4kHz$. The frequency content of the positive frequencies-channel output is depicted in figure 4.18, and we can observe frequency leakage from the negative frequency signal at $0.6 \rightarrow 0.8kHz$.

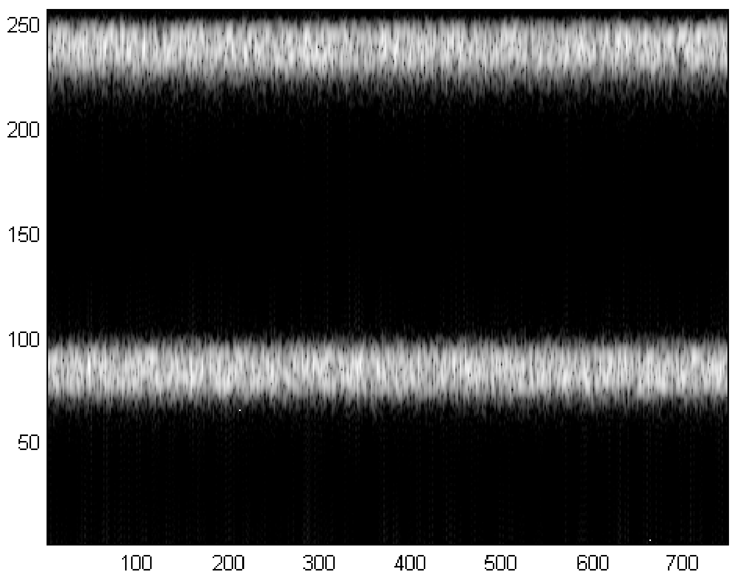


Figure 4.14: The power spectrum of the audio validation signal.

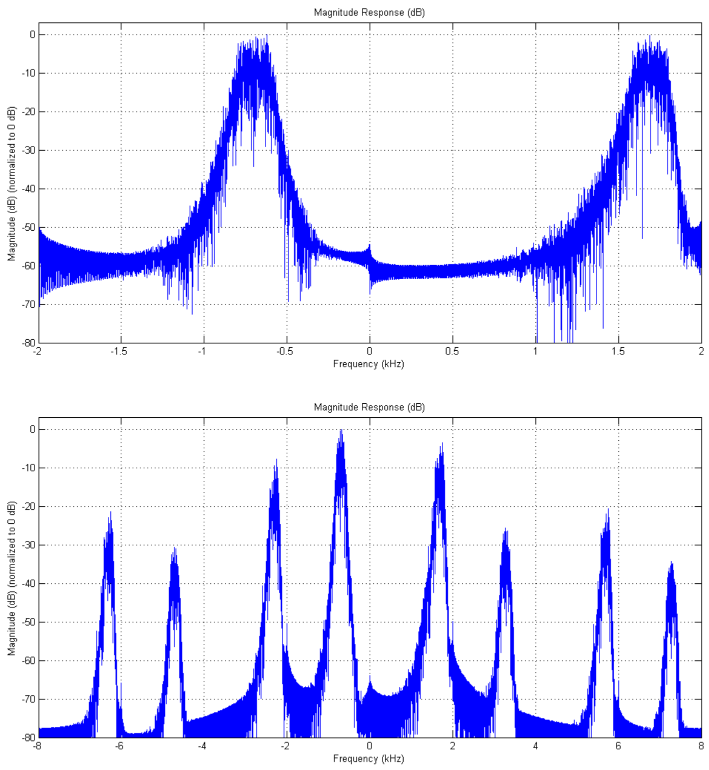


Figure 4.15: Audio test signal. The top figure depics the original test signal, while the bottom figure shows the test signal after interpolation from $4kHz$ to $16kHz$.

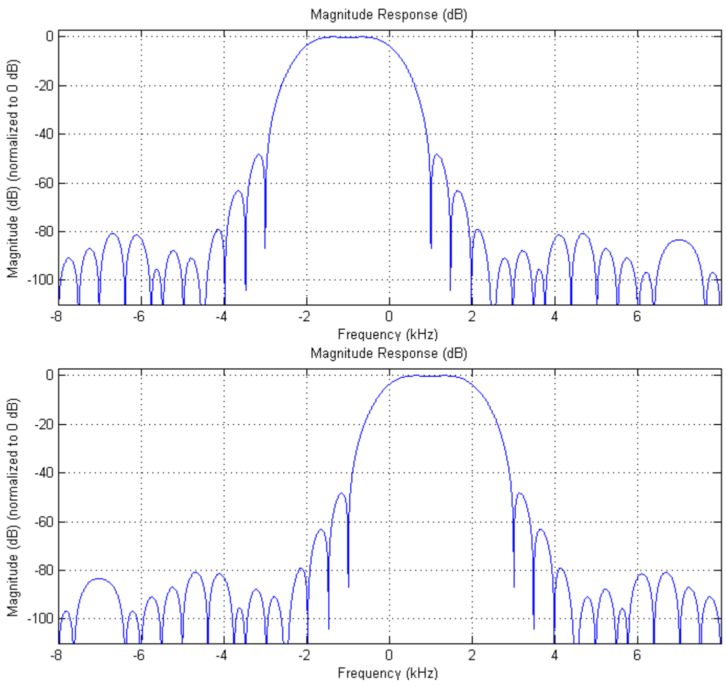


Figure 4.16: Filters used to split the positive and negative frequencies.

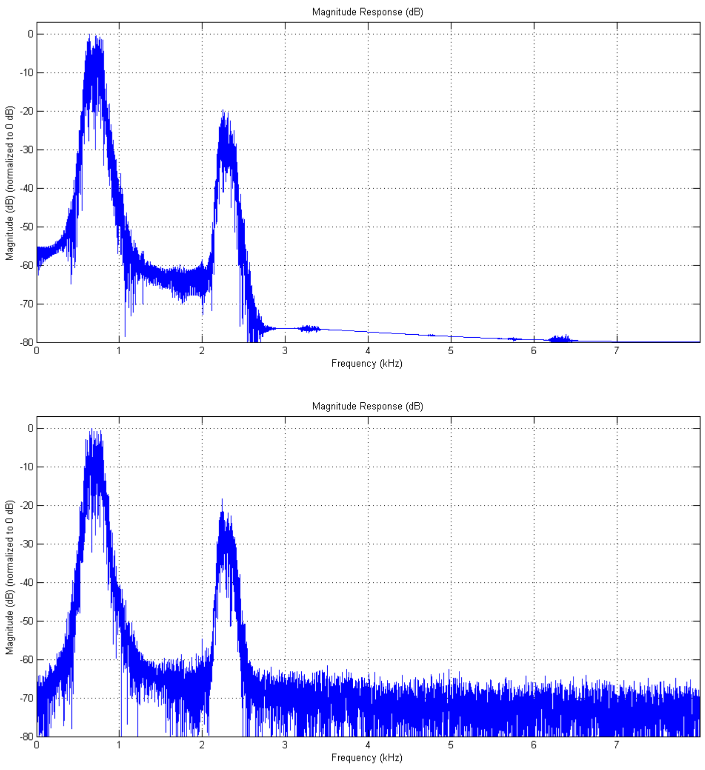


Figure 4.17: Spectrum of the audio generated from the testdata. The top figure represents the Matlab-generated channel for negative frequencies while the bottom figure is the C++-generated negative frequency channel. Both channels are computed with the same algorithm and visualized using Matlab.

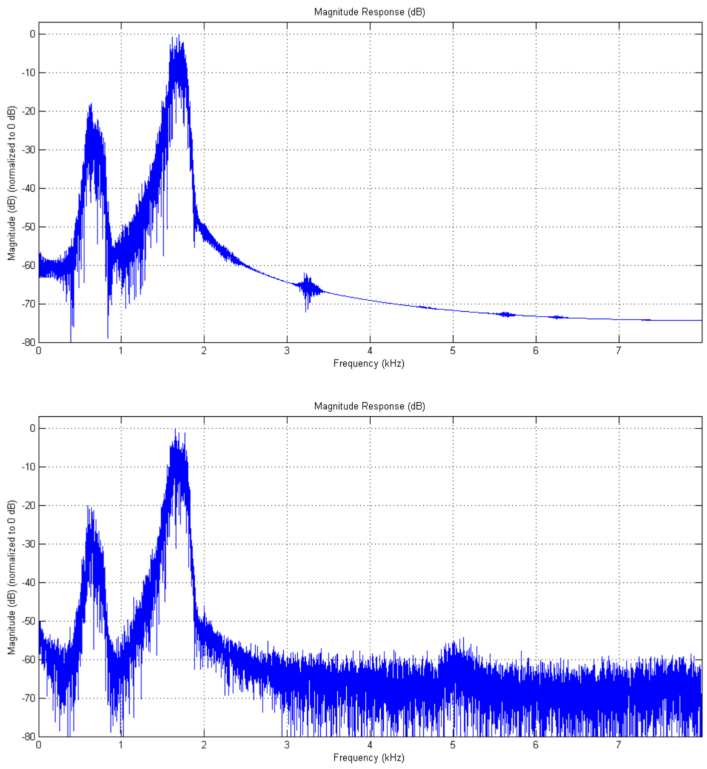


Figure 4.18: Spectrum of the audio generated from the testdata. The top figure represents the Matlab-generated channel for positive frequencies while the bottom figure is the C++-generated positive frequency channel. Both channels are computed with the same algorithm and visualized using Matlab.

4.6 Spectrum quality

To determine the quality of the produced real-time spectrum, several screenshots were taken from the Nexus 10. These screenshots can be seen in figures 4.20-4.24, where the top picture is 75% overlap and the bottom figure is 87.5% overlap in all figures. All spectrums used a $20ms$ -length window. The spectrums were created by measuring a human carotid artery in real-time, so the data differs from picture to picture. The PRF was set to 2890 and the FFT-size ranges from $64 \rightarrow 256$. Triplex mode was turned off so no synthetic signal is present in the spectrums.

The first thing to note about figures 4.19→4.24, depicting spectral quality with different parameters, is that the baseline is wrongly adjusted. This is a minor bug in the software, concerning where VTK draws the baseline, and it has no effect on the results other than that the y-axis is wrong with respect to the velocity and that the baseline is drawn at the wrong location.

Figure 4.19 is two screenshots from the Nexus 10, where both pictures are created with a FFT-size of 256 and an overlap of 75%. The top picture however is created using spectral smoothing after logarithmic compression, while the bottom picture is created by spectral smoothing before logarithmic compression.

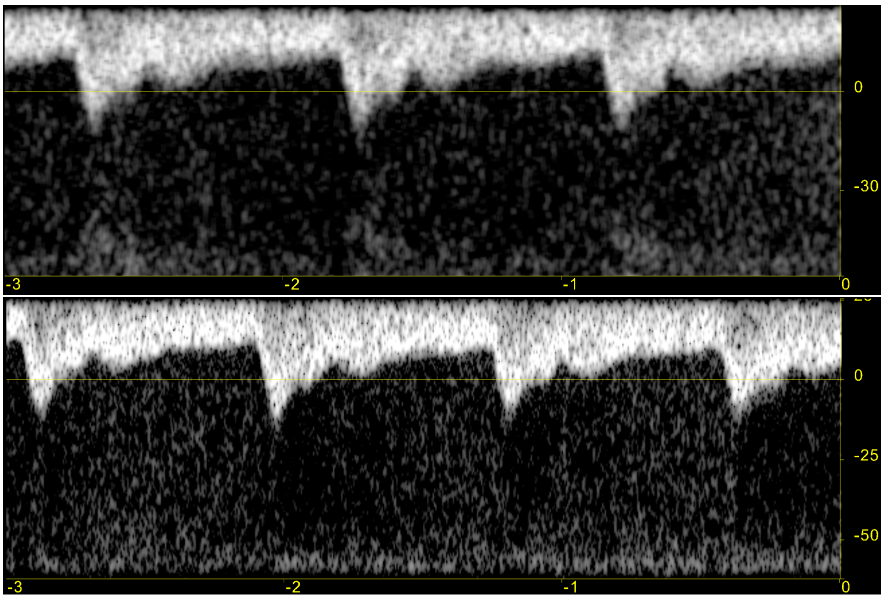


Figure 4.19: Spectrum quality of two smoothing methods. The spectrum in the top figure is created by smoothing after logarithmic compression and the bottom figure is created by smoothing before logarithmic compresion. Both figures use a 256-point FFT and 75% overlap along with 25ms smoothing. The dynamic range is set to $35dB$.

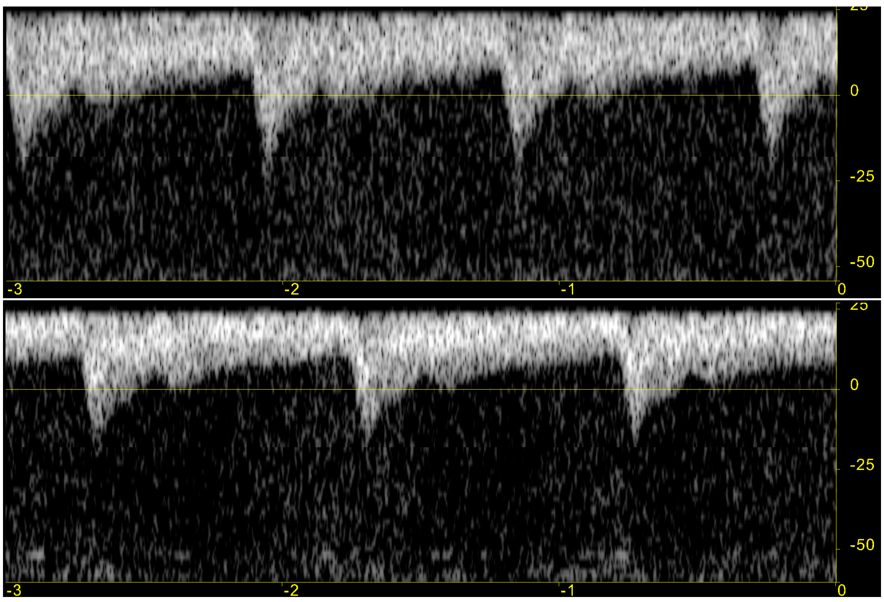


Figure 4.20: 64 point FFT with 75% overlap on the top and 87.5% overlap on the bottom. The spectrums have undergone 25ms spectral smoothing and the dynamic range is set to 35dB.

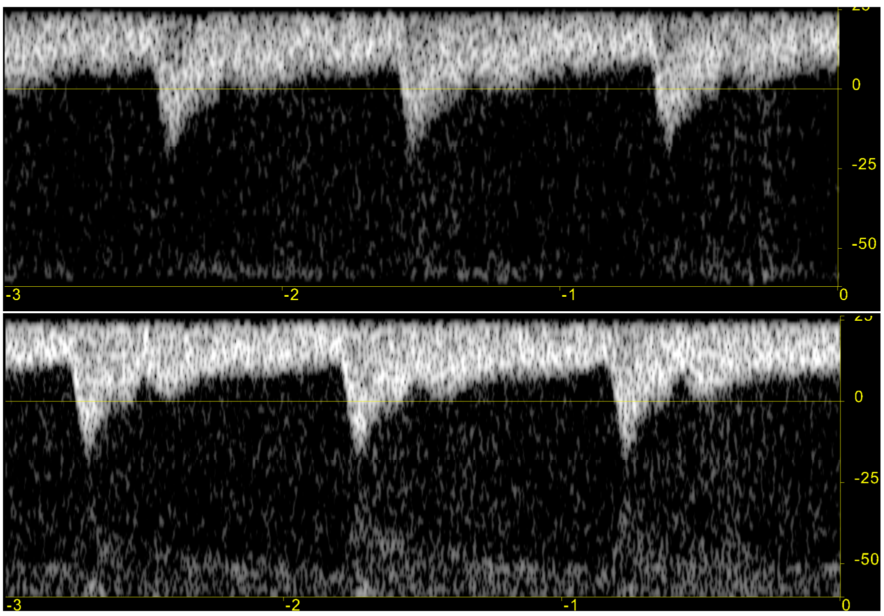


Figure 4.21: 100 point FFT with 75% overlap on the top and 87.5% overlap on the bottom. The spectrums have undergone 25ms spectral smoothing and the dynamic range is set to 35dB.

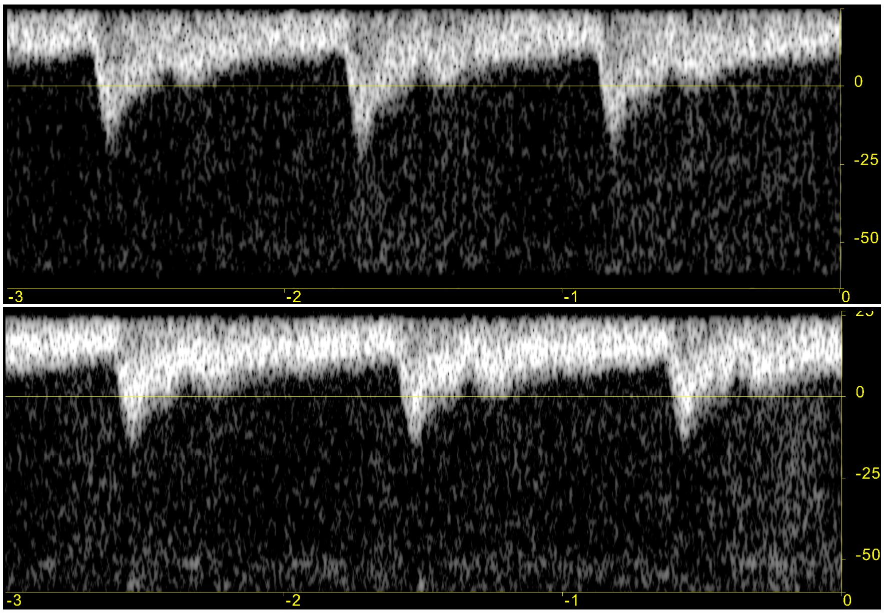


Figure 4.22: 128 point FFT with 75% overlap on the top and 87.5% overlap on the bottom. The spectrums have undergone 25ms spectral smoothing and the dynamic range is set to 35dB.

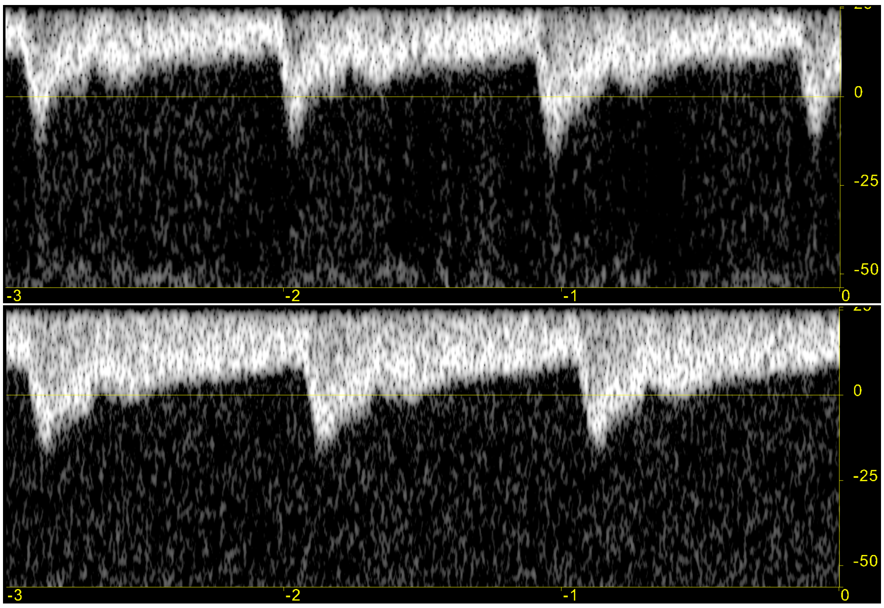


Figure 4.23: 200 point FFT with 75% overlap on the top and 87.5% overlap on the bottom.The spectrums have undergone 25ms spectral smoothing and the dynamic range is set to 35dB.

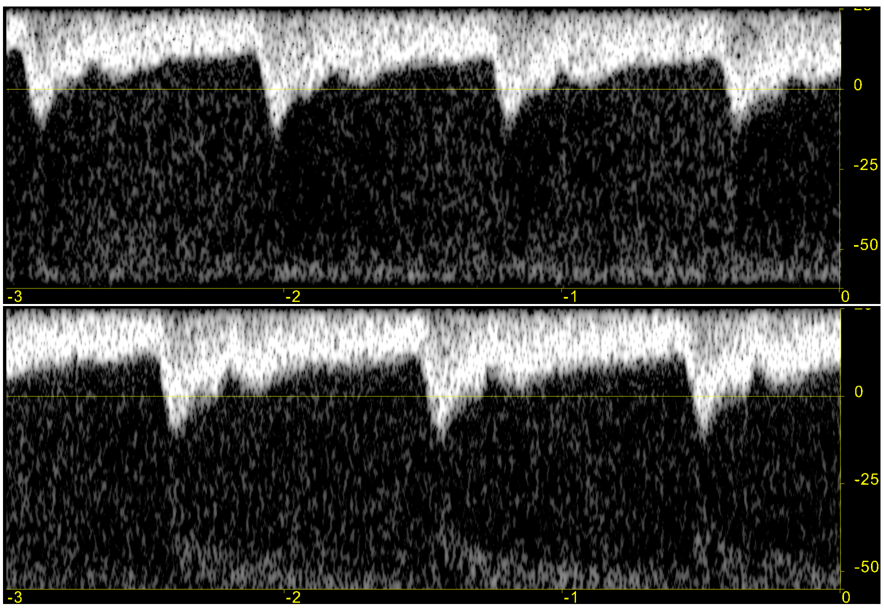


Figure 4.24: 256 point FFT with 75% overlap on the top and 87.5% overlap on the bottom. The spectrums have undergone 25ms spectral smoothing and the dynamic range is set to 35dB.

5 Discussion

5.1 Spectrum generation

Validation

The spectrum validation in figure 4.13 shows that the spectrum generator implemented in the Umoja-system performs as well as a standard Matlab-implementation when quality is concerned. The frequency- and time resolution with a (almost) full-band, highly varying (140*bpm*) signal is reproduced almost identically as the Matlab implementation. Figure 4.13 shows some very small errors, which are the brighter and darker dots scattered around the image, besides the difference in intensity, but these are likely accuracy issues such as floating point-errors[10]. The scattered dots are scaled difference-values, such that the figure 4.13 depics where there is an error, even very small ones. The total black dots are either where both images were saturated, or where there was no signal at all in both images. The original image before scaling is found in appendix G.

Spectral quality

Smoothing methods: Figure 4.19 is a comparison of two different methods for spectral smoothing. The difference is that the top picture is smoothed after the logarithmic compression, while the bottom is smoothed after finding the power of the Fourier-transformed IQ-data, but before the logarithmic compression. The latter method has been shown to minimize the standard deviation of the resulting spectrum[15]. A Matlab script which shows this effect is given in appendix F, where the standard deviation is reduced by a factor of about five when the smoothing is performed on three samples before the logarithmic compression. The bottom picture is clearer than the top picture, which is due to the lower standard deviation.

Quality vs parameters: Since the data in each spectrum in figures 4.20→4.24 is different the quality is hard to assess, but one noticeable effect of increasing the FFT-size is that the transit-regions between signal and no-signal seems to be sharper in the frequency-"direction" and the edges seem smoother in the time-"direction". The overlap percentage increases the time-resolution as both the inverted tops and dips are clearer in the 87.5%-overlap pictures compared to the 75% ones.

Parameters vs complexity: Spectral settings and processing time are closely linked as can be seen from figures 4.3 and 4.6. The FFT-size affects both how much time is spent computing the FFT alone, but also how much time which must be spent smoothing and compressing the data. The runtime overhead (the time not spent computing the FFT) seems to increase in a close to linear fashion as the FFT-size increases, which is anticipated because a larger FFT-size increases the amount of data which has to be processed after the transform.

The FFT-size also increases the time spent computing the FFT itself. This increase in run-time arises from two factors: The FFT-algorithm used by the VTK library and the amount of data which needs to be processed. Since VTK has implemented a FFT-algorithm which specializes in radix-2 transforms, the radix-2 cases will be less computationally expensive compared to their sizes as opposed to non-radix-2 cases, this can be seen from figures 4.3 and 4.6, as well as tables 4.1→4.4.

Since the window-length is set to a static size ($20ms$), the window-length in samples increases with increasing PRF. This causes the computational complexity before the FFT to increase and thus the runtime is expected to rise with increasing PRF, but only in a minimal sense since the only computation performed on the IQ-data before the FFT is multiplication by a Hamming-window and zeropadding. The increase in PRF will actually cause the for-loop performing the zeropadding to be less computationally expensive, since the number of zeros which needs to be inserted is less than with a lower PRF. As can be seen from figures 4.1→4.6 increasing the PRF does not always increase the total run-time, but the factor likely to be the reason for this is measurement variance depicted in the same figures and not reduced zeropadding complexity.

It should be noted that in figure 4.3, the total run-time for a size-200 FFT with a PRF of 2890 at 75% overlap is considerably lower than with a PRF of 6410 (~$140ms$ for $2890Hz$ and ~$225ms$ for $6410Hz$). This discrepancy cannot be explained by variance or increased complexity and must thus arise from other factors; which might include typos when measuring or faulty measurement.

Tablet comparison

Comparing figures 4.3 and 4.6 gives a clear picture of the difference in run-times for spectrum generation on the Nexus 10 and the ASUS T300. The Nexus 10 outperforms the ASUS T300 by nearly a factor of two for all FFT-sizes and overlap percentages.

A noticeable difference is also found by comparing tables 4.1→4.4 where the fraction of time spent computing the FFT on the ASUS T300 is $5-10$-percentiles higher for all FFT-sizes and overlap percentages compared to the Nexus 10.

5.2 FFT optimization

Performance gain

The FFTW-implementation was 3-10 times faster than the VTK implementation on the laptop, as figure 4.7 and table 4.5 shows. These numbers are probably the highest speedups possible with the tested FFT-sizes since the FFTW-implementation most likely utilized SIMD-instructions to optimize the codelets.

By recalculating the timers in figures 4.3 and 4.6 with the numbers in table 4.5, figures 4.8 and 4.9 were created. These latter two figures depict the optimal performance achievable by implementing FFTW on the tablets. The performance gain is subtstantial for both the Nexus 10 and the ASUS T300. The total run-time on the Nexus 10 is decreased by up to 35% and 45% on the ASUS T300.

The results involving the approximated run-times on the ASUS T300 and Nexus 10 are based on an approximately equal difference in run-time for VTKs implementation of the FFT and FFTWs implementation as on the laptop. The anticipated results might differ on different mobile platforms because FFTW constructs SIMD-instructions for their codelets, as described in section 3.3.2, and the CPU on a tablet will either be supported by FFTW SIMD-generation or not. This will have an important effect on the run-times achieveable on a mobile device and the efficiency of an FFTW-implementation.

GNU GPL

One major drawback of using FFTW is that it is released under the GNU GPL which constricts free usage of the software commercially, but the GNU GPL can be voided as described in section 3.3.2.

5.3 Audio generation and playback

Validation

The audio validation displayed in figures 4.17 and 4.18 which was created by applying the test-signal in figure 4.15 shows that the C++-implemenation matches the Matlab-implementation, but also that there is some spectral leakage from each of the channels to the other, as well as some induced noise which is from the linear interpolation[19]. The frequency-leakage is present because the split-filter's main lobe is widened by applying a Hann-window to reduce the side-lobe levels as depicted in figure 4.16. The widening of the main-lobe causes the filters to have quite high passbands around the zero-frequency line and thus spectral leakage from both high positive, high negative, low positive and low negative frequencies will appear in the opposing channel.

Since the original IQ-data is clutterfiltered at low frequencies, the near-zero frequencies will not be a problem but the high- positive and negative frequencies will leak. Even though leakage from the negative channel to the positive channel and vice versa is present, the strength of the leaked frequencies is about $20dB$ lower than the correct frequencies with the current test-signal. This is barely audible when listening to the resulting audio and in live-testing almost no leakage can be heard.

The drawback of the filter-design as it is now, is that for signals crossing the zero-line with small shifts it will be hard to hear the difference between positive and negative frequencies; low frequency-shifts, both positive and negative, will be heard in both channels. This can have implications in examinations requiring sharp audible differentiation between positive and negative low-velocity blood flow.

On the other hand, the real-time sound-tests have given clear sound and a good differentiation between regular carotid blood-flow and flow from the vein lying next to it (vena jugularis).

Possible improvements One way of improving the filter is to compute how much the Hann-window widens the main-lobe and then make the necessary reduction in the Parks-McClellan cut-off frequency to compensate. The filter-order can also be increased, but this will increase the computational complexity and add to the audio-delay and hardware requirements. Other specialized adaptive windows, like the Kaiser-Bessel window, might also be investigated to reduce the width of the main lobe while still suppressing the side lobe levels sufficiently.

To reduce the induced noise from interpolation, a shifted linear interpolation-algorithm can be implemented in high-performance devices[19].

Audio runtimes

The timing results in figure 4.10 depicts that the Nexus 10 is somewhat faster in processing the audio than the ASUS T300, which is to be expected due to the hardware differences of the two tablets.

There are two main factors which affect the complexity of the audio-generation algorithm; the PRF and the audio playback frequency. Since the linear interpolation is set to a static playback-frequency, the PRF influences how many new points which is needed to achieve the new sampling frequency. Thus the higher the PRF is the lower the complexity is. The playback frequency regulates how many samples which needs to be filtered after the interpolation, and thus a higher playback frequency should result in an increased computational complexity.

The effects of the second factor, the playback frequency, is clear from figure 4.10 while the effects of the first factor seems to be overshadowed by measurement variance.

Total system delay

The delay-test results depicted in figure 4.11 show a total delay of about $230 - 320ms$ on the ASUS T300 and $130 - 255ms$ on the Nexus 10 for audio playback frequencies of $8kHz$ and $16kHz$.

One of the factors that add a static delay is the buffer used by the audio-player which is set to $25ms$ in this implementation. Another factor is the transfer of data from the Vivid-Q to the tablets via IP which *could* be higher for the ASUS T300 since it used a wireless connection while the Nexus 10 used a direct cable connection. The filter used to remove transients and split the positive and negative frequencies is set to a static size of 32samples, which adds $\frac{32}{PRF} = 5ms$ to the delay[1].

Parsing the data within the system and the processing also accounts for some of the delay, but this expected to be quite small since the audio-player is controlled by a different thread than the processing. This means that for each packet of IQ-samples, the audio-player will start processing and playing the sound. The rest of the delay is from the audio-player itself which interacts with the audio-processing hardware and playback algorithms on the respective tablets.

There is a noticeable difference in the total delay between $8kHz$ and $16kHz$ playback and some of the added delay to the $16kHz$ delay can be accounted for by the extra processing. But all the extra delay, $\sim 90ms$ on the ASUS T300 and $\sim 125ms$ on the Nexus 10, cannot be explained by the increased computational complexity introduced by increasing the playback frequency. It is clear that an increase in the resampling-frequency will increase the number of samples which needs to be processed by the audio-player, but it is unlikely that this accounts for the rest of the additional delay since the increase in delay on the ASUS T300 from $8 - 16kHz$ is less than for the Nexus 10 in percentage[2]. The extra delay may be because of hardware/software specific configurations related to sound-playback on each of the tablets, but this needs further investigation.

[1]The PRF was set to $6410Hz$ in the test

[2]39% for the ASUS T300 and 96% for the Nexus 10

5.4 Realtime assessment

Note The realtime assessment is based on derived values taken from the performance of the spectrum- and audio-generation as well as the anticipated runtime with FFTW. The assessment is thus subject to the same performance factors.

VTK performance

The framerates achievable from utilizing the built-in Fourier-transform in VTK were acceptable for the Nexus 10, listed in table 4.8, which depicted fluent rates for all FFT-sizes with $8kHz$ playback-frequency. The Nexus 10 also displayed relatively good framerates, as can be seen in table 4.9, for $16kHz$ playback-frequency until a FFT-size of 200. Although some of the framerates were in the *orange* region, these rates are deemed acceptable since we don't need to refresh the whole frame, just update the spectrum-part with new lines. The framerates and ratings for the Nexus 10 coincides with the subjective visual quality.

For the ASUS T300, the framerates, available in tables 4.10 and 4.11, are barely acceptable at an audio-frequency of $8kHz$ and an overlap at 87.5% will make the spectral display very sluggish. The only acceptable FFT-sizes are from $64 \rightarrow 128$ at 75% overlap at $8kHz$ frequency. The rest of the framerates for $8kHz$ are considerably lower than the Nexus 10 and this will cause the user to experience sluggish update of the spectrum. For a playback-frequency of $16kHz$ the ASUS T300 performed badly for FFT-sizes above 128 at 75% overlap and FFT-sizes above 64 for an overlap of 87.5%.

The differences between the performance of the two tablets is anticipated because of the huge difference in both spectrum- and audio-generation cumulative timers.

FFTW anticipated performance

Using the FFTW library the anticipated framerates increased significantly for both the Nexus 10 and the ASUS T300. Framerates is increased by 5-15 FPS which results in fluent framerates for both $8kHz$ and $16kHz$ with the highest settings on the Nexus 10. The ASUS T300 can achieve fluent framerates for both $8kHz$ and $16kHz$ up to an FFT-size of 200 with 75% overlap, which is significantly better than the current VTK-implementation. The ASUS T300 does not achieve fluent framerates at 87.5% overlap and $16kHz$ playback frequency which indicates that the low-performance tablets should be limited to 75% overlap.

Increasing the framerate

Several steps can be taken to improve the framerate on tablets which cannot handle the workload of both spectrum- and audio-processing.

The first thing to notice is that both the tested tablets perform relatively well without audio-processing turned on, as depicted in tables 4.6 and 4.7. The ASUS T300 can operate at a good framerate until a FFT-size of 200 at 75% overlap, which indicates that the operator probably should have the ability to turn off the audio-processing if the system seems sluggish.

Another point is that not all systems have stereo capabilities, and the audio-generation implementation could be made adaptive to mono-systems. Generating mono sound instead of stereo will decrease the computational complexity and give low-medium performance devices a better spectral refresh-rate.

Automation If the system is to be adaptive to a wide range of devices, a script which automatically detects optimal settings (overlap percentage, FFT-size, stereo/mono-audio) which results in a fluent framerate could be implemented. This script could time the spectrum- and audio-generation with different settings and find the systems optimal settings by rating according to table 3.10. Such an automatic rating would ensure that the system performs optimally on the device in use.

6 Conclusions

By implementing a real-time Doppler spectrum generator and a real-time audio generator, the main objectives of this thesis has been achieved. The optimization of the Fourier-transform, and the evaluation of the influence that different spectrum-parameters have on the real-time performance of the system, renders the sub-objectives of the thesis achieved. By assessing the spectral and audio quality, as well as assessing the real-time performance of the software, the implementation has shown to be of value for the further development of the Umoja ultrasound system. No work , however, has been put into the optimization of the Doppler triplex mode.

To the writers knowledge, the ultrasound Doppler spectrum- and audio-generation as an out-of-the-box installation supporting several Android devices has given the implementation functionality that is not yet provided by any other available software.

Assessing both the spectrum and the audio-quality of the implementation showed that the spectrum is of sufficient quality when compared to a Matlab-implementation. The audio, subjectively, sounds good; but frequency leakage can be a problem for low and near-Nyquist velocity blood-flow. The audio-delay, albeit quite high for the highest playback frequency, is good enough since the displayed spectrum and the audio are synchronized.

The current implementation has proven to perform well on a mid- to high-end Android tablet but the performance on a low- to mid-end tablet was unsatisfactory. By applying a new implementation of the fast Fourier-transform it is shown that the low- to mid-end tablet can perform well. Without a new implementation of the fast Fourier-transform, the low- to mid-end tablets should be given a choice wether to process audio or not to alleviate the computational complexity and thus increase the framerate of the Doppler spectrum visualization.

A Samsung Galaxy S4 Active has also been tested to check if the implementation would work on a smartphone. The result of the test, which is a screen-recording, is included in a video-file in the digital copy of the thesis. An audio-file which is produced by generating the audio from the spectrum test signal is also included.

7 Recommendations

This is a short list of the improvements suggested in the discussion, as well as one additional recommendation.

1. To reduce the induced noise from the linear interpolation in the audio generator, we can use a shifted-linear interpolation algorithm instead of the regular linear interpolator[19]. This algorithm is a bit more computationally expensive, which means it should only be used on high-end devices.

2. Since the Umoja-system is designed as an extremely low-cost system, the mobile device which is to be used by the operator should be relatively easy and cheap to aqcuire. To assure that the system performs optimally on the system in use, a script which automatically evaluates the hardware could be implemented. This would allow the software to adapt to the hardware available by finding the optimal spectral-settings which result in a good framerate. This script could be extended to check if the current hardware has stereo capabilities and thus adapt the audio-generator, since deactivating the stereo processing will reduce the computational complexity.

3. If the mobile device owned by the user is not good enough for both spectral and audio processing, the user might be given the choice wether to process audio or not, to improve the framerate of the spectrum display.

4. By computing how much the Hann-window widens the main-lobe of the split-filter, and thus make the necessary reduction in the Parks-McClellan cut-off frequency to compensate, the frequency leakage between channels can be reduced.

5. By porting the FFTW-libraries to the mobile device, we might achieve significant improvements in the possible framerates as shown in tables 4.12→4.17.

6. My supervisor Gabriel Kiss also suggested using NEON optimizations[2] for the processing pipeline to further improve the possible framerates.

References

[1] Umoja - ultrasound for midwives in rural areas: Project description.

[2] ARM. Arm neon support in the arm compiler. White paper, September 2008.

[3] Pay-Yu Chen Chih-Chung Huang, Po-Yang Lee and Ting-Yu Liu. Design and implementation of a smartphone-based portable ultrasound pulsed-wave doppler device for blood flow measurement. *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, 59, No 1:182–187, 2012.

[4] Wikipedia contributors. Cooley-tukey fft algorithm. Wikipedia, The Free Encyclopedia., May 2014.

[5] Wikipedia contributors. Linear interpolation. Wikipedia, The Free Encyclopedia., May 2014.

[6] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comp.*, 19:297–301, 1965.

[7] Bjørn Rudi Dahl. Doppler spectrum generation and user interface evaluation of a low-cost ultrasound system.

[8] Candemir Doger. Configuring ves for eclipse and ndk-build. Blog. post 538 of the Kitware Blog.

[9] Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. *Proc. IEEE*, 93:216–231, 2005.

[10] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23, No 1:1–48, 1991.

[11] Jens M. Hovem. *Marine Acoustics.* Peninsula Publishing, Charles Wiseman, 2012.

[12] http://www.vtk.org/VTK/project/about.html.

[13] http://www.vtk.org/Wiki/VES.

[14] Smith. J.O. Physical audio signal processing. Online book, 2010. Linear Interpolation Frequency Response.

[15] Kjell Kristoffersen. Real time spectrum analysis in doppler ultrasound blood velocity measurement. *SINTEF Report*, 1984.

[16] Kjell Kristoffersen and Bjørn A.J. Angelsen. A time-shared ultrasound doppler measurement and 2-d imaging system. *IEEE Transactions on Biomedical Engineering*, 35:285–295, 1988.

[17] Steinar Bjærum & Hans Torp & Kjell Kristoffersen. Clutter filter design for ultrasound color flow imaging. *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, 49:204–216, 2002.

[18] C.M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, Volume:56, Issue: 6:1107 – 1108, 1968.

[19] Philippe Thévenaz Thierry Blu and Michael Unser. Linear interpolation revitalized. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 13:710–719, 2004.

[20] James H. McClellan Thomas W. Parks. Chebyshev approximation for nonrecursive digital filter with linear phase. *IEEE Transactions on Circuit Theory*, CT-19:189–194, 1972.

[21] B. Angelsen & H. Torp. Excerpt from ultrasound imaging -waves, signals and signal processing in medical ultrasonics. Vol I and Vol II, 2003.

[22] Peter D. Welch. The use of fast fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electroacoustics*, AU-15:70–73, 1967.

Appendix

A Synthetic signal for spectrum validation (Matlab)

```
%single frequency Doppler signal in noise
% now stationary bandpass signal in noise
%2014.03.07  Hans Torp
%clear all; %close all;
hr=140;%heart rate bpm
prf=4e3;
f0=5e6;
c=1540;
vNyquist=c*prf/(4*f0);
dt=1/prf;
t=0:dt:3;
Nt=length(t);
vmax=0.8*vNyquist;
v=vmax*sin(2*pi*hr/60*t');
dfi=v/vNyquist*pi;%phase increment per sample
%Note that s=exp(i*dfi*t/dt); will only work for
    constant dfi !!!!!!!!
fi=cumsum(dfi);%this is the way to do it
s=exp(1i*fi);%time varying single freq.signal
n=1/sqrt(2)*(randn(Nt,1)+1i*randn(Nt,1));
SNR=20;%dB
sn=10^(SNR/20)*s + n ;
```

B Synthetic signal for audio validation (Matlab)

```
%single frequency Doppler signal in noise
% now stationary bandpass signal in noise
%2014.03.07  Hans Torp
clear all; close all;
hr=140;%heart rate bpm
prf=4e3;
f0=5e6;c=1540;
vNyquist=c*prf/(4*f0);
dt=1/prf;
t=0:dt:3;
Nt=length(t);
vmax=0.8*vNyquist;
v=vmax*sin(2*pi*hr/60*t');
dfi=v/vNyquist*pi;%phase increment per sample
%Note that s=exp(i*dfi*t/dt); will only work for
    constant dfi !!!!!!!!
fi=cumsum(dfi);%this is the way to do it
%s=exp(1i*fi);%time varying single freq.signal
n1=1/sqrt(2).*(randn(Nt,1)+1i*randn(Nt,1));
[b1,a1]=butter(4,[0.8,0.9]);
s1=hilbert(filter(b1,a1,n1));
[b2,a2]=butter(4,[0.3,0.4]);
s2=hilbert(filter(b2,a2,n1));
%s=s.*exp(-1i*pi);
n=1/sqrt(2)*(randn(Nt,1)+1i*randn(Nt,1));
SNR=20;%dB s
sn=(s1+conj(s2))/2;%10^(SNR/20)*s + n ;
```

C Source code for division of IQ-data between sound and spectrum (C++)

```cpp
for (int timeSample = 0; timeSample < packetNumberOfTimeSamples; timeSample++) {
  // ------------------------------------------------------------------------------------------
  //save all IQ values (raw data) for this packet to a linear buffer
  timeOffset =(timeSample*DopplerSamplesPerTimepoint*DopplerBeamsPerTimepoint*2); //*2 because of audio
  for (int packetBeam = 0; packetBeam < DopplerBeamsPerTimepoint; packetBeam++) {

    //assumes data format: samples(I-Q-L-R) x beams x packetNumberOfTimeSamples
    sampleOffset = (packetBeam*DopplerSamplesPerTimepoint*2); //*2 because of audio data
    //add the complex value for the current timeSample and beam
    complexVal.Real = (double) tempIntDopplerArray[timeOffset+sampleOffset];
    complexVal.Imag = (double) tempIntDopplerArray[timeOffset+sampleOffset+1];
    DopplerIQSamples.push_back(complexVal);
  }

  // ------------------------------------------------------------------------------------------
  //Generate the Doppler spectrum lines for this packet
  if(GenerateDopplerSpectrum) {
    DopplerSpectrumGenerator->ProcessIQPacket(&DopplerIQSamples[0], DopplerIQSamples.size(),DopplerBeamsPerTimepoint, DopplerTimes.end()[-packetNumberOfTimeSamples+timeSample], DopplerGeometry.PRF);
  }

  // ------------------------------------------------------------------------------------------
  //Generate the Doppler sound for this packet
  if(GenerateDopplerSound) {
    DopplerSoundGenerator->ProcessIQPacket(&DopplerIQSamples[0], DopplerIQSamples.size(), DopplerBeamsPerTimepoint, AudioSamplingFrequency, DopplerGeometry.PRF);
  }
}
```

D Source code for the generation of the Doppler spectrum (C++)

```cpp
void vtkDopplerGenerateSpectrum::ProcessIQPacket(vtkImageComplex* iqSamples, int DopplerIQSamplesNumber, int DopplerBeamsPerTimepoint, double PacketTime, int DopplerPRFVal) {
  //cleanup and allocation of internals if needed
  if (!FFTin || !FFTout || (DopplerPRFVal != PRF)) {
    InitializationNeeded = true;
  }
  //call the initialization code if something has changed
  if (InitializationNeeded) {
    InitializeInternals(DopplerPRFVal);
  }
  //last sample position
  int SamplePosition = DopplerIQSamplesNumber - DopplerBeamsPerTimepoint + LastOffset;
  //check if there are enough smaples for smoothing
  bool smoothPossible = true;
  if ((int)SpectrumSamples.size() < SmoothingExtent*VelocitySamples)
    smoothPossible = false;
  //generate the spectrum lines for this packet
  double val = 0.0;
  while (SamplePosition+IQWindowInSamples < DopplerIQSamplesNumber) {
    //Apply Hamming window to the time samples
    for(int sample = 0; sample < IQWindowInSamples; sample++){
      FFTin[sample].Real = iqSamples[SamplePosition+sample].Real*HammingWindow[sample];
      FFTin[sample].Imag = iqSamples[SamplePosition+sample].Imag*HammingWindow[sample];
    }

    //zero padding if needed
    for(int sample = IQWindowInSamples; sample < VelocitySamples; sample++) {
      FFTin[sample] = zeroComplex;
    }

    //execute the FFT transform
    FftCompute->ExecuteFft(FFTin, FFTout, VelocitySamples);
    //midsample for FFT shifting
    int midSample = (int) (0.5*VelocitySamples);

    //additional Doppler processing [midSample...VelocitySamples)
    for(int sample = midSample; sample < VelocitySamples; sample++) {
      //spectrumline = |FFTout|^2 = Real{FFTout}^2 + Imag{FFTout}^2
      val = FFTout[sample].Real*FFTout[sample].Real+FFTout[sample].Imag*FFTout[sample].Imag;
      //convert to unsigned char and push
      SpectrumSamples.push_back(val);
      //smooth the spectrum if possible
      if (smoothPossible) {
        for (int extent = 1; extent <= SmoothingExtent; extent++ ) {
          val += SpectrumSamples.end()[-VelocitySamples*extent];
        }
        val /= SmoothingExtent+1.0;
      }
      //avoid negative values
      val = (Gain+10*log10(val+M_EPSILON))/DynamicRange;
      //clamp to 0 ... 1
      val = CLAMP_TO_UNIT(val);
      //add to smoothed values
      SpectrumSamplesSmoothed.push_back((unsigned char) (255.0*val));
    }

    //additional Doppler processing [0...midSample)
    for(int sample = 0; sample < midSample; sample++) {
      //spectrumline = |FFTout|^2 = Real{FFTout}^2 + Imag{FFTout}^2
      val = FFTout[sample].Real*FFTout[sample].Real+FFTout[sample].Imag*FFTout[sample].Imag;
      //convert to unsigned char and push
      SpectrumSamples.push_back(val);
      //smooth the spectrum if possible
      if (smoothPossible) {
        for (int extent = 1; extent <= SmoothingExtent; extent++ ) {
        val += SpectrumSamples.end()[-VelocitySamples*extent];
        }
      val /= SmoothingExtent+1.0;
      }
      //avoid negative values
      val = (Gain+10*log10(val+M_EPSILON))/DynamicRange;
      //clamp to 0 ... 1
      val = CLAMP_TO_UNIT(val);
      //add to smoothed values
      SpectrumSamplesSmoothed.push_back((unsigned char) (255.0*val));
    }

    //update the time vector first initialize with the packet time then increment with the actual skip time
    if (SpectrumTimes.size() <= 0)
      SpectrumTimes.push_back(PacketTime);
    else
      SpectrumTimes.push_back(SpectrumTimes.back() + IQSkipInSamples/PRF);

    //move to the next position by adding the skip value
    SamplePosition+= IQSkipInSamples;
  }

  //overlap offset for the next packet
  LastOffset = SamplePosition-DopplerIQSamplesNumber;
}
```

E Source code for audio generation (C++)

```cpp
void vtkDopplerGenerateSound::ProcessIQPacket(vtkImageComplex* DopplerIQSamples, int DopplerIQSamplesNumber, int DopplerBeamsPerTimepoint, int AudioSamplingFrequencyVal, int DopplerPRFVal) {

  //sanity check: check if there are at least DopplerBeamsPerTimepoint Doppler IQ samples
  if (DopplerIQSamplesNumber < DopplerBeamsPerTimepoint)
    return;

  //detect if we are at the start of a new processing setup
  double SamplePosition;
  if ((DopplerIQSamplesNumber == DopplerBeamsPerTimepoint) || (AudioSamplingFrequency != AudioSamplingFrequencyVal) || (DopplerPRF != DopplerPRFVal)) {
    //set the default values
    DopplerPRF = DopplerPRFVal;
    AudioSamplingFrequency = AudioSamplingFrequencyVal;
    InterpolationIncrement = DopplerPRF/AudioSamplingFrequency;
    LastInterpolationPosition = 0.0;
    SamplePosition = 0.0;
    AudioSplitFilter.clear();
    AudioSplitFilterConjugate.clear();

    //Designing the split filter as a lowpass-filter at prf/fs/2
    ParksMcClellan parksMcClellan(AudioCoefficientsNumber, InterpolationIncrement/2, 0, 0.03, LPF);

    //Generate sidelobe-reduction-window
    GenerateHanningWindow();
    //Mixing the different coefficients up, so we now have a lowpass-filter from 0->prf/fs;
    //The complex conjugate is a lowpass-filter from -prf/fs->0.
    //Also multiplying with the Hann window, to reduce sidelobe levels.
    vtkImageComplex val;
    for(int i=AudioCoefficientsNumber-1; i >= 0; --i) {
      val.Real = parksMcClellan.FirCoeff[i]*cos(i*M_PI*InterpolationIncrement/2)*HanningWindow[i];
      val.Imag = parksMcClellan.FirCoeff[i]*sin(i*M_PI*InterpolationIncrement/2)*HanningWindow[i];
      AudioSplitFilter.push_back(val);
      val.Imag = -val.Imag;
      AudioSplitFilterConjugate.push_back(val);
    }
    //clear the interpolated values
    InterpolatedIQSamples.clear();
  } else {
    SamplePosition = DopplerIQSamplesNumber - DopplerBeamsPerTimepoint -1 + LastInterpolationPosition;
  }

  //process the packet
  vtkImageComplex interpolatedVal;
  vtkImageComplex audioVal;
  while (SamplePosition < DopplerIQSamplesNumber - 1) {

    //interpolate the complex data
```

```
      interpolatedVal = InterpolateValue(DopplerIQSamples,
          SamplePosition);

      //update the interpolated IQ samples
      InterpolatedIQSamples.push_back(interpolatedVal);

      //generate the audio sample for the last value inserted and
          split the signal by applying the audio filter and its
          conjugate
      audioVal = GenerateAudioSample();

      //convert and push to audio data
      SoundData.push_back((short)(CLAMP_TO_SHORT(audioVal.Imag)));
      SoundData.push_back((short)(CLAMP_TO_SHORT(audioVal.Real)));

      //compute a new interpolation position
      SamplePosition += InterpolationIncrement;
    }

    //sample position for the next packet
    LastInterpolationPosition = SamplePosition - (
        DopplerIQSamplesNumber-1);

    //clean up in the interpolated samples only keep the needed ones
    int offset = InterpolatedIQSamples.size() - AudioSplitFilter.size
        ();

    if (offset > 0) {
      InterpolatedIQSamples.erase(InterpolatedIQSamples.begin(),
          InterpolatedIQSamples.begin()+offset);
    }

    //send the data to the audio player
    #ifdef ANDROID_BUILD
      audioPlayer->sendDataToAudioPlayer (&SoundData[0], SoundData.
          size());
    #endif

    //clear the sound vector
    SoundData.clear();
  }
```

F Smoothing methods comparison (Matlab)

```
%% powerspect. smooth
%2014  Hans Torp

N=50000;
M=5;%number of points averaging
x=randn(M,N)+i*randn(M,N);
Pest=squeeze(mean(abs(x.^2)));
dBPest=10*log10(Pest);dBPest=dBPest/mean(dBPest);
dBLogav=squeeze(mean(20*log10(abs(x))));
dBLogav=dBLogav/mean(dBLogav);

hist([dBPest;dBLogav]',200); legend('dBPest','dBLogav')
    ;
disp([std(dBPest),std(dBLogav)]);
```

G Difference image

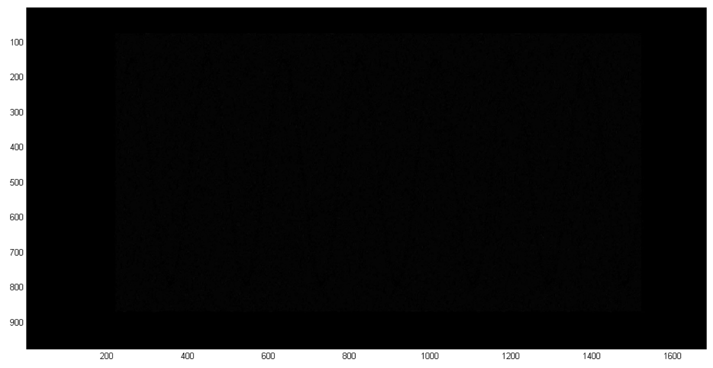


Figure G.1: Original difference image before scaling

H Raw-data from the spectrum timing

FFT-size\PRF	2890					Average
64	77.231	80.690	74.056	69.132	79.524	76.126
100	121.881	129.442	110.850	107.830	128.414	119.683
128	127.500	129.007	122.597	134.066	128.414	128.317
200	175.802	122.605	130.507	116.553	129.346	134.963
256	236.319	242.975	242.976	242.342	252.834	243.489

FFT-size\PRF	2890					Average
64	28.920	31.865	28.514	27.334	28.009	28.928
100	57.175	60.164	52.770	50.370	60.496	56.195
128	49.740	51.120	49.277	51.599	53.354	51.018
200	67.897	49.555	49.192	46.009	50.603	52.651
256	101.978	105.718	103.197	105.604	107.975	104.894

Table H.1: Nexus 10 spectral measurements for a PRF of 6410Hz and overlap of 75%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.

FFT-size\PRF	6410					Average
64	70.781	81.928	78.170	85.817	78.4730	78.077
100	117.434	120.611	121.245	132.74	123.561	123.118
128	145.278	123.868	135.067	124.031	126.036	130.856
200	240.093	224.974	210.065	237.22	219.212	226.313
256	236.269	227.58	234.962	259.133	254.162	242.421

FFT-size\PRF	6410					Average
64	26.743	28.996	29.207	28.3	27.141	78.074
100	53.338	57.163	54.648	63.073	58.152	57.275
128	57.664	47.277	48.749	48.545	49.991	50.445
200	111.135	113.241	95.396	109.799	108.085	107.531
256	99.267	93.214	97.274	112.438	111.715	102.782

Table H.2: Nexus 10 spectral measurements for a PRF of 2890Hz and overlap of 75%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.

FFT-size\PRF	2890					Average
64	134.771	137.26	144.42	135.532	136.47	137.691
100	229.193	221.451	224.85	222.908	232.507	226.182
128	238.155	252.844	237.48	256.579	255.113	248.034
200	422.018	454.673	425.749	415.583	448.78	433.361
256	482.756	481.985	444.746	449.188	454.612	462.657

FFT-size\PRF	2890					Average
64	53.962	53.811	53.593	51.407	52.334	53.021
100	107.957	105.258	105.357	105.469	110.687	106.946
128	93.015	104.68	91.962	109.349	101.304	100.062
200	203.945	206.736	209.056	203.647	218.424	208.362
256	210.29	206.263	185.447	185.083	185.598	194.536

Table H.3: Nexus 10 spectral measurements for a PRF of 2890Hz and overlap of 87.5%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.

FFT-size\PRF	6410					Average
64	160.151	140.297	140.347	140.118	143.27	144.837
100	211.075	218.625	225.282	235.166	231.3160	224.293
128	246.582	233.121	245.607	248.27	247.405	244.197
200	418.038	430.757	409.382	429.61	400.323	417.622
256	443.735	503.987	465.69	448.419	456.913	463.749

FFT-size\PRF	6410					Average
64	58.983	54.427	53.167	52.349	53.242	54.434
100	99.283	102.516	104.969	113.408	108.1	105.655
128	99.724	91.097	95.701	98.972	95.306	96.160
200	205.116	207.941	192.841	207.356	191.247	200.9
256	182.636	210.401	193.543	188.134	191.798	193.302

Table H.4: Nexus 10 spectral measurements for a PRF of 6410Hz and overlap of 87.5%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.

FFT-size\PRF	2890					Average
64	114.198	122.4	115.529	146.945	121.737	124.162
100	221.093	211.894	215.283	247.930	200.528	219.346
128	292.431	264.035	271.211	280.995	339.331	289.601
200	517.402	455.243	390.913	423.358	405.641	438.511
256	434.769	454.427	492.796	435.618	442.872	452.096

FFT-size\PRF	2890					Average
64	54.799	55.602	53.867	80.933	52.375	59.515
100	121.269	107.198	130.38	135.53	117.845	122.444
128	148.013	129.21	128.89	130.557	153.703	138.075
200	286.649	261.238	225.759	257.599	234.78	253.205
256	227.556	213.649	254.12	210.219	215.561	224.221

Table H.5: ASUS T300 spectral measurements for a PRF of 2890Hz and overlap of 75%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.

FFT-size\PRF	6410					Average
64	128.646	124.968	130.755	134.192	116.33	126.978
100	234.485	242.391	237.093	232.158	222.709	233.767
128	358.471	293.122	282.264	289.503	292.08	303.088
200	428.955	403.03	448.207	408.023	412.093	420.062
256	459.057	431.908	479.285	444.885	411.917	445.41

FFT-size\PRF	6410					Average
64	48.971	54.256	55.064	64.975	47.962	54.246
100	118.951	136.235	129.076	131.264	126.571	128.419
128	169.377	143.724	137.716	142.587	136.305	145.942
200	258.803	244.58	235.774	244.129	234.729	243.603
256	209.98	215.82	247.261	223.149	195.017	218.245

Table H.6: ASUS T300 spectral measurements for a PRF of 6410Hz and overlap of 75%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.

FFT-size\PRF	2890					Average
64	223.737	262.367	216.606	217.997	218.93	227.927
100	406.866	416.525	448.958	474.03	445.483	438.3742
128	475.702	418.26	414.506	458.491	428.781	439.148
200	823.221	849.845	812.236	823.668	853.303	832.271
256	866.52	905.122	918.027	906.166	880.52	895.271

FFT-size\PRF	2890					Average
64	103.365	93.538	96.664	94.098	100.429	97.619
100	231.944	229.26	258.171	282.371	262.586	252.866
128	242.416	186.348	193.767	200.013	208.902	439.148
200	484.766	505.021	457.784	479.663	483.261	832.455
256	429.144	420.691	498.261	459.474	438.25	895.271

Table H.7: ASUS T300 spectral measurements for a PRF of 2890Hz and overlap of 87.5%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.

FFT-size\PRF	6410					Average
64	341.01	230.295	229.758	233.536	217.913	250.502
100	439.225	451.236	435.192	418.6	419.123	432.675
128	406.069	413.401	404.36	445.317	516.775	437.184
200	827.078	804.775	794.268	802.558	765.991	798.934
256	869.833	839.458	846.402	844.023	855.43	851.029

FFT-size\PRF	6410					Average
64	256.375	111.506	113.253	107.082	91.782	136
100	118.951	261.294	226.231	230.866	243.118	216.092
128	185.903	189.833	194.362	206.661	244.363	204.224
200	480.569	468.514	448.027	460.616	440.324	459.61
256	423.497	413.568	426.833	410.612	415.607	418.024

Table H.8: ASUS T300 spectral measurements for a PRF of 6410Hz and overlap of 87.5%, the top table depics the full run-times while the bottom table is the FFT run-times. All measurements are given in miliseconds.