Scalable Audio Processing Across Heterogeneous Distributed Resources

An Investigation into Distributed Audio Processing for Music Information Retrieval

Ahmad Al-Shakarchi
School of Computer Science, Cardiff University

Declaration

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed .... (candidate) Date ....

STATEMENT 1

This thesis is being submitted in partial fulfillment of the requirements for the degree of Ph.D.

Signed .... (candidate) Date ....

STATEMENT 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references.

Signed .... (candidate) Date ....

STATEMENT 3

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate) Date ....

Acknowledgements

The work presented in this thesis could not have been completed without the help and support of various researchers, scientists, family members and close friends that I am fortunate enough to have in my life.

I would like to thank the Triana developers and researchers that I have worked closely with during my time as a PhD student, and that built the foundation for my work; Dr. Matthew Shields, Dr. Andrew Harrison, and Dr. Ian Wang. Extra special thanks are extended to Kieran Evans at Cardiff University for his help and expertise - I am eternally indebted!

I am also particularly grateful for the assistance of researchers and developers at various institutes around the world that helped me conduct such large scale experiments - researchers at: Laboratoire de l'Accélérateur Linéaire (Paris, France), SZTAKI (Hungarian Academy of the Sciences, Hungary), the Information Sciences Institute at the University of Southern California and the Center for Computation and Technology at Louisiana State University (USA), the University of Coimbra (Portugal), and not forgetting researchers and staff at Cardiff University. I would like to thank the Computer Science and Informatics department at Cardiff for allowing me the opportunity to perform this research.

I would - of course - like to extend a very special 'thank you' to my supervisor Dr. Ian Taylor, for his patience, his ongoing dedication to my work, his vision and superior intellect (!), and most of all his friendship throughout my PhD. Thank you for all the cool opportunities you've provided me! I could not have asked for a better and more understanding supervisor and am happy to have developed a great friend over the years. Rock on!

To my label mates and closest associates; Dan, Ruff, Ralph, Mud and Huw, who may not know too much about my academic career, but whose antics have provided me with the levity required to maintain my sanity[1] over the years. Thank you!

To Sarah, who sacrificed so much and showed massive commitment, love and support to me. You helped me more than you know and I will never forget it. I'm also indebted to the entire Kingman family - especially Roger and Gaynor - for their support during hard times.

Finally, I would like to thank my family - Ali, Israa, Khitam, Mazin - for supporting me in every way imaginable; I know I asked too much of you all and hope I can repay you with unwavering love, support and patience, in turn. Especially to my father, for whom all of my work and perseverance was really for. Thank you, dad.

[1]Relatively speaking.

Abstract

Audio analysis algorithms and frameworks for Music Information Retrieval (MIR) are expanding rapidly, providing new ways to discover non-trivial information from audio sources, beyond that which can be ascertained from unreliable metadata such as ID3 tags. MIR is a broad field and many aspects of the algorithms and analysis components that are used are more accurate given a larger dataset for analysis, and often require extensive computational resources.

This thesis investigates if, through the use of modern distributed computing techniques, it is possible to design an MIR system that is scalable as the number of participants increases, which adheres to copyright laws and restrictions, whilst at the same time enabling access to a global database of music for MIR applications and research. A scalable platform for MIR analysis would be of benefit to the MIR and scientific community as a whole.

A distributed MIR platform that encompasses the creation of MIR algorithms and workflows, their distribution, results collection and analysis, is presented in this thesis. The framework, called DART - Distributed Audio Retrieval using Triana - is designed to facilitate the submission of MIR algorithms and computational tasks against either remotely held music and audio content, or audio provided and distributed by the MIR researcher. Initially a detailed distributed DART architecture is presented, along with simulations to evaluate the validity and scalability of the architecture. The idea of a parameter sweep experiment to find the optimal parameters of the Sub-Harmonic Summation (SHS) algorithm is presented, in order to test the platform and use it to perform useful and real-world experiments that contribute new knowledge to the field.

DART is tested on various pre-existing distributed computing platforms and the feasibility of creating a scalable infrastructure for workflow distribution is investigated throughout the thesis, along with the different workflow distribution platforms that could be integrated into the system. The DART parameter sweep experiments begin on a small scale, working up towards the goal of running experiments on thousands of nodes, in order to truly evaluate the scalability of the DART system.

The result of this research is a functional and scalable distributed MIR research platform that is capable of performing real world MIR analysis, as demonstrated by the successful completion of several large scale SHS parameter sweep experiments across a variety of different input data - using various distribution methods - and through finding the optimal parameters of the implemented SHS algorithm. DART is shown to be highly adaptable both in terms of the distributed MIR analysis algorithm, as well as the distribution mechanism used.

Contents

Chapter 1

Introduction

1.0.1 Overview

"Myriad difficulties remain to be overcome before the creation, deployment, and evaluation of robust, large-scale, and content-based Music Information Retrieval (MIR) systems become reality. The dizzyingly complex interaction of music's pitch, temporal, harmonic, timbral, editorial, textual, and bibliographic 'facets' for example, demonstrates just one of MIRs perplexing problems" [1]
- J. Stephen Downie, Music Informtation Retrieval, Chapter 7, Annual Review of Information Science and Technology, 2003

Since the creation of the Compact Disc in 1982, music has been widely available in digital formats, and the success of the MP3 as a media format has meant that the volume of digital music available is expanding rapidly. The introduction of Napster[1] in 1999, Kazaa[2] in 2001, and other file sharing applications based on the Gnutella [2] distributed peer-to-peer framework gave users (illegal) access to an enormous library of music of all genres. This, along with other technologies such as Apple's iPod and iTunes Music Store has made digital music a convenient part of everyday life for the majority of music listeners. The iTunes Music Store opened in 2003 with over 200,000 songs available for purchase. As of April 2008, it became the number-one music vendor in the United States [3], providing over 10 billion song downloads in just under seven years [4]. iTunes now accounts for 70% of all worldwide online digital music sales, making the service the largest music retailer,

[1]`http://www.napster.com`
[2]`http://www.kazaa.com/`

and 28% of the overall US music retail market [5]. Coupled with advances in consumer computing technology and data storage, everyday music 'users' can now have a personal collection consisting of hundreds of hours of high quality digital music.

The sheer volume of purchased and downloaded music files reveals how music is both commercially and culturally important to us. The ever increasing availability of music in digital format requires the development of tools for music accessing, filtering, classifying, and retrieval of information [6][7]. *Music Information Retrieval* (MIR) is the interdisciplinary science of retrieving non-trivial information from music and audio. Audio analysis algorithms and frameworks for MIR are expanding rapidly, providing new ways to garnish this information from audio sources - well beyond that which can be ascertained from metadata such as ID3 tags (containing information such as the title, artist, album title, track number, and other information about the file to be stored). MIR research is generally focussed around these 'content-based' approaches, instead of merely analysing and indexing the metadata supplied by the creators of the files, which can prove to be unreliable and is often missing. No current search engines such as Google or Yahoo implement any content based audio search facilities - even 'specialist' music recommendation services such as Pandora[3] and Last.fm[4] (discussed in Chapter 2) employ only statistical analysis derived from embedded metadata when recommending music to listeners. Interestingly, the original MP3 format (and the current Wav or Aiff CD formats) contained no metadata fields until the format was revised. These metadata fields are not mandatory and Compact Discs cannot be automatically encoded with the (usually) correct metadata via the online Compact Disc Database (CDDB) if no internet connection is available.

The sheer volume of music available world-wide in expansive digital collections further aggravates a fundamental issue which remains problematic in the field of MIR - few effective information retrieval techniques exist to analyse and tackle the information in these collections [8] and due to the wide variety of ways that music is produced, represented and consumed, MIR is an extremely challenging area of research [9]. [8] outlines several interdisciplinary research issues in the field of MIR, including issues restricting data sharing and comparative evaluation.

Due to copyright law, music cannot be freely shared and the purchase of every single

[3]http://www.pandora.com/
[4]http://www.last.fm

piece of music that a researcher needs or wishes to analyse is often prohibitively expensive. Creative Commons[5][6] collections are available such as the Free Music Archive[7], AudioFarm[8] and Jamendo[9]. Although useful, these collections will often only reflect a small proportion of the entire scope of 'useful' audio or music files, and using specific collections can lead to *algorithm tailoring*. As such, measuring the validity of MIR algorithms and techniques has been difficult at best.

These issues were amongst the motivations that led to the establishment of MIREX [10]. The Music Information Retrieval Evaluation eXchange (MIREX) is an annual evaluation campaign for MIR algorithms, hosted by the International Music Information Retrieval Systems Evaluation (MIRSEL) Laboratory [11], part of the University of Illinois. The objective of this project is the establishment of the necessary resources for scientifically valid development and the evaluation of emerging MIR and Music Digital Library (MDL) techniques and technologies. MIREX applies a 'TREC approach' [12], where researchers and members of the MIR community can submit algorithms and applications to complete tasks that have been defined by the MIR community, which are in turn evaluated using standardised queries and evaluation techniques.

While MIREX has proven to be a valuable tool for the MIR community, there are several issues with this format of research. [13] lists several of the ongoing issues and challenges faced by MIREX, such as; a relatively small (approximately 4TB) database of music 'when compared to industrial-scale real-world problems', and the 'need to interpret the contest results achieved with care' because of this. Unlike TREC, the MIREX dataset for each task cannot be freely distributed to the participants, as explained in [10]:

> *The primary reason for the lack of freely available datasets is the current state of musical intellectual property copyright enforcement. The constant stream of news stories about the Recording Industry Association of America (RIAA) bringing lawsuits against those accused of sharing music on peer-to-peer networks has had a profoundly chilling effect on MIR research and data sharing.*

[5]Creative Commons is a nonprofit organisation that develops, supports, and stewards legal and technical infrastructure that maximises digital creativity, sharing, and innovation.

[6]`http://creativecommons.org/`

[7]`http://freemusicarchive.org/`

[8]`http://audiofarm.org/`

[9]`http://www.jamendo.com/`

This means that participants in MIREX must deliver their algorithms to be executed by the IMIRSEL team on these datasets, thus placing a large burden on the team. [10] lists several other issues that burden the IMIRSEL team, such as acquisition of new test data, data management and capacity issues for both test data and new intermediate data generated by the algorithms, and the management of the submitted algorithms, which can be submitted in a variety of programming languages, and across different platforms. Submissions must also be executed within the same eight week period each year. Although MIREX has proven to be a very valuable tool, it is clear that the job of managing a large scale MIR framework can be extremely difficult.

1.0.2 Hypothesis and Research Question

As discussed, MIR is a broad field and many aspects of the algorithms and analysis components that are used in MIR are more accurate, or can be awarded a higher level of confidence, given a larger dataset (or many different smaller datasets). The content-based analysis components of MIR that analyse the audio data can also be computer-resource intensive.

Hypothesis; through the use of modern distributed computing techniques, it is possible to design an MIR system that is scalable as the number of participants increases, which adheres to copyright laws and restrictions, whilst at the same time enabling access to a global database of music for MIR applications and research. In the context of DART, a *scalable* can be considered an improvement of performance (i.e. faster overall job execution times) as the as number of participants increase. When more workers are introduced to the system with no resulting performance benefit, then the system is no longer scalable.

A distributed implementation of an MIR research platform would provide access to potentially millions of MP3 files on target machines where files could be analysed locally, transferring back only the metadata/results of the analysis, reducing overall computation time. This solves issues with copyright, avoids bandwidth and resource restrictions, and potentially allows for more refined MIR algorithms. A scalable platform for MIR analysis would be of benefit to the MIR and scientific community as a whole.

1.0.3 Approach

To test the validity of this hypothesis, a distributed MIR platform which encompasses the creation of MIR algorithms and workflows, their distribution, results collection and analysis, is presented in this thesis. The framework, called *DART* - Distributed Audio Retrieval using Triana - is a distributed processing platform that is designed to facilitate the submission of MIR algorithms and computational tasks against either remotely held music and audio content, or audio provided and distributed by the MIR researcher. A future goal of DART involves the creation of a Music Recommendation System, giving the users an incentive to allow DART to use their computational resources. DART represents a fusion of emerging technologies, such as graphical workflow design software, audio processing techniques and algorithms, and internet scale distributed technologies such as grid computing and peer-to-peer technologies, which have grown through the development of popular applications targeted at specific services, such as Napster, Gnutella, and CPU sharing systems like SETI@Home[10] [14].

Initially, a detailed peer-to-peer distributed DART architecture is presented, along with detailed simulations to evaluate the validity and scalability of the design. A DART MIR analysis algorithm is created and used in order to replicate some of the traits of a 'standard' MIR experiment, and the idea of a parameter sweep experiment in order to find the optimal parameters of the Sub-Harmonic Summation algorithm is presented. This is tested on various distributed computer platforms and the feasibility of creating a scalable infrastructure for workflow distribution is investigated throughout the thesis, along with different workflow distribution platforms that could be integrated into the system. The DART parameter sweep experiments begin on a small scale, working up to the goal of running experiments on hundreds of thousands of nodes, in order to truly evaluate the scalability of the DART system.

1.0.4 Contributions to the State of the Art

In summary, the main contributions of the work represented in this thesis are:

- The outline for the design of a high-level, highly scalable peer-to-peer architecture

[10]`http://setiathome.berkeley.edu/`

enabling the execution of MIR and audio analysis algorithms on a large scale, as outlined in Chapter 3 and publications [15], [16], [17].

- The creation of the DART Execution Environment (DEE), which is able to take workflows or algorithms created in Triana, and enable them to be packaged into a lightweight Java JAR that can be distributed using a variety of distribution platforms. This shows an extension of the state of the art - DART is novel as it allows easy configuring of the algorithms by i) utilising Triana as a workflow platform to create the algorithms required and utilising/modifying pre-existing tools in Triana and ii) creating new units/tools to allow for further research to take place.
- The creation of a distributed MIR research platform - DART - evaluated by the execution of a large scale proof-of-concept parameter sweep experiment in order to find the optimum parameters of the Sub-Harmonic Summation pitch detection algorithm.
- The results of the parameter sweep experiments presented in Chapter 7 suggest the optimal parameters for the implemented SHS pitch detection algorithm across a range of input data.
- The investigation and comparison of the suitability of two open-source middleware platforms that are designed for distributed computing using GRID or volunteered computing resources, for the application of distributed (audio) processing/computing (BOINC & XtremWeb). Both platforms allowed for a massive reduction in overall processing time, as documented in Chapter 7. The simplification of the integration of an MIR testbed (Triana/DART) with a distributed computing platform that allows for the analysis of vast amounts of data without worry of copyright issues.
- The work with the XtremWeb team has pushed forward development and highlighted many issues with the XtremWeb platform. As documented in Chapter 7 the DART experiments were by far the largest scale experiments run on an XtremWeb platform. These experiments highlighted problems which were addressed by the XtremWeb team and made the software much more scalable, reliable and useable for large scale distributed computing.
- Chapter 8 and the entire SHS parameter sweep experiment concept shows that the DART platform allows for experiments, ideas, and research to be refined and revised,

completely independently of the distribution platform used. The work with the Pegasus team has contributed key results for [18] published in the Journal of Grid Computing (*A Case Study into Interoperable Monitoring and Analysis for Scientific Workflows*).

- The development of a platform that can be used by the scientific and MIR communities as a testbed for further research and investigation
- Simplifying the integration of an MIR testbed (Triana) with distributed computing platforms that allows for the analysis of vast amounts of data without worry of copyright issues
- The potential for the creation of a Music Recommendation System which offers everyday music 'users' an *incentive* to join the project, while allowing scientific research to take place, opening up vast resources to MIR scientists, again without the worry of law and copyright issues.

1.0.5 Project History

The majority of the research work considered in this thesis was carried out during the period from 2005-2009. This field of research is particularly fast moving, with advances made in a wide range of areas, including middleware infrastructure. Although the author is fully aware of work since the end of 2009, this document discusses the research hypothesis in the context of that period.

1.0.6 Layout & chapter plan/thesis outline

Chapter 2 discusses the background necessary to understand the thesis in detail, focussing on the basics of digital audio and the representations that allow its analysis, a review of current of Music Information Retrieval and Music Recommendation Systems, and also details the Triana workflow software which forms the basis of the DART algorithm design.

Chapter 3 gives a high level overview of DART, and explains the underlying architecture of the distributed DART system. A DART peer-to-peer case study is performed, showing the design of such a system, detailed simulations to test its scalability, and also a discussion on the system's feasibility. This chapter serves as a design for extracting some of the necessary features and requirements needed in DART, and also includes background information on relevant distributed computing technologies and reviews related works and literature.

Chapter 4 describes the final design choices that were made for DART, outlining the design of the specific Triana units which make up the DART algorithm. The design strategy used to port the units over to standalone JARs and create the DART Execution Environment, as well as the design choices for each iteration of the DART experiments (4 versions) are also described.

Chapter 5 describes the implementation of the DART system, including the Triana units, the Sub-Harmonic Summation algorithm in detail, and the details of the XtremWeb and BOINC deployment mechanisms. This chapter also details the various scripts which send, monitor, and analyse the results and look for optimal parameters of the DART pitch detection algorithm.

Chapter 6 reports the results of the initial set of DART experiments - running the DART pitch detection on an audio file 50 times with fixed parameters. This was run locally on an unconnected system, on an XtremWeb desktop grid with 5 machines, and also using the XtremWeb-EGEE bridge. The aim of these experiments is to give some indication of the performance of the DART algorithm/application on comparative platforms and in realistic scenarios, as well as to identify limitations of the current implementation.

Chapter 7 displays the results of all the large scale DART parameter sweep experiments, showing the effect of varying the parameters on the accuracy of the DART Sub-Harmonic Summation algorithm. Graphs and tables are presented and the results are discussed thoroughly. This chapter also includes a 'time analysis' section, which compares the length of time taken to submit and retrieve all jobs on both the XtremWeb and BOINC platforms, and a discussion into the advantages and disadvantages of working with both platforms.

Chapter 8 shows the results of running further DART parameter sweep experiments using the Pegasus platform, in order to extend the range of parameters analysed when using the XtremWeb and BOINC distributed computing platforms. This chapter highlights any modifications required of DART to run on the Pegasus platform, and shows DART's platform independence.

Chapter 9 summarises the conclusions of this thesis and highlights its contributions, outlines potential future work, as well as discussing more recent MIR literature and how this may affect the context of the thesis.

Chapter 2

Background

2.1 Overview

The purpose of this chapter is to familiarise the reader with concepts that are useful to understand before discussing the design or implementation carried out in order to test the hypothesis presented in this thesis. This thesis and indeed the MIR field in general, touch on aspects from many different disciplines. In order to understand this thesis' research and experiments, an overview of the concepts and technologies are explained, with references given to sources that explain the subjects beyond the scope possible in this chapter, if further reading is required.

This section begins by presenting an overview of the basic facets of sound and music that are relevant to this thesis, and continues on to explain the ideas behind the digital representation and manipulation of sound, including a basic introduction to the frequency domain and Fourier Theory. An overview of the MIR field is given, including a summary of the current challenges faced by the MIR community and a review of distributed MIR platforms and current Music Recommendation Systems.

The Triana workflow software will be introduced and discussed, giving the reader a firm basis of understanding into concepts that are discussed later on; Triana forms the basis of the DART algorithm and an overview of the audio tools in Triana (developed by the author) is presented. Finally, related music and data-flow tools such as MARSYAS, M2K, CLAM and OMRAS2 are reviewed and compared with Triana, and the advantages of using Triana are highlighted.

2.2 Review of Musical Concepts

Before discussing concepts pertaining to Music Information Retrieval and MIR techniques, it is useful to briefly review some musical concepts and terms. As suggested by Downie in [1], it is helpful to think of music as consisting of several basic 'features' (or *facets*), each of which plays a variety of roles in defining the MIR domain. The following is a brief description of some of the features that are relevant to the topics covered in this thesis.

2.2.1 Sound

Sound can be defined as: *"A travelling wave which is an oscillation of pressure transmitted through a solid, liquid, or gas, composed of frequencies within the range of hearing and of a level sufficiently strong to be heard, or the sensation stimulated in organs of hearing by such vibrations"* [1].

2.2.2 Pitch

> *"Pitch is the perceived quality of a sound that is chiefly a function of its fundamental frequency in the number of oscillations per second" [19]*

Musical instruments, with the exception of some percussion instruments, produce relatively 'periodic' vibrations through the air. The sounds produced are the result of the combination of different wave frequencies, which are all multiple integers of a 'fundamental frequency', usually called *F0*. Pitch is related to the perception of the fundamental frequency and can range from low to high. The 'fundamental frequency' of a sound wave is the lowest frequency in the waveform.

[1]The American Heritage Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company, 2006.

Other significant frequencies in a sound wave are 'overtones'[2]. of the fundamental frequency, which may include *harmonics* and *partials.* A harmonic is an integer multiple of the fundamental frequency; a 100Hz tone could have harmonics of 200Hz, 300Hz, 400Hz, and so on. In naturally occurring vibrations there is a harmonic at each multiple of the fundamental frequency - theoretically all the way up to infinity. Most instruments produce harmonics, but many instruments produce partials and inharmonic tones, such as drums, cymbals and other indefinite-pitched instruments. It is difficult for humans to perceive a sound as separate tones, especially since the amplitude of the fundamental frequency often outweighs the amplitude level of the harmonics; the harmonics decrease in amplitude as the frequency rises. However, harmonics are present in nearly all 'natural' sounds and add a uniqueness that helps to define the sound.

For example the same note (e.g. A4 or the note with a fundamental frequency of 440Hz) played on an acoustic guitar and an oboe will sound different due to the harmonics produced by each instrument, thus affecting the *timbre* (explained shortly). A guitar string can vibrate in a simple back-and-forth motion, or vibrate in more complex ways where a part of the string is moving in the opposite direction from its neighbouring strings. Waves that occur naturally when a guitar string is plucked involve many kinds of vibrations, with each type of vibration producing a simple wave with its own frequency and amplitude, for example:

1. The frequency of the simple wave produced by the simplest back-and-forth motion - the fundamental frequency and perceived as the *pitch* or *note*
2. The frequency of the wave produced by the second mode of vibration (where the string is vibrating in halves) is twice the fundamental frequency, or exactly one octave higher.
3. The frequency produced by the third mode of vibrations (where the string is vibrating in thirds) is three times the fundamental frequency.

Each of the higher-frequency simple waves is a harmonic, and these multiples of the fundamental frequency form the harmonic series. A ***sub-harmonic*** is any harmonic that has a frequency that is a fraction of the fundamental frequency, with a ratio of $1/x$.

[2]An overtone is any frequency higher than the fundamental. The fundamental and the overtones together are 'partials'. Harmonics are partials whose frequencies are whole number multiples of the fundamental

For example, a 100Hz tone would have sub-harmonics of 50Hz, 25Hz, 12.5Hz and so on, mirroring the harmonic series of the fundamental frequency.

With relevance to music, pitch has a range of approximately 20 to 5000Hz, similar to the range of the fundamental frequencies of piano strings and organ pipes. Tones with higher frequencies are audible but without definite pitch sensation. Low tones in the range of 10 to 50 Hz are often felt through the body more than they are heard. The transition from the perception of single pulses to a real pitch sensation is gradual and pitch can be perceived after very few periods of the sound wave have been presented to the ear.

When two or more pitches sound at the same time a 'harmony' or 'chord' is created. This is also known as *polyphony*. Some instruments (such as the voice) are *monophonic* and can only create one pitch at a time; more than one voice or instrument is required in order to create harmonies. A chord may have different overall qualities depending on the pitch of the different sounds and in particular, on the distances between them.

2.2.3 Intensity

The intensity of a sound is related to the amplitude of the waveform. The larger the amplitude of the wave, the louder or more intense the sound appears to be. It should be noted however, that our ears do not recognise pitch and the intensity of the pitch linearly due to the Fletcher-Munson loudness contour of the ear. Yet, these two perceptually relevant qualities of sounds can be reasonably approximated by considering the fundamental frequency and the energy of a sound.

2.2.4 Timbre

The timbre of a sound is defined as the sound characteristics or 'colour' that allow listeners to perceive two sounds with the same pitch and intensity, as different. This view was first stated by Helmholtz over a century ago and is reflected by the definition of timbre according to the American Standards Association (*Acoustic Terminology*, 1960):

> *"[Timbre is] that attribute of auditory sensation in terms of which a listener can judge that two steady-state complex tones having the same loudness and pitch are dissimilar"*

This (somewhat catchall) term can be illustrated when considering the difference between the same note played on a guitar and a piano. If two differing sounds have the same pitch and same loudness (intensity), then they must differ in timbre.

Further to the harmonic content's contribution to the timbre of a sound, it is also greatly affected by its *envelope*. The envelope is the overall amplitude structure of a sound and is defined by it's attack time and characteristics, decay, sustain, release (ADSR envelope) and transients. These controls are common place in many synthetic or electronic instruments (synthesisers) in order to allow the musician to shape the sound, changing its timbre. For example, when removing the attack portion from a piano, it becomes more difficult to identify the instrument correctly, since the sound of the piano's hammer hitting the strings forms the attack portion that is highly characteristic of the sound that is recognised to be a piano.

With many percussive instruments there is no fundamental frequency and the sound could be called *noise*. Yet, noises are perceived to be in a low, medium, or high register. Intensity and timbre are still relevant descriptors for noises.

2.3 Digital Audio

2.3.1 Digital Signal Processing

The fields of science and engineering often require the use and analysis of *signals*. Digital Signal Processing (DSP) is the science of using computers to understand these types of data to achieve a wide variety of goals, such as speech recognition, data compression, neural networks, audio processing and much more.

Digital Signal Processing, as the term suggests, is the processing and manipulation of signals by digital means. A signal in this context can mean a number of different things. More directly, it can be defined as a *physical quantity that is a function of one or more independent variables such as time, distance, temperature or pressure*. The variation of a signal's amplitude as a function of the independent variable(s), is its *waveform*.

2.3.2 Analog and digital signals

A signal is a mathematical function of one or more independent variables that usually represent time and/or space. The *independent variable* of a signal can be either continuous or discrete. A continuous-time signal (for example) is defined at every instant of time; a discrete-time signal is only defined at specific instants. When both amplitude and time are continuous, then the signal is analog. This case corresponds to the kind of signals that occur naturally in the physical world, such as sound and light. Any analog signal must be converted into digital (i.e. numerical) form before DSP techniques can be applied.

An analog electrical voltage signal, for example, can be digitised using an integrated electronic circuit device called an Analog-to-Digital Converter or ADC. This generates a digital output in the form of a binary number, whose value represents the electrical voltage input to the device. The operation of an ADC is to transform a measured sequence of analog signals into a corresponding sequence of binary numbers. These binary sequences are the kinds of signals that can be stored in the digital media used by computers, such as RAM, ROM, CDs and MP3s.

A-D conversion relies on the principle that at any point in time, an analog signal can be assigned an instantaneous value by measuring its voltage. The analog voltage that corresponds to an acoustic signal changes continuously, so that at each instance in time it has a different value. It is not possible for computers to receive the value of the voltage for every instant because of the physical limitations of both the computer and the A-D converter. Instead, an analog signal is measured at specific intervals of duration of time. To convert to digital form, we must represent the continuous data as discrete data. This process is called *Sampling.*

2.3.3 Sampling

When a sound wave strikes a microphone, a voltage is produced that varies in accordance with the pattern of the wave. The analog voltage that corresponds to an acoustic signal changes continuously so that at each instant in time it has a differing value. To be able to convert this to digital information the analog voltage is measured (sampled) at very short intervals of equal duration. The time between samples is called the *sampling interval*, and the speed at which the samples are taken is called the sampling *rate* or sampling

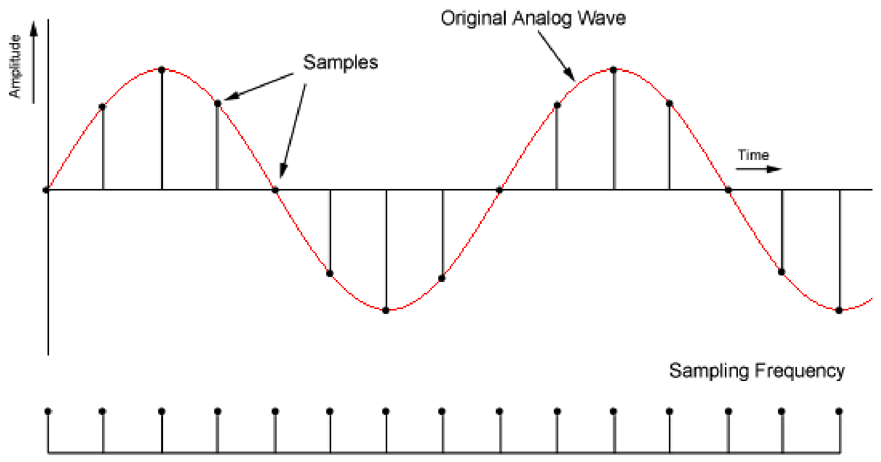


Figure 2.1: Analog and digital representations of a signal. The red wave represents the original analog signal, and the vertical lines represent the sampled values of the waveform.

frequency, measured in Hertz (samples per second). This type of sampling is known as Pulse Code Modulation (PCM); Mu-law encoding and a-law encoding are common non-linear encoding techniques that provide a more compressed version of the audio data - these encoding techniques are typically used for telephony or the recording of speech. A non-linear encoding maps the original sound's amplitude to the stored value using a non-linear function, which can be designed to give more amplitude resolution to quiet sounds than to louder sounds. A visualisation of the sampling process is given in Figure 2.1, above.

As sampling tries to accurately reproduce an analog (continuous) signal, it may feel intuitive to assume that the more samples taken per second (the higher the sample rate), the more accurately the signal could be represented. This implies that anything less than an infinite sampling rate causes some error or loss of quality in the digital signal. The Nyquist-Shannon[3] theorem attempts to solve this problem [20]. This theorem suggests that given an analog signal with a highest frequency component *fmax*, then the sampling rate of *2fmax* - twice the highest analog frequency component - will completely and accurately represent the analog signal. Therefore we can reconstruct an analog signal correctly given that:

$$SamplingFrequency \geq 2fmax \tag{2.1}$$

[3]also known as the Nyquist-Shannon-Kotelnikov, Whittaker-Shannon-Kotelnikov, Whittaker-Nyquist-Kotelnikov-Shannon, WKS, as well as the Cardinal Theorem of Interpolation Theory. It is also often referred to simply as the sampling theorem

Human hearing can perceive and recognise sounds up to or around the 20KHz region of the frequency spectrum. Therefore in accordance to the Nyquist theorem the ideal sampling rate should be greater than 40KHz - Compact Discs use a sampling frequency of 44.1KHz which gives some headroom for listeners who can hear slightly over the 20KHz 'limit'.

Failing to adhere to the Nyquist theorem results in incorrectly represented signals - when the digital signal is converted back to analog form (for example when digital sound is played back through speakers) using a DAC (Digital to Analog Converter), false frequency components appear that were not in the original analog signal. This effect is called *aliasing*. For images, this produces a jagged edge (and hence *anti-aliasing* algorithms in image processing applications), or stair-step effect. For sound, it produces a superfluous buzzing noise, or fuzz.

The number of bits used to represent each sample determines both the noise level and the amplitude range that can be handled by the system. Compact discs (or audio of equal quality) for example, uses 16-Bit numbers to represent a single sample with a sampling rate of 44.1KHz (the analog/continuous signal is sampled 44100 times every second, with 16-Bit integer accuracy).

Therefore if the sample rate is known, it is possible to reconstruct a sampled analog audio signal from only a list of its amplitude readings at each sampling interval. As such, audio data is usually expressed in the time domain.

2.3.4 Time domain representation

Sound waves produce changes of pressure in the air that vary with time. A standard method of depicting sound waveforms is to represent them in the form of a graph of air pressure (amplitude) versus time. This is called a time-domain representation and is represented in Figure 2.1.

2.3.5 Frequency Domain Representation

Frequency Domain is a term used to describe the analysis of signals with respect to frequency. A frequency-domain graph shows how much of the signal lies within each given frequency band, over a range of frequencies. An example of this is shown in Figure 2.2, whereby the frequency is represented on the X-axis (from 20Hz to 20KHz, logarithmically), and the amplitude is shown on the Y-axis (in decibels). The graph shows the average frequency content of a piece of audio at a particular point in time.

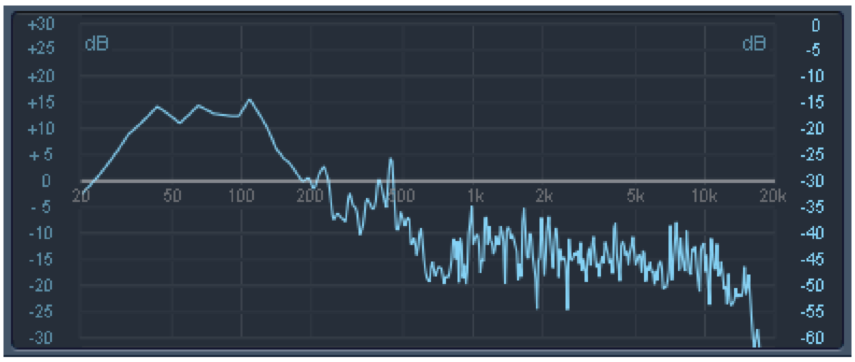


Figure 2.2: Frequency analysis using a Logic Pro software EQ FFT analyser. Frequency is represented on the X-axis (logarithmically), and the amplitude is shown on the Y-axis

A signal can be converted between the time and frequency domains with a pair of mathematical operators called a *Transform*. An example is the Fourier Transform, which decomposes a function into the sum of a (potentially infinite) number of sine wave frequency components. The *Spectrum* of frequency components is the frequency domain representation of the signal. The *inverse* Fourier Transform converts the frequency domain function back to a time function. In the next section, the Fourier Transform and general Fourier Theory are explored.

2.3.6 Fourier Transform

Joseph Fourier first discovered the idea in the 19th century, when he showed that representing a function by a trigonometric series greatly simplified the study of heat propagation [21]. Over time this idea was applied to various other applications, especially in the field of DSP, where it has become an invaluable tool. The basic premise of Fourier Theory is that: *it is possible to form any function f(x) as a summation of a series of sine and cosine terms of increasing frequency.* The attempt to understand functions by breaking them up into basic pieces that are easier to understand is one of the central themes in Fourier analysis.

The Fourier Transform is a function that transforms one complex-valued function of a real variable into another. As discussed, for audio signals the domain of the original function is typically time, and the Fourier Transform is often called the frequency domain representation of the original function, as it describes which frequencies are present in the original function[4].

An example of the Fourier Transform can be demonstrated when we consider the decomposition of a note played on an instrument, into its overtones. For example, a note played on a flute is relatively pure: a single frequency with only minor overtones, which are additional, related frequencies. It may be envisaged as a simple repeating sine wave because of the relative absence of such overtones.

However the same note played on an oboe would have a somewhat higher-pitched sound, caused by a higher level of overtones. It would appear visually more complex (in a frequency-amplitude graph representation), because of the significant presence of such overtones. In this example, a Fourier transform is the representation of the frequencies (and their amplitudes) present in that note, on that instrument.

For the flute note it would be approximately a single vertical line at the single frequency; for the oboe note there would be additional, lower amplitude vertical lines representing the overtones. This is explained in more detail in the following section, and also when the Sub-Harmonic Summation pitch detection algorithm is encountered, later in the thesis.

2.3.6.1 Fourier Transform Equations

While it is not within the scope of this thesis to present a complete overview of Fourier theory, it is useful to review some of the concepts in order to understand the implementation of algorithms presented in DART. A true overview of Fourier Theory, Discrete Fourier Transforms, Fast Fourier Transforms and all the vast related works are available in many Mathematical, Electrical Engineering, Computer Science, Physics, and related field text books. The motivation for the Fourier transform comes from the Fourier series[5], in which complicated signals (or *functions*) are written as the sum of simple waves, mathematically represented by sines and cosines. The Fourier Transform of a signal is a summation of

[4]The term Fourier Transform often refers both to the frequency domain representation of a function and to the process or formula that *transforms* one function into the other.

[5]`http://en.wikipedia.org/wiki/Fourier_series`

potentially infinite sine and cosine terms of differing frequencies. The Fourier Transform of a continuous function $f(x)$ (where x is real) is defined by the equation:

$$F(u) = \int_{-\infty}^{\infty} f(x)e^{-2\pi ixu}dx \tag{2.2}$$

where $i = \sqrt{-1}$ and u is often called the *frequency variable*. This equation can be read (slightly) more easily when the exponential term is expanded into its sine and cosine components using Euler's formula[6], giving:

$$F(u) = \int_{-\infty}^{\infty} f(x)(\cos 2\pi ux - i\sin 2\pi ux)dx \tag{2.3}$$

$F(u)$ is the data in the frequency space. Even when starting with $f(x)$ in the time domain, $F(u)$ is usually a complex value because a real number multiplied by a complex number produces a complex number. It should be noted that given $F(u)$, we can go backwards to reveal $F(x)$ by using an *inverse* Fourier transform:

$$f(x) = \int_{-\infty}^{\infty} F(u)e^{2\pi ixu}du \tag{2.4}$$

The difference between the forward and inverse Fourier transform is merely e, making it possible to go back and forth between the time (or spatial) and frequency domains.

With the expansion of the exponential term into the sine and and cosine components, it is possible to calculate the integral as two separate integrals - one for the Real part of the equation and a second for the Imaginary part.

$$F(u) = \int_{-\infty}^{\infty} f(x)\cos(ux)dx + \int_{-\infty}^{\infty} f(x)j\sin(ux)dx \tag{2.5}$$

$f(x)e^{-2\pi ixu}$ is complex, thus the sum of these terms must also give a complex number. Therefore we can view this as:

$$F(u) = R(u) + iI(u) \tag{2.6}$$

[6] $e^{i\theta} = cos\theta + isin\theta$ where e = 2.71828..., and θ is an angle which can be any real number. This is true for any real number θ

where $R(u)$ is a the real component and $I(u)$ is an imaginary component. Keeping the (u) in the equation helps to clarify that the terms are functions of u. The Fourier transform is also written in the polar form as:

$$F(u) = |F(u)|\, e^{i\theta(u)} \tag{2.7}$$

2.3.6.2 Discrete Fourier Transform

However, audio is rarely continuous in nature. The Discrete Fourier Transform (DFT) is a solution to tackle this, and is the equivalent of the continuous Fourier Transform for signals with a finite sequence of data (i.e. finite signals). The DFT replaces the integral with a finite sum, (which can be considered a **`for`** loop when programming). Therefore given N discrete samples of $f(x)$, sampled in uniform steps:

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{-2i\pi xu/N} \tag{2.8}$$

for $u = \mathit{0}, \mathit{1}, \mathit{2}, ...N - \mathit{1}$, and

$$f(x) = \sum_{u=0}^{N-1} F(u) e^{i2\pi xu/N} \tag{2.9}$$

for $x = \mathit{0}, \mathit{1}, \mathit{2}, ...N - \mathit{1}$

In general, the Discrete Fourier Transform of a real sequence of numbers will be a sequence of complex numbers of the same length.

2.3.6.3 Fast Fourier Transform

However, the Discrete Fourier Transform is not very efficient. The number of complex multiplications and additions required to implement the equations outlined is proportional to (N^{2}) (see footnote [7]). The Fast Fourier Transform solves this problem and reduces the number of computations needed for N points from N^{2} to $N \log_{\mathit{2}} N$.

[7]Converting the DFT to programming code generates two nested *for* loops; having two next loops of the size of the input means that computation time will grow with N^{2}

The FFT algorithm was published by J.W. Cooley and John Tukey in 1965 in [22], using principles first used by Gauss as early as 1805. In 1969, a 2048 point analysis of a seismic 'trace' took 13.5 hours. Using the FFT, the same task on the same machine merely took 2.4 seconds [23].

The aptly named Cooley-Tukey algorithm is the most common FFT algorithm, whereby they re-define the DFT of a signal sized $N = N1N2$ in terms of smaller DFTs of sizes $N1$ and $N2$; the FFT operates by decomposing an N point time domain signal into N time domain signals each composed of a single point. The N frequency spectra corresponding to these N time domain signals are then calculated, before the N spectra are synthesised into a single frequency spectrum. An 'interlaced decomposition' is used each time a signal is broken in two; that is, the signal is separated into its even and odd numbered samples.

Cooley and Turner noticed that if the size of the input is even, then it is possible to write N as equal to $2M$; it is possible to split the N element summation in the previous formulas into two M element ones. This splits an N transform, be it direct or inverse, into two others of half the size, one over even indexes, the other over odd ones.

If N is not a power of 2, there are two strategies available to complete an N-point FFT, such as taking advantage of factors that the number possesses. For example, if N is divisible by 3 (e.g. 48), the final decimation stage would include a 3-point transform. Another common alternative that is used in DART, is to pack the data with zeroes; e.g. include 16 zeroes with the 48 data points for $(N = 48)$ and compute a 64-point FFT.

There are a number of different ways that this algorithm can be implemented, and the discussion is beyond the scope of this chapter and thesis. For more information [22] [24] are excellent sources to reference on this subject. One of the most popular implementation in use today is the FFTW[8]; a well-optimised subroutine library programmed in C for computing the DFT in one or more dimensions, of arbitrary input size, and of both real and complex data.

2.3.7 Why the Frequency Domain?

In music analysis, the frequencies 'contained' in waveforms is obviously of paramount importance. If a signal is represented in the frequency domain, then the number of differing

[8]'Fastest Fourier Transform in the West': `http://www.fftw.org`

signals that are part of the composite signal, including their separate peak values, are revealed. Audio is a 'one dimensional' signal (an image is an example of a 2D signal) and using the frequency domain, it is possible to analyse the sound terms of the pitches or frequencies that make the sound up, recording the amplitude of each frequency. Frequency domain representation forms an important part of MIR analysis techniques.

2.4 Pitch Detection Algorithms

Pitch detection algorithms (PDAs) estimate the fundamental frequency of a given sound. PDAs are commonly used in a variety of contexts, such as Music Information Retrieval (MIR - discussed shortly in Section section:MIR), phonetics, speech coding (data compression of digital audio signals containing speech) and music production. The problem of estimating the fundamental frequency of a tone that contains 'noise' also appears in communications, and medical applications. Consequently, different results are expected from the algorithms so there may be different demands placed upon the algorithm. There has been extensive study in the field of pitch determination in speech signals, however speech and musical pitch detection algorithms pose rather different challenges and finding the fundamental frequency in the much wider context of music (monophonic and polyphony, and with a huge variety of instruments and voices) is generally more challenging. Music can span across more than 8 octaves, and the sounds produced by different musical instruments vary a lot in their spectral content and timbre. However music signals are often simpler in terms of their tempo and dynamics when compared to speech.

As there is currently no single, 'perfect' PDA that performs perfectly in all conditions and for all purposes, a range of PDAs exist. Most PDAs function well when given a clean, pitched signal, however most can fail when given a noisy or polyphonic signal. When a signal has a fundamental frequency that does not produce the highest amplitude (in the frequency domain) or the signal is not clearly periodic, it is possible to mistake an upper harmonic for the fundamental; humans can also make this error (as highlighted in Section 2.2.2). However as humans, our perception of pitch covers a wide range of frequencies and we can sense musical pitch even in the presence of noisy or complex signals. We can also follow several pitches simultaneously and detect expressive pitch deviations (such as vibrato[9]).

The field of Pitch Detection is extremely vast and cannot be covered in complete detail in this thesis, however some background knowledge is useful before considering the rest of the thesis. Both time and frequency domain based PDAs exist, and overviews of some common methods given in the following sub-sections. An excellent (and recent) overview of the field of PDAs is given in [25], a PhD thesis entitled *Fast, Accurate Pitch Detection Tools for Music Analysis*.

2.4.1 Time Domain Pitch Detection

The simplest time domain-based PDAs measure the distance between the *zero crossing points* of the signal, an idea that was presented in [26]. The zero crossing *rate* (ZCR) is a measure of how often the (time-amplitude) waveform crosses 0 per second and was initially believed that the ZCR was related to the fundamental frequency. However, this was shown to be unreliable in [27]; if a waveform is complex and contains a number of harmonics and partials (considered to be 'noise' when looking for the fundamental frequency), the accuracy of the ZCR method begins to drop. Filtering out the higher frequencies to remove the higher partials can help to improve accuracy, however the 'cutoff' frequency needs to be chosen carefully so as not to remove the fundamental frequency, while removing as much high-frequency information as possible. This method is relatively simple to implement and does not require a great amount of processing power/time. If the nature of the signal is known (for example 'voice' or 'guitar', or perhaps 'male tenor voice'), then a method can be implemented that is tailored to the waveform, reducing the level of errors.

Despite the simplicity of this technique, the information gained when measuring the ZCR can be useful in applications such as speech recognition, where a single source is assumed (the input data is a clear, monophonic voice) - the benefits of this simple method is outlined in [28] and [29]. Other rate-based pitch detection methods include the *peak rate* method, which counts the number of positive peaks per second in the waveform in order to estimate the fundamental frequency, and the *slope event rate* method. The slope event rate method looks at the periodicity of the *slope* angle of a waveform, which will also be periodic if the waveform has a fundamental frequency. These detection methods work

[9]Vibrato is a musical effect consisting of a regular, pulsating change of pitch, used to add expression.

well on simple waveforms, but again become less robust with more spectrally complex waveforms that cross zero many times or have many peaks in a cycle. Peak counting in particular has been used in hardware-based frequency-detectors for many years as the circuit is simple (and therefore cheap to implement), and when combined with a low-pass filter to remove higher partials, provides a fairly accurate solution.

Modern time-domain based pitch detection algorithms tend to build upon these methods, with additional refinements in order to increase the accuracy of the results. The YIN [30] and MPM [31] are modern time-domain PDAs that are both based upon *autocorrelation.* Autocorrelation is used in mathematics in order to recognise repeating patterns, such as the presence of a periodic signal obscured by noise, or for identifying the (missing) fundamental frequency in a signal (as implied by its harmonic frequencies). Autocorrelation represents the degree of similarity between a given time series and a time-delayed version of itself, over successive time intervals. Autocorrelation cross-correlates a signal with itself in order to show the similarity between observations as a function of the time separation between them.

Pitch detection algorithms that use the Fourier Domain can suffer from *spectral leakage* when the (finite) window chosen in the data does not contain a whole number of periods of the signal. The common solution to this is to reduce the leakage by using a windowing function [32], smoothing the data at the window edges in the time domain, before performing the FFT. This requires a larger window size for the same frequency resolution. A similar problem happens in some time domain methods[10], such as the autocorrelation, where a window containing a fractional number of periods, produces maxima at varying locations depending on the phase of the input.

2.4.2 Frequency Domain Pitch Detection

As explained in Section 2.3.6, using a Fast Fourier Transform converts time-amplitude based audio into the frequency domain. In the frequency domain, musical notes are composed of a series of harmonically related partials that can be identified and used to extract the fundamental frequency. There are many ‘competing’ frequency domain algorithms including the following, discussed in this section:

[10]PDAs such as MPM introduce a method of normalisation which is less affected by edge problems.

- Cepstrum Analysis
- Maximum Likelihood Estimators
- Component Frequency Ratios
- Filter Based Methods
- Peak Detection using the Harmonic Series
- Subharmonic-to-Harmonic Ratio
- Pair-Wise Evaluation of Spectral Peaks

Cepstrum Analysis[11] [33] is a form of spectral analysis whereby the output is *the Fourier Transform of the* ***log*** *of the magnitude spectrum of the input waveform* [34]. Naturally occurring partials in an FFT spectrum are often slightly inharmonic; Cepstrum Analysis is based on the understanding that the FFT of a note usually has several regularly spaced peaks, representing the harmonic spectrum of the signal. When the log magnitude of a spectrum is taken, these peaks are reduced, bringing their amplitude into a usable scale. This results in a periodic waveform in the frequency domain, the period of which is related to the fundamental frequency of the original sound.

The Cepstrum Analysis method assumes that the signal has regularly-spaced frequency partials; if it does not (such as with the inharmonic spectrum of a tubular bell), the method will provide erroneous results. As with most other PDAs, this method is well suited to specific types of signals - the Cepstrum Analysis method was originally developed for use with speech signals, which often have have evenly spaced partials.

Pitch detection using ***Maximum Likelihood Estimators*** is outlined in [35] and [36] and attempts to match the frequency domain characteristics to pre-defined frequency maps in order to recognise and deal with the slight inharmonicity of naturally occurring frequency partials in a pitched signal. This method works well for detecting the pitch of 'fixed' tuning instruments, such as a Piano.

The use of ***Component Frequency Ratios*** was outlined in [37] and works by examining the partials in a note using an FFT, followed by peak detection. For each pair of

[11]The name *cepstrum* was created by reversing the first four letters in the word "spectrum", indicating a 'modified spectrum'. The independent variable related to the cepstrum transform has been called "quefrency" (further continuing the letter-scrambling approach), and is often simply referred to as 'time'.

partials, the algorithm finds the 'smallest harmonic numbers' that correspond to a harmonic series with these two partials in it. Each of these harmonic number pairs are then used as a candidate for the potential fundamental frequency of the signal. After all pairs of partials are considered, the candidate most strongly suggested by the pairs of partials is chosen as the fundamental frequency. Some pairs of partials can be weighted higher, suggesting that the likelihood of their candidate being the fundamental frequency is higher than the probability given to other partials. Generally, higher amplitude pairs are counted more than lower amplitude pairs.

The Optimum Comb Filter [38] method is an accurate but computationally costly filter-based PDA. A comb filter[12] has many equally spaced pass-bands (the 'comb' effect). In the case of the Optimum Comb Filter PDA, the location of the passbands are based on the location of the first passband. For example, if the centre frequency of the first passband is 50 Hz, then there will be narrow pass-bands every 50 Hz after that, up to the Shannon frequency. The input waveform is comb filtered based on many different frequencies; if a set of regularly spaced harmonics are present in the signal, then the output of the comb filter will be greatest when the passbands of the comb line up with the harmonics. However, if the signal consists purely of the fundamental frequency, then the method will fail as there will be many comb filters that will have the same output amplitude, wherever a passband of the comb filter lines up with the fundamental frequency. The ***Tunable IIR Filter***, a more recent filter-based PDA suggested in [39] consists of a narrow, tunable band-pass filter[13], which is swept across the frequency spectrum. When the filter is in line with a strong frequency partial, a maximum output will be present in the output of the filter, and the fundamental frequency can then be read off the centre frequency of the filter. This is an extremely simple and 'manual' method of pitch detection , however the paper also presents suggestions for automating this search procedure.

[40] presents an "accurate and efficient" PDA that works for monophonic sounds, representing a PDA that uses the Harmonic Series. The method relies upon accurate partial estimates, obtained on a frame basis by means of 'enhanced' Fourier analysis. The use of state-of-the-art sinusoidal estimators allows the algorithm to work with frames of only

[12]A comb filter adds a delayed version of a signal to itself. The frequency response of a comb filter consists of a series of regularly spaced spikes, giving the appearance of a 'comb'.

[13]A band-pass filter passes frequencies within a certain range and attenuates frequencies outside that range.

two fundamental periods. The accuracy of the proposed method does not degrade for high pitched sounds, making it suitable for musical sounds.

Sub-harmonic summation (SHS) is a method of pitch measurement and detection (originally presented by DJ Hermes in [41]) that examines each n peaks in the *frequency-amplitude plot* of a piece of audio, and attempts to determine if each peak could be the fundamental frequency by - as the name suggests - summing the sub-harmonics. Sub-Harmonic Summation looks at all the integer factors of a frequency, sums them, and then searches for the lowest frequency with the highest value. The algorithm goes through each of the highest amplitude frequencies and considers: "*If this is the second harmonic, what is the fundamental?*" or "*If this is the n-th harmonic what is the fundamental?*" where n is a suitably large value.

Sub-Harmonic Summation is implemented in this thesis and the design and implementation in DART is explained in detail Chapters 4 and 5 respectively.

The ***Subharmonic-to-Harmonic Ratio*** PDA [42] is tailored to speech and voice pitch detection and employs a spectrum shifting technique to obtain the amplitude summation of both the harmonics and sub-harmonics of a signal. The algorithm compares the amplitude ratios of the subharmonics and harmonics with the pitch perception results in order to determine the pitch of normal speech. The results presented showed the algorithm to be one of the most reliable PDAs in this context; [43] presents a further improvement of the Subharmonic-to-Harmonic ratio PDA described in [42].

Working in frequency domain has also allowed for the recent implementation of PDAs that work well on *polyphonic* audio. In [44] we see a novel approach for pitch identification and estimation for the predominant voice (the 'main melody') in polyphonic music based on the ***Pair-Wise Evaluation of Spectral Peaks***. It was evaluated during MIREX (the Music Information Retrieval Evaluation eXchange) in 2006 and 2009, where the algorithm was found to have the best overall accuracy of the methods tested, as well as excellent performance measures. The method works by identifying partials in the audio that have successive (odd) harmonic numbers with well defined frequency ratios, thus deriving a possible fundamental frequency from the instantaneous frequencies of the two spectral peaks. Further, the identified harmonic pairs are rated according to harmonicity, timbral smoothness, the appearance of spectral peaks, and harmonic number. The resulting pitch strengths are then added to a pitch spectrogram, giving a visual representation of the spectrum of frequencies in the sound.

2.5 Music Information Retrieval

As explained in the introductory chapter, music is increasingly available in digital formats and consumers could feasibly possess more music than they have time to listen to it. Online music distribution is extremely easy and cheap to negotiate, even for independent and 'bedroom' artists, who have the ability and technology to distribute a song to iTunes, Amazon and other leading music distributors within a few minutes of submitting and uploading their music, though services such as TuneCore[14]. Despite album *sales* reducing, more music is being released and distributed than ever before. As explained in [45] (published in 2009):

> *Overall production figures for the creative industries appear to be consistent with this view that file sharing has not discouraged artists and publishers. While album sales have generally fallen since 2000, the number of albums being created has exploded. In 2000, 35,516 albums were released. Seven years later, 79,695 albums (including 25,159 digital albums) were published (Nielsen SoundScan, 2008).*

Furthermore, the traditional methods of listening to, discovering and purchasing music such as radio, MTV and record stores are being replaced by personalised on-demand ways to hear and learn about new music. Users expect to be able to organise and search through music easily and the sheer magnitude of available media makes this task increasingly difficult. The need to organise and provide access to this music has drawn attention from both commercial sectors hoping to use these techniques to more accurately market and sell music, and also from the MIR community, who have new opportunities to discover trends and patterns in music.

However, the increasing availability of digital music is merely an aggravating factor of a more significant issue; few truly effective information retrieval techniques exist for digital music collections. Developing these techniques for music is challenging because of the wide

[14]`http://www.tunecore.com`

variety of ways music is produced, represented, and used [9]. Basic research in MIR can be roughly categorised by the kind of music representation employed - such as symbolic musical notation (MIDI data [46], transcripts, scores and notation [47]), metadata such as ID3 tags (a metadata container for the MP3 audio file format), and the audio itself (content based).

Early MIR research focussed on the symbolic form because pertinent musical features and expressions were easier to extract from scores and MIDI data, and MP3 audio was not available or as popular. Even at present, the most common method of accessing music is through textual metadata; metadata can be extremely detailed and as such there are many scenarios where this approach is sufficient. However, as music collections become larger it becomes increasingly difficult to maintain consistent, expressive metadata descriptions. While the advent of Web 2.0 and social networking sites has helped address the problem in creating the metadata for such a large amount of content, many different users will have generated the descriptions; the variations in concepts when encoding can impact search performance drastically. Furthermore, the descriptions represent opinions (for example in the cases of 'Genre Classification' or 'Mood Classification'), therefore editorial supervision of the metadata is paramount (see *Pandora* below). For these reasons, this thesis focusses on the content-based analysis of music.

Audio analysis and MIR algorithms are rapidly expanding, providing new ways to retrieve information from audio sources. Modest successes have been made in audio-based musical genre classification algorithms [48] [49], beat detection and analysis [50], similarity retrieval [51] [49] [52], and audio fingerprinting [53].

These works generally look at low level audio features in one of three ways; frame based segments (between 10-1000ms), beat synchronous segments (features are aligned to musical beat boundaries), and statistical measures that construct probability distributions from musical features [54]. [54] also displays a schematic showing common audio low-level feature extraction processes such as a log-frequency chromagram[15], Mel-frequency cepstral coefficients, linear-frequency chromagram, and beat tracking - for all of which the first step

[15]A chromagram is defined as the restructuring of a spectral representation in which the frequencies are mapped onto a limited set of 12 chroma values in a many-to-one fashion. This is done by assigning frequencies to the 'bin' that represents the ideal chroma value of the equally tempered scale for that frequency. The 'bins' correspond to the twelve chromas in an octave. A chromagram represents the likelihood of the chroma occurrences in the audio [55]

is a Fast Fourier Transform. An FFT with a small window size is used because the phase of the spectrum is not as important for music as the magnitude.

These 'Short-Time Fourier Transforms' are used to track the means and variances of the Spectral Centroid (a measure used in DSP algorithms to characterise a spectrum, indicating where the 'center of mass' of the spectrum is), standard deviations of the spectrum around its centroid, spectral envelopes, and signal power to represent sound textures, beat and pitch content [56]. These values are then transformed into attribute-value pairs for pattern matching and semantic retrieval. [54] provides a summary of some of the approaches to bridging the gap between low-level techniques such as these, and high-level music tasks such as artist and mood recognition.

[57] surveys the state of the art in automatic *emotion recognition* (mood classification) in music, both with contextual data and content based approaches. Mood recognition can be an extremely difficult area of MIR; emotion recognition is often subjective and when compared to other music information retrieval tasks the identification of mood is still in its infancy, though it has recently received increased attention. The paper concludes that *"In the past 5 years, the performance of automated systems for music emotion recognition using a wide range of annotated and content-based features (and multi- modal feature combinations) have advanced significantly."*.

Content based MIR has even reached a widely used, commercial applications. Technologies exist such as Shazam[16], which uses a highly robust audio fingerprinting system as detailed in [58]. This system can take the *fingerprint* of an unknown audio clip as a query on a fingerprint database containing the fingerprints of a large library of songs, and then identify the song title, authors, and all the relevant metadata. At the core of the presented system is a very efficient and effective fingerprint search strategy, which enables the system to search a large fingerprint database extremely quickly, with only limited computing resources. Shazam's fingerprint extracting is based on extracting a 32-Bit sub-fingerprint every 11.8ms. The sub-fingerprints are generated by looking at energy differences along the frequency and time axes (via a Fast Fourier Transform). A fingerprint block, comprising 256 subsequent sub-fingerprints, is the basic unit that is used to identify the song. Shazam works extremely well and has been adopted by millions of smart-phone users world-wide, showing that MIR research can have a mainstream impact.

[16]`http://www.shazam.com/`

There is still much to be done in the field of MIR. Refinements to existing strategies, as well as new strategies are still needed. [8] gives an exceptional overview of the current directions and future challenges faced in MIR research, detailing the different types of representations of music and the types of research being applied to them, and Nicola Orio's paper [59] also gives an excellent overview of MIR in general, discussing the techniques commonly exploited in MIR processing, as well as evaluations of these techniques. [54] highlights the current directions and future challenges content-based music information retrieval and provides and is an excellent review of the modern state of the art. Finally, an excellent slideshow tutorial on "Music Information Retrieval" is available at `http://www.cp.jku.at/tutorials/ecir2012_mir20.html` - this was a tutorial held at the 34th European Conference on Information Retrieval in 2012 and covers all aspects of modern MIR research. These papers serve as an excellent preface to the state of MIR, and the challenges ahead.

2.5.1 Distributed MIR systems

Another challenge faced by MIR researchers is that the analysis component of MIR often requires extensive computational resources. Distributed environments and P2P networks are already being used for this purpose; in [60] a scalable P2P system is presented that provides flexible search semantics based on attribute-value pairs and supports automatic extraction of musical features and content-based similarity retrieval. The idea of using MIR over P2P was proposed in [61], however this system suffered from problems with scalability. More recently, the JXTA programming framework was used in [62] to aid in the content-based retrieval over a P2P network.

The DART system proposed in this thesis differs from the distributed MIR system proposed in [60] [62] and [61] in that only *metadata* is returned for analysis, instead of actual audio data files; DART has a different goal in that it is not intended to act as a file sharing system but instead as a distributed P2P MIR research platform, with a specific, future application scenario focussing on the recommendation of music, based on the audio files already stored on a user's hard drive. In reality, DART will be designed with flexibility in mind, and as such could implement 'any' MIR algorithm that would be well suited to a distributed computing platform.

2.5.1.1 NEMA: Development in the State of the Art

MIR is a fast growing area and new developments have taken place since this thesis was undertaken. Most relevant to the research undertaken in this thesis is *NEMA* - The Networked Environment for Music Analysis [63] - which aims to provide the MIR community with a solution to the “extreme issues of data sharing and comparative evaluation”. The NEMA project builds upon and extends MIR research conducted by IMIRSEL and hopes to provide the MIR community with a workflow environment to facilitate “computation over remote audio and resource collections; optimal code reuse, interoperability, sharing and dissemination; standardised, high-quality evaluation procedures; and the encoding of metadata, data and results in a format suitable for use in a detailed linked data system for MIR”. [63] gives an excellent overview of the system.

An in-depth look at NEMAs architecture is presented in [64], outlining the three main subsystems of the NEMA architecture; the user interface, workflow processing, and remote executors. NEMA essentially provides a Web-Interface to allow users (MIR researchers) to execute their MIR code against IMIRSEL-curated music collections. NEMA accepts user-submitted codes written in a wide variety of languages, such as Java, MATLAB, C, C++, and so on. This allows researchers to take more control over their experiments, versus the human-resource intensive MIREX, as discussed in the Chapter 1: Introduction.

NEMA facilitates the integration of music data and MIR/analytic tools that can be used by the global MIR research community, *independent of time or location.* Results and research can be done in stages and the analysis, experiments and results can be shared between institutions easily. NEMA uses Meandre[17] to implement workflow processes; Meandre is a semantic-web driven, data-intensive flow execution environment that provides a high-level language to describe workflows and both a multicore and distributed execution environment based on a service-oriented paradigm. The Meandre work flow work-bench ‘back end’ is hidden in NEMA.

NEMA also integrates with the myExperiment Virtual Research Environment[18], which aims to facilitate the easy sharing and reuse of experiment setups and partially or fully-specified implementations of those experiments, and enables the sharing and re-use of any

[17]`http://seasr.org/meandre/`
[18]`http://www.myexperiment.org`

results. Integration with myExperiment would be an excellent next-step development for DART.

NEMA is a new project, however it is built on work conducted by top researchers in the field of MIR and backed by a number of large institutions; it shows great potential. It differs from DART in nearly all aspects of its implementation and user-experience; NEMA utilises a Web Interface application to create its workflows and not a graphical Problem Solving Environment and heavily makes use of Web Services. As stated in [64], one of the ultimate goals of the NEMA user is to execute their MIR code against IMIRSEL-curated music. DART opens up the possibility of analysing data already on worker machines or providing data to analyse, if the researchers own the audio being distributed (as was the case in the SHS experiments presented in this thesis). DART as a platform for MIR research is less specific in its goal; NEMA is specifically aiming to overcome the limitations of time-specific and location specific resources when running experiments on the IMIRSEL-curated data, and sharing the results. DART is a more general MIR research platform that allows users to come up with an experiment, prototype it quickly and easily in a graphical programming environment, convert it to run standalone, and then distribute it using one of the support distributed platforms.

2.5.2 Music Recommendation Systems

[8] contains an interesting 'Critical Analysis of Coverage Gaps in MIR Research', suggesting that current MIR studies are weak on evaluation and application to real user communities. The paper states:

> *"The problem is twofold:*
>
> 1. *There are no commonly accepted means of comparing the efficacy of retrieval techniques; and,*
> 2. *There have been few if any attempts to study potential users of MIR systems to find out what they need.*
>
> *In addition to these evaluation-related gaps, there are also areas of basic research that are receiving more and less attention than they should. In particular, the amount of emphasis on QBH (Query By Humming) systems appears to be unsupportable given doubts about their usefulness (McPherson et al. 2001) and scalability (Sorsa et al.*

> *2001). **Research on recommendation systems, common in the DL and commercial communities, is inexplicably rare in the MIR community”.***

Since the publication of [8] there have been many efforts to address the first issue, such as the MIR system benchmarking system proposed in [65], which uses the Cranfield model of information retrieval evaluation [66]. However, when pertaining to MIR systems, the results are often too subjective or ‘open to interpretation’, with somewhat vague properties such as *“The precision of the system, that is, the proportion of retrieved material that is actually relevant”*. However papers such as [67] and [68], as well as annual MIREX meetings, are working with community-defined evaluation metrics to solve this issue. The MIREX evaluation results are published on a yearly basis and are presented at the International Conference on Music Information Retrieval (ISMIR[19]).

This thesis aims to produce a proof of concept MIR platform which aims to fill the final ‘coverage gap’ mentioned above in bold - novel research on music recommendations systems. At time of writing, a few notable music recommendation systems do already exist, however, and some even content-based. *Music Surfer* [69][70] is a content based music recommendation system that works by extracting descriptions related to instrumentation, rhythm, and harmony from music. [69] was published in 2005 in the *Proceedings of the 13th annual ACM international conference on Multimedia*, however the research may have ceased as no further publications seem to have emerged from the Music Surfer team, and website URLs appear broken.

[71] from 2004 investigates music recommendation based solely on the content of specific (example) input data. Four methods are proposed; one which builds a ‘model’ for each song set and recommends new songs according to their *distance* from this model. *Distance* can be defined as a similarity measure and can refer to any similarity/dissimilarity measurement between tracks. The other three methods recommend songs “according to the average, median and minimum distance to songs in the song set”. The research managed only to find a technique (called the ‘minimum distance technique’) that returned a valid recommendation as one of the ‘top 5’ 32.5% of the time. The approach based on the median distance was the next best, returning a valid recommendation as one of the top 5 29.5% of the time. Clearly, more work in the field of content based music recommendation was required.

[19]`http://www.ismir.net/`

More recently, there have been some promising advances in the field of content-based music recommendation. In [72] (*Content-based Music Recommendation Based on User Preference Examples, Bogdanov et al., 2010*), three content-based approaches to music recommendation (based on a set of songs provided by a 'user' as examples of musical preferences) are considered. Two of these approaches use a semantic 'music similarity' measure to generate recommendations, while the third approach uses a probability model of the user's preferences. Relatively high-level descriptors such as mood, tempo, genre and rhythm are taken and the three approaches are evaluated against two recommenders using state-of-the-art timbral features, and two contextual baselines, one exploiting simple genre categories, the other using similarity information obtained from collaborative filtering.

The recommendations are tested by a listening experiment to assess how 'accurate' (subjectively) the results of the recommendations are, and the paper shows that the content-based approaches outperform the low-level timbral baselines together with the genre-based recommender. The paper concludes that *"Though the proposed approaches could not reach a performance comparable to the involved collaborative filtering system* [**Last.fm** was used as a metadata based Collaborative Filtering system] *,they yielded acceptable results in terms of successful novel recommendations. We conclude that the proposed semantic approaches are suitable for music discovery especially in the long tail"*. As the content based recommendation of music is a relatively new emerging field, this is encouraging research and an area that a distributed platform such as DART will hopefully contribute towards.

Pandora[20] and Last.fm[21] are two of the most successful (widely-adopted, and commercial) recommendation systems, and are both meta-data based.

Pandora and is a music recommendation system from the makers of the Music Genome Project[22]. Pandora allows users to enter the names of artists or songs they like, and Pandora will return a play list of artists and songs that the user may like, forming a personalised 'radio station' running inside the user's web browser.

Pandora uses a team of approximately 50 expert music reviewers (each with a degree in music and 200 hours of training) to annotate songs using structured vocabularies of between 150-500 'musically objective' tags, depending on the genre of the music. The

[20]http://www.pandora.com/

[21]http://www.last.fm/

[22]http://www.pandora.com/mgp.shtml

system works using detailed (human-entered) track-level metadata enumerating musical-cultural properties for each track in the database. It is estimated that it takes around 20-30 minutes of one expert's time to enter the metadata for one song.

The cost is therefore enormous in the time taken to prepare a database to contain all the information necessary to perform similarity-based search. In this case, it would take one person approximately 50 years to enter the metadata for one million tracks.

Last.fm is also an internet radio and music community website, and uses a music recommendation system known as an *Audioscrobbler*[23]. However Last.fm works differently to Pandora, in that it builds a profile of each user's musical taste by recording details of all the songs the user listens to using their media player (iTunes, Winamp, etc) or iPod. This information is 'scrobbled' (transferred to Last.fm's central database) via a plugin installed into the user's music player, and their profile data is displayed on a personal web page. The Last.fm website offers numerous social networking features and can recommend and play artists similar to the user's favourites. As an added bonus, the Last.fm Audioscrobbler system also exposes its data via web-services so that other projects can make interesting use of the data and statistical results and recommendations in the database.

In contrast with metadata-driven music recommendation systems, *DART* would focus on the analysis of music files on the users computer in order to extract low level information based on the characteristics of the audio content and use these attributes to base the recommendations on. A DART Music Recommendation System could also look at the most frequently accessed music files and attribute more 'weight' to these files. DART would also employ a decentralised, distributed model and aims to provide an advanced, fully scalable platform for developing, testing and deploying new search and analysis algorithms on an Internet scale. Furthermore, as explained later in the thesis, the DART system can be adapted to fulfil a variety of applications other than MIR algorithms, simply by modifying the Triana workflow that is distributed to the workers. The Triana framework and Triana workflows are discussed in the next section.

[23]`http://www.audioscrobbler.net`

2.6 Triana Programming Environment

Triana[24] is a graphical Problem Solving Environment (PSE) for composing data-driven applications. Work-flows (or data-flows) are constructed by dragging programming components, called tools, from the toolbox onto a workspace and then drawing cables between these components to create a block diagram. Components can be aggregated visually to group functionality and compose new algorithms from existing components. For example, to add a digital Schroeder reverb to a piece of audio (Figure 2.3), the file could be loaded (using the LoadSound unit), then passed to a SchroderVerb group unit before being passed to the Play unit to hear the result. The SchroderVerb unit is itself a group, which consists of a number of summed comb delays and all-pass filters, representing the inner workings of the SchroderVerb algorithm.

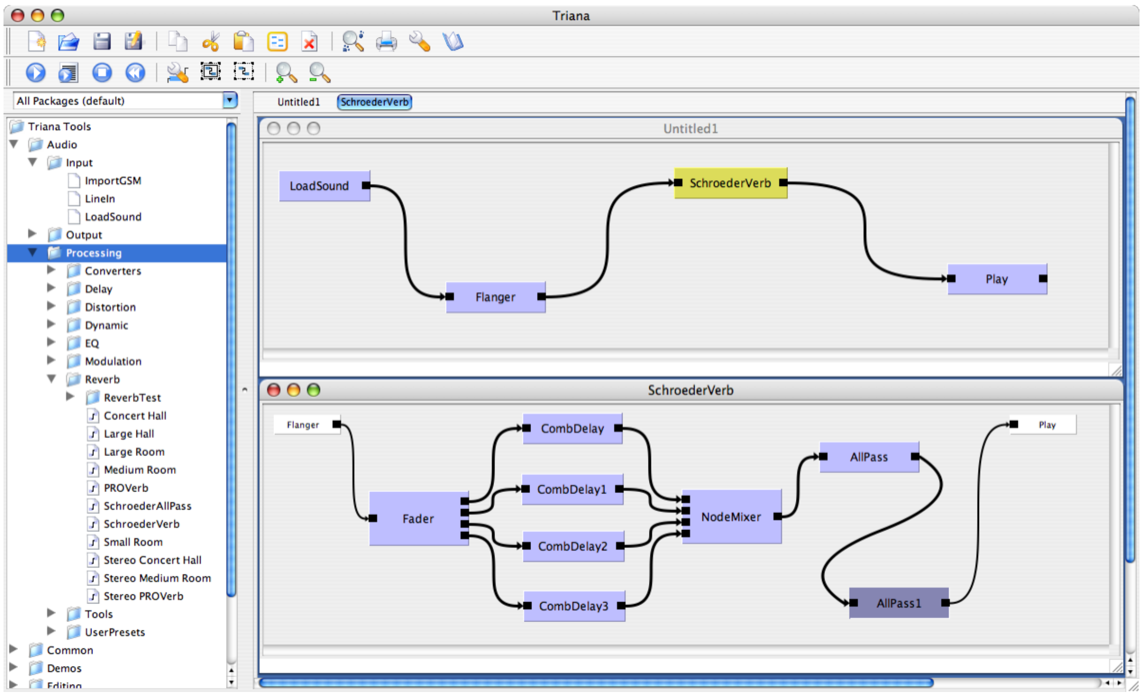


Figure 2.3: A simple audio processing work-flow, showing how a Schroeder reverb is applied to a signal by using a group, which contains the underlying algorithmic details.

Within Triana, a large suite of Java tools exist in a range of domains including signal, image, text and audio-specific processing. The signal processing tools are some of the most advanced, as Triana was initially developed for signal analysis within the GEO600 Gravitational Wave Project (GEO 600 Project, 2004), which used the system to visualise

[24] `http://www.trianacode.org`

and analyse one-dimensional signals that were similar to an audio signal, but with a lower sampling rate. Therefore a number of core mathematical, statistical and high-quality DSP algorithms already exist; there are around 300 signal processing and visualisation units, and in total there are around 500 units in Triana that cover a broad range of applications. The majority of the tools in the Triana Audio Toolkit and the features and classes they depend on - such as audio chunking and audio data types - were implemented by the author, either as part of related work or as part of this thesis.

Triana can be used to investigate and explore data in a simple and flexible way. An important aspect of Triana is that it allows non-programmers to graphically ‘code’ their own algorithms and programs, specifically suited to the tasks they wish to carry out. Programmers and researchers can take advantage of pre-written software modules within Triana to aid in the development of new algorithms. This allows the user to bypass the conventional approach to programming, i.e. creating various classes and methods/functions and coding a core program to connect the set of related method procedures together.

Triana integrates both Grid and P2P technologies and has been used in a number of domains, from bioinformatics, investigating biodiversity patterns, to gravitational wave observation using computational Grids to process signals using standard digital signal-processing techniques. The goal of the DART project is to leverage this technology such that the same kind of DSP processing can be achieved with audio rate signals for the purposes of signal analysis, feature extraction, synthesis, and MIR.

Given its modularity, its support for high quality audio, and its ability to distribute processes across a Grid of computers, Triana has the potential to be an extremely useful piece of software that allows users to implement custom audio processing algorithms from their constituent elements, no matter their computational complexity.

2.6.1 Triana Data Types

Triana units are programmed in a logical and robust manor and only ever need to be written and compiled once. When the unit is written, the programmer specifies the data type that the unit can receive and output. When connecting two units together, the input and output types of both units are checked in order to confirm the compatibility of the units. This means that programmers can create units that can be guaranteed to work sensibly with other units because the run-time type checking guarantees their compatibility. Furthermore, since the data types that are passed between the units contain the parameters associated with the particular data, each unit knows how to deal with the data object, regardless of its content. It is therefore impossible to crash a Triana network due to array size mismatches.

Triana's data type classes are fundamental to its flexibility and power. Data types are containers for the data being processed by the units. Some of the main data types in Triana (that are relevant to the processing of digital audio/MIR) include:

- **`MultipleChannel`**
- **`MultipleAudio`**
- **`VectorType`**
- **`SampleSet`**
- **`Spectrum`** and **`ComplexSpectrum`**

`MultipleChannel` is a base class for representing multiple channelled data. Each channel can have its own particular format, specified within an object that implements the **`ChannelFormat`** interface. Furthermore, each channel can contain complex data so that, for example, multiple channels of complex spectra could be stored.

`MutipleAudio` stores many channels of sampled data. Each channel can have its own particular audio format of the data e.g. the encoding, such as MU_LAW, PCM and number of bits used to record the data. This is essential for performing sound transformations and writing audio data. In essence, MultipleAudio provides the support for high quality audio to be utilised in Triana.

`VectorType` is the basic class used to represent one-dimensional array data of type double, with no extra parameters. It can hold real or complex one-dimensional data sets. If a variable is sampled uniformly, then VectorType holds only the sampled data and a **`Triplet`** indicating how the sampling is done. If the independent variable is sampled irregularly, then VectorType holds the sampling values as well. VectorType defines new methods that allow padding or interpolation of the data with zeros (as often found or required in a Fast Fourier Transform).

`SampleSet` stores a real double array or two real double arrays (representing a complex array) by extending the VectorType class to implement the Signal interface[25]. This requires two further parameters for the sampling frequency and the acquisition time. SampleSet also allows for irregularly sampled data, for which it sets the sampling frequency to zero. **`SampleSet`** is similar to **`MultipleAudio`**, however contains no 'format' data other than sampling frequency.

`Spectrum` stores one or two one-dimensional arrays of double-precision real numbers representing either a real Fourier spectrum (a power or amplitude spectrum), or a Complex Fourier spectrum (**`ComplexSpectrum`** - normally the Fourier transform of a data set). It also includes a **`Triplet`** giving the integer values of the index of this array, but introduces five new parameters: the frequency resolution, the value of the largest frequency, the number of points in the original data set from which the data here were derived, and two boolean flags; one to indicate whether the Spectrum is one-sided or two-sided, and a flag to indicate whether the data has a narrow bandwidth derived from a larger full-bandwidth spectrum.

`ComplexSpectrum` is the basic Triana class for holding one-dimensional Fourier transforms, and **`Spectrum`** is the basic class for power spectra. It implements the **`Spectral`** interface[26].

Converting between types (such as **`MultipleAudio`** and **`SampleSet`**) opens ups vast array of Triana signal programming units to the audio community, such as any of the Signal Processing or Math units. Units can also be programmed to accept a range of input types, and deal with or convert the input accordingly. Importantly, Triana also allows for any Java object to be passed between units.

2.6.2 Audio Toolkit

The Triana audio toolkit consists of several categorised hierarchical folders, each with an assortment of units based on the MultipleAudio Triana data type. This type utilises the

[25]Signal is an interface that can be implemented by any Triana data types that include data that has been acquired from a time-based data stream. It provides methods for setting and returning the sampling frequency and other appropriate information

[26]Spectral is an interface that can be implemented by any Triana data types that include data that represent frequency-domain data or that have been put through a Fourier transform

JavaSound API classes in order to allow the use of high fidelity audio. The Audio toolkit tree is split into three main folders: *Input*, *Output*, and *Processing*.

2.6.2.1 Input Tools

The audio input section houses one of the most important tools for Audio in Triana - the LoadSound unit. This unit allows the user to select an audio file (WAV, Aiff, Au) and outputs the data to the next unit. The user can select if the audio file is to be output in its entirety, or if the data should be streamed on to the next unit by chunking the data into fixed-length segments. Triana also includes a LineIn unit that can steam live audio, and a LoadMP3 unit, which allows MP3 data to be decoded on the fly and analysed, an important step for DART and MIR processing.

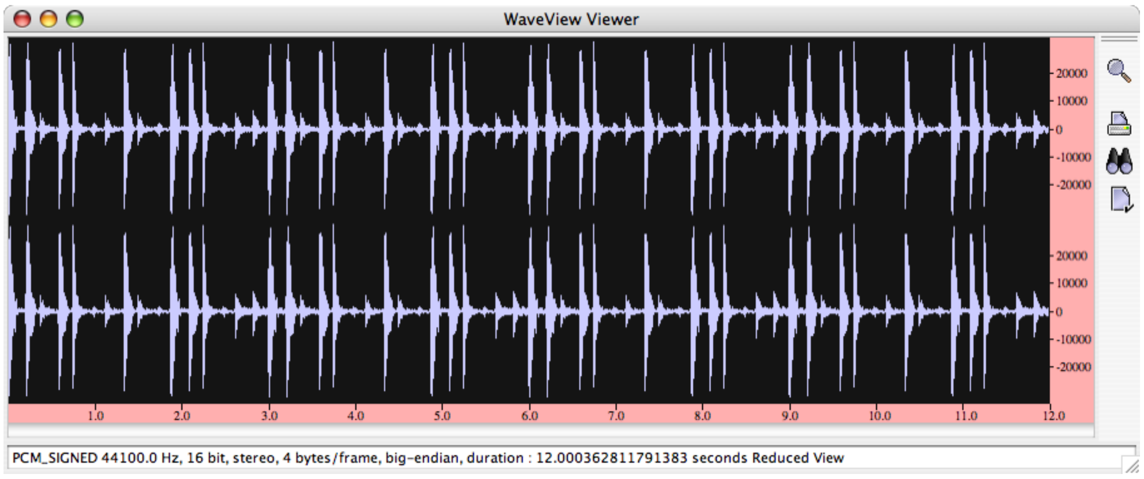
2.6.2.2 Output Tools

The output tools section contains a varied collection of tools to allow the user to hear - and see - the audio data loaded, allowing for thorough analysis of the sounds. The Play unit is one of the most often used in this toolbox and allows the user to play the audio (either streamed or as one large file). Triana gives the user a useful selection of other output tools, such as the WaveViewer unit (see 2.4), which shows the current waveform with a time-amplitude relationship. This allows for a much more thorough visual scrutiny of the audio. Also included in the Output section is a group of audio writing tools (such as WriteWav, WriteAiff), allowing the user to (re)save the audio after any amount of processing in Triana is complete.

2.6.2.3 Audio Processing Tools

The audio processing tools form the core of Triana's audio manipulation capabilities. Inside the audio processing toolkit is another level of categorised folders, covering a wide range of applications:

- **Converters:** for converting between audio formats and for separating stereo streams into two independent channels and vice versa
- **Delay:** single delays, all pass filters and comb delays



PCM_SIGNED 44100.0 Hz, 16 bit, stereo, 4 bytes/frame, big-endian, duration : 12.000362811791383 seconds Reduced View

Figure 2.4: A screenshot of the WaveViewer unit from the Audio/Output folder, displaying a waveform

- **Distortion:** distortion algorithms and fuzz boxes
- **Dynamic:** compressors, limiters and expanders
- **EQ:** low pass, high pass and parametric EQ
- **MIR:** AmplitudeSpectrum, ID3Xtractor, NoteMapper, PitchDetection, PowerSpectrum
- **Modulation:** chorus, flanger, phaser and variable delays
- **Reverb:** Schroeder verbs, and various presets from small rooms to large concert halls
- **Signal Analysis:** FFT analysis for use in spectral processing, analysis/resynthesis, and signal analysis for music information retrieval
- **Tools:** faders, resampling modules, wave inverters, rectifiers and audio reversal algorithms
- **UserPresets:** combinational effects (i.e. group units) encompassing several of the above algorithms with particular settings e.g. distorted flangers and exuberant reverbs

These categories will be recognisable to users of audio processing or production software. Each folder contains units which are in themselves processors but also allow the user to create their custom algorithms from the smaller building blocks supplied. It is possible to break down complex algorithms into a group of functions (units) that when linked together in a particular fashion, are able to perform the task as a whole. This was demonstrated earlier in Figure 2.3, with the example of Triana's ability to group the Comb and Allpass filters in order to create a Schreoder Reverberator algorithm that is then grouped, saved and reused as if it were a single unit. This unit aggregation gives the user more freedom to take advantage of Triana's modularity.

One feature of interest resides in the Converters folder; two units are available that allow the user to convert from MultipleAudio to a SampleSet Triana data type and back again (**`MAudioToSSet`** and **`SSetToMAudio`** respectively). This opens up a whole range of possibilities to the user, enabling them to utilise many of the numerous math and signal-processing units to process the audio, and then convert data back to a MultipleAudio data type for playback. One example of how this technique could be used is shown in Figure 2.5, where a **`Stereo2Mono`** unit (also in the converters folder) is used to split the stereo channels of an audio file or stream into two distinct mono channels. Each side is then converted to a SampleSet and fed into a **`Subtractor`** unit from the Math folder. This subtracts the left from the right stereo channel, which results in the removal of sound that is contained in both i.e. those panned in the middle of the stereo field.

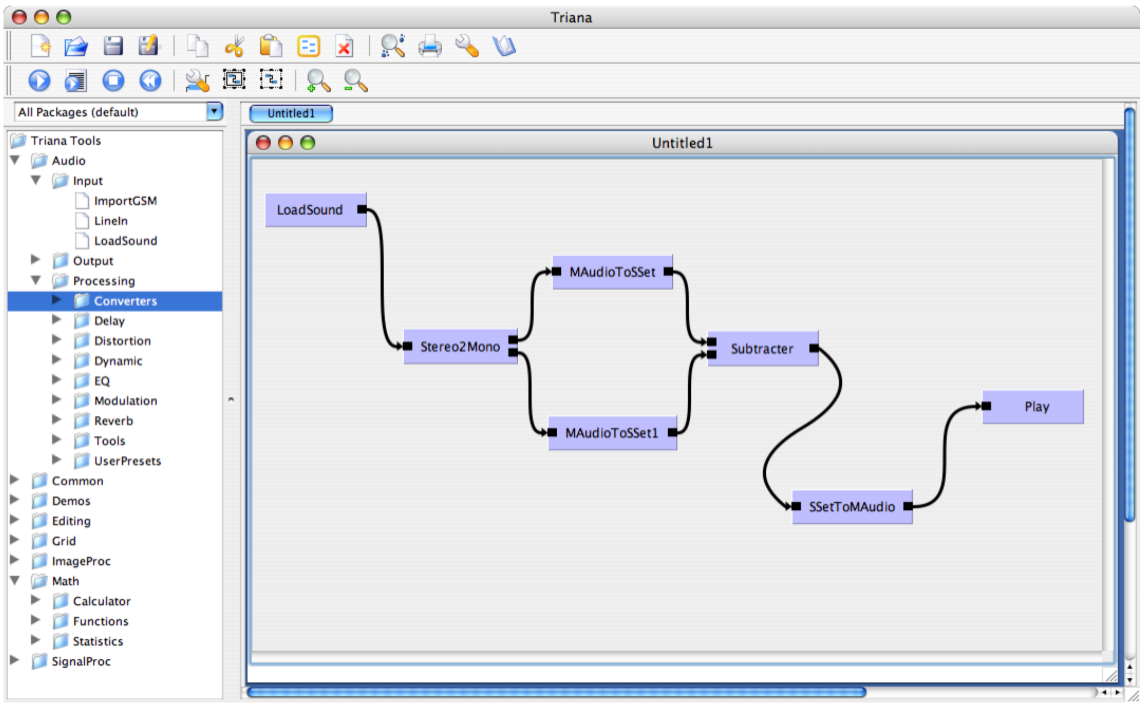


Figure 2.5: Audio is split and converted to a SampleSet data type, in order to subtract all music that is panned down the middle of the stereo field. This conversion is then reversed and played back

This is a simple way of removing vocals from many songs and leaving the (majority) of the backing track (as vocals in particular are normally panned down the centre) and is a simple example of how a few of the converter units could help users create their own algorithms. It must be remembered that the user is encouraged to try create different algorithms themselves, and experiment with different unit connection combinations, and not only using the presets given in Triana.

As mentioned previously, Triana also contains hundreds of statistical, mathematical, and signal processing units, which can be used in conjunction to all of the MultipleAudio compatible units, opening up an vast range of units to facilitate and aid MIR and the creation of useful MIR algorithms. Triana includes a range of filters, graphing and histogram viewers, spectrum analysers and more, meaning that algorithms can be broken down into their constituent elements, and programmers can take advantage of pre-written software modules within Triana to aid in the development of new algorithms.

2.6.3 Creating and Running Triana Programs

Creating an algorithm or program in Triana is simple - units can be dragged from the unit directory, and linked together in Triana's main window, as shown in 2.6.

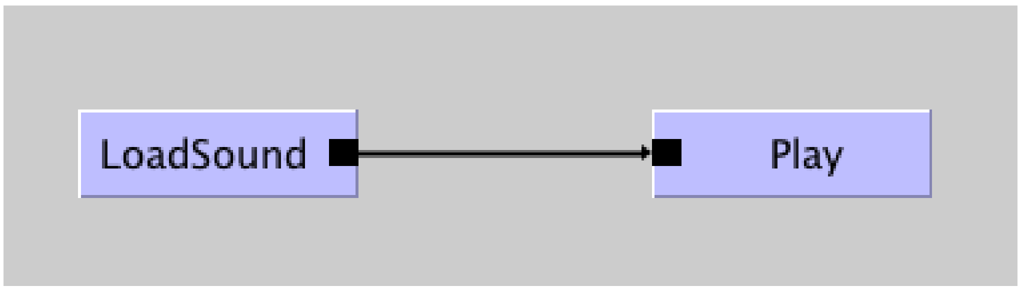


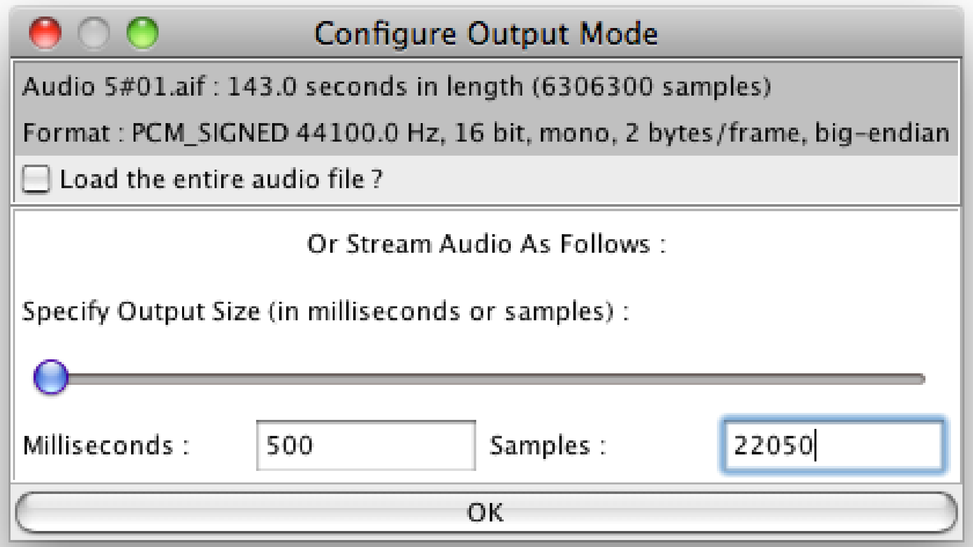
Figure 2.6: A view of the Triana workspace, showing a simple program created in Triana to play back sound files

Figure 2.6 shows a simple 'program' to play back a sound file. A LoadSound unit (one of the many pre existing sub-programs/units) has been dragged onto the workspace (the main Triana Window). Double clicking the unit will bring up the properties of a unit - in this case the LoadSound unit will open up a file browser in order to prompt the user to locate the audio file they wish to play back.

A Play unit has also been added to the workspace. The user can then finally use a virtual 'cable to connect the two units together (by connecting the nodes together), and establish the connectivity and data flow between the two units. This mini-program can then be run by pressing the play button the Triana menu bar, playing back the file.

This example may seem trivial, however demonstrates Triana's simplicity; to create a sound player in Triana (using existing units) takes a matter of seconds, however to manually program a piece of software to do this outside of Triana takes considerably longer.

The LoadSound unit in Triana also supports audio data ‘chunking’, which splits the audio file into smaller chunks of data that can be streamed to the next unit. This is demonstrated in Figure 2.7.


Configure Output Mode

Audio 5#01.aif : 143.0 seconds in length (6306300 samples)

Format : PCM_SIGNED 44100.0 Hz, 16 bit, mono, 2 bytes/frame, big-endian

☐ Load the entire audio file ?

Or Stream Audio As Follows :

Specify Output Size (in milliseconds or samples) :

Milliseconds : 500 Samples : 22050

Figure 2.7: LoadSound Unit: Chunked Data output buffer screen

Chunking splits the data into smaller chunks, which are then treated as separate/individual pieces of audio, allowing for the streaming of audio for real-time processing or for processing in smaller chunks to keep memory usage low.

2.6.4 Related Music Data-flow Tools

Other data flow systems were considered as the building blocks to create the DART workflows, namely MARSYAS, CLAM, M2K and OMRAS, as highlighted in [54].

MARSYAS[27] - Music Analysis, Retrieval and Synthesis for Audio Signals [73] - is a collection of Open Source C++ tools for extracting features and performing machine learning and MIR tasks on collections of music. It is a low-level audio framework, primarily tar-

[27]`http://marsyas.info/`

geted at MIR researchers and developers. MARSYAS 0.1 received a substantial update in 2002/2003 with MARSYAS 0.2, citing influences from CLAM (discussed shortly), and features a GUI. More recently (2005), MARSYAS 0.2 was used in a parallelisation experiment for the distributed audio feature extraction of music, as explained in [74]. The framework created facilitated the partitioning of audio computations over multiple computers (5 in this example), with the results demonstrating the effectiveness of the approach, and reducing overall computation time. 'A Case Study in Implementing MIR Systems' presented in [75] gives a detailed description of the MARSYAS software architecture, highlighting design challenges and solutions that are relevant to a wide range of MIR software.

CLAM (The C++ Library for Audio and Music) [76][77] is similar to MARSYAS 0.1 in that it provides a software framework for performing feature extraction and music synthesis, but extends the concepts by providing a graphical user interface (shown in Figure 2.8), allowing developers to produce feature rich applications targeted at less 'programming-literate' users. Despite being created in C++, CLAM is regularly compiled with gcc in Linux, Microsoft and Intel compilers in Windows and Code Warrior in Windows and MacOS. CLAM contains hundred of C++ classes aimed to *"bring the world of software design and engineering to DSP developers who could care less about it"* [76]. Since it's creation CLAM has often been used as an internal development framework for developers, and utilises a large array of open-source DSP/audio processing libraries

M2K (Music to Knowledge)[28] builds upon D2K [29], a visual programming environment that allows for rapid prototyping and algorithm development. D2K allows the user to create algorithms by wiring together computational modules into programs called itineraries, which represent data flow between modules. These itineraries can then be run, or nested within other itineraries and used as modules, allowing for the development of itineraries with arbitrary complexity. M2K is primarily targeted at evaluating music information retrieval systems [78]. It extends the ideas of MARSYAS and CLAM by providing a graphical user interface for constructing MIR systems. M2K also supports automatic evaluation of MIR tasks and is a key component of the MIREX evaluation experiments. All components are written in Java for maximum flexibility and portability and supports multiple processors (within one computer) to increase parallelisation.

[28] `http://www.music-ir.org/evaluation/m2k/`

[29] `http://alg.ncsa.uiuc.edu/do/tools/d2k`

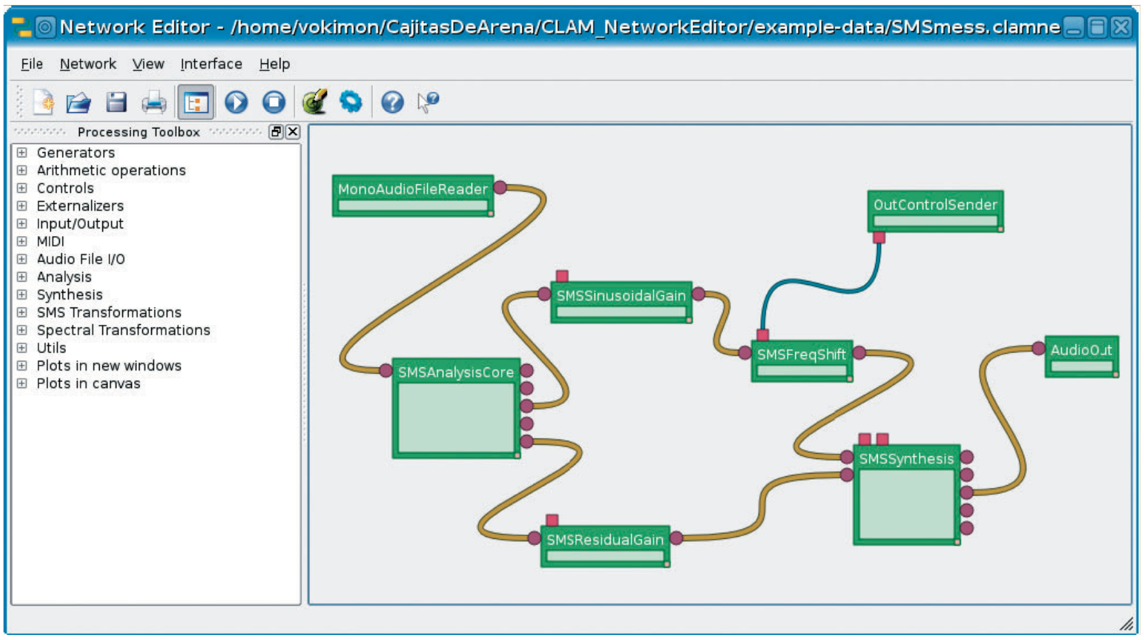


Figure 2.8: The CLAM Network Editor. Developers select processing objects, from the left menu categories and connect the objects on the 'canvas' to build a processing graph or network. Brown connections represent data flows; blue connections represent event-driven control flows.

Using the MIREX DIY MIR submission service researchers can remotely submit, execute, and evaluate their MIR algorithms against standardised datasets that are not otherwise freely distributable. This service is largely built upon the Data-to-Knowledge Web Service (D2KWS) [79] and M2K [80] libraries with the goal to reduce the heavy interaction by IMIRSEL team members in the execution, debugging, and validation of submitted code. MIREX DIY also aims web service is to put these responsibilities into the hands of submitters, and also enable the evaluations of algorithms year round, as opposed to annual exchanges. An overview of this system is presented in [81].

OMRAS2[30] [31] is a distributed MIR framework for annotating and searching collections of both recorded music and digital representations (such as MIDI). OMRAS2 looks at both meta-data and low-level audio content, via a set of extensible ontologies allowing data to be shared over the web and linked to existing resources, and also the VAMP plugin interface[32].

[30]Either 'Online Music Recognition and Searching II' or 'Ontology-driven Music Retrieval & Annotation Sharing service'

[31]`http://www.omras2.org/`

[32]`http://vamp-plugins.org/` - A development kit is available making it easy to wrap any C or C++ code as a Vamp plugin.

The OMRAS framework includes a key GUI component called SonicVisualiser[33], a low-level audio feature extraction and visualisation tool for viewing and analysing the contents of music audio files, and SonicAnnotator[34], a batch tool for feature extraction and annotation of audio files. The audio to be processed can be on the local filesystem or available over *http* or *ftp*. Other components include MusicOntology, a metadata scheme constructed using the Resource Description Framework for describing music and software resources on the Semantic Web, and also AudioDB, a low-level audio feature database that scales to storing and searching features for large music collections.

As can be deduced from the descriptions above, *Triana* has much in common with these graphical data-flow applications. The decision to use Triana was based on platform independence, deep author/developer knowledge of a working system, and also Triana's integrated distributed/grid networking capabilities. Triana has long possessed the ability to distribute the processing of units or algorithms on networks computers as highlighted in [82], and DART aims to be much more scalable than the experiments introduced in [74]. DART also does not aim to act as a file or audio-transfer mechanism, instead only choosing to exchange metadata after processing has taken place.

Triana is open source and written in Java. As such, it boasts the common advantages of the platform, such as:

- Platform Independence
- Automatic garbage collection; memory management is automatic, reducing development time
- Multi-platform and web-services support; ease of development of dynamic web applications

While C++ excels in real time performance versus Java due to the Java Virtual Machine adding an extra level of abstraction, Java (and Triana) provide the platform independence and distributed technologies which mean that huge amounts of processing can be achieved, the advantage of which vastly outweighs Java's slower DSP performance. Indeed, MARSYAS is currently being ported to Java in order to take advantage of these features of the platform.

Triana boasts many of the same advantages as the systems described above, and can also

[33]http://www.sonicvisualiser.org/
[34]http://www.omras2.org/SonicAnnotator

be used as a prototyping platform for developers, making it easy to detach code and classes and create standalone software after testing algorithms in Triana first. Furthermore, the Triana software is not solely focussed on MIR and as such, the DART platform[35] could be leveraged to tackle a wide range of problems. One application scenario is a DART system installed on a local/closed network; scanning sound effect audio files on separate systems in networked commercial studio facilities, to search for sound effect files (or for example, drum sounds) that match a specific criteria. For example, if a DART user requires a snare drum sample, usual search mechanisms cannot search for suitable sounds unless the audio files filename contains the word snare or other suitable identifier - the search mechanism would do no more than string matching.

However, given the appropriate workflow created by the DART manager, when propagated onto the network the DART system would allow the user to search all the audio content in the distributed sound library, and suggest files with suitable characteristics that match the specific search criteria set by the user. This would also be able to return further results, which would not be returned via conventional methods - any samples that match the users search criteria, can be returned. This means that when ignoring criteria such as pitch and tempo and investigating the timbre of the sound file, new sounds that could be adapted to work for the user, could be suggested. For example, a sped up sample of a car collision may easily work well in place of a drum ‘snare’ sample.

Another example consists of a scenario that requires speech recognition algorithms to detect a certain caller from a large amount of recorded telephone audio data that was distributed over several machines. Given a sophisticated enough algorithm (created in Triana as a workflow), the DART framework could be used to scan terabytes of recorded audio data to help trace calls from a specific caller. The flexibility of the DART system stems from the ability to easily refine and change the Triana workflow that dictates what the worker nodes will be processing in their screensaver/idle/nice time.

2.6.5 JavaSound API

The Triana audio framework utilises JavaSound [83], a low-level API for manipulating the input and output of sound media (both Musical Instrument Digital Interface (MIDI) data

[35]meaning the algorithm design/prototyping in Triana, the method of streamlining the workflow and recreating outside of Triana, and the distribution of work units

and audio). The Java Sound API provides the lowest level of sound support on the Java platform, and is extensible. The API gives the programmer the capabilities to build audio applications such as advanced sounds editors, or graphical wave editing tools. Included in the API are packages to support the MIDI and audio data. These two major modules of functionality are provided in separate packages:

- **`javax.sound.sampled`** - this package specifies interfaces for capture, mixing, and playback of digital (sampled) audio
- **`javax.sound.midi`** - this package provides interfaces for MIDI synthesis, sequencing, and event transport

The **`javax.sound.sampled`** package handles digital audio data, (the Java Sound API refers to this as 'sampled audio') and is used extensively in DART. A data format instructs the system on how to interpret a series of bytes of 'raw' sampled audio data, such as samples that have already been read from a sound file. A data format is represented by an **`AudioFormat`** object, which includes the following attributes:

- Encoding technique (usually PCM)
- Number of channels (1 for mono, 2 for stereo)
- Sample rate (number of samples per second, per channel)
- Number of bits per sample (per channel)
- Frame rate
- Frame size in bytes
- Byte-order of the samples (big- or little-endian)

Many of these attributes have been discussed earlier, however the frame rate and size in this context have not yet been considered. A frame contains the data for all channels at a particular time. For PCM-encoded data, the frame is simply the set of simultaneous samples in all channels, for a given instant in time, without any additional information. In this case, the frame rate is equal to the sample rate, and the frame size in bytes is the number of channels multiplied by the sample size in bits, divided by the number of bits in a byte.

2.6.5.1 File Formats

A file format specifies the structure of a sound file, not only the format of the raw audio data in the file, but also other information that can be stored in the file. In the Java sound API, a file format is represented by an AudioFileFormat object, which contains:

- The file type (WAV, AIFF, etc)
- The files length in bytes
- The length, in frames, of the audio data contained in the file
- An **`AudioFormat`** object that specifies the data format of the audio data contained in the file

It must be noted that JavaSound only natively supports 8- or 16-Bit audio data, with current recognition for sample rates of 11,025, 22,050, 44,100, or 48,000. However JavaSound has been extended with libraries by Tritonus[36] and JavaZoom [37] to support extra features such as MP3 playback and playback using circular buffers, to name a few. These libraries were used in this thesis.

For more information please refer to the JavaSound package in the Java API[38].

[36]`http://www.tritonus.org/`

[37]`http://www.javazoom.net/`

[38]`http://java.sun.com/j2se/1.4.2/docs/api/`

Chapter 3

Requirements Analysis, Underlying Infrastructures and Architecture

3.1 DART Overview

Distributed Audio Retrieval using Triana (DART) is a distributed Music Information Retrieval (MIR) research platform, designed to provide a mechanism for distributing MIR workflows across networks (both local and wide) and for the retrieval and aggregation of the results. DART can utilise a combination of distributed technologies to distribute the workflows and - if the workflow requires - any relevant data associated with it. Through the use of modern distributed computing techniques, DART aims to become massively scalable as the number of participants increases, adhere to copyright laws and restrictions, whilst at the same time enable access to a global database of music for MIR applications and research, completely free of charge.

One of the longer-term goals of the DART system is to develop a distributed Music Recommendation System (MRS) that employs the use of an underlying decentralised sub-system in order to provide a mechanism for distributing workflows across the network, and also for the retrieval and aggregation of results. Therefore this thesis aims to explore massively distributed platforms for use in MIR and investigate a number of distributed computing platforms that exhibit the following features:

1. Allow users to participate at the edges - either to analyse the music on their computers or

to provide resources for the analysis of supplied audio

2. Reduce the connectivity length between participants as much as possible (the number of hops between two end peers should be minimised)

3. Provides a scalable network for the analysis of potentially hundreds of thousands of nodes

One of the key objectives in the investigation of a proposed architecture is to attempt to design a prototype system that satisfies the desired wish list and perform a qualitative analysis on the presented architecture. From here, it is possible to take the hypothetical design and investigate its feasibility given the technologies that are currently available and that have been shown to scale to the level required by DART. This chapter investigates these technologies to examine both their feasibility and suitability for DART.

In one respect, DART at its core can be based on the volunteer computing paradigm that typically employs the use of home computers for the analysis of data. However in a DART MRS scenario, the home users would perform analysis of their own music collection by executing Triana workflows that encompass the analysis component. The MIR algorithms used for the analysis of audio and ultimately the recommendation of music, would be made available to the MIR research community for refinement, suggestions, and also to allow for the advancement of the field. MIR-algorithm researchers and specialists could provide input, ideas and offer improvements to both refine and maximise the benefit of the use of the DART system. Once established, the platform can also be used by MIR researchers to conduct their own experiments.

Both the DART 'system' and the DART workflow (Triana algorithm) are subject to constant change and refinement, especially during a proof of concept and research stage. If the DART algorithm is more suited to recommend music, for example, many more end-users would have an incentive to run the software in order to gain music recommendations - a similar incentive is offered to entice users to use the popular Last.fm and Pandora applications.

However, the goal of DART is to be flexible enough to run a wide variety of MIR applications and it is crucial that the DART environment is capable of self-updating; it is intended not only to provide music recommendations for end users but also to establish itself as a research platform that MIR researchers and scientists can use to test new ideas and algorithms in the MIR field. To this end, researchers will want to design new methods for audio analysis, potentially for both statistical correlation based on metadata and more interestingly, for the analysis of the actual audio itself to extract facets such as tempo, pitch, mood and so on. This approach allows DART to stand out from the existing recommendation systems that generally work by reviewing which songs users own or have recently listened to, as explained in Chapter 2. DART could be used in closed environments where the music content is a known, controlled variable, and also in truly distributed networks with anonymous users.

In the latter scenario, the DART algorithm must account for the fact that confirmation of the details audio files are unknown, or the file's metadata must be 'consulted'. The use of a Shazam-like fingerprinting technology (or the use of the Shazam network) could be used to cross reference the MP3 metadata in an audio file on a worker's machine in order to confirm the details of the track.

The high-level architecture of DART is presented in Figure 3.1, and a more detailed DART workflow diagram is presented in Figure 3.2. DART begins simply with an idea of an MIR experiment that would benefit from large scale distributed execution. This thesis uses a parameter sweep experiment in order to find the optimal parameters (the parameters that yield the highest accuracy) of the Sub-Harmonic Summation algorithm; this is the initial 'idea'.

Layer 1 of the DART architecture shows the prototyping and analysis stage of DART. This includes experiment design and the creation of all the required input data - if any - for the experiments. DART uses the Triana workflow software to create workflows to prototype MIR applications or algorithms, making full use of the Triana Audio Toolkit (developed to be utilised in DART/MIR experiments), as well as the many mathematical and signal processing units available in Triana. As Triana is modular, new units can be created to work alongside the existing units to solve specific goals, allowing the DART scientist to quickly and easily prototype their MIR algorithms. Non-computer science researchers can use Triana to prototype MIR workflows and algorithms visually, without necessarily having to learn how to write any programming code. This is an important facet of DART as MIR is a multidisciplinary field of research.

Layer 2 involves the streamlining and platform independence of a DART experiment - turning the MIR prototype workflow into a standalone application bundle before passing

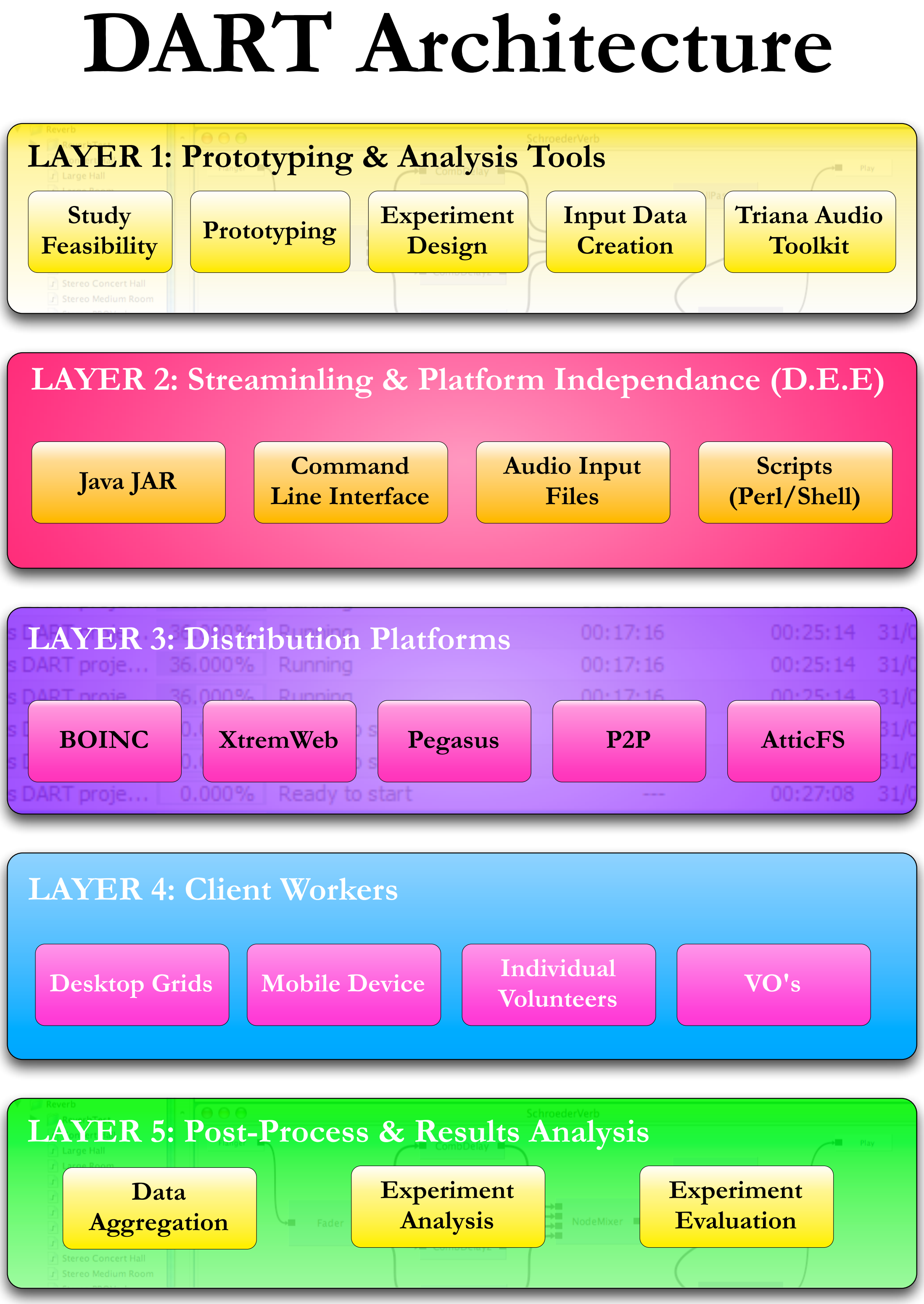


Figure 3.1: An Overview of the DART architecture model.

it on to Layer 3, which involves the use of a distributed platform that allows for the distribution of the DART bundle (which includes the DART binaries and any required input files). This layer also includes the job submission protocols and results caching schemes of any utilised distribution platform. Layer 4 represents the distributed 'workers' in the DART system that actually process the MIR experiments and eventually provide the results back to the DART researcher. The final layer represents the 'post-analysis' performed on the data by the DART researcher.

The overall DART work-flow (i.e. a workflow that a DART MIR researcher would use in order to conduct experiments using the DART system) can be illustrated in several steps and is displayed in more detail in Figure 3.2:

1. Design an MIR Experiment
2. Create a working (scalable) MIR algorithm using workflow prototyping environment (Triana)
3. Design and implement the results analysis application/algorithm
4. Streamline the (Triana) workflow and create a standalone application that can be easily distributed (DEE)
5. Distribute application and retrieve results
6. Analyse and evaluate the results

The above list can be considered as the standard work-flow to follow to create any DART application. The first step is to design an algorithm or workflow which could represent an MIR application or process, and then to extract the algorithm from Triana and streamline the application to run via the DART Execution Environment (DEE), which runs as a standalone Java application that can be easily distributed. There are two main advantages to the DEE:

1. To provide a non-GUI execution environment for Triana DART workflows
2. To streamline the execution environment to a minimal core set for execution of the workflow

It is important to reduce the memory footprint and dependancies of the MIR workflow. The conversion to the DEE can be automated and this is a feature that is currently being implemented by the Triana team (including the author), based on the research presented in this thesis. The standalone DART binary needs to be distributed by a suitable mechanism that can also retrieve the results from the workers. The final stage is to then analyse and evaluate the results - this would be subject to change depending on the type of MIR

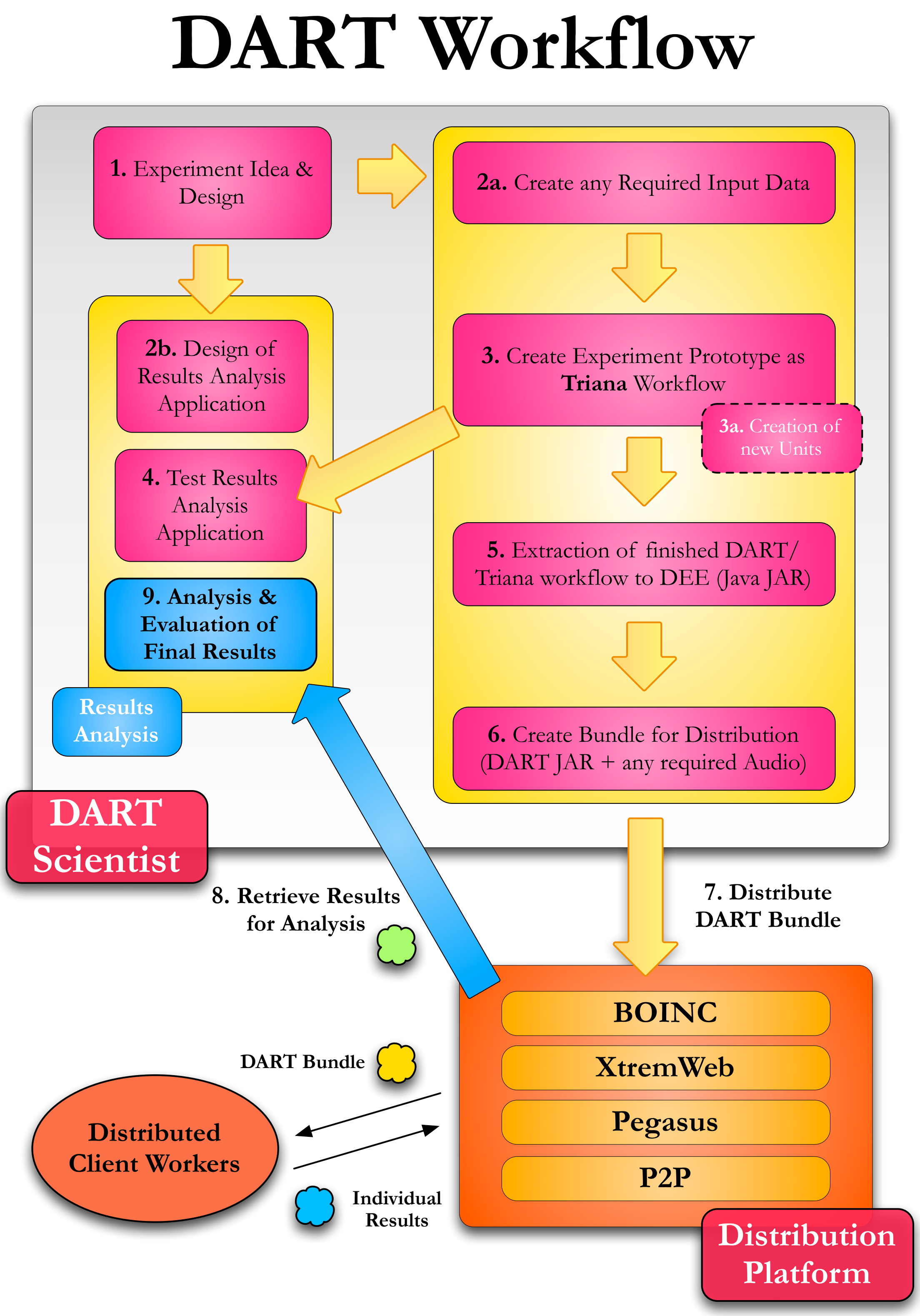

DART Workflow
1. Experiment Idea & Design
2a. Create any Required Input Data
2b. Design of Results Analysis Application
3. Create Experiment Prototype as Triana Workflow
3a. Creation of new Units
4. Test Results Analysis Application
5. Extraction of finished DART/ Triana workflow to DEE (Java JAR)
9. Analysis & Evaluation of Final Results
Results Analysis
6. Create Bundle for Distribution (DART JAR + any required Audio)
DART Scientist
8. Retrieve Results for Analysis
7. Distribute DART Bundle
BOINC
XtremWeb
Pegasus
P2P
DART Bundle
Distributed Client Workers
Individual Results
Distribution Platform

Figure 3.2: An Overview of the DART work flow.

experiment and type of evaluation required. The results could be fed back into a Triana workflow or analysed using any analysis tools that the DART researcher desires.

The goal is for the automation of as many of these steps as possible, with the most important being the automation of the streamlining of a Triana workflow to the DEE. This would enable the DART researcher to think of DART in two stages, simply:

- Algorithm/application design in Triana - *what does the MIR analysis do?*
- Application Distribution and experiment design - *what experiments am i trying to run - what am i trying to analyse?*

Both Figure 3.1 and 3.2 highlight DART's need to be (distribution) platform independent. The following section looks at the feasibility of developing a Peer-to-Peer distributed platform for DART.

3.2 Investigation into the Use of a P2P Architecture

This section details an investigation into the use of a distributed Peer to Peer (P2P) DART architecture to distributed DART work 'bundles' (consisting of a DART binary and any required input data), concentrating on the required work package assignment and results retrieval mechanism. This architecture is used as a basis for simulations carried out in the case study in Section 3.3.1. The results of this case study will help to evaluate different techniques and technologies that are required to achieve the goals of the DART system. Before the DART architecture is defined, an overview of P2P computing is given.

3.2.1 Peer to Peer Computing

Peer to Peer (P2P) computing can be used to describe the communication between two peers. However in distributed computing, a more modern definition of P2P is:

> *"P2P is a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet"*

[84]

Computers and/or devices "at the edges of the internet'" are those operating within transient and often hostile environments. Computers within this environment can come and go frequently; they can be behind a firewall or operating outside of DNS and often have to deal with differing transport protocols, operating systems and devices.

Today's uses for P2P networks range from making telephone calls over the Internet, as in the case of the popular and closed P2P network used by Skype[1], to file sharing applications such as BitTorrent[2]. P2P networks can currently be broadly classified as using "unstructured" or "structured" approaches to the problem of locating resources. Gnutella [2] and Kazaa[3] are examples of unstructured P2P networks, with hosts and resources being made available through *Super Peer* overlays, and without any global over lay planning. Distributed Hash Table (DHT) systems, such as Pastry [85] , Chord [86] and FreeNet [87], use a "structured network" overlay of peers. This structuring consists of

[1]`http://www.skype.com`
[2]`http://www.bittorrent.com/`
[3]`http://www.kazaa.com`

a logical identifier space to which peers and resources are mapped. The identifier is used to locate a particular resource in the network. Peers maintain levels of neighbourhood knowledge about each other enabling application level routing of messages through the network based on the identifier space.

Super peers are used in a number of working real-world Internet scale applications such as KaZaA, Skype, and LimeWire - all of which are capable of scaling to tens of millions of peers. Key knowledge has been gained in applying these layers and as such distributed P2P computing remains one of the only technologies that has been proven at this level of connectivity. Whilst other systems might be theoretically more efficient, super peer technology has consistently been chosen to implement large scale discovery layers for their applications. Super peer technology is simple, which perhaps is in part, a key to its success.

Super peers form a decentralised, connected set of peers that act as the discovery hub for any given application. Rather than searching central databases (e.g. Napster, BOINC), super peers act as a community of connected peers to distribute the discovery information across multiple sources. Historically, the idea of a *super* peer was motivated by the original Gnutella 0.4 protocol, for decentralised searching across networks of peers without any central administration or control. Gnutella employed a flooding mechanism within a horizon of n 'hops', to implement their decentralised connectivity and *reflector* peers were added to act as lookup tables for caching connected peers' file locations. Super peers generally accepted many more connections than regular peers and therefore when searching the network, it was only necessary to flood the super peers, rather than every node on the system. In practice only a small fraction of peers need to be super peers and the resulting structure of the system is said to share the pros and cons of power-law networks [88], where many nodes have a few connections and a smaller number of nodes have many. Whereas the original decentralised Gnutella could only really scale to tens of thousands of nodes, super peer technology could allow such networks to scale to millions of peers.

Super peer layers typically define super peer inter-connectivity, routing or forwarding policies and caching polices using *adverts*. Such properties would also be required if other layers are to be defined for other purposes, such as for application data. However, super peers only cache adverts or locations to files, which are typically stored elsewhere. Real world super peer implementations do not implement authentication policies or subscription control. In typical scenarios, the super peers implement a discovery bootstrap mechanism for data transfers, which are considered independent transactions. However for a number of applications, it is not desirable for data servers to be disassociated from the discovery and caching role process. For example: Super peers typically have a user-definable caching policy, allowing data to be replicated across the participating peers. Such a caching policy is also needed for data caching peers; Super peer connectivity rules (i.e. how many connections they accept to other super peers) and defined routing or forwarding policies are also important for both adverts and data.

P2P data sharing networks have proven to be effective in distributing both small and large files across public computing platforms in a relatively efficient manner that utilises both participants' upload and download bandwidth. Recently, BitTorrent [89] became the most widely used and accepted protocol for P2P data distribution, relying on a centralised tracking mechanism to monitor and co-ordinate file sharing. Although this approach has proved quite scalable and efficient, it might not be appropriate to scientific volunteer computing platforms due to its "tit for tat" requirement that necessitates a ratio between upload and download bandwidth, thus requiring peers to share data if they are recipients of it on the network. Such stringent requirements are likely to prove problematic for volunteer computing platforms; there are numerous security implications of opening additional ports for traffic since every client in the network becomes a server. Further, it is difficult to establish trust for data providers in the network; that is, it is difficult to stop people acting as rogue providers and serve false data across the network or disrupt the network in some way.

3.2.2 A P2P DART Case Study

A P2P DART architecture would focus on forming unstructured P2P networks and therefore needs to employ technologies that can adapt and scale within such an environment. For distribution across dynamic networks, DART proposes the use of the *P2PS* binding for Triana. Peer-to-Peer Simplified (P2PS) [90] was a response to the complexity and overhead associated with JXTA, as used in [62] to aid in the content-based retrieval over a P2P network. As the name suggests, it is a simple yet generic API for developing P2P systems. P2PS encompasses intelligent discovery mechanisms, pipe based communication and makes it possible to easily create desirable network topologies for searching, such as decentralised ad-hoc networks with super peers or rendezvous nodes. P2PS is designed to cope with rapidly changing environments, where peers may come and go at frequent intervals.

At the core of P2PS is the notion of a pipe - a virtual communication channel that is only bound to specific endpoints at connection time. When a peer publishes a pipe advertisement it only identifies the pipe by its name, ID, and the ID of its host peer. A remote peer wishing to connect to a pipe must query an endpoint resolver for the host peer in order to determine an actual endpoint address that it can contact. In P2PS a peer can have multiple endpoint resolvers (e.g. TCP, UDP etc), each resolving endpoints in different transport protocols or returning relay endpoints that bridge between protocols (e.g. to traverse a firewall). The P2PS infrastructure also employs XML in its discovery and communication protocols, which allows it to be independent of any implementation language and computing hardware. Assuming that suitable P2PS implementations exist, it should be possible to form a P2PS network that includes everything from super-computer peers to hand-held smartphone peers.

In the DART framework the distribution policy for Triana must be loosely coupled. Although Triana acts as a manager and processor in the system, the distributed functionality is provided by the DART framework, which implements a decentralised discovery and communication system based on P2PS. This allows Triana workflows to be uploaded to peers for execution and enables users to query the network to locate results and to perform custom searches. The DART system therefore manages the specifics of the network and Triana acts as a client (i.e. the DART manager or user) that both accesses DART and also acts as an end processor to execute workflows that have been previously uploaded for the analysis of the audio. Therefore, Triana does not tie the network together, rather it accesses a loosely coupled framework that allows wide-range distributed Triana entities to communicate via the Internet. The following section discusses the DART framework in more detail and details the mechanisms to facilitate so-called work package assignment and the retrieval of the results.

In order to distribute such work flows, a complex uploading, packaging, and deployment subsystem must exist; not only to bundle the workflow to the remote platform, but also to be able to fan-out these bundles to potentially millions of peers. For this to be achieved it may well not be possible to rely on the current, centralised mechanism employed for data distribution in BOINC as this is not only expensive in both cost (a fairly heavyweight server or cluster would be required) and administration time, but would also be extremely slow for updating the workflows to the network participants as the existing Triana audio toolbox alone is over 10 megabytes in size. Further, such latency is simply not acceptable for the scientists who would like to be able to prototype their ideas quickly and not have to wait a few days for everyone to be updated before they can view the results.

To address this problem, DART proposes the utilisation of a peer-to-peer approach that is based on the super peer architecture but extends this idea to employ the use of secure data servers (called *package repositories*) that cache the workflow bundles for DART, to be able to replicate and decentralise the distribution of the workflows. Other techniques were considered when creating the DART P2P prototype architecture such as bit-torrent, however this is unacceptable because of security constraints. Bit-torrent requires every user on the network to open a port for serving the data and secondly, it provides no scoping environment for DART to be able to restrict which servers can act as a data provider for their workflows. Scientists should be to develop their ideas in confidence (and also to support commercial products) and therefore a user must be able to impose control on who is authorised to distribute the data. For this X.509 certificates (issued by the Certificate Authority (CA)) would be used - in this case the DART manager - and made available to certain participants for authentication to become package repositories.

3.2.3 Proposed Work Package Assignment and Results Retrieval Architecture

The scenario discussed in this section describes the proposed architecture for a decentralised DART network, in which nodes are organised in a super peer topology using the P2PS middleware. P2PS uses the concept of *Producers*, *Consumers* and *Rendezvous Nodes*. Producers provide packages containing workflows and/or results and advertise that they have something of interest to other participants in the network. Consumers (the peers that wish to use available packages, result sets and so on) issue queries in order to search for relevant adverts, while Rendezvous Nodes are responsible for matching queries with adverts within their local cache and responding appropriately.

Consumers can receive advertisements when their query is matched, and these adverts can be used to retrieve the relevant information they require, such as downloading a new workflow package to perform new analysis.

The DART Manager node (the DART researcher) advertises the workflow package representing the new DART bundle (DART Package Adverts) containing MIR workflows that the worker nodes need to run. Workers are available to execute the workflows and issue a package query to download a package in order to start the analysis. The entire DART/Triana workflow is executed by each worker that downloads the package at least once; the workflow is not then broken down into segments and farmed out to separate nodes.

Super peers are used to transmit package queries across the network rapidly. Super peers also act as rendezvous nodes since they can also store package adverts and compare these files with queries issued to discover them, thereby acting as a rendezvous point for both package providers and consumers. Packages may require a reasonable amount of storage space and therefore it is assumed that only some of the peers in the network will cache these files. These peers are Package Repositories (PR) and can also be super peers or worker peers. The owner/administrator of each node can decide if they want their machine to be a super peer, package repository, worker, or any combination of the three.

Figure 3.3 shows a sample topology with 5 super peers (2 of which are also package repositories - normal peers are not considered as package repositories in this example), and displays the sequence of messages exchanged among different nodes when using the package submission protocol. These messages are related to the execution of a workflow by a single worker, labelled as W_0.

The DART manager puts this package on one or more package repositories and propagates a *package advert*, on the super peer network as soon as a new DART workflow package is available. The advert is an XML file that describes the properties of the algorithms to be executed such as any workflow parameters containing the units, platform requirements and information about required input audio data files.

A worker can search the network to verify that a new version of the package is available by sending *package query* - an XML document containing the hardware and software features of the worker node - that travels the network through the super peer interconnections (message 1 in Fig 3.3). A query succeeds when it matches an advert of a package that

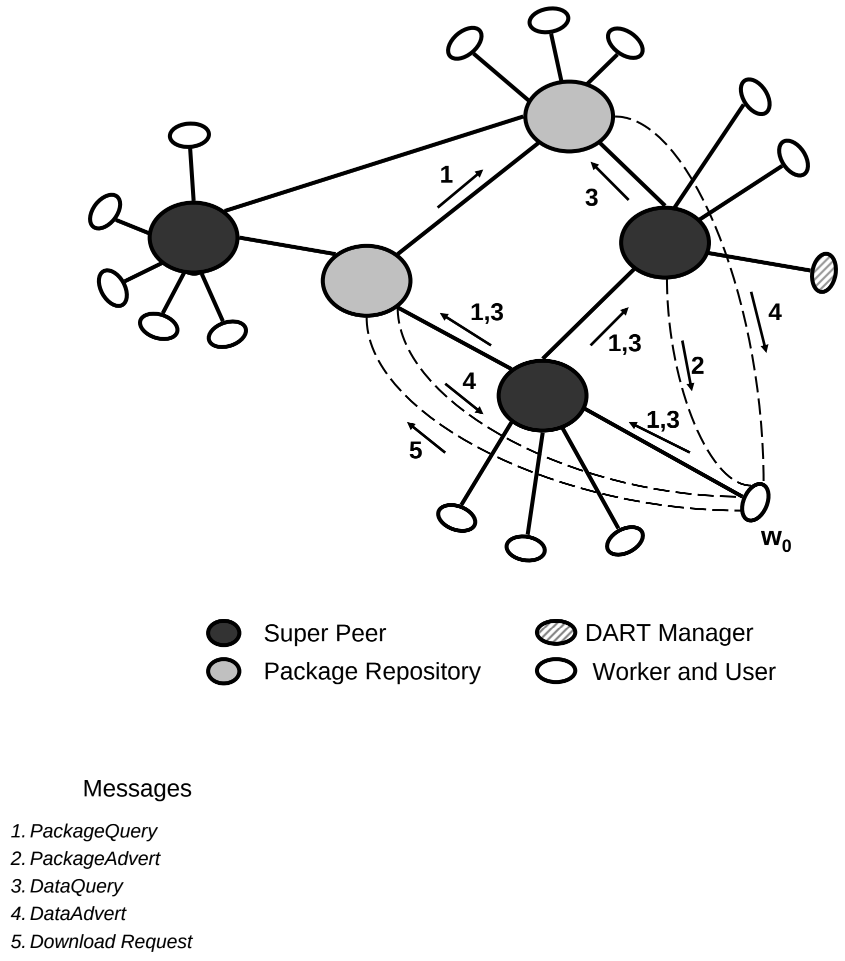


Figure 3.3: Super peer protocol for the dissemination of workflow packages: sample network topology and sequence of exchanged messages to execute one package cycle.

can be successfully executed by the requesting worker. This package advert is then sent directly to the worker.

The worker must then search for a package repository that stores the updated workflow package and sends a *data query* to the network. As multiple package repositories can match the query, a matching repository does not send the package directly to the worker in order to avoid duplicate file transmissions. Conversely, the repository returns only a *data advert* to the worker.

A worker can choose a repository based on criteria such as the distance (number of hops) of repositories, or their (available) bandwidth, and can then initiate a download operation from the selected repository. The rendezvous nodes can also query for new packages to download and store, and then re-advertise them in order to propagate them onto the network in a decentralised fashion, rather than relying on one centralised node to do this, as is the case for systems such as BOINC - although BOINC does employ some data caching schemes that optimises bandwidth usage (encountered later on in the thesis).

This protocol allows for the progressive dissemination of packages on different repositories - initially the packages are stored on one (or few) repositories - however when a worker downloads a package, if the local super peer also plays the role of a repository, the package is first downloaded and cached by this super peer, then forwarded to the worker. In the future another package query can be matched directly by this package repository, significant saving time in the querying phase and enabling the simultaneous retrieval of packages from different repositories.

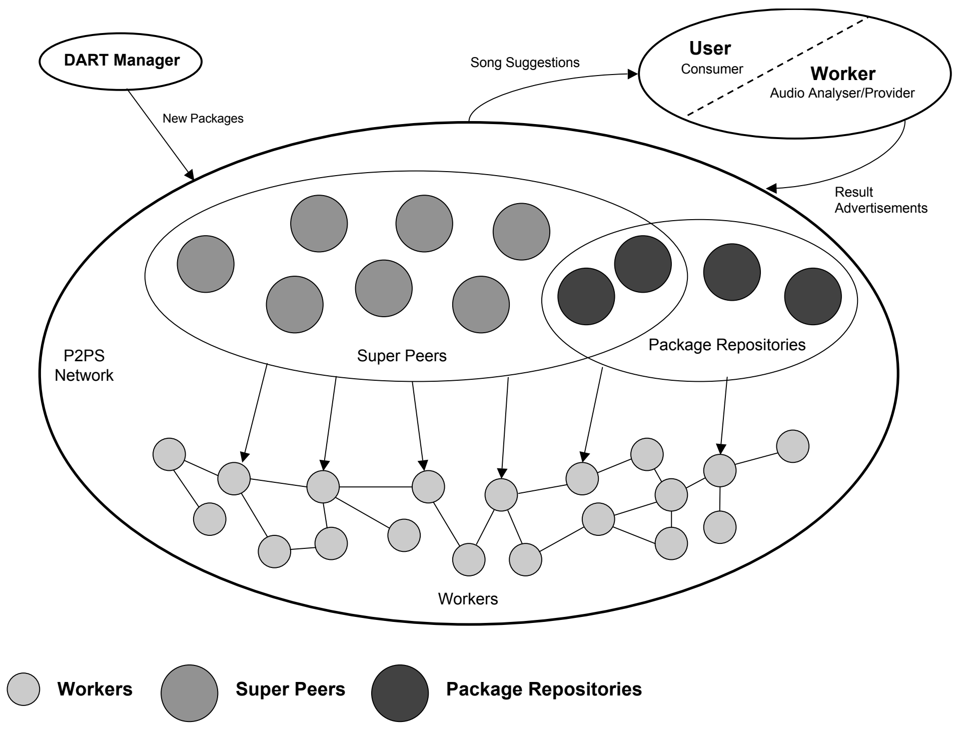


Figure 3.4: High-level overview of the DART system, showing the various peers and their connectivity.

Once a worker has received the updated package, the workflow is executed and the worker can begin to conduct the require MIR analysis. Once a work cycle is complete and there are results to present, the worker then creates an *results advertisement* containing the results and generated metadata. As the actual results generated would be extremely small in size in this DART system (estimated to be text files of only under 10KB), the super peer can cache them and make them available.

Each worker on the network can be a results provider as well as a ‘user’, as a worker can query for results (a suitable music/song suggestion as generated on the super peer, in the case of a DART Music Recommendation System). There is no central results collector, but rather a fully decentralised model is used, allowing the results to propagate through the network hop by hop, to be stored on the super peers. The super peers can process the metadata and issue an XML results advertisement on receipt of a results query from the user.

3.2.4 Peer Overview

This section gives a brief overview the role that each node on the network plays (shown in Figure 3.4 above), and the jobs associated with that role:

DART Manager

- Creates new workflows and Triana units
- Advertises new DART Packages

User

- Queries Results
- Downloads/Receives Results

Worker Nodes

- Queries New Packages
- Downloads Package
- Works/processes Workflows
- Advertises Results

Package Repositories

- Query and Download New Packages (Stored locally)
- Advertise Packages

Super Peers

- Caches Advertisements
- Performs simple analysis of results received from workers

3.2.5 Workflow Design

The workflow created by the DART manager and distributed to the other peers on the network, will differ depending on the MIR analysis that the DART researcher wishes to conduct. The algorithms used for the analysis and recommendation of music will be made available to the MIR community for refinement, suggestions, and also to allow for the advancement of this field of research as MIR-algorithm specialists provide input ideas and offer improvements to both refine and maximise the benefit of the use of the DART system.

3.3 Simulation Results

This section displays the results of the DART P2P scalability simulations, run at the Institute of High Performance Computing and Networking in Italy. The results of these simulations were published in [15]. These simulations help analyse the scalability of a P2P DART architecture based on P2PS, presented in the previous section.

3.3.1 Distributed Simulations

In order to demonstrate the scalability of the presented DART P2P architecture shown in the previous section, as well as highlight its ability to distribute workflows over a ubiquitous P2P network as the number of participants increase, a study was conducted in which a simulation of the DART scenario was run at ICAR-CNR[4] (The Institute of High Performance Computing and Networking) in Italy. A simulation analysis was performed by means of an ad-hoc event-based simulator, written in C++, to evaluate the performance of the package dissemination protocol. The simulator is designed around objects that simulate the behaviour of P2P components and are able to exchange messages to and from them. Every time an object receives a message, it performs a related procedure and

[4]`http://www.icar.cnr.it/`

can send new messages to other objects. The P2P simulator takes three files as input: a parameter file, a network file (specifying the connections between super peers), and a cluster file (specifying how many peers are attached to each super peer). As outlined in [91], the objects types defined in the simulator are the following:

- **Super Peer** - models a super peer
- **Peer** - models a simple peer
- **UserAgent** - generates queries on behalf of a user.
- **Event** - generates a message exchanged among the UserAgent, Peer and SuperPeer)
- **EventDispatcher** - manages events, stores them in a queue ordered by message delivery times, and dispatches them to destination objects.)

The DART-specific simulation scenario is described in Table 3.1. In this case, a workflow package of around 10.4MB in size (the current size of the Triana audio toolkit) is to be distributed to the worker nodes on the network. In these simulations it is assumed that the workflow can be executed by any worker and that only super peers can be package repositories.

This scenario simulates the performance and behaviour of a distributed P2P network with 1,000 to 20,000 workers, with a maximum value of 2,000 super peers (the number of super peers is assumed to be 10% of the number of workers). The simulation takes into account the transient nature of a network environment; workers can disconnect and reconnect to the network at any time. This implies that the download or execution of a workflow fails upon the disconnection of the corresponding worker. It is assumed that connections between two adjacent super peers have a larger bandwidth and a longer latency than local connections (i.e. between a super peer and a local simple node).

In the simulation scenario, each worker downloads and executes a workflow package by issuing a package query and follows the distributed protocol described earlier. If the download operation fails due to a worker disconnection, a new package query is forwarded and the procedure is repeated. The number of available package repositories was varied from 1, to 50% of the number of super peers; one repository provides the workflow package from the beginning, while the others act as *cachers*, as they can download, store and provide data on the fly.

Scenario feature	***Value***
Number of workers, or simple peers, N_{peer}	1,000 to 20,000
Number of super peers, N_{speer}	100 to 2,000
Average number of workers connected to a super peer	10
Maximum number of neighbours for a super peer	4
Average connection time of workers	4 hours
Average disconnection time of workers	1 hour
Number of package repositories	1 to 50% of N_{speer}
Size of input data files	10.4 Mbytes
Latency between two adjacent super peers	100 ms
Latency between a super peer and a local worker	10 ms
Bandwidth between two adjacent super peers	2 Mbps
Bandwidth between a super peer and a local worker	1 Mbps
Mean workflow execution time	10 hours

Table 3.1: Simulation scenario.

Simulations have been performed to analyse the overall *dissemination* time, T_{diss}, defined as the time needed to propagate the workflow package to at least 95% of the workers. The overall dissemination time is crucial to determine the rate at which workflow packages can be retrieved from the package repositories in order to guarantee that the workers are able to remain up to date with new packages. The average time needed to perform a single download operation, T_{dl}, is also calculated.

The average 'utilisation index' of package repositories, P_{act}, is defined as the fraction of time that a package repository is actually utilised - the fraction of time in which at least one download connection, from a worker (or another repository), is active with this repository. The value of P_{act} is averaged on all the repositories and can be seen as an efficiency index.

Figure 3.5 shows that T_{diss} decreases as the number of package repositories increases, as worker nodes can exploit a higher level of parallelism and download workflow packages from multiple (possibly closer) repositories and also because the repositories themselves are under less stress and therefore data download time decreases. Conversely, if the number of repositories is constant, the dissemination time increases with the number of workers due

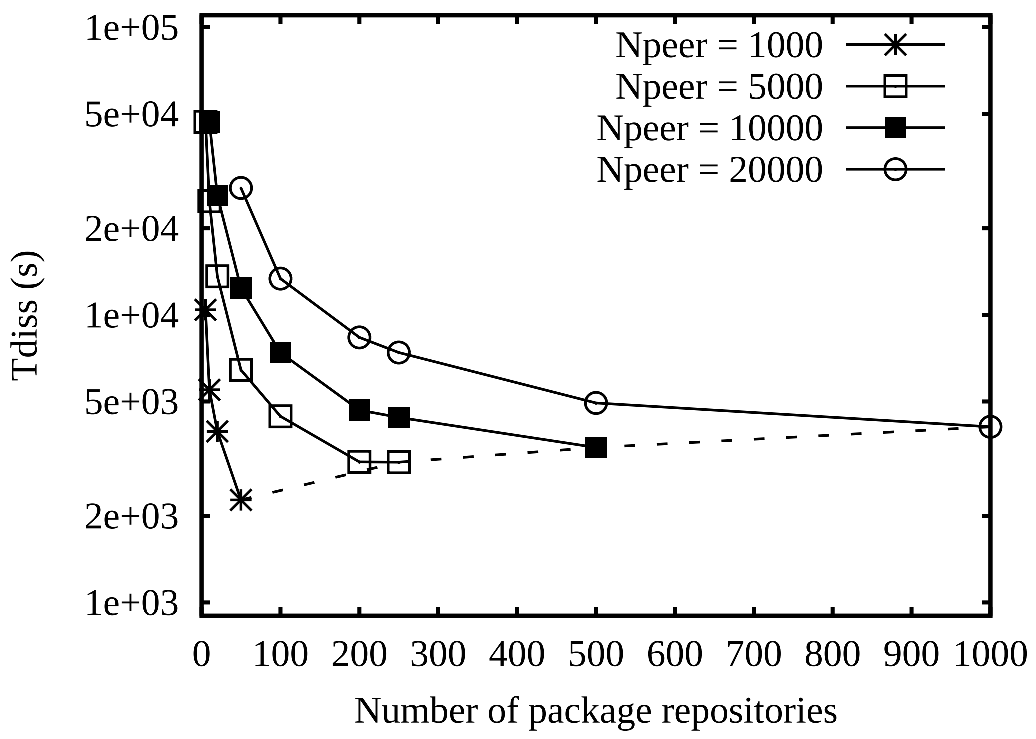


Figure 3.5: Time at which 95% of workers have downloaded a new version of the workflow package from a package repository.

to a high number of requested downloads; a single repository has to serve more workers on average.

In these simulations, the protocol is shown as *scalable* when one observes the results obtained with a fixed percentage of repositories. As an example, observe the results when the number of repositories is set to 5% of peers (which is also 50% of super peers, see dashed line in Figure 3.5). As the number of peers increases, the dissemination time also increases very slightly (but much less than the number of peers) - with 1,000 peers and 50 repositories, dissemination time is approximately 2,200 seconds; however with 20,000 peers and 1,000 repositories, dissemination time is approximately 4,100 seconds.

Figure 3.6 shows the average download time of a package from a package repository by a single worker, when a worker disconnection does not interrupt the download operation. The results here are analogous to the behaviour previously observed in Figure 3.5, however the values are lower. The download time decreases as the number of repositories increases, and the download time will increase as the number of workers increases.

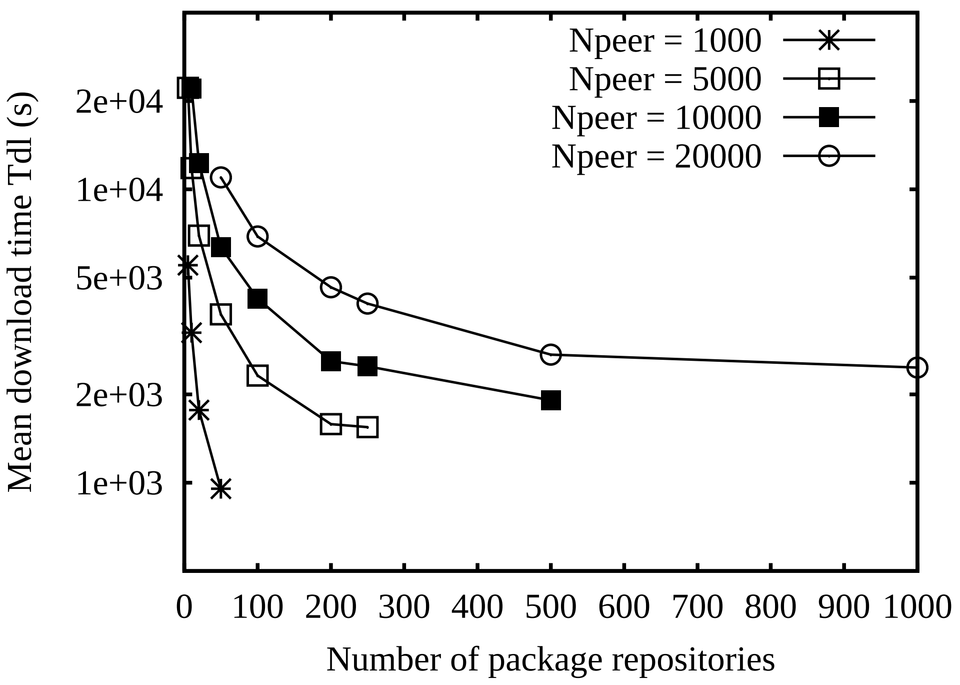


Figure 3.6: Average download time of single worker from package repository when worker disconnection does not interrupt the download.

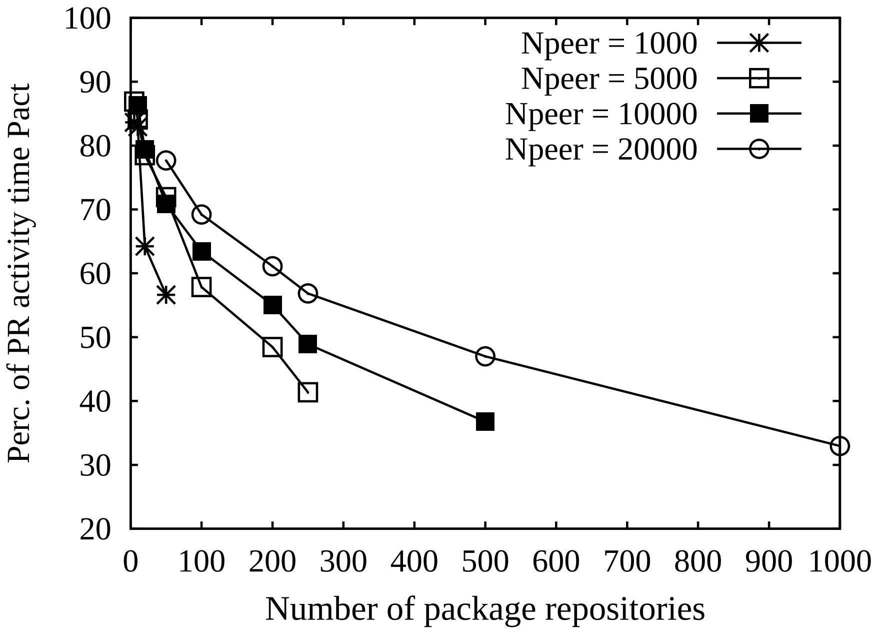


Figure 3.7: Percentage of time in which a package repository is actually exploited (at least one download in progress).

Figure 3.7 shows the percentage of time in which a repository is actually exploited (the time that there is at least one download in progress). As more repositories become available, the percentage decreases. Therefore one should be hesitant in setting up a high number of repositories, as while this can slightly decrease dissemination time, it can also lead to largely under-exploited package repositories. The utilisation of a given number of repositories also increases as the network becomes bigger and more workers need to download the workflow package. This is another verification of the scalability behaviour of the protocol [5].

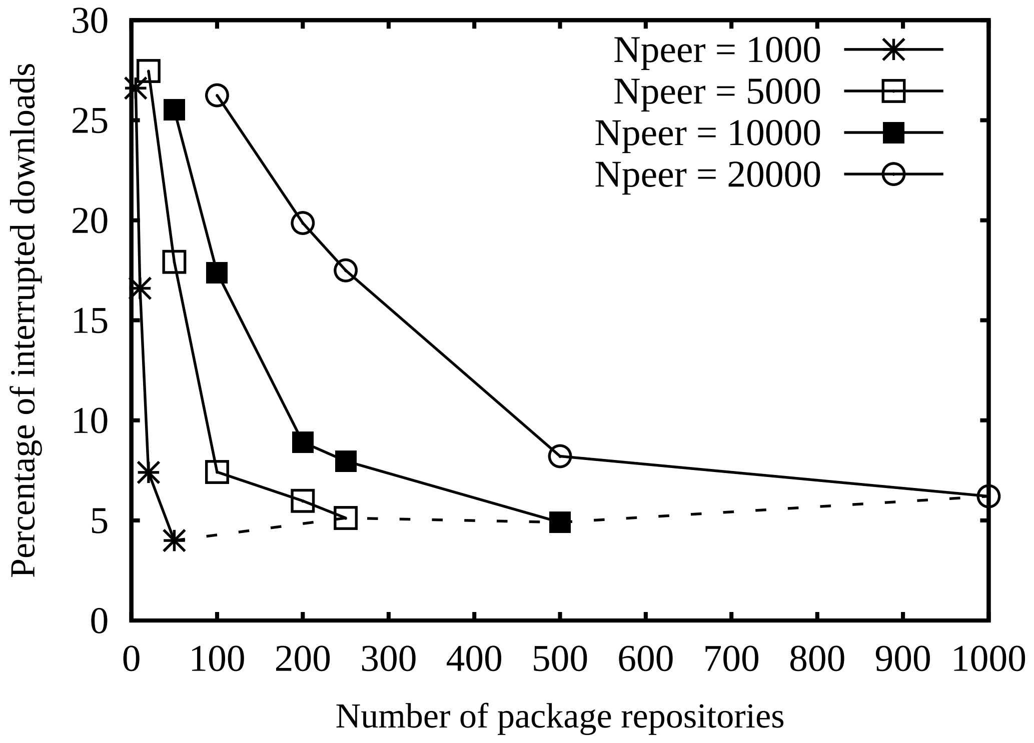


Figure 3.8: Percentage of interrupted downloads.

Figure 3.8 displays the percentage of download operations that are interrupted due to the disconnections of corresponding downloading workers. In this simulation, only results for which this percentage is lower than 30% are displayed. It was observed that the percentage of interrupted downloads decreases as the number of repositories increases. Overall, the download time decreases if more repositories are available (see Figure 3.6); a worker then has more chances to conclude its download operation. Finally, if the percentage of

[5]This percentage never reaches 100% because a data cacher (a package repository that has no data at the beginning) must download data from another repository before it can serve a worker.

repositories (with respect to the number of peers) is set to a given value (for example 5%), the percentage of interrupted downloads is almost constant (see dashed line), providing further confirmation on the scalability of the protocol.

3.4 Conclusions

The presented DART system case study makes novel use of dynamic workflows in a massively distributed P2P environment. Remote processing on peers in the DART architecture is performed through the execution of workflows designed for MIR, and dynamically propagated through the network and discovered using a super peer mechanism. The scientist or analyst (DART manager) that creates the workflow is able to modify it and retransmit it to the peers, fine tuning the application based on the previous results.

Extensive simulations have been performed using an ad-hoc event simulator to explore how the transmitted workflows will propagate throughout the network as the number of peers and super peers increases. The results show that the network will scale as the number of members increases as long as the number of super peers that act as data providers increases by the same ratio. This shows that a DART P2P platform based on the architecture outlined in this chapter, should be capable of operating at an Internet scale with acceptable performance.

3.4.1 P2P Technology Feasibility

The DART P2P simulations run at ICAR-CNR showed that:

- 1) With the requirement that both a package and data can be distributed to the nodes, using package repositories for storing code and super peers for indexing the data files, that the system can scale well as the number of participants increases.
- 2) The more repositories and super peers, the shorter the download time - up until a saturation point. Generally it was found that around 2-3% of the peers on the network must be super peers/package repositories for the system to scale well.

The use of P2PS is integral to the architecture descried in this chapter. When initially created, P2PS aimed to provide a simple platform on which to develop peer-to-peer style applications, hiding the complexity of other similar architectures such as JXTA. The functionality provided by P2PS can be considered a subset of that provided by JXTA and the more limited (yet more focussed) functionality of P2PS can be advantageous in situations whereby the complexity of the JXTA implementation creates undue difficulties. For most P2P applications, the subset of functionality provided by P2PS is sufficient, with P2PS providing a lighter and cleaner implementation than JXTA.

Unfortunately, during the term of this thesis the development of P2PS ceased and the infrastructure was no longer actively supported. As such, it would not make for a suitable distribution mechanism for DART.

The JXTA platform could potentially serve as a suitable infrastructure in lieu of the availability of the P2PS middleware support. The continuing JXTA development has made ongoing efforts in order to provide a standard way to communicate in a P2P network, and its availability to peers behind firewalls and NATs in the JXTA implementation is a useful and necessary feature for a DART implementation. Further, JXTA does not limit development to a specific programming language or networking platform and uses standard XML to create messages.

However, in [92] and [93] many of the disadvantages of the JXTA platform are listed, such as the deep complexity of the JXTA framework, stating that *"A developer may find it too time consuming and unnecessarily hard to keep track of its specification"*. [93] continues:

- *JXTA does not attempt to address how community services are invoked. Several standards for service invocation exist, such as the Web ServicesDescription Language (WSDL), but none has been specifically chosen by the JXTA Protocols Specification*
- *The network overhead of XML messaging might be more trouble than its worth for small standalone applications. It might just be easier for the developer to create their own protocols if they have no intention of taking advantage of JXTAs capability to incorporate other P2P services into the application*

JXTA has not yet become standardised, which could be partly down to the complexity of using the platform. There is also no current infrastructure for executing jobs or managing job re-runs; this would need to be developed as part of DART and is a large undertaking. Other more established distributed computing platforms such as BOINC and XtremWeb (discussed shortly) already integrate this key feature.

A paper called *BOINC on JXTA - Suggestions for Improvements* by Marcin Cieslak [94][6] outlines some of the issues that the BOINC developer faces when trying to integrate JXTA and BOINC in order to create a more easily scalable BOINC (P2P) infrastructure, mainly as a data distribution mechanism:

> *However while working with [JXTA] many mistakes can be noticed. The documentation for many important elements hardly exists. The tools for XML processing, used mainly with advertisements, are difficult to utilize. Despite of supposed popularity of JXTA there are still no professional products based on it. Implementation of JXTA protocols still has many undocumented elements. Unfortunately during the experiments it showed up how far from perfection is JXTA. Many initial solutions had to be changed. Despite of the promises from Sun and big hopes, JXTA is a platform which wont become a P2P standard to fast.*

[94] goes on to list a number of suggestions which could improve the JXTA platform and allow for easier integration with current systems, and also provides an excellent overview of distributed computing techniques, technologies, and an overview of BOINC and JXTA. *Attic* is one technology (discussed shortly) which aims to become a Peer-to-Peer (P2P) data distribution mechanism that provides a means for volunteer computing projects such as BOINC to become more scalable when running data intensive projects. The integration of BOINC with a P2P data distribution mechanism could combine the best of both worlds; a well established platform for massively distributed volunteer computing, and a P2P data distribution mechanism to allow for data intensive projects to become more scalable.

It can be concluded that the technology to support an architecture as presented in this chapter is not yet mature enough, but forms the basis of a future design for such a dynamic P2P system. The rest of the thesis will focus on the core components and features needed for DART and evaluates the more robust technology for the implementation of DART's requirements.

[6]translated from Polish, available on the BOINC website: `http://boinc.berkeley.edu/cieslak.pdf`

The next section discusses two of the main relevant distributed computing technologies that DART is able to utilise to run wide scale MIR experiments - BOINC and XtremWeb.

3.5 Relevant Distributed Computing Technologies

This thesis integrates and compares two key distributed computing technologies, namely XtremWeb and BOINC. This section aims to give a brief overview of these two technologies, showing the different approaches to distributed computing.

3.5.1 XtremWeb

Grid and Desktop Grid computing are a form of distributed computing in which an organisation uses its existing computers (desktop and/or cluster nodes) to handle its own computationally expensive tasks. This differs from *volunteer* computing (discussed shortly, whereby anonymous users donate their resources) in that the computing resources can be trusted and therefore there is typically no need for any redundant computing measures[7]. It may in fact be desirable to have the computation be completely invisible and out of the control of the user and deployment to the client is typically automated. This allows a school, university or company to setup and run a global computing or P2P distributed system for either a specific application or a range of applications or experiments.

XtremWeb[8] is open source Java software that allows users to build lightweight Desktop Grids by gathering and exploiting the unused resources of desktop computers (CPU, storage, network). XtremWeb turns a set of volatile Internet or networked resources into a runtime environment executing highly parallel applications. Like the other large scale distributed systems, the XtremWeb platform uses either remote resources (such as PCs, workstations, servers) connected to the Internet, or a pool of resources inside a local area network. Participants of an XW platform cooperate by providing their CPU idle time.

XtremWeb can be considered ‘in between’ a pure Desktop Grid system and a Volunteer Computing system; Grid Computing generally refers to the sharing of computing resources

[7]Redundant computing is a mechanism for identifying and rejecting erroneous results. Public-resource volunteer computing projects commonly deal with erroneous computational results. These results arise from malfunctioning computers or malicious participants.

[8]`http://www.xtremweb.net/`

within and between organisations, where each organisation can act as either producer or consumer of resources (frequently explained by a common analogy comparing grid computing and the electrical power grid). These organisations are mutually accountable; if one organisation misbehaves, the others can respond by refusing to share resources with them[9].

XtremWeb is not limited to centralised architectures and is currently being extended to a hierarchical design. Also, Workers may send their results directly to Clients to reduce bandwidth.

3.5.1.1 XtremWeb-HEP

XtremWeb-HEP (XtremWeb High Energy Physics) is a middleware and desktop grid platform based on XtremWeb, and developed at Laboratoire de l'Accélérateur Linéaire (LAL), Paris[10], to deploy a distributed data processing infrastructure. There is an XtremWeb testbed at LAL, making the laboratory's computation resources available; XWHEP is also installed on the GRID5000 cluster, discussed shortly.

3.5.1.2 Architecture

XWHEP uses a three-tier architecture, as shown in Figure 3.9. One tier consists of one or more XWHEP *Servers*, installed and maintained by system administrators to host centralised XWHEP Services such as the *Scheduler* and the *Result Collector*. A second tier, *Workers*, are installed by volunteers on their PCs to allow their computing resources to be aggregated within an XWHEP infrastructure. *Clients* are installed by users (MIR or DART scientists, for example) on their PCs to interact with the XWHEP infrastructure. The XWHEP software client permits users to manage the Servers and utilise the distributed resources. How DART utilises and integrates with with XWHEP will be discussed later in the thesis in the Design and Implementation chapters.

The XWHEP central services allow the XWHEP Administrator to manage registered

[9]This is different from volunteer computing. 'Desktop grid' computing - which uses desktop PCs within an organisation - is superficially similar to volunteer computing, but because it has accountability and lacks anonymity, it is significantly different.

[10]`http://www.lal.in2p3.fr/`

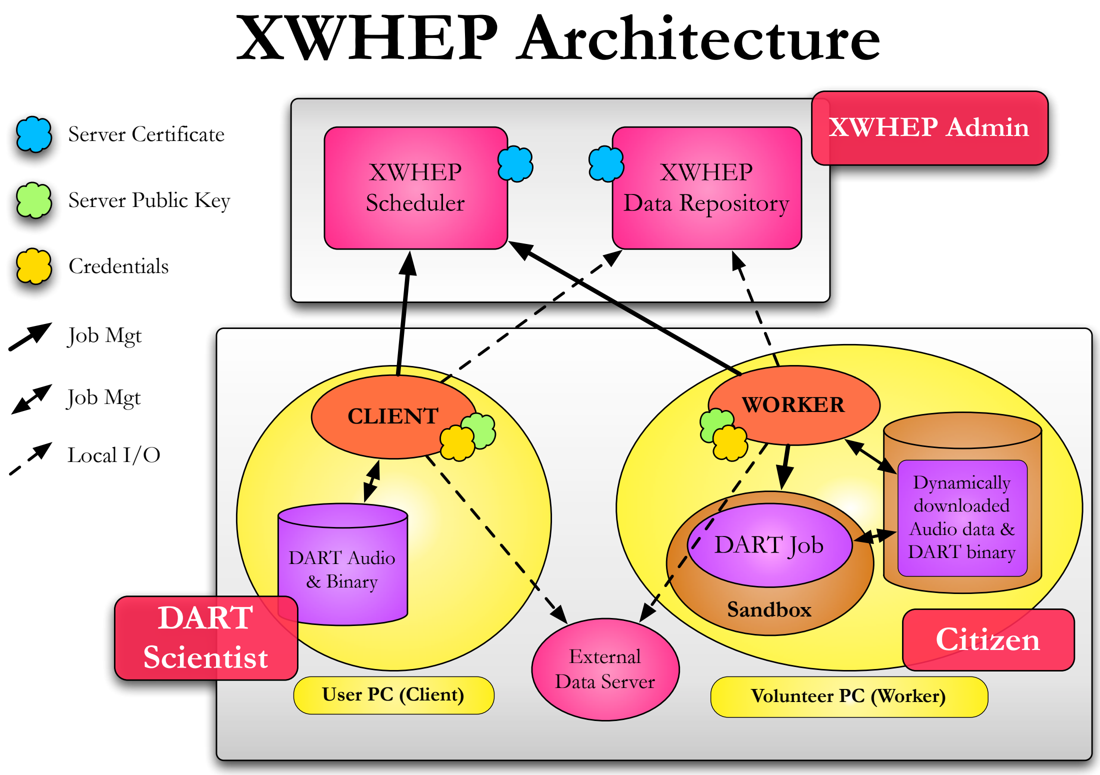


Figure 3.9: The XWHEP architecture.

applications. An XWHEP Client (such as a DART administrator) prepares data that is needed to successfully compute jobs. This data can either be stored in the XWHEP infrastructure, or in any location, as long as the data can be described by a URI and is network accessible[11]. An authorised XWHEP Client registers the application on the XWHEP infrastructure and prepares jobs containing a reference to a registered application, optional parameters, and optional references to any additional files. Finally, the Client submits the prepared jobs to the XWHEP Scheduler.

Independently, Workers contact the Scheduler to get jobs suitable for their architecture. In response, the Scheduler sends a suitable job description to a Worker. For each file referenced by the job which is not present in the local cache of the Worker yet, the Worker fetches the file from the XWHEP Data Repository or from an External Data Server. As soon as a job has finished on the Worker side, the Worker contacts the XWHEP Result Collector to send the results.

XWHEP can also 'bridge' to EGEE - the Enabling Grids for E-sciencE project. The

[11]There is no limit on data size, however the data size will naturally effect upload and downloads speeds.

European EDGeS (Enabling Desktop Grids for e-Science[12]) project [95] was created with the aim of bridging the EGEE Grid with both BOINC and XtremWeb. One of the main achievements of EDGeS is the 3G bridge (Generic Grid to Grid bridge). More information on this is presented in [96].

3.5.2 EGEE / EGI

EGEE[13] (now EGI [14]) makes grids available to scientists and engineers. The infrastructure aims to aid resource intensive research in a wide range of scientific research areas, especially high energy physics and life sciences where the computing demand is high. EGEE offers an infrastructure of more than 68,000 CPUs available to users 24 hours a day, 7 days a week, and around 20 Peta-Bytes of storage space, with a throughput of around 150,000 jobs a day. EGEE is on 250 sites across 50 countries and supports massive data transfers in excess of 1.5 GB/s

Various computing grids are powered by different, incompatible grid middleware stacks and users of one grid have difficulty accessing resources managed by other grids. An important grid type is the 'Service Grid' (SG), which aggregates geographically distributed resources and treats them as a coordinated federation of independently managed computing sites. In Service Grids the job servers do not wait for computing resources to pull jobs from servers, but instead broker incoming jobs and push them to computing resources as required. Two of the most notable SG middleware stacks are Globus and gLite.

EGEE is built on the gLite[15] middleware, which supports resource brokers, computing elements, storage elements, security services, information systems, worker nodes and user interfaces. The basic building blocks of the gLite middleware are the Worker Nodes. These machines are responsible for the execution of applications and can be considered nodes of a cluster. A group of Worker Nodes is attached to a Computing Element, which provides a gateway to the worker nodes and therefore provide the grid resources.

gLite can be more easily integrated into a Java framework (such as DART) by using *jLite*[16]. jLite is a Java library that provides a simple API for accessing gLite based grids.

[12]http://edges-grid.eu/
[13]http://www.eu-egee.org/
[14]http://www.egi.eu
[15]http://glite.cern.ch/
[16]http://code.google.com/p/jlite/

The gLite middleware can be difficult to use, especially in a Java environment and jLite helps to reduce the time and effort needed to build a cross-platform grid application on top of the EGEE grid infrastructure.

The EGEE project officially ended on April 30th 2010 and the research was continued as part of the **EGI** project. This thesis refers to work carried out before the EGI project and makes specific use of EGEE and gLite (in the form of jLite); as part of EGI the development of the gLite middleware was taken over by the European Middleware Initiative (EMI[17]). EGI now uses Grid middleware that utilises many components that came from the gLite middleware.

3.5.3 GRID5000

The GRID5000[18] project aims at building an experimental Grid platform aggregating 9 sites[19], geographically distributed across France[20]. The main purpose of this platform is to serve as an experimental testbed for research in grid computing, acting as a scientific tool for the study of large scale parallel and distributed systems. It aims to provide a highly reconfigurable, controllable and monitor-able experimental platform to its users. The initial aim of the GRID5000 project was to reach 5000 processors on the platform. It has been reframed at 5000 *cores* and was reached during Winter 2008-2009. 17 laboratories in France are involved, all with the objective of providing Grid researchers with a testbed allowing experiments in all the software layers between the network protocols up to the applications:

- Applications
- Algorithms
- Runtime
- Middleware
- Operating Systems
- Network protocols

[17]`http://www.eu-emi.eu`

[18]`https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`

[19]GRID5000 locations include: Bordeaux, Grenoble, Lille, Lyon, Orsay, Rennes, Sophia-Antipolis, Nancy and Toulouse

[20]Porto Alegre, Brazil will become the first site abroad.

The motivation for GRID5000 emerged through the analysis and discussion concerning the current methodologies used for scientific research in the Grid domain. [97] presents the rationale for GRID5000 (however the report is in French).

XWHEP can be installed on the GRID5000 infrastructure, making these resources available to XWHEP users that have been granted sufficient privileges.

3.5.4 BOINC

BOINC[21] (Berkeley Open Infrastructure for Network Computing) [98] is a software system for creating and operating public-resource computing projects. It is written in C++ and supports a diverse range of applications, including many with large storage or communication requirements. Contributors (workers) can participate in multiple BOINC projects and can specify how their resources are allocated among these projects. [99] provides an excellent insight into the computational and storage potential of volunteer computing.

BOINC has enjoyed some high profile and popular projects such as SETI@Home (Search for Extra Terrestrial Intelligence) [14], where millions of computers in homes and offices around the world are used to analyse radio signals from space, looking for patterns in order to detect intelligent life outside Earth. Folding@Home [100] studies protein folding, mis-folding, aggregation, and related diseases. It uses novel computational methods and distributed computing to simulate time scales thousands to millions of times longer than previously achieved.

The Einstein@Home project [101] was created to detect certain types of gravitational waves, such as those from spinning neutron stars, that can be detected only by using highly selective filtering techniques that require extreme computing power. Data from the Laser Interferometry Gravitational Observatory (LIGO) and the British/German GEO6000 gravitational wave detector is analysed.

Other successful (in terms of user participation) projects include multiple projects at CERN (home of the Large Hadron Collider). A list of all the projects known by the BOINC team is given at: `http://boinc.berkeley.edu/wiki/Project_list`.

A public-resource or volunteer computing project must attract participants with an in-

[21] `http://boinc.berkeley.edu`

centive - otherwise there will be very few volunteers. Interesting and worthwhile projects are required to attract and maintain a large user base. A project must explain and justify its goals, providing compelling views of local and global progress. A Music Recommendation System with high-quality recommendations based on advanced low-level audio content analysis could be just the incentive required to propel MIR into the mainstream. Users would not only be 'donating' their own time (computing resources), but also receiving useful information in return. A similar incentive is offered to entice users to use the popular Last.fm and Pandora applications. The processing of audio locally on a worker machine and sending back only metadata also opens up a vast amount of 'test data' for analysis.

3.5.4.1 Basic Architecture

A BOINC 'project' corresponds to an organisation or research group that does public-resource computing. It is identified by a single URL, which is the project's BOINC home page (e.g.: `http://boinc.berkeley.edu/wiki/Einstein@Home`). Volunteers can participate by visiting the projects web site and filling out a registration form, then running a BOINC client program on their computers (Windows, Linux & OSX), downloaded from the BOINC website[22].

The volunteer or client then 'attaches' itself to any set of projects, controls the resource share devoted to each project, and limits when and how BOINC uses their computer resources. The client software can operate in several modes:

- As a screensaver that shows graphics of the running applications
- As a service that runs even when no users are logged into the machine, and logs errors to a database
- As an application that provides a tabular view of projects, work, file transfers, and disk usage
- As a UNIX command-line program that communicates through stdin, stdout and stderr, and can be run from a cron job or startup file

An overview of the BOINC architecture is shown in 3.10. The BOINC project server consists of a relational database that stores descriptions of applications, platforms, ver-

[22]`http://boinc.berkeley.edu/download_all.php`

sions, 'workunits', results, accounts, and teams. Scheduling servers handle Remote Procedure Calls from clients, while data servers handle file uploads. File downloads are handled by HTTP. BOINC provides tools to control projects (creating, starting, stopping and querying), as well as adding new applications, platforms and application versions, creating workunits, and monitoring server performance.

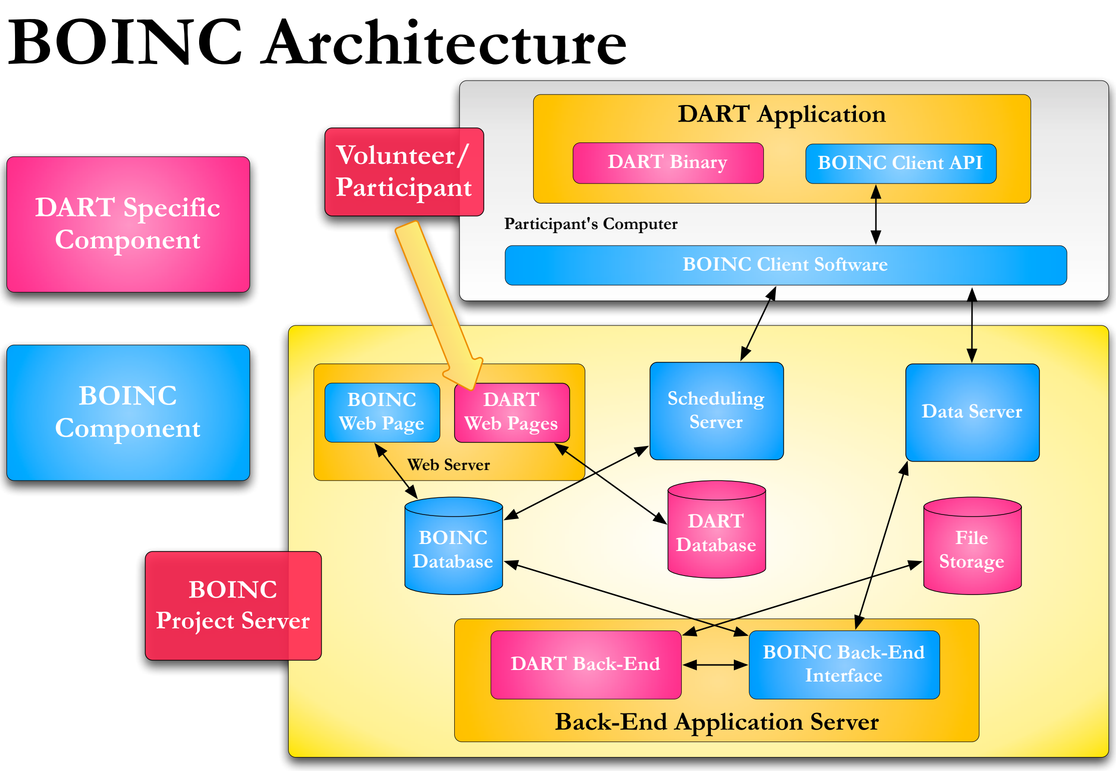


Figure 3.10: BOINC architecture.

BOINC utilises a classic client-server topology. Once the BOINC software is installed in a machine, the server starts sending tasks to the client. The client receives a workunit, the operations are performed client-side and the results are uploaded to the server-side. If there is more work, this process is repeated with a new workunit - there is no communication between clients, as is the case with P2P systems.

A workunit contains the application, a set of input files, and sets of command-line arguments and environment variables, so if the input files are already on the worker machine, the BOINC server can simply instruct the worker to compute the workunit again, but perhaps with a different set of parameters.

Each workunit has a minimum criteria such as minimum CPU, RAM and storage requirements, as well as a deadline for completion. A 'result' consists of a reference to a workunit and a list of references to output files. BOINC files have unique names and are read-only. The description of a file includes a list of URLs from which it may be downloaded or uploaded and each file can have associated attributes indicating, for example, that they should remain resident on a host after their initial use, that they must be validated with a digital signature, or that they must be compressed before network transfer.

When the BOINC client communicates with a scheduling server it reports any completed work and receives an XML document describing a collection of relevant parameters. The client then downloads and uploads files and runs the applications, with BOINC maximising concurrency by using multiple CPUs when possible and overlapping communication and computation. The BOINC client periodically contacts a scheduling server, reporting the host's hardware and availability. The scheduling server replies with a set of instructions for downloading workunits, running the applications against the input files, and uploading the resulting output files.

The BOINC middleware is especially well suited for CPU-intensive applications but can be inappropriate for data-intensive tasks due to its centralised nature, which currently requires all data to be served by a set group of centrally maintained servers. BOINC allows a project to configure a fixed and static set of data servers that are maintained and made available for data distribution. Although this scheme enables a number of servers to help load balance the network and scales well for the current applications utilising BOINC, the topology is static and has a number of problems scaling if more data-intensive applications are introduced. For example, under the current system an administrator must dedicate time to configure and maintain these data serving machines, which are generally independent for each BOINC project. Such machines are costly to purchase and maintain, are centrally administered, and therefore cannot be used by other BOINC projects. The real cost, however, lies with the expenditure required to maintain the needed network bandwidth to support a project, especially given the extremely large scale of some public resource computing projects.

An excellent overview of the BOINC client-server topology is given in [102], and an overview of BOINC's goals and design issues is given in [103].

3.5.5 AtticFS

As discussed, large numbers of workers coupled with large datasets can cause a bottleneck for the centralised BOINC data servers, which has a knock-on effect on the performance of the project by limiting the throughput of jobs. The Attic File System [104] is a Peer-to-Peer (P2P) data distribution architecture that provides a means for volunteer computing projects, such as BOINC, to also take advantage of the network and storage resources available on the network in a similar method to BitTorrent[23].

The Attic software is made up of three main elements:

- A client application that allows for the downloading of data from peers/data centres on the network
- A data serving (i.e., caching) application that replicates data on the network
- A metadata lookup service that keeps track of which peers have individual data items

Attic currently supports basic BOINC integration, and is also currently being developed to support XWHEP, to optimise data distribution further.

3.5.6 Distributed Computing Summary

Both grid computing and volunteer computing share the goal of better using existing computing resources in order to dramatically reduce processing time, however the paradigms are rather different - as are the implementations (XWHEP and BOINC) discussed here and utilised in this thesis. P2P computing uses a decentralised 'super peer' topology which enables it to be massively distributed across millions of peers, especially useful for data distribution. The details of how these distributed computing technologies have been integrated and implemented with DART is explained in later chapters.

When considering both BOINC and XtremWeb, it is believed that the requirements of DART can be met as both distribution platforms state they support the complexity of running thousands of jobs in the volunteer areas. Both are developing support for decentralised data distribution using Attic, which can be used in the future to provide the scaling necessary for data intensive experiments. Both platforms also employ a caching

[23]http://www.bittorrent.com/

scheme to minimise data downloads and maximise data reuse, mechanisms for collecting and retrieving results, and job monitoring.

These two infrastructures can now be evaluated and compared using a real-world experiment in pitch detection to see if the systems can meet the distributing computational demands that the DART MIR platform requires.

Chapter 4

Design

4.1 Chapter Overview

This chapter describes the design of a proof of concept distributed MIR platform, and of several experiments that use the DART MIR research platform to perform a parameter sweep experiment to discover the optimal parameter settings for the Sub-Harmonic Summation pitch detection algorithm. The chapter begins by giving a brief overview of the design of DART and discusses the design of the Sub-Harmonic Summation algorithm from its inception in Triana; this consists of building a workflow of Triana units and creating a standard unit design model.

The chapter then covers the steps needed in order to take a Triana MIR workflow and convert it to a streamlined execution environment for DART i.e. the DEE (DART Execution Environment) with no dependancy on the full Triana application. The design of the DART Command Line Interface is covered, as well as the design of several large scale distributed DART experiments, with an explanation given on the relevance of the experiments to the overall goals of the thesis.

The Input Data Design section looks into why certain input files, instruments and sounds were selected and how their inclusion might affect results.

A requirements analysis is then presented, covering the requirements needed at several stages of the development of DART, and finally the design of the results analysis algorithm is given.

4.2 Design Overview

DART is based on an iterative design model, whereby the complexity and scale grows over time and over several experiments.

In order to answer the research questions raised in the thesis hypothesis (Section 1.0.2), this chapter details the design of an MIR research system - DART - that scales as the number of participants increases, adheres to copyright laws, and also enables global access to a database of music for MIR research. A number of experiments have been designed in order to fully realise the goal of testing this hypothesis.

Working towards this goal, a proof of concept DART 'MIR analysis' algorithm is designed in order to replicate and represent some of the traits of a standard real-world MIR experiment. The idea of a large scale *parameter sweep experiment* in order to find the optimal parameters of the Sub Harmonic Summation (SHS) algorithm is presented; parameters such as FFT Window Type, the number of harmonics and top frequency points analysed, and variations in the harmonic content of the input data - all of which are explained later on in the chapter - are examples of some of the variables which can be manipulated.

These experiments are designed not only as a means to test the DART platform and thesis hypothesis, but also as a useful contribution to the field of MIR research - as well as providing a large number of experiments in order to test the scalability of the platform.

The SHS algorithm is distributed and tested on various distributed computing platforms and the feasibility of creating a scalable infrastructure for workflow distribution is investigated, along with different workflow distribution platforms that could be integrated into the system - this thesis investigates the best distribution policy for DART, comparing both the XtremWeb and BOINC distributed platforms. The Music Recommendation System that DART aims to work towards focusses on dealing with data that is on the user/worker machine (thus adhering to copyright laws), however not all MIR analysis algorithms work well when the input data is unknown. To begin with - and to make the DART system more flexible and allow for a wider range of potential MIR experiments, we also look into delivering the data to the worker.

The DART parameter sweep experiments begin on a small scale, iterating in complexity, towards the goal of running hundreds of thousands of experiments on multiple nodes, in order to truly evaluate the scalability of the DART system. The Sub-Harmonic Summation algorithm is the focus of the large scale, distributed DART experiments considered in this thesis, and is discussed in the next section.

4.3 The Sub-Harmonic Summation Algorithm and Triana

The DART SHS algorithm is first designed and created in Triana, which is used a development test bed. DART units are programmed in a modular way, allowing only the relevant building blocks of the workflow to be converted into the standalone DART Java application.

The algorithm performs a series of sub-algorithms on a several audio files[1] that make up the Sub-Harmonic Summation pitch detection algorithm. Ingrained in many MIR algorithms is the necessity to determine pitch, which can be computationally demanding, therefore as a feasibility study of the DART approach the algorithm performs representative (and realistic) calculations by determining the pitch of several segments of audio pieces. This represents a realistic and reusable component for any MIR system since pitch and frequency detection is at the heart of many MIR algorithms.

The complete Triana SHS Task Graph (i.e., the unit-workflow) is outlined in Figure 4.1, before each unit is looked at in more detail. The design of the SHS pitch detection workflow task graph is represented as a chain of the following units:

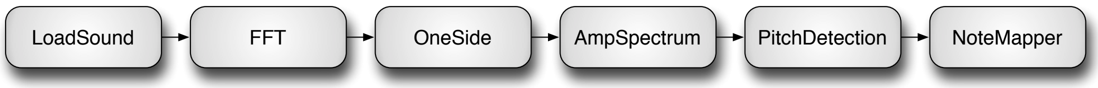


Figure 4.1: SHS pitch detection workflow

Contiguous chunks of audio flow from the LoadSound unit into the Fast Fourier Transform unit where the previously time-amplitude based audio data (as represented in the LoadSound unit) is converted to the frequency domain, as is required by the SHS algorithm. This data passed on to the OneSide unit in order to leave only the positive-frequency part of the spectrum. This is converted into a frequency/amplitude spectrum by the Ampli-

[1]downloaded by the user in the DART prototype

tudeSpectrum (AmpSpectrum in Figure 4.1) unit.

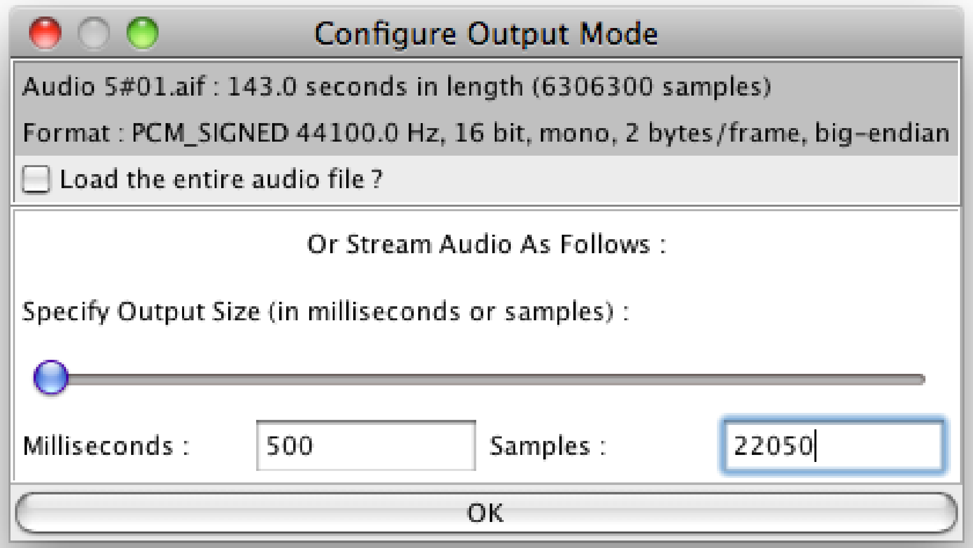
The audio data is now in a suitable format for analysis by the PitchDetection unit. The PitchDetection unit analyses the (monophonic) audio/spectrum data and determines the pitch of the current audio chunk using the SHS algorithm, and the result of the analysis (the predicted pitch of the note) for the current chunk of data (0.5 seconds or 22050 samples long) is returned as an integer value representing the pitch of the audio in Hertz (Hz). This integer value is passed on to the NoteMapper unit in order to map the frequency number value with a scientific pitch notation, where each pitch corresponds to a letter. For example, *440Hz* maps to the *A4* note, or the 'A above middle C'.

Each half a second corresponds to the analysis of one note. This would write the note to a *DARTResults.txt* file, created by the NoteMapper unit. Each input file would create one output file containing the results of the analysis of all the 0.5 second intervals. One run with one set of parameters would create one output file containing the results of the pitch analysis, which can be analysed for accuracy, when compared to the known results.

Triana units can have GUIs for graphical editing of parameters. However, such parameters can be saved along with workflows and executed without the need for external interfaces. Part of the workflow deployment into the DART distributed execution environment transfers these parameters.

4.3.1 LoadSound Unit

The LoadSound unit was discussed briefly in the background chapter of this thesis. Audio files must be loaded into the Triana workflow using a LoadSound unit, which outputs a MultipleAudio data type. MutipleAudio stores channels of sampled data, where each channel can have its own particular audio format e.g. the encoding, such as MU_LAW, PCM and number of bits used to record the data. The LoadSound unit in the DART pitch detection algorithm splits the data into 0.5-second (22050 samples) contiguous chunks for processing, as shown in Figure 4.2; on a 10 second audio file, the LoadSound unit outputs 20 audio chunks to the next unit along the data path. This output size also sets the size of the window for the FFT unit. The only chunk of data that will not usually be 22050 samples long is last chunk, which contains the remainder of the data.


Configure Output Mode

Audio 5#01.aif : 143.0 seconds in length (6306300 samples)

Format : PCM_SIGNED 44100.0 Hz, 16 bit, mono, 2 bytes/frame, big-endian

Load the entire audio file ?

Or Stream Audio As Follows :

Specify Output Size (in milliseconds or samples) :

Milliseconds : 500 Samples : 22050

Figure 4.2: LoadSound GUI, showing the splitting of an audio file into 0.5 second long segments

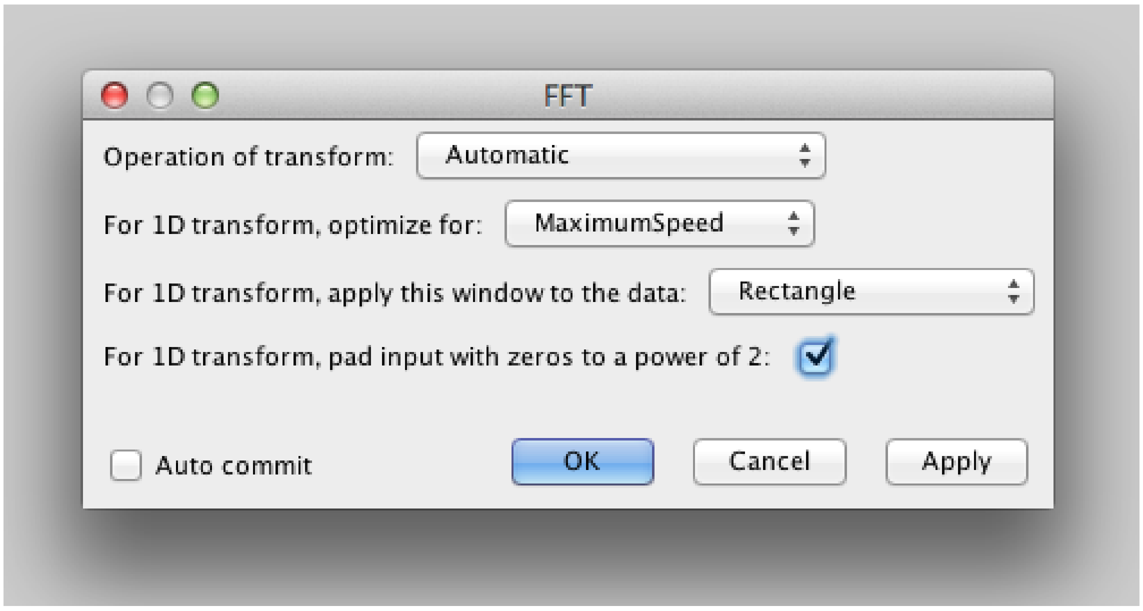
4.3.2 Fast Fourier Transform Unit

The FFT unit will take the MultipleAudio from the LoadSound and perform a Fast Fourier Transform (or its inverse) on one-dimensional or two-dimensional data sets, converting the time-domain based data to the frequency domain[2].

The FFT implemented in Triana can handle input data sets of any length; although it will work most efficiently if the prime factors of the input number are all small. The input can be any (1-Dimensional) VectorType or (2-Dimensional) MatrixType. If the input has signal information, this is used in creating the output (described shortly). The properties of spectral data sets are also discussed shortly.

If the user chooses automatic operation in the user interface window (as shown in the GUI in Figure 4.3), and if the input is a Signal or Spectral data type, then the unit automatically performs the correct type of transform. Therefore, two successive applications of the FFT unit, starting with either a SampleSet or a ComplexSpectrum, will produce a final output identical to the original input, to within round-off error.

[2]For more information on the Fourier Transform please refer to background section


FFT

Operation of transform: Automatic

For 1D transform, optimize for: MaximumSpeed

For 1D transform, apply this window to the data: Rectangle

For 1D transform, pad input with zeros to a power of 2: ✓

Figure 4.3: The GUI for the Fast Fourier Transform unit

The FFT unit can also *pad the incoming data with zeros*, a method that can improve the visualisation or interpretation of FFT results. Zero-padding appends an array of zeros to the end of the input signal before it is converted to the frequency domain in order to increase the number of data points to a power of 2; FFT algorithms are more efficient when dealing with signals that contain 2^n data points. Another reason for zero-padding is for an improved resolution in the frequency spectrum. In general, zero-padding is useful and should be used when using the FFT unit. In practice, the algorithm is many times more efficient (faster) when the input data is padded with zeros.

The FFT unit also contains a list of many different window types that can be applied to the 1D input data[3].

The different FFT Window Types shape the time portion of the to minimise edge effects that result in spectral leakage in the FFT spectrum. By using different window types the spectral resolution of the frequency-domain result could increase.

[3]Including *Rectangle, Bartlett, Blackman, Gaussian, Hamming, Hann(Hanning), Welch, Blackman-Harris92, Nuttall3, Nuttall3a, Nuttall3b, Nuttall4, Kaiser3, Kaiser4, Kaiser5, Kaiser6, Kaiser7, SFT3F, SFT4F, SFT5F, SFT3M, SFT4M, SFT5M, FTNI, FTHP, FTSRS, HFT70, HFT95*

4.3.3 OneSide

The OneSide unit converts two-sided spectra to one-sided spectrum. If the input spectrum is not conjugate-symmetric [4] then information will be lost. The input to this unit must be one-dimensional, returning data that is the positive-frequency part of the spectrum. This unit calls the `convertToOneSided` method in the FlatArray utility class.

This is useful for preparing spectral data for graphing, where the negative frequencies are often redundant. It also reduces the size of the stored data set. If the input data type of the OneSide unit is a Spectrum or ComplexSpectrum that was derived from the Fourier transform of a real data set, then no information is lost on conversion to a one-sided set. The unit called FullSpectrum inverts this and restores the original data set.

4.3.4 AmplitudeSpectrum

The amplitude spectrum of a signal is the absolute magnitude of the input complex numbers provided by the output of the OneSide unit/class. AmplitudeSpectrum outputs a *Spectrum* containing these values. The AmplitudeSpectrum unit transforms an FFT into an amplitude spectrum suitable for graphing or analysis.

The amplitude spectrum is closely related to the power spectrum (also available in Triana). It is possible to compute the single-sided power spectrum by squaring the single-sided rms[5] amplitude spectrum. Conversely, the amplitude spectrum can be computed by taking the square root of the power spectrum. The AmplitudeSpectrum was chosen for DART's implementation of the SHS algorithm.

4.3.5 PitchDetection

The PitchDetection unit performs the main part of the 'Subharmonic Summation' pitch detection algorithm. This unit takes in a ComplexSpectrum as its input and outputs an integer that denotes the overall frequency pitch (in Hertz) for the given audio chunk. As the audio chunks are 0.5 seconds (or 22050 samples) in length, this number represents the

[4]Complex conjugates are a pair of complex numbers with the same real part, but imaginary parts of equal magnitude and opposite signs. For example, 2 + 3i and 2 - 3i are complex conjugates

[5]Root-Mean-Squared average

average pitch detected by the SHS algorithm for the 0.5 second data chunk.

By examining the potential fundamental frequency (those with the highest energy in a power or amplitude spectrum) for each 0.5 second chunk, it is possible to attach scores by summing the peaks at points that lie at integer frequency multiple of those frequencies. This will result in a set of scores for each fundamental frequency and the one with the highest energy will generally be the musical pitch. The SHS algorithm has a number of parameters that can be tuned to optimise performance, such as the number possible fundamentals to examine, and the number of (sub) harmonics to sum. The algorithm design (and subsequent implementation) is based on the classic theories of pitch perception found in [105] and similar.

4.3.6 NoteMapper

The NoteMapper unit takes the number frequency value for the current chunk (given in Hertz and passed on from the PitchDetection unit) and converts it to scientific pitch notation. This method names the notes of the standard Western chromatic scale by combining a letter-name, accidentals (sharp, flat, natural), and a number identifying octave of the pitch.

The definition of scientific pitch notation in this article is that proposed to the Acoustical Society of America in 1939 [105], where C0 is in the region of the lowest possible audible frequency. An example of this in action would be mapping '440' (Hz) to 'A4'.

The NoteMapper unit adds the current note's result to a `String Vector` containing the outcome of all the current audio files results. This Vector - as well as the number frequency value (in Hz) is written to a results output file created by the NoteMapper unit, called 'SHSResults.txt'. The results file accumulates and stores all of the results for the current audio file.

4.3.7 Unit Design

All Triana units extend the Triana abstract class `Unit`, which contains a vast number of methods that units may call on to perform a wide range of functions - such as receiving and outputting objects and data.

In order to promote standardisation when implementing the units, a basic template was created and adhered to, with which all new units could be developed.

In most units designed to be integrated into the Audio processing or MIR toolboxes in Triana, the processing of input data does not take place in the `UnitName` class created by the Triana Unit Wizard, but instead takes place in the `UnitNameEffect` class. The creation of this template allows for easy expansion to the template (as is the case with the `VariableDelay` audio modulation unit, for example) and also aids in the creation of a standard development process. If each unit were implemented with vastly different class associations and interactions, the code structure would become unpredictable and difficult to manage. Future programmers wishing to adapt the code or produce further processing would find the code difficult to understand and navigate; this class template also encourages Java method and code re-use. When the template is not needed or not required, then each unit can consist of a single Java class, containing a `process()` method, where the data can be input, processed, and output.

The class naming structure has also been created to be as intuitive as possible, with the `UnitName` class acting as the main parent class, and the `UnitNameEffect` (or UnitNameRelationship) class created to implement the particular audio processing algorithm. If implementation time is reduced substantially by containing all the code inside of one class, then it may also be the most elegant solution. However, when designing Triana units the aim should not be necessarily to create standalone classes, but instead to produce units that can be used as flexible tools that can be used by the user - and future programmers - to create complex algorithms.

Several design decisions were made early on in the Triana/DART development process in order to maximise productivity when creating Triana/DART algorithms:

- All units are designed to work with 16-Bit audio WAV data only
- Any pre existing Triana units should be utilised (or modified if possible) to aid and simplify development
- The reliance on Trianas built in unit generator to create any graphical user interfaces for new units

These decisions were made in order to allow more focus on the design and implementation of the actual algorithms and digital audio processing methods, rather than trying to cater for more types of data or deliver elaborate user interfaces. It should be noted that existing Triana units, unless specifically designed for graphical applications, do not employ custom GUIs, however Triana does support them.

16-Bit data is the standard bit rate for CD quality sound (when a sampling frequency of 44.1KHz is used) and was used as the standard bit depth for audio-based units in Triana. Accounting for 8, 12 and 24-Bit data would have increased implementation time dramatically, despite their (relative) lack of popularity. 8 and 12-Bit audio data is often not sampled at 44.1KHz, and so usually would not adhere to the Nyquist theorem, making it less less likely to be used by users who wish to take the time and effort to digitally manipulate and analyse audio data. While 24-Bit audio is considered a standard in the recording industry, native 24-Bit audio support is not supported by the Java Sound Audio Engine. While there is nothing in the API that prevents Java applications dealing with 24-Bit/96KHz audio data, the time spent implementing it, in addition to the additional processing power required to deal with the data would place extra strain on memory and CPU resources; 16-Bit/44.1KHz audio emerges as the most practical bit rate to use for these tests.

Once ‘inside’ Triana, audio is processed as 32-Bit floats or doubles in order to account for any dithering or rounding effects which are added when processing the data. This is discussed further in the implementation section.

MP3 support is be available in Triana and was implemented by the author during this thesis, however using uncompressed WAV data not only allows to DART to utilise the Triana framework ‘as is’, but also tests any DART data distribution platform’s ability to provide content to the user, which may be necessary when considering some MIR tasks where the input data must be known. Integrating MP3 compatibility into the DART task graph is considered later on in the thesis.

4.4 Mapping Triana Workflows to the DART Execution Environment

Mapping a Triana workflow to run as a standalone application is necessary in order to use a distribution mechanism that is not heavily reliant on integration with Triana, removing dependancies, and lowers the user systems requirements.

Triana is a Java application and benefits from many of the advantages Java brings, as highlighted in the background chapter. It is a well coded, object oriented application and as such is structured in a way which enables components to be separated and used in isolation without many drastic modifications.

Being a *graphical* Problem Solving Environment, one of Triana's largest dependancies is on its extensive Graphical User Interface (GUI) with which the user communicates. Stripping the GUI away from the underlying code will allow for a much more streamlined operation. The GUI is used to select and connect the Triana units and components, in a particular order, and to run the algorithms. With no GUI, the Triana *Taskgraphs* (workflows or algorithms) therefore need to be finalised and then mapped (from Triana workflow to a streamlined execution environment for DART i.e. the DEE) into a sequence which follows the flow of data that was designed in Triana. This can become an automated feature in the future.

The workflow/taskgraph must be reconstructed and *get* and *set* (pass on) data from one Unit of code to another. Each Unit may have several adjustable variables, which must be set at runtime (from a command line, for example) in order to change the outcome of the algorithm. In the case of DART, it is imperative to be able to adjust these variables with an interface that is easy to use and direct, enabling the SHS parameter-sweep experiment.

Triana can be stripped of all classes and methods that are not required for the execution of the particular DART algorithm or application that is being mapped to work standalone. Only the dependancies of each unit in the workflow, as well as all of the Triana datatypes must be adhered to.

One of the most elegant methods to map over a Triana workflow to a standalone application, would be to create a Java Archive (JAR), allowing for the aggregation of several classes and associated metadata into one executable file. The elements in a JAR file can be compressed, and along with the ability to download an entire application in a single request, makes downloading a JAR file much faster than separately downloading the many uncompressed files which would form a single Java Application. The package `java.util.zip` contains classes that read and write JAR files. Each client-side machine must be capable of running the Java Virtual machine, however there should be no issues running the DART pitch detection algorithm on any Windows, Linux, or Apple Mac computer capable of running Java runtime 1.5 or higher.

4.4.1 Analysis of Triana Application Structure

In order to reduce Triana's footprint by extracting the core support interfaces to be able to execute the workflow, it is useful to briefly consider the current structure of the full-scale Triana application (recently massively streamlined), currently stands at approximately 198MB in size. Triana has seen many revisions, however the current version consists of six version-controlled IntelliJ IDEA programming modules:

- triana-app
- triana-core
- triana-gui
- triana-toolboxes
- triana-types

While the module names are somewhat self explanatory, some discussion on their purpose aids the understanding of which elements must remain in order to maintain the functionality required for the DART platform.

The ***triana-app*** module consists only of the shell and batch execution files to launch the application. The ***triana-core*** module contains many of the core classes that form the full Triana application such as the application configuration classes, discovery mechanisms for Triana toolboxes using Bonjour and HTTP services, and most importantly the enactment and taskgraph packages. The `org.trianacode.enactment` package contains a set of classes which execute a Triana taskgraph; `org.trianacode.taskgraph` contains classes to compose and control this task graph.

triana-core also contains the abstract class *Unit*. The Unit class is extended by all Triana units and contains a vast number of methods they may call on to perform a wide range of functions such as receiving and outputting objects and data.

triana-gui contains all GUI component classes, which are not required when running a DART algorithm. ***triana-toolboxes*** contains all the core Triana toolboxes, and ***triana-types*** is home to all supported Triana data types.

4.4.2 Designing a standalone Triana Workflow framework application

By using only the necessary data types and toolboxes (units), replacing the GUI with a simple command line interface, and predetermining the order of the unit connections that make up a task graph, it is possible to drastically reduce the footprint of running a Triana task graph. The design of the proposed Triana-DART standalone application contains the following java packages.

- cli
- mir
- tdeploy
- types
- util

The ***cli*** package refers to the Apache Commons CLI library[6], which provides an API for parsing command line options passed to programs. This can form the basis of the command line interface used to launch an instance of DART with a specific set of variables. More information is presented on this in the following section.

The ***mir*** package contains all of the MIR-related *Unit* classes that are used in the DART project. All of these classes extend the Unit class, the solitary class contained in the *tdeploy* package. All of the relevant Triana data type classes are contained in the **types** package, and *types.util* contains any utility packages that are required. Many of the data type classes required to process audio and spectral data are interrelated and are included in order to enable the execution of nearly any type of audio algorithm.

General utility classes required by any of the units are contained in the root level ***util*** package.

Using an IDE application such as IDEA IntelliJ[7], it is possible to easily see the dependancies that each module or class has, and integrate them into the DART framework if required.

The abstract class **`Unit`**, one of the core classes in Triana, can be drastically reduced

[6]`http://commons.apache.org/cli/`
[7]`http://www.jetbrains.com/idea/`

in size. For the DART SHS application the only methods required in this class are shown in class diagram 4.4

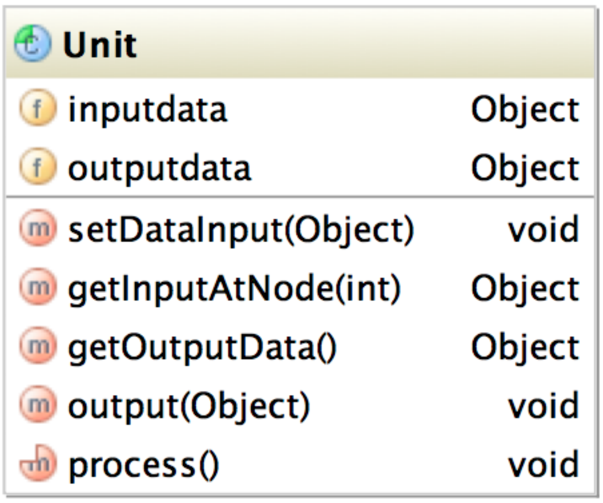


Figure 4.4: A UML class diagram showing the Unit class' fields and methods

The original Unit class contains over 85 methods and nearly 800 lines of code. This can be drastically reduced in size as nearly all are unused. All that pertains to DART is the ability for a DART Unit to be able to accept input from the previous unit, and to output data from to the next.

Following through this streamlining process in order to reduce the size and dependancies allows the overall Triana Package size to be reduced from 195MB (the size of the current Triana application) to a simple DART JAR just 1.9MB in size, a ten-fold reduction in footprint.

4.4.2.1 DART Application Class

A JAR file has an optional manifest file located in the path **`META-INF/MANIFEST.MF`**. The entries in the manifest file determine how one can use the JAR file. JAR files intended to be executed as standalone programs will have one of their classes specified as the main class; in the case of DART, the main class is simply named `Dart.java`. This class will contain the sequential instantiation of all the unit objects, and pass the output of one unit to the input of the next unit in the SHS algorithm sequence. It will also, in tandem with the `NoteMapper.java` class at the end of the DART taskgraph, write to the results file generated by the algorithm. The CLI user interface is also implemented here. The implementation details of all these aspects are explained in the next chapter.

The main class is used to create a JAR file containing the complete DART algorithm, ready for distribution.

4.4.3 User Interface

In order to allow a user to enter the parameters of the DART pitch-detection algorithm, a command line interface (CLI) is proposed in order to allow the parameters to be selected and varied. A simple perl or shell script could then be used to control the execution of the DART parameter sweep experiment by iterating through several different commands, from the command line.

4.4.3.1 Command Line Interface

The DART prototype is based around the distribution of a JAR file and (potentially multiple) audio files that are delivered to the client by distributed computing middleware (XtremWeb or BOINC). An example of the application is designed to be run from the command line by typing:

```
java -jar dart.jar -infile = inputfile.wav -outfile = dartresults.txt
```

The `java -jar dart.jar` is a prerequisite of running any Java JAR file, however the `-infile` and `-outfile` commands here allow the user to customise the input and output file names and locations.

Table 4.1 lists the full range of command line parameters for the proposed DART SHS stand-alone application. The DART pitch detection algorithm is made up from a chain of smaller, modular programming units and a reminder of the workflow is outlined briefly below:

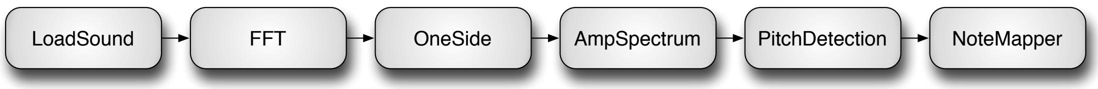


Figure 4.5: SHS pitch detection workflow

Argument Name	*Description*	**Required**
`infile`	Specifies the name & location of the audio input file	Yes
`outfile`	Specifies the name & location of the output text file generated by the DART application	Yes
`repeat_no`	Specifies the number of repetitions of the algorithm, for improved accuracy. Set to 1 by default	No
`audiodir`	Specifies the location of a directory containing audio files for analysis	No
`chunksize_ms`	The audio files are 'chunked into smaller segments and passed from unit to unit. The size of the chunk is set here in milliseconds. The default chunk size is 500ms	No
`chunksize_samples`	Allows the user to specify the chunk size in samples.	No
`fft_transform`	Alters the type of transform performed by the FFT. Automatic, Direct, Direct/Normalised(1/N), Inverse, Inverse/Normalised(1/N). The default is a forward FFT (automatic)	No
`fft_optimise`	The FFT algorithm can be optimised for MaximumSpeed or Minimum-Storage. Default = MaximumSpeed	No
`fft_window`	For a 1D transform, different Windows can be applied to the data, such Hamming, Hanning, Gaussian, etc. Default = Hann(Hanning) window	No
`fft_pad`	A Boolean argument that allows the padding of input arrays (with zeros to a power of two) to be disabled. This greatly (negatively) affects the efficiency of the FFT algorithm	No
`freqpoints`	Specifies the Number of Top Frequency Points looked into by the PitchDetection unit. Default = 30	No
`noharmonics`	Specifies te number of harmonics that are summed up from the fundamental in order to calculate the main frequency of the note. Default = 20	No

Table 4.1: DART Command Line Interface

4.5 DART Experiment Design Choices

While DART hopes to become a fully functional Music Recommendation System and MIR analysis platform, the main aim of the DART mapped prototype algorithm is to use the 'DART platform' (the fusion and complete solution of Triana, DART, and a distribution system such as BOINC or XtremWeb) to conduct a real world scientific experiment. The proposed experiments are designed to conduct an empirical study to find the optimal parameters of the Sub-Harmonic Summation algorithm. The experiments in this thesis will vary the following parameters from Table 4.2:

- **Top Frequency Points:** Vary from 1-50. This argument adjusts the number of top frequency peaks looked into by the PitchDetection algorithm

- **Number of Harmonics analysed:** Vary from 1-32 (5 Octaves). This adjusts the number of harmonics that are summed up from the fundamental frequency
- **FFT Window type:** Vary 1-28. There are 28 different FFT windows available (such as Hamming, Hanning, Gaussian, etc) in the FFT code in the pitch detection algorithm

Varying these parameters and considering all combinations gives a total of 50 x 32 x 28 = ***44,800*** jobs for each piece of input source data. In some initial DART experiments (outlined shortly) a single audio file will be used in order to reduce the number of variables and test the various distribution platforms. Keeping the input data constant ensures that each platform that runs DART will render identical results, and the feasibility of the DART platform as a whole (in terms of performance and practicality) can be investigated. However as the experiment aims to use DART to conduct an empirical study into the optimal number of harmonics, frequency points and FFT window type for the SHS algorithm, different source data must be analysed.

One of the goals of this thesis is to be able to conduct parameter sweep experiments using 6 different audio files, giving 44,800 x 6 = **268,800 total jobs**. These parameters can be iterated through using a shell script in order to generate the 268,800 different command line arguments required. An overview of design of the experiments is given in the following section.

4.5.1 Experiment Design Overview

As documented in the Chapter 3, the initial DART experiments were simulations run in Italy at ICAR-CNR. The previously outlined scenario simulated the performance and behaviour of a distributed P2P network with 1,000 - 20,000 workers with a maximum value of 2,000 super peers. This shows the architecture of the ideal distributed DART system and the simulations test the scalability of the design.

The following subsections outline the design of a series of experiments created in order to test this thesis' hypothesis and work towards a working and viable MIR platform. Each stage of experimentation contributes both knowledge and contributes to the robustness of the DART system, and is designed to evaluate different aspects. All of the experiments listed are conducted after the DART algorithm has been ported to a standalone application, and are meant to test not only the conversion from Triana to the DEE and the distribution of the application, but also the results retrieval mechanisms, before executing a large number of jobs.

The first two experiments described below are not designed to evaluate the SHS algorithm or the results of the analysis, but merely to refine the DART algorithm and workflow. In later experiments, it becomes important to keep the algorithm‘s variables, the number of experiments, and the source data the same, in order to be able to draw conclusions and comparisons between the different distribution methods (BOINC and XtremWeb).

Any problems, errors, or inability to run any of the planned experiments will be documented in the Implementation chapter.

4.5.1.1 DART Experiment 1

The first set of experiments will consist of the execution of a processor intensive DART algorithm on a single audio file *50* times, with *fixed* parameters, on various platforms. This will be run on available local machines (not distributed or processed in parallel), on an XtremWeb desktop grid with 5 nodes, and also using the XtremWeb-EGEE bridge. The purpose of this first, small scale experiment is to test the process of porting a DART algorithm to a standalone application, as well gaining familiarity with the XtremWeb distribution mechanism.

Using the Desktop Grid-EGEE bridge allows for testing of the ‘bridging procedure’ (explained more in the implementation chapter) and also allows for an investigation and comparison into the performance differences (such as overall running time or ‘makespan’) when running DART over the various platforms. The aim of these experiments is to give some indication concerning the speed-up in some realistic scenarios, to justify that the distributed versions provide significant advantages for users in terms of execution time reduction, and to identify limitations of the current implementation and suggest ways of improving the solution.

This experiment will also allow for refinement of the process of porting DART to the standalone JAR application. No results will be retrieved (for analysis) back from the workers during this stage of investigation.

4.5.1.2 DART Experiment 2

The second run of experiments will consist of 160 jobs, using a finalised DART pitch detection algorithm, executed and distributed using XtremWeb featuring a small parameter sweep to test the DART/XtremWeb infrastructure and results retrieval mechanism. The DART parameter sweep is carried out in order to evaluate the DART SHS source code and check that the resulting output produced by the experiment is in the correct and usable format. This will also verify that XtremWeb can handle the submission of multiple experiments, the correct execution of the experiments, and of the results retrieval mechanism. This will also test the script mechanism for generating all 160 DART JAR execution commands correctly.

The experiment will consist of a 2-parameter test run of the following scenario:

- Harmonics will vary only between 1-32
- Frequency candidate points will range from 10 to 50 in 10 point intervals (10, 20, 30, 40, 50) - 5 in total
- FFT Window type will remain constant
- Single audio file will be analysed
- Prototype run from a real parameter sweep application in audio pitch
- Can be run over XtremWeb Desktop Grid in Laboratoire de l'accelrateur linaire, France (LAL)
- 5 x 32 = 160 runs in total

This prototype experiment should be relatively easy to modify in order to scale up to larger experiments as the DART 'infrastructure' will now exist; all that would remain is to modify the script such that it would submit more jobs, and access more worker nodes.

4.5.1.3 DART Experiment 3

The next stage of DART development is to create a full parameter sweep experiment to discover the optimal parameters for the sub-harmonic summation pitch detection algorithm, as a proof-of-concept of the DART system.

This large scale MIR experiment will consist of 268,800 jobs, running the full range parameter sweep and using XtremWeb to distribute the DART application and the input audio files. This will require refinement of the job submission process as it will be by far the largest experiment carried out using XtremWeb to date. For clarity, the experiment will vary the following parameters:

- **Top Frequency Points: Vary 1-50**
- **Number of Harmonics: Vary 1-32 (5 Octaves)**
- **FFT Window: Vary 1-28**
- **6 different input audio files**

The results of this experiment will be retrieved for further analysis by the 'DART Manager'; please see the section entitled 'Results Analysis' for more on the analysis of the results retrieved by this experiment.

All of the experiments listed in the following sections will maintain the same sweep of the variables and use the same input data, in order to aid in the investigation of the most suitable distribution platform and technology. Without keeping all of the variables in the DART SHS algorithm the same, it would be impossible to make any useful evaluation or comparisons.

4.5.1.4 DART Experiment 4

This experiment also consists of executing the same 268,800 jobs, however the EGEE/EDGeS bridge will be used to allow the expansion from the XtremWeb Desktop Grid to the European GRID. This investigates the potential payoff of the availability of the vast EGEE resources[8], versus the potential traffic or 'queue' for resources - EGEE receives up around 330,00 jobs per day from over 14,000 users in approximately 140 Virtual Organisations.

There will be a prototype of this experiment with a run of 672 jobs, to test that the XtremWeb-EGEE bridge is fully working.

[8]The EGEE grid has over 68,000 CPUs available to users 24 hours a day, 7 days a week, and around 20Peta-Bytes of storage space

4.5.1.5 DART Experiment 5

This experiment also consists of the same 268,800 jobs, however the jobs will be distributed using BOINC. This allows for insight to be gained on the relative difficulty of the two distribution methods, and comparisons can be made on overall run times.

4.5.1.6 DART Experiments 6 & 7

These two experiments consist of the same 268,800 jobs, potentially on both XtremWeb and BOINC, however using the ADICS P2P middleware (developed by a third party at Cardiff University) for data distribution. The feasibility of these experiments relies upon the middleware being functional and complete in time for use. A P2P data distribution policy could be an excellent step towards reducing bandwidth limitations and would bring the DART prototype much further in line with the architecture outlined in Chapter 3.

4.6 Input Data Design

In some of the initial DART experiments highlighted previously, a single audio source file will be used during the initial stages of algorithm development and testing. The test file will consist of audio samples from an acoustic guitar playing various notes from the Major scale. This will be kept constant as to ensure that each system creates identical results; it is the functionality and feasibility of the DART platform (in terms of performance and practicality) that is being investigated. As one of the aims of this thesis to use DART to conduct an empirical study into the optimal number of harmonics and frequency points, different, more varied source data must be distributed and analysed.

Each note that is produced by an instrument is a mixture of many different pitches (harmonics) that blend together such that we do not hear them as separate notes. Instead, the harmonics give the note its colour or timbre. The difference is due to the difference in the relative loudness of all the different harmonics. Therefore, it is vital to test the accuracy of the DART SHS (pitch detection) algorithm across a range of input files.

The parameter sweep experiment will be conducted on six different audio files, with each file containing notes played by a particular (different) instrument. Each instrument will play all the chromatics notes possible within the playable range of the instrument - and at several amplitude/velocity levels[9]. This will allow for the analysis of the results produced by the algorithm with any given set of parameters, and an accuracy check. The following instruments/audio files will be used as input: *Acoustic Guitar*, *Distorted Guitar*, *Oboe*, *Grand Piano*, *Violin* and *Tubular Bells.*

These instruments were chosen to vary the audio being analysed by the SHS algorithm and examine a wider breadth of input data; some of these instruments produce a large number of harmonics or overtones and some are more pure in their composite sound. A composite sound is the result of the superimposing of several waves, the fundamental note that has the same frequency as the vibration of the string and determines the pitch of the sound perceived and the harmonic.

Finding a set of variables that work well for all of the instruments and notes generated will reveal the optimum settings for the algorithm. Below is a brief overview of each instrument and a description of the type of pitch range and level of inharmonicity[10] that can be expected when the instrument is played.

4.6.1 Acoustic Guitar

When an acoustic guitar string is plucked the air inside the body cavity resonates with the vibrational modes of the string and at low frequencies the acoustic chamber behaves like a Helmholtz resonator [106], increasing or decreasing the volume of the sound depending on whether the air in the chamber is moving in phase or out of phase with the strings. The sound of an acoustic guitar contains a complex mixture of harmonics that give the guitar a distinctive sound, and the string type - steel or nylon strings - each create their own timbre and frequency response. Both string types will be tested in the DART experiments.

The tension and the length of the string determine the fundamental frequency at which the string vibrates. By changing the length of a string (by placing a finger on the string, in between the frets) the player can modify the vibrating frequency, and therefore the perceived note. If the finger is placed at a whole fraction of the length of the string, the

[9]Not only will the overall amplitude of the note change, but often when playing 'harder', more or less harmonics can be generated. Varying the level of intensity ensures more rigorous testing

[10]*Inharmonicity* is the degree to which the frequencies of overtones depart from whole multiples of the fundamental frequency

vibration produces a note in harmony with the fundamental note[11].

Acoustic guitars often have around 20 frets (and is therefore an ‘equal tempered’ instrument), with notes in standard tuning (EADGBE) ranging from around E2 to C5. However the guitar is a transposing instrument; its actual pitches sound one octave lower than notated.

4.6.2 Distorted Guitar

An electric guitar uses electromagnets to convert the vibrations of its metal strings into audio signals. The signal generated by an electric guitar is not strong enough to directly drive a loudspeaker and is therefore amplified before sending it to a loudspeaker. Since the output of an electric guitar is an electric signal, the signal can be altered using electronic circuits to add colour to the sound. Often the signal is modified using effects such as chorus and distortion.

Distortion is created by compressing or ‘clipping’ the peaks of the waveform, originally discovered and exploited by guitarists when overdriving their guitar amplifiers, overloading the preamp stage, power output stage, the speakers, or a combination of all three. The waveform peaks can be manipulated by overdriving the solid state or valve amplifier circuits, creating even-(valve) or odd-(solid state) order harmonics, or by using some form of signal processing unit, such as a guitar stomp-box pedal or studio effects unit. Some amplifiers even utilise hybrid designs that employ both valve and solid-state components, adding both types of harmonics into the signal. Analysing the pitch of a heavily distorted guitar tone with a large number of harmonics would be a good challenge for any pitch detection algorithm.

Electric guitars usually have 22 frets, and are able to produce notes from E2 to D5 when in ‘standard tuning’. The effect of the distortion on the tone of the guitar will be an interesting challenge to the DART SHS algorithm.

[11]This principle was discovered by Pythagoras 2000 years ago.

4.6.3 Oboe

The oboe is a woodwind instrument that features a *conical bore*[12] giving the instrument the ability to produce a slightly piercing, relatively clear sound. The instrument also features a *double reed*[13], which not only creates many overtones, but also gives a different distribution of overtones for different notes. An oboe will approximate a sawtooth wave as the odds and evens are both relatively strong, however the fundamental frequency is often not as high in amplitude as the harmonics and overtones, making the fundamental harder to isolate. The second harmonic octave will have more energy than the first, making the overall sound like the fundamental is one octave higher. These properties make the oboe and extremely interesting test of the subharmonic summation algorithm.

Oboes typically vary in range depending on the type, such as Classical, Baroque, and Conservatoire. Generally however, an oboe has a 'written' range of Bb3 - A6.

4.6.4 Grand Piano

The grand piano uses a felt-covered hammer to strike steel strings when the player depresses a key. The hammers rebound allowing the strings to continue vibrating, transmitting through a bridge to a sounding board that couples the acoustic energy to the air more efficiently.

The different keys have different pitches, depending on a key's associated string length (shorter strings produce higher pitches), string thickness (thinner string is of higher pitch), and tension; as with all stringed instruments the higher the tension of the string, the higher the pitch. Pianos are designed to produce nearly periodic oscillations, creating overtones as close as possible to the harmonics of the fundamental tone and minimising inharmonicity.

One of the worlds most popular instruments, modern pianos have a total of 88 keys giving the instrument a large range of seven octaves plus a 'minor third', from A0 to C8. Some Bsendorfer[14] pianos can even extend the normal range down to F0, with one other

[12]The bore of a woodwind instrument is the interior chamber that defines a flow path through which air travels and is set into vibration to produce sounds. The shape of the bore has a strong influence on the instruments‘ timbre.

[13]A reed is a thin strip of material which vibrates to produce a sound on a musical instrument. The oboe has a double reed because there are two pieces of cane vibrating against each other

[14]`http://www.boesendorfer.com/`

model going as far as a bottom C0, giving a full eight octave range.

4.6.5 Violin

The violin is a wooden stringed instrument with four strings tuned in perfect fifths[15], and like the acoustic guitar has a hollow chamber which helps to amplify the resonant sound of the strings.

Sound is generated when the musician rubs the string with a bow, whereby the string undergoes a transversal and a longitudinal vibration by being tightened and then released. It also undergoes a torsion according to the force behind the rubbing, which modifies the timbre of the instrument. The sound produced often depends on the form and construction of the resonant box/body. Some players also use their fingers to pluck the strings (called *Pizzicato*) creating a sharp, short and more percussive sound[16].

The violin is capable of generating a great variety of sounds. It can produce very sharp sounds when the length of a string is reduced. Moreover as it is up to the violinist to create the sound (the violin fingerboard is fretless) - rather than to select a predefined sound as with the piano - the range between notes is unlimited much like an upright bass or fretless bass guitar. The charm of the instrument can be said to lie therein - but so does the difficulty in playing the correct notes.

The violin has a natural note range of G3 - A7. The violin can extend to notes as high as C8 by using natural or artificial harmonics (lightly touching the string with a fingertip at a harmonic node creates harmonics), therefore the E two octaves above the open E-string may be considered a practical limit for orchestral violin parts.

4.6.6 Tubular Bells

Tubular bells (or chimes) are metal, tubular percussion instruments that produce a distinctive tone when struck with a hammer or mallet. They can also be bowed at the bottom of the tube to produce a very loud, very high-pitched overtone.

The sound produced by tubular bells consists of two components: the strike note and

[15]A perfect fifth is a note interval spanning seven semitones

[16]Pizzicato notes are not analysed in the DART experiments.

the resonance. The initial note caused by the strike is short, forceful, and gives the impression of a single pitch one octave above the fundamental. The resonance however, is long and rich in overtones, with more prominence given to the pitch of the fundamental. The fundamental sounds an octave lower and is audible in the resonance along with many other higher notes. The pitches written in a score refer to the strike note and not the fundamental.

The pitch range of tubular bells can vary, depending on the number of tubes. 18 tubes produce a range of 1.5 octaves (C4 - F5), 25 tubes can span 2 octaves (F3 - F5), and a full ‘philharmonic’ consists of 29 tubes, spanning a range of 2 1/2 octaves, from Eb3 to G5.

4.6.7 Input Data Design Summary

Given six input files, each with a total of 44,800 jobs per audio file, each of the main DART experiments total 268,800 jobs.

The audio files will all be created with a tempo of 120BPM and a time signature of 4/4. This corresponds to two notes per second, as each note will play for 0.5 seconds, with no overlapping of notes (monophonic). The SHS algorithm will generate one note ‘result’ per 0.5 seconds.

In the future, DART can avoid copyright issues by analysing MP3 data on the worker machine, or by providing non-copyrighted audio to be processed (unless given consent), as is the case with the highlighted DART input files. More information on the implementation and creation of these input files is given in the Implementation chapter.

4.7 Requirements Analysis

The distributed nature of the DART experiments could place a large amount of strain on a platform, unless it is designed to be scalable. Maximising the scalability of a system necessitates a requirements for the system and therefore it is useful to briefly consider the requirements placed on the DART researchers, performance, data, and also the security requirements of the target computer platforms (workers and servers). At this stage some the requirements can be considered a ‘work in progress’; a large role that the several stages of the DART experiments will play, is the clarification of many of the requirements discussed in the following subsections. However, some consideration of the points before implementation is still worthwhile.

The execution time of the SHS experiments is largely dependant on a few simple factors: size of the audio file, the number of audio files to analyse, the cpu speed of the worker, and - perhaps - the parameters of the SHS algorithm (FFT Window, number of harmonics, and so on). Once the results of the large scale parameter sweep experiments are returned, the impact of changing each parameter on the execution time can be examined and analysed. Extending the parameter ranges may lead to exponentially growing execution times, or could have little effect on the overall processing time.

The long term aim of the adaptation and creation of DART is to study the feasibility of the analysis of massive MP3 collections with constantly refined/CPU intensive MIR algorithms. Given this it is necessary that we run the prototype DART application on a variety of different systems and platforms, with various levels of computational power in order to gauge the feasibility of the application and algorithms - what is possible and feasible, and what is not given the 'average' workers' CPU resources available to use on that platform.

DART does not require too much from the target computer platforms; the DART application is written in Java and as such each client-side machine must be running a Java Virtual machine - version 1.5 or later. There should be no issues running the DART application on any Windows (XP or Vista) machine, Linux, or OSX (10.2 or later) with the JVM installed. There are currently two cases for audio analysis - analysis of audio that is downloaded by the application (as will be implemented within the scope of this thesis) and the analysis of audio that already exists on the target-computing platform. In the case of the former scenario, no further requirements exist as the data is staged to the client by the DART application or desktop grid middleware (e.g., BOINC/XtremWeb).

The latter scenario requires that the application run on a system that has a collection of audio files that the user is willing to allow the analysis of, and the location of the top level directory that contains the MP3/audio files must be specified in the command line argument - or with a GUI once past the prototype stage.

As DART is currently a prototype of an MIR research tool, it is the responsibility of the DART algorithm developers that no copyrighted material is distributed to workers. For local files, there may be restrictions on some MP3 files that are protected by Digital Rights Management schemes (such as the songs sold in earlier iterations of Apple's iTunes store), and cannot be analysed. The BOINC website has a section describing common security issues in volunteer computing and the mechanisms used to reduce the likelihood of some of these attacks[17]. However, no confidential information is passed around the network in any of the proposed DART experiments, and so security is not considered a top priority for DART. Generally, the XtremWeb/BOINC provided security features are sufficient for the prototype experiments presented in this thesis. The XtremWeb to EGEE bridge itself is has a very tight security mechanism; only users with validated and trusted certificates may use the European Grid. This is discussed in more detail in the Implementation chapter.

The DART application must be robust; the application will be well tested before running on any distributed machines. Administrators of the XtremWeb and BOINC servers regularly maintain them a regular basis for maintenance purposes, and would be able to notice any issues caused by running the DART experiments.

4.7.0.1 DART's Data Requirements

The DART application requires one or more audio files (16-Bit/44.1KHz Wav or Aif) as input, for analysis. The worker machines must have enough free space to download all the audio input files (6 x 55MB = 330MB uncompressed) and a small JAR file ($<$ *1MB*). All the data in the DART experiments presented in this thesis is delivered to the user and does not contain any copyrighted material circumventing any legal issues caused by copyright. However even in the long term, the DART platform will be able to avoid legal issues as no copyrighted MP3 data will be passed back and forth between the workers or the DART server. DART will also not be particularly data intensive, in the traditional sense. The prototype however will require the distribution of WAV files to ensure that there is data to analyse on the worker node and that it is in the correct location, as well as to test the data distribution capabilities of the BOINC and XtremWeb middlewares.

Each audio file that will be distributed for analysis will be around 56MB[18] (5 minutes

[17]`http://boinc.berkeley.edu/trac/wiki/SecurityIssues`

[18]PCM audio with a sampling rate of 44.1KHz and a bit depth of 16-Bit generally has size of 10MB per minute of audio (uncompressed). This means that a 56MB file would be able to cover over 5 minutes 30 seconds - longer than an 'average song. A 'larger than average' audio file also helps to test the distribution

30 seconds of audio at 44KHz/16-Bit), although this can be compressed (.zip) to 35.8MB. Once the DART prototype has matured, potentially a small DART JAR executable will be distributed alone. The output text file will be negligible in size (under 6KB), and so would place very little strain on the distribution mechanism, or the users bandwidth. Furthermore, once the DART application supports MP3 decoding, a smaller compressed audio file (typically 1/10 of the size of the WAV/AIF) could be distributed around the network, if the desired DART application permits this loss in fidelity.

The following list summarises the minimum set of files that are required to allow the application to be distributed:

- **DART JAR executable:** Currently 856KB
- **DART input audio data file:** as discussed, this file is distributed in order to guarantee that there is suitable audio to analyse on the target machines, and also to establish an even test-bed for performance benchmarks and results analysis. Currently 56.6MB per file (uncompressed) / 35.8MB (zipped)
- **Result files:** text files produced by the DART, containing the results of the pitch detection analysis and basic benchmarking (time taken to complete the analysis and machine spec). The worker application sends this file back to the DART manager for inspection/further analysis. Currently 6KB

4.8 Distribution of Workflows

Once the standalone DART application is created, a shell or perl script can be written to generate all of the 268,800 command line arguments, covering all of the different variations required. Running this script would enable a single computer to sequentially run all the experiments, however it can be adapted to work with the XtremWeb and BOINC platforms to distribute these jobs over many machines. Assistance in this will be provided by the XtremWeb-HEP middleware team[19] and by scientists familiar with the creation of BOINC applications[20].

platform's ability to deal with experiments that require data to be distributed alongside the DART binary.

[19]`http://www.xtremweb-hep.org/spip.php?rubrique35`

[20]`http://www.cs.cf.ac.uk/contactsandpeople/staffpage.php?emailname=K.Evans`

4.8.1 XtremWeb

DART experiments will run on an XtremWeb (XWHEP) Desktop Grid and can be managed remotely by the 'DART Manager'. The application is registered on the XWHEP server and both the application and the input data can be stored in the XWHEP infrastructure or in any repository, as long as the data can be described by an URI and are accessible through the network. It is possible to follow the XWHEP guidelines, revealing the commands to submit the DART application - and all of the data - to the XWHEP server. URIs are returned for each of the data files inserted into the XWHEP system and the URI can be added to the command line argument script, enabling XtremWeb to find the correct input data file. An XWHEP client prepares jobs containing the reference of a registered application, optional parameters, and optional references to additional files.

This would change the CLI input commands as the input data file would be replaced with a URI. As an example, a DART CLI input command for a single job would change from:

```
java -jar Dart.jar -infile DARTAcousticG.wav -outfile DART-1-1-1-1.txt -
    nofreqpoints 1 -noharmonics 1 -fft_window Rectangle
```

to:

```
--xwsendwork DART -infile DARTAcousticG.wav -outfile DART-1-1-1-1.txt -
    nofreqpoints 1 -noharmonics 1 -fft_window Rectangle --xwenv xw://xwserv.
    lal.in2p3.fr/f10515ad-4d3c-4822-b61f-e77002cf6621
```

Where `xw://xwserv.lal.in2p3.fr/f10515ad-4d3c-4822-b61f-e77002cf6621` is the URI of the DARTAcousticG.wav audio file (input data). This job execution command is submitted to the XWHEP Scheduler. The Workers contact a Scheduler to get jobs suitable for their architecture; as all of the XWHEP systems run Java, the JAR file will be used. In response, the XWHEP Scheduler will send a suitable job description to a Worker. For each file referenced by the job which is not present in the local cache of the Worker yet, the Worker fetches the file from the XWHEP Data Repository or from an External Data Server. As soon as a job has finished on the Worker side, the Worker contacts the Result Collector to send the results.

DART requires minimal modification to work with XtremWeb, but will be by far the largest project run on the platform. Any unforeseen issues and more information on the details of the XtremWeb implementation process will be presented in the next chapter 'Implementation'.

4.8.2 BOINC

Distributing an application using BOINC is somewhat more involved than XtremWeb, and will require expert assistance. BOINC provides features that can simplify the creation and operation of distributed computing projects, however is not well suited to running Java projects. C and C++ projects can be run with little to no modification. BOINC will require several steps in order to run DART, including:

- BOINC Server Setup
- Creation of different platform applications (Windows X64, Windows X 86, Linux 32/64-Bit, Mac OSX applications; conversion from JAR)
- Work/job script creation
- Validating work
- Monitoring progress
- Retrieving Results

The BOINC team provide guides to aid the different aspects of creating a project, such as the BOINC project creation 'Cook Book' [107], the BOINC Server Intro [108] and a large master PDF document simply entitled 'Creating BOINC Projects' [109]. These documents contain a wealth of information required for the implementation of a generic BOINC project. These guides will be used to set up a BOINC server and cross platform versions of the DART application will be created created before requiring assistance (from a researcher based in Cardiff University) to fully configure BOINC and modify the DART perl script/workflow used to generate the 268,800 command line arguments, and generate the required BOINC work units.

When building a BOINC application, several versions for differing platforms are required (Windows, Mac OS X, Linux) in order to run on the widest range of volunteer computers possible. Each operating system also requires individual 32- and 64-Bit versions of the application in order to fully maximise the potential for user participation.

The multiple platform versions of the DART application will be made with the aid of software such as GenWrapper[21] or JSmooth[22]. BOINC does include some guides to run a Java application with BOINC and includes a wrapper[23] as found in [110], however this wrapper is not very flexible. It is partially configurable but can only be used to execute a list of executables (tasks) one after the other as the XML file it uses only allows describing the order of execution of the binaries. To make the wrapper more flexible this configuration file could be extended with new features to provide a required level of flexibility each time a shortcoming is discovered, but ultimately a general solution would require a generic scripting language for describing all possible configuration options.

JSmooth is a Java Executable Wrapper that creates native Windows launchers (standard .exe) for Java applications. It makes Java deployment simpler as it is able to find any installed Java VM automatically. When no VM is available, the wrapper can automatically download and install a suitable JVM, or simply display a message or redirect the user to the DART web site.

GenWrapper is a generic BOINC wrapper for legacy applications, using shell scripting and built-in commands like tar, awk, sed, zip, etc. to control and execute a 'legacy' Desktop Grid application, stating how the application is to be run and how the BOINC workunits should be processed. GenWrapper offers a solution for wrapping and executing an arbitrary set of legacy applications in a BOINC infrastructure. GenWrapper is available for Mac OS X, MS Windows, GNU/ Linux and is extremely lightweight (around 600Kb). [111] gives an excellent overview of the software.

The SHS experiments run on BOINC will be identical to the previous experiments on the XtremWeb platform.

4.9 Results Analysis

After experiments have been run the results returned from the several runs of 268,800 jobs need to be analysed in order to ascertain the best set of parameters for the SHS algorithm, taking into account run time and accuracy across the board, for all input file types (different

[21]`http://genwrapper.sourceforge.net/`

[22]`http://jsmooth.sourceforge.net/`

[23]`http://boinc.berkeley.edu/trac/wiki/WrapperApp`

instruments). The results returned from the main BOINC and the (multiple) XtremWeb experiments should be identical, allowing only for CPU error and lost results. Between the several experiments, all of the 268,800 results should be available, and can be compared with each other using a series of post-processing applications that can go through each of the 268,800 result (text) files and find the optimum parameters for the implemented SHS algorithm.

When the jobs are run on the distributed machines, each worker processes each source audio file once and generates a text file that is to be returned back to the central DART Manager. As well as the results of the SHS analysis in both integer note values (such as '440 Hz) and pitch note values (such as 'A4), each results file contains metadata listing:

- FFT Window Type
- Number Of Frequency Points
- Number Of Harmonics
- Instrument analysed (input file name)
- Run Time

However, the filename of the results files will also differ depending on the set variables. The design of the file naming scheme reveals the exact parameters of each of the 4 varied parameters. An example results file name could be: `DART-2-3-5-1.txt`.

The first number can vary from 1-6, and represents the input file type:

- 1 = DARTAcousticG.wav
- 2 = DARTOboe.wav
- 3 = DARTViolin.wav
- 4 = DARTPiano.wav
- 5 = DARTTubBells.wav
- 6 = DARTDistortG.wav

The second number represents the number of frequency points, from 1-50, and the third digit represents the number of harmonics to analyse, scaling from 1-32. The fourth number represents the window type, from 1-27, iterating through the following window types respectively:

Rectangle, Bartlett, Blackman, Gaussian, Hamming, HannHanning, Welch, BlackmanHarris92, Nuttall3, Nuttall3a, Nuttall3b, Nuttall4, Kaiser3, Kaiser4, Kaiser5, Kaiser6, Kaiser7, SFT3F, SFT4F, SFT5F, SFT3M, SFT4M, SFT5M, FTNI, FTHP, FTSRS, HFT70, HFT95

This will not only make it easier to ascertain which results may be missing, but also allow for the post processing algorithm (which will search to find the optimum parameters) to simply use the file name to categorise the results. In the example filename of `DART-2-3-5-1.txt`, we can see that this would be the results file Oboe with 3 frequency points and 5 harmonics, using the Rectangle FFT window. The overall effect of these variables on the accuracy of the analysis can then be analysed more easily.

The post processing algorithm will categorise each type of results by instrument and find the set(s) of variables that produce the least number of errors for each. The algorithm will take into account not only errors, but will also check against the known, correct results. For example, for the Number Of Harmonics variable, the effect of varying the number from 1-32 across *all* audio input files can be mapped, however the effect can also be measured *per instrument*.

A result which contains the correct note but with the wrong octave, will of course be considered more accurate than a note which has no bearing on the correct result. The correct, expected 'control' set of notes for each instrument will be written as a String array in the post processing class.

This comparison will allow for the optimal combination of variables for accuracy, per instrument. A CSV file will be created for each instrument, where each result file analysed will achieve an overall score for accuracy, both with and without octave errors. This will be done for each instrument, as well as across the entire range of results.

4.9.1 Comparison of Distribution Platforms

Once the final experiments have been run on the various platforms the overall time taken to process the 268,800 jobs can be evaluated, also taking into account the difficulty of setting up DART to work the various XtremWeb and BOINC platforms. This will be discussed in the Implementation and Results chapters.

4.10 Summary

This chapter began by giving an overview of the overall design of the DART MIR platform, and of the several experiments that use DART to perform a parameter sweep experiment to discover the optimal parameter settings for the SHS pitch detection algorithm. The chapter discusses the design of the DART Sub-Harmonic Summation algorithm, beginning with the creation of a Triana Task graph, and creating a standard unit design model.

The chapter then covers the steps needed in order to take a Triana workflow and convert it to run in the DART Execution Environement; a standalone application with no dependancy on the Triana application. The design of the DART Command Line Interface is covered, as well as the design of several important, large scale distributed DART experiments with an explanation given on the relevance of each iteration. The Input Data Design section looks into why certain input files and sounds were selected and how their inclusion might affect results, and a brief requirements analysis is presented, covering the requirements needed at several stages of the development of DART. The distribution mechanisms are discussed before finally, the design of the results analysis algorithm is given.

Chapter 5

Implementation

5.1 Implementation Overview

This section details the implementation of all the aspects of the DART system, from the creation of Triana units, to the execution and analysis of the several experiments outlined in the previous chapters.

This section begins with the implementation of the Sub-Harmonic Summation algorithm within Triana is described, including the inner workings of each unit, outlining the changes and refactoring of the pre-existing units. The PitchDetection and NoteMapper units that make up the Sub-Harmonic Summation algorithm are described in further detail. The process of creating the Dart Execution Environment and enabling the Triana DART workflow to function as a standalone JAR application is covered briefly (a full explanation of the method used to port all of the Triana Units to the Dart Execution Environment is given in Appendix A.3), revealing the structure and overall workflow of the process. The creation of the six audio input files is then discussed, explaining the methodology and reasoning behind the choice of each instrument. The DART integration and deployment with XtremWeb is then shown fully, including the installation and running of single DART jobs on a Desktop Grid, as well the creation of a script to generate the 268,800 job commands. The implementation of the Java application written to analyse the vast results from the XtremWeb experiments is shown before finally giving an overview of the DART BOINC implementation.

Further implementation details and code listings are given in the Appendix at the end of the thesis. An overview of Triana units and their basic coding structure is given in Appendix A.1. The implementation and code for the application written to find the optimal SHS parameters (post experiment analysis) is listed in Appendix A.2.

5.2 Triana Units

The creation of a Triana *Task Graph*[1] is the beginning of any DART experiment. Triana can be used to prototype algorithms and ideas, with an easy-to-use graphical interface to connect units together.

As highlighted in the design chapter, the process of creating a DART application and experiment can be described in three stages:

- Algorithm/application design in Triana - *what does the MIR analysis do?*
- Porting a Triana workflow to a standalone application with minimal dependencies
- Application distribution and experiment design - *what experiments am i trying to run - what am i trying to analyse?*

The implementation of these three stages is documented throughout this chapter. Before a Triana task graph can be created, the units that make up the required algorithm must first exist. Triana contains hundreds of pre-existing units that can be used to create new algorithms - and all number and combination of these units can be grouped together to create a *grouped unit* with new functionality that did not exist previously. However, it is also possible to create new units, as was required with DART.

5.2.1 Implementing the SHS Algorithm in Triana

The Triana Sub-Harmonic Summation pitch detection task graph is shown in Figure 5.1.

The data-flow through the algorithm is as follows: For every 0.5 seconds or 22050 samples of the input audio file (as each input file is long, this creates 672 continuous chunks of

[1]As explained in earlier chapters, a Task Graph is simply a workflow or algorithm created in Triana using the graphical user interface

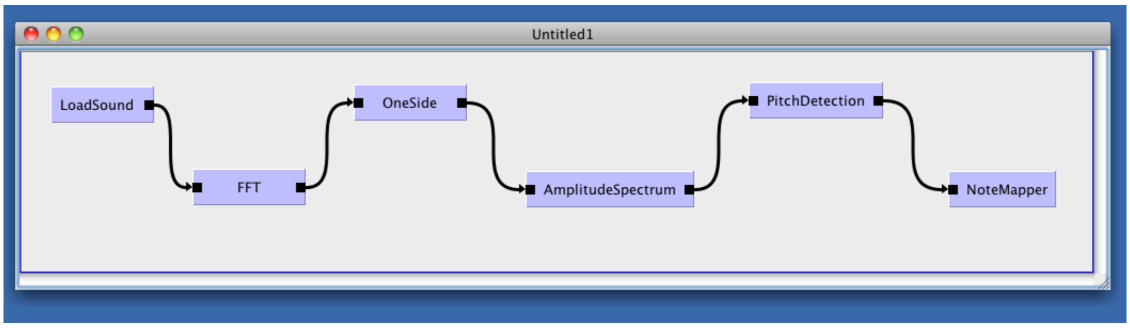


Figure 5.1: A Triana workflow showing the DART application workflow

audio per audio file), the LoadSound unit outputs a **`MultipleAudio`** data type to the FFT unit where the data is converted to a **`SampleSet`** on input and a **`ComplexSpectrum`** is output to the OneSide unit. The OneSide unit both inputs and outputs a **`ComplexSpectrum`** and the AmplitudeSpectrum unit converts the input **`ComplexSpectrum`** to a **`Spectrum`** datatype when output. The PitchDetection unit merely outputs a single **`Integer`** representing the frequency pitch value of the 0.5 second chunk that is currently being analysed. This integer is passed to the NoteMapper unit, which then maps the integer value to a note value, writing both to a results file containing the results of all 672 chunks per audio input file.

Most Triana units used a deprecated class called **`OldUnit`**.

An overview of the refractoring process is given below, after which the implementation of each unit can then be more thoroughly discussed.

5.2.2 Refactoring Revelvant Triana Units

All of the pre-existing Triana units utilised in DART required modification; namely conversion from using the deprecated Super Class **`OldUnit`** to the newer **`Unit`** class. **`OldUnit`** is a legacy class required to provided backward compatibility to old tools, and all the methods in the class are deprecated. **`OldUnit`** was heavily integrated with the GUI interface, making life very difficult for programmers wishing to create code where the processing is decoupled from the Triana-specific or Unit-specific implementation.

While not imminently vital to the implementation of the DART SHS algorithm, it was important to update all of the Triana units to work with the newer Unit class, allowing much further decoupling from the GUI, which will be required when porting the DART task-graph to work standalone. When dealing with a unit that has no GUI (e.g. the OneSide unit), converting the unit to extend the newer Unit class is relatively trivial.

Having units that depends on both **`Unit`** and **`OldUnit`** would also make the DART JAR file larger and incur unnecessary bloat. Furthermore, it will allow future DART/MIR programmers to prototype and port their designs without the trouble of supporting and using deprecated classes.

Units with a GUI built by the Triana GUI builder (usually created by using the Triana Unit Wizard) can be relatively straight forward to convert too; an excellent example is the FFT Unit. Older units such as the FFT unit have an init method to initialise a new Unit, and also a **`setGUIInformation()`** method, where the GUI's parameters are set:

```
public void init(){
  super.init();

  setUseGUIBuilder(true);
  setResizableInputs(false);
  setResizableOutputs(true);
          setRequireDoubleInputs(true);
          setCanProcessDoubleArrays(true);
}

public void setGUIInformation() {
  addGUILine("Operation of transform: $title style Choice Automatic Direct
      Direct/normalized(1/N) Inverse Inverse/normalized(1/N)");
          addGUILine("For 1D transform, optimize for: $title opt Choice
              MaximumSpeed MinimumStorage");
          addGUILine("For 1D transform, apply this window to the data: $title
               WindowFunction Choice " + SigAnalWindows.listOfWindows());
          addGUILine("For 1D transform, pad input with zeros to a power of 2:
               $title padding Checkbox false");
}
```

This can be simplified to work with the newer Triana GUI builder - in the new **`Unit`** implementation these two methods are combined into the **`init()`** method of the unit. The **`OldUnit`** method **`addGUILine(String)`** simply calls the **`Unit`** method to add a GUI line:

```
public void init() {
  super.init();
```

```
    setMinimumInputNodes(1);
    setMaximumInputNodes(1);
    setDefaultInputNodes(1);
    setMinimumOutputNodes(1);
    setDefaultOutputNodes(1);

    String guilines = "";
    guilines += "Operation of transform: $title style Choice Automatic Direct
        Direct/normalized(1/N) Inverse Inverse/normalized(1/N)\n";
    guilines += "For 1D transform, optimize for: $title opt Choice
        MaximumSpeed MinimumStorage\n";
    guilines += "For 1D transform, apply this window to the data: $title
        WindowFunction Choice " + SignalWindows.listOfWindows() + "\n";
    guilines += "For 1D transform, pad input with zeros to a power of 2:
        $title padding Checkbox false\n";
    System.out.println("guilines = " + guilines);
    setGUIBuilderV2Info(guilines);
  }
```

The Triana GUI builder can take the **`String`** and construct the GUI based on the parameters read from it. Conveniently, the String used in the **`OldUnit`** class and the newer **`Unit`** class are nearly always the same.

The newer **`Unit`** class also requires the use of the **`parameterUpdate(String name, String value)`** method instead of **`setParameter(String name, Object value)`**, and as can be seen above, gives the developer access to intuitive methods such as **`setMinimumInputNodes(int)`** and **`setMaximumInputNodes(int)`**.

The above overview gives a brief explanation of some of the changes that are required when converting the units, however each unit can possess its own idiosyncrasies and dependancies. *All* Triana units were converted before or during the implementation of the DART SHS units. Each of these units can now be more thoroughly examined and discussed.

5.2.3 LoadSound

The LoadSound unit was one of the only audio units available in Triana before the audio toolkit was created and has been updated by the author several times (to implement the chunking system that enables the processing of large audio files, as used in DART).

Before implementation of the SHS algorithm could take place the LoadSound unit required updating[2]. The LoadSound unit was using the deprecated version of the Unit superclass, **`OldUnit`**.

During the refactor of all the Triana units, LoadSound was one of the most complex to convert to using the **Unit** superclass.

LoadSound contained a method called **`doubleClick()`** and required extensive reworking in order to function as a 'new unit'; the LoadSound unit contained two custom panels which provide input data to set the variables (namely audio *chunk size*), and these panels were heavily coupled with both the implementation of the **`OldUnit`** class, as well as functionality and parameter.

When double clicking on a LoadSound unit - old or new - the first window (created by the **`doubleClick()`** method) loads a file browser to allow the user to select the audio file to load into Triana. The second (created in the **`userScreen(String fileName)`** method) prompts the user to choose to 'chunk' the data or load the entire file into memory.

Generally in Triana, units which have custom GUIs should utilise a parameter 'panel' class, for example; LoadSound.java and LoadSoundPanel.java, completely decoupling the implementation of the unit's functional code and the user interface. This approach was applied to the LoadSound unit, with the creation of the **`LoadSoundPanel`** class.

5.2.4 FFT

The FFT unit was originally developed by Dr. Ian Taylor and Dr Bernard Schulz, the creators of the Triana software. The FFT unit needed modifications to extend the newer **`Unit`** class and to support **`MultipleAudio`** data types without having to use a **`MultipleAudio`** to **`SampleSet`** converter (located in the Audio/Converters toolbox directory). Experimentation was conducted to try to increase performance and memory footprint by converting all data arrays to use **`float`** point instead of **`double`**, however this provided no performance or efficiency advantage. The FFT unit was modified to handle **`MultipleAudio`** input data types by converting the MultipleAudio data into a SampleSet (losing only audio format metadata) on the fly.

[2]All of the GUI implementation was stripped when porting the units to standalone. Please refer to the next section for the 'ported' implementation of the LoadSound unit

The FFT unit performs a Fast Fourier Transform or its inverse on both one-dimensional (a **`VectorType`**, such as a **`SampleSet`**) or two-dimensional (**`MatrixType`**, not used in the DART implementation) data sets. The Triana FFT can handle input data sets of any length, although it will work most efficiently if the prime factors of the input number are all small. If the input has signal information, this is used in creating the output (described shortly). The properties of spectral data sets are also discussed shortly.

If the user chooses automatic operation in the user interface window (as default), and if the input implements a **`Signal`**[3] (such as SampleSet) or **`Spectral`**[4] interface, then the unit automatically performs the correct type of transform. Two successive applications of the FFT unit, starting with either a **`SampleSet`** or a **`ComplexSpectrum`**, will produce a final output identical to the original input, to within round-off error.

`SampleSet` extends **`VectorType`** and implements the **`Signal`** interface. As the **`Unit`** converts from a **`MultipleAudio`** to **`SampleSet`**, then signal parameters (sampling rate, etc) are used to produce a correctly normalised spectrum, i.e an approximation to the continuous Fourier Transform. Normalisation is applied to ensure that the FFT output is a sampled representation of the continuous Fourier transform $X(f)$ of the function of time $x(t)$ represented by the input set, according to the DFT equation outlined in the Chapter 2 (Background).

The output data is a one-sided representation of the spectrum (negative frequencies are not stored) if the input is a **`SampleSet`**. The output is a **`ComplexSpectrum`**, but if the input is a **`ComplexSampleSet`** with sufficient symmetry, the output will be a real **`Spectrum`**. For display or further analysis or processing, the **`ComplexSpectrum`** should be transformed into an amplitude spectrum or a power spectrum, as is the case in the DART Sub-Harmonic Summation scenario.

`ComplexSpectrum` is the basic Triana class for holding one-dimensional Fourier transforms and stores a one-dimensional array of double-precision complex numbers representing a complex Fourier spectrum. It includes a **`Triplet`**[5] giving the integer values of the index

[3]Signal is an interface that can be implemented by any Triana data types that include data that has been acquired from a time-based data stream

[4]Spectral is an interface that can be implemented by any Triana data types that include data that represent frequency-domain data or that have been put through a Fourier transform

[5]Triplet is an Object containing three numbers that can generate a uniform sequence of numbers, which can be used to represent an array index or an independent variable in, say, graphical applications

of this array, and five new parameters:

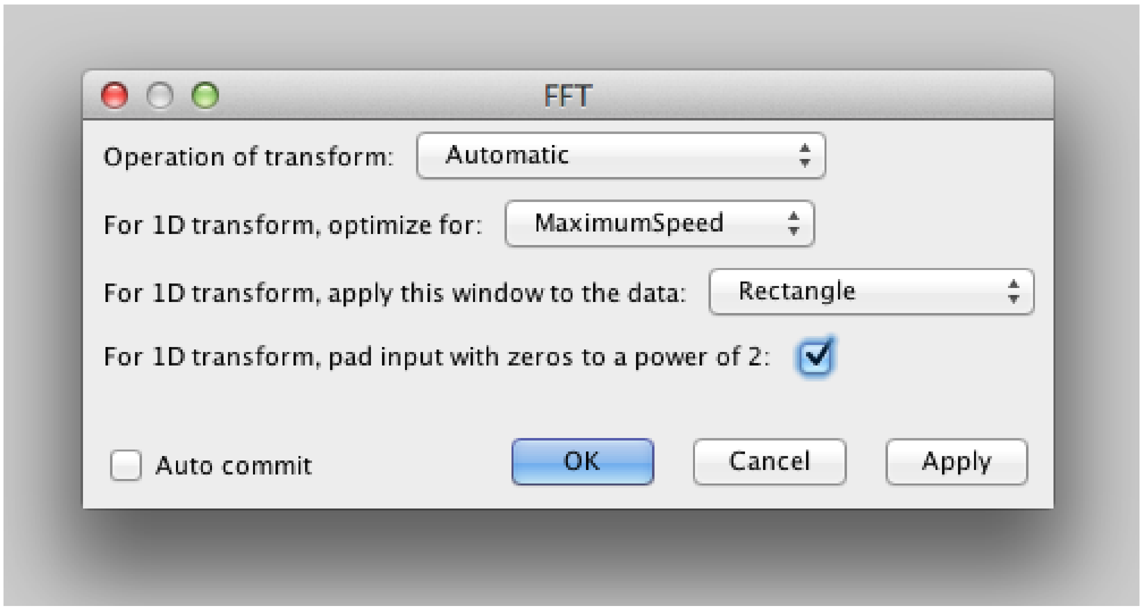
- **`resolution`** - a Double giving the frequency resolution
- **`highestFrequency`** - a Double giving the value of the largest frequency represented in the data set
- **`nFull`** - an Int giving the number of points in the original data set from which the data here were derived (used if the data have been reduced to one-sided or narrow-band spectra)
- **`twoSided`** - a Boolean flag that says whether the ComplexSpectrum is one-sided or two-sided
- **`narrow`** - a Boolean flag that says whether the data are a narrow bandwidth derived from a larger full-bandwidth spectrum

`ComplexSpectrum` is derived from the **`VectorType`** class and implements the **`Spectral`** interface, the Triana model for storing spectral data. The model is general enough to contain a multi-dimensional Fourier transform derived from a complex data set containing an arbitrary number of points. It is complete enough to ensure that inverse Fourier transforms can be done correctly automatically, even if the data set is a narrow-band spectrum extracted from a full spectrum that obeys the Triana model.

The FFT parameter window shown in Figure 5.2 allows the user to choose the type of operation of the transform. There are 5 modes:

- Automatic
- Direct
- Direct/normalised (1/N)
- Inverse
- Inverse/normalised (1/N)

For one-dimensional data sets there are three further options. One is to optimise for the speed of the calculation or the storage space of the result. If the user chooses direct or inverse operation, the unit ignores the type of the input data set and performs the requested operation. It applies normalising factors ($1/N$) if the appropriate factor is chosen.


FFT

Operation of transform: Automatic

For 1D transform, optimize for: MaximumSpeed

For 1D transform, apply this window to the data: Rectangle

For 1D transform, pad input with zeros to a power of 2: ✓

☐ Auto commit

Figure 5.2: The GUI displaying the FFT Unit parameters

For convenience, the user has an option to maximise speed of the transform in the 1D case, or to minimise storage. The minimum storage option produces a one-sided output, while the maximum speed option produces a two-sided output. A third option allows the user to specify the windowing function required (if any). For some purposes, a windowing function should be applied to data before it is transformed. The user interface gives a choice of 27 windows, including common choices such as Bartlett, Blackman, Gaussian, Hamming, Hanning, and Welch.

The final option dictates whether the unit should automatically pad out the data with zeros to the right when applying the FFT. For efficient operation, it is best that the input length should be a power of 2, significantly speeding up the processing of the input data. The FFT parameters required for SHS algorithm are:

- Operation of Transform: ***Automatic***
- For 1D transform, optimise for: ***Maximum Speed***
- Pad to Zero: ***Yes*** (checked)
- The Window Type is varied between the ***27*** options as part of the parameter sweep experiment.

5.2.5 OneSide

The OneSide unit converts two-sided spectra to one-sided spectra. If the input spectrum is not conjugate-symmetric, then information will be lost. The OneSide unit requires one-dimensional input data.

Although the FFT, given a **`SampleSet`** input (as is the case with this implementation of the SHS algorithm) should output a one-sided **`ComplexSpecrum`**, this unit is added so that other DART/MIR developers can simply change the first unit in the chain to one which does not output a **`MultipleAudio/SampleSet`**, without running into any issues.

Other than various functional changes to convert the unit to work with the newer **`Unit`** class, the OneSide unit (the processing handled in the **`process()`** method) remains unchanged from the original Triana implementation. This unit outputs the same data type it receives - in the case of the DART SHS algorithm, it outputs a **`ComplexSpectrum`** to the **`AmplitudeSpectrum`** unit.

5.2.6 AmplitudeSpectrum

In order display the results of the FFT in a graph (and for the SHS algorithm to work) an Amplitude- or Power-Spectrum is used. The SHS algorithm requires clear peaks in the spectrum, however these can be difficult to locate. Applying a power or amplitude spectrum decreases the signal-to-noise ratio of the spectrum - amplifying the peaks. Squaring the spectrum gives the power spectrum, which makes the peaks ‘stand out’.

The power spectrum gives a plot of the portion of a signal’s power (energy per unit time) falling within given frequency bins. The amplitude spectrum is closely related to the power spectrum. A (single sided) power spectrum is computed by squaring the single-sided RMS (root-mean squared) amplitude spectrum. Conversely, it is possible to compute the amplitude spectrum by taking the square root of the power spectrum.

The AmplitudeSpectrum was used in the DART scenario. The AmplitudeSpectrum is the absolute magnitude of the input complex numbers provided by the OneSide class. The AmplitudeSpectrum unit computes the amplitude spectrum from an input **`ComplexSpectrum`**. The unit outputs a **`Spectrum`** containing these values. The AmplitudeSpectrum unit transforms an FFT into an amplitude spectrum suitable for graphing

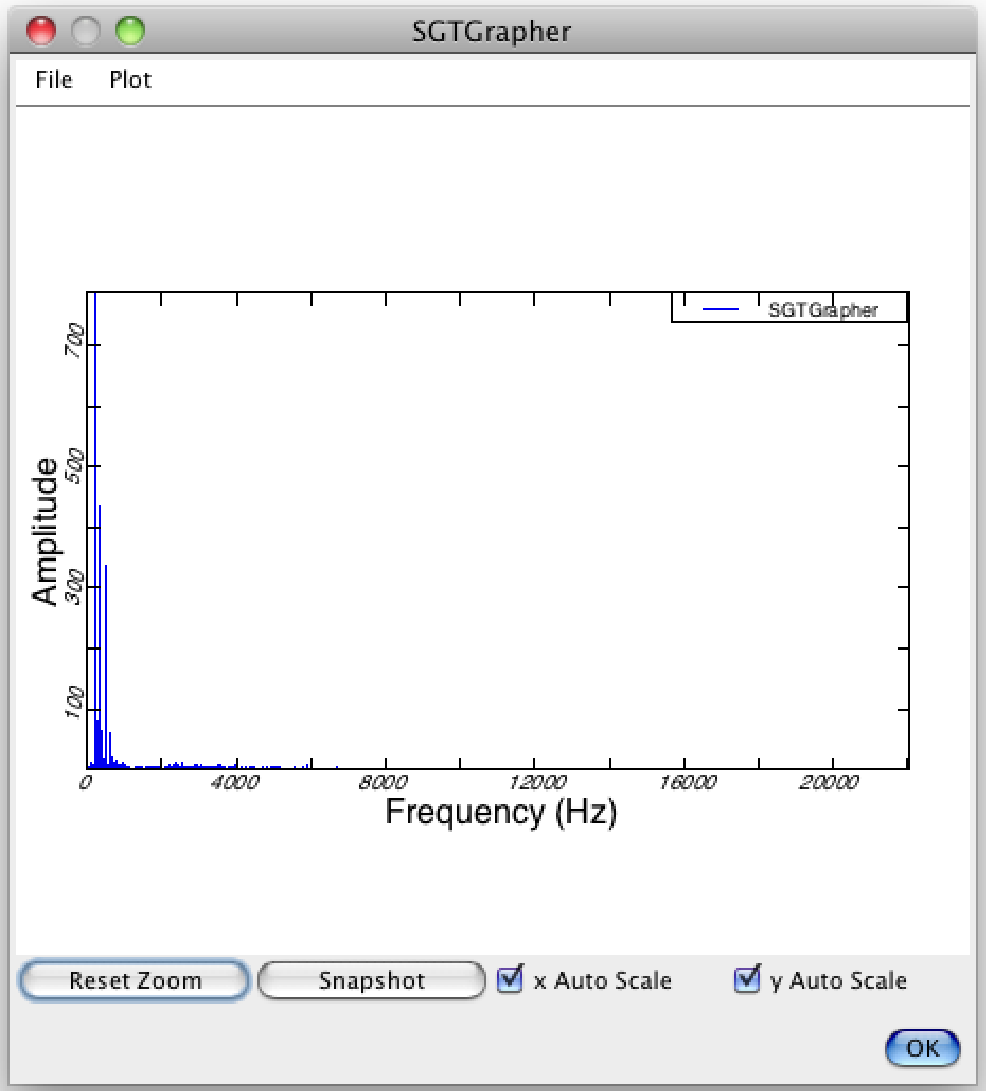


Figure 5.3: The SGTGrapher unit GUI showing the frequency-amplitude of an example audio file

or analysis.

At this point in the creation of the task graph it is now possible to display the FFT results for each chunk of audio accurately using a Triana graphing unit such as the 'SGT-Grapher'. This was done (see Figure 5.3) in order to test the algorithm so far, and demonstrate that the audio was now represented correctly in the frequency domain.

5.2.7 PitchDetection

The PitchDetection unit performs the main part of the 'Sub-Harmonic Summation' algorithm. The PitchDetection unit takes in a **`Spectrum`** as its input and outputs a number (integer) that denotes the overall pitch (in Hertz) of the current 0.5 second audio chunk. The following points summarise the SHS algorithm implemented in this unit, after accepting the incoming **`Spectrum`** array:

- Sort input Spectrum array values into ascending order
- Reverse the array, so that it is descending in size, making it easier to find the highest ***n*** values in the array
- Copy highest ***n*** values into a new array called **`topFFreqValues`**, representing potential candidates for the fundamental frequency - the default value for n is 30. This value is varied in the parameter sweep experiments (discussed shortly)
- Go through **`topFFreqValues`** and find the index/position of where each top value is located in the original Spectrum array
- Calls **`investigateSpectrumPoints()`** method and turns these array positions into frequencies (integers, in Hertz)
- Goes through each of top frequency and looks at all of its integer factors, add them together and then search for the lowest frequency with the highest value.

The final step in the algorithm highlighted above looks at first **`noOfHarmonics`** - the default is 6 harmonics, but is variable by the user and is varied in the parameter sweep experiments.

The implementation of the PitchDetection unit (and in turn the SHS algorithm) is integral to the thesis and can now be looked at in more detail. Following the steps above, after creating a new array of potential candidates for the fundamental frequency, the **`investigateSpectrumPoints()`** method is then called:

```
public void investigateSpectrumPoints() {
  freqArray= new double[topFFreqValues.length];
  freqdiv = freqSpacing / Math.floor(freqSpacing);

  for (int i = 0; i < noFreqPoints; i++) {
    freqArray[i] = indexArray[i] * (freqSpacing);
    // Gives the potential fundamental frequency
    }
    temp = Math.rint(((freqSpacing/2)+1));
    for (int i = 0; i < freqArray.length; i++) {
      detectPitch(freqArray[i]);
      for (int j = 1; j < temp; j++) {
        if (((freqArray[i] - (freqdiv*(double)j)) > 0)) {
          detectPitch(freqArray[i] - (freqdiv*(double)j) );
        }
        if (((freqArray[i] + (freqdiv*(double)j)) < 22050)){
          detectPitch(freqArray[i] +  (freqdiv*(double)j) );
        }
      }
    }
  }
```

The **investigateSpectrumPoints()** method converts each of the top n values and converts them to a frequency integer, stored in a new array. The method goes through each of these top n values and multiples them by the *frequency resolution* variable **freqSpacing**. This is set at the beginning of the class, at the top of the PitchDetection **process()** method. The value of the **freqSpacing** determines the accuracy of the results. The frequency ranges are divided into **N** sections, however only half this number are usable since the graph is symmetrical about its midpoint and one-sided, therefore the number of frequency bands is **N/2**. Since the highest frequency representable in any data is half the sampling frequency due to Nyquist/Shannon theorem, the width of each frequency band is represented by **freqSpacing**. Filling the variables with the figures that are used in the DART/SHS algorithm gives:

$$freqSpacing = (44100/2)/(16385 * 2) = 1.345743057674702 \qquad (5.1)$$

A sampling rate of 44,100Hz and an input size of 16385[6] gives a result of roughly 1.3457. Therefore the results can be accurate down to around **F0/21.83**, where the difference between each note is greater than 1.3Hz. However, if higher harmonics are present then the subharmonic summation algorithm can still work to resolve these pitches. **investigateSpectrumPoints()** multiples the **indexArray** value by the **freqSpacing**:

```
  freqArray[i] = indexArray[i] * (freqSpacing);
```

[6]This figure is larger than 11025 because the FFT unit has padded the figure with zeros

This gives an integer value which represents a frequency in Hertz, and can be thought of as a *potential* fundamental frequency. The **investigateSpectumPoints()** method then invokes **detectPitch()** on these top n frequencies and around these also +/- **freqSpacing** (the window size), as shown above.

The **detectPitch()** method is called from the **investigateSpectrumPoints()** method and takes a single frequency, considers all of its integer factors, adds them together, and searches for the *lowest frequency with the highest value.* This method looks at first **noOfHarmonics** harmonics, the default being 6 and is varied in the parameter sweep experiments.

```
void detectPitch(double fundamental) {
  freq = fundamental;
  freqbin = (int) Math.floor(freq);

  for (int i = 0; i < noOfHarmonics; i++) {
    spectrumPoint = freqToArray(freq + (freq * (double) i));
    if (spectrumPoint < power.length){
      freqResults[freqbin] = freqResults[freqbin] + power[spectrumPoint];
    }
  }
}
```

For each harmonic (1 to 6 if the default value of **noOfHarmonics** is six), each frequency in addition to its integer multiples are calculated, and then the algorithm searches for the value with both the lowest frequency and the highest amplitude. The referenced **freqToArray()** method maps a frequency value back to a value in the original data array **power**:

```
int freqToArray(double freq) {
  arrayPointDouble = freq / freqSpacing;
  arrayPointInt = Math.floor(arrayPointDouble);
  arrayPoint = (int)arrayPointInt;
  fraction = arrayPointDouble - arrayPointInt;

  if (fraction > 0.5) {
    arrayPoint = arrayPoint + 1;
  }
  return arrayPoint;
}
```

Returning to the PitchDetection **`process()`** method, the final step is to go through the results of the top summed frequency factors and find the highest summed value, called *max*. This value allows the location of the final result, which is the algorithm's chosen fundamental frequency for the current 0.5 second of input data being analysed:

```
result = 0;
for (int i = 0; i < freqResults.length; ++i){
  if (freqResults[i] > 0){
    if (freqResults[i] > max){
      max = freqResults[i];
      result = i;
    }
  }
}

Integer integer = new Integer(result);
output(integer);
resetVariables();
```

The integer result for the current chunk of audio is then passed on to the NoteMapper unit.

5.2.8 NoteMapper

The NoteMapper unit class works in tandem with the **`NoteWriter`** class. The NoteMapper **`process()`** method takes an Integer as an input from the PitchDetection unit and maps it to a musical pitch notation value (e.g. an integer value of 440 would be equal to an 'A4' note), stored in a 2-dimensional array in the **`NoteWriter`** class. The result is written to a results text file as both an integer value (e.g. 440) and a pitch notation value (e.g. A4).

The NoteMapper class simply creates a new NoteWriter object and sets the current frequency result (Integer) for the current 0.5 second chunk of audio to a value in a Vector created in the NoteWriter class by calling the **`addNote()`** method.

Once the NoteMapper class receives 672 chunks (the pre-set number of chunks per input file, as each input audio file contains 672 x 0.5 second chunks of audio), the end of the audio input file has been reached and the method calls the **findNoteMap()** and **writeFile()** methods. **findNoteMap()** goes through the vector **notesVector**, containing all of the results for the audio file, and maps each Integer to its corresponding frequency value. The notes are mapped using a 2-dimensional String array, mapping notes from C0 to D#8. A example of this static 2-dimensional array can be shown as:

```
String[][] noteMapArray = {{"begin","0"}, {"C0", "16.35"}, {"C#0", "17.32"},
     {"D0", "18.35"}, {"D#0", "19.45"}, {"E0", "20.60"}, {"F0", "21.83"}, {"
    F#0", "23.12"} ---> {"F#7", "2959.96"}, {"G7", "3135.96"}, {"G#7",
    "3322.44"}, {"A7", "3520.00"}, {"A#7", "3729.31"}, {"B7", "3951.07"}, {"
    C8", "4186.01"}, {"C#8", "4434.92"}, {"D8", "4698.64"}, {"D#8",
    "4978.03"}};
```

Where **--->** represents the 2-dimensional array values from F#0 to F#7.

The **findNoteMap()** method (the complete method is given below) creates Object and String arrays called **criteria** and **notesArray** respectively, both equal in size to **notesVector** (**notesVector** contains all of the 672 notes of the input audio file as Integer results). **notesVector** is copied into the criteria array before processing each Integer value in **notesVector**, and finding the correct note value.

```
public void findNoteMap(){

  Object[] criteria = new Object[notesVector.size()];
  notesArray = new String[notesVector.size()];
  notesVector.toArray(criteria);

  for (int i = 0; i < notesVector.size(); i++){
    int arrayFreq;
    int high = 0;
    int low = 0;

    for (int j = 0; j < noteMapArray.length; j++) {
      arrayFreq = Integer.parseInt(criteria[i].toString());
      if (j > 0){
        double highBound = Double.parseDouble(noteMapArray[j][1]);
        double lowBound= Double.parseDouble(noteMapArray[j-1][1]);
        if ((lowBound < arrayFreq) && (arrayFreq < highBound)){
          high = (int)Math.round(Double.parseDouble(noteMapArray[j][1].
              toString()));
              low = (int)Math.round(Double.parseDouble(noteMapArray[j-1][1].
                  toString()));
              if ((arrayFreq - low) < (high - arrayFreq)){
                note =  (noteMapArray[j-1][0].toString()); // the lower note
              }
              else{
                note =  (noteMapArray[j][0].toString()); // the higher note
              }
              notesArray[i] = note;
            }
            if (highBound == arrayFreq){
              note =  (noteMapArray[j][0].toString()); // the higher note
              notesArray[i] = note;
            }
          }
        }
      }
    }
```

The method considers each value in **`current`** (equal in size to **`notesVector`**), and then looks at each value in the 2D array **`noteMapArray`** (containing the mapping of each number result with the corresponding pitch notation result) and compares the current Integer in **`current`** with each **number** value (the second value) in the 2D array. The purpose of this is to find the two values that are higher and lower than the integer note value:

```
    for (int j = 0; j < noteMapArray.length; j++) {
      arrayFreq = Integer.parseInt(criteria[i].toString());

      if (j > 0) {
        double highBound = Double.parseDouble(noteMapArray[j][1]);
        double lowBound = Double.parseDouble(noteMapArray[j-1][1]);
      }
```

The algorithm aims to iterate through every value in the 2D array to try and find a suitable pair of low and high bounds between which the **`arrayFreq`** integer fits. It is important that the **`highBound`** and **`lowBound`** figures remain as **`Doubles`**; when considering low frequencies, the difference between two frequencies can be less than 1.

When the method iterates up through the **noteMapArray** array and finds a suitable pair of high and low bounds, then the following **if** statement will return true and the subsequent code will be executed:

```
if ((lowBound < arrayFreq) && (arrayFreq < highBound)){
  high = (int)Math.round(Double.parseDouble(noteMapArray[j][1].toString()));
  low = (int)Math.round(Double.parseDouble(noteMapArray[j-1][1].toString()))
      ;
}
```

The *high* and *low* integers are rounded down by calling **Math.round**, and are required in order to find out if **arrayFreq** is closer to the higher or lower bound (or if there is an exact match):

```
if ((arrayFreq - low) < (high - arrayFreq)){
  note =  (noteMapArray[j-1][0].toString()); // the lower note
} else {
  note =  (noteMapArray[j][0].toString()); // the higher note
}
notesArray[i] = note;

if (highBound == arrayFreq) {
  note =  (noteMapArray[j][0].toString()); // the higher note
  notesArray[i] = note;
}
```

Once the **findNoteMap()** method is finished running through all 672 notes, the **writeFile()** method is called to write **notesVector** and **notesArray** to a results text file, with the results separated by commas. The location and argument are set by the user - or in the case of the DART command line interface (explained later), the user can set the output results file from the command line. This produces a text file with the following format (only the first 12 notes are shown):

```
DART RESULTS


64, 72, 82, 87, 98, 109, 123, 131, 131, 145, 164, 176,...


C2, D2, E2, F2, G2, A2, B2, C3, C3, D3, E3, F3,...
```

5.2.9 MP3 Units

Several other units of note were also created, but not extensively used in the DART experiments. These include units such as **`LoadMP3`** and **`ID3Xtractor`**, allowing the user to load and analyse MP3 files instead of much larger WAV files, or to extract the ID3 Metadata in MP3 files (e.g. artist name, song and album titles), opening up the possibility of integrating some level of statistical analysis, perhaps concurrently alongside the content-based audio analysis. JavaSound does not natively allow for the decoding of MP3 files, and no libraries currently exist that allow the developer to reuse code created for WAV audio files with MP3 files, in a simple and transparent way. The implementation of the MP3 decoding unit (**`LoadMP3`**) relies on third-party libraries and is based on implementation methods suggested in [112]. The following library files were integrated into the core Triana/lib/audio directory:

- jl1.0.1.jar
- tritonus_share.jar
- mp3spi1.9.5.jar

Although sending MP3 files to the worker nodes would have lowered the download size and reduced the impact on the workers bandwidth requirements, MP3 compression is also a lossy format that works by reducing the accuracy and content of the parts of audio that are considered to be beyond the auditory resolution or ability of most humans. This compression could have influenced and undermined the results of the DART SHS experiments, given that frequency content of the input data would have been modified.

Distributing the audio files to the worker nodes is also an excellent test of the XtremWeb and BOINC distribution mechanisms, which is extremely useful for any future DART experiments which may require sending large quantities of data to the worker. The mechanism must be tested in order to fulfil the hypothesis of the thesis, investigating if a scalable MIR platform based on these distributed mechanisms is feasible - even if the resolution and quality of the MP3 files would have had no effect on the results of the DART SHS experiments.

Further experimentation of analysing any potential differences between the analysis of MP3 encoded data (at various bit depths) and uncompressed 'CD quality' 16-Bit, 44.1KHz audio would make for interesting future work.

As the **`LoadMP3`** and **`ID3Xtractor`** units were not used in the implementation of the DART SHS algorithms, their implementation will not be discussed in detail in this thesis.

5.3 Creating The DART Execution Environment

Allowing a Triana workflow to run as a standalone application is necessary in order to use a distribution mechanism that is not heavily reliant on Triana, removing dependencies on the Triana software and lowering the worker's system requirements. Triana is a Java application and benefits from many of the advantages Java brings, as highlighted in the background chapter. It is a well coded, object oriented application and as such is structured in a way which enables components to be separated and used in isolation without many drastic modifications.

XtremWeb is written in Java and accepts a JAR file as an application. BOINC requires C++ applications, however options are available to 'wrap' the Java application archive into formats that are accepted by BOINC, as will be explained in the next section. In this section, the focus of the implementation is on the creation of a standalone JAR file.

As a *graphical* Problem Solving Environment, one of Triana's largest dependancies is on its extensive Graphical User Interface (GUI) with which the user communicates. Stripping the GUI away from the underlying code will allow for a much more streamlined operation. The GUI is of course used by the user to select and connect the Triana units and components, in a particular order, and to run the algorithms. With no GUI, the Triana taskgraphs need to be finalised and then mapped into a sequence which follows the flow of data that was designed in Triana. This can of course become an automated feature in the future; this section focusses on the implementation of standalone workflows and the implementation of the design of the framework, however. The implementation of this application can serve as a framework for any Triana work-flow or task graph that is required to run as a standalone application.

Porting all of the units to work using the new superclass **`Unit`** (as explained earlier) meant that the GUI was more decoupled from the functional code, allowing the variables set by the GUI to be set using the Command Line Interface. Triana can be stripped of all classes and methods that are not required for the execution of the particular DART algorithm or application that is being ported to work standalone. Only the dependancies of each unit in the workflow, as well as all of the Triana datatypes must be adhered to.

The Triana DART workflow/taskgraph must be reconstructed and *get* and *set* data from one unit of code to another. Each unit may have several adjustable variables, which must be available to set at runtime (from the command line) in order to change the outcome of the algorithm. In the case of DART, it is imperative to be able to adjust these variables with an interface that is easy to use and direct, enabling the SHS parameter-sweep experiment.

Creating a Java Archive (JAR), aggregates several classes and associated metadata into one executable file. The Dart.java class file is the main class that is used to create the JAR executable. The passing of Triana Data types or Java objects from one instantiation of each class to the next is handled by this class, whereby each unit in the Triana task graph is instantiated as an object. After instantiating the relevant unit objects in **`TestDart.java`**, the flow of data from one unit to the next must be controlled.

Given a simple example scenario where a **`LoadSound`** unit is to pass data to a **`Play`** unit (simply enabling and initiating the playback of high quality audio), the following method calls must be used:

```
loadsound.process();
```

The **`LoadSound`** unit (the first unit in the algorithm/workflow) begins processing/initialises the audio

```
Object outdataA = loadsound.getOutputData();
```

This creates an objected containing the output from the **`LoadSound`** unit.

```
play.setDataInput(outdataA);
```

The output is set as the input for the next corresponding unit (i.e. the Play unit)

```
play.process();
```

The **`process()`** method is called to initiate the processing of the next unit. This simple structure can be used to chain any number of units together.

5.3.1 Structure of the Standalone DART JAR

The main class of the DART application (JAR) is simply named **`Dart.java`**. This class accepts the parameters that were input from the command line, sequentially instantiates of all the units, and passes the output of one unit to the input of the next unit in the SHS algorithm sequence. It also, in tandem with the NoteMapper.java class at the end of the DART task graph, writes to the results file generated by the algorithm. The overall structure of the class can be shown as:

- Create Options instance and add possible CLI variable parameters
- Create a CommandLine BasicParser
- Initialise each DART Unit object used in the workflow
- Query or *interrogate* the CommandLine in order to check if the variable has been set
- Connect the units together and begin the workflow processing
- Write the results to a text file

The Jakarta Command Line Interface (now called the Apache Commons CLI[7]) is used to create the DART interface design described in the Design chapter. There are three stages to command line processing; the definition, parsing and interrogation stages.

Each command line must define the set of variables that will be used to define the parameters to the DART application. CLI uses the **`Options`** class as a container for the **`Option`** instance. The result of the definition stage is an **`Options`** instance. Once the **`Options`** object is instantiated, the various allowed parameters must be added to it using the **`addOption`** method. The follow code snippet shows how to specify an input file for DART:

```
Options opt = new Options();

opt.addOption("h", false, "Print help for this application");
opt.addOption("infile", true, "Name/loc of the input audio file. Must be 16
    bit/44.1KHz wav/aif");
```

The **`addOption()`** method has three parameters. The first parameter is a **`String`** that represents the **`Option`**. The second parameter is a **`boolean`** that specifies whether the

[7]`http://commons.apache.org/cli/`

Option requires an argument or not. In the case of a boolean option (sometimes referred to as a **flag**), when an argument value is not present, *false* is passed[8]. The third parameter is the description of the **Option**.

The input is then *parsed*. The parse method is defined in **BasicParser**, a sub-class of **Parser**, and takes an **Options** instance and a **String[]** of arguments, returning a **CommandLine**. The class **BasicParser** provides a very simple implementation of the flatten method[9]. The parse methods of **BasicParser** are used to parse the command line arguments. The result of the parsing stage is a **CommandLine** instance.

Each unit in the Triana DART task-graph is then instantiated:

```
loadsound = (LoadSoundNoGUI)Class.forName("mir.LoadSoundNoGUI").newInstance
    ();
fft = (FFT)Class.forName("mir.processing.FFT").newInstance();
oneside = (Unit)Class.forName("mir.processing.OneSide").newInstance();
amplitudespectrum = (Unit)Class.forName("mir.processing.AmplitudeSpectrum").
    newInstance();
pitchdetector = (PitchDetection)Class.forName("mir.processing.PitchDetection
    ").newInstance();
notemapper = (NoteMapper)Class.forName("mir.processing.NoteMapper").
    newInstance();
```

During the final CLI stage, *interrogation*, DART queries the **CommandLine** to decide which DART parameter variables to use, depending on **boolean** options and uses the option values to provide the data. The result of the interrogation stage is that the code is informed by the input that was supplied on the command line and processed according to the parser and **Options** rules.

The application then checks if the specified option is present by interrogating the **CommandLine** object. The **hasOption()** method takes a **String** parameter and returns true if the option represented by the **String** is present, otherwise it returns false. The only commands that are required are the **infile**, **outfile**, **fft_window**, **nofreqpoints**, and **noharmonics**. The following listing shows the command line being queried for the

[8]The required parameters in the DART SHS algorithm are given in the design chapter

[9]http://commons.apache.org/cli/api-release/org/apache/commons/cli/BasicParser.html#flatten(org.apache.commons.cli.Options,%20java.lang.String[],%20boolean

input file:

```
if(cl.hasOption("infile")){
  System.out.println("Input file = " + cl.getOptionValue("infile"));
  loadsound.fileName = home + File.separator + cl.getOptionValue("infile");
}
```

The **`loadsound.init()`** method is then called to trigger the **`createAudioInputStream(new File(fileName))`** and **`userScreen()`** methods, creating a new audio input stream and setting the correct chunk size for the input audio.

The unit workflow is reconstructed by taking the output of the first unit and setting it as the input of the following unit, using simple **`setOutputData`** and **`getInputData`** methods from the **`Unit`** class. A timer is set in order to calculate the total processing time for the current job, which is written to the results file. This complete process is demonstrated in the listing below.

```
System.out.println("DART processing has started...");
while (!loadsound.isLastChunk()){
  currentTime = (System.currentTimeMillis() - time)/1000;

  if (currentTime >= counter && currentTime > 19){
    System.out.println((int)currentTime + " seconds have elapsed");
    counter += 20;
  }
  loadsound.process();

  Object outdataA = loadsound.getOutputData();
  fft.setDataInput(outdataA);
  fft.process();

  Object outdataB = fft.getOutputData();
  oneside.setDataInput(outdataB);
  oneside.process();

  Object outdataC = oneside.getOutputData();
  amplitudespectrum.setDataInput(outdataC);
  amplitudespectrum.process();

  Object outdataD = amplitudespectrum.getOutputData();
  pitchdetector.setDataInput(outdataD);
```

```
    pitchdetector.process();

    Object outdataE = pitchdetector.getOutputData();
    notemapper.setDataInput(outdataE);
    notemapper.process();

    System.gc();
  }
```

The elapsed processing time is reported every 20 seconds, giving visual feedback to the user, allowing them to know that the program has not stalled. The final step is to write the results of the processing to a results file.

Finally, this means that when the DART.jar is presented with the following command line argument:

```
java -jar Dart.jar -infile DARTOboe.wav -outfile DART-MyResults.txt -
    nofreqpoints 1 -noharmonics 4 -fft_window Bartlett
```

The DART application will create a results file called **`DART-MyResults.txt`** with the following format:

```
DART RESULTS

64, 72, 82, 87, 98, 109, 123, 131, 131, 145, 164, 176,...

C2, D2, E2, F2, G2, A2, B2, C3, C3, D3, E3, F3,...

The DART application took 56 seconds to run
Input Analysis File = DARTOboe.wav
FFT Window Type = Bartlett
Number of Frequency Points = 1
Number of Harmonics = 4
```

5.4 Creating Input Audio Files

As mentioned in the design section, a total of six audio files containing notes played by six different instruments are used as input data to test the accuracy of the SHS algorithm: *Acoustic Guitar*, *Distorted Guitar*, *Oboe*, *Grand Piano*, *Violin* and *Tubular Bells*.

Recording high quality samples of the all the above instruments would have been time consuming, expensive, and not quite within the scope of the thesis. Instead, Apple's *Logic Pro*[10] (shown in Figure 5.4) and Native Instruments' *Kontakt 4*[11] were used to generate the required audio files.

Logic Pro is a Digital Audio Workstation and MIDI sequencer application for the Mac OS X platform and contains over 38GB of high quality audio, samples and loops. Included in this is 4GB of multi-sampled audio data created for Logic's EXS24 MarkII built-in sampler. This audio is sampled at 16 or 24-Bit/44.1KHz and mapped across a range of keyboard keys.

This allows users to 'play' the sampled instrument by triggering the samples from live or recorded MIDI input, such as a MIDI keyboard. MIDI (Musical Instrument Digital Interface) is an industry-standard protocol that enables electronic musical instruments, computers and other electronic equipment to communicate and synchronise with each other. MIDI's primary functions include communicating event messages about musical notation, pitch, velocity, control signals for parameters (such as volume, vibrato, panning, and cues) between two devices in order to complete a signal chain and produce audible sound from a source. This allows a musician to press a key on their MIDI keyboard, send a MIDI ON message to the sampler, and trigger the appropriate sample for playback. This MIDI value can be recorded and played back, as is the case in Logic or any modern sequencer.

Kontakt 4 (the latest version of the software at the time of writing) is the industry standard for professional samplers and contains over 43GB of high quality sampled audio (24-Bit/44.1KHz), with a wide range of instruments available to use. Kontakt works as an AudioUnit [113] plugin inside of Logic Pro, allowing it to be triggered by the same input MIDI data as the EXS24 samples.

[10]`http://www.apple.com/logicstudio/logicpro/`

[11]`http://www.native-instruments.com/en/products/producer/kontakt-4/`

The Kontakt sample library also contains a subset of The Vienna Symphonic Library (VSL)[12], a high-end, research-driven music software and sample library developed and based in Vienna, Austria. The VSL features world-class samples of orchestral instruments recorded by members of the Vienna Philharmonic Orchestra, and recorded at *The Silent Stage*, a recording studio specially constructed for this purpose. The number of recorded samples for the VSL alone is over 1.3 million, including legato and repetition (round robin) articulations. This library is widely considered to be one of the best orchestral libraries available, regularly used by musicians, classical composers/producers and film composers.

The Kontakt library features even higher quality samples than the Logic EXS library [13] - the samples are longer, larger in file size, with more articulations and samples per instrument. Kontakt allows for advanced scripting to be built into instrument 'patches', enabling features such as *Round Robin sampling.*

Round Robin sampling plays back a different sampled version of the same sound each time the sample is triggered, so that just like most acoustic instruments, each note sounds slightly different each time it is played. This feature becomes particularly important when striking or playing the same note multiple times, in order to avoid an unrealistic and repetitive sound. An obvious example of their usefulness would be their use in drum libraries, avoiding a 'machine gun' effect of rapidly repeating the same (for instance) snare drum sound. By having two, four or eight slightly different samples played back in sequence, the software can help to avoid artificial-sounding effects, as repeating the same note (which has 4 samples allocated for the 'Round Robin') will play back: *Sample 1, Sample 2, Sample 3, Sample 4, Sample 1, Sample 2* and so on.

Both samplers include multi-sampled patches, whereby the sample chosen for play back is different depending on the velocity value of the incoming MIDI note. This means that the sampled instruments react intuitively when played on a MIDI keyboard. The instruments will react differently when played with more vigour; a guitar string, for example, sounds very different when plucked gently, as opposed to when plucked more violently. A sampled piano or guitar should react in a similar way to the real instrument that is being represented; when a key on a MIDI keyboard is pressed gently, the corresponding 'low

[12] `http://www.vsl.co.at`

[13] while the 'quality' of all of all audio and quality is ultimately subjective, Kontakt remains a more modern, more featured sampler with a much larger sample library featuring more articulations and larger file sizes per sample

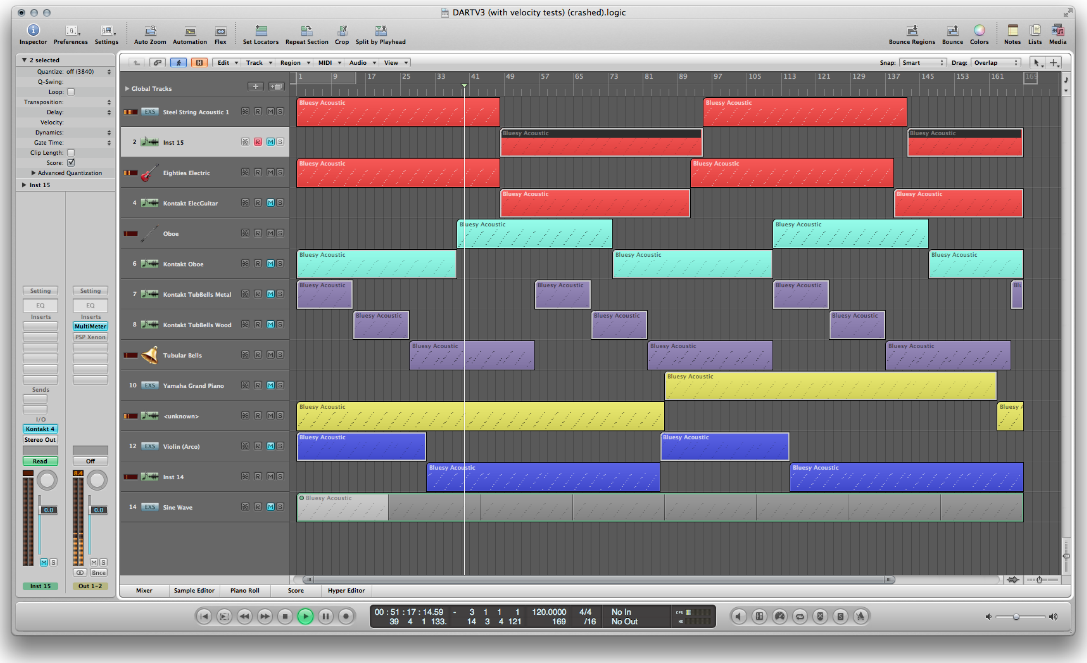


Figure 5.4: The Logic Pro *Arrange* page used to generate all size of the input audio files

velocity' sample is triggered. MIDI has an input value of 0-128 to indicate velocity, and therefore velocity 'groups' can be made within the sampler.

Using these professionally made samples, samplers, and applications as opposed to arranging to record the audio and create the samples as part of the thesis, allowed for a high level of accuracy when creating the input files. The samples were also checked with numerous pitch detection Audio Unit plugins, such as Logic's built in Tuner plugin and Native Instruments' Guitar Rig.

5.4.1 Audio Files for Initial Experiments

During the initial two prototype experiments (the runs of 50 and 160 jobs, respectively), the main focus of the implementation was to firm up the DART SHS algorithm and the distribution and deployment mechanisms. During the second experiment (created to test both the XtremWeb submission and DART parameter sweep scripts) the input audio file was not particularly important as the SHS algorithm was not being tested. A file called DARTAudio.wav was created and distributed for these experiments. This file was

5 minutes 36 long, and merely contained Acoustic Guitar samples playing C Major scale notes from C1-C7. The results were not particularly relevant and merely representative of a process intensive MIR application or algorithm. As long as the were no errors with the application, this file was suitable.

During some of the later, large-scale distributed experiments on XtremWeb, the initial input files were not as complex or 'thorough' as the files described below. These experiments failed due to bugs in the XtremWeb implementation, however this downtime allowed for an improvement to the audio input files while the XtremWeb team fixed the problems.

Initially, only sampled instruments from Logic's EXS24 library were used, and all of the sound files played whole notes from C1- C7, irrespective of the instruments real range or the range recorded by the sound designers.

While some instruments do not have the ability to play this complete range, samplers such as the EXS24 and Kontakt are able to interpolate and transpose (or 'stretch') any sample across any range of keys, pitching the sample up or down by a suitable amount. For example, if the Tubular Bells patch in the EXS24 sampler only has a range of 12 notes from E3-E4, the EXS24 - or any modern sampler - can map the lowest note (E3) across all of the keys *below* E3, and map the E4 note across all of the keys *higher* than E4. This allows the sampled instrument to have a greater range than the actual real instrument, allowing musicians and composers freedom and new creative options, at the cost of accuracy and realism.

During later implementation stages it was decided that it would be possible to improve the accuracy and robustness of these input files, by implementing the following:

- Both Kontakt and EXS24 sample libraries will be used
- Every note in the instruments natural (and sampled) range will be played back chromatically (no transposition)
- Each note for each instrument will be triggered at several MIDI velocities, thus playing back different samples with differing levels of intensity, opening the possibility of variations in harmonic content. Each note for each sampler instrument will be sampled at velocity levels of 30, 60, 90, and 120.
- Once all unique notes have been played, the audio will loop until the end of the file - if round robin samples are used, then the SHS algorithm will be able to analyse these different

samples

Despite Kontakt offering high quality or more realistic sounds, using both Kontakt and EXS24 samples to create the input files - for example - an input file containing audio from two grand pianos as opposed to just one, will allow for the results of the data analysis to be more robust and thorough. Having at least 2 sources for each instrument will guarantee a higher level of trust in the results produced by the SHS algorithm.

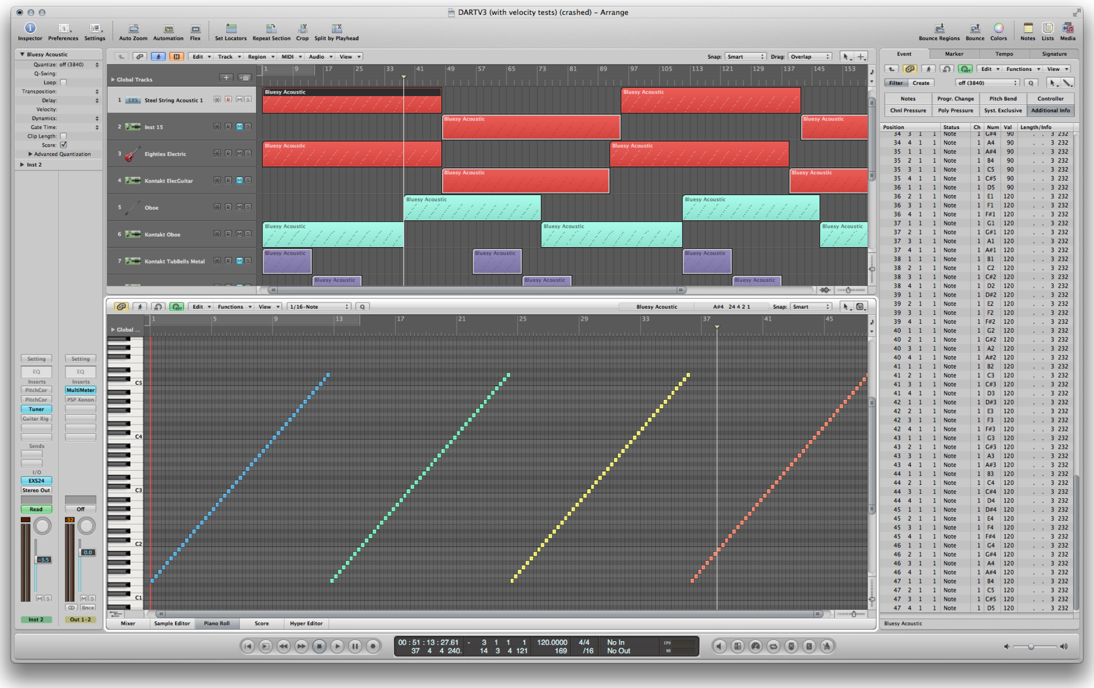


Figure 5.5: The Logic Pro MIDI Matrix showing notes at four different velocities, triggering different audio samples

Once all of the samples from both samplers have been played, the audio loops until the end of the file. Due to Round Robin sampling, this allows to test for any alternate samples playing given the same midi information as earlier on in the file.

All of the final audio files are 5 minutes 36 seconds long (59.4MB) and consist of 672 (0.5 second long) chunks of audio. All files are *monophonic*, meaning that only one note should play at a time. The decay, release and polyphony settings on the sampler was adjusted so that not only was there one note at a time, but there was no overlap, ‘tail’ or reverberation when the next note was triggered - therefore the polyphony was forced to 1 ‘voice’ at a time, and a release of 0ms.

During the downtime after the failed experiment, it also became apparent that the first audio input files contained some note overlapping errors. The release time and polyphony on each sampler hand instrument was adjusted so that no overlapping notes occurred.

5.4.2 Acoustic Guitar

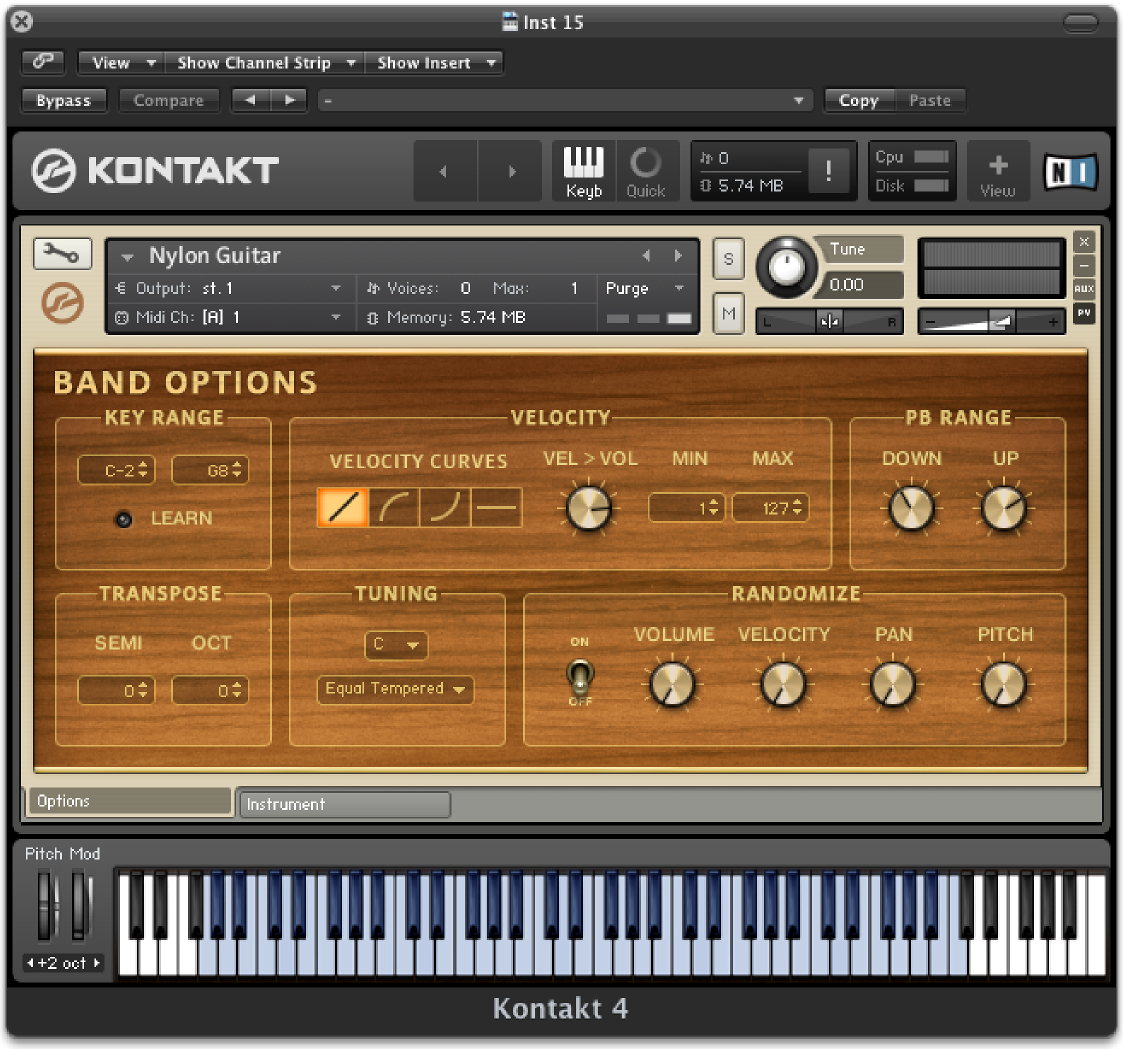
The DARTAcousticG.wav file consists of 188 notes (E1-D5) from the EXS24 sampler and 188 notes from Kontakt (the Kontakt Acoustic Guitar interface is shown in Figure 5.6). The EXS24 acoustic guitar features steel strings (giving a bright timbre), while the Kontakt samples are from a nylon string guitar. Steel strings have a defined and sharp sound that is a distinctive component of a wide range of popular music styles such as country and rock, and is often plucked with a plectrum or guitar pick. Nylon strings are usually found in classical and flamenco guitars and tend to have a very mellow, vibrant sound, played with the fingers or the fingernail.

Both guitar sample libraries are triggered and play back at the standard velocities of 30, 60, 90 and 120, giving a total of 376 unique notes before the samples begin to loop round. The samples contain a mix of plucked and fingerpicked sounds, depending on the various velocity levels and string type.

5.4.3 Distorted Guitar

The DARTDistortG.wav file has the a very similar note range to the Acoustic Guitar samples and again mixes EXS samples with Kontakt library samples. The EXS samples have a range of 47 notes (188 notes when triggered at the 4 different velocity levels) spanning from E1-D5, however the Kontakt samples have a slightly smaller range of 44 sampled notes E1-B4 (176 notes at 4 different velocity levels).

In order to create the distortion both the Kontakt and EXS samples are run through Logic‘s guitar amp simulator, Guitar Amp Pro, shown in Figure 5.8. Guitar Amp Pro can simulate the sound of popular guitar amplifiers and speakers, allowing the user to process ‘dry’ guitar signals directly. The amplifier, speaker, and EQ models emulated by Guitar Amp Pro can be combined in a number of ways to radically or subtly alter the tone. ‘Virtual microphones’ that model the characteristics of famous and well known microphones, are used to pick up the signal of the emulated amplifier and cabinet. There



Nylon Guitar
Output: st. 1
Voices: 0 Max: 1
Purge
Midi Ch: [A] 1
Memory: 5.74 MB
Tune 0.00
C-2 G8
1 127
0 0
C
Equal Tempered
+2 oct

Figure 5.6: The Kontakt 4 Sampler screen showing the GUI with the Nylon String Acoustic Guitar patch loaded

are two different microphone types, and the software allows the user to reposition them, changing the tone. Guitar Amp Pro also emulates classic guitar amplifier effects, including reverb, vibrato and tremolo, however these were not used when creating the DART files.

The EXS24 samples were very 'clean' (free from distortion) before being processed by Guitar Amp Pro, however the Kontakt 'Solo Guitar' did already have some distortion and colour to the notes, and running them through an instance of the Guitar Amp Pro plugin added yet more distortion. Of course, there are so many variables when creating a distorted guitar tone, and when creating the files, that the idea was simply to give different colour and tone options to realistic distorted guitar tones, in order to test the Sub-Harmonic Summation algorithm. Subjectively, the created files are a very good representation of

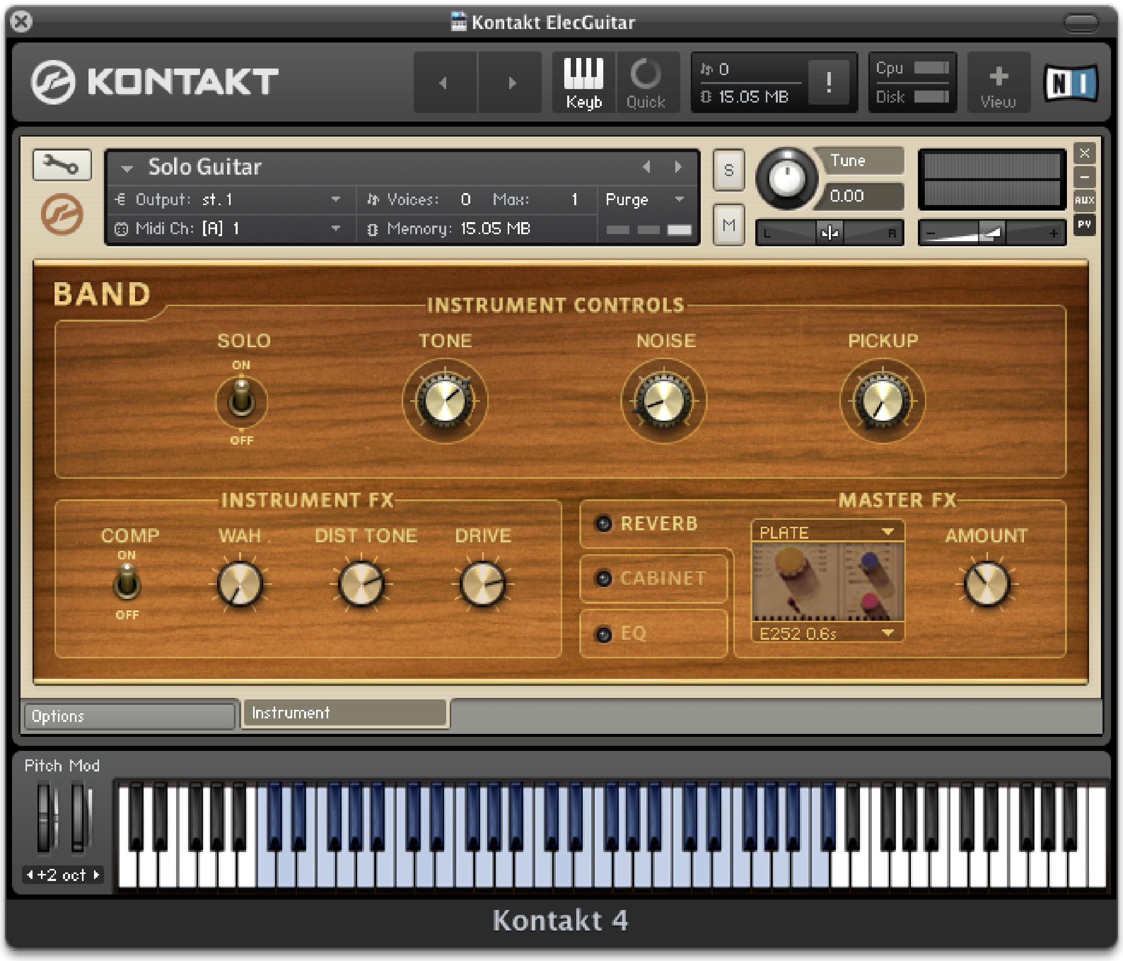
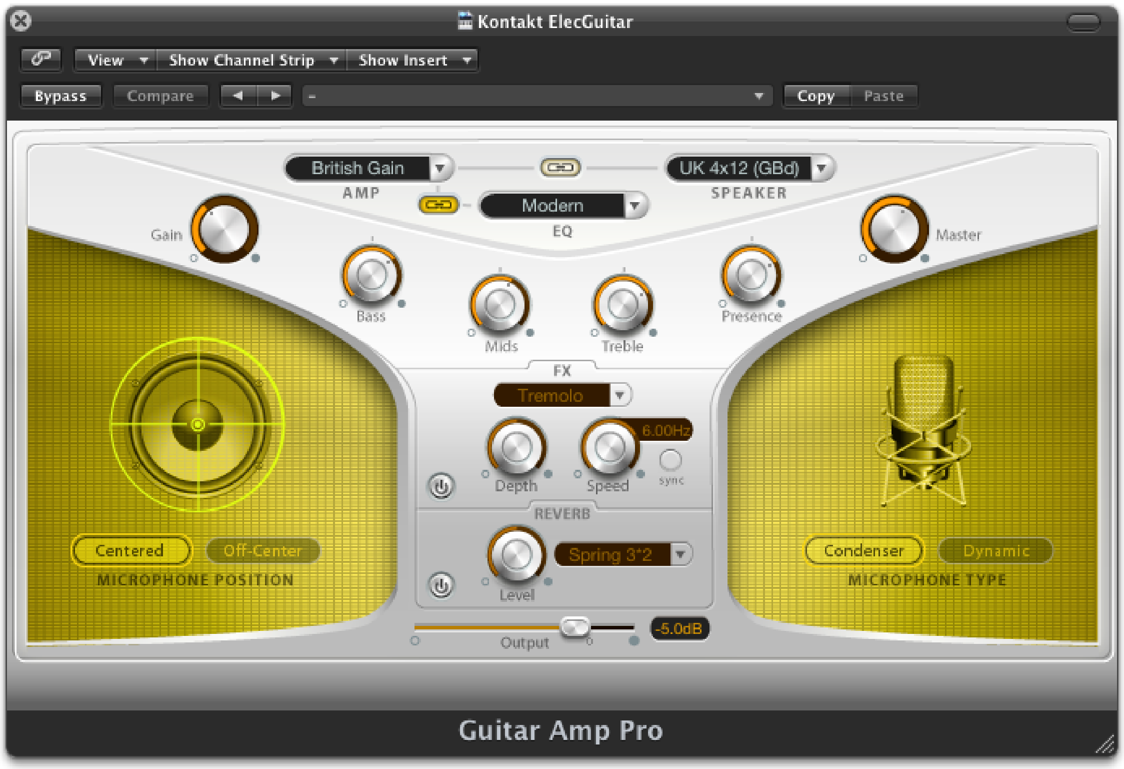


Figure 5.7: The Kontakt 4 Sampler screen showing the GUI with the Solo Electric Guitar patch loaded

two different distorted guitar tones.

5.4.4 Oboe

The Oboe can have a natural range of between A#2 to A6. The 'native' pitch range of the Kontakt and EXS (i.e. the sampled pitch range with no sampler transposition or interpolation taking place) differs, with Kontakt spanning from A#2 to A#5 (37 notes) and the EXS24 instrument spanning a higher range from A#3 to A6 (36 notes). Again, each sampled instrument is played back at 4 different velocities, giving a total of 292 notes. After the 292 notes have played back, the pattern loops until the end of the file.



AMP: British Gain
SPEAKER: UK 4x12 (GBd)
EQ: Modern
FX: Tremolo, 6.00Hz
REVERB: Spring 3*2
Output: -5.0dB

Figure 5.8: The Guitar Amp Pro screen showing the settings used on both sets of guitar samples

5.4.5 Grand Piano

The DART Piano audio file features the widest range of input data of all six input files and largely dictated the length of the overall files. The DARTPiano.wav audio file - as all DART audio input files are - is made up of 672 0.5 second notes. However the piano contains 664 unique 0.5 second samples. 304 notes are from the Logic EXS 'Yamaha Grand Piano' (range from A0 to C7, giving a total of 76 notes), while 340 unique notes (range C0 to C7) are from the SampleTekk[14] *PMI: The Emperor* sample library. The SampleTekk library is one of the largest and most thorough sample libraries used when creating the DART file.

This large, high-quality library samples a Bösendorfer 290 SE Grand Piano and contains 5.24GB of sampled audio data. The Bösendorfer 290 SE has a large dynamic range (85 piano keys - 3 less than is considered standard in modern Grand Pianos), and has 24 sample recordings for each key - 12 recorded velocity layers and with 12 separate sustain pedal down layers and release triggered samples. The sampled piano was equipped with an

[14]`http://www.sampletekk.com`

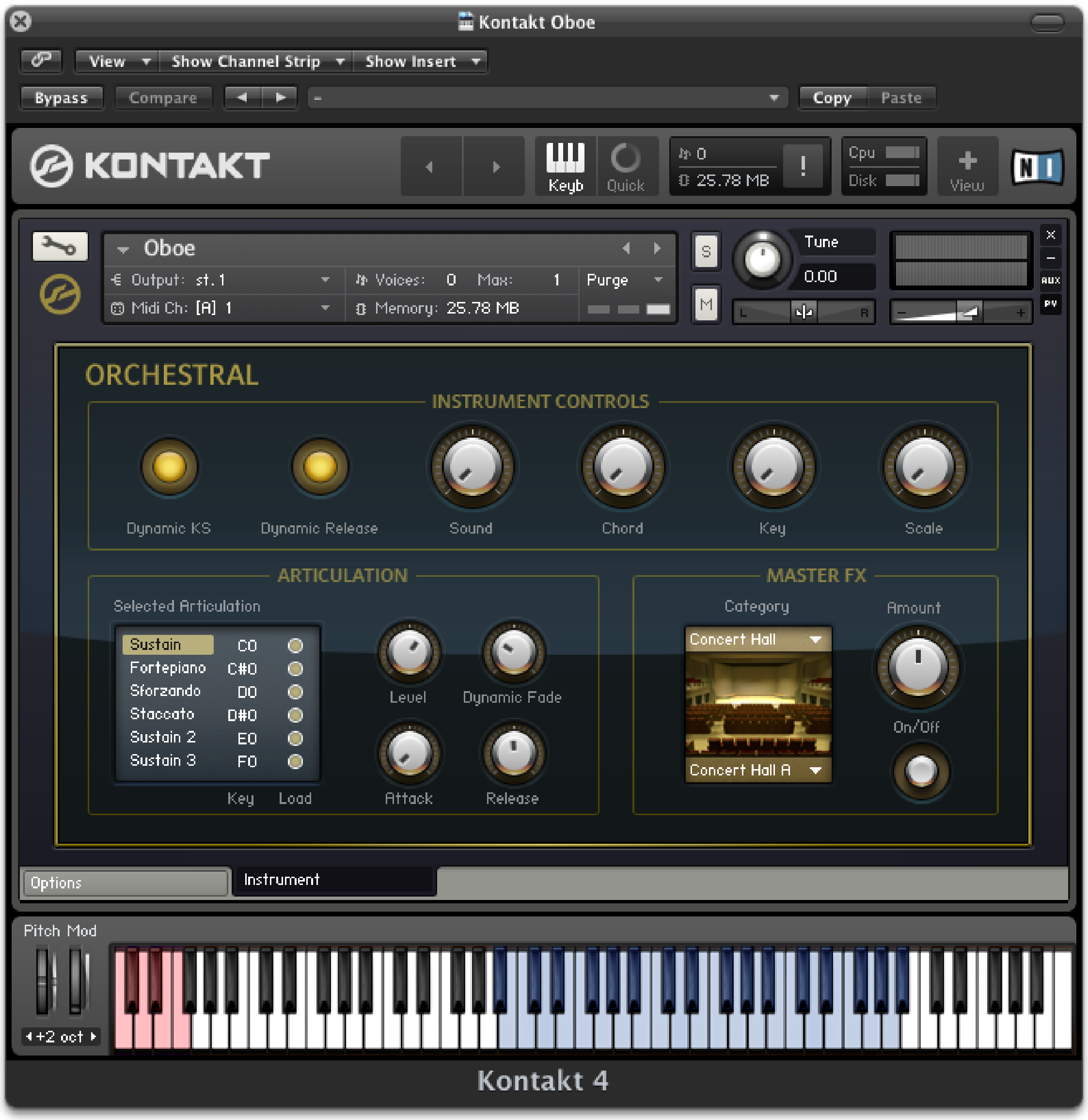


Figure 5.9: The Kontakt 4 Sampler screen showing the GUI with the Oboe patch

advanced computer operated playback mechanism that was designed by mathematician, scientist and inventor Wayne Stahnke[15]. The mechanism actually operates the piano keys and pedals with over 1,000 steps accuracy for inverse hammer velocity.

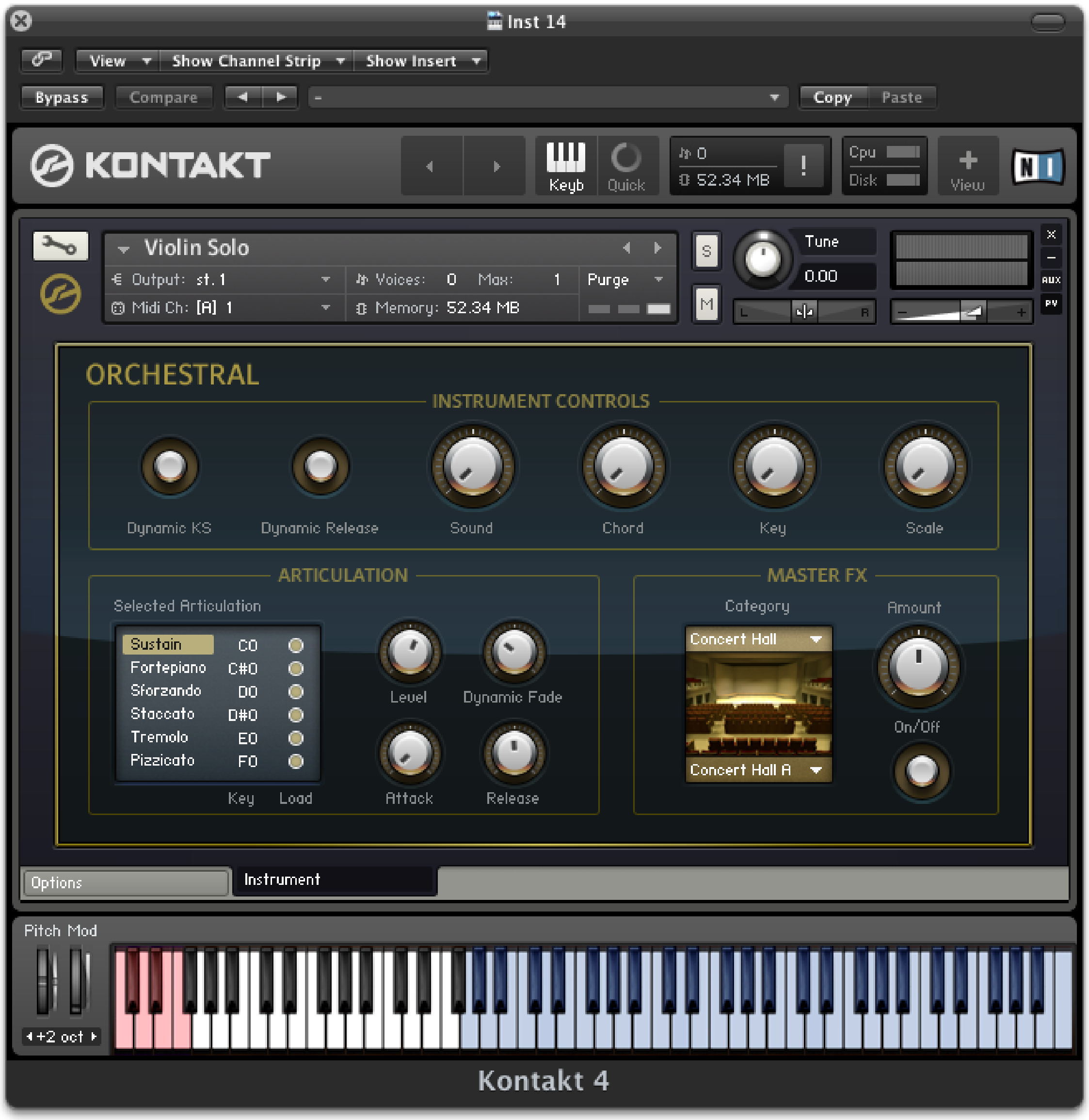
The Stahnke computer system enabled the capture of each velocity layer for this library with absolute velocity levels, guaranteeing a totally even response across the whole keyboard for all velocities. The 24 samples per-key are further divided in 64 velocity groups, each a slight variation of the underlying velocity layer sample.

[15]http://www.live-performance.com/about.html

The notes played back in the DART example were all ‘pedal up’ and each note triggered cuts off the previous note, by adjusting the polyphony and release time on the sampler interface.

5.4.6 Violin

The violin has a natural note range of G3 - A7, however the two differing sample libraries have different sampled ranges. The EXS24 has a range of G3 to C6 (30 notes), while the Kontakt ‘Solo Violin’ has a range of G2 to C7 (54 notes). When played back at all four velocities, this gives a total of 336 notes. The violin interface is shown in Figure 5.10.



Output: st. 1
Midi Ch: [A] 1
Voices: 0 Max: 1
Memory: 52.34 MB
52.34 MB
Tune 0.00

Sustain C0
Fortepiano C#0
Sforzando D0
Staccato D#0
Tremolo E0
Pizzicato F0

Concert Hall
Concert Hall A
+2 oct

Figure 5.10: The Kontakt 4 Sampler screen showing the GUI with the Violin patch

5.4.7 Tubular Bells

Tubular Bells can vary in size depending on the number of tubes or chimes, usually coming in one of three varieties:

- **1.5 Octaves: C4 - F5 (18 tubes)**
- **2 Octaves: F3 - F5 (25 tubes)**
- **2 1/3 Octaves 'Philharmonic tubular bells': Eb3 - G5 (29 tubes)**

The construction of the tubes and the material that the tubes are made of also changes the timbre of the notes massively. The Kontakt VSL library (Figure 5.11) contains both wooden and metal tubular bells, giving a total of 3 sampled instruments in the DART-Tubbells.wav input file:

- **Logic EXS24** - Metal Tubular Bells (Philharmonic) with a range of D#3 to G5 (29 notes)
- **Kontakt** - Metal Tubular bells with a one octave range from E3 to E4 (13 notes)
- **Kontakt** - Wooden Tubular bells5.11 with a one octave range from E3 to E4 (13 notes)

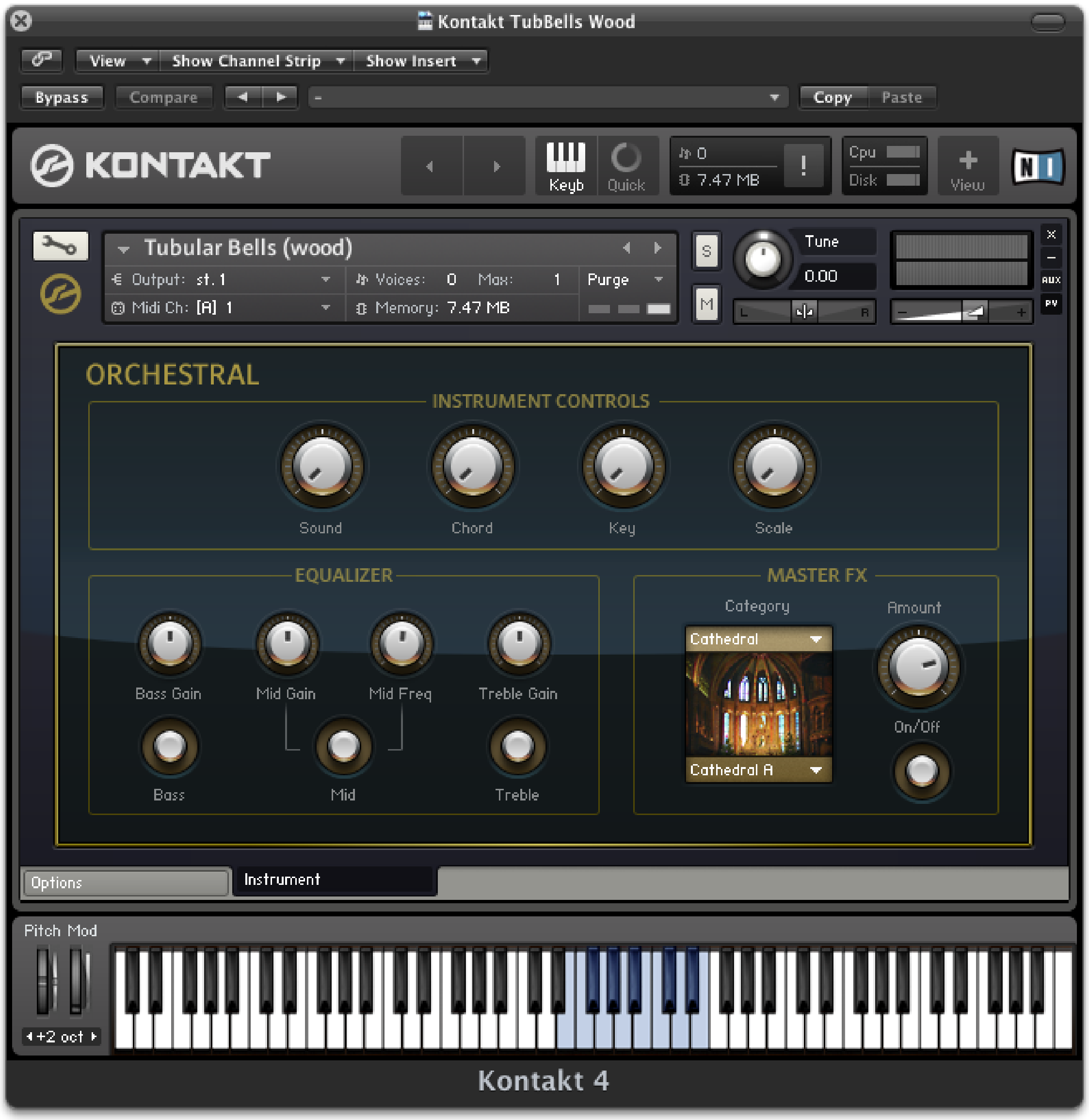
All three instruments are triggered and played back at all four MIDI velocities (30, 60, 90, 120), giving a total of 220 distinct sampled notes in the DARTTubbells.wav file, before the notes loop round until the end of the file.

5.4.8 Sine Wave

While not included in the distribution of DART or considered in the DART SHS parameter sweep tests, a pure sine wave file was created in order test for errors in the implementation of the SHS algorithm itself. A pure sine wave was generated from the Logic EXS24 (EXS24 synthesises a sine wave if no samples are loaded into it). A pure sine wave contains zero harmonics, and the DARTSinewave.wav file contained a loop of 0.5 second notes from C0 to C7. This was extremely useful when testing and validating the DART SHS algorithm.

5.4.9 Audio Input File Summary

To create the input files for the DART SHS algorithm, six audio input files were created, each containing samples from at least two sample libraries (three in the case of the Tubular



Output: st. 1
Voices: 0 Max: 1
Midi Ch: [A] 1
Memory: 7.47 MB
Tune 0.00
Cathedral
Cathedral A
+2 oct

Figure 5.11: The Kontakt 4 Sampler screen showing the GUI with the Wooden Tubular Bells patch

Bells). Each file contains the chromatic playback of every note within the sampled range of the instrument. Each instrument was played back at 4 differing MIDI input velocities (30, 60, 90 and 120) in order to trigger different samples on playback, more thoroughly testing the accuracy of the SHS algorithm.

Every file plays back all of the individual and distinct notes available at least twice, except for the Piano, due to the very large size if the input data (644 notes). This allows for any Round-Robin samples to be triggered, allowing the SHS algorithm the opportunity to analyse the variations in the notes played back.

Each file is made up of a total 672 0.5 second notes, giving a total length of 5 minutes 36 seconds for each file (59.4MB).

A sine wave file was created to test the DART SHS algorithm.

5.5 XtremWeb Configuration & DART Deployment

XtremWeb-HEP has been deployed at **LAL** (Laboratoire de l'Accélérateur Linéaire) in Paris, where local distributed resources are available to use by clients. XWHEP has also been deployed on the distributed GRID5000 platform (also) described in the background chapter, and can be bridged over a gLite based EGEE distributed european grid.

5.5.1 Installing and Configuring XtremWeb

During the first implementation stages of DART, there was no XtremWeb client application installer package. Xtremweb-HEP client 6.0 was the first version to provide installation packages for Mac OSX (used in implementing DART throughout this thesis), Linux, and Windows. Without an installer application, the XtremWeb client folder can be dragged into any directory on the user machine; the packaged OSX installer places the application in the directory: **`/Applications/xwhep.client/`**

Running the XtremWeb client requires creating and modifying the configuration file before a DART user can start to use XtremWeb to conduct their experiments. The client installation package creates a default configuration file containing all necessary configuration attributes, which must then be duplicated and stored in the OSX home directory. This default configuration file can be found in a sub-directory of the XtremWeb install, and a directory was created in the OSX home folder: **`${HOME}/.xtremweb`** directory (where **`${HOME}`** is the home directory[16], which hides the folder from view).

The default client configuration file provided within the client distribution was then copied to this folder. The main attributes of the configuration file that require modification are an XtremWeb login username and password, to register the client on the XtremWeb-HEP server via the server administrator. As soon as the administrator registers the XtremWeb client, they are provided with a login/password credential, which should be filled in the configuration file. The configuration file contains several attributes, however it is generally unnecessary to modify them.

If using an X509 certificate issued by a certificate authority belonging to IGTF[17] (`http://www.ngs.ac.uk` in the UK) installed in **`${HOME}/.globus/userkey.pem`** and

[16]The heading name must be '.xtremweb' and not 'xtremweb'

[17]The International Grid Trust Federation - `http://www.igtf.net/`

${HOME}/.globus/userkey.pem, then a login and password is not required (and should be removed from the configuration file). The X509 certificate and the creation of a gLite proxy are required in order to run jobs using the EGEE bridge, which is discussed in a following section.

As soon as the client package has been installed and configured with a username and password, it is possible to use the client tools to manage and use the XtremWeb-HEP infrastructure. Using the client, one can register users and applications, send data files, submit jobs for registered applications and retrieve job results. LAL XWHEP resources are automatically used unless GRID5000 or an X509 proxies have been created.

XWHEP does have a GUI implementation, but it was unreliable and did not allow the selection of multiple jobs or items, and so was not used. All of the use of XtremWeb was from the command line. The XtremWeb-HEP client package installs all possible XtremWeb commands in **${XWHEP_HOME}/bin**. In order to be able to access the commands from the command line the DART/XtremWeb user must enter an export command into the terminal, or store the command in the client **bash.rc** file in order to access the commands:

```
export PATH=$PATH:/Applications/xwhep.client/bin
```

The installation procedure became increasingly simpler during the implementation stages.

5.5.2 Running Jobs with XtremWeb

In order to submit the DART application to XtremWeb, the data required to successfully compute jobs - the DART JAR and audio input data - is prepared. This data may be stored in the XWHEP infrastructure or in any repository as soon as the data can be described by a URI and is accessible through the network.

In order for a DART user to submit data and jobs to XtremWeb they must:

- Register the DART application on the XWHEP Server
- Prepare jobs (units of work) containing the reference of a registered application, optional parameters, and optional references to additional files

- Submit the Application Data - the 6 input audio files
- Retrieve and keep URI's for the 6 audio files
- Submits jobs to the XWHEP Scheduler

The Workers contact a Scheduler to get jobs suitable for their architecture and in response, the Scheduler sends a suitable job description to a Worker. For each file referenced by the job which is not present in the local cache of the Worker yet, the Worker fetches the file from the XWHEP Data Repository or from an External Data Server. As soon as a job has finished on Worker side, the Worker contacts the Result Collector to send the results.

Registering an application is done using the command **`xwsendapp`**. This command requires four parameters:

- The application name
- The OS required by the application
- The CPU type required by the application
- The application (binary) file reference

For example, the command to submit an application with the name 'DART', providing the binary file for Mac OS X and Intel 64-Bit, which is stored in local disk space (in the directory **`/bin/DART`** is given by:

```
$> xwsendapp DART macosx x86_64 /bin/DART
xw://mac-89009.lal.in2p3.fr:4321/66baf2cc-f097-46ea-95b7-962a45593e8
```

The command then displays the URI of the application. Running a single DART job can be achieved by first pinging the XtremWeb server, to check if the software has been correctly installed. A simple command to view installed applications is:

```
$> xwapps
```

This returns a list of applications currently installed on the XtremWeb server:

```
UID='a87d6769-9694-46f7-9120-55b03da33095', NAME='DART'
UID='3df0653c-b3f0-439e-b5f6-c28c9bd306e3', NAME='bnbss'
UID='0cf21428-7c9a-42f3-a478-b684492cdbb9', NAME='cat'
```

```
UID='e3fdd9a2-367e-4d22-87c1-b201deec6b9d', NAME='test'
UID='afdfc5c6-12a7-4f54-9e34-e782f62d3991', NAME='VINA'
UID='8f582f85-100e-4505-8bd7-22dc0c5a6dfd', NAME='sgmuons'
UID='2e3ac23a-ed6b-4041-89bd-64af6a23be87', NAME='dsp'
UID='de7d2350-c0e1-4229-8615-1939340c4bd9', NAME='isdep'
```

To submit a DART job with a data that is available from a separate web server with the URL: **`http://www.myurl.com/DART/DARTTubBells.wav`**, the following command is used:

```
$> xwsubmit DART -infile DART/DARTTubBells.wav -outfile DART-10-1.txt -
    nofreqpoints 10 -noharmonics 1 -fft_window Hamming --xwenv http://www.
    myurl.com/DART/DARTTubBells.wav
```

It is also possible to run jobs with data that exists on the XWHEP Server. First the data must be sent to the server:

```
$> wget http://www.myurl.com/DART/DARTTubBells.wav
$> xwsenddata DARTTubBells.wav
xw://xwserv.lal.in2p3.fr/e0f5e724-52cd-4048-82ee-253db8e4db6e
```

In this example, the data is downloaded from a web server before uploading to XtremWeb. Once the data is sent to the XtremWeb server the data is reusable. After sending the data, a URI is given containing the location on the XtremWeb server: **`xw://xwserv.lal.in2p3.fr/e0f5e724-52cd-4048-82ee-253db8e4db6e`**. This allows the user to submit jobs that use that data using that new data URI:

```
$> xwsubmit DART -infile DARTTubBells.wav -outfile DART-10-1.txt -
    nofreqpoints 10 -noharmonics 1 -fft_window Hamming --xwenv xw://xwserv.
    lal.in2p3.fr/e0f5e724-52cd-4048-82ee-253db8e4db6e
xw://xwserv.lal.in2p3.fr/ff2d7889-53c6-48b8-86ea-87320fdccec8
```

After the job is submitted, a URI for the job itself is given: **`xw://xwserv.lal.in2p3.fr/ff2d7889-53c6-48b8-86ea-87320fdccec8`**. To check on and retrieve the status of the running job, the **`xwstatus`** command is used:

```
$> xwstatus xw://xwserv.lal.in2p3.fr/ff2d7889-53c6-48b8-86ea-87320fdccec8
<?xml version='1.0' encoding='UTF-8'?>
<get>
```

```
<work uid="ff2d7889-53c6-48b8-86ea-87320fdccec8" accessrights="0x755"
    isservice="false" isdeleted="false" appuid="a87d6769-9694-46f7-9120-55
    b03da33095" useruid="616d3ce1-534f-430d-9be8-3a0388527170" status="
    RUNNING" server="xwserv.lal.in2p3.fr" cmdline=" -infile DARTAudio.wav -
    outfile DART-10-1.txt -nofreqpoints 10 -noharmonics 1 -fft_window
    Hamming " dirinuri="xw://xwserv.lal.in2p3.fr/e0f5e724-52cd-4048-82ee-253
    db8e4db6e" arrivaldate="2009-08-04 17:42:08" sendtoclient="false" local
    ="true" active="true" replicated="false" />
</get>
```

Finally, the **`xwresults`** command is used to retrieve results as soon as the status is completed:

```
$> xwresults xw://xwserv.lal.in2p3.fr/ff2d7889-53c6-48b8-86ea-87320fdccec8
```

This downloads the results and deletes the job if its status returned as **`COMPLETED`**. Running the **`xwresults`** command without any arguments should download all results belonging to the XWHEP user, however should be used with caution as it can download and delete jobs and results from *all* applications due to a bug in XtremWeb. Therefore it is always recommended that a DART user should keep track of the job URI's and download results with a specific URI for a specific job.

5.5.3 Creating XtremWeb Experiment Scripts

In order to create the parameter sweep experiments described in this thesis, multiple jobs must be submitted to the XWHEP server in an automated fashion. Submitting multiple jobs is done using Perl scripts to iterate through the various command line arguments and SHS parameters, as given in the Design chapter, sending calls to work to the XWHEP Server.

The first DART experiment was simply the same set of parameters repeated 50 times by XtremWeb, which did not require a complicated script to repeat. The second stage of DART experiments prototyped the perl submission scripts; 160 experiments were conducted. The later stage included the creation of a script which iterated through the full 268,800 parameters.

5.5.4 DART 160 Runs Script

Perl was used as a simple scripting language to automate the execution of the 160 jobs. The following script was created in order to test the DART source code and check that the resulting output produced by the experiment was in the correct and usable format. The script creates a 2-parameter test run of the following scenario:

- The number of ***harmonics*** analysed will vary between 1-32
- The number of ***frequency candidate points*** will range from 10 to 50 in 10 point intervals (10, 20, 30, 40, 50)
- FFT Window type is **Hamming** (and will remain constant)
- Prototype run from a real parameter sweep application in audio pitch

Given a variation of 32 harmonics and 5 different frequency candidate point intervals, this requires 160 separate jobs. A single audio file is used used in order to reduce the number of variables being tested. Keeping the input data constant ensures that each system that runs DART will render identical results, and the feasibility of the DART platform as a whole (in terms of performance and practicality) can be investigated.

These experiments were run over the XtremWeb Desktop Grid at Laboratoire de l'Accélérateur Linéaire, France (LAL). This prototype was implemented to be easily scalable to perform more jobs by simply changing this script to create more jobs. The following script creates 160 command line execution commands:

```
#!/usr/bin/perl

$harmonics_max= 32;
$freqpoints_max = 50;

for ($i=10; $i <= $freqpoints_max; $i+=10) {
  for ($j=1; $j <= \$harmonics_max; $j++) {
      $command_line = `java -jar Dart.jar -infile DartAudio.wav -outfile
          DART-$i-$j.txt -nofreqpoints $i -noharmonics $j -fft_window
          Hamming`;
      print "\\n" . $command_line;
  }
}
```

Running this script at the command line will sequentially run the DART application 160 times on a single machine, and place the results in a 'results' directory folder. In order to convert this script to create the correct XtremWeb commands, the **`java -jar Dart.jar`** file execution command must be replaced with **`--xwsendwork DART`**, as described in the previous section. The URI of the data must be placed at the end of the command line argument, such as: **`--xwenv xw://xwserv.lal.in2p3.fr/87db1e26-d7f8-4c72-982d-54fdfc8fddbb`**. This creates 160 command line arguments that are in the correct format to successfully send 160 jobs to the XtremWeb Server. This is executed from the OSX terminal.

The results were retrieved by running the **`xwresults`** command, downloading all 160 results and deleting them from the worker machines. XtremWeb considers all created and/or modified files as results and so the results are contained in a compressed zip file. Each results file also contains a terminal output from each computer, included in the zip file. Although this information is not necessary, it could prove useful in the event of an error. Results are stored in the current working directory.

5.5.5 DART Main XtremWeb Run - 268,800 Jobs

The script presented in this section is for the implementation of the full DART parameter sweep experiment, which requires 268,800 jobs. The following parameters are iterated through:

- **Top Frequency Points: Vary from 1-50**. This argument adjusts the number of top frequency peaks looked into by the PitchDetection algorithm
- **Number of Harmonics analysed: Vary from 1-32** (5 Octaves). This adjusts the number of harmonics that are summed up from the fundamental
- **FFT Window type: Vary 1-28**. There are 28 different FFT windows available (such as Hamming, Hanning, Gaussian, etc) in the FFT code in the pitch detection algorithm
- **Audio input file: Vary from 1-6**

Varying these parameters and considering all combinations gives a total of 50 x 32 x 28 = *44,800* jobs for each piece of input source data. Using 6 different audio files gives 44,800 x 6 = **268,800 total jobs**. In contrast to the simpler perl script above, the various file names, data URI's and FFT Window types are explicitly defined as variables.

```
#!/usr/bin/perl

@infilename = ("DARTAcousticG.wav", "DARTOboe.wav", "DARTViolin.wav", "
    DARTPiano.wav", "DARTTubBells.wav", "DARTDistortG.wav");
@infileURI = ("xw://xwserv.lal.in2p3.fr/f10515ad-4d3c-4822-b61f-e77002cf6621
    ", "xw://xwserv.lal.in2p3.fr/c5960cf5-19a1-463a-bc68-c51732d18e1b", "xw
    ://xwserv.lal.in2p3.fr/87db1e26-d7f8-4c72-982d-54fdfc8fddbb", "xw://
    xwserv.lal.in2p3.fr/3af81264-8e5c-412f-a177-d5a80e7688f7", "xw://xwserv.
    lal.in2p3.fr/182c2bb4-2e54-44e4-bf2c-71e42182004a", "xw://xwserv.lal.
    in2p3.fr/36815831-1168-49d2-9428-f1c5a14d3465");
$infilelength = $#infilename + 1;

$freqpoints_max = 50;
$harmonics_max= 32;

@windowtype = ("Rectangle", "Bartlett", "Blackman", "Gaussian", "Hamming", "
    Hann(Hanning)", "Welch", "BlackmanHarris92", "Nuttall3", "Nuttall3a", "
    Nuttall3b", "Nuttall4", "Kaiser3", "Kaiser4", "Kaiser5", "Kaiser6", "
    Kaiser7", "SFT3F", "SFT4F", "SFT5F", "SFT3M", "SFT4M", "SFT5M", "FTNI",
    "FTHP", "FTSRS", "HFT70", "HFT95");
$windowlength = $#windowtype + 1;

#print "Size of infilelength: $infilelength \\n";
#print "Size of windowlength: $windowlength \\n";

for ($h=0; $h < $infilelength; $h++) {
  for ($i=1; $i <= $freqpoints_max; $i++) {
    for ($j=1; $j <= $harmonics_max; $j++) {
      for($k=0; $k < $windowlength; $k++) {
        $infilenumber = $h+1;
        $windownumber = $k+1;
                      $command_line ="--xwsendwork DART -infile @infilename[
                          $h] -outfile DART-$infilenumber-$i-$j-
                          $windownumber.txt -nofreqpoints $i -noharmonics $j
                           -fft_window @windowtype[$k] --xwenv @infileURI[$h
                          ]";
        print "\n".$command_line;
      }
    }
  }
}
```

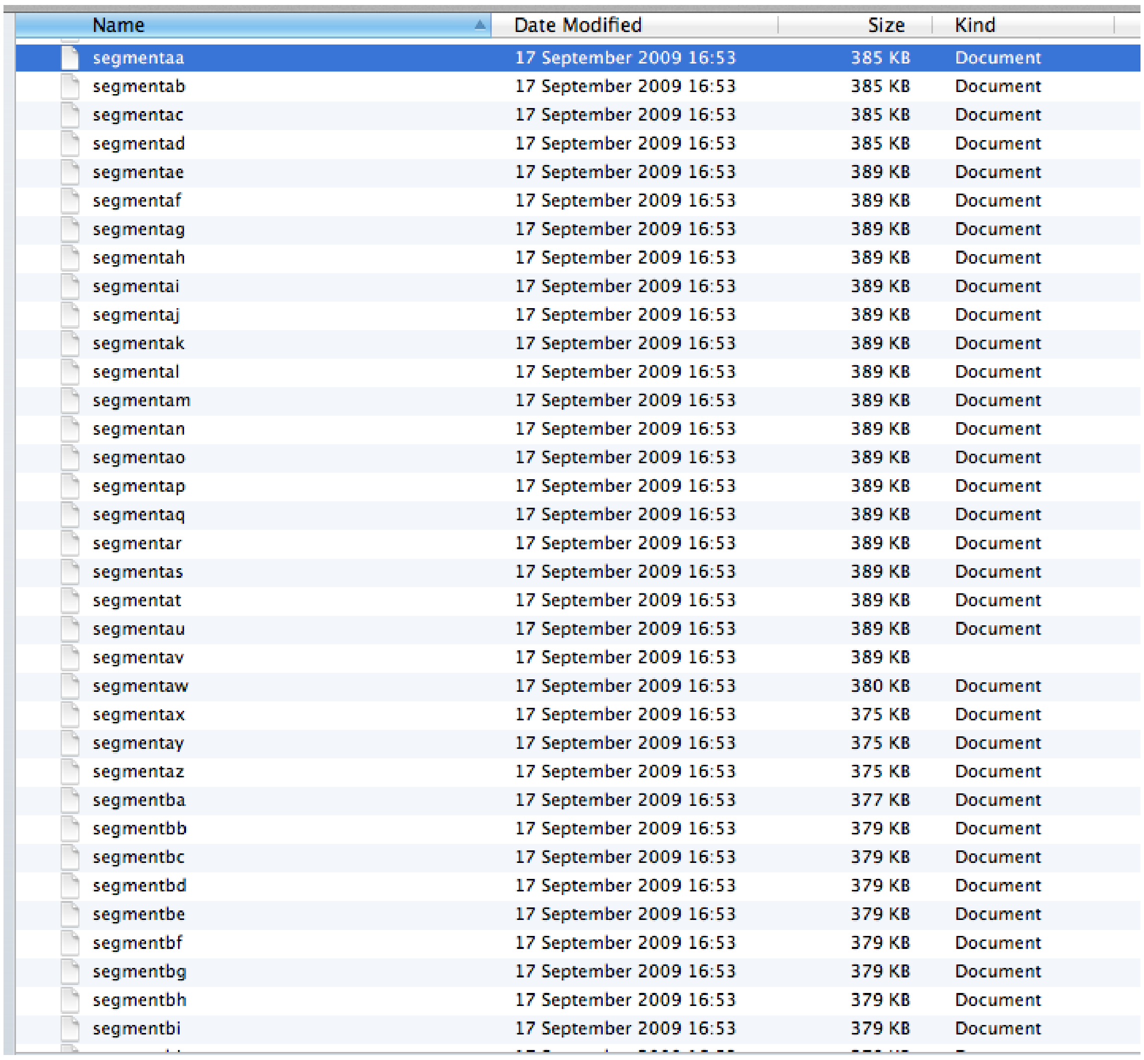
The **infilename** and **infileURI** variables are set at the top of the script, and the variables for each parameter are iterated through, dynamically create the results file name at the same time.

Submitting all 268,800 jobs to the LAL XtremWeb Server from the terminal simultaneously was not a feasible option. When attempted, it would eventually lead to either overloading the XtremWeb server, or running out of RAM on the client machine (overloading the Java Virtual Machine), due to the implementation of the XtremWeb client. Several different approaches were tried to address this.

A DART *macro* file was created by printing out the commands to the terminal, and then pasting the commands into a file called **dart.macro**. This macro file can then be split up into segments of n command line arguments, and submitted in sequence. The macro was split up into 136 smaller macro files, each consisting of 2000 lines, with the final segment consisting of 800 DART command line arguments. The **dart.macro** file can be split up using the Unix **split** command:

```
split -l 2000 dart.macro segment
```

This will create the 136 files, with the file names **segmentaa, segmentab, segmentac** etc as shown in Figure 5.12.



Name	Date Modified	Size	Kind
segmentaa	17 September 2009 16:53	385 KB	Document
segmentab	17 September 2009 16:53	385 KB	Document
segmentac	17 September 2009 16:53	385 KB	Document
segmentad	17 September 2009 16:53	385 KB	Document
segmentae	17 September 2009 16:53	389 KB	Document
segmentaf	17 September 2009 16:53	389 KB	Document
segmentag	17 September 2009 16:53	389 KB	Document
segmentah	17 September 2009 16:53	389 KB	Document
segmentai	17 September 2009 16:53	389 KB	Document
segmentaj	17 September 2009 16:53	389 KB	Document
segmentak	17 September 2009 16:53	389 KB	Document
segmental	17 September 2009 16:53	389 KB	Document
segmentam	17 September 2009 16:53	389 KB	Document
segmentan	17 September 2009 16:53	389 KB	Document
segmentao	17 September 2009 16:53	389 KB	Document
segmentap	17 September 2009 16:53	389 KB	Document
segmentaq	17 September 2009 16:53	389 KB	Document
segmentar	17 September 2009 16:53	389 KB	Document
segmentas	17 September 2009 16:53	389 KB	Document
segmentat	17 September 2009 16:53	389 KB	Document
segmentau	17 September 2009 16:53	389 KB	Document
segmentav	17 September 2009 16:53	389 KB	
segmentaw	17 September 2009 16:53	380 KB	Document
segmentax	17 September 2009 16:53	375 KB	Document
segmentay	17 September 2009 16:53	375 KB	Document
segmentaz	17 September 2009 16:53	375 KB	Document
segmentba	17 September 2009 16:53	377 KB	Document
segmentbb	17 September 2009 16:53	379 KB	Document
segmentbc	17 September 2009 16:53	379 KB	Document
segmentbd	17 September 2009 16:53	379 KB	Document
segmentbe	17 September 2009 16:53	379 KB	Document
segmentbf	17 September 2009 16:53	379 KB	Document
segmentbg	17 September 2009 16:53	379 KB	Document
segmentbh	17 September 2009 16:53	379 KB	Document
segmentbi	17 September 2009 16:53	379 KB	Document

Figure 5.12: Segments of the dart.macro file

Once the file is split into 136 segments, then a script called **`dartexec.pl`** can be run in order to execute them sequentially:

```
#!/usr/bin/perl

opendir(DIR, ".");
@FILES= readdir(DIR);
#@FILES = grep(/*segment*$/,readdir(DIR));

foreach $file (@FILES) {
  if ($file =~ m/^segment/) {
    $command_line = `./xtremweb.client --xwmacro $file > dart.uris`;
    $clearcache = `./xwclean`;
    print $command_line . "\n";
```

```
      print $clearcache . "\n";
    }
  }
```

This script will go through the current directory and find files with filenames beginning with **`segment`**. The script then calls **`xtremweb.client`**, adds the **`--xwmacro`** argument, and loops through all segment files, submitting the jobs to the XtremWeb server.

5.5.6 Submitting 100 Jobs at a time

The following script was created to submit 100 jobs at a time. Various iterations of the submission script were created until XtremWeb version 6.0, where the client software included and utilised the script by default when DART (or any XWHEP) client/users submit a macro containing more than 100 jobs. The main advantages to this script were that no file splitting was required, one simple macro file can be created containing all necessary job submission commands. Another advantage is that the job URIs are automatically stored in a separate file.

A key feature however, is that after submitting 100 jobs the script *sleeps* for 30 seconds, giving the XtremWeb server some recovery time; however, this increases the overall job submission time. Given an expected submission time of 1 second per job, it will take 100 seconds to submit 100 jobs. Theoretically, 286,800 jobs would be submitted in 74 hours 40 mins, given 1 second per submission. The additional 30 seconds wait would occur 2688 times, adding around 22 hours, 24 minutes to the total run, giving a total of **96.1 hours to submit all 268,800 jobs**.

```
  #!/bin/sh
  set -x

  # where are XWHEP binaries
  BINDIR="/Applications/xwhep.client/5.7.5/bin"

  # where is the macro to use
  MACRO="/Applications/xwhep.client/5.7.5/bin/dart.macro"

  # job URIS are stored in a file
  URIS=dartjoburis.txt
```

```
# jobs submission thanks to $MACRO. output stored in $URIS
$BINDIR/xtremweb.client --xwmacro $MACRO > $URIS

# verbose output is processed so we only have to deal with jobs URIS.
TMP1=`grep -E "^xw:" "$URIS"`
TMP1=`echo "$TMP1" | tr "n" " "`

# infinite loop checks all jobs are completed. When true, reinsert jobs
    using same $MACRO
while [ -n "$TMP1" ]
do
  TMP2=`$BINDIR/xwworks $TMP1`
  echo "$TMP2" | grep -E "WAITING|PENDING|RUNNING" > /dev/null
  if [ $? == 1 ]
  then
    $BINDIR/xwrm $TMP1
    $BINDIR/xtremweb.client --xwmacro $MACRO > $URIS
    TMP1=`grep -E "^xw:" "$URIS"`
    TMP1=`echo "$TMP1" | tr "\n" " "`
  fi
  sleep 30
done
```

A more complex and complete version of this script is now included as standard as part of the XWHEP client software.

5.5.7 Retrieving XtremWeb Results

Once all the jobs were submitted, it was then necessary to retrieve all the available results. It was especially important to keep the URI of each job, as the catchall command of **xwresults** - originally designed on retrieve all results from the user issuing command - was shown to occasionally (erronously) download and remove jobs from *all* XtremWeb users. In earlier experiments it was possible to store all the DART job URIs by copying and pasting the terminal commands into a new document, and removing every other line using the Unix AWK command:

```
awk 'NR\%2 != 0' darturis.txt
```

This left only the job URI's. Later on a command was added to allow the macro to redirect the output from the console terminal, to a file:

```
xwsubmit --xwmacro dart.macro > joburis
```

This stores all job URIs in a file called **joburis**. Note however, that it may still be necessary to remove every other line, as this simply redirects the terminal messages. With the **onehundred.sh** script - later integrated into the XWHEP client software - **dartjoburis.txt** is automatically created, which contains all the URIs for the submitted jobs.

After all of the submitted jobs are complete, the **dartgetresults.pl** file was used to retrieve the results, using the **--xwresults** command shown in the script below:

```
open (FILE, 'darturis.txt');
foreach $line (<FILE>) {
  chomp($line);
  $command_line ="--xwresults $line";
  print "\n" . $command_line;
}
```

With larger numbers of results, it became apparent that as with sending jobs, the XtremWeb server would be placed under too much strain if 268,800 result files are called at once. Similarly to the submission procedure, the **joburi.txt** document was split into smaller segments, and a newer script which sleeps for 60 seconds after sending the **xwresults** command for each segment, was used to retrieve the results:

```
#!/bin/sh
echo "start"

cd final1

for file in `ls *`
  do
  xwresults --xwmacro $file
  sleep 60
done
echo "end"
```

5.5.8 Using the EDGeS Infrastructure

XtremWeb is capable of *bridging* to Grids, as published in [114]. XtremWeb can delegate computing to EGEE resources, using 'Pilot Jobs'. The bridge is external to XtremWeb and uses the XtremWeb client to check for pending jobs for all known XtremWeb servers and the EGEE facility to submit these jobs to this infrastructure. The bridge requires:

- EGEE middleware client (gLite)
- XtremWeb middleware client
- Administrator access to XtremWeb Servers: a configuration file & associated *public key*

In order to use the bridge, XtremWeb users must *have an X509 certificate*, be *registered in a Virtual Organisation* and *submit their Jobs with a VOMS*[18] proxy. In order to ascertain a certificate a DART/XWHEP user must:

1. Request a certificate from the NGS[19] using the Mozilla Firefox web browser
2. NGS will ask for validation at the local Registration Authority (RA) at Cardiff University (or any DART user's organisation)
3. Provide the RA with identification (passport or driving licence) needed to validate the request
4. Once validated, an email with a link to the certificate will be sent
5. Open the link with the same browser used in point 1 (the certificate will be saved in the web browser)
6. Export the certificate to the client machine's file system (a Firefox feature)

This creates a **`p12`** certificate, which store X509 private keys with accompanying public key certificates, protected with a password-based symmetric key. However the **`p12`** certificate (**`PKCS12`** format) is not accepted by the EGEE security infrastructure, but can be converted into the supported standard (PEM) by splitting the file into **`usercert.pem`** and **`userkeys.pem`** files.

In order to gain permission to use the EGEE grid, the user must belong to a Virtual Organisation (VO). The **`usercert.pem`** file must be sent to a member of the Virtual Organisation with the permission and access to insert the VO. In the case of DART and

[18]`http://www.globus.org/grid_software/security/voms.php`
[19]`http://www.ngs.ac.uk/certpersonal`

the EGEE grid, this must to be done by the EDGES VOMS admin - Zoltan Balaton from MTA SZTAKI[20] at The Computer and Automation Research Institute (The Hungarian Academy of the Sciences). Sending the **`usercert.pem`** file to the VOMS admin allows them to insert the VO.

5.5.8.1 Installing jLite and Creating a Proxy

gLite can be more easily integrated into a Java framework (such as DART) by using *jLite*[21]. jLite is a Java library developed by Oleg Sukhoroslov that provides a simple API for accessing gLite based grids. The gLite middleware can be difficult to use, especially in a Java environment and jLite helps to reduce the time and effort needed to build a cross-platform grid application on top of the EGEE grid infrastructure.

jLite versions 0.1.1 to 0.2 were available to during the implementation stages of this thesis. The library can be downloaded installed in any user directory. Once the directory is downloaded, the latest list of certificates from the Certificate Authorities at **`http://www.xtremweb-hep.org/lal/cacerts.zip`** must be downloaded, by replacing the **`etc`** folder in the subdirectory of the 'jlite' installation folder. This list of certificates is update every night, and must be regularly updated.

A proxy can be created by using the **`proxy-init.sh`** script in the jLite installation. Navigating to the current directory of **`jlite/bin/`** and entering the following creates a proxy for 12 hours:

```
$./proxy-init.sh demo.vo.edges-grid.eu
Enter your private key passphrase:  *********
Created VOMS proxy: C=UK,O=eScience,OU=Cardiff,L=WeSC,CN=ahmad al-shakarchi,
    CN=proxy
Proxy is valid until: Fri Jun 3 04:15:00 2011 BST
Proxy location: /var/folders/Tk/Tk9+dUqzHa4aYeI40e2RM++++TI/-Tmp-/
    x509up_u_eddie
```

After 12 hours the proxy must be renewed, otherwise jobs will not use the XWHEP to EGEE bridge. It is also possible to create a 24 hour proxy by using: **`proxy-init -valid 24:00 demo.vo.edges-grid.eu`**. A common error is an **`AC Validation Failure`**:

[20] http://www.sztaki.hu/?en

[21] http://code.google.com/p/jlite/

```
org.glite.voms.contact.VOMSException: AC validation failed!
```

This can be caused by missing VOMS server certificate. This can often be remedied by updating the list of CA Certificates by downloading the latest **`cacerts.zip`** file from the XWHEP website and replacing the etc folder in the jLite directory.

```
$ export X509_USER_PROXY= /var/folders/Tk/Tk9+dUqzHa4aYeI40e2RM++++TI/-Tmp-/
    x509up_u_eddie
```

In order to use XWHEP with the X509 proxy instead of the standard username and password combination, the XWHEP config file must be modified. The attribute **`X509_USER_PROXY=`** must be set to: **`/var/folders/Tk/Tk9+dUqzHa4aYeI40e2RM++++TI/-Tmp-/x509up_u_eddie`**. Once this has been done, jobs sent to the XtremWeb server can be executed using the XWHEP to EGEE bridge.

5.6 BOINC Configuration & DART Deployment

A BOINC project is a group of one or more distributed applications, run by a single organisation, that uses the BOINC platform. Projects are independent; each one has its own applications, databases and servers, and is not affected by the status of other projects. Creating a working DART BOINC project was not a trivial task and required assistance in implementation. A BOINC project consists of the following components:

- **A MySQL database**
- **A directory structure**
- **A configuration file**

The configuration file specifies options, daemons, and periodic tasks. The simplest method to create a project is to use the **`make_project`** script, which creates skeletal versions of the above components. This creates a project with a master URL **`http://HOSTNAME/cplan/`** and whose directory structure is rooted at **`$HOME/projects/cplan`**. The master URL is used to identify each BOINC project. For DART, the project URL was **`http://mdesk001.cs.cf.ac.uk/dart/`** and is the home page of the project; when viewed in a browser it describes the project and contains links for registering and for downloading the core client.

To create a BOINC project, a DART researcher must do the following:

- Download the BOINC source code
- Compile the source
- Run the make_project script
- Set up a BOINC Server
- Add the DART MIR application
- Create workunits
- Start the project
- Activate the project for participants

The creation of a BOINC Server consists of a project back-end that supplies the DART application and workunits and receives the results, as well as a BOINC server *complex* that manages data distribution and collection. The BOINC server complex includes the following components:

- **Scheduling server(s)** that communicate with participant hosts
- **A relational MySQL database** that stores information about work, results, and participants
- **Utility programs and libraries** that allow the project back end to interact with the server complex
- **Web interfaces** for participants and developers
- **Data servers** that distribute input files and collect output files

The BOINC platform also provides a BOINC Server VM which automatically sets up a BOINC server, as documented in [115]. This Virtual Machine has all the BOINC prerequisites installed, the BOINC software installed and compiled, and user accounts and permissions already set up.

5.6.1 Setting Up A BOINC Server

This section covers the set up of a BOINC server, using the BOINC Virtual Machine supplied by the BOINC team (running Debian). The BOINC client (workers) attach to this server in order to retrieve work, track progress. The BOINC server was set up and deployed on the Cardiff University 'Sintero' Server cluster.

The first step in creating a BOINC server is to download the and install all the standard BOINC sever dependancies:

```
$ sudo aptitude -y install build-essential subversion m4 \
  autoconf automake gcc-4.1 g++-4.1 gcc pkg-config libtool \
  apache2-mpm-prefork libapache2-mod-php5 mysql-client-5.1 \
  mysql-server-5.1 php5-mysql php5-gd phpmyadmin\
  python-mysqldb libmysql++-dev libssl-dev \
  libapache2-mod-proxy-html

$ sudo a2enmod rewrite
$ sudo service apache2 restart
```

The **a2enmod rewrite** command enables the rewrite module for Apache before then we restart it. MySQL also has to be configured before creating a BOINC project.

```
$  mysql -u root -p
<enter password>

>GRANT ALL ON *.* TO <<mysql user>>@localhost \
IDENTIFIED BY '<<mysql password>>;

>GRANT ALL ON *.* TO <<mysql user>> IDENTIFIED \
BY '<<mysql password>>;

>quit
```

The next step is to use Subversion to check out the BOINC server source code and compile it.

```
$ svn co \ http://boinc.berkeley.edu/svn/trunk/boinc boinc_server

$ cd boinc_server

$ ./_autosetup
$ ./configure --disable-client
$ make
```

The **autosetup** and **configure disable-client** commands are used to disable the building of the BOINC client and force the server to be built, before the BOINC server side software is built.

The next stage is to actually create the DART BOINC Project. **The make_project** script (located in the tools folder in the standard directory structure) is used to create the BOINC Project:

```
$ ./tools/make_project --user_name=boincadm \

--db_user <<mysql user>> \
--db_passwd '<<mysql passwd>>' \

dart

'Eddie's Dart Project'
```

```
$ cd ../projects/dart
$ su boinc
$ sudo chmod 500 keys
$ sudo su -c "cat dart.httpd.conf >> /etc/apache2/httpd.conf"
$ sudo service apache2 restart
$ exit
```

The Make Project Script creates a project with master URL **`http://<hostname>/dart/`**, with a directory structure rooted at **`$HOME/projects/dart`**.

As highlighted earlier, the DART project URL was **`http://mdesk001.cs.cf.ac.uk/dart/`**. More specifically, **`make_project`** does the following:

- Creates the Project Directory and Subdirectories
- Creates and initialises the MySQL BOINC Database
- Copies the source and executable files
- Generates the DART config file

The script also generates a template Apache configuration file that can be inserted into **`/etc/apache/dart.httpd.conf`**, and also generates a crontab line.

The 300MB BOINC *work-package* containing the DART application and the 6 audio input files is only downloaded once by each client machine, and the work units and results files (both only 1-2KB in size) are passed back and forth to the BOINC server. Multiple work units are downloaded to each worker machine each more work is required (no new audio data is sent with the work units unless the experiment has changed), therefore after the initial download, each system should have a relatively low network activity, with just a few kilobytes of data being transferred back and forth between each worker and the BOINC server.

The BOINC team provide guides to aid the different aspects of creating a project, such as the BOINC project creation 'Cook Book' [107], the BOINC Server Intro [108] and a large master PDF document simply entitled 'Creating BOINC Projects' [109]. These documents contain the vast majority of information required for the implementation of a generic BOINC project. These guides were used to set up a BOINC server and cross platform versions of the DART application were created before assistance was required to fully configure BOINC.

5.6.2 Joining the DART BOINC Project

A **workunit** describes a computation to be performed. It is represented by a row in the workunit database table. BOINC provides a utility program and C function for creating workunits. To create a workunit an XML template file is required, describing the workunit and its corresponding results. Generally the same templates will be used for a large number work of workunits. The audio input file(s) must then be placed in the download directory, and a BOINC function or script is invoked to create a database record for the workunit. BOINC can then create the results for the workunit, distribute them to client hosts, collect the output files, find a *canonical*[22] result, assimilates the result, and delete files that are not required.

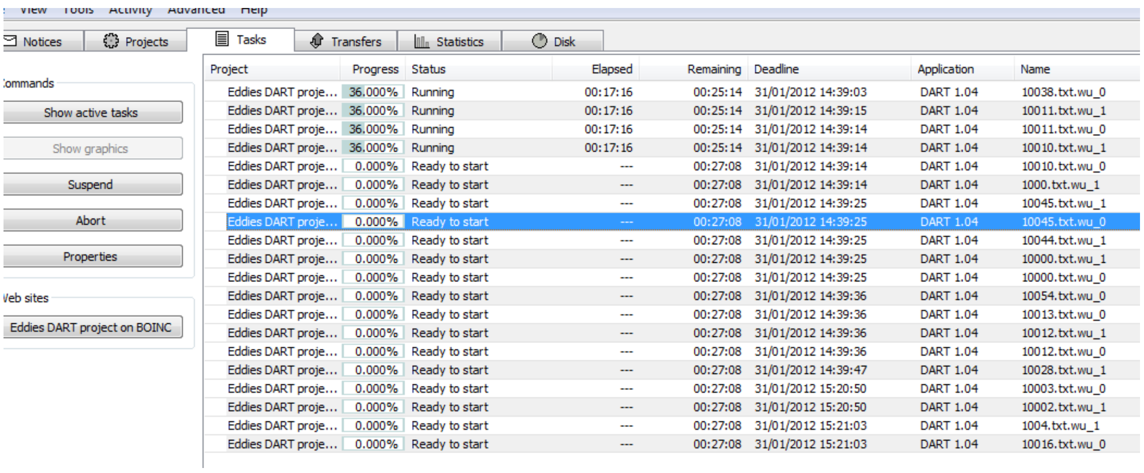
The DART BOINC project is configured to send more work units to worker machines that have more processors and processing capability. For example, an 12-Core Mac Pro based worker will download many times more work units to process in comparison to an older single or dual core machine (BOINC uses its own benchmarks to measure the speed and overall performance of the worker machine). The default amount of work downloaded by a client is also user-configurable.

In DART, the workunits were created using a script containing all 268,800 parameter variables, as described and outlined in [116]. Each DART workunit contains 20 jobs, enabling less frequent contact with the BOINC server.

The BOINC client attaches to the server in order to retrieve work, send results, and track progress. The BOINC Manager client software as seen in Figure 5.13 shows the user the workunits that are currently being processed.

For a user to participate in the DART project, the following steps were required:

[22]The Canonical Result is considered to be the model for all of the other Results for the work units that have not yet been returned.



Project	Progress	Status	Elapsed	Remaining	Deadline	Application	Name
Eddies DART proje...	36.000%	Running	00:17:16	00:25:14	31/01/2012 14:39:03	DART 1.04	10038.txt.wu_0
Eddies DART proje...	36.000%	Running	00:17:16	00:25:14	31/01/2012 14:39:15	DART 1.04	10011.txt.wu_1
Eddies DART proje...	36.000%	Running	00:17:16	00:25:14	31/01/2012 14:39:14	DART 1.04	10011.txt.wu_0
Eddies DART proje...	36.000%	Running	00:17:16	00:25:14	31/01/2012 14:39:14	DART 1.04	10010.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:14	DART 1.04	10010.txt.wu_0
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:14	DART 1.04	1000.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:25	DART 1.04	10045.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:25	DART 1.04	10045.txt.wu_0
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:25	DART 1.04	10044.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:25	DART 1.04	10000.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:25	DART 1.04	10000.txt.wu_0
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:36	DART 1.04	10054.txt.wu_0
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:36	DART 1.04	10013.txt.wu_0
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:36	DART 1.04	10012.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:36	DART 1.04	10012.txt.wu_0
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 14:39:47	DART 1.04	10028.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 15:20:50	DART 1.04	10003.txt.wu_0
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 15:20:50	DART 1.04	10002.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 15:21:03	DART 1.04	1004.txt.wu_1
Eddies DART proje...	0.000%	Ready to start	---	00:27:08	31/01/2012 15:21:03	DART 1.04	10016.txt.wu_0

Figure 5.13: The BOINC client screen for Windows 7. This shows a client machine running one work unit for each CPU core available on the machine. More work units are ready to run when the four currently running work units are complete.

1. Download and install the BOINC Manager from `http://boinc.berkeley.edu/download.php`
2. Once downloaded select 'Attach to Project'
3. When prompted, enter http://mdesk001.cs.cf.ac.uk/dart/ (ignoring the list of other BOINC projects)

The BOINC Manager software would then download the DART bundle and processing would start once the DART bundle was downloaded onto the worker's computer. The user can then specify how much of their computer's resources they allow BOINC/DART to use, which times it can be used, and so on. BOINC is very customisable which helps to gain participation from users who otherwise may worry about the software pulling resources from their machine when required.

The BOINC Manager software is available for all major operating systems, but DART for BOINC was able to run on Windows 32/64-Bit and Linux32/64-Bit. Mac OSX and Windows XP compatibility was more difficult to achieve and was not functioning during the experiments.

5.6.3 Creating Different Platform Applications

When building a BOINC application, several versions for differing platforms are required (Windows, Mac OS X, Linux) in order to run on the widest range of volunteer computers possible. Each operating system also requires individual 32- and 64-Bit versions of the application in order to fully maximise the potential for user participation.

DART is a Java application and is not supported by BOINC natively. BOINC does include some guides to run a Java application with BOINC and includes a wrapper[23] as found in [110], however this wrapper is not very flexible. It is partially configurable but can only be used to execute a list of executables (tasks) one after the other as the XML file it uses only allows describing the order of execution of the binaries. To make the wrapper more flexible this configuration file could be extended with new features to provide a required level of flexibility each time a shortcoming is discovered, but ultimately a general solution would require a generic scripting language for describing all possible configuration options.

GenWrapper[24] is a generic BOINC wrapper for legacy applications, using shell scripting and built-in commands like **`tar`**, **`awk`**, **`sed`**, **`zip`**, etc. to control and execute a 'legacy' Desktop Grid application, stating how the application is to be run and how the BOINC workunits should be processed. GenWrapper offers a solution for wrapping and executing an arbitrary set of legacy applications in a BOINC infrastructure. GenWrapper is available for Mac OS X, Microsoft Windows and Linux, and is extremely lightweight (around 600Kb). [111] gives an excellent overview of the software.

While DART is not a legacy application itself, it is a Java application and porting to run on BOINC natively would essentially call for a complete rewrite of the application in C/C++, which breaks the DART and Triana structure. GenWrapper allows applications like DART to run without 'BOINCification' by making the BOINC API available in **`POSIX`** (Portable Operating System Interface) shell scripting. This is realised by utilising an extended version of BusyBox[25], to provide the most common UNIX commands and a **`POSIX`** shell interpreter in a single executable with an applet (BusyBox extension) to make BOINC API functions accessible from the shell on Windows, Linux and Mac OS X platforms. A shell interpreter is started instead of the real application that executes a DART application script, which realises 'BOINCification' through script commands and is capable of performing any action that is possible to with a shell script.

[23]`http://boinc.berkeley.edu/trac/wiki/WrapperApp`
[24]`http://genwrapper.sourceforge.net/`
[25]`http://www.busybox.net/`

GenWrapper was used to create cross platform versions of DART; Windows 32/64 bit, Linux, and Mac OS X. The GenWrapper based DART application was then bundled along with a Java VM for each client machine except OSX (as OSX already contains Java) and the **`boinc_zip`** library. However there were bugs with the implementation that the GenWrapper developer could not rectify in time, and as such there were numerous problems creating a Mac-compatible BOINC project.

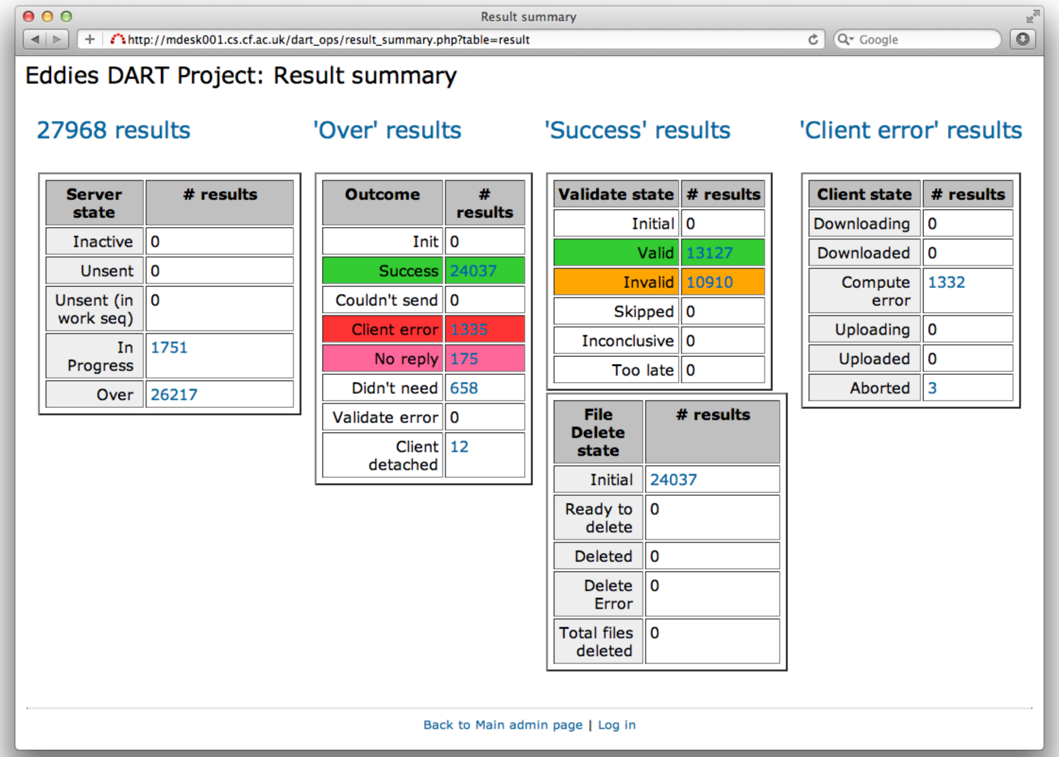
5.6.4 Validating Work

The BOINC validation process compares redundant results and decides which ones are to be considered correct, and also decides how much credit to grant to each correct result. A BOINC validator is a background (daemon) process. As DART generates exactly matching results, it is possible to use the provided BOINC **`sample_bitwise_validator`** validator. Each DART work unit is processed twice, in order to perform a bit-wise comparison of the results. The **`sample_bitwise_validator`** requires a strict majority, and regards results as equivalent only if they agree byte for byte.

The results can be monitored and summarised on the DART Admin page, as shown in Figure 5.14.

Validation works by only inspecting workunits where the **`NEED_VALIDATE`** flag is set. The Validator in contrast to the *Transitioner* daemon has also 2 project-specific functions **`check_set`** and **`check_pair`**, this means that different BOINC projects don't necessarily need to act the same way. When a workunit is successfully validated the remaining results are then compared against the *Canonical Result* if they are 'weakly similar' and are marked valid/invalid accordingly. At the end of the validation process, the Validator chooses the amount of Credit to be awarded by checking:

- If only **2** results passed validation then use the lowest Claimed Credit
- If **3 or more** results passed validation, remove highest and lowest Claimed Credit and average the rest


Result summary

http://mdesk001.cs.cf.ac.uk/dart_ops/result_summary.php?table=result

Eddies DART Project: Result summary

27968 results

Server state	# results
Inactive	0
Unsent	0
Unsent (in work seq)	0
In Progress	1751
Over	26217

'Over' results

Outcome	# results
Init	0
Success	24037
Couldn't send	0
Client error	1335
No reply	175
Didn't need	658
Validate error	0
Client detached	12

'Success' results

Validate state	# results
Initial	0
Valid	13127
Invalid	10910
Skipped	0
Inconclusive	0
Too late	0

File Delete state	# results
Initial	24037
Ready to delete	0
Deleted	0
Delete Error	0
Total files deleted	0

'Client error' results

Client state	# results
Downloading	0
Downloaded	0
Compute error	1332
Uploading	0
Uploaded	0
Aborted	3

Back to Main admin page | Log in

Figure 5.14: The BOINC admin page allows administrators to see a summary of the progress and the results of the BOINC project. This screen grab shows the results summary page after all 27968 work units have been sent out.

5.6.5 Monitoring The Progress

BOINC's administrative web interface (shown in Figure 7.55) provides an interface to allow a BOINC administrator to:

- **Browse the entire BOINC database**
- **Create user accounts**
- **Create and view user profits and recent results**
- **Screen user profiles**
- **View results**
- **Browse statistic charts and log files**

BOINC gives credits to each host and user, as seen when using the BOINC admin site in Figure 5.16. These credits give a broad overview of the performance of the host or the contribution of a particular user (or team). Two types of credits are given:

- **Total Credit Score:** The total number of Cobblestones[26] performed and granted.
- **Recent Average Credit Score:** The average number of Cobblestones per day granted *recently.* This average decreases by a factor of two every week.

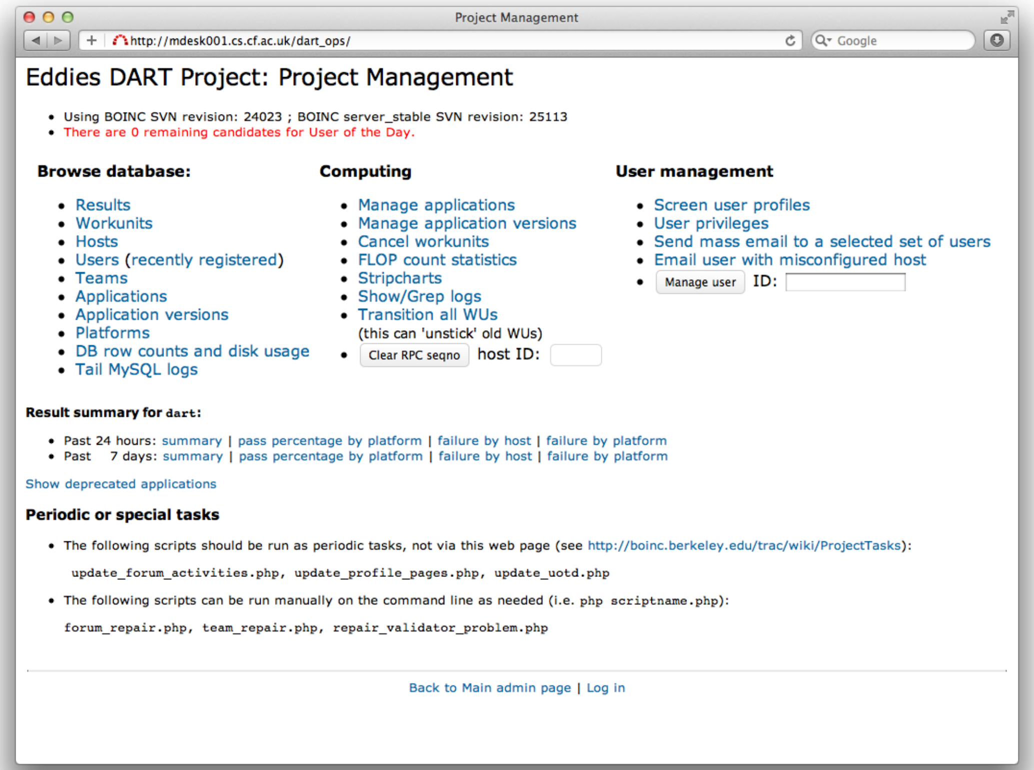
A credit is given when a valid work unit is returned by a host/worker. If a computer processes and returns a work unit it does not automatically receive a credit - it must first have that work unit validated by the project specific method - in the case of DART, each work unit was processed twice and compared with a bitwise comparison. Once validated, the host is granted a credit. This amount is immediately added to the host or user total.

BOINC uses benchmarks to measure the speed of a system and - in combination with the amount of time it required for a work unit to process can estimate at the amount of credit it should receive. Since systems have many variables including the amount of RAM, the processor speed, and specific architectures of different motherboards and CPUs, there can be wide discrepancies in the number of credits that different hosts gains when processing each work unit.

The Recent Average Credit (RAC) score is designed to give a rough estimate of the number of credits a computer, user, and team will accumulate on an average day. Additionally the RAC score is independent of computers, users, and teams, meaning that they do not simply accumulate. RAC was originally meant to help scientists understand the computational power available to them and to increase competition among users by allowing even new users to quickly move up in rank based on RAC, which should directly reflect how fast work is being processed. However the ratings often encourage more user participation with 'healthy competition'. More information on the BOINC credit system can be found in [117] and [118].

Participation with the DART/BOINC project was encouraged from the Cardiff University Computer Science department and various social networking websites.

[26]The basic unit of the BOINC credit system is the *cobblestone*, a benchmark figure named after Jeff Cobb of SETI@home



Eddies DART Project: Project Management

- Using BOINC SVN revision: 24023 ; BOINC server_stable SVN revision: 25113
- There are 0 remaining candidates for User of the Day.

Browse database:

- Results
- Workunits
- Hosts
- Users (recently registered)
- Teams
- Applications
- Application versions
- Platforms
- DB row counts and disk usage
- Tail MySQL logs

Computing

- Manage applications
- Manage application versions
- Cancel workunits
- FLOP count statistics
- Stripcharts
- Show/Grep logs
- Transition all WUs
  (this can 'unstick' old WUs)
- Clear RPC seqno host ID:

User management

- Screen user profiles
- User privileges
- Send mass email to a selected set of users
- Email user with misconfigured host
- Manage user ID:

Result summary for dart:

- Past 24 hours: summary | pass percentage by platform | failure by host | failure by platform
- Past 7 days: summary | pass percentage by platform | failure by host | failure by platform

Show deprecated applications

Periodic or special tasks

- The following scripts should be run as periodic tasks, not via this web page (see http://boinc.berkeley.edu/trac/wiki/ProjectTasks):

  `update_forum_activities.php, update_profile_pages.php, update_uotd.php`
- The following scripts can be run manually on the command line as needed (i.e. php scriptname.php):

  `forum_repair.php, team_repair.php, repair_validator_problem.php`

Back to Main admin page | Log in

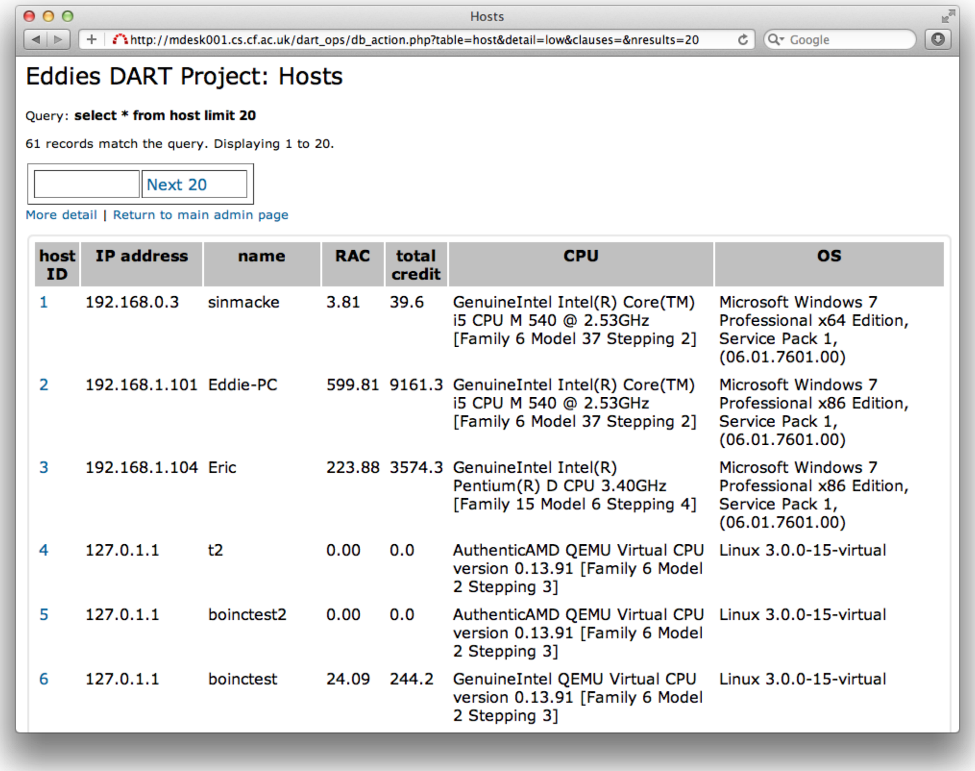
Figure 5.15: The BOINC main administration page

5.6.6 Retrieving Results

A BOINC **Assimilator** was used to handle workunits that are *Completed*. Handling a successfully completed Result often involves recording results in the database and often generating more work. The workunit results are returned to the BOINC server and the Assimilator stores all the DART result text files in one folder, which it zips up. This can then be downloaded and analysed using the exact same methods as explained for XtremWeb, in the previous section.

The BOINC server side software includes a sample Assimilator called **`sample_assimilator`** that was used to as a starting point to create the DART Assimilator.

For successful workunits, the Assimilator writes the canonical instance's output files to the directory **`dart/sample_results/`**. If there is only one output file it is named **`WU_NAME`**. If there are more than one they are named **`WU_NAME_0`**, **`WU_NAME_1`**, and so on.


Hosts
http://mdesk001.cs.cf.ac.uk/dart_ops/db_action.php?table=host&detail=low&clauses=&nresults=20
Google

Eddies DART Project: Hosts

Query: **select * from host limit 20**

61 records match the query. Displaying 1 to 20.

Next 20

More detail | Return to main admin page

host ID	IP address	name	RAC	total credit	CPU	OS
1	192.168.0.3	sinmacke	3.81	39.6	GenuineIntel Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz [Family 6 Model 37 Stepping 2]	Microsoft Windows 7 Professional x64 Edition, Service Pack 1, (06.01.7601.00)
2	192.168.1.101	Eddie-PC	599.81	9161.3	GenuineIntel Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz [Family 6 Model 37 Stepping 2]	Microsoft Windows 7 Professional x86 Edition, Service Pack 1, (06.01.7601.00)
3	192.168.1.104	Eric	223.88	3574.3	GenuineIntel Intel(R) Pentium(R) D CPU 3.40GHz [Family 15 Model 6 Stepping 4]	Microsoft Windows 7 Professional x86 Edition, Service Pack 1, (06.01.7601.00)
4	127.0.1.1	t2	0.00	0.0	AuthenticAMD QEMU Virtual CPU version 0.13.91 [Family 6 Model 2 Stepping 3]	Linux 3.0.0-15-virtual
5	127.0.1.1	boinctest2	0.00	0.0	AuthenticAMD QEMU Virtual CPU version 0.13.91 [Family 6 Model 2 Stepping 3]	Linux 3.0.0-15-virtual
6	127.0.1.1	boinctest	24.09	244.2	GenuineIntel QEMU Virtual CPU version 0.13.91 [Family 6 Model 2 Stepping 3]	Linux 3.0.0-15-virtual

Figure 5.16: The BOINC admin page allows administrators to see an overview of all registered hosts (or 'worker machines').

If the workunit failed (for example, due to too many errors or too much time has passed since the start and the job has timed out) it appends a line to **`sample_results/errors`** containing the workunit name and the error code. To create the DART assimilator, the program scheduler was linked with a C function called **`assimilate_handler()`**,

The **`assimilate_handler()`** function is called when the workunit has a nonzero error code (when there are too many error results, for example) - in which case the handler might write a message to a log or send an email to the project administrator. The function is also called when the workunit has a canonical result. In this case **`wu.canonical_resultid`** will be nonzero, and **`canonica_result`** will contain the canonical result. In both cases the BOINC vector containing the results will be populated with all the workunit's results (including unsuccessful and unsent results).

`assimilate_handler()` results either return with zero to show that the workunit was successfully completed and therefore will be marked as 'assimilated', or will return with **`DEFER_ASSIMILATION`**, meaning that the workunit will be processed again when another instance finishes. **`assimilate_handler()`** can also return with other nonzero values, causing the assimilator to log an error message and exit. **`assimilate_handler()`** should return nonzero for any error condition in order to allow the system administrator to fix the problem(s) before any completed or erroneous workunits are mis-handled by BOINC. To run the DART assimilators as a BOINC daemon a simple entry was added to the DART BOINC configuration file

```
<daemon>
   <cmd> my_assimilator -app dart </cmd>
</daemon>
```

Chapter 6

DART Initial Experiments

6.1 DART Experiment 1

This short chapter presents the results of the initial, small-scale DART experiments, whereby the DART pitch detection application (mapped from Triana to the standalone Dart Execution Environment) runs on a single audio file 50 times with fixed parameters, using both a local XtremWeb Desktop Grid and XtremWeb-EGEE bridge platforms. These experiments not only test the correct mapping of the Triana workflow (taskgraph) to the DART Execution Environment, but also allows for the initial testing and familiarity with the XtremWeb platform, as well as experience in creating a Proxy Server (detailed in the Implementation chapter) and using the XtremWeb to EGEE bridge.

Another aim of these experiments was to give some indication concerning the potential reduction in execution time when running multiple DART experiments on a distributed platform, and to identify any limitations in the current distributed implementation, helping to suggest ways of improving the solution.

The DART SHS pitch detection algorithm is a computationally intensive task, taking a reasonable amount of time to run on a single machine. A parallel execution environment can reduce the processing time of multiple runs/jobs significantly; as the future DART application is updated to work its way through a user's hard disk with potentially thousands of audio files stored, this performance increase becomes invaluable.

As the distributed experiments run independently from each other, a master-worker type of parallelisation is feasible. Theoretically, the performance gain of the XtremWeb Desktop Grid application should be close to the number of worker nodes involved in the computation, compared to the sequential version running on a single machine. The Desktop Grid produced management overhead will, of course, reduce the speed-up.

The tests performed were three fold:

1. Running the mapped DART application 50 times using the Desktop Grid - EGEE bridge in order to gauge the baseline performance of the sequential application on a variety of different machines.

2. Running the mapped DART application 50 times on a local XtremWeb Desktop Grid available at **LAL** (Laboratoire De L'Accelerateur Lineair in Paris, where local, distributed resources are available for use by clients) to ensure that the ported application works correctly, and measure the performance.

3. Running the mapped DART application 50 times using the Desktop Grid - EGEE bridge and in order to determine the speedup of the Desktop Grid version of the application compared to the original sequential version, and compare with the Desktop Grid results.

6.1.1 Background

The DART input file currently being used (DARTAudio.wav) is a 16-Bit/44.1KHz wave file containing 336 seconds of audio to analyse and process. At this stage of development, it was not necessary to analyse multiple audio files - any files of equal length, bit depth, and sampling frequency will take the same time to process, on the same machine.

The DART application was run on 4 locally available machines, in order to gauge the run-time of the application and allow for baseline projections on processing times, given more audio to analyse. The results and specification of the machines are presented in Tables 6.1, 6.2, 6.3 and 6.4.

DART is currently a single-threaded application and only one core per machine is utilised during processing, therefore the 8-core Xeon Mac Pro was not 'disproportionately' faster than the dual-core iMac, and the single core Windows-based machine did not lag

Property	*Value*
Manufacturer / Model	Apple MacBook Pro 15"
Processor	Intel Core Duo 2.16GHz
Memory	2GB
Operating System	OSX 10.5.5
DART Process Completion Time	**172 Seconds**

Table 6.1: Local Test Machine 1 (OSX based, 32-bit)

Property	*Value*
Manufacturer / Model	Apple Mac Pro (Early 2008)
Processor	Intel Xeon Dual Quad 2.8Ghz
Memory	6GB
Operating System	OSX 10.5.5
DART Process Completion Time	**115 Seconds**

Table 6.2: Local Test Machine 2 (OSX based, 64-bit)

too far behind the other machines[1].

Using *Local Test Machine 1* as the standard benchmark, it took **172** seconds to run the DART application locally, on 336 seconds of audio. This can be used to make the projection that in order to process *50* audio files (or 16,800 seconds of CD quality audio), it would take *Local Test Machine 1* 336 x 50 = **8,600 seconds** (146.3 minutes or 2.38 hours) for this machine to complete its analysis. It is common for many people to have hundreds of hours of audio data on their systems - for example *Local Test Machine 1* has around 200 hours of music store on its hard drive. Given that the projection in processing times are linear, it would take *Local Test Machine 1* over 390 hours to process each file once.

The following section displays the results of the experiments conducted on the

[1]Some distributed platforms such as BOINC are able to accept multiple jobs per worker, giving each available CPU core one workunit to process, speeding up overall processing time and maximising the use of resources. XtremWeb currently does not support this feature and leaves the multicore optimisation to the scientists using the platform. Both implementations simply represent a difference in the platform paradigm and can both be advantageous. The results of the different approaches with respect to DART are discussed in the next Chapter.

Property	*Value*
Manufacturer / Model	Apple iMac 24"
Processor	Intel Xeon Dual Quad 2.8Ghz
Memory	2GB
Operating System	OSX 10.4.11
DART Process Completion Time	**127 Seconds**

Table 6.3: Local Test Machine 3 (OSX based, 64-bit)

Property	*Value*
Manufacturer / Model	Custom PC
Processor	Intel Pentium 4 3.2GHz
Memory	2GB
Operating System	Windows XP Pro SP3
DART Process Completion Time	**216 Seconds**

Table 6.4: Local Test Machine 4 (Windows based, 32 bit)

XtremWeb Desktop Grid - both with and without using the XtremWeb-EGEE bridge - and investigates the speedup possible using the ported DART application.

6.2 Testing on the Local XtremWeb Desktop Grid

6.2.1 Test Environment

The following table summarises the specifications of the five worker machines used in the test DG environment:

Property	*Value*
Processor	AMD64 at 2.3Ghz
Memory	8 / 16GB
Operating System	Linux

Table 6.5: LAL Test Machines (Linux based, 64-Bit)

Four of the machines had 8GB of RAM installed, and one machine had 16GB, although DART is not a memory intensive application.

6.2.2 XtremWeb Desktop Grid Test Description

The XtremWeb implementation of the DART experiments used the same algorithm as the sequential DART application in the previous section, however distributes the execution of the application across the available worker machines and returns the results back to the XtremWeb-Server, as described in the Design and Implementation Chapters.

There were 5 computers on the XtremWeb DG on which to run the current DART application. In total, the DART application was run 50 times over the 5 machines.

6.2.3 XtremWeb Desktop Grid Test Results

The tests results showed that DART was successfully run on all Desktop Grid worker machines as the **`DARTResults.txt`** file was produced in each case. Figure 6.1 shows the jobs per host distribution, revealing a fairly even distribution across the 5 machines on the desktop grid.

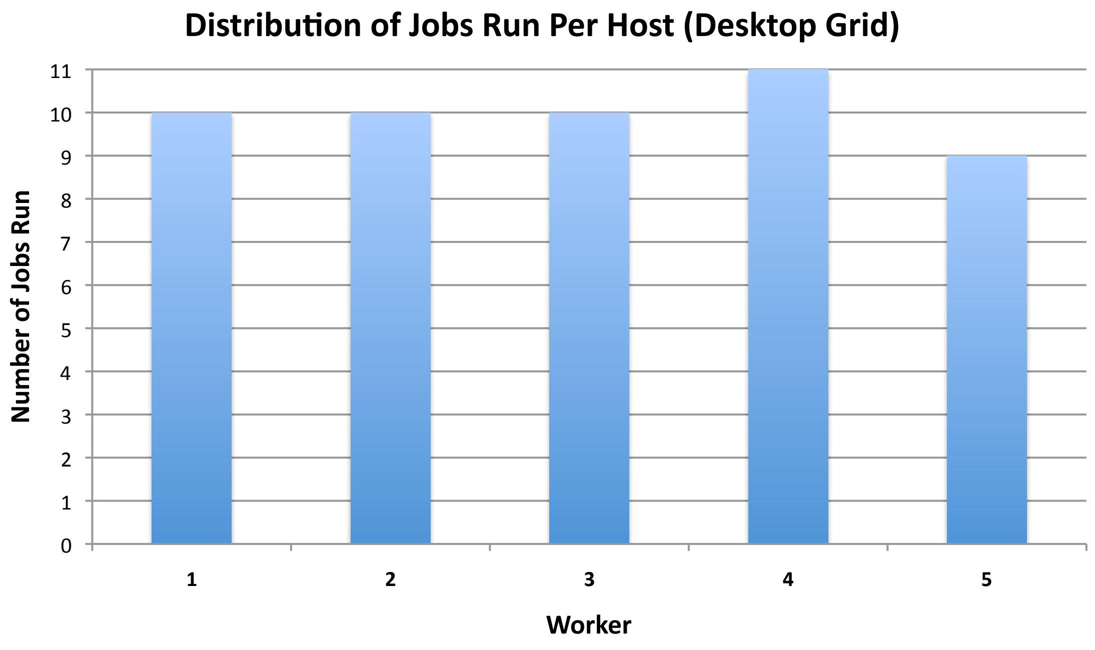


Figure 6.1: Job distribution over the 5 worker machines on the XtremWeb Desktop Grid

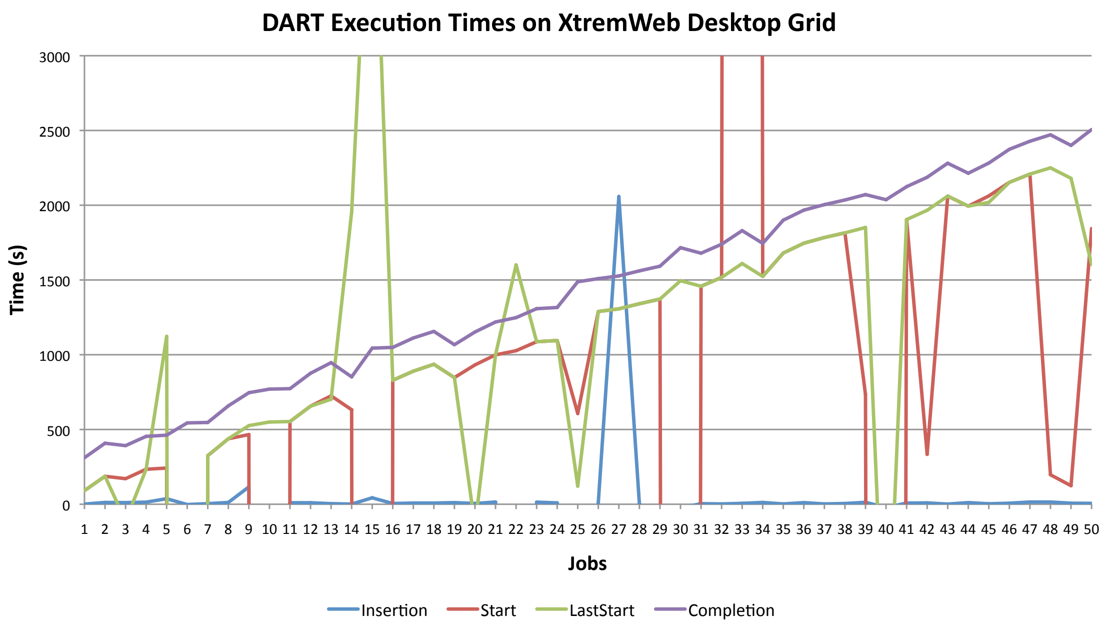


Figure 6.2: DART Execution time when run over XtremWeb Desktop Grid

It should be noted that there were some anomalies in the timings noted when generating the results from this execution of the 50 jobs over the desktop grid, as exemplified by the extreme peaks and troughs in Figure 6.2. However the **Completion** results are all valid, and these results reveal the most about how long it took to run the 50 jobs on the XtremWeb Desktop Grid. The time anomalies were reported to the XtremWeb team, who were working to resolve them.

The job time values in Figure 6.2 are all from **0** - meaning that the time measurement of the completion of the last job was started at the same time as the first job - therefore the time of the completion of the last job reveals how long it took to process the DART application 50 times.

Working through an example scenario helps to explain the results, and in the case of investigating the first job (Job 1):

- At **1** seconds the job was taken in account by the scheduler (Insertion: Blue line)
- At **91** seconds there was a first attempt to run (Start: Red line)
- At **91** seconds there was the last attempt to run and this was successful (LastStart: Green line)
- At **311** seconds the job was complete (Completion: Purple line)

- It took **311 - 91** seconds to effectively download the job to the worker (on worker request) and download all needed files (JAR + Wav audio file) and to execute and to upload result to server

- Therefore, total time for execution: **220 seconds**

Figure 6.3 shows the time taken for each job to run across the 5 available worker nodes. Again, this includes the time to download all the necessary files, execute the DART application, and then upload the results to the XtremWeb server. Figure 6.3 shows that the fastest processing time was **220** seconds the slowest processing time was **263** seconds and reveals an average processing time of **222.1** seconds.

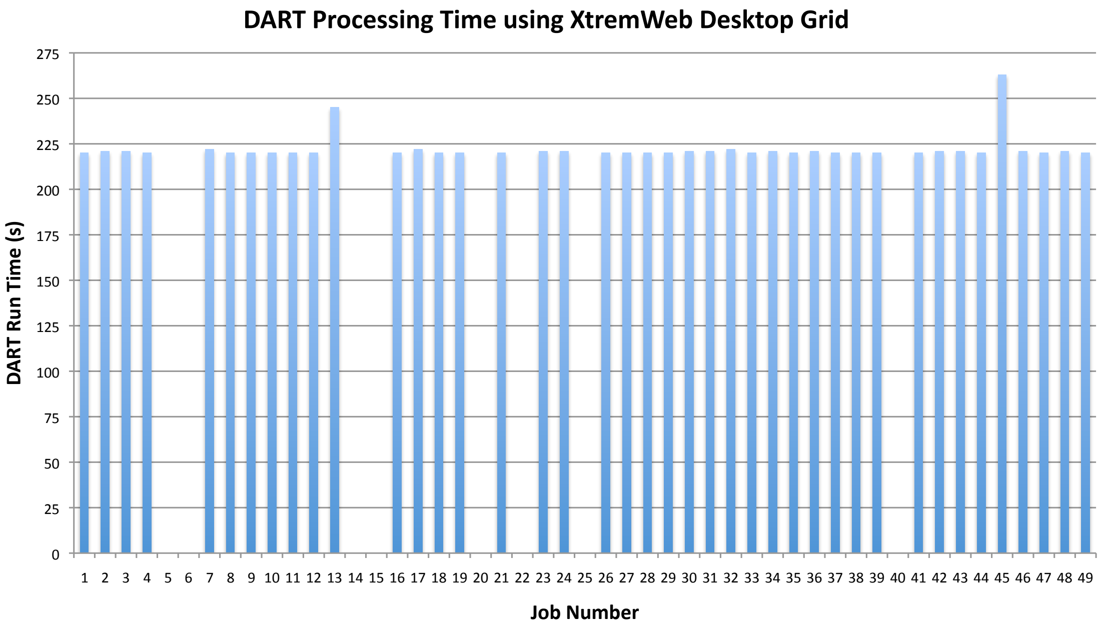


Figure 6.3: DART processing for each of the 50 jobs run over XtremWeb Desktop Grid

Of particular interest, is the overhead incurred by having to download the JAR and **`DARTAudio.wav`** data files, and the uploading of the results. The runtime of the actual DART application was shown to be approximately 165 seconds, meaning that there is approximately a 55 second 'penalty' involved each time the DART application is run. A suggestion made to the XtremWeb team was the implementation of a caching scheme in order to minimise the re-download of files that have previously been processed.

6.3 XtremWeb Desktop Grid to EGEE Bridge

6.3.1 Overview

The purpose of the XtremWeb-EGEE test was to investigate the differences in performance when running the DART application 50 times using the XtremWeb to EGEE bridge, and allow for comparison with the results of running the DART application sequentially and over the XtremWeb Desktop Grid.

6.3.2 Prerequisites

- *Linux versions of the application*: the EGEE worker nodes are working with a Linux based operating system (Scientific Linux). In order to send work units through the bridge into EGEE Linux versions (32 bit, 64 bit) of the application have to be provided. As DART is written in Java (executable JAR distribution), as is XtremWeb, this was not an issue.

- *Opening a port of the Laboratoire De L'Accelerateur Lineair firewall*: the LAL Desktop Grid is a local Desktop Grid and can be only accessed from inside the university. To make it accessible, a firewall rule had to be set up to allow connections from the bridge machine.

- *New DG user for the DG-EGEE Bridge*: this step was not mandatory, but it is advantageous. With a separate user account it is easier to monitor the work done by the bridge.

6.3.3 Test Environment

The testing utilised an experimental DG-EGEE Bridge that was set up by LAL. The bridge submitted the converted work units to the EDGeS VO that comprised of 21 free machines on the EGEE network grid. Table 6.6 summarises the capabilities of the worker machines used in the EGEE environment. Nine of the machines were running at 2GHz, the other 12 running at 2.3Ghz, all with 16GB RAM.

Property	*Value*
Processor	AMD64 / 2GHz - 2.3Ghz
Memory	16GB
Operating System	Linux

Table 6.6: EGEE Test Machine Specifications (Linux based, 64-Bit)

6.3.4 Test Results

Since the XtremWeb-EGEE bridge can act as a client or worker on the local XtremWeb Desktop Grid, using the bridge was as simple as testing on the local DG; this test helped to clarify that the bridge fetches work units and returns the correct results of those work units. This was tested for different numbers of work units and worked correctly, and as such future runs of DART should be able to utilise the XtremWeb-EGEE bridge without issue.

It should be noted that there was not time to run all 50 jobs over EGEE, demonstrating a problem with running jobs using the bridge, which is often heavily overbooked for resources. The DART application was successfully run 44 times across 21 machines over EGEE. Figure 6.4 shows the jobs per host distribution.

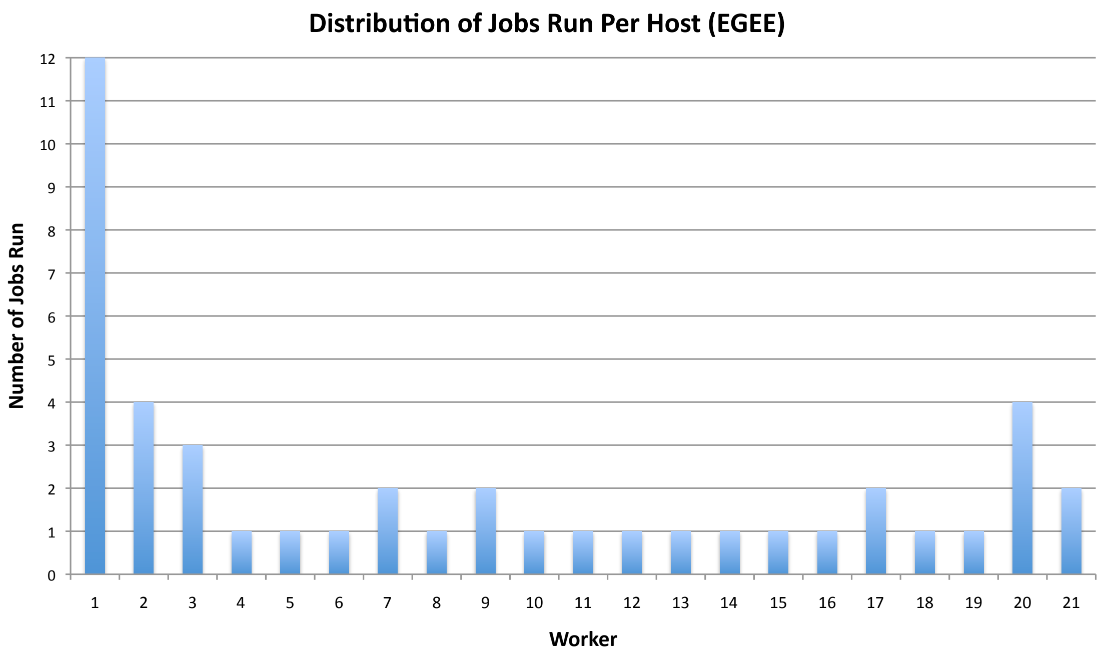


Figure 6.4: Job distribution over the 21 worker machines on the EGEE Grid

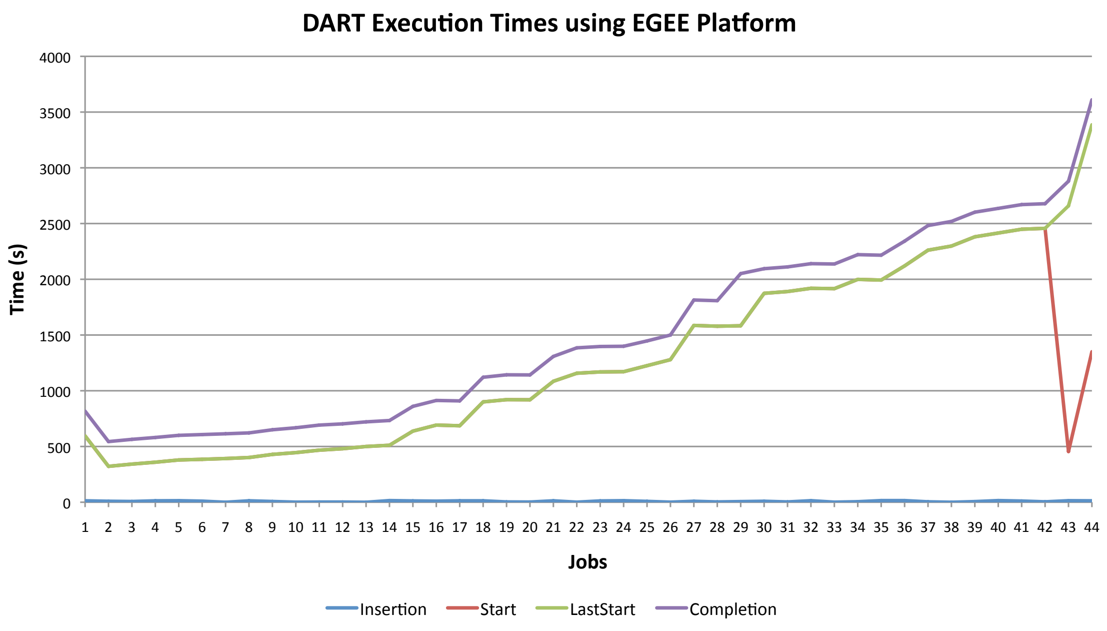


Figure 6.5: DART Execution time when run over EGEE Grid

As with the previous experiment run on the XtremWeb Desktop Grid, the job time values in Figure 6.5 are all from **0**. Investigating the case of the last job (Job 44), looking at an example scenario helps to explain the results:

- At **13** seconds the job was taken in account by the scheduler (*Insertion*: Blue line)
- At **1350** seconds there was a first attempt to run the DART job, which failed (*Start*: Red line)
- At **3386** seconds there was the last attempt to run and this was successful (*LastStart*: Green line)
- At **3610** seconds the job was complete (*Completion*: Purple line)
- So it took **3610 - 3386** seconds to effectively download the job to the worker (on worker request) and download all needed files (JAR + wav) and to execute and to upload result to server
- Total time for execution: **224 seconds**

Figure 6.6 shows the time taken for each job to run across the 21 available nodes. Again, this includes the time to download all the necessary files, execute the DART application, and then upload the results to the server. The fastest processing time was revealed to be **221** seconds, with slowest and average processing times of **486** and **228.1** seconds respectively.

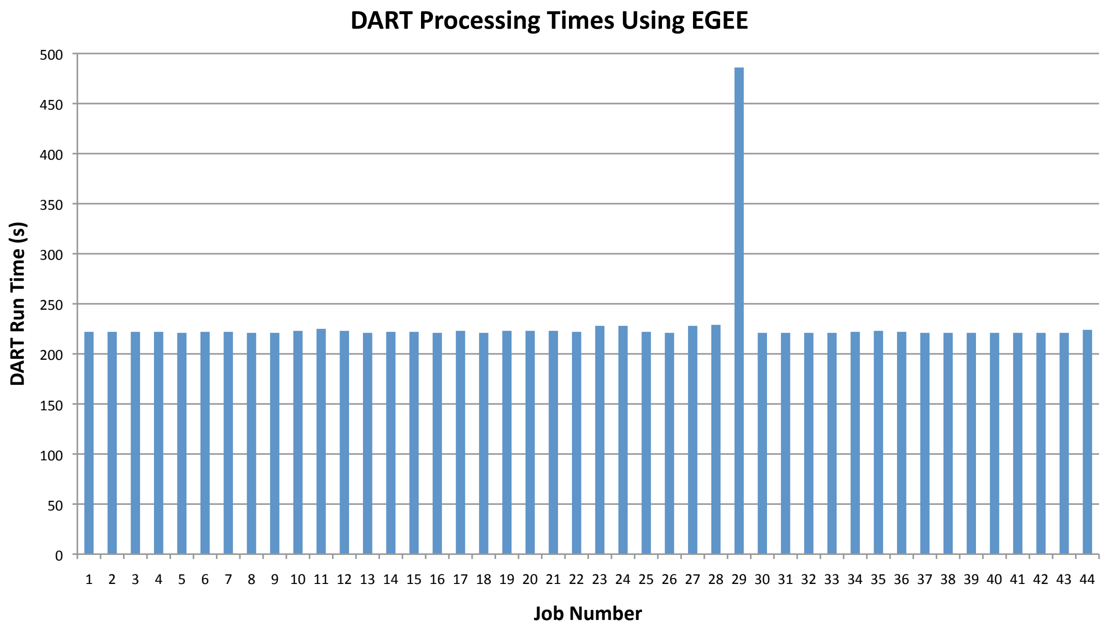


Figure 6.6: DART processing for each of the 50 jobs run over EGEE Grid

Of particular interest, is the overhead incurred by downloading the JAR and DARTAudio.wav data file, and the uploading of the results. As the specification of the worker machines running over EGEE was very similar to those running on the LAL XtremWeb Desktop Grid, the DART processing times were similar. The runtime of the actual DART application was shown to be around **165 seconds**, meaning that there is approximately a **63-70 second 'penalty'** involved each time the DART application is run, in contrast to the 55 seconds incurred by the local Desktop Grid.

6.3.5 Results Comparison

The test results presented in this chapter show that DART was successfully run on all machines as the DARTResults.txt file was produced and returned to the server without errors in each case, showing that running DART on both the XtremWeb DG and XtremWeb-EGEE bridge are feasible options for scientific research. Figure 6.7 summarises the results of the test runs, showing the total time taken to run **50** jobs on the LAL XtremWeb Desktop Grid, and **44** jobs over the EGEE network. Also included are the projected times to run the DART application 50 times (sequentially) on the 4 local test machines at Cardiff University. Figure 6.7 demonstrates the achieved 'speed-up' by the distributed DART application when compared to the sequential version. Table 6.7 presents a Platform *Key* for the Graph presented in Figure 6.7.

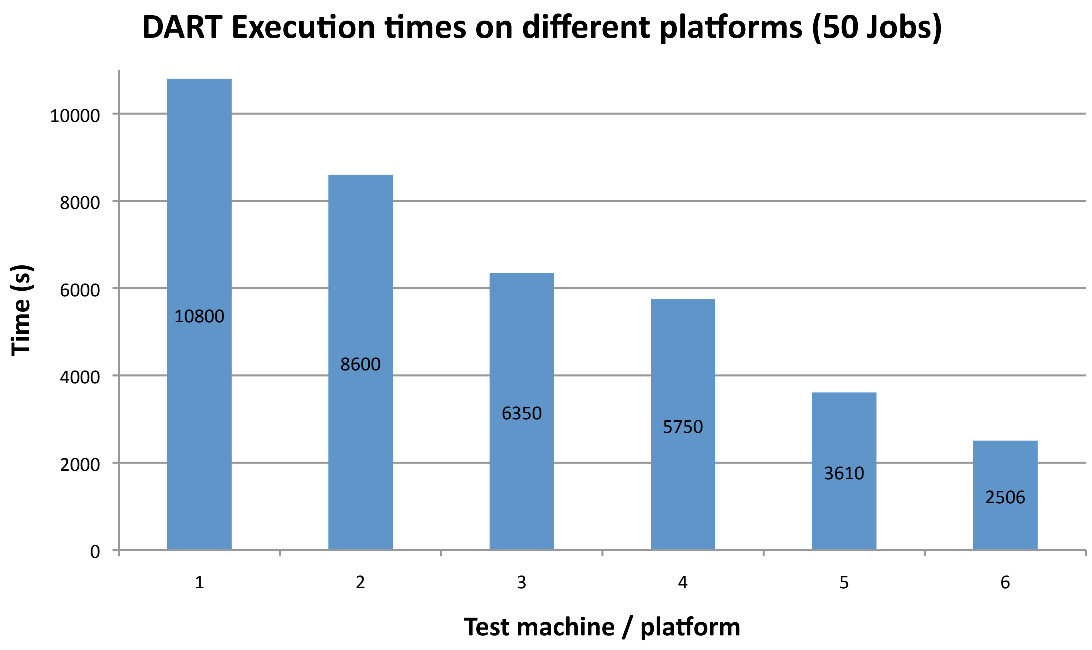


Figure 6.7: Chart to show overall time to run 50 jobs (44 for EGEE) on the 6 different test platforms

Test Platform Key Chart	
Test platform 1:	Single Windows PC Pentium 4, 3.2GHz
Test platform 2:	Single Apple MacBook Pro 2.8GHz
Test platform 3:	Single Apple iMac 2.8Ghz
Test platform 4:	Single Apple Mac Pro 2.8GHz
Test platform 5:	XtremWeb-EGEE Bridge/Grid (21 Workers)
Test platform 6:	XtremWeb Desktop Grid (5 Workers)

Table 6.7: Test Platform Key for the Graph presented in Figure 6.7

The results show that the XtremWeb DG with 5 workers is approximately 44% faster than the EGEE grid over 21 worker machines, which was only able to run 44 out of the 50 jobs in the time available for testing, demonstrating the disadvantages of running jobs on the overbooked EGEE grid.

This 44% increase in speed over the experiments running using the XW-EGEE bridge was achieved with only 5 machines versus the 21 used on EGEE, as the desktop grid was not busy waiting for other processes to complete - there were no failed attempts to run the DART jobs.

However, with a larger number of jobs and potentially more workers on both platforms, the EGEE grid could potentially process DART experiments in a shorter time. The results of further research using large scale parameter sweep experiments is given in the next chapter.

6.3.6 Conclusion and Future Work

These experiments showed that running the distributed DART application successfully produced DART results many times faster than the sequential version running on a single machine. The XtremWeb Desktop Grid application ran 44% faster than the ported EGEE version (to run 50 jobs vs. the 44 run on EGEE), despite only using 5 nodes, as opposed to the 21 used over EGEE. This is due to reduced queuing times, while DART waits for resources on EGEE to become available. The XtremWeb DG version was also 4.3x faster than the slowest Windows PC running the application sequentially.

The performance of the current version of the DART could be improved (that is to say, the running time of the DART application could be decreased) by making the application multi-processor aware. This would also speed up the processing on local, multi-core machines (all local test machines at Cardiff University are at least dual-core, other than the Windows PC), and an investigation on the performance difference between the local and DG versions would be of interest.

Furthermore, much of the performance overhead that occurred on the XW DG version was from the downloading of the DARTAudio.wav file. The sequential versions of the application did not have this overhead, and both BOINC and future versions of XtremWeb will employ a caching scheme in order to minimise the overheads and reduce bandwidth use, as a result of the DART research experiments.

As the Dart Execution Environment/XtremWeb system is in place, it is now easier to focus on developing and refining the DART pitch detection algorithm and begin the investigation into the effect of varying the parameters of the SHS algorithm and conducting large scale experiments on the XtremWeb and BOINC platforms. Given the number of variables and the size of data files, the parallelism afforded by the XtremWeb environment massively reduces the time taken to process all the data, making the MIR research feasible.

Chapter 7

DART Sub-Harmonic Summation Results

7.1 Chapter Overview

This chapter displays and discusses the results of the large scale DART Sub-Harmonic Summation (SHS) parameter sweep investigation. The purpose of this investigation was to carry out a real-world MIR experiment, showing that the DART platform could be used to perform such a task, as well as to contribute the results of the SHS parameter sweep experiments to the research community.

In this chapter, the efficacy of each modified DART parameter is analysed, allowing the optimal parameters of the SHS algorithm to be found, both across all input data files, and also individually for each of the six different instruments sampled. All accuracy results are presented as a percentage. The effect of modifying each of the four variables (audio input file, number of harmonics, number of 'top frequencies' analysed, and FFT window type) is shown for all results and also *per audio input file.* This chapter also displays the time taken to run the large scale DART experiments on both the XtremWeb and BOINC platforms.

The first results Section (7.2) shows the overall accuracy of the SHS algorithm, giving the minimum, maximum and average accuracy rates, with and without taking into account octave mistakes. Octave errors are the most common type of error in pitch detections as many of the frequency harmonics are common. Furthermore, when determining pitch, the user may also place more importance (or weight) on the accuracy of the pitch, rather than the octave. Therefore, octave errors are extracted into a different category to help understand the proportion of octave mistakes versus misclassifications in pitch. This information is then displayed in a chart, showing the minimum, maximum and average accuracy values for *each audio input file*, revealing which audio files (or which instruments) are best suited to the SHS algorithm (or alternatively, how well the SHS algorithm works for each instrument time).

The **FFT Window Type** accuracy results are then presented in Section 7.3. This section presents a graph showing the resulting accuracy of each of the 28 FFT windows (again with the minimum, maximum and average accuracy values), across all results, and again for each audio input file, allowing for insight to be gained on the optimal FFT Window Type, both for each audio input type and across all audio input.

The accuracy of the **Number Of Harmonics** and **Number of Top Frequency Points** (referred to as **NTFP**) variables is then given both overall and for each of the six audio input files. For parameters with a sweeping range such as **Number Of Harmonics** (from 1-32) and **NTFP** (from 1-50), a line graph is used to plot the accuracy. The FFT Window Type and Audio Input Files sections require column bar charts to plot their various accuracy values.

The most optimal parameter in each scenario can be found by summing the scores in the first 5 columns of each results table. The parameter or variable with the highest score is the most optimal parameter when taking into account accuracy, both with and without octave errors.

Each section in this chapter contains a graph giving a graphical overview of the presented data, as well a table containing the exact values (rounded to 2 decimal places) when relevant. The tables highlight the optimal or top values, in each case.

Finally, the results of the time taken to execute the DART jobs is given. The analysis of the results is based on the results from the final runs of the XtremWeb/GRID5000 and BOINC DART experiments. 268,796 results were successfully retrieved on the GRID5000 system, giving a 99.9985% job success rate[1], with all 268,800 results retrieved using

[1]The 4 results that could not be retrieved (failed) were from the XtremWeb server and the reasons for

BOINC[2].

Over 200 worker systems were used on XtremWeb and 48 active workers were used in the BOINC experiments - and as such an overall insight into the minimum, maximum, and average processing times (in total and per audio file) is useful - however machines of various processing power were used and as such the execution time of each job is expected to vary. The chapter ends with a discussion on the usability and the differing advantages to using the two platforms.

7.2 Overall Accuracy of SHS Algorithm Across All Audio Files

The overall results for the accuracy of the DART SHS algorithm can be summarised in Table 7.1. ***Maximum Accuracy*** is the single highest accuracy value for a results file - this means that both the note and the octave are correct. For correct notes and correct octaves, the highest accuracy achieved using the SHS algorithm is **94.95%**, which is achieved in 420 files, all of which are acoustic guitar results. However, when allowing for/including octave errors (or 'ignoring' octave errors), a total accuracy of up to **99.47%** can be achieved - again all with acoustic guitar samples. All of the 10,405 results which have a 0% minimum accuracy rate are achieved when analysing the tubular bells audio input file.

The ***Top Accuracy Value Plus Octave Errors*** category at the bottom of Table 7.1 is calculated by looking at the 420 result files that shown the maximum accuracy value of 94.95% and ignoring their octave mistakes. This gives a result of 98.14% - lower than the 99.47% accuracy rate which is given when octave errors are ignored completely. This means that the results sets which give the top 99.47% (Maximum Accuracy Inc. Octave Errors) are *not the files with singularly top result of 94.95%.* Ignoring the octave mistakes gives an extremely high accuracy rate for the acoustic guitar samples.

These results span across *all* parameter variations and includes settings that provide both accurate and inaccurate results. The most optimal settings are extracted later on in this chapter.

These results can be seen displayed in Graph 7.1, which shows the accuracy of the SHS

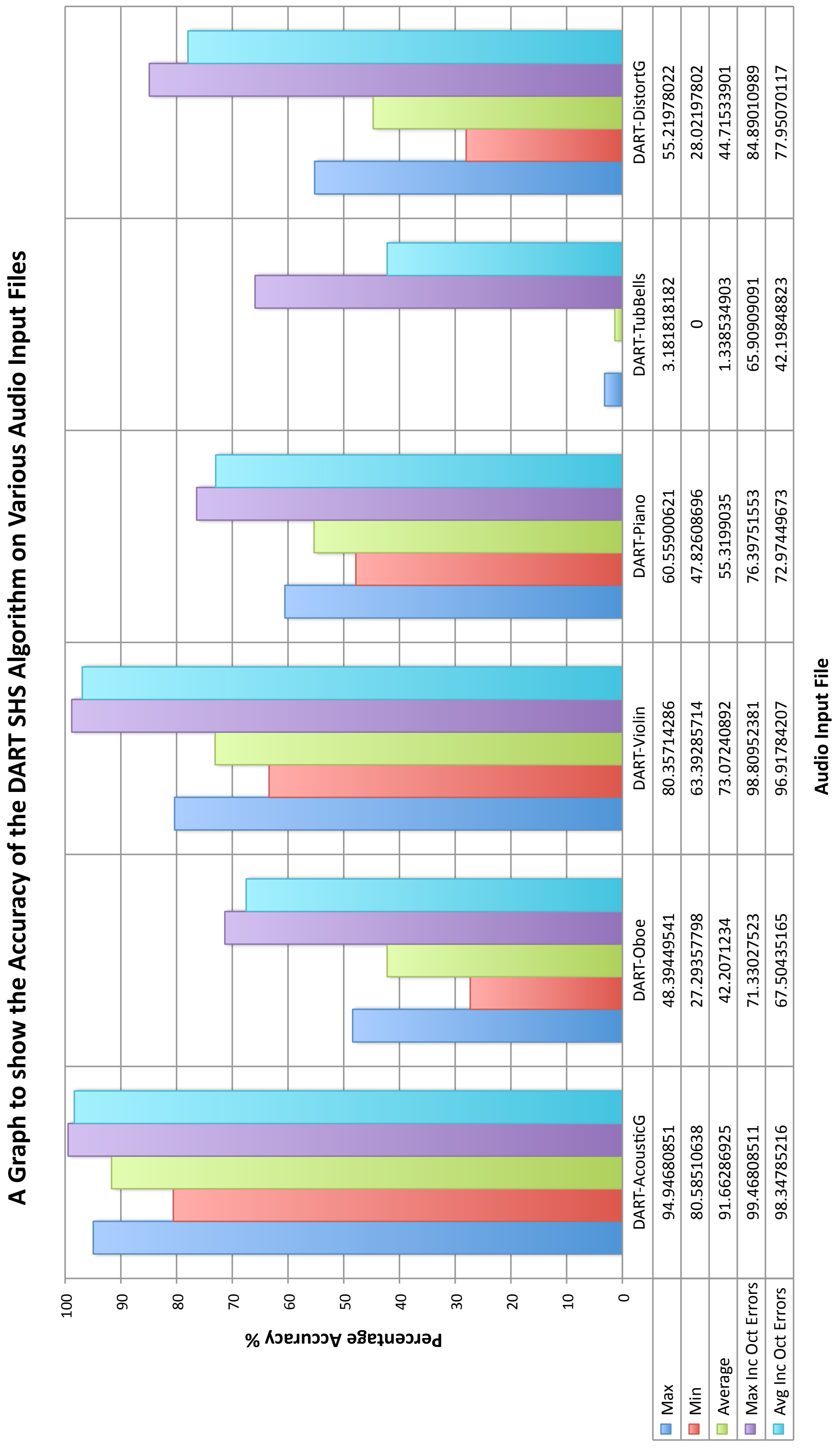
the failure are unknown - the server simply would not return those results. The XtremWeb middleware did not provide any other explanation, however these machines could have been terminated (switched off).

[2]The analysis of the XtremWeb results began during the run of BOINC experiments.

algorithm when analysing all 6 audio files. The graph shows the minimum, maximum and average accuracy values of the DART SHS algorithm, for each input file - including the maximum and average accuracy values when ignoring octave mistakes. The Acoustic Guitar audio input file has by far the highest accuracy rate, with the worst accuracy given when analysing the Tubular Bells input file. When ignoring octave errors however, the SHS algorithm implemented in DART can achieve up to 65.91% accuracy when analysing the Tubular Bells file, with an average result of 42.2% accuracy, including octave errors. The ***Top Result + Octave Error*** result is also included, to enable for the distinction between this value and the ***Max Inc. Octave Error*** value, which were often distinct.

Maximum Accuracy	94.95%	420 files
Minimum Accuracy	0.0%	10,405 files
Average Accuracy	51.39%	
Average Accuracy Including Octave Errors	75.98%	
Minimum Octave Error	3.19%	420 files
Maximum Octave Error	65.91%	352 files
Average Octave Error	24.60%	
Maximum Accuracy Inc. Octave Errors	99.47%	542 files
Top Accuracy Value Plus Octave Errors	98.14%	420 files

Table 7.1: This table details the overall accuracy statistics of the DART SHS algorithm across all six input files, and also the number of results files which give the accuracy value shown.



	DART-AcousticG	DART-Oboe	DART-Violin	DART-Piano	DART-TubBells	DART-DistortG
🟦 Max	94.94680851	48.39449541	80.35714286	60.55900621	3.181818182	55.21978022
🟥 Min	80.58510638	27.29357798	63.39285714	47.82608696	0	28.02197802
🟩 Average	91.66286925	42.2071234	73.07240892	55.3199035	1.338534903	44.71533901
🟪 Max Inc Oct Errors	99.46808511	71.33027523	98.80952381	76.39751553	65.90909091	84.89010989
🟦 Avg Inc Oct Errors	98.34785216	67.50435165	96.91784207	72.97449673	42.19848823	77.95070117

Figure 7.1: A graph to show the accuracy of the SHS algorithm, with minimum, maximum and average score values for each audio input file. Also included are the maximum and average accuracy results when ignoring octave errors.

7.3 FFT Window Accuracy Results

This section displays the effect of varying the 28 FFT Window Types on the accuracy of the SHS algorithm. Each subsection looks at the results *per audio file*, with the result across all audio files displayed in the final sub-section 7.3.7.

Each results set consists of a graph displaying the data, as well as a table containing the exact values in the graph, and more information based on the Minimum, Maximum and Average Octave Error values. The *Top Value + Octave Error* value is also given.

7.3.1 FFT Window Accuracy Results for Acoustic Guitar

Graph 7.2 shows the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-AcousticG audio (Acoustic Guitar) input file. Table 7.2 illustrates the actual values presented in Graph 7.2. The top value(s) in each column is highlighted in bold. The abbreviation **O.E** stands for *Octave Error*. The first 5 columns represent data presented in Graph 7.2, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of the chosen FFT Window.

In the case of the Acoustic Guitar, the **Welch** FFT Window Type produced the highest Minimum (85.64%), Maximum (94.95%) and Average (93.97%) values, and can be considered the most optimal Window Type when octave errors are unwanted.

The **Nuttall3b**, **Kaiser5**, **Kaiser6**, and **SFT5M** Window Types all produce a higher accuracy rate when ignoring octave errors (99.47%), with the **Kaiser5** Window type also producing the most optimal Top Result + Octave Error, alongside the **SFT5M** Window Type. The **SFT5M** Window Type produced the highest average accuracy rate when including octave errors, with 98.79% accuracy. The **Kaiser5** Window type produced a slightly less accurate Avg Inc O.E value at 98.72%, and therefore the **SFT5M** Window type can be considered the most optimal performing when octave errors are ignored.

The **FTSRS**, **HFT70** and **HFT95** Window Types produced the least accurate results, and produced the highest number of Minimum, Maximum and Average Octave Errors, as can be seen in Table 7.2.

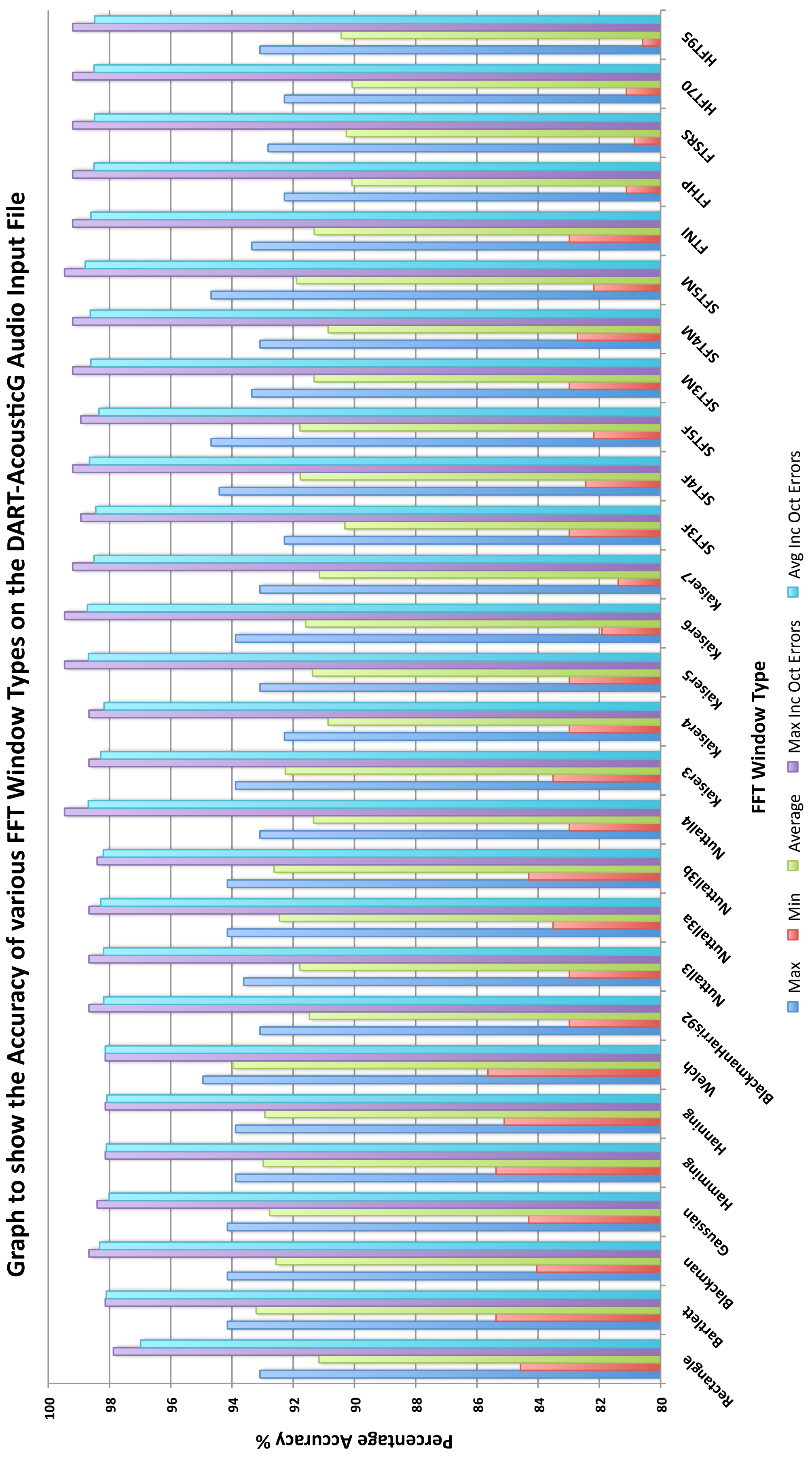


Figure 7.2: A graph to show the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-AcousticG audio input file, with minimum, maximum and average score values. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

FFTWindow	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
Rectangle	93.09	84.57	91.16	97.87	96.98	97.87	11.70	4.79	5.83
Bartlett	94.15	85.37	93.21	98.14	98.10	98.14	12.50	3.99	4.90
Blackman	94.15	84.04	92.56	98.67	98.32	98.67	13.83	4.52	5.76
Gaussian	94.15	84.31	92.77	98.40	98.01	98.14	13.56	3.99	5.24
Hamming	93.88	85.37	92.98	98.14	98.10	98.14	12.50	4.26	5.13
Hanning	93.88	85.11	92.93	98.14	98.08	98.14	12.77	4.26	5.16
Welch	**94.95**	**85.64**	**93.97**	98.14	98.14	98.14	12.50	3.19	4.16
BlackmanHarris92	93.09	82.98	91.47	98.67	98.19	98.67	14.63	5.32	6.72
Nuttall3	93.62	82.98	91.79	98.67	98.19	98.67	14.89	5.05	6.41
Nuttall3a	94.15	83.51	92.45	98.67	98.29	98.67	14.36	4.52	5.84
Nuttall3b	94.15	84.31	92.62	98.40	98.20	98.40	13.56	4.26	5.58
Nuttall4	93.09	82.98	91.33	**99.47**	98.69	99.20	14.36	6.12	7.36
Kaiser3	93.88	83.51	92.26	98.67	98.29	98.67	14.36	4.79	6.03
Kaiser4	92.29	82.98	90.86	98.67	98.18	98.67	14.63	6.12	7.32
Kaiser5	93.09	82.98	91.37	**99.47**	98.69	99.20	14.36	6.12	7.32
Kaiser6	93.88	81.91	91.59	**99.47**	98.72	**99.47**	14.89	5.59	7.13
Kaiser7	93.09	81.38	91.14	99.20	98.51	99.20	15.69	5.85	7.36
SFT3F	92.29	82.98	90.31	98.94	98.45	98.94	14.36	6.65	8.14
SFT4F	94.41	82.45	91.77	99.20	98.65	99.20	15.16	4.79	6.88
SFT5F	94.68	82.18	91.78	98.94	98.34	98.94	15.16	4.26	6.56
SFT3M	93.35	82.98	91.31	99.20	98.61	99.20	14.10	5.85	7.29
SFT4M	93.09	82.71	90.86	99.20	98.63	99.20	14.63	6.12	7.77
SFT5M	94.68	82.18	91.89	**99.47**	**98.79**	**99.47**	15.16	4.79	6.90
FTNI	93.35	82.98	91.31	99.20	98.61	99.20	14.10	5.85	7.30
FTHP	92.29	81.12	90.08	99.20	98.50	99.20	**16.22**	**6.91**	8.42
FTSRS	92.82	80.85	90.27	99.20	98.49	99.20	15.96	6.38	8.22
HFT70	92.29	81.12	90.07	99.20	98.50	99.20	**16.22**	**6.91**	**8.43**
HFT95	93.09	80.59	90.43	99.20	98.48	99.20	**16.22**	6.12	8.05

Table 7.2: This Table details the accuracy of the DART SHS algorithm while varying the FFT Window Type on the DART Acoustic Guitar audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.3.2 FFT Window Accuracy Results for Oboe

Graph (7.3) shows the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-Oboe audio input file. Table 7.3 illustrates the actual values presented in Graph 7.3. The first 5 columns represent data presented in 7.3, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of the chosen FFT Window.

In the case of the Oboe, the **Welch** FFT Window Type produced the highest Maximum accuracy result at 48.39%. The highest Minimum, Average, Maximum Including O.E, Average Including O.E, and Top + O.E, was the **Rectangle** Window Type.

The Rectangle Window Type only produced a Maximum accuracy rate of 0.22% less than the Welsh algorithm and so can be considered the most optimal overall FFT Window Type for the Oboe Audio Input File.

The **HFT70**, **Gaussian** and **HFT95** Window Types produced the least accurate results, as can be seen in Table 7.3.

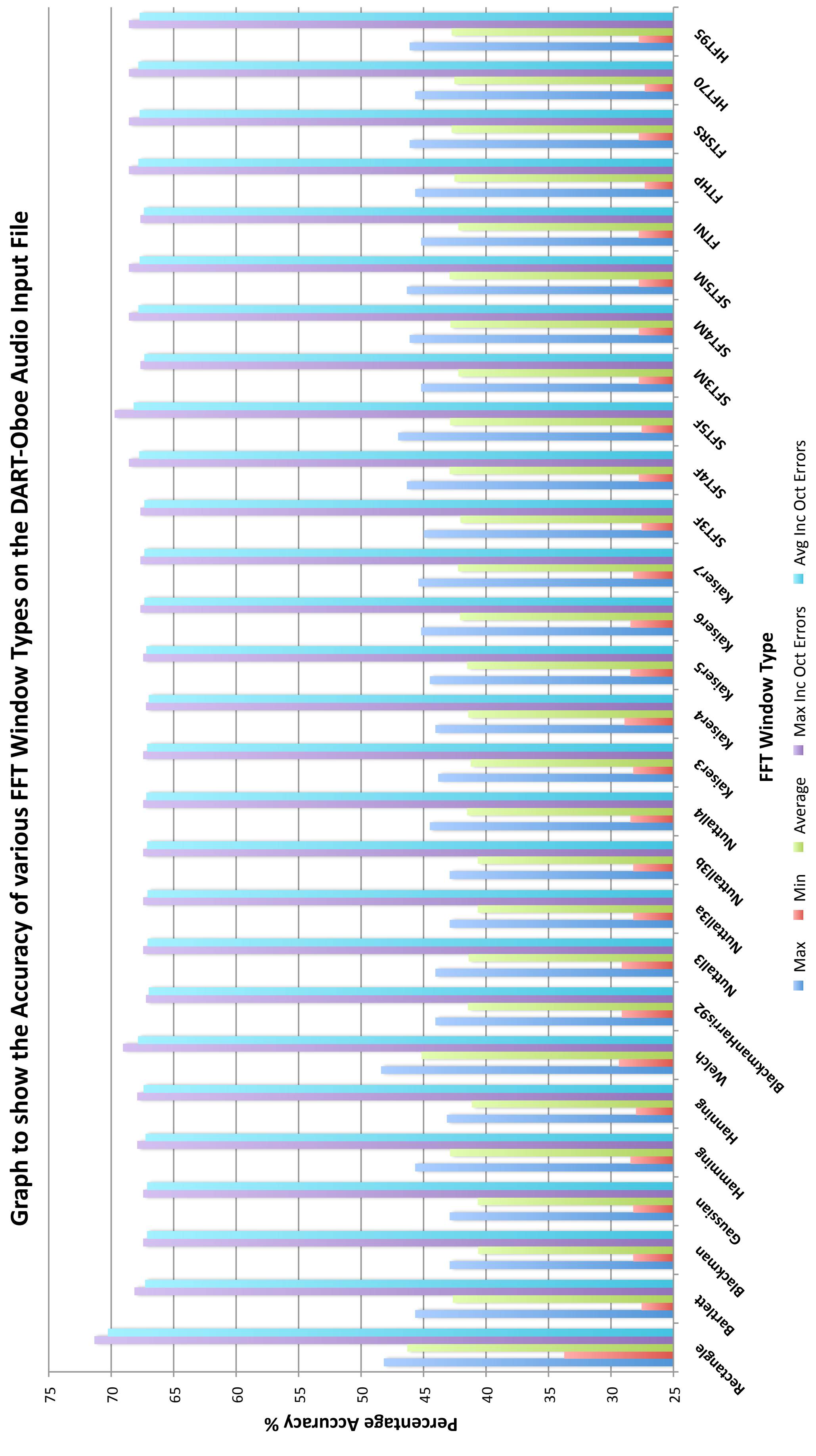


Figure 7.3: A graph to show the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-Oboe audio input file, with minimum, maximum and average score values. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

FFTWindow	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
Rectangle	48.17	**33.72**	**46.29**	**71.33**	**70.24**	**71.33**	30.05	22.02	23.95
Bartlett	45.64	27.52	42.66	68.12	67.27	68.12	36.70	22.48	24.62
Blackman	42.89	28.21	40.63	67.43	67.11	67.43	37.16	24.54	26.49
Gaussian	42.89	28.21	40.65	67.43	67.11	67.43	37.16	24.54	**26.46**
Hamming	45.64	28.44	42.87	67.89	67.24	67.89	35.78	22.25	24.37
Hanning	43.12	27.98	41.11	67.89	67.40	67.89	37.16	**24.77**	26.29
Welch	**48.39**	29.36	45.14	69.04	67.84	69.04	34.86	20.41	22.71
BlackmanHarris92	44.04	29.13	41.43	67.20	66.97	67.20	36.24	23.17	25.54
Nuttall3	44.04	29.13	41.37	67.43	67.08	67.43	36.24	23.39	25.71
Nuttall3a	42.89	28.21	40.65	67.43	67.09	67.43	37.16	24.31	26.43
Nuttall3b	42.89	28.21	40.63	67.43	67.12	67.43	37.16	24.54	26.48
Nuttall4	44.50	28.44	41.51	67.43	67.18	67.43	36.93	22.94	25.67
Kaiser3	43.81	28.21	41.21	67.43	67.11	67.43	37.16	23.62	25.89
Kaiser4	44.04	28.90	41.41	67.20	66.97	67.20	36.47	23.17	25.56
Kaiser5	44.50	28.44	41.50	67.43	67.17	67.43	36.93	22.94	25.66
Kaiser6	45.18	28.44	42.07	67.66	67.32	67.66	36.93	22.48	25.26
Kaiser7	45.41	28.21	42.23	67.66	67.32	67.66	36.93	22.25	25.09
SFT3F	44.95	27.52	42.03	67.66	67.35	67.66	37.39	22.71	25.31
SFT4F	46.33	27.75	42.89	68.58	67.76	68.58	37.16	22.25	24.87
SFT5F	47.02	27.52	42.87	69.72	68.19	69.72	37.39	22.71	25.32
SFT3M	45.18	27.75	42.19	67.66	67.34	67.66	37.16	22.48	25.15
SFT4M	46.10	27.75	42.85	68.58	67.81	68.58	37.39	22.48	24.96
SFT5M	46.33	27.75	42.90	68.58	67.70	68.58	37.16	22.25	24.81
FTNI	45.18	27.75	42.21	67.66	67.36	67.66	37.16	22.48	25.16
FTHP	45.64	27.29	42.52	68.58	67.81	68.58	37.61	22.94	25.29
FTSRS	46.10	27.75	42.75	68.58	67.72	68.58	37.16	22.48	24.97
HFT70	45.64	27.29	42.52	68.58	67.81	68.58	**37.61**	22.94	25.29
HFT95	46.10	27.75	42.73	68.58	67.72	68.58	36.93	22.48	24.99

Table 7.3: This Table details the accuracy of the DART SHS algorithm while varying the FFT Window Type on the DART Oboe audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.3.3 FFT Window Accuracy Results for Violin

Graph 7.4 shows the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-Violin audio input file. Table 7.4 shows the values presented in Graph 7.4. The first 5 columns represent data presented in 7.4, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of the chosen FFT Window.

In the case of the Violin, the **Rectangle** FFT Window Type produced the highest Minimum, Maximum, Average, Maximum Including O.E, Average Including O.E, and Top + O.E, accuracy results. The Rectangle Window Type can therefore be considered the most optimal overall FFT Window Type for the Violin Audio Input File.

The **FTHP**, **Gaussian** and **HFT95** Window Types produced the least accurate results, as can be seen in Table 7.4.

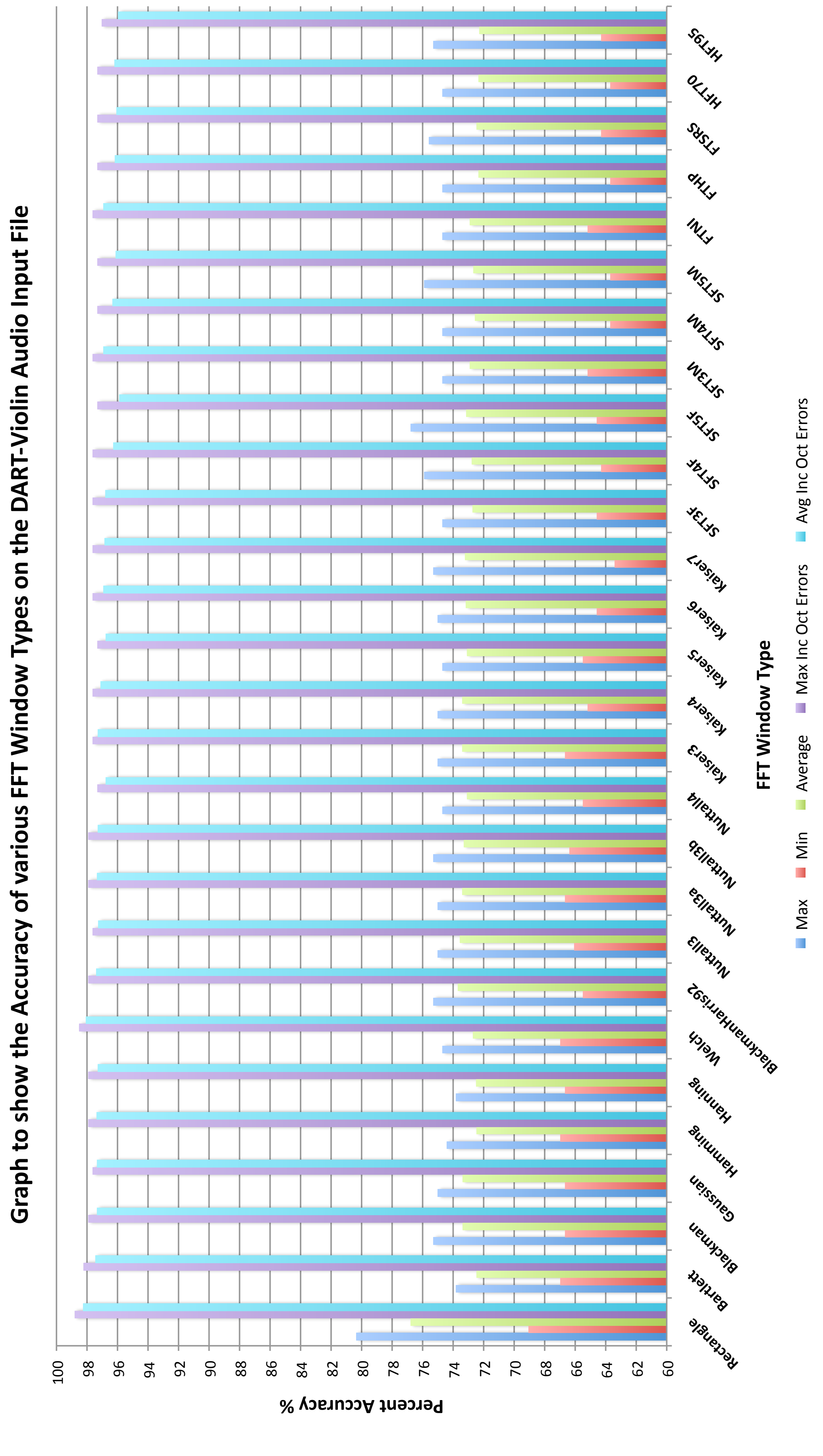


Figure 7.4: A graph to show the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-Violin audio input file, with minimum, maximum and average score values. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

FFTWindow	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
Rectangle	**80.36**	**69.05**	**76.78**	**98.81**	**98.24**	**98.81**	26.79	18.45	21.46
Bartlett	73.81	66.96	72.47	98.21	97.45	97.92	28.57	**24.11**	24.99
Blackman	75.30	66.67	73.37	97.92	97.35	97.92	28.57	22.62	23.98
Gaussian	75.00	66.67	73.38	97.62	97.35	97.62	28.57	22.62	23.97
Hamming	74.40	66.96	72.47	97.92	97.38	97.92	28.57	23.51	24.90
Hanning	73.81	66.67	72.49	97.92	97.29	97.92	28.27	**24.11**	24.80
Welch	74.70	66.96	72.70	98.51	98.06	98.51	28.57	23.81	**25.36**
BlackmanHarris92	75.30	65.48	73.69	97.92	97.38	97.92	28.87	22.62	23.69
Nuttall3	75.00	66.07	73.56	97.62	97.26	97.62	28.57	22.62	23.70
Nuttall3a	75.00	66.67	73.39	97.92	97.34	97.62	28.27	22.62	23.95
Nuttall3b	75.30	66.37	73.29	97.92	97.28	97.92	28.87	22.62	23.99
Nuttall4	74.70	65.48	73.09	97.32	96.77	97.32	27.98	22.62	23.68
Kaiser3	75.00	66.67	73.39	97.62	97.29	97.62	28.27	22.62	23.91
Kaiser4	75.00	65.18	73.39	97.62	97.11	97.62	28.87	22.62	23.73
Kaiser5	74.70	65.48	73.09	97.32	96.76	97.32	27.98	22.62	23.68
Kaiser6	75.00	64.58	73.15	97.62	96.92	97.62	28.87	22.62	23.76
Kaiser7	75.30	63.39	73.22	97.62	96.84	97.62	29.76	22.32	23.62
SFT3F	74.70	64.58	72.73	97.62	96.80	97.62	28.27	22.92	24.07
SFT4F	75.89	64.29	72.77	97.62	96.27	97.62	28.27	21.73	23.50
SFT5F	76.79	64.58	73.13	97.32	95.89	97.32	27.68	20.54	22.76
SFT3M	74.70	65.18	72.91	97.62	96.92	97.62	27.68	22.92	24.00
SFT4M	74.70	63.69	72.56	97.32	96.33	97.32	28.87	22.62	23.77
SFT5M	75.89	63.69	72.68	97.32	96.11	97.32	28.57	21.43	23.43
FTNI	74.70	65.18	72.91	97.62	96.91	97.62	27.68	22.92	24.00
FTHP	74.70	63.69	72.33	97.32	96.18	97.32	**29.17**	22.62	23.85
FTSRS	75.60	64.29	72.45	97.32	96.07	97.32	28.27	21.73	23.62
HFT70	74.70	63.69	72.34	97.32	96.19	97.32	29.17	22.62	23.85
HFT95	75.30	64.29	72.29	97.02	95.95	97.02	28.57	21.73	23.66

Table 7.4: This table details the accuracy of the DART SHS algorithm while varying the FFT Window Type on the DART Violin audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.3.4 FFT Window Accuracy Results for Piano

Graph 7.5 shows the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-Piano audio input file. Table 7.5 illustrates the actual values presented in Graph 7.5. The first 5 columns represent data presented in 7.5, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of the chosen FFT Window.

In the case of the Piano, the **Rectangle** FFT Window Type produced the highest Minimum, Maximum, Average, Maximum Including O.E, Average Including O.E, and Top + O.E, accuracy results. The Rectangle Window Type can therefore be considered the most optimal overall FFT Window Type for the Piano Audio Input File.

The **BlackmanHarris92**, **Nutall4** and **Kaiser5** Window Types produced the least accurate results, as can be seen in Table 7.5.

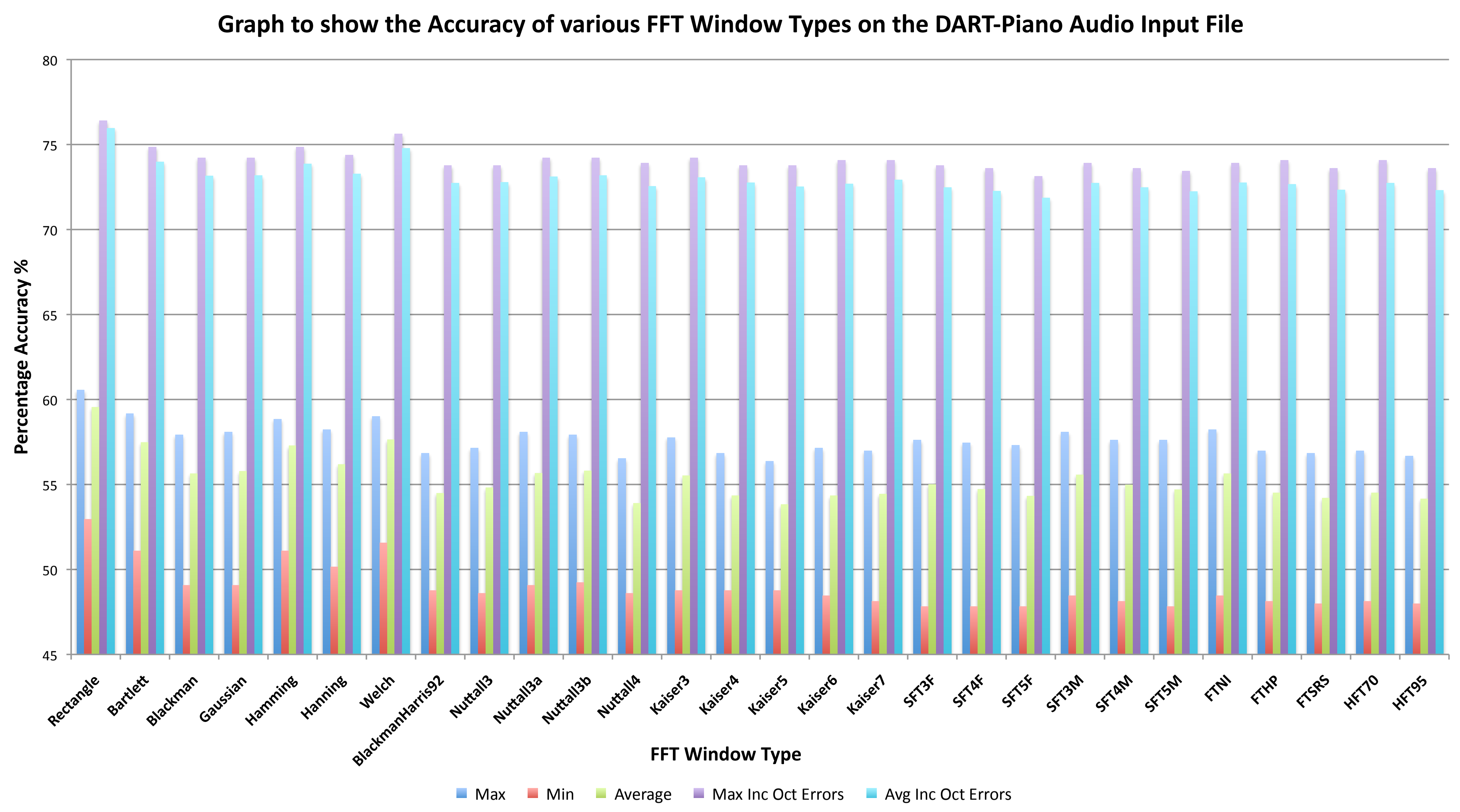


Figure 7.5: A graph to show the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-Piano audio input file, with minimum, maximum and average score values. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

FFTWindow	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
Rectangle	**60.56**	**52.95**	**59.55**	**76.40**	**75.97**	**76.40**	21.12	15.68	16.42
Bartlett	59.16	51.09	57.48	74.84	73.99	74.84	20.96	15.68	16.51
Blackman	57.92	49.07	55.64	74.22	73.16	74.07	22.20	16.15	17.52
Gaussian	58.07	49.07	55.77	74.22	73.17	74.07	22.05	15.99	17.40
Hamming	58.85	51.09	57.28	74.84	73.87	74.69	20.96	15.53	16.59
Hanning	58.23	50.16	56.18	74.38	73.27	74.22	21.58	15.99	17.09
Welch	59.01	51.55	57.64	75.62	74.77	75.62	21.74	15.99	17.13
BlackmanHarris92	56.83	48.76	54.47	73.76	72.73	73.76	**22.67**	16.93	18.25
Nuttall3	57.14	48.60	54.81	73.76	72.77	73.76	22.52	16.61	17.96
Nuttall3a	58.07	49.07	55.66	74.22	73.11	74.07	22.20	15.99	17.45
Nuttall3b	57.92	49.22	55.80	74.22	73.18	74.07	22.20	15.99	17.37
Nuttall4	56.52	48.60	53.88	73.91	72.54	73.91	22.36	**17.39**	**18.66**
Kaiser3	57.76	48.76	55.51	74.22	73.05	74.07	22.52	16.15	17.54
Kaiser4	56.83	48.76	54.35	73.76	72.76	73.76	22.36	16.93	18.41
Kaiser5	56.37	48.76	53.82	73.76	72.51	73.76	22.36	**17.39**	18.70
Kaiser6	57.14	48.45	54.33	74.07	72.69	74.07	22.36	16.61	18.36
Kaiser7	56.99	48.14	54.44	74.07	72.93	74.07	22.36	16.61	18.49
SFT3F	57.61	47.83	55.01	73.76	72.47	73.29	22.52	15.53	17.46
SFT4F	57.45	47.83	54.72	73.60	72.26	73.60	22.52	15.37	17.54
SFT5F	57.30	47.83	54.33	73.14	71.86	73.14	22.36	15.22	17.53
SFT3M	58.07	48.45	55.57	73.91	72.74	73.60	22.20	15.53	17.17
SFT4M	57.61	48.14	54.98	73.60	72.47	73.45	22.20	15.37	17.49
SFT5M	57.61	47.83	54.70	73.45	72.24	73.29	22.52	15.22	17.54
FTNI	58.23	48.45	55.64	73.91	72.75	73.60	22.20	15.37	17.11
FTHP	56.99	48.14	54.51	74.07	72.66	74.07	22.20	16.46	18.15
FTSRS	56.83	47.98	54.19	73.60	72.34	73.60	22.36	16.30	18.14
HFT70	56.99	48.14	54.52	74.07	72.73	74.07	22.20	16.46	18.21
HFT95	56.68	47.98	54.16	73.60	72.30	73.60	22.36	16.30	18.13

Table 7.5: This table details the accuracy of the DART SHS algorithm while varying the FFT Window Type on the DART Piano audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.3.5 FFT Window Accuracy Results for Tubular Bells

Graph 7.6 shows the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-TubBells audio input file. Table 7.6 illustrates the actual values presented in Graph 7.6. The first 5 columns represent data presented in Graph 7.6, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of the chosen FFT Window.

The **Rectangle** FFT Window Type produced the highest Maximum Including O.E, Average Including O.E, and Top + O.E, accuracy results. Due to the high level of overtones expected from the Tubular Bells, the Rectangle Window Type can therefore be considered the most optimal overall FFT Window Type for the Tubular Bells Audio Input File.

While other FFT Window Types produced higher Minimum, Maximum and Average accuracy scores in comparison to the Rectangle Window's 0% Maximum Accuracy, the Maximum Accuracy (with both note and octave being correct) achieved was only 3.18%, by the Kaiser5, Kaiser6, and Kaiser7 Window Types. The **FTSRS**, **HFT70**, and **HFT95** Window Types produced the highest Minimum accuracy results, however the results were still extremely low, at just 1.82%. The highest Average accuracy scores was achieved by the **Nutall4** and **Kaiser5** Window Types. If the correct note and octave is of paramount importance when using the SHS algorithm with the Tubular Bells, the **Kaiser5** algorithm will give the most optimal results, although with an extremely low success rate.

The **BlackmanHarris92**, **Nutall4** and **Kaiser5** Window Types produced the least accurate results, as can be seen in table 7.6.

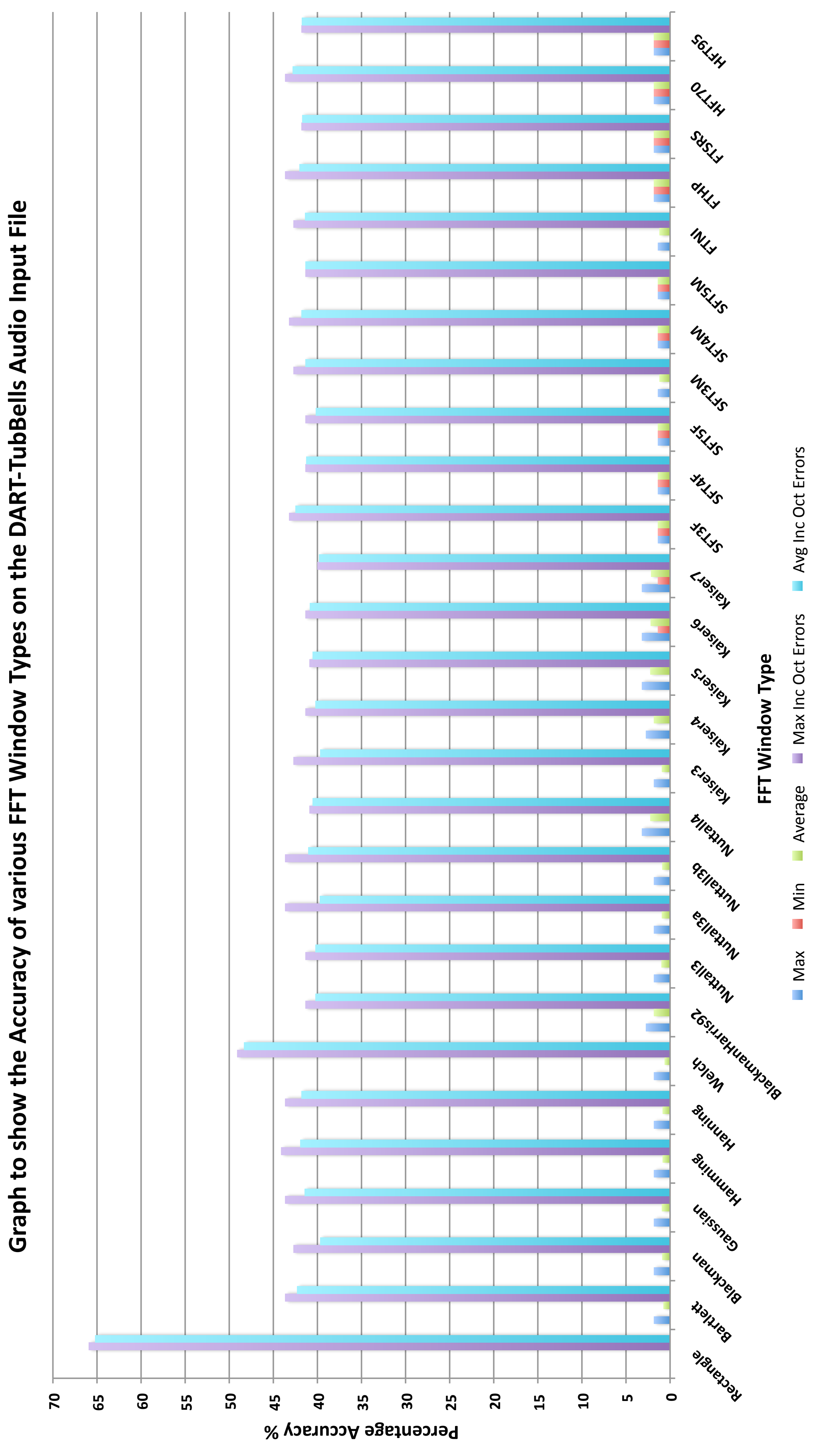


Figure 7.6: A graph to show the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-TubBells audio input file, with minimum, maximum and average score values. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

FFTWindow	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
Rectangle	0.00	0.00	0.00	**65.91**	**65.20**	**65.91**	**65.91**	**60.00**	**65.20**
Bartlett	1.82	0.00	0.76	43.64	42.28	42.27	43.64	40.00	41.52
Blackman	1.82	0.00	0.90	42.73	39.69	39.55	42.73	37.73	38.79
Gaussian	1.82	0.00	0.92	43.64	41.45	41.36	43.64	39.55	40.53
Hamming	1.82	0.00	0.83	44.09	41.93	41.82	44.09	40.00	41.10
Hanning	1.82	0.00	0.84	43.64	41.82	41.82	43.64	40.00	40.98
Welch	1.82	0.00	0.60	49.09	48.31	49.09	49.09	42.27	47.71
BlackmanHarris92	2.73	0.00	1.82	41.36	40.22	40.45	41.36	37.73	38.40
Nuttall3	1.82	0.00	0.98	41.36	40.22	40.45	41.36	38.18	39.24
Nuttall3a	1.82	0.00	0.92	43.64	39.74	39.55	43.64	37.73	38.81
Nuttall3b	1.82	0.00	0.88	43.64	41.02	40.91	43.64	39.09	40.14
Nuttall4	3.18	0.00	**2.24**	40.91	40.54	40.91	40.00	36.82	38.30
Kaiser3	1.82	0.00	0.93	42.73	39.65	39.55	42.73	37.73	38.72
Kaiser4	2.73	0.00	1.82	41.36	40.21	40.45	41.36	37.73	38.39
Kaiser5	**3.18**	0.00	**2.24**	40.91	40.53	40.91	40.00	36.82	38.29
Kaiser6	**3.18**	1.36	2.20	41.36	40.84	41.36	39.55	36.36	38.64
Kaiser7	**3.18**	1.36	2.13	40.00	39.81	40.00	38.64	35.91	37.68
SFT3F	1.36	1.36	1.36	43.18	42.50	43.18	41.82	38.18	41.14
SFT4F	1.36	1.36	1.36	41.36	41.27	41.36	40.00	38.64	39.91
SFT5F	1.36	1.36	1.36	41.36	40.16	41.36	40.00	38.64	38.80
SFT3M	1.36	0.00	1.18	42.73	41.36	41.82	42.73	39.55	40.18
SFT4M	1.36	1.36	1.36	43.18	41.81	43.18	41.82	37.73	40.45
SFT5M	1.36	1.36	1.36	41.36	41.33	41.36	40.00	38.64	39.96
FTNI	1.36	0.00	1.18	42.73	41.38	41.82	42.73	39.55	40.20
FTHP	1.82	**1.82**	1.82	43.64	42.01	43.64	41.82	36.36	40.19
FTSRS	1.82	**1.82**	1.82	41.82	41.73	41.82	40.00	37.27	39.91
HFT70	1.82	**1.82**	1.82	43.64	42.80	43.64	41.82	36.36	40.98
HFT95	1.82	**1.82**	1.82	41.82	41.75	41.82	40.00	38.18	39.93

Table 7.6: This table details the accuracy of the DART SHS algorithm while varying the FFT Window Type on the DART Tubular Bells audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.3.6 FFT Window Accuracy Results for Distorted Guitar

Graph 7.7 shows the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-DistortG audio input file. Table 7.7 shows the values presented in Graph 7.7. The first 5 columns represent data presented in Graph 7.7, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of the chosen FFT Window.

The **SFT3M** Window Type produced the highest Maximum accuracy results. The highest Minimum accuracy value of 30.22% was achieved by the **Rectangle** and **Bartlett** Window Types, and the highest average was achieved by the Welch Window Type, at 46.67%. When ignoring octave errors, the **Nutall3**, **Nutall4**, and **Kaiser5** FFT Window Types gave the Maximum Accuracy, with the Blackman window giving the highest Average Including octave errors.

Due to the high level of both odd and even harmonics in the Distorted Guitar Audio Input File, it is more difficult to choose a clear 'winner' in terms of accuracy, however the **Nutall4** FFT Window had the highest overall accuracy levels when accumulated across the first five columns in Table 7.7. This means that when adding together the accuracy scores of each of the first 5 columns in Table 7.7, the **Nutall4** Window Type gave the highest overall score (or average), as well as the highest Max and Max Inc. O.E values, making it the most suitable FFT Window Type when analysing the distorted guitar audio input file. However when not allowing for octave errors (looking at only the first three columns in Table 7.7, the highest scoring FFT Window Type is the **SFT3M** type.

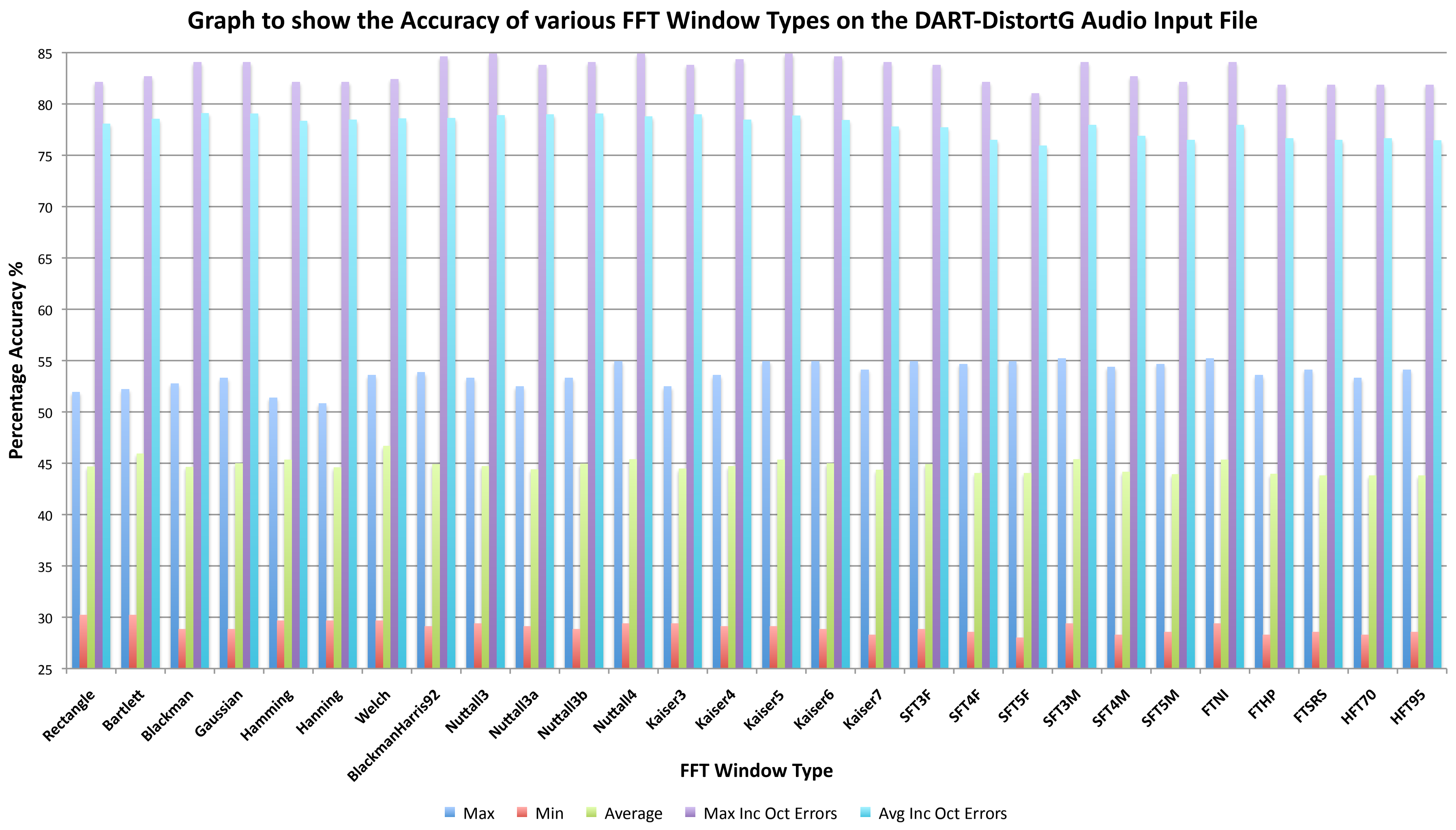


Figure 7.7: A graph to show the relative accuracy of the SHS algorithm when modifying the FFT Window Type using the DART-TubBells audio input file, with minimum, maximum and average score values. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

FFTWindow	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
Rectangle	51.92	**30.22**	44.65	82.14	78.09	82.14	**42.03**	30.22	33.44
Bartlett	52.20	**30.22**	45.92	82.69	78.53	82.69	40.66	30.22	32.61
Blackman	52.75	28.85	44.64	84.07	**79.10**	84.07	41.76	31.32	34.46
Gaussian	53.30	28.85	45.00	84.07	79.06	84.07	41.76	30.77	34.06
Hamming	51.37	29.67	45.33	82.14	78.33	82.14	41.48	30.49	33.00
Hanning	50.82	29.67	44.59	82.14	78.46	82.14	**42.03**	**31.32**	33.88
Welch	53.57	29.67	**46.67**	82.42	78.56	82.42	41.76	28.85	31.90
BlackmanHarris92	53.85	29.12	44.91	84.62	78.62	84.62	41.21	30.77	33.70
Nuttall3	53.30	29.40	44.71	**84.89**	78.91	84.89	41.21	**31.32**	34.20
Nuttall3a	52.47	29.12	44.41	83.79	78.98	83.79	**42.03**	**31.32**	**34.57**
Nuttall3b	53.30	28.85	44.96	84.07	79.05	84.07	41.76	30.77	34.08
Nuttall4	54.95	29.40	45.37	**84.89**	78.76	84.89	41.48	29.95	33.40
Kaiser3	52.47	29.40	44.45	83.79	78.99	83.79	41.76	**31.32**	34.54
Kaiser4	53.57	29.12	44.69	84.34	78.47	84.34	41.48	30.77	33.78
Kaiser5	54.95	29.12	45.35	**84.89**	78.84	**84.89**	41.48	29.95	33.49
Kaiser6	54.95	28.85	44.97	84.62	78.42	84.62	41.48	29.67	33.45
Kaiser7	54.12	28.30	44.34	84.07	77.81	84.07	41.48	29.95	33.47
SFT3F	54.95	28.85	44.90	83.79	77.72	83.79	41.21	28.85	32.82
SFT4F	54.67	28.57	44.04	82.14	76.51	82.14	40.93	27.20	32.47
SFT5F	54.95	28.02	44.02	81.04	75.93	81.04	41.48	26.10	31.91
SFT3M	**55.22**	29.40	45.36	84.07	77.94	84.07	41.48	28.85	32.58
SFT4M	54.40	28.30	44.15	82.69	76.88	82.69	41.21	28.30	32.73
SFT5M	54.67	28.57	43.93	82.14	76.47	82.14	41.21	27.47	32.54
FTNI	55.22	29.40	45.36	84.07	77.95	84.07	41.48	28.85	32.59
FTHP	53.57	28.30	43.95	81.87	76.65	81.87	41.48	28.30	32.71
FTSRS	54.12	28.57	43.81	81.87	76.50	81.87	40.93	27.47	32.69
HFT70	53.30	28.30	43.79	81.87	76.65	81.87	41.48	28.57	32.86
HFT95	54.12	28.57	43.78	81.87	76.45	81.87	40.93	27.20	32.66

Table 7.7: This table details the accuracy of the DART SHS algorithm while varying the FFT Window Type on the DART Distorted Guitar audio file. The top values in each column are highlighted bold. ‘O.E’ stands for *Octave Error*.

7.3.7 FFT Window Accuracy Results for All Audio Files

The FFT Window accuracy was also measured across all window types and audio input files, in order find the optimal setting *overall*. Due to the variation in results across the different audio input files, this results in a compromise because different Window Types perform better with different instruments - however this aids in extracting a generic optimal parameter set.

Graph 7.8 shows the relative accuracy of the SHS algorithm when modifying the FFT Window Type across *all* audio input files. Table 7.8 shows the values presented in Graph 7.8. The first 5 columns represent data already presented in Graph 7.8, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of the chosen FFT Window.

The **Welch** FFT Window Type produced the highest Maximum accuracy results. The highest Minimum accuracy value of 1.82% was achieved by the **FTSRS**, **HFT70**, and **HFT95** Window Types, and the highest average was achieved by the **Rectangle** Window Type, at 53.07%.

The **Rectangle** FFT Window had the highest overall accuracy levels when accumulated across the first five columns in Table 7.8. This means that when adding together the accuracy scores of each of the first 5 columns in Table 7.8, the **Rectangle** Window Type gave the highest overall score (or average), as well as the Average Score, both with and without octave errors. This makes it the most suitable FFT Window Type when analysing all Audio Input Files at the same time.

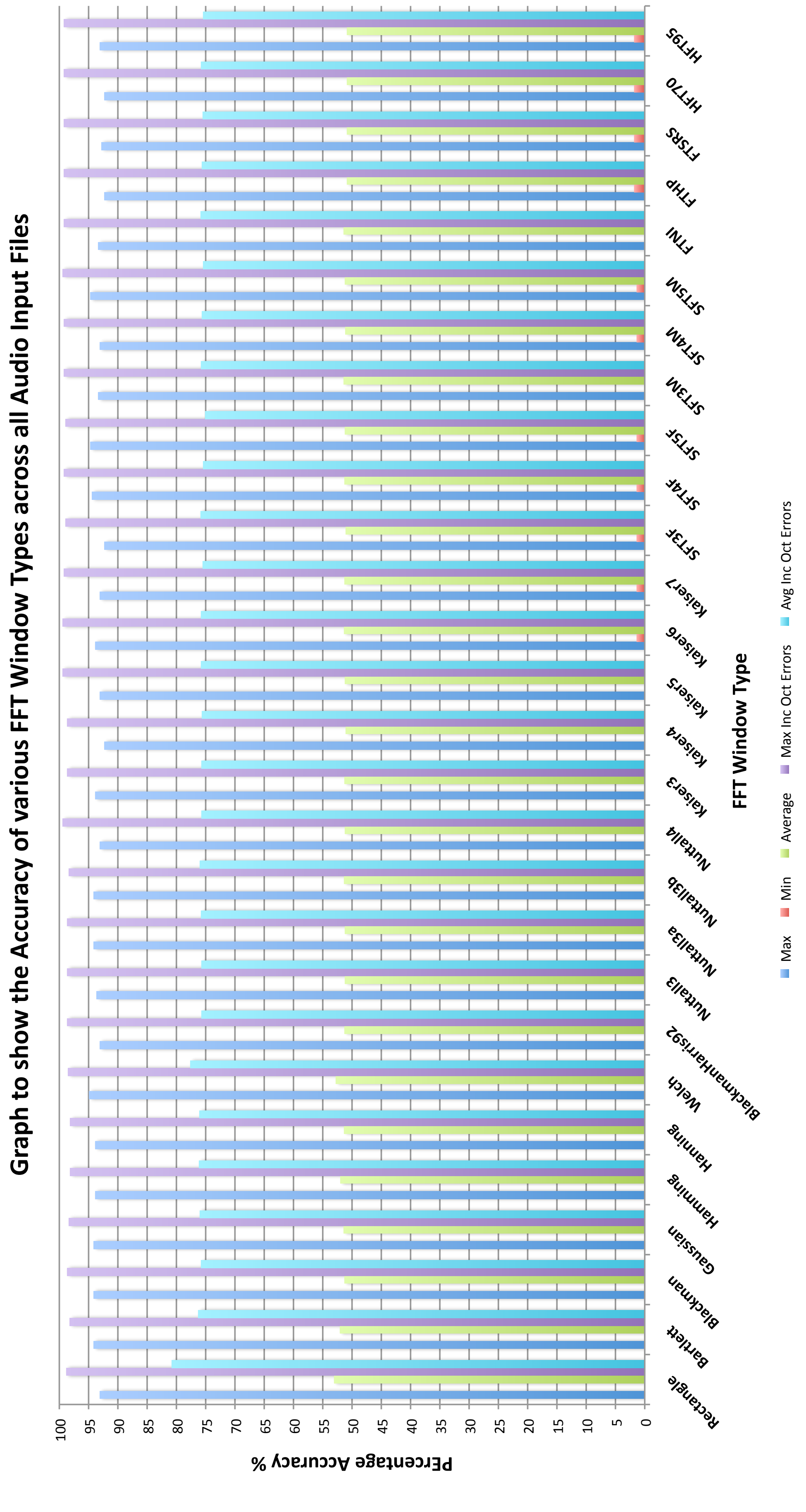


Figure 7.8: A graph to show the relative accuracy of the SHS algorithm when modifying the FFT Window Type, with minimum, maximum and average score values across all audio input files. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

FFTWindow	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
Rectangle	93.09	0.00	**53.07**	98.81	**80.79**	97.87	65.91	4.79	27.72
Bartlett	94.15	0.00	52.08	98.21	76.27	98.14	43.64	3.99	24.19
Blackman	94.15	0.00	51.29	98.67	75.79	98.67	42.73	4.52	24.50
Gaussian	94.15	0.00	51.42	98.40	76.03	98.14	43.64	3.99	24.61
Hamming	93.88	0.00	51.96	98.14	76.14	98.14	44.09	4.26	24.18
Hanning	93.88	0.00	51.36	98.14	76.06	98.14	43.64	4.26	24.70
Welch	**94.95**	0.00	52.78	98.51	77.61	98.14	49.09	3.19	24.83
BlackmanHarris92	93.09	0.00	51.30	98.67	75.68	98.67	41.36	5.32	24.38
Nuttall3	93.62	0.00	51.20	98.67	75.73	98.67	41.36	5.05	24.53
Nuttall3a	94.15	0.00	51.25	98.67	75.76	98.67	43.64	4.52	24.51
Nuttall3b	94.15	0.00	51.37	98.40	75.97	98.40	43.64	4.26	24.61
Nuttall4	93.09	0.00	51.24	**99.47**	75.75	99.20	41.48	6.12	24.51
Kaiser3	93.88	0.00	51.29	98.67	75.73	98.67	42.73	4.79	24.44
Kaiser4	92.29	0.00	51.09	98.67	75.62	98.67	41.48	6.12	24.53
Kaiser5	93.09	0.00	51.23	**99.47**	75.75	99.20	41.48	6.12	24.52
Kaiser6	93.88	1.36	51.39	**99.47**	75.82	**99.47**	41.48	5.59	24.43
Kaiser7	93.09	1.36	51.25	99.20	75.54	99.20	41.48	5.85	24.29
SFT3F	92.29	1.36	51.06	98.94	75.88	98.94	41.82	6.65	24.82
SFT4F	94.41	1.36	51.26	99.20	75.45	99.20	40.93	4.79	24.19
SFT5F	94.68	1.36	51.25	98.94	75.06	98.94	41.48	4.26	23.81
SFT3M	93.35	0.00	51.42	99.20	75.82	99.20	42.73	5.85	24.39
SFT4M	93.09	1.36	51.13	99.20	75.66	99.20	41.82	6.12	24.53
SFT5M	94.68	1.36	51.24	**99.47**	75.44	**99.47**	41.21	4.79	24.20
FTNI	93.35	0.00	51.43	99.20	75.83	99.20	42.73	5.85	24.39
FTHP	92.29	**1.82**	50.87	99.20	75.64	99.20	41.82	**6.91**	24.77
FTSRS	92.82	**1.82**	50.88	99.20	75.48	99.20	40.93	6.38	24.59
HFT70	92.29	**1.82**	50.84	99.20	75.78	99.20	41.82	**6.91**	**24.94**
HFT95	93.09	**1.82**	50.87	99.20	75.44	99.20	40.93	6.12	24.57

Table 7.8: This table details the accuracy of the DART SHS algorithm while varying the FFT Window Type across all Audio Input Files. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.3.8 FFT Window Accuracy Summary

A summary of the FFT Window results presented so far is given in Table 7.9. This table shows, for each audio input file, the most optimal FFT Window Types that give:

- **A** - The highest overall Minimum, Maximum and Average accuracy values
- **B** - The highest overall Maximum and Average Accuracy values when ignoring Octave Errors
- **Most Optimal Overall FFT Window** - The highest overall accuracy when taking both factors into account

The FFT Window that consistently produced the most accurate results across all the variables was the **Rectangle** FFT Window Type.

Audio Input File	***A***	***B***	**Most Optimal Overall FFT Window**
Acoustic Guitar	Welch	SFT5M	**Welch**
Oboe	Rectangle	Rectangle	**Rectangle**
Violin	Rectangle	Rectangle	**Rectangle**
Piano	Rectangle	Rectangle	**Rectangle**
Tubular Bells	Kaiser5	Rectangle	**Rectangle**
Distorted Guitar	SFT3M	Nutall3	**Nutall4**
All Audio Files	Welch	Rectangle	**Rectangle**

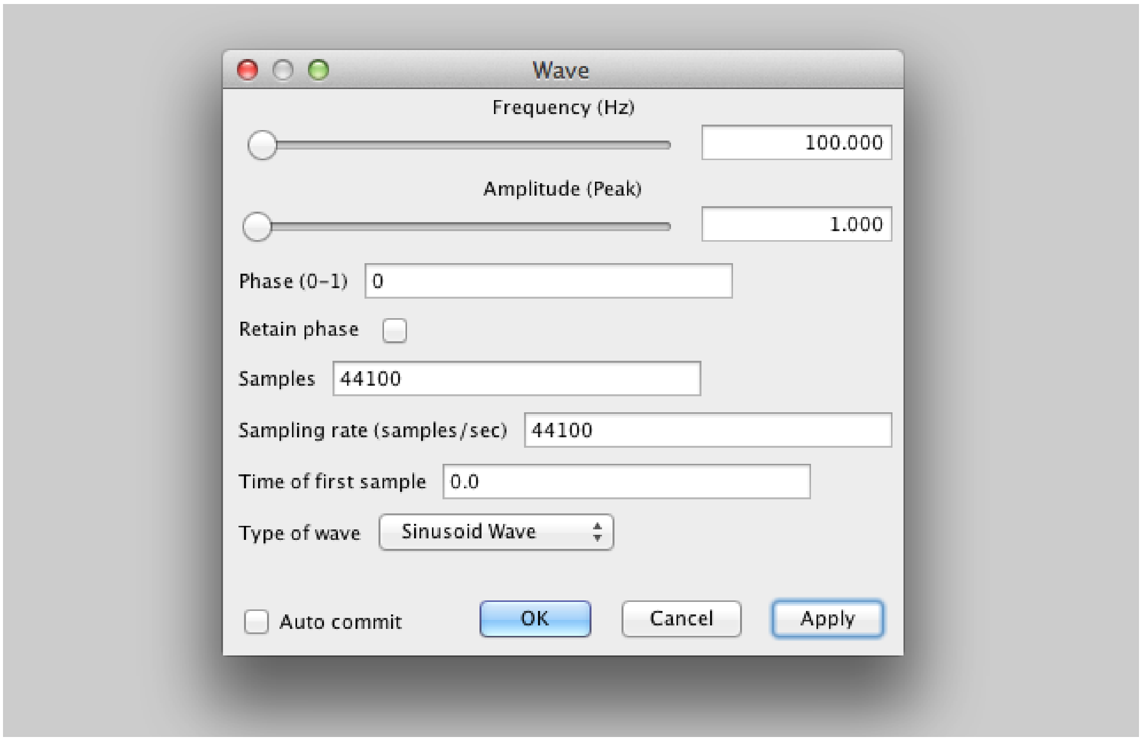
Table 7.9: This table shows a summary of the FFT Windows that give the most accurate results for each Audio Input File.

In order to understand why the Rectangle Window Type scored so well, it is helpful to review what changing the FFT Window actually achieves.

The Fast Fourier Transform in DART is the Fourier Transform of a finite chunk of time-amplitude audio data points, representing the frequency composition of the (time-based) signal. FFT algorithms work on the basis that the finite data set is exactly one period of a periodic signal. However in the real world, given any signal other than a perfect phase sine wave, the signal will not represent one perfect period of the signal. If there is a signal that is not a sine wave with an integral number of cycles, there is a discontinuity between the last and first sample, and this truncation of the waveform includes ‘false harmonics’ when compared to the original continuous-time signal. These harmonics cause *spectral leakage*, resulting in the signal energy smearing out over a wide frequency range in the FFT, when it should be in a narrow frequency range.

Windowing Functions lower the amplitudes at the beginning and the end of the (time-amplitude based) audio chunk, to reduce the harmonics caused by this discontinuity, ‘smoothing out’ what would otherwise be transients at the edges of the sampling window. This will make the endpoints of the waveform meet and therefore result in a continuous waveform without sharp transitions.

This process can be illustrated in Triana using the Signal Processing units. The **`Wave`** unit (Figure 7.9) can be used to generate a waveform that can be displayed, analysed or modified by subsequent units. **`Wave`** outputs a **`SampleSet`** containing the data and the sampling rate.


Wave

Frequency (Hz): 100.000

Amplitude (Peak): 1.000

Phase (0-1): 0

Retain phase: ☐

Samples: 44100

Sampling rate (samples/sec): 44100

Time of first sample: 0.0

Type of wave: Sinusoid Wave

☐ Auto commit

Figure 7.9: The GUI of the Triana Wave unit

Consider a 100 Hz Sine wave with amplitude 1, that is periodic. The Triana workflow created in Figure 7.10 allows for the display of the the time amplitude plot of this, as seen in Figure 7.11. The second Triana workflow displayed in Figure 7.12 allows the FFT of the wave to be plotted and displayed. The resulting FFT would show a narrow peak at 100 Hz in the frequency axis, as shown in Figure 7.14.

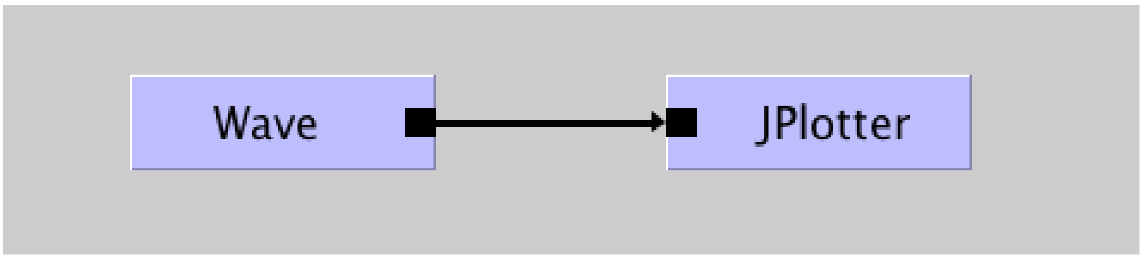


Figure 7.10: A simple Triana Workflow to display the time-amplitude graph of a 100Hz Sine wave

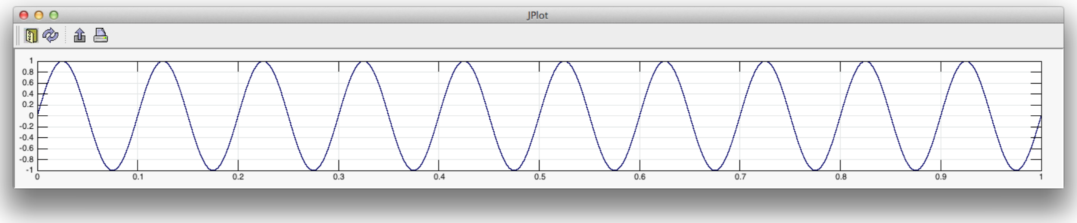


Figure 7.11: A time-amplitude graph of a perfectly periodic 100Hz Sine Wave

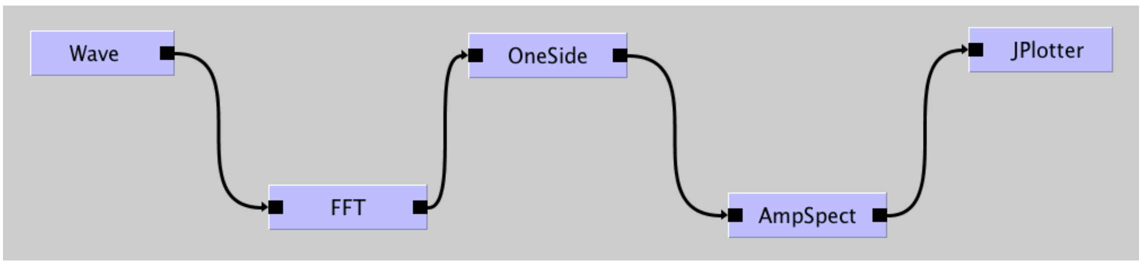


Figure 7.12: A Triana Workflow to display the FFT graph of a wave generated by the Wave unit

However when a non integer sine wave is analysed (such as a 100.533Hz sine wave as displayed in Figure 7.13), these false harmonics and spectral leakage creates noise and smears the energy out over a wider range across the frequency spectrum, as displayed in Figure 7.15. This uses a Rectangle window in the FFT unit, which is essentially the same as applying no window to the raw data.

When switching the FFT Window Type from Rectangle to Hamming, a sharper and more distinct spike at 100.533Hz is displayed, as shown in Figure 7.16. Figures 7.17 and 7.18 display the FFT of the 100.533Hz sine wave while using the Blackman-Harris and

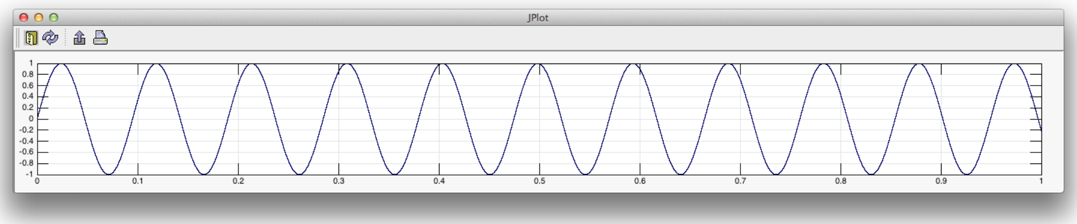


Figure 7.13: A time-amplitude graph of a 100.533Hz Sine wave. The graph does not show a perfect integer number of peaks and troughs

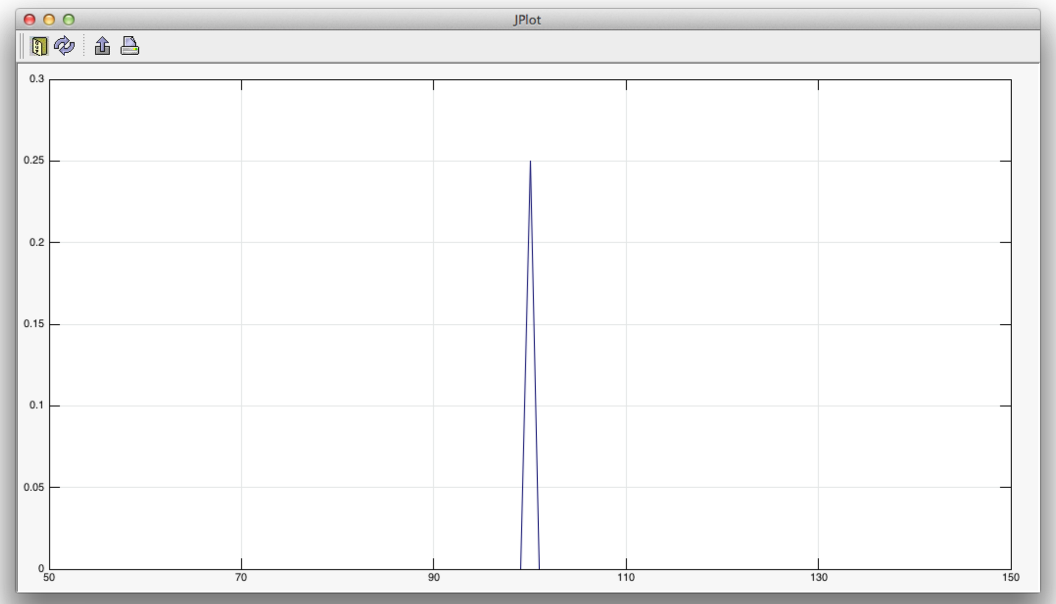


Figure 7.14: An FFT graph of a perfectly periodic 100Hz Sine Wave

Welch FFT Window Types respectively. As the FFT Window Type controls the shape of the window function applied to the raw data, each type gives a slightly different result. An excellent overview of many of the different FFT Window types and their corresponding is presented in the seminal paper [32], [119], and a shorter overview is given in [120], including a useful comparison diagram displayed in Figure 7.19 that shows the main lobe of the frequency response for various different FFT Window types, in detail. This could be is helpful when selecting a FFT Window Type for a particular application.

When analysing the DART audio input files, the FFT unit also used *zero-padding* to improve the efficiency of the algorithm, which also has an effect on the accuracy of the results when analysing a non-periodic wave. Figure 7.20 shows the FFT of a 100.533.Hz

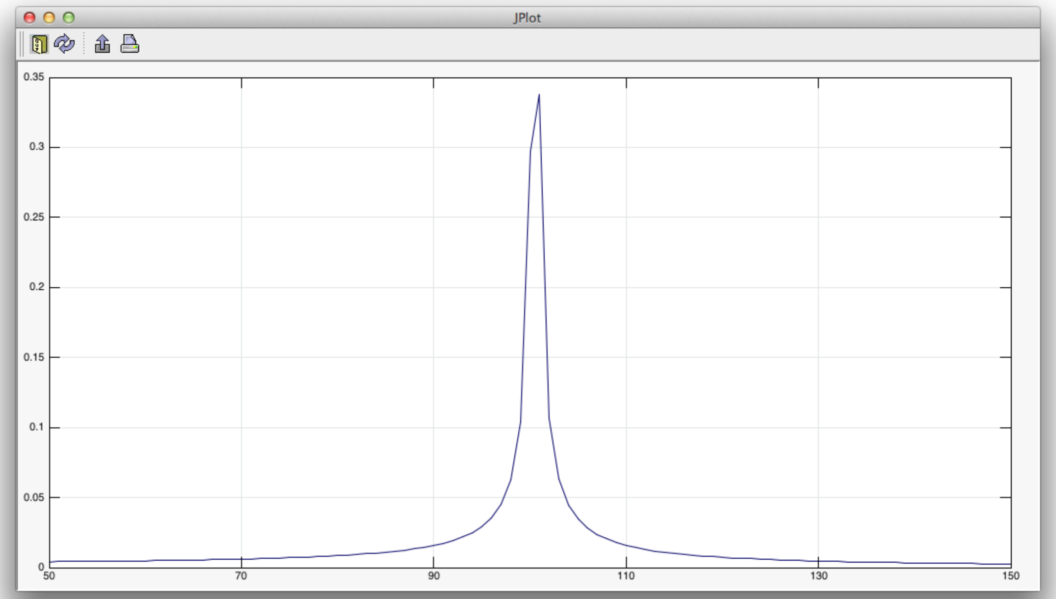


Figure 7.15: An FFT graph of a 100.533Hz Sine wave. Due to the non-integer pitch and use of the rectangle window, spectral leakage causes noise around the 100.533Hz frequency

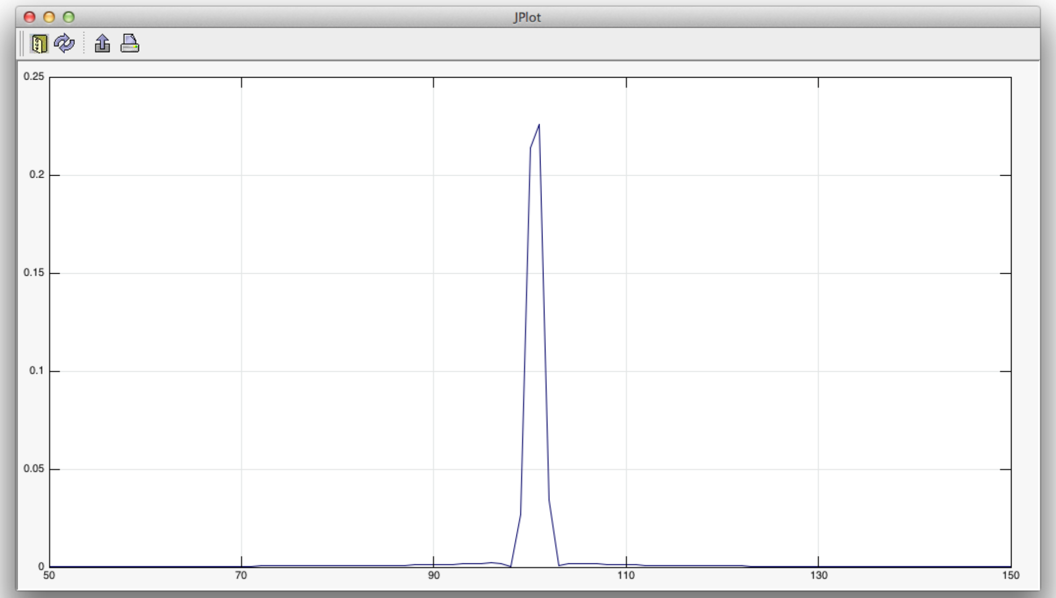


Figure 7.16: An FFT graph of a 100.533Hz Sine wave using the Hamming FFT Window Type. This results in a much cleaner frequency spike in comparison to the Rectangle FFT Window

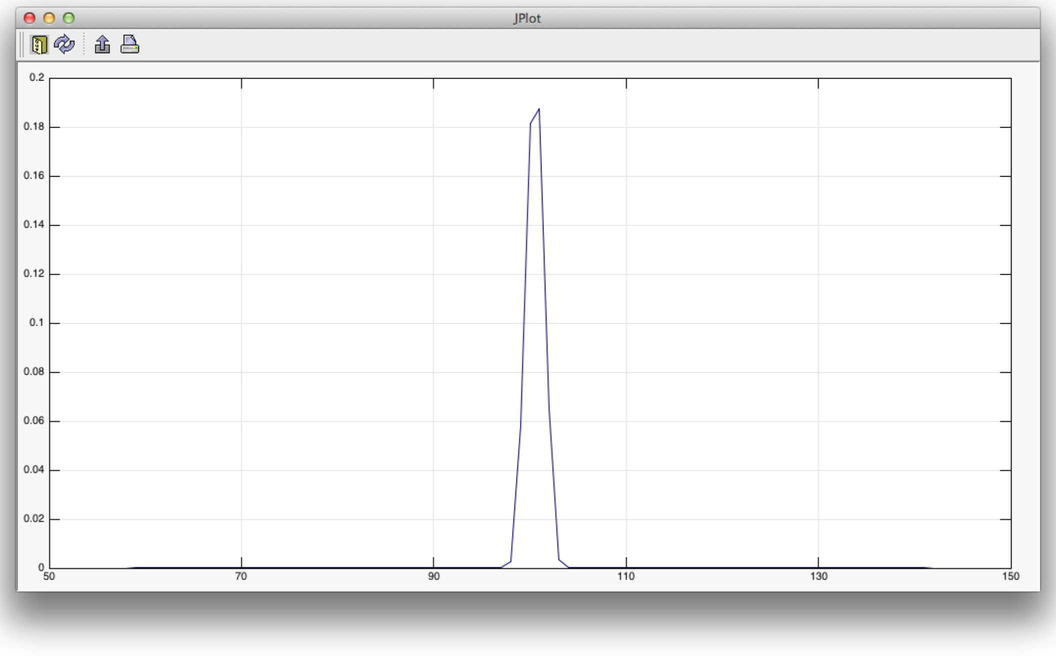


Figure 7.17: An FFT graph of a 100.533Hz Sine wave using the Blackman-Harris FFT Window Type

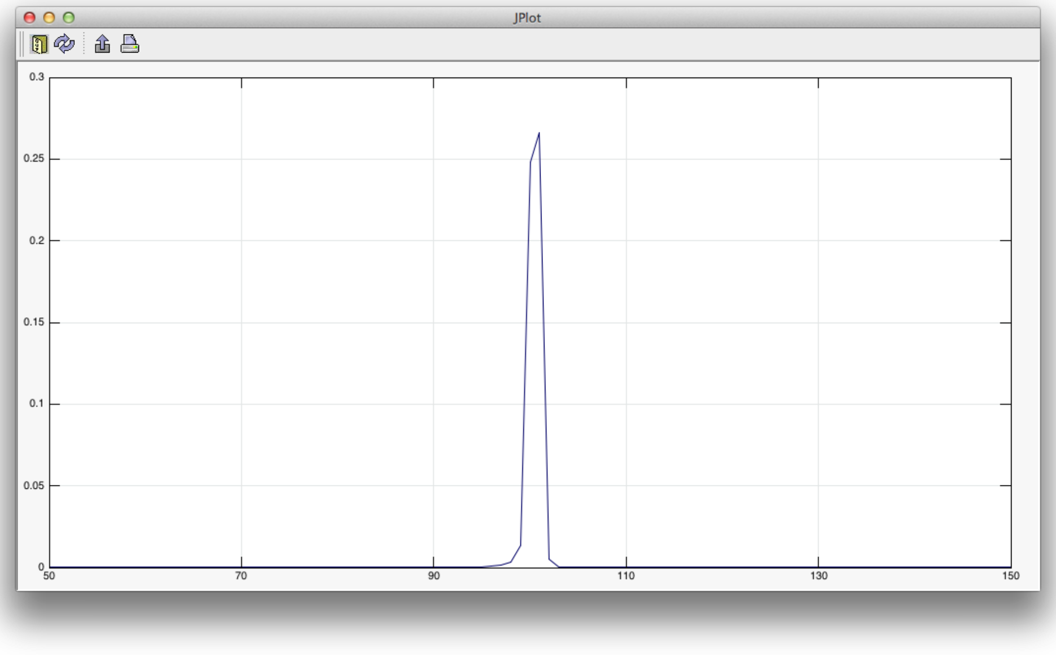


Figure 7.18: An FFT graph of a 100.533Hz Sine wave using the Welch FFT Window Type

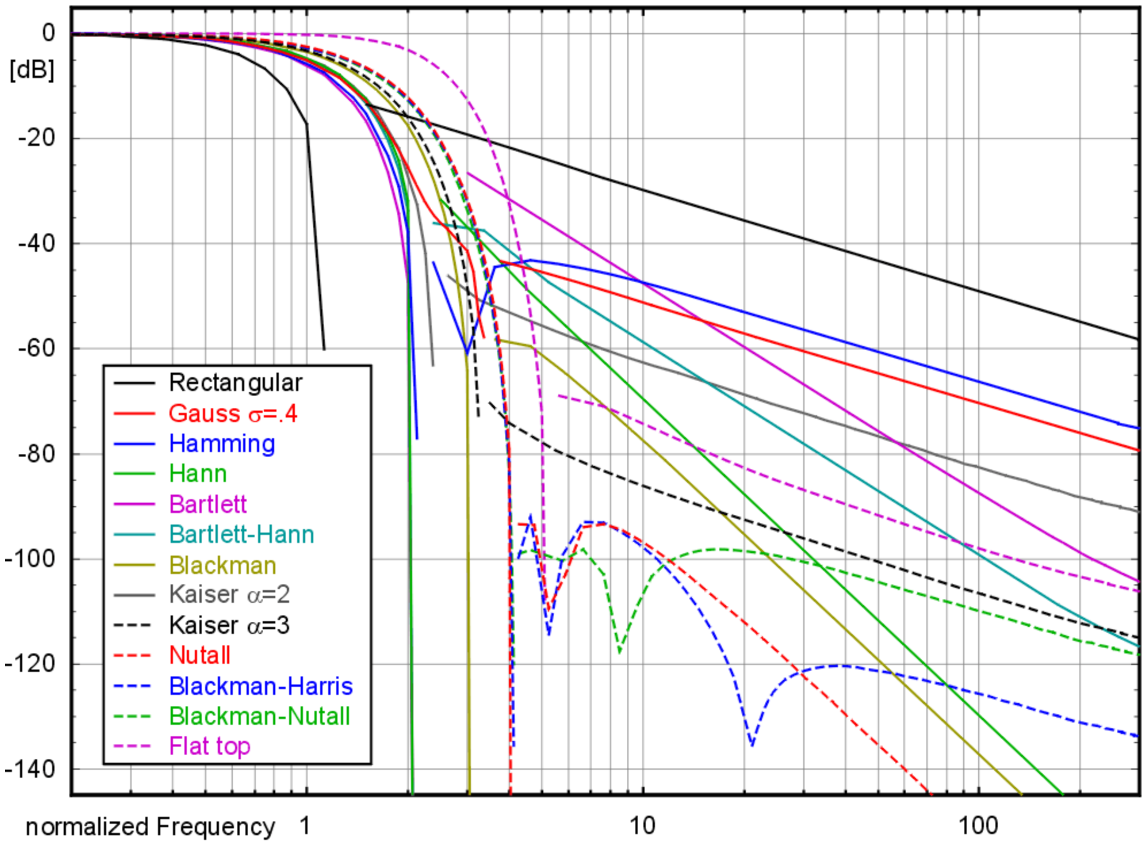


Figure 7.19: The graph shows a comparison of different window functions, showing only the main lobe of the window's frequency response in detail. Beyond that only the envelope of the sidelobes is shown to reduce clutter. This graph was used as part of the Wikimedia Commons Library

sine wave using the Rectangle FFT Window type and with *Zero Padding* ticked in the FFT Window GUI; this produces some *residue* noise either side of the 100.533Hz spike, however does improve the relative sharpness of the spike. Zero padding is essentially a frequency interpolation of the resulting spectrum and is done in the time domain to increase the number of samples over which the FFT is calculated. This gives an increased spectral resolution (more points in the frequency domain), and also a *higher fundamental amplitude peak* then the without Zero-Padding, as can be seen in Figure 7.20.

Returning back to the results presented in Table 7.9, the FFT Window that consistently produced the most accurate results across all the variables was the **Rectangle** FFT Window Type. The rational behind this result is more logical when recalling that the FFT window size was set by the **LoadSound** unit and was 0.5 seconds (22050 samples) long. Each note played for 0.5 seconds and so using the Rectangle FFT Window Type allowed for the maximum 'amount of signal' to be analysed by the Pitch Detection unit -

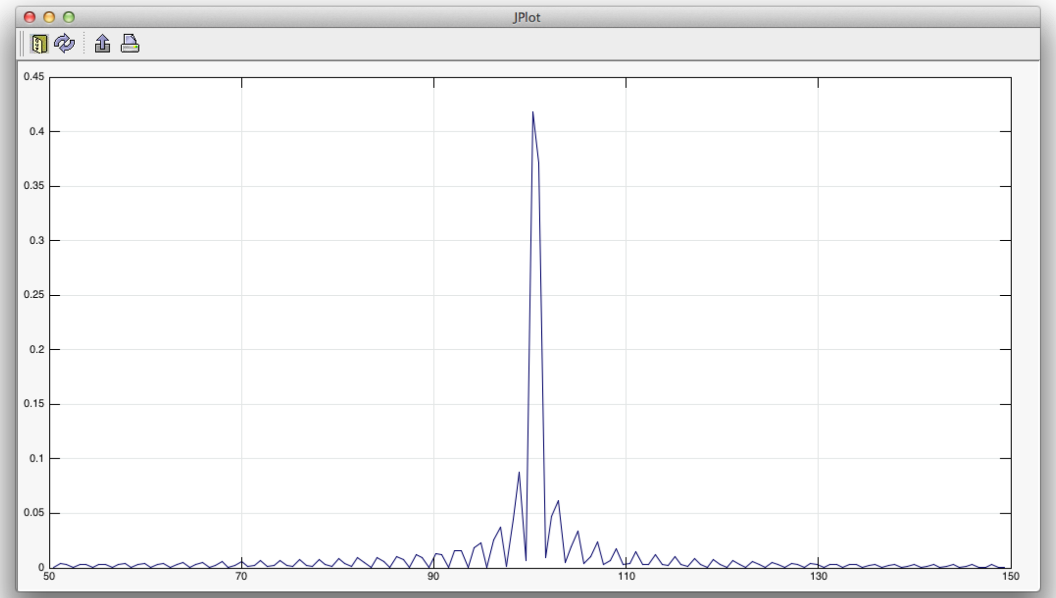


Figure 7.20: An FFT graph of a 100.533Hz Sine wave using the Rectangle FFT Window Type with Zero-Padding enabled

the complete 0.5 second long chunk of audio.

After the 22050 samples had been analysed, a new note was played. There was no polyphony, reverb effect, or 'residue' from the previous note, from each 0.5 chunk to the next, thus it seems appropriate that the Rectangle FFT Window type gives the highest accuracy, especially when adding the effect of the higher amplitude of the harmonic peak from the Zero Padding effect.

If each note were 1 second long and the chunk of audio was still 0.5 seconds, the spectral leakage would have played a bigger role and as such would have created more need for the other FFT Window Types to be used.

7.4 Number Of Harmonics Accuracy Results

This section displays the accuracy levels of the varying Number Of Harmonics (from 1-32)[3]. Each subsection looks at the results *per audio file*, with the result across all audio files displayed in the final sub-section 7.4.7.

Each results set consists of a graph displaying the data, as well as a table containing the exact values in the graph, and more information based on the Minimum, Maximum and Average Octave Error values. The Top Value + Octave Error value is also given.

7.4.1 Number Of Harmonics Results for Acoustic Guitar

Graph 7.21 shows the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics analysed by the DART SHS algorithm from 1-32, using the DARTAcousticG.wav audio input file. The yellow line indicates the optimum number of harmonics - i.e. the minimum number of harmonics that give the highest (or equal to the highest) accuracy for all accuracy measurements, reducing unnecessary processing. The orange dotted line indicates the minimum number of harmonics, above which only a minor (or statistically insignificant) increase in accuracy is gained.

Table 7.10 shows the values presented in Graph 7.21. The rows featuring the optimum value(s) whereby no further level of accuracy is gained, are highlighted in bold. The first 5 columns represent data presented in Graph 7.21, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of each number of harmonics.

In general, the accuracy results for the acoustic guitar are all high, especially in comparison to the other instruments analysed. Graph 7.21 shows that while the greatest accuracy is discovered when analysing **14** Harmonics, that looking at more than **5** harmonics does not give a significant improvement in accuracy. This is because the acoustic guitar generates a relatively pure pitch, with a stronger fundamental frequency than the other instruments. Only 2 harmonics are required in order to normalise the minimum accuracy value, with a ‘relatively large’ increase (2%) in the overall accuracy from 1 to 2

[3]While the analysing less than 4 or 5 harmonics would predictably create less accurate results, they were included as a baseline to test and verify the validity of the SHS algorithm, and thus the rest of the results.

harmonics. However, even with just 1 harmonic, the accuracy is still high at 85%.

When considering the average accuracy including octave mistakes, changing from 1 harmonic to 15 only increases the accuracy by less than 1%. This indicates a low level of pitch mistakes, with the majority of mistakes occurring due to octave errors.

To explain these results, it is useful to recap on the behaviour of a plucked string, as previously introduced in the Background and Design chapters. The fundamental frequency of a plucked string is determined by the density, length and tension of the string[4]. When the Number Of Harmonics in the DART SHS algorithm is set to *1*, the fundamental frequency alone is analysed, and no harmonics are considered.

However, a string vibrates in a complex harmonic pattern; every time a string is plucked a specific set of frequencies resonate based on the harmonic series (integer multiples of the fundamental frequency) with each harmonic (or overtone) becoming quieter the higher it is. Due to the physical nature of the strings, the higher up the overtones go, the more out of tune they are to the fundamental, which would require the SHS algorithm to look at a higher Number Of Harmonics in order to correctly find the correct pitch and octave.

A plucked string will vibrate in all of these possible resonant modes simultaneously, creating energy at all of the corresponding frequencies. Each mode of vibration (and thus each frequency) will have a different amplitude. With a guitar string the longer segments of string have more freedom to vibrate. The acoustic guitar samples were relatively free from a high number of harmonics - and hence only a small Number Of Harmonics were required to ascertain the correct notes. The SHS algorithm should therefore be able to detect most of the pitches, expect in the lowest range of notes, which is supported by the results shown.

The acoustic guitar audio was presented at 4 different velocities (as were the other 5 instruments analysed in these experiments). Different velocities can trigger a different number of harmonic overtones, and change the relative intensity of the different harmonics. As discussed, the overtones in a guitar string all sum to harmonically related frequencies, and results in a periodic wave having the fundamental frequency. However, there is clearly still a lot of extra fourth and fifth harmonics produced which provide a few extra percent of accuracy.

[4]The fundamental wavelength is twice the length of the vibrating part of the string.

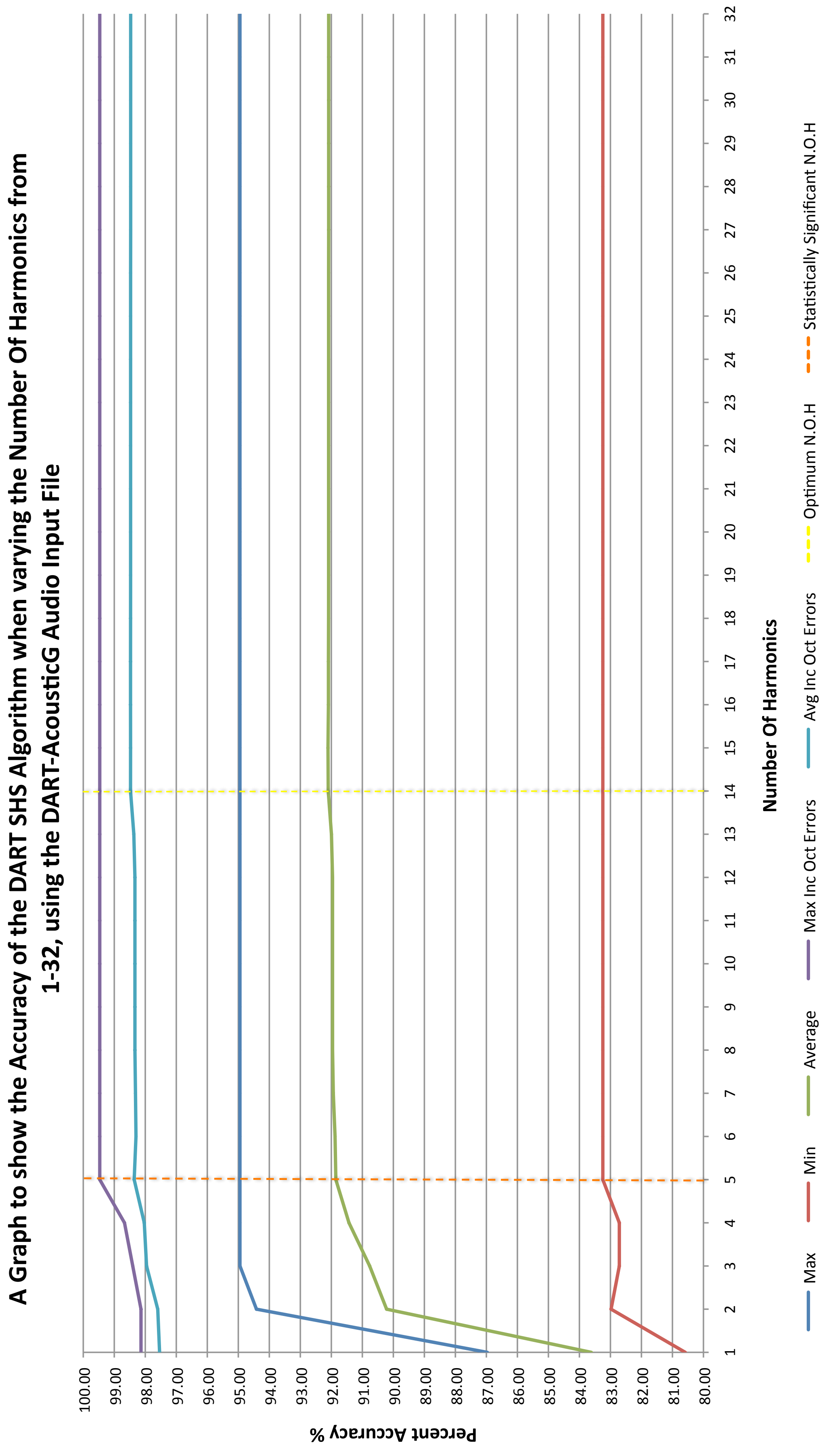


Figure 7.21: A graph to show the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics from 1-32, with minimum, maximum and average score values using the Acoustic Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NoHarmonics	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
1	86.97	80.59	83.61	98.14	97.54	98.14	16.22	11.17	13.92
2	94.41	82.98	90.22	98.14	97.60	98.14	14.63	3.72	7.38
3	94.95	82.71	90.76	98.40	97.96	98.14	14.63	3.19	7.19
4	94.95	82.71	91.43	98.67	98.03	98.14	14.63	3.19	6.60
5	*94.95*	*83.24*	*91.85*	*99.47*	*98.36*	*98.14*	*14.63*	*3.19*	*6.50*
6	94.95	83.24	91.88	99.47	98.30	98.14	14.63	3.19	6.42
7	94.95	83.24	91.94	99.47	98.32	98.14	14.63	3.19	6.38
8	94.95	83.24	91.96	99.47	98.33	98.14	14.63	3.19	6.37
9	94.95	83.24	91.96	99.47	98.33	98.14	14.63	3.19	6.37
10	94.95	83.24	91.96	99.47	98.33	98.14	14.63	3.19	6.37
11	94.95	83.24	91.96	99.47	98.33	98.14	14.63	3.19	6.37
12	94.95	83.24	91.96	99.47	98.33	98.14	14.63	3.19	6.37
13	94.95	83.24	92.00	99.47	98.37	98.14	14.63	3.19	6.37
14	**94.95**	**83.24**	**92.10**	**99.47**	**98.48**	**98.14**	**14.63**	**3.19**	**6.38**
15	94.95	83.24	92.11	99.47	98.47	98.14	14.63	3.19	6.37
16	94.95	83.24	92.09	99.47	98.48	98.14	14.63	3.19	6.38
17	94.95	83.24	92.09	99.47	98.48	98.14	14.63	3.19	6.38
18	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
19	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
20	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
21	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
22	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
23	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
24	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
25	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
26	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
27	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
28	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
29	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
31	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38
32	94.95	83.24	92.09	99.47	98.47	98.14	14.63	3.19	6.38

Table 7.10: This table details the accuracy of the DART SHS algorithm while varying the Number Of Harmonics analysed by the SHS algorithm, using the DART Acoustic Guitar audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.4.2 Number Of Harmonics Results for Oboe

Graph 7.22 shows the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics analysed by the SHS algorithm from 1-32, using the DART Oboe audio input file. Table 7.11 shows the values presented in Graph 7.22. The first 5 columns represent data presented in Graph 7.22, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of each number of harmonics.

Graph 7.22 shows that while the greatest accuracy is discovered when analysing **11** Harmonics, that looking at more than **3** harmonics does not give a significant improvement in accuracy. The results of the SHS analysis indicate a relatively high level of octave errors, with approximately 25% pitch errors (not explainable with octave mistakes).

To understand these results it is important to note that in both cases of wood and plastic oboes, the fundamental frequency is not the highest amplitude frequency present in the spectrum. The second and especially third harmonics are of the highest amplitude of any frequency in the spectrum produced by the Oboe[5]. For this reason, the Number Of Harmonics value of **3** should be able to detect most of the pitches possible, except at very low frequencies.

At 1 harmonic, we see a maximum of 34.86% accuracy, rising over 10% to 47.02% at 3 Harmonics. Jumping to 12 Harmonics only sees an increase of around 1% to 48.39% maximum accuracy. Only 2 harmonics are required in order to normalise the minimum accuracy value, with a small increase in the overall minimum accuracy from 1 harmonic (27.29%) to 2 27.52%. When allowing for octave errors, the maximum accuracy achievable was 71.33%, with an average of 67.66%. This means that even when taking into account octave errors, we have an average *inaccuracy* rate of nearly third (32.34%) across the range of all velocities.

This can be explained by the fact that the oboe produces both even and very loud odd-numbered harmonics (but in inverse proportion to their amplitudes), creating an irregular waveform that creates sounds that are not harmonic partials; giving a 'confused' perception of pitch. The result of this is a sawtooth wave, which is also the type of wave produced by most string instruments; hence a similar timbre and sound to a violin. The samples

[5]This is known as 'overblowing to the octave'.

with a high velocity (a strong ‘blow’ of the oboe) will generate more of these harmonics and could be responsible for the pitch mistakes.

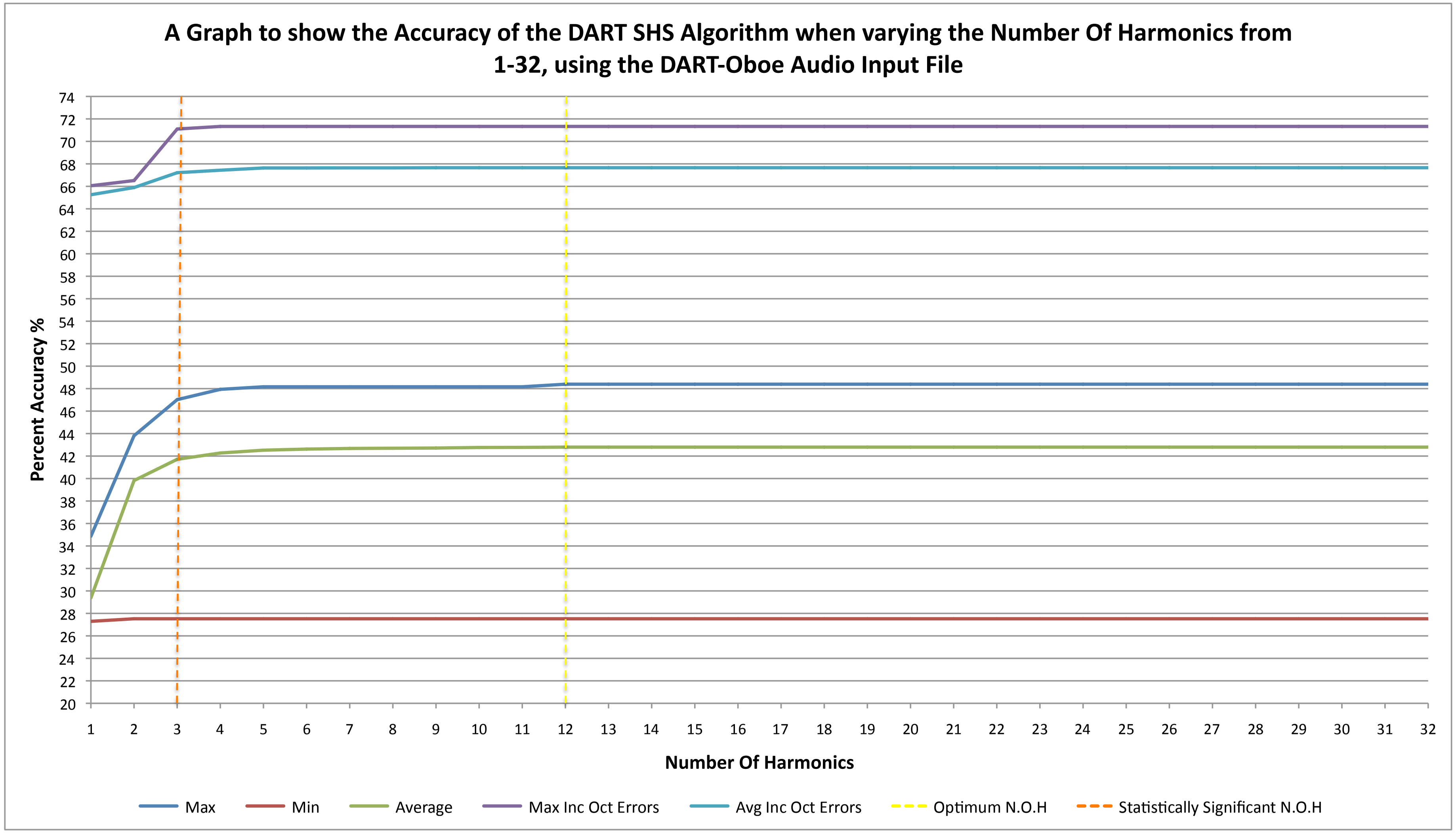


Figure 7.22: A graph to show the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics from 1-32, with minimum, maximum and average score values using DART Oboe audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	34.86	27.29	29.37	66.06	65.25	64.91	37.61	29.36	35.88
2	43.81	27.52	39.82	66.51	65.89	65.37	36.47	21.56	26.07
3	47.02	27.52	41.71	71.10	67.21	71.10	36.47	21.10	25.50
4	47.94	27.52	42.27	71.33	67.43	71.33	36.47	20.41	25.16
5	*48.17*	*27.52*	*42.52*	*71.33*	*67.63*	*69.04*	*36.47*	*20.87*	*25.11*
6	48.17	27.52	42.62	71.33	67.63	71.33	36.47	20.87	25.01
7	48.17	27.52	42.68	71.33	67.64	71.33	36.47	20.87	24.97
8	48.17	27.52	42.70	71.33	67.64	71.33	36.47	20.87	24.95
9	48.17	27.52	42.72	71.33	67.66	71.33	36.47	20.87	24.94
10	48.17	27.52	42.76	71.33	67.66	71.33	36.47	20.87	24.89
11	48.17	27.52	42.77	71.33	67.66	71.33	36.47	20.87	24.88
12	**48.39**	**27.52**	**42.79**	**71.33**	**67.66**	**69.04**	**36.47**	**20.64**	**24.86**
13	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
14	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
15	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
16	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
17	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
18	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
19	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
20	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
21	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
22	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
23	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
24	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
25	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
26	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
27	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
28	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
29	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86

NoHarmonics	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
30	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
31	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86
32	48.39	27.52	42.79	71.33	67.66	69.04	36.47	20.64	24.86

Table 7.11: This table details the accuracy of the DART SHS algorithm while varying the Number Of Harmonics analysed by the SHS algorithm, using the DART Oboe audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*

7.4.3 Number Of Harmonics Results for Violin

Graph 7.23 shows the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics analysed by the SHS algorithm from 1-32, using the DART Violin audio input file. Table (7.12) shows the values presented in Graph 7.23. The first 5 columns represent data presented in Graph 7.23, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of each number of harmonics.

Graph 7.23 shows that while the greatest accuracy is discovered when analysing **12** Harmonics, that looking at more than **5** harmonics does not give a significant improvement in accuracy. The results of the SHS analysis indicate a relatively high level of octave errors, with only 1-2% pitch errors (not explainable with octave mistakes) when analysing 12 harmonics.

The violin audio file produces a large number of octave errors, but when taking into account (or ignoring) these octave errors the SHS algorithm can produce a high accuracy rate of 97.08% and a maximum accuracy rate of 98.81% at just 5 harmonics. When not including octave errors, increasing the Number Of Harmonics from 1 to 5 produces a relatively worthwhile increase in average (66.63% to 73.29%) and maximum (from 72.32% to 80.36%) accuracy rates, however the minimum accuracy rate moves up only 3% from 1 to 5 harmonics.

These results are expected due to the complex nature of the sound produced by a violin. The acoustics of a violin body are complex, with the coupled oscillations of the strings, bridge, the top and bottom plates, the ribs, and the fingerboard. The sound from a violin comes from the resonating body, and the fundamental frequency is often not represented, with louder second and third harmonics, producing a complex, predominantly 'sawtooth' tone. Violins have very strong higher harmonics and depending upon how the string is bowed, different overtones can be emphasised, with a harder velocity producing a higher number of overtones. The violin strings themselves make very little noise: they are thin and slip easily through the air without making much of disturbance; the bridge and body of the violin transmit the vibration of the strings into the air, however when doing so, the fundamental frequency is no longer the frequency with the highest amplitude.

It seems reasonable to suggest that the lack of amplitude of the fundamental frequency and a high number of even harmonics would produce a large number of cases where the pitch is correct but the octave is incorrect.

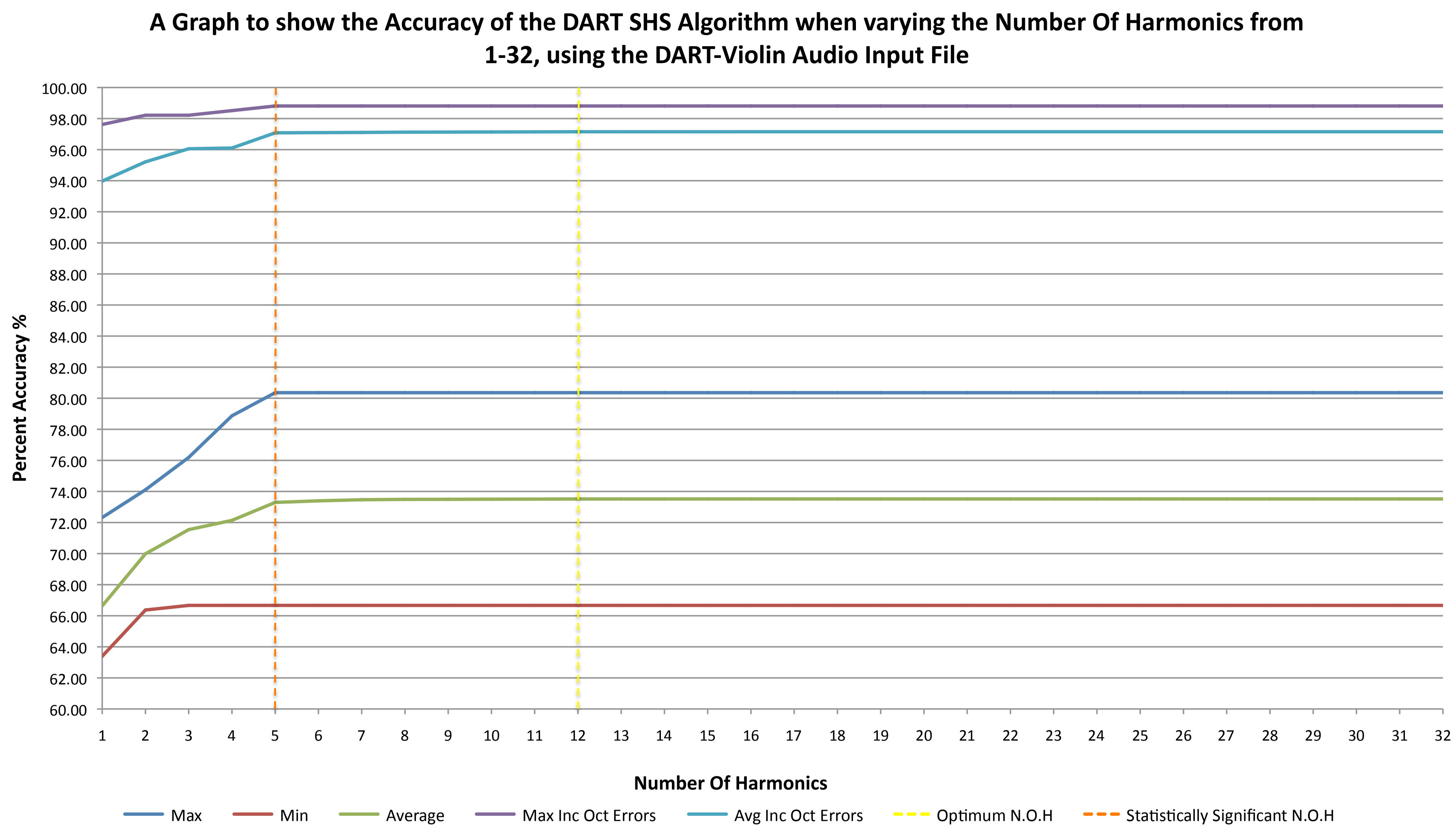


Figure 7.23: A graph to show the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics from 1-32, with minimum, maximum and average score values across all audio input files. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	72.32	63.39	66.63	97.62	93.97	97.62	29.76	25.30	27.34
2	74.11	66.37	69.99	98.21	95.21	97.02	27.98	22.92	25.23
3	76.19	66.67	71.54	98.21	96.06	98.21	27.98	22.02	24.51
4	78.87	66.67	72.14	98.51	96.11	98.51	27.98	19.64	23.97
5	*80.36*	*66.67*	*73.29*	*98.81*	*97.08*	*98.81*	*27.98*	*18.45*	*23.79*
6	80.36	66.67	73.40	98.81	97.10	98.81	27.98	18.45	23.70
7	80.36	66.67	73.47	98.81	97.11	98.81	27.98	18.45	23.64
8	80.36	66.67	73.49	98.81	97.13	98.81	27.98	18.45	23.64
9	80.36	66.67	73.49	98.81	97.13	98.81	27.98	18.45	23.64
10	80.36	66.67	73.50	98.81	97.14	98.81	27.98	18.45	23.64
11	80.36	66.67	73.51	98.81	97.14	98.81	27.98	18.45	23.64
12	**80.36**	**66.67**	**73.52**	**98.81**	**97.15**	**98.81**	**27.98**	**18.45**	**23.64**
13	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
14	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
15	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.63
16	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
17	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
18	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
19	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
20	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
21	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
22	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
23	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
24	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
25	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
26	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
27	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
28	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
29	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
31	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64
32	80.36	66.67	73.52	98.81	97.15	98.81	27.98	18.45	23.64

Table 7.12: This table details the accuracy of the DART SHS algorithm while varying the Number Of Harmonics using the DART Violin audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.4.4 Number Of Harmonics Results for Piano

Graph 7.24 shows the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics analysed by the SHS algorithm from 1-32, using the DART Piano audio input file. Table (7.13) shows the values presented in Graph 7.24, with the first 5 columns representing data presented in the graph, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of each number of harmonics.

Graph 7.24 shows that while the greatest accuracy is discovered when analysing **12** Harmonics, that looking at more than **7** harmonics does not give a significant improvement in accuracy, and even at 3 harmonics, the accuracy rate barely improves. The results of the SHS analysis indicate a relatively high level of octave errors, and with around 24% pitch errors (not explainable with octave mistakes) when analysing 12 harmonics.

The piano audio file produces a large number of octave errors, but when ignoring these octave errors the SHS algorithm can produce a slightly higher accuracy rate of 73.42% and maximum accuracy rate of 76.24% at 7 harmonics. When excluding octave errors, increasing the Number Of Harmonics from 1 to 3 produces a worthwhile increase in average (66.63% to 73.29%) and maximum (from 72.32% to 80.36%) accuracy rates, however the minimum accuracy rate does not move from 47.83% across any number of harmonics.

The piano audio input file contained by far the largest set of input data, spanning the largest range of notes, with two sampled pianos playing back their full range at 4 different velocity levels (664 distinct notes in total). A grand piano has approximately 88 keys and 230 strings. Depressing a key on the piano engages a complex mechanism called the *action*, which causes the hammer to strike the string. When a piano key is pressed gently the hammer hits the string with a small amount of force and acts as a gentle spring. The resulting sound is dominated by the fundamental frequency of the string and few higher harmonics. In contrast, when a piano key is struck with a high velocity the hammer hits the string with a greater force and acts as a harder spring, resulting in many higher frequency harmonics and a louder, brighter sound.

As 25% of the notes in the piano audio input files were at a very high velocity level this could explain the approximate 25% error rate (when including octave mistakes)

There is also a large difference in the level of *inharmonicity* across the piano keybed. Inharmonicity is the degree to which the frequencies of overtones depart from whole multiples of the fundamental frequency. A stiff string under low tension (such as those found in the bass notes) exhibits a high level of inharmonicity, while a thinner string under higher tension (such as a high pitch string in a piano) will exhibit less inharmonicity.

As explained in [121]: *"In the bass region of the piano, the string spectra contain about 50-60 harmonics and extend out to about 5,000 Hz. In the middle region the string spectra contain about 20-30 harmonics and extend out to about 7,000 Hz. In the treble region the string spectra contain less than 10 harmonics and extend out to about 10,000 Hz. The highest couple of notes on the piano may only produce a fundamental and perhaps one harmonic. This trend means that bass notes sound rich and full (since many frequencies are being produced at the same time) while treble notes sound weak and thin."*

Due to the high number of harmonics at the lower range of the piano, many pianos are tuned with the lower strings up to 30 cents flat in order to place the upper harmonics more in tune with the midrange notes. The upper harmonics of these low strings are more prominent than the fundamental, and therefore this flat tuning can sound more 'in tune'. The extreme upper end of the piano can also be tuned up to 30 cents sharp. These practices could also contribute to the lower overall accuracy levels and over 25% average pitch error rate (when including octave mistakes). It is also interesting to note with a number of harmonics higher than around 16, there is a fall off in accuracy in most of the measurements. This could be attributed to too many harmonics in the lower notes skewing the overall perception of the note being analysed.

The high level of harmonics also explain the high level of octave errors. In addition to the high level of harmonics in the *bass* region of the piano however, the instrument itself has a very wide pitch range and therefore goes down to very low frequencies, which SHS algorithm struggles with due to the extremely small differences between frequencies representing notes, especially at C0 (16.35Hz) to C#0 (17.32Hz), a difference of less than 1Hz. The SHS algorithm works with integer values to represent the frequency of the note, and cannot produce accurate results when the frequencies are extremely low and the difference between notes is small. The Grand Piano has more of these low notes than the other instruments in the DART experiments.

A final point to consider is the length of time that of each of the samples in the piano audio file. Each note only rings out for 0.5 seconds and as such the note is rich with harmonics. As any string rings out the higher harmonics will decay, leaving predominately the fundamental frequency to ring. The clearer fundamental does not have a chance to stand out, resulting in a less accurate result than would be expected for the SHS algorithm had the note been allowed to ring out for longer.

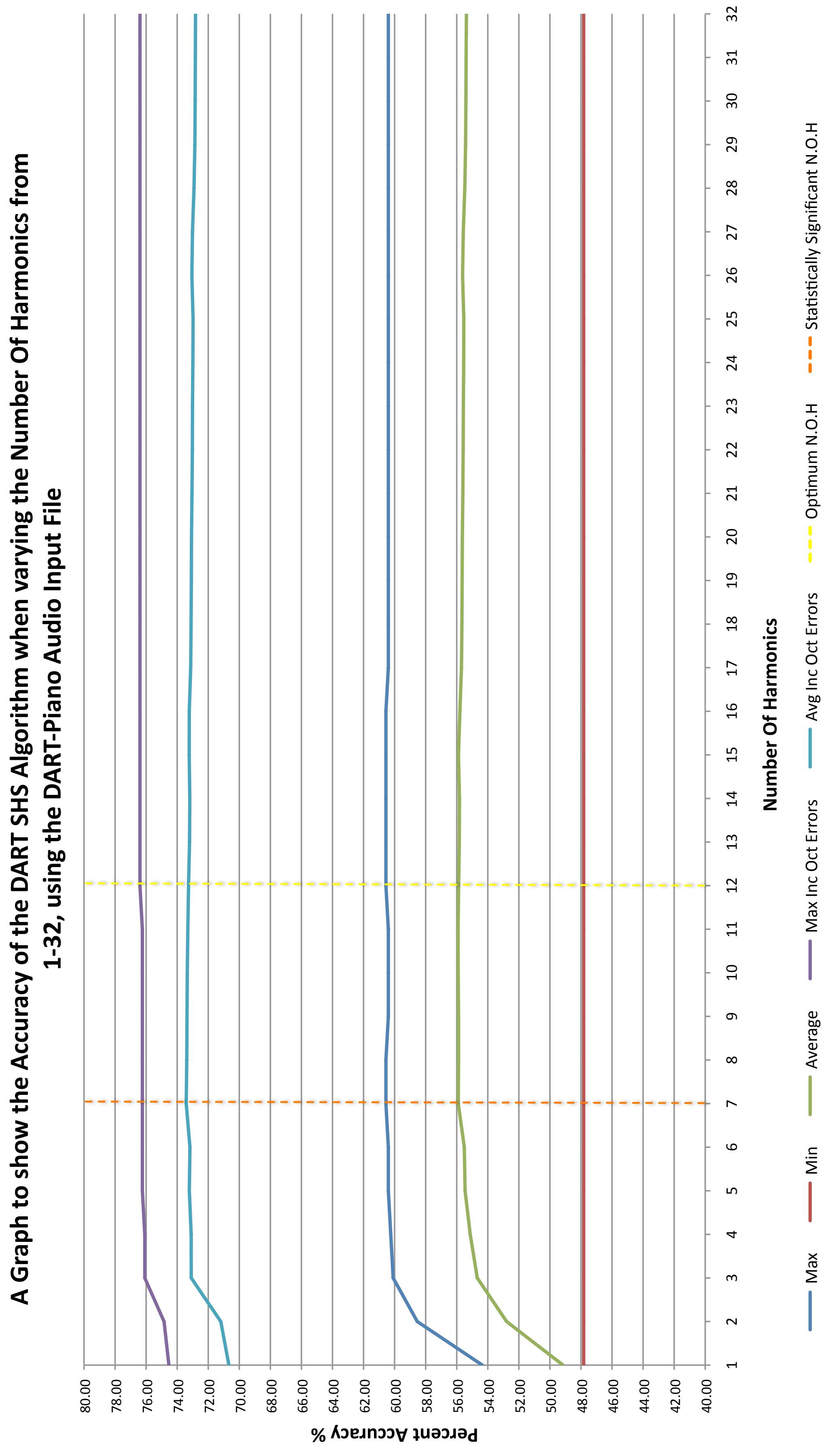


Figure 7.24: A graph to show the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics from 1-32, with minimum, maximum and average score values using the Piano input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	54.35	47.83	49.15	74.53	70.67	74.53	22.67	19.72	21.52
2	58.54	47.83	52.79	74.84	71.19	74.69	22.67	16.15	18.40
3	60.09	47.83	54.68	76.09	73.10	76.09	22.67	15.99	18.42
4	60.25	47.83	55.13	76.09	73.10	76.09	22.67	15.84	17.97
5	60.40	47.83	55.46	76.24	73.22	76.24	22.67	15.84	17.76
6	60.40	47.83	55.51	76.24	73.18	76.24	22.67	15.84	17.67
7	*60.56*	*47.83*	*55.90*	*76.24*	*73.42*	*76.24*	*22.67*	*15.68*	*17.52*
8	60.56	47.83	55.88	76.24	73.38	76.24	22.67	15.68	17.50
9	60.40	47.83	55.88	76.24	73.37	76.24	22.67	15.84	17.48
10	60.40	47.83	55.91	76.24	73.35	76.24	22.67	15.53	17.44
11	60.40	47.83	55.91	76.24	73.31	76.24	22.67	15.37	17.40
12	**60.56**	**47.83**	**55.88**	**76.40**	**73.27**	**76.40**	**22.67**	**15.22**	**17.39**
13	60.56	47.83	55.84	76.40	73.20	76.40	22.67	15.22	17.37
14	60.56	47.83	55.82	76.40	73.19	76.40	22.67	15.22	17.37
15	60.56	47.83	55.89	76.40	73.23	76.40	22.67	15.22	17.34
16	60.56	47.83	55.79	76.40	73.22	76.40	22.67	15.22	17.43
17	60.40	47.83	55.69	76.40	73.13	76.40	22.67	15.22	17.44
18	60.40	47.83	55.67	76.40	73.11	76.40	22.67	15.22	17.44
19	60.40	47.83	55.65	76.40	73.09	76.40	22.67	15.22	17.44
20	60.40	47.83	55.64	76.40	73.08	76.40	22.67	15.22	17.44
21	60.40	47.83	55.61	76.40	73.05	76.40	22.67	15.22	17.44
22	60.40	47.83	55.59	76.40	73.02	76.40	22.67	15.22	17.44
23	60.40	47.83	55.57	76.40	73.01	76.40	22.67	15.22	17.44
24	60.40	47.83	55.55	76.40	72.99	76.40	22.67	15.22	17.44
25	60.40	47.83	55.55	76.40	72.98	76.40	22.67	15.22	17.43
26	60.40	47.83	55.62	76.40	73.05	76.40	22.67	15.22	17.43
27	60.40	47.83	55.58	76.40	73.01	76.40	22.67	15.22	17.44
28	60.40	47.83	55.48	76.40	72.92	76.40	22.67	15.22	17.44
29	60.40	47.83	55.43	76.40	72.86	76.40	22.67	15.37	17.43

NoHarmonics	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
30	60.40	47.83	55.41	76.40	72.84	76.40	22.67	15.37	17.43
31	60.40	47.83	55.40	76.40	72.83	76.40	22.67	15.37	17.43
32	60.40	47.83	55.38	76.40	72.82	76.40	22.67	15.37	17.44

Table 7.13: This table details the accuracy of the DART SHS algorithm while varying the Number Of Harmonics using the DART Piano audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.4.5 Number Of Harmonics Results for Tubular Bells

Graph 7.25 shows the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics analysed by the SHS algorithm from 1-32, using the DART Tubular Bells audio input file. Table (7.14) shows the all values presented in the graph, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of each number of harmonics.

The Tubular Bells exhibited the most inharmonicity, with by far the lowest accuracy results of the six audio files analysed. Graph 7.25 shows that while the greatest accuracy is discovered when analysing **13** Harmonics, that looking at more than **4** harmonics does not give a significant improvement in accuracy. The results of the SHS analysis indicate an extremely high level of octave errors, and with an average of around 48% complete pitch errors (not explainable with octave mistakes) when analysing 13 harmonics. It is interesting to note that the even when increasing from 1 to 4 harmonics, the only improvements are seen in the maximum and average accuracy values, and the improvements are minor, at less than 2%.

The tubular bells audio file produces a large number of octave errors, but when ignoring these octave errors, the SHS algorithm produces slightly higher average and maximum accuracy rates. When including octave errors, the average accuracy rate rises to 42.19% and maximum accuracy rate of 65.91% at 4 harmonics.

Tubular bells vibrate in extremely complex ways with many different modes of vibrations, which may not necessary produce a harmonically related set of partials. If the frequencies present in a tone are not integer multiples of a single fundamental frequency, the wave does not repeat periodically and a fundamental frequency is difficult to locate. When a tone is so complex that it contains very many different frequencies with no apparent mathematical relationship, the sound is perceived as noise.

Tubular bells often have just enough sufficiently suggestive of a harmonic spectrum that we can identify a fundamental pitch, yet they contain other inharmonic partials.

It is interesting to note the minimum accuracy of 0% across all numbers of harmonics, with a near flat line or constant level of accuracy across all measurements. It is known that tubular bells exhibit very little in terms of the fundamental frequency, and as such the fourth harmonic appears to contain the most information, and analysing anymore harmonics gives no real benefit in terms of accuracy. The relatively low level of accuracy is attributed to the high level of harmonics and partials in the signals, particularly when struck with a higher velocity.

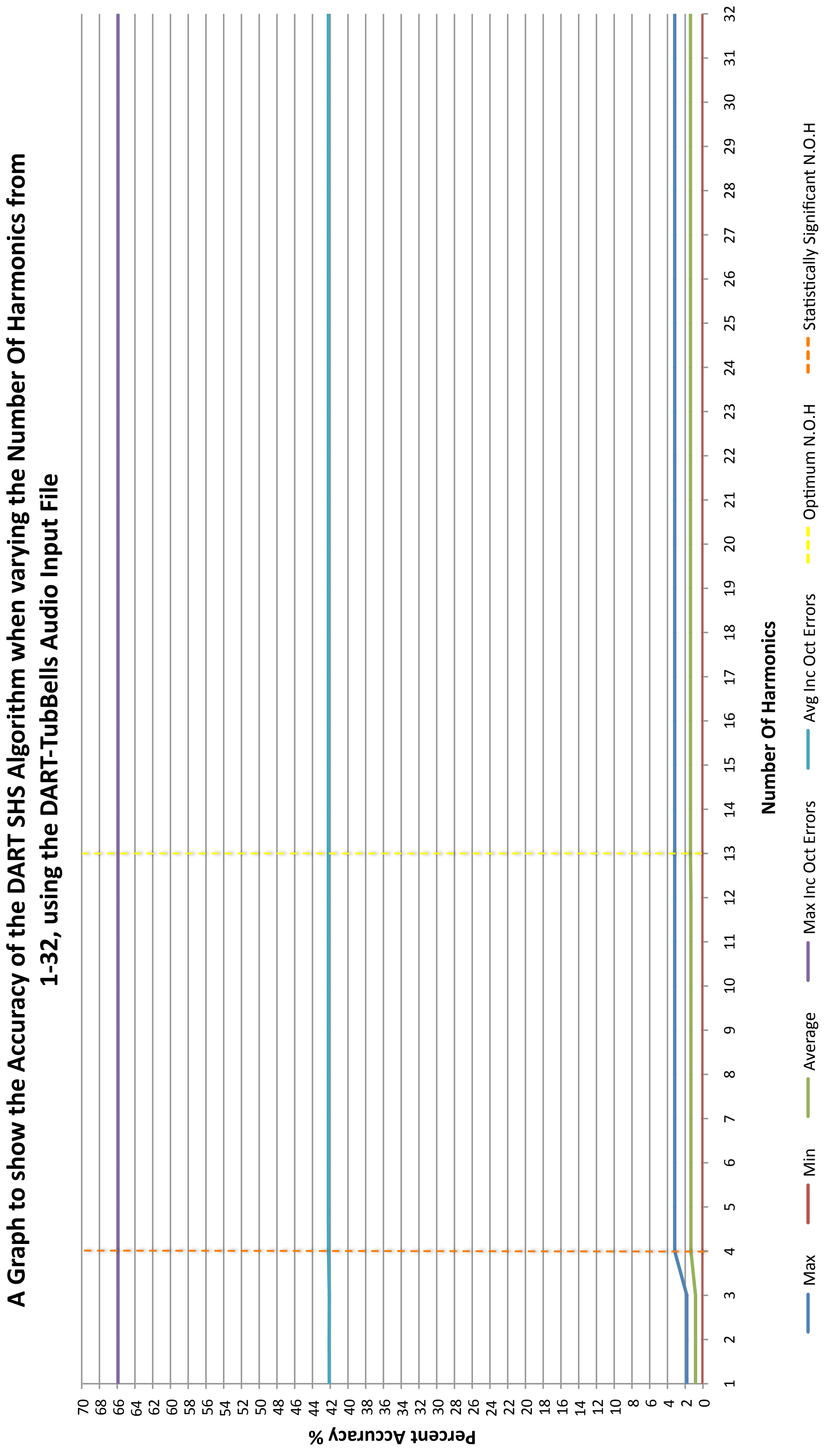


Figure 7.25: A graph to show the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics from 1-32, with minimum, maximum and average score values using the DART Tubular Bells audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	1.82	0.00	0.84	65.91	42.11	43.64	65.91	35.91	41.27
2	1.82	0.00	0.84	65.91	42.12	43.64	65.91	35.91	41.28
3	1.82	0.00	0.84	65.91	42.10	43.64	65.91	35.91	41.27
4	*3.18*	*0.00*	*1.36*	*65.91*	*42.19*	*40.91*	*65.91*	*35.91*	*40.84*
5	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
6	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
7	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
8	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
9	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
10	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
11	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
12	3.18	0.00	1.36	65.91	42.21	40.91	65.91	35.91	40.85
13	**3.18**	**0.00**	**1.41**	**65.91**	**42.21**	**41.36**	**65.91**	**35.91**	**40.80**
14	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
15	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
16	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
17	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
18	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
19	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
20	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
21	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
22	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
23	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
24	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
25	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
26	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
27	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
28	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
29	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
31	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80
32	3.18	0.00	1.41	65.91	42.21	41.36	65.91	35.91	40.80

Table 7.14: This table details the accuracy of the DART SHS algorithm while varying the Number Of Harmonics using the DART Tubular Bells audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.4.6 Number Of Harmonics Results for Distorted Guitar

Graph 7.26 shows the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics analysed by the SHS algorithm from 1-32, using the DART Distorted Guitar audio input file. Table 7.15 shows all the data values presented in Graph 7.26, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of each number of harmonics.

Graph 7.26 shows that while the greatest accuracy is discovered when analysing **15** harmonics, that looking at more than **8** harmonics gives no significant improvement in accuracy. The results of the SHS analysis indicate a relatively high level of octave errors, and with an average of around 21% pitch errors (not explainable with octave mistakes) when analysing 15 harmonics.

The distorted guitar audio file produces a large number of octave errors, but when ignoring these octave errors, the SHS algorithm can produce a much higher accuracy rate. When including octave errors, the average accuracy rate rises to 78.38% and maximum accuracy rate of 84.07% at 8 harmonics. When not including octave errors, increasing the Number Of Harmonics from 8 to 15 produces a relatively minor increase in average (45.46% to 46.04%) and maximum (from 53.85% to 55.22%) accuracy rates. The minimum accuracy rate does not increase any higher than 28.85% after 2 harmonics, rising only 0.83% from 1 harmonic.

The introduction of distortion from Logic Pro‘s guitar amplifier simulation, Guitar Amp Pro (shown in Figure 5.8) introduces a large number of odd and even order harmonics to the relatively clean electric guitar signal. The large number of overtones and harmonics introduced by the amplification simulation and distortion makes the SHS algorithm produce less accurate results, and creates a greater number of octave mistakes in comparison to the cleaner acoustic guitar samples.

Using any distortion device introduces a great number of variables, in addition to the variables taken into consideration when analysing the acoustic guitar audio input file and string vibrations in general. Different amplifiers modify the clean electric guitar signal in different ways, some producing more odd harmonics creating a more square wave shave, and some producing more even harmonics giving a more sawtooth shape. Furthermore, the amplifier can be driven harder to introduce more harmonics and partials, or can be subtle, and therefore the DART distorted guitar input file is somewhat less representative of the ‘average’ distorted guitar sound.

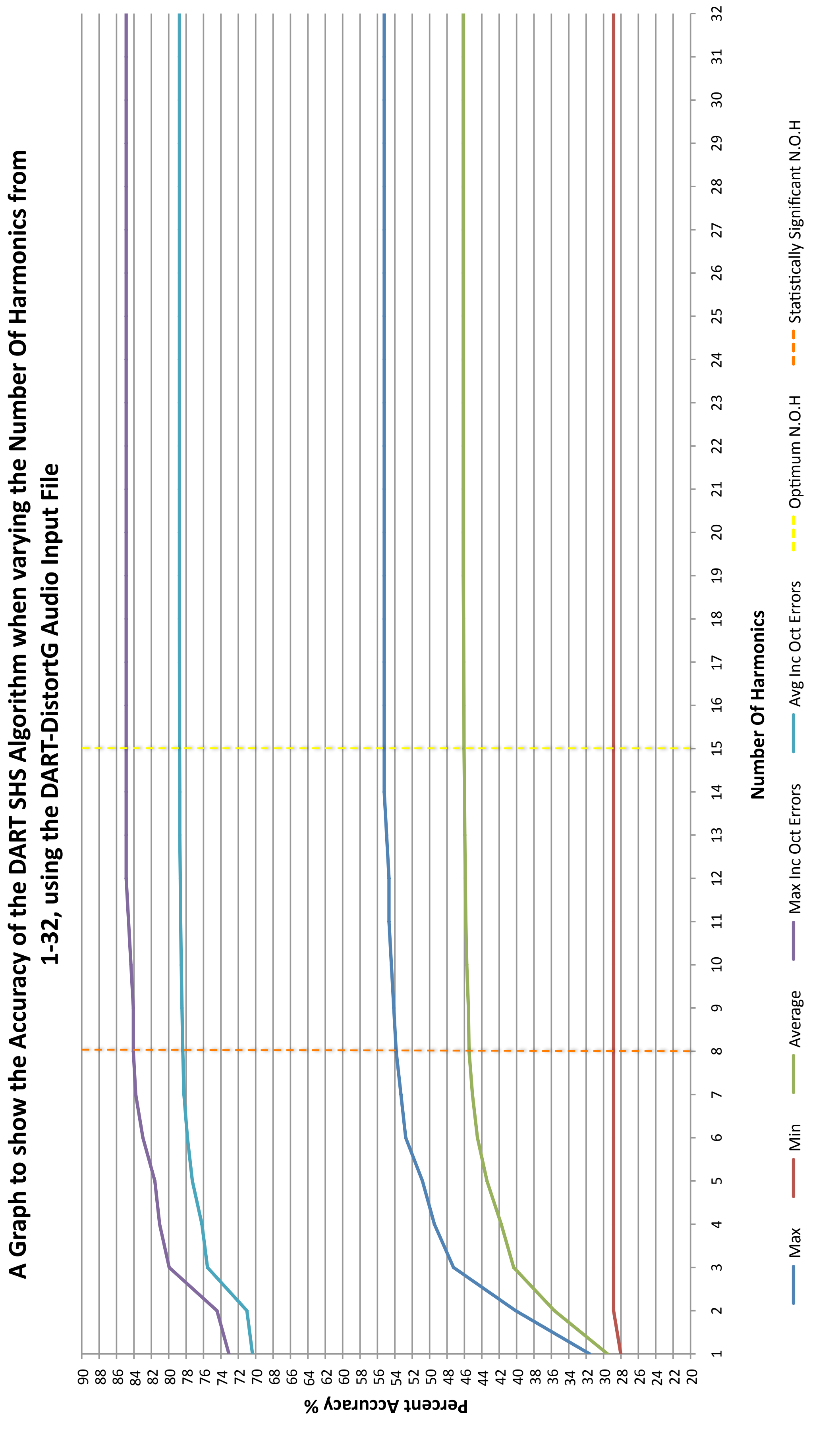


Figure 7.26: A graph to show the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics from 1-32, with minimum, maximum and average score values using the DART Distorted Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	31.59	28.02	29.54	73.08	70.37	71.70	42.03	39.29	40.83
2	40.11	28.85	35.65	74.45	71.01	73.90	41.48	32.42	35.36
3	47.25	28.85	40.34	79.95	75.54	79.67	41.48	31.87	35.21
4	49.45	28.85	41.77	81.04	76.18	80.77	41.48	30.22	34.41
5	50.82	28.85	43.40	81.59	77.28	81.59	41.48	29.12	33.88
6	52.75	28.85	44.50	82.97	77.85	82.69	41.48	28.30	33.35
7	53.30	28.85	45.08	83.79	78.25	83.52	41.48	28.02	33.17
8	*53.85*	*28.85*	*45.46*	*84.07*	*78.38*	*84.07*	*41.48*	*27.20*	*32.91*
9	54.12	28.85	45.54	84.07	78.47	83.52	41.48	27.20	32.93
10	54.40	28.85	45.73	84.34	78.56	83.52	41.48	26.65	32.83
11	54.67	28.85	45.84	84.62	78.63	83.79	41.48	26.37	32.79
12	54.67	28.85	45.90	84.89	78.67	84.89	41.48	26.37	32.77
13	54.95	28.85	45.95	84.89	78.73	84.07	41.48	26.37	32.77
14	55.22	28.85	45.99	84.89	78.73	84.07	41.48	26.37	32.74
15	**55.22**	**28.85**	**46.04**	**84.89**	**78.75**	**84.07**	**41.48**	**26.37**	**32.71**
16	55.22	28.85	46.05	84.89	78.75	84.07	41.48	26.37	32.70
17	55.22	28.85	46.08	84.89	78.76	84.07	41.48	26.37	32.69
18	55.22	28.85	46.10	84.89	78.77	84.07	41.48	26.37	32.67
19	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
20	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
21	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
22	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
23	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
24	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
25	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
26	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
27	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.37	32.63
28	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.10	32.63
29	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.10	32.63

NoHarmonics	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
30	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.10	32.63
31	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.10	32.63
32	55.22	28.85	46.14	84.89	78.77	84.07	41.48	26.10	32.63

Table 7.15: This table details the accuracy of the DART SHS algorithm while varying the Number Of Harmonics using the DART Distorted Guitar audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.4.7 Number Of Harmonics Results for All Audio Files

Graph 7.27 shows the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics analysed by the SHS algorithm from 1-32, across *all* DART audio input files. Table 7.16 shows all the values presented in Graph 7.27, however the columns *Top + O.E*, *Max Octave Error*, *Min O.E*, and *Avg O.E* give further information to highlight the accuracy of each number of harmonics.

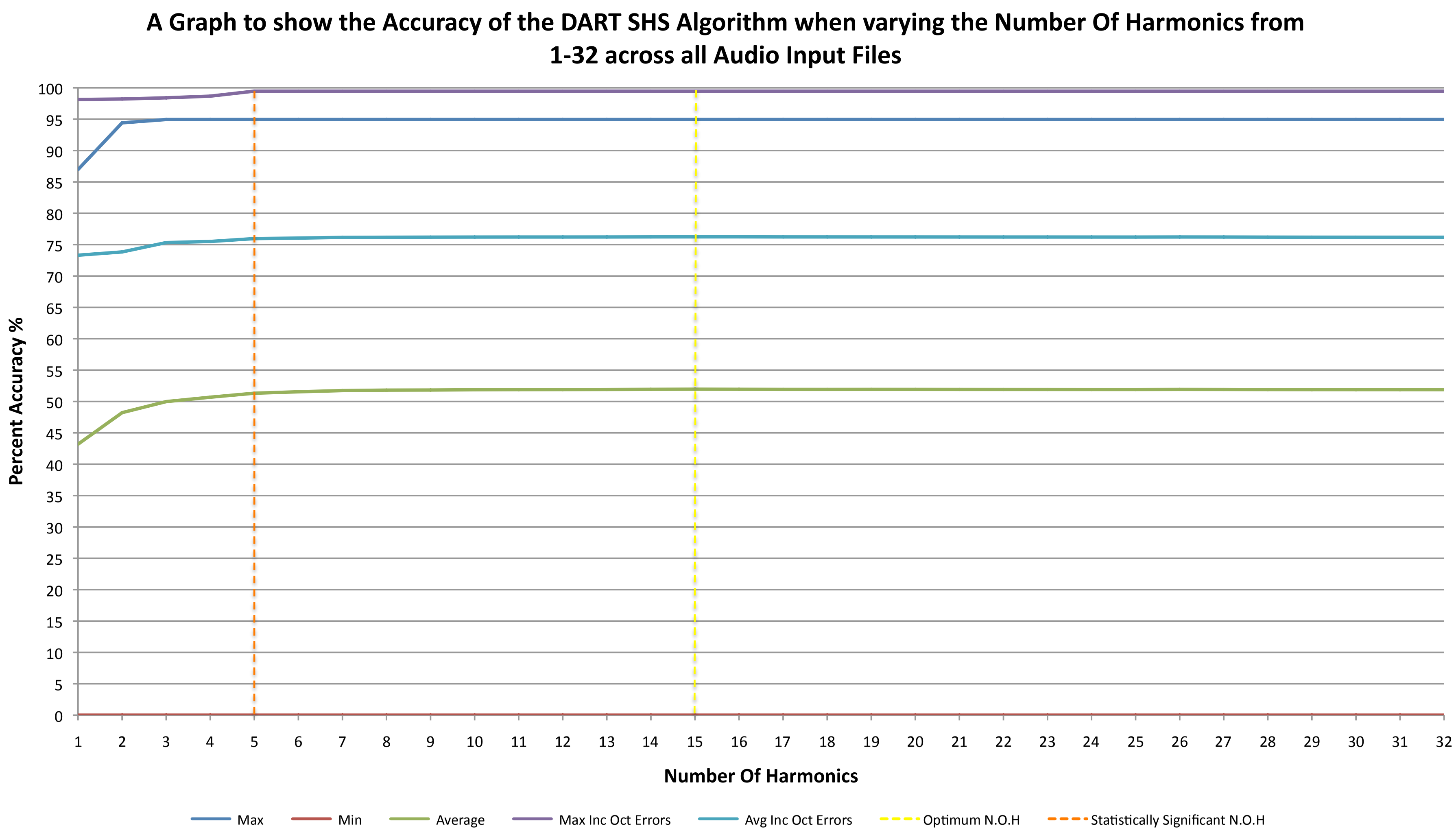


Figure 7.27: A graph to show the relative accuracy of the SHS algorithm when modifying the Number Of Harmonics from 1-32, with minimum, maximum and average score values using the DART Distorted Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	86.97	0.00	43.19	98.14	73.32	98.14	65.91	11.17	30.13
2	94.41	0.00	48.22	98.21	73.84	98.14	65.91	3.72	25.62
3	94.95	0.00	49.98	98.40	75.33	98.14	65.91	3.19	25.35
4	94.95	0.00	50.68	98.67	75.51	98.14	65.91	3.19	24.83
5	94.95	0.00	51.32	99.47	75.96	98.14	65.91	3.19	24.65
6	94.95	0.00	51.54	99.47	76.04	98.14	65.91	3.19	24.50
7	94.95	0.00	51.74	99.47	76.16	98.14	65.91	3.19	24.42
8	94.95	0.00	51.81	99.47	76.18	98.14	65.91	3.19	24.37
9	94.95	0.00	51.83	99.47	76.19	98.14	65.91	3.19	24.37
10	94.95	0.00	51.87	99.47	76.21	98.14	65.91	3.19	24.34
11	94.95	0.00	51.89	99.47	76.21	98.14	65.91	3.19	24.32
12	94.95	0.00	51.90	99.47	76.21	98.14	65.91	3.19	24.31
13	94.95	0.00	51.92	99.47	76.22	98.14	65.91	3.19	24.30
14	94.95	0.00	51.94	99.47	76.24	98.14	65.91	3.19	24.30
15	94.95	0.00	51.96	99.47	76.24	98.14	65.91	3.19	24.29
16	94.95	0.00	51.94	99.47	76.25	98.14	65.91	3.19	24.30
17	94.95	0.00	51.93	99.47	76.23	98.14	65.91	3.19	24.30
18	94.95	0.00	51.93	99.47	76.23	98.14	65.91	3.19	24.30
19	94.95	0.00	51.93	99.47	76.22	98.14	65.91	3.19	24.29
20	94.95	0.00	51.93	99.47	76.22	98.14	65.91	3.19	24.29
21	94.95	0.00	51.92	99.47	76.22	98.14	65.91	3.19	24.29
22	94.95	0.00	51.92	99.47	76.21	98.14	65.91	3.19	24.29
23	94.95	0.00	51.92	99.47	76.21	98.14	65.91	3.19	24.29
24	94.95	0.00	51.92	99.47	76.21	98.14	65.91	3.19	24.29
25	94.95	0.00	51.91	99.47	76.21	98.14	65.91	3.19	24.29
26	94.95	0.00	51.93	99.47	76.22	98.14	65.91	3.19	24.29
27	94.95	0.00	51.92	99.47	76.21	98.14	65.91	3.19	24.29
28	94.95	0.00	51.90	99.47	76.20	98.14	65.91	3.19	24.29
29	94.95	0.00	51.90	99.47	76.19	98.14	65.91	3.19	24.29

NoHarmonics	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	94.95	0.00	51.89	99.47	76.18	98.14	65.91	3.19	24.29
31	94.95	0.00	51.89	99.47	76.18	98.14	65.91	3.19	24.29
32	94.95	0.00	51.89	99.47	76.18	98.14	65.91	3.19	24.29

Table 7.16: This table details the accuracy of the DART SHS algorithm while varying the Number Of Harmonics across all DART audio input files. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.4.8 Number Of Harmonics Accuracy Summary

A summary of the Number Of Harmonics results presented so far is presented in Table 7.17. This table shows, for each audio input file, the optimal number of Harmonics (the minimum number that gives the maximum accuracy whereby increasing the Number of Harmonics shows no further improvement), as well as the 'Statistically Significant' figure whereby only minimal improvement is shown when increasing the Number of Harmonics. In Table 7.17, these two numbers are called the 'Minimum' and 'Maximum', respectively.

The Number Of Harmonics that consistently produced the most accurate results across all the variables was *15*.

The results appear to shows that the results and the optimal Number Of Harmonics for each audio input file were roughly similar. While the accuracy of the SHS algorithm differs depending on the type of audio input data, Graph 7.27 indicates that around 5 harmonics are required to achieve a high level of accuracy for the given audio type, after which only a very small increase in accuracy is found. Similarly, 15 harmonics appear to be the maximum number in order to achieve both the maximum accuracy possible (some audio files such as the tubular bells even exhibited a reduction in the accuracy when too many harmonics are analysed) and not waste resources by examining too many harmonics for no benefit in accuracy.

Audio Input File	***Minimum NoHarmonics***	***Maximum NoHarmonics***
Acoustic Guitar	5	**14**
Oboe	5	**13**
Violin	5	**12**
Piano	7	**12**
Tubular Bells	4	**13**
Distorted Guitar	8	**15**
All Audio Files	5	**15**

Table 7.17: This table shows a summary of the Number Of Harmonics that give the most accurate results for each Audio Input File.

7.5 Number Of Top Frequency Points Accuracy Results

This section displays the accuracy levels of the SHS algorithm when varying the NTFP value from 1-50. Each subsection looks at the results *per audio file*, with the result across all audio files displayed in the final sub-section 7.5.7. Each results set consists of a graph displaying the data, as well as a table containing the exact values in the graph, and extra information based on the Minimum, Maximum and Average Octave Error values. The Top Value + Octave Error value is also given.

When the SHS algorithm is searching for the fundamental frequency or pitch of the audio, it considers the top $\boldsymbol{n}$ frequency amplitude values present in the spectrum. The aim of these tests was to find the optimum value of $\boldsymbol{n}$, resulting in the highest accuracy results. The results for each instrument are presented in each sub-section, with the results discussed in the summary at the end of the section.

7.5.1 Number Of Frequency Points Results for Acoustic Guitar

Graph 7.28 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-50, using the DART Acoustic Guitar audio input file. The yellow line indicates the optimum NTFP - i.e. the minimum number of points that give the highest (or equal to the highest) accuracy for all accuracy measurements, reducing unnecessary processing. The orange dotted line indicates the minimum number of points, above which only a minor increase in accuracy is gained. Table 7.18 shows the values presented in Graph 7.28.

As when varying the Number Of Harmonics previously, the overall accuracy of the acoustic guitar was relatively high, especially when ignoring octave errors. Graph 7.28 shows that while the greatest accuracy is found when analysing **33** frequency points, that looking at more than **13** frequency points gives only a minor improvement in maximum or average accuracies (less than 1%). There is an average of only 1.5% pitch errors (not explainable with octave mistakes) when analysing 33 harmonics and including octave errors, with a maximum accuracy of 99.47% - extremely high.

When including octave errors, the average accuracy rate is 98.05% and the maximum accuracy rate is 98.40% at 13 harmonics.

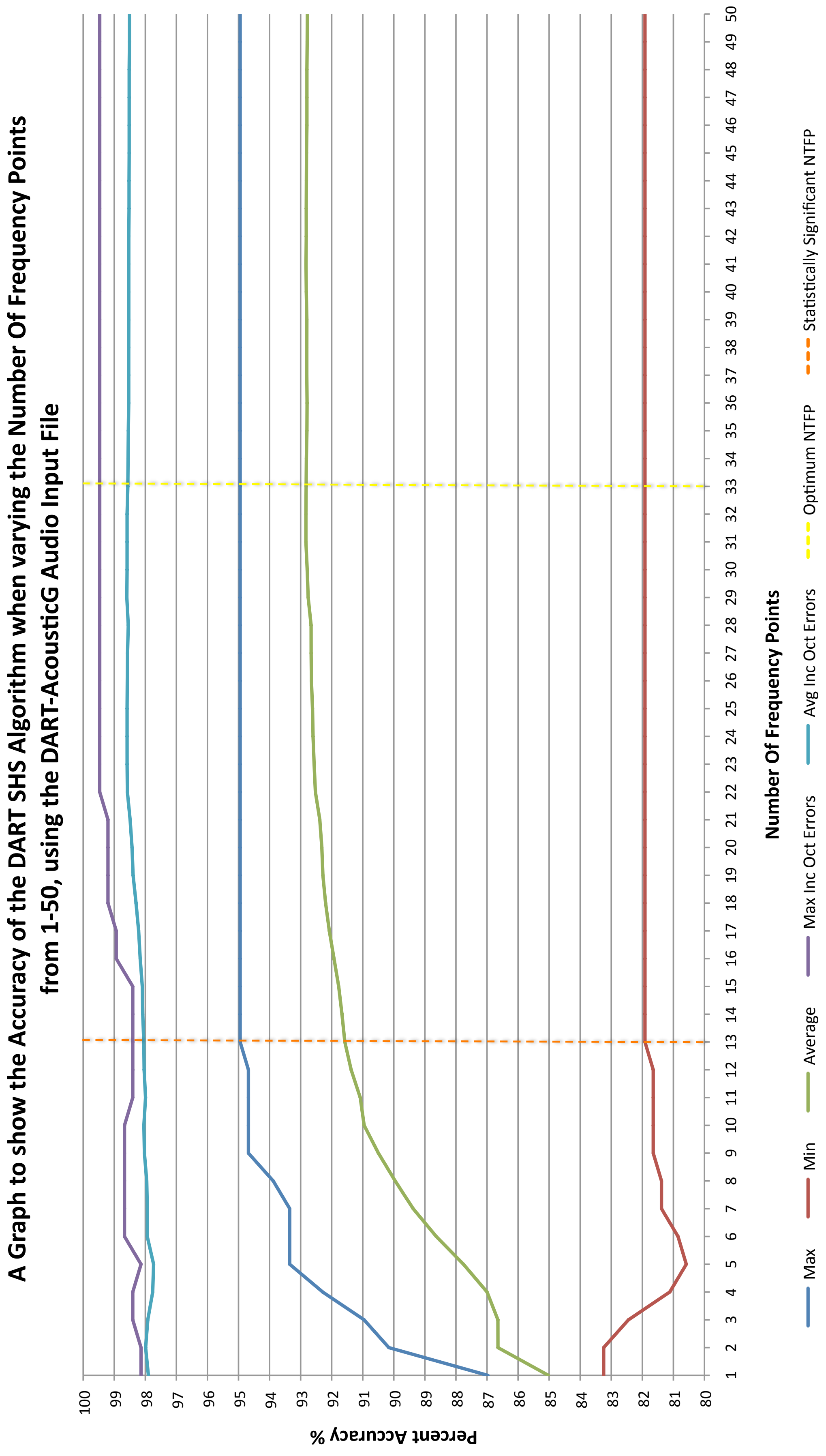


Figure 7.28: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using the DART Acoustic Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	86.97	83.24	85.02	98.14	97.90	98.14	14.63	11.17	12.88
2	90.16	83.24	86.65	98.14	97.99	98.14	14.63	7.98	11.34
3	90.96	82.45	86.65	98.40	97.92	98.14	14.89	7.18	11.27
4	92.29	81.12	87.00	98.40	97.76	98.14	15.69	5.85	10.76
5	93.35	80.59	87.76	98.14	97.73	98.14	16.22	4.79	9.98
6	93.35	80.85	88.63	98.67	97.94	98.14	15.69	4.79	9.30
7	93.35	81.38	89.38	98.67	97.94	98.14	15.43	4.79	8.56
8	93.88	81.38	89.96	98.67	97.95	98.14	15.43	4.26	8.00
9	94.68	81.65	90.50	98.67	98.03	98.14	15.43	3.46	7.53
10	94.68	81.65	90.96	98.67	98.05	98.14	15.43	3.46	7.09
11	94.68	81.65	91.08	98.40	97.99	98.14	15.43	3.46	6.91
12	94.68	81.65	91.37	98.40	98.04	98.14	15.43	3.46	6.66
13	*94.95*	*81.91*	*91.58*	*98.40*	*98.05*	*98.14*	*15.43*	*3.19*	*6.47*
14	94.95	81.91	91.67	98.40	98.08	98.14	15.43	3.19	6.41
15	94.95	81.91	91.77	98.40	98.10	98.14	15.43	3.19	6.32
16	94.95	81.91	91.93	98.94	98.16	98.14	15.43	3.19	6.24
17	94.95	81.91	92.08	98.94	98.21	98.14	15.43	3.19	6.14
18	94.95	81.91	92.20	99.20	98.30	98.14	15.43	3.19	6.10
19	94.95	81.91	92.28	99.20	98.39	98.14	15.43	3.19	6.11
20	94.95	81.91	92.32	99.20	98.43	98.14	15.43	3.19	6.11
21	94.95	81.91	92.38	99.20	98.49	98.14	15.43	3.19	6.11
22	94.95	81.91	92.52	99.47	98.58	98.14	15.43	3.19	6.05
23	94.95	81.91	92.57	99.47	98.59	98.14	15.43	3.19	6.02
24	94.95	81.91	92.60	99.47	98.59	98.14	15.43	3.19	5.99
25	94.95	81.91	92.62	99.47	98.59	98.14	15.43	3.19	5.97
26	94.95	81.91	92.65	99.47	98.58	98.14	15.43	3.19	5.93
27	94.95	81.91	92.66	99.47	98.57	98.14	15.43	3.19	5.91
28	94.95	81.91	92.66	99.47	98.55	98.14	15.43	3.19	5.89
29	94.95	81.91	92.76	99.47	98.60	98.14	15.43	3.19	5.84

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	94.95	81.91	92.79	99.47	98.59	98.14	15.43	3.19	5.80
31	94.95	81.91	92.83	99.47	98.59	98.14	15.43	3.19	5.76
32	94.95	81.91	92.83	99.47	98.59	98.14	15.43	3.19	5.76
33	**94.95**	**81.91**	**92.82**	**99.47**	**98.56**	**98.14**	**15.43**	**3.19**	**5.74**
34	94.95	81.91	92.81	99.47	98.56	98.14	15.43	3.19	5.74
35	94.95	81.91	92.80	99.47	98.55	98.14	15.43	3.19	5.75
36	94.95	81.91	92.79	99.47	98.53	98.14	15.43	3.19	5.74
37	94.95	81.91	92.80	99.47	98.53	98.14	15.43	3.19	5.73
38	94.95	81.91	92.80	99.47	98.53	98.14	15.43	3.19	5.73
39	94.95	81.91	92.80	99.47	98.53	98.14	15.43	3.19	5.73
40	94.95	81.91	92.82	99.47	98.53	98.14	15.43	3.19	5.71
41	94.95	81.91	92.83	99.47	98.53	98.14	15.43	3.19	5.71
42	94.95	81.91	92.82	99.47	98.53	98.14	15.43	3.19	5.71
43	94.95	81.91	92.82	99.47	98.52	98.14	15.43	3.19	5.70
44	94.95	81.91	92.81	99.47	98.52	98.14	15.43	3.19	5.71
45	94.95	81.91	92.81	99.47	98.52	98.14	15.43	3.19	5.71
46	94.95	81.91	92.80	99.47	98.52	98.14	15.43	3.19	5.72
47	94.95	81.91	92.80	99.47	98.52	98.14	15.43	3.19	5.72
48	94.95	81.91	92.80	99.47	98.52	98.14	15.43	3.19	5.72
49	94.95	81.91	92.79	99.47	98.51	98.14	15.43	3.19	5.72
50	94.95	81.91	92.78	99.47	98.51	98.14	15.43	3.19	5.73

Table 7.18: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Acoustic Guitar audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.5.2 Number Of Frequency Points Results for Oboe

Graph 7.29 shows the relative accuracy of the SHS algorithm when modifying the Number of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-50, using the DART Oboe audio input file. Table 7.19 shows the values presented in Graph 7.29.

Graph 7.29 shows that while the greatest accuracy is discovered when analysing the (maximum) **50** frequency points, looking at more than **24** frequency points gives only a minor improvement in maximum or average accuracies (less than 1%).

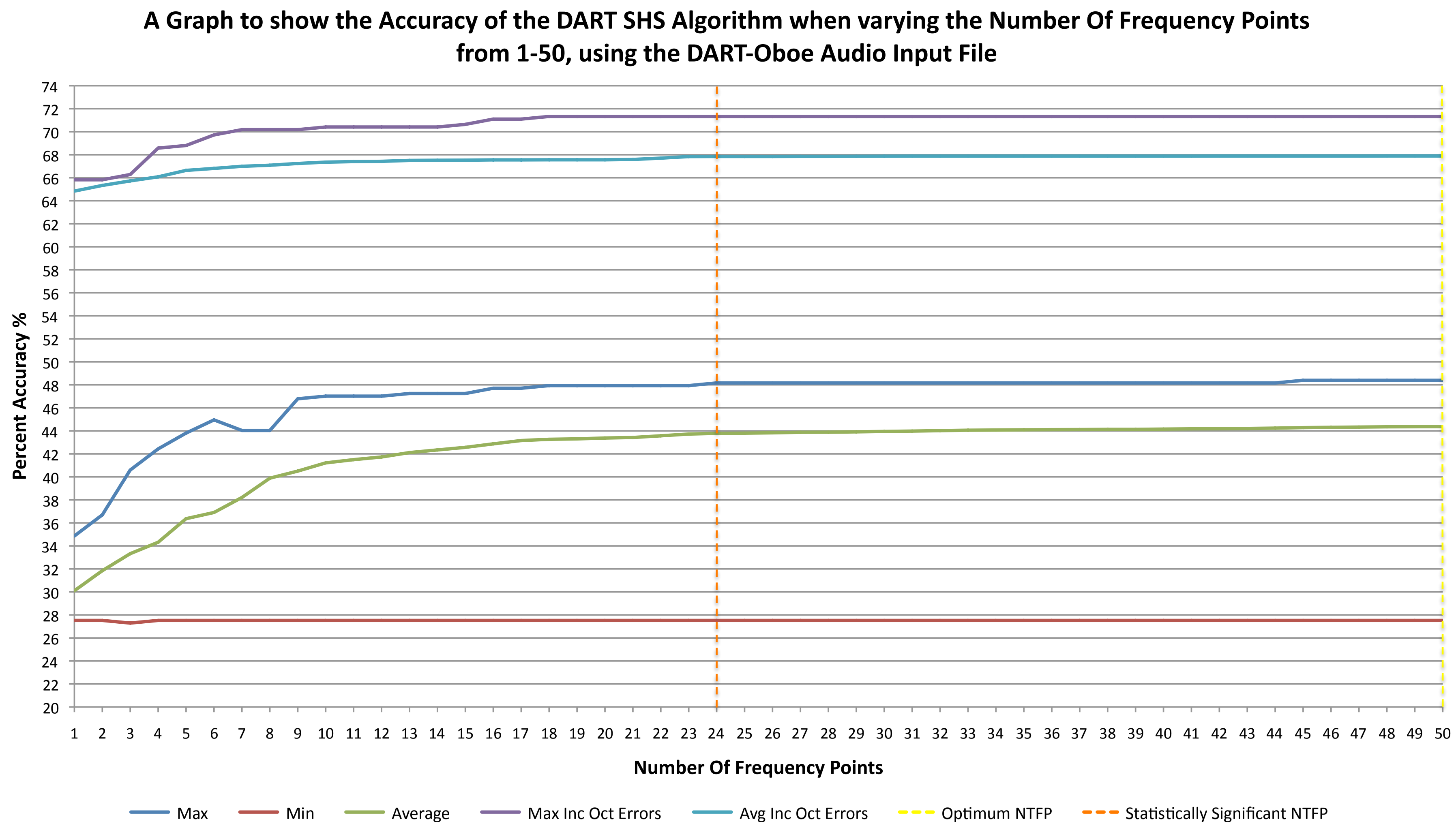


Figure 7.29: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using the DART Acoustic Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
1	34.86	27.52	30.10	65.83	64.84	64.91	36.47	29.82	34.74
2	36.70	27.52	31.85	65.83	65.33	65.14	36.47	28.44	33.49
3	40.60	27.29	33.33	66.28	65.73	66.06	36.93	25.46	32.40
4	42.43	27.52	34.32	68.58	66.08	68.58	37.61	25.00	31.76
5	43.81	27.52	36.37	68.81	66.64	68.81	36.70	23.85	30.27
6	44.95	27.52	36.91	69.72	66.81	69.72	37.39	23.85	29.91
7	44.04	27.52	38.21	70.18	67.00	70.18	37.39	25.00	28.79
8	44.04	27.52	39.89	70.18	67.09	70.18	37.39	23.39	27.20
9	46.79	27.52	40.50	70.18	67.24	70.18	37.39	22.25	26.74
10	47.02	27.52	41.22	70.41	67.36	70.41	37.39	22.25	26.14
11	47.02	27.52	41.50	70.41	67.41	70.41	37.39	22.25	25.91
12	47.02	27.52	41.73	70.41	67.43	70.41	37.39	22.25	25.70
13	47.25	27.52	42.11	70.41	67.51	70.41	37.39	21.79	25.40
14	47.25	27.52	42.34	70.41	67.52	70.41	37.39	21.56	25.18
15	47.25	27.52	42.57	70.64	67.53	70.64	37.39	21.56	24.97
16	47.71	27.52	42.87	71.10	67.56	71.10	37.39	21.56	24.68
17	47.71	27.52	43.16	71.10	67.56	71.10	37.39	21.56	24.40
18	47.94	27.52	43.27	71.33	67.56	71.33	37.39	21.33	24.30
19	47.94	27.52	43.30	71.33	67.56	71.33	37.39	21.33	24.26
20	47.94	27.52	43.38	71.33	67.56	71.33	37.39	21.33	24.18
21	47.94	27.52	43.42	71.33	67.59	71.33	37.39	21.33	24.17
22	47.94	27.52	43.57	71.33	67.71	71.33	37.39	21.33	24.14
23	47.94	27.52	43.72	71.33	67.84	71.33	37.39	21.56	24.12
24	*48.17*	*27.52*	*43.78*	*71.33*	*67.85*	*71.33*	*37.39*	*21.56*	*24.07*
25	48.17	27.52	43.80	71.33	67.85	71.33	37.39	21.33	24.05
26	48.17	27.52	43.83	71.33	67.85	71.33	37.39	21.33	24.02
27	48.17	27.52	43.88	71.33	67.86	71.33	37.39	21.33	23.98
28	48.17	27.52	43.88	71.33	67.86	71.33	37.39	21.33	23.98
29	48.17	27.52	43.91	71.33	67.87	71.33	37.39	21.10	23.95

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
30	48.17	27.52	43.95	71.33	67.87	71.33	37.39	21.10	23.92
31	48.17	27.52	43.98	71.33	67.88	71.33	37.39	21.10	23.90
32	48.17	27.52	44.02	71.33	67.88	71.33	37.39	21.10	23.86
33	48.17	27.52	44.06	71.33	67.88	71.33	37.39	21.10	23.82
34	48.17	27.52	44.07	71.33	67.88	71.33	37.39	20.64	23.81
35	48.17	27.52	44.09	71.33	67.88	71.33	37.39	20.64	23.79
36	48.17	27.52	44.11	71.33	67.88	71.33	37.39	20.64	23.77
37	48.17	27.52	44.12	71.33	67.88	71.33	37.39	20.64	23.77
38	48.17	27.52	44.13	71.33	67.88	71.33	37.39	20.64	23.75
39	48.17	27.52	44.13	71.33	67.88	71.33	37.39	20.64	23.75
40	48.17	27.52	44.15	71.33	67.88	71.33	37.39	20.64	23.73
41	48.17	27.52	44.18	71.33	67.88	71.33	37.39	20.64	23.70
42	48.17	27.52	44.19	71.33	67.89	71.33	37.39	20.64	23.70
43	48.17	27.52	44.21	71.33	67.89	71.33	37.39	20.64	23.68
44	48.17	27.52	44.24	71.33	67.89	71.33	37.39	20.64	23.65
45	48.39	27.52	44.28	71.33	67.89	69.04	37.39	20.41	23.61
46	48.39	27.52	44.31	71.33	67.89	69.04	37.39	20.41	23.58
47	48.39	27.52	44.33	71.33	67.89	69.04	37.39	20.41	23.56
48	48.39	27.52	44.35	71.33	67.90	69.04	37.39	20.41	23.54
49	48.39	27.52	44.36	71.33	67.90	69.04	37.39	20.41	23.54
50	**48.39**	**27.52**	**44.37**	**71.33**	**67.90**	**69.04**	**37.39**	**20.41**	**23.53**

Table 7.19: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Oboe audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.5.3 Number Of Frequency Points Results for Violin

Graph 7.30 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-50, using the DART Violin audio input file. Table 7.20 shows the values presented in Graph 7.30.

Graph 7.30 shows that while the greatest accuracy is discovered when analysing the (maximum) **50** frequency points, looking at more than **23** frequency points gives only a minor improvement in maximum and average accuracies (under 2%).

When allowing for octave errors, the maximum and average accuracies were very high even at a NTFP value of 3.

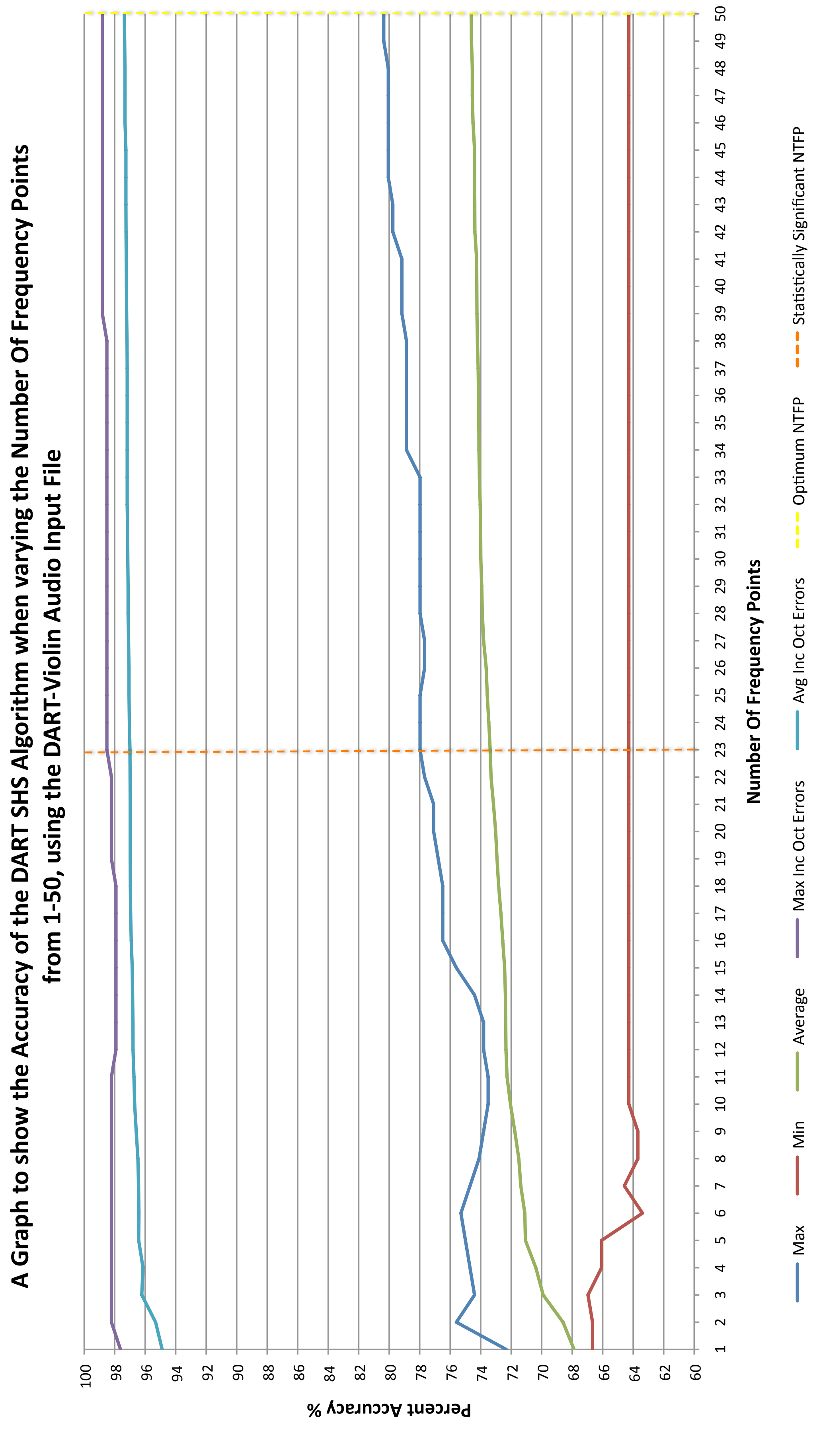


Figure 7.30: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using the DART Violin audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
1	72.32	66.67	67.88	97.62	94.89	97.62	27.98	25.30	27.01
2	75.60	66.67	68.60	98.21	95.32	98.21	27.98	22.62	26.72
3	74.40	66.96	69.90	98.21	96.22	98.21	28.27	23.81	26.32
4	74.70	66.07	70.39	98.21	96.14	98.21	27.98	23.51	25.76
5	75.00	66.07	71.07	98.21	96.43	98.21	28.57	23.21	25.36
6	75.30	63.39	71.11	98.21	96.41	98.21	29.76	22.92	25.30
7	74.70	64.58	71.37	98.21	96.44	98.21	28.57	23.51	25.07
8	74.11	63.69	71.50	98.21	96.48	98.21	28.57	24.11	24.98
9	73.81	63.69	71.76	98.21	96.59	97.92	28.57	23.81	24.83
10	73.51	64.29	72.05	98.21	96.69	97.92	28.27	23.51	24.64
11	73.51	64.29	72.27	98.21	96.74	97.92	28.57	23.51	24.47
12	73.81	64.29	72.34	97.92	96.80	97.92	28.87	23.51	24.46
13	73.81	64.29	72.36	97.92	96.80	97.92	28.57	23.51	24.44
14	74.40	64.29	72.38	97.92	96.83	97.92	28.57	23.21	24.45
15	75.60	64.29	72.43	97.92	96.85	97.92	28.57	22.32	24.42
16	76.49	64.29	72.55	97.92	96.92	97.92	28.57	21.43	24.36
17	76.49	64.29	72.67	97.92	96.96	97.92	28.57	21.43	24.28
18	76.49	64.29	72.82	97.92	96.97	97.92	28.57	21.43	24.15
19	76.79	64.29	72.93	98.21	96.98	98.21	28.57	21.43	24.05
20	77.08	64.29	73.02	98.21	96.98	98.21	28.57	21.13	23.96
21	77.08	64.29	73.16	98.21	96.98	98.21	28.57	21.13	23.82
22	77.68	64.29	73.32	98.21	96.99	98.21	28.57	20.54	23.67
23	*77.98*	*64.29*	*73.38*	*98.51*	*97.00*	*98.51*	*28.57*	*20.54*	*23.62*
24	77.98	64.29	73.47	98.51	97.04	98.51	28.57	20.54	23.57
25	77.98	64.29	73.57	98.51	97.06	98.51	28.57	20.54	23.49
26	77.68	64.29	73.64	98.51	97.06	98.51	28.57	20.83	23.42
27	77.68	64.29	73.81	98.51	97.09	98.51	28.57	20.83	23.28
28	77.98	64.29	73.89	98.51	97.12	98.51	28.57	20.54	23.23
29	77.98	64.29	73.93	98.51	97.12	98.51	28.57	20.54	23.19

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	77.98	64.29	73.99	98.51	97.14	98.51	28.57	20.54	23.15
31	77.98	64.29	74.00	98.51	97.15	98.51	28.57	20.54	23.15
32	77.98	64.29	74.03	98.51	97.18	98.51	28.57	20.54	23.15
33	77.98	64.29	74.08	98.51	97.18	98.51	28.57	20.54	23.10
34	78.87	64.29	74.11	98.51	97.18	98.51	28.57	19.64	23.07
35	78.87	64.29	74.13	98.51	97.18	98.51	28.57	19.64	23.06
36	78.87	64.29	74.14	98.51	97.18	98.51	28.57	19.64	23.04
37	78.87	64.29	74.17	98.51	97.18	98.51	28.57	19.64	23.02
38	78.87	64.29	74.22	98.51	97.19	98.51	28.57	19.64	22.98
39	79.17	64.29	74.24	98.81	97.22	98.81	28.57	19.64	22.97
40	79.17	64.29	74.26	98.81	97.23	98.81	28.57	19.64	22.97
41	79.17	64.29	74.27	98.81	97.24	98.81	28.57	19.64	22.97
42	79.76	64.29	74.38	98.81	97.26	98.81	28.57	19.05	22.88
43	79.76	64.29	74.39	98.81	97.26	98.81	28.57	19.05	22.88
44	80.06	64.29	74.40	98.81	97.26	98.81	28.57	18.75	22.86
45	80.06	64.29	74.40	98.81	97.26	98.81	28.57	18.75	22.86
46	80.06	64.29	74.50	98.81	97.32	98.81	28.57	18.75	22.82
47	80.06	64.29	74.55	98.81	97.32	98.81	28.57	18.75	22.77
48	80.06	64.29	74.55	98.81	97.32	98.81	28.57	18.75	22.77
49	80.36	64.29	74.61	98.81	97.35	98.81	28.57	18.45	22.74
50	**80.36**	**64.29**	**74.63**	**98.81**	**97.37**	**98.81**	**28.57**	**18.45**	**22.74**

Table 7.20: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Violin audio file. The top values in each column are highlighted bold. ‘O.E’ stands for *Octave Error*.

7.5.4 Number Of Frequency Points Results for Piano

Graph 7.31 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-50, using the DART Violin audio input file. Table 7.21 shows the values presented in Graph 7.31.

Graph 7.31 shows that while the greatest accuracy is discovered when analysing **37** frequency points, looking at more than **21** frequency points gives only a minor improvement in maximum and average accuracies (under 1%). Increasing the NTFP over 38 points actually results in a slight decrease in accuracy.

When allowing for octave errors, the maximum and average accuracies were very high even at low NTFP values of around 5-6.

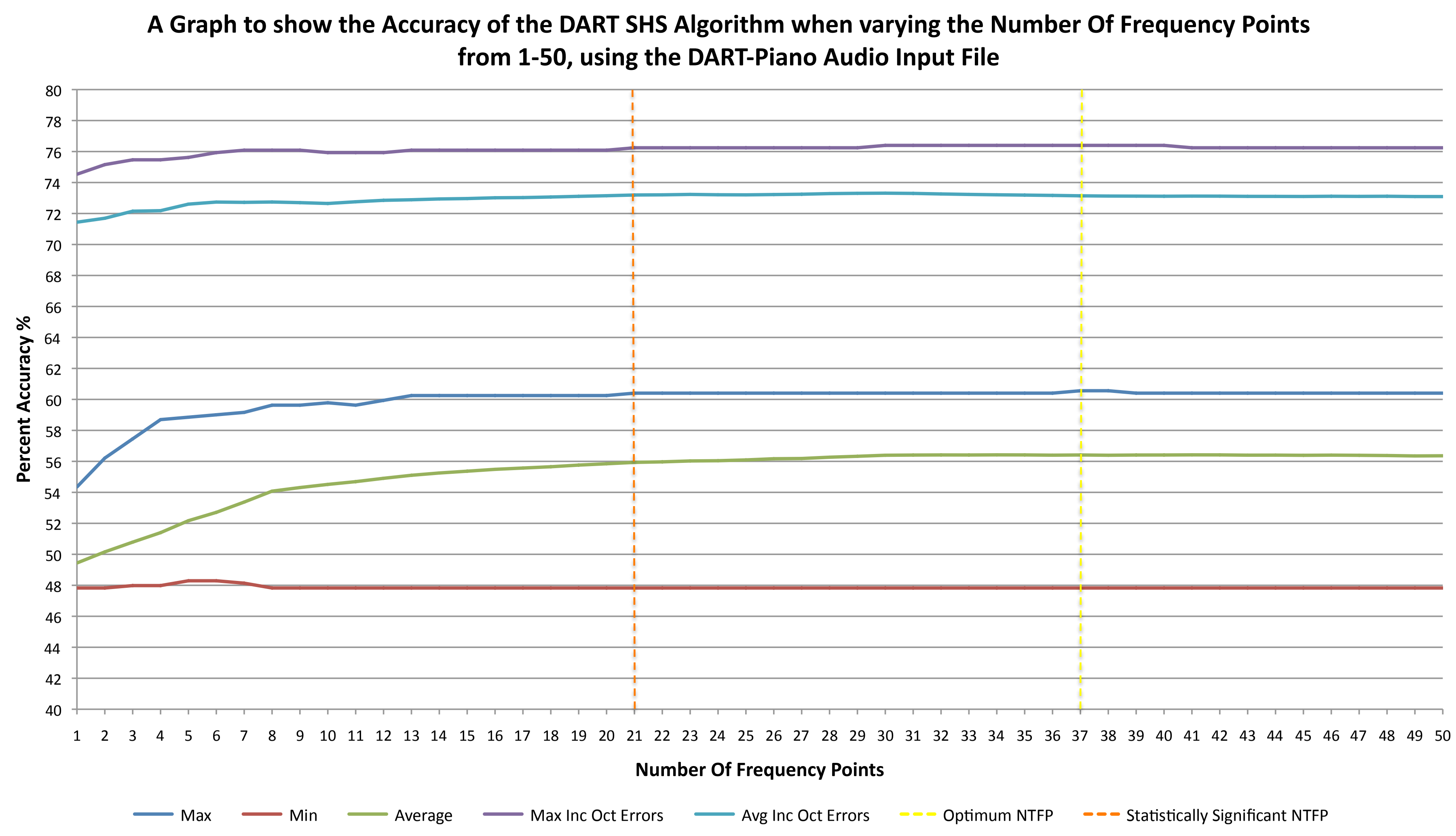


Figure 7.31: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using the DART Violin audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	54.35	47.83	49.44	74.53	71.44	74.53	22.67	20.19	22.00
2	56.21	47.83	50.16	75.16	71.69	75.16	22.67	18.94	21.54
3	57.45	47.98	50.79	75.47	72.15	75.47	22.52	18.01	21.36
4	58.70	47.98	51.40	75.47	72.18	75.47	22.52	16.77	20.78
5	58.85	48.29	52.17	75.62	72.60	75.62	22.20	16.77	20.43
6	59.01	48.29	52.71	75.93	72.74	75.93	22.52	16.93	20.03
7	59.16	48.14	53.37	76.09	72.72	76.09	21.89	16.30	19.35
8	59.63	47.83	54.08	76.09	72.74	76.09	22.20	16.30	18.66
9	59.63	47.83	54.31	76.09	72.70	76.09	22.36	16.30	18.39
10	59.78	47.83	54.51	75.93	72.65	75.93	22.36	16.15	18.13
11	59.63	47.83	54.69	75.93	72.76	75.93	22.36	16.15	18.07
12	59.94	47.83	54.91	75.93	72.85	75.93	22.36	15.99	17.94
13	60.25	47.83	55.10	76.09	72.89	76.09	22.36	15.84	17.79
14	60.25	47.83	55.25	76.09	72.94	76.09	22.36	15.84	17.69
15	60.25	47.83	55.37	76.09	72.96	76.09	22.36	15.84	17.60
16	60.25	47.83	55.49	76.09	73.02	76.09	22.36	15.84	17.53
17	60.25	47.83	55.57	76.09	73.03	76.09	22.20	15.84	17.46
18	60.25	47.83	55.65	76.09	73.06	76.09	22.20	15.84	17.41
19	60.25	47.83	55.76	76.09	73.11	76.09	22.20	15.84	17.35
20	60.25	47.83	55.85	76.09	73.15	76.09	22.20	15.84	17.30
21	*60.40*	*47.83*	*55.93*	*76.24*	*73.19*	*76.24*	*22.20*	*15.84*	*17.26*
22	60.40	47.83	55.97	76.24	73.20	76.24	22.20	15.84	17.24
23	60.40	47.83	56.03	76.24	73.23	76.24	22.20	15.84	17.21
24	60.40	47.83	56.04	76.24	73.21	76.24	22.20	15.68	17.17
25	60.40	47.83	56.09	76.24	73.20	76.24	22.20	15.68	17.11
26	60.40	47.83	56.17	76.24	73.22	76.24	22.20	15.68	17.05
27	60.40	47.83	56.18	76.24	73.24	76.24	22.20	15.68	17.06
28	60.40	47.83	56.27	76.24	73.28	76.24	22.20	15.68	17.02
29	60.40	47.83	56.33	76.24	73.30	76.24	22.20	15.68	16.98

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
30	60.40	47.83	56.40	76.40	73.31	76.40	22.20	15.68	16.92
31	60.40	47.83	56.41	76.40	73.30	76.40	22.20	15.68	16.89
32	60.40	47.83	56.41	76.40	73.26	76.40	22.20	15.68	16.85
33	60.40	47.83	56.41	76.40	73.23	76.40	22.20	15.68	16.82
34	60.40	47.83	56.42	76.40	73.21	76.40	22.20	15.53	16.79
35	60.40	47.83	56.42	76.40	73.19	76.40	22.20	15.53	16.77
36	60.40	47.83	56.40	76.40	73.17	76.40	22.20	15.37	16.77
37	**60.56**	**47.83**	**56.41**	**76.40**	**73.14**	**76.40**	**22.20**	**15.22**	**16.73**
38	60.56	47.83	56.40	76.40	73.13	76.40	22.20	15.22	16.73
39	60.40	47.83	56.41	76.40	73.12	76.40	22.20	15.22	16.71
40	60.40	47.83	56.41	76.40	73.11	76.40	22.20	15.22	16.70
41	60.40	47.83	56.42	76.24	73.12	76.24	22.20	15.37	16.70
42	60.40	47.83	56.42	76.24	73.12	76.24	22.20	15.37	16.70
43	60.40	47.83	56.40	76.24	73.11	76.24	22.20	15.37	16.71
44	60.40	47.83	56.40	76.24	73.10	76.24	22.20	15.37	16.70
45	60.40	47.83	56.39	76.24	73.10	76.24	22.20	15.22	16.71
46	60.40	47.83	56.40	76.24	73.11	76.24	22.20	15.22	16.71
47	60.40	47.83	56.39	76.24	73.10	76.24	22.20	15.22	16.71
48	60.40	47.83	56.38	76.24	73.11	76.24	22.20	15.22	16.74
49	60.40	47.83	56.35	76.24	73.10	76.24	22.20	15.22	16.75
50	60.40	47.83	56.36	76.24	73.09	76.24	22.20	15.37	16.73

Table 7.21: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Piano audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.5.5 Number Of Frequency Points Results for Tubular Bells

Graph 7.32 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-50, using the DART Tubular Bells audio input file. Table 7.22 shows the values presented in Graph 7.32.

Graph 7.32 shows that while the greatest accuracy is discovered when analysing **36** frequency points, looking at more than **24** frequency points gives only an extremely minor improvement in maximum and average accuracies (under 1%). When allowing for octave errors, the maximum and average accuracies had reached their maximum points even at low NTFP values of around 6.

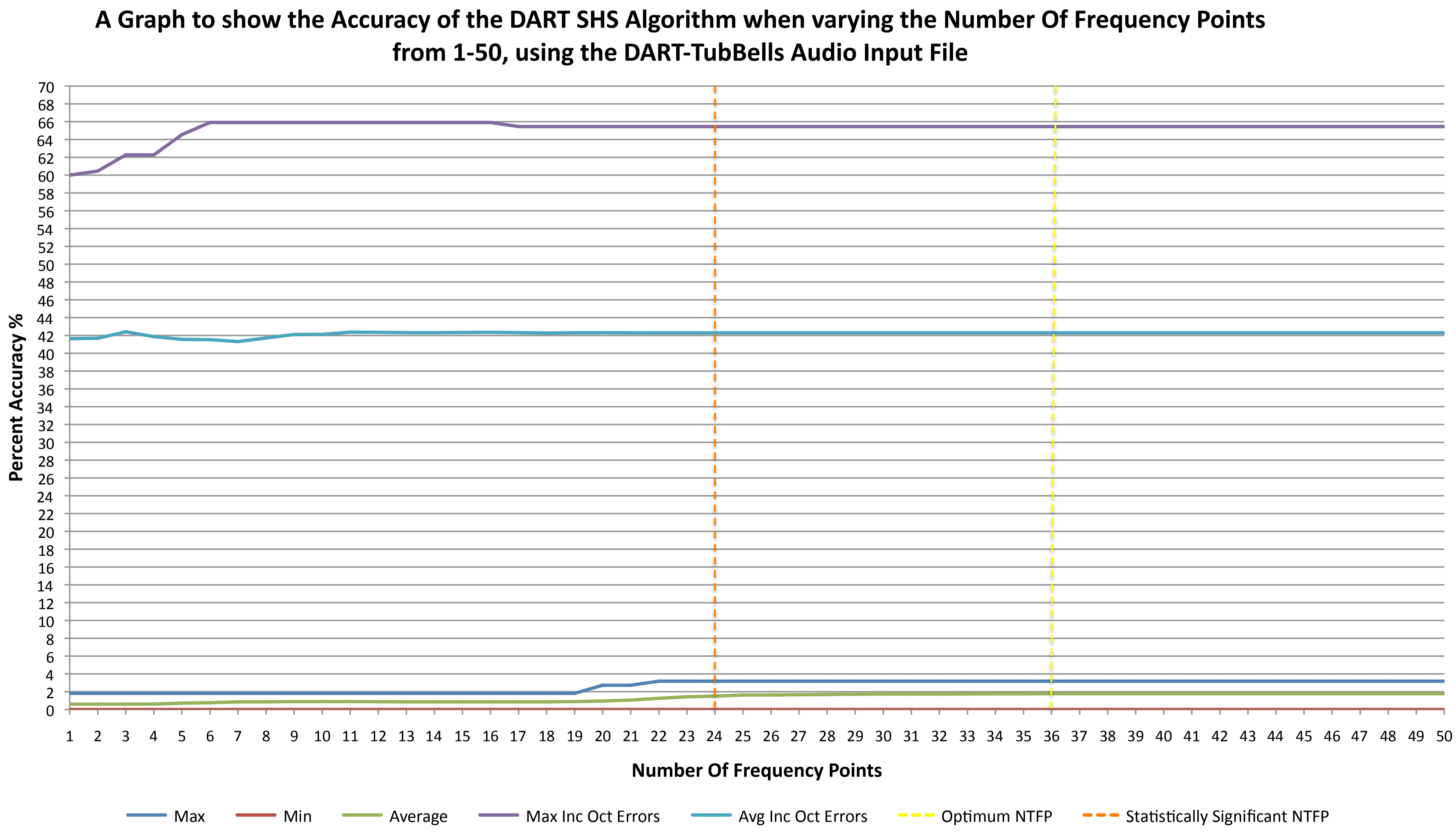


Figure 7.32: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using the DART Tubular Bells audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
1	1.82	0.00	0.60	60.00	41.64	41.82	60.00	38.18	41.04
2	1.82	0.00	0.60	60.45	41.68	41.82	60.45	38.18	41.08
3	1.82	0.00	0.60	62.27	42.42	41.82	62.27	38.18	41.81
4	1.82	0.00	0.62	62.27	41.86	41.82	62.27	36.36	41.25
5	1.82	0.00	0.71	64.55	41.56	41.82	64.55	35.91	40.84
6	1.82	0.00	0.76	65.91	41.53	40.00	65.91	37.73	40.76
7	1.82	0.00	0.86	65.91	41.31	40.45	65.91	36.36	40.45
8	1.82	0.00	0.86	65.91	41.72	41.82	65.91	37.73	40.86
9	1.82	0.00	0.89	65.91	42.11	41.82	65.91	38.18	41.21
10	1.82	0.00	0.89	65.91	42.12	41.82	65.91	38.18	41.23
11	1.82	0.00	0.89	65.91	42.37	43.64	65.91	38.18	41.47
12	1.82	0.00	0.88	65.91	42.35	43.64	65.91	38.18	41.47
13	1.82	0.00	0.86	65.91	42.32	43.64	65.91	38.18	41.46
14	1.82	0.00	0.86	65.91	42.32	43.64	65.91	38.18	41.46
15	1.82	0.00	0.86	65.91	42.33	43.64	65.91	38.18	41.47
16	1.82	0.00	0.86	65.91	42.35	43.64	65.91	38.18	41.49
17	1.82	0.00	0.86	65.45	42.32	43.64	65.45	38.18	41.46
18	1.82	0.00	0.86	65.45	42.27	43.18	65.45	38.18	41.41
19	1.82	0.00	0.89	65.45	42.30	43.18	65.45	38.18	41.41
20	2.73	0.00	0.95	65.45	42.31	40.00	65.45	37.27	41.36
21	2.73	0.00	1.06	65.45	42.29	40.00	65.45	37.27	41.24
22	3.18	0.00	1.26	65.45	42.29	40.91	65.45	37.27	41.03
23	3.18	0.00	1.43	65.45	42.29	41.36	65.45	36.82	40.87
24	*3.18*	*0.00*	*1.50*	*65.45*	*42.29*	*41.36*	*65.45*	*36.82*	*40.79*
25	3.18	0.00	1.63	65.45	42.29	41.36	65.45	36.82	40.66
26	3.18	0.00	1.63	65.45	42.29	41.36	65.45	36.82	40.66
27	3.18	0.00	1.66	65.45	42.29	41.36	65.45	36.82	40.63
28	3.18	0.00	1.69	65.45	42.29	41.36	65.45	36.82	40.60
29	3.18	0.00	1.72	65.45	42.29	41.36	65.45	36.82	40.57

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	3.18	0.00	1.73	65.45	42.29	41.36	65.45	36.82	40.56
31	3.18	0.00	1.73	65.45	42.29	41.36	65.45	36.82	40.56
32	3.18	0.00	1.73	65.45	42.29	41.36	65.45	36.82	40.56
33	3.18	0.00	1.75	65.45	42.29	41.36	65.45	36.82	40.54
34	3.18	0.00	1.76	65.45	42.29	41.36	65.45	36.82	40.53
35	3.18	0.00	1.76	65.45	42.29	41.36	65.45	36.82	40.53
36	**3.18**	**0.00**	**1.78**	**65.45**	**42.29**	**41.36**	**65.45**	**36.82**	**40.51**
37	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
38	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
39	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
40	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
41	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
42	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
43	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
44	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
45	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
46	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
47	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
48	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
49	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51
50	3.18	0.00	1.78	65.45	42.29	41.36	65.45	36.82	40.51

Table 7.22: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Tubular Bells audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.5.6 Number Of Frequency Points Results for Distorted Guitar

Graph 7.33 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-50, using the DART Distorted Guitar audio input file. Table 7.23 shows the values presented in Graph 7.33.

Graph 7.33 shows that while the greatest accuracy is discovered when analysing the maximum of **50** frequency points, looking at more than **41** frequency points gives only a minor improvement in maximum and average accuracies (under 1%). However, overall accuracies levels appeared to rise as the NTFP value increased.

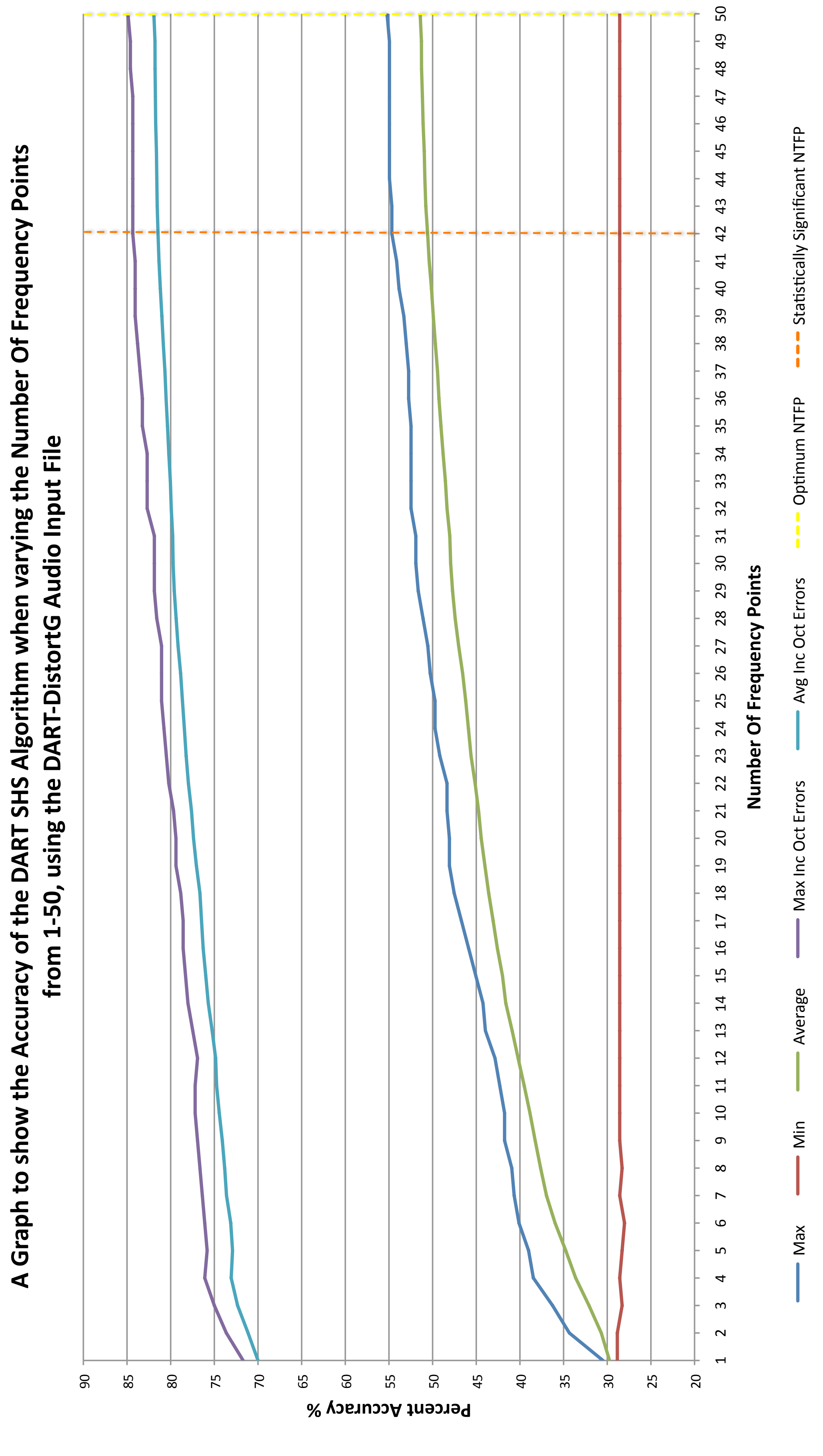


Figure 7.33: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using the DART Distorted Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	30.49	28.85	29.74	71.70	70.00	69.78	41.48	39.29	40.26
2	34.34	28.85	30.68	73.63	71.14	72.80	41.48	38.46	40.45
3	36.26	28.30	32.09	75.00	72.34	75.00	41.48	37.91	40.24
4	38.46	28.57	33.61	76.10	73.07	75.55	41.76	37.09	39.46
5	39.01	28.30	34.75	75.82	72.91	75.82	41.76	36.54	38.16
6	40.11	28.02	35.98	76.10	73.12	76.10	42.03	35.99	37.14
7	40.66	28.57	36.96	76.37	73.60	76.37	42.03	35.16	36.64
8	40.93	28.30	37.63	76.65	73.81	76.65	41.76	34.34	36.18
9	41.76	28.57	38.25	76.92	74.09	76.10	41.76	34.07	35.83
10	41.76	28.57	38.85	77.20	74.43	77.20	41.76	33.79	35.58
11	42.31	28.57	39.51	77.20	74.74	77.20	41.76	33.52	35.22
12	42.86	28.57	40.21	76.92	74.85	76.92	41.76	32.97	34.64
13	43.96	28.57	40.89	77.47	75.25	76.65	41.76	32.42	34.36
14	44.23	28.57	41.62	78.02	75.69	76.65	41.76	31.59	34.07
15	45.05	28.57	42.00	78.30	75.98	76.92	41.76	31.59	33.98
16	45.88	28.57	42.59	78.57	76.28	77.75	41.76	31.04	33.69
17	46.70	28.57	43.07	78.57	76.47	78.02	41.76	30.77	33.40
18	47.53	28.57	43.58	78.85	76.65	78.02	41.76	30.49	33.07
19	48.08	28.57	44.01	79.40	77.05	78.30	41.76	30.22	33.04
20	48.08	28.57	44.44	79.40	77.38	79.12	41.76	30.22	32.95
21	48.35	28.57	44.74	79.67	77.62	78.85	41.76	30.49	32.88
22	48.35	28.57	45.13	80.22	77.97	80.22	41.76	30.49	32.84
23	49.18	28.57	45.60	80.49	78.23	79.12	41.76	29.95	32.62
24	49.73	28.57	45.90	80.77	78.43	79.67	41.76	29.95	32.53
25	49.73	28.57	46.21	81.04	78.65	79.67	41.76	29.95	32.44
26	50.27	28.57	46.56	81.04	78.86	79.95	41.76	29.67	32.30
27	50.55	28.57	47.02	81.04	79.15	79.95	41.76	29.40	32.14
28	51.10	28.57	47.42	81.59	79.37	80.22	41.76	29.12	31.96
29	51.65	28.57	47.72	81.87	79.59	80.77	41.76	29.12	31.87

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	51.92	28.57	47.93	81.87	79.72	81.04	41.76	29.12	31.79
31	51.92	28.57	48.03	81.87	79.77	81.04	41.76	29.12	31.74
32	52.47	28.57	48.35	82.69	79.93	81.32	41.76	28.85	31.59
33	52.47	28.57	48.54	82.69	80.04	81.32	41.76	28.85	31.50
34	52.47	28.57	48.80	82.69	80.22	81.32	41.76	28.85	31.41
35	52.47	28.57	49.05	83.24	80.36	81.32	41.76	28.57	31.32
36	52.75	28.57	49.28	83.24	80.53	81.59	41.76	28.02	31.25
37	52.75	28.57	49.44	83.52	80.66	82.14	41.76	27.75	31.22
38	53.02	28.57	49.69	83.79	80.85	82.42	41.76	27.75	31.16
39	53.30	28.57	49.92	84.07	81.01	82.69	41.76	27.75	31.09
40	53.85	28.57	50.13	84.07	81.18	82.97	41.76	27.47	31.05
41	54.12	28.57	50.39	84.07	81.34	83.24	41.76	27.20	30.95
42	*54.67*	*28.57*	*50.57*	*84.34*	*81.45*	*83.52*	*41.76*	*27.20*	*30.88*
43	54.67	28.57	50.78	84.34	81.54	83.52	41.76	27.20	30.76
44	54.95	28.57	50.89	84.34	81.59	83.79	41.76	27.20	30.70
45	54.95	28.57	50.96	84.34	81.63	83.79	41.76	27.20	30.67
46	54.95	28.57	51.09	84.34	81.72	83.79	42.03	26.92	30.63
47	54.95	28.57	51.17	84.34	81.75	83.79	42.03	26.65	30.58
48	54.95	28.57	51.27	84.62	81.79	83.79	42.03	26.37	30.52
49	54.95	28.57	51.28	84.62	81.80	83.79	42.03	26.37	30.52
50	**55.22**	**28.57**	**51.43**	**84.89**	**81.93**	**84.07**	**42.03**	**26.10**	**30.50**

Table 7.23: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Distorted Guitar audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.5.7 Number Of Frequency Points Results for All Audio Files

Graph 7.34 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-50, using all DART audio input files. Table 7.24 shows the values presented in Graph 7.34.

Graph 7.34 shows that while the greatest accuracy is discovered when analysing the maximum of **50** frequency points, looking at more than **22** frequency points gives only a minor improvement in maximum and average accuracies (under 1%).

When allowing for octave errors, the maximum and average accuracies were very high even at low NTFP values of around 6-7.

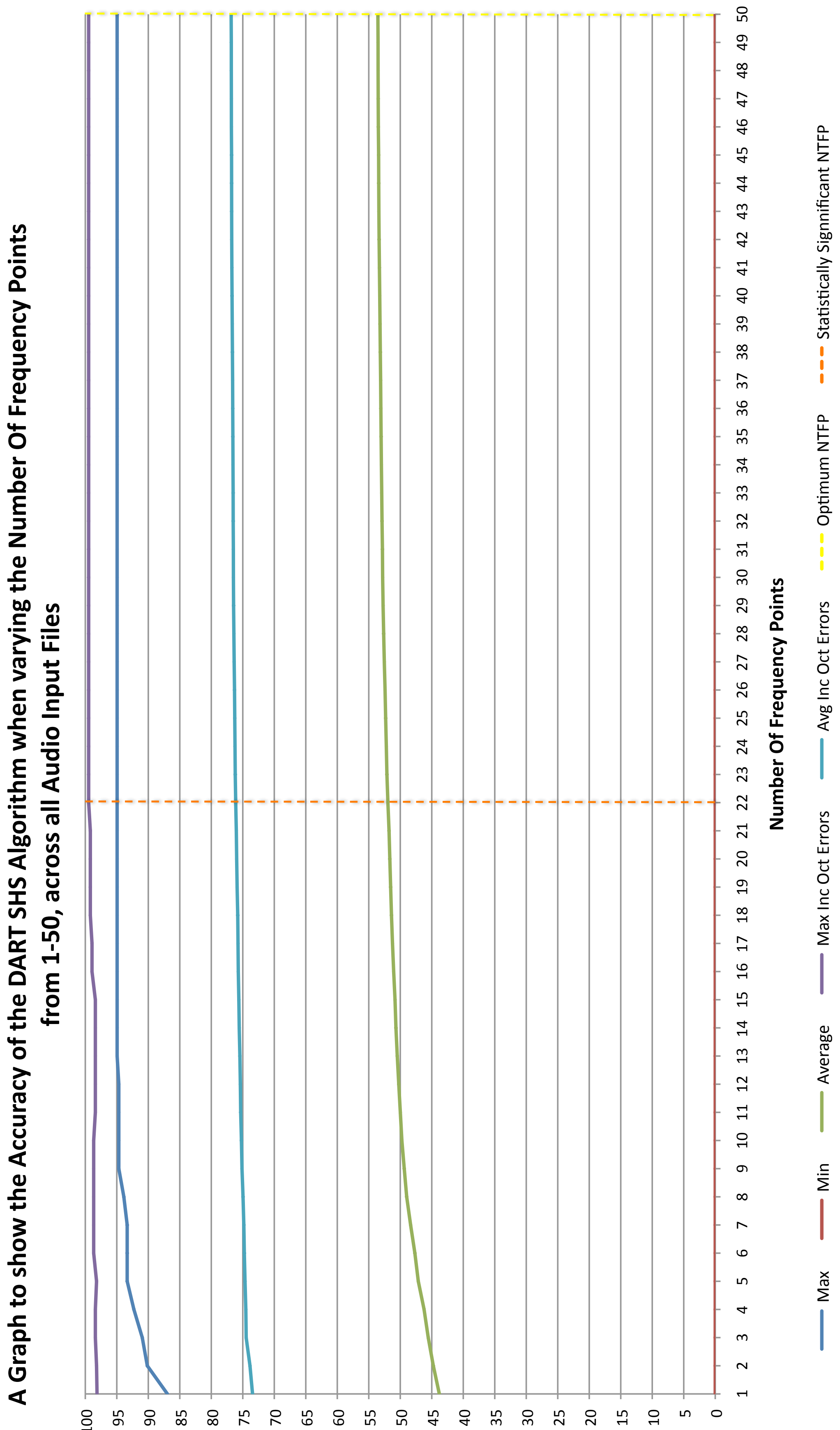


Figure 7.34: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using all DART audio input files. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	86.97	0.00	43.80	98.14	73.45	98.14	60.00	11.17	29.65
2	90.16	0.00	44.76	98.21	73.86	98.14	60.45	7.98	29.10
3	90.96	0.00	45.56	98.40	74.46	98.14	62.27	7.18	28.90
4	92.29	0.00	46.22	98.40	74.52	98.14	62.27	5.85	28.30
5	93.35	0.00	47.14	98.21	74.65	98.14	64.55	4.79	27.51
6	93.35	0.00	47.68	98.67	74.76	98.14	65.91	4.79	27.08
7	93.35	0.00	48.36	98.67	74.83	98.14	65.91	4.79	26.48
8	93.88	0.00	48.98	98.67	74.96	98.14	65.91	4.26	25.98
9	94.68	0.00	49.37	98.67	75.13	98.14	65.91	3.46	25.76
10	94.68	0.00	49.75	98.67	75.22	98.14	65.91	3.46	25.47
11	94.68	0.00	49.99	98.40	75.33	98.14	65.91	3.46	25.34
12	94.68	0.00	50.24	98.40	75.39	98.14	65.91	3.46	25.15
13	94.95	0.00	50.48	98.40	75.46	98.14	65.91	3.19	24.99
14	94.95	0.00	50.69	98.40	75.56	98.14	65.91	3.19	24.88
15	94.95	0.00	50.83	98.40	75.63	98.14	65.91	3.19	24.79
16	94.95	0.00	51.05	98.94	75.71	98.14	65.91	3.19	24.67
17	94.95	0.00	51.24	98.94	75.76	98.14	65.45	3.19	24.52
18	94.95	0.00	51.40	99.20	75.80	98.14	65.45	3.19	24.41
19	94.95	0.00	51.53	99.20	75.90	98.14	65.45	3.19	24.37
20	94.95	0.00	51.66	99.20	75.97	98.14	65.45	3.19	24.31
21	94.95	0.00	51.78	99.20	76.03	98.14	65.45	3.19	24.25
22	*94.95*	*0.00*	*51.96*	*99.47*	*76.12*	*98.14*	*65.45*	*3.19*	*24.16*
23	94.95	0.00	52.12	99.47	76.20	98.14	65.45	3.19	24.08
24	94.95	0.00	52.22	99.47	76.24	98.14	65.45	3.19	24.02
25	94.95	0.00	52.32	99.47	76.27	98.14	65.45	3.19	23.95
26	94.95	0.00	52.41	99.47	76.31	98.14	65.45	3.19	23.90
27	94.95	0.00	52.54	99.47	76.37	98.14	65.45	3.19	23.83
28	94.95	0.00	52.64	99.47	76.41	98.14	65.45	3.19	23.78
29	94.95	0.00	52.73	99.47	76.46	98.14	65.45	3.19	23.73

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
30	94.95	0.00	52.80	99.47	76.49	98.14	65.45	3.19	23.69
31	94.95	0.00	52.83	99.47	76.50	98.14	65.45	3.19	23.67
32	94.95	0.00	52.90	99.47	76.52	98.14	65.45	3.19	23.63
33	94.95	0.00	52.94	99.47	76.53	98.14	65.45	3.19	23.59
34	94.95	0.00	53.00	99.47	76.56	98.14	65.45	3.19	23.56
35	94.95	0.00	53.04	99.47	76.58	98.14	65.45	3.19	23.53
36	94.95	0.00	53.08	99.47	76.60	98.14	65.45	3.19	23.51
37	94.95	0.00	53.12	99.47	76.61	98.14	65.45	3.19	23.50
38	94.95	0.00	53.17	99.47	76.65	98.14	65.45	3.19	23.48
39	94.95	0.00	53.21	99.47	76.68	98.14	65.45	3.19	23.46
40	94.95	0.00	53.26	99.47	76.71	98.14	65.45	3.19	23.45
41	94.95	0.00	53.31	99.47	76.73	98.14	65.45	3.19	23.42
42	94.95	0.00	53.36	99.47	76.76	98.14	65.45	3.19	23.40
43	94.95	0.00	53.40	99.47	76.77	98.14	65.45	3.19	23.37
44	94.95	0.00	53.42	99.47	76.78	98.14	65.45	3.19	23.36
45	94.95	0.00	53.44	99.47	76.78	98.14	65.45	3.19	23.35
46	94.95	0.00	53.48	99.47	76.81	98.14	65.45	3.19	23.33
47	94.95	0.00	53.50	99.47	76.81	98.14	65.45	3.19	23.31
48	94.95	0.00	53.52	99.47	76.82	98.14	65.45	3.19	23.30
49	94.95	0.00	53.53	99.47	76.82	98.14	65.45	3.19	23.30
50	**94.95**	**0.00**	**53.56**	**99.47**	**76.85**	**98.14**	**65.45**	**3.19**	**23.29**

Table 7.24: This table details the accuracy of the DART SHS algorithm while varying the NTFP using all DART audio input files. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

7.5.8 Number Of Top Frequency Points Accuracy Summary

A summary of the NTFP results presented so far is presented in Table 7.25. This table shows, for each audio input file, the optimal NTFP value (the minimum number that gives the maximum accuracy whereby increasing the NTFP shows no further improvement), as well as the 'Statistically Significant' figure whereby only minimal improvement is shown when increasing the NTFP value any further. In table 7.25, these two numbers are called the **'Minimum'** and **'Maximum'**, respectively.

The NTFP that consistently produced the most accurate results across all the variables was ***50***. Generally, the overall accuracy continues to increase as the NTFP increases, however when visually analysing the graphs, only minor increases are found after approximately **15 points**. In the case of the Distorted Guitar however, the accuracy increases nearly linearly until a NTFP value of around 42.

Audio Input File	***Minimum NTFP***	***Maximum NTFP***
Acoustic Guitar	13	**33**
Oboe	24	**50**
Violin	23	**50**
Piano	21	**37**
Tubular Bells	24	**36**
Distorted Guitar	42	**50**
All Audio Files	22	**50**

Table 7.25: This table shows a summary of the Number Of Top Frequency Points that give the most accurate results for each Audio Input File.

One explanation for the continued increase of accuracy of the SHS algorithm as the NTFP value rose could be due to 'residual' or 'noisy' peaks around the base of the fundamental frequency. The audio data was not a pure sine wave as presented in Section 7.3.8.

Section 7.3.8 also shows that when a pure sine wave is analysed, if the waveform is non periodic and utilises zero-padding (both will be true in the case of the DART experiments), residual noise can be introduced either side of the main peaks. While using zero-padding gives an increased spectral resolution and a higher fundamental amplitude peak, Figure 7.20 shows the type of noise that can be introduced to an otherwise noisy signal. Add to this the effect of a less than ideal FFT Window choice for a particular audio input file and it is easy to hypothesise that these factors could explain why the accuracy of the SHS algorithm continued NTFP value continued to rise.

In order to further understand these results, 0.5 second samples of audio (piano, guitar, tubular bells from the DART audio input files) were put through a similar Triana task graph as presented in Figure 7.12, however replacing the **`Wave`** unit with a **`LoadSound`** unit. This workflow allows the FFT of an audio file to be displayed using the **`JPlotter`** unit.

The FFT of two of the audio files are displayed in detail in Figures 7.35 and 7.36. These represent a 0.5 second sample of a Piano and a Distorted Guitar playing a C2 note (65.41Hz) respectively, with the FFT displaying frequencies from 0Hz to 2000Hz to show all of the peaks.

Figures 7.37 and 7.38 zoom in on the highest amplitude peaks in each FFT. Figure 7.37 shows no peak at the fundamental of 65Hz but shows a strong peak at an octave higher, around 130Hz. This is displayed in Figure 7.37 and shows a lack of any major noise around the base of the peak. Figure 7.38 shows the highest peak at around 260Hz, an octave higher again, although there is a smaller peak at 130Hz. Both these particular results would give the correct note, but the wrong octave, however both of these samples show a lack of peaks around the base.

While these samples do not exhibit a large amount of residual peaks around the base of each main amplitude spike, others audio samples from the thousands analysed in DART may. Two more cases are presented; a high-pitched Distorted Guitar note D5 (587.33Hz), and a high-pitched Tubular Bells note G5 (783.99Hz) presented in Figures 7.39 and 7.41 respectively.

Figures 7.40 and 7.42 zoom in on the highest amplitude peaks for their respective FFTs. For the distorted guitar, Figure 7.40 shows no peak at the fundamental of 587.33Hz, however shows a strong peak at an octave higher at 1174.66Hz. This is displayed in 7.40 and shows a lack of any real noise around the base of the peak.

Figure 7.42 shows the highest peak at around 1568Hz, again an octave higher, although

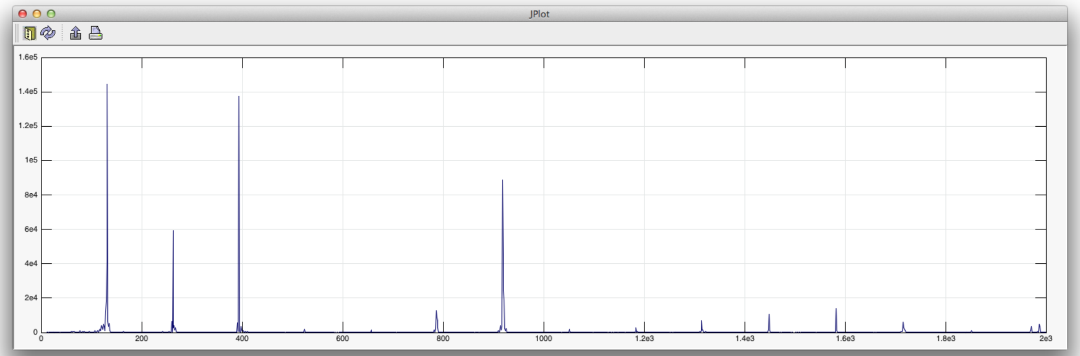


Figure 7.35: An FFT of a 0.5 second long C2 note played on a Piano, taken from the DART Piano audio input file. There was no peak at the fundamental of 65.42Hz in this example. This graph shows a frequency range of 0-2000Hz

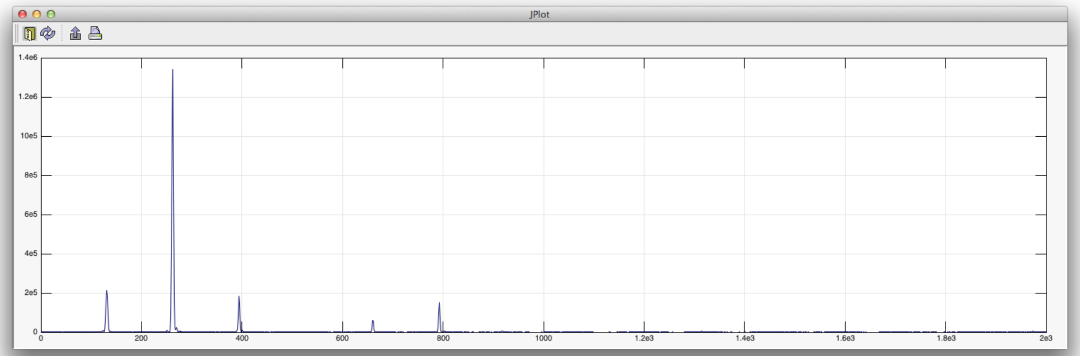


Figure 7.36: An FFT of a 0.5 second long C2 note played on a Distorted Guitar, taken from the DART DistortG audio input file. There was no peak at the fundamental of 65.42Hz in this example. This graph shows a frequency range of 0-2000Hz

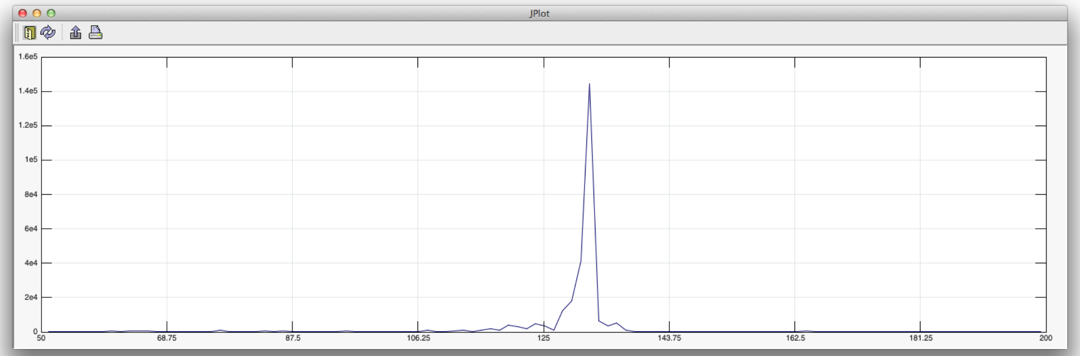


Figure 7.37: An FFT of the 0.5 second Piano C2 note shown in 7.35, zoomed in at 50-200Hz showing the peak with the largest amplitude in detail

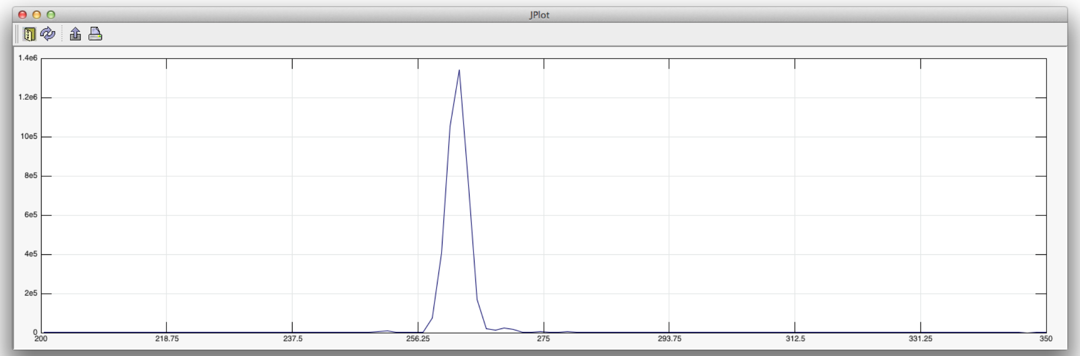


Figure 7.38: An FFT of the 0.5 second Distorted Guitar C2 note shown in 7.36, zoomed in at 200-350Hz showing the peak with the largest amplitude in detail

there is an smaller peak at 783.99Hz. Both these particular results would give the correct note, but the wrong octave. Figure 7.43 shows the second largest peak for the Tubular Bells G2 note, this time zooming on on the 2500-2900Hz range. This note *does* show a number of peaks around the base of the main peak.

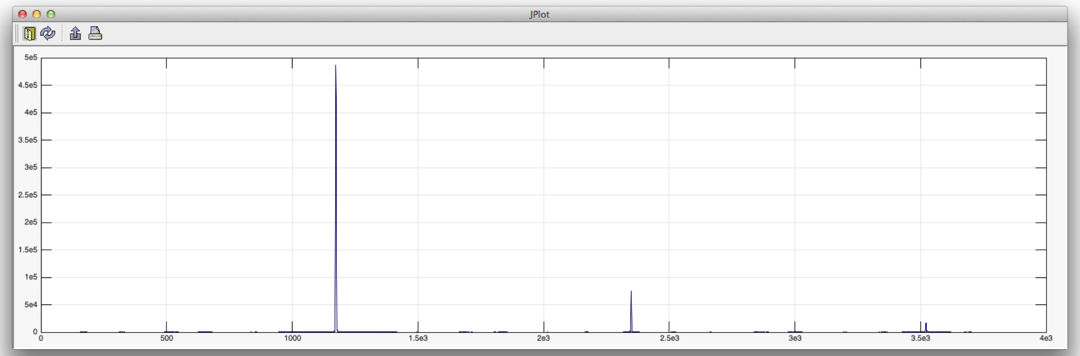


Figure 7.39: An FFT of a 0.5 second long C5 note played on a Distorted Guitar. This graph shows a frequency range of 0-4000Hz. There was no peak at the fundamental of 587.33Hz in this example

The samples examined in this section only contain a few main amplitude peak points, making it possible to visually count the critical ones when searching for around 15 NTFP. However this may not be this is true for other instruments, all of the time. In some cases for some frequencies, the situation may be a lot worse because of the frequency spacing and spread of energy and it may be possible to have 20 or 30 frequencies that are higher than the ones needed to accurately extract the pitch.

In general, these experiments have shown that the accuracy of the SHS algorithm kept

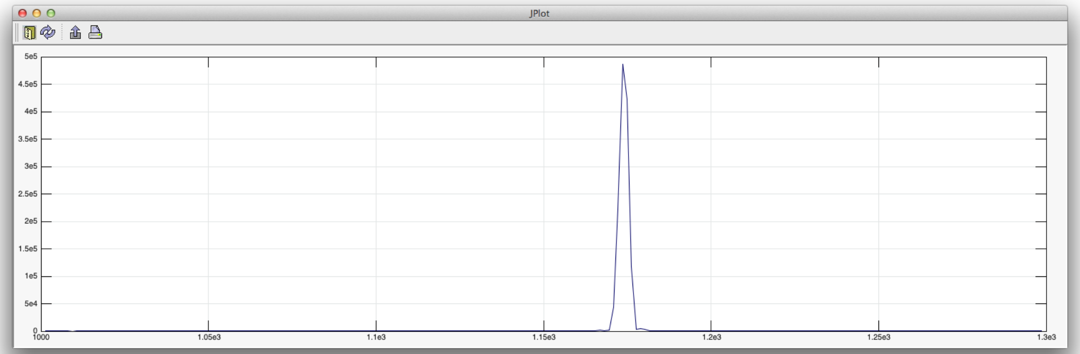


Figure 7.40: An FFT of the 0.5 second Distorted Guitar note shown above, zoomed in at 1000-1300Hz showing the peak with the largest amplitude in detail at around 1174Hz

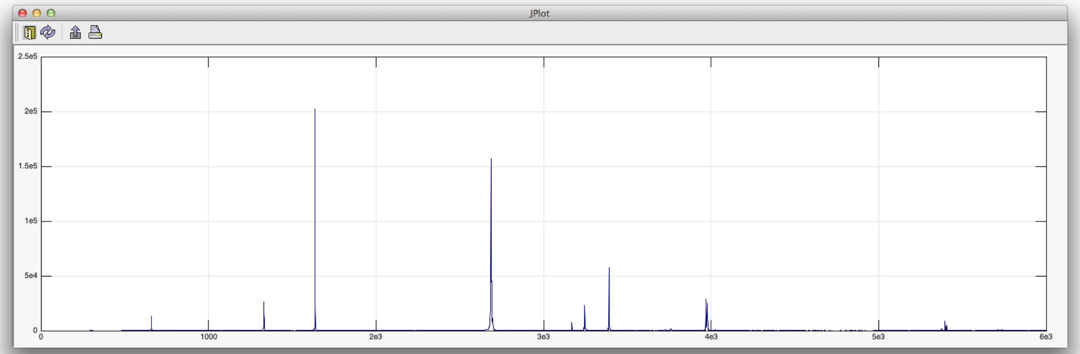


Figure 7.41: An FFT of a 0.5 second long G5 note played on a Tubular Bells. Two large peaks are prominent, with only a very small peak at the fundamental frequency of approximately 784Hz

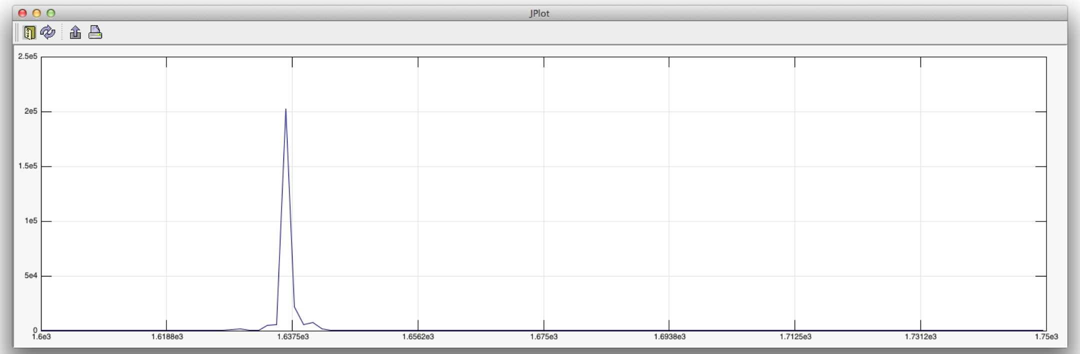


Figure 7.42: An FFT of a 0.5 second Tubular Bells G5 note zoomed in at 1600-1750Hz showing the peak with the largest amplitude in detail

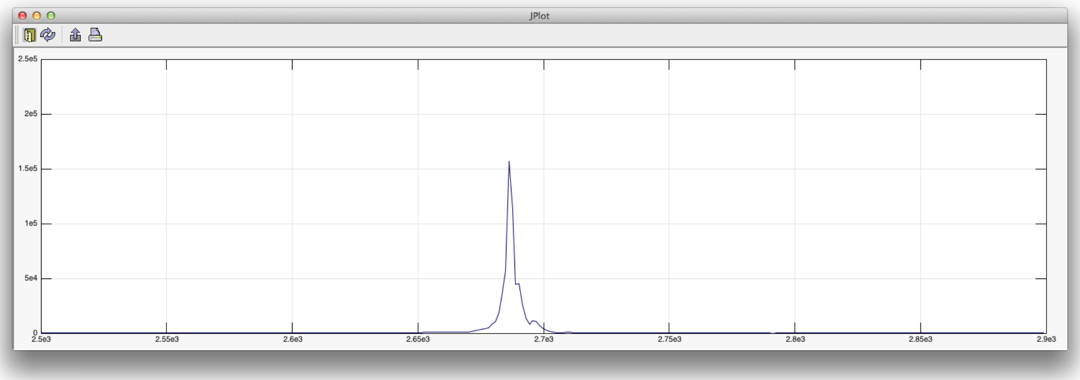


Figure 7.43: An FFT of a 0.5 second Tubular Bells G5 note zoomed in at 2500-2900Hz showing the peak with the *second largest* amplitude in detail

increasing as the NTFP value increased, however once past 12-15 points, the gains were less significant. Given a wider and higher range of NTFP values it would seem intuitive that the overall accuracy could continue to improve because the extra range would provide more robustness to using pitch templating; summing the harmonics is similar to templating because the algorithm counts frequencies at specific points. If these frequencies are not present (for example, because they are the 60th highest amplitude in the spectrum) then the SHS algorithm will not count them and this will lead to an error.

Further parameter sweep experiments to extend the range of the NTFP value would allow for further investigation and clarification of the optimal NTFP value. Chapter 7 investigates this and presents the results of the further experimentation.

7.6 SHS Analysis Summary

This section presents a summary of the results obtained through the large scale DART parameter sweep experiments to try to find the optimum values for the implemented Sub-Harmonic Summation algorithm. The most optimal variables found for maximum accuracy across all audio files so far are:

- **FFT Window**: Rectangle
- **NTFP**: 50
- **No. Harmonics**: 15

Table 7.26 shows the optimum variables for each audio input type.

Audio Input File	***FFT Window***	***NTFP***	**No. Harmonics**
Acoustic Guitar	Welch	33	14
Oboe	Rectangle	50	13
Violin	Rectangle	50	12
Piano	Rectangle	37	12
Tubular Bells	Rectangle	36	13
Distorted Guitar	Nutall4	50	15
All Audio Files	Rectangle	50	15

Table 7.26: This table shows a summary of the FFT Windows that give the most accurate results for each Audio Input File.

7.7 Time Analysis

7.7.1 XtremWeb Time Analysis

The total processing time for analysing 268,796 results over 214 worker nodes on the GRID5000 platform was **138,173,89** seconds (*3,838.17 hours, or around 160 days*). These figures are calculated by adding together the total run time, in seconds, from the 268,796 results files retrieved. These figures do not include any of the XWHEP job submission and results or data download times.

The XremWeb job submission process began on 27/07/2011 at 3:47pm and the jobs were submitted by 13/08/2011 at 7:21. This means that 160 days of DART processing was completed in just 17 days, 3 hours and 34 minutes using XtremWeb. This figure this does not show the whole picture or give the entire makespan of the XtremWeb experiments, however.

The XtremWeb macro file containing the 268,800 job submission commands was split into 27 smaller macros, each containing 10,000 jobs (with the final file containing a smaller number of jobs). These 27 macros were submitted *manually* across a 22 day period, however jobs were only submitted on 14 of the 17 days, with more than one macro being submitted per day. Figure 7.47 shows the number of jobs completed each day. The maximum number of submitted macros in a single day was 4, with 40,000 jobs completed in this single day (with no errors). Some days had an extremely low number of job executions (as low as 4) due to the resubmission of any jobs from the previous day that returned with errors. According to the SQL database logs, a total of 277,779 jobs were submitted in total; it is presumed that this is due to errors and include jobs that were automatically resubmitted. The average number of jobs completed per (active) day was 15,432.

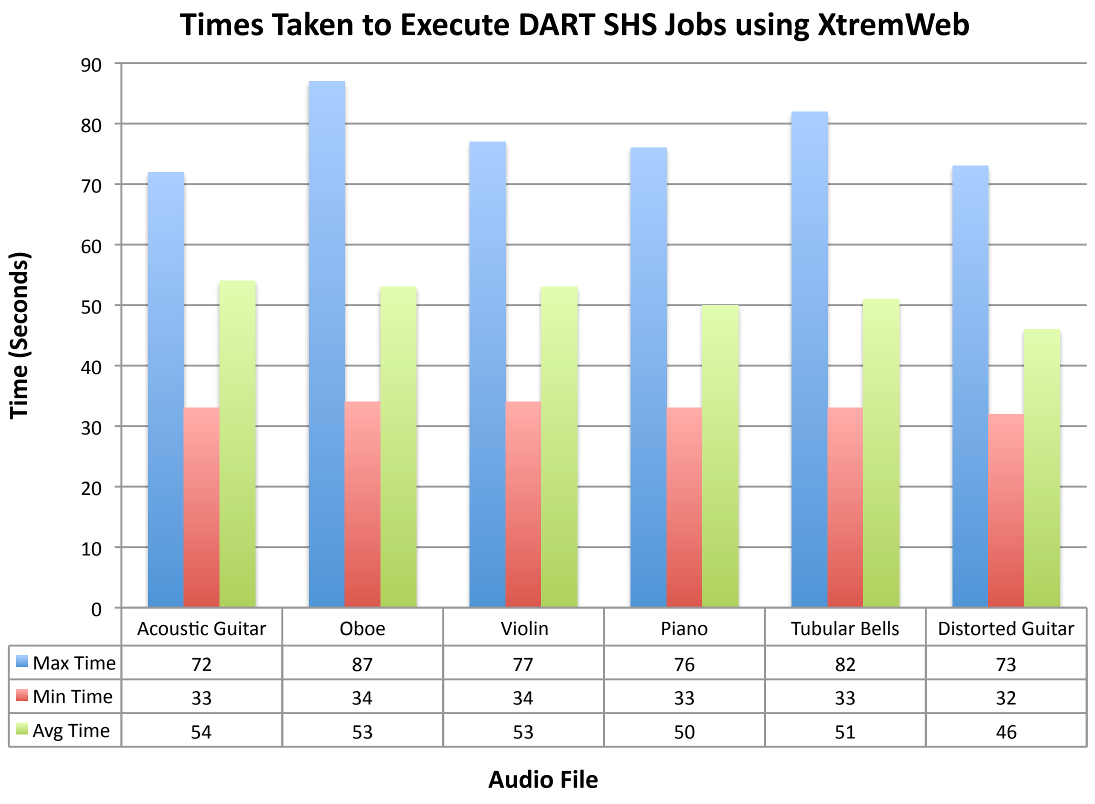
Finding the *makespan* is more difficult due to a bug in the implementation of XtremWeb. This bug made it impossible to see how many times a job was submitted before it was completed successfully (multiple submission attempts obviously extend the time taken), and due to the 'manual' nature that the 27 macros were submitted in, it becomes more difficult to calculate and remove the 'dead time' in-between job submissions and *after completion.* It was impossible to know when each job was completed, and as such it is only possible to calculate the time taken to submit all jobs, and the approximate time to retrieve the results.

The time taken to submit the 27 macros was **52.9 hours** (3,174 minutes). This time is calculated by subtracting all of the 'dead time' in-between job submissions. Results retrieval was done with a script and took approximately 67.5 hours (150mins for each set of 10,000 results). This gives a total of 120.4 hours (just over 5 days) for the job submission and results retrieval stage of XtremWeb.

Unfortunately, the 27 macros were not submitted with a script to reduce the down time in between submissions, which would have been a more efficient use of time and available resources. The manual submission of the macros was suggested by the XtremWeb team and in hindsight, made the analysis of the platform more difficult.

The minimum DART job execution time was 32 seconds for a single job, with a maximum of 87 seconds. The average job execution time was 51.0 seconds. The variation in minimum and maximum times is simply down to a variation in CPU speeds of the available GRID5000 machines, or if there were any other processing that occurred on the worker systems while the DART experiments were running - the GRID5000 machines were generally of high specification, with processing times much faster than those presented in Chapter 7, on older desktop systems. Figure 7.44 displays the total time time taken to execute each set of experiments for each audio file. Table 7.27 displays the total processing time for each Audio Input File when running the XtremWeb experiments (which are roughly equal).

Figure 7.45 shows the number of jobs processed on each worker node on the GRID5000 platform. The minimum number is 1 job, with the maximum of 4,961 jobs. The average number of jobs processed by each of the 214 worker nodes is 1,207. Across the 214 machines, a total of 928 cores were available (59% of machines had 4 cores, with only 6% of the machines having only a single core) however DART is a single threaded application and XtremWeb offers no ability to run multiple jobs on multiple cores. Figure 7.46 shows the distribution of the number of CPU cores per worker for the 214 machines used on the GRID5000 platform.



	Acoustic Guitar	Oboe	Violin	Piano	Tubular Bells	Distorted Guitar
🟦 Max Time	72	87	77	76	82	73
🟥 Min Time	33	34	34	33	33	32
🟩 Avg Time	54	53	53	50	51	46

Figure 7.44: A graph to show the time taken to execute the DART algorithm on all DART Audio Input Files using the XtremWeb platform

Audio Input File	***Total Processing Time***
DARTAcousticG.wav	243,927,7 Seconds
DARTOboe.wav	241,124,2 Seconds
DARTViolin.wav	227,592,1 Seconds
DARTPiano.wav	232,167,1 Seconds
DARTTubBells.wav	229,031,0 Seconds
DARTDistortedG.wav	207,896,8 Seconds

Table 7.27: This table displays the total processing time for each Audio Input File when running the XtremWeb experiments

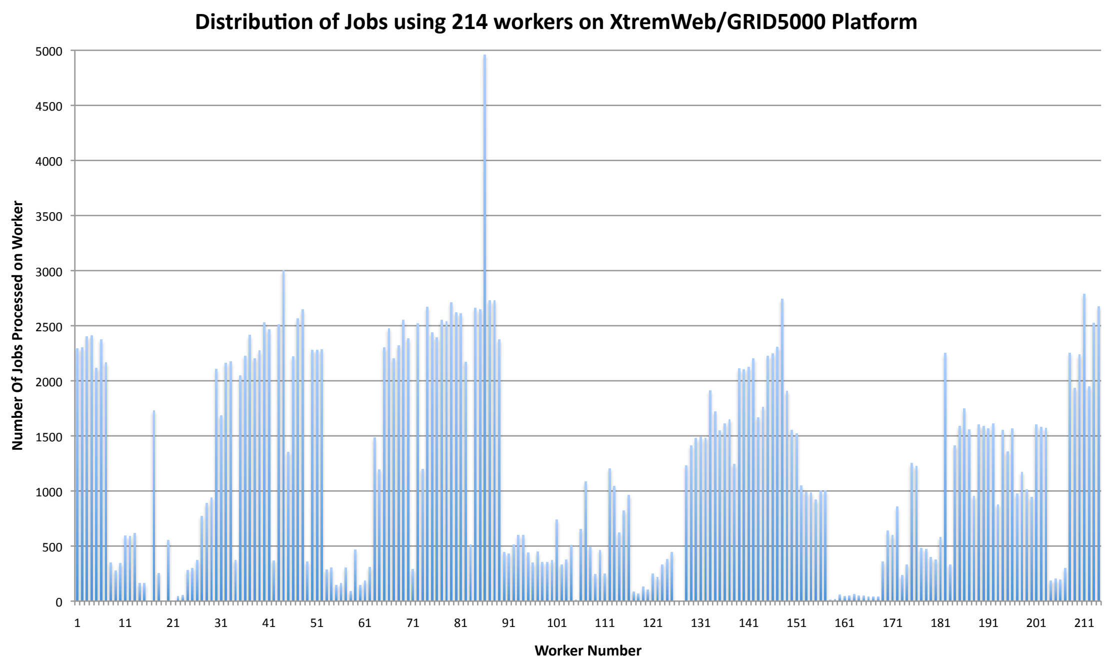


Figure 7.45: A graph to show the distribution of the number of jobs executed by each worker node on the XtremWeb platform

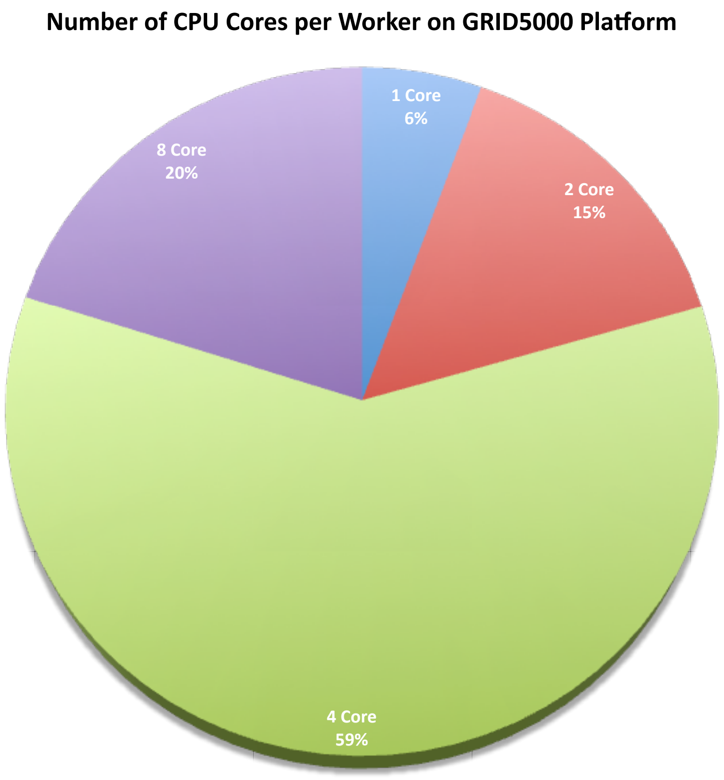


Figure 7.46: A graph to show the distribution of the number of CPU Cores on each worker node on the XtremWeb platform

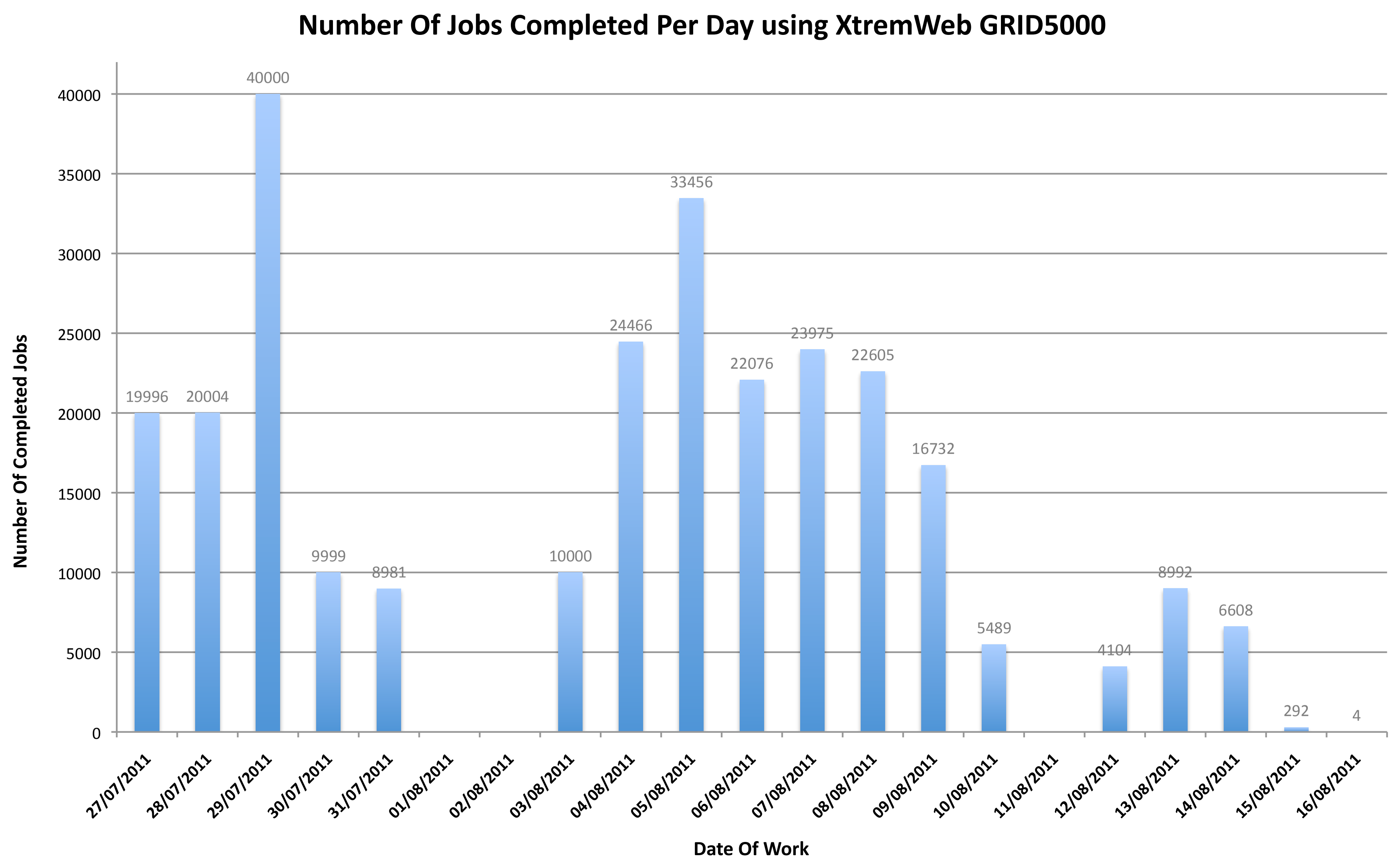


Figure 7.47: A graph to show the number of completed jobs on a day-to-day basis, throughout the 27 period that experiments were run on the GRID5000 platform

7.7.2 BOINC Time Analysis

The total processing time for analysing 268,800 results[6] over 47 active worker nodes on the BOINC platform was **169,848,33** seconds (*4,718 hours, or around 196.6 days*). These figures are calculated by adding together the total run time, in seconds, from each of the 268,800 results files retrieved. These figures do not include any of the BOINC job submission and results or data download times.

The first BOINC workunit was sent on 24 Jan 2012 at 18:18:50, and the last results were retrieved on 8 Feb 2012 at 17:19:04. It therefore took 14 days, 23 hours, 0 minutes and 14 seconds to send and complete 537,600 jobs (around 400 days of processing time) across 47 worker machines. Figure 7.50 shows the distribution of the number of work units processed on each host on the BOINC platform. Most machines were multicore machines and Figure 7.51 shows the distribution of the number of CPU Cores on each worker node on the BOINC platform. This is especially important on the BOINC platform as the BOINC Manager (client software) will detect the number of cores on the machine and begin to process the equivalent number of work units; a 4 core machine will be able to concurrently process 4 work units at the same time, with each work unit using up 100% of the resources on the respective core, in contrast to the XtremWeb platform.

Many modern processors such as i3, i5, i7 and some Xeon server class processors support Hyper-Threading[7]. Hyper-threading allows the operating system to address two virtual cores for every one physical core present, and shares the workload between them when possible. In the context of DART, this means that double the number of work units can be processed simultaneously rather than sequentially, resulting in up to a 50% speed increase[8] in comparison to similar machines which do not support Hyper-Threading.

Figure 7.53 shows the Total Credit Score for each host on the BOINC platform. The DART BOINC project gives credits to each host and user. These credits give a broad overview of the performance of the host or the contribution of a particular user (or team). Two types of credits are given:

[6]537,600 results were created in total, as each result was processed twice

[7]`http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`

[8]`http://www.ibm.com/developerworks/linux/library/l-htl/`

- **Total Credit Score**: The total number of Cobblestones[9] performed and granted.
- **Recent Average Credit Score**: The average number of Cobblestones per day granted *recently*. This average decreases by a factor of two every week.

A credit is given when a valid work unit is returned by a host/worker. If a computer processes and returns a work unit it does not automatically receive a credit - it must first have that work unit validated by the project specific method - in the case of DART, each work unit was processed twice and compared with a bitwise comparison. Once validated, the host is granted a credit. This amount is immediately added to the host or user total.

BOINC uses benchmarks to measure the speed of a system and in combination with the amount of time it required for a work unit to process can estimate at the amount of credit it should receive. Since systems have many variables including the amount of RAM, the processor speed, and specific architectures of different motherboards and CPUs, there can be wide discrepancies in the number of credits that different hosts gain when processing each work unit.

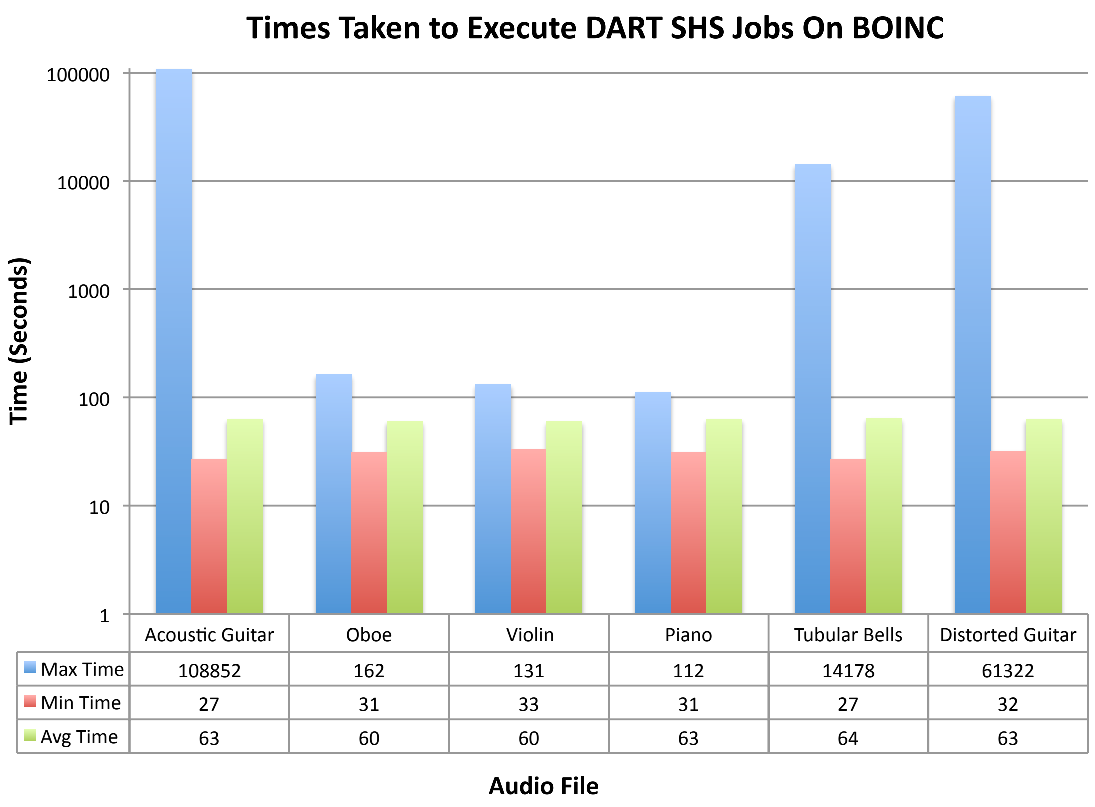
Figure 7.52 shows the Recent Average Credit score for each host on the BOINC platform. This calculation is designed to give a rough estimate of the number of credits a computer,, user, and team will accumulate on an average day. Additionally the RAC score is independent of computers, users, and teams, meaning that they do not simply accumulate. RAC was originally meant to help scientists understand the computational power available to them and to increase competition among users by allowing even new users to quickly move up in rank based on RAC, which should directly reflect how fast work is being processed. More information on the BOINC credit system can be found in [117] and [118].

The minimum DART execution time for a single *job* on the BOINC platform was **27** seconds, with the maximum of **108,852** seconds. This long execution time can be attributed to the transient nature of the availability of resources; workers can stop and start the DART BOINC Manager or their computers at will, and the job will not be complete until the worker resumes processing. There were only a handful of high maximum processing time, shown by the relatively low average job execution time of 63.19 seconds

[9]The basic unit of the BOINC credit system is the *cobblestone*, a benchmark figure named after Jeff Cobb of SETI@home

per job. The specification of the machines on BOINC is completely uncontrolled.

Figure 7.49 displays the total time time taken to execute each set of experiments for each audio file on the BOINC platform. Please note that the vertical axis (time) is displayed *logarithmically* in order to accommodate the high maximum processing time, in contrast to the linear axis in Figure 7.44.



	Acoustic Guitar	Oboe	Violin	Piano	Tubular Bells	Distorted Guitar
🟦 Max Time	108852	162	131	112	14178	61322
🟥 Min Time	27	31	33	31	27	32
🟩 Avg Time	63	60	60	63	64	63

Figure 7.48: A graph to show the time taken to execute the DART algorithm on all DART Audio Input Files on the BOINC platform.

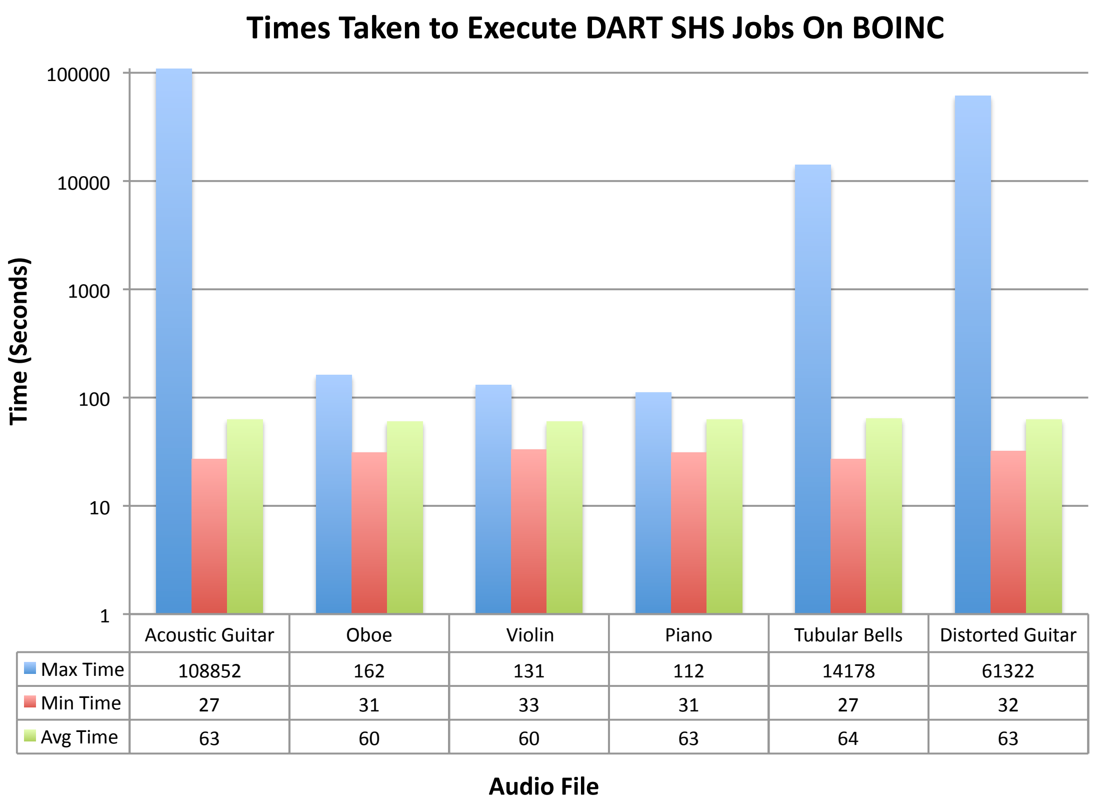
7.8 Summary & Discussion

Table 7.29 shows a comparison of results from the XtremWeb and BOINC projects.

As shown in Table 7.29, the BOINC platform calculated twice the number of experiments (537,600 vs 268,796) in less time (15 days vs 17 days), using roughly 1/3 of the machines (and 1/4 of the number of CPU cores). However, twice the number of experiments were run because BOINC results require validation as the hosts/worker machines were members of the 'public' and not granted trust automatically, therefore two jobs must

Audio Input File	***Total Processing Time***
DARTAcousticG.wav	311,172,8 Seconds
DARTOboe.wav	269,764,3 Seconds
DARTViolin.wav	270,409,6 Seconds
DARTPiano.wav	281,050,1 Seconds
DARTTubBells.wav	285,896,9 Seconds
DARTDistortedG.wav	280,189,6 Seconds

Table 7.28: This table details the total processing time for each audio input file when processed on the BOINC platform



	Acoustic Guitar	Oboe	Violin	Piano	Tubular Bells	Distorted Guitar
🟦 Max Time	108852	162	131	112	14178	61322
🟥 Min Time	27	31	33	31	27	32
🟩 Avg Time	63	60	60	63	64	63

Figure 7.49: A graph to show the time taken to execute the DART algorithm on all DART Audio Input Files on the BOINC platform.

be processed to validate a single job. The XtremWeb GRID5000 platform is secure and used trusted machines that did not require further validation. The SHS results from both platforms were identical.

The DART BOINC experiments used one of BOINC's built in validation methods called

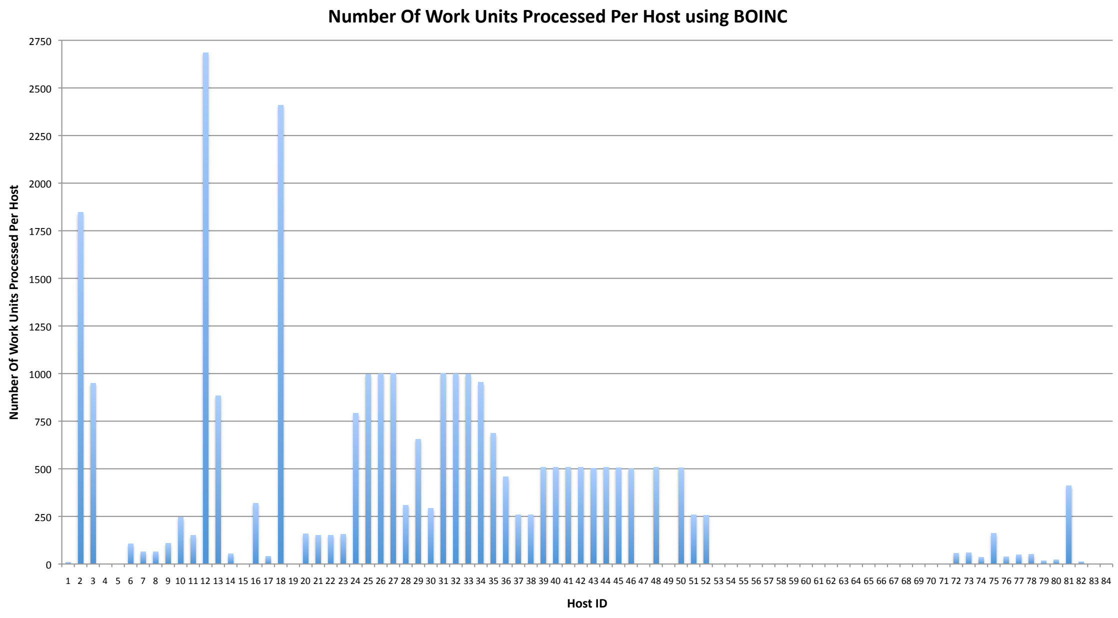


Figure 7.50: A graph to show the distribution of the number of work units processed on each host on the BOINC platform

	XtremWeb	*BOINC*
Total Number Of Jobs Processed	268,796	537,600
Total Processing Time (Hours)	3,838.17	4718
Total Running TIme (Hours)	408.5	359
Number Of Machines Used	214	68
Number Of Cores Available	916	223

Table 7.29: This table shows a summary of statistics of both large scale DART experiments, on the XtremWeb and BOINC platoform

the **`sample_bitwise_validator`**. This validator requires a strict majority, and regards results as equivalent only if they agree byte for byte, in which case the validator grants credit to the user/host for valid results. If the BOINC server was set up for 'desktop grid' computing (where all the participating hosts are trusted), then it would have been possible to use the **`sample_trivial_validator`**, which accepts all results as valid. This goes against the 'volunteer computing' paradigm, however.

BOINC provides a good level of security and protects against several types of attacks (for example, digital signatures based on public-key encryption protect against the distri-

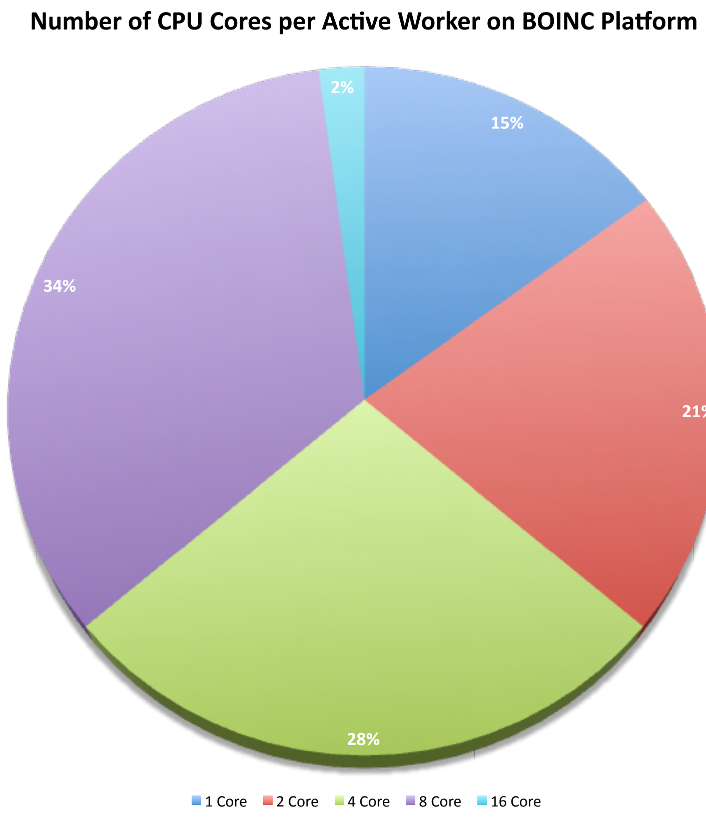


Figure 7.51: A graph to show the distribution of the number of CPU Cores on each worker node on the BOINC platform

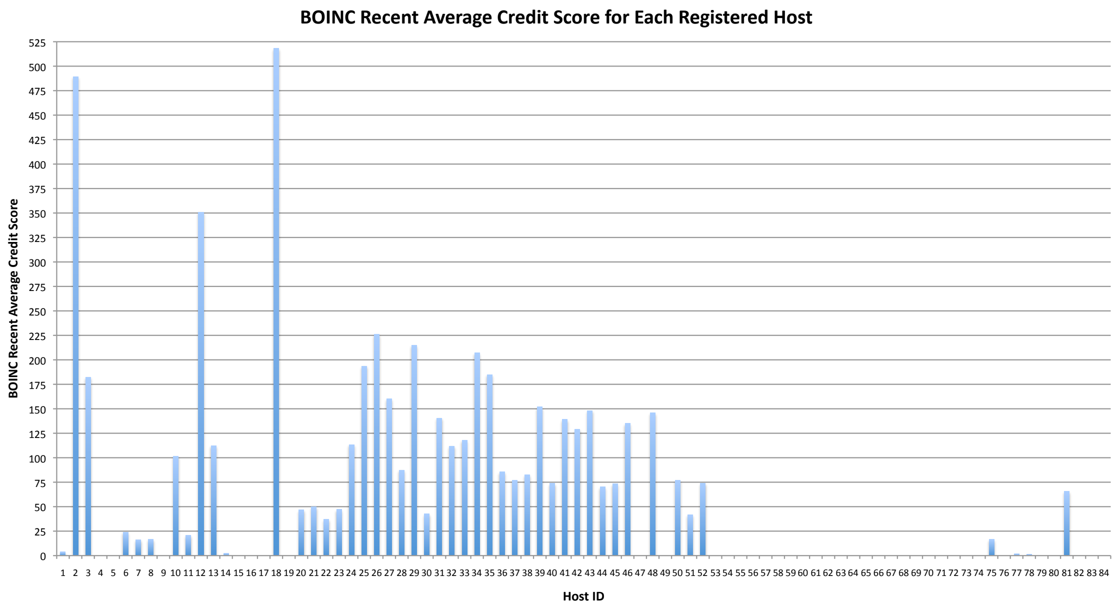


Figure 7.52: A graph to show the Recent Average Credit Score for each host on the BOINC platform

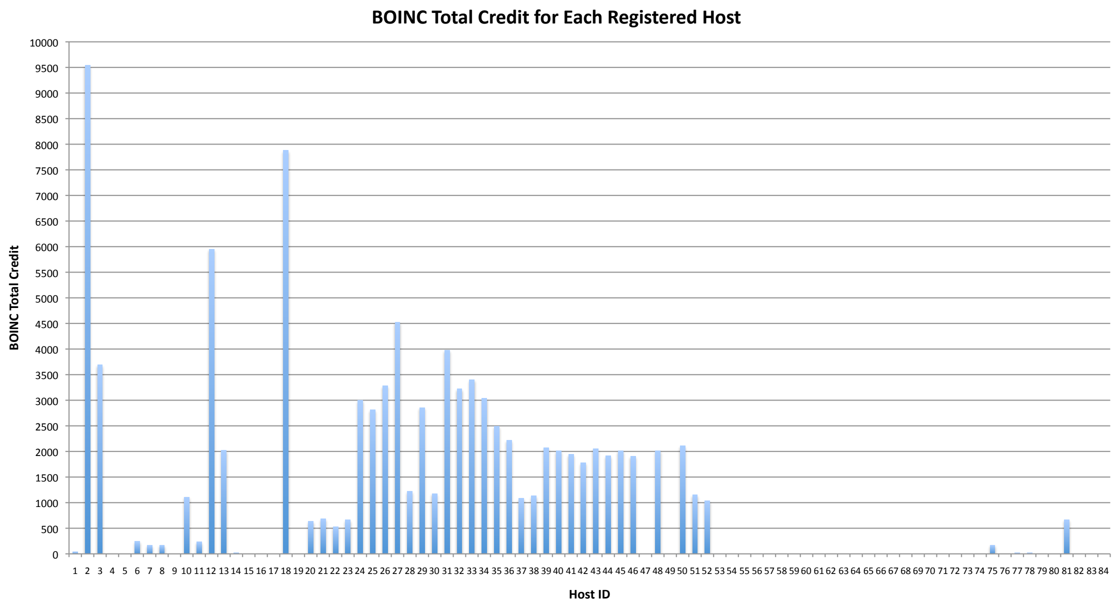


Figure 7.53: A graph to show the Total Credit Score for each host on the BOINC platform

bution of viruses). Overall, the BOINC experiments were faster (twice the number of work units overall in less time) due to number of factors; the automatic multicore processing capability of the BOINC platform allowed modern worker machines to take advantage of parallel local-execution of the work units, and BOINC also sent more work units to machines which had more processors and processing capability. For example, an 8-Core Intel Xeon based worker downloaded many times more work units to eventually process, in comparison to the single or dual core machines. The default amount of work downloaded by a client is also user-configurable.

The BOINC manager software automatically takes advantage of all the available cores on a machine by launching one work unit per available core, and can even take advantage of a hosts Graphics Processing Unit (GPU)[10] power if properly supported by the application (not supported in the DART experiments). Utilising the GPU can offer speed increases of 2x-10x over the CPU-only version[11]. BOINC makes excellent use of all the available resources on each machine.

These core-utilisation features are all missing from the XtremWeb middleware. XtremWeb does not 'get involved' in the CPU core distribution of the application running

[10]Most modern GPUs are capable are handling complex 3D and animation, and have increased in power, often making them more powerful than the computer's main processor, or CPU.

[11]`http://boinc.berkeley.edu/gpu.php`

and focusses purely on the delivery and distribution of the application itself. This is not necessarily a missing or negative feature, merely a different way of working; XtremWeb could lend itself well to heavily multi-threaded DART applications in the future, especially given that the vast majority of machines (59%) on GRID5000 platform had 4 available cores. BOINC therefore seems ideal for single threaded applications, in particular parameter sweeps, while XtremWeb could prove to be more suited to heavily multi-threaded DART applications.

The current DART SHS algorithm is a single threaded application and took on average 51 seconds (XtremWeb) to 63.9 seconds (BOINC)[12]. This indicates that even the speed per core on each XtremWeb machine was on average faster, but the lack of multicore awareness resulted in poor efficiency overall.

However, the BOINC platform required expert assistance in order to set up the BOINC server, the workunit generation script, in converting the application to work on multiple platforms, and also solving numerous bugs and re-starting the project after several failed attempts. In general, porting the DART application to be compatible with BOINC was much more labour intensive for the DART user/scientist. Several different methods and applications (code wrappers) were used to try to convert the DART application to work natively in on OSX, Windows and Linux platforms (with each OS coming in various iterations, as well as 32/64-bit support for each iteration that supports it). Mac OSX and Windows XP (the former an increasingly popular OS for both consumers and audio enthusiasts - both key target audiences for DART - the latter being one of the most popular operating systems, even after 10 years[13]) support was not available on BOINC due to ongoing issues when using the GenWrapper program, and re-coding the SHS algorithm natively in C would go against the DART paradigm and be extremely time consuming. With Mac OSX and Windows XP support, more workers would have been available.

XtremWeb has the distinct advantage of being Java based middleware that works well with the DART and Triana platform; by definition this means that the worker machines will have Java 1.6 installed. XtremWeb can actually run any type of application without requiring modification, providing that the dependancies of the application are catered for.

[12]When the DART SHS algorithm was created, average running time was approximately 2-3x longer, roughly in line with Moores Law, as shown in the previous chapter

[13]`http://www.w3schools.com/browsers/browsers_os.asp`

While a personal XtremWeb desktop grid system can be created by setting up an XtremWeb server, another advantage of XtremWeb is ready and available platforms such as those at LAL (Laboratoire de lAcclrateur Linaire, Paris), at LRI (Laboratoire de Recherche en Informatique, Paris) and GRID5000 (a distributed site in France) that allow - given suitable permission - users to simply 'plug in' their applications and use the available resources, as documented in the implementation chapter.

However, the continuous problems encountered with using the XtremWeb platform must be highlighted and cannot be ignored. The platform was buggy and is clearly a 'work in progress', taking many months to iron out the encountered issues. The EGEE bridge did not work when attempting to run any of the major DART experiments, despite numerous efforts and the consultation of several experts and developers throughout Europe. Several delays and bugs pushed the experiments back and many attempts were made before a successful run was possible - and even after this, proper statistics were unavailable to find the makespan of the project. Issues with bugs and memory leaks plagued the project throughout all experiments attempted on the platform.

XtremWeb is less established in comparison to BOINC and the DART experiments were the largest set of experiments ever run using the XtremWeb platform; there were still scalability problems that had not yet been encountered by the XtremWeb developers. Figure 7.54 shows a graph taken the XtremWeb website showing the number of jobs processed across the year from February 2011-2012. The large spike in March and smaller spike in April reflects some of the first attempts in running the large DART SHS parameter sweep experiments, stopped due to XtremWeb client memory leaks and lost results. No jobs used the EGEE bridge (for unknown reasons) and the XtremWeb staff noted a probable memory leak in the XtremWeb-HEP client because it crashed after retrieving 150K results. The results could not be retrieved.

However, some interesting information can still be gathered from running the failed experiments. This particular run lasted for 20 days and each of the 268,800 jobs were submitted from one client using 270 macro files, each containing 1000 jobs. The job submission period lasted for approximately 36 hours. A high (10%) processing error rate was noted due to a bug in XtremWeb-HEP (one suggestion being possible concurrent file access on workers which ran several jobs simultaneously). The jobs with the errors were manually resubmitted and all the jobs were reported 'completed'. Approximately 50 jobs had a constants status of RUNNING; these were resubmitted, but prolonged the overall duration. 27,000 jobs reported errors and were submitted, with a further 3200 error jobs from the resubmitted jobs. Further errors were reported and jobs were resubmitted until no errors were received.

The job average execution time is very short (approximately 2 minutes), and the average XtremWeb overhead (job completed date minus the last time the job started) of all jobs is 154 seconds. This gives an idea of the average job execution time, although the communication time and manual resubmission of error and always running jobs change this average. Most hosts had 2 cores, however there were some with between 1 and 16 cores.

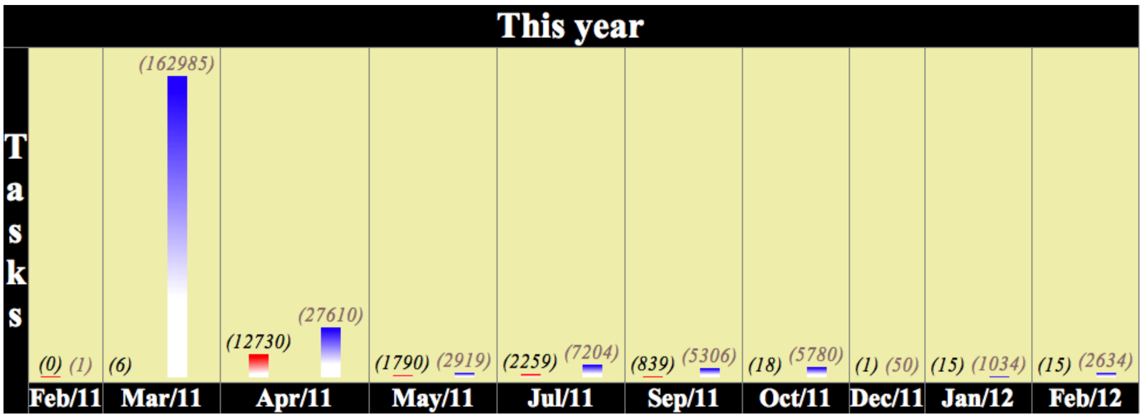


Figure 7.54: A graph from the XtremWeb website showing the number of jobs processed across the year from February 2011-2012

Although these experiments were ultimately unsuccessful, some positive research and insight came from this attempt to run the SHS parameter sweep. This attempt highlighted unseen bugs in the XtremWeb software, and the delays and setbacks also allowed for a refinement of the DART input audio data, making the eventual parameter sweep experiment much more robust.

A somewhat strange occurrence was that it was possible to retrieve statistics on the unsuccessful/unretrievable large scale experiments (such as the attempt in March/April 2011), however after this run, a bug was introduced making the statistics of the successful large scale experiments less accurate, as documented in this chapter.

EGEE support was also working in earlier tests (such as those presented in Chapter 7) and also failed when it 'really mattered' - the successful, large scale SHS experiments with

268,800 jobs. As earlier experiments (documented in Chapter 6) were conducted using the EGEE bridge it was believed that the main run of 268,000 jobs could be submitted over the XtremWeb to EGEE bridge. However this time the jobs were submitted from Cardiff University (as opposed to LAL) and run on the 30-60 available local machines at LAL. This overloaded the Desktop Grid and only 101,000 jobs were completed before the experiments halted. Again, submitting such a large number of jobs over XtremWeb also highlighted some deficiencies with the platform to the developers.

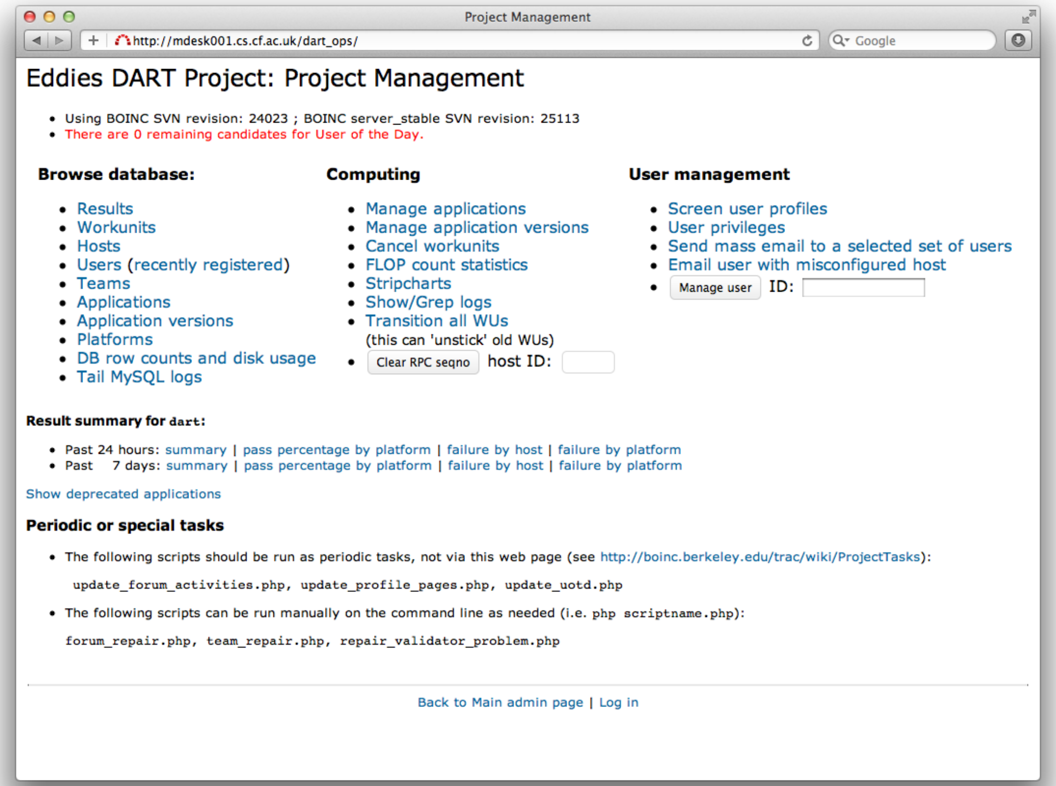
Creating a BOINC project also led to some issues, with errors in the work unit allocation script causing the experiments to halt. This was a simple scripting problem caused by human error - some of the FFT Window names contain apostrophes which were not 'escaped' properly in the script.

BOINC has a clear advantage in that it is an established platform BOINC with a clear methodology and documentation. The BOINC platform is mature and quite refined in comparison to the much newer XtremWeb platform. BOINC also a has a fully featured 'admin' section as shown in Figure 7.55 [14], displaying statistics on the number of completed jobs, errors, users, hosts; nearly all aspects required for project management and maintenance. This is a distinct advantage over the XtremWeb platform, which uses terminal commands to operate and report on the status of jobs. Each job in an XtremWeb project is seen individually and not part of a particular 'project' or run of experiments, making status checks on large numbers of experiments impractical. The XtremWeb GUI application was full of bugs and unusable. The BOINC admin page made it extremely clear when a worker or host machine had signed up to the BOINC project with an incompatible machine and had to be notified. This made the administration of the project much simpler.

BOINC provides features that simplify the creation and operation of distributed computing projects. As BOINC is a volunteer computing platform, it also provides web-based interfaces for account creation, preference editing, and participant status display. A participant's preferences are automatically propagated to all their hosts, making it easy to manage large numbers of hosts.

The *ATTIC* distributed data distribution system did not work on BOINC or XtremWeb

[14]The database for the DART BOINC project was accessible at `http://mdesk001.cs.cf.ac.uk/dart_ops/`



Eddies DART Project: Project Management

- Using BOINC SVN revision: 24023 ; BOINC server_stable SVN revision: 25113
- There are 0 remaining candidates for User of the Day.

Browse database:

- Results
- Workunits
- Hosts
- Users (recently registered)
- Teams
- Applications
- Application versions
- Platforms
- DB row counts and disk usage
- Tail MySQL logs

Computing

- Manage applications
- Manage application versions
- Cancel workunits
- FLOP count statistics
- Stripcharts
- Show/Grep logs
- Transition all WUs
  (this can 'unstick' old WUs)
- Clear RPC seqno host ID:

User management

- Screen user profiles
- User privileges
- Send mass email to a selected set of users
- Email user with misconfigured host
- Manage user ID:

Result summary for dart:

- Past 24 hours: summary | pass percentage by platform | failure by host | failure by platform
- Past 7 days: summary | pass percentage by platform | failure by host | failure by platform

Show deprecated applications

Periodic or special tasks

- The following scripts should be run as periodic tasks, not via this web page (see http://boinc.berkeley.edu/trac/wiki/ProjectTasks):

  `update_forum_activities.php, update_profile_pages.php, update_uotd.php`

- The following scripts can be run manually on the command line as needed (i.e. `php scriptname.php`):

  `forum_repair.php, team_repair.php, repair_validator_problem.php`

Back to Main admin page | Log in

Figure 7.55: The DART BOINC Project Management webpage

as the developer did not finish the implementation and have a working system within the timeframe of the project, however in the case of BOINC at least, the data distribution did not appear to be a large bottleneck in terms of overall performance and time taken to completion.

BOINC's 300Mb *work-package* containing the application and the 6 audio input files was only downloaded once by each host machine, and work units and results files (both only 1-2KB in size) were passed back and forth to the BOINC server. Many work units are downloaded to each worker machine and the work units are extremely small in size (no new audio data is sent with the work units). Therefore after the initial download, each system has relatively low network activity, with just a few kilobytes of data being transferred back and forth between each worker and the BOINC server.

While XtremWeb sees each job as an individual experiment, it also employs a caching scheme and therefore does not download the DART application and a 55mb input data (audio file) each time a new job is to be run on a the worker. This also helps reduce overall network traffic as it is not necessarily true that all workers will process a job with each audio file. This caching scheme was added to the platform as a suggestion based on results of previous DART experiments.

In comparison to BOINC, XtremWeb has some key differences in its methodology that would be useful for a DART user to understand before picking one of the two platforms to run their MIR/Audio Analysis experiments. For example, XtremWeb assumes that each user or node has the ability (if authorised) to submit a job. With BOINC, only the project manager is able to create tasks and insert new applications. As a consequence, fewer applications are supported by BOINC and require more time preparing the project and server. In contrast, XtremWeb offers scientific users an interface similar to batch system to create their jobs and get their results, but does not offer credits base system, nor general user support forum.

It is also important to take into account that the total running time of the XtremWeb project would have been massively reduced if the process had been automated, as usually the case when XtremWeb. Because of the use of the GRID5000 resources, the jobs were submitted manually using 27 macro files, each containing 10,000 job requests with the understanding that the time in-between macro submissions would be available to subtract from the total running project time. Time was unnecessarily wasted in-between submitting the macros, and due to problems with the XtremWeb middleware it was impossible to discover the total running time of the project.

However, these kind of issues are real-world, realistic problems that a DART MIR scientist would face, and as such must be taken into account.

Chapter 8

Further Investigation Results

8.1 Chapter Overview

This chapter gives an overview of the further experiments carried out on using the Pegasus platform as a result of findings in the previous chapter. These experiments extend the range of the parameters of the NTFP value in the DART SHS algorithm to 501 (from 1-50 of the previous experiments) to investigate any change to the accuracy level.

The chapter begins by describing the Pegasus platform, and then gives a description of the further parameter sweep experiments conducted using Pegasus. Modifications to DART are then outlined, before presenting the results of the Pegasus NTFP experiments.

The accuracy of the *Number of Top Frequency Points* is then given both overall and for each of the six audio input files.

8.2 Further Experimentation

Previous DART parameter sweep experiments highlighted the optimal FFT Window Type and Number of Harmonics parameters, both for each instrument analysed and more generally across all six audio input files. However, the Number Of Top Frequency Points (NTFP) value range of 1-50 did not produce conclusive results; some of the audio input files (such as the Distorted Guitar and Violin) exhibited increasing accuracy as the NTFP value rose towards the maximum NTFP value of 50.

As a result, the aim of further experiments on the Pegasus platform is to extend the range of the NTFP values and note any effect on the accuracy levels of the SHS algorithm. Extending this figure to around 500 would reveal if the accuracy continued to increase, plateaued, or even decreased.

An opportunity arose to conduct further DART experiments on more distributed computing platforms through the collaboration of the Triana development team and *Pegasus* developers, through joint participation in the SHIWA project[1]. SHIWA is a project that aims to develop workflow interoperability technologies.

However, the aim of these experiments is not to gauge the suitability of the Pegasus or related platforms for DART, but solely to focus on the results of further increasing the NTFP value and noting the effect on the accuracy of the DART pitch detection algorithm. Integration with a new distribution platform shows DART's versatility, however.

8.2.1 Pegasus

Pegasus[2] is a *Workflow Management System* (WMS) created at the Information Sciences Institute (ISI), University of Southern California for mapping and executing application workflows over a wide range of distributed resources, ranging from a single laptop to a campus cluster, a Grid, or a cloud-based system. At the time of writing, Pegasus can run workflows on systems such as the Amazon Elastic Cloud2[3], Nimbus[4], Open Science Grid[5], TeraGrid[6], and many University campus clusters. One workflow can run on a single system or across a wide range of resources. Pegasus scales well - both the size of the workflow and the resources that the workflow is distributed over - and can run workflows ranging from just a few computational tasks, to millions of jobs. The number of resources involved in executing a workflow can scale as needed without any impediments to performance.

Pegasus contains a host of features that are required to run DART, such as the ability to automatically locate the necessary input data and computational resources necessary for workflow execution. This enables scientists to construct workflows in abstract terms with-

[1]`http://www.shiwa-workflow.eu`
[2]`http://pegasus.isi.edu`
[3]`http://aws.amazon.com/ec2/`
[4]`http://www.nimbusproject.org`
[5]`http://www.opensciencegrip.org`
[6]`http://www.teragrid.org`

out worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware. Pegasus takes in an abstract workflow (called a **DAX**) and generates an executable workflow (Directed Acyclic Graph - DAG) that is run in an environment. A DAX is a description of an abstract workflow in XML format that is used as the primary input into Pegasus.

Pegasus therefore allows researchers to translate complex tasks into workflows that link and manage dependent tasks and related data files, automatically chaining dependent tasks together so that a single user can complete complex computations that once required many different stages or people.

Pegasus has a number of further features that contribute to its usability and effectiveness, such as the ability to easily run the user created workflows (DAXs) in different environments without alteration. The same workflow can be run on a single system or across a heterogeneous set of resources. Furthermore, the Pegasus *Mapper* can reorder, group, and prioritise tasks in order to increase the overall workflow performance, finding the appropriate software, data, and computational resources required for workflow execution.

The system is composed of three components:

- **Pegasus Mapper**: Generates an executable workflow based on an abstract workflow provided by the user or workflow composition system
- **Execution Engine**: Executes the tasks defined by the workflow in order of their dependencies
- **Scheduler/Task Manager**: Manages individual workflow tasks: supervises their execution on local and remote resources

A DAX can be deployed across a variety of environments:

- Local Execution - a workflow on a single computer with Internet access. Running in a local environment is quicker to deploy as the user does not need to gain access to multiple resources in order to execute a workflow
- Condor[7] Pools & Glideins - Condor is a specialised workload management system. Condor Pools and Glideins are tools for submitting and executing the Condor daemons on a Globus resource

[7]`http://www.cs.wisc.edu/condor/description.html`

- Grids
- Clouds

The experiments outlined in this chapter were carried out at ISI, University of South California using Pegasus with Condor[8]. Condor is a workload management system for compute-intensive jobs. Condor provides:

> *"a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion"*

[122]

Condor is often used to manage a cluster of dedicated compute nodes, as is the case with Pegasus, and in addition, unique mechanisms enable Condor to effectively harness wasted CPU power from otherwise idle desktop workstations in a similar way to BOINC. For instance, Condor can be configured to only use desktop machines where the keyboard and mouse are idle.

8.3 Pegasus Experiments Description

This smaller scale parameter sweep experiment on the Pegasus platform consists of 1,632 jobs per audio input file, giving a total of **9,792 jobs**. The FFT Window will remain a constant, set to **Rectangle**. The Rectangle (or lack of) window was found to give the best results overall in the large scale parameter sweep application and as such was chosen as a constant in order to reduce the number of overall jobs and focus more intensely on the effect of modifying the NTFP value. The Pegasus parameter sweep experiment will vary the following parameters:

- **Top Frequency Points**: Vary 1 to 501 in 10 point intervals (51 in total)

[8]`http://research.cs.wisc.edu/condor/`

- **Number of Harmonics**: Vary 1-32 (5 Octaves - the same range used in previous experiments)
- **Audio Input Files**: 6 audio files (the same files as used in previous experiments)

A bundle was supplied to the Pegasus team featuring the DART JAR and other relevant files:

1. **`Dart.jar`** - Main DART application
2. **6 x DART Audio files** - These are the input required for the jobs. Each job only requires one input file.
3. **6 x 'DART Commands' text files** - These contain the 1,632 job execution commands (**`java -jar Dart.jar`** etc) for the particular audio file.
4. **`dart-script.pl`** - running this script in the terminal will generate the complete range 'DART Commands' arguments, specific to the extra Pegasus SHS parameter sweep experiments. This script is useful in aiding the generation of a Pegasus DAX.

The **`dart-script.pl`** file forms the basis of the Pegasus workflow. The DART Command text files contain the relevant commands for each audio input file, at the request of the Pegasus team:

- 'DART Commands 1.txt' contains all the commands for the DARTAcousticG.wav input file
- 'DART Commands 2.txt' contains all the commands for the DARTOboe.wav input file
- 'DART Commands 3.txt' contains all the commands for the DARTViolin.wav input file
- 'DART Commands 4.txt' contains all the commands for the DARTPiano.wav input file
- 'DART Commands 5.txt' contains all the commands for the DARTTubBells.wav input file
- 'DART Commands 6.txt' contains all the commands for the DARTDistortG.wav input file

8.3.1 DART Modification

The Dart Execution Environment and resulting JAR required minor modifications in order to maintain compatibility when integrated with Pegasus. The DART CLI was modified to allow the specification of an output results directory, a requirement of pegasus. The output results directory can now be specified at the command line by adding **-outputdir yourdirectory/subdirectory** in the command line path. This would make a complete DART command line argument such as:

```
java -jar Dart.jar -infile DARTDistortG.wav -outfile DART-6-501-32-1.txt -
    nofreqpoints 501 -noharmonics 32 -fft_window Rectangle -resultsdir
    testdir/subdir2/subdir3
```

The default directory is the directory that the JAR is being run from, but it is also possible to go up in subdirectory in the standard way for a Unix based command line: **-resultsdir ../someresults**. If no output results directory is specified at the command line, DART will simply create a **/results/** folder as usual.

There are two main ways of generating a Pegasus workflow:

1. Using a DAX generating API in Java, Perl or Python (Recommended by developers)
2. Generating XML directly from your script. (Only for advance users).

It is important to note that the idea of a Pegasus workflow (DAX) is wholly separate from the Triana workflow (or taskgraph) created and encapsulated in the DART Execution Environment. The Pegasus workflow is shown in Figure 8.1 and represents a recreation of the **dart.pl** Perl script documented in sections and which generate the multiple command line arguments required to conduct the parameter sweep experiments. The perl script was modified, creating a new script (called **dart-dax-gen.pl**) that generates the DAX (Pegasus input description). A sub-workflow is created for each audio file in the input directory. The audio input data and the DART JAR will be moved by Pegasus for each of the jobs.

The job commands are split into 6 sets of 1,632 jobs with each sets of commands in the relevant text file, as requested by the Pegasus team. For each of the six audio input files a sub-workflow was created. Each of these sub-workflows contained the 1,632 jobs, following the same logic as in the original DART perl script.

8.3.2 Results Analysis Program Modification

The results generated by the Pegasus experiments were analysed using the same programs detailed in Chapter 5. As with the initial integration with Pegasus, slight modifications

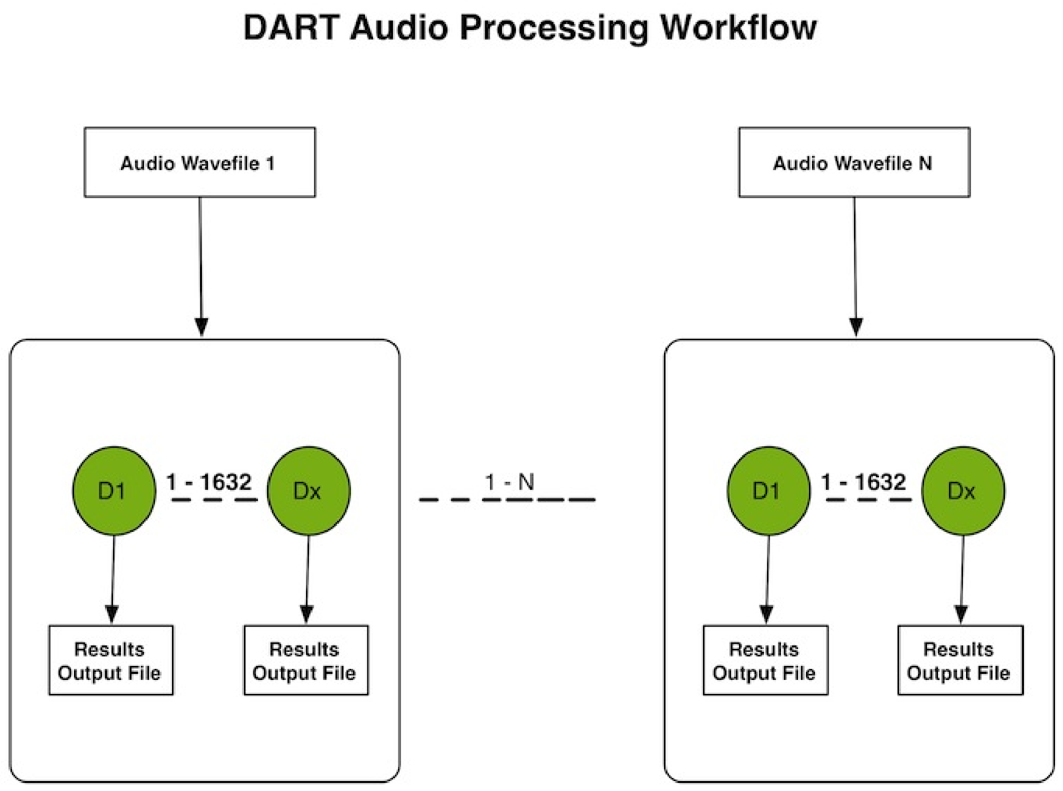


Figure 8.1: A diagram showing Pegasus DAX workflow for 1-n jobs (where n is 6), which creates a sub workflow with 1,632 jobs per audio input file

were required to enable the post-experiment analysis to take place. Modifications were made to the initial perl script given, whereby the `@infilename` command was picked up as the following:

```
print "Input Directory for wave files is : $input_dir\n";
my @infilename = `ls $input_dir/*wav`;
```

This enabled the script to automatically resize the number of computations (jobs) depending on the audio input files. For example, this would allow for the expansion of different audio input files (each with a different recorded or sampled instrument) to be placed in the input directory and the experiments would work without further modification.

While this is definitely an improvement, this did however mean that the naming scheme of the files (Acoustic Guitar is audio file 1, Oboe is 2, and so on) was modified. This lead to the modification of the DART post analysis program; as well as a change to the algorithm to take into account the extra NTFP values.

8.4 Pegasus NTFP Accuracy Results

This section displays the accuracy levels of the varying the NTFP value from 1-501, in steps of 10. Each subsection looks at the results *per audio file*, with the result across all audio files displayed in the final sub-section..

Each results set consists of a graph displaying the data, as well as a table containing the exact values in the graph, and more information based on the Minimum, Maximum and Average Octave Error values. The Top Value + Octave Error value is also given.

When the SHS algorithm is searching for the fundamental frequency or pitch of the audio, it considers the top $\boldsymbol{n}$ frequency amplitude values present in the spectrum. The aim of these tests was to find the optimum value of $\boldsymbol{n}$, resulting in the highest accuracy results.

It is important to note that there can be small discrepancies between the minimum and maximum values found in these experiments (compared with the results of the previous experiments) due to the use of only the Rectangle type of FFT Window.

8.4.1 Number Of Frequency Points Results for Acoustic Guitar

Graph 8.2 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501, using the DART Acoustic Guitar audio input file. The yellow line indicates the optimum NTFP - i.e. the minimum number of points that give the highest (or equal to the highest) accuracy for all accuracy measurements, reducing unnecessary processing. Table 8.1 shows the values presented in Graph 8.2.

The overall accuracy of the acoustic guitar samples was high at lower NTFP values, especially when ignoring octave errors. Graph 8.2 shows that while the greatest accuracy is discovered when analysing **21** frequency points, that looking at more than **11** frequency points gives only a minor improvement in maximum or average (less than 1%). There is an average of only 1.5% pitch errors (not explainable with octave mistakes) when analysing 21 harmonics and including octave errors, with a maximum accuracy of 97.87% - extremely high in comparison to the other audio files.

The NTFP mark of **21** is interestingly in-between the two values marked in the previous large scale parameter sweep experiments (where the NTFP value was varied from 1-50) as displayed in the previous chapter. The previous test noted that the Optimum NTFP value was **33**, with only minor increases in accuracy above the NTFP value of **11**. The figure of 21 found in the set of experiments run on the pegasus platform is roughly in-between these two values, providing extra evidence that the optimum NTFP value for this audio input file is in this range.

The previous experiments varying the NTFP from 1-50 appeared to show a plateauing in all of the accuracy measurements displayed in the graphs and tables. However Graph 8.2 shows a clear *decrease* in overall accuracy as the NTFP value rises above 31.

After a small decrease the *Maximum Accuracy Including Oct Errors* does plateau, however. This is evidence to suggest that given a high enough number of harmonics, the accuracy could remain high when the NTFP value is increased. The steep decrease in Minimum accuracy levels at 161 with the level of Max Inc Oct Errors remaining unchanged also supports this. Performing the experiments again but only varying the NTFP value would be able to a reliable way to confirm this.

The steep decrease in Minimum accuracy levels at a NTFP value of 161 could represent the point at which the algorithm simply begins to analyse *noise* instead of relevant harmonic data.

It should also be noted that while the use of the Rectangle window was optimal overall, the large scale parameter sweep experiments performed in the previous chapter revealed that the Acoustic Guitar audio file achieved higher accuracy rates using the Welch FFT Window Type, with 14 harmonics.

A future run tailored to the optimisation of the specific parameter being analysed (NTFP) would also be beneficial; running experiments sweeping from 1-501 while keeping the FFT Window to Welch and using 14 Harmonics for the Acoustic Guitar audio input file would make for interesting further analysis.

Given the current evidence from both experiments however, it seems that the best results for the Acoustic Guitar audio input file can be found using a low NTFP of around 21, the Welch FFT Window type, and 14 harmonics.

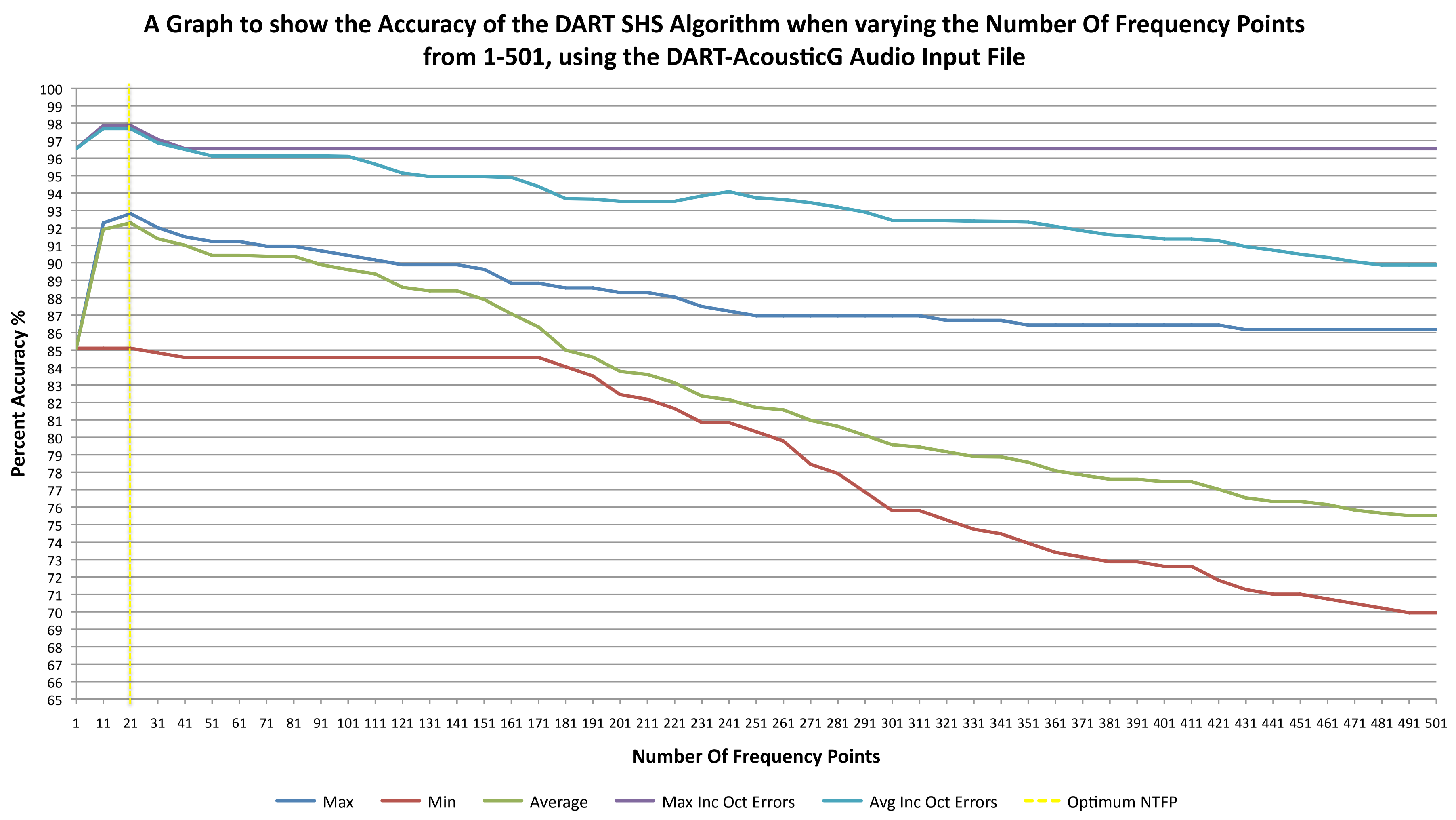


Figure 8.2: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-501, with minimum, maximum and average score values using the DART Acoustic Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
1	85.11	85.11	85.11	96.54	96.54	96.54	11.44	11.44	11.44
11	92.29	85.11	91.92	97.87	97.70	97.87	11.44	5.59	5.78
21	**92.82**	**85.11**	**92.29**	**97.87**	**97.70**	**97.87**	**11.44**	**5.05**	**5.41**
31	92.02	84.84	91.38	97.07	96.87	97.07	11.44	5.05	5.49
41	91.49	84.57	91.01	96.54	96.50	96.54	11.44	5.05	5.49
51	91.22	84.57	90.43	96.54	96.13	96.28	11.44	5.05	5.70
61	91.22	84.57	90.43	96.54	96.13	96.28	11.44	5.05	5.70
71	90.96	84.57	90.38	96.54	96.13	96.54	11.44	5.32	5.75
81	90.96	84.57	90.38	96.54	96.13	96.54	11.44	5.32	5.75
91	90.69	84.57	89.89	96.54	96.13	96.28	11.44	5.59	6.23
101	90.43	84.57	89.61	96.54	96.10	96.28	11.44	5.85	6.49
111	90.16	84.57	89.36	96.54	95.65	96.54	11.44	5.85	6.29
121	89.89	84.57	88.60	96.54	95.15	96.54	11.44	6.12	6.55
131	89.89	84.57	88.40	96.54	94.95	96.54	11.44	6.12	6.55
141	89.89	84.57	88.40	96.54	94.95	96.54	11.44	6.12	6.55
151	89.63	84.57	87.90	96.54	94.95	96.54	11.44	6.12	7.05
161	88.83	84.57	87.08	96.54	94.90	96.54	11.44	6.91	7.82
171	88.83	84.57	86.33	96.54	94.37	96.54	11.44	6.91	8.05
181	88.56	84.04	85.00	96.54	93.68	96.54	11.44	7.18	8.68
191	88.56	83.51	84.59	96.54	93.65	96.54	11.44	7.45	9.06
201	88.30	82.45	83.78	96.54	93.53	96.54	11.44	7.71	9.75
211	88.30	82.18	83.60	96.54	93.53	96.54	11.44	7.71	9.92
221	88.03	81.65	83.13	96.54	93.53	96.54	11.44	7.98	10.40
231	87.50	80.85	82.36	96.54	93.83	96.54	12.50	8.51	11.47
241	87.23	80.85	82.16	96.54	94.08	96.54	12.77	8.78	11.93
251	86.97	80.32	81.72	96.54	93.73	96.54	12.77	9.04	12.01
261	86.97	79.79	81.57	96.54	93.63	96.54	13.03	9.04	12.05
271	86.97	78.46	80.98	96.54	93.44	96.54	13.56	9.04	12.47
281	86.97	77.93	80.63	96.54	93.19	96.54	13.83	9.04	12.56

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
291	86.97	76.86	80.11	96.54	92.91	96.54	14.10	9.04	12.80
301	86.97	75.80	79.58	96.54	92.44	96.54	14.63	9.04	12.86
311	86.97	75.80	79.45	96.54	92.44	96.54	14.63	9.04	12.99
321	86.70	75.27	79.17	96.54	92.42	96.54	14.89	9.31	13.25
331	86.70	74.73	78.90	96.54	92.39	96.54	15.16	9.31	13.49
341	86.70	74.47	78.88	96.54	92.37	96.54	15.16	9.31	13.49
351	86.44	73.94	78.57	96.54	92.34	96.54	15.43	9.57	13.76
361	86.44	73.40	78.08	96.54	92.09	96.54	15.69	9.57	14.00
371	86.44	73.14	77.83	96.54	91.84	96.54	15.69	9.57	14.00
381	86.44	72.87	77.60	96.54	91.61	96.54	15.69	9.57	14.00
391	86.44	72.87	77.60	96.54	91.51	96.54	15.69	9.57	13.90
401	86.44	72.61	77.46	96.54	91.36	96.54	15.69	9.57	13.90
411	86.44	72.61	77.46	96.54	91.36	96.54	15.69	9.57	13.90
421	86.44	71.81	77.02	96.54	91.26	96.54	16.22	9.57	14.25
431	86.17	71.28	76.53	96.54	90.93	96.54	16.22	9.84	14.40
441	86.17	71.01	76.33	96.54	90.73	96.54	16.22	9.84	14.40
451	86.17	71.01	76.33	96.54	90.49	96.54	15.96	9.57	14.16
461	86.17	70.74	76.15	96.54	90.31	96.54	15.96	9.57	14.16
471	86.17	70.48	75.83	96.54	90.06	96.54	15.96	9.57	14.23
481	86.17	70.21	75.65	96.54	89.88	96.54	15.96	9.57	14.23
491	86.17	69.95	75.52	96.54	89.88	96.54	16.22	9.57	14.36
501	86.17	69.95	75.52	96.54	89.88	96.54	16.22	9.57	14.36

Table 8.1: This table details the accuracy of the DART SHS algorithm while varying the NTFP from 1-501 in multiples of 10, using the DART Acoustic Guitar audio file. The row of the optimal NTFP value is highlighted bold. 'O.E' stands for *Octave Error*.

8.4.2 Number Of Frequency Points Results for Oboe

Graph 8.3 shows the relative accuracy of the SHS algorithm when modifying the Number of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501 in increments of 10, using the DART Oboe audio input file. Table 8.2 shows the values presented in Graph 8.3.

Graph 8.3 shows that while the greatest accuracy is discovered when analysing the (maximum) **341** frequency points, looking at more than approximately **21** frequency points gives only a minor improvement in accuracy. The results seem to plateau and the increase in the NTFP value does not introduce more errors or reduce the accuracy levels, as was the case in the previous section with the acoustic guitar samples.

The Max and Average values when including octave errors plateaus slightly higher at 71 points. The NTFP value of 341 marks where the maximum and average values also plateau. This seems to indicate that increasing the NTFP value from 71 to 341 only decreases the number of octave mistakes by a small amount.

The previous experiments varying the NTFP from 1-50 appeared to show a plateauing in all of the accuracy measurements displayed in the graph and table at a NTFP value of 50, although the increase was very small.

The previous experiments also highlighted a NTFP value for the Oboe of 24 - above which only tiny improvements in accuracy were found. The results from this experiment seem to confirm that while there are can be increases in accuracy when using an NTFP value above the 21-24 range, the increases are extremely minor, less than 2% increase in max accuracy and less than 1% increase in average accuracy when jumping from a NTFP value of 21 to 341. When including octave errors, the gains in accuracy are even smaller.

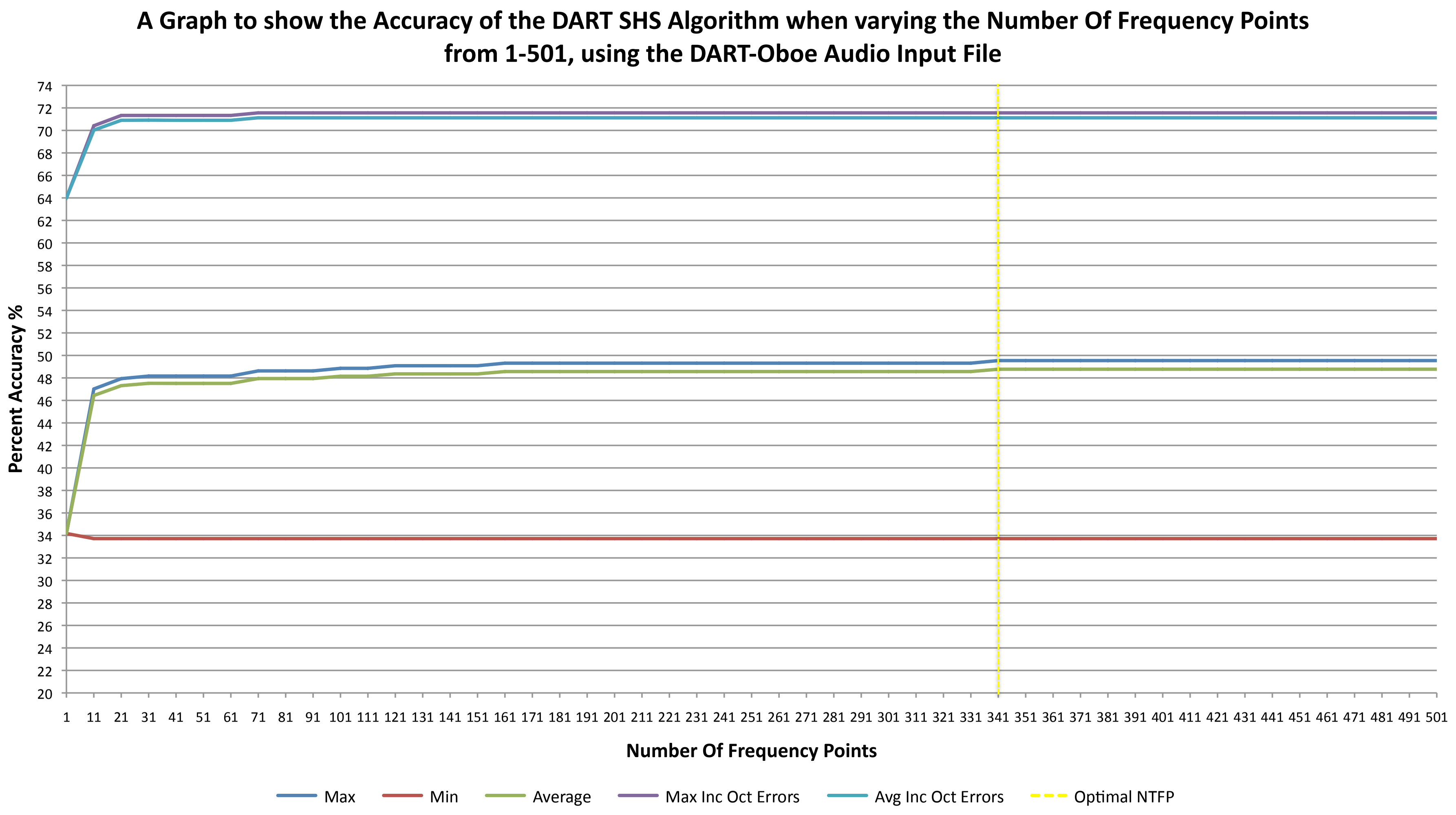


Figure 8.3: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-501 in increments of 10, with minimum, maximum and average score values using the DART Acoustic Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	34.17	34.17	34.17	63.99	63.99	63.99	29.82	29.82	29.82
11	47.02	33.72	46.44	70.41	70.03	70.41	30.05	22.25	23.59
21	47.94	33.72	47.31	71.33	70.90	71.33	30.05	22.25	23.59
31	48.17	33.72	47.52	71.33	70.91	71.33	30.05	22.25	23.39
41	48.17	33.72	47.51	71.33	70.90	71.33	30.05	22.02	23.39
51	48.17	33.72	47.51	71.33	70.90	71.33	30.05	22.02	23.39
61	48.17	33.72	47.51	71.33	70.90	71.33	30.05	22.02	23.39
71	48.62	33.72	47.94	71.56	71.12	71.56	30.05	22.02	23.18
81	48.62	33.72	47.94	71.56	71.12	71.56	30.05	22.02	23.18
91	48.62	33.72	47.94	71.56	71.12	71.56	30.05	22.02	23.18
101	48.85	33.72	48.15	71.56	71.12	71.56	30.05	22.02	22.97
111	48.85	33.72	48.15	71.56	71.12	71.56	30.05	22.02	22.97
121	49.08	33.72	48.36	71.56	71.12	71.56	30.05	22.02	22.76
131	49.08	33.72	48.36	71.56	71.12	71.56	30.05	22.02	22.76
141	49.08	33.72	48.36	71.56	71.12	71.56	30.05	22.02	22.76
151	49.08	33.72	48.36	71.56	71.12	71.56	30.05	22.02	22.76
161	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
171	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
181	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
191	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
201	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
211	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
221	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
231	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
241	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
251	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
261	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
271	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
281	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
291	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
301	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
311	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
321	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
331	49.31	33.72	48.57	71.56	71.12	71.56	30.05	22.02	22.56
341	**49.54**	**33.72**	**48.77**	**71.56**	**71.12**	**71.56**	**30.05**	**22.02**	**22.35**
351	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
361	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
371	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
381	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
391	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
401	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
411	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
421	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
431	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
441	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
451	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
461	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
471	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
481	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
491	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35
501	49.54	33.72	48.77	71.56	71.12	71.56	30.05	22.02	22.35

Table 8.2: This table details the accuracy of the DART SHS algorithm while varying the NTFP from 1-501 in multiples of 10, using the DART Oboe audio file. The row of the optimal NTFP value is highlighted bold. 'O.E' stands for *Octave Error*.

8.4.3 Number Of Frequency Points Results for Violin

Graph 8.4 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501 in increments of 10, using the DART Violin audio input file. Table 8.3 shows the values presented in Graph 8.4.

The results are similar to the results for the oboe, in that they plateau and do not begin to decrease in accuracy in a similar way to the acoustic guitar samples. However the general accuracy of the violin samples is much higher than the accuracy of the oboe, with an extremely high level of accuracy when allowing for octave errors.

As the Max and Average levels of accuracy when including octave errors plateaus at a NTFP value of **41**, it seems clear that increasing the NTFP value from 41 to **201** only decreases the number of octave mistakes.

Graph 8.4 shows that while the greatest accuracy is discovered when analysing the (maximum) **201** frequency points, looking at more than approximately **181** frequency points gives only an *extremely* minor improvement in accuracy (0.01% in the average accuracy), and any more than 41 is not required if only the pitch of the note is required, and not the octave.

The Max and Average values when including octave errors plateaus slightly higher at 71 points. The NTFP value of 341 marks where the maximum and average values also plateau. The previous experiments varying the NTFP from 1-50 appeared to show a plateauing in all of the accuracy measurements displayed in the graph and table at a NTFP value of 50, although the increase was very small.

The previous experiments also highlighted a NTFP value for the Violin of 23 - above which only tiny improvements in accuracy were found. The results from this experiment seem to show a worthwhile benefit in a higher number of NTFP values. For the violin audio input file, the optimal NTFP appears to be in the range of **181**, or **41** if octave mistakes are acceptable.

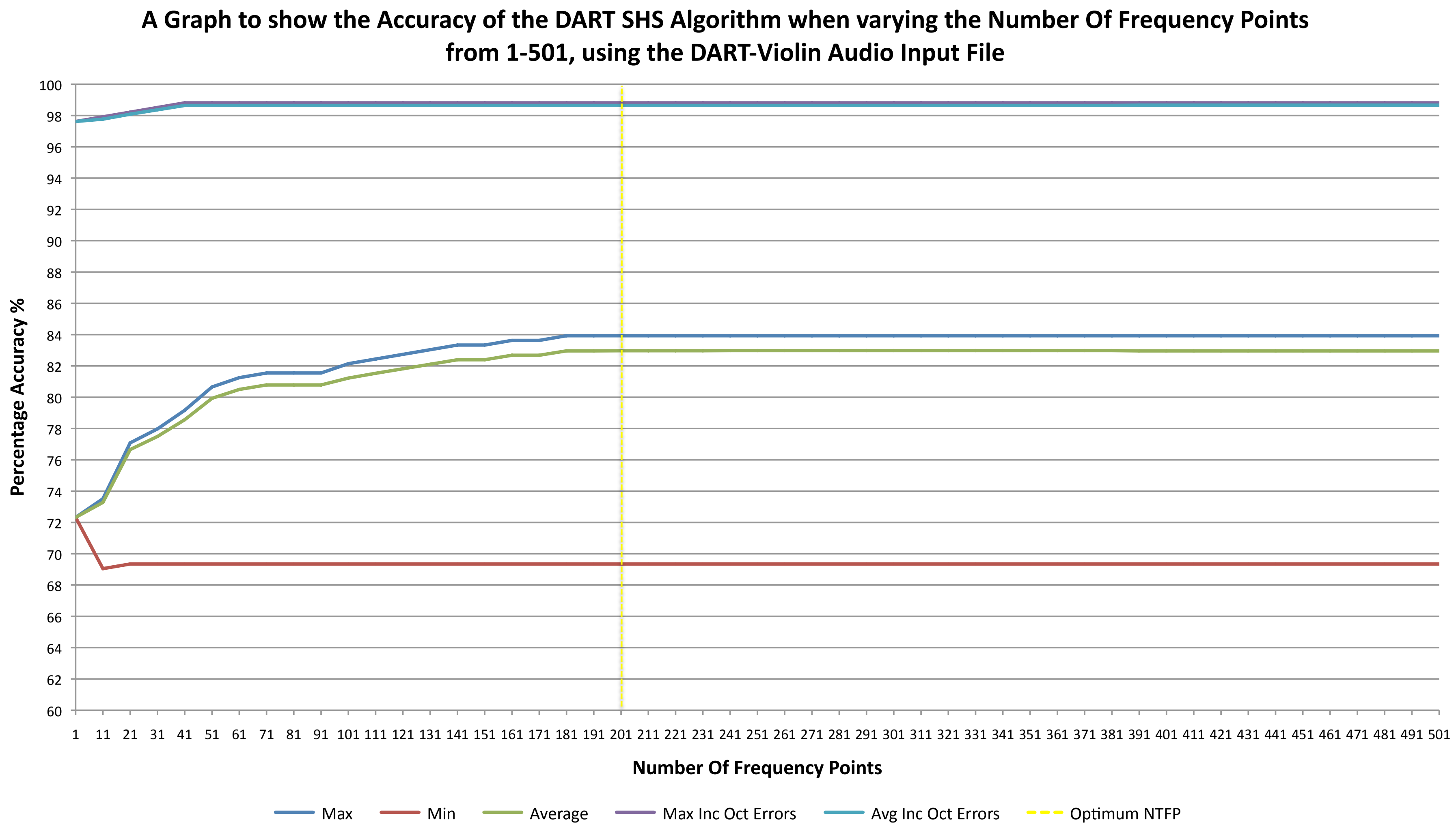


Figure 8.4: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-501 in increments of 10, with minimum, maximum and average score values using the DART Piano audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	72.32	72.32	72.32	97.62	97.62	97.62	25.30	25.30	25.30
11	73.51	69.05	73.28	97.92	97.77	97.92	26.49	24.40	24.49
21	77.08	69.35	76.66	98.21	98.08	98.21	26.79	21.13	21.43
31	77.98	69.35	77.49	98.51	98.36	98.51	26.79	20.54	20.87
41	79.17	69.35	78.56	98.81	98.64	98.81	26.79	19.64	20.08
51	80.65	69.35	79.93	98.81	98.64	98.81	26.79	18.15	18.71
61	81.25	69.35	80.50	98.81	98.64	98.81	26.79	17.56	18.15
71	81.55	69.35	80.78	98.81	98.64	98.81	26.79	17.26	17.86
81	81.55	69.35	80.78	98.81	98.64	98.81	26.79	17.26	17.86
91	81.55	69.35	80.78	98.81	98.64	98.81	26.79	17.26	17.86
101	82.14	69.35	81.22	98.81	98.64	98.81	26.79	16.67	17.42
111	82.44	69.35	81.53	98.81	98.64	98.81	26.79	16.37	17.11
121	82.74	69.35	81.82	98.81	98.64	98.81	26.79	16.07	16.82
131	83.04	69.35	82.11	98.81	98.64	98.81	26.79	15.77	16.54
141	83.33	69.35	82.39	98.81	98.64	98.81	26.79	15.48	16.25
151	83.33	69.35	82.39	98.81	98.64	98.81	26.79	15.48	16.25
161	83.63	69.35	82.68	98.81	98.64	98.81	26.79	15.18	15.96
171	83.63	69.35	82.68	98.81	98.64	98.81	26.79	15.18	15.96
181	83.93	69.35	82.96	98.81	98.64	98.81	26.79	14.88	15.68
191	83.93	69.35	82.96	98.81	98.64	98.81	26.79	14.88	15.68
201	**83.93**	**69.35**	**82.97**	**98.81**	**98.64**	**98.81**	**26.79**	**14.88**	**15.67**
211	83.93	69.35	82.97	98.81	98.64	98.81	26.79	14.88	15.67
221	83.93	69.35	82.97	98.81	98.64	98.81	26.79	14.88	15.67
231	83.93	69.35	82.97	98.81	98.64	98.81	26.79	14.88	15.67
241	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
251	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
261	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
271	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
281	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
291	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
301	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
311	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
321	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
331	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
341	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
351	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
361	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
371	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
381	83.93	69.35	82.98	98.81	98.64	98.81	26.79	14.88	15.66
391	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
401	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
411	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
421	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
431	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
441	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
451	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
461	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
471	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
481	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
491	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70
501	83.93	69.35	82.96	98.81	98.66	98.81	26.79	14.88	15.70

Table 8.3: This table details the accuracy of the DART SHS algorithm while varying the NTFP from 1-501 in multiples of 10, using the DART Violin audio file. The row of the optimal NTFP value is highlighted bold. 'O.E' stands for *Octave Error*.

8.4.4 Number Of Frequency Points Results for Piano

Graph 8.5 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501 in increments of 10, using the DART Violin audio input file. Table 8.4 shows the values presented in Graph 8.5.

Graph 8.5 shows that while the greatest accuracy is discovered when analysing **31** frequency points, that looking at more than **11** or **21** frequency points gives only a minor improvement in maximum or average (less than 1%). There is an average of only 1.5% pitch errors (not explainable with octave mistakes) when analysing 21 harmonics and including octave errors, with a maximum accuracy of 97.87% - extremely high in comparison to the other audio files.

As with the acoustic guitar samples, the NTFP mark of 31 is interestingly in-between the two values marked in the previous large scale parameter sweep experiments (where the NTFP value was varied from 1-50) as displayed in the previous chapter. The previous test noted that the Optimum NTFP value was 37, with only minor increases in accuracy above the NTFP value of 21. The figure of 31 found in the Pegasus experiments is roughly in-between these two values, providing extra evidence that the optimum NTFP value for this audio input file is in this range.

The previous experiments (varying the NTFP from 1-50) appeared to show a plateauing in all of the accuracy measurements displayed in the graphs and tables. However Graph 8.5 shows a clear decrease in overall accuracy as the NTFP value rises above 31.

After a small decrease the Maximum Accuracy Including Oct Errors does plateau at a NTFP value of 241, however. This is evidence to suggest that given a high enough number of harmonics, the accuracy could remain high when the NTFP value is increased. The steep decrease in Minimum accuracy levels at 271 with the level of Max Inc Oct Errors remaining unchanged also supports this.

Performing the experiments again but only varying the NTFP value would be able to a reliable way to confirm this. The steep decrease in Minimum accuracy levels could represent the point at which the algorithm simply begins to analyse noise instead of relevant harmonic data.

Given the current evidence from both experiments however, it seems that the best results for the Piano audio input file can be found using a low NTFP of around 31, the Rectangle FFT Window type, and 12 harmonics.

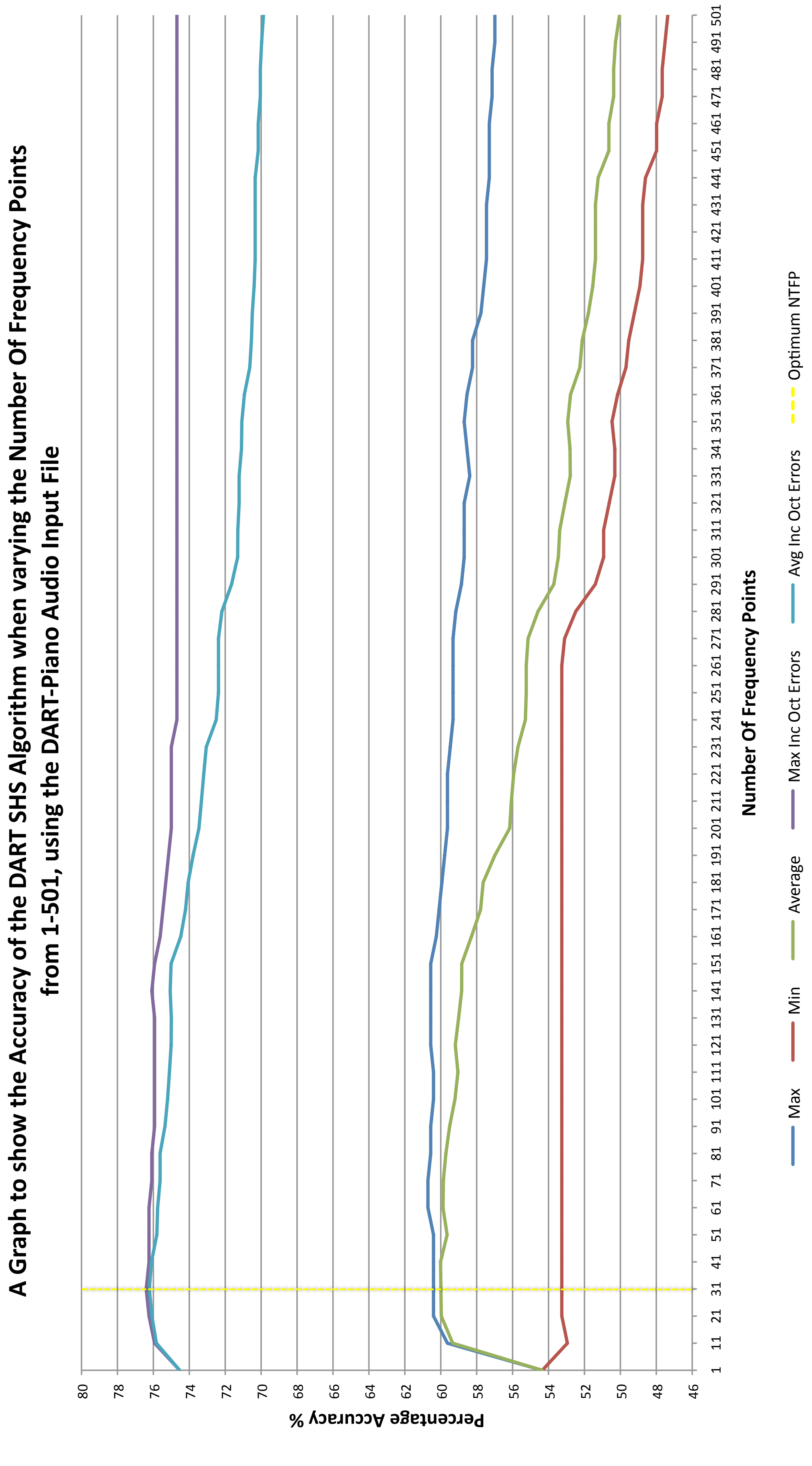


Figure 8.5: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-501 in increments of 10, with minimum, maximum and average score values using the DART Piano audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
1	54.35	54.35	54.35	74.53	74.53	74.53	20.19	20.19	20.19
11	59.63	52.95	59.34	75.93	75.83	75.93	21.12	16.30	16.49
21	**60.40**	**53.26**	**59.97**	**76.24**	**76.08**	**76.24**	**20.96**	**15.84**	**16.11**
31	60.40	53.26	59.99	76.40	76.22	76.40	20.96	15.84	16.23
41	60.40	53.26	60.01	76.24	76.11	76.24	20.96	15.84	16.10
51	60.40	53.26	59.65	76.24	75.81	76.24	20.96	15.84	16.16
61	60.71	53.26	59.87	76.24	75.76	76.24	20.96	15.53	15.89
71	60.71	53.26	59.86	76.09	75.63	76.09	20.96	15.37	15.77
81	60.56	53.26	59.71	76.09	75.63	76.09	20.96	15.53	15.91
91	60.56	53.26	59.51	75.93	75.35	75.93	20.96	15.37	15.85
101	60.40	53.26	59.21	75.93	75.21	75.93	20.96	15.53	16.00
111	60.40	53.26	59.05	75.93	75.11	75.93	20.96	15.53	16.06
121	60.56	53.26	59.19	75.93	75.01	75.93	20.96	15.22	15.82
131	60.56	53.26	59.01	75.93	75.00	75.93	20.96	15.37	15.99
141	60.56	53.26	58.84	76.09	75.06	76.09	20.96	15.53	16.23
151	60.56	53.26	58.83	75.93	75.01	75.93	20.96	15.37	16.18
161	60.25	53.26	58.29	75.62	74.48	75.62	20.96	15.37	16.18
171	60.09	53.26	57.78	75.47	74.21	75.47	20.96	15.37	16.43
181	59.94	53.26	57.64	75.31	74.06	75.31	20.96	15.37	16.43
191	59.78	53.26	56.99	75.16	73.79	75.16	20.96	15.37	16.80
201	59.63	53.26	56.16	75.00	73.46	75.00	20.96	15.37	17.30
211	59.63	53.26	56.07	75.00	73.33	75.00	20.96	15.37	17.26
221	59.63	53.26	55.94	75.00	73.19	75.00	20.96	15.37	17.26
231	59.47	53.26	55.70	75.00	73.05	75.00	20.96	15.53	17.35
241	59.32	53.26	55.29	74.69	72.50	74.69	20.96	15.37	17.21
251	59.32	53.26	55.24	74.69	72.37	74.69	20.96	15.37	17.13
261	59.32	53.26	55.24	74.69	72.37	74.69	20.96	15.37	17.13
271	59.32	53.11	55.14	74.69	72.37	74.69	20.96	15.37	17.24
281	59.16	52.48	54.59	74.69	72.19	74.69	20.96	15.53	17.60

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
291	58.85	51.40	53.71	74.69	71.65	74.69	20.96	15.84	17.94
301	58.70	50.93	53.45	74.69	71.31	74.38	20.96	15.68	17.85
311	58.70	50.93	53.38	74.69	71.30	74.38	20.96	15.68	17.92
321	58.70	50.62	53.09	74.69	71.22	74.22	20.96	15.53	18.13
331	58.39	50.31	52.79	74.69	71.22	74.22	20.96	15.84	18.43
341	58.54	50.31	52.80	74.69	71.09	74.22	20.96	15.68	18.29
351	58.70	50.47	52.93	74.69	71.07	74.22	20.96	15.53	18.14
361	58.54	50.16	52.78	74.69	70.93	74.22	20.96	15.68	18.16
371	58.23	49.69	52.26	74.69	70.63	74.07	20.96	15.84	18.38
381	58.23	49.53	52.13	74.69	70.54	74.07	20.96	15.84	18.42
391	57.76	49.22	51.78	74.69	70.49	73.91	20.96	16.15	18.71
401	57.61	48.91	51.54	74.69	70.39	73.76	20.96	16.15	18.85
411	57.45	48.76	51.39	74.69	70.33	73.76	20.96	16.30	18.94
421	57.45	48.76	51.39	74.69	70.33	73.76	20.96	16.30	18.94
431	57.45	48.76	51.39	74.69	70.33	73.76	20.96	16.30	18.94
441	57.30	48.60	51.24	74.69	70.33	73.76	20.96	16.46	19.09
451	57.30	47.98	50.64	74.69	70.16	73.76	20.96	16.46	19.52
461	57.30	47.98	50.64	74.69	70.16	73.76	20.96	16.46	19.52
471	57.14	47.67	50.37	74.69	70.05	73.76	20.96	16.61	19.67
481	57.14	47.67	50.37	74.69	70.05	73.76	20.96	16.61	19.67
491	56.99	47.52	50.27	74.69	69.97	73.76	20.96	16.77	19.70
501	56.99	47.36	50.05	74.69	69.87	73.76	20.96	16.77	19.82

Table 8.4: This table details the accuracy of the DART SHS algorithm while varying the NTFP from 1-501 in multiples of 10, using the DART Piano audio file. The row of the optimal NTFP value is highlighted bold. 'O.E' stands for *Octave Error*.

8.4.5 Number Of Frequency Points Results for Tubular Bells

Graph 8.6 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501, using the DART Tubular Bells audio input file. Table 8.5 shows the values presented in Graph 8.6.

Graph 8.6 shows that the greatest accuracy is discovered when analysing **261** frequency points. However, from 1-261 points the Maximum and Average accuracies only increase by less than 2%. The Average Inc. Octave Errors actually decreases slightly from 11 to 21 points, and remains constant at 65.45% for the remainder of the experiment. As a result of such a minor increase and a decrease in the average accuracy (including octave errors), the optimum NTFP value of 261 is slightly misleading.

The previous experiments varying the NTFP from 1-50 appeared to show a levelling in all of the accuracy measurements displayed in the graphs and tables and slightly better performance at lower NTFP values. While the use of the Rectangle window was both optimal overall and for the Tubular Bells, the discrepancies between the higher scores at lower NTFP values could have been due to a different FFT Window (which was not used this time) performing slightly better at lower NTFP values. This could also explain the reason why the Average and Maximum accuracy values are equal in this set of experiments, but not in the previous large scale experiments.

Due to the extremely low level of fundamental frequency audio created when Tubular Bells are struck, it is not surprising that the accuracy levels are extremely low when octave mistakes are not allowed. Given this, it makes little sense to factor in the less than 2% accuracy rate when analysing the results; much more weight can be given to the Max and Average accuracy rates when including octave errors. This peaks at a NTFP value of **11** (65.91% accuracy), before dropping very slightly and remaining constant at 65.45%.

The previous experiments showed an Optimum NTFP value of 36, with only a minor increase in accuracy above an NTFP value of around 24. However, when following the precedent set in this analysis and only looking at the Max and Average accuracy when including octave errors, the NTFP figure of **11** again shows itself to be the optimum value.

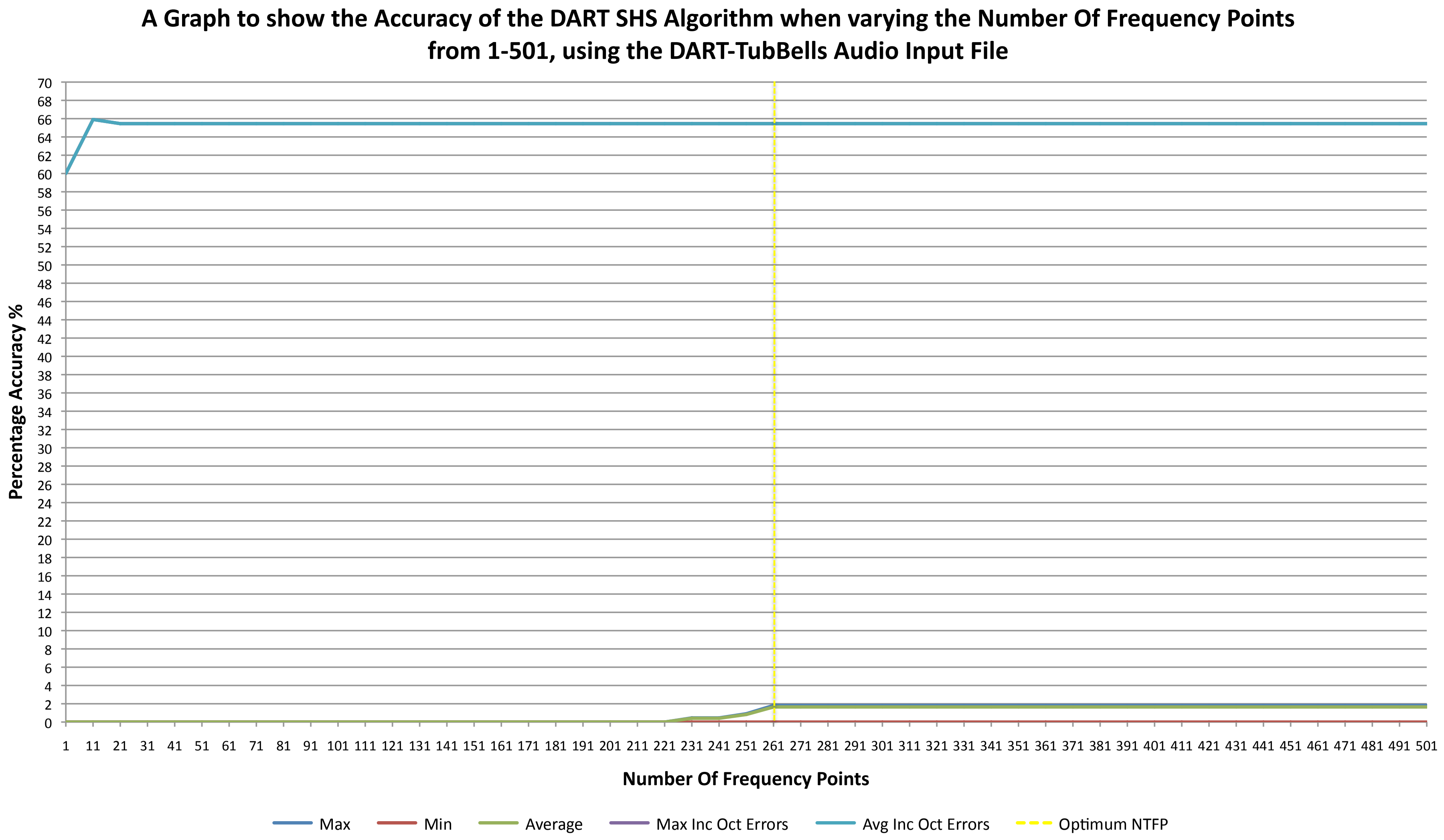


Figure 8.6: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-501 in steps of 10, with minimum, maximum and average score values using the DART Tubular Bells audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	0.00	0.00	0.00	60.00	60.00	60.00	60.00	60.00	60.00
11	0.00	0.00	0.00	65.91	65.91	65.91	65.91	65.91	65.91
21	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
31	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
41	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
51	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
61	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
71	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
81	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
91	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
101	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
111	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
121	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
131	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
141	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
151	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
161	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
171	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
181	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
191	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
201	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
211	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
221	0.00	0.00	0.00	65.45	65.45	65.45	65.45	65.45	65.45
231	0.45	0.00	0.41	65.45	65.45	65.45	65.45	65.00	65.04
241	0.45	0.00	0.41	65.45	65.45	65.45	65.45	65.00	65.04
251	0.91	0.00	0.82	65.45	65.45	65.45	65.45	64.55	64.63
261	**1.82**	**0.00**	**1.65**	**65.45**	**65.45**	**65.45**	**65.45**	**63.64**	**63.81**
271	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
281	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
291	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
301	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
311	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
321	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
331	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
341	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
351	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
361	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
371	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
381	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
391	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
401	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
411	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
421	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
431	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
441	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
451	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
461	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
471	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
481	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
491	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81
501	1.82	0.00	1.65	65.45	65.45	65.45	65.45	63.64	63.81

Table 8.5: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Tubular Bells audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

8.4.6 Number Of Frequency Points Results for Distorted Guitar

Graph 8.7 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501 in steps of 10, using the DART Distorted Guitar audio input file. Table 8.6 shows the values presented in Graph 8.7.

The original large scale experiment results for the Distorted Guitar presented in the previous chapter show a gradual increase in accuracy as the NTFP value rose to the maximum of 50. One of the main motivations for carrying out the extra NTFP experiments presented in this chapter was to see how much the accuracy would rise given a large enough value, and indeed if they would continue to rise.

The extra tests show that the increase in accuracy becomes much more gradual after a NTFP value of 50. Graph 8.7 shows that the greatest accuracy is discovered when analysing **321** frequency points - looking at more than 51 frequency points gives only a minor improvement in maximum or average (around 3% for most measurements). After an NTFP value of around 341, the overall accuracy seems to begin to decrease.

It should also be noted that while the use of the Rectangle window was optimal overall, the large scale parameter sweep experiments performed in the previous chapter revealed that the Acoustic Guitar audio file achieved higher accuracy rates using the Nutall4 FFT Window Type, with 15 harmonics.

A future run of experiments tailored to the optimisation of the specific parameter being analysed (NTFP) would also be beneficial; running experiments sweeping from 1-501 while keeping the FFT Window to Nutall4 and using 15 Harmonics for the Distorted Guitar audio input file would make for interesting further analysis.

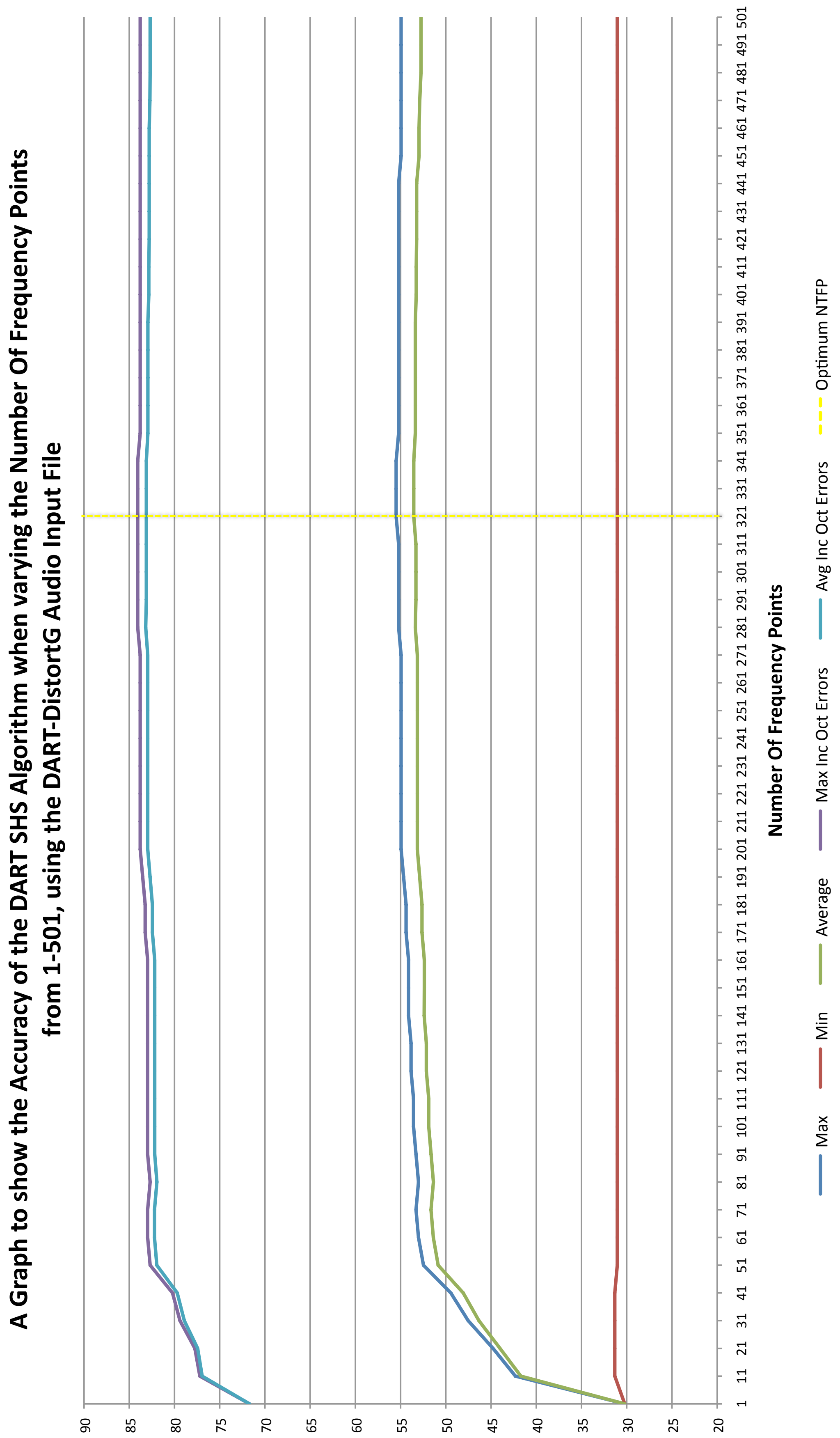


Figure 8.7: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using the DART Distorted Guitar audio input file. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	30.22	30.22	30.22	71.70	71.70	71.70	41.48	41.48	41.48
11	42.31	31.32	41.71	77.20	76.94	77.20	41.76	34.89	35.23
21	44.78	31.32	44.01	77.75	77.44	77.75	41.76	32.97	33.43
31	47.53	31.32	46.34	79.40	78.91	79.40	41.76	31.87	32.57
41	49.45	31.32	48.08	80.22	79.70	80.22	41.76	30.77	31.62
51	52.47	31.04	50.85	82.69	81.95	82.69	42.03	30.22	31.10
61	53.02	31.04	51.38	82.97	82.21	82.97	42.03	29.95	30.83
71	53.30	31.04	51.64	82.97	82.21	82.97	42.03	29.67	30.57
81	53.02	31.04	51.37	82.69	81.95	82.69	42.03	29.67	30.57
91	53.30	31.04	51.63	82.97	82.19	82.97	42.03	29.67	30.55
101	53.57	31.04	51.89	82.97	82.19	82.97	42.03	29.40	30.30
111	53.57	31.04	51.89	82.97	82.19	82.97	42.03	29.40	30.30
121	53.85	31.04	52.15	82.97	82.19	82.97	42.03	29.12	30.03
131	53.85	31.04	52.15	82.97	82.19	82.97	42.03	29.12	30.03
141	54.12	31.04	52.40	82.97	82.19	82.97	42.03	28.85	29.79
151	54.12	31.04	52.38	82.97	82.19	82.97	42.03	28.85	29.81
161	54.12	31.04	52.38	82.97	82.19	82.97	42.03	28.85	29.81
171	54.40	31.04	52.64	83.24	82.44	83.24	42.03	28.85	29.81
181	54.40	31.04	52.64	83.24	82.44	83.24	42.03	28.85	29.81
191	54.67	31.04	52.89	83.52	82.70	83.52	42.03	28.85	29.81
201	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
211	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
221	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
231	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
241	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
251	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
261	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
271	54.95	31.04	53.15	83.79	82.96	83.79	42.03	28.85	29.81
281	55.22	31.04	53.37	84.07	83.18	84.07	42.03	28.85	29.81

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
291	55.22	31.04	53.31	84.07	83.11	84.07	42.03	28.85	29.81
301	55.22	31.04	53.31	84.07	83.11	84.07	42.03	28.85	29.81
311	55.22	31.04	53.31	84.07	83.11	84.07	42.03	28.85	29.81
321	**55.49**	**31.04**	**53.55**	**84.07**	**83.11**	**84.07**	**42.03**	**28.57**	**29.57**
331	55.49	31.04	53.55	84.07	83.11	84.07	42.03	28.57	29.57
341	55.49	31.04	53.55	84.07	83.11	84.07	42.03	28.57	29.57
351	55.22	31.04	53.37	83.79	82.94	83.79	42.03	28.57	29.57
361	55.22	31.04	53.37	83.79	82.94	83.79	42.03	28.57	29.57
371	55.22	31.04	53.37	83.79	82.94	83.79	42.03	28.57	29.57
381	55.22	31.04	53.37	83.79	82.94	83.79	42.03	28.57	29.57
391	55.22	31.04	53.37	83.79	82.94	83.79	42.03	28.57	29.57
401	55.22	31.04	53.27	83.79	82.84	83.79	42.03	28.57	29.57
411	55.22	31.04	53.27	83.79	82.84	83.79	42.03	28.57	29.57
421	55.22	31.04	53.23	83.79	82.80	83.79	42.03	28.57	29.57
431	55.22	31.04	53.23	83.79	82.80	83.79	42.03	28.57	29.57
441	55.22	31.04	53.23	83.79	82.80	83.79	42.03	28.57	29.57
451	54.95	31.04	52.96	83.79	82.80	83.79	42.03	28.85	29.83
461	54.95	31.04	52.96	83.79	82.80	83.79	42.03	28.85	29.83
471	54.95	31.04	52.88	83.79	82.72	83.79	42.03	28.85	29.83
481	54.95	31.04	52.75	83.79	82.69	83.79	42.03	28.85	29.95
491	54.95	31.04	52.75	83.79	82.69	83.79	42.03	28.85	29.95
501	54.95	31.04	52.75	83.79	82.69	83.79	42.03	28.85	29.95

Table 8.6: This table details the accuracy of the DART SHS algorithm while varying the NTFP using the DART Distorted Guitar audio file. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

8.4.7 Number Of Frequency Points Results for All Audio Files

Graph 8.8 shows the relative accuracy of the SHS algorithm when modifying the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501, using all DART audio input files. Table 8.7 shows the values presented in Graph 8.8.

Across all audio files and accuracy figures, the NTFP value of **61** was found to give the highest accuracy *overall*.

The maximum accuracy when not including octave errors gradually declines after a peak of **21**. This figure matches nearly exactly with the figure presented in previous experiment presented in Graph 7.34. The previous experiment found that when analysing all files, increasing the NTFP value over **22** gave only minor (statistically insignificant) increases in accuracy. The NTFP figure of 50 was found to be optimal, however the increase was extremely minor. All other accuracy measurements increase up to a NTFP value of 51, after which they begin to decline in accuracy.

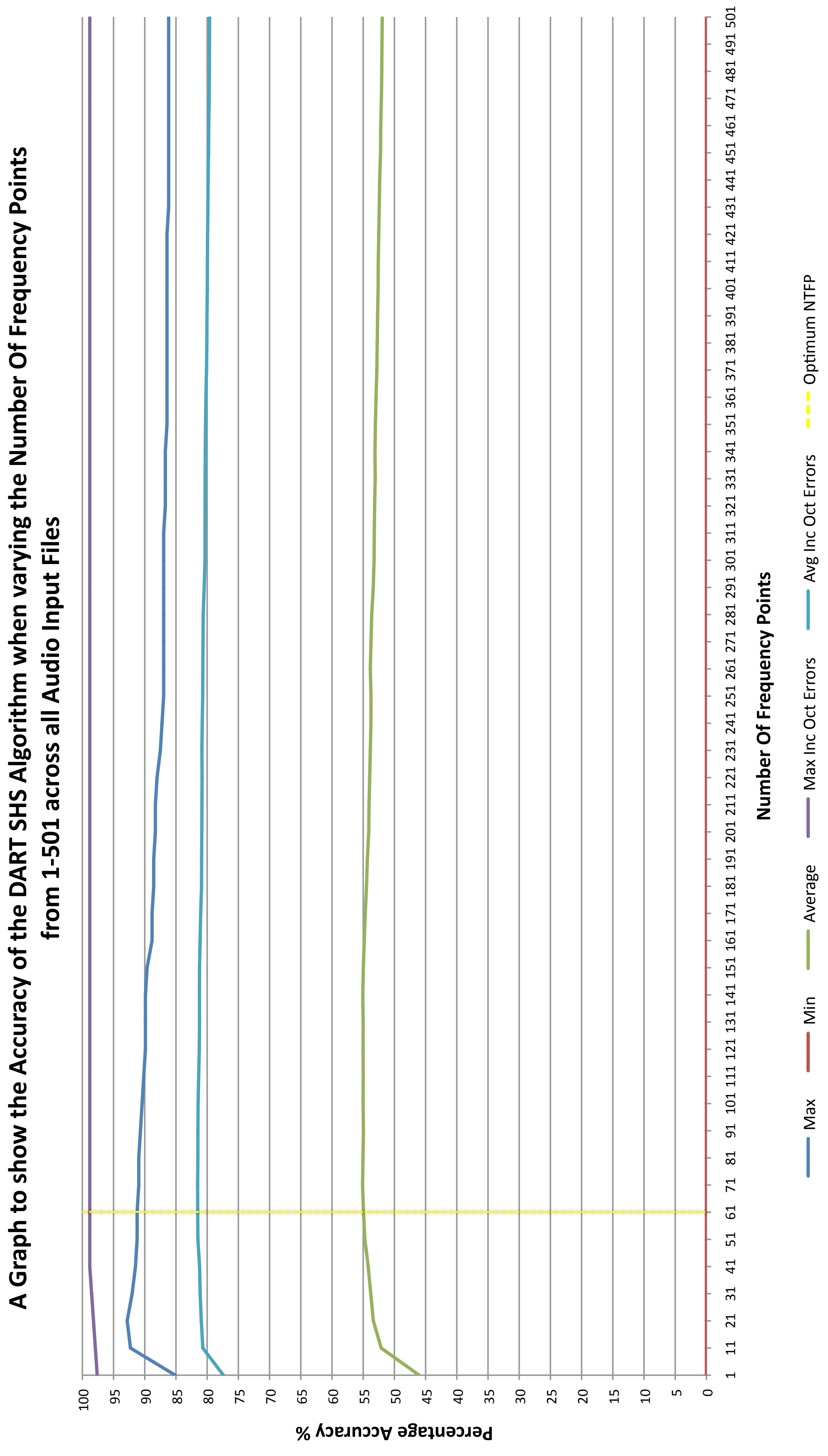


Figure 8.8: A graph to show the relative accuracy of the SHS algorithm when modifying the NTFP from 1-50, with minimum, maximum and average score values using all DART audio input files. Also included are the maximum and average accuracy results when ignoring the octave mistakes.

NTFP	Max %	Min %	Average %	Max Inc O.E %	Avg Inc O.E %	Top + O.E %	Max O.E %	Min O.E %	Avg O.E %
1	85.11	0.00	46.03	97.62	77.40	96.54	60.00	11.44	31.37
11	92.29	0.00	52.11	97.92	80.70	97.87	65.91	5.59	28.58
21	92.82	0.00	53.37	98.21	80.94	97.87	65.45	5.05	27.57
31	92.02	0.00	53.79	98.51	81.12	97.07	65.45	5.05	27.34
41	91.49	0.00	54.19	98.81	81.22	96.54	65.45	5.05	27.02
51	91.22	0.00	54.73	98.81	81.48	96.28	65.45	5.05	26.75
61	**91.22**	**0.00**	**54.95**	**98.81**	**81.52**	**96.28**	**65.45**	**5.05**	**26.57**
71	90.96	0.00	55.10	98.81	81.53	96.54	65.45	5.32	26.43
81	90.96	0.00	55.03	98.81	81.49	96.54	65.45	5.32	26.45
91	90.69	0.00	54.96	98.81	81.48	96.28	65.45	5.59	26.52
101	90.43	0.00	55.01	98.81	81.45	96.28	65.45	5.85	26.44
111	90.16	0.00	55.00	98.81	81.36	96.54	65.45	5.85	26.36
121	89.89	0.00	55.02	98.81	81.26	96.54	65.45	6.12	26.24
131	89.89	0.00	55.00	98.81	81.23	96.54	65.45	6.12	26.22
141	89.89	0.00	55.06	98.81	81.24	96.54	65.45	6.12	26.17
151	89.63	0.00	54.98	98.81	81.23	96.54	65.45	6.12	26.25
161	88.83	0.00	54.83	98.81	81.13	96.54	65.45	6.91	26.30
171	88.83	0.00	54.67	98.81	81.04	96.54	65.45	6.91	26.37
181	88.56	0.00	54.47	98.81	80.90	96.54	65.45	7.18	26.43
191	88.56	0.00	54.33	98.81	80.89	96.54	65.45	7.45	26.56
201	88.30	0.00	54.10	98.81	80.86	96.54	65.45	7.71	26.76
211	88.30	0.00	54.06	98.81	80.84	96.54	65.45	7.71	26.78
221	88.03	0.00	53.96	98.81	80.82	96.54	65.45	7.98	26.86
231	87.50	0.00	53.86	98.81	80.84	96.54	65.45	8.51	26.98
241	87.23	0.00	53.76	98.81	80.79	96.54	65.45	8.78	27.03
251	86.97	0.00	53.75	98.81	80.71	96.54	65.45	9.04	26.97
261	86.97	0.00	53.86	98.81	80.70	96.54	65.45	9.04	26.84
271	86.97	0.00	53.74	98.81	80.67	96.54	65.45	9.04	26.92
281	86.97	0.00	53.63	98.81	80.63	96.54	65.45	9.04	27.00

NTFP	**Max %**	**Min %**	**Average %**	**Max Inc O.E %**	**Avg Inc O.E %**	**Top + O.E %**	**Max O.E %**	**Min O.E %**	**Avg O.E %**
291	86.97	0.00	53.39	98.81	80.48	96.54	65.45	9.04	27.10
301	86.97	0.00	53.26	98.81	80.35	96.54	65.45	9.04	27.09
311	86.97	0.00	53.22	98.81	80.34	96.54	65.45	9.04	27.12
321	86.70	0.00	53.17	98.81	80.33	96.54	65.45	9.31	27.16
331	86.70	0.00	53.07	98.81	80.32	96.54	65.45	9.31	27.25
341	86.70	0.00	53.11	98.81	80.30	96.54	65.45	9.31	27.19
351	86.44	0.00	53.05	98.81	80.26	96.54	65.45	9.57	27.22
361	86.44	0.00	52.94	98.81	80.20	96.54	65.45	9.57	27.26
371	86.44	0.00	52.81	98.81	80.11	96.54	65.45	9.57	27.29
381	86.44	0.00	52.75	98.81	80.05	96.54	65.45	9.57	27.30
391	86.44	0.00	52.69	98.81	80.03	96.54	65.45	9.57	27.34
401	86.44	0.00	52.61	98.81	79.97	96.54	65.45	9.57	27.36
411	86.44	0.00	52.58	98.81	79.96	96.54	65.45	9.57	27.38
421	86.44	0.00	52.50	98.81	79.94	96.54	65.45	9.57	27.44
431	86.17	0.00	52.42	98.81	79.88	96.54	65.45	9.84	27.46
441	86.17	0.00	52.36	98.81	79.85	96.54	65.45	9.84	27.49
451	86.17	0.00	52.22	98.81	79.78	96.54	65.45	9.57	27.56
461	86.17	0.00	52.19	98.81	79.75	96.54	65.45	9.57	27.56
471	86.17	0.00	52.08	98.81	79.68	96.54	65.45	9.57	27.60
481	86.17	0.00	52.03	98.81	79.64	96.54	65.45	9.57	27.62
491	86.17	0.00	51.99	98.81	79.63	96.54	65.45	9.57	27.64
501	86.17	0.00	51.95	98.81	79.61	96.54	65.45	9.57	27.66

Table 8.7: This table details the accuracy of the DART SHS algorithm while varying the NTFP using all DART audio input files. The top values in each column are highlighted bold. 'O.E' stands for *Octave Error*.

8.4.8 Number Of Top Frequency Points Accuracy Summary

A summary of the NTFP results presented so far is given in Table 8.8. This table shows, for each audio input file, the NTFP value that gave the maximum accuracy (where increasing the NTFP shows no further improvement).

The NTFP that consistently produced the most accurate results across all audio input files was ***61***. Contrary to the results suggested in the previous chapter, the overall accuracy does not continues to increase continually as the NTFP increases. When visually analysing the graphs only minor increases in accuracy (a maximum of around 3-4%) are found after increasing over 51 points, and usually the results are far below this value. In the case of the acoustic guitar and piano, increasing past NTFP values of 21 and 31 (respectively) resulted in a relatively sharp decrease in accuracy.

In order to fully research the effect of the modification of the NTFP value, further experimentation is carried out in Section 8.6. This research isolates the effect of modifying only the NTFP value, whilst keeping all other variables constant

Audio Input File	***Maximum NTFP***
Acoustic Guitar	**21**
Oboe	**341**
Violin	**201**
Piano	**31**
Tubular Bells	**261**
Distorted Guitar	**321**
All Audio Files	**61**

Table 8.8: This table shows a summary of the Number Of Top Frequency Points that give the most accurate results for each Audio Input File.

8.5 Pegasus Time Analysis

The Pegasus experiments ran the DART NTFP experiments on a local 'Condor Pool' at ISI consisting of 7 nodes; 5 dual core and 2 quad core. Each core on a machine was made available as a Condor slot, giving a total of 18 slots available to run DART jobs. The specification of the two types of machines available were:

- Dual core AMD 2.4GHz / 4 GB RAM
- 2 x Dual core AMD 2.2GHz / 8 GB RAM

The total processing time for analysing 9,792 results over 6 worker nodes on the small Pegasus/Condor platform was **604,819** seconds (nearly exactly 7 days). These figures are calculated by adding together the total run time, in seconds, from the 9,792 results files retrieved. These figures do not include any of the Pegasus job submission and results or data download times. The Pegasus workflow completed processing the 9,792 jobs in only **9 hours, 41 minutes** (34,897 seconds).

The minimum DART job execution time was 44 seconds for a single job, with a maximum of 137 seconds. The average job execution time was 61.76 seconds.

In order to ascertain if increasing the NTFP value has a severe detrimental effect on the time taken to complete a single job, and investigation into the average time taken as the NTFP value increases would have to take place on a single machine, or at least on a platform where all machines are of similar (preferably exact) specification. However, it is possible to see an increase in the average time taken to process the DART SHS algorithm on the Pegasus platform in Graph 8.9. This graph displays the total time time taken to execute each set of experiments for all audio files and roughly shows a 15-20 second increase in time from the region of NTFP values of 1-50, to the 451-501 level.

Given the extremely minor increase (and in more than one case a clear decrease) in accuracy levels gained by using NTFP values of over 50, these tests reveal that using an NTFP value of over 50 would not be advantageous or recommended in most cases.

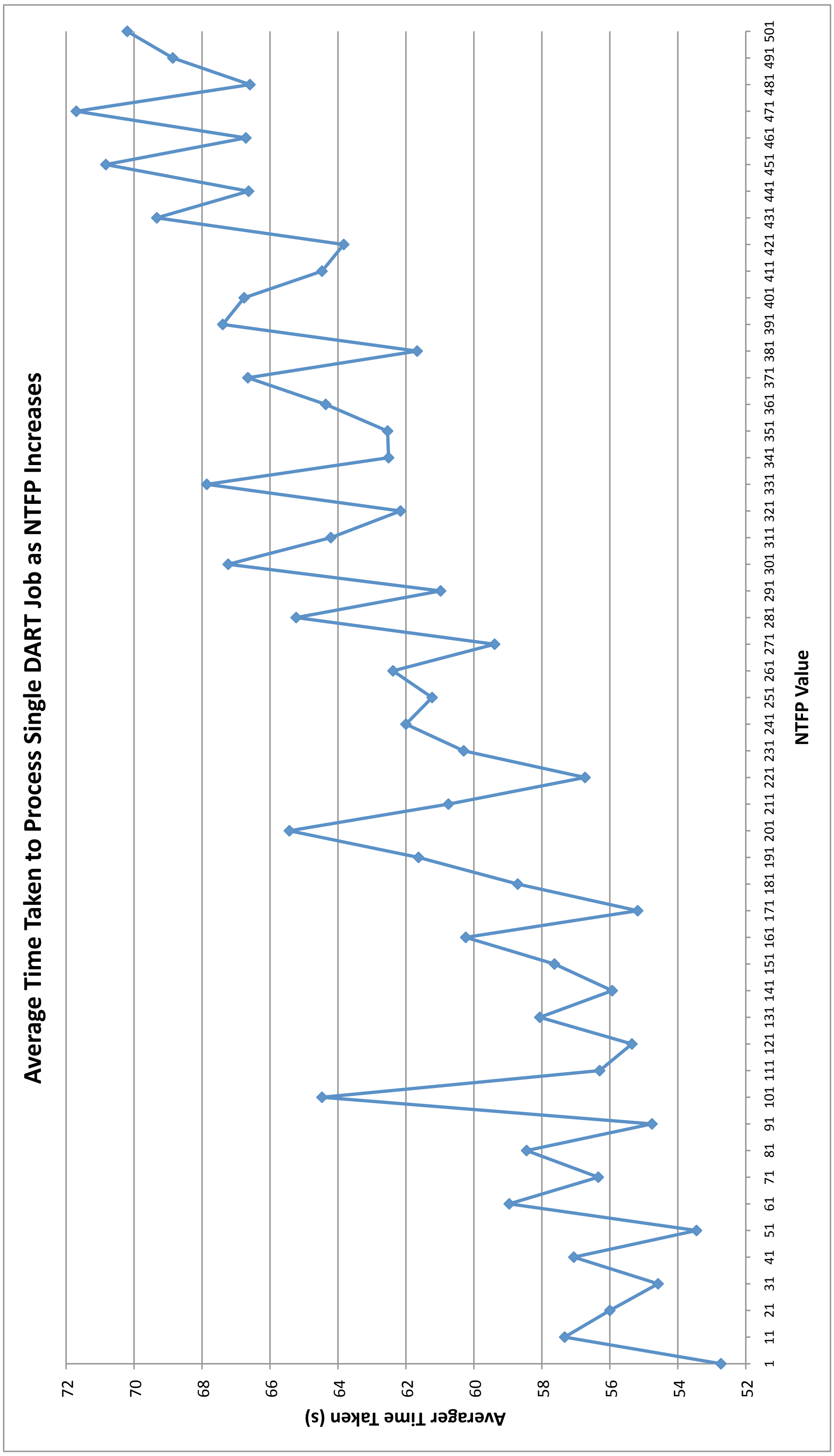


Figure 8.9: A graph to show the increase in time taken to process a single DART job on the Pegasus platform as NTFP increases from 1 to 501 in multiples of 10

8.6 Modifying Only the NTFP Value

This section displays the accuracy levels of the SHS algorithm when varying ***only*** the NTFP value from 1-501, in steps of 10, whilst maintaining the Number of Harmonics constant at 15 (the optimum value discovered in the previous chapter), and the FFT Window Type at 'Rectangle' (the overall optimal FFT Window Type also discovered in the previous chapter), allowing for a clear observation of the effect of solely modifying the NTFP value. This creates 306 experiments in total (6 audio files x 51 NTFP values). The results are presented in 7 graphs in the following subsection; one for each audio file, and a final graph showing the results across all audio files.

As the only variable in these experiments was the NTFP value, there is no minimum, maximum or average values for accuracy; only 'Accuracy' and 'Accuracy Including Octave Errors', and therefore these experiments are extremely helpful in finding the optimum NTFP value for situations where the note itself is required and octave errors are acceptable.

8.6.1 Acoustic Guitar

Graph 8.10 shows the relative accuracy of the SHS algorithm when modifying only the Number Of Top Frequency Points analysed by the SHS algorithm from 1-501 in steps of 10, using the DART Acoustic Guitar audio input file. The yellow line indicates the optimum NTFP for maximum accuracy, with the orange line showing the optimum NTFP for accuracy when including octave errors.

While the optimal FFT window type for the acoustic guitar was not used in these experiments, the overall accuracy of the acoustic guitar samples was high at lower NTFP values, especially when ignoring octave errors. The greatest accuracy is discovered when analysing **21** frequency points, with the greatest accuracy when including octave mistakes is found at only **11** frequency points. These results show the steady decrease of accuracy after these points and support the results found in previous experiments.

Given the current evidence from both experiments it is possible to confirm that the most accurate results for the Acoustic Guitar audio input file can be found using a low NTFP of around 21, the Welch FFT Window type, and 14 harmonics.

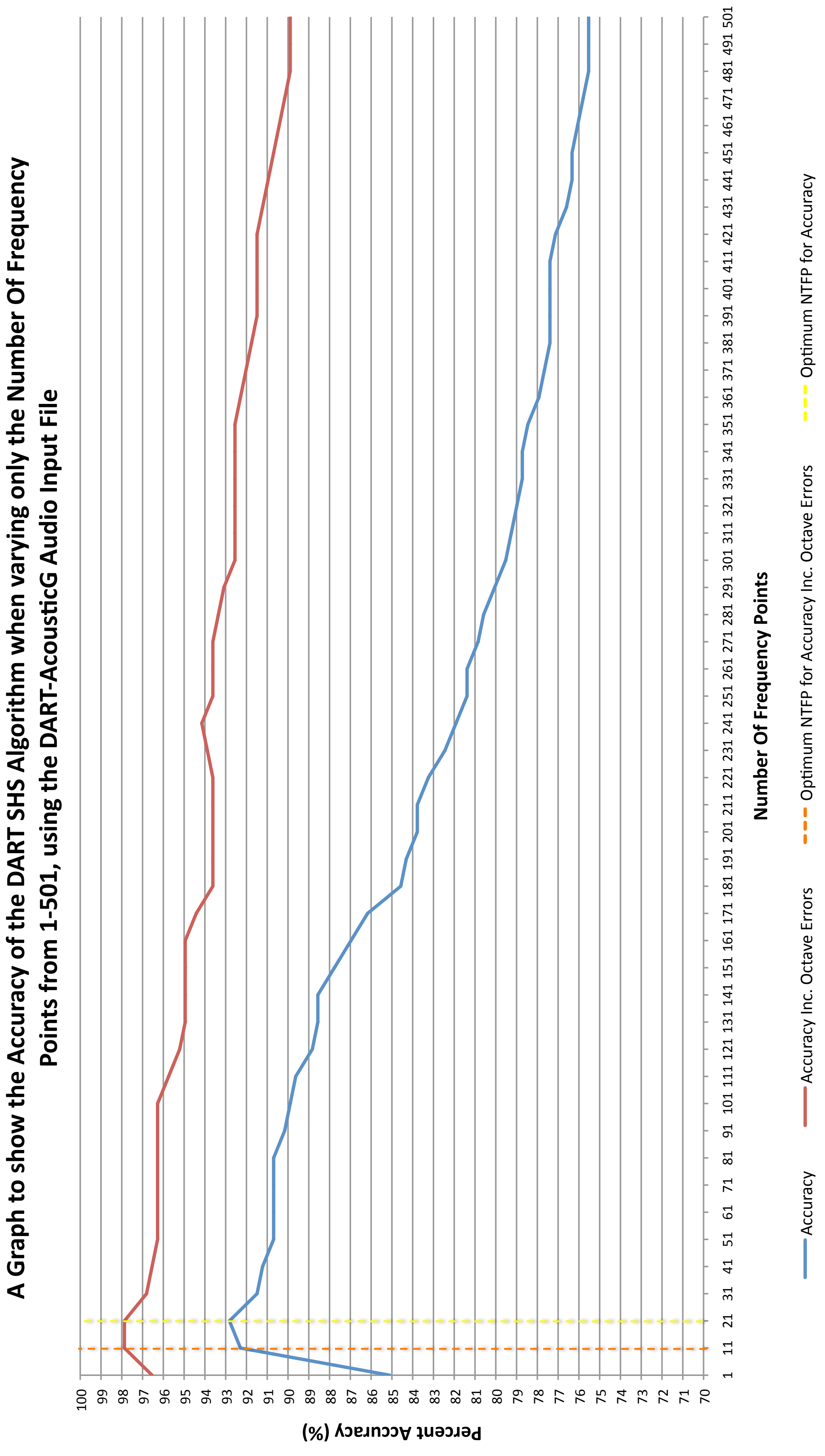


Figure 8.10: A graph showing the accuracy of the SHS algorithm when modifying the NTFP from 1-501 using the DART Acoustic Guitar audio input file

8.6.2 Oboe

Graph 8.11 shows the accuracy of the SHS algorithm when modifying only the NTFP analysed by the SHS algorithm from 1-501 in steps of 10, using the DART Oboe.

The greatest accuracy is discovered when analysing a high NTFP value of **341** (as in the previous experiment) but with the much smaller NTFP value of **71** found for accuracy when including octave mistakes. Again, looking at more than approximately 21 frequency points gives only a minor improvement in accuracy. The results plateau, however the increase in the NTFP value does not introduce more errors or reduce the accuracy levels, as was the case in the previous section with the acoustic guitar samples.

The previous experiments highlighted an NTFP value for the Oboe of *24* - above which only tiny improvements in accuracy were found. The graph from this experiment seem to confirm that while there are can be increases in accuracy when using an NTFP value above the 21-24 range, the increases are extremely minor, less than 2% increase in max accuracy and less than 1% increase in average accuracy when jumping from a NTFP value of 21 to 341. When including octave errors, the gains in accuracy are even smaller. The results imply that the increase in accuracy found when increasing the NTFP value from 41 to 181 is mainly reducing the number of octave errors.

Given the current evidence from both experiments it seems that satisfactory results for the Oboe audio input file can be found using a low NTFP of around 21, the Rectangle FFT Window type, and 13 harmonics. However increasing the NTFP value to around 371 will give a very slight increase in accuracy.

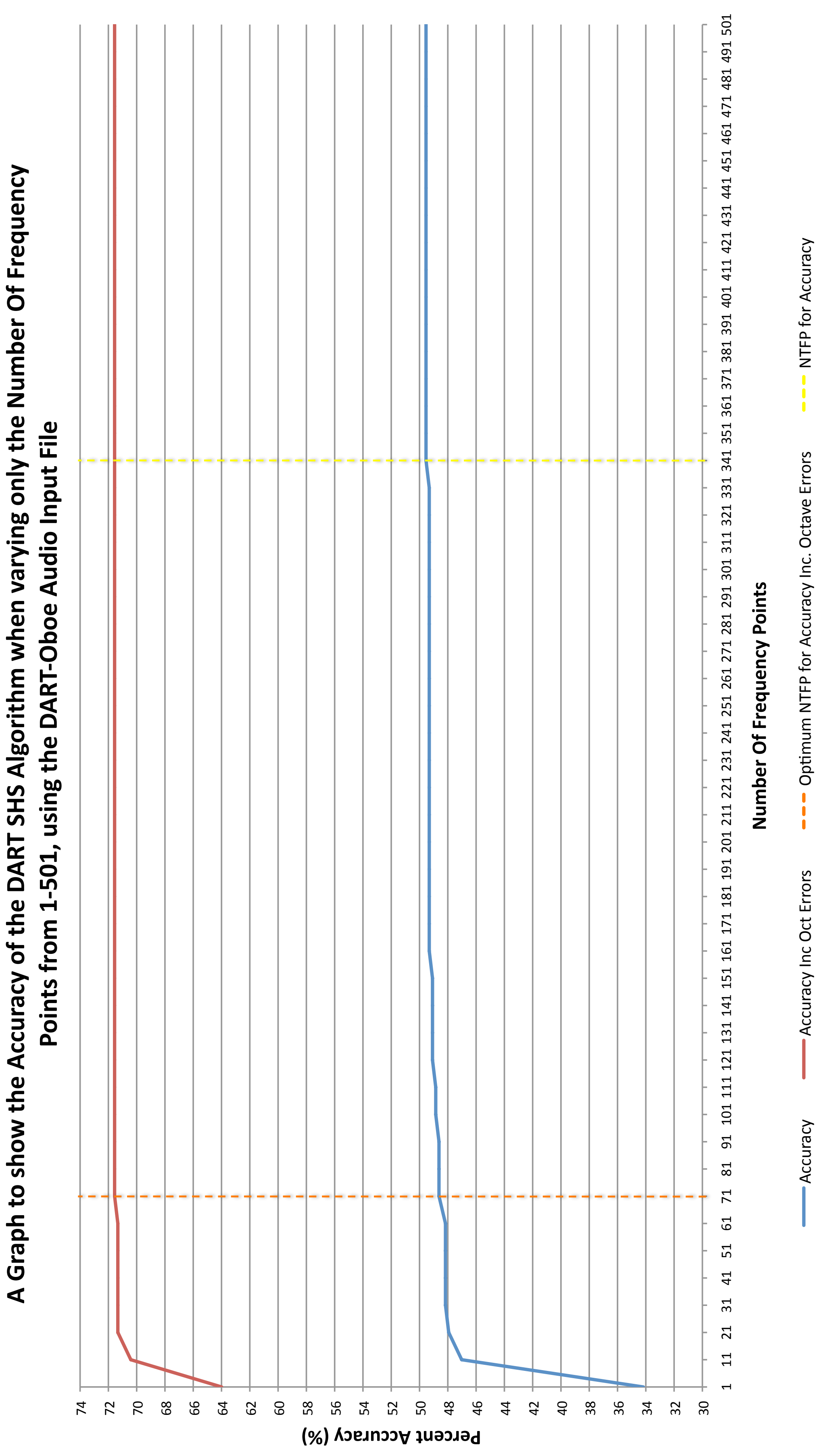


Figure 8.11: A graph showing the accuracy of the SHS algorithm when modifying the NTFP from 1-501 using the DART Oboe audio input file

8.6.3 Violin

Graph 8.12 shows the accuracy of the SHS algorithm when modifying only the NTFP analysed by the SHS algorithm from 1-501 in steps of 10, using the DART Violin audio input file.

The greatest accuracy is discovered when analysing a high NTFP value of **181** (lower than the previous experiment, however by less than 1% in accuracy) but with the much smaller NTFP value of **41** found for accuracy when including octave mistakes. Similar to the results of the oboe, the graph plateaus and does not begin to decrease in accuracy in a similar way to the acoustic guitar samples. However the general accuracy of the violin samples is much higher than the accuracy of the oboe, with an extremely high level of accuracy when allowing for octave errors. The results imply that the increase in accuracy found when NTFP value 41 to 181 is mainly reducing the number of octave errors.

The previous large scale experiments highlighted a NTFP value for the Violin of 23. However, *the results from this experiment seems to show a worthwhile benefit in a higher number of NTFP values (at around 7-8%).* For the violin audio input file, the optimal SHS parameters appear to consist of an NTFP value in the range of 181, or 41 if octave mistakes are acceptable, with the Rectangle FFT Window and 12 Harmonics.

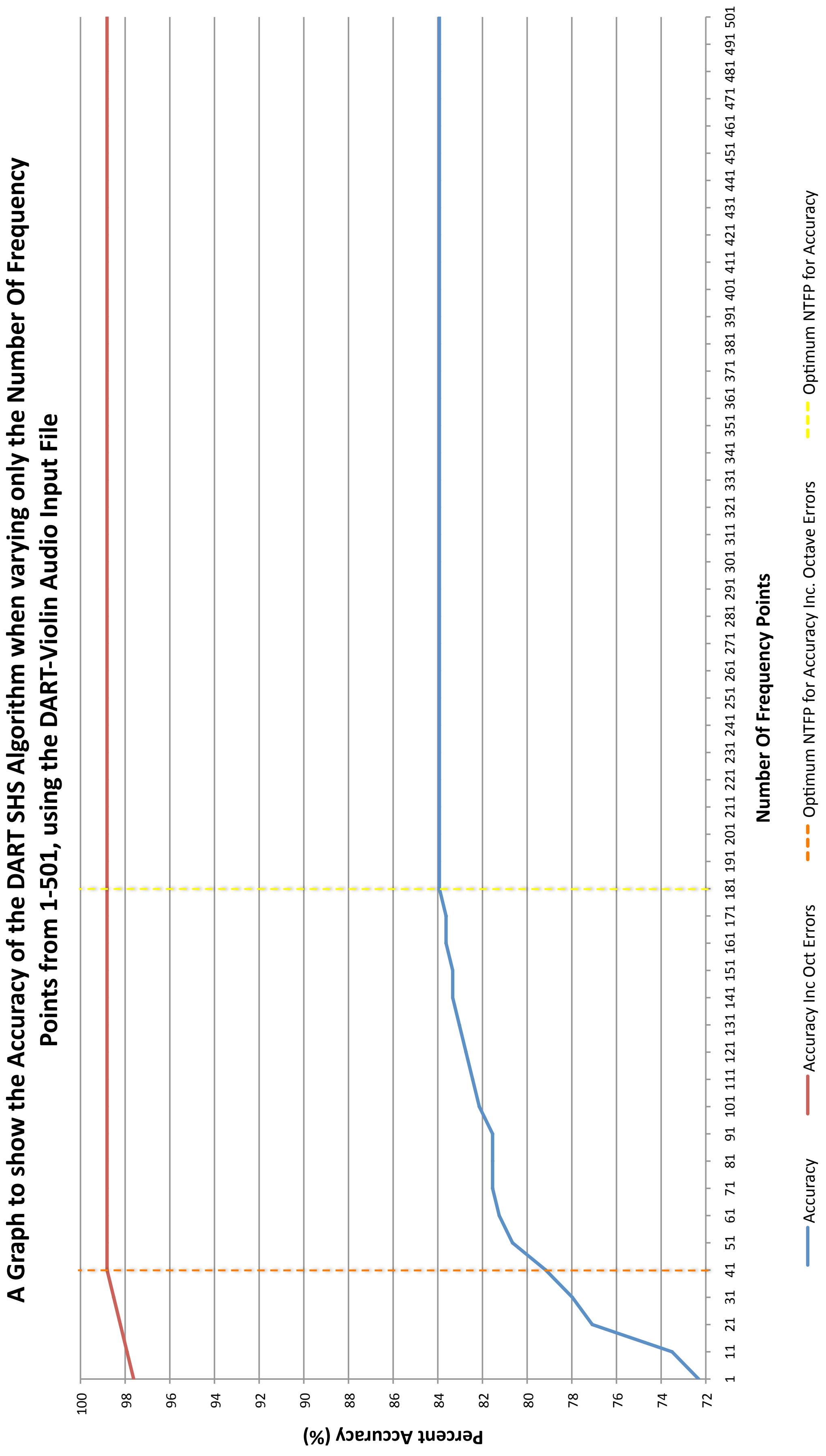


Figure 8.12: A graph showing the accuracy of the SHS algorithm when modifying the NTFP from 1-501 using the DART Oboe audio input file

8.6.4 Piano

Graph 8.13 shows the accuracy of the SHS algorithm when modifying only the Number Of Top Frequency Points (NTFP) analysed by the SHS algorithm from 1-501 in steps of 10, using the DART Piano audio input file.

The results of the Piano seem to resemble the general traits displayed by the Acoustic Guitar experiment, albeit with lower overall accuracy. The greatest accuracy is discovered when analysing a relatively low NTFP value of **31** (as in the previous experiment), and with an optimal NTFP value of **21** found for accuracy when including octave mistakes. These results show the steady decrease of accuracy after 31 points and support the results found in previous experiments.

Given the current evidence from both experiments it seems that the best results for the Piano audio input file can be found using a low NTFP of around 21-31, the Rectangle FFT Window type, and 12 harmonics.

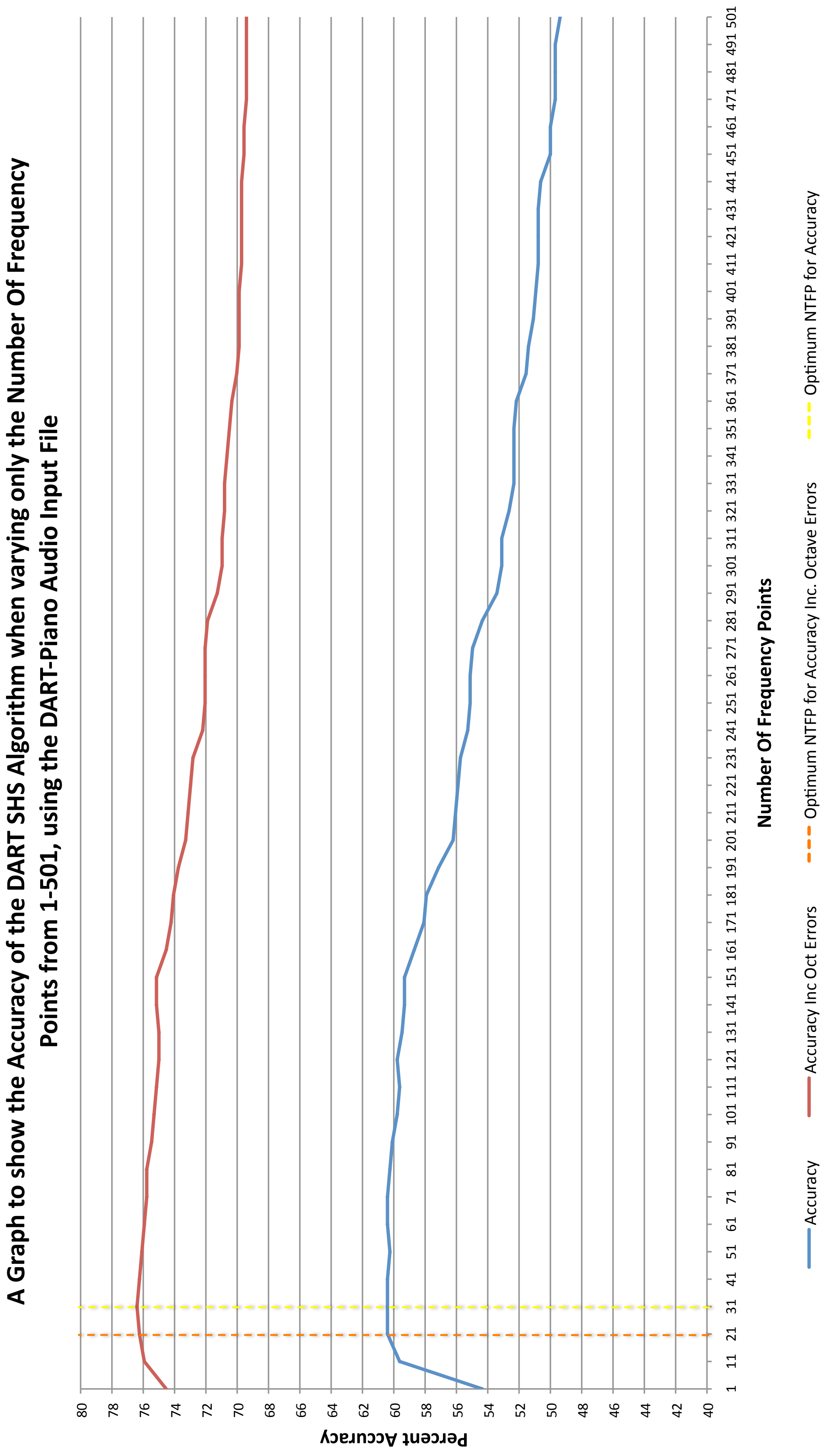


Figure 8.13: A graph showing the accuracy of the SHS algorithm when modifying the NTFP from 1-501 using the DART Piano audio input file

8.6.5 Tubular Bells

Graph 8.14 shows the accuracy of the SHS algorithm when modifying only the NTFP analysed by the SHS algorithm from 1-501 in steps of 10, using the DART Tubular Bells audio input file.

The greatest accuracy is discovered when analysing a high NTFP value of **261** but with the much smaller NTFP value of **11** found to be optimal for accuracy when including octave mistakes. Furthermore, the maximum accuracy when *not* including octave errors is extremely low at around 2%. However at a NTFP value of 11, the optimum NTFP including octave errors gives an accuracy of approximately 66%. Similar to the results of the oboe and violin previously, the graph plateaus and does not begin to decrease in accuracy in a similar way to the acoustic guitar or piano samples.

Given the current evidence from both experiments it seems that the most accurate results for the Tubular Bells audio input file can be found using a low NTFP of around 11 (possibly even as low as 7), the Rectangle FFT Window type, and 13 harmonics, with the caveat that these parameters will not be able to give the correct octave of the note, but only the note itself. If octave mistakes must be minimised at all costs, then a higher NTFP value of 261 could be considered, however this will only increase the chances of achieving the correct note and octave by a very small amount.

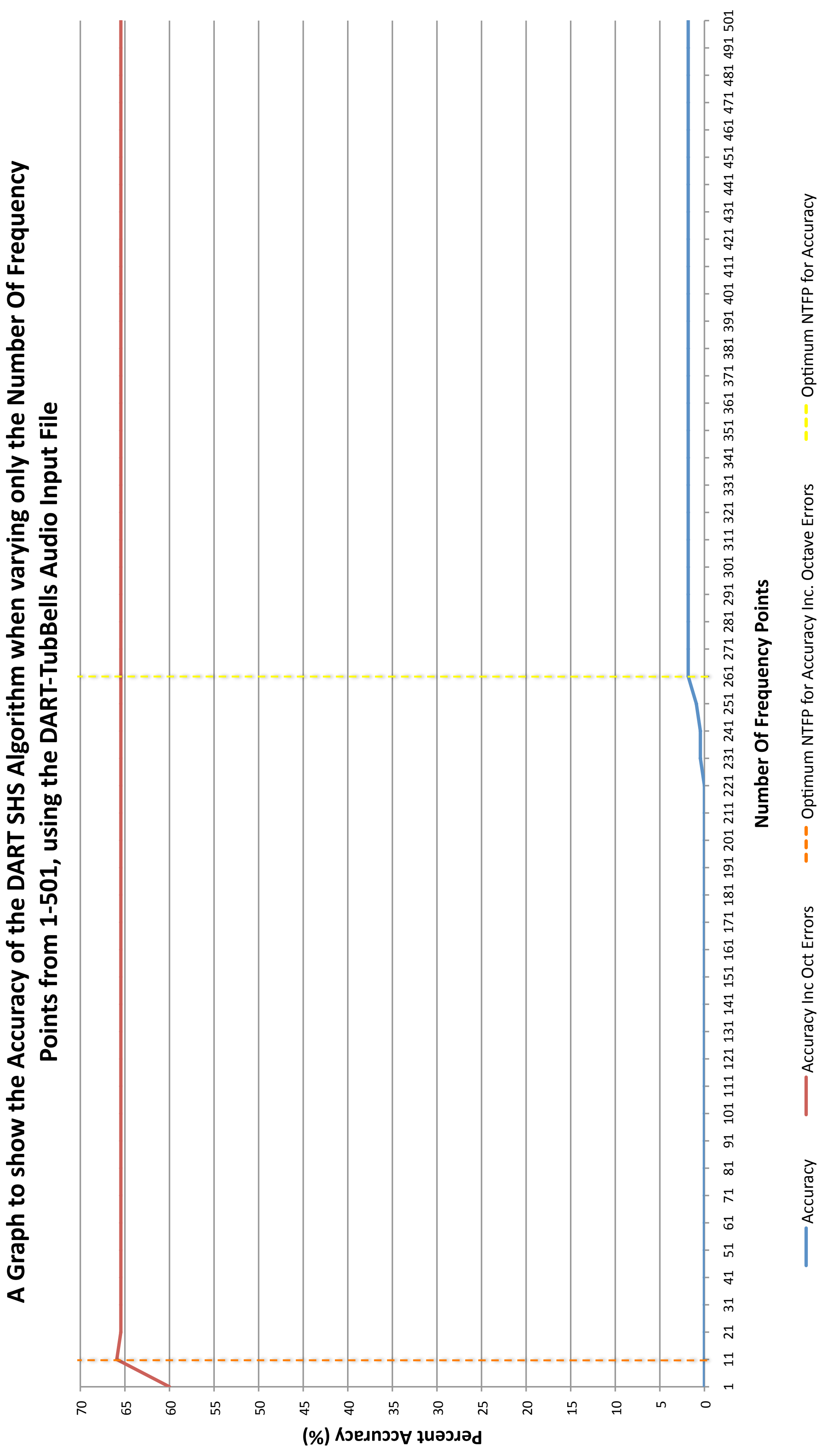


Figure 8.14: A graph showing the accuracy of the SHS algorithm when modifying the NTFP from 1-501 using the DART Tubular Bells audio input file

8.6.6 Distorted Guitar

Graph 8.15 shows the accuracy of the SHS algorithm when modifying only the NTFP from 1-501 in steps of 10, using the DART Distorted Guitar audio input file.

The greatest accuracy is discovered when analysing a high NTFP value of **321** (identical to the previous experiment), with the slightly smaller NTFP value of **281** found when including octave mistakes.

The original large scale experiment results for the Distorted Guitar presented in the previous chapter show a relatively steep increase in accuracy as the NTFP value rose to the maximum of 50. One of the main motivations for carrying out the extra NTFP experiments presented in this chapter was to see how much the accuracy would rise given a large enough value, and indeed if they would continue to rise. These extra tests show that the increase in accuracy becomes much more gradual after a NTFP value of 50.

Graph 8.15 shows that the greatest accuracy is discovered when analysing 321 frequency points - however *looking at more than 51 frequency points gives only a minor improvement in maximum or average (around 2%).* After the a NTFP value of around 341, the overall accuracy seems to begin to decrease.

It should also be noted that while the use of the Rectangle window was optimal overall, the large scale parameter sweep experiments performed in the previous chapter revealed that the Distorted Guitar audio file achieved higher accuracy rates using the Nutall4 FFT Window Type, with 15 harmonics.

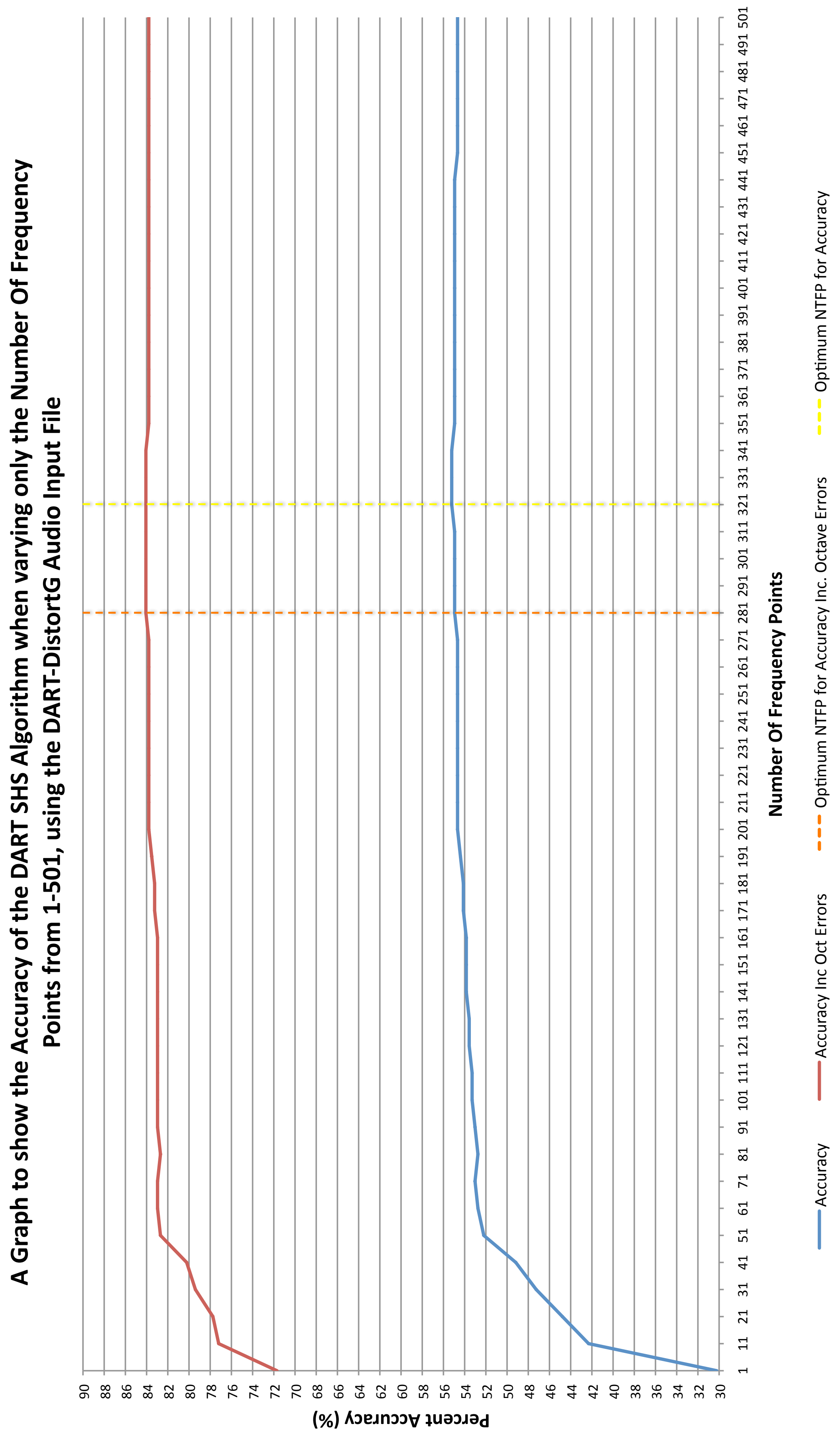


Figure 8.15: A graph showing the accuracy of the SHS algorithm when modifying the NTFP from 1-501 using the DART Distorted Guitar audio input file

8.6.7 Summary

Table 8.9 gives an overview of the results obtained from the further analysis of the SHS algorithm by increasing the range of NTFP values from 1 to 501 in increments of 10, showing both the optimum NTFP value, as well as optimum NTFP value when octave errors are allowed or included. The results from the second smaller Pegasus experiments are collated and presented in Figures 8.16 and 8.17 and display the accuracy results when excluding and including octave errors, respectively.

Contrary to early indications implied by previous experiments, the overall accuracy does not always continue to increase as the NTFP value increases. While the results in both Tables 8.9 and 8.16 appear to show that the optimal values often lie far beyond an NTFP value of 50, when visually analysing graphs of the results only relatively minor increases in accuracy are found after increasing over 51 points, around 2-4% at best, but generally far less. The Violin did show an increase of nearly 5% in accuracy from 51-181 points, however in the case of the Acoustic Guitar and Piano, once the optimum NTFP values of 21 and 31 respectively were found, analysing further NTFP values decreased the accuracy somewhat rapidly. Generally, increasing the NTFP value increases the computation time and therefore the lowest NTFP that can produce optimal or acceptable accuracy levels is preferred and when allowing for octave errors, a much lower NTFP value can be used.

Upon analysis of the frequency distribution around the harmonic progressions from a number of different pitches and instruments, the experiments empirically showed that the number of noisy peaks (the unrelated frequencies that do not contribute to this particular pitch determination) was several at worst. Therefore, investigating **50** peaks - an NTFP value of 50 - should allow for at least 10 harmonics to be considered for each pitch, which is generally considered to be enough for pitch classification.

When taking into account octave errors, a lower NTFP value of approximately 21 will allow for acceptable results in all cases except for the Distorted Guitar, which shows approximately a 10% increase from an NTFP value of 21 to the optimum of 281. This is likely due to the high level of odd and even harmonics which can effectively add noise to the signal, requiring more frequency points to be analysed.

The use of Pegasus and Triana enabled the further investigation, analysis and optimisation of the implemented of the DART Sub-Harmonic Summation pitch detection algorithm, and showed that DART experiments can be refined across multiple platforms.

Audio Input File	***Optimum NTFP Inc O.E***	***Optimum NTFP***
Acoustic Guitar	11	21
Oboe	71	341
Violin	41	181
Piano	21	31
Tubular Bells	11	261
Distorted Guitar	281	321

Table 8.9: This table shows a summary of the Number Of Top Frequency Points that give the most accurate results for each Audio Input File.

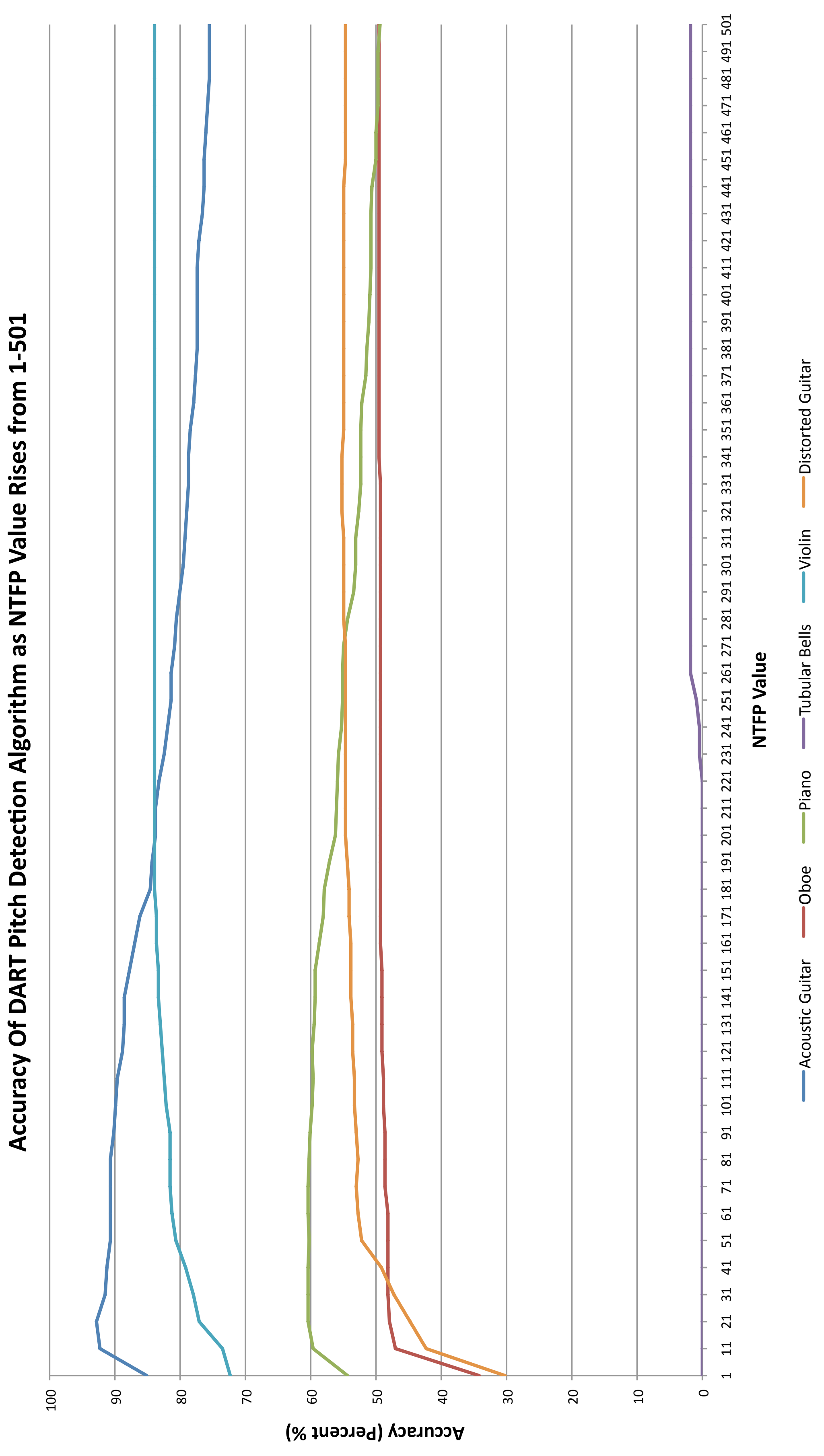


Figure 8.16: A graph to show the accuracy of the DART SHS algorithm when varying only the NTFP value from 1-501 in steps of 10

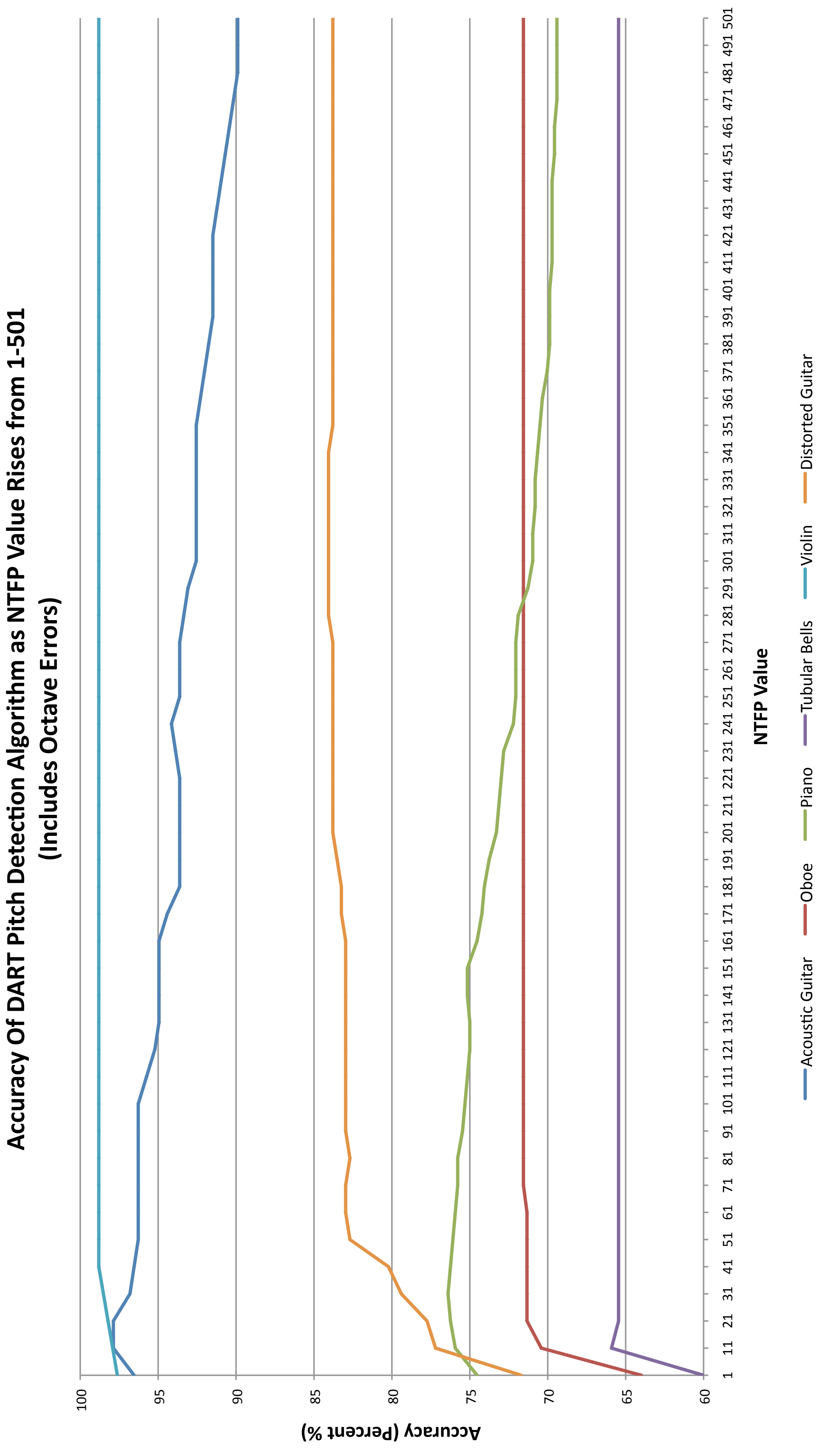


Figure 8.17: A graph to show the accuracy of the DART SHS algorithm when varying only the NTFP value from 1-501 in steps of 10, and allowing for Octave Errors

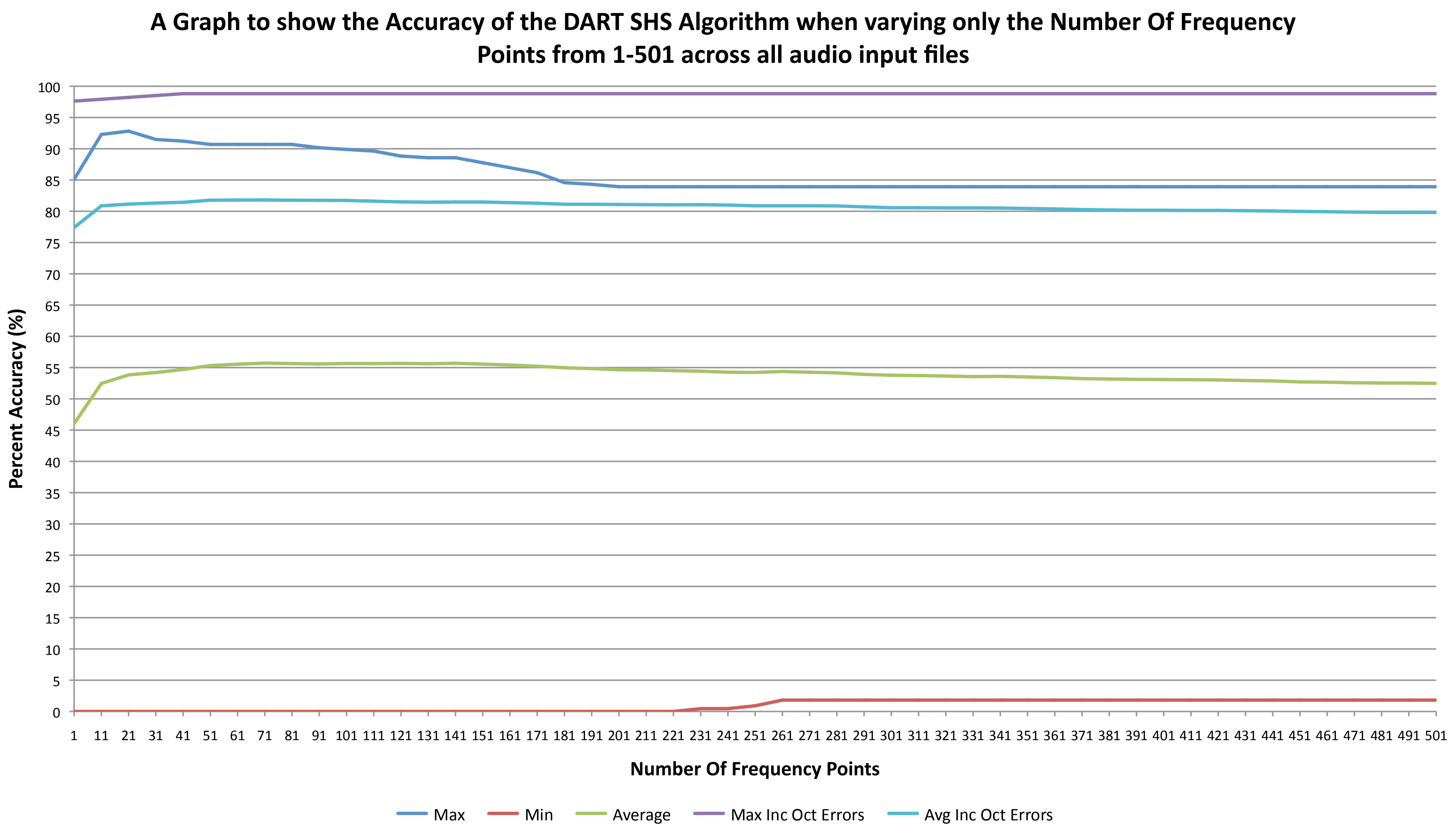


Figure 8.18: A graph showing the accuracy of the SHS algorithm when modifying the NTFP from 1-501 across all six DART audio input files

Chapter 9

Conclusion

A distributed MIR platform that encompasses the creation of MIR algorithms and workflows, their distribution, results collection and analysis, is presented in this thesis. The framework, called *DART* - Distributed Audio Retrieval using Triana - is a distributed audio processing platform that is designed to facilitate the submission of MIR algorithms and computational tasks against either remotely held music and audio content, or audio provided and distributed by the MIR researcher.

DART and the Sub-Harmonic Summation experiments were created in order to test the hypothesis set in Chapter 1, that; through the use of modern distributed computing techniques, it is possible to design an MIR system that is scalable as the number of participants increases, adheres to copyright laws and restrictions, whilst at the same time enabling access to a global database of music for MIR applications and research.

The experiments and research presented support the hypothesis and show that a scalable[1] distributed MIR platform can be created and utilised to perform and aid in real-world, scientific/MIR experiments by using *modern distributed computing technologies* such as BOINC (a volunteer and grid computing middleware), XtremWeb (a desktop grid middleware capable of bridging and using larger Grid infrastructures such as EGI), and Pegasus (capable of mapping and executing application workflows over a wide range

[1]As highlighted in Chapter 1,'scalable' in the context of DART can be considered as an improvement of performance (i.e. faster overall job execution times) as the as number of participants increase. When more workers are introduced to the system with no resulting performance benefit, then the system is no longer scalable.

of distributed resources, ranging from a single laptop to a campus cluster, a Grid, or a cloud-based system).

DART is a platform designed to be used by the scientific and MIR research communities, as a testbed for further research and investigation. A primary concept in developing DART was that its scope should only be limited by the imagination of the MIR researcher designing the Triana workflows - and for those workflows to be as created as easily as possible. The use of the Triana graphical workflow software allows for researchers who are not proficient in software programming to get up and running. The Audio Toolkit outlined in Chapter 2, created by the author, contains many Triana *Units* that an MIR researcher can utilise straight away to construct MIR algorithms/workflows - as well as the hundreds of Math and Signal Processing Units made available by utilising the data type conversion Units (in the Audio Toolkit).

In order to aid in the distribution of MIR algorithms, the DART Execution Environment (DEE) (the architecture of which is outlined in Chapter 3, with the design outlined in Chapter 4) was created to take Task Graphs (workflows or algorithms) created in Triana, and enable them to be packaged into a lightweight Java JAR that can be distributed using a variety of distribution platforms. The process of 'streamlining' a Triana workflow to the DEE highlights the key Triana classes and component structure that must remain intact in order to successfully run Triana workflows outside of Triana, with the ultimate goal of the process becoming automated in the near future (highlighted in Section 9.1).

The successful execution of large scale, 'proof-of-concept' parameter sweep experiments in order to find the optimum parameters of the Sub-Harmonic Summation pitch detection algorithm not only provides DART with a test application to investigate if large scale experiments are possible using the platform, but the results of the parameter sweep experiments presented in Chapter 7 reveal the optimal parameters for the implemented SHS pitch detection algorithm across a range of input data. Further investigation and verification of the implementation of the SHS algorithm (perhaps from experts in the field of PDAs) would be useful in order to claim definite contributions to the field of PDAs. However, the research presented here and the opportunity to further harness the power of the DART platform in order to continue to study and optimise the SHS (or any pitch detection) algorithm - as well the option to create and utilise different input data - is one which contributes towards building a more robust results set and conclusion. As a result of research presented in 7.3.8, the results of the optimal FFT window parameter would be more conclusive if the FFT window size was not set to the same size as the Triana chunk size, which gives the impression that the Rectangular window (i.e. no windowing function) produces the most accurate results. If each note were 1 second long and the chunk of audio was still 0.5 seconds, spectral leakage would have played a bigger role and created more need for the other FFT Window Types to be used. The advantage of creating a distributed MIR platform, of course, is that re-running the tests with different parameters or a modified SHS algorithm does not require any further investment or resources - the new algorithm can easily converted to run in the standalone DEE environment and distributed to the workers.

Further contributions include the design of a highly scalable peer-to-peer prototype DART architecture, as outlined in Chapter 3 and published in [15], [16], and [17], supported by multiple simulations performed at the Institute of High Performance Computing and Networking, Italy. This thesis also investigates and compares the suitability of two open-source middleware platforms that are designed for distributed computing using GRID or volunteered computing resources, for the application of distributed (audio) processing (BOINC & XtremWeb). Both platforms allowed for a massive reduction in overall processing time, as documented and evaluated in Chapter 7.

Chapter 8 and the SHS parameter sweep experiment concept shows that the DART platform allows for experiments, ideas, and research to be refined and revised, completely independently of the distribution platform used. The work with the Pegasus team has contributed key results for [18] (*A Case Study into Interoperable Monitoring and Analysis for Scientific Workflows*), published in the Journal of Grid Computing. The work with the XtremWeb team has also pushed forward development and highlighted many issues with the XtremWeb platform. As documented in Chapter 7 the DART experiments were by far the largest scale experiments run on an XtremWeb platform. These experiments highlighted problems which were addressed by the XtremWeb team and made the software much more scalable, reliable and useable for large scale distributed computing.

The DART research perhaps does not yet show 'first hand' that a DART researcher has access to a global database, however the work presented shows considerable evidence to help achieve this goal; the potential for the creation of (for example) a Music Recommendation System which offers everyday music 'users' an incentive to join the project, while allowing scientific research to take place, would open up vast resources to MIR scientists, without the worry of law and copyright issues. MIR experiments can differ in their input data requirements; some may be able to utilise any input audio data to achieve their analysis goals, while others may require specific input data - either provided by the researcher - or the data can still already exist on the worker machine (with the DART workflow only downloaded if the required audio is present on the worker machine). In any case, given a distributed implementation of an MIR research platform (i.e. DART), the files would be analysed on the worker machine, with only the metadata/results of the analysis returned to the DART researcher the overall computation time will be reduced, indirectly solving the issues of legality and copyright highlighted in the hypothesis. This also helps to avoid bandwidth and resource restrictions, and potentially allows for more refined MIR algorithms, benefitting the MIR and scientific community as a whole.

The creation of the Music Recommendation System (MRS) initially described in this thesis would require collaboration and coordination with a number of other researchers as the task is non-trivial. Whether DART becomes an MRS or remains a platform for general MIR analysis - with additional research and development and the participation of the scientific/MIR community - DART can help further develop the state of the art of a complex, inter-disciplinary field. The next section looks at the 'next step' in advancing DART.

9.1 Future Work

DART is still in the early stages of development and requires further participation from the scientific and specifically the MIR communities in order to incorporate more features that would be useful for MIR research and push forward innovation in the field. New Triana Units should be developed to cover a wide range of common MIR processes that scientists and researchers would require in order to encourage adoption of DART as a standard MIR research platform. An excellent addition to DART that would encourage adoption and simplify development would be for the automated conversion of a Triana workflow to the DART Execution Environment. If a user could package a Triana workflow into a Java JAR directly from Triana, the usability of the entire Triana/DART platform would be improved dramatically. This is possible and is a logical 'next step' for DART; this feature is currently being implemented by the Triana team, based on the research presented in this thesis.

The integration of DART with distribution platforms such as BOINC or XtremWeb proved to be the most time consuming aspect of the thesis. XtremWeb is maturing and becoming even more scalable, reliable and proves itself to be a useful option to DART users. XtremWeb merely distributes the application and data and does not require extensive modification of the application. While the XtremWeb to EGEE bridge was not functional for the large scale parameter sweep experiments, XtremWeb provides a solution for Desktop Grid environments and would bridge easily to other, functional large scale computing platforms that may exist in the future. DART users have the option of utilising a volunteer based distribution platform in BOINC. BOINCOID[2], is an interesting recent development; a Java port of BOINC. While reportedly not at the final release stage, a Java BOINC implementation could remove the need for any type of application wrapper, and working natively in Java from the beginning of the Triana workflow, to the DART Execution Environment and finally to the distribution platform, would simplify the development processes considerably.

While not implemented in time to be used as part of the experiments carried out in this thesis, the adoption of ADICS would allow for the data distribution to become fully peer-to-peer - a step closer to the proposed DART architecture outlined in Chapter 3. Although both BOINC and XtremWeb used caching schemes to minimise file transfer and bandwidth use (and as such download times were not particularly costly bottlenecks), future DART experiments may be more data intensive and benefit massively from a P2P data distribution mechanism.

Apache Hadoop[3] is another distribution platform that DART could integrate well with. Hadoop is a software framework - written in Java - that supports data intensive distributed computing applications, under a free licence. It enables applications to work across thousands of machines, with petabytes of data. Hadoop represents a decentralised database system; the data can be spread across many machines, each with their own memory or disks, with no one central data server - Hadoop keeps track of where all the data is stored and integrates redundancy mechanisms in order to ensure that no data is lost if one of the

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

[3]http://hadoop.apache.org

machines is removed. There are numerous advantages to a decentralised approach and as DART ultimately aims to be used to analyse audio files that are stored locally on volunteer or worker machines, Hadoop could potentially be utilised to *keep track* of the data that is currently in the system.

The wide scale DART SHS experiments could also have been improved. Using 1 second audio chunks instead of 0.5 second chunks would allow notes to ring out longer, giving the algorithm more chance to analyse proportionally less of the initial transient, which can contain a large number of overtones. It would have also been beneficial to include a seventh audio file containing a pure sine wave. While tests of the SHS algorithm were carried out with sine waves, including this as a seventh audio file would have tested the pure accuracy of the SHS algorithm and found the optimal parameters for a 'perfect' signal. This would have allowed for further analysis of the accuracy of the implemented SHS algorithm.

An extra set of experiments to look into the effect of different velocities on the accuracy of the results would have been interesting to incorporate into these DART experiments. This can be done without running the large scale experiments again, but by writing an application to analyse the resulting datasets as the number of notes of each velocity is already known.

Utilising 6 WAV files as audio input data was useful in testing potential distribution platforms such as BOINC and XtremWeb, making sure that any relatively 'data heavy' DART experiments could be distributed on these platforms. Furthermore, WAV files are lossless; the integrity of the audio data remains intact and the results of the experiments are not effected by any compression artefacts. However, the MP3 units created as part of DART (and MP3 input audio data) could be utilised both to lessen the bandwidth requirements of an experiment such as the SHS parameter sweep documented in this thesis, but also opens up the possibility of analysing audio files that already exist on the worker's computer. As explained in the Chapter 1, a DART Music Recommendation System would rely on audio files that already exist on a participants computers, negating problems with copyright issues or any legal restrictions.

Appendix A

Appendix

A.1 Basic Coding Structure Of Triana Units

The Triana Unit Wizard (TUW - discussed in the background chapter) is commonly used to create the framework for new units in Triana and DART. The TUW creates functional and reliable GUIs which are sufficient for the purposes of DART prototyping; the main DART application has a command line interface and so a GUI (custom or otherwise) is not necessary.

Each Triana unit contains a **`process()`** method, called by the **`Unit`** class when a Task Graph is executed. This allows the unit developer to simply write code that processes the input data and outputs the results, without having to worry about the implementation of any other aspect of the Triana workflow - as long as the input and output types for each unit are compatible.

Some Triana units (many of the Audio toolkit units, for example) consist of two classes - such as **`MyUnit`** and **`MyUnitProcessing`**. The **`MyUnit`** class would contain the **`process()`** method created by the Unit Wizard. When appropriate, the **`MyUnitProcessing`** class contains a process method that actually contains the algorithm which carries out the function of the unit. This approach allows for programmers to call on the **`process(short input[])`** method in the **`MyUnitProcessing`** class, which returns the manipulated array without forcing the array to be set to a channel and output as a **`multipleAudio`** Object. This is useful to programmers who are coding Triana units which perform many functions - if one of the methods of the unit is useful during implementation and they do not wish to get involved in grouping units, the programmer can simply call on the **`MyUnitProcessing.process()`** method when creating their others unit. All **`process()`** methods in the **`MyUnitProcessing`** classes can be overloaded in order to accept more data types.

When a unit is created, an **`input`** and/or **`output`** Object - for example a **`multipleAudio`** input type - can be instantiated automatically by setting the appropriate data types when using the Triana Unit Wizard. This creates the following code:

```
MultipleAudio input = (MultipleAudio) getInputAtNode(0);
MultipleAudio output = new MultipleAudio(input.getChannels());
```

The input **`multipleAudio`** object is set using the **`Object getInputAtNode(int nodeNumber)`** method from the Unit superclass, which returns the data at a specific input node in the Triana GUI. Another **`multipleAudio`** object can be created to output the manipulated or processed data. The new **`multipleAudio`** object called **`output`** is given the same number of channels as the **`input`** object using the **`getChannels()`** method from the **`MultipleChannel`** data type class. It may be the case that the output of the unit needs to be of a different data type than the input, such is the case in the case of the FFT unit. If the unit accepts other input types, then these can also be created accordingly.

With **`multipleAudio`** data, it is not possible to manipulate the **`multipleAudio`** object directly - the data must first be converted into an array. In the case of this example, one-dimensional arrays are created in order to represent 16-Bit audio, where each value in the array constitutes a sample of audio data and holds a value that corresponds to the sample's amplitude level. If the data is 16-Bit, this corresponds to a maximum amplitude level of 32767 and a minimum level of -32768 (see[1]). If the data has a sampling frequency of 44.1KHz this indicates that 44100 samples are read from the array per second. To convert the **`multipleAudio`** object into an array, the **`multipleAudio`** must be considered a channel at a time (if there is more than one, two channels are used for Stereo audio, one for Mono). An object (called **`in`** in the following example) must be created and assigned to the value of input for the particular channel.

For each channel in the **`multipleAudio`** object, an output **`Short`** array is created -

[1]As stated in the Design chapter, all audio units are designed to work with 16-Bit data.

this is to hold the manipulated data and insert it into the **`multipleAudio`** object called **`output`**. Continuing with this example, a **`for`** loop is used to inspect the audio one channel at a time:

```
Object in;
  for (int i=0; i<input.getChannels(); ++i) {
    in = input.getChannel(i);
      short[] out;
```

To check if the data is indeed 16-Bit, the class can use the **`in instanceof short[]`** (short data types represent 16-Bit data). If the data passes this test, then the **`in`** object can be converted to a short array called **`temp`**: **`short[] temp = (short[])in;`**. The **`out`** array is assigned to the result of the value returned from the **`process()`** method in the **`MyUnitProcessing`** class (which would have been instantiated earlier on in the method):

```
out = fader.process(temp);
```

The **`setChannel(int channelNo, short[] data, ChannelFormat format)`** method from the **`MultipleChannel`** class can be used to add the manipulated channel data into the output **`mutipleAudio`** object.

Finally, only after all channels have been set to the output **`MultipleAudio`** object, and the **`for`** loop completed, can the data then be output from the unit using the **`output(Object)`** method, from the **`Unit`** superclass. This method outputs the data across all nodes, passes the given data set to the first output node and then makes copies for any other output nodes, blocking until the data is successfully sent. This method allows the user to increase the number of output nodes as they desire, and the data will be sent equally from each output node.

Missing from the above example is the processing algorithm that occurs after data input and before the unit outputs data to the next unit. The implementation of all the DART Triana units is now presented.

A.2 Finding Optimal SHS Parameters

A Java framework was created to allow for the analysis of the effect of each parameter on the resulting accuracy of the SHS algorithm. 5 applications were created to analyse the results with each parameter in mind, using the framework:

- **AnalyseAudioFiles.java**
- **AnalyseFFTWindow.java**
- **AnalyseNoHarmonics.java**
- **AnalyseNoFreqPoints.java**
- **AnalyseTime.java**

Each application goes through the directory of 268,800 results (text) files (or a subset of these files based on specific audio input files), and analyses each results file, giving it an accuracy score, measured out of 100%. The code also calculates the minimum, maximum, and average accuracies when taking into account octave mistakes, and then automatically creates a CSV file containing the results in an easily graph-able format. This allows the results to be easily charted and displayed using spreadsheet software.

The code examples presented in this section are from the **`AnalyseNoHarmonics.java`** class, however the 5 classes are very similar in nature and the main differences in implementation are highlighted.

Each instrument's *correct* note range was stored as a static **`String[]`**, allowing the post analysis code to reference the correct notes and check against the current results file. An (shortened) example is given below for the Acoustic Guitar samples:

```
static String[] correctAcousticGArray = {"E1", "F1", "F#1", "G1", "G#1", "A1
    ", "A#1", "B1", "C2", "C#2", "D2", "D#2", ---> "G4", "G#4", "A4", "A#4",
     "B4", "C5", "C#5", "D5"};
```

The analysis code framework begins by creating the results output CSV file and locates the correct folder containing all 268,800 text results in the following manner:

```
public static void main(String[] args) throws IOException {


  long time = System.currentTimeMillis();
```

```
    File thisfolder = new File(home + "/project/DartRoot/results/");
    File[] listOfFiles = thisfolder.listFiles();

    file = new File("DART-ALL-NoHarmonicsCSV");
    boolean exists = file.exists();
    if (!exists) {
      String myNewDir = "results";
      new File(myNewDir).mkdirs();
      out = new BufferedWriter(new FileWriter(file));
    }
    else{
      out = new BufferedWriter(new FileWriter(file));
    }

    out.write("NoHarmonics, Max, Min, Average, Max Inc Oct Errors, Avg Inc Oct
         Errors, Top + Error, Max Oct Error, Min Oct Error, Avg Oct Error");
    out.newLine();
```

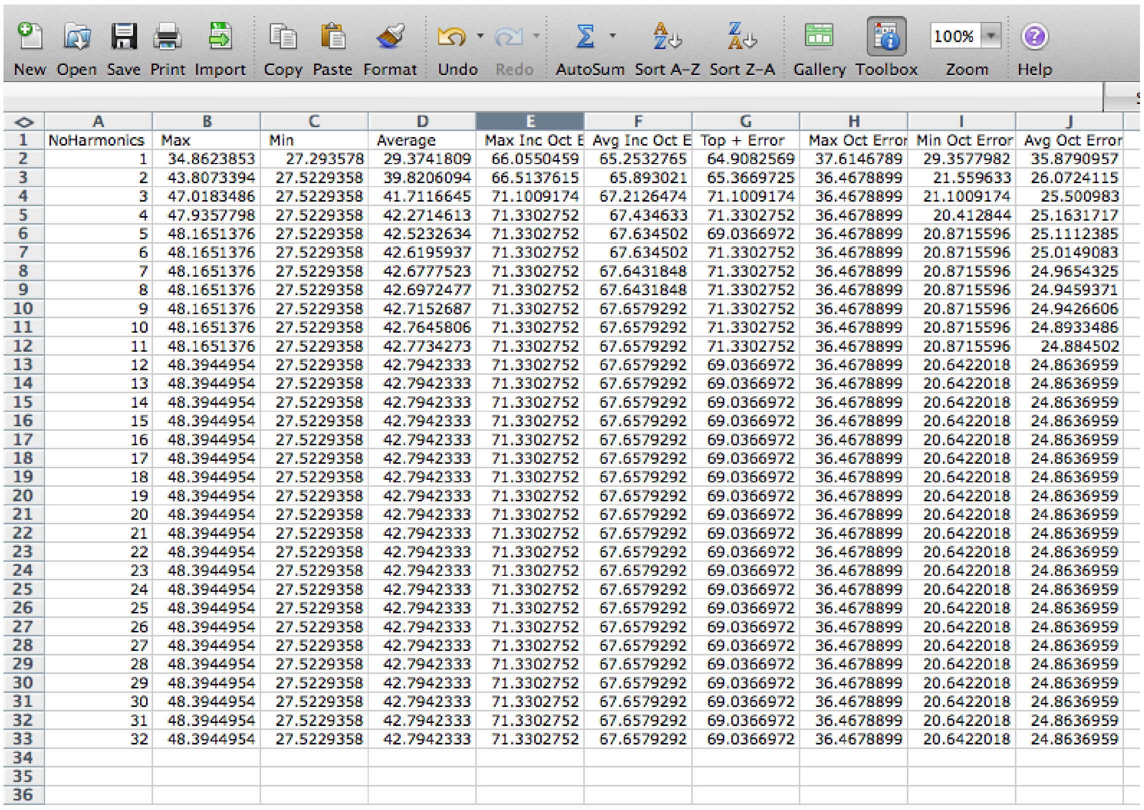
At the top of the generated CSV file the headings of a table are created and each row can then be populated, giving the results for each Number of Harmonics. In the case of *NoHarmonics*, there will be 32 rows, each giving the Minimum, Maximum, Average (and so on), values for each Number Of Harmonics, as demonstrated in Figure A.1.

The Min, Max and Average columns give the Minimum, Maximum, and Average accuracy values for the particular Number of Harmonics analysed. This particular code and image example gives the first column the name **`NoHarmonics`**, however the other classes report the **`FileNo`**, **`FFTWindow`**, or **`NoFreqPoints`**, respectively.

A Vector called **`allresultsvector`** is created to hold the results of the post-analysis. In total, the **`allresultsVector`** vector will contain all the 268,800 **`String`** arrays and their accuracy scores, both with and without octave errors. A **`for`** loop goes through each results text file in the results directory:

```
  Vector allresultsvector = new Vector();

  for (int i = 0; i < listOfFiles.length; ++i) {
    if (listOfFiles[i].isFile()) {
      String files = listOfFiles[i].getName();
      if (files.startsWith("DART-")){
```



	A	B	C	D	E	F	G	H	I	J
1	NoHarmonics	Max	Min	Average	Max Inc Oct E	Avg Inc Oct E	Top + Error	Max Oct Error	Min Oct Error	Avg Oct Error
2	1	34.8623853	27.293578	29.3741809	66.0550459	65.2532765	64.9082569	37.6146789	29.3577982	35.8790957
3	2	43.8073394	27.5229358	39.8206094	66.5137615	65.893021	65.3669725	36.4678899	21.559633	26.0724115
4	3	47.0183486	27.5229358	41.7116645	71.1009174	67.2126474	71.1009174	36.4678899	21.1009174	25.500983
5	4	47.9357798	27.5229358	42.2714613	71.3302752	67.434633	71.3302752	36.4678899	20.412844	25.1631717
6	5	48.1651376	27.5229358	42.5232634	71.3302752	67.634502	69.0366972	36.4678899	20.8715596	25.1112385
7	6	48.1651376	27.5229358	42.6195937	71.3302752	67.634502	71.3302752	36.4678899	20.8715596	25.0149083
8	7	48.1651376	27.5229358	42.6777523	71.3302752	67.6431848	71.3302752	36.4678899	20.8715596	24.9654325
9	8	48.1651376	27.5229358	42.6972477	71.3302752	67.6431848	71.3302752	36.4678899	20.8715596	24.9459371
10	9	48.1651376	27.5229358	42.7152687	71.3302752	67.6579292	71.3302752	36.4678899	20.8715596	24.9426606
11	10	48.1651376	27.5229358	42.7645806	71.3302752	67.6579292	71.3302752	36.4678899	20.8715596	24.8933486
12	11	48.1651376	27.5229358	42.7734273	71.3302752	67.6579292	71.3302752	36.4678899	20.8715596	24.884502
13	12	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
14	13	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
15	14	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
16	15	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
17	16	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
18	17	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
19	18	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
20	19	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
21	20	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
22	21	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
23	22	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
24	23	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
25	24	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
26	25	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
27	26	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
28	27	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
29	28	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
30	29	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
31	30	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
32	31	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
33	32	48.3944954	27.5229358	42.7942333	71.3302752	67.6579292	69.0366972	36.4678899	20.6422018	24.8636959
34										
35										
36										

Figure A.1: The resulting CSV file generated after running the AnalyseNoHarmonics.java program on one or more audio files.

The filename for each result file plays a key part in telling the program which parameters were used in the SHS analysis. Changing the filename from **`DART-`** to **`DART-`*`n`*** (where n represents a number from 1-6) allows for the analysis of a subset of the results files, based on a particular audio input file; when examining a results file with a filename that begins with DART-1, the application will only be analysing the results of the analysis of the Acoustic Guitar audio input files. This allows for the analysis of the optimum Number Of Harmonics for only the Acoustic Guitar, for example, or across all instruments.

The following code section goes through each note in the results file, and calls the **`findNoteMap`** method, which is taken from the pitch detection algorithm. It would have been possible to simply use the already mapped notes in the results file (the integer note values have already been mapped to a pitch, i.e. 440 has been mapped to A4 in the results file), however the **`findNoteMap()`** method was slightly easier to call and had already been implemented.

The aim of the main body of code is to check if the note in the results file matches the with the correct string array for the particular audio input file that has been analysed.

The code begins by checking if the note AND the octave are both correct. If the note is a perfect match with the note in the correct **`String`** array, then the **`resultsscore`** integer is incremented. As the number of total notes in the file is already known, it is then possible to calculate the overall accuracy of that particular results file.

```
compare = findNoteMap(baseArray);
int resultscore = 0;
int octaveerrorscore = 0;
for (int j = 0; j < (currentArrayLength ); ++j) {
  if (files.startsWith("DART-1")) {
    if (compare[j] == correctAcousticGArray[j]) {
      resultscore++;
    }
```

If however, the *note* is correct but the *octave* is incorrect, then the **`octaveerrorscore`** integer is incremented. This must take into account notes that are both one or two characters long (such as A and $A\#$), as well as the following integer to show the octave value.

```
else if (compare[j].length() == 2){
  if (compare[j].charAt(0) == correctAcousticGArray[j].charAt(0)) {
    if (compare[j].charAt(1) != correctAcousticGArray[j].charAt(1)) {
      octaveerrorscore++;
    }
  }
}
else if (compare[j].length() == 3){
  if (compare[j].substring(0,2).contentEquals(correctAcousticGArray[j].
      substring(0,2)) ) {
    if (compare[j].charAt(2) != correctAcousticGArray[j].charAt(2)) {
      octaveerrorscore++;
    }
  }
}
```

This code is repeated 6 times, once for each audio input file type and file name, such as **`DART-1-`** (shown above), **`DART-2-`**, **`DART-3-`**, and so on. The total accuracy with and without octave errors is then calculated and added to the **`allresultsvector`** Vector.

```
double score = 100 * ((double) resultscore / (double) compare.length);
double octaveerror = 100 * ((double) octaveerrorscore / (double) compare.
    length);
Object[] resultsandnamearray = new Object[] {files, score, octaveerror};
allresultsvector.add(resultsandnamearray);
```

In total, **`allresultsVector`** contains all the 268,800 string arrays and their accuracy, both with and without octave errors. The **`allresultsVector`** can now be analysed to reveal which SHS parameters yield the best results.

Two nested **`for`** loops work to calculate the the minimum and maximum values in the **`Vector`**. The **`for`** loop iterates from 1 to 33, as there are 32 harmonics to analyse. This is changed to 1-51 for the NTFP (50 values) in the **`AnalyseNoFreqPoints.java`** file, to 1-29 (28 values) for the FFT Window Type in **`AnalyseFFTWindow.java`** file, and from 1-7 (for the 6 audio file types) in the **`AnalyseAudioFiles.java`** file.

```
for (int a = 1; a < 33; ++a){

  Object[] vectorelement = null;
  double max = 0;
  double maxoctaveerror = 0;
  double min = 100;
  double minoctaveerror = 100;
  double sum = 0;
  double sum2 = 0;
  double sum3 = 0;
  double overallmax = 0;

  int noofwanted = 0;

  for (int i = 0; i < (allresultsvector.size()); ++i) { // go through each
      element in vector
    vectorelement = (Object[]) (allresultsvector.get(i));

    String[] filenamesplit = vectorelement[0].toString().split("-");
    String harmonicnumber = filenamesplit[3]; // set no harmonics;

    if (Integer.parseInt(harmonicnumber) == a){
      ++noofwanted;
      if (parseDouble(vectorelement[1].toString()) > max){
          max = parseDouble(vectorelement[1].toString());
        }
        if (parseDouble(vectorelement[1].toString()) < min){
          min = parseDouble(vectorelement[1].toString());
        }
        if (parseDouble(vectorelement[2].toString()) > maxoctaveerror){
          maxoctaveerror = parseDouble(vectorelement[2].toString());
        }
        if (parseDouble(vectorelement[2].toString()) < minoctaveerror){
          minoctaveerror = parseDouble(vectorelement[2].toString());
        }
        if (parseDouble(vectorelement[1].toString()) + parseDouble(
            vectorelement[2].toString()) > overallmax){
          overallmax = (parseDouble(vectorelement[1].toString()) + parseDouble
              (vectorelement[2].toString()));
        }
        sum = sum + parseDouble(vectorelement[1].toString());
        sum2 = sum2 + parseDouble(vectorelement[2].toString());
        sum3 = sum3 + (parseDouble(vectorelement[1].toString()) + parseDouble(
            vectorelement[2].toString()));
      }
              }

    double averagescore = sum / noofwanted;
    double averageoctaveerror = sum2 / noofwanted;
    double averagescorewithoctaveerrors = sum3 / noofwanted;
    double maxresultpluserror = 0;
  }
```

The nested **`for`** loop iterates through each element in the **`allresultsvector`** Vector in order to search for the minimum and maximum result values for each Number of Harmonics. The average scores can then be calculated after the **`for`** loop completes.

In the nested **`for`** loop the Number Of Harmonics is again ascertained by splitting the filename according to the hyphens in the filename itself. This allows the **`for`** loop to check only the values of the particular Number of Harmonics.

After this pair of **`for`** loops completes, a second nested pair is used to go through all of the results files and find the name of the files which are equal to the minimum and maximum values. This is not strictly necessary, however it can prove useful to able to visually notice patterns between the results, such as a particular type of audio input file consistently giving the best or worst results.

```
  for (int i = 0; i < (allresultsvector.size()); ++i){
    vectorelement2 = (Object[]) (allresultsvector.get(i));
    String[] filenamesplit = vectorelement2[0].toString().split("-");
    String harmonicnumber = filenamesplit[2];

    if (Integer.parseInt(harmonicnumber) == a){
      if (parseDouble(vectorelement2[1].toString()) == max){
        maxresultsvector.add(vectorelement2[0]);
        bestresultsvector.add(vectorelement2[0]);
        double temp = parseDouble(vectorelement2[1].toString()) + parseDouble(
            vectorelement2[2].toString());

        if (temp > maxresultpluserror){
          maxresultpluserror = temp;
        }
      }
      else if (parseDouble(vectorelement2[1].toString()) == min){
        minresultsvector.add(vectorelement2[0]);
      }

      if (parseDouble(vectorelement2[2].toString()) == maxoctaveerror){
        maxoctaveerrorvector.add(vectorelement2[0]);
      }
      else if (parseDouble(vectorelement2[2].toString()) == minoctaveerror){
        minoctaveerrorvector.add(vectorelement2[0]);
                      }

      if (parseDouble(vectorelement2[1].toString()) + parseDouble(
          vectorelement2[2].toString()) == overallmax) {
        bestresultsvector2.add(vectorelement2[0]);
      }
    }
  }

   outputelements = new String[]{String.valueOf(a), String.valueOf(max),
       String.valueOf(min), String.valueOf(averagescore),
    String.valueOf(overallmax), String.valueOf(averagescorewithoctaveerrors),
    String.valueOf(maxresultpluserror), String.valueOf(maxoctaveerror), String
        .valueOf(minoctaveerror),
    String.valueOf(averageoctaveerror)};

    writeToFile(outputelements);

  System.out.println("number of Results Analysed = " + allresultsvector.size()
      );
  System.out.println("DART Processing Completed");
  time = System.currentTimeMillis() - time;
  System.out.println("Total Analysis Time: " + (time/1000) + " seconds");
  out.close();
```

Finally, the **`writeToFile()`** method is called, writing **`outputelements`** to the file. After this, **`out.close()`** is called; the CSV files can then be opened in Microsoft Excel or any other spreadsheet application, and easily graphed.

A.3 Creating The DART Execution Environment

Allowing a Triana workflow to run as a standalone application is necessary in order to use a distribution mechanism that is not heavily reliant on Triana, removing dependencies on the Triana software and lowering the worker's system requirements. Triana is a Java application and benefits from many of the advantages Java brings, as highlighted in the background chapter. It is a well coded, object oriented application and as such is structured in a way which enables components to be separated and used in isolation without many drastic modifications.

XtremWeb is written in Java and accepts a JAR file as an application. BOINC requires C++ applications, however options are available to 'wrap' the Java application archive into formats that are accepted by BOINC, as will be explained in the next section. In this section, the focus of the implementation is on the creation of a standalone JAR file.

As a *graphical* Problem Solving Environment, one of Triana's largest dependancies is on its extensive Graphical User Interface (GUI) with which the user communicates. Stripping the GUI away from the underlying code will allow for a much more streamlined operation. The GUI is of course used by the user to select and connect the Triana units and components, in a particular order, and to run the algorithms. With no GUI, the Triana taskgraphs need to be finalised and then mapped into a sequence which follows the flow of data that was designed in Triana. This can of course become an automated feature in the future; this section focusses on the implementation of standalone workflows and the implementation of the design of the framework, however. The implementation of this application can serve as a framework for any Triana work-flow or task graph that is required to run as a standalone application.

Porting all of the units to work using the new superclass **`Unit`** (as explained earlier) meant that the GUI was more decoupled from the functional code, allowing the variables set by the GUI to be set using the Command Line Interface. Triana can be stripped of all classes and methods that are not required for the execution of the particular DART algorithm or application that is being ported to work standalone. Only the dependancies of each unit in the workflow, as well as all of the Triana datatypes must be adhered to.

The Triana DART workflow/taskgraph must be reconstructed and *get* and *set* data from one unit of code to another. Each unit may have several adjustable variables, which must be available to set at runtime (from the command line) in order to change the outcome of the algorithm. In the case of DART, it is imperative to be able to adjust these variables with an interface that is easy to use and direct, enabling the SHS parameter-sweep experiment.

Creating a Java Archive (JAR), aggregates several classes and associated metadata into one executable file. The Dart.java class file is the main class that is used to create the JAR executable. The passing of Triana Data types or Java objects from one instantiation of each class to the next is handled by this class, whereby each unit in the Triana task graph is instantiated as an object. After instantiating the relevant unit objects in **`TestDart.java`**, the flow of data from one unit to the next must be controlled.

Given a simple example scenario where a **`LoadSound`** unit is to pass data to a **`Play`** unit (simply enabling and initiating the playback of high quality audio), the following method calls must be used:

```
loadsound.process();
```

The **`LoadSound`** unit (the first unit in the algorithm/workflow) begins processing/initialises the audio

```
Object outdataA = loadsound.getOutputData();
```

This creates an objected containing the output from the **`LoadSound`** unit.

```
play.setDataInput(outdataA);
```

The output is set as the input for the next corresponding unit (i.e. the Play unit)

```
play.process();
```

The **`process()`** method is called to initiate the processing of the next unit. This simple structure can be used to chain any number of units together.

A.3.1 Structure of the Standalone DART JAR

The main class of the DART application (JAR) is simply named **`Dart.java`**. This class accepts the parameters that were input from the command line, sequentially instantiates of all the units, and passes the output of one unit to the input of the next unit in the SHS algorithm sequence. It also, in tandem with the NoteMapper.java class at the end of the DART task graph, writes to the results file generated by the algorithm. The overall structure of the class can be shown as:

- Create Options instance and add possible CLI variable parameters
- Create a CommandLine BasicParser
- Initialise each DART Unit object used in the workflow
- Query or *interrogate* the CommandLine in order to check if the variable has been set
- Connect the units together and begin the workflow processing
- Write the results to a text file

The Jakarta Command Line Interface (now called the Apache Commons CLI[2]) is used to create the DART interface design described in the Design chapter. There are three stages to command line processing; the definition, parsing and interrogation stages.

Each command line must define the set of variables that will be used to define the parameters to the DART application. CLI uses the **`Options`** class as a container for the **`Option`** instance. The result of the definition stage is an **`Options`** instance. Once the **`Options`** object is instantiated, the various allowed parameters must be added to it using the **`addOption`** method:

```
Options opt = new Options();

opt.addOption("h", false, "Print help for this application");
opt.addOption("infile", true, "Name/loc of the input audio file. Must be 16
    bit/44.1KHz wav/aif");
opt.addOption("outfile", true, "Output text file containing the results
    source to use");
opt.addOption("repeat_no", false, "Number of times to repeat DART algorithm.
     Run once by default.");
opt.addOption("audiodir", false, "Location of a directory containing audio
    files for analysis.");
```

The **`addOption()`** method has three parameters. The first parameter is a **`String`** that represents the **`Option`**. The second parameter is a **`boolean`** that specifies whether the

[2]`http://commons.apache.org/cli/`

`Option` requires an argument or not. In the case of a boolean option (sometimes referred to as a **`flag`**), when an argument value is not present, *false* is passed[3]. The third parameter is the description of the **`Option`**. This description will be used in the usage text of the application.

The above listing shows the general options added, in-file name (the name of the audio input file to be analysed), out-file name (the name of the results text file), number of times to repeat the job, and the directory and location of the audio. The LoadSound arguments (input audio chunk size) are given by:

```
opt.addOption("chunksize_ms", false, "Size of the chunk is set in
    milliseconds. Default = 500");
opt.addOption("chunksize_samples", false, "Size of the chunk is set in
    samples. Default = 22050");
```

The Fast Fourier Transform arguments:

```
opt.addOption("fft_transform", false, "Alters the type of transform
    performed by the FFT. Automatic, Direct, Direct/Normalised(1/N), Inverse
    , Inverse/Normalised(1/N). Default = Automatic.");
opt.addOption("fft_optimise", false, "The FFT algorithm can be optimised for
     MaximumSpeed or MinimumStorage. Default = MaximumSpeed");
opt.addOption("fft_window", true, "For a 1D transform, different windows can
     be applied to the data, such as a Hamming window, a Hanning window,
    Gaussian, etc. Default = Hann(Hanning) window.");
opt.addOption("fft_pad", false, "Boolean argument that allows default
    padding of input arrays with zeros to a power of two to be turned off.
    This greatly (negatively) affects the efficiency of the FFT algorithm.")
    ;
```

The PitchDetection arguments:

```
opt.addOption("nofreqpoints", true, "Number of top frequency peaks analysed
    by the PitchDetection module. Default = 30.");
opt.addOption("noharmonics", true, "Number of harmonics that are summed up
    from the fundamental in order to calculate the main frequency of the.
    Default = 20");
parser = new BasicParser();
cl = parser.parse(opt, args);
```

[3]The required parameters in the DART SHS algorithm are given in the design chapter

The above code listing also shows the parsing stage, where the text passed into the application via the command line, is processed. The parse method is defined in **BasicParser**, a sub-class of **Parser**, and takes an **Options** instance and a **String[]** of arguments, returning a **CommandLine**. The class **BasicParser** provides a very simple implementation of the flatten method[4]. The parse methods of **BasicParser** are used to parse the command line arguments. The result of the parsing stage is a **CommandLine** instance.

Each unit in the Triana DART task-graph is then instantiated:

```
loadsound = (LoadSoundNoGUI)Class.forName("mir.LoadSoundNoGUI").newInstance
    ();
fft = (FFT)Class.forName("mir.processing.FFT").newInstance();
oneside = (Unit)Class.forName("mir.processing.OneSide").newInstance();
amplitudespectrum = (Unit)Class.forName("mir.processing.AmplitudeSpectrum").
    newInstance();
pitchdetector = (PitchDetection)Class.forName("mir.processing.PitchDetection
    ").newInstance();
notemapper = (NoteMapper)Class.forName("mir.processing.NoteMapper").
    newInstance();
```

During the final CLI stage, *interrogation*, DART queries the **CommandLine** to decide which DART parameter variables to use, depending on **boolean** options and uses the option values to provide the data. The result of the interrogation stage is that the code is informed by the input that was supplied on the command line and processed according to the parser and **Options** rules.

The application then checks if the specified option is present by interrogating the **CommandLine** object. The **hasOption()** method takes a **String** parameter and returns true if the option represented by the **String** is present, otherwise it returns false. The only commands that are required are the **infile**, **outfile**, **fft_window**, **nofreqpoints**, and **noharmonics**. Querying the command line:

```
if(cl.hasOption("infile")){
  System.out.println("Input file = " + cl.getOptionValue("infile"));
  loadsound.fileName = home + File.separator + cl.getOptionValue("infile");
}
```

[4]http://commons.apache.org/cli/api-release/org/apache/commons/cli/BasicParser.html#flatten(org.apache.commons.cli.Options,%20java.lang.String[],%20boolean

```
  if(cl.hasOption("outfile")){
    System.out.println("Output file = " + cl.getOptionValue("outfile"));
    notemapper.outputfilename = (cl.getOptionValue("outfile"));
  }
  if(cl.hasOption("repeat_no")){
    System.out.println("No. of times to repeat DART algorithm = " + cl.
        getOptionValue("repeat_no"));
  }
  if(cl.hasOption("audiodir")){
    System.out.println("Location of a directory containing audio files for
        analysis = " + cl.getOptionValue("audiodir"));
  }
  if(cl.hasOption("chunksize_ms")){ // Default is 500ms
    System.out.println("Audio chunk size in ms = " + cl.getOptionValue("
        chunksize_ms"));
  }
  if(cl.hasOption("chunksize_samples")){
    System.out.println("Audio chunk size in samples = " + cl.getOptionValue("
        chunksize_samples"));
  }
  if(cl.hasOption("fft_transform")){
    System.out.println("FFT Transform type = " + cl.getOptionValue("
        fft_transform"));
  }
  if(cl.hasOption("fft_optimise")){
    System.out.println("FFT Optimisation type = " + cl.getOptionValue("
        fft_optimise"));
  }
  if(cl.hasOption("fft_window")){
    System.out.println("FFT Window type = " + cl.getOptionValue("fft_window"))
        ;
    fft.windowfunction = cl.getOptionValue("fft_window");
  }
  if(cl.hasOption("fft_pad")){
    System.out.println("FFT Transform True/False = " + cl.getOptionValue("
        fft_pad"));
  }
  if(cl.hasOption("nofreqpoints")){
    System.out.println("No. Of FreqPoints = " + cl.getOptionValue("
        nofreqpoints"));
    pitchdetector.noFreqPoints = Integer.parseInt(cl.getOptionValue("
        nofreqpoints"));
  }
  if(cl.hasOption("noharmonics")){
    System.out.println("No. Of Harmonics = " + cl.getOptionValue("noharmonics
        "));
    pitchdetector.noOfHarmonics = Integer.parseInt(cl.getOptionValue("
        noharmonics"));
  }
  loadsound.init();
```

The **loadsound.init()** method is also called to trigger the **createAudioInputStream(new File(fileName))** and **userScreen()** methods, creating a new audio input stream and setting the correct chunk size for the input audio.

The unit workflow is reconstructed by taking the output of the first unit and setting it as the input of the following unit, using simple **setOutputData** and **getInputData** methods from the **Unit** class (and as explained in the beginning of this section). A timer is set in order to calculate the total processing time for the current job, which is written to the results file. This process is demonstrated in the listing below.

```
  System.out.println("DART processing has started...");

  while (!loadsound.isLastChunk()){
    currentTime = (System.currentTimeMillis() - time)/1000;

    if (currentTime >= counter && currentTime > 19){
      System.out.println((int)currentTime + " seconds have elapsed");
      counter += 20;
    }

    loadsound.process();

    Object outdataA = loadsound.getOutputData();

    fft.setDataInput(outdataA);
    fft.process();

    Object outdataB = fft.getOutputData();

    oneside.setDataInput(outdataB);
    oneside.process();

    Object outdataC = oneside.getOutputData();

    amplitudespectrum.setDataInput(outdataC);
    amplitudespectrum.process();

    Object outdataD = amplitudespectrum.getOutputData();

    pitchdetector.setDataInput(outdataD);
    pitchdetector.process();

    Object outdataE = pitchdetector.getOutputData();

    notemapper.setDataInput(outdataE);
    notemapper.process();

    System.gc();
  }
```

The elapsed processing time is reported every 20 seconds, giving visual feedback to the user, allowing them to know that the program has not stalled.

The final step is to write the results of the processing to a results file. The following code writes and formats the results to a text file.

```
  System.out.println("DART processing completed");
  time = System.currentTimeMillis() - time;
  System.out.println("The DART application took " + (time/1000) + " seconds to
       run");

  // Append variable settings to Result file
  try {
    String filename = "results/" + cl.getOptionValue("outfile");
    BufferedWriter out = new BufferedWriter(new FileWriter(filename, true));
    out.newLine();
    out.newLine();
    out.write("The DART application took " + (time/1000) + " seconds to run");
    out.newLine();
    out.write("Input Analysis File = " + cl.getOptionValue("infile"));
    out.newLine();
    out.write("FFT Window Type = " + cl.getOptionValue("fft_window"));
    out.newLine();
    out.write("Number of Frequency Points = " + Integer.parseInt(cl.
        getOptionValue("nofreqpoints")));
    out.newLine();
    out.write("Number of Harmonics = " + Integer.parseInt(cl.getOptionValue("
        noharmonics")));
    out.close();
  }
```

The **Dart.java** class means that given the following command line argument:

```
java -jar Dart.jar -infile DARTOboe.wav -outfile DART-MyResults.txt -
    nofreqpoints 1 -noharmonics 4 -fft_window Bartlett
```

The DART application will create a results file called **DART-MyResults.txt** with the following format:

```
DART RESULTS

64, 72, 82, 87, 98, 109, 123, 131, 131, 145, 164, 176,...

C2, D2, E2, F2, G2, A2, B2, C3, C3, D3, E3, F3,...

The DART application took 56 seconds to run
Input Analysis File = DARTOboe.wav
FFT Window Type = Bartlett
Number of Frequency Points = 1
Number of Harmonics = 4
```

A.3.2 Porting the Units to Standalone

Before porting to work stand alone as part of the DART application, the functionality of the **LoadSound** unit was heavily integrated in the units GUI. Porting the unit to use the new **Unit** super class helped decouple the functional code from the graphical interface. However, the functional code from the **LoadSoundPanel** class needed to be integrated into a new unit called **LoadSoundNoGUI.java**. This integrated all of the JavaSound code into one class, reducing the amount of code by over 60%. Using the DART CLI it is possible to set the file (including location) and chunk size in an extremely efficient way.

The FFT unit also benefits from a large reduction in size, with the GUI options stripped.

The OneSide, AmplitudeSpectrum, Pitch Detection, and NoteMapper units have no GUI in Triana, and therefore no major modifications were required for the units to function in DART. The **`init()`** method in all units - called when the unit is created and initialises the unit's properties and parameters - can simply be bypassed.

Bibliography

[1] J. Stephen Downie. *Music Information Retrieval*, volume 37. Information Today Inc., 2003.

[2] GNU. "Regarding Gnutella". `http://www.gnu.org/philosophy/gnutella.html`.

[3] Apple Computer Inc. *iTunes Store Top Music Retailer in the US.* `http://www.apple.com/pr/library/2008/04/03itunes.html`.

[4] Apple Computer Inc. *iTunes Store Tops 10 Billion Songs Sold.* `http://www.apple.com/pr/library/2010/02/25iTunes-Store-Tops-10-Billion-Songs-Sold.html`.

[5] The NPD Group. *Amazon Ties Walmart as Second-Ranked US Music Retailer, Behind Industry-Leader iTunes.* `http://www.npd.com/press/releases/press_100526.html`, May 2006.

[6] A. Durey and M. Clements. Features for melody spotting using hidden Markov models. In *IEEE International Conference On Acoustics, Speech and Signal Processing*, volume 2. Citeseer, 2002.

[7] H. Hoos, K. Renz, and M. Gorg. GUIDO/MIR-an experimental musical information retrieval system based on GUIDO music notation. In *International Symposium on Music Information Retrieval.* Citeseer, 2001.

[8] J. Futrelle and J.S. Downie. Interdisciplinary communities and research issues in music information retrieval. In *Proceedings of the Third International Conference on Music Information Retrieval*, pages 215–221. Citeseer, 2002.

[9] R.P. Smiraglia. Musical works as information retrieval entities: Epistemological perspectives. In *Proceedings of the 2nd Annual International Symposium on Music Information Retrieval (ISMIR 2001)*, pages 85–92. Citeseer, 2001.

[10] J.S. Downie. The Music Information Retrieval Evaluation eXchange (2005–2007): A Window Into Music Information Retrieval Research. *Acoustical Science and Technology*, 29(4):247–255, 2008.

[11] University of Illinois. The International Music Information Retrieval Systems Evaluation Laboratory (IMIRSEL) Project. `http://www.music-ir.org/evaluation/`.

[12] J. Tague-Sutcliffe and J. Blustein. A statistical analysis of the TREC-3 data. In *Overview of the Third Text REtrieval Conference (TREC-3)*, pages 385–398, 1995.

[13] J.S. Downie, K. West, A. Ehmann, and E. Vincent. The 2005 Music Information Retrieval Evaluation eXchange (MIREX 2005): Preliminary Overview. In *Proceedings of the International Conference on Music Information Retrieval*, pages 320–323. Citeseer, 2005.

[14] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@Home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[15] E. Al-Shakarchi, P. Cozza, A. Harrison, C. Mastroianni, M. Shields, D. Talia, and I. Taylor. Distributing workflows over a ubiquitous P2P network. *Scientific Programming*, 15(4):269–281, 2007.

[16] E. Al-Shakarchi, I. Taylor, and S.D. Beck. Distributed Audio Retrieval using Triana (DART). In *International Computer Music Conference (ICMC)*, pages 6–11, 2006.

[17] E. Al-Shakarchi, I. Taylor, and S.D. Beck. *DART: A Framework for Distributed Audio Analysis and Music Information Retrieval.* IGI Global, 2006.

[18] Karan Vahi, Ian Harvey, Taghrid Samak, Daniel Gunter, Kieran Evans, David Rogers, Ian Taylor, Monte Goode, Fabio Silva, Eddie Al-Shakarchi, et al. A general approach to real-time workflow monitoring. *Journal of Grid Computing*, 2012.

[19] D.M. Randel. *The Harvard Dictionary of Music.* Belknap Press, 2003.

[20] H. NYQUIST. Certain Topics in Telegraph Transmission Theory. *Transactions of the AIEE*, 1928.

[21] J. Fourier. Mémoire sur la propagation de la Chaleur dans les corps solides. *Nouveau Bulletin des Sciences de la Société Philomathique de Paris*, 6:112–116, 1808.

[22] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput*, 19(90):297–301, 1965.

[23] GD Bergland. A Guided Tour of the Fast Fourier Transform. *Spectrum, IEEE*, 6(7):41–52, 1969.

[24] E.O. Brigham and RE Morrow. The Fast Fourier Transform. *Spectrum, IEEE*, 4(12):63–70, 1967.

[25] Philip McLeod. Fast, accurate pitch detection tools for music analysis. *Academisch proefschrift, University of Otago. Department of Computer Science*, 2009.

[26] Benjamin Kedem. Spectral analysis and discrimination by zero-crossings. *Proceedings of the IEEE*, 74(11):1477–1493, 1986.

[27] Curtis Roads. *The computer music tutorial*. The MIT press, 1996.

[28] Eric Scheirer and Malcolm Slaney. Construction and evaluation of a robust multifeature speech/music discriminator. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, volume 2, pages 1331–1334. IEEE, 1997.

[29] Stéphane Rossignol, Xavier Rodet, Jöel Soumagne, Jean-Luc Collette, and Philippe Depalle. Feature extraction and temporal segmentation of acoustic signals. In *Proc. ICMC*, volume 98, pages 199–202. Citeseer, 1998.

[30] Alain De Cheveigné and Hideki Kawahara. Yin, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111:1917, 2002.

[31] P. McLeod and G. Wyvill. A Smarter Way to Find Pitch. In *Proceedings of International Computer Music Conference, ICMC*, 2005.

[32] F.J. Harris. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[33] A Michael Noll. Cepstrum pitch determination. *The journal of the acoustical society of America*, 41:293, 1967.

[34] James Flanagan. *Speech Analysis, Synthesis, and Perception.* Springer-Verlag, Berlin-Heidelberg-New York, 1972.

[35] Boris Doval and Xavier Rodet. Estimation of fundamental frequency of musical sound signals. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 3657–3660. IEEE, 1991.

[36] Boris Doval and Xavier Rodet. Fundamental frequency estimation and tracking using maximum likelihood harmonic matching and hmms. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 221–224. IEEE, 1993.

[37] Martin Piszczalski and Bernard A Galler. Predicting musical pitch from component frequency ratios. *The Journal of the Acoustical Society of America*, 66:710, 1979.

[38] James A Moorer. On the transcription of musical sound by computer. *Computer Music Journal*, pages 32–38, 1977.

[39] John E Lane. Pitch detection using a tunable iir filter. *Computer Music Journal*, 14(3):46–59, 1990.

[40] Adriano Mitre, Marcelo Queiroz, and Regis Faria. Accurate and efficient fundamental frequency determination from precise partial estimates. In *Proceedings of the 4th AES Brazil Conference*, pages 113–118, 2006.

[41] Dik J. Hermes. Measurements of pitch by subharmonic summation. *Journal of the Acoustical Society of America*, 83(1):257–264, January 1988.

[42] Xuejing Sun. A pitch determination algorithm based on subharmonic-to-harmonic ratio. *A A*, 1000:1, 2000.

[43] Xuejing Sun. Pitch determination and voice quality analysis using subharmonic-to-harmonic ratio. In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, volume 1, pages I–333. IEEE, 2002.

[44] Karin Dressler. Pitch estimation by the pair-wise evaluation of spectral peaks. In *Audio Engineering Society Conference: 42nd International Conference: Semantic Audio*, 2011.

[45] F. Oberholzer-Gee and K. Strumpf. File-sharing and Copyright. *Policy*, 10:1–46, 2009.

[46] J.A. Montalvo. A MIDI Track for Music IR Evaluation. *"The MIR/MDL Evaluation Project White Paper Collection" Edition# 3*, page 32, 2003.

[47] K. MacMillan. Common Music Notation as a Source for Music Information Retrieval. *"The MIR/MDL Evaluation Project White Paper Collection" Edition# 3*, page 27, 2003.

[48] G. Tzanetakis and P. Cook. Manipulation, analysis and retrieval systems for audio signals. *Princeton University, Princeton, NJ*, 2002.

[49] Aucouturier and Pachet. Representing musical genre: A state of the art. *Journal of New Music Research*, 32:83–93, 2003.

[50] J. Foote, M. Cooper, and U. Nam. Audio Retrieval by Rhythmic Similarity. In *Proceedings of the International Conference on Music Information Retrieval*, volume 3, pages 265–266. Citeseer, 2002.

[51] B. Logan and A. Salomon. A Content-Based Music Similarity Function. *Cambrige Res. Lab*, 2001.

[52] C. Yang. Music database retrieval based on spectral similarity. In *Proceedings of the 2nd Annual International Symposium on Music Information Retrieval*, pages 37–38. Citeseer, 2001.

[53] J. Haitsma and T. Kalker. A Highly Robust Audio Fingerprinting System. In *Proc. ISMIR*, volume 2002, pages 144–148. Citeseer, 2002.

[54] M.A. Casey, R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney. Content-based music information retrieval: current directions and future challenges. *Proceedings of the IEEE*, 96(4):668–696, 2008.

[55] S. Pauws. Musical Key Extraction from Audio. In *Proc. ISMIR*, volume 4. Citeseer, 2004.

[56] G. Tummarello, C. Morbidoni, P. Puliti, and F. Piazza. Semantic audio hyperlinking: a multimedia-semantic web scenario. In *Automated Production of Cross Media*

Content for Multi-Channel Distribution, 2005. AXMEDIS 2005. First International Conference on, pages 4–pp. IEEE, 2005.

[57] Youngmoo E Kim, Erik M Schmidt, Raymond Migneco, Brandon G Morton, Patrick Richardson, Jeffrey Scott, Jacquelin A Speck, and Douglas Turnbull. Music emotion recognition: A state of the art review. In *Proc. ISMIR*, pages 255–266. Citeseer, 2010.

[58] J. Haitsma and T. Kalker. A Highly Robust Audio Fingerprinting System. In *Proc. ISMIR*, volume 2002, pages 144–148. Citeseer, 2002.

[59] Nicola Orio. Music Retrieval: A Tutorial and Review. *Foundations and Trends in Information Retrieval*, 1(1):1–90, November 2006.

[60] G. Tzanetakis, J. Gao, and P. Steenkiste. A scalable peer-to-peer system for music information retrieval. *Computer Music Journal*, 28(2):24–33, 2004.

[61] C. Wang, J. Li, and S. Shi. An Approach to Content-Based Approximate Query Processing in Peer-to-Peer Data Systems. *Grid and Cooperative Computing*, pages 348–355, 2004.

[62] S. Baumann. Music Similarity Analysis in a P2P Environment. In *Digital media processing for multimedia interactive services: proceedings of the 4th European Workshop on Image Analysis for Multimedia Interactive Services: Queen Mary, University of London, 9-11 April 2003*, page 314. World Scientific Pub Co Inc, 2003.

[63] K. West, A. Kumar, A. Shirk, G. Zhu, J. Downie, A. Ehmann, and M. Bay. The Networked Environment for Music Analysis (NEMA). In *6th IEEE World Congress on Services*, pages 314–317, 2010.

[64] NEMA Architecture. `http://www.music-ir.org/?q=nema/architecture`.

[65] J. Reiss and M. Sandler. Benchmarking music information retrieval systems. *"The MIR/MDL Evaluation Project White Paper Collection" Edition# 3*, page 37, 2003.

[66] C.W. Cleverdon, J. Mills, and M. Keen. Factors determining the performance of indexing systems. Technical report, College of Aeronautics, Indiana University, 1966.

[67] J. Futrelle. Three criteria for the evaluation of music information retrieval techniques against collections of musical material. *"The MIR/MDL Evaluation Project White Paper Collection" Edition# 3*, page 20, 2003.

[68] J.S. Downie. Toward the scientific evaluation of music information retrieval systems. In *Proceedings of the 4th International Conference on Music Information Retrieval (ISMIR 2003)*, pages 25–32. Citeseer, 2003.

[69] Pedro Cano, Markus Koppenberger, and Nicolas Wack. An industrial-strength content-based music recommendation system. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 673–673. ACM, 2005.

[70] Pedro Cano, Markus Koppenberger, and Nicolas Wack. Content-based music audio recommendation. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 211–212. ACM, 2005.

[71] Qing Li, Byeong Man Kim, Dong Hai Guan, et al. A music recommender based on audio features. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 532–533. ACM, 2004.

[72] Dmitry Bogdanov, Martín Haro, Ferdinand Fuhrmann, Emilia Gómez, and Perfecto Herrera. Content-based music recommendation based on user preference examples. In *1st Workshop On Music Recommendation And Discovery (WOMRAD), ACM RecSys, 2010, Barcelona, Spain*, 2010.

[73] G. Tzanetakis and P. Cook. Marsyas: A framework for audio analysis. *Organised sound*, 4(03):169–175, 2000.

[74] S. Bray and G. Tzanetakis. Distributed audio feature extraction for music. In *Proceedings of the International Conference on Music Information Retrieval*, pages 434–437. Citeseer, 2005.

[75] G. Tzanetakis. Marsyas-0.2: a case study in implementing music information retrieval systems. *Intelligent Music Information Systems. IGI Global*, 2007.

[76] X. Amatriain, M. De Boer, E. Robledo, and D. Garcia. CLAM: an OO framework for developing audio and music applications. In *Companion of the 17th annual*

ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 22–23. ACM, 2002.

[77] X. Amatriain. Clam: A framework for audio and music application development. *Software, IEEE*, 24(1):82–85, 2007.

[78] J.S. Downie, J. Futrelle, and D. Tcheng. The international music information retrieval systems evaluation laboratory: Governance, access and security. In *Proc. of ISMIR*, pages 9–15. Citeseer, 2004.

[79] A. Shirk. D2K Web Service Design & Implementation. *Presentation given to NCSA CyberArchitecture Working Group*, 2004.

[80] J.S. Downie, A.F. Ehmann, and D. Tcheng. Music-to-Knowledge (M2K): a prototyping and evaluation environment for music information retrieval research. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 676–676. ACM, 2005.

[81] A.F. Ehmann, J.S. Downie, and M.C. Jones. The music information retrieval evaluation exchange "Do-It-Yourself" Web Service. In *Proceedings of the International Conference on Music Information Retrieval.* Citeseer, 2007.

[82] Ian Wang Matthew Shields, Ian Taylor. Distributed Computing With Triana, A Short Course. `http://www.trianacode.org/shortcourse/index.html`.

[83] Oracle. Java Sound API: Java Sound Demo. `http://java.sun.com/products/java-media/sound`.

[84] OpenP2P.com. What Is P2P... And What Isn't? `http://openp2p.com/lpt/a/472`, 2000.

[85] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[86] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[87] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[88] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100. IEEE, 2001.

[89] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[90] I. Wang. P2PS (Peer-to-Peer Simplified). In *Proceedings of 13th Annual Mardi Gras Conference-Frontiers of Grid Applications and Technologies*, pages 54–59, 2005.

[91] Carlo Mastroianni, Domenico Talia, and Oreste Verta. A super-peer model for resource discovery services in large-scale grids. *Future Generation Computer Systems*, 21(8):1235–1248, 2005.

[92] B.J. Wilson. *JXTA*. Pearson Education, 2002.

[93] T. Morkved. Peer-to-Peer Programming with Wireless Devices. *University of New South Whales and Agder University College*, 2005.

[94] Marcin Cieślak. Boinc on jxta - suggestions for improvements. Presented at the Technical University of Wroclaw, Poland, June 2007.

[95] E. Urbah, P. Kacsuk, Z. Farkas, G. Fedak, G. Kecskemeti, O. Lodygensky, A. Marosi, Z. Balaton, G. Caillat, G. Gombas, et al. EDGeS: bridging EGEE to BOINC and XtremWeb. *Journal of Grid Computing*, 7(3):335–354, 2009.

[96] H. He, G. Fedak, P. Kacsuk, Z. Farkas, Z. Balaton, O. Lodygensky, E. Urbah, G. Caillat, F. Araujo, and A. Emmen. Extending the EGEE Grid with XtremWeb-HEP Desktop Grids. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 685–690. IEEE, 2010.

[97] Groupe de r eflexion. Plate-forme de recherche exp erimentale en informatique. Written in French, July 2003.

[98] D.P. Anderson. BOINC: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[99] D.P. Anderson and G. Fedak. The Computational and Storage Potential of Volunteer Computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 73–80. IEEE, 2006.

[100] V.S. Pande, I. Baker, J. Chapman, S.P. Elmer, S. Khaliq, S.M. Larson, Y.M. Rhee, M.R. Shirts, C.D. Snow, E.J. Sorin, et al. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68(1):91–109, 2003.

[101] B. Abbott, R. Abbott, R. Adhikari, P. Ajith, B. Allen, G. Allen, R. Amin, DP Anderson, SB Anderson, WG Anderson, et al. Einstein@ Home search for periodic gravitational waves in LIGO S4 data. *Physical Review D*, 79(2):022001, 2009.

[102] Boinc client-server technology. `http://en.wikipedia.org/wiki/BOINC_client--server_technology`.

[103] D.P. Anderson. BOINC: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[104] I. Kelley and I. Taylor. Bridging the data management gap between service and desktop grids. *Distributed and Parallel Systems*, pages 13–26, 2008.

[105] Robert W. Young. Terminology for Logarithmic Frequency Units. *The Journal of the Acoustical Society of America*, 11:134–139, July 1939.

[106] L. Rayleigh. The theory of the Helmholtz resonator. *Proceedings of the Royal Society of London. Series A*, 92(638):265, 1916.

[107] Project creation cookbook. `http://boinc.berkeley.edu/trac/wiki/CreateProjectCookbook`.

[108] Setting up a BOINC server. `http://boinc.berkeley.edu/trac/wiki/ServerIntro`.

[109] Berklee, University of California, `http://boinc.berkeley.edu/boinc.pdf`. *Creating BOINC Projects*, February 2007.

[110] Using BOINC with Java applications. http://boinc.berkeley.edu/trac/wiki/JavaApps.

[111] A.C. Marosi, Z. Balaton, and P. Kacsuk. GenWrapper: A generic wrapper for running legacy applications on desktop grids. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–6. IEEE, 2009.

[112] Q: How to play MP3 file through MP3 SPI and JavaSound? http://www.javazoom.net/mp3spi/documents.html.

[113] Apple Audio Unit Programmer's Guide. http://developer.apple.com/library/mac/#documentation/MusicAudio/Conceptual/AudioUnitProgrammingGuide/Introduction/Introduction.html, 2009.

[114] O. Lodygensky, G. Fedak, F. Cappello, V. Neri, M. Livny, and D. Thain. XtremWeb and Condor: sharing resources between Internet connected Condor pool. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 382–389. IEEE, 2003.

[115] The BOINC Server Virtual Machine. http://boinc.berkeley.edu/trac/wiki/VmServer.

[116] BOINC - Work Generation. http://boinc.berkeley.edu/trac/wiki/WorkGeneration.

[117] BOINC Credit System. http://en.wikipedia.org/wiki/BOINC_Credit_System.

[118] A New System for Runtime Estimation and Credit. http://boinc.berkeley.edu/trac/wiki/CreditNew.

[119] Characteristics of Different Smoothing Windows. http://zone.ni.com/reference/en-XX/help/371361B-01/lvanlsconcepts/char_smoothing_windows/.

[120] FFT Window Functions. http://en.wikipedia.org/wiki/Window_function.

[121] Hammer nonlinearity, dynamics and the piano sound. http://www.acs.psu.edu/drussell/Piano/Dynamics.html.

[122] What is Condor? http://research.cs.wisc.edu/condor/description.html.