A black background with a black square

Description automatically generated with medium confidence

An Interactive Tool to Teach Musical Harmony

Owen Graham

Student ID: 2051153

Course: BSc Maths and Computer Science

Supervisor: Paul Levy

Institution: School of Computer Science, University of Birmingham

Date: September 2023

**Abstract**

This report presents the development and evaluation of an innovative educational application which teaches the fundamental concepts of musical harmony, focusing on the notions of consonance and dissonance perceived when a pair of musical notes are played together and how this naturally led to development of various tuning systems, namely Pythagorean, Just, and Equal Tempered.

The project began with a review of relevant theory and history relating to musical harmony and a critical analysis of existing resources and the methods they adopt to explain the concepts involved. The application presented aims to educate more effectively than such resources by integrating interactivity, visualisation, and audio to provide an engaging experience to a broad audience.

Contents

[1 Introduction 1](#_Toc146568611)

[1.1 Motivation 1](#_Toc146568612)

[1.2 Relevant Theory 1](#_Toc146568613)

[1.3 Related Works 3](#_Toc146568614)

[1.3.1 Similar Resources 3](#_Toc146568615)

[1.3.2 Harmony Visualisation Tools 3](#_Toc146568616)

[1.4 Git Repository 5](#_Toc146568617)

[2 Design 5](#_Toc146568618)

[2.1 Methodology 5](#_Toc146568619)

[2.1.1 Max 5](#_Toc146568620)

[2.1.2 Planning 7](#_Toc146568621)

[2.2 Requirements 7](#_Toc146568622)

[2.2.1 Functional Requirements 7](#_Toc146568623)

[2.2.2 Non-Functional Requirements 7](#_Toc146568624)

[3 Implementation 8](#_Toc146568625)

[3.1 User Interface 8](#_Toc146568626)

[3.2 Pages 10](#_Toc146568627)

[3.2.1 Main Page 10](#_Toc146568628)

[3.2.2 Title Page 12](#_Toc146568629)

[3.2.3 Intervals 13](#_Toc146568630)

[3.2.4 Consonant vs Dissonant Intervals 16](#_Toc146568631)

[3.2.5 Frequency 19](#_Toc146568632)

[3.2.6 Pythagoras’ Hammers 20](#_Toc146568633)

[3.2.7 Polyrhythms 22](#_Toc146568634)

[3.2.8 Pythagorean Scale 23](#_Toc146568635)

[3.2.9 The Pythagorean Comma 27](#_Toc146568636)

[3.2.10 The Overtone Series 28](#_Toc146568637)

[3.2.11 Just Intonation 31](#_Toc146568638)

[3.2.12 Problems with Just Intonation 32](#_Toc146568639)

[3.2.13 Equal Temperament 37](#_Toc146568640)

[3.2.14 Comparison 38](#_Toc146568641)

[4 Results and Evaluation 41](#_Toc146568642)

[4.1 Testing 41](#_Toc146568643)

[4.2 Project Management 44](#_Toc146568644)

[4.3 Discussion 45](#_Toc146568645)

[4.4 Conclusion 46](#_Toc146568646)

[5 References 46](#_Toc146568647)

[6 Appendices 47](#_Toc146568648)

[6.1 Instructions to Run 47](#_Toc146568649)

[6.2 Object Descriptions 47](#_Toc146568650)

[6.3 Additional Patch Images 51](#_Toc146568651)

# Introduction

## Motivation

The advancement in ability and availability of DAWs (Digital Audio Workstations) such as Ableton Live means that many people are interested in producing music, a creative hobby that allows one access to creative expression in a way which can be good for mental health and provides a valued service to society. Sitting down in front of a DAW for the first time results in a similar thought process for many such people: creating rhythm and song structure is an intuitive process for which the subject can draw inspiration from music they themselves enjoy listening to, but there seem to be an abstract set of rules that dictate whether the notes used sound good or not. Researching the theory which is involved in uncovering the reasons behind these rules reveals a fascinating rabbit hole encompassing physics, number theory, and history, with familiar names to a maths student such as myself such as Pythagoras, Ptolemy, and Euler cropping up in unexpected places. It turns out that musical harmony is built on concrete mathematical foundations and is not as subjective as one might first expect given that appreciation of music is itself so subjective.

Because music theory is most effectively taught interactively and auditorily, the usual option that provides this is a professional music teacher. This is not an option for many people due to cost, geographical location, and availability of musical instruments. A free application which simulates this learning environment by employing interactivity and visual and auditory learning would provide a larger audience with the knowledge required to effectively produce music and could also be used to support music teachers in schools.

The goal of this project is to improve on existing resources that teach the theory behind musical harmony by providing a comprehensive all-in-one-place application that facilitates more effective learning by engaging the user’s senses with visualization and auditory feedback. If this is achieved, the application will provide an engaging and enjoyable experience that provides a wide variety of users with the basic knowledge required to begin composing and producing music.

This report begins by providing the basic theory required to understand the content of the application, followed by an analysis of existing resources with similar aims to this project and novel visualisation techniques used to aid the understanding of musical harmony. Section 2 gives a description of Max, the programming language the application was developed in, and the reason this language was chosen, followed by a specification of the application’s requirements. Section 3 gives a detailed explanation of how this was implemented, written to be understood by readers who aren’t familiar with Max. Section 4 outlines the results of the project and discusses the appropriateness of the methods employed.

## Relevant Theory

Sound is how we perceive waves of air produced by a vibrating medium hitting our ear drum. Sounds are categorised by the frequency and amplitude of these waves. When referring to sounds produced by non-percussive musical instruments, we call these sounds notes.

Humans differentiate between musical notes using three perceived factors:

1. Volume: dictated by the amplitude of the sound wave of the note, volume refers to how loud or quiet a note is heard. Measured in decibels (dB).
2. Timbre: unique to an instrument or voice, timbre describes the specific tone or quality of notes produced by that instrument or voice. (Chase, 2022)
3. Pitch: how the human ear perceives the frequency, measured in Hertz (Hz), of the sound wave of the note as high or low.

It is important to note that pitch is the perception of a sound wave’s frequency, but we do not perceive frequencies linearly. The ear is more sensitive to differences between low frequencies than high frequencies. For instance, we hear a greater difference between 100 Hz and 200 Hz than between 1000 Hz and 1100 Hz. This is because 200 Hz is double the frequency of 100 Hz, whereas 1100 Hz is only 10% higher than 1000 Hz. This describes how we perceive frequency on a logarithmic scale. (Doshi, 2021)

Musical harmony refers to the sound of two or more notes heard simultaneously (Rich, 2019). When two musical notes are played simultaneously, the perceived distance between their respective pitches is measured as an interval.

When a note is played on a musical instrument, the sound heard is not just the frequency of that note, we also hear a range of integer multiples of that frequency, known as overtones. For example, the middle A on a piano is tuned to vibrate at 440 Hz, but the sound produced by pressing this key is the Fourier series of 440 Hz, 880 Hz, 1320 Hz… etc. It is the volume of each of these overtones that changes the timbre of a note.

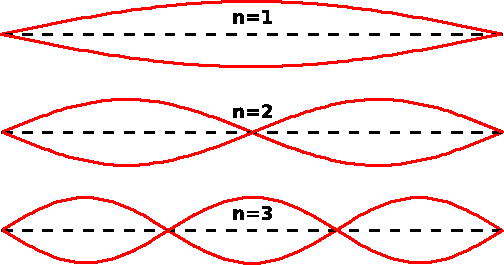


Figure ‑ The first three overtones as produced by a vibrating string. https://www.cantorsparadise.com/the-mathematical-nature-of-musical-scales-f0a6536bca5d

Human brains find patterns pleasing, and this can explain why some intervals sound ‘better’, or more ‘consonant’ to us than others. If we were to hear 1 Hz and 2 Hz played simultaneously (The human hearing range is between 20 Hz and 20,000 Hz so we wouldn’t actually be able to hear this), 2 Hz would already be included in the first note since it is the first overtone of 1 Hz and this means the resulting sound would sound organised and pleasing. We call the interval between two notes an octave when the frequency of one note is double that of the other, and this explains why we give notes that are an octave apart the same name. For example, 220 Hz and 440 Hz are both A but the first is an octave lower than the second.

The ancient Greek mathematician and philosopher Pythagoras, regarded by some as the first music theorist, observed the pleasant sound produced by notes with overlapping overtone series sounding simultaneously when he heard hammers of different weights striking an anvil in a blacksmith’s shop.

## Related Works

Research for this project took two different forms:

1. Educating myself on the theory involved in musical harmony. Since this is the theory I would include in the application, this also served as a critical analysis of resources with similar aims to mine.

2. Researching methods of visualising harmony.

### Similar Resources

Looking at other educational material with a focus on musical harmony gave valuable insight into the pedagogical aspects of what the application should include. For example, most resources, such as (Ashton, 2018), made mention to the famous story of Pythagoras observing consonance in the sounds produced by hammers whose weights were at a ratio of 2:1 and 3:2 since this is an analogy that ties abstract concepts such as frequency to the more tangible measurement of weight. Similarly, (Ashley, 2016) took advantage of the relationship between rhythm and frequency to explain frequency ratios using polyrhythms, and Adam Neely went one step further in his YouTube video titled “Harmonic Polyrhythms Explained! [ AN's Bass Lessons #27 ]” by increasing the speed of polyrhythms until they were audible as the harmonic intervals with the same ratio (Neely, 2015).

A key deficiency identified in resources that aim to inform on similar subject matter was the lack of audio, an inherent component of music theory. Resources which did implement audio were either written materials with limited sound files to demonstrate particular things, or YouTube videos. The written materials featured too few audio files which resulted in them feeling somewhat scripted and incomprehensive, while the YouTube videos suffered from the same issue whilst also being restricted to the creator’s instructional pace. It can be expected that this won’t be the same as the learning pace of the entire audience, meaning some people will be forced to frequently rewind and watch again, while others may lose interest if the pace is too slow for them.

### Harmony Visualisation Tools

Both (Malandrino, Pirozzi and Zaccagnino, 2018) and (Ciuha, Klemenc and Solina, 2010) discuss visualising musical notes using the circle of fifths, by assigning notes which are a perfect fifth apart similar colours.

A colorful circle with letters and numbers

Description automatically generated

Figure ‑ Circle of thirds assigned to colour wheel used in (Ciuha, Klemenc and Solina, 2010)

Ciuha et al. presented a method of visualizing concurrent (played simultaneously) tones using a colour wheel where colours become increasingly unsaturated towards the centre. Their goal was the same as mine; to help untrained people understand harmonic relationships in music by interconnecting similar aspects of music and visual perception. The basis for their solution was the fact that concurrent tones are not perceived as entirely separate, and so a colour can be assigned to an entire group of concurrent tones. A tone is represented by a vector whose length is proportional it’s volume and direction points to that tone in the circle of thirds. This vector is then applied to the colour wheel above to determine the colour to represent the tone with. The colour of a group of tones can be calculated by adding the vectors of each individual tone and normalising the result. The result of this method is that dissonant chords are given a colour with low saturation while consonant chords have highly saturated colours. A major strength of this method is that volume is considered when assigning a colour to a tone, meaning that louder tones have more influence on the colour of a group of concurrent tones.

Malandrino et al. conducted a usability study in which they received positive feedback on their visualization tool. Included in the study’s sample were three participants afflicted with colourblind deficiency, and their responses to the questionnaire used in the survey suggested that their tool could include a colourblind mode which assigns notes to colours which are easier for colourblind people to differentiate between.

An important harmonic visual tool is the Tonnetz grid. Created by Leonhard Euler in 1739 by unrolling circles of fifths and stacking them on top of each other so that a major triad chord is represented by an upwards pointing triangle and a minor triad is represented by a downwards pointing triangle. This provides an intuitive geometric method of playing chords that can be understood without understanding the intricacies of harmonic relationships between notes. An interactive Tonnetz is available at https://girlinbluemusic.com/interactive-tonnetz/. This tool also uses colour in its implementation by assigning notes with similar pitch to similar colours. This application of colour seems to only serve an aesthetic purpose, as little harmonic information can be extrapolated from the colours of the notes in a chord.

A diagram of a triangle with letters and numbers

Description automatically generated with medium confidence

Figure ‑ Tonnetz grid with C major triad highlighted. Taken from <https://www.researchgate.net/figure/Tonnetz-grid-as-proposed-by-Riemann-12-in-19th-century-Highlighted-is-the-C-major_fig2_321905982>

(Ashton, 2018) focussed on the intuitive and visually pleasing Lissajous Figure and harmonograph visualization of intervals, which use a two-dimensional plane and plot the frequency of one note in one axis and another in the other axis. This results in consonant intervals, where the frequencies involved are at a ratio of small integers, to produce simple, visually pleasing patters, while dissonant intervals produce messier more complicated patterns. This book was a valuable resource when researching for this project because it is similar in tone and ambition to my application and has a similar narrative structure.

## Git Repository

https://git.cs.bham.ac.uk/projects-2022-23/oxg053

# Design

## Methodology

### Max

Max, also known as Max/MSP/Jitter, is a visual programming language developed by software company Cycling ’74 for music and multimedia projects, typically used to create instruments and effects for Ableton via the Max for Live platform and to program audio responsive visuals to accompany live music performances or create music videos.

Max was chosen as the language to create the software with for a few reasons:

* My own personal interest in using it recreationally, to create effects and instruments to use in my own Ableton productions and performances, and audio-reactive video synthesis inspired by the likes of artist Weirdcore.
* The graphical interface Max patches are created in bear a resemblance to Unreal Engine blueprints. Unreal Engine is a free to use popular video game engine and real-time 3D creation platform owned by Epic Games which I am very interested in using for creative projects in the future both recreationally and potentially as a career.
* I was enthusiastic about programming in an unconventional way which was new to me as means of developing my skills and gaining a new way of approaching programming. Max’s active user community and comprehensive documentation made this a viable alternative to programming the application in a language I was already familiar with.
* MIDI (Musical Instrument Digital Interface) integration meant making the application compatible with a MIDI keyboard for user input was very easy.

Max programs (called patches) are created using object boxes connected by virtual patch chords, intended to simulate a modular analogue synthesizer. Each object represents and provides a specific function or process. Depending on the function of the object, it may have various inlets and outlets to receive inputs and send outputs to other objects via patch cords. To conclude this section, I will give a brief explanation of some of the features of Max to help the uninitiated reader understand the code presented later in this report.

The data types in Max are:

* Integers
* Floats
* Symbols, which serve the same purpose as strings in other languages such as Java.
* Lists
* Signals, such as audio
* Bangs, used to instruct an object to perform its primary function with whatever information is available.

Data is usually stored in message objects and can then be sent to objects by either clicking the message box or sending a bang to its left inlet. The right inlet of a message box can be used to set the message of a message object without causing output. Messages also have the capability to handle changeable arguments. A dollar sign followed immediately by a number in the range 1-9 represents a changeable argument, and the argument will be set as the element at that position of a list given to the left inlet of the message box. This is primarily used to dynamically change an object’s attributes. Different objects have different attributes depending on their function, and these arguments are set by sending the object a message containing the name of the attribute followed by the arguments of the attribute.

A screenshot of a computer

Description automatically generated

Figure ‑ Setting the background colour of a live.scope~ object using a message with changeable arguments.

Some objects also have similar changeable arguments, notably the expr object used for evaluating expressions and as variables in if objects, used to evaluate inputs according to a conditional statement. In these situations, the changeable argument is denoted by a dollar sign, immediately followed by either the letter i, f, or s to indicate whether the argument is an integer, floating-point number, or symbol respectively, followed immediately by a number to indicate which inlet the argument is received into.

A screenshot of a computer

Description automatically generated

Figure ‑ An expression object evaluating the expression ($i1 + $i2) / 2, where $i1 is the integer received into the left inlet, and $i2 is the integer received into the right inlet.

An important thing to note in the above figure is that the left inlet of the expr object is a ‘hot’ inlet, and the right inlet is ‘cold’, denoted by the red and blue colours of the inlets respectively. This means that only a change of $i1 will cause the object to output the evaluated expression.

One of the most important objects used in the application is the mtof object. This converts MIDI key numbers into frequencies using three attributes: the MIDI key value of the reference note, the frequency of the reference note, and the scale data used to find the rest of the frequencies using the reference note. Throughout the application, the reference note of every mtof object is A4, which has MIDI key number 69. The frequency of this note is set as 440 Hz, the standard used to tune instruments today. The scale data is given as a pair of numbers for each degree of the scale used, representing the numerator and denominator of the ratio/fraction to multiply the root frequency by to obtain the frequency for that note. This is explained in more detail in section 3.2.8 of this report.

### Planning

I chose to employ an agile workflow when developing my program for two primary reasons:

1. I came into this project with very little knowledge of music theory beyond studying it at GCSE level, an ABRSM grade 4 in jazz trombone, and teaching myself to produce dance music using Ableton when I have the time. At the beginning of the project period, I invested some time into researching the relevant theory and formulated a preliminary plan of which concepts to include in the system and how to present them to the user, as well as in what order. However, over the course of the development process there were times when I found that my understanding of a concept was not adequate and I had to review my reading materials, which often led to me changing my plan of how to present the concept to the user.

2. I also had no prior experience of programming with Max and taught myself over the duration of the project using the official documentation, tutorial files, and community posts on Cycling ‘74’s public forum. Working iteratively allowed me to continually refine the components of the application as I improved my proficiency in Max and was able to identify and implement more efficient solutions than I had initially employed.

## Requirements

### Functional Requirements

1. The application must be available as a standalone and not require any additional software.
2. The application must not require an internet connection to run.
3. The application should always open on the title page.
4. When the application is opened all outputs should be empty.
5. Audio should be switched on when the application opens.
6. All note inputs can be given by a MIDI keyboard or a clickable on-screen keyboard.
7. On-screen keyboards which display 25 keys or fewer should display keys in the range 48-72 to match with the standard key range of a smaller MIDI keyboard.
8. Input from a MIDI keyboard should only affect the currently visible page.
9. Audio should only be output from the currently visible page.
10. Buttons which would not influence the state of the application (e.g., clicking the next page button when on the last page) should appear different to buttons which do.

In addition to these requirements, the user must be able to:

1. Control the audio volume of the application from every page.
2. Mute or unmute the audio of the application from every page.
3. Navigate one page backwards and forwards through the system with one click.
4. Navigate to any page of the system with one click.

### Non-Functional Requirements

1. The application should take no longer than 3 seconds to open.
2. The application should not crash on any modern computer.
3. The application should be compatible with all MIDI keyboards.
4. The application should take less than half a second to calculate intervals on any modern computer.
5. Audio from the application must not clip.
6. The volume level of the application should be the same on every page of the system.
7. All text should be a legible size and font.
8. The UI should be minimal to ensure it doesn’t distract from the content of each page.

# Implementation

## User Interface

Max is not a language typically used to make programs with a user interface except when using the Max for Live platform to create instruments and effects for Ableton Live, in which case the main consideration when creating a patch’s UI is for it to match the aesthetic of Ableton. For example, see below the UI of an LFO (Low Frequency Oscillator) effect written in Max which comes included in Ableton Live.

A screenshot of a computer

Description automatically generated

Figure ‑ LFO effect made in Max for live by Ableton.

Early in the project, it was considered to create the entire program as a Max for Live instrument, so that if someone was about to start producing a piece of music in the DAW but was unsure of how to choose which notes to use, they could first run through the program in the same environment. A prototype of how that might have looked was created and is shown below.

A screenshot of a computer

Description automatically generated

Figure ‑ Interval ratio calculator and visualiser Max for Live device Prototype.

It was eventually decided not to create the program as a Max for live instrument for two reasons:

1. Max for live device user interfaces are subject to a height restriction, to allow them to fit in the window at the bottom of Ableton’s UI. This would have meant that not as much information could be shown to the user.
2. Max for Live does not include all the objects that can be used in a standalone Max-patch, which would have severely restricted the functionality I could have included in the program.

The user interface was instead implemented as the presentation mode of the patch. I approached designing the program as an exploration of how to best explain the concepts involved in the theory behind the project, and as such the UI was not the main focus. As such, Max’s default colour scheme and graphics were used for most objects in the UI. When creating the application’s UI most consideration went to making sure the UI didn’t distract from the content of each page.

Each page is structured from top to bottom as: the page title, a body of text explaining a concept, and an interactive activity which builds the user’s understanding of the concept. This ensures that the user reads the page title first, drawing them into the content of the page and giving them an idea of what the keywords to pay particular attention to on the page are. It is vital that the user reads the body text of a page before engaging with any of the interactive activities on the page so that they can understand how to interact with them and how they should comprehend and interpret the feedback interacting with them gives. This is why the text correlating to an activity is always positioned above the activity.

A screenshot of a computer

Description automatically generated

Figure ‑ The first screen of the user interface.

Each page is included in the main patch as a subpatch and is given the same coordinates and dimensions. Also, UI objects in each subpatch such as headings and body text were given the same coordinates, font, and font size. This ensured a smooth transition between pages. The title of each page has a font size of 35 pts to make clear the purpose of each page and allow for easy navigation of the system. All body text is 14 pts Ariel to ensure that it is easy to read while allowing lots of information to fit on each page. All buttons were given the same font, font size, and roundness of box edges.

Care was taken to ensure that the text colour of buttons indicates whether clicking them influences the state of the program or not. For example, in the Pythagorean Scale page, the text of the ‘restart’ button is grey when the animation is on the first frame and light blue otherwise, as clicking the button in the first case has no effect on the state of the program. It was important that the text colour of all “clickable” buttons was the same throughout the system to emphasise this. This was done as clicking a button and not seeing any effect can be frustrating to the user.

The graphics for the hammers on the ‘Pythagoras’ Hammers’ page, and the string and feather on the ‘The Overtone Series’ page were found on the internet. Similarly, the kick drum sound used on the ‘Frequency’ and ‘Polyrhythms’ pages and the hammer sound on the ‘Pythagoras’ Hammers’ page were found online, and the plucked string sound on the ‘The Overtone Series’ page was sampled from an Ableton guitar instrument. Were the program to be shipped commercially in the future, it would be desirable to create original graphics and sound effects to avoid any copyright issues.

The sliders on the ‘Frequency’ and ‘Polyrhythms’ pages had their knob shape set to ‘Indicator’. This was the most appropriate shape as it made clear that the frequency was being set as a discrete value rather than an accumulation of values.

## Pages

The application is structured in a modular fashion, split into 12 self-contained patches, each presenting an individual concept like a page in a book. This approach was taken so that the application would function like a lesson, introducing concepts at a carefully thought-out pace so that each one will be fully understood by the user before it is used to explain further concepts.

Each patch is set to “Open in presentation” and included in the main patcher as bpatcher objects, described in the Max documentation as “Embed a subpatcher with a visible UI”. This allowed me to conveniently replace pages with newer versions and insert new pages where I felt necessary. The main patcher provides navigation tools to change which bpatcher is visible to the user, and controls to turn audio on or off and change the system’s volume.

This section of the report explains what each page attempts to teach and the method chosen to do this. The parts of program which were more challenging to implement are explained at a lower level, with sections of the patch included as figures. Because Max is not a well-known language, the code has been explained in detail in parts so that it can be understood by people not familiar with Max. Each time an object is first mentioned, I have included the object’s description from Max’s documentation, as found at the bottom of the objects reference tab. All object descriptions, as well as additional patch images can be found in the appendices of this report.

### Main Page

This page is designed to provide a user interface allowing the user to navigate the system and control the audio coming from the system.

A screenshot of a computer

Description automatically generated

Figure ‑ Main page patch.

bpatchers output audio as either a signal from an outlet, or as a MIDI note number from one outlet and a velocity value (0-127) from another.

A screenshot of a computer

Description automatically generatedIn the case where a bpatcher outputs a MIDI note number and velocity pair, each value is sent to a *gate* object.

Figure ‑ MIDI note messages are gated so that only the currently open page outputs note sounds.



Both gates have the following arguments:

* outlets (number of outlets): 1
* initial-state: 1 (open)

The page number is received by a receive object and passed into an *if* object.

The condition of the if object is given as:



so that the gate is only open if this page is the currently visible one.

If the page is the one currently displayed to the user, the MIDI note number and velocity value are passed to a *noteout* object.



By default (Windows), this transmits the MIDI info to Microsoft GS Wavetable Synth, and causes a simple piano sound to play the selected note(s).

A screenshot of a computer

Description automatically generatedA similar algorithm controls the signal output of bpatchers, only a *selector~* object is used in place of the gate object, and the output of this selector~ is a send object named “sound”.

Figure ‑ Audio signals are routed through a selector~ object so that only the currently open page outputs audio.

This ensures that only audio from the currently open page is output by the application.

A screenshot of a music player

Description automatically generatedThe signal from whichever page’s selector~ object is currently open is sent to a receive object, and from there through a *live.gain~* object to an *ezdac~* object.

Figure ‑ Volume and audio on/off controls for the whole application.





The audio controls are located at the bottom of the page, to the right of the bpatcher window.

In an earlier prototype of the patch, each bpatcher’s output signal was sent to its own inlet of a single selector~ object, and the open inlet was chosen by the current page number and that signal was routed directly to the live.gain~ object. However, an early issue with the program was that the noteout objects that produce piano sounds for the 2nd, 3rd, and 12th pages were initially included in their respective subpatches. This meant that pressing a key on a MIDI keyboard on any page of the system would produce the piano sound. To solve this, the noteout objects were moved to this patch and the pitch and velocity values were sent through outlets of the subpatches, so that they could be routed through gate objects. This was why the if objects were initially added for these subpatches, and the decision was made to add a corresponding if object for every subpatch and to use this to control a selector~ for each individual subpatch, as this made the patch more elegant and easier to understand.

### Title Page

The title page of the system serves only as an introduction to the system and gives instructions on how to navigate the system and control audio with the live.gain~ and ezdac~ objects.

This page contains only comment objects to present text.

### Intervals

This page is the first to provide the user with any of the theory fundamental to understanding musical harmony and it is the natural first step in teaching harmony to introduce the concept of an interval. Without comprehending the meaning of the word interval in a harmonic sense, a student would be unable to grasp any of the further concepts of musical harmony, and it is for this reason that this page serves as a gentle introduction to the application and theory therein.

From a pedagogical point of view, it is important to ease the student into the content of a topic by introducing concepts one at a time to ensure they understand which are the most important concepts and don’t become overwhelmed, and so they fully understand a concept before it is used to explain another concept. This is why this page displays a minimal amount of text and only one activity, with user interaction resulting in one visual output at a time.

A computer screen shot of a keyboard

Description automatically generated

Figure ‑ The Intervals page displaying the result of the user clicking a C and G on the keyboard.

On this page the user enters a pair of notes. When the first key is pressed the selected key will be highlighted, and the corresponding note heard as though the key were pressed on a real piano. When the second key is pressed, both notes will be heard together, and the name given to the interval between the pair of notes is displayed below the keyboard.

The biggest challenge faced when implementing this page came from ensuring that both notes selected by the user are heard together and are highlighted at the same time on the on-screen keyboard. This was first attempted by adding the output of the keyboard to a list and when the length of the list reached two, sending it back into the keyboard as a chord message. This resulted in a stack overflow error as the notes in the chord message would get stuck in an infinite cycle. The final implementation of this function uses two different methods of achieving the desired result depending on whether input is given by a MIDI keyboard or the on-screen keyboard.

A screenshot of a computer

Description automatically generated

Figure ‑ Intervals page patch.

The pair of notes are inputted from either a connected external MIDI keyboard or a *kslider* object, plays them together, and displays the interval between them.

The kslider object can be used for input or to display which keys have been pressed on a MIDI keyboard. The kslider has been set to polyphonic mode, so that the two keys the user selects are highlighted at the same time. In this mode, kslider can be told to play chords by passing it a *chord* message which must be in the following format:



Where ki is the MIDI key number of the ith note in the chord, and vi is the velocity the ith note of the chord is to be played at.

In polyphonic mode, a selected key will stay highlighted on the kslider until either it is selected again or the kslider receives a *clear* message.

The kslider is set to have 13 keys, with the lowest key outputting MIDI key number 48 when pressed.

When using a MIDI keyboard, the key pressed by the user and the velocity at which they pressed it are received through a *notein* object.

notein outputs a MIDI key number the velocity the key was pressed with when the user presses a key, and the MIDI key number and a 0-velocity value when the key is released. These are called note-on and note-off messages. The pitch and velocity outlets of the notein object are connected to the respective inlets of a *stripnote* object to filter out the note-off messages.



The output of the stripnote object goes to two objects from here:

1. the kslider object, so that the pressed key is highlighted and the selected key’s MIDI number is passed to a *makenote* object.



This makenote object has the following arguments:

* + Velocity: 127
  + Duration: 500

The pitch and velocity outputs from this makenote object are connected to their respective outlets to be received in the main patch.

1. A *zl.group* object with maximum list length 2.

This outputs each consecutive pair of MIDI key numbers it receives from the stripnote object.

The pair of MIDI key numbers are sent as arguments to a message object containing the text

and this is passed to the kslider object.

The kslider outputs the MIDI key numbers of either the key the user has clicked, the key the user has pressed on their MIDI keyboard, or each of last two keys pressed (from the *chord* message), and these are sent to another zl.group object with maximum list length 2. Each time two keys have been inputted, this zl.group object will send the MIDI key numbers of the keys to an *unpack* object.

This splits the pair into two individual MIDI key numbers, which are both passed to a makenote object with arguments:

* + Velocity: 127
  + Duration: 1000

The pair of MIDI key numbers is also sent to a *–* object to find the difference between them, and the absolute value of the difference is calculated by passing the result through an *abs* object. This finds the number of keys the higher note is above the second note. This value is converted to the name of the corresponding interval (INCLUDE COLL IN APPENDIX) using a *coll* object.



The name of the interval is displayed to the user in a comment box, positioned centrally below the kslider in presentation mode. This is done by taking the output of the coll object as an argument to a *set* message so that it is prepended to the message “set “.

When the lower zl.group object outputs a pair of MIDI key numbers, this output is converted to a bang by passing it to a *button* object.



This bang is delayed by a second using a *delay* object with delay time set to 1000ms.



This bang does two things:

1. Triggers a message object to send a *clear* message to the kslider to un-highlight the selected notes after a second.
2. Trigger a message object to send a *set* message to the comment object so that the interval name of the previous pair of notes is cleared after a second.

In later pages zl.stream is used instead of zl.group to accumulate and output pairs of MIDI note numbers. The difference between these objects is that after the first two inputs, zl.stream outputs the last two inputs for every new input it receives, whereas zl.group outputs each *new* pair i.e., every two keys that are pressed correspond to one output. zl.group was used in this patch so that to see the name of an interval two keys must be pressed as feedback showed that this was the natural way users would expect it to at this stage. Later pages use zl.stream as once the user is more familiar with the application and concepts involved, it becomes more intuitive and natural to see the information for each successive interval. It is also necessary to use zl.stream when analysing the intervals in a sequence of more than two notes.

### Consonant vs Dissonant Intervals

This page offers the same interval-calculation functionality as the previous page whilst also introducing the notions of consonance and dissonance. A button can be clicked to play a consonant pair of intervals and another button plays a dissonant pair of intervals. The pairs of notes in either interval were chosen as particularly blatant examples of consonance and dissonance to make clear to the user what each means.

A screenshot of a computer

Description automatically generated

Figure ‑ Code for buttons which play a consonant or dissonant interval.

Three *cycle~* objects generate cosine waves with frequencies 440 Hz, 660 Hz, and 495 Hz respectively.



The signals outputted by these cycle~ objects are sent to the input inlets of a selector~ object like so:

* Input 1 receives a cosine wave cycling at 440 Hz and another cycling at 660 Hz. This is a perfect fifth interval.
* Input 2 receives a cosine wave cycling at 440 Hz and another cycling at 495 Hz. This is a major second interval in Pythagorean tuning.

The selector object initially has both inputs closed. The “Play Consonant Interval” textbutton causes a message object to send the message “1” to the selector~, opening the first input. The “Play Dissonant Interval” textbutton causes a message object to send the message “2” to the selector~, opening the second input.

Rather than just playing the two notes corresponding to which button has been pressed at a constant volume, perhaps until a ‘stop’ button is pressed, the amplitude of the signal outputted by the selector object is modulated to have an ADSR (attack, decay, sustain, release) envelope, to make it sound more like a note generated by a musical instrument.

This is done by multiplying the signal by the output of a *line~* object.



The ramp time of this line~ object comes from a *function* objectThe points in the function object were chosen to mimic a conventional synthesiser ADSR envelope.

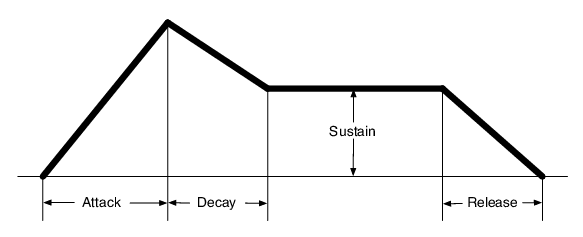


Figure ‑ ASDR envelope diagram from https://www.researchgate.net/figure/The-elements-in-an-ADSR-envelope\_fig22\_270819567

The points of the function object are output in line format via it’s second-leftmost outlet when the object receives a bang in its leftmost inlet. To achieve this, both the “Play Consonant Interval” and “Play Dissonant Interval” textbuttons are also connected to the leftmost inlet of the function object.

The amplitude modulation is done by passing the cosine waves to the left inlet of a *\*~*  object and using the line~ object’s output ramp as the right operand of this operation.

The resulting signal is sent to the patcher’s outlet.

After playing an interval on a MIDI keyboard or the kslider object, the user can click a button if they think the interval is consonant, or another button if they think the interval is dissonant. After the user clicks one of the buttons, a message will tell them whether they are correct or incorrect. This feature wasn’t part of the initial plan for this page but was added after feedback suggested that more interactivity could be included here to increase user engagement.

This patch uses nearly the same algorithm to calculate and display the name of the interval between the two keys entered by the user. However, in this patch the kslider has 24 keys and so the difference between the MIDI key values of the two keys pressed is reduced MOD 12 using the % object with argument 12 before being passed to the coll object. This means that if the two notes are more than an octave apart, the interval name displayed will be the same as if the notes were in the same octave. For example, if the two notes entered are a major ninth apart, the displayed interval would be the major second.

The lookup table the coll object uses is also slightly different, so that the entry for each interval also contains either the word “Consonant” or the word “Dissonant”, depending on whether the corresponding interval is generally considered or dissonant.

Finally, the user can click a play button to play an excerpt from Chopin's Nocturne in C sharp minor, which is an example of using dissonance to create tension and releasing this tension with consonance. This was added after one of the people I asked to use the application suggested that an example would demonstrate how dissonance can be used deliberately in a piece of music.

### Frequency

This page explains the notion of frequency by likening the frequency of a sine wave to the periodicity of a looped kick drum sample. When the frequency is increased to a point where the sine wave is audible and the kick drum becomes a tone with a discernible pitch, the two will have the same pitch. This is largely used as setup for the Polyrhythms page that comes later in the system. This was inspired by Adam Neely’s YouTube video titled “Harmonic Polyrhythms Explained! [ AN's Bass Lessons #27 ]”, where he similarly speeds up a looped kick drum pattern, but allowing the user to control the speed themselves improves the learner’s understanding.

A screenshot of a computer

Description automatically generated

Figure ‑ Frequency page patch.

The slider sets the frequency of a cycle~ object and this is used as the sine wave sound the user hears.

The kick drum sample is played using a *sfplay~* object.



The kick drum sample is stored as DubK3DHit-01.wav file in the same folder as the main patch file and is loaded into the sfplay~ object using a *loadmess* object.



The message that the loadmess object sends to the sfplay~ is “open DubK3DHit-01.wav”.

Once the file is loaded into sfplay~, it is played when the sfplay~ receives a 1 into its left inlet, since it is the first (and only) file in its queue. This 1 comes from the same cycle~ object that produces the cosine wave that the user hears. This ensures that the sine wave and kick drum repeat at the same frequency. However, the cycle~ object never outputs 1, so its output must be doubled using a \*~ object with the argument 2. This has the effect of doubling the amplitude of the cosine wave so that it now reaches 1 but has no effect on the frequency so the kick drum still plays at the same frequency as the cosine wave.

When the kick drum sample is looped at a high frequency, we only hear the start of the sample, which results in the tone that is produced being very quiet since the amplitude of the sample waveform is small at the beginning of the sample. This is negated by multiplying the output of sfplay~ by the output of the frequency slider. The result of this is that at high frequencies, the sound of the looping kick drum sample is far louder than the cosine wave and is very unpleasant to listen to. This was solved by passing the signal through a *omx.peaklim~* object.

This means that the output of the looping kick drum never exceeds the volume level the user has selected with the live.gain~ object in the main patch.

The output of the selector~ object which controls whether the cosine wave or kick drum is sent out of the patch is also passed to a *live.scope~* object to visualise the signal currently passing through.



### Pythagoras’ Hammers

A screenshot of a computer

Description automatically generated

Figure ‑ Pythagoras' hammers page patch.

This page introduces the concept of consonant intervals resulting from two notes whose frequencies are at a ratio of small integers using the famous analogy of Pythagoras hearing hammers whose weights have a ratio of 3:2 striking an anvil. This is an often-used analogy as it interconnects the seemingly abstract concept of frequency with the tangible and familiar measurement of weight, and because it is essential to understanding one of the most culturally significant tuning systems in western music: Pythagorean tuning.

The hammers are implemented using *pictctrl* objects.



The image files used for the pictctrl objects must be in a specific format:

A screenshot of a cell phone

Description automatically generated

Figure ‑ The format needed for images used for a pictctrl button.

Only the top row is necessary since the button doesn’t have an inactive state or mask. The image used was therefore:

A hammer hitting a bowl with a bowl

Description automatically generated

Figure ‑ Formatted images for use with pictctrl button. When not clicked, the button displays the image on the left. When clicked, the button displays the image on the right.

The sound produces when the user clicks either of the hammers comes from the file anvil-hit.mp3. This mp3 file is played using a sfplay~ object much like the kick drum sound in the frequency page.

The sound produced when the smaller hammer is clicked is pitched up by a perfect fifth by sending the corresponding sfplay~ object’s output to a *pitchshift~* object which is in turn sent the message “pitchshiftcent 702”.

This shifts the pitch of the mp3 file by 702 cents. The number of cents is calculated as

This achieves the frequencies played by clicking the hammers having a ratio of 2:3.

A textbutton labelled “Together” can be clicked to play the original sound and the pitch-shifted sound at the same time. This is achieved by sending the bangs outputted by the textbutton to a pair of *toggle* objects.



The bang from the textbutton is also sent through a delay object with delay time 100ms. The bang is held for 100ms and then sent to the two toggles. This has the effect of switching the toggles on when the Together button is clicked, switching the pictctrl buttons to their “on” state, and then switching the toggles back off 100ms later, switching the pictctrl objects back to their “off” state.

### Polyrhythms

This page further clarifies the meaning of frequency ratios by returning to the analogy of a repeating kick drum. It also introduces the interval of the octave, with a frequency ratio to the root of 2:1.

A screenshot of a computer

Description automatically generated

Figure ‑ Frequency control on the Polyrhythms page.

The frequency that the sample loops at is given by the user using a slider object with output minimum 1 and range 440. This is sent to the three oscillators using a send object named freq.

The frequency each oscillator cycles at is displayed to the user by multiplying it by the relevant coefficient and sending the result as an argument to a message object with the message “set $1 Hz”.

A screenshot of a computer

Description automatically generated

Figure ‑ This kick drum loops at a frequency 1.5x higher than the first on this page, so that when they are both sped up, they produce tones at an interval of a perfect fifth.

The looping kick drum sample is created in the same way as on the frequency page. This is done three times; once to play the sample at the given frequency, one at 1.5 times the given frequency, and one at double the given frequency. This achieves three rhythms whose frequencies are at a ratio of 1:1.5:2 which is the same as 2:3:4.

Each signal is sent to its own live.scope~ object so the soundwaves of each looping kick drum can be visualized and compared.

The looping samples can be switched on and off using their corresponding Play/Pause textbutton. These textbuttons are set to toggle mode, so that they display “Play” when in “off” mode, and “Pause” when in “on” mode. The output of these textbuttons is used to open and close a gate which will either allow the signal of the looping sample through or not depending on the button’s state.

A glaring issue with this page of the program is that when more than one rhythm is playing together, the rhythms are slightly out of sync. A solution to this issue was attempted by using *metro* objects instead of cycle~ to play the sample.

This proved more effective for a simple 3:2 polyrhythm, but after adding the slider for tempo modulation, it was found that the extra processing caused the bangs sent by the metro objects to become out of sync. Because using metro instead of cycle~ didn’t solve the sync issue and required more calculation to set the time interval of each metro, the final program uses cycle~ objects.

### Pythagorean Scale

This page demonstrates how the ratios and frequencies of the Pythagorean scale can be calculated using the ratios 3:2 and 2:1 by showing the process of starting from A at 440 Hz, moving up in perfect fifths by multiplying the frequency by 1.5, and moving down in octaves by halving the frequency where necessary so that all the frequencies are between those of the starting note, A (440 Hz), and the A one octave above (880 Hz). The process is visualised on a keyboard and as an animation showing the construction of the circle of fifths, a fundamental part of harmonic music theory.

A circular chart with numbers and letters

Description automatically generated

Figure ‑ The final frame of the animation, displaying the circle of fifths with the ratio of each note's frequency to A in Pythagorean tuning.

Each of the 12 notes in the chromatic scale are assigned a colour by overlaying the circle of fifths over the colour wheel. Notes adjacent to each-other in the circle of fifths sound most consonant when played together, so they are assigned similar colours. Notes on opposite sides of the circle of fifths sound most dissonant when played together, so they are assigned complementary colours. These colours are represented as vectors and combined using vector addition to assign a colour to a pair of notes later in the system, but they are introduced here to give a visual aid to emphasise how the perfect fifth is the most consonant interval after unison and the octave and to build familiarity with the colour assignment in the user before it is used later.

A screenshot of a computer

Description automatically generated

Figure ‑ The animation is played using a metro object and a counter object. This is the typical implementation of iteration in Max.

The animation has 19 individual frames. To start the animation the user clicks a textbutton, set to toggle mode and labelled “Play” when in “off” mode, and “Pause” when in “on” mode. The textbutton is used to start/stop a metroobject. The metro object’s time interval argument is set as 1500ms.

The metro’s bangs are sent to a *counter* object.



The counter is set to count from 1 to 19 by incrementing by 1 every time it receives a bang.

A textbutton labelled “Restart” sends a bang to the second right-most inlet of the counter when clicked. This causes the counter to reset to 1.

The current count outlet of the counter object is sent to an if object which causes to the text of the Restart textbutton to be greyed out if the current count is 1 i.e., if the animation is on the first frame.

A screenshot of a computer

Description automatically generated

Figure ‑ The note numbers of each note in the process are stored as a list and converted to frequencies using an mtof object.

Each tick of the metro causes a key to be highlighted on a kslider object set to polyphonic mode and cause the corresponding note to play out of the speakers.

The current count of the counter is sent to a route object with one selector, set as 1. This means that when the counter outputs 1, a bang is sent from the leftmost outlet of the route object which sends a clear message to the kslider, clearing any highlighted keys.

The route object also sends a bang to a message box which sends 0 to a selector~ object. This closes the selector’s inlet and stops audio being sent out of the patch’s outlet.

If the counter’s current count is not 1, it will be sent out the right outlet of the route object. This is used to open the selector’s inlet so that audio is sent out the patch’s outlet, and to iterate through elements of a list and send the result to the left inlet of the kslider. The list is stored in a message object and the iteration is done using a *zl.nth* object.



1 is subtracted from the output of the route object, so that when the current count of the counter is 2, the first element of the list is extracted and sent to the kslider. The elements of the list are MIDI key numbers in the order they should be played as part of the animation.

When the kslider receives a MIDI key number from zl.nth it highlights that key and sends the MIDI key number to an *mtof* object.



The scale argument of the mtof object is set using a loadmess object. This sends the message “scale 12 256. 243. 9. 8. 32. 27. 81. 64. 4. 3. 729. 512. 3. 2. 128. 81. 27. 16. 16. 9. 243. 128. 2. 1.”

This represents a scale in Scala format and can be interpreted as a scale with 12 notes where the second degree of the scale’s frequency has a ratio of 256:243 to the tonic, the third degree has a ratio of 9:8 to the tonic and so on.

The frequency reported by the mtof object is sent to a cycle~ object, and the signal generated is sent through the selector~ object to the patch outlet. The frequency value is also set as the text of a comment object in presentation mode.

A screenshot of a computer

Description automatically generated

Figure ‑ Each frame of the animation is an image file bundled with the application and displayed using an fpic object.

The output of the counter is sent to a *sel* object to control the animation.

The sel object has 19 selectors, corresponding to each frame of the animation. For example, when the metro outputs 1 the sel object will send a bang out of its left-most outlet.

Each frame of the animation is stored as a PNG file in the application’s folder. The frames are displayed using an *fpic* object.



When the metro outputs the number of the current frame, the sel outputs a bang out the corresponding outlet, which sends a message containing the name of the image file for that frame as an argument to a *pic* message. This updates the image the fpic displays to that of the current frame.

Including every image in the animation as a separate fpic object and stacking them on top of each other, then making them visible one at a time, would allow for every image to be embedded in the patch so that they wouldn’t need to be included in the application folder. However, since the images are PNGs with transparent backgrounds, and the text in the images isn’t aligned for each image, this would cause the text to overlap and be very messy and hard to read. This could be solved by hiding each image when the next frame is made visible, but this would have made the patch very messy and complicated, so it was decided to just use one fpic object and change the image it shows using pic messages.

An earlier prototype of this page included a slider that was used to set the time interval of the metro object so that the user could change the speed of the animation. However, this was omitted from the final patch because it made the UI to crowded.

### The Pythagorean Comma

This page explains the discrepancy between the theoretical result of stacking 12 perfect fifths i.e., returning to the starting note, and the actual result obtained when using the perfect fifth with a ratio of 3:2. This small but significant discrepancy is known as the Pythagorean comma and is the reason we rarely use Pythagorean tuning today. Describing the comma mathematically does not suffice to emphasise its importance, so this page allows the user to click buttons to play two notes at an interval of the Pythagorean comma to demonstrate that it is an audible difference.

A screenshot of a computer

Description automatically generated

Figure ‑ The Pythagorean Comma page patch

The sound for the two notes is produced by a cycle~ object. When the textbutton labelled “440 Hz” is clicked, 440 is sent to the cycle~ object. The frequency of the note a Pythagorean comma above 440 Hz is calculated as

This calculation is done using an *expr* object.

Powers are calculated in an expr object using the pow operator.

The expr object outputs the result of the expression when the textbutton labelled “446 Hz”, and this is sent to the cycle~ object.

The output of the cycle~ object is sent to the right inlet of a selector~ object. The input of the selector~ object is closed when the program opens using a loadmess object to send a 0 into its left inlet. Clicking the Stop button also triggers this loadmess object to output. Clicking either the 440 Hz button or the 446 Hz button causes a message object to send a 1 to the selector~’s left inlet, allowing the output of cycle~ to pass through to the patch’s outlet.

### The Overtone Series

This page explains the overtone series using the analogy of a vibrating string. A feather can be moved along the string to partition the string into two lengths that vibrate separately. This is a common analogy used to explain overtones as the correlation between the length of the string and the frequency it vibrates at when plucked e.g., halving the length of the string causes it to vibrate twice as fast.

A computer graphics of a computer program

Description automatically generated with medium confidence

Figure ‑ The implementation of changing the length of a vibrating portion of a piece of string.

The string and feather were implemented using a *pictslider* object which outputs integers in the range 0-600.



The value given by the slider is sent around the patch with a send object named sVal.

The string on either side of the feather can be “plucked” by clicking a *ubutton* object which is positioned in front of the string in presentation mode.

The ubutton objects had to be dynamically re-sized when the user moves the feather along the string so that they are the same size as the string either side of the feather.

The left ubutton’s position should be constant, but its width must change to be the same width as the string to the left of the feather. This is done by passing receiving the pictslider’s value with a receive object and setting this as the width of the ubutton object. The range of the pictslider was set as 600 to match its width, so that moving the feather one pixel to the right causes the width of the left ubutton to increase by one pixel.

The ubutton on the right has dynamic width and x position. The width, w, and the x position, x, are calculated as

These values are collected in a list (w, x) using a *pak* object.



This allows them to be given as arguments to the message box containing the message “$2 313. $1 30.” Which is appended to the message “rect “, setting the ubutton’s coordinates as (x, 313), and its width and height as w and 30 respectively.

The feathers position can be set to all the way, halfway, a third of the way, and a quarter of the way along the string by clicking the corresponding textbuttons, which send the corresponding value to the left inlet of the pictslider.

A screenshot of a computer

Description automatically generated

Figure ‑ The sound played when one section of the string is clicked is pitch-shifted by an amount proportional the length of that section.

The sound produced by clicking either of the ubuttons is the guitar-a.mp3 file in the application’s folder. It is played using sfplay~ objects and pitch-shifted using pitchshift~ objects. Clicking the string to the left of the feather results in the sound file playing but pitch shifted up by

Clicking the string to the right of the feather results in the sound file playing but pitch shifted up by

The resulting audio is sent out the patch’s outlet.

A screenshot of a computer

Description automatically generated

Figure ‑ The ratio of the length of the section of string to the length of the entire string is displayed below that section.

The ratio of the length of the string on either side of the feather to the string’s full length is shown below the midpoint between the feather and the end of the string. To the left of the feather, the ratio is calculated by finding the highest common factor of the 600 and the slider value using the sadam.gcd object from the Sadam library. This is the only code used in the application that was not written by myself. The ratio is calculated using the algorithm:

and the ratio is then given by a:b.

The x position of the comment box that displays this ratio is calculated as

For the string to the right of the feather, the ratio is calculated as:

and the ratio is then given by a:b.

The x position of the comment box that displays this ratio is calculated as

Initially the string and feather were implemented using a pictctrl button to represent the string either side of the feather, so that the string could be animated to appear to vibrate by changing the image file the clicked pictctrl uses. This made the patch more immersive and made it very clear which length of string the user had clicked. However, this meant the slider had to be placed below the string unless a pictslider could be made with a transparent background and a knob image with a transparent background, but without the software capable of making these images, I decided the current implementation was more elegant. In a future implementation, it would be desirable to indicate which length of string has been ‘plucked’ with some kind of visual indication, such a vibration animation.

### Just Intonation

This page introduces the ratios of the Just scale, which are calculated by combining combinations of the ratios found in the overtone series and allows the user to visualise any of them with a Lissajous figure by clicking the button labelled with that ratio.

A computer screen shot of a computer

Description automatically generated

Figure ‑ The Just Intonation page patch.

The Lissajous figure is created by passing the output of two cycle~ objects into the left and right inlets of a *scope~* object.

There is a textbutton for each given ratio, and clicking one of these sends the numerator of the ratio to the cycle~ object that is plotted in the x axis of the scope~ object, and the numerator to the other cycle~ which is plotted in the y axis.

### Problems with Just Intonation

Just intonation is generally considered to be the most pure and harmonious tuning system because of its simple ratios and prevalence of intervals found in the overtone series. However, it has several issues that make it challenging to use in practice. For one, the Pythagorean comma is also present in Just intonation, and this makes it difficult to define how many notes are in a scale. More importantly, Just intonation is inherently tied to specific keys, specifically the key of the note used as the root when calculating the frequencies of the other notes in the scale. This means that when playing music that is written in a different key, or perhaps changes key midway through a song, or when playing in an ensemble with instruments that are tuned do a different key, the purity of intervals can be compromised.

This page visualises the second issue outlined above by providing a keyboard tuned using Just intonation and visualising the interval between successive notes using Lissajous figures. The ratios of the resulting intervals can be compared to their expected values (displayed at the bottom of the screen) so the user can see how they differ for intervals which don’t involve the root note (A).

A piano keyboard with a black and white keyboard

Description automatically generated

Figure ‑ The last two keys pressed are accumulated using a zl.stream object with argument 2.

The velocity and pitch outputs of a kslider are first sent to a stripnote object to filter out note-off messages. The MIDI key number of any key pressed is then sent to a *zl.stream* object.



The list length of the zl.stream object is set as 2. Once two keys have been pressed, zl.stream will output their MIDI key numbers as a list. Every subsequent time a key is pressed, zl.stream outputs the last two keys to be pressed.

The zl.stream sends a list containing the last two keys pressed around the patch using a send object named pair.

A screenshot of a computer

Description automatically generated

Figure ‑ The pair of note numbers is converted to the name of each note, and each of these is coloured with the colour corresponding to that note.

Each note entered by the user is assigned a colour. These colours are chosen by mapping the circle of fifths to the colour wheel. The pair of MIDI key numbers is put into ascending order of pitch using *zl.sort*.



The pair is then split into individual messages using unpack. Each MIDI key number is reduced MOD 12 so that notes that are octaves apart of given the same colour. Each number is then converted to an RGBA colour code using a coll item with the lookup table given by the file colour\_wheel.txt. The colour codes are appended to a *textcolor* message and sent around the patch using send objects named colour1 and colour2 for the higher and lower notes respectively.

A screenshot of a computer

Description automatically generated

Figure ‑ The colours of each note in the pair are combined by performing vector addition on their RGB codes.

A colour is also assigned to every pair of notes. This is calculated using vector addition. Each value in the RGB code of this colour is calculated by adding the corresponding values of each individual colour and normalising the result by dividing it by the maximum resultant length, namely 2.

The three values are collected in a list using a pak object, which has been set to have three inlets that accept floating-point numbers by giving it the arguments “f f f”.

The resulting RGB is sent elsewhere in the patcher using a send object named combColour.

This is a poor implementation of the method presented in (Ciuha, Klemenc and Solina, 2010), since the method they present performs uses the result of the vector addition to represent a point in a colour wheel which has no saturation in the centre, resulting in more dissonant pairs of notes being assigned a less saturated colour. This allows the user to interpret harmonic information from the colour assigned to the pair of notes, whereas little can be extrapolated from the colour calculated by my application. One possible solution to this problem would be to scale the alpha value of the combined colours proportionally to the length of the vector representing the pair of notes.

A screenshot of a computer

Description automatically generated

Figure ‑ The ratio of the frequency of each note in the pair is calculated using the subpatch ratio.maxpat.

The ratio of the interval between the pair of notes is calculated using the encapsulated sub-patch below. This was done because this algorithm was used several times in the program, so including it as a subpatch using the patcher object meant I didn’t have to write it more than once, and it also keeps the patch more organised and easier to understand.

A screenshot of a computer

Description automatically generated

Figure ‑ The subpatch ratio.maxpat. This calculates the ratio of the interval between a pair of given notes in a given tuning system.

The patcher takes the name of a lookup table .txt file and a pair of MIDI note numbers as arguments and outputs a list containing the numerator and denominator of the frequency ratio between the two notes in the given tuning system.

The entries of the lookup table are indexed from -12 to 12, corresponding to difference between two notes in semitones. Each entry contains the numerator and denominator of the frequency ratio of the note that many semitones above or below the reference note (always A4 at 440 Hz in this application).

The % operation works differently to the conventional mod operation when dealing with negative numbers, in the sense that a negative input will always result in a negative output. For example, with the conventional mod operation we would have that:

since

With Max’s % operation we instead get

because

The lookup table is read into two coll objects by appending the name of the text file to the message read.

The frequency ratio between two notes is calculated as the quotient between each note’s ratio to the reference note.

The pair of notes is split into each individual note using unpack. The difference in semitones from the reference note is calculated by subtracting 69 from each MIDI note value (69 is the MIDI note value of A4). This is then reduced to a number between -12 and 12 using the % operation with right operand 12.

These values are then input to the coll objects, which outputs the frequency ratio of each note to A4 in the given tuning system as a list containing the numerator and denominator.

The quotient between the two fractions is calculated by multiplying the numerator of the first fraction by the denominator of the second to be the numerator of the new ratio and multiplying the denominator of the first fraction by the numerator of the second to be the denominator of the new fraction.

Each operand for the \* operations must first be packed into a list using pak. This is because the left inlet of the \* is a “hot” inlet (denoted by the orange colour) and the right inlet of the \* is a “cold” inlet (denoted by the blue colour). This means only a change in the left operand of the operation causes an output, and so if the left operand is updated before the right operand, the output will be calculated using the old right operand. Packing both operands into a list and passing this to the left inlet of the \* object causes the output to update when either operand changes.

The new numerator and denominator are packed into a list using pak, and the list is then sorted into descending order using zl.sort with the argument -1. The argument -1 tells the zl.sort object to sort in descending order. This is done to always give the ratio in terms of the frequency of the lower note to the frequency of the higher note, as convention for this application. The list is then sent to the patch’s outlet to be received by the parent patch.

Each component of the ratio is displayed in a separate comment object, and the text colour of each is set using the RGB codes received from colour1 and colour2.

A screenshot of a computer

Description automatically generated

The pair of MIDI note numbers is converted to the corresponding frequency given by the just intonation ratios using a mtof object with scale argument given by the Just scale in Scala format. This outputs a fair of frequencies given as floating-point numbers. This pair is separated using unpack. The frequencies are sent to cycle~ objects to set the frequency of the cosine wave the cycle~ objects output. One of the resulting signals is sent to the left inlet of a scope~ object and the other is sent to the right inlet.

Each signal is also sent to the right inlet of a selector~ object. The selector~’s right inlet is opened whenever a key is pressed. There is a textbutton labelled “Clear note selection”, clicking this button sends a 0 to close the selector~’s right inlet, send set messages to all comment objects to clear their text, and send a *zlclear* message to the zl.stream object, so that two new keys must be pressed for any output to be given.

If the zlclear message wasn’t sent, when the user next presses a key, the system would output information about interval between this note, and the previous key pressed. This is not the result a user would expect from clicking a button labelled clear note selection.

### Equal Temperament

A computer screen shot of a keyboard

Description automatically generated

Figure ‑ The Equal Temperament page patch.

This page introduces the equal temperament tuning system and the unit of cents that was developed with it. In this tuning system, the octave is divided into 12 notes equally spaced apart, sacrificing some harmonic purity in order to solve the issues presented on the previous page. The user plays a pair of notes and the number of cents in the corresponding interval in both equal tempered and just intonation is calculated and displayed, along with the difference between the two cent values.

The pair of MIDI note numbers is collected using zl.stream. The frequency ratio of the Just interval is calculated using the ratio subpatch, and the corresponding cents value is calculated using an expr object.

The number of cents in the equal tempered interval is calculated by finding the difference between the two MIDI note values and multiplying by 100.

The difference between the two cent values is calculated using an expr object with the expression

Here Max’s abs operator is used to find the absolute value of the difference.

The sound of each note is produced by sending the pitch output of the kslider to a makenote object with the arguments 127 (velocity) and 500 (duration).

The note-on and note-off pitch and velocity values from the makenote are sent out of the patcher through a pair of outlets.

### Comparison

This page offers the same functionality as the ‘Problems with Just Intonation’ page for each of the three tuning systems discussed in the application. It also allows the user to record a melody and play it back. The user can select which tuning system playback uses or turn on the ‘optimiser’ which causes the program to automatically select the tuning system for which the sum of the numerator and denominator of the ratio for the given interval is smaller.

A screenshot of a computer

Description automatically generated

Figure ‑ The seq object is used to allow the user to record and play back a sequence of notes.

Melody recording and playback is done using the *seq* object.



When the user presses a key, the pitch and velocity values are sent from the kslider to a *pack* object to be collected as a list.



This list is sent to the leftmost inlet of a *midiformat* object which converts it to the correct format to be understood by seq.



seq will start recording incoming data when it receives a *record* message and stop recording when it receives a *stop* message. seq outputs the MIDI data for the recording when it receives a *start* message. These messages are sent to the seq object when the user clicks the textbuttons labelled “Record”, “Stop”, and “Play” respectively.

When the Play button is clicked, the note-on and note-off pitch values of the recorded sequence are sent to the left inlet of the kslider object so that interval data can be calculated and presented as if the user were playing the sequence live on the kslider. The MIDI data from the seq object is converted to note-on/off messages using a *midiparse* object.

A screenshot of a computer

Description automatically generated

Figure ‑ The frequencies of the notes in each tuning system are calculated using mtof objects given the intervals for that tuning system in Scala format as an argument.

The frequencies of each note in the current pair are calculated simultaneously using mtof objects. Note that the mtof object on the right isn’t given a scale argument. This is because this is used to generate the frequencies for the notes in equal temperament tuning, which is the default scale that mtof objects use.

The frequencies are sent around the patch using send objects which are named using a letter; P for Pythagorean, J for Just, and E for Equal tempered, and a number; 1 for the first note in the pair and 2 for the second.

A screenshot of a computer

Description automatically generated

Figure ‑ Clicking the Clear note selection button will set the frequency of each note to 0 Hz.

A textbutton labelled “Clear note selection” can be clicked by the user to set the frequency of all cycle~ objects to 0 so that they stop producing audio. Clicking this button also sends a set message to all comment objects displaying information about the current pair of notes.

A screenshot of a computer

Description automatically generated

Figure ‑ The user selects which tuning system the notes use by clicking a ubutton overlayed on the Lissajous figure for the interval in that tuning system.

The audio of only one tuning system at a time will play from this page. This is achieved by storing a 0, 1, 2, or 3 in a number object, 0 corresponding to no selection, 1 to Pythagorean tuning, 2 to Just intonation, and 3 to equal tempered tuning. The user selects which tuning is used by clicking a ubutton which is positioned in front of the Lissajous figure and interval information for that tuning system.

Any change to the pair stored in the zl.stream object causes the button on the left to output a bang, which sends the number 0, 1, 2, or 3 around the patcher using a send object named mode.

A screenshot of a computer

Description automatically generated

Figure ‑ Only the audio of the currently selected tuning system passes through the selector~ object to the patches outlet.

The frequency of each of the pair of notes for all three tuning systems are set as the frequency of cycle~ objects and routed through a pair of selector~ objects so that the signals from only one tuning system are send out of the patcher at a time. The open inlet of the selector~ objects are dictated by the mode value explained above.

A loadmess object sets the selector~s to be initially have all inlets closed, and clicking the Clear note selection button will also do this.

A computer screen shot of a computer

Description automatically generated

Figure ‑ The optimiser selects the tuning system for which the current interval's ratio is comprised of smaller integers.

The optimiser can be stitched off and on using a textbutton in toggle mode. The sum of the numerator and denominator output by the ratio subpatches for both Pythagorean and Just tuning are compared using an if object. If the sum is smaller for the Pythagorean interval, a 1 is output and if it’s smaller for the Just interval a 2 is output. The output of this is used as the mode value but is only sent to all receive mode objects if the gate is open, controlled by the textbutton.

A screenshot of a computer

Description automatically generated

*Figure 3‑39 Only the selected tuning system's Lissajous figure is given the colour of the pair of the notes.*

The RGB code for the current pair of notes is sent to the left inlet of a message box, so it sets the message without outputting. A receive object receives the mode and sends this to a button, so that whenever the user changes the mode a bang is sent to the left inlet of the message box, causing it to output the RGB code. The code is sent to the right inlet of a gate object with three outlets. Which outlet is open at any time is dictated by the mode. The code is sent to a message object above the Lissajous figure of the chosen tuning system and set as the arguments to an fgcolor message.

To correctly update the phosphor colour of the currently selected scope~, the RGB code is routed directly through another gate object controlled by the mode and sent to the message box of the selected tuning system.

A sel object receives the mode value as input, and bangs out the leftmost inlet if it receives a 0, the second leftmost inlet if the input is a 1, and so on for values 0,1,2,3. The leftmost inlet sends a bang to the three message objects that send “fgcolor 0.35 0.35 0.35”, This sets the phosphor colours of all three scope~ objects to grey. If the sel receives a 1, bangs are sent out the second leftmost outlet, which are sent to the fgcolor message with the RGB code of the calculated colour, and the grey fgcolor message for the other two scope~ objects. The next two outlets of the sel object are connected similarly so that selecting a tuning system sets the phosphor colours of the three scope~s are set appropriately.

# Results and Evaluation

## Testing

The relatively low computational complexity of the application and the small range of inputs it accepts meant that manual testing was an appropriate strategy for the quality assurance of the application. This was done by comparing the application’s performance to the system functional and non-functional requirements outlined in section 2.2 of this report, manually giving the full range of possible inputs to each page of the application and checking for erroneous outputs, and by asking friends and family to use the system and give feedback about its useability. This process proved the system to be very robust, without ever crashing, but it did highlight several incorrect outputs and useability issues. This section of the report outlines the issues identified.

Firstly, the interval between the currently entered pair of notes is updated and displayed on each page of the application that takes notes as input. For example, if the user enters notes at an interval of a perfect fifth on the Intervals page of the application and then clicks the next page button, “Perfect 5th” will be displayed below the keyboard on the Consonant vs Dissonant Intervals page. This is a failure to meet functional requirement 8. All output boxes should be blank when a page is first opened and only display an output when the user gives an input to that page, so that it is clear where feedback is being given to the user in response to an input.

When the user presses keys on a MIDI keyboard that are outside of the range of keys displayed on the on-screen keyboard on any page which features one, no keys are highlighted on the on-screen keyboard. This is not technically an error, but user feedback highlighted that it made it unclear whether the program was registering their inputs. This could be solved by taking the MIDI key number of the key the user presses and reducing it modulo the number of keys on the on-screen keyboard, then adding the lowest key offset of the on-screen keyboard. This would mean that playing the next key higher than the highest included in the on-screen keyboard would cause the lowest key on the on-screen keyboard to be highlighted.

In several parts of the application, when the user selects two keys that are an octave apart, the system outputs that they are at an interval of the unison instead of the octave either by displaying “Unison” on the Intervals and Dissonant vs Consonant Intervals pages, or by reporting the frequency ratio of the two notes as 1:1 instead of 2:1 on the Problems with Just Intonation and Comparison pages. As well as these erroneous results, the Equal Temperament page reports the incorrect number of cents in the given interval in Just tuning when the interval is larger than 10 semitones. This happens because of the modular arithmetic used in the ratio.maxpat subpatch used to calculate ratios not producing the expected result. In particular, it interprets the interval between C and the B♭ above as a minor second instead of a major seventh, as if it were the B♭ below the C. Because of this, an automatic systematic testing suite was created to evaluate the performance of the ratio subpatch.

A screenshot of a computer

Description automatically generated

Figure ‑ Automated test of ratio subpatch.

The test uses a counter object to test every possible pair of notes that could be given by a keyboard with keys in the range 36-84, the same key range as the largest on-screen keyboard included in the application. The test counts how many interval ratios fall outside of the range 1/1-2/1 for both Pythagorean and Just tuning. It should be noted that this doesn’t count every incorrect result, just the ones that fall outside of the acceptable range. To test that every result is correct, a Python script could be created to find the correct ratio between every possible pair of notes and check that ratio.maxpat gives this result.

The number of pairs tested was and the test found that 2185 of the ratios calculated fell outside of the acceptable range, a 47.4% error margin, indicating that the algorithm used needs modification.

There are several audio issues throughout the application. Firstly, audio generated by cycle~ objects produce a click sound when turned on or off, or when switching between tuning systems on the Comparison page. This is a minor issue but brings down the overall quality of the application and should be fixed in any future implementations. Another audio issue is that the piano sounds from the Intervals and Consonant vs Dissonant intervals page play even when the application’s audio is tuned off by the ezdac~ object.

Below are briefly listed the other issues of each page of the application:

Intervals

* Intervals page doesn’t name intervals from MIDI keyboard larger than an octave.
  + Should be prepared for any input, if the first interval the user enters is larger than an octave, they may think the application isn’t working properly.
  + Could use compound interval by doing mod 12 name or number above 8.

Dissonant vs Consonant Intervals

• Giving inputs quickly can cause system to lose sync i.e., not playing two pair of notes at the same time as giving interval name (On intervals page too).

• Giving inputs quickly causes more than two notes to be highlighted on keyboard (On intervals page too).

• Examples of consonant and dissonant intervals play too loudly causing a buzzing noise at highest point of ADSR envelope.

• Chopin excerpt plays very quietly.

Pythagoras’ Hammers

* The sound isn’t synced with the button animation.
* When the ‘Together’ button is clicked the sound of the larger hammer plays slightly before that of the smaller hammer.

Polyrhythms

* Rhythms aren’t quite synced up, as mentioned in the code explanation for this page.

Pythagorean Scale

* Too much information on screen at once can be overwhelming for user. This could be solved by either implementing the circle of fifths explanation on a separate page, or by adding controls to change the speed that the animation plays at.

The Overtone Series

* Pitch shifting the string sound produces strange unwanted sound.
* Clicking the right side of the string and then clicking the root button produces a very high-pitched unpleasant sound.
* Transition between ubuttons isn’t aligned with point where feather touches string.

Just intonation

* Lissajous figures are slow to update.

Comparison

* Sequencer doesn’t play final note for full length.

## Project Management

Because the application is based on theory which I had little previous understanding of and is written in a language I wasn’t previously familiar with, balancing time between learning the theory and the language and designing the application was difficult. A Gantt chart was presented as part of the project’s proposal, and this was approximately followed. With the benefit of hindsight, I believe the project would have been more successful if more time had been dedicated to researching the theory and related existing works so that a more formal plan for the system could have been designed, including a clear specification of what each page should do and UI mock-ups.

On the other hand, having a flexible plan was beneficial during the implementation stage of the project because an agile workflow was necessary to modify the specification of the application as I became more familiar with the capabilities of Max. Were I aiming to create a more refined product rather than use the project as an educational experience, it would have been preferable to create the program in a language that would be more suitable for creating user-friendly user interfaces.

## Discussion

To properly analyse the results of the project it would be desirable to conduct a survey study by providing a sample of the intended audience of the project with the application and asking them to provide feedback. The time needed to conduct such a survey was not available to me, I did however ask some friends and family to test the software and provide feedback, and using this and the original project aims and specification several strengths and deficiencies of the program were identified.

Inputs are given to the application using keyboards, sliders and button clicks, meaning there is little opportunity for a user to give unexpected inputs. This along with the relatively low computationally complexity of the program has allowed the application to be quite robust, giving few unexpected or incorrect outputs and not crashing.

One of the main motivations behind the project was to provide people interested in music production a cheaper alternative to in-person music lessons from a professional tutor. The project was successful in this aspect in that the application doesn’t require an internet connection or a powerful computer to run, making it accessible to people from lower income backgrounds. Improvements that could make the application more accessible include adding a colour-blind mode which would assign colours that are easier to differentiate between to users afflicted with a colour visual deficiency to the 12 notes of the chromatic scale, as discussed by Malandrino et al. (Malandrino, Pirozzi and Zaccagnino, 2018).

The theory taught in the application was chosen to give insight into why the choice of notes in a piece of music is important and how to make these choices. Choosing the specific concepts and how to explain them required a degree of nuance to make the application useful and enjoyable to an audience of many ages and levels of prior knowledge. The finished application does well at meeting this requirement by taking a mathematical and historical approach to the theory rather than purely musical, and by using analogies like that of the blacksmiths’ hammers Pythagoras observed to make ideas like frequency ratios seem less abstract. However, the application omits many fundamental aspects of musical harmony and on its own would not provide the knowledge needed to compose a piece of music. Some aspects of the theory are left underdeveloped and feedback given by those I asked to test the application indicated that some areas would benefit from further explanation, for instance the ‘Intervals’ page mentions the quality of an interval but this is not explained, and several references are made to the root note used as a reference when calculating a note’s frequency from the ratio of an interval but this is not explained thoroughly enough.

Reflecting on the finished application highlights areas of the theory that could have been explained more clearly using different explanations. For example, using stave notation rather than a piano’s keyboard to represent Just and Pythagorean scales may have been more appropriate since these tuning systems can split an octave into an arbitrary number of notes. For the same reason, in just intonation A♯ and G♭ are two distinct notes, but this application treats them as if they are the same. This was a conscious decision to keep the scope of the project manageable in the available time frame. Were more time available I would have liked to expand the educational scope of the application to cover the Tonnetz grid, non-western music, harmony in chords comprised of more than two notes. Finally, the explanation of why transposition fails when using just intonation could have been improved by implementing a function where the user records a sequence of notes, and the sequence would be transposed by moving each note up by a set number of semitones. The resulting loss of consonance could then be visualised by showing the Lissajous figure of each successive interval in the sequence, as is done on the ‘Comparison’ page of the application.

The application’s UI could also be improved to make the application more engaging and feel more trustworthy. Most importantly, it would be desirable for the size of the UI to scale with the screen of the user’s monitor. The buttons of the UI could also be customised using JS Painter files.

## Conclusion

This project has found that building a comprehensive interactive application to teach musical harmony is a feasible improvement on existing resources that aim to teach the same content, and basic trials show that it is a more effective learning method although further studies would be required to verify this. Furthermore, creating the application using Max was a valuable experience that strengthened my software development skills, but to create a more user friendly application more conventional programming languages may be more suitable.

# References

Chase, S. (2022). *What Is Timbre In Music? | HelloMusicTheory*. [online] https://hellomusictheory.com/. Available at: https://hellomusictheory.com/learn/timbre/.

Rich, A. (2019). *Harmony - Classical Western harmony*. [online] Encyclopedia Britannica. Available at: https://www.britannica.com/art/harmony-music/Classical-Western-harmony.

Doshi, K. (2021). *Audio Deep Learning Made Simple (Part 2): Why Mel Spectrograms perform better*. [online] Medium. Available at: https://towardsdatascience.com/audio-deep-learning-made-simple-part-2-why-mel-spectrograms-perform-better-aad889a93505 [Accessed 21 Sep. 2023].

Ashton, A. (2018). *Harmonograph*. Glastonbury, Somerset: Wooden Books Ltd.

Ashley, J. (2016). *Using Harmonic Polyrhythms – Jack Ashley*. [online] jackashley.com. Available at: https://jackashley.co.uk/2016/12/12/using-harmonic-polyrhythms/ [Accessed 30 Jun. 2023].

Neely, A. (2015). *Harmonic Polyrhythms Explained! [ AN’s Bass Lessons #27 ]*. [online] www.youtube.com. Available at: <https://www.youtube.com/watch?v=_gCJHNBEdoc&t=109s>.

Malandrino, D., Pirozzi, D. and Zaccagnino, R. (2018). Visualization and Music Harmony: Design, Implementation, and Evaluation. In: *2018 22nd International Conference Information Visualisation (IV)*. [online] 2018 22nd International Conference Information Visualisation (IV). pp.498–503. Available at: https://ieeexplore-ieee-org.bham-ezproxy.idm.oclc.org/document/8564210 [Accessed 9 Jul. 2023].

Ciuha, P., Klemenc, B. and Solina, F. (2010). *Visualization of Concurrent Tones in Music with Colours*. [online] *ACM Digital Library*, pp.1677–1680. Available at: https://dl.acm.org/doi/abs/10.1145/1873951.1874320?casa\_token=nnx9sB3m2HYAAAAA:WX\_mPYpgJ\_9L7OHCOVIVznxEf1qvdL4HYzE5r6ryqY4\_4RnmdpkGj4bkUD1H6M4Ky5b0GsQyawxDpQ [Accessed 9 Jul. 2023].

# Appendices

## Instructions to Run

A standalone application and the .maxpat versions of the application can be found at <https://git.cs.bham.ac.uk/projects-2022-23/oxg053>. The standalone application can be run on any windows machine, but the .maxpat file requires a licensed copy of Max to be installed on the machine. Max can be purchased from https://cycling74.com/shop/max.

To run the standalone application, download the zip file ‘Musical Harmony Explained.zip’, unzip it, and run the file ‘Musical Harmony Explained.exe’ contained within.

To run the application in Max, download the folder ‘Owen Graham Diss 2023’, and move the folder ‘Sadam Library’ to your Max packages location (defaults to ~/Documents/Max 8/Packages). The top-level patcher you need to open to run the program is ‘Full System.maxpat‘, but all subpatches and media are included in the Owen Graham Diss 2023 folder for inspection.

## Object Descriptions

|  |  |
| --- | --- |
| **Object** | **Description from Max documentation** |
| *\*~* | \*~ is a signal multiplier-operator that outputs a signal which is the multiplication between two signals. |
| *bpatcher* | Abstracts the contents of a patcher or subpatcher for use in other patchers, displaying only those visual elements which are desired. The number of inlets and outlets in a bpatcher is determined by the number of inlet and outlet objects contained in its subpatch window. |
| *button* | Button blinks when you send it any message, and it sends out a bang when you click on it. |
| *coll* | Allows for the storage, organising, editing, and retrieval of different messages. |
| *counter* | Outputs the current count of bang message constrained to a specified range. |
| *cycle~* | Use the cycle~ object to generate a periodic waveform. The default waveform is one cycle of a cosine wave. |
| *delay* | Holds a bang for a specified amount of time before sending it to the next object. |
| *expr* | Evaluate an expression using a C-like language. Variables and operators are used to create output values. |
| *ezdac~* | ezdac~ works as a user interface version of the dac~ object. It appears as a button which can be clicked with the mouse to turn audio on or off. |
| *fpic* | Display an image. |
| *function* | Draw or store a set of x, y points as floating-point numbers. The output the entire function is useful as an input for line~. |
| *gate* | Pass input to an outlet |
| *if* | Evaluates input according to a conditional statement specified in an if-then-else form. |
| *incdec* | Increment or decrement a value. When connected to a number box, click the upper part of the object to increment, click the lower half to decrement. |
| *kslider* | Outputs and displays note and velocity information using an on-screen keyboard. |
| *line~* | Use the line~ object to generate a signal ramp or envelope. |
| *live.gain~* | live.gain~ is a slider that scales input audio signals and provides a visual indication of the current sound level on a decibel scale. |
| *live.scope~* | Use live.scope~ to visualise an audio signal. |
| *loadbang* | Outputs a bang automatically when the file is opened or when the patch is part of another file that is opened. |
| *loadmess* | Outputs a message automatically when the file is opened, or when the patch is part of another file that is opened. |
| *loadmess* | Outputs a message automatically when the file is opened, or when the patch is part of another file that is opened. |
| *makenote* | Outputs a MIDI note-on message paired with a velocity value followed by a note-off message after a specified amount of time. This allows for generative MIDI output without having to manage note-off generation. |
| *message* | Message displays and sends any given message with the ability to handle specified arguments. |
| *metro* | Acts as a metronome which outputs bangs at a regular, specified interval. |
| *midiformat* | Numbers received in the inlets are used as data for MIDI messages. The data is formatted into complete MIDI message (with the status byte determined by the inlet) and sent out the outlet as individual bytes. |
| *midiparse* | Separates raw MIDI bytes into standard message types. |
| *mtof* | Performs a MIDI-note-number to frequency conversion. Frequency is reported as a float in Hertz (Hz). |
| *notein* | Receives its input from a MIDI note-on or note-off message sent by a MIDI input device. |
| *noteout* | Transmits note-on and note-off messages to a MIDI device. |
| *omx.peaklim~* | omx.peaklim~ is a peak-limiter which allows for the specified control of signal amplitude. |
| *pack* | Combine elements into an output list. |
| *pak* | The pak object (pronounced “pock”) offers much of the functionality of pack, but outputs the entire list whenever input is received in any inlet. |
| *pictctrl* | Creating buttons, switches, knobs, and other controls using images from a picture file for its appearance. |
| *pictslider* | A slider control that uses pictures in external files for its appearance. It uses two pictures—one for the “knob” and one for the background over which the knob moves. |
| *pitchshift~* | Use the pitchshift~ object to load a perform pitch-shifting on an input signal. |
| *receive* | Receives and outputs messages from send objects sharing the same name. This allows you to send any kind of message between Patcher windows or within a window without using patch cords. A receive object can be instantiated with its short-form: the letter "r". |
| *route* | Tries to match a message’s first argument to the route object’s own arguments. |
| *scope~* | Use the scope~ object to visualise an oscilloscope-style display. |
| *sel* | Selectively output a bang in response to any input which matches its arguments and will output non-matching messages out its right-most outlet. |
| *selector~* | Use the selector~ object to choose between one of several input signals (or none). The selector~ is similar to the Max switch object but for signals, however if no input is chosen, it outputs a signal composed of zero values. |
| *send* | Send will transmit given messages to receive objects which are named by the same argument and will allow you to send any kind of message between Patcher windows or within a window without using patch chords. A send object can be instantiated simply by typing into an object box the short form letter “s”. |
| *seq* | seq is a sequencer of raw MIDI bytes. You can control the speed of playback (only at the time you start it), read and write from files, and record from live MIDI input. |
| *sfplay~* | Use the sfplay~ object to play audio files from disk. |
| *slider* | Resembles a sliding potentiometer, outputting numbers restricted to a specified range, offset by a specified number, and multiplied by a specified number. |
| *stripnote* | Only pass note-on messages: those having any velocity above 0. |
| *textbutton* | Button with text |
| *thispatcher* | Allows modification of a patcher window with Max messages. |
| *toggle* | When clicked, toggle outputs a 0 when turned off and a 1 when turned on. When giving input, a non-zero number will turn it on, a 0 will turn it off, and a bang will alternate the state of the toggle. |
| *ubutton* | Creates a transparent click-able region that can be placed over graphics or other objects. Produces a bang message when clicked. |
| *unpack* | Breaks a list into its elements, and sends each item out a separate outlet. |
| *zl.group* | Output a list after the number of items specified by the maximum output length are received. |
| *zl.nth* | Extract an item from an incoming list, based on the index specified (starting at 1). |
| *zl.sort* | zl.sort is used to sort the contents of a list. |
| *zl.stream* | zl.stream accepts a number in the right inlet which specifies the length of the output list. Following the reciept of this number, the object will collect this number of items through the left inlet. After the list-length is complete, and with each subsequent input, the list will be output the left outlet. |

## Additional Patch Images

A screenshot of a computer

Description automatically generated

Figure ‑ Consonant vs Dissonant Intervals patch

A screenshot of a computer

Description automatically generated

Figure ‑ Polyrhythms patch.

A screenshot of a computer

Description automatically generated

Figure ‑ Pythagorean Scale patch.

A screenshot of a computer

Description automatically generated

Figure ‑ The Overtone Series patch.

A screenshot of a computer

Description automatically generated

Figure ‑ Problems with Just Intonation patch.

A screenshot of a computer

Description automatically generated

Figure ‑ Comparison patch.