Henry Mattinson

Excello: End-user music programming in Excel

Computer Science Tripos – Part II

Christ's College

April 18, 2019

Chapter 1

Introduction

1.1 Motivation

There exist many programs for music notation and composition. Sibelius allows users to write scores using traditional western music notation, whilst music is produced in the live programming interface Sonic Pi by real-time editing of Ruby code [1]. These require users to gain familiarity with a new interface, often with a large threshold to creating simple musical ideas. Spreadsheet users significantly outnumber programmers [15] being the preferred programming language for many people [12]. I believe that this ubiquitousness, along with the affordances of the spreadsheet, would enable new ways to interact with musical notation that capitalise on existing familiarities with spreadsheets and their data handling capabilities.

The use of grid structures is an established concept in music programs, with most sequencing software using one axis of the screen for time and the other for pitch or musical parts. Chris Nash's Manhattan [11] uses a grid structure where formulae can be defined in the cells to change the cell value, much like in a spreadsheet. However it is limited to columns defining tracks and rows corresponding to different times. Advait Sarkar's SheetMusic [14] investigated how formulae with sound output can be included within the spreadsheet paradigm. This also introduced abstracting time away from the grid, in this case using an incrementing global `tick` variable which could be referred to in the formulae. Both axes can be used interchangeably for SheetMusic notation or markup that the user wishes to include which is not interpreted musically, a concept idiomatic to Excel usage. Simple formulae such as `if(tick%2==0) p('snare') else p('kick')` allow musical structures to be defined without advanced programming knowledge but quickly become unwieldy for defining larger pieces, especially if they are not highly repetitive. Whilst other spreadsheet music projects exist [4], these simply use the spreadsheet as the medium for conventional sequencing with an auxiliary script used to parse the grid and create musical output.

Excello is an Excel add-in for end-user music programming where users define music in the spreadsheet and can play it back from within Excel. It maintains the abstraction of time from the grid to keep the flexibility spreadsheets offer but was designed so that the complexity of an individual cell was limited. Existing functionality within Excel can be used, both accelerating the learning curve and increasing the available functionality.

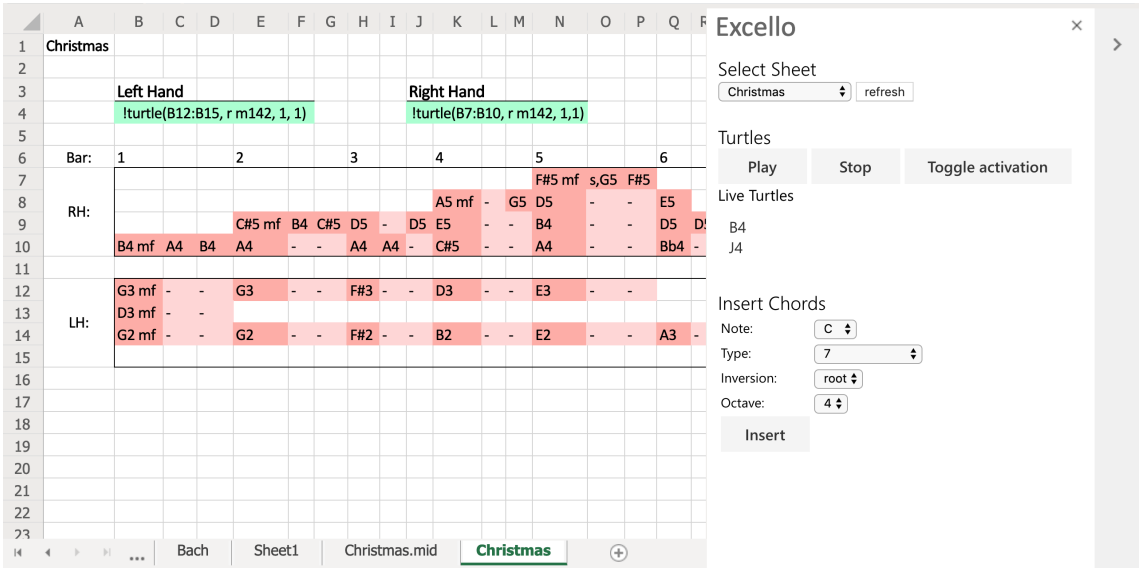

Figure 1.1: The Excello notation and add-in

1.2 Outline of work

1. I design a system for musical expression and playback within Excel. An initial prototype is built satisfying the all the success criteria of the system itself: play individual notes and chords with defined durations, define multiple parts, play loops, define sequences of notes and chords and be able to call these for playback, and define the tempo of playback.

2. Participatory design commences using the initial prototype as a discussion point. Following formative evaluation sessions with 21 participants, issues and feature requests are identified. Users continue to use the system and give feedback following their introductory session.

3. A series of additional extensions are implemented to solve the problems identified by participants and add features that are requested to improve the system.

4. A converter from MIDI to the Excello format is built to aid in the translation of a large existing corpus of MIDI files to the Excello notation.

5. Summative evaluation is carried out with the participants. The success of the features implemented as part of the participatory design phase is evaluated. The usability of Excello is analysed using the Cognitive Dimensions of Notation framework [3], focusing on Sibelius as a comparison.

Chapter 2

Preparation

This chapter shall first address the refinements made to the project proposal. Then I shall explain the tests that were performed to establish Excel's suitability for musical development. Next, the design decisions for Excello itself shall be explained. The software engineering tools and techniques employed will then be introduced. Finally, the research that was conducted to decide to implement a converter from MIDI to Excello shall be summariesed.

2.1 Proposal Refinements

The project shall invent a notation by which music can be defined within a spreadsheet along with a system for interpreting the notation in the spreadsheet to produce audio output. This shall continue to explore ways in which time can be abstracted away from the grid.

The aim shall be to implement the project as an Excel add-in subject to successful initial testing. An add-in is a web application displayed within Excel that is able to read and write to the spreadsheet using the Office Javascript API. This shall be implemented in a way that allows arbitrary additional data and markup to be included in the spreadsheet. Also, no information beyond the spreadsheet shall need providing for playback via the add-in to be possible. Tests shall be carried out to verify that suitable audio output can be produced for music end-user music programming within Excel to be possible.

A sizeable addition to the project not included in the initial proposal was to perform participatory design [10] to advise on improvements that can be made beyond the initial prototype. The prototype would be introduced to users and from this, new features and improvements implemented. A subset of these participants who gain sufficient familiarity with the project can then be used for more informed summative evaluation. As a result, the proposed extension of incorporating live-coding will only be implemented if there are no other issues or feature requests raised by the userbase that are deemed higher priority.

MIDI shall be the formal notation for which a converter shall be implemented to translate into a CSV file that can then be opened in Excel. Additional explaination on the choice of MIDI is provided below. The choice of MIDI was motivated by participants

who wished to be able to integrate Excello in to their use of digital audio workstations such as Logic Pro, Ableton Live and GarageBand.

2.2 Initial Tests

The following section outlines the libraries I explored and the tests carried out to assess the feasibility of synthesising notes given data in a spreadsheet using an Excel add-in. All tests were carried out in Excel Online using Script Lab, an add-in that allows users to create and test simple add-ins experimenting with the Office Javascript API. These add-ins have an HTML front end and can access libraries and data elsewhere online.

A simple add-in that played a wav file stored online was used to verify that an add-in was capable of creating sound.

2.2.1 Note Synthesis Library

The Web Audio API allows audio to be synthesised from the browser using Javascript [9]. To create a program for users to define and play musical structures will require synthesising arbitrary length, pitch and volume notes. In order to avoid the lower-level audio components (e.g. oscillators), I researched libraries that would allow me to deal with higher level musical abstractions of the synthesised notes. Sarkar's SheetMusic used the library tones[1] which provides a very simple API where only the pitch and volume envelope[2] of all notes. Other limitations included no definition of volume and only including simple waveform synthesisers.

Tone.js[3] is a library built on top of the Web Audio API providing greater functionality than tones. An `Instrument` such as a `Synth` or `Sampler` is defined. The `triggerattackrelease` release method of these instruments allows a note of a given pitch, volume and duration to be triggered at a particular time. Notes are defined using scientific pitch notation (e.g. `F#4`), the notes name (`F#`) combined with the its octave (`4`). Script Lab is able to reference libraries from the Node Package Manager (NPM). Therefore, I was able to test creating notes with pitches defined in the add-in Javascript to confirm Tone.js was suitable for an add-in.

2.2.2 Office Javascript API

In order to create a program for users to produce music from within Excel, the musical output must be informed by the data in the spreadsheet. Tests up to this point had created notes defined within the add-in Javascript. To test the Excel Javascript API, I outputted a note with the Tone.js library, the pitch of which was defined in the spreadsheet. This was extended so the instruction to play a note, not just the pitch, being defined within a cell, detected and executed.

[1] https://github.com/bit101/tones
[2] A description of how the note volume changes over its duration
[3] https://tonejs.github.io/

Next, I was able to play a sequence of constant length notes with the notes defined in consecutive cells. The range of cells was accessed using the Excel API and the values were played using the Tone `Sequence` object. Having carried out the above tests, I confirmed Tone.js combined with the Excel API had the functionality required to assist in the implementation of the project.

2.3 Excello Design and Language

2.3.1 Abstracting Time

Dave Griffith's Al-Jazari [7] takes place in a three-dimensional world where robotic agents navigate around a two-dimensional grid. The hight and colour of the blocks over which the agents traverse determines the sound that they produce. The characteristics of the blocks are modified manually by users at run-time whilst the agents are moving. Whilst there are more complex conditional instructions, the basic instructions have the agents rotate and move forwards and backwards in the direction that they are facing. There therefore exists a dual formalism as both the instructions given to an agent and the state of each block. This design is intended to make live coding more accessible, both when viewing performances and becoming a live coder.

In Al-Jazari, the agents are programmed by placing symbols corresponding to different movements in thought bubbles that appear above them. This is not suitable for programming within spreadsheets where all data must exist alphanumerically within cells. What's more if an agent was to continue moving forwards many times in a row, it would become tiresome to keep adding the move forward symbol. This is less of an issue in Al-Jazari where the grid within which the agents navigate only measures ten cells wide and long.

The concept of having a curser navigate around a cartesian plane is the method used by turtle graphics. Just as this concept is used in Al-Jazari to play the cell the agents occupy rather than colour it, it is suitable for spreadsheets. The turtle abstraction is employed by Excello by having notes defined in cells and defining agents, known as turtles, to move through the spreadsheet activating them. In order to play a chord, multiple turtles must be defined to pass through multiple cells corresponding to the note of the chord. This method maintains high notational consistency but sacrifices the abstractions for musical structures that are available in languages like Sonic Pi - `chord('F#', 'maj7')`. By implementing methods in the add-in to add the notes of chords to the grid, the use of the abstractions is maintained whilst preserving consistency and cleanness in the spreadsheet itself.

The turtle is the crux of the Logo programming language [2]. In Logo, turtles are programmed entirely by text. For example `repeat 4 [forward 50 right 90]` has a turtle move forwards 50 units and turn 90 degrees to the right. This is repeated four times to draw a square. A similar method is employed in Excello but the language is designed to be much more concise.

2.3.2 Initial Prototype Design

In Excello, notes are placed in the cells of the spreadsheet and pathways through the grid are defined using a language for programming turtle movement. The notes in the cells will be played when a turtle moves through that cell. When the program is run, the melodic lines produced by all turtles defined in the grid will be played concurrently. Turtles are defined with a start cell, movement instructions, the speed with which they move through the grid (cells per minute) and the number of times they repeat their path. As in Al-Jazari, distance in space maps to time [8], Excello extends upon this by allowing different turtles to navigate at different speeds. This allows parts with longer notes to be defined more concisely and for phase music[4] to be easily defined.

As in Logo, turtles begin facing north. The move command `m` moves the turtle forward one cell in the direction that it is facing. Just like in Logo, the turtle always moves in the direction it is facing. The commands `l` and `r` turn the turtle 90 degrees to the left and right respectively. Repeats are implemented in Logo with the command `repeat` followed by the number of repeats and the instructions to be repeated [2]. In order to create more concise instructions, single ommands can be repeated in succession by placing a number immediately after it. For example, the command `m4` will have the turtle move forwards four cells in the direction that it is facing. The direction a turtle is facing can be defined absolutely using the commands `n`, `e`, `s` and `w` to face the turtle north, east, south and west. This could have instead moved the turtle in that direction, but this would have lost the consistency that the turtle always moves in the direction it is facing. In order to change the volume notes are played at, dynamics (`ppp`, `pp`, `p`, `mp`, `mf`, `f`, `ff`, `fff`) can be placed within the turtle instructions. Any notes played after this will be played at that dynamic. In the same way the dynamics in western notation are a property of the staff and not individual notes, dynamics were originally designed to be a property of the turtle. In order to repeat multiple instruction sequences, these are placed in brackets and the number of repeats put immediately after the bracket. For example, `(r m50)4` would define a path going clockwise around a fifty by fifty square. This 8 character example is equivlant to the Logo example above that requires 30 characters. The ability to repeat larger series of instruction is why the relative movements `l` and `r` are included in the language despite being less explicit than the compass based directions.

It may not be convenient for each melodic line to be defined by a single path of adjacent cells. Just as conventional score notation often spans across multiple lines, the splitting of parts is a useful form of secondary notation. This requires the turtle to navigate to non-adjacent cells and then proceed playing. For graphic drawing in Logo, the pen can be lifted, allowing the turtle to navigate without colouring the space beneath it. This is suitable for a graphical output where the number of steps the turtle takes has no effect on the output, only the cells it colours. However, the musical output is dependent on when the turtle is in certain cells, so this would not be convenient as it would introduce large rests. Analogous to lifting the pen for graphical turtles, one could set the turtle in a mode where it doesn't play the cells it navigates through and passes through them immediately until it is placed back in a playing mode. However, in this case the actual path that the

[4]Music where identical parts are played at different speeds

Table 2.1: Grammar rules for turtle movement instructions. $z \in \mathbb{Z}, n \in \mathbb{N}, c \in$ `[A-Za-z]`$^+$.

Grammar Rule	**Left Symbol Meaning**
$\mathbf{S} \rightarrow \mathbf{Y}$	Starting symbol
$\mathbf{Y} \rightarrow \mathbf{X} \mid \mathbf{X}\ \mathbf{Y}$	A series of instructions
$\mathbf{X} \rightarrow (\mathbf{Y})z \mid \mathbf{I}$	A single command or bracketed series of instructions
$\mathbf{I} \rightarrow$ `m`$z \mid \mathbf{R} \mid \mathbf{R}z \mid \mathbf{A} \mid \mathbf{D} \mid$ `j`$\mathbf{C} \mid$ `j`$\mathbf{P}n\mathbf{P}n$	A single command
$\mathbf{R} \rightarrow$ `l`$\mid$`r`	Relative rotation
$\mathbf{A} \rightarrow$ `n`$\mid$`e`$\mid$`s`$\mid$`w`	Absolute rotation
$\mathbf{P} \rightarrow$ `+`$\mid$`-`	Sign
$\mathbf{C} \rightarrow cz$	Cell reference
$\mathbf{D} \rightarrow$ `ppp`$\mid$`pp`$\mid$`p`$\mid$`mp`$\mid$`mf`$\mid$`f`$\mid$`ff`$\mid$`fff`	Dynamic

turtle takes is insignificant only the cell it ends up in. I have therefore added jumps to the language. This can be defined in absolute terms where the destination cell is given (e.g. `jA5`), or relatively (e.g. `j-7+1`), where the number of rows and columns jumped is given instead. An absolute jump may be more explicit to the human reader but defining jumps relatively allows them to be repeated, jumping to different cells in each repeat. For example `r (m7 j-7+1)9 m7` plays 10 rows of 8 cells from top to bottom playing each row left to right.

The language for turtle movement instructions can be summarised by the follwoing context-free grammar, $(N, \Sigma, S, \mathcal{P})$. Where the non-terminal symbols $N = (\mathbf{S}, \mathbf{Y}, \mathbf{X}, \mathbf{I}, \mathbf{R}, \mathbf{A}, \mathbf{P}, \mathbf{C}, \mathbf{D})$, terminal symbols $\Sigma =$ ($z \in \mathbb{Z}, n \in \mathbb{N}, c \in$ `[A-Za-z]`$^+$, `m`, `j`, `l`, `r`, `n`, `e`, `s`, `w`, `+`, `-`, `ppp`, `pp`, `p`, `mp`, `mf`, `f`, `ff`, `fff`) and starting symbol S. The set of grammar rules are shown in table 2.1:

Notes are defined in the cells using scientific pitch notation - the note name (with accidental[5] if required) followed by the octave number. Empty cells are interpreted as rests. In order to create notes longer than a single cell, the character `s` in will sustain the note that came before it. This is used to create notes longer than the duration of a single cell. A cell can be sub-divided time-wise into multiple notes by placing multiple notes separated by commas into a cell. The motivation for this design decision was so the length each cell corresponds to is not bound by the length of the smallest note in the piece. For example, a piece defined primarily with crotchets (one unit) but with a single instance of a quaver (half a unit) and dotted crotchet (one and a half units) can define these two notes with `C4,C4` and `s` in two cells. Without this, representing this single quaver would require double the number of cells and introducing many additional `s` cells in the entire piece.

[5]Sharp or flat symbol used to define a black note on a piano keyboard.

2.4 Software Engineering

2.4.1 Requirements

The success criteria of the project are as follows:

1. Implementation of an API for music playback within a spreadsheet allowing users to:
    - Play individual notes and chords and define their durations.
    - Define multiple parts.
    - Play loops.
    - Define sequences of notes and chords and be able to call these for playback.
    - Define the tempo of playback.
2. Implementation of a converter from MIDI to the spreadsheet representation.
3. Performance of participatory design sessions.
4. Usability testing using participants who have gained familiarity with the system.

   In addition to these, the following extension work was completed:
5. Extensions:
    - Implement additional features from issues and requests that arise from participatory design.
    - Explore a compressive conversion from MIDI to the Excel system.

Tools and Technologies Used

Initial tests were written in Javascript in the Script Lab add-in for Excel. Excello was written in Typescript as this is readily compiled into the Javascript required to run the add-in but provides static type-checking. It also allows the large collection of existing Javascript libraries to be utilised. Using the Yeoman generator I created a blank Excel add-in project. I used NodeJS to manage dependencies to other Javascript libraries. During development I ran the add-in on a localhost. To allow participants to run Excello on their own machines, I hosted a version of the add-in online using Surge. To run the add-in in Excel, a manifest.xml file is imported which instructs Excel where the add-in is hosted. The converter from MIDI to Excello was implemented in Python using Jupyter Notebooks.

The tone.js library was used to synthesis and schedule sound production via the Web Audio API. The Javascript music theory library tonal was used to produce the notes that make up chords. This prevented the hardcoding of the intervals present in the 109 chords available. The Python library Mido was employed to read python files. All of these

libraries have an MIT license. I used the Salamander Grand Piano V3[6] sample pack in order to sythesise realistic piano playback. This is under the creative commons liscence[7]

Starting Point

Having used the Yeoman generator to create an empty Excel add-in, all of the code used to produce Excello and the MIDI converter is produced from scratch using the tools and technologies described above.

I had written simple Javascript for small web pages, but no experience using Node, libraries or building a larger project. I had never used any of the libraries before, therefore, reviewing the documentation was required before and during development. I had gained significant experience with Python and Jupyter Notebooks from a summer internship.

Evaluation Practices

In order to best tune the design of Excello to the needs of potential users, formative evaluation sessions were carried out with participants. As a result, the project followed a spiral development methodology. Due to the number of participants involved and the timeframe of the project, there were only two major development iterations. The first prototype following the design described above, and the second fixing issues and implementing requests brought up by the participants.

The users from the participatory design phase of the project were involved in summative evaluation at the end of the project. By using the same users, I could carry out tests using experienced users of Excello despite the product not yet being released in the public domain.

2.4.2 MIDI files

Musicical Instrument Digital Interface (MIDI) details a communications protocol to connect electronic musical instruments with devices for playing, editing and recording music. A MIDI file consists of event messages describing on/off triggerings for a device or program to control audio [6]. MIDI files were designed to be produced by MIDI controllers such as an electric keyboard. As such, a MIDI file contains a lot of controller specific information that is not necessary for the creation of an Excello file. There exist musical formats such as MusicXML that specify the musical notation and as such may be more suitable for conversion to Excello.

Many musical programs support the importing and exporting of MIDI files. By allowing MIDI files to be converted to the Excello notation, Excello is more integrated into the environment of computer programs for playing, editing and composing music. Furthermore, there exist many datasets available for MIDI [5] which can immediately be played back for comparison.

[6] https://freepats.zenvoid.org/Piano/acoustic-grand-piano.html
[7] http://creativecommons.org/licenses/by/3.0/

Chapter 3

Implementation

This chapter shall first explain how turtles are defined along with the remaining features of the initial prototype. The format and results of the formative evaluation using this initial prototype shall be summarised. I shall then cover the design decisions and changes that were made to the Excello prototype during this participatory design process. Then, the technical details of Excello and the MIDI to Excello converter will be covered. Concluding with an overview of the project repository.

3.1 Initial Prototype

Notes and turtles can be defined in any cell. The interpretation of cell contents by turtles is shown in table 3.1. When the Excello add-in is opened, a window will open in the right side of Excel. A play and stop button can be used to launch all the turtles defined in the spreadsheet and initiate playback. Playback is a realistic piano sound.

3.1.1 Turtles

Turtles are defined as follows:

```
!turtle(<Starting Cell>, <Movement>, <Speed>, <Number of Loops>)
```

Table 3.1: Definition of notes in cells.

Interpretation	**Format**
Note	Note name (A-G), optional accidentals and octave number e.g. `F#4`
Sustain	`s`
Multiple notes subdivided in time	Notes, rests or sustains separated by a comma. Rests must be a space or an empty string e.g. `E4,,C4,s`
Rest	Any cell not interpreted as a note, sustain or multi-note.

Activation

"`!`" dictates that the turtle will be activated when the play button is pressed. Just as digital audio workstations allow muting and soloing of tracks, this can be used to quickly modify which turtles will play without losing their definitions.

Starting Cell

The starting cell of the turtle, which is also played, is given by the cell reference. As with Excel formulae, this is a concatenation of letters for the column and numbers for the row.

As each turtle only plays one note at a time, multiple turtles must be defined to play polyphonic music such as chords. It was believed that user would define turtles following identical paths but in adjacent rows or columns. Multiple turtles following identical paths but starting from adjacent cells can be defined using the existing Excel range notation for the starting cells. "`A2:A5`" would define four turtles in the cells A2,A3,A4,A5. This prevents the writing of multiple turtle definitions differing in only the start cell row.

Movement

Turtles start facing north. The language for programming turtle movement has been discussed in the preparation chapter. Using brackets to repeat movements within the turtle's instructions was not implemented by the start of the participatory design process.

Speed

An optional third argument declares the speed the turtle moves through the grid relative to 160 cells per minute. The default is 1 corresponding to 160 cells per minute. If the argument "`2`" was provided, this would move through the grid at 320 cells per minute. This relative system was used so it would be easier to tell the speed relation between two turtles. This would be particularly beneficial for phase music. Arbitrary maths can be provided for this argument also allowing turtles' speeds to be irrational multiples of each other.

Number of Loops

An optional fourth argument defines the number of repetitions of the turtle's path. By default, the turtle will loop infinitely. This was included so that repeating parts (e.g. the cello of Pachabel's Canon in D) need only defining once.

3.1.2 Highlighting

To assist in the recognition of notes and turtles, when the play button is pressed, cells are highlighted, conditional on their contents. Cells containing activated or deactivated turtle definitions are highlighted green. Cells containing definitions of notes, or multiple notes,

are highlighted red. Sustain cells are highlighted a lighter red, to show a correspondance to notes whilst maintaining differentiation.

3.1.3 Chord input

In order to use musical abstractions of chords and arpeggios [1] whilst keeping the paradigm of a turtle being responsible for up to one note at any time, a tool to add them is included. The note, type, inversion and starting octave of the chord are inputted in four drop-down selectors and the insert button enters the notes making up that chord into the grid. If a single cell or range taller than it is wide is highlighted in the spreadsheet, the notes will be inserted vertically starting at the top-left of the range. Otherwise, the notes will be inserted horizontally. This means whether the turtles are moving horizontally or vertically both chords and arpeggios can be easily defined. Thus, helpful musical abstractions are still available whilst keeping the cleanness of the turtle system.

3.2 Formative Evaluation

To guide development to best suite users, participants were involved in formative evaluation. 21 participants took place in the participatory design process. Participants were all musical University of Cambridge students, across a range of subjects. Initially, one-on-one tutorials of the initial prototype were given followed by the user carrying out of a short exercise. After both the tutorial and the exercise, users were interviewed on how they found Excello, drawing particular attention to actions that they found particularly unintuitive or requiring notable mental effort. Comparisons were made to the musical interfaces that participants were already familiar with. The sessions were audio recorded in order to prevent the jotting down of notes causing delays. Notes were later made from these recordings. The ethical and data handling procedures shall be discussed in the evaluation chapter.

For realistic simulation of the ways in which Excello would be used, participants were given the freedom to carry out an exercise of their choice. In many cases this was transcribing an exiting piece from memory or traditional western notation into the Excello notation. Two tasks were provided to choose from if participants had no immediate inspiration; transcribing a piece of western notation music or making changes to existing Excello notation.

These sessions were carried out at the beginning of Lent term 2019. Participants were asked if they would be willing to continue using Excello personally until the summative evaluation sessions, eight weeks later. This allowed additional feedback to be given as participants used Excello in their own time. It also ensured that the summative evaluation would be done using users with sufficient experience of the interface.

[1] Where the notes of a chord are played in rising or descending order

3.2.1 Issues and Suggestions

The issues and suggestions from the participatory design process have been summarised below.

Turtle Notation

Dynamics in the turtle instructions made it harder to extract the turtle's path as not all instructions related movement. As the dynamics weren't next to the notes they corresponded to, it was challenging to know the of volume a note or where to place the dynamics within the turtle to affect a certain note elsewhere in the spreadsheet. The initial prototype had no way to assign a dynamic to the first note without having the starting cell being empty. This empty cell could be inconvenient for looping parts as it would be included in the loop. Users not familiar with the dynamics of western notation found them unintuitive. Furthermore, these discrete markings do not make available the continuous volume scale possible with the interface.

When transcribing a piece with exact tempo, dividing the speed by 160 to enter the relative speed caused unnecessary work. There was also forgetfulness as to whether relative speed referred to how long the turtle spent in each cell or how quickly the turtle moved. Having completed the tutorial, users often had to check the position and meaning of turtle arguments.

As the number of dynamics and movement instructions grew, the instructions became long and it could be tough to parse and then establish turtle behaivour. Also, becuase "`s`" could be used to indicate sustain within cells, some users confused the "`s`" within the turtle instructions to mean sustain and not south.

Feedback

It was often unknown if pressing play registered, especially if the Excel workbook was saving delayed Excello being able to access the spreadsheet. Users also requested to see a summary of where the active turtle were in addition to the highlighting. If a turtle had accidentally been left activated, the entire grid had to be searched in order to locate it.

MIDI conversions

Many users who use production software said importing and exporting MIDI files would be helpful. If working with an existing MIDI file, it would be convenient to be able to convert that into the Excello notation. Exportation would allow Excello to be used to create chord sequences, bass lines and the piece structure, before adding additional effects and recorded lines in their digital audio work stations.

Sources of effort when writing

Once notes had been inputted into the grid, the number of cells the turtle had to move had to be counted. This is often in a straight line. Whilst the Excel status bar allows users to highlight a selection of cells and immediately see how many cells are highlighted, this is unproductive. This was particularly inconvenient when users were writing out a piece and periodically testing what they had written so far. Some users simply instructed turtles to move forward significantly more steps than required to prevent this counting. This is not feasible for looping parts. It was suggested that turtles figure out how far they should move.

Instructions involving repeated movements such as moving to the end of a line and jumping down to the beginning of a line below, instructions within the turtles required a lot of repetition.

Many of the notes in melodic lines would take place in the same octave. As such, repeatedly writing out the octave number was tiresome. One user made a comparison to LilyPond [13] where if the length of a note is not defined, the last defined note length would be used.

Some users find it more intuitive to think of a melodic line as the intervals between notes as opposed to note names. A modulated [2] melody line required the modulated part to be written out again and could not be derived quicker from the original version.

Chords

Most users used a very small subset of the available chords but had to scroll through the whole list to find these. Separating the more common chords for easier access was requested. Initially, notes inserted vertically had the lowest note at the top with notes increasing in pitch proceeding down the column. In western staff notation, higher pitch notes appear higher up the staff. As a result, it was suggested that inverting the order would be more intuitive. In the initial interface it was also unclear what the different drop downs corresponded to, with some users selecting the 7 from the octave number in order to try and insert a Maj7 chord.

Activation of turtles

When toggling the activation of a turtle, entering the edit mode for each cell containing a turtle definition to add or remove the exclamation mark was very tedious.

3.3 Second Prototype

Following the formative evaluation sessions and feedback, a series of additions and modifications were made to solve the problems and opportunities brought up.

[2] Where every note has been moved up or down in pitch by the same amount.

3.3.1 Dynamics

To help extract the path that the turtles follow and pair notes with their volume, dynamics are instead inserted in the cells along with the notes. A dynamic instruction is added after the note, separated by a space as in Manhattan [11]. As before, this will persist for all following notes until the volume is redefined. A single turtle definition with multiple start cells can now play parts of different volume. However, notes in the grid are limited to only being played at their given volume. To play the same notes at a different volume, a new path must be made where the cells defining the volume are replaced. Overall, the new system was believed to be more preferable.

In order to be able to make use of a full continuous dynamic scale, in addition to the existing dynamic symbols, a number between 0 and 1 can be provided where 0 will be silent and 1 is equivalent to fff.

3.3.2 Nested Instructions

Nested instructions with repetitions reduces the length of turtle instructions and allows for repeated sections or movements to be more easily incorporated. A series of instructions placed within parentheses with a number immediately following the closing parenthesis will be repeated that number of times. Whilst the fourth argument of the turtle will simply repeat the entire musical output of the turtle, repetitions within the turtle instruction allow paths to be defined more concisely.

3.3.3 Absolute Tempo

The turtle's speed is defined by cells per minute, rather than the relative value used initially. However, values less than 10 were interpreted in the original relative way to maintain backwards compatibility for the participant's existing work. To maintain consistency in a production version, this is removed so speed must be defined absolutely. The values given for speed and dynamics will be of different orders of magnitude and hence reduce the confusion that can occur between them.

3.3.4 Custom Excel Functions

Two custom Excel functions were implemented to aid composition. One to insert turtles into the grid and a second to transpose notes.

Excello.Turtle

By adding custom Excel functions, the existing formulae writing tools provided within Excel can be utilised. When using a built in formula, a prompt appears informing users which arguments go where and whether they are optional. The output of this function is text used to define a turtle if written manually. Other cells can be referenced for the arguments of the turtle function. For example all turtles could reference a single cell for

their speeds. This allows relative tempos to be easily implemented as the speed argument of each turtle could be a relative speed mutliplied by this global speed.

Excello.Modulate

A function to modulate notes provides easy modulation of existing sections of a piece and also the definition of a melodic line by the intervals between the notes. The function takes a cell and an interval and outputs any notes defined in that cell transposed by that interval, maintaining any dynamic definition. A section can be modulated by calling this function on the first note with a provided interval and using the existing drag-fill functionality of Excel to modulate all notes. By using the previous note that has just been transposed and one of a series of intervals as the arguments, a melodic line can quickly be produced from a starting note and a series of intervals.

3.3.5 Sustain

To prevent confusion between the instruction for a turtle to face south and sustains. The symbol "`-`" sustains a note. This was chosen because it is light and also has some similarity to a tie [3]. "`s`" in a cell is still interpreted as a sustain to maintain backwards compatibility for the existing work of the participants.

3.3.6 Active Turtles

In order to provide feedback that turtle definitions have been recognised, a list of the active turtles is given below the play button. This also helps find spurious turtles that were not intended to be activated.

3.3.7 Automatic Movement

To prevent counting the number of cells in a line, `m*` instructs a turtle to move as far as there are notes defined in the direction it is currently facing. If more notes are added on this line, the turtle instructions do not need editing before pressing play. There may be cases where a part is meant to finish with a number of rests. As a rest is notated with a blank cell, a method of increasing the length of the path to include these rests is required. A cell can be explicitly defined as a rest with "`.`". This would be required if multiple turtles were defining a repeating section where one does not have the final cell of the section being a note, sustain or multi-note cell. Without an explicit rest the turtle would stop and repeat too soon and the parts would be out of phase.

3.3.8 Inferred Octave

The octave number can be inferred by the program if omitted. Two methods were under consideration. Firstly, given that most intervals within music are small, the nearest note could be played. Whilst this method would likely require the least explicit statement

[3] A line to increase the length of a note by joining to another.

of octave number it would be non-trivial to figure out the octave of a given note. The last defined octave in the path would need finding and then all subsequent notes walked through keeping track of the octave. The second consideration was to always use the last defined octave. Whilst this may require many octave definition around the boundary between octaves, it is easier to find what octave a note is played at as it is simply the last defined octave in the path. The second option was implemented.

3.3.9 Chords

To aid entering common chords, common types are repeated in a separate group at the top of the type drop-down. The layout of the chord drop-downs was improved with labels added making it clearer what the values refer to. If the notes were entered vertically, the order was reversed to have a greater correspondence with traditional staff notation.

3.3.10 Activation of turtles

A "Toggle Activation" button was added to the add-in window. When a cell or range is highlighted in the spreadsheet, the activation of any turtle definitions in this range will be toggled when the button is pressed. This significantly increases the ease with which turtles can be seleted as only two clicks are required as opposed to having to enter the cell edit mode and add or remove an exclamation mark.

3.4 Final Prototype Implementation

This section discusses the underlying implementation of the final prototype, following the participatory design. Excello consists of three main parts. The first, and largest, is the turtle system for playing the grid contents. The second is the method for inserting the notes of chords into the grid. Thirdly, turtle input and modulation is made available through the custom Excel functions.

When the play button in the add-in window is pressed, the turtle definitions within the grid are identified. For each, the starting cell and movement instructions are used to establish the contents of the cells which it passes through. This is converted to a series of note definitions - pitch, start time, duration, volume. The speed and loop parameter are used to create the structure interpreted by the Tone.js library to schedule and initiate playback. An overview of the data flow and subtasks required to create the musical playback is show in figure 3.1.

An extension of the Tone instrument class is a Sampler. This interpolates between a set of pitched samples to create notes of arbitrary pitch and length. A sampler is loaded using the Salamander grand piano samples [4] which includes four pitches (out of a possible 12) per octave. This accurately interpolates notes whilst reducing loading times and storage requirements.

[4] https://freepats.zenvoid.org/Piano/acoustic-grand-piano.html

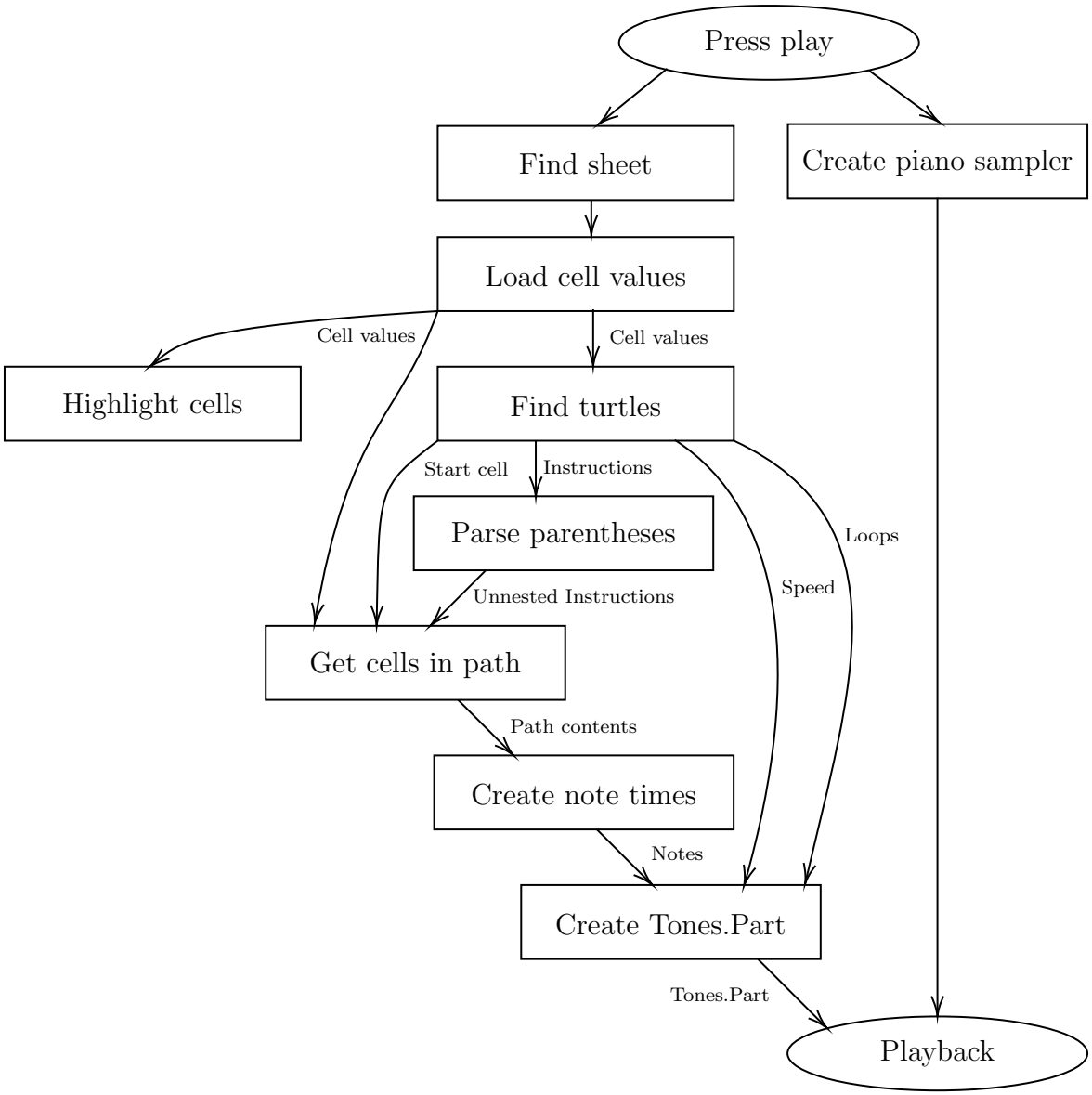

Figure 3.1: An overview of the playback algorithm and dataflow of Excello

3.4.1 Identifying Cells

The drop-down menu for sheet selection is populated with the current sheets in the workbook. When a user presses play, the cell values from the selected sheet are loaded using the Office API. Once the sheet contents have been loaded, the cell contents can be analysed for highlighting and calculating turtle paths. Cells containing at least one definition of a note are highlighted red. A cell defining a note must contain a note name. There may also be an accidental, octave number, or a volume instruction in the form of a dynamic marking or number between 0 and 1. This is tested using the following regular expression:

```
'^[A-G](#|b|)?[1-9]?( (0(\.\[0-9]+)?|1(\.0)?|ppp|pp|p|mp|mf|f|ff|fff))?$'
```

Cells containing multiple note definitions are split using commas. All subsequent strings are trimmed of starting and ending whitespace and then must either ratify the test for a note or be a sustain ("`-`" or "`s`"), explicit rest ("`.`") or an empty string from an implicit rest. Cells matching "`-`","`.`" or "`s`" are highlighted a lighter red. Turtle definitions are tested using:

```
'^(!turtle\().*(\))$'
```

and cells containing turtle definitions are highlighted green. The same regex is used to identify definitions of turtles. The address of cells containing a turtle definition are added as text nodes to the live turtle section of the add-in window.

3.4.2 Parsing Movement Instructions

The arguments given in the turtle are separated. First the movement instructions are converted to a single list of movements, without bracketed instructions, (e.g. "`(r m2)2`" becomes "`[r, m2, r, m2]`") so that the path of the turtle can be established. It initially seemed that the `parse` method of the Parenthesis[5] library would be suitable for aiding in this string manipulation. This parses strings containing parenthesis into an array with nested format. For example, `parse('a(b[c{d}])')` gives `['a(', ['b[', ['c{', ['d'], '}'], ']'], ')']`.

This suggests that a string like "`(r m2)2`" would become `['(', ['r m2'], ')2']`. By removing the brackets from the strings within the array, a simple recursive method could be built to output `'r m2 r m2'` from `[['r m2'], '2']`. However upon testing this, an array with undefined contents was outputted. From further tests and investigation of the source code I established that strings with a number following a closing parenthesis would all cause such an error. A possible solution allowing the library to be used would be to substitute characters for numbers or place a symbol before all numbers and then to later revert this change. Instead, using the method employed by the Parenthesis function as inspiration, I implemented my own parsing function.

This works in two main steps. First the deepest bracketed expression (one containing no brackets within it) is identified and stored in an array with the brackets removed. This expression is replaced in the original string with the string '$\_\_\_x\_\_\_$' where x is the index expression stored in the array. This is repeated until the original string contains no brackets. Secondly, a recursive algorithm uses the indices places between the '`___`' to reconstruct the string in the desired array format. This method is outlined in algorthitm 1. The Typescript implementation is in appendix **??**.

Having submitted an issue on the Parenthesis Github reporting the bug, and implemented my own method for parsing the turtle movement instructions, I implemented a fix to the Parenthesis library. The existing function performed the initial replacement with

[5] https://www.npmjs.com/package/parenthesis

Algorithm 1 Parsing bracketed expression. `str.replace(regex,f)` performs `f(s)` on the first substring, `s`, of `str` matching the regular expression `regex`.

```
1: procedure parseBrackets(str)
2:     idPadding ← '___'
3:     unnestedStr ← []
4:     deepestLevelBracketsRE ← RegExp('\\([^\\(\\)]*\\)')
5:     replacementIDRE ← RegExp('\\' + idPadding + '([0-9]+)' + idPadding)
6:
7:     procedure replaceDeepestBracket(x)
8:         unnestedStr.push(x.substring(1, x.length-1))
9:         return idPadding + (unnestedStr.length - 1) + idPadding
10:    end procedure
11:
12:    while deepestLevelBracketsRE.test(str) do
13:        str = str.replace(deepestLevelBracketsRE,replaceDeepestBracket)
14:    end while
15:    unnestedStr[0] = str
16:
17:    procedure reNest(outerStr)
18:        renestingStr ← []
19:        while There is a match of replacementIDRE in outerStr do
20:            matchIndex ← index of the match in outerStr
21:            matchID ← ID of the match (number between padding)
22:            matchString ← matched string
23:
24:            if matchIndex > 0 then
25:                renestingStr.push(outestStr.substring(0, matchIndex))
26:            end if
27:            renestingStr.push(reNest(unnestedStr[firstMatchID]))
28:            outestStr = outestStr.substring(matchIndex + matchString.length)
29:        end while
30:        renestingStr.push(outestStr)
31:        return renestingStr
32:    end procedure
33:
34:    return reNest(unnestedStr[0])
35: end procedure
```

the string '___x'. Therefore the x and following numbers would concatenate forming a single number, causing the library to fail. By modifying the existing Parenthesis code so that it utilised my method of having an identifier before and after the index number I was able to fix the issue. I added additional tests to the project to verify that this worked and ensured that previous tests all passed before submitted a pull request to the developers. This has since been merged into the library and published.

I wrote an additional recursive method to unnest the outputted array of this function into a single stream of instructions. An empty string, s, is initialised. For each item in the array, if it is an array, unnest the contents recursively. If not, it will be one or more single movement instructions. If the last item was an array the result of the array being unnested is added to s. If the first item in the single movement instructions is a number, the result of the array being unnested is added to s that number of times. The remaining instructions are added to s. This is outlined in algorithm 2. The implementation is shown in appendix **??**.

Algorithm 2 Unnesting a parsed bracketed expression.

```
1:  procedure processParsedBrackets(arr)
2:      s ← ''
3:      previousArr
4:      for v in s do
5:          if v is an array then
6:              previousArr ← processParsedBrackets(v)
7:          else
8:              if previous instruction was an array then
9:                  s ← s + previousArr
10:                 if next instruction is a number then
11:                     s ← s + previousArr, that number of times minus one
12:                 end if
13:             end if
14:             s ← s+ remaining instruction in v
15:         end if
16:     end for
17:     return s
18: end procedure
```

3.4.3 Getting Cells in Turtle's path

If the first argument in the turtle is defining a range of starting cells the cell addresses within this range are calculated. For each starting cell, the unnested instructions and sheet values are used to determine the contents of the cells the turtle passes through. When dynamics were defined within the turtle instructions this also returned the volume at which each cell was returned. Volume is now handled in the next step. This process

models the movement of the turtle within the grid. Keeping track of where it is positioned and which was it is facing. For each instruction the position and direction is updated as required and the contents of any new cells entered added to a list of notes.

Additional computation is required when the "`m*`" instruction is used, as the number of steps the turtle should take must be computed. Given the current position of the turtle and direction it is facing, all the cells in front of it are taken from the sheet values. The turtle should step to the last cell that defined a note, sustain or explicitly defines a rest. The number of steps is the length of the array minus the index of the first element satisfying this criteria in the array reversed.

3.4.4 Creating Note Times

For each turtle, for each starting cell, the cells moved through are calculated. This is used to create a data structure containing the information required for each note for playback to be initiated using the Tone library. For each turtle, the following array is produced: `[<Note 1>,,<Note N>], <number of cells>]`. Each note is as follows: `[<start time>, [<pitch>, <duration>, <volume>]]`. This is also the point at which the dynamics and the octave are added to each note if it had been omitted from a cell.

The Tone library has many different ways in which time can be represented. I opted to use Transport Time for all time measurements - start times and durations. This is in the form `'BARS:QUARTERS:SIXTEENTHS'` where the numbers used do not have to be integer values. This allowed me to simply use the quarters value to represent number of cells and not have to worry about exact times, ticks or what musical note a length corresponded to (tricky for arbitrary subdivisions).

The note sequence array is initiated by counting the number of notes that are defined in the cells passed to the method. This is done using the regular expressions for identifying notes and multi-note cells. The cells are iterated through in one scan keeping track of the active note and adding it to a sequence of notes when it ends. Outside of this loop, Variables are defined to keep count of how many cells and notes through the process the algorithm is and whether the current value is a rest or note. Variables keep track of the note currently being played - when it started (`currentStart`), the pitch (`currentNote`) and volume (`currentVolume`). As volume and octave number, variables are also required to keep track of these.

Table 3.2 outlines the actions carried out when a cell is read. When a note is added to the note sequence, it is added in the form `[currentStart, [currentNote, "0:" + noteLength + ":0", currentVolume]]`.

The same method can be used for multi-note cells, except the note length and cell count must be incremented by the appropriate fraction of a cell rather than by one for each note that is processed.

Table 3.2: The actions taken when processing each cell to create note times per turtle

Cell	State	Action		
Note	Note	Note, octave and volume established from cell contents and previous values	Previous note added to note sequence	currentStart = '0:'+beatCount+':0'
				currentNote = value
	Rest		inRest = false	noteLength = 1
				currentVolume = volume
Sustain	Row	noteLength++		
	Rest	Nothing (has no semantic value)		
Rest	Note	Previous note added to note sequence inRest = true		
	Rest	Nothing		

If at the end of the final cell, the state is in a note, this must be finished and added to the note sequence.

The notes definitions in the note sequence are sufficient to play a note using the piano sampler using the triggerAttackRelease function. The Tone.Part allows a set of calls to this method to be defined which can be started, stopped and looped as a single unit. Using the note sequence ("noteTimes"), and number of cells ("beatsLength") from creating the note times, playback is scheduled with the following code:

```
var turtlePart = new Tone.Part(function(time: string, note: [string, string, number])
  piano.triggerAttackRelease(note[0], note[1], time, note[2]);
}, noteTimes).start();
if (repeats>0){
  turtlePart = turtlePart.stop("0:" + (repeats*beatsLength/speedFactor) + ":0");
}
turtlePart.loop = true;
turtlePart.loopEnd = "0:" + beatsLength + ":0";
turtlePart.playbackRate = speedFactor;
```

3.4.5 Chord Input

When the insert chord button is pressed the note, type, inversion[6] and octave of the chord are extracted from their HTML elements. The tonal library can then be used to generate the notes of the scale:

```
var chordNotes = Chord.notes(chordNote, chordType).map(x => Note.simplify(x));
```

The tonal simplify function is used to simplify reduce note definition involving multiple accidentals to contain at most one, thereby conforming to the notation interpreted by Excello. This provides a list of notes in ascending order but without octave or taking into

[6]which note of the chord is the lowest, the chord ascends from this. No inversion is root position, then first inversion, second inversion, etc.

account the inversion of the chord. In order to reach the correct inversion of the chord, the array of notes is rotated by the inversion number.

Octave numbers are added by iterating through the notes produced. A dictionary matches note names to position in the chromatic scale starting at C (the first note of the octave in scientific pitch notation). This is also accounts for enharmonic notes[7]. The given octave number is appended to the first note in the chord. For each preceding note, if it appears in an equal or lower position in the octave than its predecessor, the octave number is incremented before appending it to the note name. Otherwise, it is in the same octave and hence the octave number is appended without modification.

The Office API is used to acquire the range that has been selected by the user. The notes of the chord will be entered starting at the top-left corner of this range. If the hight of the range is greater or equal to its width, the notes are entered vertically going down from the starting cell. Otherwise they are entered horizontally going right. This is done by building the 2D array where the chord will be entered and setting that range using the Office API.

3.4.6 Custom Excel Functions

I built another Excel add-in in order to implement custom functions. As opposed to offering a separate window as the main Excello add-in does, this one allows additional functions to be used by users in the cells using the prefix "=EXCELLO.". The file structure was generated with the Yeomann generator. The name, description, result type, and parameter names and types are store in a JSON schema. This is used by Excel to provide argument prompts and autofill to the user when editing the formula. Functions were also given an identifier used to link them to a typescript file where the functions were defined.

The turtle argument simply concatenates the given arguments into the correct format for Excello to recognise as a turtle. This allows other cells to be referenced, for example the speed variable can reference a global tempo variable as shown in figure **??**.

The modulate function first establishes if the cell if a note or multi-note. For a note, if there is a volume defined, the note is separated, modulated using the tonal transpose function and then combined back with the volume. This is performed for every element in the multi-note that is a note definition. This allows the drag fill feature of Excel to be employed by the user for transposing sections or to define melodic lines using the interval between notes as shown in figure **??**.

[7]Notes that are the same pitch but different names, such as Ab and G#

3.5 MIDI Converter

The following section shall document how the Python converter from a MIDI file to CSV file suitable for Excello playback works. A MIDI file is divided into up to 16 parallel channels [**?**]. Each channel contains a series of messages defined using predefined status bytes and data bytes. I used the Python Mido library to read MIDI files and abstract away from the underlying byte representations. The onset and offset of notes are treated as two separate events with two separate messages [**?**]. A note onset or offset message includes the note pitch and velocity, channel and time in ticks since the last message [**?**]. The channel is not relevant for this conversion. Other messages define information not relevant for the conversion for Excello, but the message times must still be taken into account.

The first step converts a list of messages to a list of notes defined by onset and offset time, pitch and velocity. For each track, the messages are iterated through, using the time value in every message to update a variable tracking time. This is performed for every message as piano pedal presses or meta messages have non-zero time. If the message defines a note onset, this is added to a dictionary mapping pitches to a list of currently active notes. Lists are used because a pitch can be active multiple times at once. For a note offset message, or onset message with zero velocity, the note popped from the active notes at that pitch has its end time added and then it is added to the list of all notes defined in the file.

As each turtle can only define one note at a time, the notes are then split into separate lists such that no list contains two notes which are playing at the same time. As long as the list of notes is non empty, a new list is created. The first remaining note is added to the new note and removed from the list of all notes. The next remaining note starting after the previous note ends is added to this new list. All remaining notes are iterated over. The number of iterations required corresponds to the number of turtles required, n.

It would be simple to have a tick correspond to a cell. With this, any combination of note onsets and offsets could be accurately represented in Excello. In order to achieve better compression, start and end times are converted to the cell number within the path of the turtle. For many MIDI files, the duration of a notes, will be different to the time it is notated to occupy. For example, a note immediately followed by another note may have an end time significantly less than the start time of the next note. As a result a method is required to account for this. For all notes, before separation into the streams for different turtles, the length of the notes in ticks and differences between consecutive start times are found. The minimum or modal values for these times are calculated depending on the level of compression giving the $lengthStat$ and $differenceStat$.

$$ratio = \max(lengthStat, differenceStat) / \min(lengthStat, differenceStat)$$

$$ratio_{int} = \lfloor ratio \rfloor$$

$$correction = ratio/ratio_{int}$$

For each note, the times are adjusted as follows:

$$\begin{aligned} length &\leftarrow (start - end)/lengthStat \quad \texttt{(rounded to the nearest 0.1)} \\ start &\leftarrow start/difference \times ratio_{int} \quad \texttt{(rounded to the nearest 0.1)} \\ end &\leftarrow start + length \end{aligned}$$

The next step converts the streams, with note start and end times corresponding to cells, into a CSV file to be run with Excello. The path for each turtle is an array of empty strings is initialised. The length of these arrays is the maximum end time for a note in any turtle, T. For each note the turtle plays, this is entered into the corresponding cell. MIDI defines pitch using the integers. I used the library audiolazy [8] to convert from MIDI number to scientific pitch notation. If the velocity of the note if different to the previous note played by the turtle (or the first note played), the eight-bit velocity as defined by MIDI is mapped to the range [0,1] as used by Excello. If the note is greater than one cell, a sustain is placed in the following cells so the note will be the correct length. These paths go right starting in column A, with the first in row 2.

Finally the definition of the turtle must be placed in the spreadsheet. The start cell range is 'A2:A$(n+1)$'. The movement instruction is "r mT". The MIDI file contains meta data for the tempo (milliseconds per beat) and ticks_per_beat. Cells per minute is calculated as follows:

$$\begin{aligned} &cells\ per\ tick \times ticks\ per\ beat \times beat\ per\ minute \\ &= \frac{ratio_{int}}{mode\ difference} \times \texttt{ticks\_per\_beat} \times \frac{60 \times 10^6}{\texttt{tempo}} \end{aligned}$$

Using one as the number of repeats, the turtle definition is placed in cell A1 and the CSV exported.

3.6 Repository Overview

TODO

[8] audiolazy library

Bibliography

[1] Samuel Aaron, Alan F. Blackwell, and Pamela Burnard. The development of sonic pi and its use in educational partnerships: Co-creating pedagogies for learning computer programming. *Journal of Music, Technology and Education*, 9:75–94, 05 2016.

[2] Ron Goldman, Scott Schaefer, and Tao Ju. Turtle geometry in computer graphics and computer-aided design. *Computer-Aided Design*, 36:1471–1482, 2004.

[3] Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: a tutorial. Technical Report Version 1.2, BCS HCI Conference, 1998.

[4] Sven Gregori. Never mind the sheet music, heres spreadsheet music, 2019.

[5] Allen Huang and Raymond Wu. Deep learning for music. *CoRR*, abs/1606.04930, 2016.

[6] D.M. Huber. *The MIDI Manual: A Practical Guide to MIDI in the Project Studio*. Taylor & Francis, 2012.

[7] Alex Mclean, Dave Griffiths, Foam Vzw, Dave@fo Am, Nick Collins, and Geraint Wiggins. Visualisation of live code. 01 2010.

[8] Alex Mclean and Geraint Wiggins. Texture: Visual notation for live coding of pattern. 01 2011.

[9] Mozilla. Web audio api. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API`, 03 2019. Accessed: 2019-04-02).

[10] Michael Muller and Sarah Kuhn. Participatory design. *Communications of the ACM*, 36:24–28, 06 1993.

[11] Chris Nash. Manhattan: End-user programming for music. In *NIME*, 2014.

[12] Simon Peyton Jones, Margaret Burnett, and Alan Blackwell. A user-centred approach to functions in excel. June 2003.

[13] Erik Sandberg, Examensarbete Nv, Reviewer Arne Andersson, and Examiner Anders Jansson. Separating input language and formatter in gnu lilypond, 2006.

[14] Advait Sarkar. Towards spreadsheet tools for end-user music programming. In *PPIG*, 2016.

[15] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 207–214, Sep. 2005.