MUSIC INFORMATION RETRIEVAL AND AI: AUTOMATING MUSIC PERFORMANCE

Exploring audio analysis and machine creativity through
an automatic DJ system for house music.

June 2021

Alexei Smith

Supervised by:
Prof Andy Hunt
&
Dr Yihua Hu

4th Year Project Final Report for Degree of MEng in
Electronic Engineering with Music Technology Systems

Department of Electronic Engineering,
University of York

Abstract

Music Information Retrieval (MIR) allows musical knowledge to be extracted directly from audio data. The field of research is maturing and there are now many powerful techniques in temporal, tonal and higher-level analysis, which can be used in combination to build intelligent music systems. A number of automatic DJ systems have been developed in the past, but the primary focus has been on modelling a human DJ using MIR algorithms. This project looks at automatic DJ software from a user perspective, investigating how it can be designed to meet user needs, as well as where it fits into the larger world of DJing and music performance. AutoDJ is a fully-automatic desktop app, written in C++, which can produce a coherent mix of dance music tracks. The mixing decisions are driven by a combination of beat tracking, key signature detection, track segmentation and danceability analysis. Powerful off-beat transients are a stylistic feature of house music and surrounding dance-focused genres, which often confuse the phase estimation of beat-tracking algorithms. A novel method for correcting beat phase estimations is discussed, which uses correlation with pulse trains (Percival & Tzanetakis, 2014) to establish whether an existing estimate is offset by half a beat period. A pipeline for modelling the creativity of a human performer is proposed, with a discussion of how this can be integrated into AutoDJ. In a test with 18 users, the software was rated highly and results suggested there is a strong use case for automatic DJ software on desktop. Future open-source development objectives are proposed, with the aim of readying AutoDJ for a wider test and subsequent release. The testing methodology used here can serve as a framework for further investigations into the automation of music performance.

Note

A video demonstration of the system can be accessed here, which includes a comparison to a real DJ performance, using the same tracks.

All of the citations, sections and figure numbers in this report are clickable - please use them for quick navigation around the document. Other hyperlinks are shown in a blue font.

Dedicated to Derek Smith, for his incredible support throughout my education.

Declaration

I declare that this report is a presentation of original work and I am the sole author. All sources are acknowledged as References.

With the exception of the Introduction and Literature Review sections, which have been adapted from my initial project report, this work has not previously been presented for an award at this, or any other, University. The Literature Review has been significantly expanded in all research areas.

This project was examined in line with the University of York's ethics protocol and no issues were found.

Acknowledgements

Many thanks to my supervisor, Prof Andy Hunt, for his guidance in this project and my wider studies. His passion and energy inspires every student in the department, and has pushed me to achieve my best work. Thanks also to my second supervisor, Dr Yihua Hu, for his useful assistance in this project.

Thank you to the London Audio Team at Sony Interactive Entertainment - in particular, Mike Jones, Marina Villanueva-Barreiro and Danjeli Schembri. Your help and advice during previous projects laid the foundations for the development of AutoDJ.

Thank you to Elizabeth Smith and Katie Maddock for their proofing work, and Yasmine Leary for her endless support. Oli Still and Oli Clarke provided great company and all-important motivation to keep pushing in the latter stages of the project.

Finally, I'd like to thank the 18 test participants, who helped greatly with their feedback on AutoDJ.

Contents

1 Introduction

Music performance is the oldest and perhaps purest form of music. Long before the inception of writing or recording music, people were congregating to perform together. Studies have shown that music has played a part in every known society, past and present [6], so its performance is clearly of great cultural and historical importance.

Being such an integral part of society, it makes sense that music performance has evolved throughout history, responding to developments in culture, lifestyle and technology. In the last few decades, we have witnessed possibly the greatest ever shift in music performance: the rise of digital technology. From digital effects pedals that can manipulate analogue signals, to powerful sequencers that can be used to construct a full song on the fly, digital technology has had a profound impact on performance. The technology is versatile, in that it can be used as a simple augmentation to performance, or as the centrepiece. Furthermore, the user can involve themselves to varying degrees, building a system that requires continuous manual control, or just small, sporadic interventions.

Figure 1.1: A live music performance using a mixture of analogue and digital hardware: watch on YouTube [13].

This project examines the extreme case, in which the power of modern technology is used to fully automate a musical performance. Here, software has been developed that can analyse a music library and make its own creative decisions on how to manipulate music, in order to create a continuous soundtrack. The art of creating a continuous soundtrack from pre-recorded music, particularly in the context of live performance, is called Disc-Jockeying (DJing).

1.1 A Brief History of the Disc-Jockey

Disc-Jockey originally referred to a controller (jockey) of circular vinyl records (discs). They play a series of songs, usually mixing them together to create a continuous soundtrack. The term was coined in the 1930s to describe the first radio host that aired shows of fully pre-recorded music, Martin Block [34]. He introduced the world to the idea that you could perform using recorded music, without any musicians present.

These days, the term is almost always shortened to DJ, and its definition extends to any performer who uses some form of recorded music – analogue or digital – to create a continuous soundtrack. As well as on the radio, DJs are found in nightclubs, where they guide the emotions of a crowd through a carefully orchestrated mix of dance-focused tracks. DJing has come a long way since its inception and is now surrounded by a rich culture of music lovers worldwide. The practice has become something of an art, as the diverse community has developed a number of different techniques and styles – many of which are showcased in Red Bull's 3Style World DJ Championship.

Modern technology allows DJs to manipulate music in new ways. A professional system typically provides a great number of DSP effects and filters, which can be used to seamlessly blend two tracks, or to create dramatic climaxes in the performance. While professional systems remain prohibitively expensive for hobbyists and amateurs, entry-level options are becoming increasingly accessible.

Digital systems, such as the one shown in Figure 1.2, also offer convenience features, which can automate certain parts of the DJing process – something particularly relevant to this project. Automation is sometimes looked at with disdain, and there is an air of elitism surrounding vinyl DJing: its fully manual nature is seen by some as the only authentic method. This project challenges that viewpoint and investigates whether there is a place where automated DJ performances can be embraced.

Figure 1.2: A professional DJ setup with vinyl turntables (outside), digital turntables (inside) and a mixer (centre) that optionally communicates with mixing software on a laptop [89].

1.2 Musical Terminology

Before going any further, it is useful to define some of the musical terminology to be used:

Track	A music recording, i.e. a song.
Rhythm	Repeating musical patterns in time.
Beats	The primary 'pulses' within a rhythm.
Downbeat	The first beat in a measure
Time Signature	The number of beats within a measure.
Tempo	Musical speed, the rate of beats.
BPM	Beats-per-minute, the unit of measurement for tempo.
Melody	A sequence of musical notes.
Harmony	Combinations of musical notes.
Key Signature	The set of musical notes used in a piece of music.
Tonality	The mood / emotion conveyed by a harmony or key signature.
Dynamics	The contrast of loud and quiet, i.e. energy variation.
Artefacts	Audible fragments of noise, caused by errors in audio data.
MIDI	Musical Instrument Digital Interface: A communication standard for music hardware.

1.3 DJ Fundamentals

A high-level overview of the DJ workflow is shown in Figure 1.3. The yellow component represents the preparation stage, which can either be done in advance, or during a performance; the green section is an integral part of the performance itself.

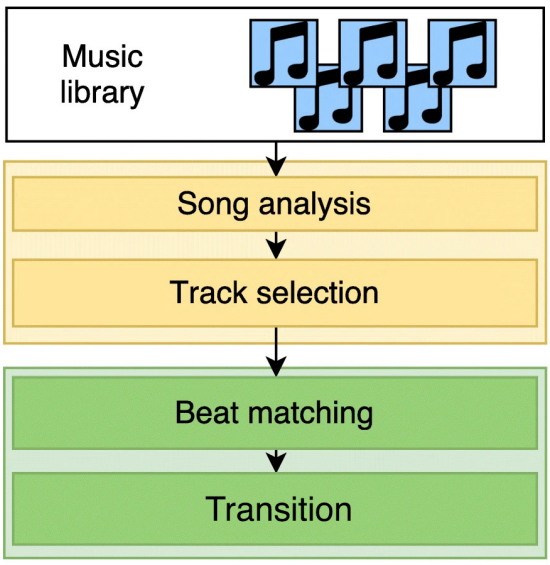


Figure 1.3: The fundamental performance workflow of a DJ [100] (modified).

First, a DJ will analyse their music library, through manual or automatic means, to build up a database of musical knowledge. This could include information about the tempo, dynamics and key signature of each track, for example. Tracks that are in the same key signature will give a harmonically pleasing sound, because they use the same set of musical notes.

The extracted information then informs the track selection stage, where the DJ chooses which tracks to play and in what order. Some DJs will curate their whole performance into a playlist in advance, but most will bring a larger set of tracks to draw from, allowing them to adapt to the mood of the crowd in real-time.

During a performance, the primary task is making transitions between consecutive tracks. Before this can happen, though, the DJ must *beat-match* the music. This involves time stretching one

or both tracks until they are at the same tempo, allowing for synchronised playback during the transition. Figure 1.4 illustrates the importance of beat-matching: notice that synchronised pulses result in a rhythmically coherent output, which in turn helps listeners keep time with the music, especially when dancing.

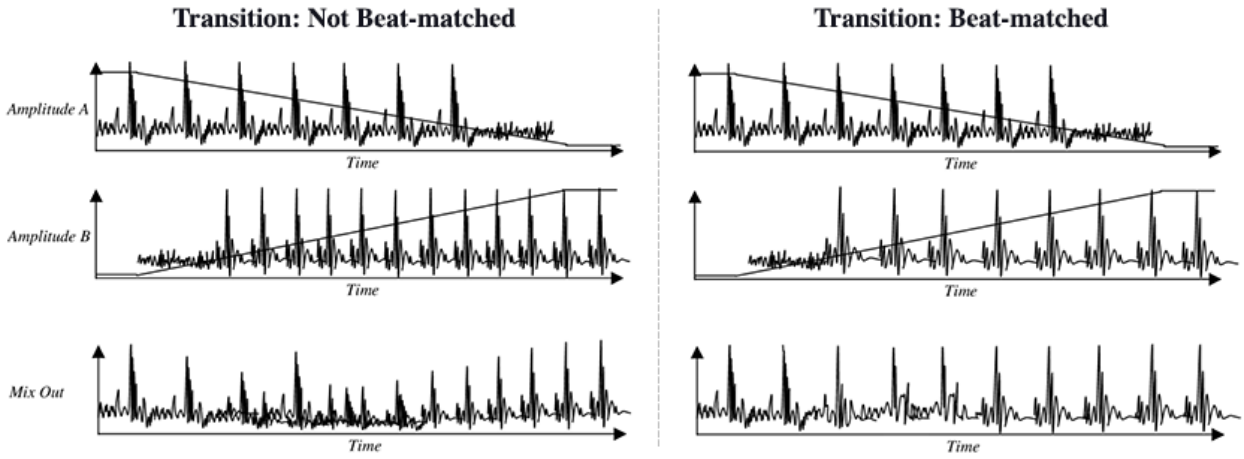


Figure 1.4: Audio magnitude plots to show the impact of beat-matching concurrent tracks. When beat-matched, the rhythmic pulses of both tracks are synchronised to produce a coherent output [20] (modified).

Where in each track to make a transition is an important consideration; the DJ usually chooses sections with similar energy or rhythm. The DJ waits for the current track to reach the chosen location, and then begins the transition. At the most basic level, the volume of the outgoing track is gradually reduced, while the incoming track volume is increased. This process, called a *crossfade*, is represented by the ramping lines in Figure 1.4. Of course, a skilled DJ will incorporate more advanced techniques to make a transition interesting. In the world of dance music, you usually hear long, gradual blends of rhythmic drum beats and chords, while genres such as hip-hop lend themselves to much shorter transitions, perhaps involving scratching.

Transitions are usually synchronised with distinct musical sections, that is, they often start and end at moments where the music changes significantly. In dance music, most tracks follow a repeating arrangement of builds and drops - i.e., tension and release. DJs tend to start and/or end their transitions at the *boundaries* between these sections.

1.3.1 Golden Rules

As with any art form, the artist can express themselves however they like - there is no 'correct' way to do things. Nevertheless, there are a set of unspoken rules which will generally help improve a performance and avoid the dreaded *empty dancefloor*.

Keeping Synchronisation

The beat-match between two tracks may not be perfectly accurate, meaning their tempos could differ by a small margin. The is often the case when mixing with vinyl records, as the DJ can't precisely measure the BPM shifts being applied. In this case, they must keep the tracks synchronised by listening to the transient elements, and shifting one of the tracks if they slip out of time.

One Voice

The communication between a singer and a listener is something of a conversation. The singer conveys ideas and emotions through their choice of words and melodies, and the listener(s) responds with their own emotions, dancing, and perhaps singing along.

Everyone can agree that listening to two conversations at once is a difficult task, and the same applies in music. DJ transitions that result in conflicting vocals are generally avoided. DJs will often choose a section with no singing in one track, to make space for it in another. This is part of what determines the length of a transition, which helps to explain why vocal-heavy genres like pop and hip-hop see much quicker transitions than their instrumental counterparts.

One Bassline

There is a reason bands tend to have only one bassist. The frequency range of 250Hz and below is an important one, but it can become cluttered, or *muddy*, very easily. Like the 'one voice' technique, a DJ usually chooses one bassline to bring forward during a transition, so that listeners need only focus on a single melody in the low end. This is achieved by choosing a section of one track without a bass element, or by using filters to manipulate the low frequencies.

1.3.2 Getting Creative

Aside from the fundamentals of mixing, a DJ should embellish their performance with more advanced techniques to keep the listener's interest. The performer can play with the crowd's expectations, creating the ebb and flow of tension and release. Filters and effects can be used to create climactic moments and ethereal breakdowns throughout the set.

Modern mixing hardware, as shown in Figure 1.5, provides a great variety of DSP audio effects to apply to single channels, or the whole mix. In this example, the effects are controlled through a number of parameters (from top down): effect type, audio channel, time length (e.g. reverb size, echo decay rate) and intensity.

Figure 1.5: The effects section of an industry-standard Pioneer mixer [82].

1.4 Research Aims

This project investigates the automation of musical performance, through the lens of accessible DJ software. The software perspective was chosen because it provided a strong, flexible framework in which to conduct practical research. The primary research aims of the project were as follows:

- Explore how an automatic music performance engine can mimic human creativity
- Investigate how audio analysis techniques can be used to build a digital understanding of recorded music
- Design a software interface that is accessible to users of any musical ability level

These high-level aims presented three primary research areas, which are shown in Figure 1.6. To encourage a varied investigation, each area was deliberately broad. The areas also overlap somewhat, to keep the project coherent as a whole. Each topic will be examined in detail in the following section.

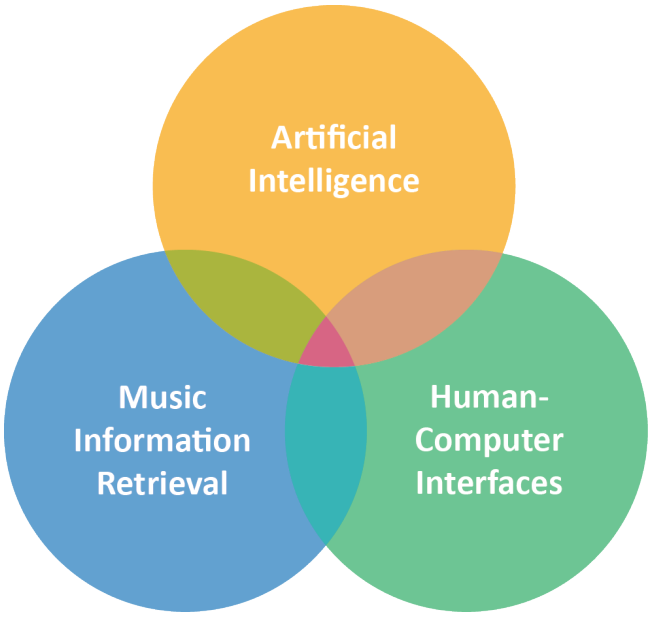


Figure 1.6: Venn diagram to show the high-level research areas corresponding to the project aims.

2 Literature Review

2.1 Artificial Intelligence

In the first half of the 20th Century, the world became familiar with the idea of artificial intelligence: machines that could reason and make decisions by themselves. This was not through a ground-breaking paper or the muses of a great philosopher, but with the boom of science fiction. Before we could fathom how one might achieve artificial intelligence, or whether it was even possible, such characters existed in popular culture, such as the humanoid robot in the 1927 film Metropolis. This meant that "by the 1950s, we had a generation of scientists, mathematicians, and philosophers with the concept of artificial intelligence (or AI) culturally assimilated in their minds" [10].

Early research theorised that the lofty ambitions of fiction writing might someday become a reality. Turing's 1950 paper, "Computing Machinery and Intelligence", discussed how we might test a machine's ability to think and to solve problems in a human manner [96]. Shortly after, Simon, Newell and Shaw developed the *Logic Theorist*, which was "perhaps the first working program that simulated some aspects of peoples' ability to solve complex problems" [44]. The late 1960's saw the inception of Moore's law, which fortified enthusiasm in the field of AI research.

Figure 1.1. Timeline of early AI developments (1950s to 2000)

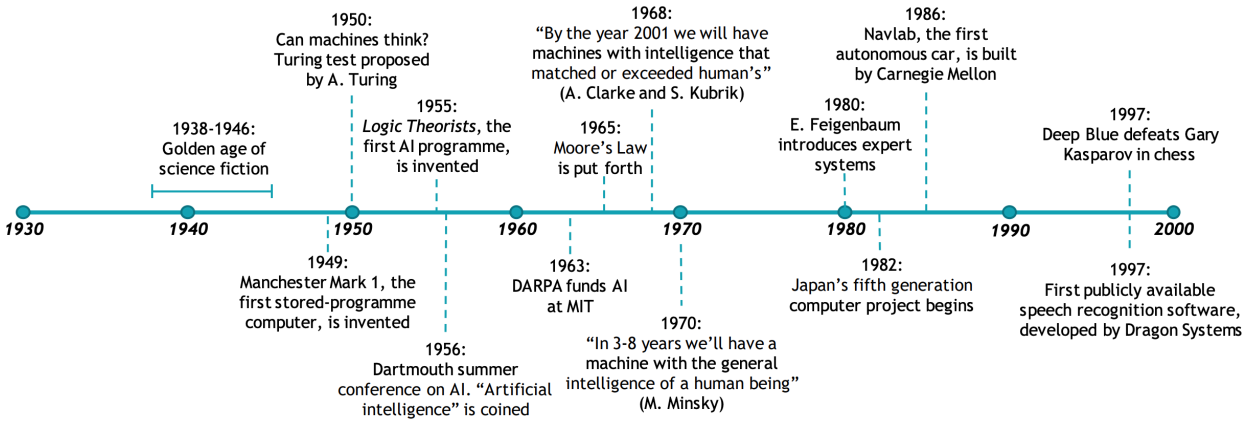


Figure 2.1: Timeline of early AI developments [31].

Today, we still don't have artificial systems matching the comprehensive cognitive abilities of humans, but there are certainly some that can outperform humans in specific tasks. This is called Artificial Narrow Intelligence (ANI) [55], which is an important distinction from Artificial General Intelligence (AGI), because ANI solutions must be tailored to a very limited range of problems. We have not progressed very far towards achieving AGI and some argue it is, in fact, impossible [35].

Nevertheless, ANI has accomplished many feats that once seemed impossible. A significant milestone was the 1997 defeat of the then World Chess Champion, Garry Kasparov, by IBM's Deep Blue supercomputer. More recently, Google's Alpha Go was victorious in its 2017 matches against the world-leading Go player, Ke Jie. Go is "considered to be one of the world's most complex games, and is much more challenging for computers than chess" [77].

Interestingly, [10] suggests that the approach to developing intelligent systems has not changed significantly, but that increasing computing power is the main factor in AI progress. It could be argued that, while increased power is essential for achieving machine intelligence, it is innovation in system architecture that has led to most of the recent breakthroughs - the most popular example being deep learning.

2.1.1 A Note on Machine Learning

Machine Learning (ML) is an area of artificial intelligence that is experiencing ever-increasing research interest, for good reason. ML, especially deep learning, has unlocked impressive new capabilities in the computing space, solving many problems that were previously unapproachable. ML is particularly good at building an abstract understanding of data, which can be used to drive real-time decisions in a human-like manner - something of particular relevance to this project.

Unfortunately, the use of ML introduces several development challenges, which made it impractical for use in the present work, given the time and hardware constraints. These include:

- **Training time:** Unless very powerful hardware is available, it can take days or even weeks to train an advanced ML model. Furthermore, a number of tuning iterations are typically required before a suitable architecture is reached.
- **Training data:** Large data sets are required for training the model. It would be difficult to find data sets relevant to the project aims, and creating bespoke sets is a long process.
- **End-user hardware:** Despite increasing adoption of ML hardware acceleration, it is not guaranteed that a model can be run efficiently on end-user hardware. For a real-time audio application like the one developed here, efficiency is a key requirement.

2.1.2 A Moving Target

One of the key figures in the field of AI, John McCarthy, said: "As soon as it works, no one calls it AI anymore" [73]. Rodney Brooks, the director of MIT's Artificial Intelligence Laboratory concurs: "Every time we figure out a piece of [AI], it stops being magical; we say, 'Oh, that's just a computation'" [58]. There is indeed a trend that, if a new AI system presents value to the masses, it will gradually become commonplace, after which it is not considered to be especially intelligent. The idea is well illustrated in an inquiry by UK Parliament on the state of AI, which describes a series of activities that would have been inconceivable a decade ago, but are very much a reality today [21]. This concept is somewhat present in the dictionary definition of AI:

"Artificial Intelligence: The theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision making and translation between languages." [25]

As technology progresses, our idea of what 'normally requires human intelligence' is forever changing, and the interpretation of AI is adjusted accordingly. On the other hand, the mention of decision making here captures an important trait of AI that is much more constant. The intelligence of an artificial system is not determined only by what novel functions it can perform, but its ability to make its own decisions about the data it receives. These are systems where decisions are not explicitly programmed, or controlled by the user, but determined by the systems themselves, based on a combination of factors, such as input data, past and present state, and randomness.

The decision-making aspect of AI is important in this project. It can influence the route taken by a music performance system in a way that mimics the artistic choices of a human, hopefully resulting in a more interesting and stimulating performance. This artistic decision making can be compared to a phenomenon deeply ingrained in all music: creativity.

2.1.3 Creativity

It is generally agreed that creativity is a universal trait of human intelligence - something that is "grounded in everyday capacities" [14, p. 347]. Creativity can be found in even the most mundane situations; [1] compares the task of efficiently packing shopping bags into a car boot, with that of an engineer trying to fit components onto a circuit board. While research grapples with difficult questions such as determining the source of creativity, we only need to form a simple model of it for this project.

A creative idea can be defined as "one which is novel, surprising and valuable" [14, p. 347]. The value depends on the creative context – perhaps the idea is interesting, beautiful or simply relevant to the greater work. The definition suggests that creativity is dependent on 'breaking the mould' and moving away from existing concepts to some new solution – something that computers are inherently bad at doing. Nevertheless, we can form models of creativity which can be applied in computing. [14] builds on earlier work in [15] to describe three methods through which computers can be creative:

- **Combinational:** The system forms novel combinations of familiar ideas.
- **Exploratory:** The system generates novel ideas by exploring a structured conceptual space.
- **Transformational:** Similar to exploratory, but the system can also transform itself to form a new conceptual space where previously impossible ideas can be formed.

There is a clear gradient of complexity in these models. For most engineering problems, it is easy to see the value in a transformational system that can adapt itself and generate fundamentally different ideas depending on the situation. In a music performance context, however, there is a lesser need for reorganization: music theory and long-established performance techniques provide a strong framework within which the system can be constrained. Here, we are

not aiming to create a system that finds revolutionary new DJ techniques, but one that simply emulates human tendencies.

In the case of DJing, the AI must make decisions on which tracks to mix and how to mix them. For example, where to start and end a transition, which filters or effects to apply, as well as the parameters of these effects, such as depth or delay time. These decisions can be equated to the 'familiar ideas' from the combination model.

With this model, novelty can be reached if the AI has enough decisions to make. The greater the variety of ideas that can be combined, the more unique each output can be. Given that the combinational model is the simplest to implement, and serves as a suitable analogue to human creativity, it was used in this project.

Making Mistakes

It is often said that imperfections are what makes art feel 'human'. Defects in a work of art can make us feel closer to the artist, as they are usually symptoms of their personal technique or the situation in which the art was created [33]. Imperfections can also help ground things in reality - as the Zen Buddhist, Shunryū Suzuki, said: "Nothing we see or hear is perfect. But right there in the imperfection is perfect reality" [98].

If an AI performance system is to feel human, perhaps mistakes should be embraced. The definition of a mistake is highly context-dependent, but the ability to make them can be built into our creative model. A system that harnesses the combinational model of creativity could exhibit mistakes if it had enough freedom in the ideas it can draw from. Some element of randomness could also help mistakes to be made, resulting in a more organic and human output.

Rather than integrating deliberate mistakes into an AI system, an 'imperfect' implementation or training process can also give similar results. Figure 2.2 shows two paintings from a Paris-based collective who created a series of portraits using machine learning. The neural network they used was not trained as rigorously as a typical image generation network, which resulted in very organic, almost incomplete-looking outputs.

Figure 2.2: Portraits of the fictional Belamy family, created by a neural network [19].

Another artist has developed a system which tries to generate images that are different to anything it sees in its training data. This has an equally creative, but much more abstract result compared to the portraits above.

Figure 2.3: Abstract images created by a neural network that tries to output something different to its training data [19].

2.1.4 Related Work

Much existing research into creative music systems is focused on music composition. One such system is Cypher, which is described as "a real-time interactive music system" [85, p. 43], which composes novel musical output based on the user's input. The real-time nature of this work makes it particularly relevant, as real-time composition can be thought of as a type of music performance.

Cypher has two components: a listener and a player. The listener analyses a stream of input MIDI data, tracking certain features over time, including dynamics, harmony and rhythm. The analysis is then sent to the player, which produces a new melody in response. The system provides several ways for the player to interpret the listener's analysis, based on different internal algorithms, which can be selected by the user.

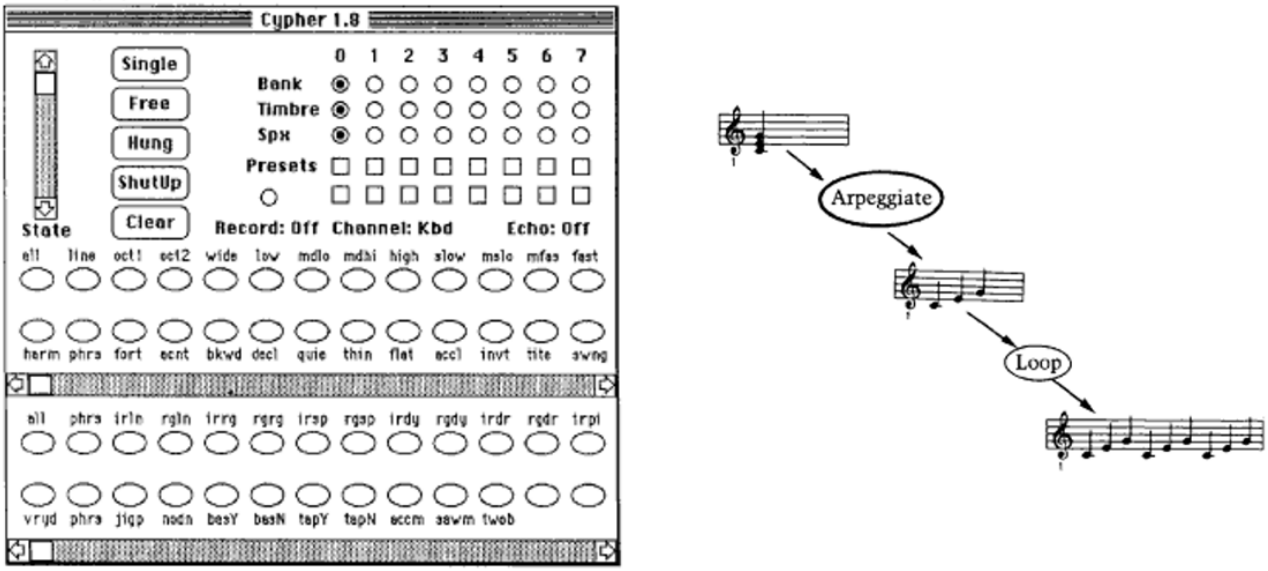


Figure 2.4: The interface of Cypher (left) and a basic example of how a MIDI input chord might be transformed into a melody (right) [85].

One of the algorithms pieces together a vocabulary of musical elements based on the listener's messages, with an element of randomness involved. This is closely related to the combinational model of creativity described in the previous section, in that existing ideas are combined in novel ways to create a larger piece.

Another creative AI system is described in [22]. Here, dynamic programming is employed to generate a musical accompaniment to an input melody. The use of dynamic programming is interesting: it can be compared to the transformational model of creativity, because dynamic programming allows algorithms to self-adjust, producing outputs they previously could not. The benefits of using AI accompaniment over a human are discussed, noting that "computers can follow extremely detailed and precise directions without error" [22, p. 189]. While the accuracy of a computer certainly presents opportunities for novel performance methods, following 'precise directions without error' might interfere with the emulation of a natural human performance, because there is no room for the mistakes described in the previous section.

On the subject of organic human performance, the work in [68] attempts to reproduce the expressiveness of a real performance, using an AI system. They define expressiveness as the dynamic 'touch' of a human performer, which is a product of years of practice where the performer observes and imitates other, more skilled players. They use this idea of observation and imitation to form a novel approach that transforms a 'flat' input melody into a more natural, expressive output.

The system uses spectral modelling and a note detection model to analyse recordings of human players, building knowledge of what expressive transformations might be applied to certain note patterns. After this training process, a case-based reasoning (CBR) system is used to generate appropriate transformations to apply to unseen input data. CBR operates by looking at solutions to similar cases from the past, and adapting them to fit new ones.

This work raised the question: could an observation-imitation model with CBR be used in the present work? While it might function well to produce 'appropriate' DJ mixing decisions, the use of CBR could hinder creativity, by repeatedly reusing ideas for certain situations. Furthermore, it would be difficult to source human DJ performance data to observe in the first place. Instead, we continued with a set of explicitly programmed mixing ideas, which can be combined in a creative manner. Note that an external input is required to drive the systems mentioned here, whereas the present work moves towards a more autonomous solution, where no external input is required.

2.2 Music Information Retrieval

To make appropriate DJ decisions, our system needs to build an understanding of the music it has available. Music Information Retrieval (MIR) is a technique where raw audio data is analysed to extract knowledge about the characteristics of the music. These characteristics may be constant throughout a track, or they might change with time and/or frequency.

There are several types of data that can be extracted using MIR techniques. These include low-level *parameters*, such as energy level and fundamental frequency, which can be measured directly from the audio; as well as higher-level *descriptors*, such as tempo and key signature, which require more complex modelling techniques.

Rhythmic synchronization, or *beat-matching*, of concurrent tracks is an essential part of the DJ workflow. To align the period and phase of the beats in each track, information about their tempo and beat positions must be extracted using temporal analysis techniques. From this process, we get a series of beat positions called a *beat grid*.

2.2.1 Temporal Analysis

The measurement of musical pulses and subsequent estimation of tempo is known as *beat tracking*. The first beat tracking system that could operate on complex polyphonic audio was presented in 1994 [39]. Prior solutions typically operated on recordings of single instruments, or MIDI data. A year later, the work was extended to a real-time implementation – also the first of its kind [40]. See [86] for a comprehensive review of temporal MIR research up to 1998.

By 2001, there was growing demand for high-performance temporal analysis algorithms, as more and more commercial use cases were emerging. But the systems available tended "to be fairly unsophisticated, as they seem[ed] to rely mostly on the presence of a strong and regular kick drum at every beat" [64]. Around this time, the rise in popularity of electronic music also marked a shift in some MIR techniques. The heavily time-quantized nature of electronic music contrasts with traditional acoustic music, in that the tempo is constant throughout the track. This allows for simplifications to be made to MIR algorithms that specifically target electronic music.

As the research field matured, common frameworks for temporal analysis began to emerge. A fundamental component in temporal analysis is the concept of note onsets; these are the time points at which notes start playing. The process of finding significant note onsets within a track forms the basis of most modern approaches, as the onsets hold much of the important rhythmic information. The onsets can be represented as a discrete list of time points [39], [27], [29], [28]; or they can be represented in a more continuous manner, using an onset strength signal (OSS), which is sampled at discrete intervals [47], [24], [53], [81].

A typical OSS pipeline is shown in Figure 2.5. The input audio is considered in 2048-sample batches called *windows*. Using various methods, the content of each window is analysed to produce a single sample of the OSS - notice that the sample rate of the OSS, F_{sO}, is therefore:

$$F_{sO} = \frac{F_s}{hopsize}$$

Once note onsets are extracted, there are numerous techniques for finding rhythmic periodicities within them. A popular method is the multi-agent approach [39], [40], [53], [52]. Here, multiple tempo hypotheses – or agents – are considered, and a measurement of their confidence is scored across the whole track. The agent with the highest final score is taken as the overall tempo estimate.

A highly efficient method for evaluating different tempo candidates is presented in [81]. A *pulse train* is constructed for each tempo candidate, where ideal impulses are distributed in time with the beats of a given tempo. Each pulse train is cross-correlated with the OSS to produce a similarity score, as shown in Figure 2.6. This is efficient because the correlation only needs to happen at the impulse points - the rest is 0. The last plot represents the score for one pulse train, correlated at different phases (time offsets) with the OSS. The peak magnitude and variance of each similarity plot is used to evaluate which tempo candidate should be taken

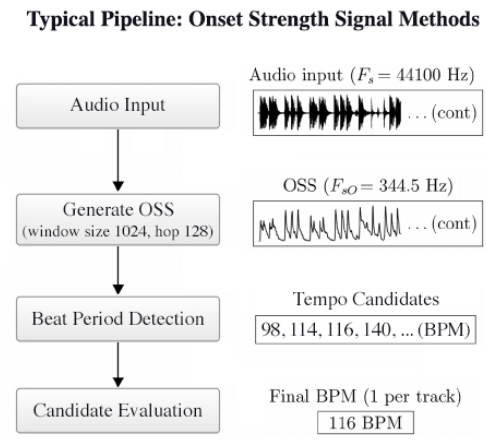


Figure 2.5: A typical Onset Signal Strength (OSS) pipeline for extracting note onsets from raw audio [81] (modified).

Autocorrelation is another popular method for finding periodicities within an OSS [81], [97], [38], [43]. Instead of measuring the similarity between separate signals, autocorrelation measures the *self-similarity* of one signal. It compares the OSS with a time-shifted version of itself; time shift values that give a high score correspond to rhythmic periodicities within the signal. The time shift that gives the highest score is considered equal to the beat period of the music. The tempo, T, can be found from the beat period, P, as follows:

$$T_{bpm} = \frac{60 * F_s}{P}$$

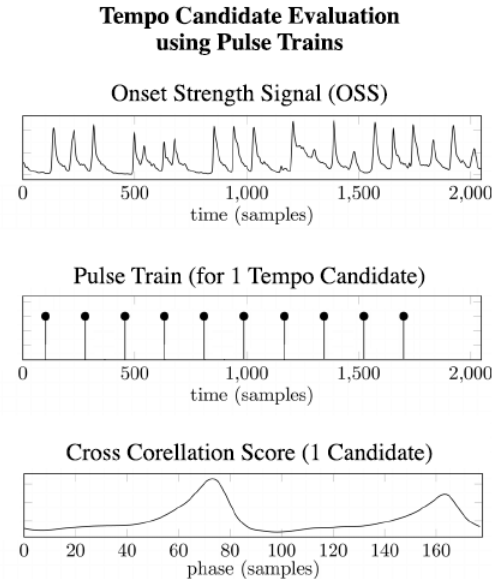


Figure 2.6: Correlating an ideal series of impulses with the OSS, over a full rotation of phase [81] (modified).

2.2.2 Evaluation Methods

In order to choose the best algorithms for this project, it was important to have a reliable framework for comparing different techniques. **Accuracy** and **computation time** are both critical metrics which helped to determine which algorithms to use.

One of the first reliable evaluations of a large set of MIR algorithms was conducted at the 2004 International Conference on Music Information Retrieval (ISMIR) [3]. This established a strong methodology which was used in many subsequent works. The data sets it used were frequently applied in subsequent research, which led to concerns that algorithms could be overfitting the common test data. [81] was the first piece of research to address this, by conducting a comprehensive review of temporal algorithms using an expanded group of data

sets.

The Music Information Retrieval Evaluation eXchange (MIREX) is a community-built framework for the evaluation of MIR systems. It provides annual insights into the progress of this research field and was used as reference during this project.

2.2.3 Automatic DJ Systems

Once temporal MIR techniques had matured enough to produce somewhat reliable results, researchers saw their potential for automating the workflow of a DJ. Cliff [20] was the first to propose an automatic DJ architecture in 2000, laying the foundations for how one might approach a simple beat-match and crossfade system. This paper describes which music characteristics would be required for successful automation, without prescribing the specific MIR algorithms to be used.

Fraser [36] implemented such a system in software, with basic algorithms for tempo matching, beat phase alignment, and crossfading. Interestingly, the user could draw the volume envelope to be used for each crossfade.

Andersen [9] created an open-source framework, *Mixxx*, for the development and evaluation of novel DJ interfaces. This is a software implementation of traditional DJing, with a modular design that allows different interfaces and add-ons to be tested. Their later study [8] used Mixxx to examine the benefits of automating the beat-matching process. They found that "on average, 47% of the time used by professional DJs playing live [is] used on a task related to beat matching" [8, p. 44]. Figure 2.7 shows two mappings of a DJ console used to control Mixxx: one where beat-matching is automated and the other representing a traditional setup. On average, DJs preferred the interface where beat matching was automated. As mentioned in section 1.1, automation can be considered an escape from 'real' DJing, so this favourable result is interesting. The test was conducted with only 10 professional DJs; it would be beneficial to extend a similar investigation to a wider, more diverse group. Similar features have since been integrated into commercial offerings, including industry-standard hardware by Pioneer.

Jehan [56] was the first to use advanced MIR techniques to generate more natural DJ transitions. As well as extracting tempo information, an algorithm was used to determine the downbeat positions in each track. This allows for better alignment of rhythms and melodies during transitions, as they tend to begin and/or repeat on the downbeats. A method is proposed for finding the ideal moments to transition between consecutive tracks, by measuring the rhythmic similarity between different sections - a technique that human DJs sometimes use.

Bouckenhove [16] continued to make use of high-level descriptors to improve the automation. A vocal detection algorithm is used to avoid playing different vocals simultaneously - one of the golden rules of DJing. A significant advance here is that energy and chord information are used to inform the playlisting of tracks, i.e., which tracks should be mixed together and in what order.

Researchers continued to integrate more advanced MIR techniques to improve the automation of the DJ workflow. Lin [46] and Hirai [92] aim to maximise the musical similarity of the track sections to be mixed, with the latter considering both rhythm and tonality. Ishizaki [45] aims to minimise the uneasiness that tempo modulation can cause, by using a perceptual

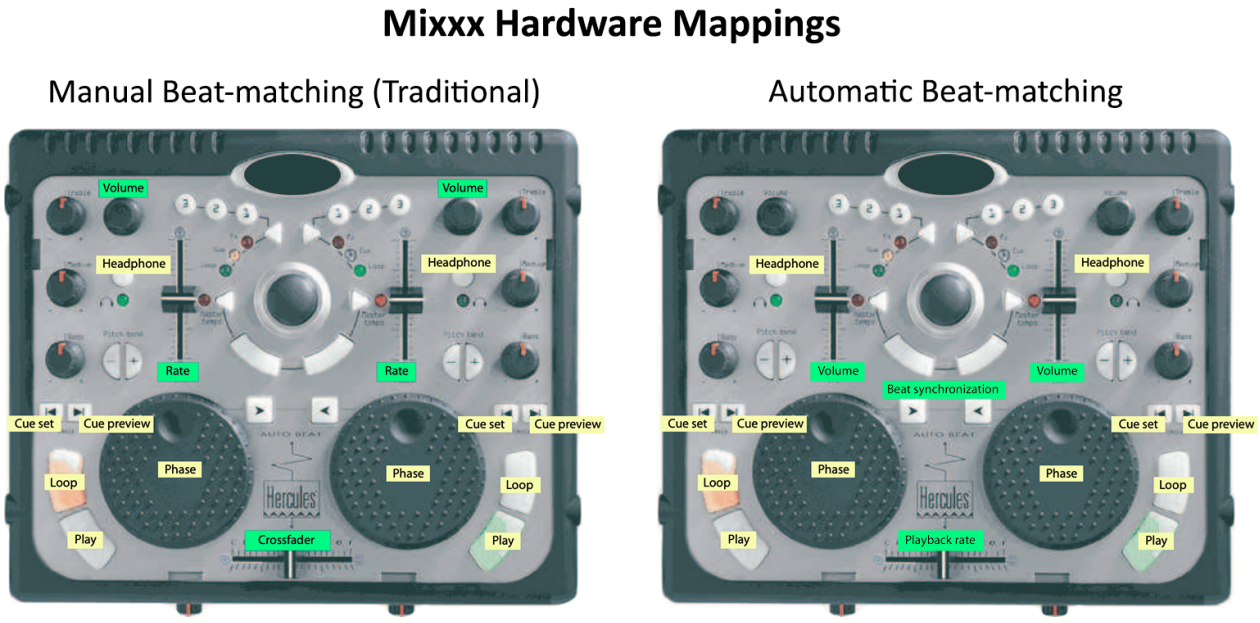


Figure 2.7: DJ hardware mappings with manual and automated beat-matching, tested by 10 professional DJs [8] (modified).

model of user discomfort. Wooller [104] describes how transition can be embellished with new musical elements, by gradually morphing the note sequences of the outgoing track to match the incoming track.

Clearly, extracting more musical information allows more advanced mixing techniques to be used, which helps us move towards a human-like DJ performance. There is a steadily increasing complexity in the proposed systems, which can be seen in the research timeline in Figure 2.8. An interactive version of this timeline can be found here, which provides short descriptions of each paper.

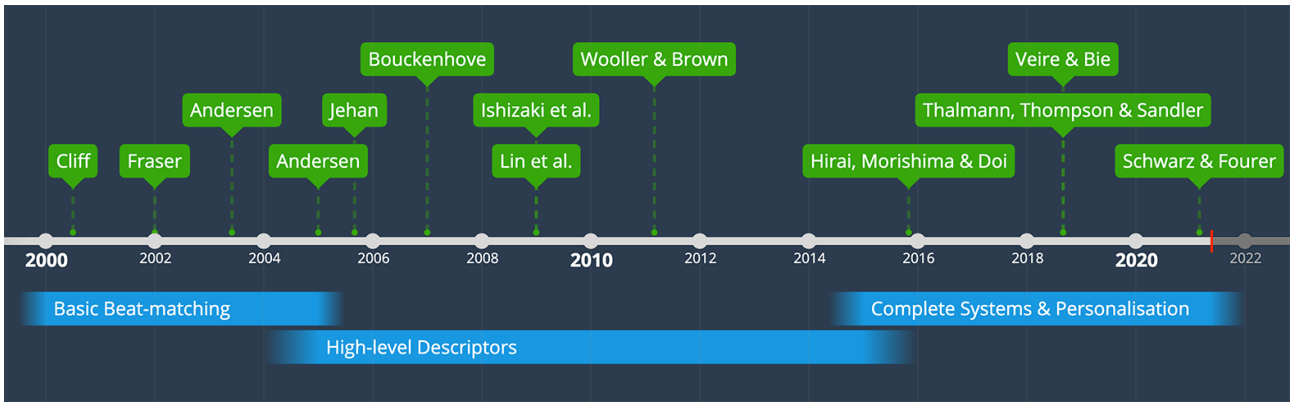


Figure 2.8: A timeline of MIR-based automatic DJ systems - interactive version.

Complete Systems

Recent research has focused on building complete systems, where a powerful set of MIR algorithms extract almost all the information that a human DJ would usually use to inform their performance. Note that, while an automated system like this is explicit in what information is extracted and used, the equivalent thought process by a human DJ is often at a *subconscious* level. For example, they might mix together two tracks because it 'feels' right, rather than explicitly considering the relative energy levels or rhythmic content of the music.

[101] describes a comprehensive, fully-automatic DJ system that makes use of a number of MIR techniques. They claim that by "leveraging the understanding of the music tracks offered by ... state-of-the-art MIR techniques, the proposed system surpasses existing automatic DJ systems both in accuracy and completeness." They also examine 'mash up' systems, which make use of similar algorithms to produce songs which are made up of other tracks - like a very short and dense DJ mix.

The beat tracking system in [101] scores an impressive accuracy of 98.2% - that is, the tempo and beat phase is estimated correctly for 98.2% of tracks. They acknowledge that the efficacy is, in part, due to the focus on a single genre, Drum & Bass, which allows them to make certain assumptions about the music. This exploitation of genre-specific characteristics is inspired by [51], which was one of the first papers to acknowledge the advantages of genre-specific approaches within MIR research. The system encourages musical continuity between consecutive tracks by determining their style. The analysis annotates each track with a style descriptor, which represents the mood or atmosphere of the track. The spectral contrast method for music genre classification [99] is used to generate these descriptors. Interestingly, the track selection engine must then make a trade-off between musical continuity and progression/change over time.

Another recent study looks at automated DJing from a new perspective. [32] was conducted by the Centre for Digital Music at Queen Mary University of London, a world-leading research centre for MIR. They propose a system with a minimal user-interface, that "adapts to the user's preference via simple interactive decisions and feedback on taste." The primary research goal is to collect data on the types of transitions favoured by listeners, and in what contexts they are best applied. An important point they raise is that previous research has focused on making transitions as smooth as possible, whereas real DJs use a more diverse range of transitions as "devices for adding musical interest or an element of surprise, in order to keep the crowd engaged" [32, p. 2].

A decision tree is used to determine the transition type for each mix, which is informed by high-level MIR descriptors extracted from the music library. A typical decision tree is shown in Figure 2.9, with a number of the possible transitions described. Only a preliminary experiment has been carried out with the system so far, which showed favourable results for the decision tree system, versus random transition choices. The results of a full investigation could be compared with a recently released data set of DJ mixes [88], to gain insights on whether the listeners' preferences align with techniques commonly found in human DJ mixes. It would be interesting to see whether they would support the notion that brisk or abrupt transitions are best suited to vocally-dense genres such as pop and hip-hop, as described in section 1.3.

There are a small number of existing commercial systems that automate DJ performance: Djay Pro AI and Pacemaker are the most popular examples. Both systems offer varying levels of user-involvement, including fully automatic modes. These are private projects with vague

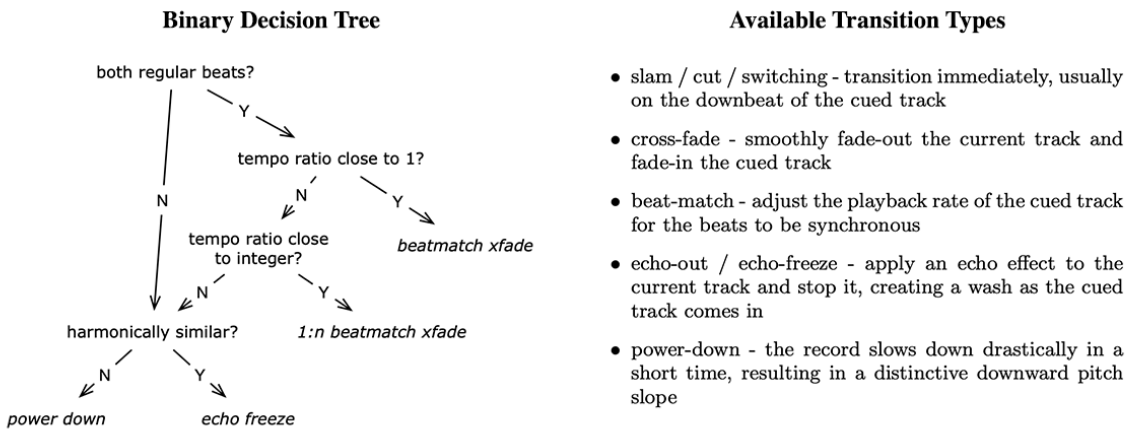


Figure 2.9: A binary decision tree to decide the style of transition between two tracks, and the available transitions [32].

technical descriptions, so it is unclear what audio analysis algorithms are used, but there is certainly temporal MIR involved. Through brief qualitative testing, it seems that both offerings perform very well at automatic beat-matching of various dance music genres. An interesting feature in Djay Pro is Neural Mix, which offers real-time source separation, meaning the audio can be separated into instruments, drums, vocals and so on. This uncovers a world of possibilities, because individual components of the tracks can be mixed separately, or in various combinations.

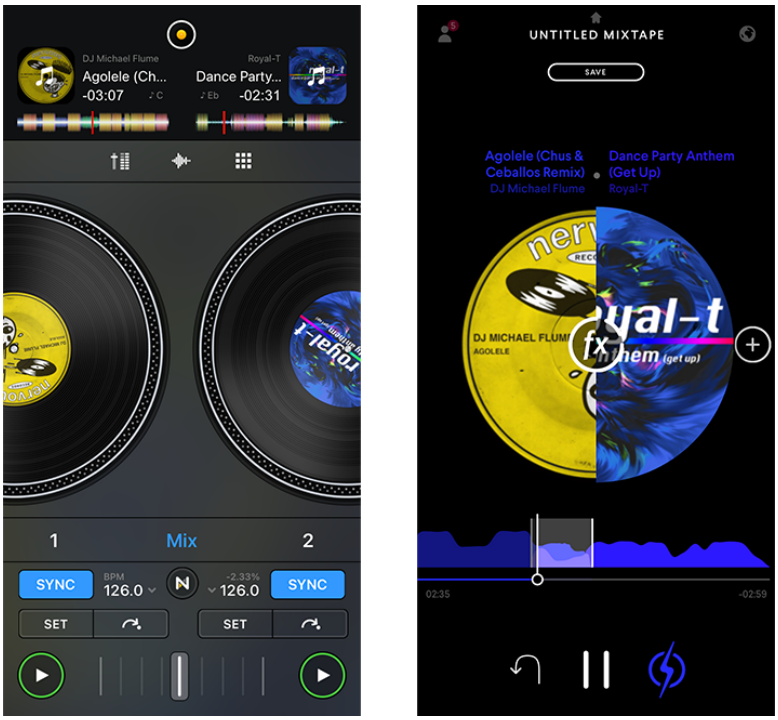


Figure 2.10: Screenshots of Djay Pro AI (left) and Pacemaker (right) mixing the same two tracks. They are both iOS apps capable of automatic DJ mixing.

2.3 Human Computer Interfaces

Many aspects of music, especially performance, can sometimes feel like an 'exclusive club'. There is often a high cost of entry, whether that be prerequisite knowledge and skills, or the actual monetary value of musical equipment. While many people are interested in music and its performance, these barriers to entry surely obstruct a great deal of them, perhaps to the point of never getting involved.

Software can be made accessible to the masses, both in its design and distribution, and this presents an opportunity to lower the cost of entry to the music performance club. While it is easy to cater exclusively to the experienced digital musician, the job of the developer is to take a step back and think from the perspective of a newcomer. Most users will not seek training or open a guide, so it is important that software is designed from the ground up to support *learning-by-doing*. The automation aspect of this project only reinforced the importance of accessible design, because we welcome users with no musical experience at all.

Through automation, we are essentially taking control away from the user. The user interface (UI) developed in this project is therefore simpler than that of a manual system. Typical music software is notoriously packed full of parameters, menus and visualisers. Nevertheless, these complex interfaces, as well as universal design practices, served as important learning resources for how to design an accessible UI.

2.3.1 Communicating with the Subconscious

Oxford definition of intuitive: "Using what one feels to be true even without conscious reasoning; instinctive."

At the core of any intuitive UI design is a communication with the subconscious. In an ideal world, the user could launch a piece of software for the first time and immediately know how to harness it to meet their specific needs. In reality, many never realise the full potential of their software; this can be due to a lack of understanding of certain features, or complete unawareness of said features. We can get closer to the ideal case by considering how the mind works, to minimise cognitive load on the user and guide them to an ideal workflow.

Figure 2.11: Proximity between shapes triggers groupings, even when the shapes are dissimilar as in (C). Even the proximity of the labels allows us to reference groups in this figure [57].

Visual structure is perhaps the most powerful tool available to the interface designer. The human brain is constantly searching for patterns and groupings in visual information, and this can be used to guide the user from the entry point to their desired destination. In Designing

with the Mind in Mind, Johnson shows how simple proximity between shapes, even those that are dissimilar, triggers grouping processes in the brain [57]. These groupings can be used to indicate relationships between interface elements such as buttons, icons or tabs.

Spatial placement can also communicate *hierarchy* within a block of data. In Figure 2.12, the unstructured flight details are an unorganised slurry of information, which the viewer must linearly scan through to find what they want. The structured layout brings hierarchy to the data, allowing the viewer to take shortcuts to what they want. The benefit of this simple improvement only multiplies when there is a long list of flights.

Figure 2.12: Unstructured and structured text showing flight details [57].

The concept of hierarchy extends much further than text. A tool bar, for example, is considered a *top*-level element that sits above the content it controls; conversely, document icons represent *low*-level objects in a file system - the leaves of a folder tree. Spatial placement helps to communicate these hierarchies to the user, even if they are unaware of it.

In most cases, the user is here to perform a task, not in a leisurely manner, but as fast as possible. They are on a mission to achieve some result, and we must make that journey as easy as possible. In Don't Make Me Think, Krug points out the reality that only a fraction of the information presented by an interface is actually absorbed [61]. Appendix A illustrates how a user might parse a page to find what they want; text is not read, it is scanned, simply because "we know we don't *need* to read everything."

Contrary to popular belief, reading is not a natural human ability like speaking, so it is best to avoid text whenever possible. The neural structures that support spoken language have evolved over hundreds of thousands - if not millions - of years [57]. Written language, on the other hand, did not exist until much later, and most of the world population was illiterate until a few centuries ago [83]. Furthermore, reliance on written language can exclude the blind, and more commonly, users of different languages. Alternatives to text include shapes/icons, images and sounds.

In the multitude of situations where writing is a necessity, *plain language* should be used. This refers to language that is understood by a very wide audience. The Plain Writing Act of 2010 [41] was passed by the US government to encourage clearer communication through all state channels. They have since published comprehensive guidelines which aim to help users "find what they need, understand what they find, and use what they find to meet their needs." The simple guidelines, which are based on academic research, can easily be applied in interface design. For example, on choosing vocabulary, the guide quotes [60]: "Prefer the familiar word to the far-fetched. Prefer the concrete word to the abstraction. Prefer the short word to the long." While it might be tempting to insert clever or playful language, it is best avoided in a UI.

Design conventions should be used to our advantage when communicating with the subcon-

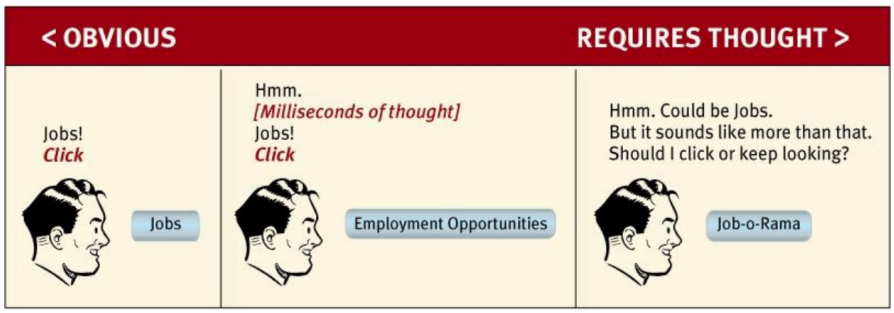


Figure 2.13: Plain language makes a UI understandable for a wider audience [61].

scious. These conventions include how things look, where they are, and how they work. Take a road stop sign for example: they all look the same, always placed on the same side of the road, and they all mean *Stop!* This level of consistency takes cognitive load away from the driver, no thought is required as to what the sign means [61]. To relate this to software, consider the classic layout of audio playback controls in Figure 2.14. If the layout or appearance of the icons is changed, things become unnecessarily confusing. To avoid confusion, conventions must be embraced when appropriate.

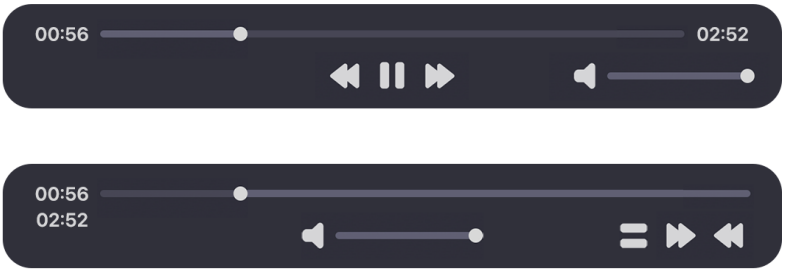


Figure 2.14: An audio toolbar which embraces convention (top) and rejects it (bottom) - which one feels intuitive?

2.3.2 The Paradox of Control

Complex software often includes many parameters that can be controlled by the user. There is a point where the addition of more controls results in the user being less *in control*. This issue is especially prevalent in music software, where the user is presented with a sea of buttons, knobs and sliders. There are a number of ways to alleviate confusion, firstly by making it clear exactly *what* can be controlled.

Perceived Affordance

Affordance is defined as the relationship between the properties of an object and the capabilities of a potential user [80]. This determines if and how a user can interact with an object, whether physical or virtual. Perceived affordance is therefore if and how a user *thinks* they can interact with something. The context, appearance and perhaps labelling of an object all influence its perceived affordance. It is important to consider these things so that the user is clear on what they can control.

In The Life of a Button, Apple's prototype designer Julian Missig discusses the careful thought that goes into the subtleties of the simplest UI control. A toast-making app is used as an example, where the user simply presses a button to remotely start their toaster. Missig notes that, in many cases like this, buttons are indirect controllers of action, which makes it even more important to consider perceived affordance.

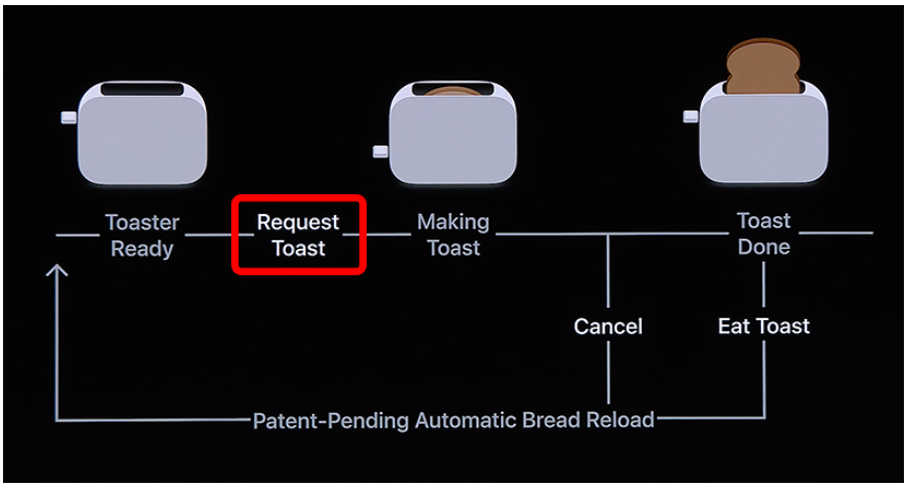


Figure 2.15: The pipeline of Missig's toast-making app, with the most important interaction highlighted [12] (modified).

While there are several processes happening in the app, shown in Figure 2.15, Missig says we should consider the most important interaction first, the *Request Toast* stage. If we get the details of this interaction right, "the rest will flow from it." He adds: "If people aren't able to start the toaster, the rest of it doesn't really matter." While this methodology is primarily aimed at smartphone app developers, it can be extended to more complex desktop software, by considering a *number* of important interactions - perhaps one per screen/view.

Figure 2.16: The design iterations for the *Toast Request* button [12] (modified).

Figure 2.16 shows the design iterations for the toast request button. Each iteration experiments with how we can convey the affordance of the button, until the final iteration where there is no doubt that it makes toast. Note that, while iteration #2 is more concise than #4, it is both a noun and a verb, so the action is unclear. Iteration #3 attempts to use imagery - a great alternative to language, as described in the previous section - but the icon is unfamiliar and unclear, so text is preferred in this case.

Divide the problem

In his YouTube series on music software design, Tantacrul challenges the widespread design ideal that "nothing should be more than three clicks away" [94]. If the problem is divided into sections, we can reduce cognitive load on the user, guiding them through a number of simple steps to reach their goal. These steps should be considered carefully, so that memory

Figure 2.17: Perceived affordance is how a user *thinks* they can interact with things [61].

load is also minimised - the user should not have to remember what they typed three screens previously [87].

Preventing user error is another important way we can streamline the user experience [87], and dividing the problem helps with this. If we know exactly where the user is and what they are doing, we can apply certain restrictions which prevent them from making errors. Such errors might include invalid input values or misuse of UI controls.

2.3.3 Music Software Interfaces

Most existing research into music software interfaces is focused on Digital Audio Workstations (DAWs). The DAW is the most widely used software tool for creating contemporary music; it is a complex editor, which operates as both a high-level instrumentation and arrangement tool, and a low level manipulator of audio. Their design is primarily informed by their historical origins in multitrack tape recorders and mixing desks [95], and many of the visual elements reference these physical systems. Marrington describes it as a "repository of virtual tools that refer, metaphorically or otherwise, to both the pre- and post-digital hardware traditions of music technology" [71, p. 8]. While DAWs are primarily used for music *creation*, some can also be used as music performance tools - Ableton Live being a popular example.

Many UIs, such as word processors, deal with direct data representations, where paradigms like WYSIWYG offer an almost tangible experience. Music software, however, has the more difficult task of abstracting music and audio into visual representations [67]. Consider the Ableton piano roll shown in Figure 2.18, which contains a number of abstractions. Time is represented by distance along the horizontal axis; pitch is distance along the vertical; and note intensity is represented by brightness. As a whole, it is equivalent to a traditional musical score, which in turn is an abstract representation of music.

Music can be thought of as the organisation of sound events in time. Some argue it is also time *in* sound, given that we can hear time patterns from rhythm itself [69]. Time representation is therefore an important part of music software. Users must be able to manipulate short phrases or individual notes, all the way up to entire song sections. Duignan writes "the explicit representation of blocks of material at multiple temporal levels is critical to reducing complexity, particularly since it is required to enable the quick rearrangement and editing of material at various levels" [67, p. 26]. He found that representations above 'song' level are either problematic or non-existent in DAWs, which leads to problems if people want to use them for live performance. "In an ideal world, producers could move between entire musical

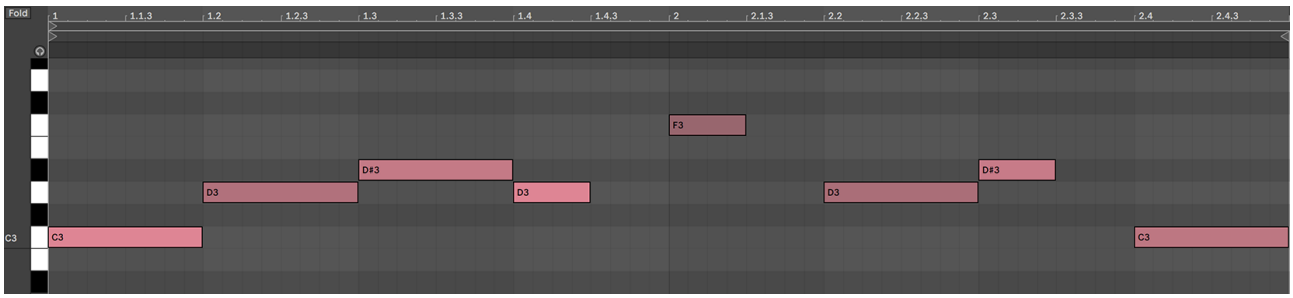


Figure 2.18: The piano roll / MIDI editor in Ableton. The black and white blocks on the vertical axis represent piano keys.

pieces on an ad hoc basis and rearrange their set list on the fly without any interruption to the performance" [67, p. 26]. This is not an issue in dedicated DJ software, but it indicates that DAWs have untapped potential for powerful performance workflows.

Another common metaphor in music software is the use of colour to communicate timbre. Timbre is defined as the "character or quality of a musical sound or voice as distinct from its pitch and intensity" [26] - for example, the difference in sound between a trumpet and a piano playing the same note. Helmholtz, a pioneering researcher in sound perception, referred to it as *klangfarbe* or 'tone colour' [62], so a mapping between timbre and visual colour feels very natural.

Timbre is determined by the frequency content of a sound, and how it changes over time. We can therefore make a directly proportional mapping between light frequency and sound frequency, where the lowest bass is red and the highest treble is blue. Both of the most popular DJ applications, Rekordbox and Serato, use this idea as part of their track waveforms, as shown in Figure 2.19. These help the user find different sections within a track, as well as see what frequency content is up ahead, so they can make mixing decisions in advance. Serato offers a muted colour palette compared to the saturated Rekordbox waveform, which can make for more comfortable viewing after long periods.

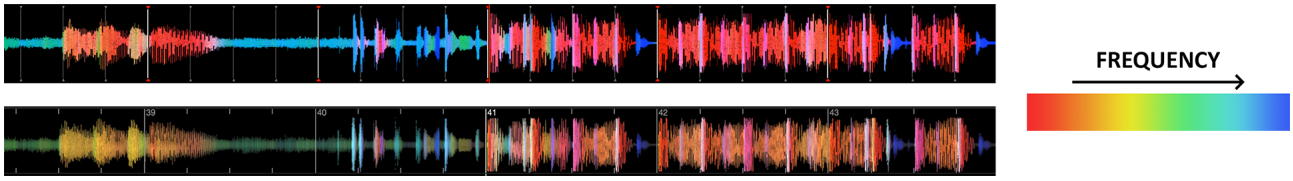


Figure 2.19: Audio waveforms in Rekordbox (top) and Serato (bottom), where frequency content is mapped to colour as shown.

Since the horizontal and vertical axes are also used to delineate time and amplitude respectively, these waveforms are a perfect example of using multiple visual dimensions to represent audio. While many users might not consider what each dimension corresponds to, they can easily learn to interpret certain elements without much thought. For example, hearing quiet sections whenever the waveform is thin, and heavy bass whenever the waveform is red, will quickly build connections in the brain. The user can then harness this knowledge for future mixing decisions, whether consciously or unconsciously.

Serato makes use of another colour mapping for key signature. This system is called the Camelot wheel, developed by Mixed in Key [59]. It is based on the circle of fifths, where keys are distributed around a wheel, but the placements have been shuffled so that 'compatible' keys

are adjacent. The more notes that two keys share, the more compatible they are. The idea is that tracks with compatible keys will work together harmoniously during a DJ transition. While this system is slightly less intuitive than the coloured waveforms - perhaps requiring the user to research how it is used - it certainly presents high value when harnessed correctly.

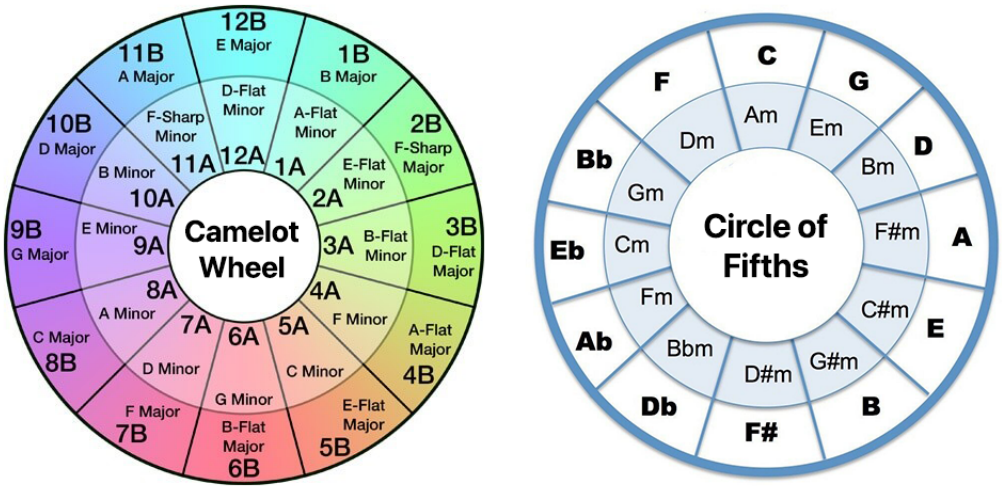


Figure 2.20: The Camelot wheel (left) and the circle of fifths (right), which both distribute key signatures around a circle [59].

Tools as a Participant

> *"People who make music on computers don't realise how powerful the visual element is. Whether you like it or not, your mind starts to think in terms of patterns, because it's a natural human way to do things, and you start seeing the way drums are lining up on the screen, and it becomes completely instinctive to line them up in a certain way."* - Four Tet [49]

The affordances mentioned in the previous section extend beyond individual UI elements, all the way up to high-level software functionality. Mooney proposes a framework for analysing music creation tools, both digital and analogue, from the perspective of affordance [74]. This allows us to focus on how the design of music software affects the creative process.

A key insight is that no design is a completely "transparent, neutral mediator of artistic expression" - all of them have "a spectrum of affordance whereby certain objectives are easier to achieve than others" [74, p. 145]. For this reason, the software can be thought of as a participant in the creative process. It has been shown experimentally that software design influences the creative approach taken by musicians, and therefore the musical outcome [70]. Furthermore, some systems offer auto-generation and randomisation features which produce novel ideas ex nihilo. In these cases, the software becomes a partial collaborator with the musician [48].

Many electronic music producers begin by playfully experimenting with their software, in an open-ended period of parameter-fiddling [48]. This can lead to new discoveries or unexpected twists in existing ideas, which often result in unique musical outcomes. Such workflows can even harness quirks of software which might typically be seen as bugs or limitations. This is supported by the much-quoted idea that 'limitations breed creativity.' Artist Phil Hansen

summarised this well: "If you treat the problems as possibilities, life will start to dance with you in the most amazing ways" [72].

This does not mean that limitations should deliberately be built into software, because they often hinder the creative process. Tantacrul examines the problems with the infamously complex *Sibelius* in his YouTube series on music software interfaces. This tongue-in-cheek critique is certainly not unbiased, but the issues he mentions can serve as a good learning experience. Even at the first step - adding an instrument - Sibelius makes it difficult for the user to achieve their goal. The following figures give examples of what Tantacrul found problematic. The subsequent section of the video details the addition of a simple tempo marking, which incurs further difficulty. Some suggestions are made for how the system could be improved; these relate primarily to clearer naming and categorisation, as well as simplified layouts and workflows.

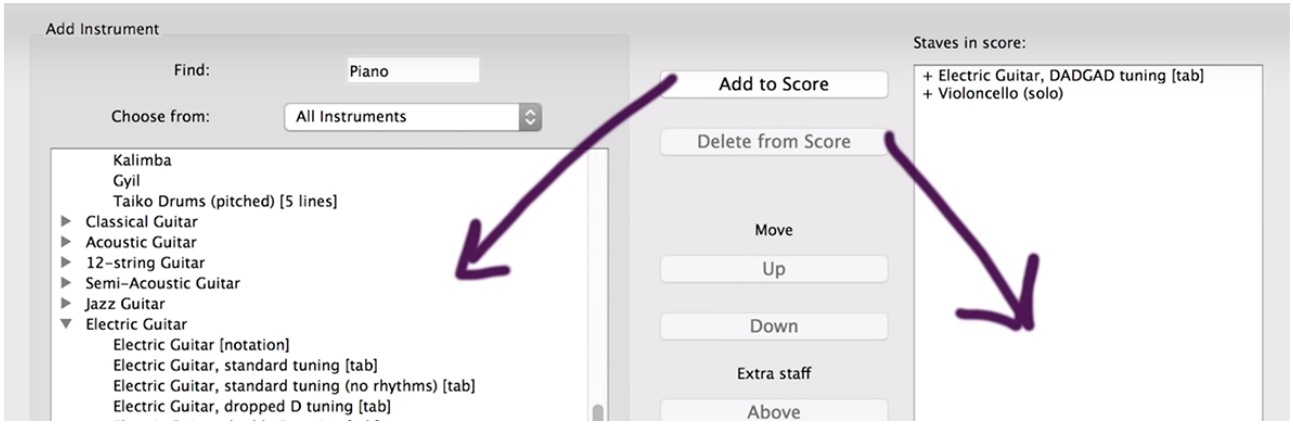


Figure 2.21: In the Add Instruments window of Sibelius, buttons that are close together refer to different parts of the screen. [93].

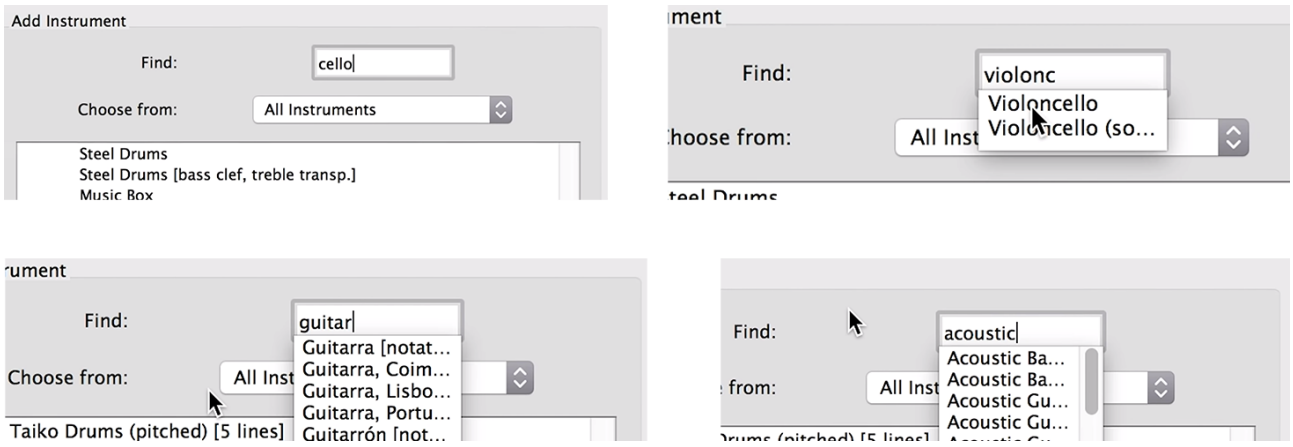


Figure 2.22: Searching for a 'cello' does not return any results, so one must type the rarely-used full name 'violincello'. Searching for an acoustic guitar returns a plethora of confusing results that are cut short, despite the text box having ample room [93].

3 Specification

After considering the project aims and literature review, the following software specifications were decided:

- Model a human DJ using artificial intelligence
  - Automatically create a continuous DJ mix, in real time
  - Use the combinational model for creativity to deliver an entertaining mix
  - Integrate Gaussian randomness to encourage novel decision making
  - Allow the user to override the choice of the next track to be played
- Use MIR algorithms to extract information from a library of music
  - Temporal analysis for beat matching tracks
  - Tonal analysis to extract emotion/mood information for the track selection process
  - Segmentation analysis to detect distinct musical sections
  - Exploit genre-specific features to increase reliability
  - Provide a method for manually correcting the temporal data
  - Wait-free system that analyses tracks in the background while playing
- Provide a GUI which is accessible to users with no musical training
  - Make use of HCI best practices to ensure the UI is intuitive
  - Include basic audio controls (start, stop, volume)
  - A skip button to fast-forward the mix
  - Provide coloured waveforms to communicate timbre and audio amplitude
  - High level view of mix trajectory, with controllable direction

To further focus the research aims, a set of questions were also set out:

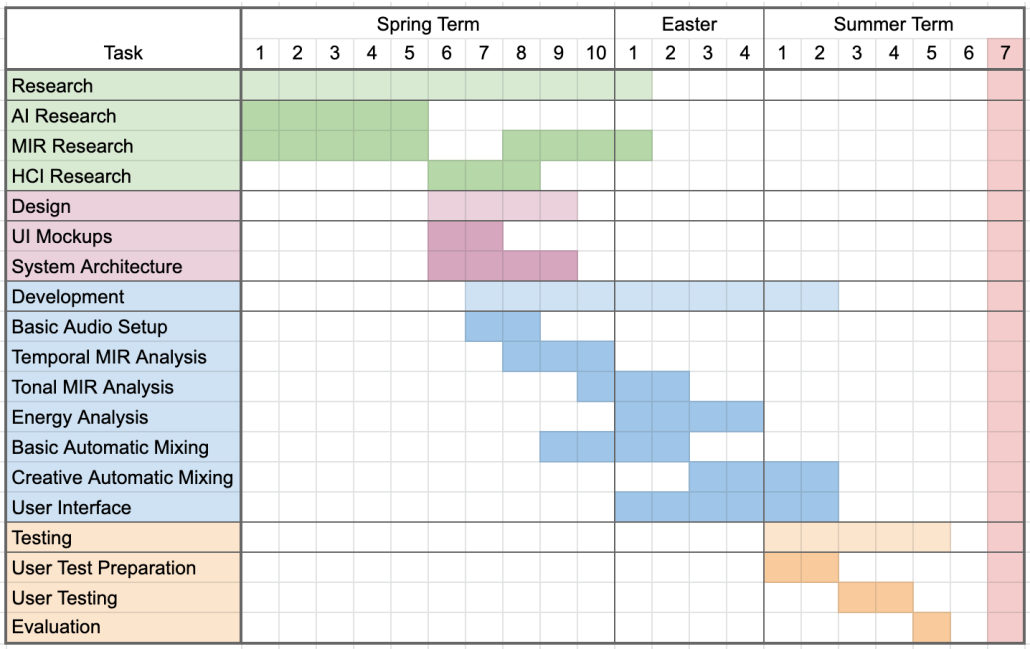
- Is there a use case for automatic DJ software, and how much involvement do users want?
- Can an AI DJ be entertaining?
- Can expensive MIR algorithms be processed reliably on a range of machines and material?
- Are mistakes and errors a major issue in an AI music performance system?
- How can we help non-musical users get involved in music performance?

4 Approach

An iterative development approach was used for the project. It is based on the Agile methodology, with relaxed rules surrounding collaboration and product delivery, given the academic nature of the project. The core principles are as follows:

- Working software is the primary measure of success
  - Focus on getting results, with as little overhead as possible
  - Results are defined by the project specification
  - Tackle the project from various angles so that all areas see progress
- Be highly adaptable
  - Embrace changes in requirements/direction, rather than adhering to a strict plan
  - Use weekly meetings with supervisor to discuss progress and re-evaluate priorities
- Complete work in batches/sprints
  - Break tasks up into smaller, more manageable problems
  - Test each solution as soon as it is implemented

With the flexible design methodology in mind, a Gantt chart was used to plan out the areas of work. This was a 'live' time plan, meaning it was reviewed at various stages throughout the project and adapted according to progress, challenges and changes in priorities.



Git was used for version control, meaning that the codebase was always backed up with full history available. A feature branch approach was used in certain cases. The team-related benefits of feature branching were irrelevant in this solo project; instead, it allowed for safe experimentation on features that might not make it into the final product.

The GitHub repository for the project can be found here. Since open-source code libraries have been used, the source code for this project must be open source, too.

The project has been developed and tested on Mac, but plans are in place to build and test it for Windows. This should be relatively easy given that a cross-platform framework was used - see section 4.2.1.

4.1 Software Tools

A variety of software was used during the project for designing, researching, coding and testing. The applications were carefully chosen based on the tools and workflows suited to each task.

XMind A mind-mapping tool used for keeping track of research and design ideas. Supports the insertion of images, hyperlinks, comments and colour-coded flags and into mind map. See appendix B for a screenshot of the map of HCI research.

Adobe XD Used for designing/prototyping the user interface. Very good for quickly testing out designs, with the ability to make objects clickable to link different views together.

Adobe Illustrator & Photoshop A powerful pair of vector and image manipulation environments, used for designing user interface icons and logos.

XCode The primary IDE used for programming. Offers a powerful set of debugging tools, as well as convenience features when navigating and typing code.

Projucer A JUCE application for managing source files, project settings and resources. The XCode project is created by this application - see section 4.2.1.

Visual Studio Code A lightweight IDE by Microsoft, used for its powerful search functionality. After choosing a directory, you can search for a term within *any* type of file.

Terminal Command-line tool used for git version control, as well as building static libraries. Static libraries allow dependencies to be packaged *within* the application binary, meaning the user does not need to have the libraries installed on their machine.

Audacity A utilitarian audio editor used for examining input music and the output of the app. Able to find the sample number of specific points in audio, which is useful while debugging audio processing code.

DB Browser A simple application for viewing and editing SQL database files.

Diagrams.net (formally Draw.io) A web interface for creating block diagrams and flow charts. Used to map out the system architecture during the design phase.

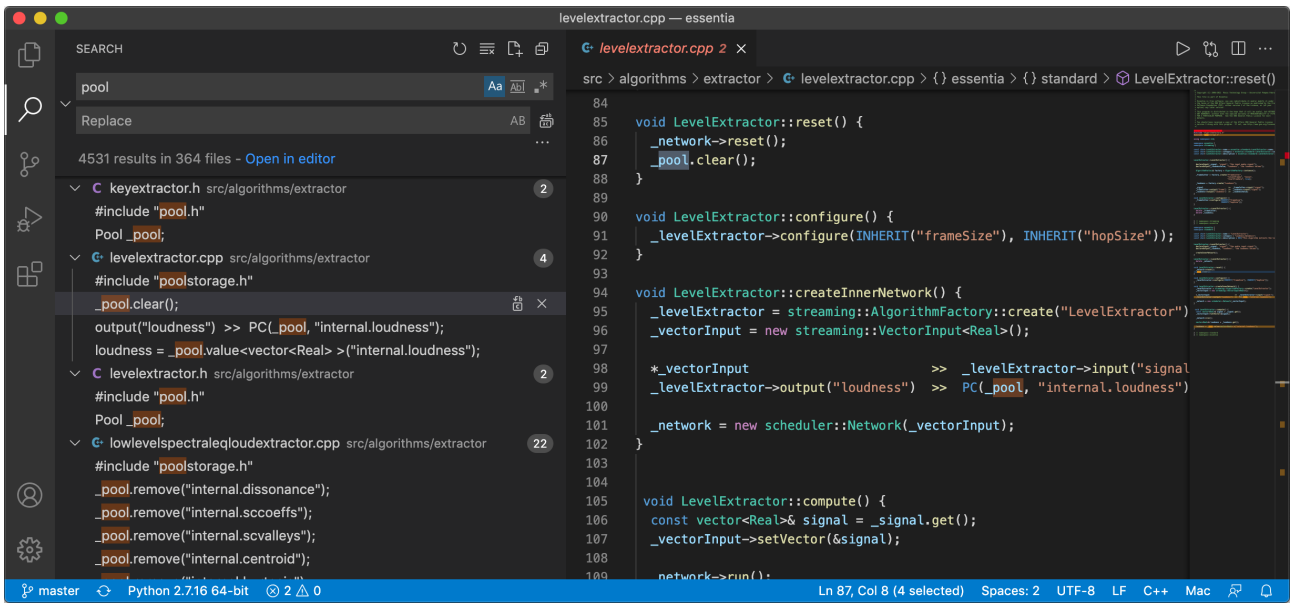


Figure 4.1: The powerful search function in Visual Studio Code can parse almost any file type.

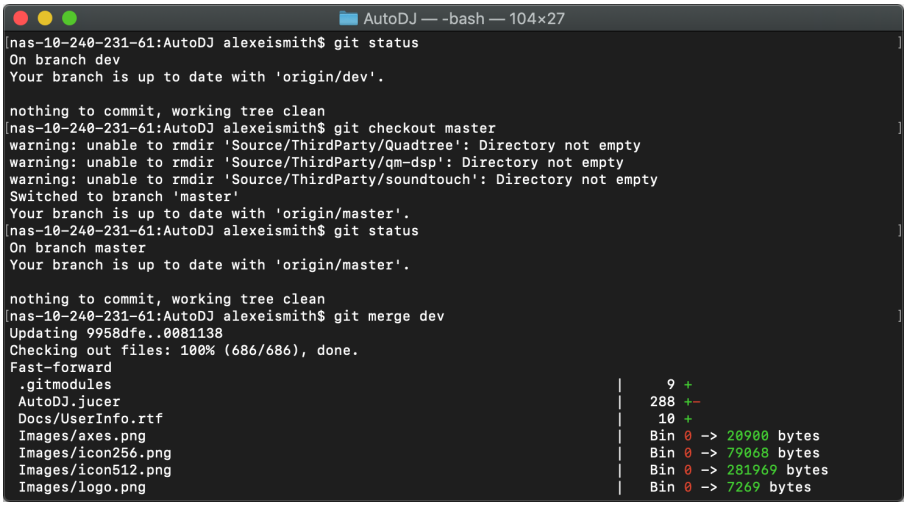


Figure 4.2: The Mac Terminal natively supports git version control.

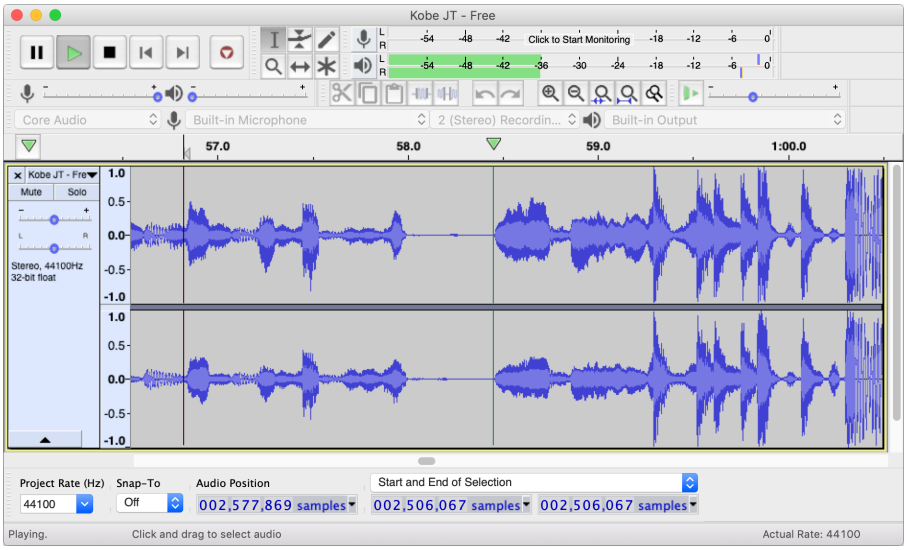


Figure 4.3: Audacity can view and edit raw audio down to the sample level.

4.2 C++ Audio Programming

C++ was chosen as the programming language for this project, primarily for its performance capabilities. Unlike high-level languages such as Python and Java, the core C++ libraries are very lightweight and come without frills such as automatic memory management. While this means more work for the developer, it allows them to prioritise efficient code over unnecessary features. Performance optimisation is essential for this project because it involves complex analysis algorithms and real-time audio processing. The manual memory management is also well-suited for low-level audio manipulation, because individual samples can be addressed and processed in an explicit manner.

The lightweight nature of C++ means that a lot of boilerplate coding work is required to get started. To avoid this overhead, it is useful to use a framework which handles the basic application setup, as well as a variety of extra functions.

4.2.1 The JUCE Framework

JUCE is a partially open-source, cross-platform framework for C++ applications. It allows developers to target Mac, Windows, Linux, iOS and Android from a single codebase, providing utility functions that operate the same across all these platforms. An array of optional modules can be added, which provide powerful tools for working with graphics, audio, DSP and data processing. It is known primarily for its use in VST audio plugin development.

There is a dedicated community surrounding JUCE, which supports bug fixes and the development of new features. They run an annual conference for audio developers, aptly named the Audio Developer Conference. Comprehensive documentation and a number of tutorials are provided to assist newcomers, as well as seasoned developers. The JUCE Forum can be used to seek help on personal projects, or to raise wider issues.

The Projucer

JUCE comes with an application called the Projucer, shown in Figure 4.4, which manages the state of an entire JUCE project. It brings together all the source code files, resources and build settings into one environment. It also comes with a number of example projects which are ready to build and run without any changes.

The Projucer automatically manages the packaging of resources - such as images, audio and databases - so that they are included in the application build. While this does increase the file size of the app, it ensures independence from any external files, meaning the user can run the application without an install process.

The actual programming happens in a traditional IDE of the developer's choice. The IDE project file is *generated* by the Projucer, which supports a number of major options including XCode, Visual Studio, Code::Blocks and Android Studio. In this sense, a JUCE project can be made truly platform-independent, as anyone can open a Projucer file and export it to their IDE on Mac, Windows or Linux. The IDE files can therefore be excluded from version control,

because only the Projucer file is required (as well as the supporting material, such as source code and images). The folder structure of a typical JUCE project is shown in Figure 4.5.

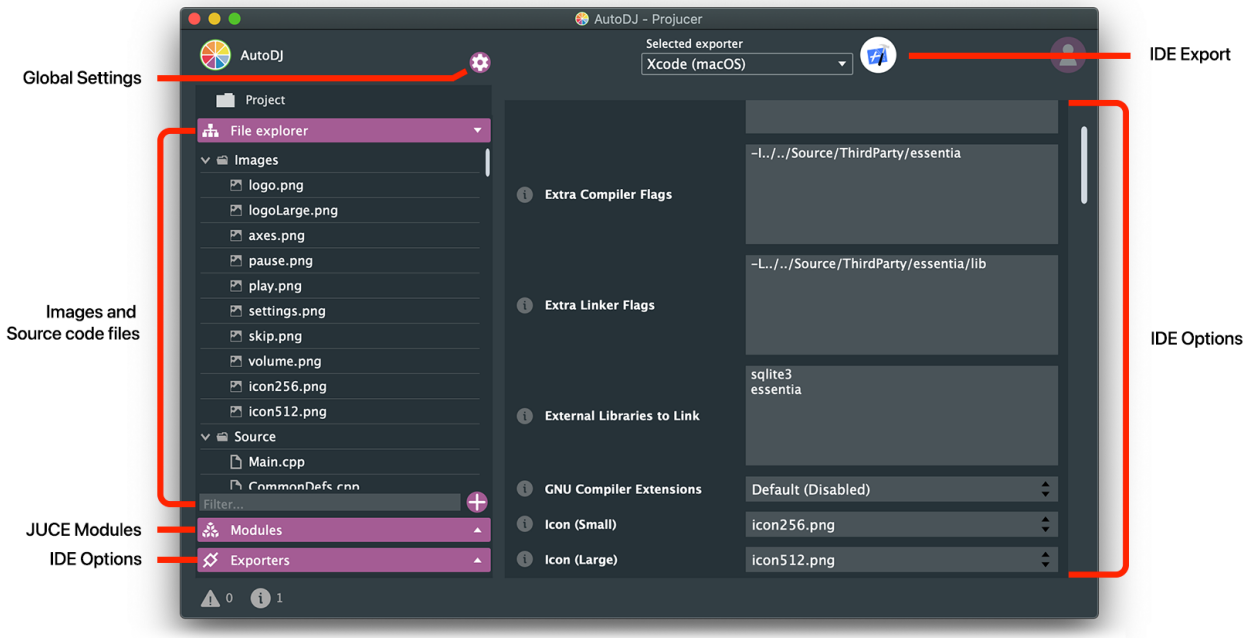


Figure 4.4: The Projucer manages all the resources and settings of a JUCE project.

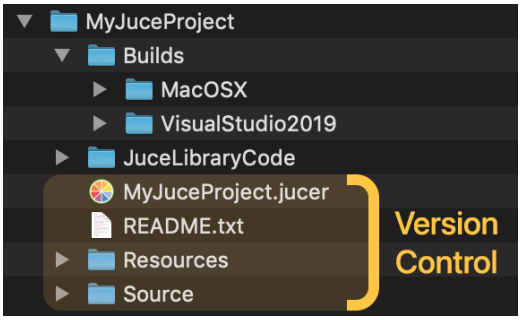


Figure 4.5: The typical folder structure of a JUCE project. Note that the build folder, which is excluded from version control, contains the IDE project files and any builds of the app being developed.

4.2.2 Troubleshooting

Troubleshooting is a central activity of software development, so it is important to have a well-defined approach to problem solving. The following resources were used in this project:

XCode Debugging

The XCode debugging tools provide a variety of methods for understanding what is happening while an application is running. Breakpoints were regularly used to pause execution and

examine the values of variables, as well as the call stack (the hierarchy of functions called by each thread). Conditional breakpoints were sometimes used to pause execution when a variable reached a given value.

JUCE Resources

The JUCE documentation provided essential information on the classes used in this project, including details of their functions and member variables. The tutorials and example projects served as inspiration for how to approach a number of common tasks relating to audio and graphics rendering. The forum was used to search for answers to any remaining issues that were encountered.

C++ Documentation

CppReference provides detailed C++ documentation, complete with examples. This was a useful resource for finding appropriate functions and tackling general C++ issues.

Programming Forums

Stack Overflow and similar programming forums were indispensable when tackling bugs related to C++ code.

Other Projects

Existing open-source projects were used as reference when integrating Music Information Retrieval algorithms. Their GitHub repositories were cloned (downloaded) locally, so the relevant sections of code could be found using the search function in Visual Studio Code. See section 5.3 for details on which algorithms/libraries were used.

5 Design & Implementation

The design and implementation of the software, named *AutoDJ*, will be detailed in this section. Despite the relatively simple appearance of the user interface, the underlying system is quite complex. In addition to the JUCE modules and third-party algorithms in use, 43 C++ classes were written, totalling over 8,000 lines of code. It is impractical to examine the entire codebase in this report; instead, we will focus on the most important features, as well as the high-level design decisions.

5.1 System Architecture

A simplified top-level class diagram is shown in Figure 5.1. In a JUCE application, the first class to be instantiated is MainComponent, which owns all of the other objects. AutoDJ is based on a simplified version of the Model View Controller (MVC) architecture, where the model and views interact directly - essentially, the controller layer is part of each view. This was found to simplify the overall design, while maintaining enough modularity.

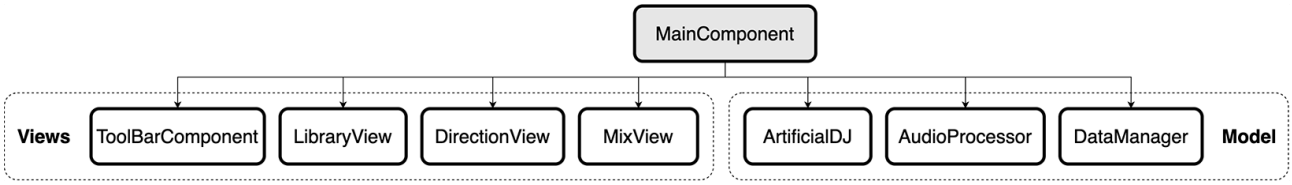


Figure 5.1: Diagram to show the hierarchy of the most important classes in AutoDJ.

The model is split into three areas of functionality: DJ mixing decisions, audio processing and data management. Figure 5.2 gives a closer look at the classes that manage these activities. Please see Appendix C for a more complete set of class diagrams, which show class inheritance, member variables and functions. Note that those are still simplified compared to the code implementation, with the aim of better communicating the system design.

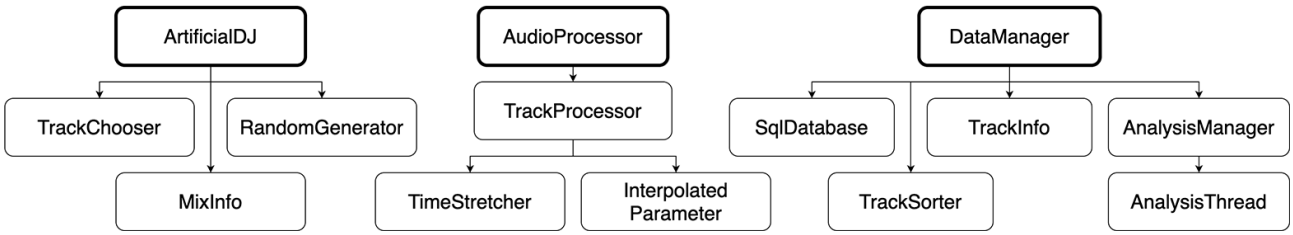


Figure 5.2: Diagram to show the hierarchy of the model classes in AutoDJ. Please see the appendix for full class diagrams.

ArtificialDJ handles the choosing of tracks to play, as well the deciding when and how they should be mixed together. It has a random generator which encourages novelty in the decision

making, using Gaussian-weighting to lean it towards 'sensible' decisions, where appropriate. See section 5.4 for more information.

AudioProcessor delegates most of the audio signal processing to two instances of TrackProcessor, which apply time-stretching, filtering and gain attenuation - the parameters of which are determined by the decisions from ArtificialDJ.

Both ArtificialDJ and AudioProcessor interact frequently with DataManager, which handles the logistics of data storage throughout AutoDJ including track information and audio. It also owns the MIR analysis pipeline, since this primarily deals with track data.

5.1.1 Data Management

DataManager owns all the track information used by ArtificialDJ and AudioProcessor. It has a central array to store the TrackInfo objects relating to every track shown in the library. Elements in this array are updated when a track is added, analysed or played by the system.

The TrackInfo array is only operational during runtime; for persistent storage, DataManager uses an SQLite database. This is also updated whenever a track is added or analysed. Note that the database filename starts with a '.', which hides it from view by default on a Mac. This helps prevent the average user from being confused by its presence, or even interfering with the data inside.

While a filename is assured to be unique within a given directory, we cannot assume it always points to the same data. An audio file could have been modified or replaced since AutoDJ was last launched. If we don't check the data inside, AutoDJ could associate old analysis results with unseen audio data, which would result in poor DJ mixing. To prevent this situation, the system stores a hash of every audio file, which represents the data in a single value. The hash value of existing track data can be compared to the data found when a directory is selected at launch. If the hashes differ, the audio file will be marked for re-analysis.

The algorithm in use is xxHash32, which compresses the data into a 32-bit representation. This does not guarantee that every possible file will generate a unique hash, but it is highly unlikely that the program would encounter two different audio files that produce the same value. More importantly, this algorithm is highly efficient, which allows it to be processed on hundreds of audio files per second.

5.1.2 Multi-threading & Concurrency

AutoDJ makes extensive use of multi-threading, which allows the same program to execute multiple tasks simultaneously - this is called *concurrency*. The relevant benefits of concurrency are as follows:

- **Uninterrupted UI:** Expensive tasks executed on the UI thread will cause it to lock up and become unresponsive while they run. Moving these tasks to background threads will ensure a smooth user experience.

- **Audio Priority:** A typical audio processing loop must execute every 10-20ms, otherwise the user will hear unpleasant artefacts in the output. It is standard practice to use a dedicated audio thread, which executes only the necessary, performance-optimised audio code.
- **Delegation:** Batch processes, such as the audio analysis in AutoDJ, can be split up between multiple threads, so that the overall task is executed more quickly.

Several threads are present in AutoDJ, for everything from UI interaction to the generation of DJ mix ideas. Figure 5.3 shows a summary of these, along with the primary tasks they perform.

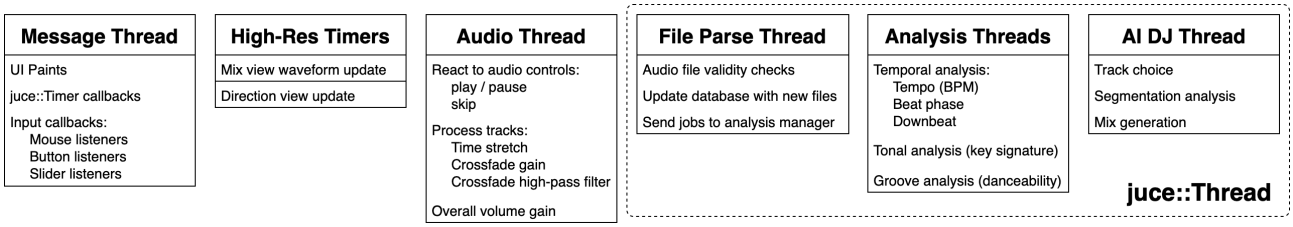


Figure 5.3: The threads used in AutoDJ, with their main processes listed.

JUCE was originally based on the Java concurrency model [90], which revolves around a dispatch loop. The dispatch loop, called the UI or 'message' thread in JUCE, handles core app functions such as graphics rendering, input handling and other utility messages. JUCE provides the Timer class which, when inherited, will trigger a callback function within the inheriting class at regular intervals. As shown in Figure 5.3, the timers are triggered by the message thread. To keep the graphics and input pipelines running efficiently, the message thread should not be clogged with too much extra functionality to process. Hence, most other tasks are handled by the surrounding threads. For regular updates where expensive operations are carried out, JUCE's HighResolutionTimer can be used, which runs on its own thread.

To define a custom thread, a class can inherit from juce::Thread. This requires that the class has a function called `run()`, which will be invoked when the thread is started. As shown in Figure 5.3, the file parsing, audio analysis and AI DJ threads all make use of juce::Thread. Below is the code for the Artificial DJ thread, which simply runs in a loop, generating new mixes when there is space in the mix queue. It is important for a thread to check whether it should keep running, so that other threads can terminate it if needed. Most commonly, this is the case when the user has pressed exit - the main thread must then stop all other threads running before it can close.

In most cases, the use of multi-threading for batch processes is only effective if there are multiple processing units to use. These processing units could be separate CPUs, CPU cores, or even the virtual cores found in hyper-threaded architectures. If there are more threads than processing units, the threads are just competing for processing time, rather than working simultaneously. For this reason, AnalysisManager, checks how many units are available, before deciding the number of analysis threads to launch. JUCE provides a simple interface for fetching this statistic, as shown:

```
int numThreads = juce::SystemStats::getNumCpus();
```

Concurrency comes with some major complications which must be considered in the system design. The first is that they increase the likelihood of race conditions. A race condition is

a situation where the output of an algorithm depends on its components being executed in a certain order, but the order is not constant. If threads are working concurrently towards the same goal, the variable speed at which they execute could determine the outcome. Race conditions can somewhat be avoided by modular code design, where threads do not work together towards a single outcome. If such a situation is unavoidable, for example in multi-threaded batch processes, there are methods for making a thread wait for its counterpart(s) before it continues operation.

The second, more common issue occurs when separate threads try to access the same data. To be clear: classes/objects and threads are not equivalent, and different threads can execute code within the same class simultaneously. Similarly, separate threads can attempt to access the same data simultaneously, whether that data belongs to a class, or is globally defined. If one thread writes data to a block of memory while another reads from it, *value tearing* can occur. This is where only some of the data bits have been written to, when the read operation occurs. The result is *undefined behaviour*, which means there may be an error, depending on the computer's architecture. Undefined behaviour is considered an error in itself and should be avoided at all costs. There are two solutions that can prevent value tearing...

Atomic Variables

Atomic variables are the only thread-safe variables in C++. They ensure that only one thread can read/write to the associated memory, preventing torn values. If a thread tries to access an atomic that is already engaged, it will wait momentarily until the atomic is free. There are a number of atomic types which correspond to the 'plain old data' (POD) types familiar to any C++ programmer, for example: `atomic<bool>`, `atomic<int>` and `atomic<double>`.

Atomics are used extensively throughout AutoDJ. The following code shows the declaration of an atomic boolean, `trackDataUpdate`, which DataManager uses to indicate whether there is updated track information to show in the UI. Typically, this is set by the analysis thread when a track analysis is completed, and then reset by the UI thread when the update has been received. Atomics are part of the Standard library in C++, hence they are declared in the 'std' namespace.

```
std::atomic<bool> trackDataUpdate = false;
```

Another example is `skipFlag`, which belongs to AudioProcessor. This flag is set by the UI thread when the skip button is pressed, indicating that the user wishes to skip to the next mix event. The audio thread checks the flag once every loop and initiates the skip operation if the flag is true, which also resets the flag ready for the next request.

```
void AudioProcessor::getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill)
{
    ...
    // If a skip has been requested, skip to the next mix event
    if (skipFlag.load())
        skipToNextEvent();
```

Mutex Locks

While atomics protect a single data variable when it is busy, a mutex lock can protect a set of objects for an arbitrary amount of time. A mutex can be used to lock a class and all its data while a thread executes a block of code, hence it typically blocks any competing threads for longer than an atomic would. Locks are particularly useful for preventing race conditions, or for keeping the state of an object protected while a critical operation is executed.

It is important to consider thread priority when using locks; the audio thread, for example, should not run into any locked code. If the audio thread shares resources with a lower-priority thread, such as the UI, it should be the UI that gets blocked, allowing the audio thread to access the resources whenever it needs to.

JUCE provides a convenient system for locking class objects while inside a given scope. A juce::ScopedLock can be instantiated within a function, which will protect the data of the associated class while in the scope of said function.

Here, a ScopedLock is used to protect the data of AnalysisManager while its function `getNextJob()` is called. This prevents a race condition, where multiple analysis threads could request a new job at the same time, possibly resulting in the same job being delegated to multiple threads, or even a job being missed entirely.

```
TrackInfo* AnalysisManager::getNextJob()
{
    // Initiate a scoped mutex lock to protect
    // class data while this function executes
    const juce::ScopedLock sl(lock);

    // Fetch the index of the next job
    int job = nextJob;

    // If there are no more jobs, return
    if (job >= jobs.size())
    {
        return nullptr;
    }
    else
    {
        // Otherwise, increment the job counter
        nextJob += 1;
        // Return the next job
        return jobs.getUnchecked(job);
    }
}
```

5.1.3 Third Party Libraries

AutoDJ makes use of a few third-party libraries to achieve a range of tasks, as follows:

- **Essentia:** Audio analysis library used for its MIR capabilities
- **Queen Mary's DSP:** Audio analysis library used for its MIR capabilities
- **Mixxx:** One class from this open-source DJ software is used to adjust the temporal MIR results
- **SoundTouch:** Time-stretching of audio, also capable of pitch shifting
- **Quadtree:** distributes tracks in a conceptual 2D plane of tempo and groove, and allows for highly efficient nearest-neighbour searching
- **xxHash32:** Efficient hashing algorithm for checking whether audio files are the same

These libraries have all been statically linked to AutoDJ, which means they are compiled *into* the executable application. The alternative is dynamic linking, where the user must have the library pre-installed on their own computer, before they can launch the application. While static linking does increase the file size of AutoDJ by around 5mb, it makes the software much easier to distribute, especially to non-technical users.

5.1.4 Pre-processor Directives

There are certain parameters that have a global effect on the operation of AutoDJ. These include audio settings, such as sample rate, as well as musical assumptions, such as the number of beats per bar. To avoid redundant and/or duplicate definitions, AutoDJ uses a single file to define these global parameters, named CommonDefs.hpp. This file also contains some common utility functions that don't belong to any specific objects.

```
#define SUPPORTED_SAMPLERATE (44100) // Sample rate to use (Hz)
#define TRACK_LENGTH_SECS_MIN (60) // 1 minute minimum length
#define TRACK_LENGTH_SECS_MAX (600) // 10 minutes maximum length
#define BEATS_PER_BAR (4) // Number of beats per bar (4/4 time assumed)
#define NUM_TRACKS_MIN (6) // Minimum track required to launch mix
```

These centralised values can be adjusted in one place, rather than in multiple, disparate files. Most of the parameters are defined using C++ pre-processor macros, which do not create symbolic links between CommonDefs.hpp and the other files. Instead, the parameter value is *inserted* at locations where the macros are used, during the build phase. This means the program architecture is not made any more complex when using the centralised parameters.

Another C++ pre-processor facility is used to choose between certain blocks of code. These conditional inclusions are useful for testing different algorithm configurations, because entire sections of code can be swapped depending on the value of a single pre-processor macro. Below is the function where the time-stretcher prepares its sample rate and audio channel configuration. If the macro `STRETCHER_MONO` has been defined somewhere, the stretcher will initialise only one channel. Here, `STRETCHER_MONO` is not defined, so the stretcher initialises two channels.

```
TimeStretcher::TimeStretcher()
{
    // Set the samplerate of SoundTouch using the
    // globally-supported sample rate
    shifter.setSampleRate(SUPPORTED_SAMPLERATE);

    // Set the number of channels depending on
    // the pre-processor mono/stereo macro
#ifdef STRETCHER_MONO
    shifter.setChannels(1);
#else
    shifter.setChannels(2);
#endif
```

Notice the use of `SUPPORTED_SAMPLERATE`, which is the globally-defined sample rate from CommonDefs.hpp. During compilation, the code will become:

```
TimeStretcher::TimeStretcher()
{
    // Set the samplerate of SoundTouch using the
    // globally-supported sample rate
    shifter.setSampleRate(44100);

    // Set the number of channels depending on
    // the pre-processor mono/stereo macro
    shifter.setChannels(2);
```

5.2 Audio Pipeline

As mentioned in the previous section, MainComponent is the top-level object in AutoDJ. Although it mainly deals with UI processes, JUCE architecture dictates it should also handle audio input and output. As such, it is the entry point for the audio thread, and is given a buffer to fill with the desired output audio.

The top-level pipeline of audio processing is shown in Figure 5.4. First, MainComponent checks that the audio output device is configured correctly. This is currently just a sample rate check, but there could be more requirements added in the future. The buffer is then handed to the top-level audio model - AudioProcessor.

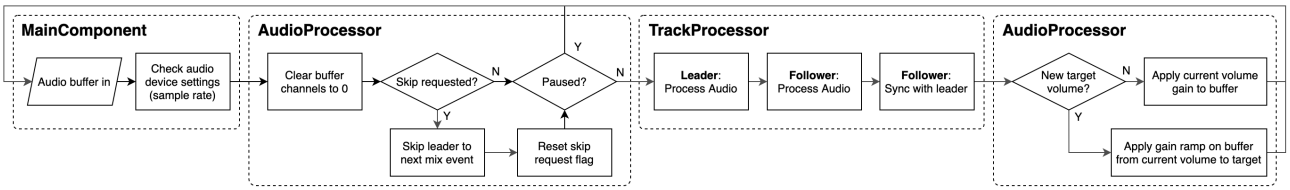


Figure 5.4: Flowchart to show the high-level audio pipeline in AutoDJ.

The buffer received by AudioProcessor is not guaranteed to be empty, in fact, it is likely to contain garbage data that has been left in the memory location of the buffer. It is therefore important to clear all channels to 0 before continuing, otherwise loud, distorted artefacts could be sent to the output.

AudioProcessor handles any master audio processing, and delegates the track-specific operations to its two child TrackProcessors. At any given time, one TrackProcessor is **leading** the mix, and the other is **following**. These are important states that affect which processor receives certain commands. For example, the next step is handling a skip request. If the skip flag is true, AudioProcessor tells the **leader** to skip to the next event in the mix. The follower will then be adjusted by the leader if necessary. See section 5.4 for more information on mix events.

The last operation by AudioProcessor is a volume ramp. If the user changes the volume level, the UI thread will notify AudioProcessor using an atomic 'target' value. If the target value differs from the current volume, AudioProcessor will ramp between the values over the length of the audio buffer. This is a form of interpolation, where the gain applied to each audio sample is calculated using its distance from the beginning and end values. Without the ramping, a large, instantaneous change in volume would introduce a high-frequency shift in the output waveform - i.e., an audible artefact.

Once again, efficiency is mind throughout the design of the audio pipeline. At the default sample rate of 44.1kHz and an audio buffer size of 512 samples, the pipeline will be processed around 86 times a second. Simple, explicit code should therefore be used to avoid any bottlenecks.

5.2.1 Track Processing

TrackProcessor is responsible for all the audio processing that happens to the individual tracks being mixed. It manages the fade in and out, using gain attenuation and a subtle high-pass filter, and delegates the time-stretching of audio to its child TimeStretcher object. This processing pipeline is shown in Figure 5.5.

After receiving the audio output buffer, TrackProcessor first checks that the track audio is ready. The time-stretched track audio is then fetched from TimeStretcher, which will be detailed shortly. The track BPM, gain and high-pass filtering are updated based on the number of samples being processed - i.e., the audio buffer size. Afterwards, TrackProcessor applies the gain attenuation and high-pass filtering, if active.

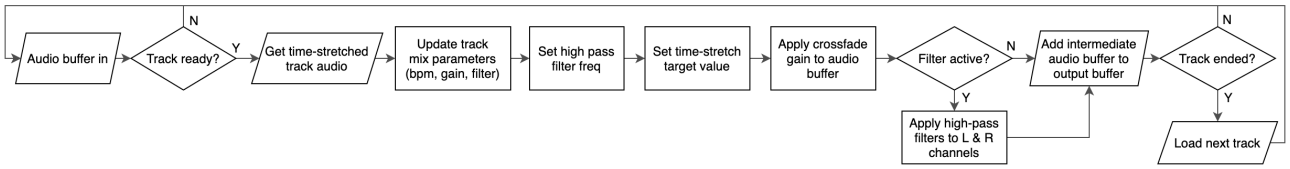


Figure 5.5: Flowchart to show the processing pipeline for TrackProcessor.

Note that TrackProcessor must use an intermediate audio buffer for all the processing up to this point. If it used the main output buffer, the audio from the leading TrackProcessor would be *replaced* when the following TrackProcessor did its processing. Once the crossfade effects are applied, TrackProcessor *adds* the audio from its intermediate buffer into the output, rather than replacing any values.

Once processing is complete, TrackProcessor checks whether the track it is playing has finished. If so, it triggers the loading of the next track in the mix. The load cannot happen on the audio thread, because it would halt operation for a few seconds. The loading of audio is instead handled by DataManager on a separate thread. Audio buffers are typically very large, containing 2 channels that are millions of samples in length. To minimise the amount of RAM during runtime, DataManager releases the audio once it is no longer required.

Ramped Parameters

During a crossfade, TrackProcessor must interpolate between the start and end values for gain and high-pass filtering. It must also gradually modulate the track bpm between transitions, if they are at different tempos. Since there are three parameters to be interpolated, and potentially a lot more in future versions, it made sense to create a dedicated class for this type of value: InterpolatedParameter.

```
void moveTo(double targetValue, int startSample, int numSamples);
void update(int currentSample, int numSamples);
void resetTo(double value);
```

The InterpolatedParameter class handles the modulation of values over time using the three functions shown in the code above. First, the parameter is set to a starting value, using `resetTo()`. The `moveTo()` function can then create a new ramp to a given target value. The position in the track where the ramp starts, as well as the ramp length, are both defined in terms of audio samples.

The class stores the arguments, and computes the rate of change that should be applied during the ramp. The rate of change is defined *per sample*, as follows: MATHS

In the `update()` function, the rate of change is added to the current value, for every sample moved. MATHS

TrackProcessor must be able to smoothly modulate the track parameters, sometimes over a period of a few minutes. For a long, shallow ramp, the rate of change could be extremely small, so the associated variables are stored as `doubles`, to maximise precision.

Time Stretching

Two tracks must be at the same BPM value to enable beat-matching. If AutoDJ were only able to mix together tracks that are naturally equal in BPM, it would run out of options very quickly. We must therefore be able to *change* a track's BPM, by using a time-stretching algorithm.

The simplest solution would be a resample operation, where a track's sample rate is changed by interpolating between the audio samples. This is great for preserving transient and rhythmic information, but it also changes the pitch of the whole track. Since musical tonality is a key part of the track choice engine, it was decided that the pitch of all tracks should remain constant. This means we require a time-stretching that doesn't modify pitch - this is known as time-scale modification (TSM).

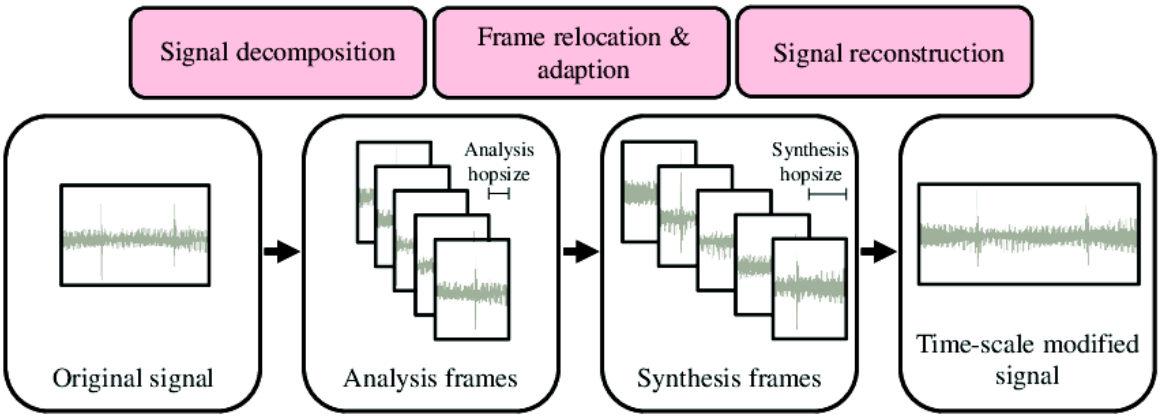


Figure 5.6: The high-level processes found in most audio time-stretching solutions [30].

Most TSM algorithms follow a similar pipeline of deconstruction, analysis and reconstruction. The input audio is divided into frames, and some form of temporal and/or tonal analysis is used to determine how to relocate and adapt these frames, forming a time-stretched version at the output. A comprehensive overview of TSM techniques is given in [30].

The most basic form of TSM is the Overlap-Add (OLA) technique. This involves taking frames of the input audio at regularly-spaced intervals, and blending them together at a different interval spacing. As shown in Figure 5.7, a windowing function is used to ensure smooth transitions between the frames, which prevents audible discontinuities in the output waveform.

The OLA method has been shown to produce good results for purely percussive audio [50], but it does not perform well when there are harmonic periodicities in the input signal. The splicing of audio frames without concern for signal periodicity results in a 'warbling' effect,

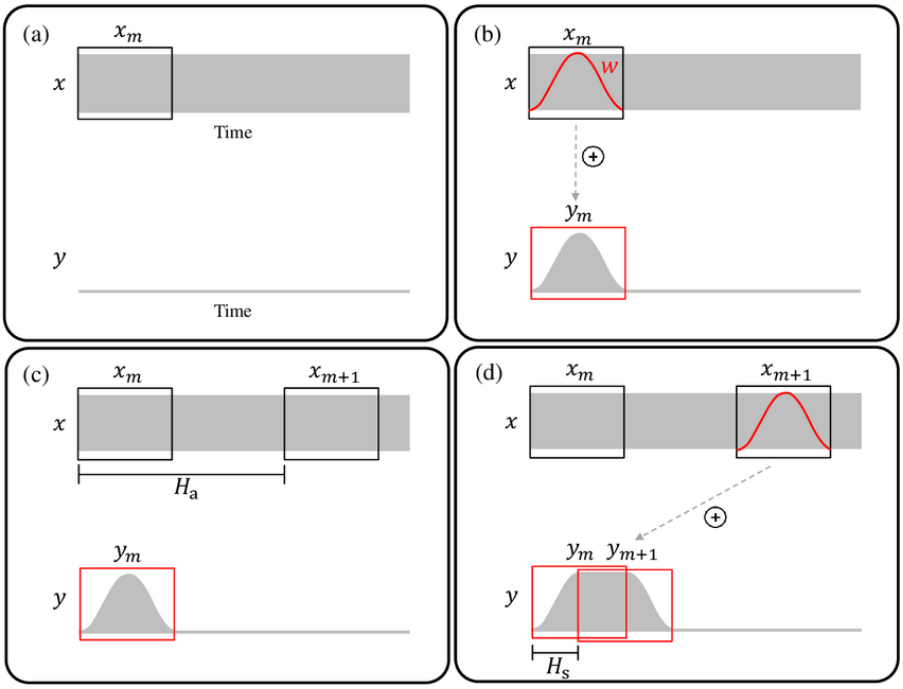


Figure 5.7: Processing steps of the OLA time-stretching algorithm, which splices frames of audio [30].

as the phase of the waveform jumps around at each frame boundary [30]. This effect can be considered a form of periodic frequency modulation [65].

For the complex musical material that AutoDJ operates on, we must preserve harmonic periodicities and minimise artefacts from time-stretching. There are several approaches that improve on OLA by allowing some tolerance in the frame sampling locations, to maximise the phase synchronization of the overlapped audio. These include Synchronised-OLA [84], Pitch-Synchronised OLA [75] and correlation-based approaches [63] [102].

Wave-Synchronised OLA (WSOLA) was used for this implementation, which uses cross correlation to find ideal framing positions. As shown in Figure 5.8, the sampling frame, x_{m+1}, is given a range in which it can move. Cross correlation is used to test the similarity of x_{m+1} with the previous frame, x_m, with at various positions. The position with the highest score is sampled, and the process repeats for the next frame.

Due to the focus on harmonic synchronisation, WSOLA does not preserve transient material as well as some other TSM methods. The algorithm was chosen despite this issue, because it is highly efficient, and existing implementations are available in a number of programming languages.

SoundTouch is a C++ library that offers time-stretching based on the WSOLA method, as well as independent pitch-shifting. The TimeStretcher class in AutoDJ acts as a wrapper for this library, managing its parameters and processing calls.

SoundTouch requires an interleaved audio buffer for stereo operation. This means audio samples for the left and right channels are packed into a single buffer in sequential pairs, rather than existing in separate channels. TimeStretcher must therefore interleave the audio it sends to SoundTouch, and also de-interleave the stretched audio it receives back.

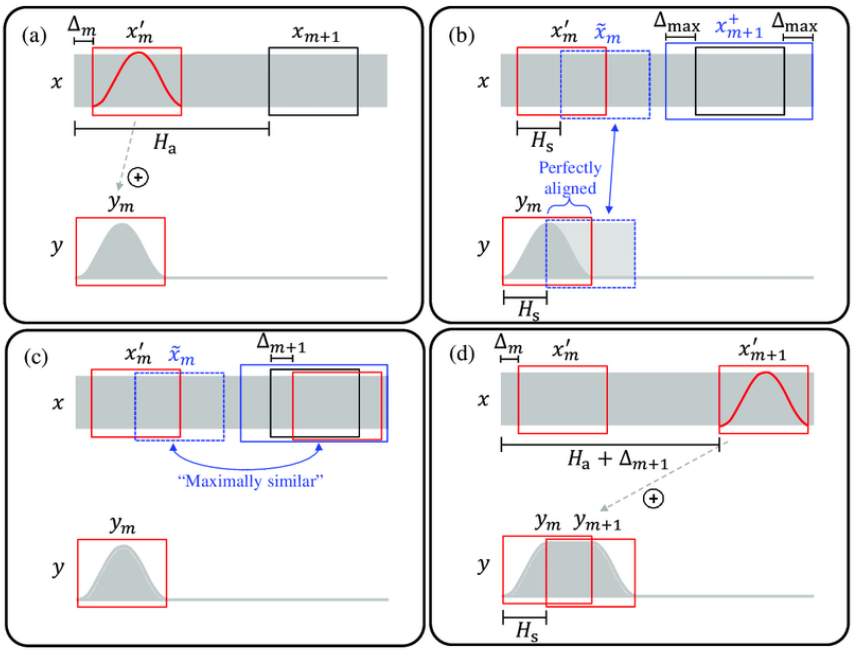


Figure 5.8: Processing steps of WS-OLA time-stretching algorithm, which aims for similar alignment of frames [30].

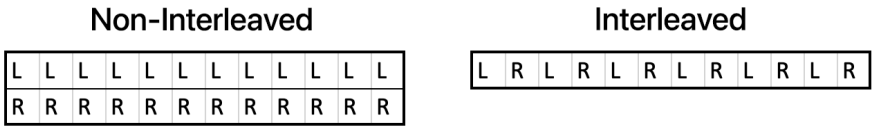


Figure 5.9: A normal stereo audio buffer, compared to an interleaved stereo buffer. Notice that the interleaved buffer of the same length contains half the amount audio data.

The processing pipeline for TimeStretcher is shown in Figure 5.10. Note that due to the time stretch, the ratio of input samples to output samples is not equal. TimeStretcher must therefore keep track of the current position in the track *pre-stretch*. Furthermore, SoundTouch has a processing latency, whereby a number of input blocks must be sent before an output block is received. Note the loop in 5.10, which happens when SoundTouch has not yet received enough input samples. This latency fortifies the need for a dedicated 'input playhead' to store the current position in the input audio.

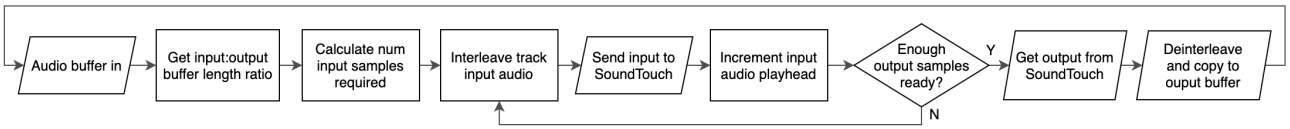


Figure 5.10: Processing pipeline of the TimeStretcher object.

5.2.2 Effects

Unfortunately, there was not enough time to implement the intended mixing effects. This has implications for the AI DJ system, which are discussed in the next section. A full set of effects found in a typical DJ system is intended for future versions of AutoDJ, the design of which will be discussed here.

Professional DJ systems, such as the industry-standard Pioneer DJM-900, have a vast array of DSP effects, as demonstrated in this video. These are usually constructed using some combination of reverb, delay/looping, and filtering. As such, a diverse effects system could be formulated using different configurations of these three elements.

Firstly, it matters where in the audio pipeline an effect is applied. At a high-level, there is a choice between applying effects at the master output, or on the individual tracks. For the purposes of a DJ transition, individual track effects are much more common, because they can smooth the fading of a track more effectively in this configuration. As mentioned in section 1.3.1, though, there are no steadfast rules on how to DJ, so master effects should not be disregarded.

Even within the signal chain of an individual track, effects placement has important implications. If a reverb of delay is placed before the gain attenuation, the signal will decay quickly as the track is faded out; if it is placed after the gain, the effect's decay will not be affected by the attenuation, which can result in a smoother fade out.

Another important aspect of effect chains is serial versus parallel operation. In a serial setup, the effects will combine to give an output which is generally more 'blurred'. In a parallel configuration, the dry signal might be audible for longer - if a long delay is used on parallel with a short reverb, for example.

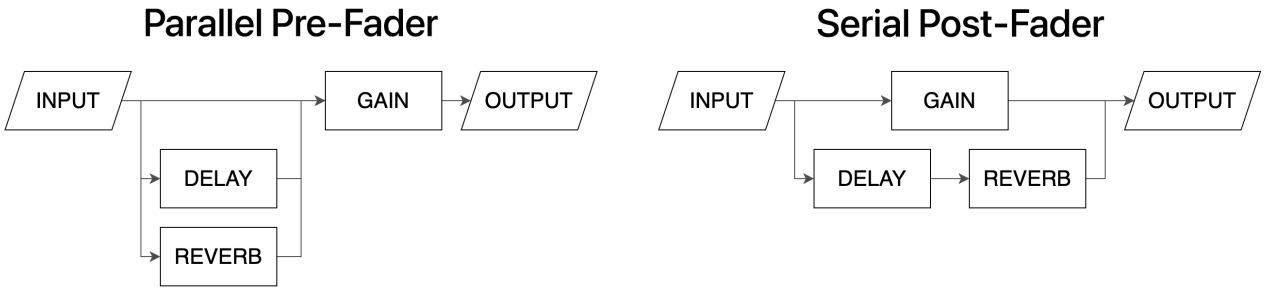


Figure 5.11: Possible configurations for DSP effects in AutoDJ.

A modular effects unit should be integrated into AutoDJ, which allows for different placements and signal chains depending on the mixing decisions from the AI DJ. JUCE provide a graph structure, AudioProcessorGraph, which could be used. It can arrange multiple DSP units in any configuration - as discussed in this tutorial. JUCE also comes with basic DSP effects units, which could be combined to create the more complex setups described in this section.

5.3 Audio Analysis Algorithms

After an extensive review of MIR algorithms and automatic DJ systems during the research phase, it was clear which methods would be useful for a given purpose. In AutoDJ, temporal analysis is used for beat-matching purposes; tonal analysis informs the track selection process, and segmentation is used to find appropriate transition points. An experimental parameter, *danceability*, is also part of the track selection process - note that this has been renamed to 'groove' in the UI.

There was not time to design bespoke algorithms for this project, or to write code implementations of existing designs. Thus, it was necessary to look for existing code solutions to integrate into the system.

Many researchers use Python to develop their algorithms, so there are limited options for C++. Unfortunately, the library of greatest interest, Madmom [7], is a Python implementation. These algorithms performed very well in their 2016 MIREX evaluation, scoring the highest F-measure score for beat and downbeat tracking.

In a search for alternatives, the design of Mixxx was used as reference. Mixxx is a modular DJ software examined in section 2.2.3, founded by a scholar at Queen Mary University of London (QM). For beat tracking, it makes use of their in-house audio processing library, QM-DSP.

5.3.1 Queen Mary's DSP

QM-DSP is a signal processing library which focuses heavily on MIR. It includes an extensive set of algorithms that cover most areas of MIR functionality. The design could be considered a little dated, since it was conceived in 2006, but open-source development has continued to this day.

While there is no dedicated documentation for this library, it is used as part of the QM Vamp plugins package, which has some useful supporting material. Here, we can see which research papers the algorithms are based on.

In the current configuration of AutoDJ, QM-DSP is used for the following:

- Downbeat detection in AnalyserBeatsEssentia - based on [23].
- Key signature estimation in AnalyserKey - based on [79].
- Track segmentation in AnalyserSegments - based on [66].

As mentioned, the Mixxx source code served as a useful reference for how to integrate this library. In fact, some utility functions from Mixxx were also included in AutoDJ, which can perform post-processing on beat tracker results. Their primary purpose is to convert variable-tempo beat positions to a constant beat grid. This works by finding near-constant sections in the variable grid, and evaluating what tempo value is most common throughout these sections. The dominant beat phase alignment is then also extracted from the constant sections.

In order to map out AutoDJ's mix trajectory on a 2D plane, some kind of high-level music descriptor relating to energy was required. QM-DSP does not provide such a metric, so another library is used in conjunction with this one.

5.3.2 Essentia

Esssentia, developed at Pompeu Fabra University, is another library which offers a combination of DSP and MIR functionality [2]. It is more modern than QM-DSP, and provides extensive documentation. There is also a greater range of algorithms available, with extra categories, such as filters, file I/O, and machine learning models.

In the current configuration of AutoDJ, Essentia is used for the following:

- Beat tracking in AnalyserBeatsEssentia - based on [76].
- Beat phase correction in AnalyserBeatsEssentia - a novel method using pulse trains from [81], see section 5.3.3.
- Danceability analysis in AnalyserGroove - based on [91].

The danceability analysis is used in the mix trajectory plane, as the second dimension to tempo. This algorithm uses detrended fluctuation analysis (DFA), which originated in the medical field, for revealing correlation in data across varying time scales [18]. DFA was first applied to music in [4], where it showed promise for music genre classification tasks. [91] built on this idea by applying DFA to an annotated data set. The manual annotations included a number of high-level descriptors of each track, such as "danceable", "energetic" and "melancholic". They found strong correlations between the "danceable" annotations and the DFA results, which prompted the development of the algorithm in use.

The danceability algorithm does not distinguish between tracks of the same genre very effectively, and is intended for broader categorisation of a varied library. Nevertheless, it was employed in AutoDJ as somewhat of an experiment, to see whether it gave some kind of meaningful distribution of the tracks. The versatility of DFA has already been demonstrated in the cited research, so it holds promise here. The metric was given a more abstract name within AutoDJ - *groove* - so as not to confuse users if the results do not correlate strongly with what they feel is danceable.

Data Visualisation

When developing or integrating an algorithm in an environment such as MATLAB, the data pipeline can be examined in a number of ways. It is easy to create a graph of an intermediate function output, for example, to get a sense of whether that part of the code is performing as intended. To mimic this functionality, a dedicated graph utility class was created, which displays in a separate window to AutoDJ, if the `SHOW_GRAPH` macro is defined. This was indispensable during the integration of MIR algorithms.

The static texttttGraphComponent::store() function can be called from anywhere in the program, and will accept an array of POD type to display. The graph of an onset strength signal (OSS) produced by QM-DSP is shown in Figure 5.12. The x-axis corresponds to the elements of the provided array - the OSS frames, in this case - and the y-axis is the magnitude of the data. Mouse hovering can be used to examine individual values, which are displayed in the top-left corner. Here, the mouse is over the 6447th frame of the signal, which has a magnitude

of 670.862. The shape of the data looks appropriate, when compared to the waveform of the same track shown in AutoDJ - notice the decreased onset activity in the mid section.

Figure 5.12: The custom graphing tool used to examine data during MIR algorithm integration.

5.3.3 Temporal Analysis

The most important analysis component is certainly the temporal aspect. Without this, the beat-matching falls apart, which would give a much more unpleasant result than if the track choice was not tonally coherent, for example.

Before deciding on the temporal analysis approach, several assumptions were made about the input audio:

- **4/4 Time Signature:** All input tracks will contain 4 beats per bar, meaning downbeats occur every 4 beats.
- **Constant Tempo:** The BPM will remain perfectly constant throughout a track.
- **Integer Tempo:** The BPM will be a whole number.
- **Tempo Range:** A restricted tempo range will be used to prevent octave errors, where the estimation is double or half the true value.

These assumptions are based on the characteristics of the target music genre - house. They help to simplify the design process, as well as potentially increase accuracy when analysing the intended material. The disadvantage of such assumptions is they limit the system to a certain class of input data, effectively decreasing overall robustness.

A number of analysis configurations were tested during development - see section 6.1. Here, we will look at the *current* analysis implementation, which gave the best results over all.

The pipeline of the temporal analyser, AnalyserBeatsEssentia, is shown in Figure 5.13. After receiving the track audio, it resets the state of all analysis objects, including the Essentia and QM-DSP analysers. The Essentia beat tracker in use is called RhythmExtractor2013, which implements the method described in [76]. First, it must be passed the input audio, as well as several variables that will receive its output values. Once the algorithm is configured, RhythmExtractor2013 is ready for beat tracking computation. The bpm value it returns is rounded to an integer and taken as the final bpm estimate.

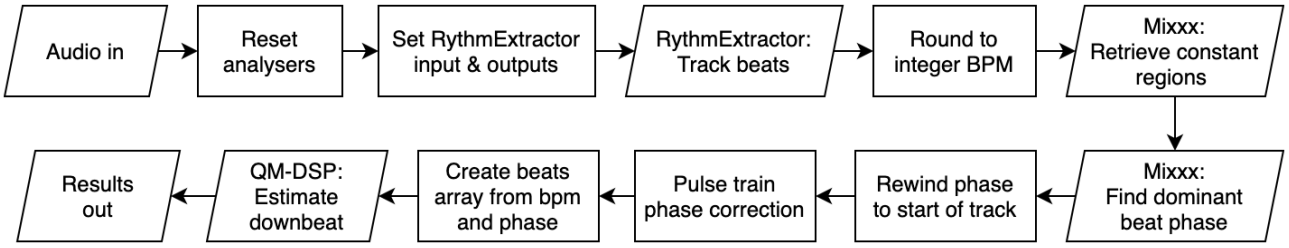


Figure 5.13: The temporal analysis pipeline in AutoDJ.

RhythmExtractor2013 also returns a series of beat positions, which are passed to the Mixxx utility functions, which find constant regions and then the dominant beat phase, as described in section 5.3.1. The resultant phase is sometimes offset by an arbitrary number of beats, so 'rewinding' is necessary to bring it back to the track start. This is achieved by subtracting the maximum number of whole beat periods that fit into the phase value. Afterwards, the phase is corrected through a novel method described shortly.

Essentia does not provide a downbeat extraction algorithm, so the solution from QM-DSP is used, which is based on [23]. As well as the raw track audio, this algorithm requires a beat grid, so one is constructed using the BPM and phase estimates from the prior steps:

```
void AnalyserBeatsEssentia::getDownbeat(juce::AudioBuffer<float>* audio, int bpm, int beatPhase, int& downbeat)
{
    // The downbeat algorithm requires a grid of beat position, so we need to construct one
    std::vector<double> beats;
    // Find the number of downbeat frames that will fit into the audio length
    int numFrames = (audio->getNumSamples() - STEP_SIZE_DOWNBEAT) / STEP_SIZE_DOWNBEAT;

    // For each frame to be analysed by the downbeat algorithm
    for (int i = 0; i < numFrames; i++)
    {
        // If the frame contains a beat, add the frame number to the beat grid array
        if (isBeat(i, bpm, beatPhase))
        {
            beats.push_back(i);
        }
    }

    // Find downbeats using QM-DSP algorithm
    downBeat->findDownBeats(downsampled, downLength, beats, downbeats);
```

5.3.4 Beat Phase Correction using Pulse Trains

Even before the formal analysis tests were conducted (see section 6.1), it was clear through using AutoDJ that the beat tracking algorithms have a tendency to synchronise with the off-beat of the music. Off-beats are the time points exactly halfway between each beat. A beat tracker that is locked to the off-beat will report the correct tempo, but a phase that is offset by half a beat period.

The music in use tends to have powerful transient onsets on every off-beat, due to the loud hi-hat cymbals which are a common stylistic feature in house music. This suggests that the behaviour exhibited by the beat trackers was not an error in implementation, but a symptom of the target genre. It was therefore necessary to develop a bespoke method for correcting these phase offsets.

Practically any hi-hat cymbal will produce a high-frequency 'tiss' sound when struck. The first correction technique proposed is to low-pass filter the audio given to the beat tracker. Unfortunately, this is detrimental to the overall accuracy of the beat tracker, so it was not appropriate for use.

The correction method that was implemented is believed to be a novel use of the pulse trains proposed by [81]. These pulse trains are originally used to rate the confidence of a set of tempo estimates, by cross correlating a pulse train for each tempo with the onset strength signal (OSS). See section 2.2.1 for full details on this method. Crucially, the algorithm tests the pulse trains at a full rotation of phases, giving a cross-correlation score for every phase alignment against the OSS.

In AutoDJ the pulse train algorithm is used to evaluate only *one* tempo, the final BPM estimate, against an OSS, to determine whether the beat phase is half a period out of synchronisation. The pipeline for this correction process is shown in Figure 5.14.

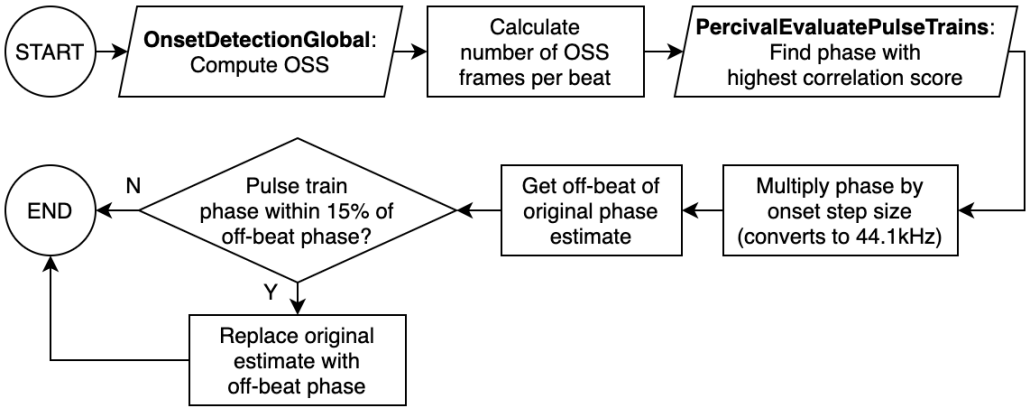


Figure 5.14: Processing pipeline for off-beat phase correction.

No OSS is available from the previous analysis steps, so the OnsetDetectionGlobal function from Essentia is used to compute one. The pulse train algorithm requires the tempo estimate as a beat period, so that it can be used at the spacing between the impulses. Furthermore, this beat period must be at the sample rate of the OSS, so effectively, we need to find the number of OSS frames per beat. This can be found by dividing the beat period by the OSS step size (the number of samples between each OSS frame).

The OSS and beat period are then handed to PercivalEvaluatePulseTrains, which returns the phase alignment that gave the highest cross correlation score with the OSS. Again, this phase is in terms of OSS frames, so it must be *multiplied* by the OSS step size, to bring it back to a 44.1kHz sample rate.

Next, we compare the pulse train result with the original phase estimate from the beat tracking algorithm. If the pulse train result is close to the off-beat of the phase estimate, we deduce that the original estimate is actually incorrect. In this case, the 'off-beat' is considered to be *on* the beat, and is taken as the corrected estimate, as shown in Figure 5.15. If the pulse train is *not* close to off-beat phase, the original estimate is considered valid.

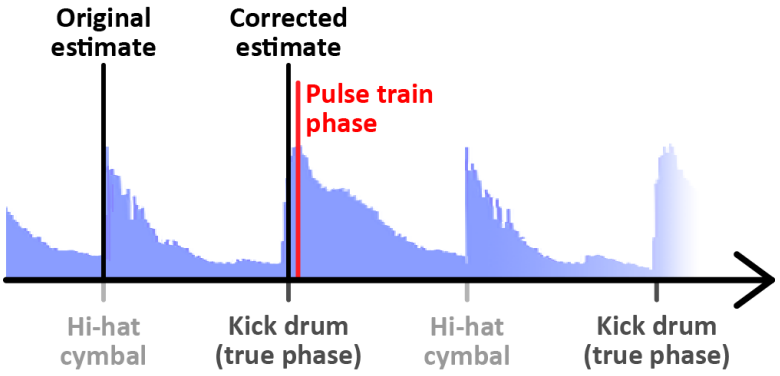


Figure 5.15: Diagram to show an off-beat estimation and the corrected version.

The pulse train phase is considered 'close' to the off-beat if it is within 15% of the beat period, that is, +/- 15% of the number of audio samples per beat. While the pulse train correlation is not accurate enough to use as the overall phase estimation, testing shows that it is very effective at *correcting* an existing phase estimate using this tolerance. However, more rigorous testing is needed to determine the ideal parameter settings.

5.4 Artificial Intelligence

The mixing decisions in AutoDJ are all handled by ArtificialDJ. This class manages the choosing of tracks to play, and how they are mixed together. It takes into account the tempo, key and danceability of the tracks to encourage a coherent flow of music. Segmentation analysis influences the transition timings, with the aim of starting and finishing transitions at the boundaries between musical sections - a foundational DJ technique. To encourage novelty, randomness is integrated into every decision, with a Gaussian weighting to keep the average result close to the tendencies of a human DJ.

As mentioned in section 5.2.2, the intended DSP effects have not yet been implemented, which means the AI does not have a full set of mixing parameters to control. Furthermore, only one type of crossfade is possible at present, which is a simple linear volume transition with subtle high-pass filter modulation. This means that the combinational model of creativity is somewhat incomplete, because there are not enough elements for the AI to combine into novel ideas.

5.4.1 Mix Generation

ArtificialDJ operates on its own thread, queueing up a series of mixing decisions that are fetched by the audio processing objects when required. As such, the system does not make its decisions in real time. This advance processing does not affect the final output, primarily because ArtificialDJ is the *sole* controller of the system. If the user could influence the decisions in some way, then real time operation might be necessary. At present, real-time decisions would add unnecessary complexity, because the efficient audio processing routines would have to wait for a decision to be made when they need it, potentially resulting in audio drop-outs.

ArtificialDJ generates all the decisions relating to a mix transition in one batch process. The resultant data is stored in a `struct` called MixInfo. In C++, a `struct` is much like a simplified class, where all the internal data is publicly accessible by default. The MixInfo objects are stored in an array belonging to ArtificialDJ, called the mix queue. When TrackProcessor finishes playing a track, it will use the information at the start of the mix queue to ready itself for the next DJ transition.

Once AutoDJ has begun playback, the mix generation thread runs in a loop to keep the queue full. The DJ sleeps for 1 second in between each iteration, to avoid unnecessary computational load on the system. If the DJ runs out of analysed, unplayed tracks to choose from, it enters an ending state. If more tracks are analysed, normal operation is restored; if not, the TrackProcessors will be given a null mix, which results in the last track playing until it ends.

The mix generation pipeline is shown in Figure 5.16. Firstly, the DJ checks that the mix queue is not full, and that the mix is not ending. If the checks pass, it calls on TrackChooser to pick the next track to be played (more on this shortly). A null track choice indicates that there are no more tracks to play, and the mix should end.

Once a track is chosen, the tempo to be used during the transition is set as the average of the new track BPM and the one before it. The audio data for the next track is then loaded into RAM by DataManager. AnalyserSegments find the boundaries between musical sections

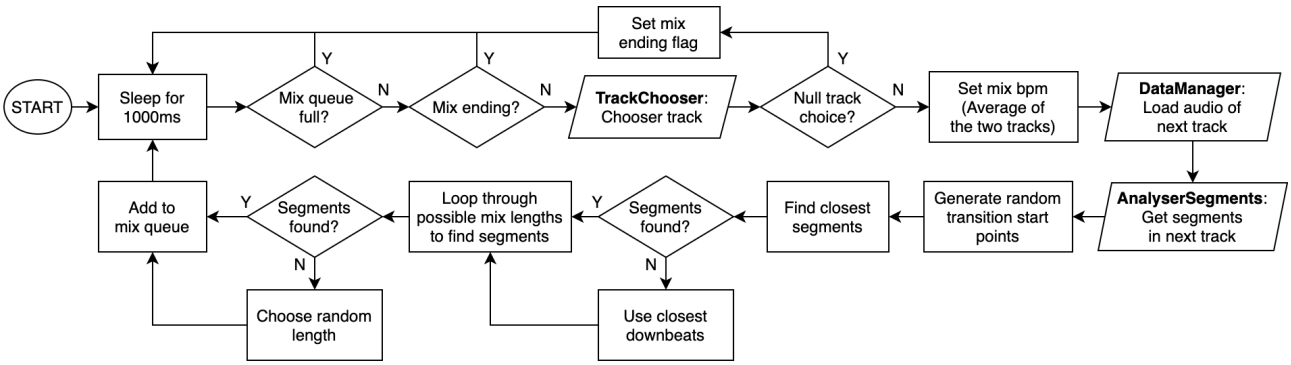


Figure 5.16: ArtificialDJ's mix generation pipeline. This class keeps a queue of transition ready for the audio processing pipeline to fetch.

in the track, which then inform that transition start and end points.

ArtificialDJ starts by choosing random points in both tracks to start the transition. The closest segments to these points, within a given range, are then taken as the mix start points. If no segments are in range, then the downbeat nearest to the random start point is used. Next various transition ending points are tested to see if they coincide with any segments. The DJ starts with an 8 bar length, then increments by 4 bars until the maximum length is reached. If a particular length coincides with segments in both tracks, that one is chosen. Otherwise, the longest transition that hit one segment is chosen, or a random length if none were hit.

5.4.2 Track Choice

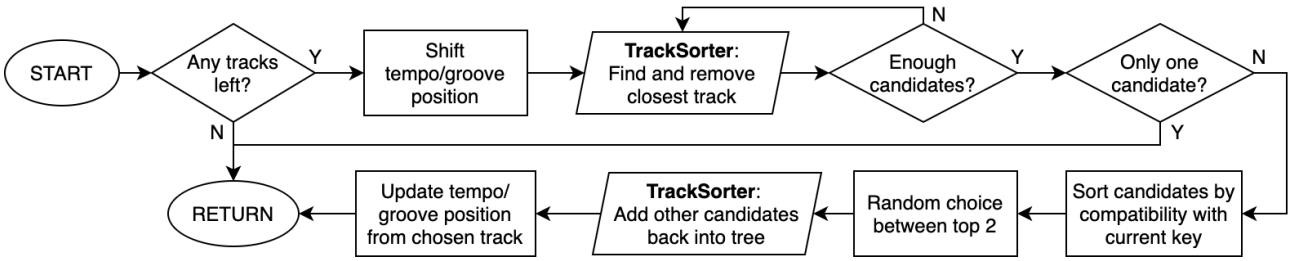


Figure 5.17: The track choice pipeline.

The TrackChooser class is used by ArtificialDJ to choose a new track for each mix. Figure 5.17 shows the steps it takes, starting with a simple check that there is at least one analysed, unplayed track to choose from. The *Direction* view in AutoDJ, shown in Figure 5.18, serves as a good metaphor for how TrackChooser goes about the remainder of the choosing process. TrackChooser stores its current position in the 2D plane of tempo and groove. To keep the DJ mix interesting, the system slowly wanders around this plane, resulting in a gradual evolution of speed and musical 'feeling'. As such, TrackChooser also stores its velocity and acceleration in terms of both tempo and groove. The velocity is added to the current position in the plane before the track choice is made.

A small set of tracks nearest to the shifted position are taken as candidates for the new mix. TrackSorter stores the distribution of tracks in the mix trajectory plane. It uses a quadtree representation, which allows for extremely efficient searching in 2D space. To avoid

reinventing the wheel, a third-party quadtree implementation is used, which is simply wrapped by TrackSorter into a much cleaner interface.

The track candidates are removed from the tree and sorted by their compatibility with the key signature of the previously chosen track. A random choice between the top two candidates is returned as the next track to be played. Any candidates not chosen are added back into the quadtree.

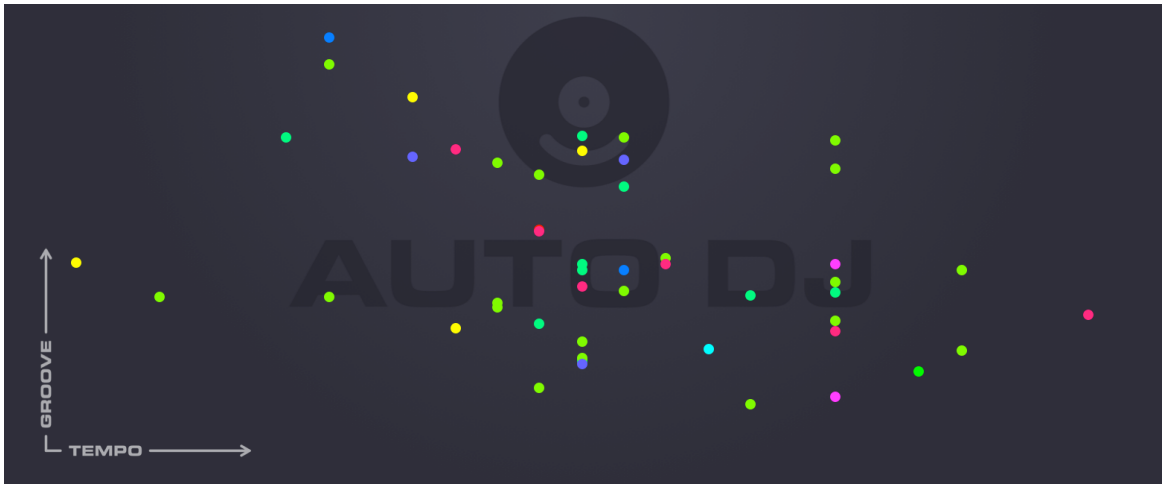


Figure 5.18: The Direction view in AutoDJ distributes tracks in terms of tempo and groove.

5.4.3 Gaussian Weighting

As mentioned, randomness is used to encourage novelty in the decisions made by ArtificialDJ. If given too much power, this randomness could get out of hand, resulting in too many deviant decisions that might decrease the overall coherence of the DJ mix.

A 'sensible' decision can be defined as the one most likely to be taken by a human DJ in a given context. For example, most of the time, a DJ will start a track from the beginning - or the first downbeat, to be precise. We mimic this tendency by using a Gaussian distribution to weight the initial track positions towards the first downbeat, as shown in Figure 5.19. This concept can be extended to most decisions made by ArtificialDJ.

The RandomGenerator object is used to fetch random values. For the Gaussian-weighted versions, it makes use of `std` library, which provides a function called `normal_distribution`. The mean, standard deviation, and limits of the random value can all be controlled using RandomGenerator.

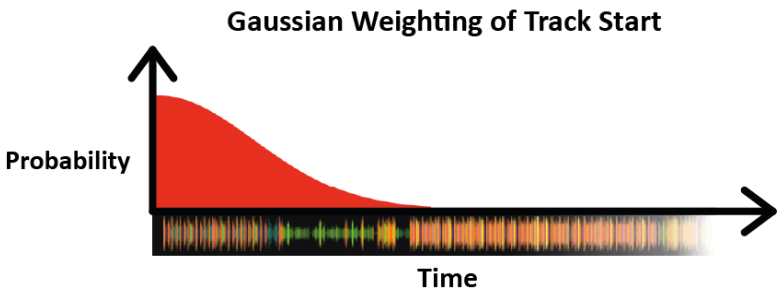


Figure 5.19: Gaussain distributions are used to weight randomness in the mixing decisions.

5.5 User Interface

Since the initial development platform is MacOS, Apple's Human Interface Guidelines were used as reference for the user interface (UI) design. These are a comprehensive set of guidelines covering everything from high-level design themes, down to when and how to use coloured icons in the MacBook TouchBar. Given the potential for cross-platform development in future, the generalised high-level concepts were the focus here.

One of the most important decisions to make before designing a UI is whether it should be resizable, because this influences the distribution of UI elements on the screen. Unlike the constrained world of iOS, the guidelines state that MacOS apps should be "flexible and expansive" [11]. JUCE includes support for resizable windows, so they are used here.

Another major choice was whether to use multiple views, or keep the app to a single screen. The guidelines say "MacOS is designed to keep the current task clear and in focus" [11], which suggests an inclination to breaking the application up into focused sections. Traditionally, DJ software shows all the information in a single view, so that DJs can quickly glance at anything they need. However, in an automated system, this is not so important. Dividing the problem into manageable sections can reduce cognitive load on the user, which is especially relevant for a music app aimed at a partially non-musical audience. For this reason, it was decided the app would use multiple views.

5.5.1 Initial Design

Pencil sketches were made while brainstorming the initial UI designs. The most relevant sketch is shown in Figure 5.20, which formed the basis of the final design. This version includes two views: one dealing with the music library and mix direction, the other for viewing the live DJ mix.

The direction view shows the high-level trajectory of the mix, distributing the music library in a 2D plane of tempo and energy level. Here, the user should be able to control the direction in which the mix moves, for example, they might want to slow down the tempo over a period of time. To the left of the 2D plane are tables showing the music library and queued tracks.

In the second view, users can watch the tracks currently being mixed. The basic metadata of the tracks is shown, as well as coloured audio waveforms. The 'full' waveform is a thin strip which shows the full track; the zoomed version is a much larger view of the current position in the track. These are inspired by the waveforms in Serato, which give a high-level and low-level view of the music in parallel. Beside the waveforms are buttons for shifting the beat grid to correct temporal analysis error.

Notice the 'DJ Log', which is a rolling list of live updates about the decisions the AI is making. This could be a powerful debugging/development tool, as well as an entertaining feature for the user.

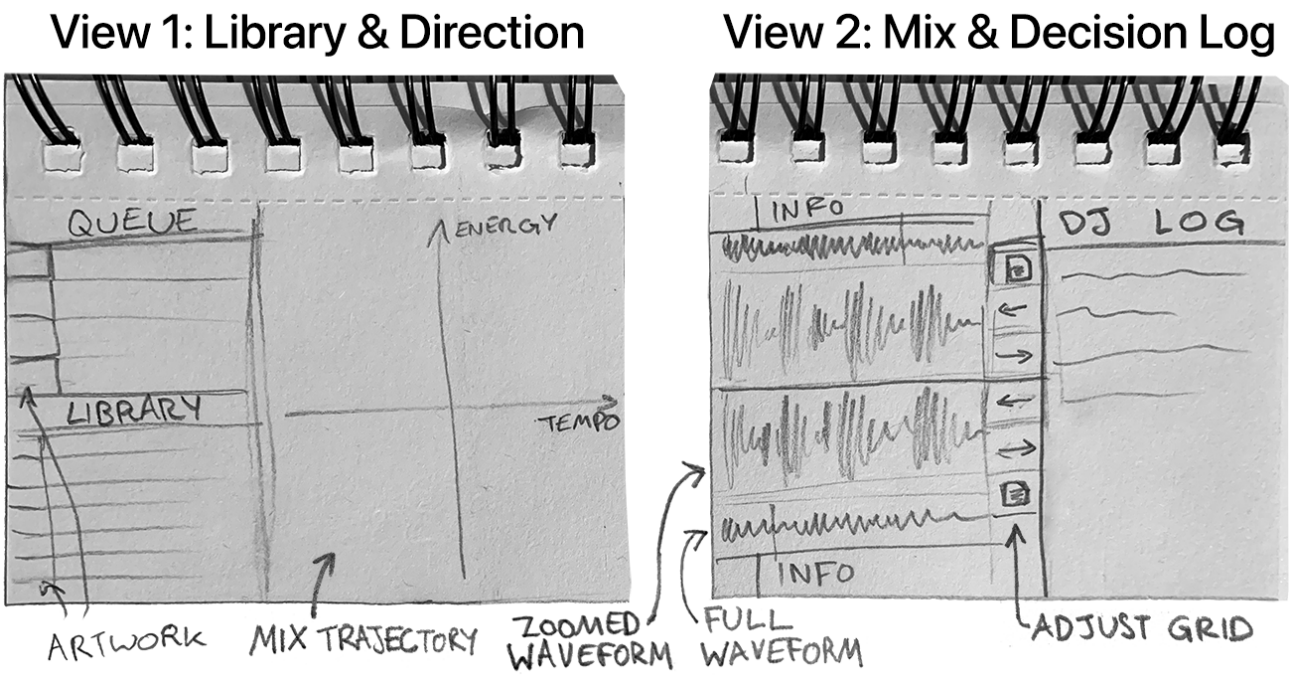


Figure 5.20: Sketches from the initial UI brainstorming, which formed the basis of the final design.

5.5.2 Mock-up Design

The initial UI design felt quite cluttered. There is a lot of information packed into each screen, which would be difficult for the user to absorb at first glance. For the revised mock-up, the application was further split up into three views. An interactive version of the design can be found here - click anywhere on the preview to see which buttons you can interact with.

As shown in Figure 5.21, the revised design is split into three areas of thought, dealing with the music library, mix trajectory/direction, and the current tracks. These views are selected using tab style buttons in the bottom left. The tab buttons are part of a toolbar which is shown on all screens. The toolbar makes use of the spatial groupings to indicate relationships between the controls. The tabs are close together, as well as the audio transport controls, and there is plenty of space between the groups to disassociate them.

Row A of Figure 5.21 shows the Library view, which contains the same tabular elements from the initial UI design, listing the music library and queued tracks. Given the number of columns in these tables, the extra space they get here is welcome. The user could drag tracks from the main library table into the queue, as well as reorder items in the queue. Temporal analysis correction has been moved to the Library view in this revised design. Users can select a track and view/edit the beat grid using the waveform panel, which replaces the queue table - shown in row B. This would provide an optional metronome click so the user can verify the beat grid through listening as well as looking.

The Direction view still shows a distribution of tracks based on tempo and energy, with the addition of colour coding against key signature, where each colour represents a position on the Camelot wheel. The user can click on a dot to view the track details, as shown in row B. Note that the queue table is carried across to this view, because the queue is somewhat determined *by* the mix direction. The user should be able to change the mix direction by dragging the

Figure 5.21: The revised UI mock-up design, created in Adobe XD - interactive version.

black arrow to point at a new position. Note the *Velocity* slider, which controls the rate at which the mix progresses to the chosen location.

The Mix view is essentially the same as the original design, with improved use of space. The brightness of the zoomed waveforms should be used as a metaphor for the volume level of each track, similar to the Ableton note brightness in section 2.3.3. An improvement could be made here, as there is no clear hierarchy between the track details - spatial hierarchy could help the user find the information they want at a glance. The Title and Artist labels should be the focus, with the other data underneath.

5.5.3 Current Design

We will now examine the user interface in its current state, shown in Figure 5.22. Note that some features from the original design are not yet implemented - see chapter 7 for a full review of progress. At present, the system is fully automatic, with no user control over queued tracks or mix direction. There is no artwork displayed in the UI; this could be added in future by integrating a third-party metadata extraction library, such as FFmpeg.

The most striking difference from the mock-up is the dark colour scheme. 'Dark mode' design themes have become popular in recent years; they make use of inky background and panel colours to put focus on the *content*, rather than the surrounding window. A lack of bright white areas is also more ergonomic during the night, because it helps to prevent squinting. After testing early iterations of the software, it was decided that a dark background would be appropriate, primarily because DJ software is often used in dark environments.

The settings page, shown on screen 8 in Figure 5.22, was not considered in the original design. After an audio output issue was encountered by a test participant, this page was added to allow users to change their audio device settings.

Figure 5.22: Screenshots of the current user interface implementation. Screens 1 and 2 show the first step where the user must choose a folder of music files and wait for them to be parsed.

Early testing also indicated that the text-only tab buttons from the original design did not clearly invite interaction. Rectangular outlines were added to help with the perceived affordance of these buttons, with a highlight to indicate the current selection.

Another HCI best practice, plain language, was used throughout the UI design. Words were chosen carefully to cater to the wide target audience. Consider screen 1 in Figure 5.22, 'Choose Folder' was chosen over 'Choose Directory' here, because it is the more widely used term. Similarly, track 'length' was preferred over 'duration' throughout the design. Where possible, icons were used instead of text because they improve the *glanceability* of the UI.

'Tempo' versus 'BPM' was a more difficult choice: both are musical terms that are not universally understood, and more general terms - such as 'speed' - would not be appropriate. BPM was chosen because it is more commonly used in DJ software and communicates the units of measurement - beats per minute - in a concise acronym.

Loading States

The MacOS design guidelines give recommendations for dealing with loading times. The user should be able to "gauge how long a process will take to complete," hence the use of progress bars for the initial load and library analysis, shown on screens 2 and 3 respectively. The user is shown the library of tracks even if analysis is incomplete, because MacOS apps should "show content as soon as possible." This helps the UI feel smooth and responsive.

When the user first presses play, the AI takes a short period to generate the first DJ mix before audio starts playing. One of the guidelines recommends instant acknowledgement of user interactions, so a pulsating animation was added to the play button during this brief background loading period. This confirms to the user that their command has been registered and the app is processing something.

When there are not enough mixing decisions loaded by the AI DJ, skipping ahead would cause an error. To avoid this situation, the skip button is disabled during these periods, rather than showing a retrospective error message to the user. This approach is in line with the prevention of user error recommended in section 2.3.2.

5.5.4 Waveform Implementation

The audio waveforms in AutoDJ communicate audio levels and timbre in a single view. They are inspired by the waveforms in Serato, shown in Figure 5.23, and produce a very similar colour palette in response to the frequency content of audio. Note that, for efficiency, AutoDJ uses half-waveforms which are reflected in the horizontal axis. Serato uses full waveforms with positive *and* negative values.

A class diagram for WaveformView is shown in the appendix, which holds both the full and zoomed waveforms, as well as a background thread which loads the data to display. Figure 5.24 shows the loading process that occurs on this background thread. On a basic level, the waveforms are generated by measuring the audio levels in three filtered bands: low, medium and high. Each band determines the amount of red, green and blue for each colour point respectively.

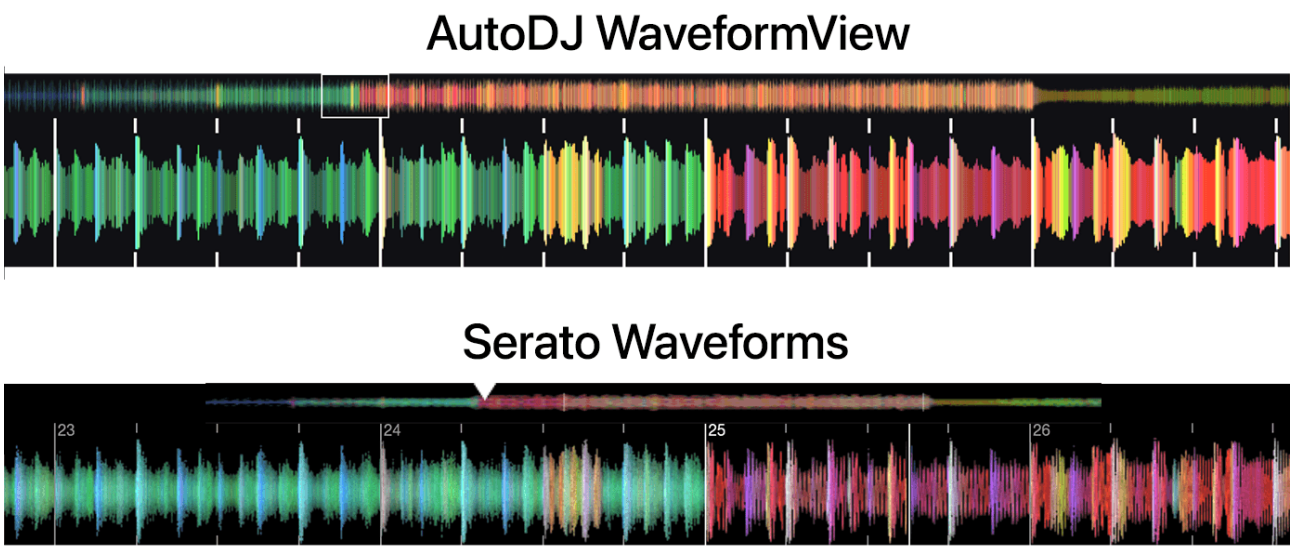


Figure 5.23: A comparison of the waveform views in AutoDJ (top) and Serato (bottom).

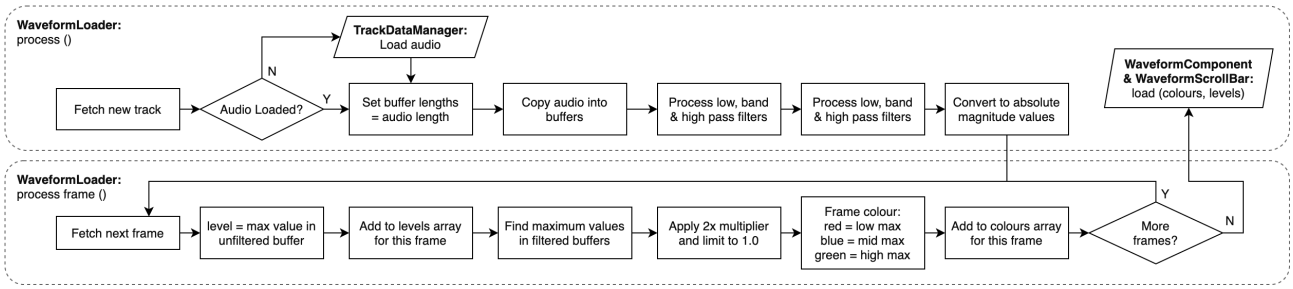


Figure 5.24: Flow chart showing the waveform load process, which executes on a background thread.

First, the track audio is copied into four buffers, which are: unfiltered, low-pass filtered, band-pass filtered and high-pass filtered. This process occurs every time a new track is loaded, so it must be as efficient as possible. Infinite Impulse Response (IIR) filters are typically more efficient to process than their Finite Impulse Response (FIR) counterparts, but they are more complex to design. Luckily, JUCE has this covered with the IIRCoefficients class. This can be used to set the coefficients of a juce::IIRFilter, using a different function depending on the type of filter: makeLowPass(), makeBandPass() or makeHighPass(). The following code sets the coefficients using these arguments: sample rate (Hz), cut-off frequency (Hz) and resonance.

```
// Generate coefficients for the low-, band- and high-pass IIR filters
filterLow.setCoefficients(juce::IIRCoefficients::makeLowPass(SUPPORTED_SAMPLERATE, 200, 1.0));
filterMid.setCoefficients(juce::IIRCoefficients::makeBandPass(SUPPORTED_SAMPLERATE, 500, 1.0));
filterHigh.setCoefficients(juce::IIRCoefficients::makeHighPass(SUPPORTED_SAMPLERATE, 10000, 1.0));
```

After filtering, the audio amplitude values are converted to magnitude - that is, they are made positive, absolute values. This is the digital equivalent of full wave rectification.

Every pixel along the horizontal axis of the waveform corresponds to a window, or *frame*, of audio. The frame size is currently set to 380, meaning 380 audio samples are processed to produce the height and colour of each waveform frame. The height of each frame is calculated from audio level, which is simply the maximum value of the unfiltered audio within a frame.

The colour of each frame is a combination of the low, medium and high frequency content. The maximum value of each filtered audio band is taken for every frame, and then scaled to an appropriate range. The colour is then constructed using RGB values, where the red, blue and green channels are determined by the low-, band- and high-pass maximums respectively. This means that bass-heavy sections of music have warm, reddish tones. Higher-pitch, 'airy' sounds have cool, blue or green tones. When there is activity in multiple frequency bands, the colours combine to produce oranges, yellows, whites and so on.

Figure 5.25: The red, green and blue channels of the waveform are determined by the audio frequency content.

Once the colour and level data is calculated for all frames, it is sent to the two waveforms to be drawn. At various stages in the loading process, the thread checks for a new request. If there is a new track to load, it aborts the current process, resets the associated data and starts loading the new track.

Efficiency

The waveform loading and drawing processes are very expensive, typically taking a few seconds to execute. They should therefore be carried out only once - when the track is loaded. This presents a challenge for the zoomed waveform, because it responds dynamically to track position, time-stretch and volume level.

The solution is to draw the waveforms to an image object, which is stored for as long as required. JUCE provides measures for manipulating this image to respond to the track parameters:

- **Track position:** The image is simply drawn to the screen at an offset along the x-axis. This means we see only the relevant portion of the track in the visible area.
- **Time-stretch:** The image is stretched by the same amount as the audio, using the rescale() method of juce::Image.
- **Volume level:** The image is drawn with brightness which is directly proportional to volume.

Once the waveform image has been manipulated, the beat grid is drawn on top. For each visible waveform frame, the class uses its isBeat() function to determine whether the frame contains a beat. If it does, a white marker is drawn above and below the waveform. If it is a downbeat, a white marker is drawn across the full height of the waveform.

6 Testing

We will now look at the testing methodology and results. The software was tested from two perspectives: audio analysis and user experience. Both of these are equally valuable, and led to important insights on the current performance of AutoDJ, as well how it could be improved.

6.1 Audio Analysis

To test the performance of the temporal analysis algorithms in AutoDJ, ground truth reference data was required. There are some public data sets of relevance, for example, one developed in 2015 giving tempo and key data for 664 electronic dance music tracks [5], as well as a very recent set of annotated DJ mixes [88]. Unfortunately, none of the existing options encapsulate all the parameters necessary to fully evaluate the temporal algorithms. Beat phase and downbeat were notably absent from most sets.

To navigate this issue, a bespoke data set was created for the project. This is a manually-annotated set of 50 dance music clips, each 2:30 in length. The annotations include tempo (in BPM), beat phase (in samples) and downbeat (in beats), which are stored in the same type of database as the normal AutoDJ analysis results (see section 5.1.1).

	filename	hash	length	bpm	beatPhase	downbeat	key	groove
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
40	Prospa - Ecstasy (Over & Over) [Special Request ...	1005395051	150	130	7500	3	13	1.30900
41	Raito - Ocean Tides.mp3	1781119260	150	128	7700	0	2	1.43877
42	Reuben James - I Know You Too Well (Karma Kid 12" ...	-608753458	150	123	5000	0	13	1.76626
43	salute - Syl.mp3	1347431414	150	133	4700	0	13	1.28828
44	Second City - I Wanna Feel (Radio Edit).mp3	638515698	150	122	2350	0	13	1.80117
45	Serge Funk - Can't Get Enough.mp3	626694714	150	121	13550	2	24	1.83832
46	Soul Reductions - Got 2 Be Loved.mp3	489861723	150	122	2600	1	13	1.40895
47	Taiki Nulight - Hold On.mp3	528875863	150	124	5700	0	2	1.52341
48	TCTS - Live for Something.mp3	-777970802	150	123	5000	0	13	1.18743
49	Tommy Theo - Ain't He Bad!.mp3	-983141308	150	125	4450	0	13	1.80756
50	Todd Edwards - You're Sorry (Earsling Dub).mp3	-1267232886	150	124	4200	0	13	1.18743

Figure 6.1: An excerpt from the temporal analysis data set, with manual annotations for BPM, beat phase and downbeat location.

Since audio samples are extremely dense in time, it is almost impossible to establish an exact truth for a metric such as beat phase. The phase of beats was measured visually, using Audacity, which gave a precision of about 100 samples. This corresponds to 2ms at a 44.1kHz sampling rate.

The downbeat metric indicates the phase of downbeats, with a valid range of 0-3. This indicates which is a downbeat out of the first four beats in a track - the rest of the downbeats are assumed to be multiples of 4 beats away from this one.

6.1.1 Evaluation Metrics

When evaluating parameters such as beat phase, it is important to consider the smallest margin perceptible by a human listener, known as just-noticeable difference (JND). JND is defined as the smallest value difference which is detected 50% of the time by humans. [37] investigated the JND for detecting a phase difference between two periodic series of pulsing tones (beats). They found that listeners could not distinguish the phase difference if it was below 2.5% of the beat period, for periods larger than 240ms.

The maximum tempo of interest for AutoDJ is around 140BPM, which is a beat period of 429ms - well clear of the 240ms threshold. We can therefore take 2.5% as the JND constant for beat phase, meaning an estimate can deviate from the ground truth by up to **2.5% of the beat period**, before it is considered inaccurate. Note the beat period is specific to the tempo of a given track.

To collect test data, an extension of AnalysisManager was created: AnalysisTest. Instead of analysing tracks and storing the results in a database, this class analysis tracks and *compares* the results with the ground truth database. Once analysis is performed on the entire data set, AnalysisTest performs some post-processing on the results, and produces the following scores:

- **BPM Accuracy:** Since the tempo is stored as an integer, this metric gives the percentage of estimates that were exactly equal to the ground truth.
- **Average BPM Error:** The average magnitude of BPM error in the estimates.
- **Phase Accuracy within 1x JND:** The number of phase estimates that were within 2.5% of the ground truth.
- **Phase Accuracy within 2x JND:** The number of phase estimates that were within 5% of the ground truth.
- **Phase Accuracy within 2x JND or off-beat:** The same as the above metric, with the inclusion of estimates that were within 5% of the off-beat.
- **Average Phase Error:** The average beat phase error, measured in audio samples.
- **Downbeat Accuracy:** The number of downbeat estimates that were correct.
- **Average Time:** The time taken to complete the temporal analysis of a track, averaged over the 50 items in the data set.

Each category of estimate here depends on the previous. If BPM is not estimated correctly, the beat phase is meaningless; and if beat phase is not close to the truth, the downbeat is meaningless. To give a better view of how the different analysis components performed, it was necessary to decouple these dependencies: only the tracks with correct BPM are included in the phase results, and only those with correct phase (within 2x JND) are included in the downbeat result.

As described in section 5.1.4, conditional inclusions can be used to choose which parts of code are included in the app build. A set of macros is defined in BeatTests.hpp which determine the configuration of the temporal analysis setup, including which MIR algorithms to use, and whether to filter the input audio.

6.1.2 Results

The results of these test are shown in appendix D. Three beat-tracking algorithms were tested, all from the third-party libraries detailed in section 5.3. They are listed in the results table as follows:

- **QM-DSP:** The beat tracker included as part of QM-DSP, which in based on [24].
- **MultiFeature:** An Essentia algorithm, based on [54].
- **Degara:** An Essentia algorithm, based on [76].

All three gave very strong tempo detection results on the custom data set, with QM-DSP and Degara scoring 100% accuracy across the 50 clips. This is a fairly surprising result, as the documentation for Essentia suggests that the MultiFeature aglorithm is more advanced than Degara. Perhaps MultiFeature would show greater efficacy on a more diverse data set of various music genres. Nevertheless, the difference in the scores is marginal.

Degara gave the best beat phase analysis, scoring 46% accuracy within the 1x JND, and 50% within 2x JND. MultiFeature was the worst performing, with 20% accuracy within 1x JND. Downbeat accuracy was close, with QM-DSP, MultiFeature and Degara scoring 55%, 50% and 48% respectively. Computational efficiency is highly important in the analysis pipeline, because the user is likely to analyse hundreds of tracks in one batch. Degara was by far the more efficient algorithm, taking around 1 second to complete, whereas the other were both above 3 seconds.

It was hypothesised that focusing on lower frequencies might improve results, because dance music is very bass-driven. Two low-pass filtering stages were tested with all of the algorithms. Firstly, the input audio to the entire pipeline was low-passed with an 800Hz cut-off frequency; then, only the audio given to the downbeat analysis was filtered.

While the filtering had detrimental effects on most results, there was one major improvement to note. The 2x JND phase accuracy of QM-DSP increased from 44% to 0.92%, when the input audio was filtered. With this, the tempo accuracy decreased and the downbeat improved, both by a small margin.

Without filtering, Degara has tempo, phase and downbeat scores comparable to QM-DSP, and better than MultiFeature. Its massive lead in computation time was the most significant result in the whole test, so Degara was the most promising candidate. The only issue was its phase accuracy of 50% (2x JND), which was not adequate for beat-matching purposes. As explained in section 5.3.3, it was clear through using AutoDJ that many of the phase estimates were catching the off-beat of the music. It was therefore decided that an extra phase metric should be tested, which includes the off-beat within a 2x JND window. This proved very successful, Degara scoring 100% accuracy across the data set, and QM-DSP and MultiFeature close behind with 0.98% and 0.85% respectively.

The high proportion of off-beat phase estimates sparked the development of the novel phase correction technique. Once this pulse train method was implemented, as described in section 5.3.3, Degara gave an impressive phase accuracy of 84% within 1x JND, and 100% within 2x JND. The correction did take computation time from 0.98s to 1.37s, but this is still under half that of the other algorithm, so this configuration was chosen and the final setup.

6.2 User Test

User testing is one of the best methods for evaluating the state of a software development project. While certain user testing is appropriate to carry out on a very large group, initial user tests are often more valuable in a small group, as each tester can be given more focus. Figure 6.2 illustrates the diminishing returns found as test group size increases, in relation to the number of usability issues discovered. There were a set of research questions to be answered in this project, so the testing examines much more than usability, meaning it was appropriate to aim for the higher end of the scale in terms of group size.

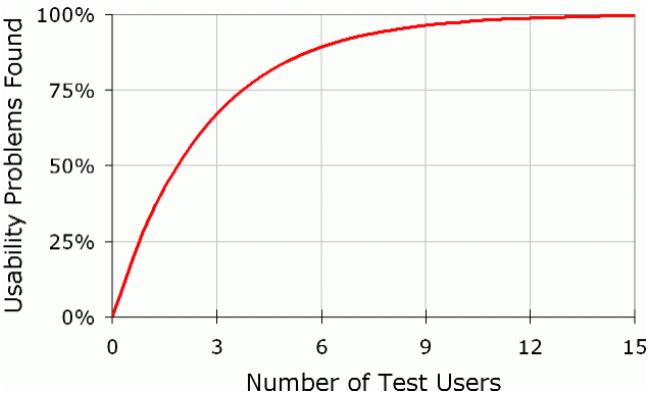


Figure 6.2: A graph to show the number of issues found against user test group size [78].

AutoDJ was sent out in its current configuration to 18 users. They were asked to explore the app for at least 20 minutes, to get a sense of the DJ mixing quality, as well as the general user experience. The 50 clips of dance music from the custom data set were sent with the application, as well as a brief user guide. The prototype build and user guide have since been tagged as a release on GitHub, which can be viewed and downloaded here.

After exploring AutoDJ, the users answered a survey, which can be viewed here. With the aim of collecting comprehensive feedback about the system, the survey covered four distinct sections:

- **About You:** Information on the user's background knowledge and skills, as well as their computer hardware.
- **User Experience:** Feedback on the experience of using the AutoDJ interface - whether it was intuitive, smooth, and so on.
- **AutoDJ Performance:** Subjective evaluation of the DJ mixing performance.
- **Final Thoughts:** Overall feelings about the software and its potential.

The survey was designed with best practices in mind, to avoid biasing the results. Leading questions were avoided, for example: *"Rate the system's beat-matching"* was chosen over *"How good was the system's beat-matching"*. Such questions were also accompanied by an short explanation of the musical terminology.

The test package was first sent to two users for preliminary trials, to ensure a smooth distribution to the wider group. This trial uncovered a major issue that would prevent the application from launching on the majority of systems. The Essentia library was dynamically linked, meaning the user had to have it separately installed on their machine before AutoDJ could run. The linking was changed to a static technique, as explained in section 5.1.3, which fixed the problem.

Full results for the survey can be found in the appendix. They will be examined in the following section.

7 Evaluation

The proposed specification proved to be an intense workload for a single developer with a 4-month time frame. In its current state, the software meets most of the specification points, with some key exceptions...

Completed Partially complete Not implemented

- Model a human DJ using artificial intelligence
  - Automatically create a continuous DJ mix, in real time
  - Use the combinational model for creativity to deliver an entertaining mix
  - Integrate Gaussian randomness to encourage novel decision making
  - Allow the user to override the choice of the next track to be played
- Use MIR algorithms to extract information from a library of music
  - Temporal analysis for beat matching tracks
  - Tonal analysis to extract emotion/mood information for the track selection process
  - Segmentation analysis to detect distinct musical sections
  - Exploit genre-specific features to increase reliability
  - Provide a method for manually correcting the temporal data
  - Wait-free system that analyses tracks in the background while playing
- Provide a GUI which is accessible to users with no musical training
  - Make use of HCI best practices to ensure the UI is intuitive
  - Include basic audio controls (start, stop, volume)
  - A skip button to fast-forward the mix
  - Provide coloured waveforms to communicate timbre and audio amplitude
  - High level view of mix trajectory, with controllable direction

The core mixing system is complete, and the software is able to create a continuous DJ mix when given a library of music. Several MIR algorithms are used to model the techniques of a human DJ. The system is fully automatic at present, and most of the incomplete points relate to user control over the mix. This gave an interesting opportunity for testing, where feedback was collected on the fully automatic experience, to find out whether users would like more control.

The other incomplete aspect is the AI modelling of creativity. To recap, the aim is to model human creativity by allowing the system to form novel combinations of familiar ideas. In the case of DJing, these would be stylistic decisions, such as the application of different effects and filters, as well transition styles. See section 2.1.3 for more details.

At present, there are not enough 'ideas' to draw from, making novelty-by-combination a difficult task. No creative effects or filters have been implemented, which would generate a significant variety of parameters to set for every mix. The situation is similar to real life: a human DJ can get much more creative with a complex, feature-packed mixer, than with a very simple one.

7.1 User Experience

Overall, the feedback on AutoDJ was very promising. 100% of users said their experience was 'great' or 'good', while 66.7% said they would use the software again, and pay a small price to do so.

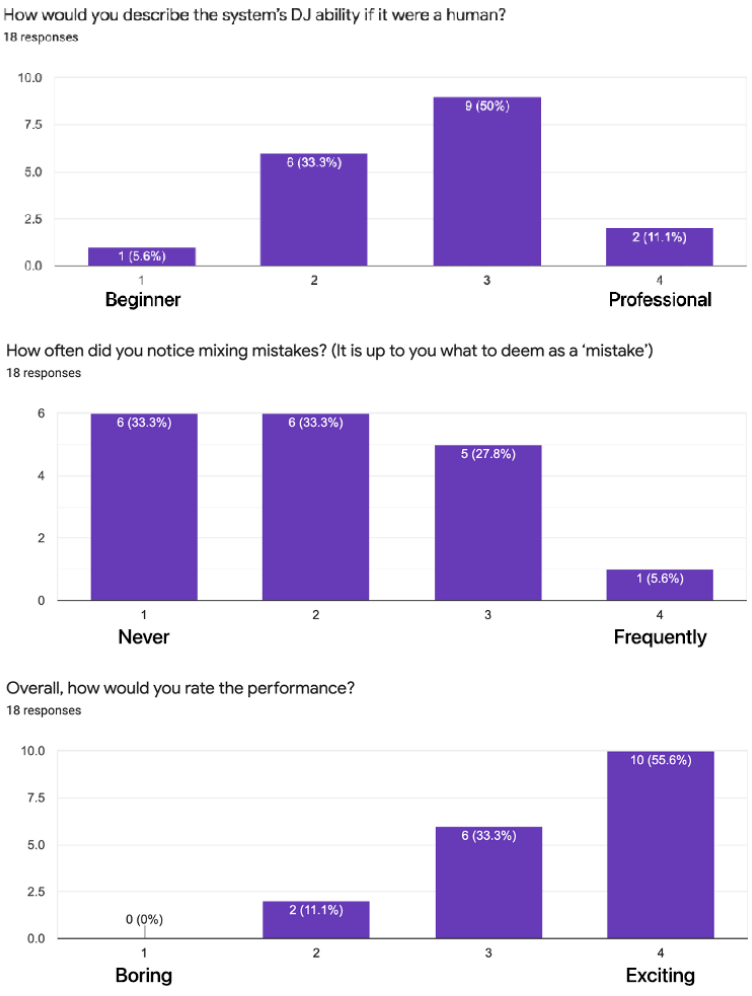


Figure 7.1: Results from the user test survey, showing overall opinions on the DJ performance.

Number scales were used to rate the DJ's mixing ability, some of which are shown in Figure 7.1. Interestingly, while users did notice mistakes fairly often, they still mostly regarded the overall performance as exciting and nearly at professional DJ standard. This suggests that mistakes are perhaps an accepted part of musical performance, and perhaps they make a performance feel more human, as discussed in section 2.1.3.

The data on the users' background experience gave insights on some of the statistics. Those who rated AutoDJ's ability as 'professional' were generally not DJs themselves; and those that were DJs typically gave lower ratings throughout the performance evaluation.

A DJ has to make a trade-off between coherence and novelty in their track choices [100], and in the case of AutoDJ, this is govern by speed at which it traverse the 2D plane of tempo and groove. The coherence of the track choices was rated highly, with 61.6% rating it as 'impressive'.

Given that most users also rated the performance as 'exciting', the results suggest that AutoDJ strikes a good balance between coherence and novelty.

The ratings for the quality of beat-matching and transition timings were lower than the track choice rating. This indicates there is room for improvement in both the temporal analysis pipeline and the mix generation. The lower beat-matching results correlated with users that imported their own music library, rather than just using the test tracks provided. While some users stuck to house music, other imported a diverse range of genre, which supports fears that the temporal analysis pipeline might perform poorly on generalised data.

An option to indicate preference for a smartphone version of the app was provided, but, surprisingly, it was only selected by one user. There are existing solutions for automatic DJing on smartphones - detailed in section 2.2.3 - so perhaps users sense a gap in the market for desktop implementations.

On the subject of desktop hardware, users reported good performance, with 72% rating it 'smooth'. There was no correlation between users that gave lower performance ratings, and those that had less RAM or a low-end CPU. This suggests that lower ratings were specific to the way in which the users treated the software, or their standards of what is 'smooth' performance. Impressively, only 1 out of 18 users experienced a crash in the software.

Users gave some very useful comments and suggestions, which are listed in the appendix. Only half of users rated the navigation of AutoDJ as completely intuitive, and a few of the comments were related to this. The issue seems the be that the purpose of each view is unclear. Given that the user can't influence the track choice, the Library and Direction views might feel out of place. Hopefully, allowing the user to influence track choice, as originally intended, will aid this situation.

One of the most important findings from the user test is that most users would like more control over the mix. 64.7% said they would prefer 'some control', while 29.4% said it depends on the listening situation. Only one user said they prefer this fully automatic experience. Given that 57.1% of users left AutoDJ playing in the background while they did another activity, the system needs the *ability* to mix with full automation, but it could certainly allow the user to get involved to some degree, such as re-ordering chosen tracks, or inserting their own choices into the queue.

7.2 AutoDJ vs Human DJ

A video demonstration of AutoDJ can be viewed here. This reviews the key features of the software, while it mixes a set of six tracks. The software is then compared to the same series of tracks mixed by a human DJ. The set of tracks chosen by AutoDJ happened to have fully-correct temporal analysis results, which biases the comparison slightly, because correct analysis is not always achieved.

When the temporal analysis is correct, it seems that the beat matching by AutoDJ tends to be better - or less noticeable - than the human DJ. Human DJs often listen out for beat-matching error and correct it in retrospect, meaning that listeners can also hear the momentary synchronisation error. Furthermore, a DJ might *forget* to beat-matching tracks, which results

in a scramble to get them in time, as seen at 14:05 in the video. In the case of AutoDJ, the tempos are matched prior to each transition, and stay constant throughout. Having said this, there is sometimes an audible 'clang', where the transient elements in the music sound slightly out of time (see 4:08). This is believed to be due to the framing technique of the time-stretching algorithm, which may place transients slightly out of line.

The modulation of tempo shifts by AutoDJ is generally smoother than that of a human DJ. Notice that at 15:15 in the video, the DJ uses both hands to adjust the tempo of the tracks. There are a multitude of other processes that the DJ needs their hands free for, meaning they tend to only spend a short time on such tempo changes.

While the smooth, linear tempo modulation by AutoDJ is advantageous, the linear-only modulation of other parameters is a major detriment to the creativity of the mix. The human DJ applies a continuous, non-linear adjustment of volume at 9:13, with the intention of highlighting an incoming vocal. Later, at 10:45, they start a track with the volume already set high, and subsequently cut the volume of the previous track. AutoDJ is not capable of such transitions, which means many of its transitions sound and, importantly, *feel* the same. If the artificial DJ had a range of modulation curves to use for any of the available parameters, it might make for a more engaging and exciting listening experience.

The comparison reinforces the appetite for DSP effects in AutoDJ. The human DJ makes extensive use of stereo delay, reverb, and a phaser, which add great interest and variety into the transitions. As discussed in section 5.2.2, an effect placed after the volume fader, known as *post-fader*, can continue long after the track has been stopped. This is technique gives a very spacious effect after the last transition at 16:10.

AutoDJ's lack of awareness of vocals is highlighted by the transition 5:53, where it mixes two singers together. This example is not disastrous, though, because one of the tracks involves vocal 'chopping' where there are no coherent sentences to absorb. The human DJ avoids mixing these vocals together by *looping* the leading track at 14:00, which allows them to keep it playing a section with no vocals, while the new track is mixed in.

While a rather critical evaluation of AutoDJ's mix has been given here, the performance does stand up to the human DJ fairly well. There are clear areas for improvement, and some DJ golden rules are broken, but as stated already: *there are no rules!* It could be argued that coherent high-level performance choices are much more important than individual transitions, and AutoDJ delivers in this respect.

The software was used to set the tone for a COVID-safe gathering outside. It proved very effective in providing a stimulating soundtrack while the DJ was setting up their own equipment. This highlights another possible use case for AutoDJ: filling the gaps before, after, or in between acts at live events. The software could serve as an easy way for venues to keep the audience stimulated during these times. Such venues usually need to report the tracks they play to a music royalty organisation, such as PRS, who handle the payment of artists. It would be convenient for AutoDJ to output a text list of the tracks it plays, so that the venue does not have to do this manually.

Figure 7.2: AutoDJ can provide a soundtrack while a DJ sets up their equipment.

7.3 Research Findings

A set of research questions were set out after the project specification. We can now use our findings from the development and testing of AutoDJ to shed light on these points...

Is there a use case for automatic DJ software, and how much involvement do users want?

There is a strong use case for automatic DJ software. Users often opted to play AutoDJ in the background while they did another task, sometimes listening along with other people. The vast majority of users did want more control than the fully-automatic experience currently available. It is also apparent that AutoDJ would be suitable for intermissions at live events, or social gatherings where a DJ is not present.

Can an AI DJ be entertaining?

The majority of users found the AI DJ very entertaining, and qualitative evaluation found that it can compete with a human DJ in various areas. The current implementation of AutoDJ has a lack of creativity involved, but this is a symptom of the project time constraints, and can be addressed in future versions.

Can expensive MIR algorithms be processed reliably on a range of machines and material?

Most users rated AutoDJ as running smoothly on their machine, but most of their hardware was similar in power. A user test on Windows machines would be more revealing in this respect, as it would offer much greater diversity in hardware. While the analysis algorithms performed well on the custom house music data set, there are signs they fell apart in response to different genres.

Are mistakes and errors a major issue in an AI music performance system?

Testing showed that mistakes are much less of a problem than typically regarded. Users noticed AutoDJ make mixing mistakes fairly often, but still rated the performance highly. Mistakes are also made in the human DJ performance examined in the previous section.

How can we help non-musical users get involved in music performance?

The degree to which users are currently 'involved' with AutoDJ is debatable, since they have no control over the mix. In future versions, however, the user might be able to influence track choice or stylistic mixing decisions. In this respect, AutoDJ shows promise for helping non-musical users get involved in music performance. The interface design and user help material does need improving, though, since a number of users reported confusion about the UI.

7.4 Task List

In response to the discoveries made about AutoDJ, the following set of tasks are suggested. These have been added as issues to the GitHub repository, which means any developer can tackle them and propose a solution to be integrated into the main code-base.

Preserve transient alignment in time-stretching algorithm

The time-stretching algorithm in use, WS-OLA, slices the audio into frames which are then moved in time. This is possibly the cause of the imperfect beat-matching in AutoDJ, which is sometimes present on even when using ground truth analysis data. There are a number of transient-preserving methods for time stretching [103], [17], [42], [30], some of which model/decompose the audio signal into harmonic and percussive component, allowing for pitch and phase preservation, as well as transient preservation. These should be investigated for use in AutoDJ.

Investigate Segmentation Options

Only one track segmentation algorithm was tested with AutoDJ, and the results are not always ideal, with many musical section boundaries being missed. There are several methods available within this one algorithm, listed here, as well as other options, such as Essentia's highly-customisable solution, which should be evaluated against the current configuration.

Creative Parameter Modulation

AutoDJ should have a set of custom value curves for modulating its parameters, to add more interest to the mix. These should be designed with human DJ tendencies in mind, to encourage a 'sensible' mixing style. The system could even generate its own curves in real-time, or allow the user to draw them, as in [36].

Help Page

While the user interface is designed to be intuitive, user testing showed that navigation can be confusing, and the feature set is not always clear. To avoid a reliance on a separate user guide, there should be a help page inside the app. Additional help could also be added through

tooltips.

Equal Loudness

Currently, the loudness of input music is not checked, and there is no level-matching performed during playback. Once users start feeding AutoDJ with their own libraries, this might become an issue, because the volume of consecutive tracks is not guaranteed to be close in level. Essentia provides an equal loudness filter, which could be used to measure the loudness of input audio, and adjust it to meet a reference level.

Waveform Jitter

The waveforms in the Mix view sometimes jump around in a 'jittery' fashion. Extensive effort has been made to optimise the update and drawing pipeline of the waveforms, so performance is unlikely to be the cause of this behaviour. This suggests there is a bug somewhere in the update pipeline, where the waveform receives an incorrect playhead or time-stretch value.

Windows Support

The JUCE framework used to create AutoDJ supports cross-platform development. It should be relatively easy to add support for Windows users, but there are some key considerations:

- **Audio Drivers:** All Macs ship with adequate audio drivers that can handle the audio processing carried out in AutoDJ. Many Windows users have to install custom drivers, such as Asio4All, before they can use advanced audio software. Check that AutoDJ is compatible with these custom drivers.
- **Low-Resolution Screens:** All modern Mac systems are known to have screens with high pixel density, but Windows systems are incredibly diverse in comparison. AutoDJ's user interface should be checked on a range of low-resolution screens, to ensure all elements are sharp enough - especially the thin tracks waveforms.
- **Third-Party Libraries:** The static third party libraries in use are currently built for Mac only. These would need to be compiled separately on Windows *before* building AutoDJ.

Danceability Analysis

The danceability, or 'groove', algorithm is not designed to distinguish between tracks of the same genre. A modification to the algorithm is suggested, which might allow a better classification of similar tracks: "By averaging the DFA exponent function we used an extremely compact and simple representation in the experiments. It should be possible to improve the results by a more sophisticated reduction of the function, or by choosing a less compact representation" [91, p. 8]

8 Conclusion

Overall, this project can be considered a success. The original research aims have been thoroughly investigated, by answering a set of more focused questions. The software development perspective allowed for discoveries to be made in all three research areas, and an entertaining AI DJ was produced in the process.

The iterative development approach helped to prioritise useful work and minimise overhead. It encouraged joint progress in all areas, which actually assisted the development process. For example, the integration of MIR algorithms was made much easier because the UI, audio pipeline and data management were all partly complete. It was also easier to keep the big picture in mind, as the application progressed in a balanced way towards the intended specification.

The incomplete state of the software is believed to be a symptom of an over-ambitious specification, rather than a fault in the development approach. Before adequate analysis accuracy was reached, a lot of time was spent experimenting with different MIR analysis configurations. This meant there was not enough time to work on the creative mixing abilities. It was worth getting the temporal analysis right, though, as the mix falls apart without strong knowledge in this area.

Notably, a novel beat phase correction technique was developed, which greatly increased temporal analysis accuracy within the software.

While the 'AI' mixing brain is relatively basic compared to modern AI standard, the system as a whole brings together a number of advanced techniques which result in a decent emulation of a human DJ mix. As such, the system can be considered moderately intelligent.

Furthermore, as development continues, the performance of AutoDJ can only improve. New MIR techniques are developed every year, which unlock new possibilities in the extraction of musical knowledge. In the future, AutoDJ could even integrate machine learning techniques, which have proven to be very effective in a number of MIR applications.

8.1 Future Work

Development of AutoDJ will continue as an open-source project. The task list suggested in section 7.4 is available on the public GitHub repository, to which anyone can offer contributions. There has already been some interest expressed by other developers, who offered help with adding Windows support.

The results of the tests carried here could be compared to future versions of AutoDJ, as well as other projects. The approach taken here could also act as a framework for future projects, which could even build on some of the audio processing pipelines designed here.

Appendix

A The Reality of User Interfaces

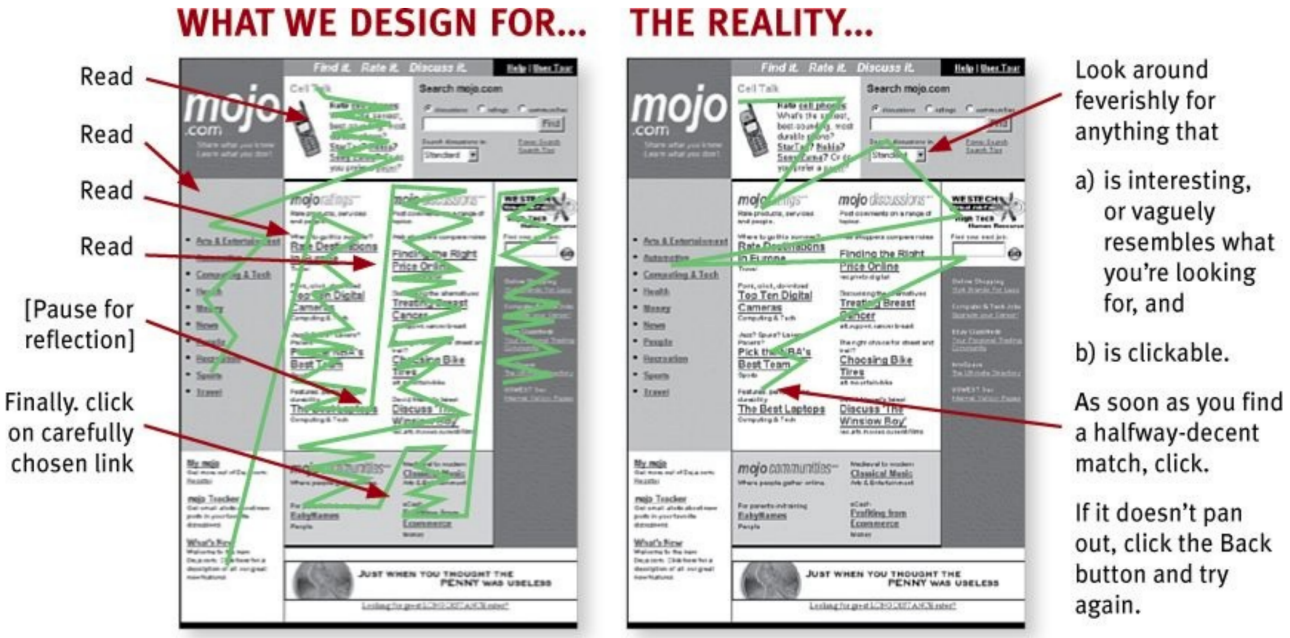


Figure 1: The expectation versus reality of how a user absorbs information [61].

B Human Computer Interfaces Mind Map

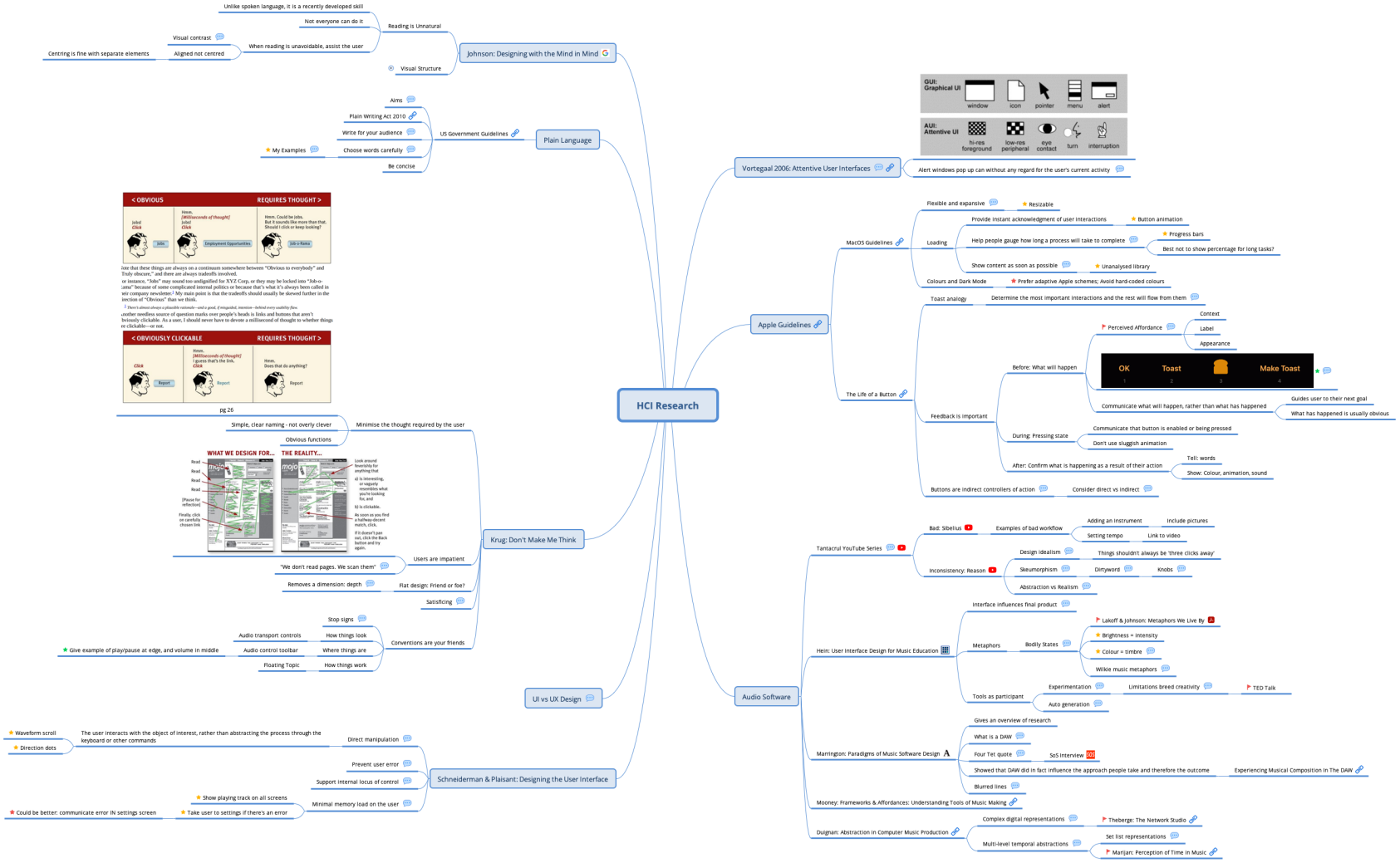
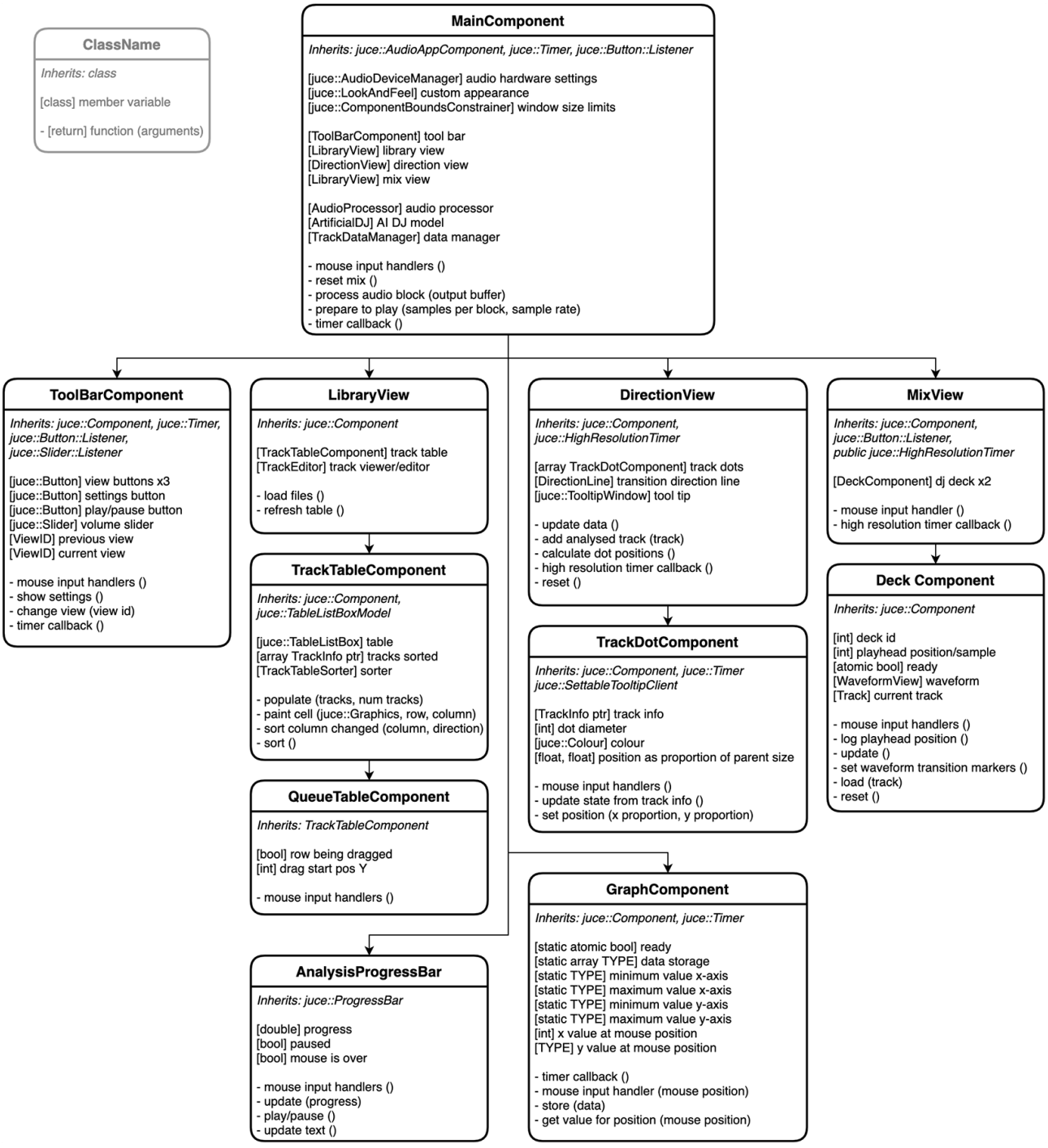
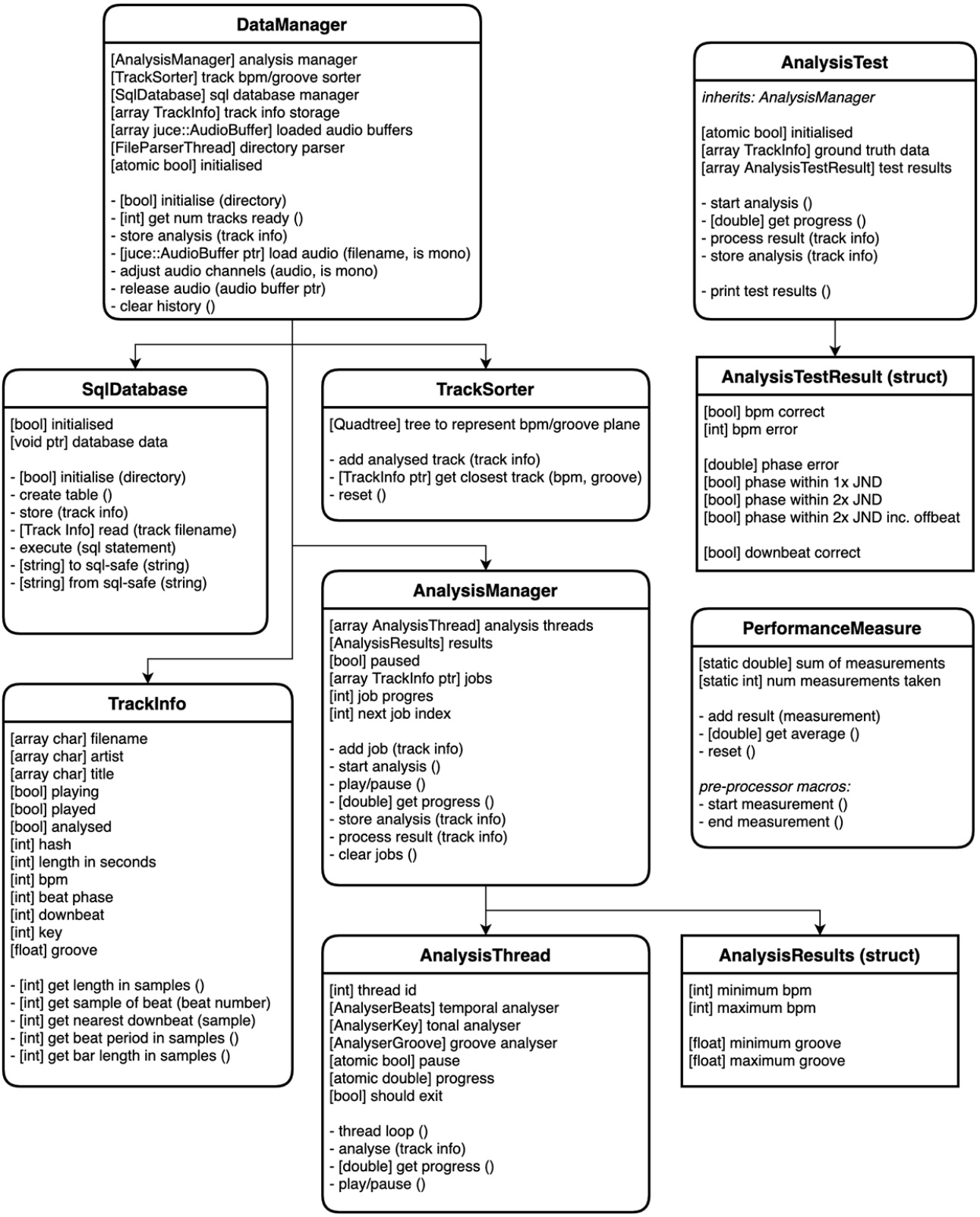
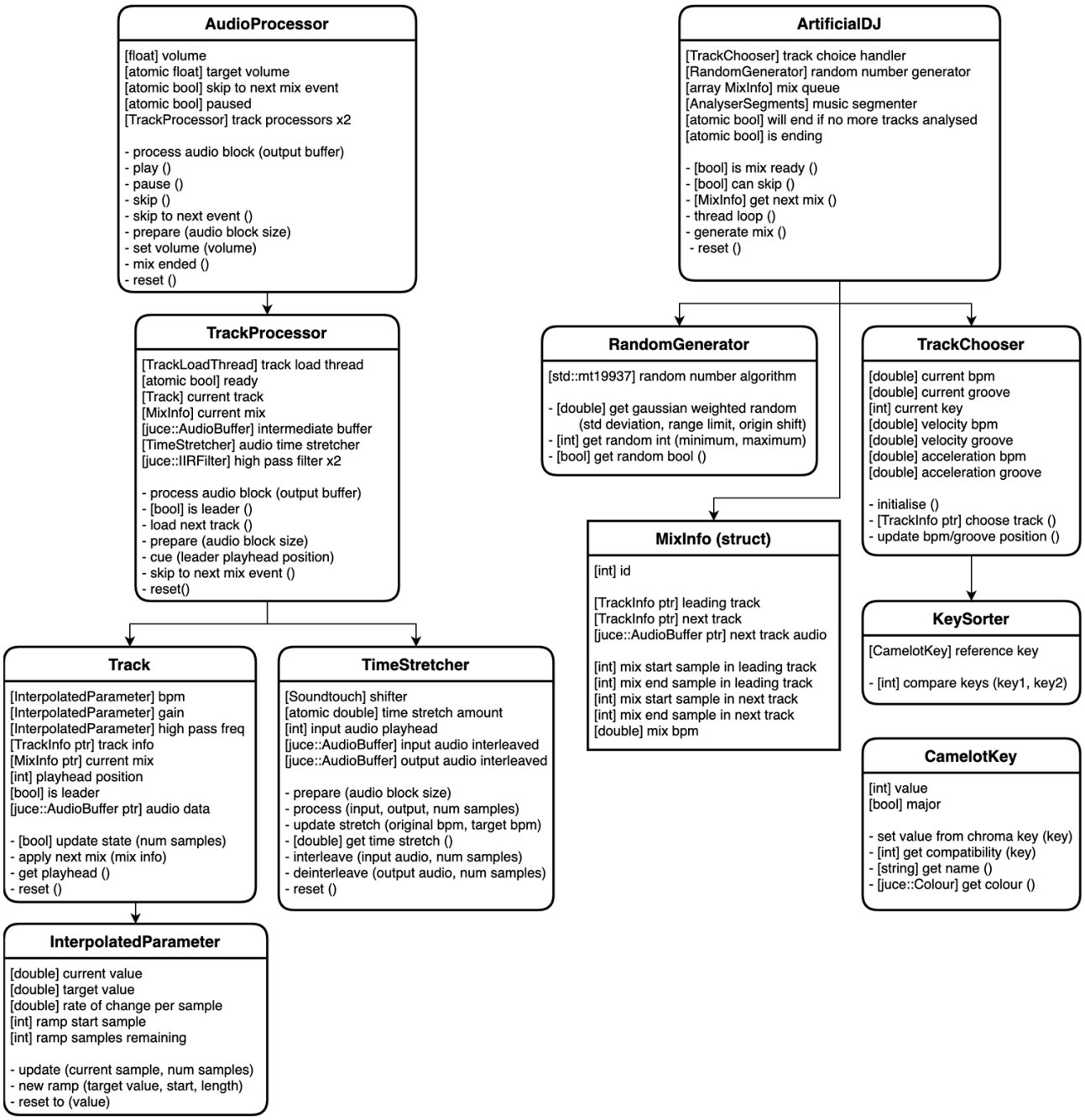


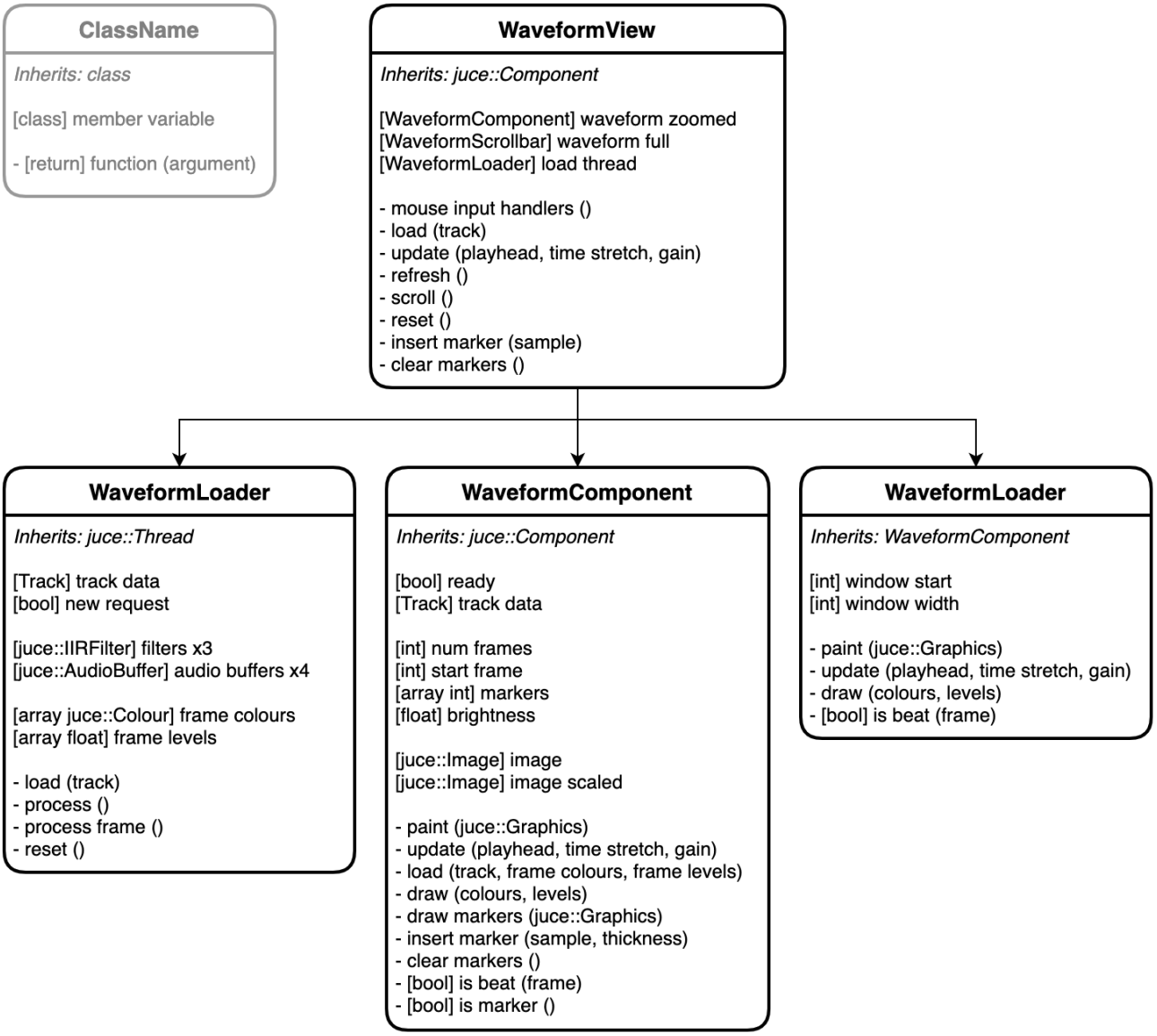
Figure 2: Mind map of research into human computer interfaces, created in XMind.

C AutoDJ Class Diagrams




DataManager
[AnalysisManager] analysis manager
[TrackSorter] track bpm/groove sorter
[SqlDatabase] sql database manager
[array TrackInfo] track info storage
[array juce::AudioBuffer] loaded audio buffers
[FileParserThread] directory parser
[atomic bool] initialised
- [bool] initialise (directory)
- [int] get num tracks ready ()
- store analysis (track info)
- [juce::AudioBuffer ptr] load audio (filename, is mono)
- adjust audio channels (audio, is mono)
- release audio (audio buffer ptr)
- clear history ()
AnalysisTest
inherits: AnalysisManager
[atomic bool] initialised
[array TrackInfo] ground truth data
[array AnalysisTestResult] test results
- start analysis ()
- [double] get progress ()
- process result (track info)
- store analysis (track info)
- print test results ()
SqlDatabase
[bool] initialised
[void ptr] database data
- [bool] initialise (directory)
- create table ()
- store (track info)
- [Track Info] read (track filename)
- execute (sql statement)
- [string] to sql-safe (string)
- [string] from sql-safe (string)
TrackSorter
[Quadtree] tree to represent bpm/groove plane
- add analysed track (track info)
- [TrackInfo ptr] get closest track (bpm, groove)
- reset ()
AnalysisTestResult (struct)
[bool] bpm correct
[int] bpm error
[double] phase error
[bool] phase within 1x JND
[bool] phase within 2x JND
[bool] phase within 2x JND inc. offbeat
[bool] downbeat correct
AnalysisManager
[array AnalysisThread] analysis threads
[AnalysisResults] results
[bool] paused
[array TrackInfo ptr] jobs
[int] job progres
[int] next job index
- add job (track info)
- start analysis ()
- play/pause ()
- [double] get progress ()
- store analysis (track info)
- process result (track info)
- clear jobs ()
PerformanceMeasure
[static double] sum of measurements
[static int] num measurements taken
- add result (measurement)
- [double] get average ()
- reset ()
pre-processor macros:
- start measurement ()
- end measurement ()
TrackInfo
[array char] filename
[array char] artist
[array char] title
[bool] playing
[bool] played
[bool] analysed
[int] hash
[int] length in seconds
[int] bpm
[int] beat phase
[int] downbeat
[int] key
[float] groove
- [int] get length in samples ()
- [int] get sample of beat (beat number)
- [int] get nearest downbeat (sample)
- [int] get beat period in samples ()
- [int] get bar length in samples ()
AnalysisThread
[int] thread id
[AnalyserBeats] temporal analyser
[AnalyserKey] tonal analyser
[AnalyserGroove] groove analyser
[atomic bool] pause
[atomic double] progress
[bool] should exit
- thread loop ()
- analyse (track info)
- [double] get progress ()
- play/pause ()
AnalysisResults (struct)
[int] minimum bpm
[int] maximum bpm
[float] minimum groove
[float] maximum groove


AudioProcessor
[float] volume
[atomic float] target volume
[atomic bool] skip to next mix event
[atomic bool] paused
[TrackProcessor] track processors x2
- process audio block (output buffer)
- play ()
- pause ()
- skip ()
- skip to next event ()
- prepare (audio block size)
- set volume (volume)
- mix ended ()
- reset ()
TrackProcessor
[TrackLoadThread] track load thread
[atomic bool] ready
[Track] current track
[MixInfo] current mix
[juce::AudioBuffer] intermediate buffer
[TimeStretcher] audio time stretcher
[juce::IIRFilter] high pass filter x2
- process audio block (output buffer)
- [bool] is leader ()
- load next track ()
- prepare (audio block size)
- cue (leader playhead position)
- skip to next mix event ()
- reset()
Track
[InterpolatedParameter] bpm
[InterpolatedParameter] gain
[InterpolatedParameter] high pass freq
[TrackInfo ptr] track info
[MixInfo ptr] current mix
[int] playhead position
[bool] is leader
[juce::AudioBuffer ptr] audio data
- [bool] update state (num samples)
- apply next mix (mix info)
- get playhead ()
- reset ()
TimeStretcher
[Soundtouch] shifter
[atomic double] time stretch amount
[int] input audio playhead
[juce::AudioBuffer] input audio interleaved
[juce::AudioBuffer] output audio interleaved
- prepare (audio block size)
- process (input, output, num samples)
- update stretch (original bpm, target bpm)
- [double] get time stretch ()
- interleave (input audio, num samples)
- deinterleave (output audio, num samples)
- reset ()
InterpolatedParameter
[double] current value
[double] target value
[double] rate of change per sample
[int] ramp start sample
[int] ramp samples remaining
- update (current sample, num samples)
- new ramp (target value, start, length)
- reset to (value)
ArtificialDJ
[TrackChooser] track choice handler
[RandomGenerator] random number generator
[array MixInfo] mix queue
[AnalyserSegments] music segmenter
[atomic bool] will end if no more tracks analysed
[atomic bool] is ending
- [bool] is mix ready ()
- [bool] can skip ()
- [MixInfo] get next mix ()
- thread loop ()
- generate mix ()
- reset ()
RandomGenerator
[std::mt19937] random number algorithm
- [double] get gaussian weighted random (std deviation, range limit, origin shift)
- [int] get random int (minimum, maximum)
- [bool] get random bool ()
TrackChooser
[double] current bpm
[double] current groove
[int] current key
[double] velocity bpm
[double] velocity groove
[double] acceleration bpm
[double] acceleration groove
- initialise ()
- [TrackInfo ptr] choose track ()
- update bpm/groove position ()
MixInfo (struct)
[int] id
[TrackInfo ptr] leading track
[TrackInfo ptr] next track
[juce::AudioBuffer ptr] next track audio
[int] mix start sample in leading track
[int] mix end sample in leading track
[int] mix start sample in next track
[int] mix end sample in next track
[double] mix bpm
KeySorter
[CamelotKey] reference key
- [int] compare keys (key1, key2)
CamelotKey
[int] value
[bool] major
- set value from chroma key (key)
- [int] get compatibility (key)
- [string] get name ()
- [juce::Colour] get colour ()


ClassName
Inherits: class
[class] member variable
- [return] function (argument)
WaveformView
Inherits: juce::Component
[WaveformComponent] waveform zoomed
[WaveformScrollbar] waveform full
[WaveformLoader] load thread
- mouse input handlers ()
- load (track)
- update (playhead, time stretch, gain)
- refresh ()
- scroll ()
- reset ()
- insert marker (sample)
- clear markers ()
WaveformLoader
Inherits: juce::Thread
[Track] track data
[bool] new request
[juce::IIRFilter] filters x3
[juce::AudioBuffer] audio buffers x4
[array juce::Colour] frame colours
[array float] frame levels
- load (track)
- process ()
- process frame ()
- reset ()
WaveformComponent
Inherits: juce::Component
[bool] ready
[Track] track data
[int] num frames
[int] start frame
[array int] markers
[float] brightness
[juce::Image] image
[juce::Image] image scaled
- paint (juce::Graphics)
- update (playhead, time stretch, gain)
- load (track, frame colours, frame levels)
- draw (colours, levels)
- draw markers (juce::Graphics)
- insert marker (sample, thickness)
- clear markers ()
- [bool] is beat (frame)
- [bool] is marker ()
WaveformLoader
Inherits: WaveformComponent
[int] window start
[int] window width
- paint (juce::Graphics)
- update (playhead, time stretch, gain)
- draw (colours, levels)
- [bool] is beat (frame)

D Temporal Analysis Results

The test results of the temporal analysis pipeline are shown in the following table. Degara was chosen as the final configuration - see section 5.3 for details.

Algorithm	BPM		Phase Accuracy				Downbeat	Average
	Accuracy (%)	Avg. Error (samples)	1x JND (%)	2x JND (%)	2x JND inc. Off-beat (%)	Avg. Error (samples)	Accuracy (%)	Time (s)
QM-DSP	1.00	0.00	0.40	0.44	0.98	0.28	0.55	3.47
QM-DSP + Lowpass	0.96	1.48	0.65	0.92	0.96	0.04	0.61	3.51
QM-DSP + Lowpass Downbeat	1.00	0.00	0.40	0.44	0.98	0.28	0.55	3.63
MultiFeature	0.96	0.04	0.20	0.41	0.85	0.28	0.50	3.22
MultiFeature + Lowpass	0.90	0.12	0.29	0.56	0.78	0.18	0.32	3.30
MultiFeature + Lowpass Downbeat	0.96	0.04	0.20	0.41	0.85	0.28	0.50	3.31
Degara	1.00	0.00	0.46	0.50	1.00	0.26	0.48	0.98
Degara + Lowpass	0.94	0.74	0.68	1.00	1.00	0.02	0.55	1.10
Degara + Lowpass Downbeat	1.00	0.00	0.46	0.50	1.00	0.26	0.52	1.08
Degara + Phase Correct	1.00	0.00	0.84	1.00	1.00	0.02	0.58	1.37

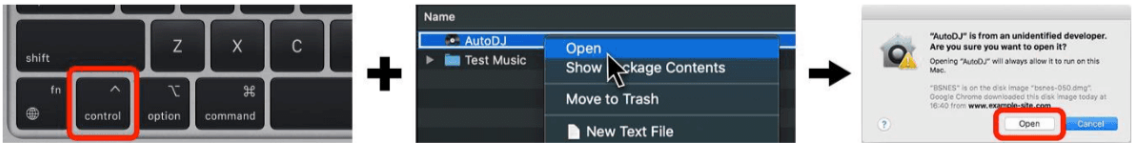
For the Degara configuration, the following downbeat step size values were tested.

Step Size	Downbeat Accuracy
1024	0.72
2048	0.76
4096	0.8
6144	0.76
8192	0.76

E User Guide for AutoDJ Prototype

AutoDJ User Guide

AutoDJ is a fully-automatic DJ system that uses audio analysis to construct a digital understanding of your music library. The software was designed by Alexei Smith as part of a study into '*Music Information Retrieval and AI: Automating Music Performance.*' This is an open-source project, so the code can be viewed on GitHub.



The first time you launch AutoDJ, dismiss the warning message. Then you need to hold *control* and click on the file, then select *Open* in the same message window.

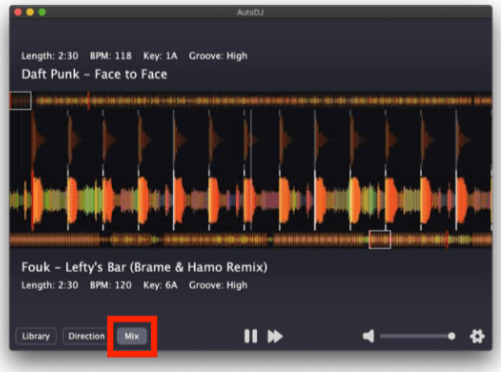
Select a folder with at least 6 tracks in .mp3 or .wav format. The provided Test Music folder contains 50 clips you can use. You can import any kind of dance music, but keep in mind that AutoDJ is optimised for house.

The **Library** view shows a table of all valid tracks in the selected folder. As they are analysed, more information will appear, including the tempo (BPM), key and groove. Groove is an experimental parameter related to the 'danceability' of the music.

To start the automatic DJ mix, simply press play.

The **Direction** view distributes music based on its tempo and groove. Here, you can watch the path that AutoDJ takes through your library. It tries to create pleasing mixes by combining tracks that are similar in tempo, groove and key.

Hover over a point to see the track title.



The **Mix** view shows the tracks currently being mixed, where brightness corresponds to volume.

The red markers show the start and end points for each mix. AutoDJ tries to find sensible mixing points based on musical sections.

Figure 3: The simple User Guide which was distributed with the AutoDJ user testing package.

F User Survey Comments

"A tutorial or guide in using the application would be good. UI is simple, clear and effective, but I was initially lost."

"Data visualisation map was very interesting, would like some interaction or control on the 'Direction' page."

"I think the software needs a queue and skip track function. You expect it to have one as part of the browser, this creates some confusion with the UI. Generally really good though! Awesome prototype of the DJing algorithm. I have DJay Pro 2 AI mixer and this is comparable. I like the fact it chooses similar tracks to mix really effectively."

"In terms of adding to the mixing algorithm. Maybe a rapid fade out with reverb tail mix type for tracks which are very different."

"Loved using it! Great Project!"

"Great idea and very impressive! Would be cool if you could see the route of mixes in direction view. Maybe having faded lines so you can remember what tracks blended nicely with others."

"Needs a function to allow users to fast forward/rewind within a track (seek). Beatmatching algorithm needs improving, some problems with this when using two songs of different bpm."

"The app is amazing! I'd just add a previous song function. Also, it would be great to be able to hoover over the track to choose a specific point in the track."

"Suggestions: FX you can control in real time. Control over sense of direction. iOS Compatible"

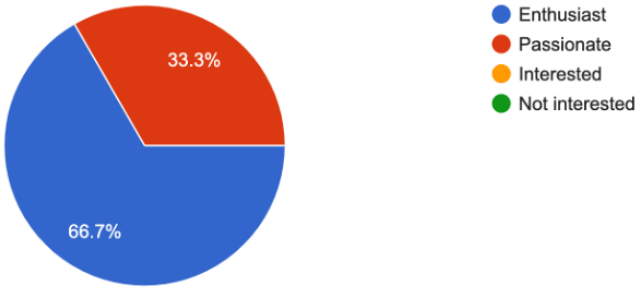
"I think the idea is really cool, the fact that you have to change windows to get the different views staggers work flow, also when on the library window it should be clearer that song and where in the song you are, its almost like the library window and mix window should be one window. bare in mind I am not a dj and know very little about how to correctly mix music. I mainly create and use logic so I think I found it less native than someone who had spent a lot of time mixing."

"Overall a very cool idea, would be cool to use for parties if no-one wanted to have to dj."

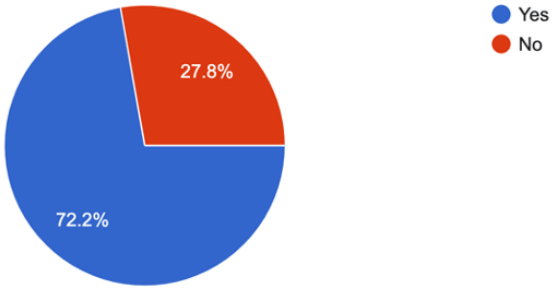
"Initially when I started using the software with bluetooth headphones it would not output music through the headphones. It only outputted music through the Macs speakers. When I changed the sample rate in order to output the sound through my headphone, the software generated an error message and it would not play the music. In ordered to rectify this I discounted my headphones and played AutoDJ through my computers speakers. There was option in the program to change sample rate and tempo in the software, this may have allowed me to play music through my headphones. However, my limited musical knowledge meant I was not able to rectify the issue. It may also be helpful in the settings to include key to explain colours in the mix section (or if not in the help document). This would be useful as I would like to learn more about the music as I use the software. A Possible rewind ('pull up') function would be a great additional feature."

G User Survey Results

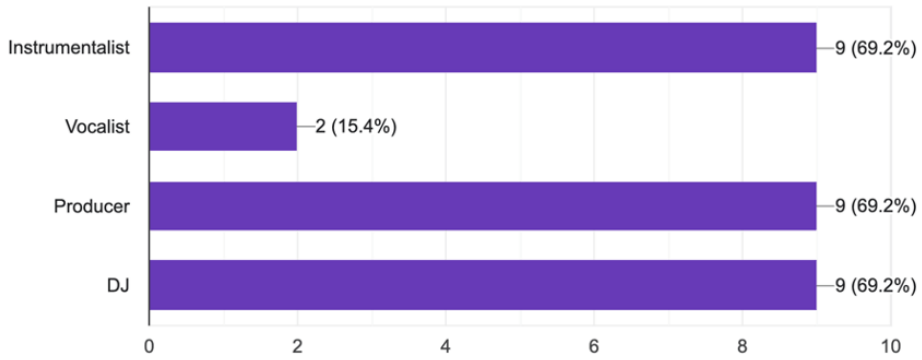
How would you describe your interest in music?
18 responses



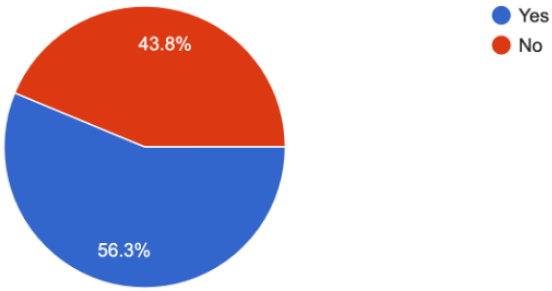
Do you make or perform music?
18 responses



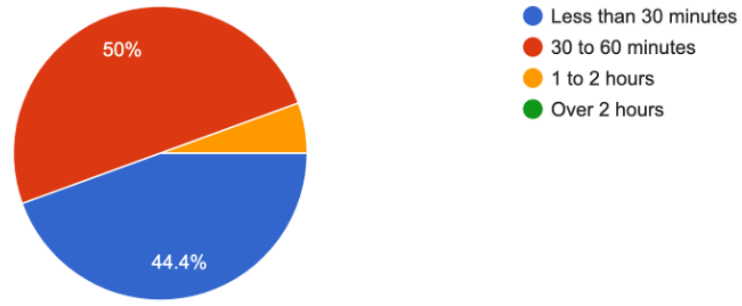
If so, tick all that apply...
13 responses



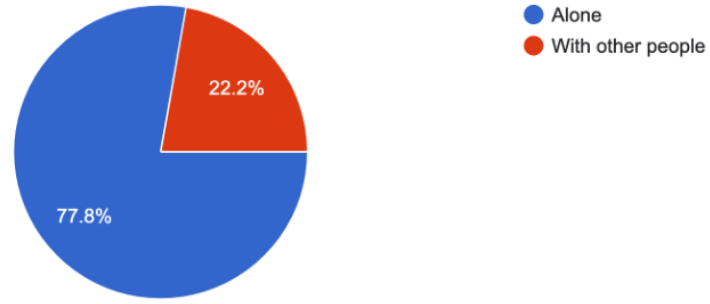
Do you have any software development experience? (Not required for this test!)
16 responses



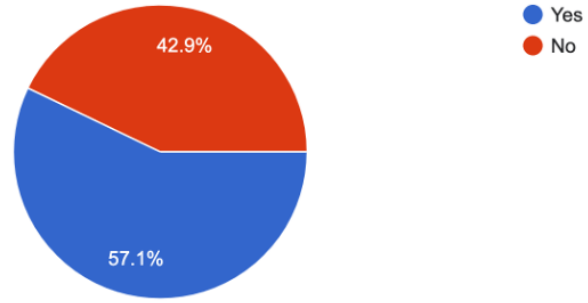
How long did you spend listening to AutoDJ?
18 responses



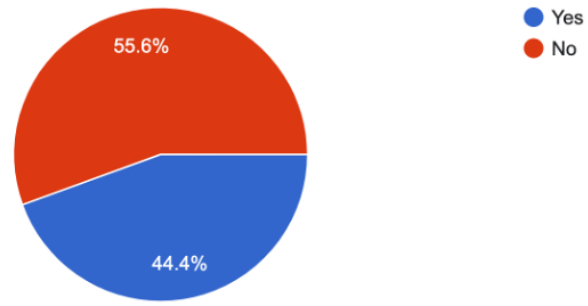
Did you listen alone or with others?
18 responses



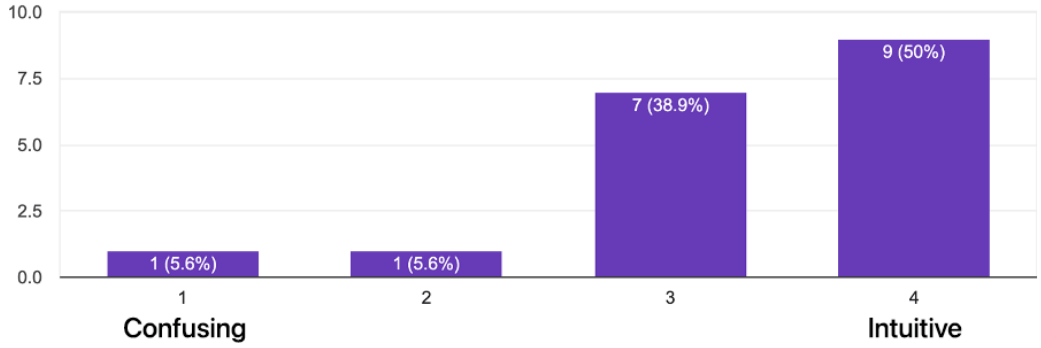
Did you leave AutoDJ playing in the background while you did another activity?
14 responses



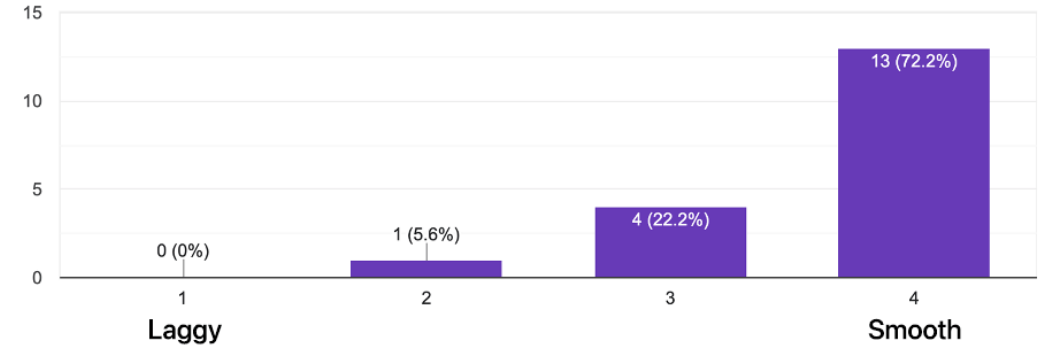
Did you import music from your own library?
18 responses



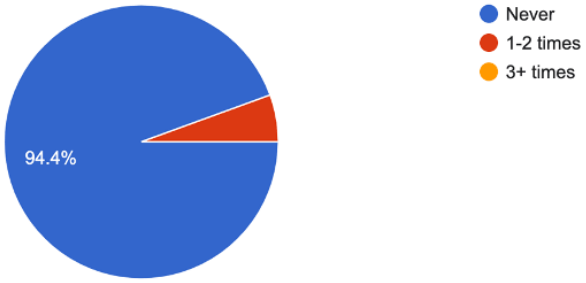
How difficult was it to navigate and use AutoDJ?
18 responses



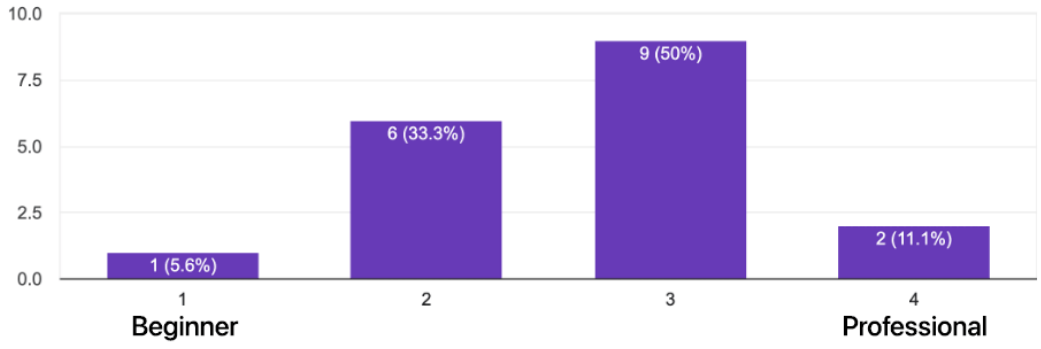
How would you describe the speed/performance of the software?
18 responses



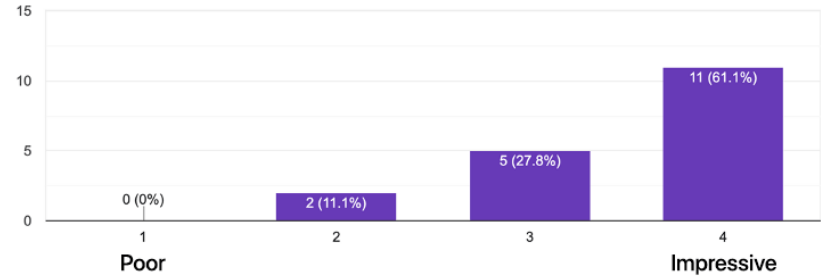
Did the software crash?
18 responses



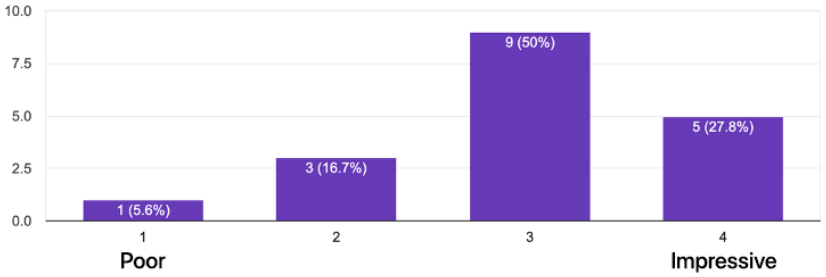
How would you describe the system's DJ ability if it were a human?
18 responses



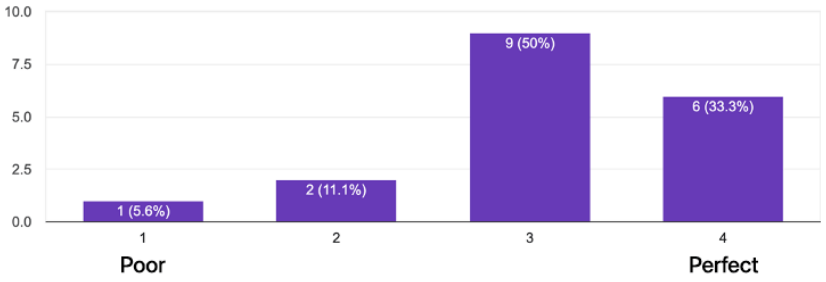
Rate the system's ability to deliver a coherent mix. (Choosing which tracks to mix together)
18 responses



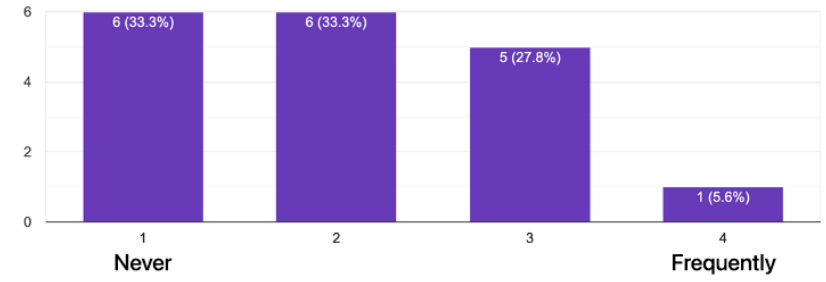
Rate the system's choice of mix timing. (The sections of the tracks that were mixed together)
18 responses



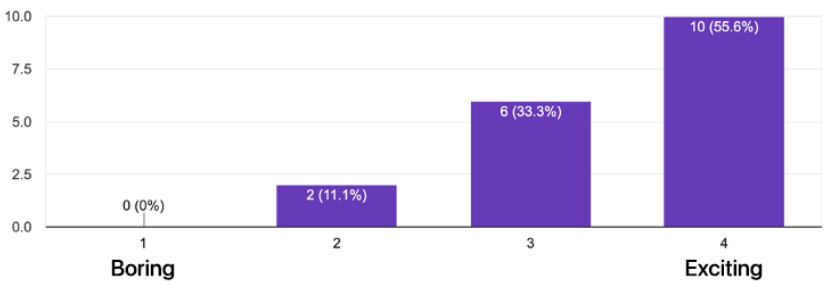
Rate the system's beat-matching. (Synchronisation of tracks)
18 responses



How often did you notice mixing mistakes? (It is up to you what to deem as a 'mistake')
18 responses

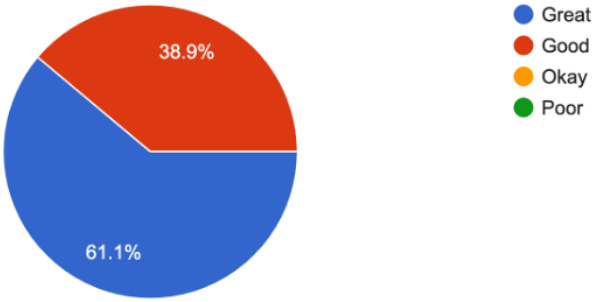


Overall, how would you rate the performance?
18 responses



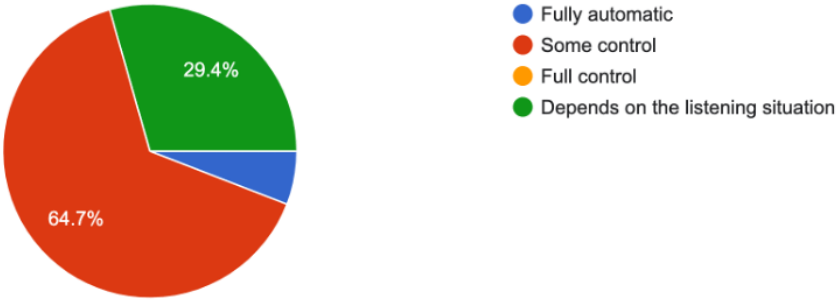
How was the overall experience of using AutoDJ?

18 responses



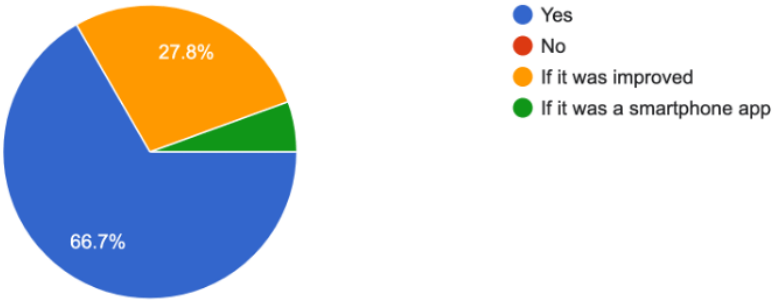
Did you enjoy the fully-automatic experience, or would you prefer more control?

17 responses



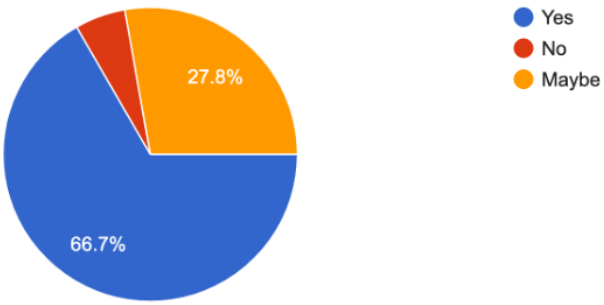
Would you use AutoDJ again?

18 responses



If so, would you pay a small price for it?

18 responses



References

[1] A. Williamon et al. "Creativity, originality, and value in music performance". In: *Musical Creativity: Multidisciplinary Research in Theory and Practice*. Ed. by I. Deliège and G. Wiggins. London: Psychology Press, Aug. 2006, pp. 161–180. ISBN: 978-1841695082.

[2] D. Bogdanov et al. "ESSENTIA: an Audio Analysis Library for Music Information Retrieval". In: *Int. Society for Music Information Retrieval Conference*. Nov. 2013.

[3] F. Gouyon et al. "An experimental comparison of audio tempo induction algorithms". In: *IEEE Transactions on Speech and Audio Processing* 14 (5 Oct. 2006), pp. 1832–1844. DOI: 10.1109/TSA.2005.858509.

[4] H. Jennings et al. "Variance fluctuations in nonstationary time series: A comparative study of music genres". In: *Physica A: Statistical Mechanics and its Applications* 336 (Mar. 2004), pp. 585–594. DOI: 10.1016/j.physa.2003.12.049.

[5] P. Knees et al. "Two Data Sets for Tempo Estimation and Key Detection in Electronic Dance Music Annotated from User Corrections". In: *Proc. of 16th Int. Society for Music Information Retrieval Conference*. 2015, pp. 364–370.

[6] S. A. Mehr et al. "Universality and diversity in human song". In: *Science* 366 (6468 Nov. 2019). DOI: 10.1126/science.aax0868.

[7] S. Böck et al. "madmom: A New Python Audio and Music Signal Processing Library". In: *Proc. of 24th ACM Int. Conference on Multimedia*. Oct. 2016. DOI: 10.1145/2964284.2973795.

[8] T. Andersen. "Interaction with Sound and Pre-Recorded Music: Novel Interfaces and Use Patterns". PhD thesis. University of Copenhagen, 2005.

[9] T. Andersen. "Mixxx: Towards Novel DJ Interfaces". In: *Proc. of New Interfaces for Musical Expression Conference*. May 2003.

[10] Rockwell Anyoha. *The History of Artificial Intelligence*. Aug. 2017. URL: https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/.

[11] Apple. *Human Interface Guidelines: MacOS*. 2021. URL: https://developer.apple.com/design/human-interface-guidelines/macos/overview/themes/.

[12] Apple. *WWDC: The Life of a Button*. June 2018. URL: https://developer.apple.com/videos/play/wwdc2018/804/.

[13] ARRA. *Live performance #1. Roland TR-8, Elektron Digitakt, DSI Mopho, Waldorf Pulse 2 and Korg Minilogue*. Jan. 2018. URL: https://www.youtube.com/watch?v=2YxWMdBPrA8.

[14] M. Boden. "Creativity and artificial intelligence". In: *Artificial Intelligence* 103 (1-2 Aug. 1998). DOI: 10.1016/S0004-3702(98)00055-1.

[15] M. Boden. *The Creative Mind: Myths and Mechanisms*. Routledge, 1991. ISBN: 9780415314534.

[16] D. Bouckenhove. "Automatisch Mixen van Muzieknummers op Basis van Tempo, Zang, Energie en Akkoordinformatie". MA thesis. Universiteit Gent, 2007.

[17] M. Davies C. Duxbury and M. Sandler. "Improved time-scaling of musical audio using phase locking at transients". In: *Proc. of Audio Engineering Society Convention*. May 2002.

[18] S. Buldyrev C. Peng and S. Havlin. "Mosaic organization of DNA nucleotides". In: *Physical review. E, Statistical physics, plasmas, fluids, and related interdisciplinary topics* 49 (2 Mar. 1994). DOI: 10.1103/PhysRevE.49.1685.

[19] Christie's. *Is artificial intelligence set to become art's next medium?* Dec. 2018. URL: https://www.christies.com/features/A-collaboration-between-two-artists-one-human-one-a-machine-9332-1.aspx.

[20] D. Cliff. *Hang the DJ: Automatic Sequencing and Seamless Mixing of Dance-Music Tracks*. Bristol, July 2000.

[21] Artificial Intelligence Committee. *AI in the UK: ready, willing and able?* 2018. URL: https://publications.parliament.uk/pa/ld201719/ldselect/ldai/100/10004.htm.

[22] R. Dannenberg. "Dynamic Programming for Interactive Music Systems". In: *Readings in Music and Artificial Intelligence*. Ed. by M. Reck. London: Routledge, 2000, pp. 189–206. ISBN: 9789057550942.

[23] M. Davies and M. Plumbley. *A spectral difference approach to downbeat extraction in musical audio*. Mar. 2006. URL: https://ieeexplore.ieee.org/document/7071189.

[24] M. Davies and M. Plumbley. "Context-Dependent Beat Tracking of Musical Audio". In: *IEEE Transactions on Audio, Speech and Language Processing* 15.3 (2007), pp. 1009–1020.

[25] Oxford Dictionary. *Definition of artificial intelligence*. URL: https://www.lexico.com/definition/artificial_intelligence.

[26] Oxford Dictionary. *Definition of timbre*. URL: https://www.lexico.com/definition/timbre.

[27] S. Dixon. "Automatic Extraction of Tempo and Beat From Expressive Performances". In: *Journal of New Music Research* 30 (1 Aug. 2001), pp. 95–114. DOI: 10.1076/jnmr.30.1.39.7119.

[28] S. Dixon. "Evaluation of the Audio Beat Tracking System BeatRoot". In: *Journal of New Music Research* 36 (1 Mar. 2007), pp. 39–50. DOI: 10.1080/09298210701653310.

[29] S. Dixon. "Onset detection revisited". In: *Proc. of 9th international conference on digital audio effects*. Sept. 2006, pp. 133–137.

[30] J. Driedger and M. Müller. "A Review of Time-Scale Modification of Music Signals". In: *Applied Sciences* 6 (2 Feb. 2016). DOI: 10.3390/app6020057.

[31] Organisation for Economic Co-operation and Development. *Artificial Intelligence in Society*. Paris: OECD Publishing, 2019. ISBN: 978-92-64-58254-5.

[32] L. Thompson F. Thalmann and M. Sandler. "A User-Adaptive Automated DJ Web App with Object-Based Audio and Crowd-Sourced Decision Trees". In: *Proc. of Web Audio Conference*. Sept. 2018.

[33] L. Figes. *Unfinished: why are we drawn to imperfection?* Aug. 2019. URL: https://artuk.org/discover/stories/unfinished-why-are-we-drawn-to-imperfection.

[34] M. Fisher. *Something in the Air: Radio, Rock, and the Revolution That Shaped a Generation*. Random House, Jan. 2007.

[35] R. Fjelland. "Why general artificial intelligence will not be realized". In: *Humanities and Social Sciences Communications* 7 (1 June 2020). DOI: 10.1057/s41599-020-0494-4.

[36] M. Fraser. *An Automatic Software DJ*. Sheffield, 2002.

[37] A. Friberg and J. Sundberg. "Time discrimination in a monotonic, isochronous sequence". In: *The Journal of the Acoustical Society of America* 98 (5 Nov. 1995). DOI: 10.1121/1.413218.

[38] G. Essl G. Tzanetakis and P. Cook. "Human perception and computer extraction of musical beat strength". In: *Proc. of 5th Int. Conference on Digital Audio Effects*. Sept. 2002.

[39] M. Goto and Y. Muraoka. "A beat tracking system for acoustic signals of music". In: *Proc. of 2nd ACM international conference on Multimedia*. Oct. 1994, pp. 365–372. DOI: 10.1145/192593.192700.

[40] M. Goto and Y. Muraoka. "A Real-Time Beat Tracking System for Audio Signals". In: *Proc. of 1995 Int. Computer Music Conference*. Sept. 1995, pp. 171–174.

[41] United States Government. *Public Law 111 - 274 - Plain Writing Act of 2010*. Oct. 2010. URL: https://www.govinfo.gov/app/details/PLAW-111publ274.

[42] S. Grofit and Y. Lavner. "Time-Scale Modification of Audio Signals Using Enhanced WSOLA With Management of Transients". In: *IEEE Transactions on Audio, Speech, and Language Processing* 16 (1 Feb. 2008). DOI: 10.1109/TASL.2007.909444.

[43] P. Grosche and M. Müller. "Extracting Predominant Local Pulse Information From Music Recordings". In: *IEEE Transactions on Speech and Audio Processing* 19 (6 Sept. 2011), pp. 1688–1701. DOI: 10.1109/TASL.2010.2096216.

[44] L. Gugerty. "Newell and Simon's Logic Theorist: Historical Background and Impact on Cognitive Modeling". In: *Proc. Hum Factors Ergon Soc Annu Meet* 50 (9 Oct. 2006). DOI: 10.1177/154193120605000904.

[45] K. Hoashi H. Ishizaki and Y. Takishima. "Music Paste: Concatenating Music Clips based on Chroma and Rhythm Features". In: *Proc. of 10th Int. Society for Music Information Retrieval Conference*. Jan. 2009, pp. 135–140.

[46] M. Tien H. Lin Y. Lin and J. Wu. "Music Paste: Concatenating Music Clips based on Chroma and Rhythm Features". In: *Proc. of 10th Int. Society for Music Information Retrieval Conference*. Jan. 2009.

[47] S. Hainsworth. "Techniques for the Automated Analysis of Musical Audio". PhD thesis. University of Cambridge, 2004.

[48] Ethan Hein. *User Interface Design for Music Learning Software*. Feb. 2013. URL: https://www.ethanhein.com/wp/2013/user-interface-design-for-music-learning-software/.

[49] S. Inglis. *Four Tet*. July 2003. URL: https://www.soundonsound.com/people/four-tet.

[50] M. Müller J. Driedger and S. Ewert. "Improving Time-Scale Modification of Music Signals Using Harmonic-Percussive Separation". In: *IEEE Signal Processing Letters* 21 (1 Jan. 2014). DOI: 10.1109/LSP.2013.2294023.

[51] M. Davies J. Hockman and I. Fujinaga. "One in the jungle: Downbeat detection in hardcore, jungle, and drum and bass". In: *Proc. of 13th Int. Society for Music Information Retrieval Conference*. Jan. 2012, pp. 169–174.

[52] F. Gouyon J. Oliveira M. Davies and L. Reis. "Beat Tracking for Multiple Applications: A Multi-Agent System Architecture With State Recovery". In: *IEEE Transactions on Speech and Audio Processing* 20 (10 Dec. 2012), pp. 2696–2706. DOI: 10.1109/TASL.2012.2210878.

[53] L. Martins J. Oliveira F. Gouyon and L. Reis. "IBT: A Real-time Tempo and Beat Tracking System". In: *Proc. of 11th Int. Society for Music Information Retrieval Conference*. Jan. 2010, pp. 291–296.

[54] M. Davies J. Zapata and E. Gómez. "Multi-Feature Beat Tracking". In: *IEEE Transactions on Audio, Speech, and Language Processing* 22 (4 Apr. 2014). DOI: 10.1109/TASLP.2014.2305252.

[55] T. Jajal. *Distinguishing between Narrow AI, General AI and Super AI*. May 2018. URL: https://medium.com/mapping-out-2050/distinguishing-between-narrow-ai-general-ai-and-super-ai-a4bc44172e22.

[56] T. Jehan. "Creating Music by Listening". PhD thesis. Massachusetts Institute of Technology, 2005.

[57] J. Johnson. *Designing with the Mind in Mind*. Boston: Morgan Kaufmann, 2014. ISBN: 978-0-12-407914-4.

[58] J. Kahn. *It's Alive!* Jan. 2002. URL: https://www.wired.com/2002/03/everywhere/.

[59] Mixed in Key. *Official Harmonic Mixing guide*. URL: https://mixedinkey.com/harmonic-mixing-guide/.

[60] J. Kimble. *Lifting the Fog of Legalese*. Durham: Carolina Academic Press, 2006. ISBN: 1594602123.

[61] S. Krug. *Don't Make Me Think, Revisited*. New York: Pearson Education, Jan. 2014. ISBN: 9780321965516.

[62] J. Kursell. "Experiments on Tone Color in Music and Acoustics: Helmholtz, Schoenberg, and Klangfarbenmelodie". In: *Osiris* 28 (1 Jan. 2013), pp. 191–211. DOI: 10.1086/671377.

[63] J. Laroche. "Autocorrelation method for high-quality time/pitch-scaling". In: *Proc. of IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. Oct. 1993. DOI: 10.1109/ASPAA.1993.379979.

[64] J. Laroche. "Estimating tempo, swing and beat locations in audio recordings". In: *Proc. of 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics*. Oct. 2001, pp. 135–138. DOI: 10.1109/ASPAA.2001.969561.

[65] J. Laroche and M. Dolson. "Improved phase vocoder time-scale modification of audio". In: *IEEE Transactions on Speech and Audio Processing* 7 (3 May 1999). DOI: 10.1109/89.759041.

[66] M. Levy and M. Sandler. "Structural Segmentation of Musical Audio by Constrained Clustering". In: *IEEE Transactions on Speech and Audio Processing* 16 (2 Feb. 2008). DOI: 10.1109/TASL.2007.910781.

[67] J. Noble M. Duignan and R. Biddle. "Abstraction and Activity in Computer-Mediated Music Production". In: *Computer Music Journal* 34 (4 Dec. 2010), pp. 22–33. DOI: 10.1162/COMJ_a_00023.

[68] R. Mantaras and J. Arcos. "AI and Music: From Composition to Expressive Performance". In: *AI Magazine* 23 (3 2002), pp. 43–57. DOI: 10.1609/aimag.v23i3.1656.

[69] M. Marijan. "The Perception and Organization of Time in Music". In: *Accelerando Belgrade Journal of Music and Dance* 3 (4 Feb. 2018).

[70] M. Marrington. "Experiencing Musical Composition In The DAW: The Software Interface As Mediator Of The Musical Idea". In: *Journal on the Art of Record Production* 5 (July 2011).

[71] M. Marrington. "Paradigms of Music Software Interface Design andMusical Creativity". In: *Innovation in Music II*. Ed. by J. L. Patersonand R. Hepworth-Sawyer J. Hodgson and R. Toulson. Shoreham-by-sea: Future Technology Press, 2016, pp. 52–63. ISBN: 1911108042.

[72] K. May. *Can limitations make you more creative? A Q&A with artist Phil Hansen*. Mar. 2013. URL: https://blog.ted.com/can-limitations-make-you-more-creative-a-qa-with-artist-phil-hansen/.

[73] Bertrand Meyer. *Obituary: John McCarthy*. Oct. 2011. URL: https://cacm.acm.org/blogs/blog-cacm/138907-john-mccarthy/fulltext.

[74] J. Mooney. "Frameworks and affordances: Understanding the tools of music-making". In: *Journal of Music Technology and Education* 3 (2-3 Apr. 2011), pp. 141–154. DOI: 10.1386/jmte.3.2-3.141_1.

[75] E. Moulines and F. Charpentier. "Pitch-synchronous waveform processing techniques for text-to-speech synthesis using diphones". In: *IEEE Transactions on Speech and Audio Processing* 9 (5-6 Dec. 1990). DOI: 10.1016/0167-6393(90)90021-Z.

[76] A. Pena N. Degara Quintela E. Rúa and S. Torres-Guijarro. "Reliability-Informed Beat Tracking of Musical Signals". In: *IEEE Transactions on Audio, Speech, and Language Processing* 20 (1 Jan. 2012). DOI: 10.1109/TASL.2011.2160854.

[77] BBC News. *Google AI defeats human Go champion*. May 2017. URL: https://www.bbc.co.uk/news/technology-40042581.

[78] Jakob Nielsen. *Why You Only Need to Test with 5 Users*. Mar. 2000. URL: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

[79] K. Noland and M. Sandler. "Signal Processing Parameters for Tonality Estimation". In: *Journal of the Audio Engineering Society* (May 2007).

[80] D. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. New York: Basic Books, 2013. ISBN: 9780465050659.

[81] G. Percival and G. Tzanetakis. "Streamlined Tempo Estimation Based on Autocorrelation and Cross-correlation With Pulses". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 22 (12 Dec. 2014), pp. 1765–1776. DOI: 10.1109/TASLP.2014.2348916.

[82] Pioneer. *DJM-900NXS2: 4-channel professional DJ mixer*. URL: https://www.pioneerdj.com/en-gb/product/mixer/djm-900nxs2/black/overview/.

[83] M. Roser and E. Ortiz-Ospina. "Literacy". In: *Our World in Data* (2016). https://ourworldindata.org

[84] S. Roucous and A. Wilgus. "High quality time-scale modification for speech". In: *IEEE Int. Conference on Acoustics, Speech, and Signal Processing*. 1985.

[85] R. Rowe. "Machine Listening and Composing with Cypher". In: *Computer Music Journal* 16 (1 1992). DOI: 10.2307/3680494.

[86] E. Scheirer. "Tempo and beat analysis of acoustic musical signals". In: *Journal of the Acoustical Society of America* 103 (1 Sept. 1997), pp. 588–601. DOI: 10.1121/1.421129.

[87] B. Schneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. New York: Pearson, 2005. ISBN: 9780321197863.

[88] D. Schwarz and D. Fourer. "Methods and Datasets for DJ-Mix Reverse Engineering". In: *Proc. of 14th Int. Symposium on Perception, Representations, Image, Sound, Music*. Mar. 2021, pp. 31–47.

[89] Serato. *Serato DJ - Getting started with the Club Kit*. Apr. 2015. URL: https://www.youtube.com/watch?v=rjG-5Paqoas.

[90] Julian Storer. *JUCE Forum: Which architecture serves as a model for JUCE GUI code?* 2019. URL: https://forum.juce.com/t/which-architecture-serves-as-a-model-for-juce-gui-code/32196.

[91] S. Streich and P. Herrera. "Detrended Fluctuation Analysis of Music Signals: Danceability Estimation and further Semantic Characterization". In: *Journal of the Audio Engineering Society* (2005).

[92] H. Doi T. Hirai and S. Morishima. "MusicMixer: computer-aided DJ system based on an automatic song mixing". In: *Proc. of 12th Int. Conference on Advances in Computer Entertainment Technology*. 41. Nov. 2015, pp. 1–5.

[93] Tantacrul. *Music Software & Bad Interface Design: Avid's Sibelius*. Nov. 2018. URL: https://www.youtube.com/watch?v=dKx1wnXClcI.

[94] Tantacrul. *Music Software & Interface Design: Propellerhead's Reason*. July 2018. URL: https://www.youtube.com/watch?v=7PFRyONURSo.

[95] P. Theberge. "The Network Studio: Historical and Technological Paths to a New Ideal in Music Making". In: *Social Studies of Science* 34 (5 Oct. 2004), pp. 759–781. DOI: 10.1177/0306312704047173.

[96] A. Turing. "Computing Machinery and Intelligence". In: *Mind* 51 (236 Sept. 1950). DOI: 10.1093/mind/LIX.236.433.

[97] G. Tzanetakis and P. Cook. "Musical Genre Classification of Audio Signals". In: *IEEE Transactions on Speech and Audio Processing* 10 (5 Aug. 2002), pp. 293–302. DOI: 10.1109/TSA.2002.800560.

[98] Regents of the University of Michigan. *Less Than Perfect: Deliberate Imperfection*. 2017. URL: https://exhibitions.kelsey.lsa.umich.edu/less-than-perfect/deliberate.php.

[99] J. Serrà V. Akkermans and P. Herrera. "Shape-based spectral contrast descriptor". In: *Proc. of 6th Sound and Music Computing Conference*. July 2009, pp. 143–148.

[100] L. Veire and T. Bie. "From raw audio to a seamless mix: creating an automated DJ system for Drum and Bass". In: *EURASIP J. Audio Speech Music Process.* (Sept. 2018). DOI: 10.1186/s13636-018-0134-8.

[101] L. Veire and T. Bie. "From raw audio to a seamless mix: creating an automated DJ system for Drum and Bass". In: *EURASIP Journal on Audio, Speech, and Music Processing* 13 (Sept. 2018).

[102] W. Verhelst and M. Roelands. "An overlap-add technique based on waveform similarity (WSOLA) for high quality time-scale modification of speech". In: *IEEE Int. Conference on Acoustics, Speech, and Signal Processing*. 1993.

[103] T. Verma and T. Meng. "Time scale modification using a sines+transients+noise signal model". In: *Proc. of Digital Audio Effects Workshop*. Nov. 1998, pp. 49–52.

[104] R. Wooller and A. Brown. "Note sequence morphing algorithms for performance of electronic dance music". In: *Digital Creativity* 22 (1 Mar. 2011), pp. 13–25. DOI: 10.1080/14626268.2011.538704.