Sound and music

T. Blackwell, M. Grierson, M. Pearce,
C. Rhodes and A. Smaill

2910**346**

2009

BSc in Computing and Related Subjects

The material in this subject guide was prepared for the University of London External System by:

T. Blackwell; M. Grierson; M. Pearce; C. Rhodes, Goldsmiths, University of London.
A. Smaill, University of Edinburgh.

The guide was edited and produced by J. Forth and S. Rauchas, Department of Computing, Goldsmiths, University of London.

This is one of a series of subject guides published by the University.

This subject guide is for the use of University of London External System students registered for programmes in the field of Computing. The programmes currently available in these subject areas are:

BSc(Honours) in Computing and Information Systems
BSc(Honours) in Creative Computing
Diploma in Computing and Information Systems
Diploma in Creative Computing

Published 2010

Publisher:
The External System
Publications Office
University of London
Stewart House
32 Russell Square
London
WC1B 5DN

www.londonexternal.ac.uk

Contents

Preface

At level 3, the Creative Computing programme has two emphases: audio and visual. *343 Computing art and image effects* focuses on the visual, while this course is about audio.

The approach taken in our study of the subject is a topic-based one. Each chapter of this study guide focuses on one of five areas within the field of digital sound, and looks at it with a creative perspective.

At level 3, you are expected to understand topics more deeply, and to be able to make the connections between different areas. The big picture is not painted explicitly by the authors of the guide; you are expected to do some of this yourself, to obtain a broad appreciation of the material and its links.

You should read widely; you should not read only this subject guide and the explicit readings that it lists, but you should also look for related material yourself. This will develop your ability to decide for yourself whether a paper or article or textbook is of value, and will give you useful skills for carrying out your project and in your future career.

The examination will consist of questions from the material introduced in this study guide, as well as the broader reading and courseworks. A sample examination paper is included to give you an idea of the kind of questions to expect.

This is a fast-moving area and an exciting one. We hope you find the subject interesting and stimulating!

Syllabus

This subject guide presents material on topics within the subject of sound and music in a creative computing context. These are:

- computational models of music cognition
- interactive sound using Pure Data
- algorithmic composition
- understanding musical interaction
- music retrieval and searching.

You also are expected to complete two courseworks that contain relevant material from the syllabus. Some of the material for the courseworks may be covered in the study guide; for some of it you are also expected to read and research independently.

The courseworks are an integral part of your learning experience in this subject.

Reading

There is no one prescribed text for this course. Sometimes specific readings are suggested in relevant sections of the subject guide; sometimes mention is made of books and papers that explain concepts more. The bibliography at the end of this volume is a useful resource for you. You are not expected to know all of the references it lists in great detail, but it provides you with an overview of the area, as well as the ability to examine in more depth any topic that is of particular interest.

A guided reading list accompanies this guide, to help you synthesise the material from an exciting and fast-growing area.

Finally . . .

We hope that topics covered in the subject guide, together with the courseworks, will whet your appetite for further investigation into the use of computers in music, which is a broad and stimulating area. Best wishes for your studies!

Sarah Rauchas and Jamie Forth, Goldsmiths, University of London

Chapter 1

Computational models of music cognition

1.1 Introduction

Although music has an objective physical reality as sound pressure waves, it is only through being perceived by a mind that sound becomes music. Thus music is a fundamentally psychological phenomenon and it is this psychological dimension that distinguishes music from sound and leads to the diversity of musics around the world, each particular to its own cultural context. The goal of cognitive science is to understand how the mind works and it does this by building computer models of cognitive processes and comparing the behaviour of these models to that of humans. Music cognition is the study of the workings of the musical mind using a cognitive scientific approach. In this chapter, we consider some general issues relating to the construction of representations and algorithms in models of music cognition. These issues are illustrated in a review of three concrete examples of music cognition:

- the perception of grouping structure in music
- melodic pitch expectations
- key-finding.

First, however, we must consider the various motivations for building computer models of music. This is important because different motivations imply different goals and, therefore, different methods for achieving these goals. This will help frame our discussion of the cognitive scientific approach and its application to music perception in broader terms.

1.2 Motivations and methods

People build computer programs that process music for many different reasons. A composer might be motivated by artistic concerns to do with the generation of novel musical structures for a new composition, whereas a musicologist might be concerned to find structural relationships between two movements of an existing piece of music, for example. However, in spite of their different motivations, they might well face very similar challenges in writing programs to do these tasks. In both cases, for example, they might need to write programs to break musical passages up into musically meaningful chunks (e.g. phrases or motifs). Our two individuals might therefore end up with programs that are superficially quite similar.

However, it is important to understand that different motivations imply different objectives, which in turn imply different methods for achieving those objectives (Pearce et al. 2002). These methods include the way the program is written and how it is tested. This is not to say that programs cannot be used for more than one purpose but it is important to be clear about one's own motivations before sitting down to write a program that processes music. This should be made clear by the following examples of computer programs written to generate music.

Algorithmic composition Many who write programs for the composition of music are motivated by artistic goals: computer programs are used to generate novel musical structures, compositional techniques and even genres of music. David Cope, for example, writes that:

> *EMI [Experiments in Musical Intelligence] was conceived . . . as the direct result of a composer's block. Feeling I needed a composing partner, I turned to computers believing that computational procedures would cure my block.* Cope (1991, p.18)

Here the computer program is an integral part of the process of composition (and perhaps even of the composition itself). Since the program is written for the composer's own use, there are very few methodological constraints on its development: it need not be well-written or general-purpose (indeed it may only work for a single composition and only run on the composer's computer). Ultimately the criteria for success are one and the same as those for the compositions produced using the program: audience reactions, record sales, critical reviews, and so on.

Design of compositional tools If we intend to write a program that will be more generally useful in composition, then we are firmly into the domain of software engineering and, accordingly, we need a more sophisticated methodological approach. This includes an explicit analysis of what the system is to achieve and why (who the users are, why they need such a system, and so on) and testing to evaluate whether the software achieves its design objectives. Other issues that may need to be considered include documentation, support, bug-tracking, code management, release schedules, and so on.

Computational music analysis Turning now from practical applications to theoretical ones, we return to our musicologist. There is a branch of musicology where computer programs are used to propose and verify hypotheses about the stylistic attributes defining a body of works. Such hypotheses might concern, for example, metrical or contrapuntal structure. A computer program that implements the hypotheses is a model of the style and potentially offers the musicologist several benefits, some of which are practical: the program can perform routine, time-consuming aspects of the analysis and may allow the analyst to study a much larger corpus of compositions than they would be able to by hand. Other benefits are methodological. In writing a program, the musicologist must explicitly specify all aspects of the stylistic theory including premises and hypotheses – self-evident or otherwise. It is important that the program is written in such a way that the stylistic hypotheses are clearly expressed and easily changed and extended. Finally, the model can be tested in a number of ways. One method of evaluation is called **analysis by synthesis**: we use the program to generate new pieces and compare them to existing pieces in the style, using discrepancies to identify shortcomings of the model and, subsequently, improve it. Ultimately, a well-tested model can be used to address questions of real interest to musicologists. For example, computer programs have been used to re-attribute pieces of music to other composers and to identify recordings which fraudulently contain extracts of other performers' renditions of the same piece.

Music cognition Could a computer program that generates music help us understand how our minds work? A psychologist, for example, might be interested in the cognitive processes involved in improvising a jazz solo (Johnson-Laird 1991). One way of proceeding would be to propose a theory consisting of some hypotheses about the mental processes involved, including the physical and mental constraints (e.g. timing and memory), the inputs and outputs to the cognitive procedure and the processing steps that transform inputs to outputs (e.g. are the pitch and timing of the next note generated independently; if so, which comes first). These hypotheses can then be implemented in a computer program whose behaviour–including the solos it generates–can be compared to human jazz musicians to test the adequacy of the theory as an account of the cognitive process of jazz improvisation. This is the primary methodological approach in cognitive science and its application to music is the topic of the remainder of this chapter.

1.3 Cognitive science

The goal of cognitive science is to explain how the mind works. Broadly speaking, where psychologists tend to focus on designing experiments to measure human responses to particular situations to infer the underlying psychological processes, in cognitive science the emphasis is on building computational models of psychological processes and then comparing the responses of the models to certain inputs with human responses to the same stimuli.

The basic assumption underlying the cognitive approach is that mental processes can be conceived of as computational procedures (Newell & Simon 1976, Johnson-Laird 1988, Pylyshyn 1989). This is sometimes called the **computational metaphor** according to which the mind can best be understood as a sequence of functional procedures progressively transforming inputs (e.g. perception) to outputs (e.g. behaviour). This metaphor has two implications: first, so long as the physical substrate provides for an appropriate degree of computational power, its physical nature places no constraints on theories of cognition; and second, any scientific theory of cognition may be simulated by a computer program.

Ultimately, then, a cognitive theory is implemented as a model consisting of a set of algorithms operating on a series of representations. A cognitive scientist cannot observe the operation of these algorithms and representations directly in the human mind, but can constrain the design of the model on the basis of what is known about the mind (from psychological experiments) and the brain (from neuroscientific studies). On this basis, Marr (1982) introduced three levels at which the mind can be studied:

1. the computational theory;
2. the representation and algorithm (the computer program);
3. the hardware implementation (the physical level: the brain or computer hardware).

The first level deals with the **what** and the **why** of the system. What is the goal of the computation? Why is it appropriate? What is the logic of the strategy by which it can be carried out? At this level, the computational theory attempts to describe the intrinsic nature and computational requirements of a cognitive task through a formal analysis of the various outputs resulting from different inputs. Through understanding the nature of the problem to be solved, appropriate constraints can be placed on the representational and algorithmic levels of the theory. Cognitive scientists often argue that the representation and algorithm (the second level) depend much more on the first level than on the third level:

> *. . . trying to understand perception by studying only neurons is like trying to understand bird flight by studying only feathers: it just cannot be done.*
>
> *Marr (1982, p. 27)*

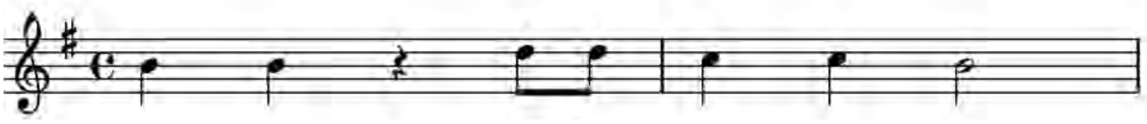
Note that the metaphor doesn't state that the mind **is** a computational process, only that theories of how it works should be so. Computational theories of cognitive processes have many advantages. Because cognitive models are implemented as computer programs (rather than, say, verbal descriptions or box-and-arrow diagrams), they explicitly contain **all** the machinery required to perform the task of interest. They take little for granted and any assumptions made are concretely expressed in the program. Also, because we can run the program, we can look at its behaviour, measure it and compare it quantitatively to human behaviour. Any discrepancies can be used to improve the model.

1.4 Music cognition

Music cognition is the application of the cognitive scientific method to study how the mind represents and processes music. We first consider some general issues in designing musical representations and algorithms and then review some examples of cognitive models of music perception. As a generality, it is good practice to maintain a clear distinction between representations and algorithms, so that one can easily apply new algorithms to the same representations or apply the same algorithms to new representations. Apart from the practical advantages, this may also help to make one's cognitive models more general.

1.4.1 Representations

Musical representations vary along many different dimensions (Wiggins et al. 1993). One dimension is the level of abstraction involved. The most concrete and complete representation would be a specification of the analogue sound pressure waves arriving at each ear. Although it allows us to reproduce a piece of music (indeed a particular performance of a piece) very accurately, this representation does not correspond very closely to our subjective experience of music which makes it unsuitable for many kinds of task. A performer, for example, would find it difficult to use as a set of performance instructions, while a composer would have trouble using it to reorchestrate the piece. In these cases, Western musicians would use a score (or perhaps a piano roll sequencer), which contains abstract information about the pitch, timbre, timing and dynamics of notes in the piece of music. The score is one of an infinite range of abstract representations of music, which both discards information deemed unnecessary but also introduces more abstract structure (e.g. notes, chords, key signatures, time signatures, clefs, rests, and so on). The explicit representation of abstract structures allows the music to be manipulated in different



Onset	0	2	6	7	8	10	12
Duration	2	2	1	1	2	2	4
Pitch	71	71	71	74	72	72	71
IOI	⊥	2	4	1	1	2	2
Rest	⊥	0	2	0	0	0	0
Pitch interval	⊥	0	3	0	-2	0	-1
Contour	⊥	0	1	0	-1	0	-1
Scale degree	4	4	7	7	5	5	4

Figure 1.1: An example of an abstract, symbolic representation scheme for a simple melody. See the text for details.

ways but the loss of information means that many different concrete sound pressure waves can be produced from a single score.

Another distinction that is often made is between symbolic and non-symbolic representations and this dimension is orthogonal to the degree of representational abstraction (although symbolic representations tend to be used at higher levels of abstraction). A symbolic representation scheme is one in which each concept or object of interest is represented using a discrete local symbol and relationships between symbols (e.g. has-a, is-a or temporal relations) are explicitly represented. For example, we could represent a melody using a sequence of discrete symbols representing notes ordered in time, each of which has a number of discrete symbols representing onset time, duration, pitch, loudness, and so on. Alternatively, we could represent the same melody as a matrix of numerical values, whose rows represent musical attributes such as pitch, loudness, timbre and whose columns represent observations or samples of these attributes at 200 millisecond intervals. This is an example of a non-symbolic representation.

Finally, some non-symbolic representations may be distributed in that a particular concept may be represented by the pattern of activity across (and weights between) a number of different processing units. Such representations are common in neural network models. Distributed representations are attractive because they share some features in common with actual neuronal circuits in the brain although most neural network models used in cognitive science are better described as neuronally inspired than neuronally plausible.

Here, we focus on abstract, symbolic and local representations of music in which the **musical surface** (the lowest level of abstraction of interest) consists of a collection of discrete notes, each of which possesses a finite number of properties including an onset time, a duration and a pitch which might be represented as shown in Figure 1.1. For our purposes, time can be represented in temporal units (e.g. milliseconds) or metrical units such as bars, **tactus** beats or in terms of the **tatum** (the smallest note duration in a sample of music). In Figure 1.1, a single time unit is the tatum for this short phrase, which is a quaver or eighth note. The time signature (common time or $\frac{4}{4}$) tells us that a tactus beat is 2 of these time units while a bar (or measure) lasts for 8 of these time units. Given a sequence of non-overlapping notes, a melody, we can use the onset time and duration of each note to calculate the time interval between the onset of one note and that of the one before it. This is called the **inter-onset interval** (IOI). We can also calculate the time interval between the end of one note and onset of the next, which corresponds to a musical **rest** or silence between consecutive notes. As shown in Figure 1.1, the IOI and rest are undefined (indicated by $\perp$) for the first note in a melody since two notes are required to compute an interval.

Chromatic pitch is usually represented in log frequency units with 12 units in an octave, or doubling of frequency. The MIDI standard defines pitch in the range of 0 to 127, where middle C on the piano (C_4 where 4 is the octave number) is 60. This is the representation used in Figure 1.1. Given a melody with pitch represented in such a way, we can compute the **pitch interval** between consecutive notes by subtracting the pitch of the earlier note from that of the later one. As shown in Figure 1.1, pitch interval is undefined for the first note in a melody. Note also that pitch intervals have both a magnitude (corresponding to the number of semitones between the two pitches) and a **registral direction** that distinguishes ascending intervals, descending intervals and unisons. We can use pitch intervals to compute an even more abstract pitch representation called **contour** (sometimes known as Parsons code), which simply represents registral direction (up, down, or same, or often, 1, -1 and 0 as in Figure 1.1). Finally, we can represent pitch in a way that is relative to a particular note, usually the **tonic** in Western music. In Figure 1.1, for example, the key signature indicates that the melody is in the key of G Major, whose tonic is G ($G_4 = 67$). The chromatic scale degrees shown in Figure 1.1 are computed by subtracting the pitch of each note from the G below it. For example, the first and last note are both B_4, so the scale degree is $71 - 67 = 4$.

1.4.2 Algorithms

In principle, any kind of algorithm or computational procedure can be used to build cognitive models: existing work in music cognition includes grammars, expert systems, evolutionary algorithms, neural networks, graphical models and many more. One distinction that is commonly made is between static models whose representations and algorithms are designed by hand and models that acquire their representations or algorithms through a process of learning from experience. A learning model can make strong predictions about how a particular cognitive process is acquired which can be tested by studying, for example, developmental changes in behaviour as children accumulate more musical experience or cross-cultural differences in behaviour given different kinds of musical experience. In the case of a static model, it may be that the representations and algorithms are thought to be innate properties of the mind or simply that their acquisition is not of interest. One common approach in music cognition has been to draw on music theory to propose static rule-based models of music cognition, Lerdahl & Jackendoff (e.g. 1983), Narmour (e.g. 1990), Temperley (e.g. 2001*a*).

1.5 Examples

In this section, we cover some concrete examples of computer models that have been built to study music cognition.

1.5.1 Grouping

Many aspects of cognition require an ability to group items together or to segment complex objects into simpler components. Examples include the grouping of visual components into complete objects in visual perception, identifying word boundaries in speech or planning a complex sequence of movements. Even something as simple as remembering a phone number usually involves grouping the individual numbers into a smaller number of larger 'chunks'.

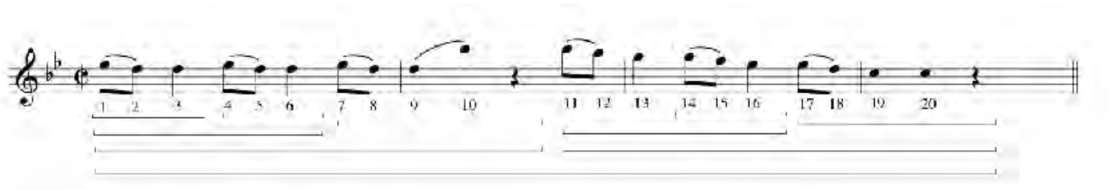


Figure 1.2: An example of melodic grouping structure: the opening of Mozart's G Minor Symphony, K. 550, Lerdahl & Jackendoff (after 1983, Figure 3.1)

Various kinds of grouping process operate in music perception, including the **fusing** of simultaneously occurring events into a single percept (e.g. the fusing of notes into a chord), the segregation of polyphonic music into parallel streams (usually called **stream segregation** or **streaming**) and the sequential grouping of notes into motifs and phrases, which are subsequently grouped into sections and movements. An example of the latter kind of grouping process is shown in Figure 1.2. This is thought to be one of the fundamental cognitive processes involved in music perception:

> *When a listener has construed a grouping structure for a piece, he has gone a long way toward "making sense" of the piece . . . grouping can be viewed as the most basic component of musical understanding.*
>
> *Lerdahl & Jackendoff (1983, p. 13)*

The most venerable cognitive theory about musical grouping structure is inspired by the Gestalt principles originally developed to account for visual perception. Figure 1.3 illustrates the Gestalt principles of proximity and similarity as they apply to the grouping of objects in visual perception. Note that the two principles can conflict with each other.

Just as the visual system tends to group together objects that are close together in space or similar (in colour, shape, size and so on), the auditory system tends to group together notes that are close together (proximate in time) and similar in some way (e.g. in terms of pitch, timbre, loudness, duration and so on). Figure 1.4 illustrates the Gestalt principles of (temporal) proximity and (pitch) similarity as they apply to the grouping of notes in melody perception. The goal of the computation is to exhaustively partition a passage of music into a sequence of non-overlapping groups containing two or more notes each. These low-level groups may then be combined into higher-level groups (although here we consider only a single level grouping analysis).

These principles of proximity and similarity can be used to construct a cognitive model of grouping processes in music perception. One recent version of this approach is the **Local Boundary Detection Model** (LBDM) developed by Cambouropoulos (2001) which assigned each note in a melody a boundary strength:

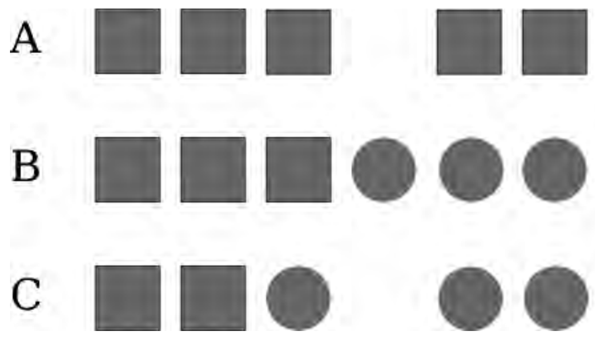


Figure 1.3: The Gestalt principles of proximity (A) and similarity (B) in visual perception; the two principles may produce conflicting groups (C).

the higher the strength the more likely the note is to be preceded by a boundary. The LBDM consists of two rules: first, a **change** rule, which assigns boundary strengths in proportion to the degree of change between consecutive intervals; and second, a **proximity** rule, which scales the boundary strength according to the size of the intervals involved. The LBDM takes as input three aspects of a symbolically represented melody: the first represents the pitch interval between successive pairs of notes, the second represents the temporal inter-onset interval (IOI) between successive notes and the third is the temporal interval between the end of one note and the onset of the next (a rest). These features are represented as vectors with one element for each pair of consecutive notes in the melody, $P_k = [x_1, x_2, \dots, x_n]$ where $k \in \{\text{pitch, IOI, rest}\}$, $x_i > 0, i \in \{1, 2, \dots, n\}$. The boundary strength at interval x_i is given by:

$$s_i = x_i \times (r_{i-1,i} + r_{i,i+1}) \tag{1.1}$$

where the degree of change between two successive intervals:

$$r_{i,i+1} = \begin{cases} \frac{|x_i - x_{i+1}|}{x_i + x_{i+1}} & \text{if } x_i + x_{i+1} \neq 0 \wedge x_i, x_{i+1} \geq 0 \\ 0 & \text{if } x_i = x_{i+1} = 0. \end{cases}$$

For each parameter k, the boundary strength profile $S_k = [s_1, s_2, \dots, s_n]$ is calculated and normalised in the range $[0, 1]$. A weighted sum of the boundary strength profiles is computed using weights derived by trial and error (.25 for *pitch* and *rest*, and .5 for *IOI*), and boundaries are predicted where the combined profile exceeds a predefined threshold.

A model of melodic grouping can be tested by comparing its output to human behaviour. There are various ways of achieving this. For example, Cambouropoulos (2001) found that notes falling before predicted boundaries were more often lengthened than shortened in pianists' performances of Mozart piano sonatas and a Chopin étude, in accordance with music-theoretic predictions.

Another way would be to ask people to indicate where they perceive boundaries in a collection of melodies. For example, we could give a musicologist the score and ask her to mark the boundaries on the paper. Or we might try to obtain more perceptual (and less intellectual) judgements by asking non-musicians to indicate grouping boundaries (with a mouse click, for example) while they listen to a collection of melodies, and then combine their responses in some way. Either way, we would end up with a set of data indicating, for each note in the collection, whether or not that note falls directly after a boundary. Therefore, we can represent each melody as a binary vector, where the entry at index *i* of the vector indicates whether the *i*th note of the melody falls after a boundary (1) or not (0). Given the same melody, a cognitive model such as the LBDM also produces such a vector, so we can compare its output to that of our listeners.

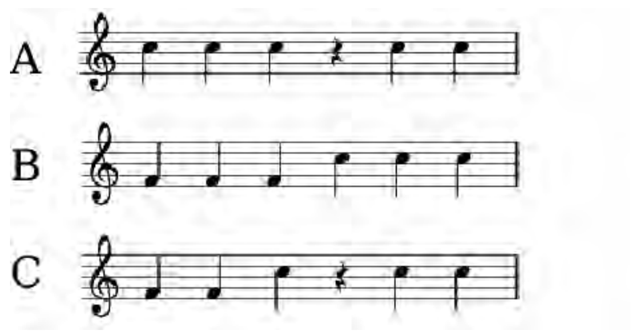


Figure 1.4: The Gestalt principles of temporal proximity (A) and pitch similarity (B) in music perception; the two principles may produce conflicting groups (C).

There are many general purpose methods for comparing the similarity between binary vectors, many of which rely on the following classification of agreements and disagreements between the vectors. There are two cases in which the model's predictions may agree with the perceptual data: a **true negative** (TN) is where the model correctly predicts the absence of a boundary (both vectors contain 0); a **true positive** (TP) is where the model correctly predicts the presence of a boundary (both vectors contain 1). Similarly, the model may make two kinds of error: a **false positive** (FP) is where it incorrectly predicts a boundary while a **false negative** (FN) is where it incorrectly fails to predict a boundary.

Two commonly-used methods for comparing binary vectors are **precision** and **recall**. Precision reflects the true positives as a proportion of the positive output of the model (i.e. the proportion of boundaries predicted by the model that are correct); while recall reflects the true positives as a proportion of the positives in the perceptual data (i.e. the proportion of perceived boundaries correctly predicted by the model). **F1** is the harmonic mean of the two.

$$\begin{aligned} Precision &= \frac{TP}{TP + FP} \\ Recall &= \frac{TP}{TP + FN} \\ F1 &= \frac{2 \cdot precision \cdot recall}{precision + recall} \end{aligned}$$

Cambouropoulos (2001) found that the LBDM obtained a recall of up to 74% of the boundaries marked on a score by a musician (depending on the threshold and weights used) although precision was lower at 55%. Because it reflects listeners' perceptions reasonably well, the LBDM can also be used as a tool for automatically segmenting melodies into phrases.

1.5.2 Expectation

Our minds are very good at predicting what will happen next; in an evolutionary sense, they have to be to allow us to avoid negative outcomes and seek out positive ones: "All brains are, in essence, anticipation machines." Dennett (1991, p. 177).

When we listen to a piece of music, our minds anticipate forthcoming structures, based on what has gone before. This process of expectation is crucial to our experience of musical tension and resolution, corresponding to violation and confirmation of our expectations respectively, which may in turn influence the emotional impact of the music and its aesthetic effect.

So the question of how the mind generates expectations is of considerable interest in music cognition. While listeners generate expectations about many aspects of musical structure, including harmony, timing and dynamics, here we focus on pitch expectations in melody. The goal of the computation is to predict the pitch of the next note in a melody given the pitches of the preceding notes. Several models of melodic pitch expectation have been proposed, many of which use the pitch interval between two consecutive notes to anticipate the new pitch interval formed when the next note arrives. The first interval is known as the **implicative** interval because it generates an implication for the second interval, called the **realised** interval. One of the most recent models consists of just two rules:

Pitch proximity: pitches are expected to be similar to the pitch of the previous note (i.e. we expect small melodic intervals);

Pitch reversal: pitches are expected to be similar to the first pitch of the penultimate note and large implicative intervals are expected to be followed by a change in registral direction.

This is known as the **two-factor** model of melodic expectation (Schellenberg 1997). The first factor is another example of the transfer of Gestalt principles from visual perception to music cognition. The two principles are quantified in terms of the size (in semitones) of the implicative and realised intervals as shown in Figure 1.5. The output of the two principles can be averaged to produce an overall model that produces quantitative pitch expectations for the pitch of the note following any two pitched notes.

The two-factor model is inherently static: it does not depend in any way on the musical experience of the listener and its expectations will not change with increasing or changing musical experience. It can be compared with a very simple learning based model based on n-gram statistics. An n-gram is a sequence of n symbols and an n-gram model is simply a collection of such sequences each of which is associated with a frequency count. The quantity $n - 1$ is known as the **order** of the model and represents the number of symbols making up the context within which a prediction is made. During the **training** of the statistical model, the frequency counts are acquired through an analysis of some corpus of sequences (the training set) in the target domain (melodies in this case). When the trained model is exposed to a melody, it uses the frequency counts associated with n-grams to estimate a probability distribution governing the identity of the next note in the melody given the $n - 1$ preceding symbols.

The most elementary n-gram model of melodic pitch structure (a monogram model where $n = 1$) simply tabulates the frequency of occurrence for each chromatic pitch encountered in a traversal of each melody in the training set. During prediction, the expectations of the model are governed by a zeroth-order pitch distribution derived

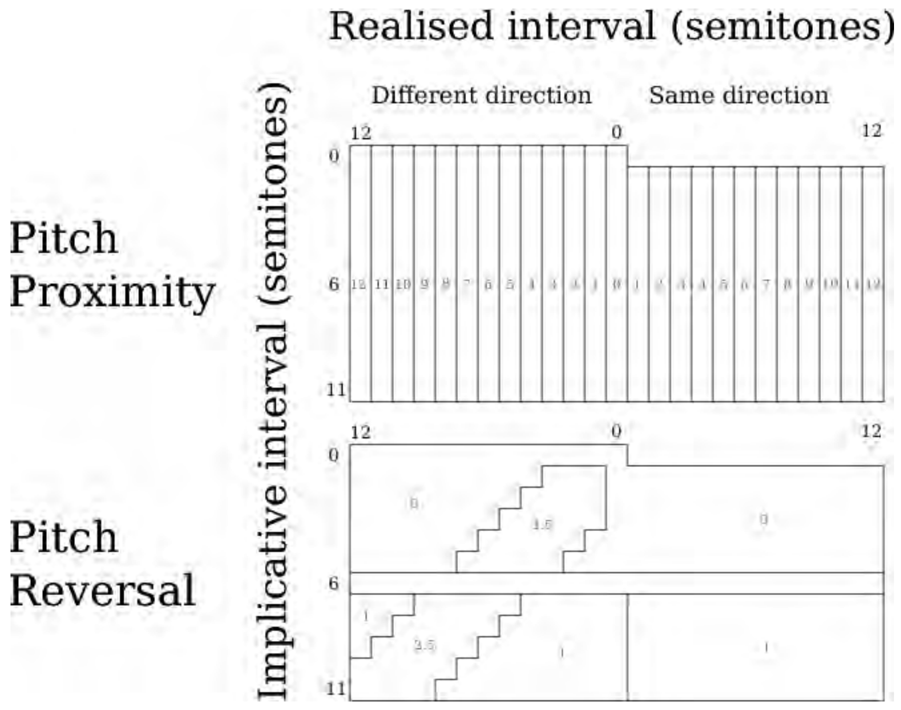


Figure 1.5: The quantification of the principles of pitch proximity and pitch reversal in the two-factor model Schellenberg (after 1997, Figure 4). The model predicts the expectedness of the last of three consecutive notes in a melody as a function of the size and direction of the two pitch intervals (the implicative then the realised) formed by the three notes.

from the frequency counts, and they do not depend on the preceding context of the melody. In a digram model (where $n = 2$), however, frequency counts are maintained for sequences of two pitch symbols and predictions are governed by a first-order pitch distribution derived from the frequency counts associated with only those digrams whose initial pitch symbol matches the final pitch symbol in the melodic context.

Given a pair of consecutive pitches in a melody, x_1 and x_2, we can compute the conditional probability $p(x_1|x_2)$ as being the number of times that x_1 is followed by x_2, divided by the number of times that x_1 occurs regardless of what follows it.

Note that this is different from simply computing the overall probability of the pattern $x_1 x_2$ in the corpus of melodies. Huron (2006) discusses other applications of n-gram models in music cognition.

Rather than building a digram model of pitches, we could just as easily build one of pitch intervals. The trained model would look something like the table shown in Table 1.1 which shows the frequency counts of realised intervals given a preceding implicative interval. We can use the table to compute conditional probabilities. For example, the conditional probability of a realised ascending perfect fifth (7 semitones) given an implicative descending major third (-4 semitones) is 97/314 = 0.31.

This would be a good learning model to compare with the two-factor model as both require as input two intervals and return as output expectations about the pitch of the third note. To see which is the best cognitive model, we can compare their

	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	
-12	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	5	0	2	0	0	0	0	0	10
-11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-9	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	4	0	0	0	0	0	0	0	6
-8	0	0	0	0	0	0	0	0	0	0	0	0	6	2	0	2	0	2	0	0	0	0	0	0	0	12
-7	0	0	0	0	0	0	0	1	0	0	0	0	14	0	7	0	3	8	0	16	0	5	1	0	1	56
-6	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	3	0	0	2	0	0	0	0	0	0	8
-5	0	0	0	0	0	0	0	0	1	11	11	0	72	1	3	1	2	44	0	7	5	14	1	0	10	183
-4	0	0	0	0	0	0	0	30	0	2	0	14	75	0	33	0	23	18	0	97	0	16	0	0	6	314
-3	0	0	0	2	0	0	0	0	138	4	77	0	143	38	17	186	0	49	0	5	21	0	0	0	0	680
-2	0	0	0	0	0	19	2	41	1	197	269	228	373	0	96	17	51	34	0	25	0	4	0	0	0	1357
-1	0	0	0	2	0	0	0	5	54	0	153	0	52	50	0	59	0	12	0	0	2	0	0	0	0	389
0	2	0	0	1	3	22	0	53	34	307	498	84	827	57	396	92	76	98	0	34	3	20	3	0	1	2611
1	0	0	0	0	0	0	1	17	0	23	0	20	44	0	102	0	4	1	0	5	0	1	0	0	0	218
2	1	0	0	0	0	8	0	4	52	47	191	0	312	90	115	19	0	11	0	0	0	0	0	0	0	850
3	1	0	0	0	0	5	5	9	0	45	69	29	235	0	26	1	2	9	0	1	0	2	0	0	0	439
4	0	0	0	1	0	1	0	0	32	0	37	0	52	4	0	57	0	3	0	0	0	0	0	0	0	187
5	2	0	0	0	4	8	0	17	0	22	71	16	142	0	24	1	15	0	0	2	0	0	0	0	0	324
6	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	2
7	3	0	0	0	0	1	0	4	0	16	21	0	152	0	9	0	0	0	0	0	0	0	0	0	0	206
8	0	0	0	0	5	0	0	5	0	7	3	1	9	0	0	0	0	0	0	0	0	0	0	0	0	30
9	0	0	0	0	0	0	0	1	13	0	17	0	31	0	0	0	0	0	0	0	0	0	0	0	0	62
10	0	0	0	0	0	0	0	1	0	1	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	5
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0	0	2	5	3	6	0	1	0	0	0	0	0	0	0	0	0	0	18
	10	0	0	6	12	64	8	188	325	684	1423	398	2553	242	830	438	176	298	2	194	31	62	5	0	18	7967

Table 1.1: A digram frequency table for pitch intervals in 213 German nursery rhymes. The rows represent the implicative interval and the columns the realised interval in semitones; negative numbers represent descending intervals and positive ones ascending intervals. The entries in each cell show the number of times the realised interval follows the implicative interval in the corpus.

output with the behavioural responses of human listeners collected in psychological experiments. Such experiments usually involve a **probe-tone** task where listeners are asked to listen to a melodic context and state how expected they find the final note (on a scale from 1 to 7, for example). By varying the final probe tone (say between an octave above and an octave below the final tone of the context), expectations can be gathered for a range of notes in the same melodic context. The procedure is repeated for a number of different melodic contexts and the whole experiment is done by several different listeners whose responses are averaged for each pair of contexts and probe-tones. We can then produce expectation predictions from each of our models for each pair of contexts and probe-tones and compare these to the average expectations of our listeners.

One way to do the comparison is using correlation which is a statistical procedure for identifying linear relationships between two properties $X = x_1, x_2, \ldots, x_i$ and $Y = y_1, y_2, \ldots, y_i$ of a set of datapoints. A correlation is a straight line $Y = a + bX$ where the intercept a and coefficient b are fitted to the data so as to minimise the deviation of each datapoint from the line. The correlation coefficient b may vary between -1 (strong inverse relationship) to 1 (strong positive relationship) where 0 represents no relationship (a horizontal line). Figure 1.6 shows an example where each point plotted is a note in a melody, the y axis shows the average strength of expectation of a group of listeners for that note and the x axis shows the expectations predicted by a more sophisticated version of the digram model (Pearce & Wiggins 2006). The correlation coefficient here is 0.91 indicating a strong relationship between the predictions of the model and the listeners' expectations. If we are comparing two models, then we can correlate each one's predictions with the average expectations of the listeners and, simply put, whichever model has the highest correlation coefficient b wins.

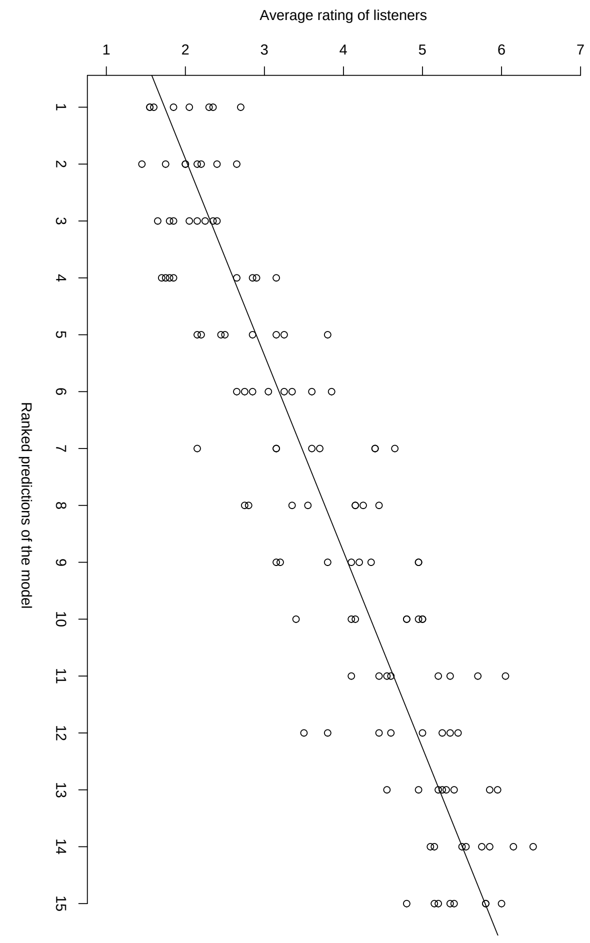


Figure 1.6: Correlation between the average expectations of listeners (y axis) and the predictions of a computational model (Pearce & Wiggins 2006).

1.5.3 Key finding

When we listen to a piece of music, we usually perceive that some notes are more important and occur more frequently than others. In Western tonal music, we identify one note as the **tonic** and perceive other notes in relation to this note. In the Western tonal idiom, the key signature of a piece of music defines both the tonic but also the **mode** of the piece: together, the tonic and the mode place constraints on the set of pitches which are likely to appear in the music. For example, a piece of music written in C major will have the note C as its tonic and will tend to be restricted to the white keys on the piano. These are the notes, or degrees, of the C major scale and pitches represented in relation to a tonic are known as **scale degrees**. Although the key signature is encoded in a score, the tonic and mode are not explicit in the auditory signal and must be inferred from the pitches that occur in the music. The probe-tone paradigm introduced in §1.5.2 has been used to demonstrate that listeners infer tonal relationships spontaneously when they listen to passages of music.

Krumhansl & Kessler (1982) derived the perceived strength of scale degrees as experimentally quantified **key profiles** using a **probe tone** experimental paradigm. Ten musically trained Western listeners were asked to supply ratings of how well a variable probe tone fitted, in a musical sense, with an antecedent musical context. The probe tones in Krumhansl & Kessler's experiments were all 12 tones of the chromatic scale which were presented in a variety of contexts, including complete diatonic scales, tonic triads and a number of chord cadences in both major and minor keys and using a variety of tonics. The results exhibited high consistency across contexts and both inter- and intra-subject consistency were also high. Furthermore, in a substantial number of instances, the same patterns were found for the different major and minor contexts once the ratings had been transposed to compensate for the different tonics. Because of this consistency, the data can be averaged across subjects and keys to derive quantitative profiles for the stability of tones in major and minor keys from the experimental data. The resulting profiles are shown in Figure 1.7.

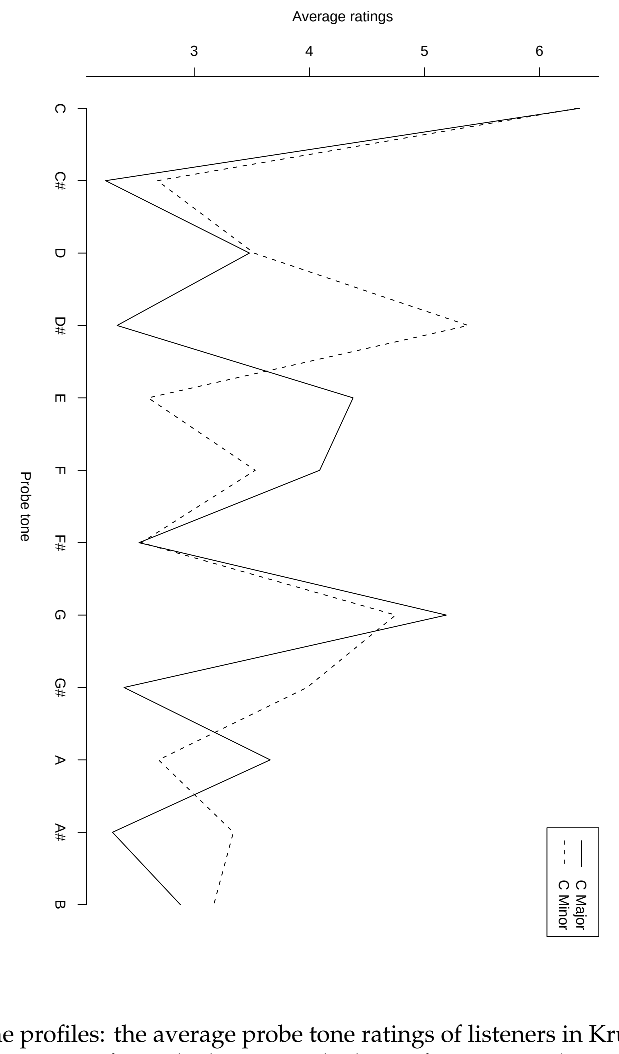


Figure 1.7: Tone profiles: the average probe tone ratings of listeners in Krumhansl & Kessler's experiment for scale degrees in the keys of C major and minor.

These profiles substantially conform to music-theoretic accounts of decreasing relative stability from the tonic, through the third and fifth scale degrees, the remaining diatonic scale degrees (white piano keys in C major) and finally to the non-diatonic tones (the black keys on the piano in C major). These key-profiles are frequently added to the two-factor model in order to represent the effects of tonal perception on melodic pitch expectations. They can also be used to implement a method of finding the key of a piece of music known as the *Krumhansl-Schmuckler* key-finding algorithm Krumhansl (1990, pp.77–110). The input to the algorithm is a selection of music symbolically represented as notes with pitches and durations. The algorithm proceeds as follows:

1. Compute a 12-dimensional vector $I = (d_1, d_2, \ldots, d_{12})$ specifying the total duration (in seconds or metric units, such as beats) of each of the 12 chromatic pitches in the sample.
2. Generate 12 major key profiles $K_1, K_2, \ldots, K_{12}$ corresponding to each of the 12 possible tonics (C, C#, D,. . . , B) by rotating the major key profile 11 times.
3. Generate 12 minor key profiles $K_{13}, K_{14}, \ldots, K_{24}$ corresponding to each of the 12 possible tonics by rotating the minor key profile 11 times.
4. Correlate I with each of the key profiles corresponding to the 24 major and minor keys $K_1, K_2, \ldots, K_{24}$ and store the resulting correlation coefficients in a new vector $(R_1, R_2, \ldots, R_{24})$.
5. The predicted key signature is the one with the highest correlation coefficient.

Although it is very simple, it has been shown that the algorithm is quite good at predicting key signatures – both as notated in the score and as perceived by listeners – in a variety of musical genres. However, there is something rather circular about using the results of a psychological experiment (the key profiles) directly as a component in a cognitive model. If our minds do contain stored representations of the major and minor key profiles, how did they arise? One possibility is that they are acquired through statistical learning of the kind discussed in §1.5.2 and indeed it has been shown that the key profiles correlate strongly with monogram distributions of the pitches in various collections of music.

1.6 Summary

In this chapter we have:

- examined different reasons for writing computer programs that process music and their associated goals and methods;
- introduced the field of cognitive science and discussed some features that distinguish it from other ways of studying the mind, in particular its use of computational metaphors and models;
- introduced music cognition as a sub-field of cognitive science and looked in general at issues involved in selecting algorithms and representations for modelling the musical mind;
- surveyed three examples of music cognition research:
  - melodic grouping;
  - melodic pitch expectation;
  - key-finding.

The first two examples illustrate the use of ideas from visual perception and music theory in building cognitive models of music perception and how such models can be tested against human responses. The second example also introduces two very different models, one based on static rules and the other on learning, and how they can be compared. The final example illustrates the development of a cognitive model from quantitative perceptual data, and how it can be used as a practical tool.

1.7 Exercises

1. What are the limitations of the digram model? How could they be addressed?
2. Think of two ways of combining the principles of the two-factor model and describe the advantages and disadvantages of each.
3. Describe two ways of combining the grouping boundaries indicated by 24 listeners for a single melody into a single binary vector. Discuss the advantages and disadvantages of each.
4. How could you test the hypothesis that the tone profiles of Krumhansl & Kessler (1982) are acquired through statistical learning of pitches in the music we listen to?
5. Neither precision nor recall use the true negatives (TN). Why is this appropriate for comparing two grouping analyses of the same melody?
6. Can you think of another way of representing the digram table shown in Table 1.1? How would the two representations generalise to higher order models? How do the representations compare in terms of, for example, time and space complexity?

Chapter 2

Interactive sound using Pure Data

2.1 Introduction

This chapter explores interactive sound and music using the Pure Data real-time graphical program language. It gives you a solid background in computational approaches to sound and music, including music representation systems such as MIDI (Musical Instrument Digital Interface), and DSP (Digital Signal Processing). In addition, this chapter will teach you to create your own electronic music systems, and to build software that will record, play back and independently generate sound and music.

This chapter takes a broad view of music. For the purposes of this text, music is a special category of sound that can have specific acoustic features (detailed below). It follows on from the *Creative Computing 2: interactive multimedia* module (2910227), and as such assumes some knowledge of sound and acoustics. However, no specialist knowledge of music is required beyond that explained in the text. As such, this chapter does not require any high-level musical training, although knowledge of music will occasionally be advantageous.

2.2 Equipment requirements

This course does not require specialist equipment. Pure Data is compatible with any computer running recent versions of Windows, Linux or Mac OS X. At times, Musical Instrument Digital Interface (MIDI) equipment is mentioned. Access to a MIDI keyboard and interface may be of use, but is not required. Please use the alternative methods indicated if you do not have access to a MIDI device.

2.3 The Fundamentals of sound

Sound is a longitudinal vibration – a wave of varying pressure. Waves travel through any medium at various speeds, depending on both the density **and** elasticity of the medium. We tend to associate sound with longitudinal vibrations travelling through air. These waves have important properties we need to understand before we can start using digital audio.

Objects moving in air cause air molecules to move away from them. Air molecules are pushed into other air molecules (**compression**). This leaves space for new air molecules to move in (**rarefaction**). These new molecules push against the original air molecules. They in turn push more molecules, leaving more space to be filled. This starts a process of compression and rarefaction, and a pressure wave is formed that propagates through the air at a speed of approximately 340 metres per second.

This is similar to what happens when you throw a stone into a pool of water. The rock displaces the water, which pushes more water away, creating space for more water to rush in. The ripples move away from the source. This is similar (but not at all times and in all ways identical) to what happens in air.

We can make a drum by taking a section of flexible material and stretching it over the hollow end of a cylindrical object. If we strike the surface (the skin) of the drum with an object, this causes the surface to move very quickly back and forth. As this energy dissipates, the speed of this movement slows down. This causes air molecules to move either side of the drum skin in the way described above, creating sound waves. What we hear depends on the type of material, the amount of force, and the surface tension.

Waves have three important properties: amplitude, frequency and phase, discussed below.

2.3.1 Amplitude

The amplitude of a wave can be thought of as corresponding to the **amount of energy** in the skin of the drum. When we make a sound, for example by clapping our hands, we perceive the sound as having **loudness**. This roughly correlates to the energy being generated by our hands when they hit each other, or when we strike a drum skin, and by extension, the force with which the air molecules are pushed together. For example, if we drop a large rock in a pool, the waves will be higher than if we drop a small one. If we hit the drum skin hard, there will be more initial energy in the skin, and the sound will be louder than if we hit it softly.

We measure the amplitude of sound waves in **decibels** (dB – tenth of a Bel) after Alexander Graham Bell (1847–1922). This unit of measurement has many different scales, depending on whether we are measuring air pressure (which is also measured in **pascals**), voltages, or industry standard **Volume Units**. In sound recording, we use a scale whereby the loudest sound that can be recorded is known as 0 dBFS (decibels relative to full scale). This is equivalent to the maximum linear amplitude of 1. To calculate the amplitude value between 0 and 1 in dBFS, we use the expression:

$$x = 20 \log_{10} y$$

where y is the value in linear amplitude between 0 and 1, and x is the value in dB. Note that 0.5 is obviously not equivalent to half the perceived loudness, corresponding instead to a logarithmic amplitude of -6.0206 dBFS. For more information please refer to Section 3.4.3 in *Creative Computing 2: interactive multimedia Vol. 1* and Section 2.2 in *Creative Computing 2: interactive multimedia Vol. 2*.

2.3.2 Frequency

Waves propagate at particular rates. We can think of this rate as equal to the number of compression and rarefaction cycles each second. Different types of events have different numbers of cycles per second. When we strike a drum hard, there is a lot of energy in the surface, creating a great deal of tension. This causes the skin to move quickly. As the energy gets used up, the movement of the skin gradually slows down. This causes the frequency of vibration to slow down. This creates the fast change of pitch that we associate with many types of drums.

We measure cycles per second (frequency) in **Hertz** (Hz), after Heinrich Rudolf Hertz (1857–1894). The human ear can detect frequencies between 20 Hz and 20,000 Hz, depending on age and ear function. Although air pressure waves exist way below and above these boundaries, our ears are incapable of perceiving them as sounds. More information on frequency can be found in Section 3.4.2 of *Creative Computing 2: interactive multimedia Vol. 1*, and in Sections 2.2 and 2.3 of *Creative Computing 2: interactive multimedia Vol. 2*.

2.3.3 Phase

Sound waves can not only be measured in terms of amplitudes and frequencies. They also have a phase. Waves are periodic – that is, they repeat themselves in cycles. They start with a steady state, go into compression, go back to a steady state, go into rarefaction, and then reach a steady state once more, where the cycle begins again.

We can measure the stages (or phases) of each cycle in degrees (or radians). We can measure the phase of a wave at any point in its cycle. A wave that begins with a phase of 0, completes half a cycle at 180 degrees (π radians), and completes its cycle at 360 degrees (2π radians). Further information on phase can be found in Section 3.4.4 of *Creative Computing 2: interactive multimedia* Vol. 1.

2.3.4 Musical sounds

Almost all naturally occurring sounds are the sum of many seemingly unrelated frequencies. The mathematical relationship between these different frequencies is complicated in that they are most often not easily described in terms of integer ratios. Compared to harmonic sounds, such sounds can be considered complex. Good examples of these types of sounds are the wind, the sea, bangs, crashes, thuds and explosions. As such, they do not bear much relation to what some people think of as music, despite appearing in many forms of music all the time.

Importantly, human voices can produce sounds that have far fewer combinations of frequencies and amplitudes (i.e. are non-complex). In addition, the human voice can produce sounds that maintain relatively steady frequencies and amplitudes consistently. Furthermore, the human voice can produce sounds whereby the relationships between the component frequencies approximate relationships that can be described by integer ratios relatively easily. Over time, we have learned to make instruments that produce sounds with these three qualities for the purposes of making certain types of music.

Many people associate an organised series of non-complex sounds exhibiting relative stability in terms of frequency and amplitude, and where the frequencies are mostly describable using integer ratios, with that which is commonly thought of as music. Systems exist which help us to organise these types of sounds in terms of fundamental frequency, volume, complexity and timing. For more information on these systems, you should refer to Chapter 2 of *Creative Computing 2: interactive multimedia Vol. 2*.

Music traditionally features combinations of complex and non-complex sounds, occurring at various times in relation to some form of recognisable pattern or sequence. Most importantly, in this context, frequency and amplitude are not thought of in terms of Hz (hertz) and dB (decibels). Instead, they are thought of in terms of **note-pitch** and **dynamics**.

Importantly, some modern music does not feature any non-complex, pitched sound whatsoever, instead favouring patterned combinations of complex sounds. This is a more recent phenomenon, facilitated by the development of music and sound technology that can record, sequence and play back any sound of any type. This type of music making is exemplified by the work of Pierre Schaeffer and the approach he refers to as **musique concrète** (Schaeffer 1952).

Throughout this chapter, you will be encouraged to engage with complex and non-complex sounds both in terms of engineering and creativity. You should feel free to create sound and music in any way you choose given the criteria laid out in the exercises. On occasion, you may wish to use systems of pitched sounds (scales) in relation to some sort of timing and rhythm, in which case you should refer to Chapter 2 of *Creative Computing 2: interactive multimedia Vol. 2*.

2.4 Sound, music and computers

We can measure sound using any electrical device that is sensitive to vibrations. Microphones work by converting the movements of a diaphragm suspended in an electromagnetic field into electrical signals. As the diaphragm moves in response to changes in pressure, this causes similar changes in the electromagnetic field. These variations can be passed down a conductive cable where the variations can be stored in some electrical or electronic medium.

Computers take these electrical signals and convert them into a series of numbers. This process is called analogue to digital conversion (ADC – a standard term for the digitisation of audio signals). The analogue signal is measured a set number of times per second (see Section 2.4.1 below), and the amplitude of the signal is stored as a 16 bit binary number. The bit depth is sometimes greater, for example 24 bit in professional systems, although 16 bits is considered sufficient to represent signals within a theoretical 96dB range, with each bit representing a range of 6dB. Regardless of resolution, we normally encounter each measured value as a floating-point number between 1 and -1.

Crucially, a single sinusoid at maximum amplitude with a phase of zero will be measured as having an amplitude of zero at the start of a cycle. This sounds confusing, but it is a very important to remember. What you hear as pitched information is actually cycling amplitudes, and so you should always be clear that there is no absolute way to separate these two aspects of sound waves.

At 90 degrees (0.5π radians), the amplitude will be 1. At 180 degrees (π radians), the amplitude will be zero, and at 270 degrees (1.5π radians) the amplitude will be -1. Finally, at 360 degrees (2π radians) the amplitude will again be zero. In this way, from 0–180 degrees, the phase of the wave is positive, and from 180–360 degrees, the phase of the wave is negative.

These **samples** are stored in a continuous stream either in RAM or on disk. They are played back in sequence to reconstruct the waveform. The result is a staircased-shaped output (also called a piecewise constant output). This must be interpolated through the use of a **reconstruction filter** (see Section 2.4.2 below). The filtered signal is then converted back to an electrical signal before being sent to a speaker. The speaker cone is suspended in an electromagnetic field that fluctuates in response to the electrical signal. This makes the cone move back and forth in line with the amplitude of the waveform. The movements of the speaker cone cause air to compress and expand in front of the speaker at audible frequencies, turning the signal back into sound.

There are a number of issues that cause problems when representing analogue audio signals digitally. Two important issues are sampling frequency, and aliasing.

2.4.1 Sampling frequency

As can be seen above, a single sinusoid can be represented digitally by only two measurements – its maximum positive amplitude and its maximum negative amplitude. This is because at 0 and 180 degrees, the connecting line passes through zero amplitude as described above. The highest frequency that a human can hear is approximately 20,000 Hz. We can think of this as the upper boundary of the signal's **bandwidth**.

We know that to represent a wave with this frequency, two measurements are required. This means that the sampling frequency must be at least twice the bandwidth of the signal – 40,000 Hz. In practice, most digital audio systems have a sampling frequency of at least 44,100 Hz, with professional 'high-definition' systems sampling at rates as high as 192,000 Hz (commonly referred to as **over-sampling**). Crucially, if the sampling frequency is twice the bandwidth of the human ear, all audible frequencies can be represented.

2.4.2 Aliasing

If the sampling rate is less than twice the bandwidth of the signal, strange things happen. If a 200 Hz sinusoid waveform is sampled at a frequency of 300 Hz, 1.5 times the frequency of the signal, the resulting signal representation is lower than the original – by exactly 0.5 of its original frequency (100 Hz). The wave is effectively 'folded over' by the difference between half the sampling frequency and the frequency of the sampled signal. This is referred to as **aliasing** or **foldover**.

For these two reasons, when sound is sampled, it is usually filtered so that its bandwidth is exactly half the sampling rate. This prevents aliasing in cases where the sampling rate is not above the audible frequency range (which is not very often in modern contexts, but is important to understand). Aliasing often occurs during synthesis or sound manipulation, and filtering can be useful to help reduce its effects.

These issues are commonly referred to as the **Nyquist-Shannon Sampling Theorem**, after Harry Nyquist (1889–1976) and Claude Elwood Shannon (1916–2001). Additionally, the term **Nyquist Rate** is used to refer to a sampling rate that is twice the bandwidth of a signal, and the term **Nyquist Frequency** is always half this value. In almost all cases, the Nyquist frequency should always be above the maximum frequency you want to record, unless aliasing is a desired creative result. Please refer to Section 3.4.1 in *Creative Computing 2: interactive multimedia*

Vol. 1 for more information.

2.5 MIDI – Musical Instrument Digital Interface

Despite being originally commercially developed in 1969 by Peter Zinovieff, David Cockerell and Peter Grogono at London's EMS studio in Putney, digital audio systems remained prohibitively expensive until the early 1980s. As a result, most electronic musical instruments were built using traditional synthesis methods, by combining simple waveforms, occasionally with small amounts of sampled sound stored in ROM.

By this time it had been realised that a unified method for transmitting musical information would be useful for allowing electronic musical instruments to talk to one another. Before this, instruments used Control Voltage to transmit pitch information, with Gate signals to switch events on and off. Other systems in use included Roland's DCB (Digital Control Bus). This allowed for basic sequencing, but had severe limitations, particularly with respect to compatibility.

MIDI was first proposed in 1981 by David Smith, and became an industry standard communications protocol for the transmission of musical information in 1983. The fact that it is still in use today is testament to its excellent design and applicability.

MIDI does not transmit audio signals of any kind. It communicates performance events, synchronisation information and some system information. The most important performance events are note-on (which occurs when a note is played), note-off (when a note is released), velocity (occurs simultaneously with note-on to denote how hard the key is pressed), pitch bend and overall volume. In addition, MIDI controllers can be assigned to a large number of sliders for a wide range of functions.

Almost every MIDI message is contained in a single byte, and uses 7 bit resolution. This means that almost all MIDI messages contain data values between 0 and 127, with 0 being equivalent to 0000000, and 127 being equivalent to 1111111. Each note is assigned a particular number between 0 and 127. Table 2.1 shows 8.176 Hz (C five octaves below middle C) as MIDI note 0, and 12,544 Hz (G five octaves above the G above middle C) as MIDI note 127. The frequency of a midi note number, $f(p)$, can be calculated using Equation 2.1.

$$f(p) = 2^{\frac{p-69}{12}} \cdot 440 \tag{2.1}$$

Note velocity ranges from 0 to 127, with a velocity of 0 indicating a note-off message. In addition, there are 16 MIDI channels, all of which can send and receive almost all types of MIDI data simultaneously.

There are cases where MIDI information is contained in two bytes, with a resolution of 14 bits, but these will not be used in this chapter. For more information, consult the MIDI Manufacturers Association at `http://www.midi.org/`

2.6 Using Pure Data

As it cannot be assumed that you will have had any experience with Pure Data, what follows is an extremely simple introduction to using the software, suitable for

MIDI note number	Pitch	Frequency (Hz)
0	C-1	8.176
12	C0	16.352
24	C1	32.703
36	C2	65.406
48	C3	130.813
60	C4	261.626
69	A4	440.000
72	C5	523.251
84	C6	1,046.502
96	C7	2,093.005
108	C8	4,186.009
120	C9	8,372.018
127	G9	12,543.854

Table 2.1: The relationship between MIDI note numbers, pitch, and frequency.

anyone who has never had any experience of programming real-time sound and music software before in PD.

Pure Data is based on the Max programming language (Puckette 1988, 2002). It is a free, open source sound, music and multimedia visual programming environment. The version used to generate these examples is PD-0.41.4-extended, and it can be obtained from `http://www.puredata.info/`

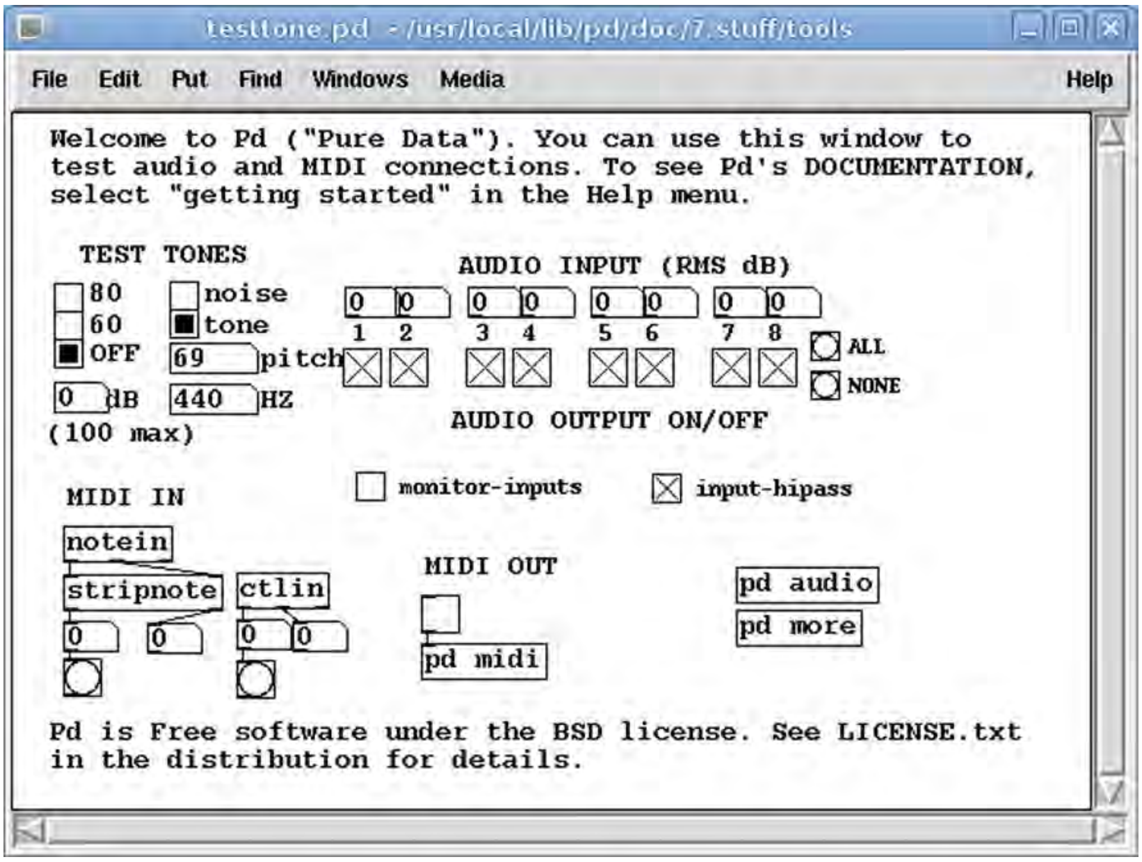
When you launch Pure Data, you will notice a window appears on the left hand side called PD. This window gives you feedback on how the PD system is running. You can use it to test functions, and to print information. In addition, any errors that occur will be posted in this window.

2.6.1 Testing your system

Before we begin you should check that your soundcard is correctly configured. To do this, locate the Media menu, and select 'Test Audio and Midi'. A new window will open. This is a 'patcher' window. You will notice that the patch is called 'testtone.pd'. At the left of the patch you will find some controls under the label TEST TONES (see Figure 2.1). Click in the box labelled '60'. You should hear a test tone. If you do not hear anything, check that your system volume is not muted, and that your soundcard is correctly configured before trying again. If there is still a problem, go to Audio Settings, and make sure that your soundcard is selected.

If you have a MIDI keyboard attached to your computer, press some keys down. You should see numbers appear under the section labelled 'MIDI IN'. If you do not, try checking that your MIDI device is selected under MIDI settings, and that it is correctly installed.

Now that you have tested your settings, you can close the testtone.pd patch. This is important, as PD patches will run in the background, even if you cannot see them.



Welcome to Pd ("Pure Data"). You can use this window to test audio and MIDI connections. To see Pd's DOCUMENTATION, select "getting started" in the Help menu.

Pd is Free software under the BSD license. See LICENSE.txt in the distribution for details.

Figure 2.1: testtone.pd from Pd version 0.42-5 (Linux)

2.6.2 Getting started

To begin, choose file → new. This will create an empty patch. You can resize the patch window by clicking the bottom right corner and dragging the mouse.

2.6.3 Edit mode

You should notice that when you click the mouse within the patch area, the hand pointer appears. This means that you are in edit mode. In order to write and edit your program, you need to be in Edit mode. In order to use your program, you need to exit Edit mode. You can switch in and out of Edit mode by finding the Edit menu, and choosing the Edit mode option. When you do this, make a note of the keyboard shortcut, as you will be doing it a lot.

2.6.4 Number boxes

Make sure you are in Edit mode and that your patch is selected. You can do this by clicking on it.

Locate the ‘Put’ menu and choose ‘Number’. Take a note of the shortcut key. You should find that a box with a zero in it has appeared in your patch. This is a number box. Leave Edit mode and click in the number box with the mouse. You should find that when you drag up and down, the number in the box changes.

Now press the shift key whilst dragging. You will notice that a decimal point has appeared, and that you are now changing the numbers to the right of the decimal point. If you stop and let go of shift, you should find that when you click and drag, the whole numbers are now being changed, whilst the numbers after the decimal point are staying the same.

Finally, try typing numbers into the box. You will need to press the enter key for the numbers you have typed in to have an effect.

Number boxes (or **Atoms**) handle integers and floating point values. They can be used to enter data or to display results. If you right click on the number box and select properties, you will notice that the number box can have predefined limits. It can also have a label, send and receive messages. We will deal with this in more detail later.

2.6.5 Messages

Enter Edit mode and choose Put → Message, taking note of the shortcut key. You will notice that a new box has appeared, similar to the number box, but with different shaped corners on the right hand side. This is a **Message** box. Type the number 9 into the message box. When you have finished, click outside the message box.

Try grabbing the message box with the mouse. You should be able to move it around. Position it above the number box. You will notice that both the message box and the number box have small rectangular inputs and outputs on top and underneath. These are called **inlets** and **outlets**.

Make sure your patch is active (by clicking on it), and move your mouse over the outlet at the bottom of the message box. You should see a small black ring appear. Click and drag from the message box's outlet to the inlet on the top of the number box. You should see a thin black line appear. Let go of the mouse button just above the number boxes inlet. You should see something that looks like this (Figure 2.2).

Now leave Edit mode and click on the message box. You should see that the message has sent the data '9' to the number box. Now change the number in the number box, then click the message box again.

Messages are one way of storing and recalling data in PD. You can type long lists of messages into message boxes with the data separated by spaces. You can also separate messages inside a message box with commas.

2.6.6 Objects

Most of the time, you will use number boxes and message boxes to send data from place to place. However, they do not handle or manipulate data. To do something useful with data, you need to use a function. In PD, functions are performed by **objects**. PD has many hundreds of objects, each which performs different functions.

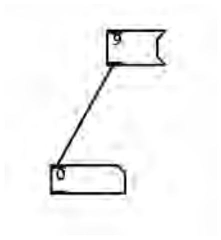


Figure 2.2: Message and number boxes

Objects are functions that take inputs and generate specific outputs. They can also be told how to behave through the use of messages.

Enter Edit mode and choose Put → Object, taking note of the shortcut key. You will see a square with a dashed blue outline. Position this in some whitespace and type 'print'.

Now click outside the object box. Move the [print] object underneath your message box, and above your number box.

Move your mouse until it is over the connection between the message box and the number box. You will see that a black cross has appeared. Click the mouse button. You will notice that the line has gone blue. Now, press backspace to delete your previous connection.

Make a new connection between the message box and the [print] object. Click inside the message box until the number '9' is highlighted. Type 'Hello World' and exit edit mode.

Look in the PD window and click the message box. You should see the words 'Hello World' in the PD window. You have written your first program.

Print is an object that prints any kind of text-based data to the PD window. This is useful for tracking down errors, or just finding out what things do. If you are trying to understand what an object does, it is always worth connecting a print object to it to have a look.

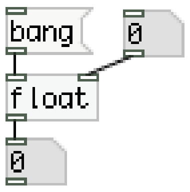
Another way of learning what an object does is to right click on it and select 'help'. This will bring up the object reference patch, which is a great way of getting extra information. You can also copy and paste patches or sections of patches. This makes building patches much easier.

We are going to start by using a few simple objects to store information and count numbers. We are also going to look at how we organise when information gets processed.

2.6.7 Storing and counting numbers

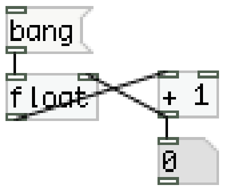
You do not need to use messages to store numbers. You can store a number in an object and recall it later. This is useful if you want to change the number every so often, something you can not do so easily with a message.

Make the patch below. Notice that 'bang' is a message, not an object.



Try changing the number box attached to the right inlet of the [float] object. Remember to exit Edit mode to interact with your patch. You should find that nothing happens. This is because in most cases, PD objects only produce output when they receive something in their leftmost inlet – the 'hot' outlet. Now press the 'bang' message. You should see the result in the number box at the bottom.

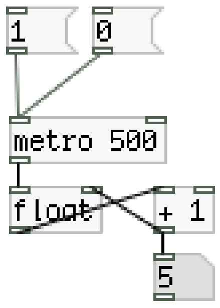
Now make the following changes to your patch until it looks like this. There is a new object here, called [+]. This adds a number to any number it receives in its hot (leftmost) inlet. In this case, it has been initialised with a value of one. This value would be changed if a number were sent to the right inlet of the [+] object.



Now click on the bang message repeatedly. Notice what happens. The number goes up by one each time you click 'bang'. The [float] object initially produces a zero when it gets banged. This goes into the [+] object, which sends the value 1 to the number box, and also to the 'cold' inlet of the [float] object. The next time you click bang, the [float] object spits out the 1 that was stored in its cold inlet, and sends it to the [+] object, which adds one, and sends the result (2) to the number box and the cold inlet of [float] for storage. What would happen if a number were sent to the right (cold) inlet of the [+] box? Try it.

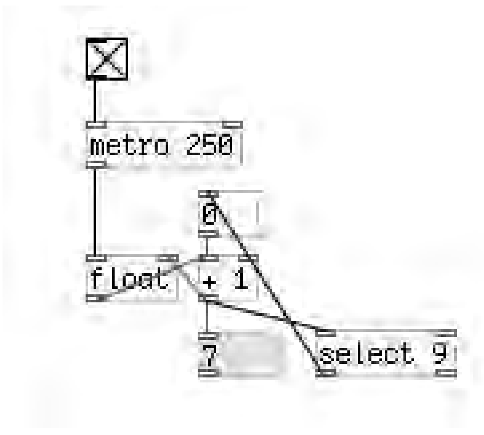
Bang is a special type of message that is very important in PD. Bang is so important that it even has its own user interface object – the bang button. You can create a bang button by choosing Put → Bang. Clicking on this button has exactly the same effect as clicking on a message with the word 'bang' in it. You can test this by connecting it to the left inlet of your [float] object and clicking it, or by using the [print] object, or both.

There is another special object called [metro] which produces bangs at regular intervals, specified in milliseconds. You can set the size of the time interval with either an initialising argument, or a number in the right inlet. [metro] starts when it receives a 1 (on) in its left inlet, and stops when it receives a 0 (off).



You can also use a special object called a **toggle** to start and stop [metro]. You can create a toggle by choosing Put → Toggle. When toggle is checked, it sends a 1 out its outlet. When it is unchecked, it sends a zero.

Another important object that sends out bangs is [select]. You can use select to make things happen at specific times, such as reset your counter. Here select sends a bang when the count reaches 9. This resets the next stored value to 1 by immediately sending 0 to the [+] object, which in turn adds 1 and sends it to the [float]. This means that the counter goes from one to eight.



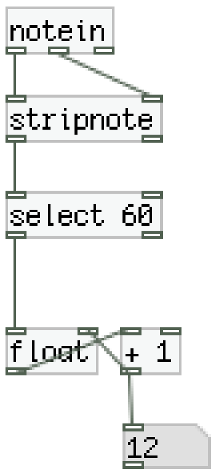
[select] can have more than one argument. This means you can use one [select] object to make lots of different things happen at different times. This is the simplest method of sequencing events in PD.

2.6.8 Using keyboard events

Another way of making things happen in PD is to generate events. You can grab events from the computer keyboard or from a MIDI device. The easiest way to do this is with the [key] object. This reports the key code for any key you press. You can use [select] to generate a bang when a certain key is used.

2.6.9 Getting MIDI events

If you have a MIDI keyboard attached, you should be able to use MIDI to send control messages to PD. [notein] receives pitch and velocity pairs when any keys are pressed on the MIDI keyboard. Be aware that MIDI note messages come in two types – note-on and note-off. If you want to use the MIDI keyboard to make things happen, you may want to strip out the note-offs with the [stripnote] object. Here is an example where the counter increments each time note number 60 is played (middle C). See if you can find it.



There are a number of important MIDI objects. In addition to [notein], the two objects which are of most use when starting out are [ctlin], which takes controller input from MIDI controller sliders, and [bendin], which converts data from the pitch-bend wheel to values between 0 and 127.

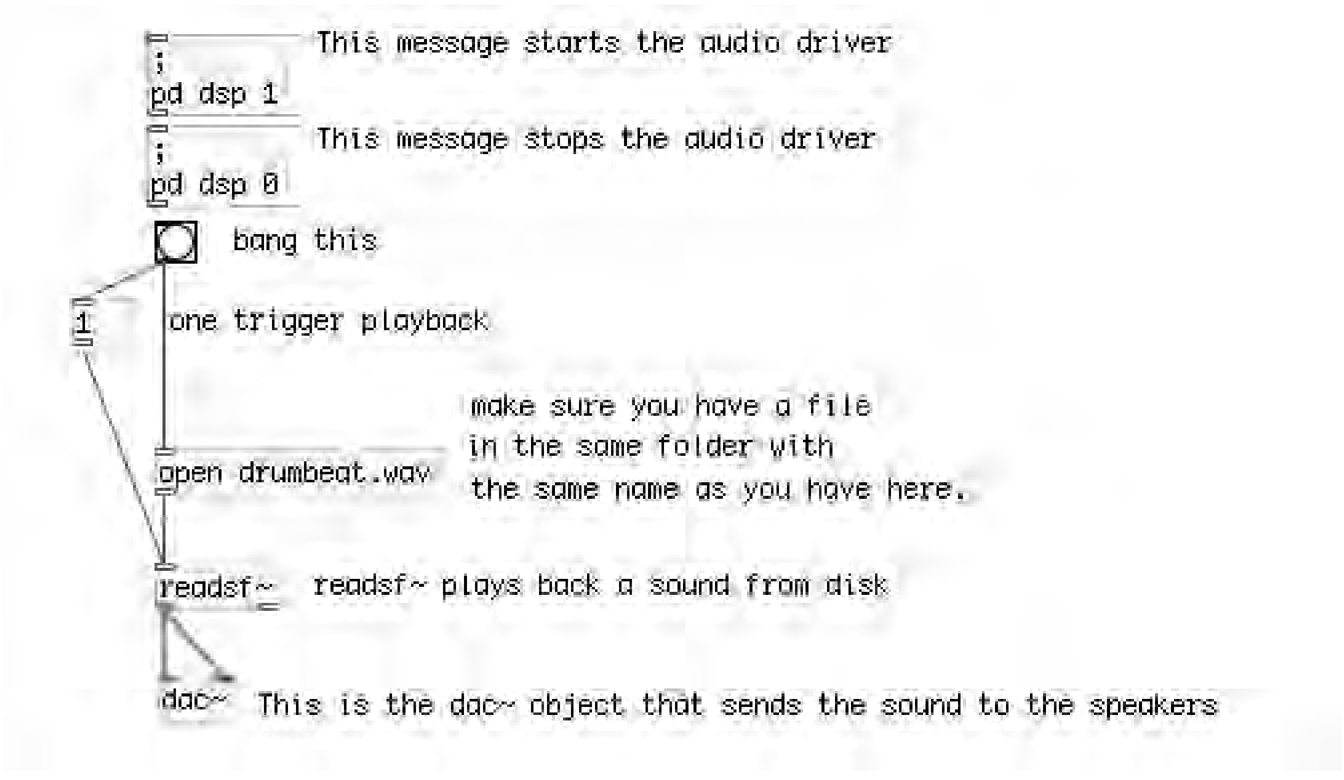
2.6.10 Signal objects

Basic PD objects come in two different types. Standard objects, like [metro], [select] and arithmetic objects ([+],[-],[/],[*],[SQRT] etc.), and signal objects. Signal objects are different from standard objects in the same way that DSP and MIDI is different. Standard objects handle control data. This data represents events that are occurring. However, signal objects pass around and modify actual sound signals. In this way, PD can be used to create any number of samplers, synthesisers and effects processers that can generate and modify sound and music in real-time.

In PD, signal objects are indicated in two main ways. First, signal object names always have a tilde (˜) character appended to their name. This is good to remember, as it looks like a little wave. Secondly, signal object connections are slightly thicker than standard object connections. In addition, it is worth remembering that many PD objects have signal variants that look identical apart from the tilde symbol. For example, you can multiply sounds by using the [*˜] (times tilde) object.

2.6.11 Playing back sounds from disk

The easiest way of playing back sounds from disk is to use the [readsf˜] object. The patch below demonstrates the simplest method for doing this. In order for [readsf˜] to work, you should save your patch in a folder with some uncompressed .wav files. In the example below, the patch has been saved to a folder containing a sound file called ‘drumbeat.wav’.



```
;
pd dsp 1          This message starts the audio driver
;
pd dsp 0          This message stops the audio driver
bang this
1    one trigger playback
                  make sure you have a file
                  in the same folder with
open drumbeat.wav the same name as you have here.
readsf~  readsf~ plays back a sound from disk
dac~  This is the dac~ object that sends the sound to the speakers
```

Learning activity

Build a simple sequencer that plays back a selection of sounds from disk at different times. Use [metro], [select], [float], [+], [readsf˜], plus any other objects you think you need.

Advanced exercise to test your deeper understanding of the topic:
Try to trigger some of the sounds from either the computer keyboard or a MIDI device.

Learning activity

The following are some questions for revision, that you should attempt and use to consolidate, before studying further:

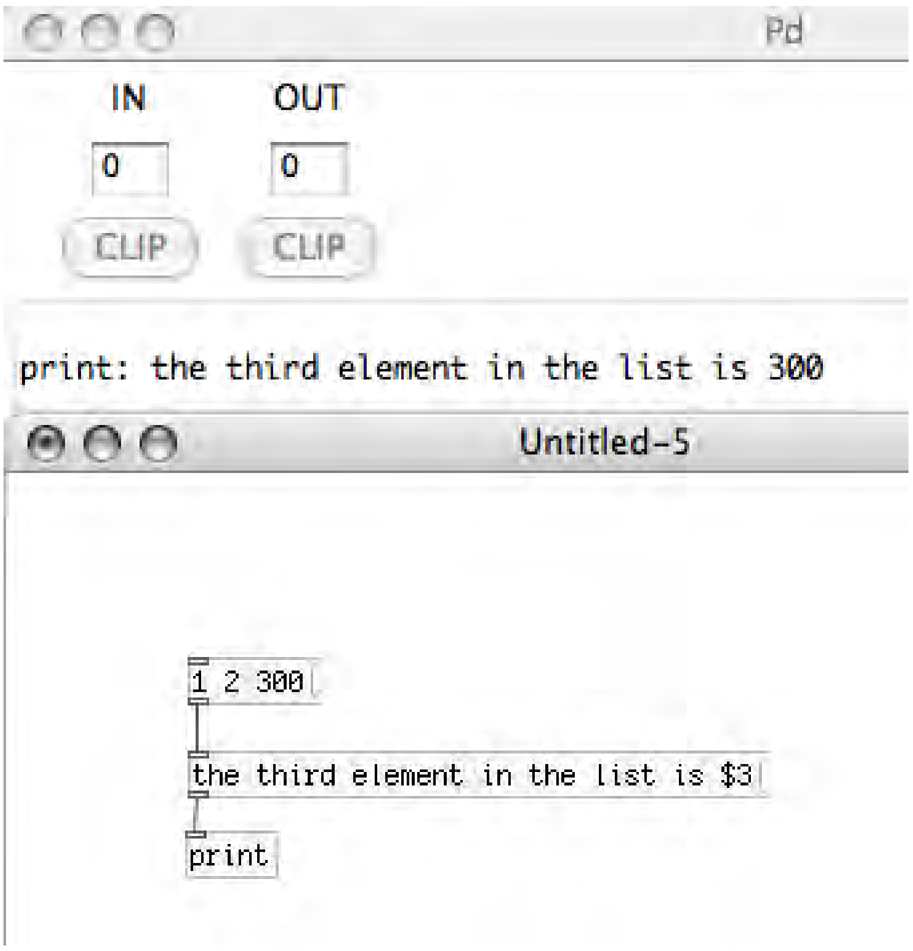
- What is a message?
- What is an object?
- Name two types of data
- What does bang do?
- What is the output of a toggle?
- What is the difference between a hot and a cold inlet?
- Name two fundamental properties of acoustic sound and describe how they are measured.
- What is the difference between the Nyquist Rate and the Nyquist Frequency?
- A frequency of 500 cycles per second is sampled at a rate of 600 samples per second. What frequency will be recorded?

2.6.12 Lists and arrays

Lists

Sometimes you need to store and recall more than one number. Lists of numbers can be written into a message box. We can retrieve individual numbers from a list in many ways. The easiest way is to use dollar signs ($).

When you place a dollar sign in a message box, followed by an index number, it acts as a placeholder. The dollar sign is then replaced by any incoming data with that index. For example, if a list has three elements, and you send the list to a message box with '$3' written in it, it will spit out the third element in the list, as in the example below. Notice the printed output in the PD window.

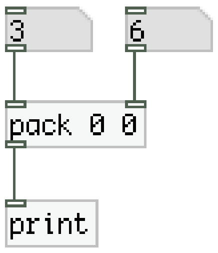


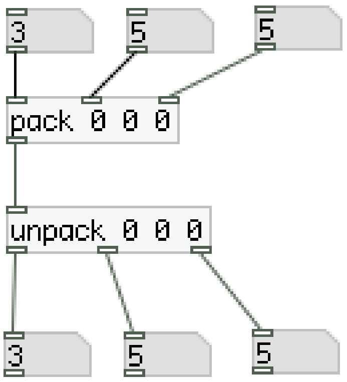
```
print: the third element in the list is 300
```

```
1 2 300
the third element in the list is $3
print
```

This is a great method for getting numbers and words out of lists. However, it is not very efficient if you need to change numbers in a list dynamically. Luckily, there is a special object called [pack] that allows us to create dynamic lists.

[pack] takes initialising arguments that define the starting values for all elements in the list. The object [pack 2 4 6 8 10] creates a list with 5 elements that are initialised with the values 2 4 6 8 and 10 respectively. The object will therefore have 5 inlets, one for each element in the list. Importantly, **only the left inlet is hot**. This means that although data arriving at the inlets will change corresponding values, they will not be output unless inlet 1 (the leftmost inlet) either receives a number or a bang message.





You can see in the above example that lists created by [pack] can be dynamically unpacked using the [unpack] object. This is a great way of building and accessing numerical data in lists effectively. However, it must be noted that the [pack] and [unpack] objects only handle numerical data. If you want to make a list using text, you should use the dollar sign method.

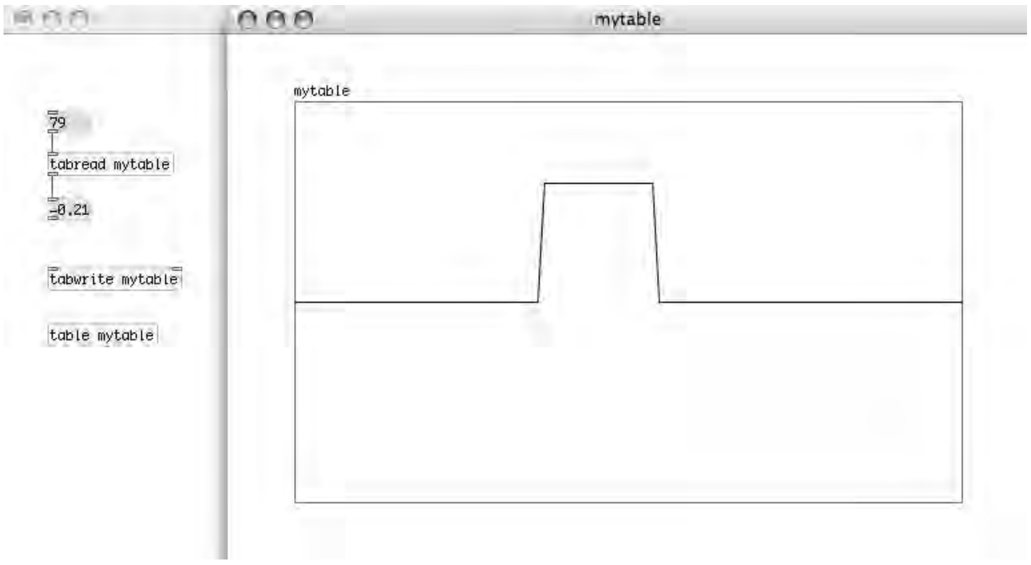
Using arrays

You can store numbers in an array and recall them by using an index.

Arrays are stored in [table] objects, and can be filled with data by using the [tabread] and [tabwrite] objects. You can also put data in an array by clicking and dragging in the display with the mouse.

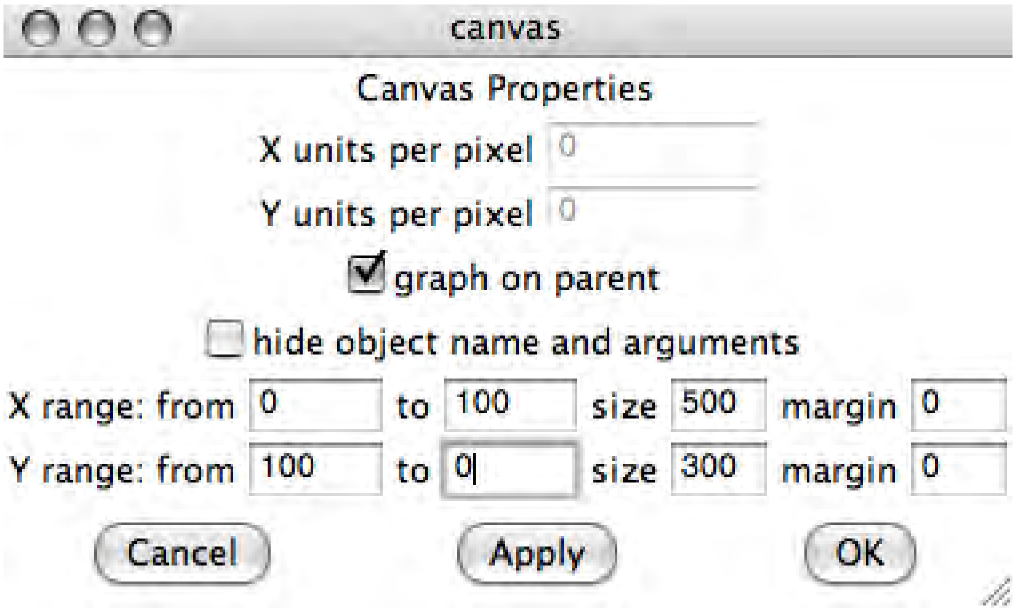
Create a [table] object, remembering to give it a name as an argument, like [table mytable]. Also create a [tabread mytable] object and a [tabwrite mytable] object. The argument names are very important. They link the [tabread] and [tabwrite] objects to your array which is now stored in the [table] object. Now create two number boxes, and connect them to both the inlet and outlet of the [tabread mytable] object. The number in the top of [tabread] can be used to ask for the value that is stored at a particular index in the array.

Now click on the [table mytable] object. This will open a window with the array inside. You can enter values into the table by clicking on the line and dragging with the mouse.



Now use the number box connected to the inlet of the [table mytable] object. You can see the result appear in the number box below.

By default, arrays in a [table] object can store up to 100 floating point values on its x-axis. These numbers can be any floating point value, but by default, the graph will only contain numbers between 1 and -1. If you want to change the size of the array, you can do so easily by right clicking on it and selecting 'properties'. This brings up two windows, one called 'array' and one called 'canvas'. Using the canvas window, change the y-axis range to values between 100 and 0. It should now look like this:


canvas

Canvas Properties

X units per pixel 0

Y units per pixel 0

✓ graph on parent

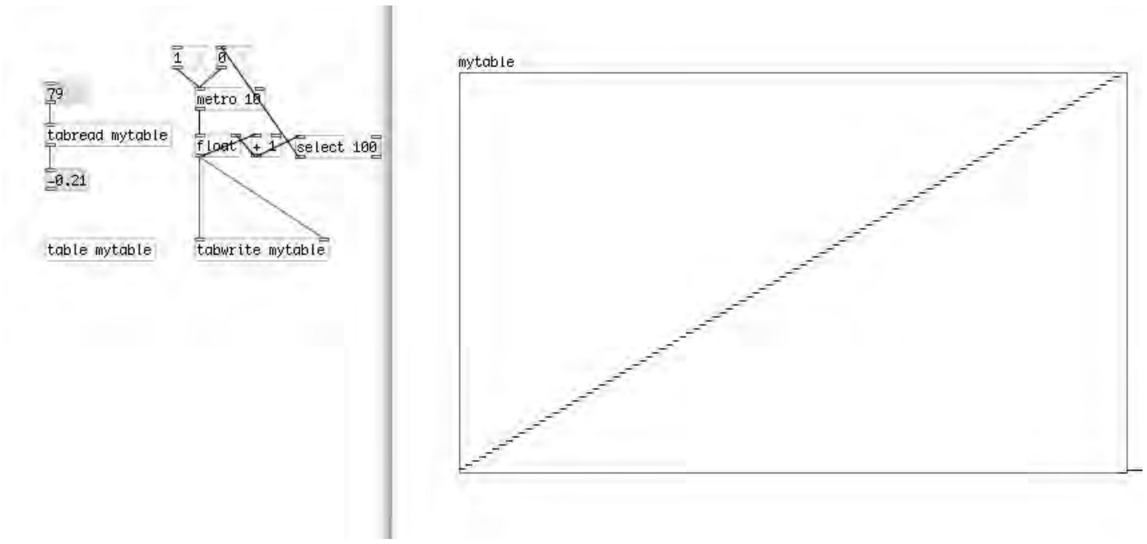
☐ hide object name and arguments

X range: from 0 to 100 size 500 margin 0

Y range: from 100 to 0 size 300 margin 0

Click 'Apply', and then 'OK'. Draw some values into the table with the mouse. You should find that the range of data that the table can display is now much greater.

We can fill an array with data by using the [tabwrite mytable] object. [tabwrite] has two inlets. The left inlet takes the value you want to store, and the right inlet takes the index. We can fill the array with values by using a counter. In the example below, the counter is used to create a stream of numbers between 1 and 100. When it reaches 100, it stops. These numbers are used as both the value and the index. This creates the diagonal line we see displayed in the table.



Copy this example and press the ‘1’ message connected to the [metro] object. You should get the same result. You will then be able to retrieve the values using [tabread].

You do not need to use [table] to create an array. If you like, you can just go to the Put menu and select ‘Array’. You can set the size and properties of the array in exactly the same way, and arrays created in this manner will still work with [tabread] and [tabwrite] objects.

Learning activity

Arrays are a great way of making music in PD. Can you think of a way of using an array to play back a sequence of events? Modify your previous sample player so that it uses an array to play back your sounds. Hint: you will need a counter and a [select] object, just like before, but they may need to be configured differently.

2.7 Sound synthesis

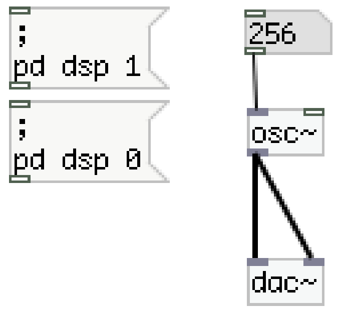
One of the best ways of making sound and music in Pure Data is through sound **synthesis**. You can use PD’s basic synthesis components to carry out any number of sound creation tasks, using a variety of different methods. In this section, you will learn how to create synthesised sound of varying complexity by emulating well-known synthesis techniques, some of which have been used in sound and music for over 100 years.

2.7.1 Oscillators and waveforms

All sounds are made up of combinations of sinusoids (sine waves). It is therefore theoretically possible to synthesise any sound by combining sine waves with constantly varying amplitudes. This form of synthesis is known as **Fourier** synthesis, after the mathematician and physicist, Joseph Fourier (1978–1830). Fourier synthesis is often referred to as **resynthesis**, and normally requires an analysis technique known as **Fourier Analysis** to be performed first. For more information on Fourier analysis see Section 4.3.3 of *Creative Computing 2: interactive multimedia Vol. 1*, and Section 2.3.1 of *Creative Computing 2: interactive multimedia, Vol. 2*.

A far easier way of creating sounds is to start with waveforms that model the basic qualities of natural sounds. In reality this is incredibly difficult to do, as the number of frequencies and their varying amplitudes can be deceptively difficult to model. However, by combining waveforms in interesting ways, we can create a number of excellent sounds, many of which are impossible to produce in the natural world.

In PD, the most basic synthesis object is the sine wave oscillator, or [osc˜]. This object creates a test tone with a frequency determined either by an argument, or by a floating-point number in its left inlet. Connect an [osc˜] to both inlets of a [dac˜] object to ensure that sound will come out of both speakers. Connect a number box to the left inlet of [osc˜] and start the DSP using the message {;pd dsp 1]. You can just copy and paste this from your sampler patch if you want. You should end up with something like this.

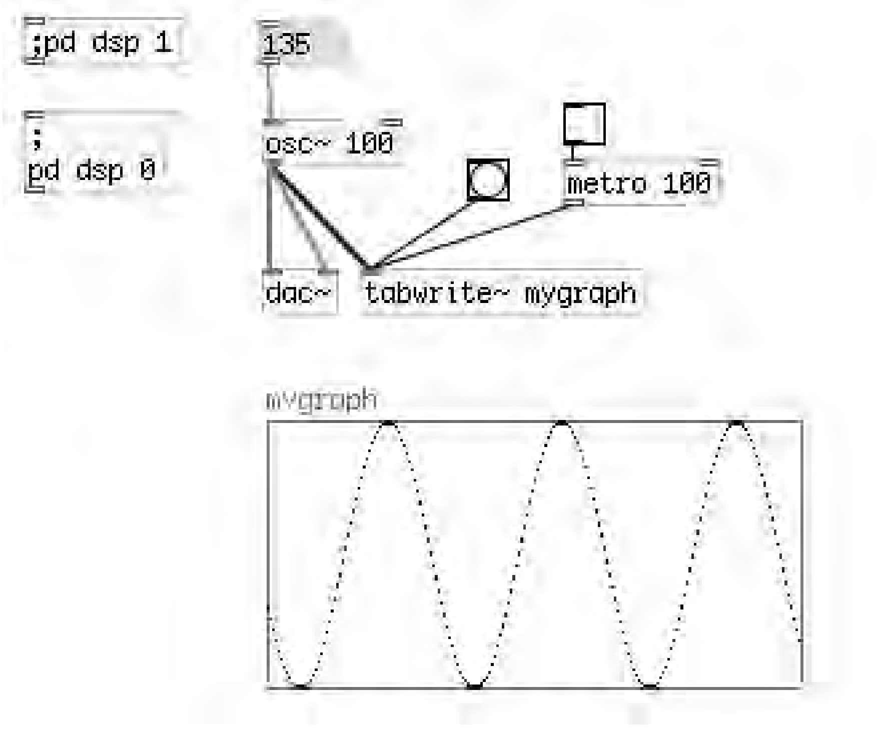


Use the number box to change the frequency and listen to the result. You should be able to tell that as the frequency increases, the tone of the oscillator gets higher in pitch.

All musical pitches relate directly to specific fundamental frequencies. For example, concert A, the note that an orchestra most often uses as a reference, is equal to 440 Hz. Similarly, middle C (MIDI note number 60), normally considered the middle of a piano keyboard and staff system, is equal to 261.626 Hz.

At this point it would be great if we could graph the sounds we are making. We can use an array to do this. All we need to do is create an array, and send our sound signals to an object called [tabwrite˜]. Notice that this object is essentially the same as the [tabwrite] object, but for signals. Do not forget to name your array.

When [tabwrite˜] receives a bang, it will graph the signal going into it. Create a bang button and connect it to the input of [tabwrite]. You can automate the bangs with a [metro] object, but take care not to send the bangs every millisecond, as this will waste valuable CPU power. A timing interval of around 100 should be fine. Now your patch should look like this:



Here, I have changed the dimensions of the graph array to make it smaller. I have also increased the number of values that it displays on its x-axis from 100 to 1000. Note: you must change both the size of the canvas and also the size of the array, otherwise it will not work!

Notice that when you change the frequency of the [osc˜] object, the width of the waveforms in the display changes accordingly. You can see that lower frequencies are wider than higher frequencies. This is because lower frequencies have longer wavelengths.

We will continue to use this graph throughout the rest of the examples. We may also occasionally change its parameters to allow us to view sounds differently.

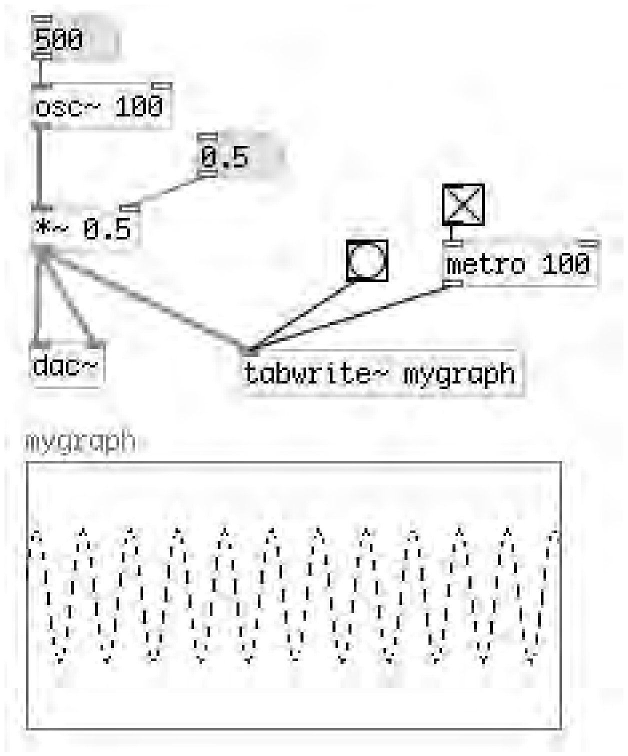
2.7.2 Amplitude

So far we have learned how to control the frequency of an oscillator and graph its output. However, we have as yet no way of controlling the amplitude of the oscillator.

By default, all oscillators output their signal at maximum possible amplitude. We can see this by looking at the graph, as we know that it is displaying values between 1 and -1, and the oscillator is peaking at the top and the bottom of the graph.

We need a method of lowering the amplitude of the signal. Most people imagine that if we subtract a value from the signal that it will lower its amplitude. This is not the case. We could use a [-˜ 1] object to do this, but this would have the function of changing the signal from an oscillation between 1 and -1, to an oscillation between 0 and -2. This can be very useful. However, it would be unwise to send such a signal to a speaker, as they are incapable of responding to signals beyond 1 and -1 for prolonged periods without being damaged.

When we want to change a signal's amplitude, we are trying to scale the values it contains. The best method of scaling a range of values is to multiply them. For this reason, we should use the [*˜] object to reduce the amplitude of a signal, as in the example below.

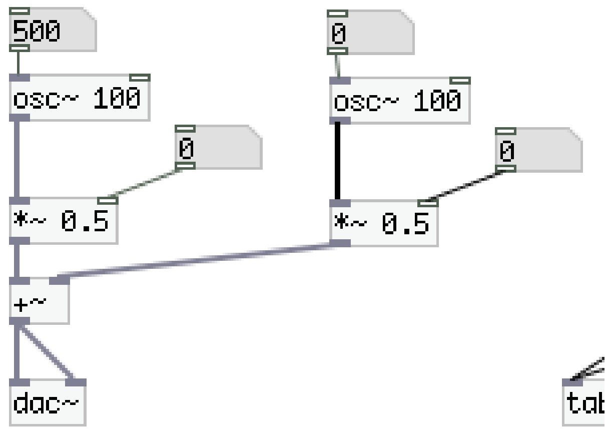


Try altering the value in the number box attached to the [*˜] object by shift clicking and dragging. Notice what happens to the signal. At this point, you may have unintentionally multiplied the output signal by a value greater than one. This is not a great idea. If you start to panic, just hit the {;pd dsp 0} message.

There are times when multiplying a signal by greater than one is a good thing to do. However, you should avoid doing this to a signal that is being sent to a speaker unless it has already been turned down somewhere else. We can mitigate against this sort of problem by using the [clip˜] object with the arguments -1 and 1 to constrain the signal between these two values.

2.7.3 Adding signals together

When we add two signals together, they combine in interesting ways depending on their frequency content. Add a second oscillator to the patch, but do not connect it to the [tabwrite˜] object just yet. Try to think of a way of adding the oscillators together. If you manage to do this correctly, that section of your patch should look something like this.



Notice that both oscillators are multiplied by 0.5 before being added together. This is important, as when we add sounds together their amplitudes are summed, so the output oscillates between 2 and -2. Therefore we must scale them first. Here we have scaled them both by the same amount, halving their values so that when we add them together with the [+˜] object, the output oscillates between 1 and -1. Also notice that the output of the +˜ object is going to both inlets of the [dac˜] so we can hear the sound in both speakers.

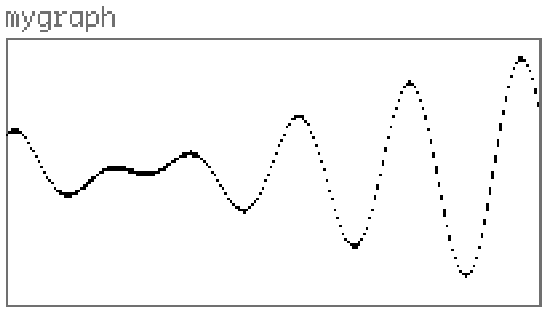
Now, change the frequencies of both oscillators to 200 Hz. You should hear one tone. Now, change one of the oscillators to a frequency of exactly 201 Hz and listen to the effect. What is happening? You may have expected to hear two separate tones, but you do not. This is because there is very little difference between a signal of 220 Hz and a signal of 221 Hz. However, there is enough of a difference to create a rather strange effect.

Beating

You should hear that the tone is fading in and out. Look at a clock or watch with a second hand and work out how often the sound fades in and out. You should hear it happening once a second. This interval is called the 'beat frequency'. When two tones of similar frequency are combined, they 'beat' together, noticeably interacting to create a third 'beat' frequency. The beat frequency is always equal to the difference between the two summed frequencies. In this case the difference is 1 Hz – one cycle per second.

Connect the outlet of the [+˜] object to the inlet of the [tabwrite˜] object and take a look at the graph. You should be able to see the changing amplitude created by the frequencies beating together. Now, slowly change the frequency of the second oscillator from 221 to 222 Hz (use the shift key). The beat frequency will change from 1 Hz to 2 Hz.

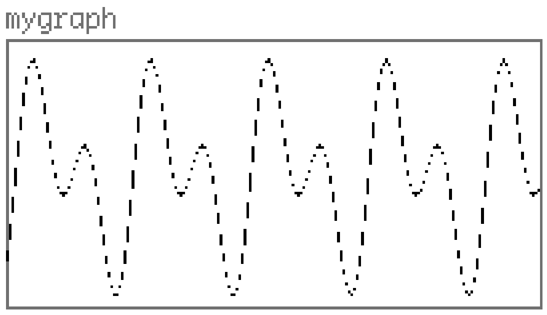
Now, slowly increase the frequency once more, 1 Hz at a time until you get to 240 Hz. You should now hear the frequency getting higher in pitch, as you would expect. However, you should also notice that the beat frequency is increasing. At 240 Hz, the beat frequency is 20 Hz, and should look something like this.



20 Hz is the lower boundary of the human hearing range. Beyond this point, depending on the quality of your speakers, you should begin to hear the beat frequency as a third audible tone increasing in pitch as the frequency rises. This is called **heterodyning**. The audible tones caused by beating are also known as **difference tones**, as they are equal to the difference between the two frequencies. Now increase the frequency further, slowing down as you approach 440 Hz.

Harmonics

Just before you reach 440 Hz, you should be able to hear some interesting changes in the audio signal. You will hear the beating again, but this time, you will not hear much change in the amplitude. Instead, the quality of the sound changes. Stop at 439 Hz, and listen to the speed at which the changes occur. It is 1 Hz – once per second. Now look at the graph. You should be able to clearly see the changes in the oscillation of the sound. Finally, move from 439 to 440. The beating will stop, and the shape of the waveform will stabilise. It should look something like this.



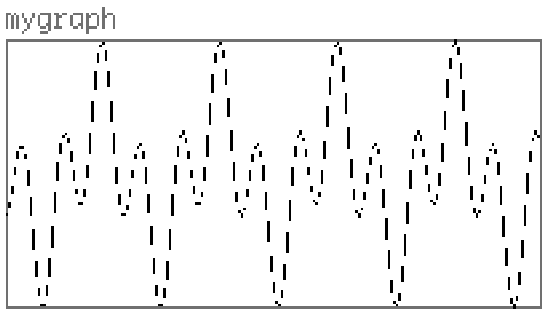
Notice that the waveform has two peaks, and that these peaks are equally spaced. This is because the second oscillator is producing a frequency that is exactly twice that of the first oscillator. They are in a mathematical relationship that can be described in terms of integer ratios, combining to produce what we call a **harmonic** sound. Furthermore, because the second frequency is a multiple of the first, we can refer to it as a **harmonic**. More specifically, as the second frequency is exactly double the first, being the first multiple, we refer to it as the **first harmonic**.

The pitches produced by both waveforms sound similar. You may not actually be able to tell that there are two pitches at all. Technically, it could be argued that they are of the same pitch class, and if we were to play them on a piano keyboard, they would both be the note A, spaced exactly one **octave** apart. In MIDI terms, they are note numbers 57 and 69. The difference between 57 and 69 is 12. This is the number of musical notes in an octave in the Western music tradition. Importantly, this is only the number of notes in this particular case, other musical traditions divide the octave into different numbers of notes.

Western musical scales do not normally feature all twelve notes of the system, instead traversing each octave in eight steps, with the eighth note being of the same pitch class as the first. Crucially, the Western music tradition is based on the premise that a doubling of frequency is equal to an octave, with the first and eighth note being of the same type. The fact that you may not be able to detect these two pitches as separate is testament to the power of the pitch class system.

The harmonic series

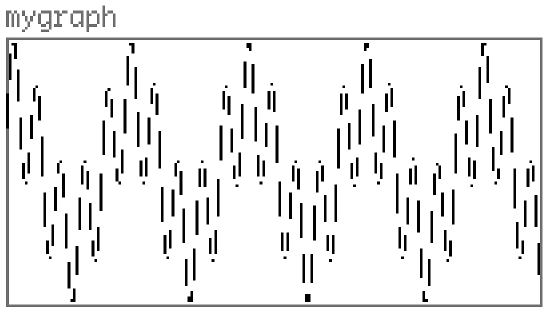
If we now multiply the first frequency by three, we will get the second harmonic – 220 * 3 = 660. Notice the affect this has on the waveform, and try to pay attention to the sound.



This waveform shares some characteristics with the previous one. Again, there is no beating, the sound is harmonic and the signal is stable. However, you can see from the waveform graph that the combined signal has three regular peaks. This is significant, as what we are hearing is the fundamental frequency, combined with the second harmonic. Also, you can tell that there are two distinct pitches of different types, whereas before, even though we knew there were two frequencies, it only sounded like a single pitch.

The second harmonic produces a pitch equivalent to the musical interval of a fifth above the first harmonic. In this case, the second harmonic is almost equivalent to MIDI note number 76, or the note E. From this point on, the exact mathematical relationships differ in small ways from the Western musical tuning system for reasons explained elsewhere (See Section 2.3.2 *Creative Computing 2: interactive multimedia, Vol. 2* for more information.)

If we add together the fundamental and the third harmonic, you will see that there are four peaks in the waveform, and the pitch will be two octaves above, being exactly four times the fundamental. If we add the fourth harmonic instead, the waveform has five peaks, and the interval is almost exactly what is referred to in the Western music tradition as a **major third**.



Here you can see clearly why the peaks in the waveform increase as we ascend through the harmonic series. In this case, the fourth harmonic has a frequency exactly five times that of the fundamental, so for every cycle the fundamental makes, the fourth harmonic makes five.

Importantly, if we combine the pitch class results from the first four notes in the harmonic series inside one octave, the result is a major chord, which is found in many musical systems throughout the world. In this way, we can think of musical sounds as those that mainly contain frequencies that are close to or exactly mathematically related, and many real-world instruments create sounds that are mainly composed of harmonics. However, the effects of beating are quite beautiful in their own right, and could be described as highly musical despite their complexity.

Exercises:

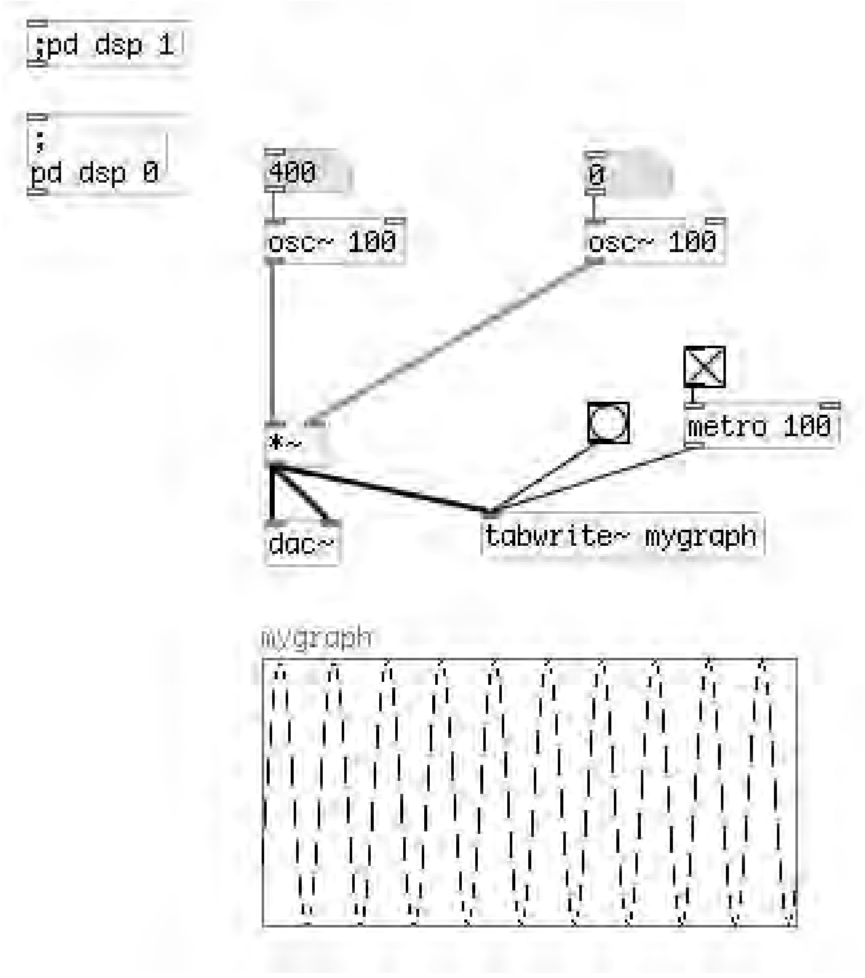
Create a patch that repeatedly ascends through the first 10 harmonics of a 100 Hz Fundamental.

Create a patch that plays the fundamental and first four harmonics, so that the fundamental can be dynamically changed whilst the harmonics remain in ratio. Hint: use [*] to set the values of several [osc~] objects.

Try loading sounds into [readsf~] and graphing them in your array. Look at the waveform and see if you can spot when they have mathematically related frequency content. It will be easiest to do this with recordings of solo acoustic instruments, particularly woodwind instruments like flute and/or clarinet. Alternatively, have a look at the sound of someone singing. What can you tell about it by looking at the waveform?

2.7.4 Multiplying signals together

Instead of adding signals together to synthesise sound, we can multiply them. Doing this is normally referred to as **modulation**, and it is used for all sorts of reasons. Create a patch with two [osc~] objects. This time, instead of scaling and then adding them together, just multiply them with a [*~] object and send them to the [dac~] and the array so we can see the waveform. Your patch should look like the one below.



Amplitude Modulation Now you should set the frequency of the first oscillator to something reasonably high, such as 400, otherwise the affect might be difficult to hear. Now, starting from 0, slowly turn up the frequency of the second oscillator. Stop at 1 Hz. You should hear that the sound is being faded in and out.

This may appear to be similar to the affect of adding sounds. However, there are two crucial differences. First, you cannot as yet hear the frequency of the second oscillator, as it is well below the audible frequency range. Secondly, the speed of the modulation is 2 Hz, not 1 Hz. This is because the first oscillator is being attenuated by the second. As the second oscillator has a positive and negative phase, this attenuation happens twice per cycle, with the original oscillator being initially modulated by the positive phase of oscillator 2, and then by the negative phase.

Now raise the frequency of oscillator 2. At first, you will hear a tremolo affect, with the speed of the tremolo increasing as you raise the frequency. As you approach the audible frequency threshold (20 Hz), you will notice that the sound begins to change. You will start to hear two separate tones, one ascending and one descending.

This is a form of amplitude modulation known as **ring modulation**. In the case of ring modulation, the frequencies you hear are equal to the carrier frequency (oscillator 1) plus and minus the modulation frequency (oscillator 2). For example, if you ring modulate a 400 Hz sine wave with a 100 Hz sine wave, the tones you will hear will be 300 Hz and 500 Hz. In this way, ring modulation has the same properties as aliasing.

A slightly more advanced form of amplitude modulation, know as classic AM, requires that the amplitude of the modulating waveform be adjusted so that it oscillates between 0 and 1 as opposed to -1 and 1. This should be a simple task. It changes the sonic characteristics of the output considerably. Using a [*~] object and a [+~] object, implement this form of AM. Can you describe the ways in which it differs from RM?

Learning activity

Ring modulation sounds especially interesting on acoustic signals. Give it a try using [readsf~] as the carrier. The composer Karlheinz Stockhausen (1928–2007) used the technique extensively, particularly on vocal works such as *Microphonie II*. He also claimed to have invented the technique.

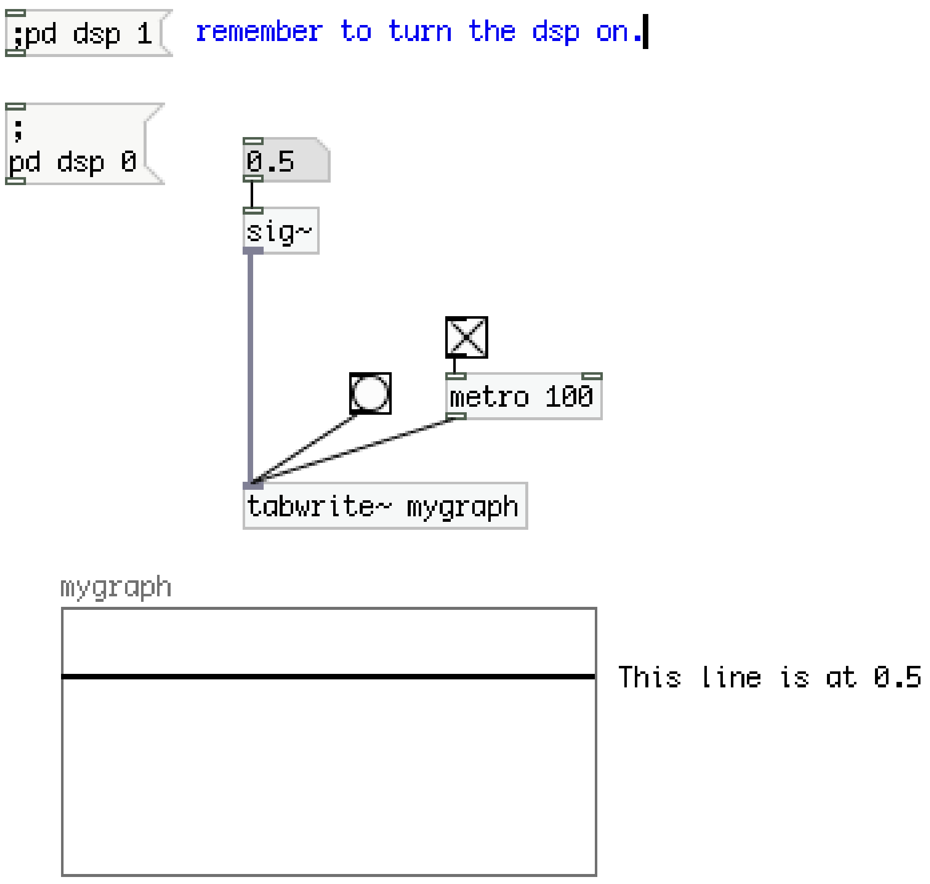
Does this make aliasing occasionally creatively desirable, despite being bad practice from an engineering perspective?

2.7.5 Frequency modulation

Sounds have amplitudes. They also have frequencies. We have learned that there are many creative applications for amplitude modulation. This is equally true of the following technique, **frequency modulation** (FM), discovered for musical purposes by John Chowning at Stanford University in 1967 (Chowning 1973).

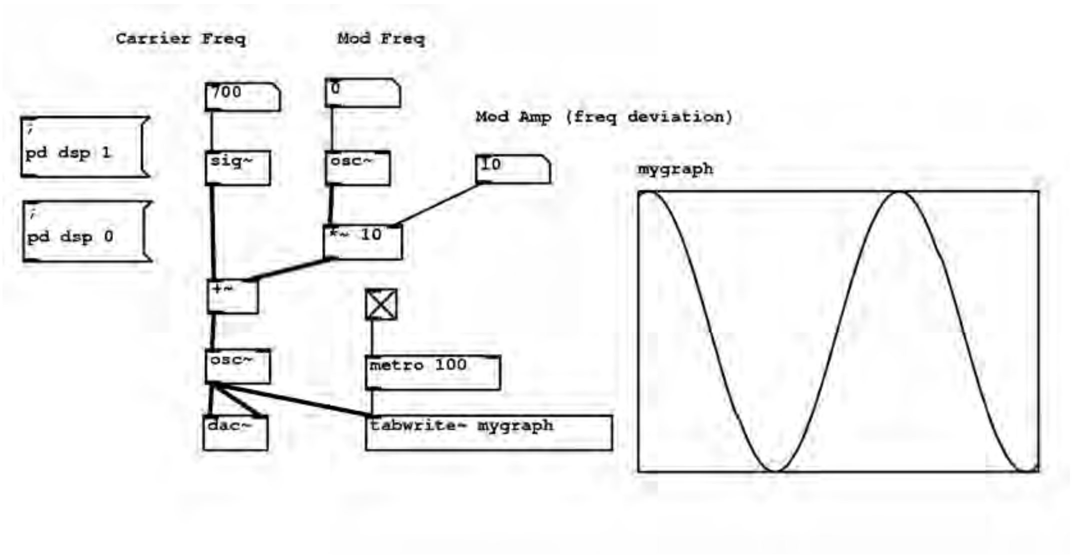
To create FM, we have to modulate the frequency of oscillator 1 (the carrier), with the frequency and amplitude of oscillator 2. To do this properly, we should control the frequency of oscillator 1 with a signal, using an object called [sig~].

[sig~] takes any floating point number and turns it into a stable signal value. Have a look at the graph below. The [sig~] object is creating a constant signal of 0.5. It has no audible frequency, as it is not oscillating.



We can use [sig˜] for all sorts of things. Most importantly, we can use it to control an oscillator. In addition, we can combine the output of [sig˜] with other signals.

Create a patch with two [osc˜] oscillators, both with their own amplitude controls ([*˜]). Connect oscillator 1 (the carrier) to the [dac˜]. Connect oscillator 2 (the modulator) to a [+˜] object that takes the output of a [sig˜] as its first input. You should end up with something like this.



Now, give the carrier a frequency that you can comfortably hear, somewhere between 200 and 750 Hz depending on your speakers. Now, slowly increase the frequency of the modulator (i.e. the **modulation frequency**), until it is around 0.3 Hz (holding down shift). You should be able to hear the pitch oscillating up and down around the carrier frequency (±10 Hz). You should also be able to see the sine wave squeezing and stretching in the window, as the distance between each peak is modified by oscillator 2.

Click on the number box going into the modulator's amplifier ([*~ 10]). Slowly turn the amplifier down to 0. As you reduce this value, you will hear the range of the modulation decreasing until the pitch is steady. Now slowly increase the value again until it is back at 10. You will hear the pitch modulating once more. In FM terminology, the amplitude of the modulating signal is called the peak frequency deviation (Chowning 1973, p. 527).

Now, slowly increase the modulation frequency and listen to what happens. You may be surprised by what you hear. The sound will begin to stabilise from a constantly varying pitch to a stable pitch once you reach the audible frequency threshold of 20 Hz. In addition, as you continue to increase the modulation frequency, you should find that the frequency content of the sound remains quite complex.

Finally, slowly increase the frequency deviation (the amplitude of oscillator 2) and listen to what happens. You will hear new partials becoming audible in the sound.

Now, set the carrier frequency at 400 Hz, and the modulator at 800 Hz – the first harmonic. You will realise immediately that the sound is now stable and ordered. Now adjust the frequency deviation. You should hear that the sound gets brighter or duller with the rise and fall of the frequency deviation, but the sounds remain harmonic.

In a similar way to amplitude modulation, FM synthesis produces sidebands around the carrier frequency. However, FM does not only produce a single pair of sideband frequencies, as in the case in AM, but instead produces sidebands at the carrier frequency plus and minus **multiples of** the modulator frequency: $f_c + kf_m$ and $f_c - kf_m$ where f_c is the frequency of the carrier, f_m is the frequency of the modulator, and k is an integer greater than zero.

The amplitude of individual partials (the carrier frequency itself as well as all sideband pairs) are calculated using the modulation index, i. The modulation index is defined as the ratio between the frequency deviation, d, and the modulation frequency: $i = d/f_m$. Scaling factors for partials (relative to the amplitude of the carrier) are determined using the modulation index and the set of Bessel functions $B_n(i)$. For example, $B_0(i)$ determines the scaling factor for the carrier frequency, $B_1(i)$ the scaling factor of the first pair of sidebands, $B_2(i)$ the scaling factor for the second pair of sidebands, and so on. As a rough guide, the number of audible sideband pairs in an FM generated signal can be predicted as $i + 1$ (Miranda 2002, pp. 26–8).

If the modulator is a harmonic of the carrier frequency, or if they are both harmonics of a lower fundamental, the sidebands will all be from the harmonic series, creating an ordered and stable sound texture. If you adjust the frequency deviation you will hear frequencies along the harmonic series being added and subtracted from the sound. Likewise, when the carrier and modulator are not harmonically related, you will experience inharmonic bell-like tones.

FM tends to be best suited to creating rich, textured sounds such as bells, and metallic noises. It is also a highly expressive synthesis method, and is very useful in a variety of electronic music contexts, particularly in combination with other synthesis approaches.

Learning activity

The object [mtof] takes midi note numbers and converts them to their frequency equivalents. Build an FM synthesiser controlled by MIDI note information, either from a MIDI device, or alternatively from a series of note-number messages triggered by [key] objects.

Create three arrays and use them to alter the carrier frequency, modulation frequency and peak frequency deviation respectively. Automate this process with [metro] and a counter.

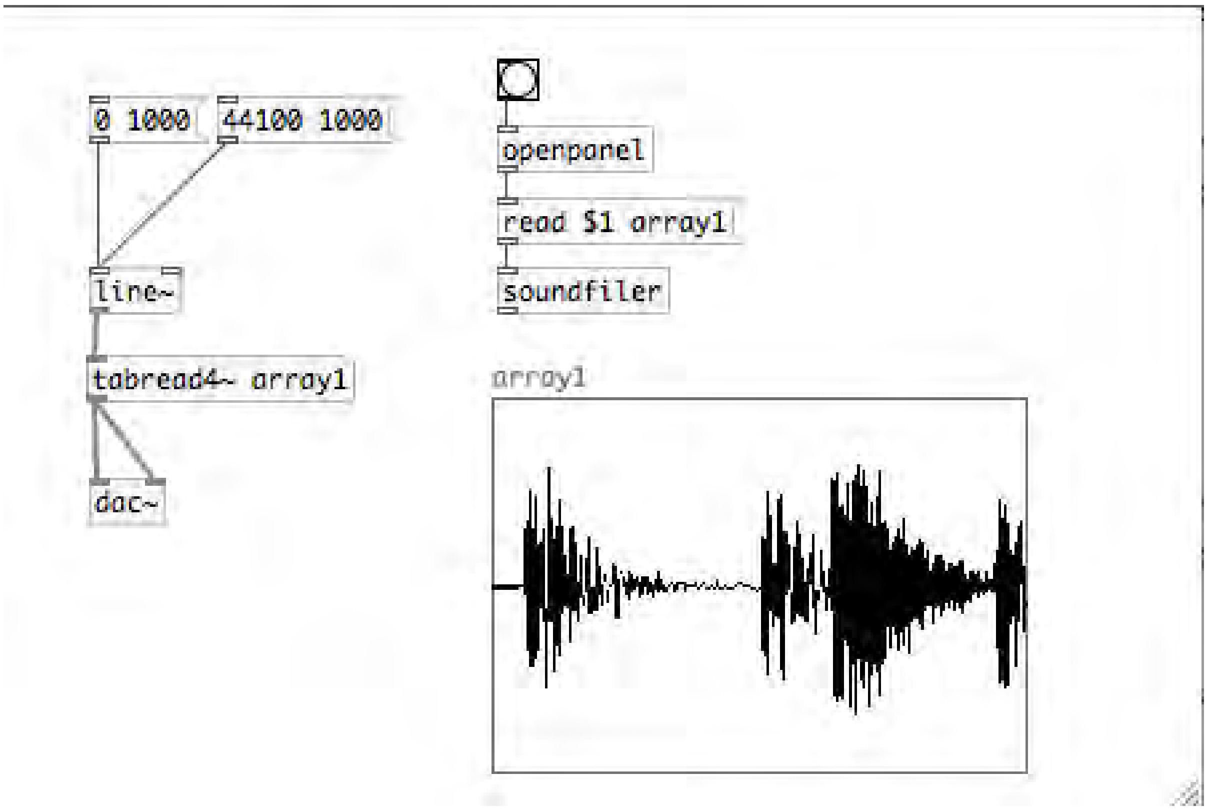
2.7.6 Playing back sampled sound with line

You can play back the contents of an array at signal rate by using the [tabread4] object and a control signal known as a **signal ramp**. You can provide the name of the array as an argument to [tabread4] (just like [tabread]). [tabread4] will allow you to treat the contents of the array as a waveform. In this way, you can use any data you like as a soundwave!

The simplest form of signal ramp in PD is called [line]. It creates a signal ramp between its current value and a new value, over a period of time specified in milliseconds.

So, if you want to play the first second of a sample over the period of one second, you would pass the message '44100 1000' to [line], and plug this object into [tabread 4]. If you then passed the message '0 1000' to your line object, it would play the sound backwards from its current position, to sample 0, in one second. Likewise, if you then passed the message '88200 10000', it would play the first two seconds of the sample forwards over a period of 10 seconds. As a result, it would play back at 1/5th of the speed. Can you understand why?

You can load sampled sound from disk into an array by using an object called [soundfiler]. [soundfiler] takes a message to read samples from anywhere on your computer and passes all the data into an array. So if your array is called **array1**, all you need do is pass the message 'read (insertsamplelocation) array1' to [soundfiler].



2.7.7 Enveloping

You can use array data to control any signal, including the amplitudes of signals (by of course using the [*] object). In this way, you can control the volume of your sounds over time with data in an array. This can be done even more simply by using the [line] object to create ramps between signal values of 0. and 1. You can then use these ramps to fade a sample or signal of any kind between 0. and 1. and back again over any period of time, simply through multiplying your signal ramp with your carrier signal.

Controlling the way in which a sound fades up and down like this is often referred to as envelopes. Crucially, you should not restrict the use of envelopes to the control of volume alone. As we have already seen, they can be easily deployed to control the playback position and speed of a sample stored in RAM.

2.7.8 Using the VCF object

Once you are used to using envelopes to control frequency and volume, you should try using them to control a filter (you will have covered the mathematics of filtering in *Creative Computing 2: interactive multimedia*).

The easiest filter for creative use in PD is called [vcf]. This allows you to attenuate certain frequencies in a signal above the cutoff frequency specified in Hz. In addition, you can selectively adjust the filter with the q function. This increases the amplitude of frequencies around the centre frequency. The value of q itself is the ratio of the centre frequency to the upper and lower frequency bands boosted to a level of at least -3db below the amplitude of the centre frequency. This means that all frequencies around the centre frequency are boosted. The q simply describes the width of the curve as a ratio. This width is adjustable with the [vcf] object, and it is this that gives it its distinctive sound. The vcf is the central component of the subtractive synthesis model, allowing you to control the frequency content of your synthesised sound.

You should now be able to create additive, wavetable-based, and subtractive synthesisers. You should also be able to sequence them and arrange them using signal rate controls.

Chapter 3

Algorithmic composition: swarm music

3.1 Introduction

In algorithmic composition, a composer creates or utilises abstract processes in order to generate sequences of musical events. As such, algorithmic composition can be described as a form of 'meta-composition' (Taube 2004, Xenakis 1992), because the composer engages in composition at a higher, or more abstract, level than might otherwise be the case. To contrast algorithmic composition with non-algorithmic composition, a composer not working with algorithmic techniques will typically create a composition by explicitly specifying an arrangement of notes, perhaps by producing a musical score or by entering musical notes into a computer-based sequencer. The key point is that the composer of non-algorithmic music is working directly at the level of musical notes – they may write down a melody that they have imagined in their head, or that they have created by improvising with a musical instrument. In contrast, a composer using algorithmic techniques is less concerned about specifying the exact arrangement of musical events in time, and instead focuses attention towards higher level abstract processes. It is these abstract processes that are responsible for generating the musical sequences that are heard as the music. Composing music using algorithms opens up many fascinating avenues of exploration; it also presents many challenges.

It is important to remember that the use of abstract processes and algorithms in musical composition is not a new idea, and certainly predates the invention of the computer. The role of abstract processes in the creation of music dates back to the Middle Ages, with the **canon** being one of the earliest examples (Harley 2009, p. 109). In the fourteenth-century, composers of the Ars Nova style developed a technique now known as **isorhythm** (Bent 2010). This process involves the combination of a rhythmic pattern (called a **talea**) with a melodic phrase (called a **color**), which are typically of different lengths, in order to generate musical variation.

Music throughout the twentieth-century has significantly expanded the range of process-based compositional techniques; for example, those of the serialist and aleatoric traditions. It is also important to remember that even when a computer is used to execute algorithmic processes, the resulting composition can still be performed by human musicians (see Harley (2009) for an in-depth survey). For example, the *Illiac Suite* for string quartet (1955–56), programmed by Lejaren Hiller and Leonard Isaacson, relied on a computer to generate the score, employing a range of innovative algorithmic techniques. Similarly, during the composition of the string quartet *ST/4,1-080262*, Iannis Xenakis used a computer to carry out stochastic calculations in order to generate the musical material (Griffiths 1995, p. 207).

When a computer is used to realise an algorithmic work, a considerable range of possibilities opens up. As well as using algorithmic processes to determine notes, pitches, rhythms, dynamics and instrumentation, the whole palette of sound synthesis techniques becomes amenable to algorithmic control. There has been a rapid development in dedicated music programming languages and environments over recent decades offering the composer easy access to this exciting new world. Widely used software includes: ChucK[1], Common Music[2], CSound[3], Impromptu[4], Max/MSP[5], Pure Data[6], and SuperCollider[7]. One aspect of the design of these and other similar software is to meet the needs of algorithmic composers. The following quotation from James McCartney, the original creator of SuperCollider, describes the key objectives behind the software design.

> *Motivations for the design of SuperCollider were the ability to realize sound processes that were different every time they are played, to write a piece in a way that describes a range of possibilities rather than a fixed entity, and to facilitate live improvisation by a composer/performer. (McCartney 2002, p. 61)*

Many music programming systems today are highly dynamic and allow the composer to interact in real time with the generative process. This has allowed composers to go one step further and take algorithmic composition into live performance. The practice of live coding brings together algorithmic composition and live performance (Collins et al. 2003). Live coding involves writing and interacting with algorithm in front of an audience in order to generate music or visual animation. Typically, a performer projects their screen simultaneously to help give the audience some insight into the abstract processes they are creating. Live coding not only places tough demands on the performer (they might introduce some code that causes their system to crash!), but also on the programming language itself. A system must be robust and efficient, but importantly must also be flexible enough to allow the performer to concisely express their musical ideas. Live coding is a rich application domain for research into language design and also the psychology of programming (Blackwell & Collins 2005).[8]

As Robert Rowe notes, the range of techniques employed in composition is much wider than that used to analyse and understand music (Rowe 2001, p. 201). Explanatory models are required to be grounded in more general terms, in the case of music, often supported by cultural, perceptual, cognitive or musicological theories. Whereas compositional techniques require no such external validation, and Rowe gives the example of a composer utilising a telephone directory in the act of composition. As such, the evaluation of algorithmic composition techniques need only rest on 'whether they output structures that make musical sense' (Rowe 2001, p. 7). The methodological considerations of algorithmic composition are also discussed in Section 1.2 of this study guide.

The very open-ended range of possibilities available to the algorithmic composer presents opportunities and challenges. The composer is free to explore arbitrarily complex abstract processes, which might generate surprising and high quality musical results. However, processes that include elements of randomness may vary considerably in the quality of their output if the indeterminacy is not tightly controlled. Control itself may be an issue, for if the algorithmic process is too complex, the composer may lose sight of how the process relates to the music it

[1]`http://chuck.cs.princeton.edu/`
[2]`http://commonmusic.sourceforge.net/`
[3]`http://csound.sourceforge.net/`
[4]`http://impromptu.moso.com.au/`
[5]`http://cycling74.com/`
[6]`http://puredata.info/`
[7]`http://supercollider.sourceforge.net/`
[8]Also see the TOPLAP website for more information on live coding: `http://www.toplap.org/`

generates. In other words, the process could become a 'black box' instead of a transparent process that the composer is able to engage with – however, this might not necessarily be a bad thing, and may even be an artistic aim. One under researched topic in algorithmic composition is the issue of larger scale musical form. Algorithms used to generate musical material are often constrained to producing one particular kind of musical patterning or texture, and lack a means of generating structured variation over larger time frames (Collins 2009).

Algorithmic composition also raises issues for listeners and musicologists. To consider listeners: is it necessary that a listener should be able to hear and understand the process that generated an algorithmic work in order to fully appreciate it? See Lerdahl (1992) for one argument on this subject from the point of view of a cognitive scientist. A similar issue is raised concerning the musicological analysis of generative music: how is it possible to analyse a piece of music that might be different each time it is performed (Collins 2008)? Issues of authorship are also brought into question by the practice of algorithmic composition. In most cases it seems reasonable to assume that even if an entire musical work has been generated by an algorithm, we would still attribute authorship to the composer since they created the algorithm. However, does the situation become different when an algorithm becomes extremely complex, or has the ability to learn and modify its own behaviour to the extent that it could be said to be creative (Wiggins 2006)?

The remainder of this chapter explores swarm process for algorithmic composition. Swarm music is an example of a class of algorithmic techniques that are inspired by behaviour observed in the natural world.

3.2 Swarms, music and improvisation

Swarming locusts, buffalo herds and anchovy schools are just three examples of the many kinds of groupings of social animals. The incredible and seemingly choreographed behaviour of starling flocks, filmed at Otmoor, Oxfordshire, UK[9] is a particularly striking and beautiful example.

This synchronised, co-ordinated dance seems so perfectly executed that we can only assume a leader bird has choreographed the display. However the swirling, darting patterns are believed to occur spontaneously, deriving from simple and local rules that do not in themselves contain features of the overall, global behaviour of the group. Flocks, and systems displaying similar behaviour, are said to **self-organise** (Bonabeau et al. 1999).

Self-organisation (SO) has been observed in many systems in physics and chemistry as well as in market economics, motorway traffic flow, and in biological systems such as animal groups and immune systems. In flocks, the collective order arises from the motions of individual birds that only react to the motions of nearby neighbours. Individual starlings do not have a sense of the overall flock. In general, self-organising systems display emergent structures that are not imposed by external ordering influences, but result from local interactions and without reference to the global pattern.

This chapter explores the idea that music may be viewed as a self-organising system. At first glance, music does not seem to be a candidate for emergence.

[9]http://www.youtube.com/watch?v=XH-groCeKbE

Systems that self-organise are also usually **decentralised**. Antibodies in an immune system mobilise against the pathogen in a way that is quite unlike our own centralised way of organising armies, with top down control flowing through a command hierarchy, headed by the General (read *Turtles, Termites and Traffic Jams* (Resnick 1997) for a convincing and heartfelt account of decentralisation).

Western art music is highly centralised, headed by the composer. However, music outside this tradition, for example jazz and folk music world-wide, is highly improvisational. Much of the musical content is not planned ahead of the performance but is generated afresh and in real time. Improvised music is much more akin to the decentralised, bottom up, structuring that we see in flocks, swarms and motorway bunching.

Imagine replacing starlings by musical notes (crotchets, quavers etc.) and letting these notes 'fly' across a sheet of musical manuscript: what might this sound like? Examine (almost) any musical score you can find. If the score has been composed, you will find a tight structure, governed by statement, repetition and variation. However, if you look at a transcription of an improvisation (the transcriptions of the alto saxophonist Charlie Parker[10] provide exemplary examples) you will notice a looser, more organic flow. Perhaps a swarm of interacting notes might generate such a solo?

Let us examine in more detail the local concerns of individuals in a swarm. The overriding concern is surely not to collide! Beyond this is the need to fly towards other individuals, especially for those individuals on the outside of the swarm. (We will focus on swarms rather than flocks because they are simpler. In dynamical terms, individuals in flocks attempt to match flight direction, as well as attract, and avoid collisions.) Further, notes in music also avoid collisions, or else the music would be too bland. Notes do not wander aimlessly across the stave with large jumps separating them; they clump together as if mutually attracted.

Apart from attraction and collision avoidance, there are a couple of other noteworthy parallels. The spatial patterns produced by a swarm resemble previous patterns, but are unlikely to be exact repetitions. This is also a feature of improvised music, where strict repetition is unusual, if not undesirable, yet some degree of similarity is apparent.

A further property of SO is the 'amplifiication of fluctuations'. In a swarming example, neighbouring individuals might become attracted towards a wayward straggler (a fluctuation); and then neighbours of the neighbours are diverted, eventually causing the entire population to fly towards the outlier (amplification). Similarly, improvised music can take on unpredictable twists and turns with the entire ensemble spontaneously moving in new musical directions, as if carefully planned.

These analogies are suggestive of links between SO and improvised music that might be made at some level of analysis. The question for the computer musician is this: can we use principles of SO to generate sequences of notes that, although unfamiliar and improvisational in character, we might hear as musical?

[10]*The Charlie Parker Omnibook*, Jamie Aebersold, pub. Criterion (1976) ISBN 0769260543

3.3 Project: MIDI synthesiser

This chapter contains a number of programming projects culminating in a swarm-based improvisational system. It is a simple version of the author's own Swarm Music (Blackwell 2007).

First, you will need to build a synthesiser for the production of sound. A MIDI synthesiser will be sufficient for our purpose. We will use Java since it has convenient MIDI and graphics libraries (the latter will become important when coding the swarm simulation) and because you are familiar with this language.

MIDI allows messages between devices to be transmitted over one of 16 channels. Each channel may be set to a different instrument (piano, guitar, etc.) by sending a program change message. Notes are played by sending note-on messages to a particular channel, along with the number of the note to play and its velocity (how hard a key was pressed for example, which is usually interpreted by a synthesiser as loudness, although it may also affect the timbre of the sound). MIDI note numbers increase in semitones and range from 0 to 127. Velocity values also range from 0 to 127, with 0 corresponding to the quietest level, and 127 to the loudest. Sounding notes are terminated by sending note-off messages.

Channels also respond to control messages. These do not play notes but alter the instrument characteristics in some way. For example 'control change 10 72' will set control number 10 – which corresponds to pan (stereo position) – to a value of 72. There are 128 controls, and each has a value in the range 0–127.

The Java `javax.sound.midi` package contains several classes which represent channels, synthesisers, messages and other MIDI technology that need not concern us here. You should now take a look at this package in the Java API, and in particular study the `MidiChannel`, `MidiSystem` and `Synthesizer` classes.

The following program shows a single channel MIDI synthesiser. A `Synthesizer` object is obtained from the `MidiSystem` class, which interacts directly with your machine. Notice that the synthesiser must be opened, and a `close` method shuts the synthesiser down. This method should be called when the program is terminated. This synthesiser has a single channel.

Add some code where indicated to play a few notes. You will have to pause the program for a while in between sending note-on messages. This can be achieved by asking the program thread to 'sleep' for a given time, as measured in milliseconds.

Try to pan the sound so that low notes are towards the left and high notes are towards the right. (You will have to send a `controlChange` message to the `MidiSynth.channel`.)

3.3.1 Simple MIDI synthesiser

```java
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Synthesizer;

public class SimpleMidiSynth {

  Synthesizer synthesizer = null;
  MidiChannel channel;

  SimpleMidiSynth() {

    try {
      synthesizer = MidiSystem.getSynthesizer();
      synthesizer.open();
    } catch (MidiUnavailableException e) {
      e.printStackTrace();
    }
    MidiChannel[] channels = synthesizer.getChannels();
    channel = channels[0];
  }

  void close() {
    synthesizer.close();
  }

  public static void main(String[] args) {

    MidiSynth synth = new SimpleMidiSynth();

    // Play some notes by calling
    // noteOn(int noteNumber int velocity)
    // on this midi synth's single available channel.
    // Remember to call noteOff(int noteNumber)
    // before sending the next noteOn.

    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
    }

    synth.close();
  }
}
```

3.4 Rules of swarming

There are many ways of formulating the swarming rules alluded to in the Introduction. The idea is to find a set of rules that describes a swarm **at some level**, and, when coded, produces a swarm-like animation. The correct level of description for our purposes is in terms of a **dynamical system**.

In the dynamical system model, swarm individuals are represented as **particles**. Particles are simple dynamic entities of zero size, moving in space, and subject to forces due to inter-particle interaction. In other words they are described by position ($\mathbf{x}$), velocity ($\mathbf{v}$) and acceleration ($\mathbf{a}$) vectors. They have no internal state or any other property.

Time is necessarily discrete in this model. Each particle is updated by calculating an acceleration vector and adjusting velocity and position. The **dynamical state** at time $t + 1$ is obtained from the state at t according to a discrete particle kinematics:

$$\begin{aligned}\mathbf{v}(t+1) &= \mathbf{v}(t) + \mathbf{a}(t)\\ \mathbf{x}(t+1) &= \mathbf{x}(t) + \mathbf{v}(t)\end{aligned}$$

An animation proceeds by a sequence of frames; each frame is a snapshot of the swarm at time t. The animation is paused for a short time between frames, the swarm is updated and the new frame is displayed.

The acceleration vector can be calculated in various ways, but here we specify a method that leads to good results and has its origins in biological modelling of actual swarming animals.

Each particle is considered to have two zones of perception, an inner and an outer zone. The particle cannot 'see' any other particle that is outside the outer zone. This means that the interactions are **localised**. Particles are repelled from the centre of mass of any particles within the inner zone, and attracted towards the centre of mass of any particles within the outer zone, but outside the inner zone. The Figures 3.1 and 3.2 below illustrate the intention.

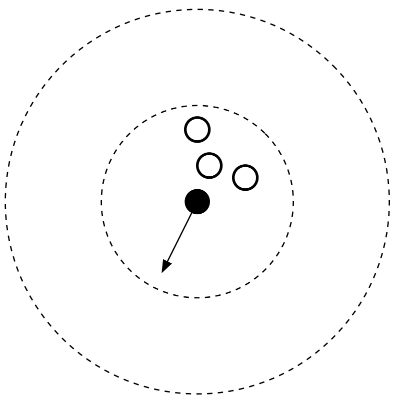


Figure 3.1: Repulsion away from three neighbours within the inner zone.

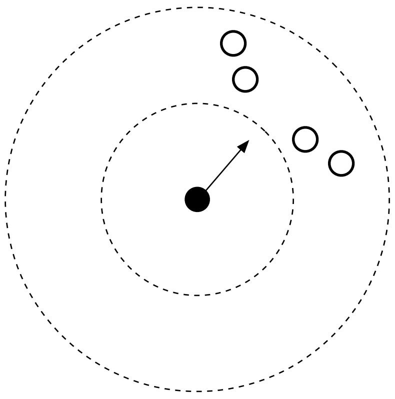


Figure 3.2: Attractions towards four neighbours within the outer zone.

In detail, the algorithm is:

- **if** the inner zone of the particle at $\mathbf{x}$ is occupied, find the neighbour centre of mass $\mathbf{y}_{CM}$ where $\mathbf{y}$ is the position of each neighbour. The acceleration is then given by $\mathbf{a} = \mathbf{x} - \mathbf{y}_{CM}$. The centre of mass of a set of points $\mathbf{y}_1, \mathbf{y}_2, \dots \mathbf{y}_n$ is $\mathbf{y}_{CM} = \sum_{i=1}^{n} \mathbf{y}_i$
- **else if** the outer zone of the particle at $\mathbf{x}$ is occupied, find the neighbour centre of mass $\mathbf{y}_{cm}$ where $\mathbf{y}$ is the position of each neighbour. The acceleration is then given by $\mathbf{a} = \mathbf{y}_{CM} - \mathbf{x}$
- **else** $\mathbf{a} = 0$.

Notice that avoidance always has precedence: if both zones are occupied, a single avoidance acceleration is calculated. Finite accelerations may be 'clamped' to a fixed magnitude a_0.

In practice the velocity is also clamped to a constant. Individuals in a swarm or flock tend to move at a steady speed. Velocity clamping prevents both high speeds and stagnation, neither of which are observed in nature. Velocity clamping ensures that particles always move at the same speed; accelerations instigate turning, not speeding or slowing.

The complete particle update kinematics are specified by the following steps:

$$\mathbf{a} = \begin{cases} \mathbf{x} - \mathbf{y}_{CM} \\ \mathbf{y}_{CM} - \mathbf{x} \\ 0 \end{cases} \tag{3.1}$$

$$\mathbf{a} \leftarrow a_0 \frac{\mathbf{a}}{|\mathbf{a}|} (|\mathbf{a} \neq 0) \tag{3.2}$$

$$\mathbf{v}(t+1) = \mathbf{v}(t) + \mathbf{a}(t) \tag{3.3}$$

$$\mathbf{v}(t+1) \leftarrow v_0 \frac{\mathbf{v}_{t+1}}{|\mathbf{v}_{t+1}|} \tag{3.4}$$

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{v}(t) \tag{3.5}$$

3.5 Project: vector maths

Although not necessary, the development of a vector maths library will keep your code tidy when you begin to implement the full swarm animation.

We can identify the following operations:

- vector addition and subtraction
- multiplication of a vector by a scalar
- finding the length of a vector
- clamping a non-zero vector to a scalar magnitude
- finding the distance $|\mathbf{a} - \mathbf{b}|$ between points at positions $\mathbf{a}$ and $\mathbf{b}$.

Implement these operations using static methods and place them in a class `VecMaths`. Write a test program to verify that the methods are behaving as expected. The following class `SimpleVecMaths` provides a skeleton. For our purposes a vector is an array of `floats`. The animation package that we will use draws to single precision accuracy (easily sufficient resolution on a computer screen). Working to single precision accuracy avoids excessive casting later on.

There are a few method prototypes with three arguments. The first argument is a pre-defined vector, used to carry the result of the calculation. This saves the method from creating a new array at each call. A reference to the result is also returned, enabling the method call to be used in an expression.

3.5.1 Simple vector maths

```java
import java.util.Arrays;

public class SimpleVecMaths {

  static float length(float[] vec) {

    float mag = 0;
    for (int d = 0; d < vec.length; d++)
      mag += vec[d] * vec[d];
    return (float) Math.sqrt(mag);
  }

  static void clamp(float[] vec, float length) {
    // Place your code here.
  }

  static float[] add(float[] result, float[] vecA, float[] vecB) {
    // Place your code here.
    return result;
  }

  static float[] subtract(float[] result, float[] vecA, float[] vecB
     ) {
    // Place your code here.
    return result;
  }

  static float[] scalarMult(float[] result, float scalar, float[]
     vec) {
    // Place your code here.
    return result;
  }

  static float dist(float[] vecA, float[] vecB) {
    // Place your code here.
    return 0;
  }

  public static void main(String[] args) {

    float[] vec = new float[] { 3, 4 };
    System.out.println("length of " + Arrays.toString(vec) + " is "
        + length(vec));
  }
}
```

3.6 Project: particle animation

We are almost ready to begin coding the animation. We shall make use of *Processing*, a Java animation and graphics environment which you have used extensively in previous units. This will enable us to ignore many details of Java graphics and threading, and concentrate on what we want to do, namely program a swarm animation, and turn it into music.

Open the *Processing* editor. Type `ParticleApplet.java` directly into the editor and save the 'sketch' as `ParticleApplet`. A *Processing* sketch is actually a Java **Applet**, i.e. a Java program which runs inside a webpage. *Processing* will do the embedding for you if you click on the 'export' button.

`ParticleApplet` refers to two other classes, `Particle` and `VecMaths`. `Particle` conveniently holds position, velocity vectors and the particle size (the size is used purely for rendering and takes no part in the dynamics). The class definition is given below; `VecMaths` has already been defined (by you!). Both of these classes need to be loaded into the Processing editor.

Click on the arrow button on the top right of the tab menu bar. Select 'New Tab' and type (or copy and paste) the sourcecode for `Particle` into the blank editor pane. Save the new file (this dialogue appears at the foot of the editor) as `Particle.java`. It is very important to remember the `.java` ending. Repeat the procedure with `Vecmaths`.

The `PApplet`'s `setUp()` method is performed when the applet is launched, rather like a constructor. Here, position and velocity vectors (floating point arrays) are initialised. The velocity is clamped, and a `Particle` object is created. The call to `smooth` asks the graphics to perform antialiasing so that edges are sharp, and `noStroke` means that shapes are drawn without an outline.

`PApplet` repeatedly calls `draw` at a default rate of 60 times per second. This is known as the frame rate. In this case `draw` paints the entire frame black,

`background(0),`

and prepares to draw and fill with white any requested shape,

`fill(255),`

and finally draws a circle where the particle is located,

`ellipse((particle.position[0], particle.position[1], particle.diam, particle.diam).`

If all goes to plan, when you click on run, you will see a black background with a stationary white disc. We have drawn the particle, but it does not move, because the position does not change.

Add your own code where indicated to make the particle move. The update is simple. In the absence of interactions, just add the velocity to the position.

After a while the particle will leave the frame. Your code will therefore need to take action when this happens. One possibility is that the particle is reflected, like a billiard ball; another possibility is that the particle reappears on the opposite edge.

3.6.1 Particle applet

```
import processing.core.PApplet;

public class ParticleApplet extends PApplet {

  private static final long serialVersionUID = 1L;

  int D, L;
  float speed, particleDiam;

  Particle particle;

  public void setup() {

    size(L, L);

    float[] x = new float[D];
    float[] v = new float[D];

    for (int d = 0; d < D; d++) {

      x[d] = (float) (L * Math.random());
      v[d] = (float) (Math.random() - 0.5f);
    }
    VecMaths.clamp(v, speed);
    particle = new Particle(x, v, particleDiam);

    smooth();
    noStroke();
  }

  public void draw() {

    background(0);

    // Place your code here.

    fill(255);
    ellipse(particle.position[0], particle.position[1], particle.
        diam,
        particle.diam);
  }

  public static void main(String args[]) {

    PApplet.main(new String[] { "SimpleSwarmMusic" });
  }
}
```

3.6.2 Particle

```
public class Particle {

  float[] position, velocity;
  float diam;

  Particle(float[] x, float[] v, float d) {

    position = x;
    velocity = v;
    diam = d;
  }
}
```

3.7 Project: swarm animation

Now that you have succeeded in animating a single particle, it is time to implement a swarm of interacting particles. Since a swarm is a collection of particles, it makes sense to introduce a *Swarm* class to store the `particle` objects. `SimpleSwarm` shows an outline class. The associated `PApplet`, `SimpleSwarmAnimation`, transfers the drawing to the swarm object by calling `swarm.draw`, and passing a reference to itself. Your code for reflection or doubling-back should be pasted in the corresponding empty methods. The hardest part is to code the particle interactions, as defined in the algorithm above, and to choose the constants so that the animation proceeds at a reasonable pace (not too fast or too slow), and gives interesting swarm behaviour.

The kinematic parameters and other constants affecting the animation and swarm behaviour are:

f	frame rate
a_0	acceleration magnitude
v_0	speed
r_{avoid}	radius of inner avoidance zone
r_{attr}	outer radius of attraction zone
N	number of particles
D	dimension of space
L	size of space in each dimension

Additionally we note the two algorithms:

- particle kinematics steps (1)–(5)
- boundary rules.

To help you decide on parameter values, note that v_0 is the speed in pixels/frame. The speed in pixels/second is obtained by multiplying v_0 by the frame rate; the actual speed in cm/sec is found by multiplying this by the size of the pixels on your display. You can calculate the pixel size by dividing the resolution of your display by the linear measurement of your screen. A speed of a few cm/sec is about right.

3.7.1 Simple swarm

```
import java.util.ArrayList;
import processing.core.PApplet;

public class SimpleSwarm {

  int D, L;
  float speed, particleDiam;

  private ArrayList particles;

  public SimpleSwarm(int numParticles, int dim, int xMax) {

    D = dim;
    L = xMax;

    particles = new ArrayList();
    for (int p = 0; p < numParticles; p++) {

      float[] x = new float[D];
      float[] v = new float[D];

      for (int d = 0; d < D; d++) {

        x[d] = (float) (L * Math.random());
        v[d] = (float) (Math.random() - 0.5f);
      }
      VecMaths.clamp(v, speed);
      particles.add(new Particle(x, v, particleDiam));
    }
  }

  void wrap(float[] vec) {
    // Code to make the particle reappear on the opposite edge.
  }

  void reflect(float[] pos, float[] vel) {
    // Code to make the particle reflect at the edge.
  }

  void draw(PApplet pApplet) {

    for (int p = 0; p < particles.size(); p++) {

      Particle particle = (Particle)particles.get(p);

      for (int n = 0; n < particles.size(); n++) {

        if (n == p)
          continue;
        Particle neighbour = (Particle)particles.get(n);

        // Particle interaction code goes here.
      }

      wrap(particle.position);
      // reflect(particle.position, particle.velocity);
      pApplet.ellipse(particle.position[0], particle.position[1],
          particleDiam, particleDiam);
    }
  }

  public Particle getParticle(int p) {
    return (Particle)particles.get(p);
  }

  public int getNumParticles() {
    return particles.size();
  }
}
```

3.7.2 Simple swarm applet

```
import processing.core.PApplet;

public class SimpleSwarmApplet extends PApplet {

  private static final long serialVersionUID = 1L;

  int D, L, numParticles;
  SimpleSwarm swarm;

  public void setup() {

    size(L, L);

    swarm = new SimpleSwarm(numParticles, D, L);

    smooth();
    noStroke();
  }

  public void draw() {

    background(0);

    fill(255);
    swarm.draw(this);
  }
}
```

3.8 Project: swarm music

Hopefully you will now have a swarm animation running. It is now time to interpret particle positions as notes. Once more, keep your code tidy by defining a new class, an `Interpreter` class with a method `void interpret(Swarm swarm)`. This class will contain a `MidiSynth` object. The purpose of `interpret` is to take particle positions and form notes from the position components. A simple and direct scheme is to extract a note number from one coordinate, and a note velocity from the other.

Suppose the particles are flying in a box $[0, L]^2$. A linear interpretation would imply

$$n = n_{min} + \frac{x}{L}(n_{max} - n_{min}) \tag{3.6}$$

$$l = l_{min} + \frac{x}{L}(l_{max} - l_{min}) \tag{3.7}$$

where n and l are midi note numbers and midi velocities corresponding to a position (x, y). Notes and velocities are scaled between maximum and minimum values which can be adjusted as the user wishes.

Since `draw` will be called 60 times a second, and since your swarm may have 10 (or more) particles, you certainly don't want to interpret each particle. Instead, you will have to **sample** the swarm, interpreting a particle every δt seconds where δt is the required duration of the note. Particles can be chosen in turn. Clearly we wish to sample the swarm often enough so that the interpretation is a good representation of the configuration of the particles. It is helpful to draw a blob at each interpreted position; erase and redraw a new blob when the next particle sounds.

3.9 Swarm music as a live algorithm

We close this chapter by considering how a musician might interact with the music system that you have just built. The version of swarm music described in the previous pages is an example of a **generative** music system. In other words, the output depends only on the choice of internal parameters, the sequence of random numbers, and the details of the algorithm. The music is generated as if by machinery.

Swarm music[11] was conceived as an interactive application. The intention is that musicians play along with the system just as if taking part in a duet with another (human) musician. Such systems have been termed **Live Algorithms**.[12] The term is meant to convey the impression that the system behaves autonomously, almost as if alive, and also alludes to the frequent use of artificial life algorithms (which includes swarming simulations) in this context.

Swarm music employs an interactive model that is inspired by the indirect interaction known to social entomologists as **stigmergy**. A stigmergetic interaction

[11] `www.timblackwell.com` for further information, including downloads

[12] Swarming and Music. In Miranda E. and Biles A. (eds.): *Evolutionary Computer Music.* Springer Verlag 2007 ISBN 978-1-84628-599-8

is mediated by the environment rather than directly by the near (or actual) contact of interacting individuals. A chance encounter with a disturbance made sometime in the past influences future behaviour. A well known example is the laying and then following of pheromone trails by ants.

In swarm music, the musical output of the musician is mapped onto particle movement within the state space of the system. Notes are interpreted as positions within the internal space. The musician leaves a trail of particles behind her, just as if she were actually moving in the internal space. Each deposited particle then becomes an attractor for swarm particles. A particle will experience an acceleration towards any attractors that are within perceptual range; the swarming rules can be easily extended to include this scheme.

The field of live algorithm research is still developing but some desirable properties are immediately apparent.

- The system must be able to **reflect** musically what it hears. The output must bear some resemblance to the playing of the human partner. At the simplest level, a direct echo would suffice, although this would become tedious.
- The system must also be able to **innovate**. Its improvisations should suggest novelty; 'ideas' that the human partner can engage with. A quite random output would appear to satisfy this criterion but would again become tedious because of the relative lack of predictability and patterning.
- **Autonomy**, as opposed to automation, implies that the system may play, or be quiet, at any moment: this is not contingent on the activity of the interacting partner. Automatic responses will lead to predictability and musical tedium. On the other hand, apparently random responses become unpredictable and uninteresting.
- The system should have a degree of **transparency**; its internal patterning should be apparent in the interpreted sonic output, and the relation between system response and musical input should not be too obscure.

The stigmergetic model attempts to satisfy reflection, innovation and transparency. The musical output of a musician will surely lie in some region of state space, and this region will shrink somewhat, perhaps expand, and move around. In fact it appears to be a swarm (compare with the earlier remarks concerning SO and music). If the internal swarm is able to track the swarm-like image of the external musician, the output of the system will reflect what it hears. Conversely, if the swarm, or perhaps a breakaway subswarm, flies to an unrelated region of state space, the output will bear less resemblance to the input, and might be regarded as novel. Of course the musician must be able to infer some regularity concerning when this behaviour might happen; the tracking should not be too predictable or the breakaway too random.

The requirement of autonomy is satisfied to some extent in swarm music by including time-related parameters such as note duration and time between event onsets as additional dimensions in the space.

3.10 Project: interacting with swarm music

Although you could develop your program to include MIDI input, it can be quite tricky to set up the hardware, and to get Java to recognise this hardware. However, you can interact with the swarm by using the mouse. Attractors can be deposited by mouse clicks, or by pressing the mouse button and moving the cursor across the screen. The *Processing* API reveals two methods, `mouseClicked()` and `mouseDragged()` which you can override in your `PApplet`. In either case, the variables `mouseX` and `mouseY` yield the co-ordinates of the cursor. You will need to create attractor objects (these are just particles, except that they have zero velocity, and a different appearance) and store them in a variable. You should consider the lifetime of the attractors. Do they get consumed after a certain number of visits, or is there a fixed number, with the oldest being replaced by the newest?

Chapter 4
Understanding musical interaction

4.1 Introduction

When thinking about a discussion between a group of people, we can look at the words (sounds) that were spoken, and make some sense of those; we can look at the focus of the discussion (turn-taking), and how that moves around; we can look at the sorts of things being done with words – giving information, disagreeing, persuading . . .

We can also imagine various ways to record such a discussion, ranging from a sound/video record, to a written (tidied-up) account of who said what, to a summary of the points raised. For a written account, the transcription process can be done by people (on the basis of lengthy education in the language involved), or by current natural language processing techniques that do this more or less accurately.

Interaction between musicians in performance raises similar issues, with important differences too. Here we want ways of analysing the musical material in terms of some building blocks of music; we may be interested in the grammar associated with particular styles, and the rhythmic and pitch organisations employed. Automation of these processes is a challenging field, and in this chapter we will look at some current approaches.

We will look at some of the techniques and algorithms used in the analysis of music, taken as sounds organised in a structured way similar to that of natural language, and also ideas involved in understanding musical discourse.

Current computational technology allows machine-generated music to be incorporated into musical performance and improvisation in interesting and musically productive new ways, and this topic is also covered in this chapter.

Mostly we will use as examples music from **Western tonal music** (WTM) – roughly, music based on the organisation of musical pitch and rhythm common in Western Europe from the 16th century, in classical music up to the early 20th century, and used in most current (Western) popular, jazz and folk music. There are many other musics in the world, raising similar questions, and contemporary composers often work outside this framework.

For those without a basic background in terms such as **metre, rhythm, pitch and key**, you should look at an introduction to music theory, for example the first fifteen very short lessons on the `8notes.com` website.[1] You should bear in mind that American and British English have some differences in vocabulary, listed as "Alternative terms".[2]

[1]`http://www.8notes.com/theory`
[2]`http://dolmetsch.com/introduction.htm`

4.2 Sound, music, score . . .

Here we take music to be primarily an experience of sound in time; but this is not enough to distinguish music from sound in general. There is no general agreement as to what differentiates music from sound. A start to an answer is, for example, given by Roger Scruton:

> *Music is an art of sound. . . . Nor is it the work of a musician to write poetry, even though poetry too is an art of sound. So what distinguishes the sound of music?*
>
> *The simple answer is 'organization'. But it is no answer at all if we cannot say what kind of organization we have in mind. . . .*
>
> *(Scruton 1997, p 16)*

In this chapter we look at some of the ways Western tonal music is organised. There are different aspects to this organisation, which are of differing importance in other musics. Some of these are listed below.

Rhythm This concentrates on the time dimension of music, and is seen most clearly in drum and percussion music in general. It underlies nearly all WTM, from Beethoven to the blues.

Pitch An individual note, say one note played on a piano, is associated with a given pitch, associated with the physical frequency of sound waves in the air – each note on the piano or guitar has a different pitch. The way that pitches are organised together is fundamental to many musics.

Intensity The same note or drum beat may be played louder or softer; this dimension of organisation is called **intensity** and is associated with the energy level of the sound waves involved.

Timbre Here we are interested in the difference between notes of the same pitch and intensity played by different instruments, or sung by different people. The physics of this is more complicated; when guitarists change guitar between numbers, it is often about using a different timbre as part of a different atmosphere.

Much music is passed between musicians by playing, listening, and imitating. There are some spoken or sung conventions in describing instrumental music to other musicians, which constitute local specialised languages; see for example bols associated with Indian tala[3] (this music is rhythmically complex).

Written notation for WTM is traditionally central to classical musical performance. It reflects in particular the ways in which the pitch and rhythm are organised in this music. It is not important, for this chapter, to be able to imagine music on the basis of seeing such notation (this is something like learning a new language). However, the so-called **score**, which is a (fairly loose) specification of what the musicians are to perform, is important for many kinds of musical activity.

Historically, the task of producing a score involved detailed and laborious work. Nowadays, while we can do a reasonable job of turning spoken speech into written text automatically, the corresponding problem for music, say for producing a score from music improvised at the piano, has not seen such progress.

[3]`http://kksongs.org/talamala.html`

When we look at musical interaction later, we will need the notion of a score which gives the performing musicians some shared understanding of what musical actions are involved at different moments. If you want to see what an orchestral score looks like, look at Beethoven's Symphony no. 5.[4]

We will now look at some of the issues in machine analysis of music; in particular, ways of getting machines to help with recognition of the local structure of music in terms of rhythm and of pitch.

4.3 Rhythm, beat and metre

By **rhythm** we mean a pattern of accentuation of events in time, for example the 'short, short, short, long' at the start of Beethoven's Fifth symphony. These are concrete patterns, which suggest and are supported in WTM by longer-term structuring of accents in time. Here we look at two of the ways in which music in WTM is structured at the medium scale.

4.3.1 Beat

One ability needed when making sense of musical interaction is a way to recognise where the beat is when listening – this is where the listener will tap her finger or toe, and the regular beat that coordinates most dance. Most people do this naturally, but this is not easy for computer analysis. This basic ability is a basis for the coordination of musical performance where different musicians play in an ensemble, or where virtual computer musicians produce music in time with other musicians, human or virtual. This is the problem of **beat tracking**.

Some computer-generated music produces music where the beat is perfectly regular: within the limits of the technology, beats occur exactly every 0.75 seconds, for example. This is fine if this is what is intended. But this is unnatural for human musicians, who find keeping time alongside music produced this way reduces the possibility of shaping the tempo, which is basic to musical performance. It also means that the human must follow the computer. In a small ensemble of human musicians, the players listen to each other to coordinate the beat, which will vary by some amount during playing; the lead in shaping the tempo is taken by different players at different times.

For human listeners, it is not hard to pick up the beat when listening to music in a known style in real time. For machine beat-tracking in real-time, the input can be taken in two ways:

1. as audio input;
2. by tracking the physical gestures of the performers, rather than the sounds produced.

The latter can simplify the problem, because the places of the start of notes in time are usually then made obvious, and this is the main information used in beat-tracking algorithms.

[4] `http://www.dlib.indiana.edu/variations/scores/bgp5237/large/`

The most usual format used here is that of MIDI (Musical Instrument Digital Interface), which started out as a way of describing key press and release actions on piano-like keyboards. Nowadays MIDI versions of many instruments are available, and there are several software libraries supporting manipulation of MIDI data, for example in Java.[5]

Some sample `.wav` files used for testing beat-tracking algorithms can be downloaded by following the instructions from an ISMIR conference.[6] These give an idea of the variation of genre that such algorithms attempt to cover. You might like to try tapping along with some of these. It is not the case that everyone taps along in the same place – for example sometimes it is possible to tap twice as fast as someone else, and both make musical sense. We will look at this issue again later in this chapter.

There are some simple cases, such as some music with loud regular percussion (drum) beats, where attending to these sounds is the key to detecting the beat. But other cases, such as jazz guitar, can play without strong accents on the beat, with some accents off the beat (syncopation), and the human listener still follows the musical beat.

There are several very different algorithms that have been used here. Let us look at the case where we have MIDI input.

A system of virtual musical agents is described by Wulfhorst et al. (2003), where a simple beat-tracking algorithm based on MIDI tracking of the notes played by other musical agents is given. Suppose an agent keeps a record of recent events (note onsets) by time, and suppose the agent has an initial expectation of how frequent the beats will be. The agent can consider mapping these events as radial lines on a circle, for a notional beat at times $t, t+I, t+2I, \ldots$

- the angle on the circle corresponds to the times with respect to the beat, so that events occurring time I apart are at the same angle.
- more recent events are weighted more strongly (or events from a small time window are used).

If I is a good estimate of the time between beats, we expect to see events clustering at angles around the circle; if I is a bad estimate, the events will not show a pattern.

So by looking at a number of candidate intervals around the expected beat, the agent can find the best estimate to the current beat — and then use this estimate to make a musical gesture itself in relation to the current beat.

The clustering in the example on the right suggests that there are events happening regularly at each quarter beat. This happens often (think of percussion **filling-in**), and suggests that there is more going on than a level repeated beat; we now look at this aspect.

4.3.2 Metre

Typically in WTM the underlying pulses of music are organised hierarchically; in the case above, the tapping level beat is subdivided into four. Beats themselves fit into larger repeated units, usually in groups by 2, 3 or 4, and they themselves in

[5] http://www.midi.org/aboutmidi/index.php

[6] http://www.music-ir.org/mirex/2006/index.php/Audio_Beat_Tracking

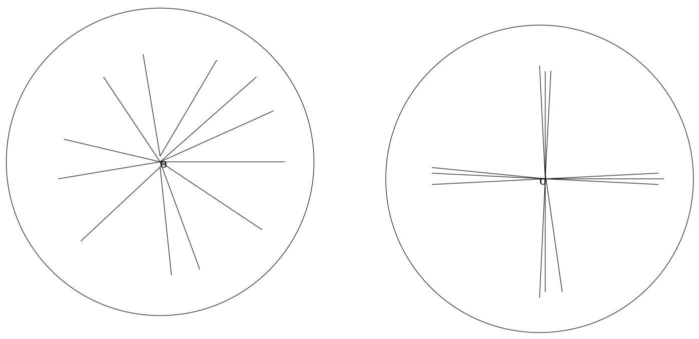


Figure 4.1: Bad (left) and good (right) choice of period

larger groupings. The bar (measure), marked in standard notation by a vertical line, is usually one or two levels above the beat; bars themselves fit into regular patterns too, as in the 12-bar blues. This hierarchical structure is called the **metrical** structure of the music. The so-called **time signature** written at the start of a piece indicates this hierarchy at the level of bars and beneath.

Here is an example of rhythmically simple music, based on a dance pattern. The rows of dots beneath indicate levels of metrical grouping. The beat is usually heard at row 2, which is divided in four. The beats group together two by two. In each case, the larger constituent grouping is taken to give the more important temporal grid – not necessarily where accents fall most often, as syncopated music shows. The relative importance, or salience, of different levels of metrical organisation has been studied extensively by psychologists (London 2004). It has been shown that the most salient level of grouping in a metrical hierarchy tends to have an inter-onset interval of around 600 ms. This means that groupings that are separated by around 600 ms in time are more likely to be perceived as the beat (Parncutt 1994).

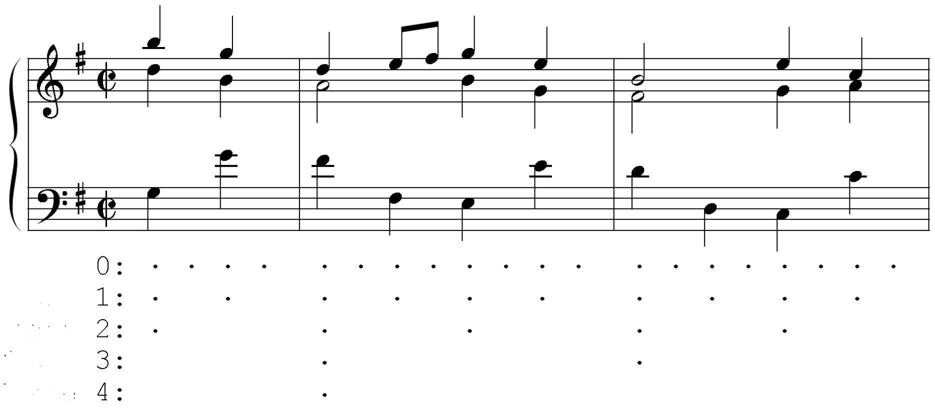


Figure 4.2: Metrical hierarchy in a Bach gavotte

There is ambiguity at level 3 for many listeners: while there is agreement that the level two units are grouped in twos, it is ambiguous how that grouping is done – either the first two units at level two are combined, or the second and third. You can listen to this and decide what you think (here played rather quickly).[7] It is a challenge to find these metrical levels automatically, in a way that fits with natural human abilities; in cases like a waltz, we find it fairly easy to recognise that there are three beats in a bar, and to tell where the first, strong, beat occurs. Again, there are different approaches to this task, which is in general harder than beat tracking. The interested reader can look, for example, at David Temperley's book on musical cognition (Temperley 2001*b*).[8]

There are many other ways to organise rhythm in music – see the brief overview mostly within WTM.[9]

4.4 Pitch and key

We now look at how different pitches play a role in WTM, and how computer analysis can support interactions that depend on this organisation. Each note on a piano or guitar corresponds to a different pitch, and these instruments are designed to play only a subset of the possible pitches (unlike the human voice, or violin). The way notes on the piano are laid out with white and black notes, and a pattern repeating after seven white notes, comes from the particular organisation of pitch in terms of **keys**.

Keys are organised in two main ways (major and minor) around a key note (called the **tonic**); the tonic acts as a pitch centre or home note, where the melody typically ends. Listen to these musical examples.[10]

Take a look at this accurate account.[11] The bluffer's very short guide is here.[12]

Knowing the current key is important in musical interaction – for example, in folk music, when someone wants to follow without a break to a new tune of their choice, they will announce the key (rather than the tune), and the musicians then have the right context to accompany the new melody.

Notes which differ in that the fundamental frequency of one as a sound wave is twice that of the other are given the same name, and not distinguished as tonic notes (they are said to be an octave apart). It is natural when men and women sing a melody together that they sing an octave apart, yet are understood as singing 'the same notes'.

There is a geometric way to think of the organisation of pitches that leads to an interesting approach to key detection. In this approach, different pitches appear several times, because their role is ambiguous depending on which key they are analysed as belonging to. By looking at the assignments of pitches which occur, to their possible places in a 2 or 3-dimensional structure, and looking for the assignment which gives the most compact representation in that structure, a candidate can be computed as the most likely key for the set of pitches concerned.

A recent system that exploits this approach is given by Mardirossian & Chew

[7]http://www.we7.com/#/track/French-Suite-No-5-in-G-BWV-816--Gavotte!trackId=2179255
[8]http://www.link.cs.cmu.edu/cbms/index.html
[9]http://reference.findtarget.com/search/Meter%20(music)/
[10]http://ask.metafilter.com/6444/Music-Just-what-exactly-is-key#132028
[11]http://wapedia.mobi/en/Key_(music)
[12]http://www.soundfeelings.com/products/music_instruction/eproducts/key.htm

(2008),[13] where Table 1 shows the arrangement of pitches in a two-dimensional array.

4.5 Musical grammar and style

4.5.1 Grammar

Given some understanding of how pitch and rhythm are organised, we can now think of how music is structured at higher levels. One approach here is to make use of notions from **grammars**, as applied to both natural languages like English, and computer languages like Pascal.

A grammar gives a set of rules by which combinations of some entities (words, symbols, sounds . . .) can be categorised as OK or not, according to the grammar. When a compiler complains about a syntax error, it uses the syntax rules of the language at hand to determine whether the input really is a statement in that language or not. Similarly a musical grammar can be used to characterise music in a given style, say – what is it that makes one piece a blues piece and another not?

Grammars have been very productive in providing ways to manipulate natural and computer languages, and they can play an important role in machine understanding and generation of music also. A grammar can be used not only for recognition, but also for generation: given a grammar for, say, Java, it is possible to generate automatically well-formed Java programs from a specified subset of the language. There is no guarantee that these programs do anything interesting, but we can ensure that there are no syntax errors; if some randomness is built into the generation process, then this gives some sort of fair sample of the possible programs. Musical grammars have been used for both recognition and generation.

An example of this is Mark Steedman's grammar for the blues, at the level of chord sequences (Steedman 1996).[14] The initial grammar rule given there is this:

$$\textbf{12bars} \quad \rightarrow \quad \textbf{I} \quad \textbf{I7} \quad \textbf{IV} \quad \textbf{I} \quad \textbf{V7} \quad \textbf{I}$$

It says that the twelve bars can be made up of six successive sections, based on the chords:

$$\textbf{I} \quad \textbf{I7} \quad \textbf{IV} \quad \textbf{I} \quad \textbf{V7} \quad \textbf{I}$$

(in C major, this is C, C with flattened 7th, F, C, G with flat 7th, returning to C). Other rules allow variations by substituting different chords, in given contexts; these substitutions are known (in different form) to blues musicians, and are here incorporated into a grammar. The discussion in the paper on different grammar formalisms is beyond the scope of this chapter, but the ideas are important. Given a sequence of chords in this way, the grammar can be used to see if this is a possible blues sequence.

This grammar does not operate on the level of the individual notes played and sung, but already assumes a more abstract understanding in terms of successive chords in a given key – this is called the **harmonic** level of description.

[13] `http://en.scientificcommons.org/42491827`
[14] `ftp://ftp.cogsci.ed.ac.uk/pub/steedman/music/batat.ps.gz`

A much more elaborate approach to WTM using grammars is given by Lerdahl & Jackendoff (1983); in principle this applies across the board, with metrical as well as harmonic and melodic aspects included. It is, however, given somewhat informally, and does not get us easily to a practical way to parse WTM using all these aspects.

4.5.2 Style

There has been a lot of work on determining musical style automatically, from sound or from symbolic description. Statistical and machine learning techniques are proving to be good at this recognition task, by looking for patterns in the music. (A survey of some work using audio is given by Tzanetakis & Cook (2002).) This does not give a way to **generate** music in a given style.

David Cope has shown how it is possible to generate music in the style of a given composer, or genre, by a combination of analysing a number of pieces in the target style to look for common patterns (in this case, pairs of combinations of notes that listeners are likely to take as similar, with small changes in rhythm or pitch outline, for example). These become indicators of the style, but for the overall shape that the music should take, he supplies a hand-written grammar. Thus we can imagine new piano blues by combining a grammar, like the one mentioned above, with patterns of piano notes repeated across performances by a particular blues piano player.

See David Cope's homepage,[15] especially the section on Experiments in Musical Intelligence, with MP3s of machine-generated music in various styles, using the techniques he describes (Cope 1996, 2001).

4.6 Musical discourse

Let's now consider what might be used to support musical interactions mediated by machine. There are several scenarios where people want to do this sort of thing.

Accompaniment Systems to provide accompaniment to a soloist where the accompaniment and soloist are following a musical score. In this case, the system needs to track tempo changes made by the soloist, and deal with some amount of deviation from the scored notes, as a human accompanist would do. There are some impressive systems that achieve this; see this short article[16] and Christopher Raphael's 'Music plus one' page.[17]

Distributed ensembles There is also a desire to allow musical collaboration between people and virtual musicians who are in physically distant locations, perhaps as part of some larger production. The technology used for remote meetings in general is an obvious place to start, but there are possibilities for more sophisticated interactions that support musical interaction better. For a taster of this work, visit the telematic Circle.[18]

Real-time transformation of musical material With powerful signal processing available, musical sounds can be transformed in real time in a myriad of ways. Thus the sounds of a human performer can be mutated during performance, by

[15]http://artsites.ucsc.edu/faculty/cope

[16]www.jstor.org/stable/20055566

[17]http://xavier.informatics.indiana.edu/~craphael/music_plus_one

[18]http://www.deeplistening.org/site/telematic

someone controlling these transformations, or fed as input into other music sources. What is more interesting here is to have these transformations under the control of the performer themself, say by using some physical gesture that triggers a change, or doing this by some feature of the music itself. A very brief description of the former is here.[19]

New interfaces to (new) instruments There are new possibilities of ways of making sounds from traditional instruments, or from new instruments that may only exist in virtual, digital form. The physical acts involved in making music, by singing, blowing a wind instrument, and so on are part of the experience of making music which feel 'left out' with a push-button interface. Look at the short description of various interfaces at the start of the Stanford CCRMA course web page.[20]

Human/machine improvisation People have been working for some time towards the idea of having virtual musicians join in with human musicians, especially in music with a significant element of improvisation, where the music does not follow a score or a memorised model, but is produced spontaneously during performance. As we have seen, in music in the WTM tradition the improvised music is expected to make sense in its rhythmic and harmonic context (although this does not hold for all music, by any means). Given that human performers have some knowledge of the style in which the improvisation will evolve, it makes sense for a machine participant to have knowledge also of the style, both to follow the playing of others, and to generate suitable music itself. Having a grammar of the style is one way to achieve this.

In mixed ensembles of this sort, the aim is to enable shared guidance of the improvised music, so that the human is not subservient to the machine, nor the machine to the human. One interesting approach is the MIMI system from the University of Southern California. This page has some videos of its performance while improvising with a human in several styles.[21]

4.7 Exercises

1. **Beat tracking**
   Some music is intended to have a regular underlying pulse, but other forms of music are organised without any such perceptible regularity; sometimes there is a mixture; sometimes the beat is not obvious initially, but once established is easy for people to follow.

   Listen to the start of the following extracts, and see how easy or hard it is to tap along regularly with the music. What features of the music help or hinder in this task? How could a computational listening device make use of the helpful features?

   (a) Byrd *Pavane* [22]

   (b) Bach *Two part invention* [23]

   (c) Debussy *Clair de lune* [24]

   (d) Boulez *Dialogue de l'ombre double.* [25]

[19] http://www.infomus.org/Research/Mapping.html
[20] https://ccrma.stanford.edu/courses/250a/lectures/survey/
[21] http://www-rcf.usc.edu/~mucoaco/MIMI/
[22] http://www.mfiles.co.uk/mp3-downloads/Earle-of-Salisbury.mp3
[23] http://www.mfiles.co.uk/mp3-downloads/invention2part-no4-el.mp3
[24] http://www.mfiles.co.uk/mp3-downloads/debussy-clair-de-lune.mp3
[25] http://www.last.fm/music/Pierre+Boulez/_/Strophe+VI

2. **Hierarchical metre**
   (a) Often rhythms are organised around a hierarchy, above and below the level of the beat. For example, the following music is written with an underlying beat of about 100 beats per minute, with the beat subdivided on 4, while the beats are organised in groups of three, where the first of each group of 3 is stronger. Listen to the start of this and listen out for this structure.

   Albeniz, *Leyanda*. [26]

   (b) Usually beats are grouped in groups of 2, 3 or 4. Grouping in 5 or 7, for example, is unusual. Listen for the unusual grouping here – the conductor's movements reflect the grouping (but not very obviously). Tchaikovsky's *6th symphony, second movement*. [27]

   Also try Pink Floyd, *Money*. [28]

3. **Pitch organisation**
   Western tonal music takes the pitches it uses from the notes playable on a piano and guitar. Other traditions make different distinctions, fewer in some folk traditions, more in Indian music.

   Listen to this piece which uses twice as many pitches as normal, by having pianos so that the notes on one are slightly higher than the notes on the other. [29]

   Because we are not used to making sense of this sort of music, it can sound 'out of tune'. Does this music 'make sense' to you? What, if anything, does it mean for something to "make sense" as music?

4. **The musical dice game**
   A method of putting together a large number of possible pieces of music from basic building blocks was produced in the 18th century. There is information about the game here.[30]

   You can use the first implementation to generate a piece; you will need to be able to play midi sound files to listen to the result.

   This can be compared to a way of generating poetry by selecting from possible first lines, possible second lines, and so on — Queneau did a version in French; there is an English version on-line.[31] Notice that this respects the form of a sonnet, in terms of syllables and rhyme scheme.

   In each case, the object is to produce an output which is well-formed (as a minuet, or as a sonnet). Explain how the building blocks are designed so that they will fit together appropriately. In the musical case, this should involve the organisation of metre, and of pitch and key – other organisational properties may be relevant also.

4.8 Further reading

For mathematical, rather than computational background, the work of Fauvel et al. (2003) provides a readable account of the interaction between music and mathematics. It is accessible for those with some knowledge of musical notation and comfortable with non-specialised mathematics.

[26] http://www.mfiles.co.uk/mp3-downloads/leyenda.mp3
[27] http://www.youtube.com/watch?v=2SEDU8AyxVU&feature=related
[28] http://www.last.fm/music/Pink+Floyd/_/Money
[29] http://www.youtube.com/watch?v=EU85bUyDPWs
[30] http://webplaza.pt.lu/public/mbarnig/pages/dicemus.html
[31] http://www.bevrowe.info/Poems/QueneauRandom.htm

Two important research centres for work in this area are IRCAM in Paris[32] and the Stanford CCRMA.[33] Both have several research groups, and wide interests, as well as strong links to musical performance.

Current research is scattered across several areas and journals. *The Computer Music Journal*[34] has many relevant articles, and is available on-line.

[32] `http://www.ircam.fr/?L=1`
[33] `https://ccrma.stanford.edu/`
[34] `http://www.mitpressjournals.org/cmj`

Chapter 5

Music information retrieval

5.1 Introduction

Volume 2 of the subject guide for 2910227, *Interactive multimedia* introduced the very basics of metadata- and content-based multimedia information retrieval. This chapter expands on that introduction in the specific context of music, giving an overview of the field of music information retrieval, some currently existing systems (whether research prototypes or commercially-deployed) and how they work, and some examples of problems yet unsolved.

Figure 5.1 enumerates a number of tasks commonly attempted in the field of music information retrieval, arranged by 'specificity', which can be thought of as how discriminating a particular task is, or how clear the demarcation between relevant and non-relevant (or 'right' and 'wrong') retrieval results is. As will become clear through the course of this chapter, these and other tasks in music information retrieval have applications in domains as varied as digital libraries, consumer digital devices, content delivery and musical performance.

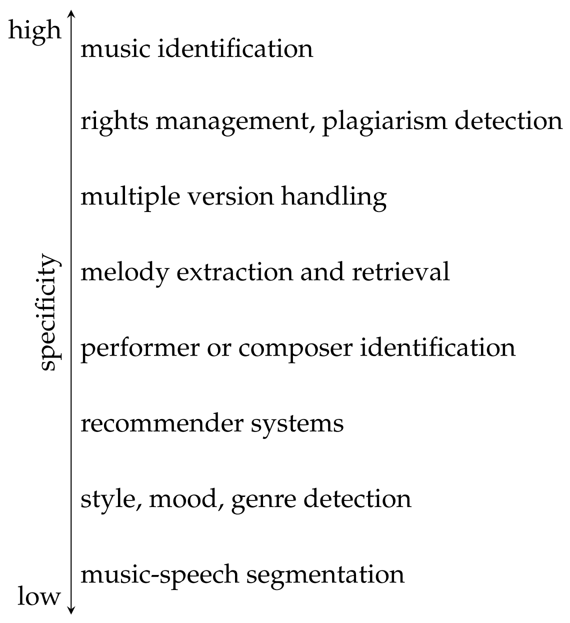

high
music identification
rights management, plagiarism detection
multiple version handling
specificity
melody extraction and retrieval
performer or composer identification
recommender systems
style, mood, genre detection
music-speech segmentation
low

Figure 5.1: An enumeration of some tasks in the general field of music information retrieval, arranged on a scale of 'specificity' after Byrd (2008) and Casey et al. (2008). The specificity of a retrieval task relates to how much acoustic and musical material a retrieved result must share with a query to be considered relevant, and how many documents in total could be considered to be relevant retrieval results.

This chapter includes a number of references to the published scientific literature. Where possible, those references have been made to papers or other documents which were freely accessible online and findable by basic web searches at the time this chapter was written. In addition to this, the International Symposium of Music Information Retrieval publishes the complete cumulative proceedings of the 10 (so far) annual events held, available at `http://www.ismir.net/proceedings/`, where much more information on the topics introduced here can be found.

Additional reading

Müller, M., *Information Retrieval for Music and Motion*. (Berlin: Springer-Verlag, 2007) [ISBN 9783540740476 (hbk)].

Witten, I.H., A. Moffat and T.C. Bell *Managing Gigabytes: Compressing and Indexing Documents and Images*. (San Francisco: Morgan Kaufmann Publishing, 1999) [ISBN 1558605703 (hbk)].

Van Rijsbergen, C.J. *Information Retrieval*. Available at `http://www.dcs.gla.ac.uk/Keith/Preface.html`.

5.2 Genre classification

A huge variety of different kinds of sounds are considered to be music: the very definition of music is fraught with difficulty. Faced with such diversity, between musics created at different times, in different geographical locations, in response to different social contexts, it is natural to attempt to classify music into different varieties or **genres**. For example, one can classify by the geographical area of origin; or by cultural standards (such as the distinction between art music, popular music and traditional music). Each of these broad categories still contains a diverse set of sounds, and can be divided into many smaller sub-categories.

The Western music industry produces genre labels (and indeed genre **taxonomies**: particular divisions of the category of music) for its own purposes. For example, physical shops must organise their music collections somehow, and so Western music retailers divide up the space available to them for categories such as ‘Classical’, ‘Jazz’, ‘Rock’ and ‘Soundtracks’; in some cases dividing the space within those categories alphabetically by composer or artist; and in some cases having subgenres. This physical division of the space guides the shopper towards the items they are most likely to want to purchase (Pachet & Cazaly 2000). The same steering also occurs in online music retail; Internet music sellers often allow the user the option to browse the available collection by genre, sometimes to quite a depth of categorisation.

A commonly-attempted task in music information retrieval is to automatically associate a genre label with a given track, working solely or primarily from its audio content. One approach to this is to treat it as a classification problem, and to model genres as clusters in some feature space; by learning the parameters of those clusters from labelled training data, one can then assign a cluster (and hence a genre label) to an unknown track.

For example, a track’s acoustic content might be modelled by an overall average spectral feature such as the **cepstrum**, which is a d-dimensional quantity. If for a given genre label we have a set of N tracks for which we have extracted the cepstrum, and we make the assumption that the clusters are represented by single

Gaussians, we can represent the cluster's density function as:

$$p^{(i)}(\mathbf{x}|\mathbf{m},\mathbf{S}) \propto \frac{1}{\sqrt{|\mathbf{S}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{m})\mathbf{S}^{-1}(\mathbf{x}-\mathbf{m})^T\right)$$

by estimating $\mathbf{m}^{(i)}$ and $\mathbf{S}^{(i)}$ from the training data as:

$$\begin{aligned}\hat{\mathbf{m}}^{(i)} &= \frac{1}{N}\sum_N \mathbf{x}^{(i)} \\ \hat{\mathbf{S}}^{(i)} &= \frac{1}{N-1}\sum_N (\mathbf{x}^{(i)} - \hat{\mathbf{m}}^{(i)})^2.\end{aligned}$$

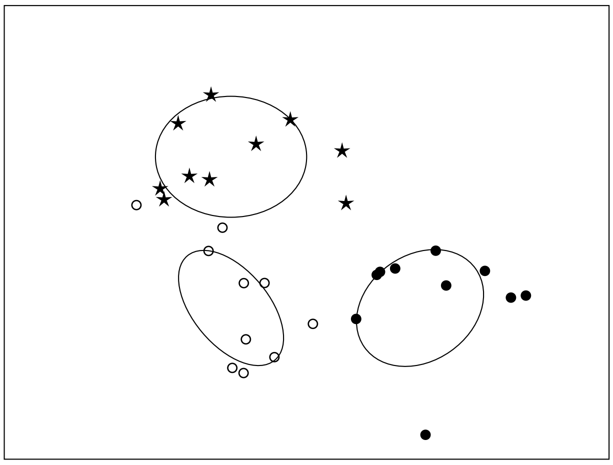


Figure 5.2: Schematic illustration of clustering in a feature space: the individual points represent the position of a piece of music in that feature space, in this context attempting to capture the notion of 'genre'. Because the clusters (whose central support is represented by the ellipses) do not overlap very much, this clustering can be used as a classifier.

Figure 5.2 shows this estimation and clustering schematically: the ellipses represent the concentration of the clusters corresponding to each of the three genre labels present in the model. In this schematic case, the clustering can be used as a classifier; however, in the application of clustering to the genre classification task, the picture is more often like Figure 5.3, where there is significant overlap between the clusters in the feature space, and hence significant mislabelling if the clusters are used to classify unknown data.

While improvements can be made to the situation in Figure 5.3 by the use of more sophisticated features and clustering models (Pampalk et al. 2005), it should be said that because of the different possible meanings and uses of 'genre' that there will always be ambiguity present.

Firstly, the genre of a track is not necessarily related to how something sounds: in many cases, the genre of a track was chosen by a recording label essentially to attempt to maximise the exposure of that track in shops. If a shop had a particular genre near the front, or if the newspapers had been talking about a specific musical genre in the recent past, that genre label would be chosen for more tracks.

In addition, genre being a personal and social construct, the identification of a track's genre differs when performed by different people: not because people do not

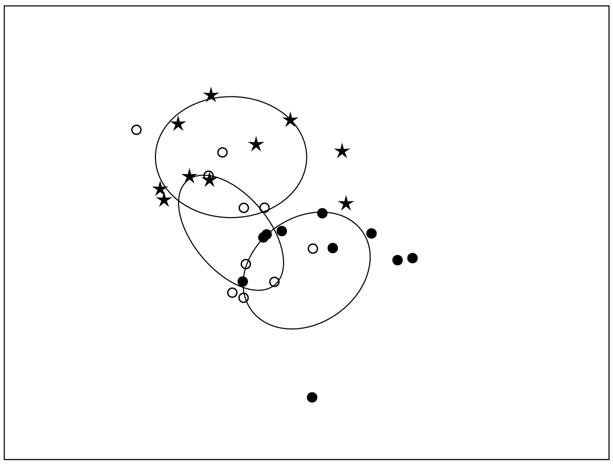


Figure 5.3: A more realistic depiction of clustering in the case of genre: features extracted from the musical content or metadata (apart from the ground truth 'genre' label itself, of course) are not sufficiently discriminating to allow separation of genre clusters.

know the 'right answer', but because there is no *a priori* right answer. People have internal, consistent models of genre (Craft et al. 2007), but they do not necessarily line up perfectly with the industry's labels or with any individual retailer's shop organisation plan. The social aspect of genre suggests a different method, to use community metadata and collaborative filtering as the way to categorise. This can be seen in action in social radio Internet sites such as last.fm or Spotify, which use genre to inform the user's navigation through the collection of available music, and can even be used to build a genre classifier based on words shared in common in text about songs as returned from web searches (Whitman & Rifkin 2002).

5.3 Recommender systems

The primary use of a recommender system is probably in the commercial arena, where the music information retrieval task is to find items which the user would be interested in buying. In this context, there is also a useful evaluation metric for any system purporting to perform this task: success or failure can be quantified in terms of the amount of money spent by users of the system.

Perhaps the most obvious example of such a recommender system, then, is the system generating suggestions of other items to buy at the Amazon[1] online retailer. This is also an interesting system in that it performs no analysis of content or even musical metadata, but works exclusively through association: the website tracks information from a large number of users, remembering details such as users' paths through the collection of available items to purchases and items purchased by the same user; recommendations are then made by establishing a similarity between a given user's profile and some subset of the stored aggregate user data, allowing prediction of suitable items to present for inspection.

One weakness of the system behind Amazon is that its only information about any user is the previous interactions that that user has had with the Amazon service. In particular, Amazon does not know about any purchases made elsewhere, and so

[1]`http://www.amazon.com/`

will naturally suggest items which the user already owns (there is provision for the user to feedback 'I own this'). A more fundamental weakness, from the point of view of music information retrieval if not necessarily from a cost-benefit analysis, is in the fact that the Amazon recommender does not use any musical content or metadata information. This leads to suboptimal treatment of, for example, multiple performances of the same work: given a pair of popular pieces of music (often recorded or covered) there will be many subsets of users with one recording of each pair, which leads Amazon to suggest many instances of the same piece of music if a user shows an interest in any of them.

From the point of view of Music Information Retrieval, Amazon's recommender suffers through its generality: it is applicable to everything from lawnmowers to jewellery. A more targeted recommender is the 'Genius' feature in Apple's iTunes music player, which establishes commonalities between tracks on a user's system with iTunes playlists containing those tracks created by other iTunes users. Again, the commercial motivation is clear: the recommendations generated by the 'Genius' feature come with the ability to purchase recommendations corresponding to tracks not already in the user's collection. Soundbite[2] is another system currently working as an iTunes plugin, similar in nature, but using a summarisation of the musical content of a track ((Levy & Sandler 2006)) rather than the collaboratively-filtered playlist metadata.

The above recommenders use a large, previously acquired set of data, whether from having access to all playlists, purchase data or prior musical feature extraction and analysis: they will therefore only be able to recommend tracks (or however musical works are represented) that exist in their own databases, which consequently are not so useful for exploring niche, targetted collections (or in the case of Apple's iTunes, recommendations based on the Beatles collection). By way of contrast, there exist recommender systems based on navigating a standalone collection, where the emphasis is not in finding new material to offer a user, but in providing a path to navigate through the existing material: a necessary tool in this age of consumer electronic devices with tens of thousands of tracks on them.

There have been many systems designed to visualise and navigate through collections of music, some with more specialised capabilities than others. A recent survey (Donaldson & Lamere 2009) demonstrated over 85 collection visualisers, with different aims and different visual representations, but all aiming to illustrate the range of a collection and in most cases helping to navigate through that collection. Among them are Islands of Music (Pampalk 2001), showing how tracks are grouped by particular acoustic features; the Musicream (Goto & Goto 2005) playlist generator and editor; and the mHashup search interface (Magas et al. 2008) for locating not just tracks but portions of a track by similarity (see the next section) or by geographic location.

5.4 Musical document retrieval

Figure 5.4 illustrates the typical workflow in content-based document retrieval. A database of music 'documents' (usually the audio content, along with whatever metadata is available) has some musical features (such as the **audio spectrogram**, or more usually some transformation of that spectrogram such as a **chromagram**) extracted, and those musical features are stored alongside the documents

[2]http://www.isophonics.net/

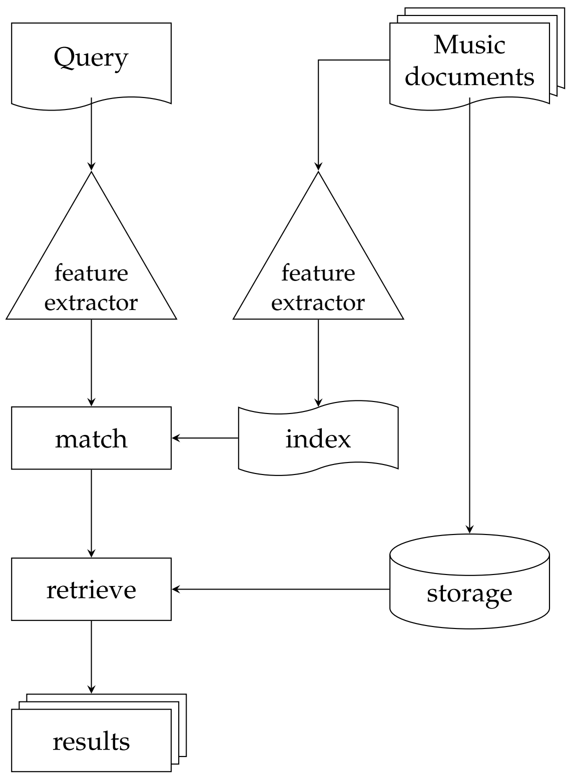


Figure 5.4: An illustration of the typical workflow for content-based document retrieval.

themselves. The retrieval process starts by extracting the same musical features from the query track, resulting in a **feature vector** which can be compared in some distance space with the database document features; if the distance space is indexable, this can be done quickly. Then the retrieval itself merely looks up from storage the documents whose feature vectors are closest to the query's feature vector, up to some maximum distance, and those documents are presented to the user as the results of the query.

This workflow boils down to one of several generic problems: **identity retrieval** for when an exact match is being searched for; the **nearest-neighbour search** for the 'closest' match; ***k*-nearest neighbours** for a *k*-element list of nearest matches; and ***r*-near neighbours**, where near neighbours are only considered for retrieval if their distance from the query is less than *r*.

5.4.1 Music identification

One popular and commonly-used approach to music identification relies on information about the content rather than the content itself. In this case, the documents are not individual tracks but albums, usually in CD form. In this case, used by such online services as freedb[3] and Musicbrainz,[4] the feature extracted, stored and indexed is the disc **TOC** (Table Of Contents), a representation of the start positions and lengths of the tracks on the disc. This feature is highly specific,

[3]Started when the CDDB service was taken proprietary by its owner; see `http://www.freedb.org/`
[4]`http://www.musicbrainz.org`

because it is extremely rare for different albums to share the same lengths of tracks in the same order; thus, an exact match in TOC space identifies an album (and consequently the component tracks) with very high confidence. One weakness of this approach is that slight differences in the generation of CDs, even from the same source audio material, can produce different TOCs, which will then fail to match each other.

Musicbrainz additionally provides identification and metadata retrieval based on an analysis of the musical content of the file, using the AmpliFIND (formerly MusicDNS) acoustic fingerprinting service[5] to retrieve a unique identifier for individual tracks. This allows for more robust album matching, at the cost of higher computational requirements (for computing and matching audio fingerprints rather than merely using the very simple TOC information).

The Shazam service[6] is a purely content-based music identification service, aiming to be robust not only against different pressings of the same CD but also against certain kinds of ambient and digital noise; the recognition service offered even works when the music is transmitted over a mobile phone line from a noisy environment such as a nightclub.

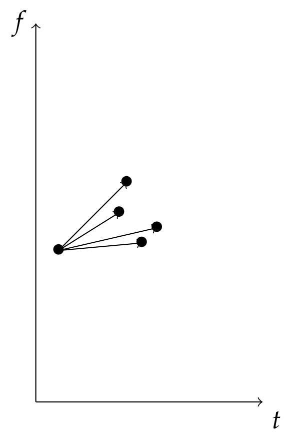


Figure 5.5: A geometric spatial feature, similar to those used in the Shazam music recognition service. The points considered as part of the feature are intensity peaks in the spectrogram, in a frequency range that is carried over telecommunications links such as mobile telephones.

Shazam (Wang 2003, 2006) works by storing an index of groups of intensity peaks in the spectrogram of a track, illustrated schematically in Figure 5.5. The presence or absence of these peaks are not audible in themselves to listeners, but provide a clear marker or fingerprint to the track, as the precise arrangement of those peaks is distinctive. Although the presence of a single such feature is not enough to identify a track with confidence, the specificity of the search can be increased by requiring the query to match not just one such arrangement but two with the same temporal spacing as in the document database (Figure 5.6).

The Shazam identification service is, as described above, robust to ambient and digital noise. However, it will only successfully match against the same *recording* as is present in the Shazam database: live performances or alternate recordings

[5] `http://www.musicip.com/`
[6] `http://shazam.com/`

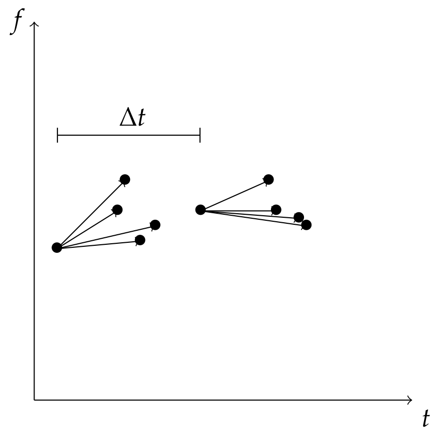


Figure 5.6: The specificity of the Shazam features is increased by requiring not only that the features be the same in query and database index, but also that they be separated in time by the same amount.

(bootlegs from a concert, say) will have a slightly different expressive performance, and so will not be considered to have the same fingerprint. It is for this kind of match, slightly down the specificity scale where we talk about handling multiple versions of the same work, that we need to explicitly perform similarity search, described in the next section.

5.4.2 Content-based similarity search

The systems described in the previous section have the function of searching and retrieving the **identity** of a track, possibly resistant to certain sorts of noise, through a combination of collection metadata and audio fingerprinting. However, many music document retrieval tasks are better described in terms of **similarity** rather than identity: multiple performances of the same work, remixes or mashups of samples from a work; cover versions and variations on a theme will all share more or less of their nature with another of the same class, without being identical. The approach to similar document retrieval is again summarised in Figure 5.4, but here the distance space is not simply defined by an equality predicate but includes the notions of relative closeness.

The concept of similarity is less specific than identity; additionally, the less specific the particular variant of similarity being considered, the more subjectivity comes into play. Therefore, it would be wrong to consider audio similarity search as a single monolithic task; instead, a similarity search must be defined more precisely before it can be performed. Some uses of similarity search include navigation through collections (by association); digital rights management (detecting distorted or sampled copies of copyrighted works); and performance tools through concatenative synthesis, such as in the Soundspotter[7] tool (Casey 2009) – and indeed the classification approach to genre identification described in Section 5.2 can be viewed as a content-based similarity search.

This leads to many different approaches being taken in the general area of similarity search. For some applications, the order in which musical 'events' happen is highly

[7] `http://www.soundspotter.org`

significant: for example, it would be almost impossible to consider two tracks to be multiple performances of the same work if the notes did not happen in the same temporal order, even if the overall note content was the same; by constrast, one track can be a remix of another while using the musical content of the original in a different sequence.

This difference is mirrored in a pair of significantly different approaches to content-based similarity retrieval: the **bag-of-frames** approach (Aucouturier et al. 2006), where the signal is represented as the long-term distribution of the 'local' (frame-based) acoustic features, discards any information about short-term organisation (such as might be found in musical structures such as bars or phrases); **sequence** approaches, instead, perform matches while keeping the time ordering information, at a cost in computational requirements. This difference in approaches can be characterised as the difference between representing the overall sound of a track, compared with the trajectory taken; the bag-of-frames approach will work well when there is little short-term structure, or when that short-term structure is not relevant to the kind of similarity match being performed.

5.5 Segmentation

In order to perform other analyses of various kinds on musical data, it is sometimes desirable to divide it up into coherent segments; see Figure 5.7 for an illustration. For example, transcription (see the next section) depends on being able to identify individual notes, and any musically ambitious transcription algorithm will also wish to group those notes into bars. Meanwhile, some portable music players attempt to synchronise the music played with the listener's movements (for example, when exercising or running), aiming to select music with the same **tempo** or beats per minute as the repetitive motions. Segmenting at **phrase** or **section** boundaries, identifying high-level musical structure, is of particular use in particular performance tools, such as karaoke machines (Goto 2006),[8] but also for finding a representative segment to use as a 'thumbnail' when a small portion of the track is needed (for example, as a sample in an online music store).

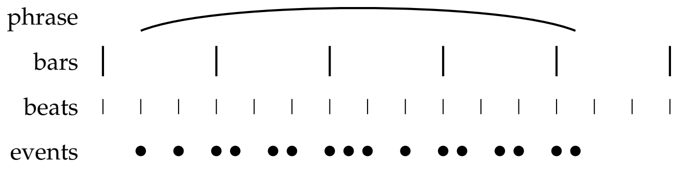


Figure 5.7: An illustration of various levels of possible segmentation: a **phrase** is made up of multiple **note** events, which can be divided into **beats** and **bars**. Importantly, observe that these segmentations do not form a single hierarchy: in this example, the phrase boundary is not aligned with bars.

Onset or event detection is normally phrased as a task in signal processing, where low-level characteristics of the signal are analysed for differences indicative of the introduction of a new signal component (Dixon 2006). However, there should be a distinction made between detecting at what point there is a difference in the signal's characteristics and determining automatically the time at which a listener would

[8]such as in SmartMusicKIOSK, `http://staff.aist.go.jp/m.goto/SmartMusicKIOSK/`

hear the event (Klapuri 1999): the latter can often be relevant for certain applications.

There are many approaches to beat detection, such as in Davies & Plumley (2007); for many kinds of music where there are strong, regular percussive elements, beat detection algorithms can attain a high accuracy, and indeed for the kinds of music which include electronically-generated regular drum tracks (such as many forms of Western popular music), beat detection can be nearly perfect. However, even in such simple cases there is often a simple source of ambiguity in the difference between **tatum** (the fastest regular sequence of events) and **tactus** (the frequency at which people would tap along): often there is a difference of a factor of two, with each tactus beat being composed of two events. Generally, people tend to tap along at speeds between 60 and 180 beats per minute, and music is generally written to conform to that range, but that range does leave scope for differences of opinion as to what the beat actually is.

The identification of musically relevant segments in music is a particularly interesting one: in that in general, a large amount of contextual information must be used to assess what distinguishes different sections from each other. For example, in Western popular music, verses, choruses and other sections are often distinguished from each other by the instrumentation (and hence the timbre) in each section; while in classical Sonata Form, the 'subjects' (segments) are distinguished by their position in the development of the piece. For popular music, a technique (related to the simple approach to genre classification described in Section 5.2) based on classifying individual audio frames to genre labels works adequately, particularly if supplemented by some smoothing (Abdallah et al. 2006) to avoid short sequences of frames being labelled as a section; for classical works, attempting to identify repeated musical material (Rhodes & Casey 2007, Müller & Clausen 2007) is a more fruitful entry point than timbral cues.

5.6 Transcription

Transcribing a musical audio signal into a form of musical notation familiar to practitioners is a seductive goal. Experienced practitioners can perform that task themselves, at least for moderately simple signals, and amateur listeners can memorise and sing along to their favourite tracks. Yet even such apparently simple things such as melody extraction from **polyphonic** audio (multiple sources, as opposed to single source **monophonic** music – not to be confused with **mono** and **stereo** recordings) are beyond the capabilities of current systems and of current understanding.

Even the transcription of a single musical instrument's audio is a difficult task (Cemgil 2004); real musical instruments are all different, have acoustic content different from a pure sinusoid, and allow for expressive performance techniques such as vibrato which further removes their sound from an 'ideal'. The basic idea in analysing monophonic audio is to identify intensity peaks in the spectrogram, and identify high-intensity regions of approximately the same frequency as coming from a note; this hypothesis would be reinforced by finding harmonically-related frequency peaks, which also allow an estimate of the **fundamental frequency** (corresponding to the pitch of the note).

This basic idea for transcription needs considerable refinement for even special cases of polyphonic music, where all the notes have been sampled individually beforehand (Abdallah & Plumbley 2006). The reason that this is such a difficult task is because the human transcriber benefits from years of experience of listening to music generally, and usually the same kind of music as they transcribe; those years of experience provide a strong set of expectations, allowing inference of a clean transcription from the noisy acoustic evidence. To perform transcription automatically to a similar level of accuracy requires encoding of these prior expectations; it appears not to be the kind of problem that can be solved merely by applying more computational resources.

Adequate transcription would have a number of uses, such as in **query by humming** systems, metadata access, and as a compositional or notational tool, as well as a front-end to other music information retrieval systems.[9] However, perhaps surprisingly, even when working from a high-level or **symbolic** representation of music, it can be surprisingly hard to extract information such as the melody (Wu & Li 2007) or chord labels (Rhodes et al. 2007) for that music.

5.7 Summary and learning outcomes

This chapter surveys a number of techniques used in the field of music information retrieval, and discusses the implementation of particular systems intended to perform certain music information retrieval tasks, including strengths and weaknesses of particular approaches.

With a knowledge of the contents of this chapter and its directed reading and activities, you should be able to:

- explain the concept of specificity in the context of music information retrieval
- understand the different ways in which genre labels are generated, and the implications that this has for automatic retrieval of those labels
- describe different forms of recommender systems, suggesting particular approaches for particular recommender applications
- summarise the workflow for content-based similarity retrieval
- describe content-based and metadata-based approaches to music identification and fingerprinting
- describe applications of audio similarity search
- explain the uses of different levels of segmentation, and techniques used to obtain them
- describe a simple method for obtaining a transcription of monophonic music, and the difficulties in general music transcription.

[9]such as the Musipedia database, `http://www.musipedia.org/`

Appendix A

Sample examination paper

Important note. The information and advice given in the following section are based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check the current *Regulations* for relevant information about the examination. These are available on the External System website at: `http://londonexternal.ac.uk/current_students/general_resources/regulations/index.shtml` You should also carefully check the rubric/instructions on the paper you actually sit and follow those instructions.

Using this sample examination paper. This paper, and the examiner's notes contained within, are intended as a guide to help you study. Where sample answers are provided, it should be noted that they are only one possible example, intended to show the kind of response that is required. They are not definitive answers, and there may well be other approaches that a student might take in answering them, in an examination, which might also be correct. The comments are intended for guidance, not prescription.

Sample paper and examiner's notes

UNIVERSITY OF LONDON **2910346**

BSc/Diploma Examination
for External Students

CREATIVE COMPUTING

Sound and Music: Sample Paper

Dateline:

Duration: 2 hours, 15 minutes

There are five questions in this paper. You should answer no more than **THREE** questions. Full marks will be awarded for complete answers to a total of **THREE** questions. Each question carries 25 marks. The marks for each part of a question are indicated at the end of the part in [.] brackets.

There are 75 marks available on this paper.

Electronic calculators must not be programmed prior to the examination. Calculators which display graphics, text or algebraic equations are not allowed.

Sample paper and examiner's notes

Question 1 Computational Models of Music Cognition

(a) Within cognitive science, what is a computational theory, and what is its purpose? [6]

A computational theory is a formalisation of the computational requirements of a cognitive task, and is the highest level at which cognitive scientists typically study the mind. A cognitive theory is formulated by analysing the various outputs of a cognitive task resulting from different inputs. The specification of a computational theory guides experimental design, and the implementation of computer simulations, and provides a necessary explanatory framework in which to interpret empirical results.

(b) What do cognitive scientists mean by the *musical surface*? [4]

The musical surface is the level at which cognitive scientists study music. It is an abstraction of the physical musical signal into a sequence of discrete, musically salient events, such as notes and rests. It a methodological idealisation based on the psychological understanding of how music is perceived by the human mind, which allows scientists to study musical processes while not necessarily being concerned with the functionality of lower level perceptual mechanisms.

(c) The following table shows the symbolic representation of a melody in terms of basic attributes.

chromatic pitch	65	67	69	69	65	72	70	69	67	62	64	65
tonic reference	5	5	5	5	5	5	5	5	5	5	5	5
onset	0	4	8	10	12	20	22	24	30	32	35	36
duration	3	2	2	2	6	2	2	4	2	2	1	12

Extend this representation by writing down the values of the following two attributes:

i. chromatic interval [2]

ii. scale degree (note: tonic reference 5 is the key of F). [2]

chromatic interval	$\perp$	*2*	*2*	*0*	*-4*	*7*	*-2*	*-1*	*-2*	*-5*	*2*	*1*
scale degree	*0*	*2*	*4*	*4*	*0*	*7*	*5*	*4*	*2*	*9*	*11*	*0*

(d) In a symbolic music processing system, you should assume that the variables `chromaticInterval` and `onset` exist, each of which has been assigned an array containing the corresponding values in the table above and from your answer to question c.(i).

Write down a method for each of the following attributes, in Java code. The methods should accept a single array argument, and return a new array representing:

i. the contour of the melody; [4]

Sample paper and examiner's notes

ii. the inter-onset-intervals of the melody. [4]

Example method for contour.

```
static int[] contour(int[] intervals)
{
  int[] contour = new int[intervals.length];
  for(int i=0; i<intervals.length; i++){
    int interval = intervals[i];
    if(interval == undefined){
      contour[i] = undefined;
    } else if (interval == 0){
      contour[i] = 0;
    } else if (interval > 0){
      contour[i] = 1;
    } else {
      contour[i] = -1;
    }
  }
  return(contour);
}
```

(e) Give an example of a learning model that has been published in the music cognition literature. Describe what and how the model is able to learn. [3]

The statistical model proposed within the IDyOM project is a learning model developed by Marcus Pearce and other researchers based in the Intelligent Sound and Music Systems research group at Goldsmiths. The model learns the conditional probabilities of a range of musical surface melody attributes, using the Predication by Partial Match (PPM) algorithm. The model has a long-term model, which is trained on a corpus of music, representing a listener's musical knowledge, and a short-term model, which is trained on the melody being analysed. The model can be used to predict the likelihood of pitches given a melodic context, and by measuring uncertainty, the model can also be used to predict phrase boundaries.*

Sample paper and examiner's notes

Question 2 Computer Music

(a) Name three early computer music pioneers, and for each, provide an example of why they are significant figures in the history of computer music. [6]

Example responses:

Pierre Schaeffer was a leader in the development of 'musique concrète', an approach to musical composition that began in the late 1940s, and utilised sound recordings as compositional material. He pioneered the use of early recording equipment for creative purposes, and also published theoretical writing on acousmatic music, which have had a considerable impact on later developments in electroacoustic and computer music aesthetics.

Iannis Xenakis was a composer and architect, who pioneered the use of mathematical models in compositional practice. He began composing seriously in the 1950s, and published his influential book Formalized Music *in 1971. Also in the 1970s, Xenakis developed the UPIC system, which allows the user to draw waveforms and amplitude envelopes as a means of generating new sounds and musical compositions.*

(b) i. What is the distinction between ring modulation and classic amplitude modulation? [2]

In amplitude modulation, the modulator signal is unipolar – i.e. it is a constant shifted signal typically in the range [0, 1]. This addition of DC means the frequency component of the carrier is present in the output (unlike ring modulation).

ii. Draw an accurately labelled diagram, in the frequency domain, of the signal resulting from the amplitude modulation of the following two signals: [6]

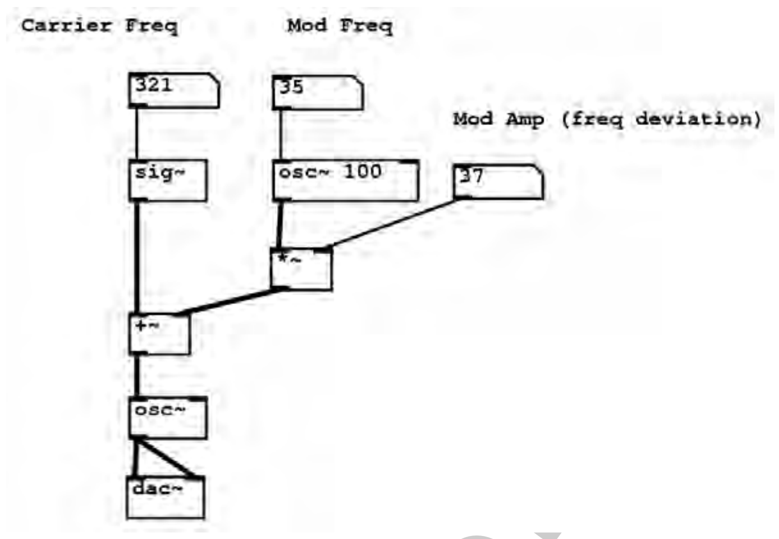
- carrier frequency $f_c = 1000$ Hz
- carrier amplitude $a_c = 1$
- modulator frequency $f_m = 360$ Hz
- modulator amplitude $a_m = 1$.

The diagram should have the carrier frequency component at 1000 Hz, and contain two sidebands of 0.5 amplitude, at 640 Hz and 1360 Hz respectively. Additional mark for correct labelling.

(c) Draw a simple Pure Data patch producing the frequency modulation of two sinusoid signals, and sending the resulting signal to the computer's sound output. [5]

2910346 PAGE 4 of 10

Sample paper and examiner's notes



(d) Describe the following synthesis methods, and state the strengths and weaknesses of each technique from the point of view of a sound designer: [6]

i. subtractive synthesis;

ii. additive synthesis.

Subtractive synthesis is a method of generating new sounds by filtering harmonics from a source sound with a rich harmonic spectrum. Traditionally, oscillators such as pulse wave or sawtooth wave generators are used as sources, which create pitched sounds with strong overtones. Noise sources can also be used, which are particularly useful for synthesising percussive sounds. Such oscillators require only relatively simple analogue circuitry to build, or can be easily and efficiently implemented in software, so they offer a lot of flexibility to the sound designer at little cost. Also, the method is very intuitive, since the model of source + filter is the same as that of the human vocal tract. Although several oscillators can be combined, and low-frequency oscillators can be used to modify control signals in order to add variation, achieving very naturalistic or harmonically complex timbres can be difficult.

A similar description should also be provided for additive synthesis, which also makes similar observations about usability, practicality, and the sonic possibilities and limitations.

Sample paper and examiner's notes

Question 3 Swarm music

(a) Write an essay discussing the use of principles of self-organisation in the generation of music. Consider both technical and aesthetic considerations, and also any potential limitations of the technique. [25]

The essay should contain a brief introduction, which summaries the argument and broader theme of the essay. There should also be a conclusion, again summarising key points, and offering some wider perspective. Intermediate paragraphs should each make a clearly defined statement, which together form a structured argument.

Some key points that might be addressed.

- *Self-organisation is a process often observed in the natural world.*
- *Self-organising systems are dynamic systems comprised of particles.*
- *Particles follow very simple reactive rules; there is no external or higher-level control.*
- *Complex coordinated behaviour emerges from a swarm of particles interacting within some environment.*
- *There are many approaches to generating music, some aim to explicitly model or learn aspects of musical style, others may be extremely abstract, or take inspiration from other domains (e.g. natural phenomena).*
- *Self-organisation may be an appropriate model for certain kinds of music – particularly improvised music.*
- *Swarm systems offer great interactive potential.*
- *There are problems in applying concepts from one domain (nature) to another (music).*
- *Practical decisions when using swarms to generate music. Using swarms to trigger notes? Or to generate separate pitch, rhythm, or dynamics data? Monophonic or polyphonic? Melodic or percussive?*
- *What kinds of tasks are swarms best suited for?*
- *Are swarms capable of generating sophisticated compositions? Or would they need combining with other generative techniques?*
- *Could swarms ever be genuinely creative?*

Sample paper and examiner's notes

Question 4 Music information retrieval

(a) A digital music library typically contains a range of different kinds of digitised documents, ranging from audio recordings to scanned manuscripts.

List five different kinds of documents, each of which could be said to be an example of a particular piece of traditional folk music. [5]

Examples of digitised documents relating to a single piece of folk music might include:

- *field audio recordings of the work performed by traditional musicians;*
- *audio recordings of the work by other performers; for example, professional musicians in a recording studio, with possibly very different interpretations intended for specific audiences; for example, pop music releases;*
- *audio recordings of re-orchestrations; for example, of melodies used in classical or film music;*
- *audio recording which use samples of folk music; for example, remixes;*
- *scanned pages of traditional notations; for example canntaireachd;*
- *scanned pages of published anthologies of folk songs collected and notated by ethnomusicologists;*
- *scanned pages of re-orchestrations or piano reductions;*
- *computer files containing machine readable music notation;*
- *text files containing lyrics;*
- *MIDI file transcriptions.*

(b) Define the following evaluative criteria: [6]

i. precision;
ii. recall;
iii. F1.

i. Precision reflects the true positives as a proportion of the positive output of the model.

ii. Recall reflects the true positives as a proportion of the positives in the original data.

iii. F1 is the harmonic mean of precision and recall:

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}.$$

(c) In a music retrieval system containing audio recordings of pop songs and cover versions, describe two problems that will typically lead to poor recall performance. [6]

Sample paper and examiner's notes

Two common problems would be global tempo variation (cover versions played at different speeds to the original); and also transposed variations (cover versions played in a different key). If both these issues were not addressed, recall performance would be low because the system would fail to recognise matches, resulting in a high number of false positives.

(d) Imagine that you have been hired by a record company to develop a system for identifying illegally used samples taken from recordings in their catalogue. Provide an overview design for such a system, and justify why the techniques you would use are suitable for this task. [8]

The answer to this question requires a sensible suggestion of infrastructure to enable the system to operate. For example, a database containing all of the record label's recordings must be accessible, and another database must be created containing all the extracted features from the record label's recordings. This second database should be automatically updated periodically as new material is added to the first database. If the system is to identify samples illegally used in recordings not owned by the company, a third database of features must be accessible or created, either in agreement with another record company, or from otherwise legally gathered audio.

Points to discuss.

- *An appropriate definition of 'similarity' needs to be given for this system.*
- *A range of appropriate features will be needed in order to capture such a potentially broad range of matches (exact samples through to heavily transformed samples).*
- *A sequence of features might be more appropriate than a bag-of frames.*
- *Computational and efficiency issues – e.g. indexable features.*
- *The system is not operating under 'real time' constraints, so speed is not necessarily a major factor.*
- *How will the results be presented to the user in a useful way?*

Sample paper and examiner's notes

Question 5 Music information retrieval

(a) Twelve-tone equal temperament is the tuning system typically used in Western classical and pop music.

Assuming the standard reference pitch A4 (MIDI note number 69) equals 440 Hz, calculate the frequency of the following pitches: [6]

i. A5

ii. D4

iii. A♯2

Requires calculating the MIDI note number for each pitch, and applying the following formula: $f(p) = 2^{\frac{p-69}{12}} \cdot 440$.

Pitch	*MIDI note number*	*Frequency (Hz)*
A5	*81*	*880.00*
D4	*62*	*293.66*
A♯2	*46*	*116.54*

(b) Explain the process of classical dynamic time warping (DTW). [6]

Complete answer should include the following points.

- *The objective of DTW is to find an optimal alignment between two time-dependent sequences (of length N and M).*
- *A local cost measure must be defined in order to compute the similarity between pairs of values.*
- *Compute a cost matrix (size $N \times M$) by applying the cost measure to each pair of elements of the sequences.*
- *Find an (N, M)- warping path with the minimal overall cost.*
- *A warping path must satisfy the boundary condition, monotonicity condition, and step size condition. These conditions should be briefly explained in words or mathematical notation.*
- *Describe the algorithm, based on dynamic programming, which recursively computes an accumulated cost matrix.*

(c) How might optical music recognition technology be combined with audio-based music information retrieval technology? Describe an application that could be built using a combination of both OMR and MIR technologies. [7]

OMR technology is able to take scanned images of musical scores, and generate symbolic representations of the musical notation. The quality of the generated data is variable, and errors can be frequent, particularly for low quality images or

Sample paper and examiner's notes

non-standard scores. However, the availability of the symbolic information opens up possibilities for interesting applications.

The answer to describe how the OMR generated symbolic data can be related to audio data (i.e. a music synchronisation task). The answer should also provide an interesting application, of genuine practical use, which combines scanned images of musical scores, and synchronised symbolic and audio music data.

(d) Describe the process of generating cepstrum features from an audio file. [6]

i. Apply a windowing function to convert the signal into frames.

ii. Calculate the discrete Fourier transform of each frame.

iii. Take the absolute values.

iv. Take the logarithm of the magnitudes.

v. Apply inverse discrete Fourier transform.

Additional mark for clarity and extra details.

Bibliography

Abdallah, S. & Plumbley, M. (2006), 'Unsupervised analysis of polyphonic music using sparse coding', *IEEE Transactions on Neural Networks* **17**(1), 179–196.

Abdallah, S., Rhodes, C., Sandler, M. & Casey, M. (2006), 'Using duration models to reduce fragmentation in audio segmentation ', *Machine Learning* **65**, 485–515.

Aucouturier, J.-J., Defreville, B. & Pachet, F. (2006), 'The bag-of-frames approach to audio pattern recognition: A sufficient model for urban soundscapes but not for polyphonic music', *Journal of the Acoustic Society of America* **122**(2), 881–891.

Bent, M. (2010), Isorhythm, *in* 'Grove Music Online, Oxford Music Online', Oxford University Press, Oxford, UK. Accessed: 19 Apr. 2010.
URL: *http://www.oxfordmusiconline.com/subscriber/article/grove/music/13950*

Blackwell, A. & Collins, N. (2005), The programming language as a musical instrument, *in* 'Proceedings of PPIG05 (Psychology of Programming Interest Group'.
URL: *http://www.informatics.sussex.ac.uk/users/nc81/research/proglangasmusicinstr.pdf*

Blackwell, T. M. (2007), Swarming and music, *in* E. Miranda & A. Biles, eds, 'Evolutionary Computer Music', Springer Verlag.

Bonabeau, E., Dorigo, M. & Theraulaz, G. (1999), *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, Oxford.

Byrd, D. (2008), A Similarity Scale for Content-Based Music IR. Available online.
URL: *http://www.informatics.indiana.edu/donbyrd/MusicSimilarityScale.html*

Cambouropoulos, E. (2001), The local boundary detection model (LBDM) and its application in the study of expressive timing, *in* 'Proceedings of the International Computer Music Conference', ICMA, San Francisco, pp. 17–22.
URL: *http://www.ofai.at/cgi-bin/get-tr?paper=oefai-tr-2001-11.pdf*

Casey, M. (2009), Soundspotting: A New Kind of Process?, *in* R. Dean, ed., 'The Oxford Handbook of Computer Music', Oxford University Press.

Casey, M., Veltkamp, R., Goto, M., Leman, M., Rhodes, C. & Slaney, M. (2008), 'Content-Based Music Information Retrieval: Current Directions and Future Challenges', *Proceedings of the IEEE* **96**(4), 668–695.

Cemgil, A. (2004), Bayesian Music Transcription, PhD thesis, Radboud Universiteit Nijmegen.

Chowning, J. M. (1973), 'The synthesis of complex audio spectra by means of frequency modulation', *Journal of the Audio Engineering Society* **21**(7), 526–534.

Collins, N. (2008), 'The analysis of generative music programs', *Organised Sound* **13**(03), 237–248.

Collins, N. (2009), 'Musical form and algorithmic composition', *Contemporary Music Review* **28**(1), 103–114.

Collins, N., McLean, A., Rohrhuber, J. & Ward, A. (2003), 'Live coding in laptop performance', *Organised Sound* **8**(3), 321–330.

Cope, D. (1991), *Computers and Musical Style*, Oxford University Press, Oxford.

Cope, D. (1996), *Experiments in Musical Intelligence*, A-R Editions, Madison, WI.

Cope, D. (2001), *Virtual Music: Computer Synthesis of Musical Style*, MIT Press.

Craft, A. J. D., Wiggins, G. A. & Crawford, T. (2007), How many beans make five? The consensus problem in music-genre classification and a new evaluation method for single-genre categorisation systems, *in* 'Eighth International Symposium on Music Information Retrieval', Vienna, pp. 73–76.

Davies, M. E. P. & Plumley, M. D. (2007), 'Context-dependent beat tracking of musical audio', *IEEE Transactions on Audio, Speech and Language Processing* **15**(3), 1009–1020.

Dennett, D. (1991), *Consciousness explained*, Little, Brown and Co., Boston.

Dixon, S. (2006), Onset Detection Revisited, *in* 'Ninth International Conference on Digital Audio Effects', pp. 133–137.

Donaldson, J. & Lamere, P. (2009), 'Using Visualizations for Music Discovery', Tutorial at Tenth ISMIR. Slides.
URL: *http://www.slideshare.net/plamere/using-visualizations-for-music-discovery*

Fauvel, J., Flood, R. & Wilson, R. (2003), *Music and Mathematics: From Pythagoras to Fractals*, Oxford University Press.

Goto, M. (2006), 'A chorus section detection method for musical audio signals and its application to a music listening station', *IEEE Transaction on Audio, Speech and Language Processing* **14**(5), 1783–1794.

Goto, M. & Goto, T. (2005), Musicream: New music playback interface for streaming, sticking, sorting and recalling musical pieces, *in* 'Sixth International Symposium on Music Information Retrieval', London, pp. 404–411.

Griffiths, P. (1995), *Modern Music and After: Directions Since 1945*, Oxford University Press, Oxford, UK.

Harley, J. (2009), Computational approaches to composition of notated instrumental music: Xenakis and the other pioneers, *in* R. T. Dean, ed., 'The Oxford Handbook of Computer Music', Oxford University Press, Oxford, UK, chapter 5, pp. 109–132.

Huron, D. (2006), *Sweet Anticipation: Music and the Psychology of Expectation*, MIT Press, Cambridge, MA.

Johnson-Laird, P. N. (1988), *The Computer and the Mind: An Introduction to Cognitive Science*, Harvard University Press, Cambridge, MA.

Johnson-Laird, P. N. (1991), Jazz improvisation: A theory at the computational level, *in* P. Howell, R. West & I. Cross, eds, 'Representing Musical Structure', Academic Press, London, pp. 291–325.

Klapuri, A. (1999), Sound onset detection by applying psychoacoustic knowledge, *in* 'IEEE International Conference on Audio, Speech and Signal Processing', pp. 3089–3092.

Krumhansl, C. L. (1990), *Cognitive Foundations of Musical Pitch*, Oxford University Press, Oxford.

Krumhansl, C. L. & Kessler, E. J. (1982), 'Tracing the dynamic changes in perceived tonal organisation in a spatial representation of musical keys', *Psychological Review* **89**(4), 334–368.

Lerdahl, F. (1992), 'Cognitive constraints on compositional systems', *Contemporary Music Review* **6**, 97–121.

Lerdahl, F. & Jackendoff, R. (1983), *A Generative Theory of Tonal Music*, The MIT Press, Cambridge, MA.

Levy, M. & Sandler, M. (2006), Lightweight measures for timbral similarity of musical audio, *in* 'First ACM workshop on Audio and Music Computing Multimedia', Santa Barbara, pp. 27–36.

London, J. (2004), *Hearing in time: psychological aspects of music metre*, Oxford University Press, Oxford, UK.

Magas, M., Casey, M. & Rhodes, C. (2008), mHashup: fast visual music discovery via locality sensitive hashing, *in* 'ACM SIGGRAPH 2008 New Tech Demos'.

Mardirossian, A. & Chew, E. (2008), 'Visualizing music: Tonal progressions and distributions', *Proceedings of the 8th International Conference on Music Information Retrieval* .
URL: *http://www.scientificcommons.org/42491827*

Marr, D. (1982), *Vision*, W. H. Freeman, San Francisco.

McCartney, J. (2002), 'Rethinking the computer music language: Supercollider', *Computer Music Journal* **26**(4), 61–68.

Miranda, E. R. (2002), *Computer Sound Design: Synthesis techniques and programming*, Music Technology Series, second edn, Focal Press, Oxford.

Müller, M. & Clausen, M. (2007), Transposition-Invariant Self-Similarity Matrices, *in* 'Eighth International Symposium on Music Information Retrieval', pp. 47–50.

Narmour, E. (1990), *The Analysis and Cognition of Basic Melodic Structures: The Implication-realisation Model*, University of Chicago Press, Chicago.

Newell, A. & Simon, H. A. (1976), 'Computer science as empirical enquiry: Symbols and search', *Communications of the ACM* **19**(3), 113–126.

Pachet, F. & Cazaly, D. (2000), A Taxonomy of Musical Genres, *in* 'Content-Based Multimedia Information Access Conference'.

Pampalk, E. (2001), Islands of Music: Analysis, Organization and Visualization of Music Archives, PhD thesis, Vienna University of Technology.
URL: *http://www.ofai.at/ elias.pampalk/music/thesis.html*

Pampalk, E., Flexer, A. & Widmer, G. (2005), Improvements of audio-based music similarity and genre classificaton, *in* 'Sixth International Symposium on Music Information Retrieval', London, pp. 628–633.

Parncutt, R. (1994), 'A perceptual model of pulse salience and metrical accent in musical rhythms', *Music Perception* **11**, 409–464.

Pearce, M. T., Meredith, D. & Wiggins, G. A. (2002), 'Motivations and methodologies for automation of the compositional process', *Musicæ Scientiæ* **6**(2), 119–147.
URL: *http://www.titanmusic.com/papers/public/mussci02.pdf*

Pearce, M. T. & Wiggins, G. A. (2006), 'Expectation in melody: The influence of context and learning', *Music Perception* **23**(5), 377–405.
URL: *http://www.doc.gold.ac.uk/ mas01mtp/papers/PearceWigginsMP06.pdf*

Puckette, M. (1988), The patcher, *in* 'Proceedings of the International Computer Music Conference 1988', Köln, Germany, pp. 420–429.

Puckette, M. (2002), 'Max at seventeen', *Computer Music Journal* **26**(4), 31–43.

Pylyshyn, Z. (1989), Computing in cognitive science, *in* M. I. Posner, ed., 'Foundations of Cognitive Science', MIT Press, Cambridge, MA, pp. 51–91.

Resnick, M. (1997), *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press.

Rhodes, C. & Casey, M. (2007), Algorithms for determining and labelling approximate hierarchical self-similarity, *in* 'Eighth International Symposium on Music Information Retrieval', pp. 41–46.

Rhodes, C., Lewis, D. & Müllensiefen, D. (2007), Bayesian Model Selection for Harmonic Labelling, *in* 'Mathematics and Computation in Music', number 37 *in* 'CICS', Berlin: Springer-Verlag.

Rowe, R. (2001), *Machine Musicianship*, The MIT Press, Cambridge, MA, USA.

Schaeffer, P. (1952), *À la recherche d'une musique concrète*, Éditions du Seuil, Paris, FR.

Schellenberg, E. G. (1997), 'Simplifying the implication-realisation model of melodic expectancy', *Music Perception* **14**(3), 295–318.

Scruton, R. (1997), *The Aesthetics of Music*, Oxford University Press.

Steedman, M. (1996), The blues and the abstract truth: Music and mental models, *in* A. Garnham & J. Oakhill, eds, 'Mental Models In Cognitive Science', Erlbaum, Mahwah, NJ, pp. 305–318.
URL: *ftp://ftp.cogsci.ed.ac.uk/pub/steedman/music/batat.ps.gz*

Taube, H. L. (2004), *Notes from the Metalevel: Introduction to Algorithmic Music Composition*, Taylor & Francis Group, London, UK.

Temperley, D. (2001*a*), *The Cognition of Basic Musical Structures*, MIT Press, Cambridge, MA.

Temperley, D. (2001*b*), *The Cognition of Basic Musical Structures*, MIT Press, Cambridge, MA.

Tzanetakis, G. & Cook, P. (2002), 'Musical genre classification of audio signals', *IEEE Transactions on Speech and Audio Processing* pp. 293–302.

Wang, A. (2003), An Industrial Strength Audio Search Algorithm, *in* 'Fourth International Symposium on Music Information Retrieval', Baltimore, pp. 7–13.

Wang, A. (2006), 'The Shazam music recognition service', *Communications of the ACM* **49**(8), 44–48.

Whitman, B. & Rifkin, R. M. (2002), Musical query-by-description as a multiclass learning problem, *in* 'IEEE Workshop on Multimedia Signal Processing', pp. 153–156.

Wiggins, G. A. (2006), 'A preliminary framework for description, analysis and comparison of creative systems', *Journal of Knowledge-Based Systems* **19**(7), 449–458.

Wiggins, G. A., Miranda, E., Smaill, A. & Harris, M. (1993), 'A framework for the evaluation of music representation systems', *Computer Music Journal* **17**(3), 31–42.
URL: *http://www.soi.city.ac.uk/ geraint/papers/CMJ93.pdf*

Wu, X. & Li, M. (2007), A QBSH system based on a three-level melody representation, *in* 'Eighth International Symposium on Music Information Retrieval', Vienna.

Wulfhorst, R. D., Nakayama, L. & Vicari, R. M. (2003), A multiagent approach for musical interactive systems, *in* 'AAMAS 2003: Second International Joint Conference on Autonomous Agents & Multiagent Systems', ACM, pp. 584–591.

Xenakis, I. (1992), *Formalised Music: Thought and Mathematical Composition*, Pendragon Press, Hillsdale, NY, USA.

Comment form

We welcome any comments you may have on the materials which are sent to you as part of your study pack. Such feedback from students helps us in our effort to improve the materials produced for the External System.

If you have any comments about this guide, either general or specific (including corrections, non-availability of essential texts, etc.), please take the time to complete and return this form.

Title of this subject guide: ....

Name ....

Address ....

....

Email ....

Student number ....

For which qualification are you studying? ....

Comments

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

Please continue on additional sheets if necessary.

Date: ....

Please send your comments on this form (or a photocopy of it) to:
Publishing Manager, External System, University of London, Stewart House, 32 Russell Square, London WC1B 5DN, UK.