Computing Pitch Names in Tonal Music: A Comparative Analysis of Pitch Spelling Algorithms

David Meredith

St. Anne's College, University of Oxford

Dissertation submitted for degree of D.Phil.

Computing Pitch Names in Tonal Music: A Comparative Analysis of Pitch Spelling Algorithms

David Meredith
St. Anne's College, University of Oxford
Submitted for degree of D.Phil.

Abstract

A *pitch spelling algorithm* predicts the pitch names (e.g., C♯4, B♭5 etc.) of the notes in a passage of tonal music, when given the onset-time, MIDI note number and possibly the duration and voice of each note. A new algorithm, called *ps13*, was compared with the algorithms of Longuet-Higgins, Cambouropoulos, Temperley and Chew and Chen by running various versions of these algorithms on a 'clean', score-derived test corpus, $\mathcal{C}$, containing 195972 notes, equally divided between eight classical and baroque composers. The standard deviation of the accuracies achieved by each algorithm over the eight composers was used to measure style dependence (SD). The best versions of the algorithms were tested for robustness to temporal deviations by running them on a 'noisy' version of the test corpus, denoted by $\mathcal{C}'$.

A version of *ps13* called PS13s1 was the most accurate of the algorithms tested, achieving note accuracies of 99.44% ($SD = 0.45$) on $\mathcal{C}$ and 99.41% ($SD = 0.50$) on $\mathcal{C}'$. A real-time version of PS13s1 also out-performed the other real-time algorithms tested, achieving note accuracies of 99.19% ($SD = 0.51$) on $\mathcal{C}$ and 99.16% ($SD = 0.53$) on $\mathcal{C}'$. PS13s1 was also as fast and easy to implement as any of the other algorithms.

New, optimised versions of Chew and Chen's algorithm were the least dependent on style over $\mathcal{C}$. The most accurate of these achieved note accuracies of 99.15% ($SD = 0.42$) on $\mathcal{C}$ and 99.12% ($SD = 0.47$) on $\mathcal{C}'$. It was proved that replacing the spiral array in Chew and Chen's algorithm with the line of fifths never changes its output.

A new, optimised version of Cambouropoulos's algorithm made 8% fewer errors over $\mathcal{C}$ than the most accurate of the versions described by Cambouropoulos himself. This algorithm achieved note accuracies of 99.15% ($SD = 0.47$) on $\mathcal{C}$ and 99.07% ($SD = 0.53$) on $\mathcal{C}'$. A new implementation of the most accurate of the versions described by Cambouropoulos achieved note accuracies of 99.07% ($SD = 0.46$) on $\mathcal{C}$ and 99.13% ($SD = 0.39$) on $\mathcal{C}'$, making it the least dependent on style over $\mathcal{C}'$. However, Cambouropoulos's algorithms were among the slowest of those tested.

When Temperley and Sleator's `harmony` and `meter` programs were used for pitch spelling, they were more affected by temporal deviations and tempo changes than any of the other algorithms tested. When enharmonic changes were ignored and the music was at a natural tempo, these programs achieved note accuracies of 99.27% ($SD = 1.30$) on $\mathcal{C}$ and 97.43% ($SD = 1.69$) on $\mathcal{C}'$. A new implementation, called TPRone, of just the first preference rule in Temperley's theory achieved note accuracies of 99.06% ($SD = 0.63$) on $\mathcal{C}$ and 99.16% ($SD = 0.52$) on $\mathcal{C}'$. TPRone's performance was independent of tempo and less dependent on style than that of the `harmony` and `meter` programs.

Of the several versions of Longuet-Higgins's algorithm tested, the best was the original one, implemented in his `music.p` program. This algorithm achieved note accuracies of 98.21% ($SD = 1.79$) on $\mathcal{C}$ and 98.25% ($SD = 1.71$) on $\mathcal{C}'$, but only when the data was processed a voice at a time.

None of the attempts to take voice-leading into account in the algorithms considered in this study resulted in an increase in note accuracy and the most accurate algorithm, PS13s1, ignores voice-leading altogether. The line of fifths is used in most of the algorithms tested, including PS13s1. However, the superior accuracy achieved by PS13s1 suggests that pitch spelling accuracy can be optimised by modelling the local key as a pitch class frequency distribution instead of a point on the line of fifths, and by keeping pitch names close to the local tonic(s) on the line of fifths rather than close on the line of fifths to the pitch names of neighbouring notes.

Acknowledgements

This project could not have been completed without the steadfast support and encouragement of Susanne Meredith, George and Priscilla Meredith, Geraint Wiggins and Ian Cross.

I thank Emilios Cambouropoulos, Elaine Chew and David Temperley for being willing to engage in detailed and lengthy discussions about their work.

I thank the members of the ISMS group at Goldsmiths College, University of London, for many stimulating and entertaining discussions.

Contents

N.B. Some of the material included in this chapter has been published in the followng article published by Taylor & Francis: Meredith, D. (2006). The ps13 pitch spelling algorithm. Journal of New Music Research, 35:2, 121-159. http://dx.doi.org/10.1080/09298210600834961

Chapter 1

Introduction

1.1 Motivation

A *pitch spelling algorithm* attempts to compute the correct pitch names (e.g., C♯4, B♭5 etc.) of the notes in a passage of tonal music, when given only the onset-time, MIDI note number and possibly the duration and voice of each note.

There are good practical and scientific reasons for attempting to develop a reliable pitch spelling algorithm. First, until such an algorithm is devised, it will be impossible to construct a reliable *MIDI-to-notation transcription system*—that is, a system that reliably computes a correctly notated score of a passage when given only a MIDI file of the passage as input. Second, existing audio transcription systems generate not notated scores but MIDI-like, 'piano roll' representations as output (Abdallah and Plumbley, 2004; Davy and Godsill, 2003; Plumbley et al., 2002; Walmsley, 2000). So, if one needs to produce a notated score from a digital audio recording, one needs not only an audio transcription system but also a MIDI-to-notation transcription algorithm (incorporating a pitch spelling algorithm).

Third, knowing the letter-names of the pitch events in a passage is useful in music information retrieval and musical pattern discovery (see, for example, Meredith et al., 2002, pp. 328–330). In particular, two occurrences of a motive on different degrees of a scale might be perceived to be similar even if the corresponding chromatic intervals in the patterns differ. In Figure 1.1, for example, the three patterns, A, B and C, are perceived as being three occurrences of the same motive even though the corresponding chromatic intervals are different in the three patterns.

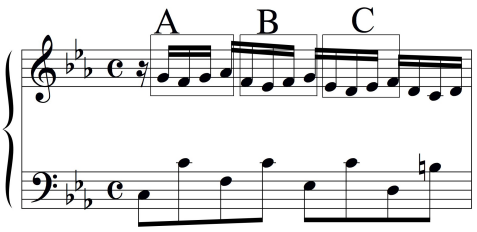


Figure 1.1: Three perceptually similar patterns with different chromatic pitch interval structures (from the first bar of the Prelude in C minor (BWV 871/1) from Book 2 of J. S. Bach's *Das Wohltemperirte Clavier*).

Note that, in this example, one important aspect of the perceived similarity between patterns A, B and C is nicely represented in the notation by the fact that they all have the same scale-step interval structure (i.e., a descending step followed by two ascending steps). In other words, one result of the choice of pitch names for the notes in this passage is that the scale-step interval structures are the same for these three perceptually similar but chromatically different patterns. This illustrates the fact that a correctly notated score is not simply a set of instructions for the performer (cf. tablature). A correct Western staff notation score of a passage of tonal music is a structural description that represents certain important aspects of the way that the passage is intended to be perceived and interpreted. As Longuet-Higgins (1976, p. 646) pointed out, "much can be learned about the structural relationships in any ordinary piece of music from a study of its orthographic representation".

If the pitch names of the notes are encoded, matches such as the ones in Figure 1.1 can be found using fast, exact-matching algorithms such as those described by Knuth et al. (1977), Galil (1979) and Boyer and Moore (1977). However, if the pitches of the notes are represented using just MIDI note numbers, matches such as the ones in Figure 1.1 can only be found using slower and more error-prone approximate-matching algorithms such as those described by Crochemore et al. (2001) and Cambouropoulos et al. (2002). Thus, if a reliable pitch spelling algorithm existed, it could be used, for example, to compute the pitch names of the notes in the huge number of MIDI files freely available on the internet, allowing these files to be searched more effectively by a music information retrieval system.

The development of a reliable pitch spelling algorithm can also serve the scientific purpose of furthering our understanding of music perception and cognition. For example, Temperley's pitch spelling algorithm forms one part of a computational model of music cognition that attempts to explain how "we extract basic kinds of musical information ... from music as we hear it" (Temperley, 2001, p. ix). In Temperley's view, "recognizing spelling distinctions" (i.e., identifying the pitch names of the notes in a piece) is "of direct experiential importance, for pitches, chords, and keys" and "provides useful input in harmonic and key analysis" (Temperley, 2001, p. 122).

Cambouropoulos (2003, p. 412) similarly argues that distinguishing between enharmonic spellings reflects "underlying tonal qualities of pitch and, thus, may facilitate other musical tasks such as harmonic analysis, melodic pattern matching, and motivic analysis". He also claims that "it allows higher level musical knowledge to be represented and manipulated in a more precise and parsimonious manner." Indeed, Cambouropoulos developed his pitch spelling algorithm in order to test two hypotheses about the perception and cognition of tonal music: first, "that a strong link exists between enharmonic pitch spelling and tonal structure"; and second, "that listeners internalize frequencies of occurrence of musical intervals" and use this information "for inferring higher-level tonal information" (Cambouropoulos, 2003, p. 413).

In a similar way, Longuet-Higgins's (1976, p. 646) pitch spelling algorithm was developed as part of a computer program that was "intended to embody, in computational form, a psychological theory of how Western musicians perceive the rhythmic and tonal relationships between the notes of ... melodies".

Those who are against using computer programs to model cognitive processes point out that the fact that a computer program successfully simulates some aspect of human behaviour does

not imply that it accomplishes its task in the same way as the human mind does (see, for example, Searle, 1980). This observation is true but it does not imply that developing computational models is pointless. In fact, this observation is symptomatic of a basic misunderstanding of the nature of a scientific theory. Indeed, one might just as well say that the fact that a particular theory has so far managed to explain all observed examples of the phenomenon that it was developed to explain does not imply that the theory is true. Again, this statement is true but has no bearing on whether or not the theory is a properly testable scientific hypothesis. A computational model for a certain observable aspect of human behaviour has the status of a proper scientific theory if it is a refutable hypothesis (in the sense of Popper, 1983) that the behaviour arises as the result of a mechanism that is correctly described at some level of detail by the computational model.[1] As Temperley (2001, p. 5) states,

> [o]ften, the aim is simply to devise a computational system which can perform a particular process (for example, yielding a certain desired output for a given input); while there is no guarantee that such a program performs the process the same way humans do it, such an approach may at least shed some light on the psychological mechanisms involved.

In the vast majority of cases, those who study and perform Western tonal music agree about how a note should be spelt in a given context. This prompts us to wonder about the nature of the cognitive processes involved when someone who understands Western music notation determines the correct pitch name of a note in a passage of tonal music, while he or she is, for example, transcribing or composing. One way of trying to learn more about these cognitive processes is to attempt to construct a pitch spelling algorithm that

1. assigns the same pitch names to notes as human experts do; and
2. works in a way that is consistent with what we have learned through experimental work about the perception and cognition of tonal music.

Clearly, an algorithm that fails to assign the same pitch names to notes as human experts do is not a plausible model of the human pitch spelling process. Therefore such a model could be rejected without worrying about whether it is consistent with the results of experimental work on tonal music perception and cognition. As Temperley (2001, p. 6) has pointed out,

> the mere fact that a model performs a process successfully certainly does not prove that the process is being performed cognitively in the same way. However, if a model does *not* perform a process successfully, then one knows that the process is *not* performed cognitively in that way. If the model succeeds in its purpose, then one has at least a hypothesis for how the process might be performed cognitively.

Consequently, even if the motivation for developing a pitch spelling algorithm is to construct a psychological theory, it makes sense to set one's first goal to be that of developing an algorithm which reliably assigns the same pitch names to notes as human experts do. Once this has been achieved, one can then go on to consider whether or not the algorithm works in a way that

[1] For an in-depth discussion of these and related issues, see Boden (1988, 1990).

is consistent with what is known about the perception and cognition of tonal music. Even if it is found that the algorithm's mechanism is *not* consistent with the results of psychological experiments, then at least one knows what such an algorithm needs to compute in order to produce the correct output. One can then attempt to design a new algorithm that computes the same output in a way that *is* psychologically plausible.

It is therefore accepted by those in the field that the development of a reliable pitch spelling algorithm may provide us not only with a useful tool, but also with "useful insights into possible cognitive principles and mechanisms ... that may be at work during the pitch transcription task" (Cambouropoulos, 2003, p. 413–414).

1.2 Overview of dissertation

In this study, pitch spelling algorithms proposed by several authors were analysed, evaluated and, in some cases, improved. The algorithms considered were those of Longuet-Higgins (1976, 1987a, 1993), Cambouropoulos (1996, 1998, 2001, 2003), Temperley (2001), Chew and Chen (2003a,b, 2005) and myself (Meredith, 2003, 2005; Meredith and Wiggins, 2005).

In Chapter 2, I analyse and evaluate the pitch spelling algorithm that Longuet-Higgins (1976, 1987a, 1993) incorporated into his `music.p` transcription program. In Chapter 3, I analyse and compare various versions of the pitch spelling algorithm proposed by Cambouropoulos (1996, 1998, 2001, 2003) and develop a new, more accurate version of this algorithm. In Chapter 4, I discuss and evaluate Temperley's (2001) theory of pitch spelling and show how a simple implementation of just part of this theory compares favorably with the much more complicated implementation in Sleator and Temperley's *Melisma* system. In Chapter 5, I analyse and attempt to optimize the real-time pitch spelling algorithm proposed by Chew and Chen (2003a,b, 2005), considering in some detail the way that the performance of this algorithm depends on the values assigned to its various parameters. In Chapter 6, I describe and analyse a number of variants of my *ps13* algorithm and identify a new version of the algorithm which is not only the simplest of the versions tested, but also the most accurate and the most robust to stylistic variation and 'noise' in the data. Finally, in Chapter 7, I compare the best versions of the algorithms tested and run these on noisy data in order to test their robustness. I then draw some general conclusions and suggest some possible directions for further work.

In the remainder of this chapter, I first discuss the methodology used to evaluate the algorithms. I then introduce various preliminary concepts, terminology, notation and functions that will be used without further comment throughout the chapters that follow.

1.3 Evaluating and comparing pitch spelling algorithms

1.3.1 Introduction

The first step in the development of an evaluation procedure is to identify relevant *evaluation criteria*—that is, specific ways in which the performance of one algorithm might be considered interestingly different from that of another. Then appropriate *performance metrics* have to be defined for these evaluation criteria. A performance metric for a particular evaluation criterion

is a way of measuring the performance of an algorithm with respect to that criterion.

In practice, each of these performance metrics is used to measure how well the algorithm performs on some specified test corpus of works that, ideally, should form a large, representative sample of the population of musical works that we intend to use the algorithm to process in the future. I shall therefore also discuss the problem of constructing a test corpus that is suitable for evaluating and comparing pitch spelling algorithms.

1.3.2 Evaluation criteria

The best pitch spelling algorithm to choose for a particular task may depend on the nature of the task.

For example, one might need to implement a pitch spelling algorithm in a music processing system that will be used to process music in many different tonal styles that one does not know beforehand. In this situation, one would probably choose an algorithm that has been shown to make the least number of errors when run on music in a wide variety of different styles. Alternatively, if one has a particularly large quantity of data to process or a close deadline, one might wish to use the fastest algorithm that achieves some minimum level of accuracy.

On the other hand, one might need to process a specific collection of pieces that is known beforehand. In this case, if one's first priority is spelling accuracy, then one would probably want to choose the algorithm that has been shown to make the fewest errors when run on music in the styles represented in the collection of pieces to be processed.

Another possibility is that one has a collection of MIDI files generated from performances on MIDI instruments. In such cases, the onset times and durations of the notes will not, in general, be strictly proportional to the notated onset times and durations as a result of intentional expressive deviations and unintentional errors. In this situation, one would use an algorithm that has been shown to work well on this sort of data. For example, in this situation, one would probably avoid using an algorithm that relies on the duration of each note being close to the notated value, or one that relies heavily on the onsets of notes within a single chord being strictly simultaneous.

If one is primarily interested in modelling music perception and cognition, then one's first priority might be that the algorithm should be a plausible model of the human pitch spelling process. On the other hand, to a computer programmer looking for an algorithm for incorporation into a music processing system, the most attractive algorithm might be the one that is easiest to implement. For a music theorist or educator, the highest priority might be that the theory underlying the algorithm should be elegant, intuitive or possible to state in a parsimonious way.

Finally, if one wishes to transcribe a piece in real-time as it is being played, then one would need to use a real-time pitch-spelling algorithm that can determine the pitch name of a note without requiring information about any notes that follow it in the music.

There are thus a number of different criteria that could be used to evaluate the performance of a pitch spelling algorithm, including, for example,

1. *spelling accuracy*—how well the algorithm predicts the pitch names of the notes;

2. *computational complexity*—how much time and memory the algorithm requires to process

the data and whether or not the algorithm works in real-time;

3. *cognitive plausibility*—how plausible it is that the human pitch spelling process works in a way that is correctly described by the algorithm;

4. *parsimony*—how simply or briefly the algorithm may be expressed in words, pseudocode or mathematics;

5. *ease of implementation*—how easy the algorithm is to implement as a working computer program;

6. *style dependence*—how much the performance of the algorithm is affected by the style of the music being processed; and

7. *robustness to noise in the data*—how little the performance of the algorithm is affected by 'noise' in the input data such as missing notes, temporal deviations, etc.

Table 1.1 identifies the criteria that have been used in those publications to date in which pitch spelling algorithms have been evaluated or compared. In this table, "Qual" indicates that a criterion is given some qualitative consideration and "Quant" indicates that the performance of one or more algorithms with respect to a criterion has been quantitatively measured in some way. An entry in parentheses indicates that the criterion is given such slight consideration that it barely qualifies for inclusion in the table.

As is clear from Table 1.1, the most commonly used criterion for comparing and evaluating pitch spelling algorithms is spelling accuracy, and, in most publications, this has been evaluated quantitatively in some way. For most applications, spelling accuracy is the most important criterion.

Cambouropoulos (1996, 1998, 2001, 2003), Temperley (2001) and Chew and Chen (2005) give some consideration to time complexity, however only Cambouropoulos (1996, 1998) considers this in any depth.

With regards to cognitive plausibility, only Temperley (2001) considers this criterion in any depth. However, Longuet-Higgins (1976, 1987a, 1993), Cambouropoulos (1996, 1998, 2001, 2003) and Chew and Chen (2003b, 2005) also consider it briefly.

No publication so far formally recognizes the importance of the extent to which the principles underlying a pitch spelling algorithm may be expressed parsimoniously. However, all the authors have made an effort to express their algorithms as parsimoniously as possible, either in the form of a set of rules or principles (Cambouropoulos, 1996, 1998, 2001, 2003; Longuet-Higgins, 1976, 1987a, 1993; Stoddard et al., 2004; Temperley, 2001) or as a short sequence of algorithmic steps (Chew and Chen, 2003a,b, 2005; Meredith, 2003, 2005).

Cambouropoulos (2001, 2003) and Meredith (2003, 2005) both briefly discuss the ease with which certain algorithms may be implemented.

A number of authors have considered the degree to which some algorithm's performance depends on the style of music being processed (Cambouropoulos, 1996, 1998, 2003; Chew and Chen, 2003b, 2005; Meredith, 2005; Temperley, 2001). However, only Meredith (2005) and Meredith and Wiggins (2005) have made a serious attempt to measure this quantitatively.

Publication	Spelling accuracy	Computational complexity	Cognitive plausibility	Parsimony	Ease of implementation	Robustness to style	Robustness to noise
Longuet-Higgins (1976, 1987a, 1993)	(Qual)		(Qual)				
Cambouropoulos (1996, 1998)	Qual	Quant	(Qual)			(Qual)	
Cambouropoulos (2001)	Quant	(Quant)	(Qual)		(Qual)		
Cambouropoulos (2003)	Quant	(Quant)	(Qual)		Qual	Quant	
Temperley (2001)	Quant	(Qual)	Qual			(Qual)	
Chew and Chen (2003a)	Quant						
Chew and Chen (2003b)	Quant		(Qual)			(Quant)	
Chew and Chen (2005)	Quant	Qual	(Qual)			(Quant)	
Meredith (2003)	Quant				Qual		
Meredith (2005)	Quant				Qual	Quant	
Meredith and Wiggins (2005)	Quant				(Qual)	Quant	
Stoddard et al. (2004)	Quant						

Table 1.1: Summary of evaluation criteria used to date in publications. "Qual" indicates that a criterion is given some qualitative consideration and "Quant" indicates that the performance of one or more algorithms with respect to a criterion has been quantitatively measured in some way. An entry in parentheses indicates that the criterion is given such slight consideration that it barely qualifies for inclusion in the table.

Finally, although a number of the algorithms proposed in the literature are designed to be robust to noise of the type typical in data automatically derived from performances or audio data, none of the authors attempt to evaluate just how robust the algorithms are to such noise.

In this dissertation, I shall be primarily concerned with evaluating and comparing algorithms in terms of those criteria that can readily be measured quantitatively, namely,

1. spelling accuracy,
2. computational complexity,
3. style dependence and
4. robustness to noise in the data.

Note that, in this study, the extent to which the performance of an algorithm depends on the *composer* of a work will be used as an indicator of style dependence (see section 1.3.6). However, the composer of a work is only one possible indicator of its "style". Other properties of a work that could have been used instead of (or in combination with) its composer as indicators of its style include its genre, form or instrumentation, where and when it was composed, its predominant rhythm, tempo, tonality or modality, and so on.

Before discussing performance metrics for these evaluation criteria, I shall first explore the problem of selecting an appropriate test corpus.

1.3.3 Test corpora

1.3.3.1 Introduction

In the vast majority of cases, those who study and perform Western tonal music agree about how a note should be spelt in a given tonal context. Correspondingly, the vast majority of notes in authoritative published editions of scores of common practice tonal works are generally agreed to be spelt correctly by those who understand Western staff notation.

Therefore, the spelling accuracy of a pitch spelling algorithm can be evaluated by running it on tonal works and comparing the pitch names it predicts with those of the corresponding notes in authoritative published editions of scores of the works. In other words, such authoritative scores can provide us with a 'ground truth' that we can compare with the output of a pitch spelling algorithm.

However, this can only be done accurately and quickly if one has access to accurate encodings of these authoritative scores in the form of computer files that can be compared automatically with the pitch spelling algorithm's output. MIDI files cannot be used for this purpose as the correct pitch name of each note is not typically encoded in such files, which means that the pitch names computed by the algorithm cannot be compared automatically with the correct pitch names in the scores.

Currently there exist few publicly available, high quality collections of encodings of authoritative editions of musical scores (one example is the MuseData collection available online at `http://www.musedata.org`). However, if real progress is to be made in the development and testing of systems for music analysis, retrieval and transcription, it is necessary for much larger

and more varied corpora of encoded scores to be made publicly available (or at least available for research purposes). Moreover, it is imperative that these corpora should be highly accurate. Unfortunately, building such corpora is currently an extremely time-consuming and error-prone process despite the existence of, for example, optical music recognition (OMR) systems.[2] There is therefore an urgent need for tools and techniques that will allow notated scores to be encoded more quickly and more accurately in a format that can be used for evaluating music-processing systems.

Apart from being a collection of works for which high quality score encodings exist, a test corpus for evaluating and comparing pitch spelling algorithms should be chosen so that it is a properly representative sample of the population of works on which one intends to run the algorithms. If this condition is not satisfied, it will not be possible to use the results obtained when the algorithms are run on the test corpus to predict with confidence how well they will perform on the population from which the corpus is a sample.

1.3.3.2 Test corpora used in previous publications

Table 1.2 summarises the test corpora that have been used in previous publications for evaluating and comparing pitch spelling algorithms. As can be seen from this table, Longuet-Higgins (1976, 1987a, 1993) used a minuscule corpus that is clearly not representative of any interesting wider population of works. His results therefore tell us next to nothing about how well we can expect his algorithm to perform on other works.

The collection of melodic fragments used by Cambouropoulos (1996, 1998) was larger than that used by Longuet-Higgins (1976, 1987a, 1993) but, nonetheless, still very small—almost certainly fewer than 1000 notes (see Table 1.2). Moreover, this corpus consisted of short monophonic extracts from works rather than complete polyphonic movements. It is hard to see what interesting wider population of works this corpus could reasonably be considered to represent. Therefore Cambouropoulos's (1996, 1998) evaluation does not allow us to predict with any confidence how well his algorithm will perform in general on some wider population.

The test corpus used by Cambouropoulos (2001) consisted of eight complete piano sonatas by Mozart (K 279–83 and K 331–3) and thus contained 24 substantial movements from polyphonic works (see Table 1.2). This corpus therefore contained a much larger number of notes than that used by Cambouropoulos (1996, 1998). However, it is not clear what population of works this test corpus was intended to represent. The eight sonatas are closely related in terms of genre, instrumentation, date and location of composition—K 279–283 were composed in Salzburg in 1774 and K 331–333 were composed in Paris in 1778. They may thus be too closely clustered even to be considered representative of Mozart's piano music, let alone some wider population such as Mozart's works or works in the classical style. Nevertheless, this corpus could reasonably be considered representative of Mozart's piano sonatas.

The test corpus used by Cambouropoulos (2003) consisted of the eight piano sonatas used by Cambouropoulos (2001) together with two more of Mozart's piano sonatas (K 284 and K 330) and three of Chopin's waltzes (Op. 64, Nos. 1–3). Both the ten sonatas and the three waltzes form sets of works that are very closely related in terms of genre, instrumentation, composer,

[2] For an overview, see David Bainbridge's web page at `http://www.cs.waikato.ac.nz/~davidb/omr/`.

Publication	*Description of corpus*	*Number of notes*	*Number of independent movements or passages*
Longuet-Higgins (1976, 1987a, 1993)	"Shave and haircut, two bits"; two extracts from *cor anglais* solo from *Prelude* to Act III of Wagner's *Tristan und Isolde*.	73	3
Cambouropoulos (1996, 1998)	Themes from Fugues in Book 1 of J. S. Bach's *Das Wohltemperirte Clavier* (BWV 846–869); theme from J. S. Bach's *Musikalisches Opfer* (BWV 1079); opening theme of Chopin's *Ballade*, Op. 23; excerpt from *cor anglais* solo from *Prelude* to Act III of Wagner's *Tristan und Isolde*.	?	27
Cambouropoulos (2001)	Eight complete piano sonatas by Mozart (K 279–283 and K 331–333).	40058	24
Cambouropoulos (2003)	Ten complete piano sonatas by Mozart (K 279–284 and K 330–333); three Chopin Waltzes (Op. 64, Nos. 1–3).	59294 (Mozart: 54418; Chopin: 4876)	33 (Mozart: 30; Chopin: 3)
Temperley (2001)	Encodings of excerpts of 8 or more bars in length from theory workbook by Kostka and Payne (1995), including passages from works by Bach, Beethoven, Brahms, Campbell, Chopin, Grieg, Haydn, Mozart, Schubert, Schumann and Tchaikovsky.	8747	46
Chew and Chen (2003a,b)	Third movement of Beethoven's Piano Sonata in G major, Op. 79; first movement of Beethoven's Piano Sonata in E major, Op. 109.	2891 (Op. 79, Mvt. 3: 1375; Op. 109, Mvt. 1: 1516)	2
Chew and Chen (2005)	Third movement of Beethoven's Piano Sonata in G major, Op. 79; first movement of Beethoven's Piano Sonata in E major, Op. 109; You-Di Huang's *Song of Ali-Shan* for piano and voice or violin.	4462 (Op. 79, Mvt. 3: 1375; Op. 109, Mvt. 1: 1516; *Ali-Shan*: 1571)	3
Meredith (2003)	Book 1 of J. S. Bach's *Das Wohltemperirte Clavier* (BWV 846–869).	41544	48
Meredith (2005)	Movements from various works by Corelli, Vivaldi, Telemann, Bach, Handel, Marcello, Haydn, Mozart and Beethoven.	1729886 (Corelli: 31390; Vivaldi: 223678; Telemann: 89542; Bach: 627083; Handel: 449793; Marcello: 2962; Haydn: 84682; Mozart: 172097; Beethoven: 48659)	1655 (Corelli: 88; Vivaldi: 162; Telemann: 157; Bach: 644; Handel: 501; Marcello: 3; Haydn: 34; Mozart: 56; Beethoven: 10)
Stoddard et al. (2004)	Selection of movements from works by Bach, Beethoven, Haydn, Mozart and Schubert.	22593	12 (Bach: 1; Beethoven: 2; Haydn: 2; Mozart: 6; Schubert: 1)

Table 1.2: Summary of test corpora used in past publications.

style and date of composition. Therefore neither of these two subsets of the test corpus could be considered properly representative of significantly wider populations of works. At most, the Mozart subset of the corpus could be considered reasonably representative of Mozart's piano sonatas and rather less representative of his piano music in general. The Chopin subset of the corpus could be considered weakly representative of Chopin's Waltzes. It is hard to see what wider population the complete corpus could represent. Therefore, again, one cannot generalize with any confidence from the results obtained by Cambouropoulos (2003). Also, the Chopin subset of the corpus may be too small and too different in size from the Mozart subset for the results obtained by an algorithm on the two corpora to be comparable. This would mean that these two corpora are not appropriate for testing the degree to which the performance of an algorithm is affected by the difference in style between Mozart's and Chopin's music.

Temperley's (2001, p. 43) test corpus consisted of 46 excerpts from Kostka and Payne's (1995) theory workbook. This corpus consisted of excerpts of eight or more bars in length from works by a variety of 18th and 19th century composers (see Table 1.2). Nineteen of the excerpts were for solo piano and the rest were mostly from "chamber pieces for small ensembles". Presumably, this corpus was intended to be representative of Western tonal music from Bach to Tchaikovsky. However, it seems very unlikely that such a small corpus of such small musical fragments could be truly representative of such a large and varied population. Also, the fact that each excerpt in this corpus is an example from a theory text book suggests that each of the excerpts was chosen for inclusion in this text book because it illustrated some specific theoretical point. This implies that the excerpts themselves may be special or non-typical in some way which casts further doubt upon the validity of generalizing from the results obtained on this corpus.

The test corpus used by Chew and Chen (2003a,b) consisted of just two movements from Beethoven's piano sonatas (see Table 1.2). This corpus is clearly far too small to be representative of any wider population of works. Therefore, no generalizations can justifiably be made from the results obtained by Chew and Chen (2003a,b). Chew and Chen's (2005) test corpus consisted of the same two Beethoven piano sonata movements together with the *Song of Ali-Shan*, a set of variations on a Taiwanese folksong by You-Di Huang composed in 1967. The *Song of Ali-Shan* contains just 1571 notes so the complete test corpus contained only 4462 notes in total. Again, this test corpus is far too small to be representative of any wider population of works. Therefore one cannot confidently generalize from the results obtained.

Meredith (2003) compared four algorithms by running them on a test corpus consisting of all the works in the first book of J. S. Bach's *Das Wohltemperirte Clavier*[3] (BWV 846–869). These works are closely related in terms of genre, texture, style, composer, instrumentation and date and location of composition. Therefore this corpus cannot be considered properly representative of any wider population of works other than, perhaps, J. S. Bach's keyboard works. Again, therefore, one cannot confidently generalize from the results obtained by Meredith (2003).

Meredith (2005) compared the same four algorithms considered by Meredith (2003) by running them on a much larger test corpus containing 1729886 notes and consisting of 1655 movements from works by 9 baroque and classical composers ranging from Corelli to Beethoven (see Table 1.2). This test corpus is over 30 times larger than Cambouropoulos's (2003), which is the

[3]This is Bach's original spelling for the title of this collection.

Algorithm	*Meredith (2003)*	*Meredith (2005)*
Cambouropoulos	93.74%	98.71%
Temperley	99.71%	97.67%
ps13 (Meredith)	99.81%	99.33%
Longuet-Higgins	99.36%	97.65%

Table 1.3: Summary of spelling accuracy results obtained by Meredith (2003, 2005). Each entry gives the percentage of notes in the corpus spelt correctly.

next largest test corpus used to date. Over 80% of the music in Meredith's (2005) test corpus is baroque and the rest is classical, so this corpus could only be considered representative of Western tonal music composed between about 1675 and 1825. Also, the corpus is rather unevenly distributed between the 9 composers, ranging from 2962 notes from works by B. Marcello to 627083 notes from works by J. S. Bach. Therefore the corpus as a whole is not an ideally representative sample of all the works composed by all the composers represented in the corpus. However, apart from the music by Marcello, each of the nine subsets of the corpus that contains the works by a particular composer can reasonably be considered representative of an interesting wider population of works. For example, the Bach subset of the corpus contains vocal, choral, orchestral and chamber works that span his career. So this subset of the corpus can probably be considered a reasonably representative sample of Bach's works. Similarly, the Corelli subset contains a large number of movements from his trio sonatas and can thus be considered a fairly representative sample of his work in this genre.

Table 1.3 summarises the results obtained by Meredith (2003, 2005). This table shows the percentage of notes spelt correctly by four different algorithms on two different test corpora. As noted by Chew and Chen (2005), the results obtained by Meredith (2003) are quite different from those obtained when the same algorithms were run on the much larger and more varied corpus used by Meredith (2005). For example, Cambouropoulos's algorithm performed worst on the test corpus used by Meredith (2003) but was second best on the larger corpus used by Meredith (2005). These differences in the results suggest that the corpora used by Meredith (2003) and Meredith (2005) are representative of different populations of works. A test for statistical significance (e.g., a t test) could be used to estimate the probability of this being the case. As already mentioned, the corpus used by Meredith (2003) consists of works in a single genre by a single composer, whereas the corpus used by Meredith (2005) contains a much larger number of works in various genres by a number of different composers. It would therefore not be surprising if the two corpora represented different wider populations of works. In particular, because the corpus used by Meredith (2005) is much larger and more varied than that used by Meredith (2003) and, indeed, contains the corpus used by Meredith (2003) as a subset, we may reasonably assume that the corpus used by Meredith (2005) is representative of a larger and more varied population of works that is a superset of that represented by the corpus used by Meredith (2003). However, the fact that the works are rather unevenly distributed between the composers in the corpus used by Meredith (2005) may mean that this corpus is not properly representative of an easily definable large population of works. One could therefore not reliably

assume that the results obtained by Meredith (2005) provide a better estimate than the results obtained by Meredith (2003) of the accuracies that would be achieved by the algorithms over, say, all baroque and classical works.

Stoddard et al. (2004) tested their classifier on a test corpus consisting of 12 movements from various chamber works by Bach, Beethoven, Haydn, Mozart and Schubert. Again, it is hard to see what wider population of works this test corpus was intended to represent. Such a small corpus could hardly be considered representative of all chamber works by the composers of the works in the corpus. For example, the corpus only contains 1 movement from a work by Bach. Therefore, we cannot use the results obtained by Stoddard et al. (2004) to predict with any confidence how well their classifier would work in general on some wider population of works.

1.3.3.3 Test corpus used in this study

In this study, the algorithms considered were evaluated and compared by running them on a test corpus containing 195972 notes and consisting of 216 movements from works by 8 baroque and classical composers (Corelli, Vivaldi, Telemann, J. S. Bach, Handel, Haydn, Mozart and Beethoven). Table 1.4 gives a complete listing of the music in this corpus.

This corpus was derived by automatic conversion from the MuseData collection of encoded scores (Hewlett, 1997).[4] In the MuseData collection, each movement of a work is typically encoded in a set of files, each voice in the movement being encoded in a separate file. Each set of files encoding a single movement in the MuseData collection was automatically converted into a single file in what I call *OPNDV format*. Each OPNDV file contains a single Lisp list representing an *OPNDV dataset* (see Figure 1.4). Each OPNDV dataset is a set of *OPNDV datapoints* and each OPNDV datapoint is a 4-tuple, $\langle t, n, d, v\rangle$, giving the onset time, t, the pitch name, n, the duration, d, and the voice, v, of a single note (or sequence of tied notes) in the score (see Figure 1.3). "OPNDV" simply stands for "Onset, Pitch Name, Duration, Voice". Because the onset time and duration of each note are provided, it is not necessary to encode rests explicitly in an OPNDV dataset. Note also that an OPNDV dataset is an *un*ordered set, so the order in which the datapoints occur in the OPNDV file is arbitrary. The onset time and duration of each note are expressed as integer multiples of the greatest common divisor of all the notated onset times and note durations in the score. The voice to which a given note belongs is encoded as an integer greater than 0. If a movement contains N voices, then the voices are numbered in some arbitrary order from 1 to N. The allocation of notes to voices in each OPNDV file in the corpus used here was the same as that in the Musedata files from which the OPNDV file was derived. The original Musedata files were encoded manually from particular editions of scores of the works. Details of the encoder and source edition for each work are given in the Musedata files. In those cases where the parts or voices are not unambiguously indicated in the score (e.g., in some music for keyboard), it seems that the allocation of notes to voices was left to the discretion of the encoder.[5] As the criteria for allocating notes to voices in such cases does not seem to have been formally specified anywhere, it is possible that this task was not done consistently across all the works in the corpus. Nevertheless, I have assumed that the allocation

[4] Available online at `http://www.musedata.org`.

[5] See `http://www.musedata.org/formats/musedata`.

Composer	*Work title*	*Work no.*	*Mvt.*	*Year*	*File name*	*No. notes*	*No. notes (mvts) for composer*
J. S. Bach	Church Cantata (Es ist das Heil uns kommen her)	BWV 9	5	fp 1732–5	bachbgcant000905m	1663	24505 (26)
	Church Cantata (O Ewigkeit, du Donnerwort)	BWV 20	3	fp 1724	bachbgcant002003m	1200	
	Church Cantata (Am Abend aber desselbigen Sabbats)	BWV 42	6	fp 1725	bachbgcant004206m	2354	
	Church Cantata (Sie werden euch in den Bann tun)	BWV 44	7	fp 1724	bachbgcant004407m	199	
	Church Cantata (Ich geh und suche mit Verlangen)	BWV 49	2	fp 1726	bachbgcant004902m	1979	
	Church Cantata (Das neugeborne Kindelein)	BWV 122	1	fp 1724	bachbgcant012201m	1886	
	Church Cantata (Erforsche mich, Gott, und erfahre)	BWV 136	6	fp 1723	bachbgcant013606m	274	
	Church Cantata (Schau, lieber Gott, wie meine Feind)	BWV 153	3	fp 1724	bachbgcant015303m	307	
	Church Cantata (Schau, lieber Gott, wie meine Feind)	BWV 153	8	fp 1724	bachbgcant015308m	1487	
	Church Cantata (O heilges Geist- und Wasserbad)	BWV 165	3	fp 1715	bachbgcant016503m	477	
	Church Cantata (Gott, wie dein Name, so ist auch dein Ruhm)	BWV 171	4	fp ?1729	bachbgcant017104m	1973	
	Chorale (Christ ist erstanden)	BWV 276	1	pb 1784–7	bachbgchoral027601m	562	
	Chorale (Ein feste Burg ist unser Gott)	BWV 302	1	pb 1784–7	bachbgchoral030201m	230	
	Chorale (Für deinen Thron tret ich hiermit)	BWV 327	1	pb 1784–7	bachbgchoral032701m	146	
	Chorale (Jesu, der du meine Seele)	BWV 352	1	pb 1784–7	bachbgchoral035201m	238	
	Chorale (Mach's mit mir, Gott, nach deiner Güt)	BWV 377	1	pb 1784–7	bachbgchoral037701m	153	
	Chorale (O Mensch, bewein dein' Sünde groß)	BWV 402	1	pb 1784–7	bachbgchoral040201m	349	
	Chorale (Wenn ich in Angst und Not)	BWV 427	1	pb 1784–7	bachbgchoral042701m	214	
	Prelude and Fugue No. 7 in E♭ major from Book 1 of *Das Wohltemperirte Clavier*	BWV 852	Fugue	co 1722	bachbgkeybdwtc-i085202m	886	
	Prelude and Fugue No. 19 in A major from Book 1 of *Das Wohltemperirte Clavier*	BWV 864	Fugue	co 1722	bachbgkeybdwtc-i086402m	1172	
	Prelude and Fugue No. 20 in A minor from Book 1 of *Das Wohltemperirte Clavier*	BWV 865	Prelude	co 1722	bachbgkeybdwtc-i086501m	608	
	Prelude and Fugue No. 9 in E minor from Book 2 of *Das Wohltemperirte Clavier*	BWV 878	Prelude	co 1738–1742	bachbgkeybdwtc-ii087801m	904	
	Prelude and Fugue No. 21 in B♭ major from Book 2 of *Das Wohltemperirte Clavier*	BWV 890	Fugue	co 1738–1742	bachbgkeybdwtc-ii089002m	958	
	Concerto for two violins in D minor	BWV 1043	2	co 1717–1723	bachbgorch104302m	2447	
	Brandenburg Concerto No. 6 in B♭ major	BWV 1051	2	co ?1713–1721	bachbgorch105102m	1271	
	Two part Invention in G minor	BWV 782	1	co 1723	bachrasmusinvent078201m	568	
Beethoven	Symphony No. 1 in C major	Op. 21	2	pb 1801	beetbhsymno102m	4162	24493 (5)
	Symphony No. 1 in C major	Op. 21	3	pb 1801	beetbhsymno103m	1789	
	Symphony No. 1 in C major	Op. 21	4	pb 1801	beetbhsymno104m	6432	
	Symphony No. 3 ('Eroica') in E♭ major	Op. 55	2	co 1803–4	beetbhsymno302m	7999	
	Symphony No. 5 in C minor	Op. 67	3	co 1807–8	beetbhsymno503m	4111	
Corelli	Sonata da chiesa a tre in F major	Op. 1, No. 1	1	pb 1681	corellchrytrioop1n0101m	206	24493 (71)
	Sonata da chiesa a tre in F major	Op. 1, No. 1	2	pb 1681	corellchrytrioop1n0102m	633	
	Sonata da chiesa a tre in F major	Op. 1, No. 1	3	pb 1681	corellchrytrioop1n0103m	211	
	Sonata da chiesa a tre in F major	Op. 1, No. 1	4	pb 1681	corellchrytrioop1n0104m	687	
	Sonata da chiesa a tre in E minor	Op. 1, No. 2	1	pb 1681	corellchrytrioop1n0201m	241	
	Sonata da chiesa a tre in E minor	Op. 1, No. 2	2	pb 1681	corellchrytrioop1n0202m	394	
	Sonata da chiesa a tre in E minor	Op. 1, No. 2	3	pb 1681	corellchrytrioop1n0203m	154	
	Sonata da chiesa a tre in E minor	Op. 1, No. 2	4	pb 1681	corellchrytrioop1n0204m	659	
	Sonata da chiesa a tre in A major	Op. 1, No. 3	2	pb 1681	corellchrytrioop1n0302m	947	
	Sonata da chiesa a tre in A major	Op. 1, No. 3	3	pb 1681	corellchrytrioop1n0303m	181	
	Sonata da chiesa a tre in A major	Op. 1, No. 3	4	pb 1681	corellchrytrioop1n0304m	744	
	Sonata da chiesa a tre in A minor	Op. 1, No. 4	1	pb 1681	corellchrytrioop1n0401m	369	
	Sonata da chiesa a tre in A minor	Op. 1, No. 4	2	pb 1681	corellchrytrioop1n0402m	152	
	Sonata da chiesa a tre in A minor	Op. 1, No. 4	3	pb 1681	corellchrytrioop1n0403m	537	
	Sonata da chiesa a tre in A minor	Op. 1, No. 4	4	pb 1681	corellchrytrioop1n0404m	421	
	Sonata da chiesa a tre in B♭ major	Op. 1, No. 5	1	pb 1681	corellchrytrioop1n0501m	258	
	Sonata da chiesa a tre in B♭ major	Op. 1, No. 5	2	pb 1681	corellchrytrioop1n0502m	569	

continued on next page

continued from previous page

Composer	*Work title*	*Work no.*	*Mvt.*	*Year*	*File name*	*No. notes*	*No. notes (mvts) for composer*
Corelli (*cont.*)	Sonata da chiesa a tre in B♭ major	Op. 1, No. 5	4	pb 1681	corellchrytrioop1n0504m	489	
	Sonata da chiesa a tre in B minor	Op. 1, No. 6	1	pb 1681	corellchrytrioop1n0601m	138	
	Sonata da chiesa a tre in B minor	Op. 1, No. 6	2	pb 1681	corellchrytrioop1n0602m	669	
	Sonata da chiesa a tre in B minor	Op. 1, No. 6	3	pb 1681	corellchrytrioop1n0603m	257	
	Sonata da chiesa a tre in B minor	Op. 1, No. 6	4	pb 1681	corellchrytrioop1n0604m	492	
	Sonata da chiesa a tre in C major	Op. 1, No. 7	1	pb 1681	corellchrytrioop1n0701m	780	
	Sonata da chiesa a tre in C major	Op. 1, No. 7	2	pb 1681	corellchrytrioop1n0702m	134	
	Sonata da chiesa a tre in C major	Op. 1, No. 7	3	pb 1681	corellchrytrioop1n0703m	657	
	Sonata da chiesa a tre in C minor	Op. 1, No. 8	1	pb 1681	corellchrytrioop1n0801m	143	
	Sonata da chiesa a tre in C minor	Op. 1, No. 8	2	pb 1681	corellchrytrioop1n0802m	360	
	Sonata da chiesa a tre in C minor	Op. 1, No. 8	3	pb 1681	corellchrytrioop1n0803m	344	
	Sonata da chiesa a tre in C minor	Op. 1, No. 8	4	pb 1681	corellchrytrioop1n0804m	250	
	Sonata da chiesa a tre in G major	Op. 1, No. 9	2	pb 1681	corellchrytrioop1n0902m	559	
	Sonata da chiesa a tre in G major	Op. 1, No. 9	3	pb 1681	corellchrytrioop1n0903m	245	
	Sonata da chiesa a tre in G minor	Op. 1, No. 10	1	pb 1681	corellchrytrioop1n1001m	103	
	Sonata da chiesa a tre in G minor	Op. 1, No. 10	3	pb 1681	corellchrytrioop1n1003m	406	
	Sonata da chiesa a tre in G minor	Op. 1, No. 10	4	pb 1681	corellchrytrioop1n1004m	142	
	Sonata da chiesa a tre in D minor	Op. 1, No. 11	1	pb 1681	corellchrytrioop1n1101m	162	
	Sonata da chiesa a tre in D minor	Op. 1, No. 11	3	pb 1681	corellchrytrioop1n1103m	210	
	Sonata da chiesa a tre in D major	Op. 1, No. 12	1	pb 1681	corellchrytrioop1n1201m	275	
	Sonata da chiesa a tre in D major	Op. 1, No. 12	3	pb 1681	corellchrytrioop1n1203m	125	
	Sonata da chiesa a tre in D major	Op. 1, No. 12	4	pb 1681	corellchrytrioop1n1204m	975	
	Sonata da camera a tre in D major	Op. 2, No. 1	1	pb 1685	corellchrytrioop2n0101m	190	
	Sonata da camera a tre in D major	Op. 2, No. 1	2	pb 1685	corellchrytrioop2n0102m	314	
	Sonata da camera a tre in D major	Op. 2, No. 1	3	pb 1685	corellchrytrioop2n0103m	217	
	Sonata da camera a tre in D major	Op. 2, No. 1	4	pb 1685	corellchrytrioop2n0104m	103	
	Sonata da camera a tre in D minor	Op. 2, No. 2	1	pb 1685	corellchrytrioop2n0201m	424	
	Sonata da camera a tre in D minor	Op. 2, No. 2	2	pb 1685	corellchrytrioop2n0202m	348	
	Sonata da camera a tre in C major	Op. 2, No. 3	1	pb 1685	corellchrytrioop2n0301m	193	
	Sonata da camera a tre in C major	Op. 2, No. 3	2	pb 1685	corellchrytrioop2n0302m	437	
	Sonata da camera a tre in C major	Op. 2, No. 3	3	pb 1685	corellchrytrioop2n0303m	195	
	Sonata da camera a tre in E minor	Op. 2, No. 4	1	pb 1685	corellchrytrioop2n0401m	261	
	Sonata da camera a tre in E minor	Op. 2, No. 4	2	pb 1685	corellchrytrioop2n0402m	295	
	Sonata da camera a tre in E minor	Op. 2, No. 4	4	pb 1685	corellchrytrioop2n0404m	456	
	Sonata da camera a tre in B♭ major	Op. 2, No. 5	1	pb 1685	corellchrytrioop2n0501m	163	
	Sonata da camera a tre in B♭ major	Op. 2, No. 5	3	pb 1685	corellchrytrioop2n0503m	118	
	Sonata da camera a tre in B♭ major	Op. 2, No. 5	4	pb 1685	corellchrytrioop2n0504m	338	
	Sonata da camera a tre in G minor	Op. 2, No. 6	1	pb 1685	corellchrytrioop2n0601m	357	
	Sonata da camera a tre in G minor	Op. 2, No. 6	2	pb 1685	corellchrytrioop2n0602m	211	
	Sonata da camera a tre in G minor	Op. 2, No. 6	3	pb 1685	corellchrytrioop2n0603m	547	
	Sonata da camera a tre in F major	Op. 2, No. 7	1	pb 1685	corellchrytrioop2n0701m	261	
	Sonata da camera a tre in F major	Op. 2, No. 7	2	pb 1685	corellchrytrioop2n0702m	292	
	Sonata da camera a tre in F major	Op. 2, No. 7	3	pb 1685	corellchrytrioop2n0703m	252	
	Sonata da camera a tre in B minor	Op. 2, No. 8	1	pb 1685	corellchrytrioop2n0801m	194	
	Sonata da camera a tre in B minor	Op. 2, No. 8	2	pb 1685	corellchrytrioop2n0802m	445	
	Sonata da camera a tre in B minor	Op. 2, No. 8	3	pb 1685	corellchrytrioop2n0803m	188	
	Sonata da camera a tre in B minor	Op. 2, No. 8	4	pb 1685	corellchrytrioop2n0804m	276	

continued on next page

continued from previous page

Composer	*Work title*	*Work no.*	*Mvt.*	*Year*	*File name*	*No. notes*	*No. notes (mvts) for composer*
Corelli (*cont.*)	Sonata da camera a tre in F♯ minor	Op. 2, No. 9	1	pb 1685	corellchrytrioop2n0901m	384	
	Sonata da camera a tre in F♯ minor	Op. 2, No. 9	3	pb 1685	corellchrytrioop2n0903m	339	
	Sonata da camera a tre in E major	Op. 2, No. 10	1	pb 1685	corellchrytrioop2n1001m	147	
	Sonata da camera a tre in E major	Op. 2, No. 10	3	pb 1685	corellchrytrioop2n1003m	130	
	Sonata da camera a tre in E major	Op. 2, No. 10	4	pb 1685	corellchrytrioop2n1004m	354	
	Sonata da camera a tre in E♭ major	Op. 2, No. 11	2	pb 1685	corellchrytrioop2n1102m	395	
	Sonata da camera a tre in E♭ major	Op. 2, No. 11	3	pb 1685	corellchrytrioop2n1103m	391	
Handel	Ariodante	HWV 33	10 (Qui d'amore nel suo linguaggio)	pb 1734	handelchryarioda10m	852	24500 (27)
	Ariodante	HWV 33	14 (Volate, amori)	pb 1734	handelchryarioda14m	1220	
	Ariodante	HWV 33	37 (Tu vivi, vivi punito)	pb 1734	handelchryarioda37m	1107	
	Ariodante	HWV 33	81 (Bramo aver mille vite)	pb 1734	handelchryarioda81m	1224	
	Messiah	HWV 56	1,22 (He shall feed his flock)	co 1741	handelchrymessia1-22cm	1278	
	Messiah	HWV 56	2,15 (The Lord gave the word)	co 1741	handelchrymessia2-15m	1167	
	Messiah	HWV 56	3,4 (The trumpet shall sound)	co 1741	handelchrymessia3-04m	2106	
	Trio Sonata in B minor	HWV 386b (Op. 2, No. 1b)	1	co before 1727	handelchryorchop201b01m	796	
	Trio Sonata in F major	HWV 392 (Op. 2, No. 3)	3	co ?1706–7	handelchryorchop20303m	496	
	Trio Sonata in F major	HWV 389 (Op. 2, No. 5)	3	co ?1718–22	handelchryorchop20503m	296	
	Trio Sonata in E major	HWV 394 (Op. 2, No. 9)	4	pb ????	handelchryorchop20904m	1749	
	Concerto Grosso in B♭ major	HWV 313 (Op. 3, No. 2)	4	co ?1715–18	handelchryorchop30204m	615	
	Concerto Grosso in D minor	HWV 316 (Op. 3, No. 5)	4	co ?1717–18	handelchryorchop30504m	915	
	Trio Sonata in A major	HWV 396 (Op. 5, No. 1)	4	pb 1739	handelchryorchop50104m	703	
	Trio Sonata in E minor	HWV 398 (Op. 5, No. 3)	3	pb 1739	handelchryorchop50303m	227	
	Trio Sonata in F major	HWV 401 (Op. 5, No. 6)	4	pb 1739	handelchryorchop50604m	1186	
	Susanna	HWV 66	16 (Without the swain's)	co 1748	handelchrysusann16m	1286	
	Susanna	HWV 66	34 (Ask if yon damask rose)	co 1748	handelchrysusann34m	324	
	Susanna	HWV 66	41 (The torrent that sweeps)	co 1748	handelchrysusann41m	1639	
	Susanna	HWV 66	59 (The blood of innocence)	co 1748	handelchrysusann59m	136	
	Clori, Tirsi e Fileno	HWV 96	2,13 (Come la rondinella)	co 1707	handelhicksclori27m	1241	
	Judas Maccabaeus	HWV 63	8b (Pious orgies, pious airs)	co 1746	handelhicksjudasm08bm	664	
	Judas Maccabaeus	HWV 63	19 (Come, ever-smiling liberty)	co 1746	handelhicksjudasm19m	433	
	Ottone, re di Germania	HWV 15	2 (Overture)	co 1722	handelhicksott02m	1443	
	Ottone, re di Germania	HWV 15	38 (Vieni oglio)	co 1722	handelhicksott38m	696	
	Ottone, re di Germania	HWV 15	69 (Si, mi traete)	co 1722	handelhicksott69m	91	
	Semele	HWV 58	31 (There from mortal cares)	co 1743	handelarnoldsemele31m	610	
Haydn	Symphony No. 1 in D major	Hob. I:1	2	co 1758	haydnbhsyms-00102m	1504	24490 (10)
	String Quartet in C major	Hob. III:57 (Op. 54, No. 2)	3	co 1788–89	haydndoverquartop54n203m	628	
	String Quartet in A major	Hob. III:60 (Op. 55, No. 1)	1	co 1788–90	haydndoverquartop55n101m	2512	
	String Quartet in B♭ major	Hob. III:62 (Op. 55, No. 3)	2	co 1788–90	haydndoverquartop55n302m	1514	
	String Quartet in C major	Hob. III:65 (Op. 64, No. 1)	2	co 1790–91	haydndoverquartop64n102m	652	
	String Quartet in B minor	Hob. III:68 (Op. 64, No. 2)	2	co 1790–91	haydndoverquartop64n202m	1333	
	Symphony No. 99 in E♭ major	Hob. I:99	2	co 1793	haydndoversyms-09902m	3391	
	Symphony No. 100 in G major ('Military')	Hob. I:100	4	co 1794	haydndoversyms-10004m	6384	
	Symphony No. 102 in B♭ major	Hob. I:102	2	co 1794–5	haydndoversyms-10202m	2343	
	Symphony No. 103 in E♭ major ('Drumroll')	Hob. I:103	2	co 1795	haydndoversyms-10302m	4229	

continued on next page

continued from previous page

Composer	*Work title*	*Work no.*	*Mvt.*	*Year*	*File name*	*No. notes*	*No. notes (mvts) for composer*
Mozart	Piano Concerto No. 19 in F major	K 459	3	co 1784	mozartbhconck45903m	8194	24494 (9)
	Clarinet Concerto in A major	K 622	2	co 1791	mozartbhconck62202m	2072	
	Duo for violin and viola in G major	K 423	2	co 1783	mozartbhduosk42302m	896	
	Duo for violin and viola in B♭ major	K 424	2	co 1783	mozartbhduosk42402m	634	
	String Quartet No. 1 in G major	K 80	3	co 1770	mozartbhqrtetsk08003m	514	
	String Quartet No. 3 in G major	K 156	2	co 1772	mozartbhqrtetsk15602m	1012	
	String Quartet No. 5 in F major	K 158	2	co 1772–3	mozartbhqrtetsk15802m	1296	
	String Quartet No. 7 in E♭ major	K 160	2	co 1773	mozartbhqrtetsk16002m	942	
	Symphony No. 38 in D major ('Prague')	K 504	1	co 1786	mozartbhsymk50401m	8934	
Telemann	Sonata in F major for violin or flute with figured bass	TWV 41:F3	2	pb 1734	telemabrusse1734_40102m	1070	24500 (48)
	Sonata in F major for violin or flute with figured bass	TWV 41:F3	3	pb 1734	telemabrusse1734_40103m	227	
	Sonata in E minor for violin or flute with figured bass	TWV 41:e4	2	pb 1734	telemabrusse1734_40202m	809	
	Sonata in E minor for violin or flute with figured bass	TWV 41:e4	3	pb 1734	telemabrusse1734_40203m	249	
	Sonata in A major for violin or flute with figured bass	TWV 41:A5	2	pb 1734	telemabrusse1734_40302m	904	
	Sonata in A major for violin or flute with figured bass	TWV 41:A5	3	pb 1734	telemabrusse1734_40303m	585	
	Sonata in C major for violin or flute with figured bass	TWV 41:C4	2	pb 1734	telemabrusse1734_40402m	692	
	Sonata in C major for violin or flute with figured bass	TWV 41:C4	3	pb 1734	telemabrusse1734_40403m	256	
	Sonata in G minor for violin or flute with figured bass	TWV 41:g7	3	pb 1734	telemabrusse1734_40503m	579	
	Sonata in D major for violin or flute with figured bass	TWV 41:D8	3	pb 1734	telemabrusse1734_40603m	269	
	Sonata in D minor for violin or flute with figured bass	TWV 41:d3	4	pb 1734	telemabrusse1734_40704m	642	
	Sonata in G major for violin or flute with figured bass	TWV 41:G8	4	pb 1734	telemabrusse1734_40804m	827	
	Sonata in B minor for violin or flute with figured bass	TWV 41:h5	3	pb 1734	telemabrusse1734_40903m	292	
	Sonata in B minor for violin or flute with figured bass	TWV 41:h5	4	pb 1734	telemabrusse1734_40904m	669	
	Sonata in E major for violin or flute with figured bass	TWV 41:E6	3	pb 1734	telemabrusse1734_41003m	282	
	Sonata in E major for violin or flute with figured bass	TWV 41:E6	4	pb 1734	telemabrusse1734_41004m	898	
	Sonata in A minor for violin or flute with figured bass	TWV 41:a5	4	pb 1734	telemabrusse1734_41104m	657	
	Sonata in F♯ minor for violin or flute with figured bass	TWV 41:fis1	4	pb 1734	telemabrusse1734_41204m	787	
	Germania mit ihrem Chor	TWV 12:1c	1b	co 1716	telemamagdebgerman01bm	1886	
	Germania mit ihrem Chor	TWV 12:1c	4	co 1716	telemamagdebgerman04m	1360	
	Germania mit ihrem Chor	TWV 12:1c	5	co 1716	telemamagdebgerman05m	36	
	Germania mit ihrem Chor	TWV 12:1c	9	co 1716	telemamagdebgerman09m	115	
	Germania mit ihrem Chor	TWV 12:1c	13	co 1716	telemamagdebgerman13m	24	
	Germania mit ihrem Chor	TWV 12:1c	19	co 1716	telemamagdebgerman19m	77	
	Germania mit ihrem Chor	TWV 12:1c	21	co 1716	telemamagdebgerman21m	78	
	Germania mit ihrem Chor	TWV 12:1c	22	co 1716	telemamagdebgerman22m	901	
	Germania mit ihrem Chor	TWV 12:1c	26	co 1716	telemamagdebgerman26m	35	
	Germania mit ihrem Chor	TWV 12:1c	27	co 1716	telemamagdebgerman27m	1048	
	Germania mit ihrem Chor	TWV 12:1c	30	co 1716	telemamagdebgerman30m	68	
	Germania mit ihrem Chor	TWV 12:1c	34	co 1716	telemamagdebgerman34m	93	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	1.1.2	co 1726–36	telemamagdeborpheu1012m	218	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	1.2.1	co 1726–36	telemamagdeborpheu1021m	1037	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	1.2.5	co 1726–36	telemamagdeborpheu1025m	106	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	1.5.1	co 1726–36	telemamagdeborpheu1051m	827	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	1.8.3	co 1726–36	telemamagdeborpheu1083m	649	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	1.9.2	co 1726–36	telemamagdeborpheu1092m	980	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	1.10.3	co 1726–36	telemamagdeborpheu1103m	388	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	2.1.5	co 1726–36	telemamagdeborpheu2015m	24	

continued on next page

continued from previous page

Composer	*Work title*	*Work no.*	*Mvt.*	*Year*	*File name*	*No. notes*	*No. notes (mvts) for composer*
Telemann (*cont.*)	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	2.3.1	co 1726–36	telemamagdeborpheu2031m	684	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	2.6.2	co 1726–36	telemamagdeborpheu2062m	670	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	2.8.1	co 1726–36	telemamagdeborpheu2081m	97	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	3.1.3	co 1726–36	telemamagdeborpheu3013m	283	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	3.3.1	co 1726–36	telemamagdeborpheu3031m	99	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	3.3.4	co 1726–36	telemamagdeborpheu3034m	589	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	3.5.2	co 1726–36	telemamagdeborpheu3052m	831	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	3.7.4	co 1726–36	telemamagdeborpheu3074m	270	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	3.7.6	co 1726–36	telemamagdeborpheu3076m	223	
	Opheus oder Die wunderbare Beständigkeit der Liebe	TWV 21:18	3.8.2	co 1726–36	telemamagdeborpheu3082m	110	
Vivaldi	Concerto in D major from 'L'Estro Armonico'	RV 549 (Op. 3, No. 1)	2	pb 1711	vivalddoverop30102m	590	24497 (20)
	Concerto in D minor from 'L'Estro Armonico'	RV 578 (Op. 3, No. 2)	2	pb 1711	vivalddoverop30202m	511	
	Concerto in E minor from 'L'Estro Armonico'	RV 550 (Op. 3, No. 4)	1	pb 1711	vivalddoverop30401m	526	
	Concerto in E major ('La primavera') from 'Le quattro stagioni'	RV 269 (Op. 8, No. 1)	1	pb ?1725	vivalddoverop8011m	2291	
	Concerto in F major ('L'autunno) from 'Le quattro stagioni'	RV 293 (Op. 8, No. 3)	2	pb ?1725	vivalddoverop8032m	139	
	Concerto in F minor ('L'inverno) from 'Le quattro stagioni'	RV 297 (Op. 8, No. 4)	1	pb ?1725	vivalddoverop8041m	3072	
	Concerto in C major ('Il piacere') from 'Il cimento dell'armonia e dell'inventione'	RV 180 (Op. 8, No. 6)	3	pb ?1725	vivalddoverop8063m	1558	
	Concerto in D minor from 'Il cimento dell'armonia e dell'inventione'	RV 236 (Op. 8, No. 9)	2	pb ?1725	vivalddoverop8092m	256	
	Concerto in D major from 'Il cimento dell'armonia e dell'inventione'	RV 210 (Op. 8, No. 11)	3	pb ?1725	vivalddoverop8113m	2529	
	Concerto in C major from 'Il cimento dell'armonia e dell'inventione'	RV 178 (Op. 8, No. 12)	2	pb ?1725	vivalddoverop8122m	158	
	Concerto in G minor ('La notte')	RV 439 (Op. 10, No. 2)	4	pb 1728	vivaldleceneop100204m	82	
	Concerto in D major ('Il gardellino')	RV 428 (Op. 10, No. 3)	2	pb 1728	vivaldleceneop100302m	200	
	Concerto in G major	RV 435 (Op. 10, No. 4)	1	pb 1728	vivaldleceneop100401m	1515	
	Concerto in G major	RV 437 (Op. 10, No. 6)	3	pb 1728	vivaldleceneop100603m	2051	
	Concerto in G major from 'L'estro armonico'	RV 310 (Op. 3, No. 3)	2	pb 1711	vivaldleceneop3032m	440	
	Concerto in A minor from 'L'estro armonico'	RV 359 (Op. 3, No. 6)	2	pb 1711	vivaldleceneop3062m	290	
	Concerto in D major from 'L'estro armonico'	RV 230 (Op. 3, No. 9)	1	pb 1712	vivaldleceneop3091m	1193	
	Concerto in E major from 'L'estro armonico'	RV 265 (Op. 3, No. 12)	1	pb 1712	vivaldleceneop3121m	1941	
	Concerto in G minor ('L'estate') from 'Le quattro stagioni'	RV 315 (Op. 8, No. 2)	3	pb ?1725	vivaldleceneop8023m	3165	
	Concerto in C major ('Il piacere') from 'Il cimento dell'armonia e dell'inventione'	RV 180 (Op. 8, No. 6)	1	pb ?1725	vivaldleceneop8061m	1990	

Table 1.4: The test corpus used in this study. Each entry in the *Year* column is prefixed with 'co', 'pb' or 'fp', indicating that the year is that in which the work was, respectively, composed, published or first performed. Information for this column was obtained from Boyd (1983) (Bach), Arnold (1983) (Beethoven, Handel, Vivaldi), Piperno (1987) (Corelli), `http://gfhandel.org` (Handel), `http://home.wxs.nl/~cmr/haydn/catalog/main.htm` (Haydn), `http://www.openmozart.net` (Mozart), `http://www.heartflyer.com/mozartsworks.htm` (Mozart), `http://infopuq.uquebec.ca/~uss1010/catal/telemann/telgp.html` (Telemann), `http://www.andrews.edu/~mack/pnotes/dec1501.html` (Telemann, *Germania*), `http://www.classical.net/music/recs/reviews/h/hmu01618a.html` (Telemann, *Orpheus*) and `http://www.antonio-vivaldi.org/VIVtimeline.htm` (Vivaldi, Op. 10).

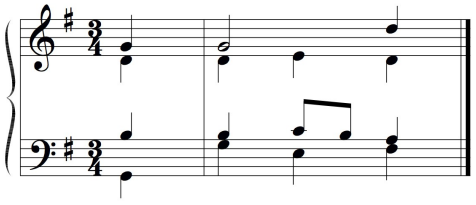


Figure 1.2: The opening of J. S. Bach's chorale harmonisation, "Aus meines Herzens Grunde" (BWV 269).

```
{  ⟨0, "Gn2", 2, 4⟩, ⟨0, "Bn3", 2, 3⟩, ⟨0, "Dn4", 2, 2⟩, ⟨0, "Gn4", 2, 1⟩,
   ⟨2, "Gn3", 2, 4⟩, ⟨2, "Bn3", 2, 3⟩, ⟨2, "Dn4", 2, 2⟩, ⟨2, "Gn4", 4, 1⟩,
   ⟨4, "En3", 2, 4⟩, ⟨4, "Cn4", 1, 3⟩, ⟨4, "En4", 2, 2⟩,
   ⟨5, "Bn3", 1, 3⟩,
   ⟨6, "Fs3", 2, 4⟩, ⟨6, "An3", 2, 3⟩, ⟨6, "Dn4", 2, 2⟩, ⟨6, "Dn5", 2, 1⟩   }
```

Figure 1.3: The OPNDV dataset representing the score in Figure 1.2.

of notes to voices was done reasonably for all the works in the corpus used here. As an example, Figure 1.3 gives the OPNDV dataset representing the score in Figure 1.2 and Figure 1.4 shows the Lisp list that represents the OPNDV dataset in Figure 1.3 in the OPNDV file in the test corpus.

In order to run Temperley's (2001) pitch spelling algorithm on this corpus, each OPNDV file had to be converted into the format accepted by his and Sleator's `meter` and `harmony` programs. The output generated by these programs depends on the absolute values of the durations and onset times of the notes in milliseconds. However, the onset times and durations in an OPNDV file are expressed relatively as multiples of some unspecified base time unit. Therefore, a reasonable tempo was determined for each movement in the test corpus and this was used to generate a file in the format accepted by Temperley and Sleator's programs. It was also found that Temperley and Sleator's programs were unable to cope with input files in which two or more notes with the same pitch begin simultaneously. Before converting to Temperley and Sleator's format, such cases therefore had to be removed. Thus, in each OPNDV file, wherever two or more notes with the same pitch p began simultaneously at time t, all notes with pitch p and onset time t were removed except the one with the longest duration. This generated for each OPNDV file, a new OPNDV file that was suitable for converting to Temperley and

```
(  (0 "Gn2" 2 4) (0 "Bn3" 2 3) (0 "Dn4" 2 2) (0 "Gn4" 2 1)
   (2 "Gn3" 2 4) (2 "Bn3" 2 3) (2 "Dn4" 2 2) (2 "Gn4" 4 1)
   (4 "En3" 2 4) (4 "Cn4" 1 3) (4 "En4" 2 2)
   (5 "Bn3" 1 3)
   (6 "Fs3" 2 4) (6 "An3" 2 3) (6 "Dn4" 2 2) (6 "Dn5" 2 1)  )
```

Figure 1.4: A Lisp list representing the OPNDV dataset in Figure 1.3.

Sleator's format. Note that the resulting set of 216 files, in which duplicate notes with the same onset and pitch had been removed, formed the corpus on which *all* the algorithms considered here (not just the algorithms of Temperley and Sleator) were tested. The effect on algorithm performance of removing such duplicate notes in the manner just described is discussed further in Appendix A.

As discussed in section 1.3.3.1 above, a test corpus should ideally be as large as possible and should represent as well as possible the population of works that the algorithms to be evaluated will be used to process. However, it was also pointed out above that, in order to compare the output of an algorithm with the 'ground truth' spellings accurately and efficiently, one needs to have accurate encodings of authoritative scores of the music in the corpus. The MuseData collection was the only publicly available collection of encodings of authoritative scores that was both sufficiently large and sufficiently accurate for the purposes of this study. I therefore used a subset of this collection to generate the test corpus used here. This subset was chosen so that it satisfied the following constraints.

1. The corpus had to be sufficiently small for it to be processed within a reasonable length of time by the large number of different algorithms considered in this study.

2. The corpus had to be small enough for me to 'proof-listen' to each encoding in order to ensure accuracy and determine an appropriate tempo for conversion to Temperley and Sleator's format.

3. In order to evaluate the style dependence of the algorithms, it was decided that the proportion of notes in the corpus in works by a particular composer had to be as similar as possible for each composer. This was to ensure that the results obtained by an algorithm on the music of one composer in the corpus were comparable with those obtained for another composer.

4. The number of composers represented had to be as high as possible in order to test for style dependence and in order for the population represented by the corpus to be as large as possible.

5. In order for the subset of the corpus by a particular composer to be as representative as possible of that composer's works, the movements selected for a given composer were as evenly distributed as possible among the suitable movements by that composer in the MuseData collection.

6. Only complete movements from works were used.

7. The corpus was as large as possible given the foregoing constraints.

The resulting corpus is defined in Table 1.4. From now on, this test corpus will be denoted by the symbol $\mathcal{C}$. Note that $\mathcal{C}$ contains almost exactly 24500 notes for each of the eight composers represented. The works in $\mathcal{C}$ were composed between 1681 and 1808. A corpus containing works in a wider range of styles, including, for example, romantic, jazz and popular tonal works composed since 1808, would have been preferable. Unfortunately, there does not yet exist a

reliable and sufficiently representative collection of publicly available encodings of tonal works from this later period.

Nevertheless the test corpus used here is over 3 times larger than the largest corpus used by other authors to evaluate pitch spelling algorithms (Cambouropoulos, 2003). Overall, the corpus may be considered reasonably representative of music composed in Europe between 1681 and 1808 and quite strongly representative of music composed between these dates by the composers whose works appear in the corpus. The set of movements in the corpus by Bach span a variety of different genres, including secular and sacred, instrumental, choral and orchestral works. They also span most of his career, ranging in date from 1715 to about 1742. This subset of the corpus may therefore be considered fairly representative of Bach's works in general. The set of movements by Beethoven in the test corpus consists of 5 movements from 3 of his symphonies. These movements are too closely related in terms of genre, instrumentation and date of composition to be representative of all of Beethoven's works. Nevertheless, this set of movements may be considered reasonably representative of Beethoven's symphonic works. The set of movements by Corelli consists of 71 movements from his Opp. 1 and 2 Trio Sonatas. About half of these are from his church sonatas and half are from his chamber sonatas, so both of Corelli's two main styles of composition in this genre are represented. The collection of movements in the corpus by Handel contains movements from his operas, cantatas, orchestral works and chamber works and range in date of composition across his career from about 1706 to 1748. This collection may therefore be considered fairly representative of his works in general. The set of movements in the corpus by Haydn contains movements from his string quartets and his symphonies. Apart from the movement from his first symphony composed in 1758, the movements are from works composed between 1788 and 1795. Therefore this set of works is primarily representative of Haydn's later chamber and orchestral works. The set of works by Mozart in the corpus again consists only of movements from instrumental and orchestral works. However, these movements range in date from 1770 to 1791. This small corpus may therefore be considered reasonably representative of Mozart's chamber and orchestral works. The movements in the corpus by Telemann include movements from instrumental, operatic and choral works. This set of movements may therefore be considered weakly representative of Telemann's works in general. Finally, the movements in the corpus by Vivaldi are taken from his Opp. 3, 8 and 10 concerti and therefore constitute a reasonably representative sample of his work in this genre.

1.3.4 Measuring spelling accuracy

1.3.4.1 Note error count, note error rate and note accuracy

The most obvious way of measuring spelling accuracy is to count the total number of notes in the corpus that are spelt incorrectly by the algorithm. Let us therefore define the *note error count* of an algorithm A over a set of movements, S, denoted by $\mathrm{NEC}(A, S)$, to be the total number of notes in S spelt incorrectly by A.

The main problem with using the note error count as a measure for spelling accuracy is that, for a given algorithm and a given population, we expect the note error count to increase roughly proportionally with the number of notes in the corpus—provided, of course, that the corpus is truly representative of the population. However, for a given algorithm and a given

population, we really want to be able to expect the value returned by our performance metric for spelling accuracy to be roughly constant for any properly representative corpus taken from the population. Nevertheless, the note error count achieved by an algorithm on the test corpus will always be included in the results given in this study.

A better performance metric for spelling accuracy might therefore be what I shall call the *note error rate*. The *note error rate* of an algorithm A over a set of movements S, denoted by $\mathrm{NER}(A, S)$, is the proportion of notes in S spelt incorrectly by A. That is,

$$\mathrm{NER}(A, S) = \mathrm{NEC}(A, S)/\mathrm{NNOTES}(S) \tag{1.1}$$

where $\mathrm{NNOTES}(S)$ denotes the number of notes in S. Within the bounds of sampling error, it seems reasonable to expect the note error rate for a given algorithm and a given population to be roughly constant for any properly representative corpus taken from the population. This suggests that we can use the note error rate of an algorithm over any particular representative corpus taken from a population as an estimate of the note error rate that the algorithm would achieve on the population as a whole.

In most publications to date in which pitch spelling algorithms have been evaluated or compared, the authors have used the proportion of notes in a corpus spelt correctly as a measure of spelling accuracy (Cambouropoulos, 2003; Chew and Chen, 2003a,b, 2005; Meredith, 2003, 2005; Meredith and Wiggins, 2005; Stoddard et al., 2004; Temperley, 2001). Let's therefore define the *note accuracy*, $\mathrm{NA}(A, S)$, of an algorithm A over a set of movements S to be the proportion of notes in S spelt correctly by A. For a given algorithm and corpus, the sum of the note accuracy and note error rate is always 1. That is,

$$\mathrm{NA}(A, S) = \frac{\mathrm{NNOTES}(S) - \mathrm{NEC}(A, S)}{\mathrm{NNOTES}(S)} = 1 - \frac{\mathrm{NEC}(A, S)}{\mathrm{NNOTES}(S)} = 1 - \mathrm{NER}(A, S). \tag{1.2}$$

The note accuracy therefore gives us exactly the same information as the note error rate. In this study, I shall always include in the results the note accuracy achieved by an algorithm, expressed as a percentage.

If two algorithms, A_1 and A_2, are both run on a corpus, C, and achieve note error rates of NER_1 and NER_2, respectively, then it is also interesting to consider the *proportional difference* between these note error rates, which is given by

$$\Delta\mathrm{NER}_{\mathrm{Pr}}(NER_1, NER_2) = \frac{NER_2 - NER_1}{NER_1}. \tag{1.3}$$

The proportional difference, $\Delta\mathrm{NER}_{\mathrm{Pr}}(NER_1, NER_2)$, gives the proportional increase in the note error rate caused by changing from A_1 to A_2. A positive value of $\Delta\mathrm{NER}_{\mathrm{Pr}}(NER_1, NER_2)$ implies that changing from A_1 to A_2 *increases* the note error rate by $(100 \times \Delta\mathrm{NER}_{\mathrm{Pr}}(NER_1, NER_2))$ per cent. In this study, proportional differences in note error rates will always be expressed as percentages.

1.3.4.2 Instances where a movement is spelt in a different key from the original

If every pitch name in a movement is transposed by the same interval, the resulting score will still be correctly notated—it will just be in a different key. For example, if every note in a correctly notated movement in G♯ minor is transposed up a diminished second, the resulting score will be in A♭ minor but it will still be correctly notated. Therefore, if every pitch name assigned in a movement by an algorithm is the same interval away from the corresponding pitch name in the original 'ground truth' score, the computed spelling for the movement should be considered correct. Consequently, in this study, for every movement and every algorithm, three spellings were generated:

1. the spelling s generated directly by the algorithm;
2. another spelling generated by transposing s up a diminished second; and
3. a third spelling generated by transposing s down a diminished second.

The note error counts were then calculated for all three of these spellings and the spelling with the smallest number of errors was defined to be the spelling generated by the algorithm for that movement for the purpose of the results reported here.

This point is discussed further in Appendix B.

1.3.4.3 Enharmonic changes for notational convenience

Occasionally in tonal music, a modulation occurs that results in a passage being in an extremely flat or extremely sharp key that, if notated correctly, would require many double flats or double sharps to be used. Composers often choose to notate such passages in enharmonically equivalent keys that require fewer accidentals because this often makes the music easier to read. When this happens, a modulation that sounds as though it is to a closely related (but extreme) key is notated as though it were a modulation to a distantly related but less extreme key, resulting in a sudden *enharmonic change* in the notated score that does not correspond to the way in which the music is interpreted by a listener (Temperley, 2001, p. 135). Indeed, in some cases, such an enharmonic change is notated where there is no perceived modulation. For example, Figure 1.5 shows a passage from the fourth movement of Haydn's Symphony No. 100 in G major ('Military') (Hob. I:100), where the notated key suddenly changes from D♭ major to C♯ major at the beginning of bar 166, even though no change in key is perceived by the listener.[6] Presumably, the music is notated in this way in order to avoid having to write much of the remainder of the movement in extremely flat keys. For example, if the music were notated without this enharmonic change, there would be a passage in F♭ major starting at bar 174. It could be argued that this enharmonic change in the original score is "incorrect" because it fails to represent correctly the tonal relationship perceived by the listener between the last notes in bar 164 and the first note in bar 166. Thus, if all the notes in this movement from bar 166 onward were transposed up a diminished second, the resulting score would still be correctly notated. On the other hand, the original notation containing the enharmonic change should

[6] The perceived key does change to C♯ minor at bar 167, but this only happens after the minor mediant sounds in that bar. In bar 166, the key is perceived to be the same as in the previous bar despite the enharmonic change.

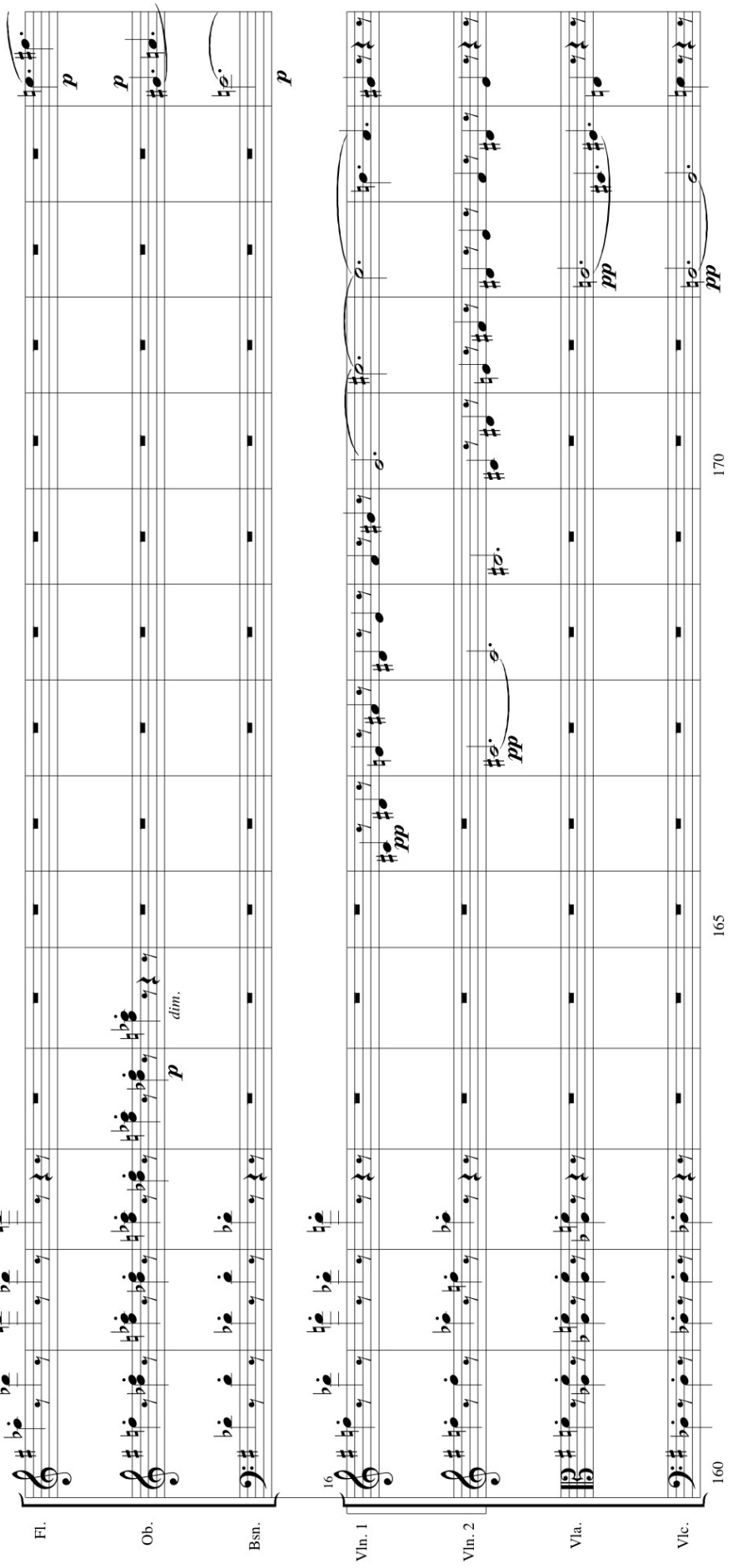


Figure 1.5: Bars 160–174 from the fourth movement of Haydn's Symphony No. 100 in G major ('Military') (Hob. I:100). Note the sudden enharmonic change from D♭ major in bar 164 to C♯ major in bar 166. Note also that the modulation to C♯ minor does not happen until bar 167—that is, after the enharmonic change.

also be considered "correct" because, first, this is how Haydn and numerous subsequent expert editors chose to notate this music and it would be highly presumptuous of us to decide that their notation is "incorrect"; and, second, the resulting score is indeed easier to read and therefore more serviceable from a purely practical point of view.

The sudden enharmonic change in the fourth movement of Haydn's Symphony No. 100 shown in Figure 1.5 is, in fact, the only instance of such a change that I could find in the test corpus. Therefore, I compared the output of each algorithm for this movement with two "correct" spellings: one in which the notes are spelt as they are in the original score; and a second, modified version, in which all the notes in the original score up to bar 165 are transposed down a diminished second. When an algorithm performed better on the modified version than on the original, I give alternative values for its note error count and note accuracy over the test corpus and over the music by Haydn in the corpus.

1.3.4.4 Measuring the significance of the difference between two note accuracies

Let's suppose that we run two algorithms, A_1 and A_2, on a corpus, C, which is a representative sample of movements taken from a population, P, and we find that $\mathrm{NA}(A_1, C) < \mathrm{NA}(A_2, C)$. This suggests that $\mathrm{NA}(A_1, P)$ would also be less than $\mathrm{NA}(A_2, P)$, however, it's not immediately clear how confident we can be that this is actually the case. Intuition tells us that the larger $\mathrm{NNOTES}(C)$ and the larger the difference between $\mathrm{NA}(A_1, C)$ and $\mathrm{NA}(A_2, C)$, the more confident we can be that $\mathrm{NA}(A_1, P)$ is less than $\mathrm{NA}(A_2, P)$. However, this intuition does not allow us to estimate quantitatively the degree of confidence that we are justified in feeling that $\mathrm{NA}(A_1, P) < \mathrm{NA}(A_2, P)$ given $\mathrm{NA}(A_1, C)$ and $\mathrm{NA}(A_2, C)$ and the fact that $\mathrm{NA}(A_1, C) < \mathrm{NA}(A_2, C)$. To do this, we would need to find an appropriate method for measuring the statistical significance of the difference between the spelling accuracies of two pitch spelling algorithms when they are both run on the same test corpus.

To date, only Meredith (2005) has attempted to measure the significance of the difference between the spelling accuracies achieved by two pitch spelling algorithms. He ran four algorithms on a large test corpus and used McNemar's test (Dietterich, 1998; Everitt, 1992, pp. 19–22; Fleiss, 1973, pp. 73–77; McNemar, 1969, pp. 54–8) to measure the significances of the differences between the spelling accuracies of these algorithms. McNemar's test is typically used to measure the significance of the difference between the frequencies with which a particular dichotomous outcome occurs within two matched samples. For example, let's suppose that an election is approaching and an investigator takes a poll of 100 people in which each person is asked if he or she will vote Labour. Then each participant is asked to watch a Labour party political broadcast and again asked if he or she will vote Labour after seeing the broadcast. The investigator wishes to know whether or not the broadcast has caused a significant change in the proportion of people who intend to vote Labour. In this situation, McNemar's test is an appropriate test to use to measure the significance of the difference between the proportion of participants who responded "Yes" before the broadcast and the proportion who responded "Yes" after it.

On the face of it, this polling example might appear to be exactly analogous to that in which one runs two pitch spelling algorithms on the same set of movements and each note in each movement is spelt either correctly or incorrectly by each algorithm. The two algorithms

correspond to the "Before broadcast" and "After broadcast" conditions in the polling example, the notes correspond to the participants and the dichotomous outcomes of "correctly spelt" and "incorrectly spelt" correspond to the "Yes" and "No" responses of the participants in the polling example. However, McNemar's test is only appropriate when the responses or outcomes are independent of each other. In the polling example, we can validly assume that each participant's response is not affected by or dependent on the responses of any other participants. However, for a number of the pitch spelling algorithms considered in this study, the pitch name assigned to a note may depend to some extent on the pitch names that the algorithm assigns to other notes around it. Therefore, strictly speaking, McNemar's test may not be appropriate for measuring the significance of the difference between the proportions of notes spelt correctly by two pitch spelling algorithms when they are run on the same test corpus. The significances reported by Meredith (2005) should therefore be interpreted with some caution.

A more appropriate test for measuring the significance of the difference between the spelling accuracies achieved by two algorithms over the same test corpus is the matched-sample t test (Howell, 1982, p. 126). This test is typically used to determine whether there is a significant difference between the conditions in an experiment that uses a within-participants design. For example, let's suppose that we want to know whether listening to classical music improves one's ability to learn arbitrary information rapidly. We carry out an experiment in which each participant writes down all the words he or she can remember from a word list presented under two different conditions. In one condition, the participant has to learn the word list while listening to classical music; whereas, in the other condition, he or she learns the list in silence. For each participant, we obtain two scores: the number of words recalled with and without the music. We want to know whether the mean difference between these two scores over all the participants is significantly different from zero. We can determine this by using a matched-sample t test. This test involves dividing the mean difference, $\bar{D}$, between the scores under the two conditions by the standard error in the mean, and then using the resulting t statistic to compute a p value which gives an estimate of the probability that the difference between $\bar{D}$ and zero is due to chance.

Determining whether one pitch spelling algorithm is more accurate than another when run on a set of movements would seem to be analogous to determining whether listening to classical music enhances rapid learning when a number of participants take part in an experiment such as the one sketched in the previous paragraph. The two algorithms would correspond to the two conditions (i.e., with or without music) and each movement would correspond to a participant. Thus, the number of words recalled by a particular participant under a particular condition would correspond to the percentage of notes spelt correctly in a particular movement by a particular algorithm. We could then use the t test to determine whether the mean difference between the corresponding movement note accuracies for the two algorithms is significantly different from zero.

The t test is only appropriate if we can assume that the responses of the different participants are of equal importance—that is, that they each provide the same amount of evidence. However, the movements in $\mathcal{C}$ range in size from 24 notes to 8934 notes, and it hardly seems reasonable to assume that the note accuracy achieved over a 24-note movement is as important (i.e., provides

the same amount of evidence) as that achieved over an 8934-note movement. Therefore, I believe it would be inappropriate to treat each movement in $\mathcal{C}$ as though it were a participant in a within-participants experimental design.

We therefore have the problem of partitioning $\mathcal{C}$ into 'chunks' that can reasonably be treated as 'participants'. The number of notes in each of these chunks should be approximately the same so that we can reasonably assign equal importance to the note accuracies that an algorithm achieves over the different chunks. Also, the note accuracy that an algorithm achieves over one chunk should not depend on the note accuracy it achieves over any other chunk, since the t test is only appropriate when the responses or outcomes are independent of each other. This therefore eliminates the possibility of using each note as a chunk. In fact, it means that each chunk must contain some number of complete movements, since, if a single movement were divided between two chunks, an algorithm's note accuracies over these two chunks might not be independent.

The next question that needs to be answered is whether the precise way in which $\mathcal{C}$ is partitioned into chunks is important, provided that each chunk contains approximately the same number of notes and each chunk is a set of complete movements. We can try to answer this question by, first, generating various different partitions of $\mathcal{C}$ in which each chunk contains approximately the same number of notes and each chunk is a set of complete movements; and then comparing two algorithms using the t test on each of these partitions. If the values of significance returned by the t test for the different partitions are approximately the same, then we can conclude that the precise way in which $\mathcal{C}$ is partitioned into chunks is not important.

Five partitions of $\mathcal{C}$ were generated as follows:

1. the first, G_2, contained two chunks, each containing 97986 notes;
2. the second, G_4, contained four chunks, each containing 48993 notes;
3. the third, G_8, contained 8 chunks, of which 4 contained 24496 notes and 4 contained 24497 notes;
4. the fourth, G_{16}, contained 16 chunks, of which 12 contained 12248 notes and 4 contained 12249 notes; and
5. the fifth, G_{Comp}, contained 8 chunks, each containing the works in the corpus by a particular composer.

The partitions G_2, G_4, G_8 and G_{16} were constructed so that all the chunks in a particular partition had as nearly as possible the same number of notes.[7] Every chunk in every partition was a set of complete movements.

Tables 1.5(a)–(f) give the note accuracies achieved by three algorithms, A, B and C, over the whole test corpus, $\mathcal{C}$, and each of the five partitions just described. Table 1.5(g) shows the p values returned by the matched-sample t test when these algorithms were compared over each of the five partitions. Figure 1.6 shows the information in Table 1.5(g) in graphical form. These results suggest that the p values returned by the matched-sample t test vary considerably depending on the precise way that the test corpus is partitioned into chunks. For example, we see

[7] This was done using the online implementation of the Complete Greedy Algorithm at <http://www.chez.com/starshoot/partition/index.php3>.

(a) Note accuracies over complete test corpus $\mathcal{C}$.

Algorithm	*NA* (%)
A	99.439
B	99.310
C	99.429

(b) Note accuracies over each chunk in G_2.

Algorithm	*NA* (%)	
	1	2
A	99.661	99.216
B	99.537	99.084
C	99.644	99.216

(c) Note accuracies over each chunk in G_4.

Algorithm	*NA* (%)			
	1	2	3	4
A	99.620	99.702	98.963	99.469
B	99.506	99.567	98.832	99.335
C	99.549	99.739	98.988	99.445

(d) Note accuracies over each chunk in G_8.

Algorithm	*NA* (%)							
	1	2	3	4	5	6	7	8
A	99.604	99.637	99.673	99.731	98.567	99.359	99.706	99.233
B	99.424	99.588	99.559	99.575	98.396	99.269	99.559	99.110
C	99.465	99.633	99.718	99.759	98.555	99.420	99.653	99.237

(e) Note accuracies over each chunk in G_{16}.

Algorithm	*NA* (%)															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	99.600	99.608	99.845	99.429	99.608	99.739	99.510	99.951	99.469	97.665	99.184	99.535	99.584	99.829	98.759	99.706
B	99.306	99.543	99.739	99.437	99.592	99.526	99.257	99.894	99.184	97.608	98.988	99.551	99.396	99.722	98.628	99.592
C	99.355	99.575	99.804	99.461	99.747	99.690	99.567	99.951	99.453	97.657	99.233	99.608	99.429	99.878	98.800	99.673

(f) Note accuracies over each chunk in G_{Comp}.

Algorithm	*NA* (%)							
	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi
A	99.837	98.440	99.988	99.735	99.163	99.412	99.608	99.326
B	99.861	98.273	99.931	99.731	98.881	99.057	99.539	99.208
C	99.914	98.359	99.996	99.751	99.151	99.273	99.682	99.314

(g) p values obtained using the matched-sample t test for each pair of algorithms and each partition.

Algorithms compared	G_2	G_4	G_8	G_{16}	G_{Comp}
A and B	0.01989	0.0001222	7.416e-05	0.0001206	0.02924
B and C	0.06635	0.02527	0.000427	0.0001105	0.005406
C and A	0.5	0.7606	0.7083	0.7066	0.748

Table 1.5: Note accuracies for each of three algorithms over (a) the whole test corpus, (b) the chunks in G_2, (c) the chunks in G_4, (d) the chunks in G_8, (e) the chunks in G_{16} and (f) the chunks in G_{Comp}. Table (g) shows the p value returned by the matched-sample t test for each pair of algorithms and each partition.

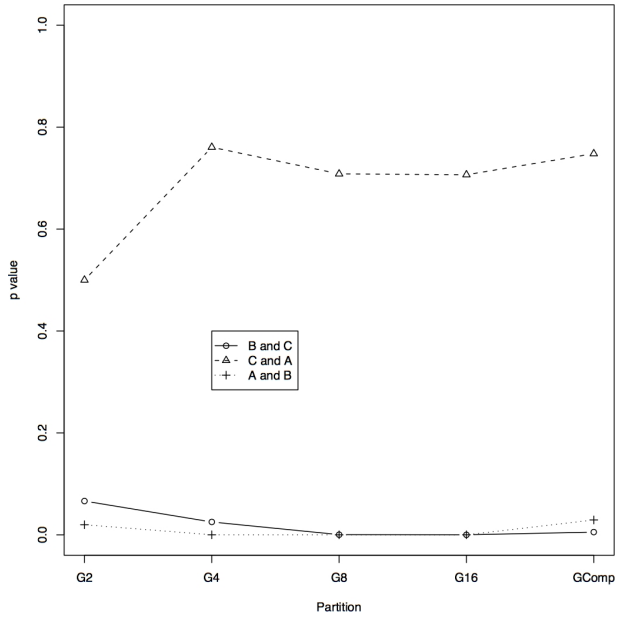


Figure 1.6: Graph showing the p values returned by the matched-sample t test for each of the partitions G_2, G_4, G_8, G_{16} and G_{Comp} and each pairwise comparison of the algorithms A, B and C.

that, for 2 of the 3 comparisons, the p values for G_2 disagree considerably with the other values. This is presumably because the t test is not very reliable when the number of observations is as low as 2. However, for each of the three comparisons, the values for G_4, G_8, G_{16} and G_{Comp} agree with each other well enough for us to be able to get some indication of the significance of the difference in each case. Therefore, in this study, the statistical significance of the difference in note accuracy between two algorithms will be estimated by the mean of the four p values returned by the t test when it is computed over G_4, G_8, G_{16} and G_{Comp}. However, it must be remembered that, in general, the p value returned by the t test depends quite heavily on the way that the test corpus is partitioned and that there is no strong *a priori* reason for choosing one partition over any other. Therefore, the p values computed using this method should be interpreted as no more than rough indicators of significance. Also, computing these estimates of significance for every pair of algorithms considered in this study would be far too cumbersome and time-consuming. Moreover, presenting and discussing these computed significance values would over-burden the results presented. For these reasons, I have decided to give estimates of significance only when presenting my comparison in the final chapter of the best versions of each algorithm (see section 7.3).

1.3.5 Measuring computational complexity

In this dissertation, the worst-case space and time complexities of each algorithm will be calculated in the usual way and expressed using O-notation (Cormen et al., 1990, pp. 6–10, 23–41).

1.3.6 Measuring style dependence

The extent to which the performance of each algorithm depends on musical style is evaluated in this study by comparing the note accuracies that the algorithm achieves on the eight different composers in $\mathcal{C}$. Specifically, if we let $\mathcal{C}_x$ denote the subset of $\mathcal{C}$ containing movements by composer x, then let us define the *style dependence* of A over $\mathcal{C}$ to be the standard deviation of $\mathrm{NA}(A, \mathcal{C}_x)$ over the eight composers in the corpus. This value, which will be denoted by $\mathrm{SD}_{\mathrm{Sty}}(A, \mathcal{C})$, is given by

$$\mathrm{SD}_{\mathrm{Sty}}(A, \mathcal{C}) = \sqrt{\frac{\sum_{x \in X} \left(\mathrm{NA}(A, \mathcal{C}_x) - \overline{\mathrm{NA}(A, \mathcal{C}_x)} \right)^2}{|X| - 1}} \tag{1.4}$$

where X is the set of composers represented in the corpus and $\overline{\mathrm{NA}(A, \mathcal{C}_x)}$ is the mean value of $\mathrm{NA}(A, \mathcal{C}_x)$ over the eight different composers—that is,

$$\overline{\mathrm{NA}(A, \mathcal{C}_x)} = \frac{\sum_{x \in X} \mathrm{NA}(A, \mathcal{C}_x)}{|X|}. \tag{1.5}$$

In this study, $\mathrm{SD}_{\mathrm{Sty}}(A, \mathcal{C})$ and $\overline{\mathrm{NA}(A, \mathcal{C}_x)}$ will always be calculated with the note accuracies expressed as percentages. Recall from Table 1.4 that $\mathrm{NNOTES}(\mathcal{C}_x)$ is very nearly equal for all the composers. This means that $\overline{\mathrm{NA}(A, \mathcal{C}_x)}$ is always very nearly equal to $\mathrm{NA}(A, \mathcal{C})$. It also means that the values of $\mathrm{NA}(A, \mathcal{C}_x)$ for the different composers may reasonably be given equal weighting. A higher value of $\mathrm{SD}_{\mathrm{Sty}}(A, \mathcal{C})$ indicates that the values of $\mathrm{NA}(A, \mathcal{C}_x)$ are more widely spread, suggesting that the spelling accuracy of A is more heavily dependent on the style of the music being processed.

Note that, as mentioned above, the composer of a work is only one possible indicator of its "style". Other properties of a work that could have been used instead of (or in combination with) its composer as indicators of its style include its genre, form or instrumentation, where and when it was composed, its predominant rhythm, tempo, tonality or modality, and so on. Using composer as the single indicator of style therefore leads to only a rather crude performance metric for the evaluation criterion of style dependence. Nevertheless, this study is the first to make a serious attempt to provide a meaningful quantitative measure of style dependence by using a reasonable statistical measure and a corpus in which each "style" (in this case, composer) is represented by the same quantity of music (measured here in terms of the number of notes).

1.3.7 Measuring robustness to temporal deviations in the data

The OPNDV files in $\mathcal{C}$ were generated automatically from encodings of musical scores. Therefore, the onset times and durations of the notes in these encodings are strictly proportional to those in the notated scores which they represent. However, in an encoding which is generated from a performance on a MIDI instrument or transcribed from an audio file, the onsets and durations of the notes are typically not precisely proportional to their notated values. Also, a MIDI file of a passage derived by transcription from an audio file will typically have both some notes missing and other notes with incorrect pitches.

Unfortunately, "naturally noisy" encodings of the movements in the test corpus $\mathcal{C}$, generated

from performances or transcribed from audio files, were not available. I therefore automatically generated an "artificially noisy" version of the test corpus by introducing temporal deviations of the type typically found in MIDI files derived from human performances on MIDI instruments. Specifically, a 'noisy' version of $\mathcal{C}$, which I shall denote by $\mathcal{C}'$, was generated, by

1. assigning a 'natural' tempo to each OPNDV file in $\mathcal{C}$ and re-expressing the onset times and durations in milliseconds;

2. selecting, for each note, a value at random from the set $\{-50, -25, 0, 25, 50\}$ and adding this value to the onset of the note;

3. selecting, for each note, a value at random from the set $\{0.5, 0.75, 1.0, 1.25, 1.5\}$ and multiplying the duration of the note by this value.

This was intended to provide a *very* crude simulation of the spreading of chords and inaccuracies in synchronicity typical in performances (Gabrielsson, 1999, pp. 527–528). In practice, the 'performances' in $\mathcal{C}'$ were rather more error-ridden (at least in terms of temporal deviation) than would usually be expected in even an amateur human performance.

There was insufficient time to run every version of every algorithm on $\mathcal{C}'$. Therefore, only the best versions of each algorithm considered were run on this noisy corpus. For each algorithm, A, the proportional difference, $\Delta\mathrm{NER}_{\mathrm{Pr}}(\mathrm{NER}(A,\mathcal{C}),\mathrm{NER}(A,\mathcal{C}'))$ (see Eq. 1.3) was used to measure the effect that the noise had on the spelling accuracy of the algorithm. $\Delta\mathrm{NER}_{\mathrm{Pr}}(\mathrm{NER}(A,\mathcal{C}),\mathrm{NER}(A,\mathcal{C}'))$ gives the proportional increase in the note error rate caused by the introduction of noise into the data. A higher positive value of $\Delta\mathrm{NER}_{\mathrm{Pr}}(\mathrm{NER}(A,\mathcal{C}),\mathrm{NER}(A,\mathcal{C}'))$ indicates that the algorithm's spelling accuracy is more strongly reduced by the introduction of noise into the data. The results of running the best algorithms on $\mathcal{C}'$ are discussed in section 7.4 below.

1.4 Preliminary definitions

1.4.1 Introduction

In this section, I define various basic concepts, notation, terminology and functions that will be used throughout the remainder of this dissertation. I shall start by discussing ordered sets and strings. I shall then discuss the pseudocode that will be used to describe algorithms. I shall then define some basic mathematical functions. Finally, I shall define certain concepts, terminology and functions relating to the representation of pitch and pitch interval information.

1.4.2 Ordered sets and strings

It will be assumed here that the reader is familiar with the concepts and terminology of basic set theory (for a quick reminder, see Cormen et al., 1990, pp. 77–81).

The concept of an *ordered set* (also called a *sequence*, *list* or *array*) will be used often throughout this dissertation. An ordered set in which all the elements are numbers may be called a *vector*. The names of ordered sets will always be written using **bold-face** font. An ordered

set may be written out in full by listing its elements between angle brackets and separating the elements by commas. For example, the ordered set

$$\mathbf{A} = \langle 1, 2, 3, 4 \rangle$$

contains the natural numbers between 1 and 4, inclusive, each exactly once, sorted into increasing order. The *empty ordered set* is denoted by $\langle\,\rangle$. The ith element of an ordered set, $\mathbf{A}$, is denoted by $\mathbf{A}[i-1]$. Thus $\mathbf{A}[0]$ denotes the first element in an ordered set $\mathbf{A}$. This 'square bracket' notation can also be used to access elements of an ordered set when it is written out in full. For example, $\langle 1, 2, 3, 4 \rangle [2] = 3$, $\langle 1, 2, 3, 4 \rangle [0] = 1$ and so on. One fundamental difference between an ordered set and a set is that the elements in an ordered set are not necessarily distinct. The *length* of an ordered set, $\mathbf{A}$, is the number of elements in $\mathbf{A}$ and it is denoted by $|\mathbf{A}|$. Thus, if $\mathbf{A} = \langle 1, 2, 3, 4 \rangle$ then $|\mathbf{A}| = 4$. Note that $|\langle 1, 1, 2, 3 \rangle| = |\langle 1, 2, 3, 4 \rangle| = 4$. An ordered set whose length is 2 may be called an *ordered pair* and an ordered set of length 3 may be called an *ordered triple*. In general, an ordered set of length n may be called an *n-tuple*. Two ordered sets, $\mathbf{A}$ and $\mathbf{B}$, are defined to be *equal* if and only if

$$(|\mathbf{A}| = |\mathbf{B}|) \wedge (\mathbf{A}[i] = \mathbf{B}[i] \text{ for all } 0 \leq i < |\mathbf{A}|).$$

(The symbols $\wedge$, $\vee$ and $\veebar$ represent the logical connectives 'and', 'or', and 'xor' (i.e., 'exclusive or'), respectively). The square bracket notation for accessing elements of an ordered set can be extended logically for accessing elements of ordered sets of any dimensionality (e.g., ordered sets of ordered sets (2 dimensions), ordered sets of ordered sets of ordered sets (3 dimensions), and so on). Thus, if

$$\mathbf{A} = \langle \langle 1, 2, 3 \rangle, \langle 1, \langle 2, 3 \rangle \rangle, \langle 1, \langle 2, \langle 3 \rangle \rangle \rangle \rangle,$$

then $\mathbf{A}[0][1] = 2$, $\mathbf{A}[1][1] = \langle 2, 3 \rangle$ and $\mathbf{A}[2][1][1][0] = 3$. If

$$\mathbf{A} = \langle a_0, a_1, a_2, \ldots a_{n-1} \rangle$$

then $\mathbf{A}[i, j]$ denotes the ordered set $\langle a_i, a_{i+1}, \ldots a_{j-1} \rangle$. That is,

$$\mathbf{A}[i, j] = \langle \mathbf{A}[i], \mathbf{A}[i+1], \ldots \mathbf{A}[j-1] \rangle.$$

If $\mathbf{A}$ and $\mathbf{B}$ are ordered sets such that

$$\mathbf{A} = \langle a_0, a_1, \ldots a_{n-1} \rangle \text{ and } \mathbf{B} = \langle b_0, b_1, \ldots b_{m-1} \rangle,$$

then the *concatenation of* $\mathbf{B}$ *onto* $\mathbf{A}$, denoted by $\mathbf{A} \oplus \mathbf{B}$, is equal to

$$\mathbf{A} \oplus \mathbf{B} = \langle a_0, a_1, \ldots a_{n-1}, b_0, b_1, \ldots b_{m-1} \rangle.$$

That is, if $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$, then $|\mathbf{C}| = |\mathbf{A}| + |\mathbf{B}|$, $\mathbf{C}[i] = \mathbf{A}[i]$ for all $0 \leq i < |\mathbf{A}|$ and $\mathbf{C}[i] = \mathbf{B}[i - |\mathbf{A}|]$

for all $|\mathbf{A}| \leq i < |\mathbf{C}|$. If $\mathbf{A}_1, \mathbf{A}_2, \ldots \mathbf{A}_n$ are all ordered sets, then

$$\bigoplus_{i=1}^{n} \mathbf{A}_i = \mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \ldots \oplus \mathbf{A}_n.$$

An ordered set in which every element is a character may be called a *string*. When a string is written out in full, the characters may be written adjacent to each other between double quotes. Thus

$$\texttt{"abcdefg"} = \langle \texttt{`a'}, \texttt{`b'}, \texttt{`c'}, \texttt{`d'}, \texttt{`e'}, \texttt{`f'}, \texttt{`g'} \rangle .$$

When written out in full, strings and characters will be printed in teletype font and characters will be written between single quotes. The empty string (which is equal to the empty ordered set) may be written `""`. All notation relating to ordered sets may be used for strings. For example, $\texttt{"abcdefg"}[2] = \texttt{`c'}$, $\texttt{"abcdefg"}[1,4] = \texttt{"bcd"}$ and $\texttt{"abc"} \oplus \texttt{"def"} = \texttt{"abcdef"}$. If $\mathbf{Y}$ is an ordered set, then the function $\textsc{Pos}(x, \mathbf{Y})$ returns

- nil if x is not an element of $\mathbf{Y}$; and
- the least value of i for which $\mathbf{Y}[i] = x$ if x *is* an element of $\mathbf{Y}$.

1.4.3 Pseudocode conventions

The pseudocode used in this dissertation should be easy to read for anyone who has done some programming in a procedural language such as C, PASCAL or BASIC. Each algorithm is expressed as a function which begins with a header line in which the name of the function is printed flush-left in CapitalizedSmallCaps font (e.g. the word 'BitVector' on the first line of Figure 1.10 on page 36). The name of the function is followed on the same line by a list of parameters. For example, the function BitVector in Figure 1.10 takes two parameters, *NonNegInt* and *RequestedWidth*. The lines following the header line of a function define the instructions carried out by the function on its parameters. Each line is labelled with a number and then indented at least one tab stop.

The looping constructs, **while** and **for**, and the conditional constructs, **if** and **else**, have the same interpretation as in PASCAL. Block structure is indicated solely by indentation. This reduces clutter by dispensing with the need for reserved words like **do**, **begin**, **end** and **then**. It also means there is no need for punctuation such as semi-colons for terminating logical lines of code. Thus, the body of the **while** loop that begins on line 8 in Figure 1.10 consists of lines 9–12. This also applies to the **if**...**else** construct, so that, for example, if the condition in line 9 of Figure 1.12 on page 43 is satisfied, then lines 11 to 15 are skipped; and if the condition in line 9 is not satisfied and the condition in line 12 *is* satisfied, then lines 14 and 15 are skipped. If a logical line of pseudocode is too long to fit on the page, then it is split over two printed lines with the second printed line indented more than one tab position beyond the first printed line (see, for example, lines 16–20 in Figure 2.5 on page 59). If a logical pseudocode line is split over two or more lines, then the printed lines may be numbered separately (as in Figure 2.5) or as a single logical line (as in Figure 3.5 on page 98).

Everything occurring on a printed line after the symbol '▶' is a comment and is not executed (see, for example, lines 11 and 14 in Figure 2.7 on page 62 or lines 22–23 in Figure 3.12 on

```
CallingFunction
1   x ← 1
2   y ← 2x                          ▶ y becomes 2.
3   z ← CalledFunction(x)           ▶ z becomes 8.
4   return ⟨x, y, z⟩                ▶ Returns ⟨1, 2, 8⟩.

CalledFunction(x)                   ▶ Called with x = 1 by CallingFunction.
1   x ← 2x                          ▶ This x becomes 2.
2   y ← 2x                          ▶ This y becomes 4.
3   return (x × y)                  ▶ Returns 2 × 4 = 8.
```

Figure 1.7: Illustration of pass-by-value policy.

page 105). Comments are always written in `teletype` font.

The symbol '←' is used to denote assignment. Thus, the expression '$x \leftarrow y$' means that the value of the variable called x becomes equal to the value of the variable called y. Note that, after assignment, x does not refer to the same object as y. Multiple assignment on one line of pseudocode is also permitted. In this case, the assignments are executed right-to-left. Thus, $x \leftarrow y \leftarrow z$ means that the value of the object referred to by y becomes equal to the value of the object referred to by z and *then* the value of the object referred to by x becomes equal to the value of the object referred to by y. So, if $x = 1$, $y = 2$ and $z = 3$, $x \leftarrow y \leftarrow z$ sets the values of x and y to 3.

All variables and parameters in a function are local to the function. Variables are not declared before they are used. If a variable's value is an ordered set (but not a string), then it is printed in **bold-face**, otherwise it is printed in *italic*. Global variables will not be used. Parameters are passed to a function *by value*. This means the called function receives its own copies of the parameters, so that if the value of a parameter is changed, this change occurs only within the called function, not in the calling function (Cormen et al., 1990, p. 5). For example, in line 1 of the function CallingFunction in Figure 1.7, the value of the local variable x becomes 1. Then, in line 2, the value of the local variable y becomes $2 \times 1 = 2$. Then, in line 3, the function CalledFunction is called with x (which is equal to 1) as its argument. In line 1 of CalledFunction, the value of the local variable x is multiplied by 2 to give 2. However, this does not change the value of the variable x in the function CallingFunction which remains 1. Then, in line 2 of CalledFunction, the value of y is set to equal twice the value in x, that is, 4. However, this does not change the value of y in CallingFunction, which remains 2. In line 3 of CalledFunction, the value $x \times y = 2 \times 4 = 8$ is returned, and this value is assigned to the variable z in CallingFunction which therefore returns the ordered set $\langle 1, 2, 8 \rangle$.

As illustrated in Figure 1.7, if a function contains a line beginning with the word '**return**', then the function returns a new object with the same value as whatever is printed on this line following the word '**return**'. A **return** statement has the same meaning here as it does in C (Kernighan and Ritchie, 1988, p. 25–26). That is, execution returns to the calling function immediately after a **return** statement is executed.

Compound data are sometimes organised into various types of *record*. Each record of a particular type consists of a set of named *fields*, which are defined in the record definition. All records of a given type have the same fields, but the values stored in these fields may be

Student
NAME
YEAR
MODULELIST

Figure 1.8: The Student record type.

Module
NAME
MARK

Figure 1.9: The Module record type.

different for each record. For example, Figure 1.8 defines a record type that could be used to store information about a student. The first line of the definition gives the name of the record type, printed in sans-serif font, with a capitalised first letter. Each subsequent line of the definition gives the name of a field within the record. Thus, every record of the Student record type contains three fields, NAME, YEAR and MODULELIST. Also, let's suppose that the MODULELIST field of each Student type record is a list of Module type records, where the Module record type is defined in Figure 1.9. Each Module record contains two fields, NAME and MARK, and gives the mark that a particular student scores on a particular module. Whenever a new record type is defined, it is assumed that a new function is also defined, called MAKEXXXX, which simply returns a new record of type Xxxx in which all the fields are empty. For example, we could create a new Student type record as follows:

$S1 \leftarrow$ MAKESTUDENT

We could then use $S1$ to store information about a first year student called Alice who got 50% in Maths and 75% in Physics. To do this, we could use the following lines of pseudocode:

$S1$.NAME $\leftarrow$ `"Alice"`
$S1$.YEAR $\leftarrow 1$
$S1$.MODULELIST $\leftarrow \langle$MAKEMODULE, MAKEMODULE$\rangle$
$S1$.MODULELIST[0].NAME $\leftarrow$ `"Maths"`
$S1$.MODULELIST[0].MARK $\leftarrow 50$
$S1$.MODULELIST[1].NAME $\leftarrow$ `"Physics"`
$S1$.MODULELIST[1].MARK $\leftarrow 75$

Note that the expression *RecordVariable*.FIELD accesses the field called FIELD in the record variable called *RecordVariable*. Note also, that this type of expression can be used on either side of an assignment expression. Thus

```
BitVector(NonNegInt, RequestedWidth)
1   if NonNegInt ≠ 0
2       MinWidth ← 1 + ⌊log2(NonNegInt)⌋
3   else
4       MinWidth ← 1
5   ActualWidth ← Max({RequestedWidth, MinWidth})
6   BitVec ← ⟨ ⟩
7   j ← MinWidth − 1
8   while j ≥ 0
9       k ← ⌊2^−j NonNegInt⌋
10      BitVec ← BitVec ⊕ ⟨k⟩
11      NonNegInt ← NonNegInt − 2^j k
12      j ← j − 1
13  for i ← 1 to ActualWidth − MinWidth
14      BitVec ← ⟨0⟩ ⊕ BitVec
15  return BitVec
```

Figure 1.10: The BitVector algorithm.

$S1.\textsc{Year} \leftarrow 1$

sets the value of the Year field in the record variable called $S1$ to 1; and

$y \leftarrow S1.\textsc{Year}$

makes the value of the variable y equal to the value stored in the Year field in the record variable $S1$.

1.4.4 Some basic mathematical functions and algorithms

A few basic mathematical functions will be used often throughout this dissertation. These will now be defined.

The function $\textsc{Abs}(x)$ takes a real-number x as its argument and returns x if $x \geq 0$ and $-x$ if $x < 0$.

The *floor* of x, denoted by $\lfloor x \rfloor$, returns the largest integer less than or equal to x. The *ceiling* of x, denoted by $\lceil x \rceil$, returns the smallest integer greater than or equal to x.

The binary operation mod is defined as follows:

$$x \bmod y = x - y \lfloor x/y \rfloor .$$

The function BitVector, defined in Figure 1.10 takes two arguments: *NonNegInt*, which must be a non-negative integer; and *RequestedWidth*, which must be an integer greater than zero. It returns a *bit vector* (i.e., an ordered set in which each element is either a zero or a one), which represents the binary number that is equivalent to *NonNegInt*. If the number of digits in the binary representation of *NonNegInt* is less than *RequestedWidth*, then the bit vector is padded

to the left with zeros until its length is *RequestedWidth*. Otherwise, the representation is left unpadded. For example, BitVector$(11, 3) = \langle 1, 0, 1, 1\rangle$ and BitVector$(11, 6) = \langle 0, 0, 1, 0, 1, 1\rangle$.

1.4.5 Pitch and pitch interval representations

In this dissertation, various different ways are used to represent various different types of pitch and pitch interval information. In this section, I shall introduce certain concepts relating to pitch and pitch interval representation that will be used without further comment throughout the remainder of this dissertation.

An object may be called a *pitch letter name* if and only if it is a member of the set

$$\{\texttt{"A"}, \texttt{"B"}, \texttt{"C"}, \texttt{"D"}, \texttt{"E"}, \texttt{"F"}, \texttt{"G"}, \texttt{"a"}, \texttt{"b"}, \texttt{"c"}, \texttt{"d"}, \texttt{"e"}, \texttt{"f"}, \texttt{"g"}\}.$$

An object may be called an *accidental* if and only if it is a string, A, which satisfies one of the following conditions:

1. $A = \texttt{""}$;
2. $A \in \{\texttt{"n"}, \texttt{"N"}\}$;
3. $|A| \geq 1 \wedge A[i] \in \{\texttt{'b'}, \texttt{'B'}\}$ for all $0 \leq i < |A|$;
4. $|A| \geq 1 \wedge A[i] \in \{\texttt{'f'}, \texttt{'F'}\}$ for all $0 \leq i < |A|$;
5. $|A| \geq 1 \wedge A[i] \in \{\texttt{'s'}, \texttt{'S'}\}$ for all $0 \leq i < |A|$; or
6. $|A| \geq 1 \wedge A[i] = \texttt{'\#'}$ for all $0 \leq i < |A|$.

For example, `"bbb"`, `"###"`, `"fFf"` and `"SSS"` are all valid accidentals.

If x is a a number representable as a finite decimal, then Num2Str(x) is a function which returns a string representation of the number x as a decimal. For example, Num2Str$(-5) =$ `"-5"`, Num2Str$(-5.32) =$ `"-5.32"` and Num2Str$(-0.32) =$ `"-0.32"`. The function Str2Num(s) is the inverse of Num2Str(x)—it takes a string representation s of a numerical value and returns the numerical value represented by s. For example, Str2Num(`"-5"`) $= -5$, Str2Num(`"-5.32"`) $= -5.32$ and Str2Num(`"-0.32"`) $= -0.32$.

An object, *PN*, may be called a *pitch name* if and only if *PN* is a string of the form $\ell \oplus a \oplus o$, where ℓ is a pitch letter name, a is an accidental and there exists some integer, i, called the *octave number* of *PN*, such that $o =$ Num2Str(i). The pitch name of a note in a score of a passage of western tonal music can be derived from the note in the usual way, by considering the position of the note-head of the note on the staff, the key signature and clef in operation where the note occurs, whether or not the music is for a transposing instrument and the presence of any explicit accidentals that apply to the note. Thus, the pitch name of middle C is `"Cn4"` (which is equivalent to `"cN4"`, `"cn4"`, `"c4"` and `"C4"`). The pitch name of 'concert A' is `"An4"`. The lowest and highest notes on a normal piano keyboard are `"An0"` and `"Cn8"`. The octave number of a pitch name is the same as the highest `"Cn"` not above it *on the staff*, which is not necessarily the highest `"Cn"` not above it *in frequency*. For example, in an equal-tempered tuning system, `"Bs3"` has the same frequency as `"Cn4"`. This system of pitch naming is equivalent to that proposed by

the Acoustical Society of America (Young, 1939) and accepted by the US Standards Association (Backus, 1977, p. 154; Brinkman, 1990, p. 122).

If $PN = \ell \oplus a \oplus o$ is a pitch name such that ℓ is a pitch letter name, a is an accidental and o represents the octave number of *PN*, then $\ell \oplus a$ is called the *pitch name class* of *PN*. In general, a string, *PNC*, may be called a *pitch name class* if and only if *PNC* has the form $\ell \oplus a$ where ℓ is a pitch letter name and a is an accidental. Thus, the pitch name class of `"Cn4"` is `"Cn"`. A pitch name class, *PNC*, represents the set of pitch names that have the same pitch letter name and accidental as *PNC*.

The *chromatic pitch* of a pitch name, *PN*, is an integer that represents the key on a normal piano keyboard which would have to be pressed in order to produce the pitch represented by the pitch name *PN*. Raising the pitch by one semitone increases the chromatic pitch by 1 and lowering the pitch by one semitone decreases the chromatic pitch by 1. In this study, the chromatic pitch of `"An0"` is defined to be 0, the chromatic pitch of `"Bf0"` is 1, the chromatic pitch of `"Gs0"` is -1, the chromatic pitch of middle C is 39, and so on. Enharmonically equivalent pitch names therefore have the same chromatic pitch. For example, `"Cn4"`, `"Bs3"`, `"Asss3"` and `"Dff4"` all have a chromatic pitch of 39. Note that it is possible for a note to have a higher position on a staff but a lower chromatic pitch than another note. For example, the chromatic pitch of `"Cn4"` is 39 but the chromatic pitch of `"Bss3"` is 40. The chromatic pitch of a note can be understood as being a representation of the log frequency of the fundamental of a note in an equal-tempered tuning system. If p_c is the chromatic pitch of a note, then the *continuous pitch code* of the note in Brinkman's (1990, p. 122) system is $p_c + 9$ and the *MIDI note number* of the note is $p_c + 21$ (The MIDI Manufacturers' Association, 1996, p. 10).

If $p_{c,1}$ and $p_{c,2}$ are two chromatic pitches, then the *chromatic pitch interval* from $p_{c,1}$ to $p_{c,2}$ is defined to be $p_{c,2} - p_{c,1}$.

The *morphetic pitch* of a note is an integer that is determined by

1. the vertical position of the note-head on the staff,
2. the clef in operation on the staff at the location of the note and
3. the transposition of the staff.

The morphetic pitch of a note is independent of the sounding pitch of a note and the chromatic pitch of the note. Moving a note one step up on the staff (while keeping the clef constant) increases its morphetic pitch by 1, whilst moving it down one step decreases its morphetic pitch by 1. The morphetic pitch of `"An0"` is defined to be 0. The morphetic pitch of a pitch name is independent of its accidental. Thus `"An0"`, `"Afff0"` and `"As0"` all have a morphetic pitch of 0. The morphetic pitch of middle C is 23. Note that it is possible for a note to have a higher frequency but lower morphetic pitch than another note. For example, `"Bss3"` has a lower morphetic pitch (22) but a higher frequency than `"Cn4"` (whose morphetic pitch is 23). If p_m is the morphetic pitch of a note, then the *continuous name code* of the note in Brinkman's (1990, p. 126) system is $p_m + 5$ and the *diatone* of the note in Regener's (1973, p. 32) system is $p_m - 17$.

If $p_{m,1}$ and $p_{m,2}$ are two morphetic pitches, then the *morphetic pitch interval* from $p_{m,1}$ to $p_{m,2}$ is defined to be $p_{m,2} - p_{m,1}$.

The *chromamorphetic pitch* of a note is simply an ordered pair, $\langle p_{\mathrm{c}}, p_{\mathrm{m}} \rangle$, in which p_{c} is the note's chromatic pitch and p_{m} is the note's morphetic pitch. If the chromamorphetic pitch of a note is $\langle p_{\mathrm{c}}, p_{\mathrm{m}} \rangle$, then its *continuous binomial representation* in Brinkman's (1990, pp. 133–135) system is $\langle p_{\mathrm{c}} + 9, p_{\mathrm{m}} + 5 \rangle$.

If $\langle p_{\mathrm{c},1}, p_{\mathrm{m},1} \rangle$ and $\langle p_{\mathrm{c},2}, p_{\mathrm{m},2} \rangle$ are two chromamorphetic pitches, then the *chromamorphetic pitch interval* from $\langle p_{\mathrm{c},1}, p_{\mathrm{m},1} \rangle$ to $\langle p_{\mathrm{c},2}, p_{\mathrm{m},2} \rangle$ is $\langle p_{\mathrm{c},2} - p_{\mathrm{c},1}, p_{\mathrm{m},2} - p_{\mathrm{m},1} \rangle$.

Chromamorphetic pitches and intervals behave in exactly the same way as (i.e., they are strictly isomorphic to) pitch names and pitch interval names, respectively.

The *chroma* of a note is the least non-negative residue, modulo 12, of its chromatic pitch. That is, if p_{c} is the chromatic pitch of a note, then its chroma is p_{c} mod 12. For example, the chroma of `"Cn4"` is 3. Therefore, if the chroma of a note is c, then its *pitch class* (as this term is used by Babbitt (1965), Forte (1973), Rahn (1980), Morris (1987), Brinkman (1990, p. 119) and many others) is $(c - 3)$ mod 12.

If c_1 and c_2 are two chromas, then the *chroma interval* (or *pitch class interval*) from c_1 to c_2 is defined to be $(c_2 - c_1)$ mod 12.

The *morph* of a note is the least non-negative residue, modulo 7, of its morphetic pitch. That is, if p_{m} is the morphetic pitch of a note, then the morph of the note is p_{m} mod 7. The morph of a note is simply a numerical representation of its pitch letter name, with `"A"` corresponding to a morph of 0, `"B"` corresponding to a morph of 1, `"C"` corresponding to a morph of 2 and so on. If the morph of a note is m, then its *name class* in Brinkman's (1990, p. 124) theory is $(m - 2)$ mod 7 and its *diatonic note class* in Regener's (1973, p. 34) theory is $(m + 2)$ mod 7.

If m_1 and m_2 are two morphs, then the *morph interval* from m_1 to m_2 is defined to be $(m_2 - m_1)$ mod 7.

If p_{c} is the chromatic pitch of a note, then its *chromatic octave* is $\lfloor p_{\mathrm{c}}/12 \rfloor$. If p_{m} is the morphetic pitch of a note, then its *morphetic octave* is $\lfloor p_{\mathrm{m}}/7 \rfloor$. Note that the morphetic and chromatic octaves of a note may not always be equal. For example, the chromatic octave of `"Af3"` is 2, but its morphetic octave is 3. If the morphetic octave of a note is o_{m}, then the octave number of its pitch name is

1. o_{m} if the morph of the note is less than 2; and
2. $o_{\mathrm{m}} + 1$ otherwise.

Note that one flaw in Brinkman's (1990) system of pitch representation is that he does not distinguish between chromatic and morphetic octave.

In this study, *pitch interval names*, such as "rising major third", "falling perfect fifth" and "perfect prime", will be represented by strings of the form $d \oplus q \oplus s$, where:

1. d, the *direction* of the pitch interval name, is a member of the set {`"r"`, `"f"`, `""`};
2. q, the *quality* of the pitch interval name, is either a member of the set {`"ma"`, `"mi"`, `"p"`} or a string of any length in which every character is an `'a'` or a string of any length in which every character is a `'d'`; and
3. s, the *diatonic size* of the pitch interval name, is a string representation of an integer greater than zero.

Pitch name class	···	F♭	C♭	G♭	D♭	A♭	E♭	B♭	F	C	G	D	A	E	B	F♯	C♯	G♯	D♯	A♯	E♯	B♯	···
LOF position, quint (Regener)	···	−7	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	···
TPC (Temperley)	···	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	···
Sharpness (Longuet-Higgins)	···	−8	−7	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	···

Figure 1.11: Line of fifths, showing the line-of-fifths position, Regener's quint, Temperley's tonal pitch class and Longuet-Higgins's sharpness for each pitch name class.

If the direction of a pitch interval name is `"r"`, this indicates that the pitch interval name is rising; if it is `"f"`, the pitch interval name is falling. Primes can be neither rising nor falling, so their direction is `""`. Note that the direction of a pitch interval name is

1. rising (`"r"`) if its morphetic pitch interval is positive;
2. falling (`"f"`) if its morphetic pitch interval is negative; and
3. absent (`""`) if its morphetic pitch interval is zero.

The direction of a pitch interval name is therefore independent of its chromatic pitch interval. Thus, the chromatic pitch interval of a rising pitch interval name may be negative. For example, a rising triply-diminished second has a chromatic pitch interval of -1 but a morphetic pitch interval of 1.

If an interval is major, then its quality is `"ma"`; if it is minor, its quality is `"mi"`; if it is perfect, its quality is `"p"`; if it is augmented, its quality is `"a"`; and if it is diminished, its quality is `"d"`. The quality of an n-tuply augmented interval is a string of length n in which every character is an `'a'`. The quality of an n-tuply diminished interval is a string of length n in which every character is a `'d'`.

The diatonic size of a pitch interval name is `"1"` for a prime, `"2"` for a second, `"3"` for a third, `"4"` for a fourth and so on. The diatonic size of a pitch interval name must be a string representing any integer greater than 0.

As examples, the pitch interval name `"raaa3"` represents a rising triply-augmented third (e.g., the interval from `"Cn4"` to `"Esss4"`); the pitch interval name `"dd1"` represents a doubly-diminished prime (e.g., the interval from `"Cn4"` to `"Cff4"`); and the pitch interval name `"fd10"` represents a falling diminished tenth (e.g., the interval from `"Cn4"` to `"As2"`).

Two pitch interval names are members of the same *pitch interval name class* if and only if the difference between them is an integer number of rising or falling perfect octaves. In this study, a pitch interval name class is represented by writing the rising pitch interval name with the lowest diatonic size in the class between square brackets. For example, the pitch interval name class [`"rmi3"`] contains, amongst others, the pitch interval names `"rmi3"`, `"rmi10"`, `"fma6"` and `"fma13"`. A pitch interval name class containing a prime is represented by writing that prime interval between square brackets (e.g., [`"a1"`] contains `"a1"`, `"ra8"`, `"fd8"` and so on).

Figure 1.11 shows the *line of fifths*. The line of fifths is a 1-dimensional pitch name class space in which each pitch name class is adjacent to the pitch name classes that are a perfect fifth above and below it. The *line-of-fifths position* of a note is an integer that indicates the position on the line of fifths of the pitch name class of the note. `"Fn"` is defined to have a line-of-fifths position of 0. Transposing a pitch name class by a rising perfect fifth increases its

line-of-fifths position by 1 and transposing it by a rising perfect fourth decreases its line-of-fifths position by 1. If ℓ is the line-of-fifths position of a note, then its *quint* in Regener's (1973, p. 33) theory is ℓ, its *tonal pitch class* in Temperley's (2001, p. 118) theory is $\ell + 1$ and its *sharpness* in Longuet-Higgins's (1987a, p. 111) theory is $\ell - 1$.

If ℓ_1 and ℓ_2 are two line-of-fifths positions, then the *line-of-fifths interval* or *line-of-fifths displacement* from ℓ_1 to ℓ_2 is $\ell_2 - \ell_1$. If $\delta\ell$ is the line-of-fifths displacement of a pitch interval name, then the *line-of-fifths displacement class* is $\delta\ell \bmod 7$ and the *line-of-fifths displacement cycle* is $\lfloor \delta\ell/7 \rfloor$. The line-of-fifths displacement class, $\delta\ell \bmod 7$, is related to the diatonic size, s, of a pitch interval name class by the equation

$$\delta\ell \bmod 7 = (2(s-1)) \bmod 7.$$

The line-of-fifths displacement cycle of a pitch interval name class is directly related to its quality. Specifically,

1. if the quality is n-tuply augmented and the diatonic size is not 4, then the line-of-fifths displacement cycle is n;

2. if the quality is n-tuply augmented and the diatonic size is 4, then the line-of-fifths displacement cycle is $n - 1$;

3. if the quality is n-tuply diminished and the diatonic size is 1 or 5, then the line-of-fifths displacement cycle is $-n$;

4. if the quality is n-tuply diminished and the diatonic size is neither 1 nor 5, then the line-of-fifths displacement cycle is $-n - 1$.

If ℓ is the line-of-fifths position of a pitch name, then the *morphetic line-of-fifths class* of the pitch name is $\ell \bmod 7$, the *morphetic line-of-fifths cycle* of the pitch name is $\lfloor \ell/7 \rfloor$, the *chromatic line-of-fifths class* of the pitch name is $\ell \bmod 12$ and the *chromatic line-of-fifths cycle* of the pitch name is $\lfloor \ell/12 \rfloor$. Two pitch names have the same morph if and only if they have the same morphetic line-of-fifths class. Two pitch names have the same chroma if and only if they have the same chromatic line-of-fifths class. Two pitch names have the same accidental if and only if they have the same morphetic line-of-fifths cycle.

The *displacement* of a note (not to be confused with the line-of-fifths displacement of a pitch interval) is a numerical representation of its accidental. The displacement of a note is zero if the accidental is natural, 1 if it is sharp, -1 if it is flat, 2 if it is double-sharp, -2 if it is double-flat, and so on.

The *undisplaced pitch name* of a note is the natural pitch name with the same morphetic pitch as the note. The *undisplaced chromatic pitch, MIDI note number, chroma, pitch name class, etc.* of a note are, respectively, the chromatic pitch, MIDI note number, chroma, pitch name class, etc. of the undisplaced pitch name of the note.

The *undisplaced pitch interval name* of a pitch interval name, *PIN*, is the major or perfect pitch interval name that has the same direction and diatonic size as *PIN*. The *undisplaced quality, chromatic pitch interval, chroma interval, pitch interval name class, etc.* of a pitch

interval name, *PIN*, are, respectively, the quality, chromatic pitch interval, chroma interval, pitch interval name class, etc. of the undisplaced pitch interval name.

The *interval displacement* of a pitch interval name, *PIN*, (on an analogy with the displacement of a pitch name) is an integer that indicates the difference between the quality of *PIN* and the undisplaced quality of *PIN* (just as the displacement of a pitch name indicates the difference between the accidental of the pitch name and natural). Specifically, if the undisplaced quality is `"ma"`, then the interval displacement is:

- 0, if the quality of *PIN* is `"ma"`;
- -1, if the quality of *PIN* is `"mi"`;
- $-n-1$, if the quality of *PIN* is n-tuply diminished; and
- n if the quality of *PIN* is n-tuply augmented.

If the undisplaced quality is `"p"`, then the interval displacement is:

- 0, if the quality of *PIN* is `"p"`;
- n, if the quality of *PIN* is n-tuply augmented; and
- $-n$ if the quality of *PIN* is n-tuply diminished.

1.4.6 Algorithms for converting between different pitch and pitch interval representations

I shall now present a selection of algorithms for converting between different types of pitch and pitch interval representations. The following is not intended to constitute a complete and exhaustive conversion system. Only those functions that are used in this study will be presented.

The chromamorphetic pitch can be derived from the pitch name of a note using the function PN2P, defined in Figure 1.12. PN2P has a single parameter, *PN*, which must be a valid pitch name. In lines 1–6 of PN2P, *PN* is parsed into its letter name, accidental and octave number which are stored in the variables *LetterName*, *Accidental* and *ASAOctaveNumber*, respectively. In line 7, the morph is computed directly from the pitch letter name using the function UpCase(x), which has a single parameter, x, which must be either a string or a character. If x is a string, then UpCase(x) returns the string that results when every alphabetic character in x is replaced with its upper case equivalent and the other characters are left unchanged. If x is an alphabetic character, then UpCase(x) returns the character that is the upper case version of x. If x is a non-alphabetic character, then UpCase(x) returns x. Then, in line 8, the undisplaced chroma (i.e., the chroma of the natural pitch name with the same letter name as *PN*) is computed from the morph. In lines 9–15, the displacement is computed from the accidental of *PN*. Then, in lines 16–19, the morphetic octave is derived from the octave number and morph of *PN*. In line 20, the chromatic pitch is computed from the morphetic octave (which is the same as the chromatic octave of the undisplaced chromatic pitch), the undisplaced chroma and the displacement. In line 21, the morphetic pitch is derived from the morphetic octave and morph.

```
PN2P(PN)
1   LetterName ← PN[0]
2   OctavePos ← 1
3   while PN[OctavePos] ∉ {'-', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}
4       OctavePos ← OctavePos + 1
5   Accidental ← PN[1, OctavePos]
6   ASAOctaveNumber ← Str2Num(PN[OctavePos, |PN|])
7   Morph ← Pos(UpCase(LetterName), {'A', 'B', 'C', 'D', 'E', 'F', 'G'})
8   UndisplacedChroma ← ⟨0, 2, 3, 5, 7, 8, 10⟩ [Morph]
9   if (Accidental = "") ∨ (UpCase(Accidental[0]) = 'N')
10      Displacement ← 0
11  else
12      if UpCase(Accidental[0]) ∈ {'F', 'B'}
13          Displacement ← −|Accidental|
14      else
15          Displacement ← |Accidental|
16  if Morph < 2
17      MorpheticOctave ← ASAOctaveNumber
18  else
19      MorpheticOctave ← ASAOctaveNumber − 1
20  ChromaticPitch ← Displacement + UndisplacedChroma + (12 × MorpheticOctave)
21  MorpheticPitch ← Morph + (7 × MorpheticOctave)
22  return ⟨ChromaticPitch, MorpheticPitch⟩
```

Figure 1.12: The PN2P algorithm.

```
P2PN(CMP)
1   Morph ← CMP[1] mod 7
2   LetterName ← ⟨"A", "B", "C", "D", "E", "F", "G"⟩ [Morph]
3   UndisplacedChroma ← ⟨0, 2, 3, 5, 7, 8, 10⟩ [Morph]
4   Displacement ← CMP[0] − 12 ⌊CMP[1]/7⌋ − UndisplacedChroma
5   Accidental ← ""
6   if Displacement ≠ 0
7       if Displacement < 0
8           AccidentalChar ← "f"
9       else
10          AccidentalChar ← "s"
11      for i ← 0 to Abs(Displacement) − 1
12          Accidental ← Accidental ⊕ AccidentalChar
13  else
14      Accidental ← "n"
15  ASAOctaveNumber ← ⌊CMP[1]/7⌋
16  if Morph > 1
17      ASAOctaveNumber ← ASAOctaveNumber + 1
18  return LetterName ⊕ Accidental ⊕ Num2Str(ASAOctaveNumber)
```

Figure 1.13: The P2PN algorithm.

```
PN2PNC(PN)
1   i ← |PN| − 1
2   while PN[i] ∈ {'-', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}
3       i ← i − 1
4   return PN[0, i + 1]
```

Figure 1.14: The PN2PNC algorithm.

```
PNC2LOF(PNC)
1   MorpheticLOFClass ← Pos(UpCase(PNC[0]), "FCGDAEB")
2   if (|PNC| = 1) ∨ (UpCase(PNC[1]) = 'N')
3       MorpheticLOFCycle ← 0
4   else
5       if UpCase(PNC[1]) ∈ {'S', '#'}
6           MorpheticLOFCycle ← |PNC| − 1
7       else
8           if UpCase(PNC[1]) ∈ {'B', 'F'}
9               MorpheticLOFCycle ← 1 − |PNC|
10  return MorpheticLOFClass + 7MorpheticLOFCycle
```

Figure 1.15: The PNC2LOF algorithm.

Having derived the chromatic pitch and morphetic pitch, the chromamorphetic pitch can be returned in line 22.

The pitch name that corresponds to a chromamorphetic pitch, *CMP*, can be determined using the function P2PN, defined in Figure 1.13. This function has one parameter, *CMP*, which must be a chromamorphetic pitch. In this function, the morph is first determined from the morphetic pitch in line 1. Then the letter name of the pitch name is derived directly from the morph in line 2. The undisplaced chroma can then be computed from the morph in line 3. The chromatic pitch, morphetic pitch and undisplaced chroma are then used in line 4 to compute the displacement which is subsequently used in lines 6–14 to compute the pitch name accidental. Finally, the pitch name octave number is computed from the morph and morphetic pitch in lines 15–17. The resulting pitch name is then constructed in line 18 by concatenating the letter name, accidental and a string representation of the octave number.

The pitch name class of a note can be derived from its pitch name using the trivial function, PN2PNC, defined in Figure 1.14.

The line-of-fifths position of a pitch name class can be computed using the function PNC2LOF, defined in Figure 1.15. PNC2LOF has a single parameter, *PNC*, which must be a valid pitch name class. In line 1 of PNC2LOF, the morphetic line-of-fifths class is computed directly from the pitch letter name of *PNC*. Then the morphetic line-of-fifths cycle is computed from the accidental of *PNC* (lines 2–9). The line-of-fifths position can then be computed from the morphetic line-of-fifths class and cycle (line 10).

The pitch name of a note can be computed from its pitch name class and MIDI note number using the function PNCMIDI2PN, defined in Figure 1.16. This function takes two arguments: a pitch name class, *PNC*; and a MIDI note number, *MIDINoteNumber*. In lines 1–4 of PNCMIDI2PN, the accidental of the pitch name class *PNC* is isolated and stored in the vari-

```
PNCMIDI2PN(PNC, MIDINoteNumber)
1    if |PNC| = 1
2        Accidental ← ""
3    else
4        Accidental ← PNC[1, |PNC|]
5    if (Accidental = "") ∨ (Accidental[0] ∈ {'n', 'N'})
6        Displacement ← 0
7    else
8        if Accidental[0] ∈ {'s', 'S', '#'}
9            Displacement ← |Accidental|
10       else
11           Displacement ← −|Accidental|
12   UndisplacedMIDINoteNumber ← MIDINoteNumber − Displacement
13   ASAOctaveNumber ← ⌊UndisplacedMIDINoteNumber/12⌋ − 1
14   return PNC ⊕ Num2Str(ASAOctaveNumber)
```

Figure 1.16: The PNCMIDI2PN algorithm.

```
TPCMIDI2PN(TPC, MIDINoteNumber)
1    ChromaticPitch ← MIDINoteNumber − 21
2    Morph ← (4TPC + 1) mod 7
3    Displacement ← ⌊(TPC − 1)/7⌋
4    UndispChromPitch ← ChromaticPitch − Displacement
5    MorpheticOctave ← ⌊UndispChromPitch/12⌋
6    MorpheticPitch ← Morph + 7MorpheticOctave
7    return P2PN(⟨ChromaticPitch, MorpheticPitch⟩)
```

Figure 1.17: The TPCMIDI2PN algorithm.

able *Accidental*. Then, in lines 5–11, the displacement of the pitch name class *PNC* is computed and stored in the variable *Displacement*. In line 12, the displacement is subtracted from the MIDI note number to give the undisplaced MIDI note number. For natural pitch names, the chromatic and morphetic octaves are equal. Therefore the pitch name octave number can be computed directly from the undisplaced MIDI note number in line 13 of PNCMIDI2PN. This octave number can then be represented as a string and concatenated onto the pitch name class *PNC* to generate the full pitch name.

The pitch name of a note can be determined from its MIDI note number and its tonal pitch class (Temperley, 2001, p. 118) using the function TPCMIDI2PN which is defined in Figure 1.17. TPCMIDI2PN takes two parameters, *TPC*, which must be a tonal pitch class; and *MIDINoteNumber*, which must be a MIDI note number. The basic strategy used in TPCMIDI2PN is to compute the appropriate chromamorphetic pitch and then use the function P2PN, defined in Figure 1.13, to compute the pitch name from this chromamorphetic pitch. In order to compute the chromamorphetic pitch, the chromatic pitch and morphetic pitch must be computed. The chromatic pitch can be computed easily by subtracting 21 from *MIDINoteNumber*. This is done in line 1 of TPCMIDI2PN. Computing the morphetic pitch is slightly more complicated. First, the morph and displacement are computed directly from *TPC* in lines 2 and 3 of TPCMIDI2PN. Then the undisplaced chromatic pitch is computed in line 4 by subtracting the displacement from the chromatic pitch. The morphetic octave can then be

```
LOFCP2PN(LOF, CP)
1    return TPCMIDI2PN(1 + LOF, CP + 21)
```

Figure 1.18: The LOFCP2PN algorithm.

computed in line 5 since this is the same as the chromatic octave of the undisplaced chromatic pitch. The morph and morphetic octave can then be combined to give the morphetic pitch in line 6.

The pitch name of a note can be computed from its line-of-fifths position and its chromatic pitch using the function LOFCP2PN, defined in Figure 1.18. This function has two parameters: *LOF*, which must be a line-of-fifths position; and *CP*, which must be a chromatic pitch. The MIDI note number is derived by adding 21 to *CP* and the tonal pitch class is derived by adding 1 to *LOF*. The MIDI note number and tonal pitch class are then given as arguments to the function TPCMIDI2PN (defined in Figure 1.17) which computes the pitch name.

The function PIN2PI, defined in Figure 1.19, takes one argument, *PIN*, which must be a valid pitch interval name, and computes the chromamorphetic pitch interval of *PIN*. In lines 1–13, the pitch interval name, *PIN*, is parsed into its component direction, quality and diatonic size, which are stored in the variables *DirStr*, *Quality* and *DiatSize*, respectively. The diatonic size is derived from its string representation in line 13 using the function Str2Num(s). In lines 14–16, the morphetic pitch interval of *PIN* is computed directly from the diatonic size of *PIN*. Then, in lines 17–43, the chromatic pitch interval of *PIN* is computed. The first step in computing this chromatic pitch interval is to find the absolute undisplaced chromatic pitch interval (lines 17–19) of *PIN*, which is the absolute chromatic pitch interval of the perfect or major pitch interval name that has the same diatonic size as *PIN*. The next step is to find the undisplaced quality of *PIN*, which is the quality of the major or perfect interval that has the same diatonic size as *PIN* (line 20). The chromatic pitch interval of *PIN* is then found by adding the absolute interval displacement of *PIN* to its undisplaced chromatic pitch interval and adjusting the sign of the chromatic pitch interval appropriately, depending on whether *PIN* is rising or falling (lines 40–43). The interval displacement of *PIN* is computed in lines 21–39 of PIN2PI.

If *CMPI* is a chromamorphetic pitch interval, then the function PI2PIN, defined in Figure 1.20, can be used to compute the pitch interval name that corresponds to *CMPI*. PI2PIN has one parameter, *CMPI*, which must be a valid chromamorphetic pitch interval. In lines 1–2 of PI2PIN, *CMPI* is parsed into its component morphetic and chromatic pitch intervals which are stored in the variables *MorpheticPitchInt* and *ChromaticPitchInt*, respectively. *MorpheticPitchInt* is then used in lines 3–9 to determine the direction of the pitch interval name: if *MorpheticPitchInt* is 0, then the pitch interval name is a prime and its direction is `""`; if *MorpheticPitchInt* is positive, then the direction of the pitch interval name is `"r"`; and if *MorpheticPitchInt* is negative, the direction is `"f"`. In line 10, the diatonic size is computed directly from *MorpheticPitchInt* and stored in the variable *DiatSize*. The function Num2Str is then used in line 11 to obtain the string representation of *DiatSize* which is stored in the variable *DiatSizeStr*. The absolute morph interval is computed from *MorpheticPitchInt* in line 12 and

```
PIN2PI(PIN)
1   DirStr ← PIN[0, 1]
2   if DirStr ∉ {"f", "r"}
3       DirStr ← ""
4   Quality ← ""
5   if DirStr = ""
6       i ← 0
7   else
8       i ← 1
9   while (i < |PIN|) ∧ (PIN[i] ∉ {'1', '2', '3', '4', '5', '6', '7', '8', '9'})
10      Quality ← Quality ⊕ PIN[i, i + 1]
11      i ← i + 1
12  DiatSizeStr ← PIN[i, |PIN|]
13  DiatSize ← Str2Num(DiatSizeStr)
14  MorpheticPitchInt ← DiatSize − 1
15  if DirStr = "f"
16      MorpheticPitchInt ← −MorpheticPitchInt
17  AbsMorphInt ← Abs(MorpheticPitchInt) mod 7
18  UndispChromaInt ← ⟨0, 2, 4, 5, 7, 9, 11⟩ [AbsMorphInt]
19  UndispCPInt ← UndispChromaInt + (12 ⌊Abs(MorpheticPitchInt)/7⌋)
20  UndispQual ← ⟨"p", "ma", "ma", "p", "p", "ma", "ma"⟩ [AbsMorphInt]
21  if UndispQual = "p"
22      if Quality = "p"
23          IntDisp ← 0
24      else
25          if Quality[0] = 'a'
26              IntDisp ← |Quality|
27          else
28              IntDisp ← −|Quality|
29  else
30      if Quality = "ma"
31          IntDisp ← 0
32      else
33          if Quality = "mi"
34              IntDisp ← −1
35          else
36              if Quality[0] = 'a'
37                  IntDisp ← |Quality|
38              else
39                  IntDisp ← −|Quality| − 1
40  if MorpheticPitchInt < 0
41      ChromaticPitchInt ← −(IntDisp + UndispCPInt)
42  else
43      ChromaticPitchInt ← IntDisp + UndispCPInt
44  return ⟨ChromaticPitchInt, MorpheticPitchInt⟩
```

Figure 1.19: The PIN2PI function.

```
PI2PIN(CMPI)
1   MorpheticPitchInt ← CMPI[1]
2   ChromaticPitchInt ← CMPI[0]
3   if MorpheticPitchInt = 0
4       DirStr ← ""
5   else
6       if MorpheticPitchInt > 0
7           DirStr ← "r"
8       else
9           DirStr ← "f"
10  DiatSize ← 1 + ABS(MorpheticPitchInt)
11  DiatSizeStr ← NUM2STR(DiatSize)
12  AbsMorphInt ← ABS(MorpheticPitchInt) mod 7
13  UndispChromaInt ← ⟨0, 2, 4, 5, 7, 9, 11⟩ [AbsMorphInt]
14  UndispQual ← ⟨"p", "ma", "ma", "p", "p", "ma", "ma"⟩ [AbsMorphInt]
15  if MorpheticPitchInt ≥ 0
16      IntDisp ← ChromaticPitchInt − 12 ⌊MorpheticPitchInt/7⌋ − UndispChromaInt
17  else
18      IntDisp ← −ChromaticPitchInt − 12 ⌊−MorpheticPitchInt/7⌋ − UndispChromaInt
19  Quality ← ""
20  if UndispQual = "p"
21      if IntDisp = 0
22          Quality ← "p"
23      else
24          if IntDisp > 0
25              QualChar ← "a"
26          else
27              QualChar ← "d"
28          for i ← 1 to ABS(IntDisp)
29              Quality ← Quality ⊕ QualChar
30  else
31      if IntDisp = 0
32          Quality ← "ma"
33      else
34          if IntDisp = −1
35              Quality ← "mi"
36          else
37              if IntDisp < −1
38                  for i ← 1 to ABS(IntDisp) − 1
39                      Quality ← Quality ⊕ "d"
40              else
41                  for i ← 1 to IntDisp
42                      Quality ← Quality ⊕ "a"
43  return DirStr ⊕ Quality ⊕ DiatSizeStr
```

Figure 1.20: The PI2PIN algorithm.

```
PIN2PINC(PIN)
1   CMPI ← PIN2PI(PIN)
2   ChromaticPitchInt ← CMPI[0]
3   MorpheticPitchInt ← CMPI[1]
4   MorpheticIntOctave ← ⌊MorpheticPitchInt/7⌋
5   PIC ← ⟨ChromaticPitchInt − 12MorpheticIntOctave, MorpheticPitchInt − 7MorpheticIntOctave⟩
6   return [PI2PIN(PIC)]
```

Figure 1.21: The PIN2PINC algorithm.

```
PIN2LOFDisp(PIN)
1   return PINC2LOFDisp(PIN2PINC(PIN))
```

Figure 1.22: The PIN2LOFDisp algorithm.

stored in the variable *AbsMorphInt*. *AbsMorphInt* is then used in lines 13 and 14 to compute the undisplaced chroma interval and undisplaced quality which are stored in the variables *UndispChromaInt* and *UndispQual*, respectively. *UndispChromaInt* is then used in lines 15–18 to compute the interval displacement which is stored in the variable *IntDisp*. *IntDisp* and *UndispQual* are then used in lines 20–42 to compute the quality which is stored in the variable *Quality*. The *DirStr*, *Quality* and *DiatSizeStr* are then concatenated in line 43 to give the pitch interval name which is returned.

The pitch interval name class of a pitch interval name can be computed using the function PIN2PINC, defined in Figure 1.21. PIN2PINC takes a single parameter, *PIN*, which must be a pitch interval name. The chromamorphetic pitch interval of *PIN* is first computed in line 1 using the function PIN2PI (defined in Figure 1.19) and stored in the variable *CMPI*. Then, in lines 2–3, *CMPI* is parsed into its component chromatic and morphetic pitch intervals which are stored in the variables *ChromaticPitchInt* and *MorpheticPitchInt*, respectively. The morphetic interval octave is then computed in line 4 and stored in the variable *MorpheticIntOctave*. In line 5, the chromamorphetic pitch interval $\langle 12, 7\rangle$, which corresponds to a rising perfect octave, is subtracted *MorpheticIntOctave* times from *CMPI* to give the chromamorphetic pitch interval of the rising pitch interval name with the least diatonic size in the pitch interval name class to which *PIN* belongs. This chromamorphetic pitch interval is stored in the variable *PIC*. The function PI2PIN, defined in Figure 1.20, is then used to compute the pitch interval name that corresponds to *PIC*.

The line-of-fifths displacement of a pitch interval name can be computed using the function PIN2LOFDisp, which is defined in Figure 1.22. This function has a single parameter, *PIN*, which must be a pitch interval name. PIN2LOFDisp first computes the pitch interval name class of *PIN* using the function PIN2PINC, defined in Figure 1.21. The function PINC2LOFDisp (defined in Figure 1.23) is then used to compute the line-of-fifths displacement.

The line-of-fifths displacement that corresponds to a pitch interval name class can be computed using the function PINC2LOFDisp, defined in Figure 1.23. This function has one parameter, *PINC*, which must be a pitch interval name class. In lines 1–9, *PINC* is parsed into its component quality and diatonic size, which are stored in the variables *Quality* and *DiatSize*,

```
PINC2LOFDisp(PINC)
1   if PINC[0] = 'r'
2       QualStart ← 1
3   else
4       QualStart ← 0
5   QualEnd ← 1 + QualStart
6   while PINC[QualEnd] ∉ {'1','2','3','4','5','6','7','8','9'}
7       QualEnd ← QualEnd + 1
8   Quality ← PINC[QualStart, QualEnd]
9   DiatSize ← Str2Num(PINC[QualEnd, |PINC|])
10  LOFDispClass ← (2(DiatSize − 1)) mod 7
11  if Quality = "p"
12      if DiatSize = 4
13          LOFDispCycle ← −1
14      else
15          LOFDispCycle ← 0
16  else
17      if Quality = "ma"
18          LOFDispCycle ← 0
19      else
20          if Quality = "mi"
21              LOFDispCycle ← −1
22          else
23              if Quality[0] = 'd'
24                  if DiatSize ∈ {1, 5}
25                      LOFDispCycle ← −|Quality|
26                  else
27                      LOFDispCycle ← −|Quality| − 1
28              else
29                  ▶ So Quality[0] = 'a'.
30                  if DiatSize = 4
31                      LOFDispCycle ← |Quality| − 1
32                  else
33                      LOFDispCycle ← |Quality|
34  return LOFDispClass + 7LOFDispCycle
```

Figure 1.23: The PINC2LOFDisp algorithm.

```
P2PI(P1, P2)
1    return ⟨P2[0] − P1[0], P2[1] − P1[1]⟩
```

Figure 1.24: The P2PI algorithm.

```
PNC2PINC(PNC1, PNC2)
1    PN1 ← PNC1 ⊕ "1"
2    PN2 ← PNC2 ⊕ "1"
3    P1 ← PN2P(PN1)
4    P2 ← PN2P(PN2)
5    if P2[1] < P1[1]
6        P2 ← ⟨P2[0] + 12, P2[1] + 7⟩
7    return PI2PIN(P2PI(P1, P2))
```

Figure 1.25: The PNC2PINC algorithm.

respectively. The line-of-fifths displacement class is then computed directly from the diatonic size in line 10 and stored in the variable *LOFDispClass*. Then, in lines 11–33, the line-of-fifths displacement cycle is computed from the quality and diatonic size and stored in the variable *LOFDispCycle*. The line-of-fifths displacement class and cycle are then used in line 34 to compute the line-of-fifths displacement which is returned.

If P_1 and P_2 are chromamorphetic pitches, then the chromamorphetic pitch interval from P_1 to P_2 can be computed using the simple function in Figure 1.24.

Given two pitch name classes, PNC_1 and PNC_2, the pitch interval name class of the interval from PNC_1 to PNC_2 can be computed using the function PNC2PINC, defined in Figure 1.25. This function takes two arguments, PNC_1 and PNC_2, which must both be valid pitch name classes. In lines 1–2 of this function, the variables PN_1 and PN_2 are set to equal the pitch names with octave number 1 in the pitch name classes PNC_1 and PNC_2, respectively. Then, in lines 3–4, the chromamorphetic pitches of PN_1 and PN_2 are computed using the function PN2P (defined in Figure 1.12) and stored in the variables P_1 and P_2, respectively. The pitch interval name class of the interval from PNC_1 to PNC_2 must be represented by the rising pitch interval name with the smallest diatonic size in the class. Therefore, if the interval from P_1 to P_2 is falling (i.e., the morphetic pitch of P_2 is less than that of P_1), P_2 must be transposed up a perfect octave (i.e., by the chromamorphetic pitch interval $\langle 12, 7\rangle$) and this is done in lines 5 and 6 of PNC2PINC. Finally, in line 7 of PNC2PINC, the chromamorphetic pitch interval from P_1 to P_2 is computed using the function P2PI (defined in Figure 1.24) and then this chromamorphetic pitch interval is converted into a pitch interval name using the function PI2PIN (defined in Figure 1.20). Note that the function PNC2PINC actually outputs the smallest rising pitch interval name in the pitch interval name class of the interval from PNC_1 to PNC_2—it does not enclose this name in square brackets to give a strictly correct pitch interval name class representation. This allows for pitch interval name classes to be manipulated using functions designed for processing pitch interval names without having to first strip away the square brackets.

1.5 Summary

A pitch spelling algorithm attempts to compute the correct pitch names (e.g., C♯4, B♭5 etc.) of the notes in a passage of tonal music, when given only the onset-time, MIDI note number and possibly the duration and voice of each note. Pitch spelling algorithms are essential for transcribing music from MIDI to notation. They can also be used to improve the effectiveness and efficiency of music information retrieval and analysis systems. Solving the problem of designing an effective pitch spelling algorithm may also allow us to learn more about the mental processes involved when an expert listener interprets Western tonal music.

The best pitch spelling algorithm to choose for a particular task may depend on the nature of the task. There are therefore a number of different criteria that could be used to evaluate the performance of a pitch spelling algorithm. In this study, I shall be primarily concerned with evaluating and comparing algorithms in terms of criteria that can be measured quantitatively. In particular, I shall focus on spelling accuracy, computational complexity, style dependence and robustness to noise in the data.

In the vast majority of cases, those who study and perform Western tonal music agree about how a note should be spelt in a given tonal context. Correspondingly, the vast majority of notes in authoritative published editions of scores of common practice tonal works are generally agreed to be spelt correctly by those who understand Western staff notation. Therefore the spelling accuracy of a pitch spelling algorithm can be evaluated by running it on tonal works and comparing the pitch names it predicts with those of the corresponding notes in authoritative published editions of scores of the works. In other words, such authoritative scores can provide us with a 'ground truth' that we can compare with the output of a pitch spelling algorithm.

A test corpus for testing pitch spelling algorithms should therefore be a large collection of works for which high quality score encodings exist. It should also be a representative sample of the population of works on which one intends to run the algorithms. The test corpora used in previous studies on pitch spelling algorithms have been deficient in various ways—for example, some have been too small, others have been stylistically too restricted and others have failed to be representative of some recognized, interesting wider population of works. In this study, the algorithms considered were evaluated and compared by running them on a test corpus containing 195972 notes and consisting of 216 movements from works by 8 baroque and classical composers (Corelli, Vivaldi, Telemann, J. S. Bach, Handel, Haydn, Mozart and Beethoven). I denote this test corpus by $\mathcal{C}$ throughout the remainder of this text. Table 1.4 gives a complete listing of the music in $\mathcal{C}$. In order to evaluate the style dependence of the algorithms, it was decided that the proportion of notes in the corpus in works by a particular composer had to be as similar as possible for each composer. The test corpus used in this study therefore contains almost exactly 24500 notes of music for each of the eight composers represented.

In this study, I use mainly the note accuracy and note error count as measures of spelling accuracy. The note error count is simply the number of notes spelt incorrectly by an algorithm over a particular corpus. The note accuracy is the proportion of notes within a corpus spelt correctly by an algorithm. I define the style dependence of an algorithm to be the standard deviation of the note accuracies (expressed as percentages) that the algorithm achieves over the 8 different composers represented in $\mathcal{C}$.

No entirely satisfactory method seems to be available for measuring the statistical significance of the difference between the note accuracies achieved by two algorithms over the test corpus $\mathcal{C}$. Nevertheless, the statistical significances of the differences between the note accuracies achieved by the best algorithms considered here will be estimated by the average of the p-values returned by the matched-sample t test over four different partitions of $\mathcal{C}$.

To evaluate robustness to temporal noise in the data, the best versions of the algorithms will be run on a 'noisy' version of $\mathcal{C}$ in which the onset times and durations have been randomly adjusted in order to simulate the types of temporal deviation that typically occur in data derived from human performances.

Finally, I provided definitions of various basic concepts, notation, terminology and functions that will be used throughout the remainder of this text.

Chapter 2

Longuet-Higgins's pitch spelling algorithm

2.1 Introduction

Pitch spelling is one of the tasks performed by Longuet-Higgins's (1976, 1987a, 1993) `music.p` program. This program takes a melody as input and generates a structural description that indicates the phrasing, articulation, pitch names and metrical structure of the melody. Longuet-Higgins (1987a, p. 114) emphasizes that the `music.p` program was intended to be used only on monophonic melodies and explicitly warns against using it on "accompanied melodies" or what he calls "covertly polyphonic" melodies (i.e., compound melodies).

The melody to be processed by `music.p` must first be encoded in the form of a list of triples, $\langle p, t_{\text{on}}, t_{\text{off}} \rangle$, each triple giving the *keyboard position*, p, together with the onset time, t_{on}, and the offset time, t_{off}, in centiseconds of a note (or sequence of tied notes) (Longuet-Higgins, 1987a, p. 115). If N is a note, then the keyboard position of N, denoted by $p(N)$, is an integer indicating the key that would have to be pressed on a normal piano keyboard in order to perform N, with C♮3 mapping onto 0, C♯3 and D♭3 mapping onto 1, C♮4 mapping onto 12 and so on. The keyboard position of a note is therefore simply 48 less than its MIDI note number. The order in which these triples occur in the input list corresponds to the order in which the notes occur in the input melody.

The pitch spelling algorithm implemented in `music.p` uses the keyboard positions of the notes in the melody to estimate a value of *sharpness*, q, for each note (Longuet-Higgins, 1987a, p. 111). If N is a note, then the sharpness of N, denoted by $q(N)$, is an integer indicating the position of the pitch name class of the note on the line of fifths (Temperley, 2001, p. 117) (see

					Key of C																	
Pitch name class	⋯	F♭	C♭	G♭	D♭	A♭	E♭	B♭	F	C	G	D	A	E	B	F♯	C♯	G♯	D♯	A♯	E♯	B♯	⋯
Sharpness	⋯	−8	−7	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	⋯
Chroma	⋯	7	2	9	4	11	6	1	8	3	10	5	0	7	2	9	4	11	6	1	8	3	⋯

Figure 2.1: The 'line of fifths' showing the sharpness and chroma associated with each pitch name class. The region corresponding to the key of C in Longuet-Higgins's theory of tonality is also indicated.

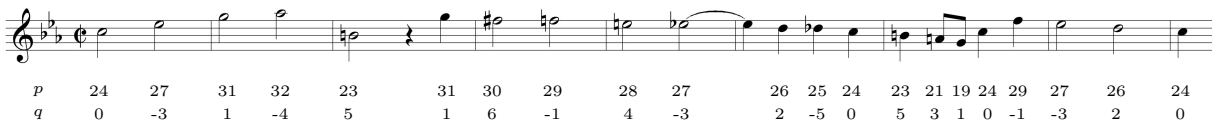


Figure 2.2: The theme from J. S. Bach's *Musical Offering* (BWV 1079) showing the keyboard position and sharpness of each note.

Figure 2.1). Longuet-Higgins's concept of sharpness is essentially the same as Temperley's (2001, p. 118) concept of *tonal pitch class* and Regener's (1973, p. 33) concept of *quint*. The pitch name class of a note can be determined unambiguously from its sharpness using the algorithm in Figure 2.13 (see discussion in section 2.3.4 below). Figure 2.2 shows the theme from Bach's *Musical Offering* (BWV 1079) together with the keyboard position, p, and the sharpness, q, of each note. The complete pitch name (i.e., including the octave number) of each note can be determined unambiguously from its keyboard position and its sharpness using the algorithm in Figure 2.15 (see section 2.3.5 below).

2.2 Longuet-Higgins's theory of tonality

The pitch spelling algorithm implemented in `music.p` is based on a "theory of tonality" (Longuet-Higgins, 1987a, p. 115), which I shall re-cast here in the form of six rules. The first of these rules is as follows.

Longuet-Higgins's Rule 1 *If X is a note in the input melody and k is the keyboard position of the putative local tonic at the point in the melody where X occurs, then X is assigned a sharpness, $q(X)$, such that $-5 \leq q(X) - q_k \leq 6$, where q_k is the sharpness assigned to the putative local tonic (Longuet-Higgins, 1987a, pp. 112–113).*

A hidden assumption in the theory is that, at the beginning of the melody, $|q_k|$ is the minimum value possible for the initial value of k. Longuet-Higgins's Rule 1 implies that every note is spelt so that its pitch name class is as close as possible to that of the putative local tonic on the line of fifths. Also, if $(p(X) - k) \bmod 12 = 6$, then $q(X)$ is set to $q_k + 6$ rather than $q_k - 6$. This implies that every note is spelt in the same way as it is in the harmonic chromatic scale beginning on k (Associated Board of the Royal Schools of Music, 1958, p. 78). Therefore, as Longuet-Higgins states, "a note is regarded as belonging to a given key if its sharpness relative to the tonic lies in the range -5 to $+6$ inclusive" (Longuet-Higgins, 1987a, p. 114). For example, in Longuet-Higgins's theory, the key of C is considered to contain all the pitch name classes on the line of fifths from D♭ to F♯, inclusive (see Figure 2.1).

Let X and Y be any two notes and let $\vec{XY}$ denote the pitch interval from X to Y. Longuet-Higgins defines the *degree* of $\vec{XY}$, denoted by $\delta q(\vec{XY})$, to be given by $q(Y) - q(X)$ (Longuet-Higgins, 1987a, p. 111). $\delta q(\vec{XY})$ is therefore the vector displacement of the pitch name class of Y from the pitch name class of X along the line of fifths.[1] $\delta q(\vec{XY})$ is negative if the pitch name

[1] For a definition of the term 'vector displacement', see Borowski and Borwein (1989, p. 171, s.v. **displacement**, sense 1).

class of Y is 'flatter' than that of X (i.e., the pitch name class of Y is to the left of the pitch name class of X on the line of fifths diagram in Figure 2.1); and $\delta q(\vec{XY})$ is positive if the pitch name class of Y is 'sharper' than that of X. For example, the degree of the interval from F♯ ($q = 6$) to D♭ ($q = -5$) is $-5 - 6 = -11$; and the degree of the interval from B♭ ($q = -2$) to A ($q = 3$) is $3 - (-2) = 5$. Longuet-Higgins (1987a, p. 112) defines the interval between X and Y to be *diatonic* if $|\delta q(\vec{XY})| < 6$, *diabolic* if $|\delta q(\vec{XY})| = 6$ and *chromatic* if $|\delta q(\vec{XY})| > 6$.

The other rules in Longuet-Higgins's theory of tonality are as follows.

Longuet-Higgins's Rule 2 *If X, Y and Z are three consecutive notes in the input melody then $\vec{XY}$ and $\vec{YZ}$ must never both be chromatic. If this condition is not satisfied, then the local key and the spelling of Y must be changed so that $\vec{XY}$ and $\vec{YZ}$ are both diatonic and the new spelling of Y is in the new key (Longuet-Higgins, 1987a, p. 113).*

Longuet-Higgins's Rule 3 *If $WXYZ$ is a sequence of four consecutive notes in the input melody and $\vec{XY}$ is chromatic, then $\vec{WX}$ and $\vec{YZ}$ must both be non-chromatic and $\vec{WY}$ and/or $\vec{XZ}$ must be diatonic. If these conditions are not satisfied, then the local key and the spelling of Y must be changed so that $\vec{XY}$ is diatonic and the new spelling of Y is in the new key (Longuet-Higgins, 1987a, pp. 113–114).*

Longuet-Higgins's Rule 4 *If the interval from a note X to the following note Y is an ascending semitone and $2 \leq q(Y) - q_k \leq 5$ where q_k is the sharpness assigned to the local tonic, then $q(X)$ is made equal to $q(Y) + 5$ and no modulation is deemed to have taken place, even though $q(X)$ is now outside the current key (Longuet-Higgins, 1987a, p. 114).*

Longuet-Higgins's Rule 4 is very similar to Temperley's (2001, p. 129) TPR 2 (see section 4.3 below).

Longuet-Higgins's Rule 5 *If two consecutive notes in a melody are an integer number of octaves apart then, for the purposes of determining pitch names, the second note should be assumed to have the same sharpness as the first and it should be ignored when considering the other rules (Longuet-Higgins, 1987a, p. 114).*

Longuet-Higgins (1987a, p. 114) expresses this rule by stating that, "for the purposes of establishing tonality one may conflate repeated notes, or notes separated by an octave".

Longuet-Higgins's Rule 6 *The tonic at the beginning of the melody can be determined from the first two notes and it will be either the first note or the note a fifth below it (Longuet-Higgins, 1987a, p. 114).*

Note that, in discussing his theory, Longuet-Higgins does not describe precisely *how* the tonic is to be determined from the first two notes. However, this can be discovered by examining the `music.p` source code directly (see below).

```
1   recordclass note pitch onset offset span deg index;
...
2   function res x;
3       loopif x<0 then x+12->x close;
4       erase (x//12);
5   end;

6   function int x;
7       res(7*x+5)-5;
8   end;

9   vars flag k l m n;

10  function modulate;
11      if m>2 then y-1->k;
12      elseif m<(-1) then y+6->k;
13      else exit;
14      int(x-k)->l; int(y-k)->m; int(z-k)->n;
15  end;

16  function hark;
17      m->l; n->m; int(z-k)->n;
18      if flag and abs(n-l)>6 then .modulate
19      close; false->flag;
20      if abs(n-m)<7 then return
21      elseif abs(m-l)>6 then .modulate
22      elseif abs(n-l)>6 and l<7 then true->flag
23      elseif n-m=7 and n<6 then m+12->m
24      close;
25  end;

26  function simplify tune; vars y;
27      tune.hd-1->y;
28      maplist(tune, lambda x;
29          if res(x-y)>0 then x
30          close; x->y; end);
31  end;

32  function intervals tune; vars ints x y z;
33      tune.simplify->tune;
34      false->flag; nil->ints;
35      tune.hd->y; tune.tl->tune;
36      if tune.null then return
37      else tune.hd->z; tune.tl->tune
38      close;
39      y->k; 0->m; int(z-k)->n;
40      if n=3 or n<0 and not(n=(-3))
41          then k+5->k; 1->m; n+1->n
42      close;
43      loopif tune.null.not
44      then y->x; z->y;
45          tune.hd->z; tune.tl->tune;
46          .hark; (m-l)::ints->ints
47      close;
48      rev((n-m)::ints);
49  end;

50  vars place;

51  function tuneup nlist; vars ints x0;
52      maplist(nlist, pitch).intervals->ints;
53      nlist.hd->x0; x0.pitch.int->place;
54      applist(nlist, lambda x;
55          x.pitch-x0.pitch->x.span;
56          if res(x.span)=0 then 0
57          else ints.hd; ints.tl->ints
58          close->x.deg;
59          x->x0;
60      end);
61  end;
```

Figure 2.3: The `tuneup` function from Longuet-Higgins's `music.p` program.

```
1   vars max min; 17->max; -13->min;

2   vars symbols:  [Fbb Cbb Gbb Dbb Abb Ebb Bbb Fb Cb Gb Db Ab Eb Bb
3       F C G D A E B Fs Cs Gs Ds As Es Bs Fx Cx Gx Dx Ax Ex Bx]
4       ->symbols;

5   vars symbol; newarray([%-15,19%],
6       lambda x; symbols.hd; symbols.tl->symbols end)->symbol;

7   function name note;
8       place+note.deg->place;
9       if place>max then "enh"; -12
10      elseif place<min then "enh"; 12
11      else 0
12      close+place->place;
13      place->note.index;
14      note.span; place.symbol;
15  end;
```

Figure 2.4: The `name` function from Longuet-Higgins's `music.p` program.

2.3 Detailed discussion of the algorithm

2.3.1 Introduction

Longuet-Higgins has published the full POP-11 source code of `music.p` (Longuet-Higgins, 1987a, pp. 120–126; Longuet-Higgins, 1993, pp. 486–492). The parts of this source code that deal with pitch spelling are shown in Figures 2.3 and 2.4.[2] In Figure 2.5, I give pseudocode for the complete pitch spelling algorithm implemented in `music.p`.

2.3.2 The input and the **Note** record type

In the LHPitchSpell algorithm in Figure 2.5, it is assumed that the encoded input melody has been read into memory and stored as an ordered set, **NList**, in which each element is a Note record (see Figure 2.6). Each Note record represents a single note (or a single sequence of tied notes) in the music and contains 8 fields: the 6 fields in the note recordclass defined in the `music.p` source code (see Figure 2.3, line 1) together with two additional fields, PNC and Enh. The PNC field is used to store the pitch name class of the note computed by the LHPitchSpell algorithm. The Enh field is used to indicate whether or not the pitch name class computed by the algorithm has had to be enharmonically altered as a result of it being outside the permitted range of pitch name classes (G𝄫 to A𝄪). In the `music.p` program, the pitch name class and the enharmonic alteration flag are written to output as soon as they are computed so there is no need for them to be stored.

[2]The following errors occur in the source code printed in Longuet-Higgins (1993).

1. Line 18 in Figure 2.3 is incorrectly printed in Longuet-Higgins (1993, p. 487) as:
   `if flag and abs(n-1)>6 then .modulate`
   That is, the `l` in line 18 of Figure 2.3 is incorrectly printed as a `1`.
2. The semi-colon at the end of line 30 in Figure 2.3 is omitted in Longuet-Higgins (1993, p. 488).
3. Line 55 in Figure 2.3 is incorrectly printed in Longuet-Higgins (1993, p. 488) as:
   `x.pitch - x0.pitch - ×.span;`

```
LHPitchSpell(NList)
    Tune ← ⟨ ⟩                                          ▶ tuneup function begins here.
    for i ← 0 to |NList| − 1
        Tune ← Tune ⊕ ⟨NList[i].PitchLH⟩
    Ints ← Intervals(Tune)
    x0 ← NList[0]
    Place ← IntLH(x0.PitchLH)
    j ← 0
    for i ← 0 to |NList| − 1
        NList[i].Span ← NList[i].PitchLH − x0.PitchLH
        if NList[i].Span mod 12 = 0
            NList[i].Deg ← 0
        else
            NList[i].Deg ← Ints[j]
            j ← j + 1
        x0 ← NList[i]                                   ▶ tuneup function ends here.
    Symbols ←⟨"Fff", "Cff", "Gff", "Dff", "Aff", "Eff", "Bff",
              "Ff", "Cf", "Gf", "Df", "Af", "Ef", "Bf",
              "F", "C", "G", "D", "A", "E", "B",
              "Fs", "Cs", "Gs", "Ds", "As", "Es", "Bs",
              "Fss", "Css", "Gss", "Dss", "Ass", "Ess", "Bss"⟩
    MaxPlace ← 17
    MinPlace ← −13
    for i ← 0 to |NList| − 1
        Place ← Place + NList[i].Deg                    ▶ name function begins here.
        NList[i].Enh ← false
        if Place > MaxPlace
            NList[i].Enh ← true
            Place ← Place − 12
        if Place < MinPlace
            NList[i].Enh ← true
            Place ← Place + 12
        NList[i].Index ← Place
        NList[i].PNC ← Symbols[Place − MinPlace + 2]    ▶ name function ends here.
    return NList
```

Figure 2.5: Longuet-Higgins's pitch spelling algorithm expressed in pseudocode.

Note
PITCH$_{LH}$
ONSET
OFFSET
SPAN
DEG
INDEX
ENH
PNC

Figure 2.6: The Note record type used in LHPITCHSPELL.

At the start of LHPITCHSPELL (see Figure 2.5), only the values of the PITCH$_{LH}$, ONSET and OFFSET fields of each Note record in **NList** are defined. If x is a Note record in **NList**, then the PITCH$_{LH}$, ONSET and OFFSET fields of x are used to store the keyboard position, onset time and offset time, respectively, of the note represented by x. In the LHPITCHSPELL algorithm, it is assumed that the order in which the Note records occur in **NList** corresponds to the order in which the notes occur in the input melody.

2.3.3 Lines 1–15 of LHPITCHSPELL

2.3.3.1 Overview

If X and Y are any two consecutive notes in the input melody (Y following X) and $p(X)$ and $p(Y)$ are the keyboard positions of X and Y respectively, then Longuet-Higgins (1987a, p. 111) defines the *span* of the interval $X\vec{Y}$, which I shall denote by $\delta p(X\vec{Y})$, to be the interval in keyboard semitones from the keyboard position of X to the keyboard position of Y. The span of $X\vec{Y}$ is therefore given by $\delta p(X\vec{Y}) = p(Y) - p(X)$.

The purpose of lines 1–15 of LHPITCHSPELL (see Figure 2.5) is to compute the values of the DEG and SPAN fields of every Note record in **NList**. In `music.p`, this portion of LHPITCHSPELL is implemented in the function `tuneup` (see lines 51–61 of Figure 2.3). If x and y are the Note records in **NList** corresponding to two consecutive notes, X and Y, in the input melody, then the SPAN field of y, denoted by y.SPAN, is set to equal $\delta p(X\vec{Y})$ and the DEG field of y (i.e., y.DEG) is set to equal $\delta q(X\vec{Y})$ (i.e., the degree of the interval from X to Y). The value of the SPAN and DEG fields of the first Note record in **NList** (i.e., **NList**[0].SPAN and **NList**[0].DEG) are both set to zero. Since the keyboard positions of X and Y are stored in x.PITCH$_{LH}$ and y.PITCH$_{LH}$, respectively, the value of y.SPAN is given by y.PITCH$_{LH}$ $- x$.PITCH$_{LH}$. Computing the value of y.DEG is rather more complicated as it involves estimating the sharpnesses of X and Y and then calculating $\delta q(X\vec{Y})$. In lines 1–4 of LHPITCHSPELL, the value of $\delta q(X\vec{Y})$ is computed for each pair of consecutive notes XY in the input melody for which $\delta p(X\vec{Y}) \bmod 12 \neq 0$ and stored in the ordered set, **Ints**.

2.3.3.2 Detailed discussion of lines 1–4 of LHPITCHSPELL

Lines 1–4 of LHPITCHSPELL are implemented in the first line of the `tuneup` function in `music.p` (see Figure 2.3, line 52). In lines 1–3 of LHPITCHSPELL, an ordered set, **Tune**, is computed

which satisfies the following condition

$$(|\mathbf{Tune}| = |\mathbf{NList}|) \wedge (\mathbf{Tune}[i] = \mathbf{NList}[i].\textsc{Pitch}_{\text{LH}} \text{ for all } 0 \leq i < |\mathbf{Tune}|).$$

That is, **Tune** is an ordered set of keyboard positions such that the ith element of **Tune** is the keyboard position of the ith element of **NList**. In line 4 of LHPitchSpell, the ordered set, **Tune**, is passed as an argument to the Intervals function, which returns the ordered set, **Ints**, which contains all the non-zero degrees of intervals between consecutive pairs of elements in **NList**.

2.3.3.2.1 The Intervals algorithm The algorithm Intervals is defined in Figure 2.7 and implemented in `music.p` in the code that appears in lines 9–25 and 32–49 in Figure 2.3. The Intervals algorithm implements all six rules in Longuet-Higgins's theory of tonality described above. Note that I have combined the `hark`, `modulate` and `intervals` functions in Figure 2.3 into the single algorithm in Figure 2.7. This makes it necessary to repeat a few lines of code in the Intervals algorithm (see Figure 2.7, lines 27–34 and 38–45). However, it avoids the use of global variables and functions that return multiple values.

The first step in the Intervals algorithm is to run the ordered set of keyboard positions, **Tune**, through the Simplify algorithm (defined in Figure 2.8). In `music.p`, the Simplify algorithm is implemented in the `simplify` function (see lines 26–31 of Figure 2.3). Simplify removes every keyboard position p from **Tune** that is preceded by a keyboard position with the same chroma as p. That is, for $0 < i < |\mathbf{Tune}|$, Simplify deletes $\mathbf{Tune}[i]$ if and only if

$$(\mathbf{Tune}[i] - \mathbf{Tune}[i-1]) \bmod 12 = 0.$$

The Simplify algorithm therefore implements Longuet-Higgins's Rule 5, defined above.

For each value of $0 \leq i \leq |\mathbf{Tune}| - 3$, the Intervals algorithm uses

1. the variable k to store the keyboard position of the current tonic;
2. the variables x, y and z to store the three consecutive keyboard positions $\mathbf{Tune}[i]$, $\mathbf{Tune}[i+1]$ and $\mathbf{Tune}[i+2]$; and
3. the variables ℓ, m and n to store the sharpnesses of x, y and z, respectively, relative to that of the current tonic, k.

If X is a note in the input melody and q_k is the sharpness assigned to the current tonic, whose keyboard position is k, then the sharpness of X relative to that of the current tonic is given by $q(X) - q_k$. Longuet-Higgins's Rule 1, defined above, states that $-5 \leq q(X) - q_k \leq 6$, and for any given pair of values, $p(X)$ and k, there is a unique value of $q(X) - q_k$ that satisfies this inequality. Furthermore, this unique value of $q(X) - q_k$ can be computed from $p(X)$ and k using the formula

$$q(X) - q_k = \textsc{Int}_{\text{LH}}(p(X) - k) = ((7(p(X) - k) + 5) \bmod 12) - 5.$$

The variables ℓ, m and n in the Intervals algorithm are therefore (nearly) always set to equal

```
INTERVALS(Tune)
1   Tune ← SIMPLIFY(Tune)
2   Flag ← false
3   Ints ← ⟨ ⟩
4   y ← Tune[0]
5   j ← 1
6   if j ≥ |Tune|
7      return ⟨ ⟩
8   else
9      z ← Tune[j]
10     j ← j + 1
11  k ← y                          ▶ i.e., assume first note is the tonic.
12  m ← 0
13  n ← INT_LH(z − k)
14  if (n = 3) ∨ (n < 0 ∧ n ≠ −3)  ▶ ...then assume first note is the dominant.
15     k ← k + 5
16     m ← 1
17     n ← n + 1
18  while j < |Tune|
19     x ← y
20     y ← z
21     z ← Tune[j]
22     j ← j + 1
23     ℓ ← m                       ▶ hark begins here.
24     m ← n
25     n ← INT_LH(z − k)
26     if Flag ∧ (ABS(n − ℓ) > 6)
27        if m > 2                 ▶ modulate begins here.
28           k ← y − 1
29        if m < −1
30           k ← y + 6
31        if (m > 2) ∨ (m < −1)
32           ℓ ← INT_LH(x − k)
33           m ← INT_LH(y − k)
34           n ← INT_LH(z − k)    ▶ modulate ends here.
35     Flag ← false
36     if ABS(n − m) > 6
37        if ABS(m − ℓ) > 6
38           if m > 2              ▶ modulate begins here.
39              k ← y − 1
40           if m < −1
41              k ← y + 6
42           if (m > 2) ∨ (m < −1)
43              ℓ ← INT_LH(x − k)
44              m ← INT_LH(y − k)
45              n ← INT_LH(z − k)  ▶ modulate ends here.
46        else
47           if (ABS(n − ℓ) > 6) ∧ (ℓ < 7)
48              Flag ← true
49           else
50              if (n − m = 7) ∧ (n < 6)
51                 m ← m + 12     ▶ hark ends here.
52     Ints ← Ints ⊕ ⟨m − ℓ⟩
53  return Ints ⊕ ⟨n − m⟩
```

Figure 2.7: Longuet-Higgins's INTERVALS algorithm.

```
SIMPLIFY(Tune)
1   y ← Tune[0] − 1
2   NewTune ← ⟨ ⟩
3   for i ← 0 to |Tune| − 1
4       if (Tune[i] − y) mod 12 > 0
5           NewTune ← NewTune ⊕ ⟨Tune[i]⟩
6       y ← Tune[i]
7   return NewTune
```

Figure 2.8: Longuet-Higgins's SIMPLIFY algorithm.

$\text{INT}_{\text{LH}}(x-k)$, $\text{INT}_{\text{LH}}(y-k)$ and $\text{INT}_{\text{LH}}(z-k)$, respectively. In `music.p`, the INT_{LH} function is implemented in the function `int` (see lines 6–8 of Figure 2.3). The `int` function in `music.p` calls the function `res`, defined in lines 2–5 of Figure 2.3. `res` simply returns the least positive residue modulo 12 of its argument (i.e., $\texttt{res}(x) = x \bmod 12$).

Let X, Y and Z be the notes in the input melody whose keyboard positions are stored in the INTERVALS variables x, y and z, respectively. The degree of the interval from X to Y, $\delta q(\vec{XY})$, was defined above to be given by $q(Y) - q(X)$. But

$$q(Y) - q(X) = q(Y) - q_k - q(X) + q_k = (q(Y) - q_k) - (q(X) - q_k)$$

and $q(Y) - q_k = \text{INT}_{\text{LH}}(y-k)$ and $q(X) - q_k = \text{INT}_{\text{LH}}(x-k)$, as explained in the previous paragraph. Therefore

$$\delta q(\vec{XY}) = \text{INT}_{\text{LH}}(y-k) - \text{INT}_{\text{LH}}(x-k).$$

But $\ell = \text{INT}_{\text{LH}}(x-k)$ and $m = \text{INT}_{\text{LH}}(y-k)$, therefore

$$\delta q(\vec{XY}) = m - \ell.$$

That is, in the INTERVALS algorithm, the degree of the interval from X to Y is given by $m - \ell$. Similarly, $\delta q(\vec{XZ}) = n - \ell$ and $\delta q(\vec{YZ}) = n - m$.

For each value of i in turn from 0 to $|\mathbf{Tune}| - 3$, the INTERVALS algorithm

1. sets x, y and z to equal the keyboard positions $\mathbf{Tune}[i]$, $\mathbf{Tune}[i+1]$ and $\mathbf{Tune}[i+2]$, respectively;
2. calculates the current tonic keyboard position, k;
3. calculates the relative sharpnesses, $\ell = \text{INT}_{\text{LH}}(x-k)$, $m = \text{INT}_{\text{LH}}(y-k)$ and $n = \text{INT}_{\text{LH}}(z-k)$; and
4. appends the value $m - \ell$ (i.e., the degree of the interval from x to y) to the ordered set, **Ints**, which is initially set to equal $\langle\,\rangle$.

Finally, the degree of the interval between the last two notes, given by the final value of $n - m$, is appended to the end of **Ints**.

In lines 4–17 of the INTERVALS algorithm, the initial value of k, the keyboard position of the tonic, is determined using the keyboard positions of the first two notes in the melody, in

accordance with Longuet-Higgins's Rule 6. Initially, y and z are set to equal the keyboard positions of the first and second notes in the melody, respectively (lines 4–10). The variable j is used throughout INTERVALS to store the index of the element of **Tune** following that whose value is currently stored in z. Lines 4–10 in INTERVALS correspond to lines 35–38 in Figure 2.3.

In accordance with Longuet-Higgins's Rule 6, the pitch of the first note is initially assumed to be the tonic, so k is initially set to equal the keyboard position of the first note, which is initially stored in y (see Figure 2.7, line 11). At line 12, $y = k$ therefore $\text{INT}_{\text{LH}}(y - k) = 0$ so m, the sharpness of y relative to the tonic, is initially set to zero. n is used to store the sharpness of z relative to the tonic so, in line 13, n is set to equal $\text{INT}_{\text{LH}}(z - k)$. Lines 11–13 in INTERVALS correspond to line 39 in Figure 2.3.

By the time we arrive at line 14 in INTERVALS, the value of k has been provisionally set to the keyboard position of the first note (currently stored in y). However, if the interval between the first and second notes is appropriate, the algorithm changes k to the keyboard position of the note a perfect fourth above the first note—that is, it reinterprets the first note as being the dominant of the opening key. In the text, Longuet-Higgins does not explain how this decision is made. However, from line 14 of INTERVALS, it is evident that there are two conditions that will cause the algorithm to reinterpret the first note as the dominant of the opening key.

First, the algorithm will reinterpret the first note as the dominant if $n = 3$ (that is, if the sharpness of z (the second note) is 3 above that of the first note). If $n = 3$, then the interval from the first note in the melody to the second will be a member of the pitch interval name class, [`"rma6"`]. Presumably this reflects the intuition that a falling minor third or rising major sixth at the beginning of a melody suggests the interval from the dominant to the mediant in a major key.

Second, the algorithm will reinterpret the first note as the dominant if $n < 0$ and n is not equal to -3, that is, if the interval from the first note to the second note is in one of the following pitch interval name classes

$$\{[\texttt{"rp4"}], [\texttt{"rmi7"}], [\texttt{"rmi6"}], [\texttt{"rmi2"}]\}\,.$$

This seems to reflect the intuition that these intervals are more likely to occur at the beginning of a melody whose first note is the dominant than one whose first note is the tonic.

The algorithm does not change the tonic if $n = -3$ (that is, if the interval from the first note to the second is a member of [`"rmi3"`]). Presumably this reflects the intuition that a rising minor third at the beginning of a melody suggests the interval from the tonic to the mediant in a minor key.

These criteria for determining the opening tonic seem rather *ad hoc* and unprincipled. As Longuet-Higgins (1987a, p. 114) admits, "this rule... [is] undoubtedly [one of] the weakest links in the tonal section of the program."

If $n = 3$ or if $n < 0$ and $n \neq -3$ in line 14 of INTERVALS, then lines 15–17 are executed. In line 15, k, the keyboard position of the tonic, is changed so that it is 5 semitones higher than the first note. This means that the first note is now the dominant, so m, its sharpness relative to the current tonic, is made equal to 1 in line 16. Similarly, since the tonic has effectively been shifted one step 'flatwards' along the line of fifths, n, the sharpness of the second note relative

to the tonic, is increased by one in line 17. Lines 14–17 in INTERVALS correspond to lines 40–42 in Figure 2.3.

On each iteration of the 'while' loop in lines 18–52 of INTERVALS, x, y, z and j are each advanced by one element of **Tune** (lines 19–22). This means that ℓ and m have to be updated and a new value of n has to be calculated corresponding to the new value of z (lines 23–25). However, before storing the degree of the interval from x to y (i.e., $m-\ell$) in **Ints**, the algorithm determines whether or not the local tonic has changed (lines 26 and 36–37). If it has, then the keyboard position of the new tonic is computed and stored in k (lines 27–30, 38–41) and the values of ℓ, m and n are recalculated for the new tonic (lines 32–34, 43–45). Lines 18–52 in INTERVALS correspond to lines 43–47 in Figure 2.3. Lines 23–51 in INTERVALS correspond to the `hark` function (see lines 16–25 in Figure 2.3). The calls to the `modulate` function in lines 18 and 21 in Figure 2.3 correspond to lines 27–34 and lines 38–45 in Figure 2.7, respectively.

Let us consider in more detail what happens on the $(i+1)$th iteration $(0 \leq i \leq |\mathbf{Tune}|-3)$ of the 'while' loop in lines 18–52 of INTERVALS. In lines 19–21, the variables x, y and z are set to equal three consecutive keyboard positions, $\mathbf{Tune}[i]$, $\mathbf{Tune}[i+1]$ and $\mathbf{Tune}[i+2]$, respectively. In line 22, j is updated so that it stores the index of the element of **Tune** following that whose value is currently stored in z (i.e., j is set to $i+3$). Lines 19–22 in INTERVALS correspond to lines 44–45 in Figure 2.3.

Because the values of x, y and z have been changed in lines 19–21 of INTERVALS, the values of ℓ, m and n have to be updated accordingly in lines 23–25. These three lines correspond to the first line of the `hark` function in `music.p` (line 17 in Figure 2.3), which is called in line 46 of Figure 2.3.

Let's assume that *Flag* is false at line 26 on the $(i+1)$th iteration of the 'while' loop in INTERVALS so that control skips to line 35 where *Flag* is reset to false.

If X, Y and Z are the notes in the input melody corresponding to the keyboard positions currently stored in x, y and z, then $n-m = \delta q(\vec{YZ})$, as has already been shown. The expression ABS$(n-m)$ in line 36 is therefore equal to $|\delta q(\vec{YZ})|$. This implies that ABS$(n-m) > 6$ if and only if $\vec{YZ}$ is chromatic. Similarly, ABS$(n-\ell) > 6$ iff $\vec{XZ}$ is chromatic and ABS$(m-\ell) > 6$ iff $\vec{XY}$ is chromatic. If $\vec{YZ}$ is not chromatic (i.e., ABS$(n-m) \leq 6$) then lines 37–51 are skipped and the degree of $\vec{XY}$ (i.e., $m-\ell$) is appended to **Ints** in line 52. If, however, $\vec{YZ}$ *is* chromatic in line 36 of INTERVALS, then control passes to line 37, which checks if $\vec{XY}$ is also chromatic. If both $\vec{XY}$ and $\vec{YZ}$ are chromatic then, according to Longuet-Higgins's Rule 2 (see above), the local key and the spelling of Y must be changed so that $\vec{XY}$ and $\vec{YZ}$ are both diatonic and the new spelling of Y is in the new key (Longuet-Higgins, 1987a, p. 113). These changes are supposed to be accomplished by lines 38–45 of INTERVALS, which correspond to the `modulate` function in `music.p` (see lines 10–15 in Figure 2.3). However, as I shall now demonstrate, the `modulate` function in `music.p` does *not*, in fact, provide an accurate implementation of Longuet-Higgins's Rule 2.

If $\vec{XY}$ and $\vec{YZ}$ are both chromatic, then, according to Longuet-Higgins's Rule 2, the local tonic (i.e., k) has to be changed so that $\vec{XY}$ and $\vec{YZ}$ become diatonic. This change of local key is supposed to be executed in either line 39 or line 41 of the INTERVALS algorithm. However, the local tonic k is only changed if either $m > 2$ or $m < -1$ (see lines 38 and 40 in Figure 2.7)

```
elseif m<0 then y+6->k;
```

Figure 2.9: In order to implement Longuet-Higgins's Rule 2 correctly, it may be sufficient to replace line 12 in Figure 2.3 with the line shown here.

if $m < 0$
    $k \leftarrow y + 6$
if $(m > 2) \vee (m < 0)$

Figure 2.10: In order to implement Longuet-Higgins's Rule 2 correctly, it may be sufficient to replace lines 29–31 and lines 40–42 in INTERVALS with the lines shown here.

and it is possible for m to equal -1 when $X\vec{Y}$ and $Y\vec{Z}$ are both chromatic. For example, this occurs on the second iteration of the 'while' loop when **Tune** $= \langle 0, 6, 5, 6 \rangle$. When $m = -1$ and $X\vec{Y}$ and $Y\vec{Z}$ are both chromatic, the change of local key (i.e., k) required by Longuet-Higgins's Rule 2 is not carried out. The necessary change of key would also not be carried out if m were equal to 2 when $X\vec{Y}$ and $Y\vec{Z}$ were chromatic. For example, at first sight, it might seem possible for us to arrive at line 38 in the INTERVALS algorithm with the variables l, m and n having the values -5, 2 and -5, respectively. It can be shown that if $m = 2$ in line 38 of the INTERVALS algorithm, then ℓ and n must both equal -5 (i.e., ℓ cannot equal 9 or 10 in this case). If we assume further (with no loss in generality) that $k = 0$ in line 38, then x, y and z must equal 1 mod 12, 2 mod 12 and 1 mod 12, respectively. All possible values of **Tune** containing between 3 and 7 elements and ending with the sequence $\langle 1, 2, 1 \rangle$ were checked and in none of these cases was m ever equal to 2 in line 38 of the INTERVALS algorithm. Consequently, it would seem that a correct implementation of Longuet-Higgins's Rule 2 could be achieved simply by changing line 12 in Figure 2.3 to that shown in Figure 2.9. Correspondingly, it would seem that changing lines 29–31 and lines 40–42 in INTERVALS to those shown in Figure 2.10 would be sufficient to ensure a correct implementation of Longuet-Higgins's Rule 2. However, I have not yet been able to prove that m is never equal to 2 in line 38 of INTERVALS. Therefore, with a risk of erring on the side of safety, I would propose replacing lines 11–12 in Figure 2.3 with those shown in Figure 2.11 and lines 29–31 and lines 40–42 in INTERVALS with those shown in Figure 2.12.

Let's assume that line 35 has just been executed on the $(i+1)$th iteration of the 'while' loop in INTERVALS, that $X\vec{Y}$ and $Y\vec{Z}$ are both chromatic and that *Flag* = false. If $m > 2$ then k, the keyboard position of the tonic, is changed in line 39 so that it becomes one less than y, the keyboard position of Y. This causes m to become -5 (i.e., the 'flattest' note in the new key). In other words, the key is changed by the smallest amount necessary for the pitch name class of Y to be enharmonically changed to the next 'flatter' pitch name class possible for the keyboard

```
if m>1 then y-1->k;
elseif m<0 then y+6->k;
```

Figure 2.11: If lines 11–12 in Figure 2.3 are replaced with those shown here, then Longuet-Higgins's Rule 2 will be correctly implemented in the `music.p` program.

if $m > 1$
    $k \leftarrow y - 1$
if $m < 0$
    $k \leftarrow y + 6$
if $(m > 1) \vee (m < 0)$

Figure 2.12: If lines 27–31 and lines 38–42 are replaced with those shown here, then Longuet-Higgins's Rule 2 will be correctly implemented in the INTERVALS algorithm.

position y (e.g., B♯ would be changed to C, F♯ would be changed to G♭ and so on). Similarly, if $m < -1$ then k is changed in line 40 so that m becomes 6 (i.e., the 'sharpest' note in the new key). In other words, if the pitch name class of Y before the key change is 'flatter' than that of the tonic before the key change, then the key is 'sharpened' by the smallest amount necessary for the pitch name class of Y to be enharmonically changed to the next 'sharper' pitch name class possible for the keyboard position y (e.g., C would become B♯, G♭ would become F♯ and so on). Clearly, if k is changed in lines 39 or 41 then the relative sharpnesses of X, Y and Z must be recalculated and this is done in lines 43–45.

I shall now explain how Longuet-Higgins's Rule 3 is implemented in `music.p`. Let's suppose that W, X, Y and Z are the four consecutive notes in the input melody that correspond to **Tune**$[i]$, **Tune**$[i+1]$, **Tune**$[i+2]$ and **Tune**$[i+3]$, respectively. On the $(i+1)$th iteration of the 'while' loop in INTERVALS, x, y and z will equal **Tune**$[i]$, **Tune**$[i+1]$ and **Tune**$[i+2]$, respectively, and therefore correspond to notes W, X and Y. Let's suppose that *Flag* = false at the beginning of the $(i+1)$th iteration of the 'while' loop in INTERVALS, so that, on this iteration, lines 27–34 are skipped. Longuet-Higgins's Rule 3 stipulates that, if $\vec{XY}$ is chromatic, then $\vec{WX}$ and $\vec{YZ}$ must both be non-chromatic and $\vec{WY}$ and/or $\vec{XZ}$ must be diatonic. The value of *Flag* is set to true in line 48 on the $(i+1)$th iteration iff

1. ABS$(n-m) > 6$ in line 36 (i.e., $\vec{XY}$ is chromatic); and
2. ABS$(m-l) \leq 6$ in line 37 (i.e., $\vec{WX}$ is non-chromatic); and
3. ABS$(n-l) > 6$ and $\ell < 7$ in line 47 (i.e., $\vec{WY}$ is chromatic and ℓ is not greater than 7 as a result of line 51 being executed on the previous iteration).

Then, lines 27–34 are executed on the $(i+2)$th iteration iff *Flag* was set to true on the $(i+1)$th iteration and ABS$(n-\ell) > 6$ in line 26. On the $(i+2)$th iteration, x, y and z are set to equal the keyboard positions of X, Y and Z, respectively. Therefore, ABS$(n-\ell) > 6$ in line 26 on the $(i+2)$th iteration iff $\vec{XZ}$ is chromatic. The effect of lines 27–34 on the $(i+2)$th iteration is to change the local tonic and the spelling of Y so that $\vec{XY}$ becomes diatonic. To recap, if

1. $\vec{XY}$ is chromatic and
2. $\vec{WX}$ is non-chromatic and
3. $\vec{WY}$ is chromatic and
4. $\vec{XZ}$ is chromatic,

then the spelling of Y is changed in lines 27–34 on the $(i+2)$th iteration in accordance with Longuet-Higgins's Rule 3. Note that, if $X\vec{Y}$ is chromatic and $W\vec{X}$ is chromatic, then this violates Longuet-Higgins's Rule 2 and the spelling of X will be changed in lines 38–45 on the $(i+1)$th iteration. Similarly, if $X\vec{Y}$ is chromatic and $Y\vec{Z}$ is chromatic, then this violates Longuet-Higgins's Rule 2 and the spelling of Y will be changed in lines 38–45 on the $(i+2)$th iteration.

It remains for me to describe how Longuet-Higgins's Rule 4 is implemented in the INTERVALS algorithm. This rule states that, if the interval from a note to the following note is an ascending semitone and the sharpness of the second note relative to the tonic is between 2 and 5, then the sharpness of the first note is made 5 steps greater than that of the second note and the tonic is not changed (Longuet-Higgins, 1987a, p. 114). This rule is implemented in lines 50–51 of INTERVALS which correspond to line 23 in Figure 2.3. If X, Y and Z are the notes in the input melody corresponding to the keyboard positions in x, y and z, respectively, on some given iteration of the 'while' loop in INTERVALS, then line 51 is executed iff

1. $n - m = 7$ and $n < 6$ in line 50 (i.e., $Y\vec{Z}$ is an ascending chromatic semitone and the sharpness of Z relative to the current tonic is less than 6);
2. $\text{ABS}(n - \ell) \leq 6$ or $\ell \geq 7$ in line 47 (i.e., $X\vec{Z}$ is non-chromatic or ℓ is greater than 6 as a result of line 51 being executed on the previous iteration); and
3. $\text{ABS}(m - \ell) \leq 6$ in line 37 (i.e., $X\vec{Y}$ is non-chromatic).

$n - m = 7$ iff $Y\vec{Z}$ is an ascending chromatic semitone and, if this is the case, then m is increased by 12 in line 51, causing the value of $n - m$ to become -5, which implies that $Y\vec{Z}$ becomes a diatonic semitone, as required by Longuet-Higgins's Rule 4. Note that there is no need to check that $n \geq 2$ in line 50, as this will necessarily be true if $n - m = 7$. Note also that, if $Y\vec{Z}$ is an ascending chromatic semitone and $X\vec{Y}$ is also chromatic, then the key and the spelling of Y will be changed in lines 38–45 in accordance with Longuet-Higgins's Rule 2.

2.3.3.3 Detailed discussion of lines 5–15 of LHPITCHSPELL

The primary purpose of lines 5–15 of LHPITCHSPELL is to assign the correct values to the SPAN and DEG fields of each Note record in **NList**. On the first iteration of the 'for' loop in lines 8–15, **NList**[0].SPAN becomes zero in line 9 because x_0 has been initialised to **NList**[0] in line 5. This means that the condition in line 10 is satisfied, so **NList**[0].DEG becomes zero in line 10. Control then skips to line 15, where x_0 is reset to **NList**[0]. On the $(i+1)$th iteration of the 'for' loop ($1 \leq i \leq |\textbf{NList}| - 1$), the SPAN field of the Note record representing the $(i+1)$th note in the input melody is set, in line 9, to equal the span of the interval from the ith to the $(i+1)$th note (i.e., **NList**$[i]$.PITCH_{LH} − **NList**$[i-1]$.PITCH_{LH}). If the ith and $(i+1)$th note have the same chroma (i.e., **NList**$[i]$.SPAN mod 12 = 0 in line 10) then the degree of the interval from the ith to the $(i+1)$th note is also zero, so the DEG field of the Note record representing the $(i+1)$th note (i.e., **NList**$[i]$.DEG) is set to zero in line 11. If, however, the span of the interval from the ith to the $(i+1)$th note is not an integer number of octaves, then the value of the DEG field of the Note record representing the $(i+1)$th note (i.e., **NList**$[i]$.DEG) is obtained

in line 13 from the ordered set, **Ints**, which was returned by the INTERVALS algorithm in line 4. The variable j is used to store the position in **Ints** of the next non-zero degree and, each time a value is drawn from **Ints** in line 13, the value of j is incremented accordingly in line 14.

2.3.4 Detailed discussion of lines 16–34 of LHPITCHSPELL

The purpose of lines 16–34 of LHPITCHSPELL is to compute the sharpness and pitch name class of each note in the input melody. Lines 24–33 in Figure 2.5 correspond to the `music.p name` function given in lines 7–15 of Figure 2.4. Lines 16–20 of LHPITCHSPELL correspond to lines 2–4 in Figure 2.4 and lines 21–22 in LHPITCHSPELL correspond to line 1 in Figure 2.4.

The 'for' loop in lines 23–33 of LHPITCHSPELL iterates over all the Note records in **NList**, calculating the appropriate values for the ENH, INDEX and PNC fields of each record. By the time this 'for' loop has terminated, the sharpness and pitch name class of each note, X, is stored in the INDEX and PNC fields, respectively, of the Note record representing X. The variable *Place* is initialised in line 6 to equal the sharpness of the first note in the input melody, which is calculated on the assumption that the opening tonic is C. In other words, the pitch name class of the first note in the melody is chosen to be as close as possible to C on the line of fifths. The expression **NList**$[i]$.DEG in line 24 gives the degree of the pitch interval from the ith to the $(i+1)$th note in the input melody. If X and Y are the ith and $(i+1)$th notes in the input melody, respectively, then, as defined above, the degree of the interval from X to Y, denoted by $\delta q(X\vec{Y})$ is given by $q(Y) - q(X)$. Therefore, the sharpness of Y is given by $q(Y) = q(X) + \delta q(X\vec{Y})$. In other words, the sharpness of the $(i+1)$th note in the input melody can be found by adding the sharpness of the ith note to the value of **NList**$[i]$.DEG. The sharpness of the $(i+1)$th note is calculated in this way and stored in the variable *Place* in line 24 on the $(i+1)$th iteration of the 'for' loop in lines 23–33.

Longuet-Higgins decided to restrict his algorithm so that it is only able to assign pitch name classes between G𝄫 and A𝄪, inclusive, on the line of fifths. The sharpness of A𝄪 is 17 and this is stored in the variable *MaxPlace* on line 21 of LHPITCHSPELL. Similarly, the sharpness of G𝄫 is -13 and this is stored in the variable *MinPlace* on line 22 of LHPITCHSPELL. If the sharpness of the $(i+1)$th note is greater than that of A𝄪, then $Place > MaxPlace$ in line 26 and *Place* is reduced by 12 in line 28. That is, the pitch name class of the $(i+1)$th note is changed so that it is the next 'flatter' pitch name class possible for the note's keyboard position (e.g., E𝄪 becomes F♯). Also, the ENH field of the $(i+1)$th Note record is set to true in line 27 in order to indicate that the spelling of the note has been enharmonically changed in order to keep the pitch name classes within the range permitted by the program. Similarly, if the sharpness of the $(i+1)$th note is less than that of G𝄫, then $Place < MinPlace$ in line 29, so *Place* is increased by 12 in line 31 and the ENH field of the $(i+1)$th Note record is set to true in line 30. In line 32 the (possibly enharmonically modified) value of *Place* is stored in the INDEX field of the $(i+1)$th Note record. In line 33, the value of *Place* is used to select the appropriate pitch name class from the 'line of fifths' table, **Symbols**, defined in lines 16–20. Given that the algorithm cannot assign pitch name classes flatter than G𝄫 or sharper than A𝄪, I do not understand why **Symbols** extends from F𝄫 to B𝄪, since it would seem that the pitch name classes F𝄫, C𝄫, E𝄪 and B𝄪 cannot be assigned by the algorithm.

```
Q2PNC(q)
1    i ← ⌊(q + 1)/7⌋
2    if i > 0
3        c ← "s"
4    else
5        c ← "f"
6    IStr ← ""
7    j ← 1
8    while j ≤ Abs(i)
9        IStr ← IStr ⊕ c
10       j ← j + 1
11   return ⟨"C", "D", "E", "F", "G", "A", "B"⟩ [(4q) mod 7] ⊕ IStr
```

Figure 2.13: The Q2PNC algorithm.

```
1    for i ← 0 to |NList| − 1
2        Place ← Place + NList[i].Deg
3        NList[i].Index ← Place
4        NList[i].PNC ← Q2PNC(Place)
```

Figure 2.14: Proposed replacement for lines 16–33 in LHPitchSpell to enable the assignment of any possible pitch name class.

In fact, as Regener (1973, p. 34) pointed out, if one knows the sharpness (or *quint*, as Regener calls it) of any note, then its pitch name class can be computed without resorting to looking it up in a predefined table. If q is the sharpness of a note and $i = \lfloor (q+1)/7 \rfloor$, then the inflection of the corresponding pitch name class is

1. $\bigoplus_{i=1}^{\text{Abs}(i)}$ `"f"` if $i \leq 0$; and
2. $\bigoplus_{i=1}^{\text{Abs}(i)}$ `"s"` if $i > 0$.

The letter name of the pitch name class is given by

$$\langle \texttt{"C"}, \texttt{"D"}, \texttt{"E"}, \texttt{"F"}, \texttt{"G"}, \texttt{"A"}, \texttt{"B"} \rangle \, [(4q) \bmod 7].$$

The function Q2PNC(q) defined in Figure 2.13 returns the pitch name class for any value of q. This implies that, if lines 16–33 in Figure 2.5 were replaced with the lines of pseudocode shown in Figure 2.14, then the resulting algorithm would be capable of assigning any possible pitch name class and would not need to enharmonically alter pitch name classes in order to keep them within some arbitrary range on the line of fifths. Clearly, in this modified version, the Enh field in each Note record would be unnecessary.

2.3.5 Computing the pitch name of each note in the input melody

The output of the LHPitchSpell algorithm is a list of Note records giving the keyboard position, onset time, offset time, sharpness and pitch name class of each note in the input melody. Each Note record also gives the degree and span of the melodic interval ending on the note represented by the Note record and indicates whether or not the pitch name class of the

```
QP2PN(q, p)
1    i ← ⌊(q + 1)/7⌋
2    p′ ← p − i
3    o ← 3 + ⌊p′/12⌋
4    if i > 0
5        c ← "s"
6    else
7        c ← "f"
8    istr ← ""
9    j ← 1
10   while j ≤ Abs(i)
11       IStr ← IStr ⊕ c
12       j ← j + 1
13   OStr ← Num2Str(o)
14   LStr ← ⟨"C", "D", "E", "F", "G", "A", "B"⟩ [(4q) mod 7]
15   return LStr ⊕ IStr ⊕ OStr
```

Figure 2.15: The QP2PN algorithm for computing the pitch name of a note from its sharpness, q, and its keyboard position, p.

note has had to be enharmonically altered in order to be within the range G𝄫 to A𝄪 on the line of fifths.

The output of LHPitchSpell does *not* explicitly give the pitch name of each note in the input melody. Fortunately, however, the pitch name of a note can be computed unambiguously from its sharpness q and its keyboard position p using the function QP2PN(q, p) given in Figure 2.15.

This implies that, if x is a Note record in the ordered set **NList** returned by the LH-PitchSpell algorithm in Figure 2.5, then the pitch name of the note represented by x is given by QP2PN$(x.\textsc{Index}, x.\textsc{Pitch}_{\text{LH}})$.

2.4 Evaluating Longuet-Higgins's pitch spelling algorithm

2.4.1 Longuet-Higgins's own evaluation of `music.p`

In his published accounts of the `music.p` program, Longuet-Higgins only reports the results of running the program on three short melodic fragments, namely, "Shave and a haircut, two bits" (Longuet-Higgins, 1987a, p. 116) and the first and second halves of the *cor anglais* solo from Act III of Wagner's *Tristan und Isolde* (Longuet-Higgins, 1987a, pp. 118–119). The program assigned the correct pitch names to all the notes in these examples apart from the note marked with an asterisk in Figure 2.16, which Wagner spelt as a D♭. This test corpus is very small indeed. Also, in my opinion, it is not well chosen. The virtue of the "Shave and a haircut" fragment is that there is no doubt that the fragment should be spelt as shown in Figure 2.17. Figure 2.17 therefore provides us with a very solid ground truth for this extract. However, this example is, in my opinion, too simple—any remotely feasible pitch spelling algorithm would spell it correctly. Conversely, the Wagner extracts are inappropriate for use in such a small test corpus because they are so chromatic as to be almost non-tonal. In any non-tonal passage, the pitch name assigned to a note is essentially arbitrary. Even Wagner's own notation of the passage therefore does not provide us with a solid ground truth spelling for these extracts. For

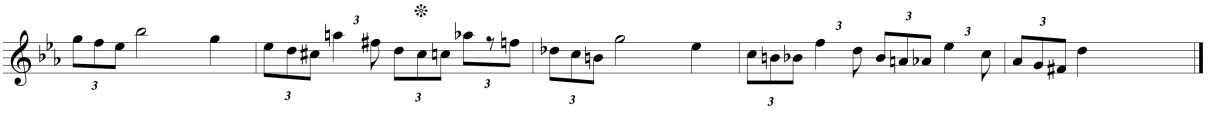


Figure 2.16: The second half of the cor anglais solo from Act III of Wagner's *Tristan and Isolde* used by Longuet-Higgins (1987a, p. 119) to evaluate his algorithm. The C♯ marked with an asterisk was spelt as a D♭ by Wagner.

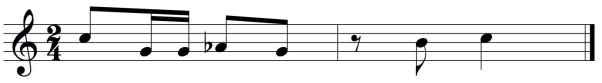


Figure 2.17: The "Shave and a Haircut" fragment used by Longuet-Higgins (1987a, p. 106) to evaluate his algorithm.

example, the C♯ predicted by the `music.p` program for the note marked with an asterisk in Figure 2.16 seems just as feasible as the D♭ used by Wagner. The problem is exacerbated by the fact that the extracts are purely melodic. It might be possible to disambiguate some of the spellings if a more complete harmonic context were provided.

2.4.2 A more thorough evaluation of Longuet-Higgins's pitch spelling algorithm

2.4.2.1 The versions of the algorithm tested

Six versions of the LHPitchSpell algorithm were implemented and tested on the test corpus, $\mathcal{C}$, defined in Table 1.4 above. These six versions of the algorithm are shown in Figures 2.18, 2.20, 2.22, 2.24, 2.25 and 2.26.

The LHPitchSpell$_1$ algorithm in Figure 2.18 is, in all essential aspects, the same as the LHPitchSpell algorithm implemented in the `music.p` program (see Figure 2.5). The only difference between the two algorithms is that the **NList** returned by LHPitchSpell$_1$ explicitly specifies the pitch name of each note rather than just its pitch name class. In order to do this, the variable **NList** in LHPitchSpell$_1$ is an ordered set of NewNote records, unlike the **NList** in LHPitchSpell, which is a list of Note records. The NewNote record type is defined in Figure 2.19. The NewNote record differs from the Note record defined in Figure 2.6 in that it has a field called PitchName instead of a field called PNC. The PitchName field of a NewNote record, x, is used to store the pitch name computed by the algorithm for the note in the input melody represented by x. In LHPitchSpell$_1$, the pitch name of each note is computed from its sharpness and keyboard position in line 28 using the QP2PN algorithm defined in Figure 2.15. Line 28 in LHPitchSpell$_1$ replaces line 33 in LHPitchSpell (see Figure 2.5). Also, because the pitch name is computed in LHPitchSpell$_1$ from the sharpness and keyboard position of each note, there is no need for the line-of-fifths table stored in **Symbols** in LHPitchSpell. Lines 16–20 in LHPitchSpell are therefore omitted from LHPitchSpell$_1$.

The LHPitchSpell$_2$ algorithm in Figure 2.20 is identical to the LHPitchSpell$_1$ algorithm except that the **Ints** variable is computed in line 4 of the LHPitchSpell$_2$ algorithm using the

```
LHPitchSpell1(NList)
1   Tune ← ⟨ ⟩
2   for i ← 0 to |NList| − 1
3       Tune ← Tune ⊕ ⟨NList[i].PitchLH⟩
4   Ints ← Intervals(Tune)
5   x0 ← NList[0]
6   Place ← IntLH(x0.PitchLH)
7   j ← 0
8   for i ← 0 to |NList| − 1
9       NList[i].Span ← NList[i].PitchLH − x0.PitchLH
10      if NList[i].Span mod 12 = 0
11          NList[i].Deg ← 0
12      else
13          NList[i].Deg ← Ints[j]
14          j ← j + 1
15      x0 ← NList[i]
16  MaxPlace ← 17
17  MinPlace ← −13
18  for i ← 0 to |NList| − 1
19      Place ← Place + NList[i].Deg
20      NList[i].Enh ← false
21      if Place > MaxPlace
22          NList[i].Enh ← true
23          Place ← Place − 12
24      if Place < MinPlace
25          NList[i].Enh ← true
26          Place ← Place + 12
27      NList[i].Index ← Place
28      NList[i].PitchName ← QP2PN(NList[i].Index, NList[i].PitchLH)
29  return NList
```

Figure 2.18: The LHPitchSpell$_1$ algorithm.

NewNote
- PitchLH
- Onset
- Offset
- Span
- Deg
- Index
- Enh
- PitchName

Figure 2.19: The NewNote record type used in LHPitchSpell$_{1–6}$.

```
LHPitchSpell2(NList)
1   Tune ← ⟨ ⟩
2   for i ← 0 to |NList| − 1
3       Tune ← Tune ⊕ ⟨NList[i].PitchLH⟩
4   Ints ← Intervals2(Tune)
5   x0 ← NList[0]
6   Place ← IntLH(x0.PitchLH)
7   j ← 0
8   for i ← 0 to |NList| − 1
9       NList[i].Span ← NList[i].PitchLH − x0.PitchLH
10      if NList[i].Span mod 12 = 0
11          NList[i].Deg ← 0
12      else
13          NList[i].Deg ← Ints[j]
14          j ← j + 1
15      x0 ← NList[i]
16  MaxPlace ← 17
17  MinPlace ← −13
18  for i ← 0 to |NList| − 1
19      Place ← Place + NList[i].Deg
20      NList[i].Enh ← false
21      if Place > MaxPlace
22          NList[i].Enh ← true
23          Place ← Place − 12
24      if Place < MinPlace
25          NList[i].Enh ← true
26          Place ← Place + 12
27      NList[i].Index ← Place
28      NList[i].PitchName ← QP2PN(NList[i].Index, NList[i].PitchLH)
29  return NList
```

Figure 2.20: The LHPitchSpell$_2$ algorithm.

```
INTERVALS2(Tune)
1   Tune ← SIMPLIFY(Tune)
2   Flag ← false
3   Ints ← ⟨ ⟩
4   y ← Tune[0]
5   j ← 1
6   if j ≥ |Tune|
7       return ⟨ ⟩
8   else
9       z ← Tune[j]
10      j ← j + 1
11  k ← y
12  m ← 0
13  n ← INTLH(z − k)
14  if (n = 3) ∨ (n < 0 ∧ n ≠ −3)
15      k ← k + 5
16      m ← 1
17      n ← n + 1
18  while j < |Tune|
19      x ← y
20      y ← z
21      z ← Tune[j]
22      j ← j + 1
23      ℓ ← m
24      m ← n
25      n ← INTLH(z − k)
26      if Flag ∧ (ABS(n − ℓ) > 6)
27          if m > 2
28              k ← y − 1
29          if m < 0
30              k ← y + 6
31          if (m > 2) ∨ (m < 0)
32              ℓ ← INTLH(x − k)
33              m ← INTLH(y − k)
34              n ← INTLH(z − k)
35      Flag ← false
36      if ABS(n − m) > 6
37          if ABS(m − ℓ) > 6
38              if m > 2
39                  k ← y − 1
40              if m < 0
41                  k ← y + 6
42              if (m > 2) ∨ (m < 0)
43                  ℓ ← INTLH(x − k)
44                  m ← INTLH(y − k)
45                  n ← INTLH(z − k)
46          else
47              if (ABS(n − ℓ) > 6) ∧ (ℓ < 7)
48                  Flag ← true
49              else
50                  if (n − m = 7) ∧ (n < 6)
51                      m ← m + 12
52      Ints ← Ints ⊕ ⟨m − ℓ⟩
53  return Ints ⊕ ⟨n − m⟩
```

Figure 2.21: The INTERVALS2 algorithm.

```
LHPitchSpell3(NList)
1    Tune ← ⟨ ⟩
2    for i ← 0 to |NList| − 1
3        Tune ← Tune ⊕ ⟨NList[i].Pitch_LH⟩
4    Ints ← Intervals3(Tune)
5    x0 ← NList[0]
6    Place ← Int_LH(x0.Pitch_LH)
7    j ← 0
8    for i ← 0 to |NList| − 1
9        NList[i].Span ← NList[i].Pitch_LH − x0.Pitch_LH
10       if NList[i].Span mod 12 = 0
11           NList[i].Deg ← 0
12       else
13           NList[i].Deg ← Ints[j]
14           j ← j + 1
15       x0 ← NList[i]
16   MaxPlace ← 17
17   MinPlace ← −13
18   for i ← 0 to |NList| − 1
19       Place ← Place + NList[i].Deg
20       NList[i].Enh ← false
21       if Place > MaxPlace
22           NList[i].Enh ← true
23           Place ← Place − 12
24       if Place < MinPlace
25           NList[i].Enh ← true
26           Place ← Place + 12
27       NList[i].Index ← Place
28       NList[i].PitchName ← QP2PN(NList[i].Index, NList[i].Pitch_LH)
29   return NList
```

Figure 2.22: The LHPitchSpell$_3$ algorithm.

Intervals$_2$ algorithm defined in Figure 2.21 instead of the Intervals algorithm defined in Figure 2.7.

As discussed in section 2.3.3.2.1 above, the Intervals algorithm in Figure 2.7 does not provide a correct implementation of Longuet-Higgins's Rule 2. It was suggested that, in order to implement Longuet-Higgins's Rule 2 correctly, it may be sufficient to replace lines 29–31 and lines 40–42 in Intervals with the lines shown in Figure 2.10. In the Intervals$_2$ algorithm, these replacements have been made. Apart from these changes, the Intervals$_2$ algorithm is the same as the Intervals algorithm.

The LHPitchSpell$_3$ algorithm in Figure 2.22 is identical to the LHPitchSpell$_1$ algorithm except that the **Ints** variable is computed in line 4 of the LHPitchSpell$_3$ algorithm using the Intervals$_3$ algorithm defined in Figure 2.23 instead of the Intervals algorithm defined in Figure 2.7.

As discussed in section 2.3.3.2.1 above, the Intervals algorithm in Figure 2.7 does not provide a correct implementation of Longuet-Higgins's Rule 2. However, it was pointed out that this rule would certainly be implemented correctly, if lines 27–31 and lines 38–42 in Intervals were replaced with those shown in Figure 2.12. In the Intervals$_3$ algorithm these replacements have been made. Apart from these changes, the Intervals$_3$ algorithm is the same as the Intervals algorithm.

LHPitchSpell$_{4–6}$ (see Figures 2.24, 2.25 and 2.26) are the same as LHPitchSpell$_{1–3}$,

```
INTERVALS3(Tune)
1   Tune ← SIMPLIFY(Tune)
2   Flag ← false
3   Ints ← ⟨ ⟩
4   y ← Tune[0]
5   j ← 1
6   if j ≥ |Tune|
7       return ⟨ ⟩
8   else
9       z ← Tune[j]
10      j ← j + 1
11  k ← y
12  m ← 0
13  n ← INTLH(z − k)
14  if (n = 3) ∨ (n < 0 ∧ n ≠ −3)
15      k ← k + 5
16      m ← 1
17      n ← n + 1
18  while j < |Tune|
19      x ← y
20      y ← z
21      z ← Tune[j]
22      j ← j + 1
23      ℓ ← m
24      m ← n
25      n ← INTLH(z − k)
26      if Flag ∧ (ABS(n − ℓ) > 6)
27          if m > 1
28              k ← y − 1
29          if m < 0
30              k ← y + 6
31          if (m > 1) ∨ (m < 0)
32              ℓ ← INTLH(x − k)
33              m ← INTLH(y − k)
34              n ← INTLH(z − k)
35      Flag ← false
36      if ABS(n − m) > 6
37          if ABS(m − ℓ) > 6
38              if m > 1
39                  k ← y − 1
40              if m < 0
41                  k ← y + 6
42              if (m > 1) ∨ (m < 0)
43                  ℓ ← INTLH(x − k)
44                  m ← INTLH(y − k)
45                  n ← INTLH(z − k)
46          else
47              if (ABS(n − ℓ) > 6) ∧ (ℓ < 7)
48                  Flag ← true
49              else
50                  if (n − m = 7) ∧ (n < 6)
51                      m ← m + 12
52      Ints ← Ints ⊕ ⟨m − ℓ⟩
53  return Ints ⊕ ⟨n − m⟩
```

Figure 2.23: The INTERVALS3 algorithm.

```
LHPitchSpell4(NList)
1    Tune ← ⟨ ⟩
2    for i ← 0 to |NList| − 1
3        Tune ← Tune ⊕ ⟨NList[i].Pitch_LH⟩
4    Ints ← Intervals(Tune)
5    x0 ← NList[0]
6    Place ← Int_LH(x0.Pitch_LH)
7    j ← 0
8    for i ← 0 to |NList| − 1
9        NList[i].Span ← NList[i].Pitch_LH − x0.Pitch_LH
10       if NList[i].Span mod 12 = 0
11           NList[i].Deg ← 0
12       else
13           NList[i].Deg ← Ints[j]
14           j ← j + 1
15       x0 ← NList[i]
16   for i ← 0 to |NList| − 1
17       Place ← Place + NList[i].Deg
18       NList[i].Index ← Place
19       NList[i].PitchName ← QP2PN(NList[i].Index, NList[i].Pitch_LH)
20   return NList
```

Figure 2.24: The LHPitchSpell$_4$ algorithm.

respectively, except that lines 16–17 and lines 20–26 in LHPitchSpell$_{1–3}$ have been deleted in LHPitchSpell$_{4–6}$. The effect of these deletions is to remove the restriction in LHPitchSpell of only permitting notes to have pitch name classes lying between G𝄫 and A𝄪, inclusive, on the line of fifths.

The relationships between the versions of LHPitchSpell tested here may be summarised as follows (see Table 2.1).

1. LHPitchSpell$_1$ is essentially identical to the algorithm implemented in the `music.p` program.

2. LHPitchSpell$_{1–3}$ are the same as LHPitchSpell$_{4–6}$, respectively, except that, in LHPitchSpell$_{1–3}$, the computed pitch names are restricted to being between G𝄫 and A𝄪 (inclusive) on the line of fifths; whereas, in LHPitchSpell$_{4–6}$, there are no such restrictions as to the positions of the computed pitch names on the line of fifths. For this reason, I shall refer to LHPitchSpell$_{1–3}$ as the 'restricted' algorithms and to LHPitchSpell$_{4–6}$ as the 'derestricted' algorithms in the discussion below.

3. LHPitchSpell$_2$ is the same as LHPitchSpell$_1$ except that it incorporates a modification which *may* be sufficient to ensure that Longuet-Higgins's Rule 2 is correctly implemented (see Figure 2.10). The effect of this modification is that a modulation (i.e., a change of tonic) is triggered when a subdominant is preceded and followed by a raised subdominant (e.g., F♯ - F - F♯ when the tonic is C) as well as in all other cases where a modulation is triggered in LHPitchSpell$_1$.

4. Finally, LHPitchSpell$_3$ is the same as LHPitchSpell$_2$ except that it incorporates a further modification that definitely ensures that Longuet-Higgins's Rule 2 is correctly implemented (see Figure 2.12). The effect of this modification is that a modulation is

```
LHPitchSpell5(NList)
1   Tune ← ⟨ ⟩
2   for i ← 0 to |NList| − 1
3       Tune ← Tune ⊕ ⟨NList[i].Pitch_LH⟩
4   Ints ← Intervals2(Tune)
5   x0 ← NList[0]
6   Place ← Int_LH(x0.Pitch_LH)
7   j ← 0
8   for i ← 0 to |NList| − 1
9       NList[i].Span ← NList[i].Pitch_LH − x0.Pitch_LH
10      if NList[i].Span mod 12 = 0
11          NList[i].Deg ← 0
12      else
13          NList[i].Deg ← Ints[j]
14          j ← j + 1
15      x0 ← NList[i]
16  for i ← 0 to |NList| − 1
17      Place ← Place + NList[i].Deg
18      NList[i].Index ← Place
19      NList[i].PitchName ← QP2PN(NList[i].Index, NList[i].Pitch_LH)
20  return NList
```

Figure 2.25: The LHPitchSpell$_5$ algorithm.

```
LHPitchSpell6(NList)
1   Tune ← ⟨ ⟩
2   for i ← 0 to |NList| − 1
3       Tune ← Tune ⊕ ⟨NList[i].Pitch_LH⟩
4   Ints ← Intervals3(Tune)
5   x0 ← NList[0]
6   Place ← Int_LH(x0.Pitch_LH)
7   j ← 0
8   for i ← 0 to |NList| − 1
9       NList[i].Span ← NList[i].Pitch_LH − x0.Pitch_LH
10      if NList[i].Span mod 12 = 0
11          NList[i].Deg ← 0
12      else
13          NList[i].Deg ← Ints[j]
14          j ← j + 1
15      x0 ← NList[i]
16  for i ← 0 to |NList| − 1
17      Place ← Place + NList[i].Deg
18      NList[i].Index ← Place
19      NList[i].PitchName ← QP2PN(NList[i].Index, NList[i].Pitch_LH)
20  return NList
```

Figure 2.26: The LHPitchSpell$_6$ algorithm.

	'Restricted' algorithms: pitch name classes restricted to being between G𝄫 and A𝄪, inclusive, on the line of fifths.	'Derestricted' algorithms: no restriction on positions of pitch name classes on line of fifths.
Uses INTERVALS algorithm: incorrect implementation of Longuet-Higgins's Rule 2.	LHPITCHSPELL$_1$	LHPITCHSPELL$_4$
Uses INTERVALS$_2$ algorithm: possibly correct implementation of Longuet-Higgins's Rule 2; $\sharp\hat{4}-\hat{4}-\sharp\hat{4}$ triggers modulation.	LHPITCHSPELL$_2$	LHPITCHSPELL$_5$
Uses INTERVALS$_3$ algorithm: definitely correct implementation of Longuet-Higgins's Rule 2; $\flat\hat{2}-\hat{2}-\flat\hat{2}$ and $\sharp\hat{4}-\hat{4}-\sharp\hat{4}$ trigger modulation.	LHPITCHSPELL$_3$	LHPITCHSPELL$_6$

Table 2.1: Summary of relationships between versions of LHPITCHSPELL tested here.

triggered when a supertonic is preceded and followed by a flattened supertonic (e.g., D♭ - D - D♭ when the tonic is C) as well as in all other cases where a modulation is triggered in LHPITCHSPELL$_2$.

2.4.2.2 Computational complexity

It is easy to see that for every version of the LHPITCHSPELL algorithm tested here, both the worst-case running time and the worst-case space complexity are $O(|\mathbf{NList}|)$.

2.4.2.3 Converting OPNDV datasets into NLists

Each of the six versions of the LHPITCHSPELL algorithm tested takes as input an ordered set, **NList**, of NewNote records as described above. Each OPNDV dataset in the test corpus therefore had to be converted into such an **NList** before being processed by one of the tested algorithms. As mentioned above, Longuet-Higgins (1987a, p. 114) intended the `music.p` program to be used only on monophonic melodies and explicitly warns against using it on "accompanied melodies" or what he calls "covertly polyphonic" melodies (i.e., compound melodies). It was therefore decided that two **NList**s should be derived from each OPNDV dataset, the first generated using the algorithm in Figure 2.27, and the second generated using the algorithm in Figure 2.28.

In the OPNDV2NLIST$_1$ algorithm in Figure 2.27, the OPNDV dataset, *OPNDV*, is first sorted in line 1 using the SORT$_{\text{onset}}$ function. The SORT$_{\text{onset}}$ function takes an unordered OPNDV dataset, *OPNDV*, as its argument and returns an ordered set, **OPNDV**, which only contains every element of *OPNDV*, sorted so that, for any pair of consecutive elements, **OPNDV**$[i]$ and **OPNDV**$[i+1]$, in **OPNDV**, $(0 \leq i < |OPNDV| - 1)$,

1. the onset time of **OPNDV**$[i]$ is less than the onset time of **OPNDV**$[i+1]$; or
2. the onset time of **OPNDV**$[i]$ is the same as that of **OPNDV**$[i+1]$ and the chromatic pitch of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$; or

```
OPNDV2NList1(OPNDV)
1    OPNDV ← Sort_onset(OPNDV)
2    for i ← 0 to |OPNDV| − 1
3        p ← (PN2P(OPNDV[i][1]))[0] − 27
4        t_on ← OPNDV[i][0]
5        t_off ← t_on + OPNDV[i][2]
6        NList ← NList ⊕ ⟨MakeNewNote⟩
7        NList[|NList| − 1].Pitch_LH ← p
8        NList[|NList| − 1].Onset ← t_on
9        NList[|NList| − 1].Offset ← t_off
10   return NList
```

Figure 2.27: The OPNDV2NList$_1$ algorithm.

```
OPNDV2NList2(OPNDV)
1    OPNDV ← Sort_voice(OPNDV)
2    for i ← 0 to |OPNDV| − 1
3        p ← (PN2P(OPNDV[i][1]))[0] − 27
4        t_on ← OPNDV[i][0]
5        t_off ← t_on + OPNDV[i][2]
6        NList ← NList ⊕ ⟨MakeNewNote⟩
7        NList[|NList| − 1].Pitch_LH ← p
8        NList[|NList| − 1].Onset ← t_on
9        NList[|NList| − 1].Offset ← t_off
10   return NList
```

Figure 2.28: The OPNDV2NList$_2$ algorithm.

3. the onset time and chromatic pitch of **OPNDV**$[i]$ and **OPNDV**$[i+1]$ are the same but the duration of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$; or

4. the onset time, chromatic pitch and duration of **OPNDV**$[i]$ and **OPNDV**$[i+1]$ are the same but the voice number of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$; or

5. the onset time, chromatic pitch, duration and voice number of **OPNDV**$[i]$ and **OPNDV**$[i+1]$ are the same but the morphetic pitch of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$.

In lines 2–10 of OPNDV2NList$_1$ (see Figure 2.27), an ordered set of NewNote records, **NList**, is constructed which satisfies the following conditions:

1. $|\mathbf{NList}| = |OPNDV|$; and

2. **NList**$[i]$.Pitch$_{\text{LH}}$ is 27 less than the chromatic pitch of the note represented by **OPNDV**$[i]$ for all $0 \leq i < |OPNDV|$ (this is calculated in line 3); and

3. **NList**$[i]$.Onset is equal to the onset time of the note represented by **OPNDV**$[i]$ for all $0 \leq i < |OPNDV|$ (this is calculated in line 4); and

4. **NList**$[i]$.Offset is equal to the offset time of the note represented by **OPNDV**$[i]$ for all $0 \leq i < |OPNDV|$ (this is calculated in line 5); and

5. all other fields in **NList**$[i]$ are set to nil (for all $0 \leq i < |OPNDV|$).

The OPNDV2NLIST$_2$ algorithm in Figure 2.28 is the same as the OPNDV2NLIST$_1$ algorithm in Figure 2.27 except that, in line 1 of OPNDV2NLIST$_2$, *OPNDV* is sorted using the function SORT$_{\text{voice}}$ instead of the function SORT$_{\text{onset}}$ used in line 1 of OPNDV2NLIST$_1$. The SORT$_{\text{voice}}$ function takes an unordered OPNDV dataset, *OPNDV*, as its argument and returns an ordered set, **OPNDV**, which only contains every element of *OPNDV*, sorted so that for any pair of consecutive elements, **OPNDV**$[i]$ and **OPNDV**$[i+1]$, in **OPNDV**, $(0 \leq i < |OPNDV| - 1)$,

1. the voice number of **OPNDV**$[i]$ is less than the voice number of **OPNDV**$[i+1]$; or
2. the voice number of **OPNDV**$[i]$ is the same as that of **OPNDV**$[i+1]$ and the onset time of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$; or
3. the voice number and onset time of **OPNDV**$[i]$ and **OPNDV**$[i+1]$ are the same but the chromatic pitch of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$; or
4. the voice number, onset time and chromatic pitch of **OPNDV**$[i]$ and **OPNDV**$[i+1]$ are the same but the duration of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$; or
5. the voice number, onset time, chromatic pitch and duration of **OPNDV**$[i]$ and **OPNDV**$[i+1]$ are the same but the morphetic pitch of **OPNDV**$[i]$ is less than that of **OPNDV**$[i+1]$.

The SORT$_{\text{voice}}$ function therefore effectively partitions the OPNDV dataset into voices and sorts the datapoints within each voice into order of increasing onset time.

Longuet-Higgins's claim that his algorithm is inappropriate for use on polyphonic music leads naturally to the hypothesis that it would perform better when the voices are arranged 'end-to-end' in **NList** (as they are when **NList** is generated using OPNDV2NLIST$_2$) than when the voices are 'interleaved' (as they are when **NList** is generated using OPNDV2NLIST$_1$). It was possible to test this hypothesis by comparing the performance of each algorithm when run on the set of **NList**s generated from the test corpus using OPNDV2NLIST$_1$, with its performance when run on the set of **NList**s generated using OPNDV2NLIST$_2$.

2.4.2.4 Results and discussion

Tables 2.2, 2.3 and 2.4 summarise the results obtained when the six algorithms LHPITCHSPELL$_{1–6}$ were run on the test corpus, $\mathcal{C}$, defined in Table 1.4 above. Each algorithm was run on both the set of 'interleaved' **NList**s generated from the test corpus using OPNDV2NLIST$_1$ and the set of 'end-to-end' **NList**s generated using OPNDV2NLIST$_2$.

Tables 2.2 and 2.3 show the note error counts and note accuracies for the various versions of the algorithm tested, for the complete test corpus, $\mathcal{C}$, and for each subset of $\mathcal{C}$ containing movements by one of the eight composers. In the first column of each of these tables, an entry of the form "LHn" represents algorithm LHPITCHSPELL$_n$, run on the set of 'interleaved' **NList**s; and an entry of the form "LHnV" represents algorithm LHPITCHSPELL$_n$, run on the set of 'end-to-end' **NList**s. For example, the code "LH4V" represents LHPITCHSPELL$_4$, run on the set of 'end-to-end' **NList**s. Each entry in Table 2.4 gives the percentage increase in the note error rate caused by changing from the algorithm at the head of the entry's row to the algorithm at the

Algorithm	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
LH1V	1045	334	74	41	1202	263	210	339	3508
LH2V	1045	390	74	41	1252	263	210	339	3614
LH3V	1045	390	74	41	1252	263	210	339	3614
LH4V	90	2270	74	192	2827	1450	119	1720	8742
LH1	116	890	102	681	2662	3689	418	1024	9582
LH2	116	890	102	681	2662	3689	418	1024	9582
LH3	116	890	102	681	2662	3689	418	1024	9582
LH5V	90	3458	74	192	3939	1450	119	1720	11042
LH6V	90	3458	74	192	3939	1450	119	1720	11042
LH4	199	1913	110	681	6192 (3220)	5319	418	1630	16462 (13490)
LH5	199	1913	110	681	6192 (3220)	5319	418	1630	16462 (13490)
LH6	199	1913	110	681	6192 (3220)	5319	418	1630	16462 (13490)

Table 2.2: Note error counts for algorithms in column labelled "Algorithm" for the complete test corpus (last column), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus. The algorithms are sorted into increasing order of overall note error count. See text and sections 1.3.4 and 1.3.6 for further details.

Algorithm	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\overline{\text{NA}}$	SD_{Sty}
LH1V	95.74	98.64	99.70	99.83	95.09	98.93	99.14	98.62	98.21	98.21	1.79
LH2V	95.74	98.41	99.70	99.83	94.89	98.93	99.14	98.62	98.16	98.16	1.84
LH3V	95.74	98.41	99.70	99.83	94.89	98.93	99.14	98.62	98.16	98.16	1.84
LH4V	99.63	90.73	99.70	99.22	88.46	94.08	99.51	92.98	95.54	95.54	4.56
LH1	99.53	96.37	99.58	97.22	89.13	84.94	98.29	95.82	95.11	95.11	5.28
LH2	99.53	96.37	99.58	97.22	89.13	84.94	98.29	95.82	95.11	95.11	5.28
LH3	99.53	96.37	99.58	97.22	89.13	84.94	98.29	95.82	95.11	95.11	5.28
LH5V	99.63	85.88	99.70	99.22	83.92	94.08	99.51	92.98	94.37	94.36	6.43
LH6V	99.63	85.88	99.70	99.22	83.92	94.08	99.51	92.98	94.37	94.36	6.43
LH4	99.19	92.19	99.55	97.22	74.72 (86.86)	78.28	98.29	93.35	91.60 (93.12)	91.60 (93.12)	9.73 (7.39)
LH5	99.19	92.19	99.55	97.22	74.72 (86.86)	78.28	98.29	93.35	91.60 (93.12)	91.60 (93.12)	9.73 (7.39)
LH6	99.19	92.19	99.55	97.22	74.72 (86.86)	78.28	98.29	93.35	91.60 (93.12)	91.60 (93.12)	9.73 (7.39)

Table 2.3: Note accuracies expressed as percentages for each algorithm in the column headed "Algorithm" for the complete test corpus (column 10), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The columns headed $\overline{\text{NA}}$ and SD_{Sty} give the mean and standard deviation, respectively, of the values in columns 2 to 9. The algorithms are sorted into decreasing order of overall note accuracy. See text and sections 1.3.4 and 1.3.6 for further details.

	LH1V	LH2V	LH3V	LH4V	LH1	LH2	LH3	LH5V	LH6V	LH4	LH5	LH6
LH1V	0.00	3.02	3.02	149.20	173.15	173.15	173.15	214.77	214.77	369.27 (284.55)	369.27 (284.55)	369.27 (284.55)
LH2V	-2.93	0.00	0.00	141.89	165.14	165.14	165.14	205.53	205.53	355.51 (273.27)	355.51 (273.27)	355.51 (273.27)
LH3V	-2.93	0.00	0.00	141.89	165.14	165.14	165.14	205.53	205.53	355.51 (273.27)	355.51 (273.27)	355.51 (273.27)
LH4V	-59.87	-58.66	-58.66	0.00	9.61	9.61	9.61	26.31	26.31	88.31 (54.31)	88.31 (54.31)	88.31 (54.31)
LH1	-63.39	-62.28	-62.28	-8.77	0.00	0.00	0.00	15.24	15.24	71.80 (40.78)	71.80 (40.78)	71.80 (40.78)
LH2	-63.39	-62.28	-62.28	-8.77	0.00	0.00	0.00	15.24	15.24	71.80 (40.78)	71.80 (40.78)	71.80 (40.78)
LH3	-63.39	-62.28	-62.28	-8.77	0.00	0.00	0.00	15.24	15.24	71.80 (40.78)	71.80 (40.78)	71.80 (40.78)
LH5V	-68.23	-67.27	-67.27	-20.83	-13.22	-13.22	-13.22	0.00	0.00	49.09 (18.15)	49.09 (18.15)	49.09 (18.15)
LH6V	-68.23	-67.27	-67.27	-20.83	-13.22	-13.22	-13.22	0.00	0.00	49.09 (18.15)	49.09 (18.15)	49.09 (18.15)
LH4	-78.69 (-74.00)	-78.05 (-73.21)	-78.05 (-73.21)	-46.90 (-35.20)	-41.79 (-28.97)	-41.79 (-28.97)	-41.79 (-28.97)	-32.92 (-18.15)	-32.92 (-18.15)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
LH5	-78.69 (-74.00)	-78.05 (-73.21)	-78.05 (-73.21)	-46.90 (-35.20)	-41.79 (-28.97)	-41.79 (-28.97)	-41.79 (-28.97)	-32.92 (-18.15)	-32.92 (-18.15)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
LH6	-78.69 (-74.00)	-78.05 (-73.21)	-78.05 (-73.21)	-46.90 (-35.20)	-41.79 (-28.97)	-41.79 (-28.97)	-41.79 (-28.97)	-32.92 (-18.15)	-32.92 (-18.15)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)

Table 2.4: Each entry in this table gives the percentage increase in the note error rate caused by changing from the algorithm at the head of the entry's row to the algorithm at the head of the entry's column. A negative value means that the note error rate was reduced by the change in algorithm. See text and section 1.3.4 for further details.

head of the entry's column. For example, changing from 'end-to-end' **NList**s to 'interleaved' ones increased the note error rate for LHPitchSpell$_4$ by 88.31%. More details are given in the table captions. The algorithms are arranged in these tables in non-increasing order of note accuracy.

As discussed in section 1.3.4.3 above, because of the sudden enharmonic change at bar 166 in the fourth movement of Haydn's Symphony No. 100 in G ('Military') (Hob. I:100), the output of each algorithm was compared with two "correct" versions of this movement: one in which the notes are spelt as they are in the original score; and a second, modified version, in which all the notes in the original score up to bar 165 are transposed down a diminished second. Where an algorithm performed better on the modified version than on the original, the results obtained with the modified version have been given in parentheses in Tables 2.2, 2.3 and 2.4.

When the 'derestricted' algorithms, LHPitchSpell$_{4–6}$, were run on the interleaved **NList**s, they all generated a spelling for the fourth movement of Haydn's 'Military' Symphony in which 3007 notes were assigned pitch names different from those in the original score (i.e., 52.90% correct) but only 35 notes were assigned names different from those in the modified version (i.e., 99.45% correct). All the other versions of the algorithm made more errors when compared with the modified version than when compared with the original score. The modified "correct" version of the fourth movement of Haydn's 'Military' Symphony has passages in extreme keys (it begins in F𝄪 major and ends in G major). Therefore it is not surprising that the modified correct spelling for this movement was more similar to the output of the derestricted algorithms than it was to the output of the restricted algorithms when the **NList**s were interleaved. However, when the derestricted algorithms were run on the 'end-to-end' **NList**s, 3752 notes were assigned pitch names that were different from those in the modified version of the movement (i.e., 41.23% correct). This is also not surprising when one remembers that in the 'end-to-end' **NList** representation of the modified spelling of this movement, there is a sudden enharmonic change at the beginning of each voice (apart from the first) in which the key suddenly changes from G major at the end of one voice to F𝄪 major at the beginning of the next. In other words, in this 'end-to-end' representation of the modified version of the movement, there are many more sudden enharmonic changes than in the 'interleaved' representation of the original score. Consequently, the derestricted versions of the algorithm, which do not attempt to predict such enharmonic changes but rather attempt only to represent the key changes that the listener actually hears, make many more note errors on the 'end-to-end' representation of the modified version of this movement than they do on the 'interleaved' representation.

The next observation to be made is that, for both the 'end-to-end' and the 'interleaved' **NList**s, the note error counts for LHPitchSpell$_2$ were the same as those for LHPitchSpell$_3$ and the note error counts for LHPitchSpell$_5$ were the same as those for LHPitchSpell$_6$ (compare LH2 with LH3, LH2V with LH3V, LH5 with LH6, and LH5V with LH6V in Table 2.2). It was confirmed that these identical note error counts were not coincidental—LHPitchSpell$_2$ made exactly the same errors as LHPitchSpell$_3$ and LHPitchSpell$_5$ made exactly the same errors as LHPitchSpell$_6$. This implies that the variable m in the Intervals function (Figure 2.7) was never equal to 2 in line 38 of this function when LHPitchSpell$_1$ was run on the test corpus. This is not surprising given that m was never equal to 2 in line 38 of this function

when LHPitchSpell was run on all possible sequences between 3 and 7 notes in length (see section 2.3.3.2.1 above). It would therefore seem that a correct implementation of Longuet-Higgins's Rule 2 can be achieved simply by changing line 12 in Figure 2.3 to that shown in Figure 2.9 and lines 29–31 and lines 40–42 in Intervals to those shown in Figure 2.10. That is, Longuet-Higgins's Rule 2 would seem to be correctly implemented in the Intervals$_2$ algorithm (see Figure 2.21).

However, correcting the implementation of Longuet-Higgins's Rule 2 had no effect on note accuracy when the algorithms were run on the interleaved **NList**s (compare LH1 with LH2 and LH4 with LH5 in Table 2.2). When run on the end-to-end **NList**s, correcting the implementation of Longuet-Higgins's Rule 2 in the restricted algorithm (i.e., going from LH1V to LH2V) actually increased the note error count by 16.77% for Beethoven and by 4.16% for Haydn but otherwise had no effect. Similarly, correcting the Rule 2 implementation in the derestricted algorithm (i.e., changing from LH4V to LH5V), increased the note error count by 52.33% for Beethoven and 39.33% for Haydn but otherwise had no effect. There would therefore seem to be no point in modifying LHPitchSpell so that Longuet-Higgins's Rule 2 is more accurately implemented.

Why should correcting the implementation of Longuet-Higgins's Rule 2 cause a decrease in note accuracy? Also, why should it only affect the note accuracies for the music of Beethoven and Haydn and then only when the notes in the data are sorted so that the voices are arranged 'end-to-end'? The only difference between LHPitchSpell$_1$ and LHPitchSpell$_2$ is that LHPitchSpell$_2$ uses Intervals$_2$ (Figure 2.21) instead of the Intervals algorithm (Figure 2.7) used in LHPitchSpell$_1$. And the only difference between Intervals$_2$ and Intervals is that, in Intervals$_2$, lines 30 and 41 execute when $m = -1$. It can be proved that m is never equal to -1 in line 29 of Intervals$_2$ since this would imply that the two previous notes had the same keyboard position which is not possible because the Simplify algorithm removes all such cases in the first line of Intervals$_2$. m is only equal to -1 in line 40 of Intervals$_2$ when the sequence of scale degrees $\langle \sharp\hat{4}, \hat{4}, \sharp\hat{4} \rangle$ occurs in the input, and, when this occurs, this triggers a change of key which results in this sequence being reinterpreted as $\langle \hat{5}, \sharp\hat{4}, \hat{5} \rangle$ in the new key. In other words, the only difference between LHPitchSpell$_1$ and LHPitchSpell$_2$ is that the sequence $\langle \sharp\hat{4}, \hat{4}, \sharp\hat{4} \rangle$ is reinterpreted as $\langle \hat{5}, \sharp\hat{4}, \hat{5} \rangle$ in LHPitchSpell$_2$ but not in LHPitchSpell$_1$. This reinterpretation implies at least a local tonicization of the leading note or the sharpened subdominant which is extremely unlikely. It is therefore not surprising that such reinterpretations often led to a note being mis-spelt. It is also not really surprising that the sequence $\langle \sharp\hat{4}, \hat{4}, \sharp\hat{4} \rangle$ never occurs in the interleaved **NList**s as one would not expect such a sequence of intervals to occur between adjacent notes within a chord or between the highest notes in one chord and the lowest notes in the next.

A more detailed analysis shows that correcting the implementation of Longuet-Higgins's Rule 2 only affects the pitch names in two of the movements in the entire test corpus: the second movement of Beethoven's Symphony No. 1 in C major (Op. 21) published in 1801; and the second movement of Haydn's Symphony No. 102 in B♭ major (Hob, I:102) composed in 1794–5. For the restricted algorithm, LHPitchSpell$_1$, correcting the Rule 2 implementation corrected none of the errors made by LHPitchSpell$_1$ and caused an extra 56 errors to be made in the Beethoven movement and an extra 50 errors to be made in the Haydn movement. Interestingly, both

the Beethoven and Haydn movements are slow, second movements in F major in symphonies composed at around the same time—sketches suggest that Beethoven started composing his first symphony in 1795–6 (Kerman et al., 2004), shortly after he had received some tuition from Haydn. Moreover, in both movements, the extra errors introduced by correcting the Rule 2 implementation in LHPitchSpell$_1$ are all contained within a segment of a few bars where there is a chromatically embellished dominant pedal that leads into the main recapitulation of the opening theme in the home key.

Correcting the Rule 2 implementation in the derestricted algorithm, LHPitchSpell$_4$, corrected 871 of the 887 errors made by LHPitchSpell$_4$ in the Beethoven movement but then caused 2059 new errors to be made. In the Haydn movement, correcting LHPitchSpell$_4$ corrected 29 of the 42 errors made by this algorithm but then caused 1141 new errors to be made. In contrast to the results obtained with the restricted algorithm, the errors introduced by correcting the Rule 2 implementation in LHPitchSpell$_4$ are distributed throughout the two movements, not localised to one small segment.

As discussed above, Longuet-Higgins's (1987a, p. 114) warning against using his algorithm on polyphonic music leads us to expect it to work better on the end-to-end **NList**s than the interleaved **NList**s, and, overall, this was indeed found to be the case: changing from interleaved to end-to-end **NList**s substantially reduced the overall note error count for all the algorithms, reducing it by over 62% for LHPitchSpell$_{1–3}$, about 47% for LHPitchSpell$_4$ and about 33% for LHPitchSpell$_{5–6}$ (see entries in Table 2.4 for changing from LHn to LHnV). Indeed, for the restricted algorithms (i.e., LHPitchSpell$_{1–3}$) changing from interleaved to end-to-end **NList**s eliminated over 90% of the errors for Handel and Mozart.

This makes it all the more remarkable that, for the same restricted algorithms, LHPitchSpell$_{1–3}$, changing from interleaved to end-to-end **NList**s *increased* the number of errors on Bach's music *ninefold* from 116 to 1045. However, it was found that this increase was entirely due to an increase from 19 to 977 errors (a drop in note accuracy from 99.04% to 50.63%) in the second movement from the Church Cantata, "Ich geh und suche mit Verlangen", BWV 49 (first performed in 1726). The total note error count over the remaining movements by Bach actually dropped by 29.90% when the voices in the **NList**s were changed from being interleaved to being arranged end-to-end. The large drop in note accuracy observed in the second movement of BWV 49 was primarily due to almost every note in one voice being spelt a diminished second below its correct spelling. This sort of error results when the key is incorrectly determined at the beginning of a voice and the algorithm goes on to spell the notes so that the intervals between consecutive intervals within the voice are correct. This sort of error is only likely to happen when the voices are arranged end-to-end since, when they are interleaved, the algorithm ensures that intervals between notes in different voices are spelt correctly.

Another important observation to make is that 'derestricting' the algorithms so that they are not constrained to assigning pitch names between G𝄫 and A𝄪 on the line of fifths severely increased the overall note error count. For the interleaved **NList**s, Table 2.4 shows that derestricting the algorithms increased the note error count by 71.80% (see entries for changing from LH1 to LH4, LH2 to LH5 and LH3 to LH6). For the end-to-end **NList**s, derestricting the algorithms had an even worse effect on the note error count: changing from LH1V to LH4V

increased the note error count by about $2\frac{1}{2}$ times (149.20%); and the note error count was more than tripled (205.53% increase) when LH2V was changed to LH5V. It would seem that constraining the pitch names to being within a restricted range on the line of fifths in LHPitchSpell$_{1–3}$ generally prevents the algorithms from allowing the tonality to drift along the line of fifths into extreme keys.

However, in specific cases, this constraint can cause incorrect enharmonic changes that can lead to whole segments being spelt a diminished second away from their correct spelling. As just discussed above, most of the errors made by LHPitchSpell$_1$ on Bach when it was run on the end-to-end **NList**s were the result of a single voice in the second movement of BWV 49 being spelt a diminished second below its correct spelling. This error resulted from the key being incorrectly determined at the beginning of the voice. By derestricting LHPitchSpell$_1$ to produce LHPitchSpell$_4$, this incorrect key determination was corrected and the note error count for this movement was reduced from 977 to 22. This suggests that the incorrectly determined key which resulted in the mis-spelt voice when LHPitchSpell$_1$ was run on the second movement of BWV 49 was the result of an enharmonic change triggered by a correct pitch name being outside the permitted range on the line of fifths.

An examination of the values of $\mathrm{SD}_{\mathrm{Sty}}(A, \mathcal{C})$ in the final column of Table 2.3 reveals that, for the versions of Longuet-Higgins's algorithm tested here, the style dependence increases monotonically (indeed, almost linearly) with the note error rate.

2.5 Summary and conclusions

In this chapter I have provided a detailed analysis and evaluation of the pitch spelling algorithm implemented in Longuet-Higgins's (1976, 1987a, 1993) `music.p` program. More specifically, I recast Longuet-Higgins's (1987a, pp. 112–114) theory of tonality as six rules, re-expressed his algorithm in pseudocode and then explained precisely how he had implemented the rules in the algorithm. This analysis revealed that one of the rules in Longuet-Higgins's theory, Rule 2, had not been accurately implemented in the algorithm. Two different modifications were proposed for correcting the implementation of Rule 2. It was also observed that the pitch names generated by Longuet-Higgins's algorithm were arbitrarily restricted to being between G𝄫 and A𝄪 on the line of fifths. An evaluation was then carried out which was designed to explore the effects on note accuracy of the two proposed Rule 2 implementation corrections and of derestricting the algorithm so that it was capable of generating any possible pitch name. Another goal of this evaluation was to test Longuet-Higgins's (1987a, p. 114) claim that his algorithm was not appropriate for processing polyphonic music. To this end, six versions of the algorithm were run on two versions of the test corpus $\mathcal{C}$ defined in Table 1.4. In one version of the test corpus used, the notes were sorted so that the voices were arranged 'end-to-end'; in the other version, the notes were sorted so that the voices were 'interleaved'. The main results of this evaluation were that

1. the two Rule 2 implementation corrections proposed had exactly the same effect;
2. correcting the Rule 2 implementation had no effect when the voices were interleaved and increased the number of errors when the voices were arranged end-to-end;

3. the number of errors made when the voices were end-to-end was over 62% less than when the voices were interleaved for the restricted algorithms, and 33-47% less for the derestricted algorithms, supporting Longuet-Higgins's prediction that the algorithm would work less well on polyphonic music;

4. removing the restriction that pitch names must be between G𝄫 and A𝄪 on the line of fifths more than doubled the note error rate when the voices were end-to-end and increased it by over 70% when the voices were interleaved; and

5. the style dependence of the versions of the algorithm tested increased monotonically with note error rate so that the most accurate version of the algorithm was also the one whose accuracy was least affected by musical style.

The results of this study therefore indicate that the original version of Longuet-Higgins's algorithm implemented in `music.p` is both more accurate and less affected by style than the other versions of the algorithm tested here. When tested on the test corpus $\mathcal{C}$ defined in Table 1.4 with the voices arranged end-to-end, this original version of the algorithm spelt 98.21% of the notes correctly and its style dependence (i.e., $\mathrm{SD}_{\mathrm{Sty}}(A, \mathcal{C})$) was 1.79. When the voices were interleaved, the algorithm spelt 95.11% of the notes correctly and its style dependence was 5.28.

Chapter 3

Cambouropoulos's pitch spelling algorithms

3.1 Introduction

Cambouropoulos (1996, 1998, 2001, 2003) has published descriptions of three different variants of his pitch spelling algorithm. In all three of these variants, it is assumed that the passage of music to be processed has been represented as a MIDI file and that a sequence of MIDI note numbers has been derived from this file in which the ordering of the note numbers corresponds to the order in which the notes occur in the input passage (notes that begin simultaneously being ordered arbitrarily). This sequence of MIDI note numbers is then processed using a "shifting overlapping windowing technique" (Cambouropoulos, 2003, p. 420), in which each window contains a certain number (typically 9 or 12) of contiguous elements in the input sequence (see Figure 3.1). Each window is positioned so that the first two thirds of the window overlap the last two thirds of the previous window. In other words, the window position advances each time by a third of the window size. Cambouropoulos adopted a windowing technique in order that the overall running time of the algorithm should be linear rather than exponential in the size of the input sequence (Cambouropoulos, 1996, pp. 244–245). He chose to use overlapping windows in order to avoid abrupt changes (e.g., from flat to sharp pitch names) at window boundaries (Cambouropoulos, 2003, p. 420).

Cambouropoulos allows 'white note' pitch classes (i.e., 0, 2, 4, 5, 7, 9 and 11) to be spelt in three different ways. For example, pitch class 0 can be spelt as B♯, C♮ or D𝄫. He allows 'black note' pitch classes to be spelt in two different ways. For example, pitch class 6 can be spelt as

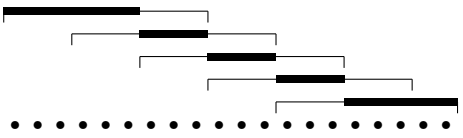


Figure 3.1: Cambouropoulos's own caption to this figure reads: "Shifting overlapping window technique. For each window only the middle section of spelled pitches (bold line) is retained. Dots represent the pitches of the input sequence." (Reproduced (with minor corrections and alterations) from Cambouropoulos, 2003, p. 420, Fig. 6 and Cambouropoulos, 1996, p. 245, Fig. 8.)

F♯ or G♭ (see, for example, Cambouropoulos, 1996, p. 242). Cambouropoulos's method involves computing all the possible spellings for each window that are permitted given the restricted sets of possible pitch name classes just described and given various other restrictions specific to the particular version of the algorithm used (see, for example, Cambouropoulos, 2003, p. 421).

A penalty score is then computed for each of these possible window spellings. The penalty score for a given window spelling is found by computing a penalty value for each pitch interval in the window and summing these interval penalty values. A given interval in a particular window spelling is penalised more heavily the less frequently it occurs in the major and minor scales, a principle that Cambouropoulos (2003, p. 421) calls *interval optimization*. An interval is also penalised if either of the pitch names forming the interval is a double-sharp or a double-flat, a principle that Cambouropoulos (2003, p. 421) calls *notational parsimony*. For each window, the algorithm chooses the spelling that has the lowest penalty score and appends the pitch name classes in the middle third of this best window spelling to the end of the sequence of pitch name classes that the algorithm eventually generates as output.

The earliest published version of this method (Cambouropoulos, 1996, 1998) was designed for processing monophonic melodies and includes a rule, based on Krumhansl's (1990, pp. 150–151) principle of contextual asymmetry, that is applied as a 'tie-breaker' when two or more spellings for a given window achieve the least penalty score. The two more recent published versions (Cambouropoulos, 2001, 2003) are adapted for processing polyphonic music. However, they differ from each other in that the earlier of the two (Cambouropoulos, 2001, p. 5) uses a "variable length window" which will be described in more detail in section 3.3 below.

Unlike Longuet-Higgins, Cambouropoulos has not published the source code of any of his own implementations of his pitch spelling method. He was also unable to provide me with this source code when I requested it. I therefore had to implement the algorithms myself in order to test them. In the next three sections, I describe in detail my own implementations of the published versions of Cambouropoulos's method, starting with the most recent (and simplest).

3.2 The algorithm described by Cambouropoulos (2003)

3.2.1 The CAM03 algorithm

In this section I present my own implementation, expressed in pseudocode, of Cambouropoulos's (2003) most recent published description of his pitch spelling method. The complete algorithm, which I call CAM03, is given in Figure 3.2.

Cambouropoulos (2003, p. 420) assumes that the input to his algorithm is a sequence of MIDI note numbers in the order in which they appear in a MIDI file. In CAM03, this input sequence is implemented by the ordered set of MIDI note numbers called **MidiList** (see Figure 3.2). Cambouropoulos (2003, pp. 411, 414) states that the algorithm outputs "a sequence of 'correctly' spelled pitches", although it actually only assigns a pitch name *class* to each note. CAM03 therefore returns a sequence of pitch name classes, **PNCList**, in which the ith element is the pitch name class assigned by the algorithm to the note represented by the ith MIDI note number in **MidiList** (see line 12 in Figure 3.2). The complete pitch name of each note (i.e., including octave number) can be computed from its pitch name class and MIDI note number using the

```
CAM03(MidiList, WinSize)
1    MidiListSize ← |MidiList|
2    LastWindow ← false
3    FirstWindow ← true
4    WinStart ← 0
5    PNCList ← ⟨ ⟩
6    while |PNCList| < MidiListSize
7        WinEnd ← MIN({ WinStart + WinSize, MidiListSize})
8        if WinEnd = MidiListSize
9            LastWindow ← true
10       if FirstWindow
11           WinPNCList ← ⟨ ⟩
12           WinMidiList ← MidiList[WinStart, WinEnd]
13       else
14           WinPNCList ← PNCList[WinStart, WinStart + (WinSize/3)]
15           WinMidiList ← MidiList[WinStart + (WinSize/3), WinEnd]
16       PNCList ← PNCList ⊕ SPELLWIN03(WinPNCList, WinMidiList,
17                                      FirstWindow, LastWindow)
18       FirstWindow ← false
19       WinStart ← WinStart + (WinSize/3)
20   return PNCList
```

Figure 3.2: Cambouropoulos's (2003) pitch spelling algorithm expressed in pseudocode.

algorithm in Figure 1.16.

As mentioned in section 3.1 above, the algorithm uses a "shifting overlapping windowing technique" (Cambouropoulos, 2003, p. 420) as illustrated in Figure 3.1. In CAM03, each window contains a fixed number of contiguous elements of **MidiList** and this number is provided by the user in the argument *WinSize*. Cambouropoulos (2003, p. 420) recommends that this value should be set to either 9 or 12, but points out that allowing a larger window

> gives greater stability to the pitch-spelling process because a larger pitch context is taken into account and abrupt changes at the edges of the window are avoided.

In line 1 of CAM03 (see Figure 3.2), the length of the sequence **MidiList** is stored in the variable *MidiListSize* so that this quantity can be accessed in constant time later in the algorithm. If the programming language used to implement the algorithm allows for the length of a list to be accessed in constant time (i.e., without having to traverse the list) then this step is unnecessary.

On each iteration of the 'while' loop in lines 6–19, the best spelling for a particular window is computed and the pitch name classes for a segment of the window are retained and appended to **PNCList**, which is initialised to equal the empty sequence in line 5. The best spelling for each window is computed in line 16 using the function SPELLWIN03 which is defined in Figure 3.4. Figure 3.1 shows that, for the first window, the pitch name classes for the first two thirds of the window are retained, whereas, for all other windows apart from the last, only the middle third of each window is retained. For the last window, all pitch name classes are retained starting with the $(1 + WinSize/3)$th. The portion of the best spelling for a window returned by SPELLWIN03 therefore depends on whether it is the first window, the last window or another window. The variables *FirstWindow* and *LastWindow* are used to flag whether the current window is the first

window or last window, respectively. These variables are passed as arguments to SpellWin03 and determine the portion of the best spelling for the current window that this function returns. *FirstWindow* is initialised to true in line 3 and then set to false in line 18 after the first window has been processed. *LastWindow* is initialised to false in line 2 and set to true in line 9 when the end of the current window coincides with the end of **MidiList**.

The variables *WinStart* and *WinEnd*, (see lines 4, 7 and 19) are used to keep track of the position of the window that is currently being processed. *WinStart* is used to store the index of the first element in the current window and *WinEnd* is set to be one greater than the index of the last element in the current window. *WinStart* is initialised to zero in line 4. On each iteration of the 'while' loop, *WinEnd* is set for the current window in line 7 and then, after the window has been processed, *WinStart* is advanced by one third of *WinSize* in line 19 so that the first two thirds of the new window overlap the last two thirds of the previous window. In line 7, if the number of elements in **MidiList** following the $(WinStart + 1)$th element is not enough to fill a window of size *WinSize*, then *WinEnd* is set to equal the length of **MidiList**. If this happens, then the current window must be the last window so *LastWindow* is set to true in line 9.

Cambouropoulos (2003, p. 422) states that

> in order to provide more stability to the transcription process... in each window, the first three [i.e., $WinSize/3$] notes retain the spellings defined in the immediately previous window... that is, for these pitches, the spellings are fixed and all possible pitch spellings are only tried out for the remaining six [i.e., $2\,WinSize/3$] pitches.

Therefore, on the second and subsequent iterations of the 'while' loop in lines 6–19 of Cam03, the pitch name classes for the first third of the window (which were computed on the previous iteration) are stored in the ordered set **WinPNCList** in line 14 and the MIDI note numbers for the remainder of the current window are stored in the ordered set **WinMidiList** in line 15. These two ordered sets are then passed as arguments to the SpellWin03 function in line 16. Obviously, for the first window, **WinPNCList** must be empty (see line 11) and **WinMidiList** must contain the MIDI note numbers for the entire window (see line 12).

The 'while' loop in lines 6–19 continues to iterate until **PNCList** contains a pitch name class for each MIDI note number in **MidiList** (i.e., $|\mathbf{PNCList}| = MidiListSize$ in line 6) and then **PNCList** is returned in line 20.

3.2.2 The SpellWin03 algorithm

Let us now look more closely at the SpellWin03 function, called in line 16 of Cam03 and defined in Figure 3.4. SpellWin03 is my own implementation of the method used by Cambouropoulos (2003, pp. 421–422) to obtain the best spelling for a given window. This function takes four arguments:

1. **WinPNCList** is an ordered set containing the pitch name classes assigned in the previous window to the notes in the first third of the current window;

2. **WinMidiList** is an ordered set containing the MIDI note numbers for the last two thirds of the current window;

3. *FirstWindow* is a boolean flag whose value indicates whether the current window is the first window; and

4. *LastWindow* is a boolean flag whose value indicates whether the current window is the last window.

Cambouropoulos (2003, p. 421) states that "[f]or each window, *all* possible spelling sequences are computed", however,

> sequences that contain *both* double sharps *and* double flats are disallowed by creating the different spelling sequences only within two different tonal-pitch-class [i.e., pitch name class] areas wherein one area excludes double sharps and the other excludes double flats.
>
> (Cambouropoulos, 2003, p. 421)

These two pitch name class regions are shown in Figure 3.3 (cf. Cambouropoulos, 2003, p. 422, Fig. 7). I call the region that contains double flats, the flatside region, and the region that contains double sharps, the sharpside region. Figure 3.3 shows that, within each of these regions, there are always two possible pitch name classes for each pitch class. For example, in the sharpside region, pitch class 0 can be spelt as either `"Bs"` or `"Cn"`. This implies that the total number of possible sharpside spellings for a sequence of n MIDI note numbers is 2^n. As the pitch name classes for the first third of each window (apart from the first) are fixed when the previous window is spelt, the total number of spellings for each window is $2 \times 2^{2\,WinSize/3} = 2^{1+(2\,WinSize/3)}$. So, for a window size of 9, computing the best spelling for a single window involves computing the penalty score for $2^7 = 128$ different window spellings. Cambouropoulos's (2003, p. 421) claim that $2^{10} = 1024$ spellings have to be computed for each 9-note window is based on the assumption that the first third of each window is not retained from the previous window. As he points out (Cambouropoulos, 2003, p. 422), retaining the spellings for the first third of each window from the previous window not only makes it "more difficult for the algorithm to flip over from the flat region to the sharp region" but also improves the time complexity of the algorithm "as the actual number of pitches to be spelled is decreased in each window".

As each possible window spelling is constrained to being within either the flatside or sharpside region and as, within each of these regions, each pitch class can be spelt in two different ways, each possible spelling for a MIDI note number sequence of length n can be uniquely specified by

1. stating whether the spelling is flatside or sharpside and

2. giving a bit vector of length n (corresponding to a number between 0 and $2^n - 1$) in which each element (0 or 1) indicates one of the two possible pitch name classes for a particular MIDI note number (for example, a zero could always indicate the 'sharper' of the two possible pitch name classes).

For example, if we have the MIDI note number sequence

$$\langle 60, \quad 63, \quad 67, \quad 68, \quad 59, \quad 67 \rangle ,$$

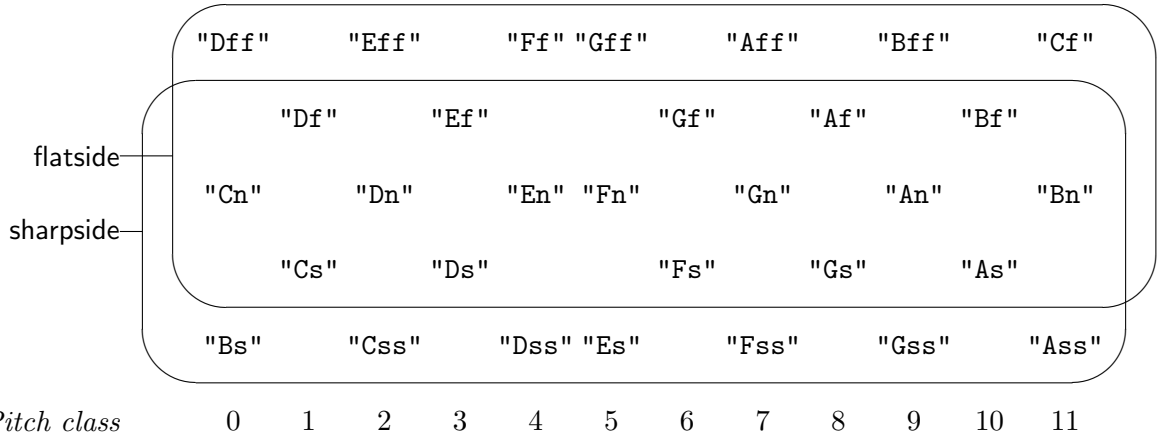


Figure 3.3: All the pitch name classes in any single window spelling must be completely contained within one of the two regions in this figure. The region that contains double sharps is the sharpside region, the other is the flatside region.

then the flatside spelling corresponding to the bit vector $\langle 0, 0, 1, 0, 1, 1 \rangle$ (i.e., the decimal integer 11, expressed in binary with two padding zeros) would be

$$\langle \texttt{"Cn"}, \quad \texttt{"Ds"}, \quad \texttt{"Aff"}, \quad \texttt{"Gs"}, \quad \texttt{"Cf"}, \quad \texttt{"Aff"} \rangle$$

assuming, as suggested above, that a zero indicates the sharper of the two possible spellings for a given MIDI note number. All the possible spellings for a window can therefore be generated by

1. computing the bit vector representation of length $|\mathbf{WinMidiList}|$ for each integer between 0 and $2^{|\mathbf{WinMidiList}|} - 1$, then
2. finding the flatside and sharpside spellings of **WinMidiList** corresponding to each of these bit vectors, and
3. concatenating each of these two spellings onto **WinPNCList** for each bit vector.

The total penalty score for each spelling can be computed immediately after it has been generated and the spelling that achieves the least penalty score can be stored and returned once all the spellings have been evaluated.

SpellWin03 (see Figure 3.4) implements the strategy just described. In line 1, the length of **WinMidiList** is stored in the variable *WinMidiListSize* so that it can be accessed in constant time later on in the algorithm. If the algorithm is implemented in a programming language in which the length of a list can be accessed in constant time without traversing the list then this step is unnecessary (cf. discussion above of *MidiListSize* variable in Cam03). Next, in line 2, the variable *MaxBitVecInt* is made equal to $2^{WinMidiListSize} - 1$, this being the maximum integer for which a bit vector representation has to be generated. In line 3, the variable **BestSpelling** is initialised to the empty sequence, $\langle\,\rangle$. The variables **BestSpelling** and *BestPenalty* are used throughout the function to store, respectively, the spelling of **WinMidiList** that has achieved the least penalty score so far and the penalty score for this best spelling.

```
SpellWin03(WinPNCList, WinMidiList, FirstWindow, LastWindow)
1   WinMidiListSize ← |WinMidiList|
2   MaxBitVecInt ← 2^WinMidiListSize − 1
3   BestSpelling ← ⟨ ⟩
4   for i ← 0 to MaxBitVecInt
5       BitVec ← BitVector(i, WinMidiListSize)
6       Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
7       SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
8       if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
9           BestSpelling ← Spelling
10          BestPenalty ← SpellingPenalty
11      Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
12      SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
13      if SpellingPenalty < BestPenalty
14          BestSpelling ← Spelling
15          BestPenalty ← SpellingPenalty
16  ▶ Now return appropriate part of BestSpelling.
17  BSEnd ← WinMidiListSize/2
18  if FirstWindow
19      BSEnd ← 2WinMidiListSize/3
20  if LastWindow
21      BSEnd ← WinMidiListSize
22  return BestSpelling[0, BSEnd]
```

Figure 3.4: The SpellWin03 algorithm.

In the 'for' loop in lines 4–15 of SpellWin03, for each integer i from 0 to *MaxBitVecInt*, the bit vector representation of i of length *WinMidiListSize* is computed and stored in the variable **BitVec** (line 5). This is done using the function BitVector, defined in Figure 1.10 above. The flatside window spelling corresponding to **BitVec** is then computed and stored in the variable **Spelling** (line 6) and the total penalty score for this window spelling is calculated and stored in the variable *SpellingPenalty* (line 7). Each spelling is computed using the ComputeSpelling algorithm defined in Figure 3.5 and the penalty score for each spelling is computed using the ComputeSpellPen03 function defined in Figure 3.6. On the first iteration of the 'for' loop, **BestSpelling** is equal to the empty sequence, $\langle\,\rangle$, in line 8, so in lines 9 and 10, **BestSpelling** and *BestPenalty* are set to equal the initial values of **Spelling** and *SpellingPenalty*, respectively. On subsequent iterations, **BestSpelling** and *BestPenalty* are set to equal **Spelling** and *SpellingPenalty* in lines 9 and 10 only if the value in *SpellingPenalty* is less than that currently stored in *BestPenalty* in line 8. Then the sharpside window spelling corresponding to **BitVec** is computed and stored in **Spelling** (line 11), and the total penalty score for this spelling is calculated and stored in *SpellingPenalty* (line 12). Again, if the penalty score in *SpellingPenalty* is less than that in *BestPenalty* (line 13), this implies that the sharpside window spelling just computed in line 11 is superior to that currently stored in **BestSpelling**, so **BestSpelling** is set to equal **Spelling** and *BestPenalty* is set to equal *SpellingPenalty* (lines 14–15). When the 'for' loop terminates, the variable **BestSpelling** contains the first window spelling computed that achieves the minimum total penalty score. Note that it is possible for more than one spelling to achieve this minimum score. However, in his most recent description of the algorithm, Cambouropoulos does not suggest any way of deciding between these different 'best' window spellings (cf. discussion in section 3.4 below of the 'tie-breaker' rule in the earliest version of the method

```
ComputeSpelling(WinMidiList, BitVec, SharpOrFlatSide)
1    if SharpOrFlatSide = sharpside
2       PNCs ← ⟨⟨"Bs", "Cn"⟩, ⟨"Cs", "Df"⟩, ⟨"Css", "Dn"⟩, ⟨"Ds", "Ef"⟩,
                ⟨"Dss", "En"⟩, ⟨"Es", "Fn"⟩, ⟨"Fs", "Gf"⟩, ⟨"Fss", "Gn"⟩,
                ⟨"Gs", "Af"⟩, ⟨"Gss", "An"⟩, ⟨"As", "Bf"⟩, ⟨"Ass", "Bn"⟩⟩
3    else
4       PNCs ← ⟨⟨"Cn", "Dff"⟩, ⟨"Cs", "Df"⟩, ⟨"Dn", "Eff"⟩, ⟨"Ds", "Ef"⟩,
                ⟨"En", "Ff"⟩, ⟨"Fn", "Gff"⟩, ⟨"Fs", "Gf"⟩, ⟨"Gn", "Aff"⟩,
                ⟨"Gs", "Af"⟩, ⟨"An", "Bff"⟩, ⟨"As", "Bf"⟩, ⟨"Bn", "Cf"⟩⟩
5    Spelling ← ⟨⟩
6    for i ← 0 to |WinMidiList| − 1
7       PitchClass ← WinMidiList[i] mod 12
8       Spelling ← Spelling ⊕ ⟨PNCs[PitchClass][BitVec[i]]⟩
9    return Spelling
```

Figure 3.5: The ComputeSpelling algorithm.

(Cambouropoulos, 1996, p. 243)). Consequently, SpellWin03 simply selects the first spelling computed with the minimum penalty score.

As shown in Figure 3.1 and discussed above, the part of **BestSpelling** that SpellWin03 returns depends on whether the window being processed is the first window, the last window or another window, as indicated by the values of the flags *FirstWindow* and *LastWindow*. If the current window is neither the first nor last window, then the pitch name classes for the middle third of the current window should be returned, therefore SpellWin03 must return **BestSpelling**[0, *WinMidiListSize*/2]. In lines 17–22, the variable *BSEnd* is used to store the position of the end of the part of **BestSpelling** that is returned. This value is initialised to *WinMidiListSize*/2 in line 17 as this is the value that it should have for every window apart from the first and the last. If the current window is the first window, then **WinPNCList** is empty and SpellWin03 must return the first two thirds of **BestSpelling**—that is, **BestSpelling**[0, 2 *WinMidiListSize*/3]. Therefore, if *FirstWindow* is true in line 18, *BSEnd* is set to equal 2 *WinMidiListSize*/3 in line 19. If the current window is the last window (i.e., *LastWindow* is true in line 20), then the whole of **WinMidiList** must be returned so *BSEnd* is set to equal *WinMidiListSize* in line 21.

3.2.3 The ComputeSpelling algorithm

The ComputeSpelling algorithm, called in lines 6 and 11 of SpellWin03, is defined in Figure 3.5. This function computes a sequence of pitch name classes (i.e., a 'spelling') for the sequence of MIDI note numbers, **WinMidiList**, given to it as an argument. The specific spelling returned by ComputeSpelling is determined in the way described in section 3.2.2 above by the bit vector argument, **BitVec**, and the argument *SharpOrFlatSide* which determines whether the spelling is flatside or sharpside. The length of **BitVec** must be identical to that of **WinMidiList** and the argument *SharpOrFlatSide* must be equal to either flatside or sharpside. In lines 1–4, the variable **PNCs** is set to equal either the flatside or sharpside pitch name class region depending on the value of *SharpOrFlatSide* (see Figure 3.3). In either case, **PNCs** becomes an ordered set of 12 ordered pairs of pitch name classes, the ith ordered pair in **PNCs** being the two possible pitch name classes for the pitch class $i - 1$. For example, when *SharpOrFlatSide* is equal to

```
ComputeSpellPen03(Spelling)
1    n ← |Spelling|
2    SpellingPenalty ← 0
3    for i ← 0 to n − 2
4        for j ← i + 1 to n − 1
5            if IsEnharmonicSpelling(Spelling[i])
6                SpellingPenalty ← SpellingPenalty + 2
7            if IsEnharmonicSpelling(Spelling[j])
8                SpellingPenalty ← SpellingPenalty + 2
9            PINC ← PNC2PINC(Spelling[i], Spelling[j])
10           if not IsClassAOrBPINC(PINC)
11               if IsClassCPINC(PINC)
12                   SpellingPenalty ← SpellingPenalty + 1
13               else
14                   SpellingPenalty ← SpellingPenalty + 2
15   return SpellingPenalty
```

Figure 3.6: The ComputeSpellPen03 algorithm.

sharpside, the third element of **PNCs** is ⟨"Css", "Dn"⟩ (see line 2), "Css" and "Dn" being the two possible sharpside pitch name classes that Cambouropoulos's algorithm can assign to the pitch class 2. In each ordered pair of pitch name classes, the first pitch name class is the sharper of the two. In line 5, the variable **Spelling** is initialised to the empty sequence. Then, in lines 6–8, the 'for' loop iterates over **WinMidiList**, calculating the pitch class of each MIDI note number (line 7) and using this pitch class and the appropriate element of **BitVec** to select from **PNCs** a pitch name class which is then appended to **Spelling** (line 8). When this 'for' loop has terminated, the variable **Spelling** is returned in line 9.

3.2.4 The ComputeSpellPen03 algorithm

The total penalty score for each window spelling is computed using the function ComputeSpellPen03 which is called in lines 7 and 12 of SpellWin03 and defined in Figure 3.6. This function implements Cambouropoulos's (2003, p. 421) principles of 'notational parsimony' and 'interval optimization' mentioned in section 3.1 above. ComputeSpellPen03 takes one argument, **Spelling**, which is an ordered set of n pitch name classes, and computes a penalty value for each of the $n(n-1)/2$ (unordered) pairs of elements in **Spelling**—that is, both contiguous and non-contiguous pairs of pitch name classes are considered (see Cambouropoulos, 2003, p. 422). The penalty values for all these pitch name class pairs are summed to give the total penalty score for **Spelling**. This is accomplished by the nested 'for' loops in lines 3 to 14 of ComputeSpellPen03. On each iteration of the inner 'for' loop (lines 4–14), the penalty value for a particular pair of pitch name classes (**Spelling**[i] and **Spelling**[j]) is calculated and added to the total penalty score (stored in *SpellingPenalty*). When all pairs of pitch name classes have been considered, the outer 'for' loop (lines 3–14) terminates and the value in *SpellingPenalty* is returned (line 15).

The principle of 'notational parsimony' is implemented by increasing the penalty applied to a pair of pitch name classes by 2 for each pitch name class in the pair that is 'enharmonic'—Cambouropoulos defines a pitch name to be 'enharmonic' if it has one or more sharps or flats and is enharmonically equivalent to a 'natural' pitch name (Cambouropoulos, 2003, p. 421).

```
IsEnharmonicSpelling(PNC)
1    return (PNC ∈ {"Bs", "Dff", "Css", "Eff", "Dss", "Ff", "Es",
                    "Gff", "Fss", "Aff", "Gss", "Bff", "Ass", "Cf"})
```

Figure 3.7: The IsEnharmonicSpelling algorithm.

Modality Class	A	B				C				D		
Intervals	P4	m2	M2	m3	M3	A2	d3	d4	A4	d1	A3	d2
	P5	M7	m7	M6	m6	d7	A6	A5	d5	A1	d6	A7

Table 3.1: Table used by Cambouropoulos (2003, p. 417) to categorise intervals into modality classes.

For example, in the sharpside region (see Figure 3.3), the 'enharmonic' pitch name classes are `"Bs"`, `"Css"`, `"Dss"`, `"Es"`, `"Fss"`, `"Gss"` and `"Ass"`. This is achieved in lines 5–8 of ComputeSpellPen03 using the function IsEnharmonicSpelling (defined in Figure 3.7) to determine whether the pitch name classes are enharmonic.

According to Cambouropoulos's (1996) *General Pitch Interval Representation* (*GPIR*), the interval between any two pitch names can be categorised into one of four *modality classes* (Cambouropoulos, 1996, p. 235) according to the frequency with which the interval occurs in the major and minor (harmonic and melodic) scales (Cambouropoulos, 2003, p. 416). According to Cambouropoulos (2003, p. 416), an interval belongs to

1. class A if it is perfect;
2. class B if it is major or minor;
3. class C if it is a diminished or augmented interval that occurs in the major and minor scales; and
4. class D otherwise.

In the most recent published version of his method, Cambouropoulos (2003, p. 417) actually categorises each interval according to the table shown in Table 3.1. In this table, each interval is denoted using a code in which 'M' denotes major, 'm' denotes minor, 'A' denotes augmented and 'd' denotes diminished. It will be assumed here that each of these intervals is actually supposed to represent the pitch interval name class containing the rising interval with the indicated quality and diatonic size. For example, it is assumed that 'P4' in Table 3.1 actually denotes [`"rp4"`], 'M7' denotes [`"rma7"`], 'A2' denotes [`"ra2"`] and so on. It is also assumed that [`"p1"`] is in class A. It should also be noted that there are many intervals that occur between pitch name classes allowed by Cambouropoulos's algorithm that do not appear in the table. For example, within a single flatside window spelling, one might have both `"Gs"` and `"Aff"` (see section 3.2.2 above). If this happened, a penalty value would have to be computed for either [`"rdd2"`] or [`"raa7"`], neither of which appear in Table 3.1. This is not really a problem as Cambouropoulos (2003, p. 416) specifies that "intervals not encountered between scale degrees... form class D". There

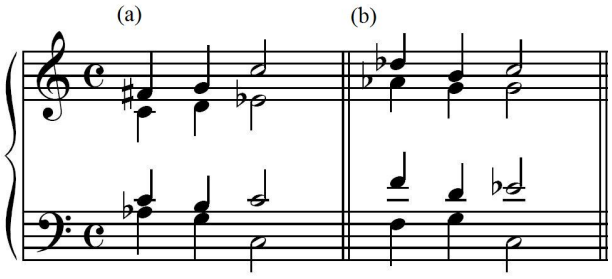


Figure 3.8: Diminished thirds and augmented sixths occur reasonably often in minor-mode works between the sharpened subdominant and the flattened submediant in augmented sixth chords (example (a)) and between the flattened supertonic and leading note in contexts where the Neapolitan sixth is used (example (b)).

is, however, an error in Table 3.1: the diminished third and augmented sixth are placed in class C whereas they should be in class D as they do not occur in any of the major and minor scales.[1]

It should be noted, however, that diminished thirds and augmented sixths occur reasonably often in minor-mode works between the sharpened subdominant and the flattened submediant in augmented sixth chords and between the flattened supertonic and leading note in contexts where the Neapolitan sixth is used (see Figure 3.8). This highlights the fact that the frequency with which an interval occurs in a set of scales does not directly relate to the frequency with which the interval is used in works that putatively use that set of scales.

In any case, the results published by Cambouropoulos (2003) were produced using an algorithm that employs the categorisation specified in Table 3.1, therefore the CAM03 algorithm (which is intended to be an accurate implementation of the algorithm described by Cambouropoulos, 2003) also uses Table 3.1 to classify the intervals. In section 3.6.3 below, a modified version of CAM03 will be described that uses the correct categorisation and the results obtained with this algorithm on the test corpus will be compared with those achieved by CAM03.

In COMPUTESPELLPEN03, the pitch interval name class of the interval between each pitch name class pair, **Spelling**[i] and **Spelling**[j], is calculated in line 9 using the function PNC2PINC (defined in Figure 1.25) and stored in the variable *PINC*. Then, in lines 10–14, in accordance with the penalty values specified by Cambouropoulos (2003, p. 421), the principle of 'interval optimization' is implemented by increasing the penalty applied to the pair, **Spelling**[i] and **Spelling**[j], by

- one, if *PINC* is a member of modality class C as specified in Table 3.1 (see line 12 of COMPUTESPELLPEN03); and
- two, if *PINC* is not a member of modality classes A, B or C in Table 3.1 (see line 14 of COMPUTESPELLPEN03).

The modality class of *PINC* is determined in lines 10 and 11 of COMPUTESPELLPEN03 using the functions ISCLASSAORBPINC and ISCLASSCPINC, defined in Figures 3.9 and 3.10, respectively.

[1] Cambouropoulos (2004) agrees that ["ra6"] and ["rd3"] should be in class D.

```
IsClassAOrBPINC(PINC)
1    return (PINC ∈ {["p1"], ["rp4"], ["rp5"], ["rmi2"], ["rma7"], ["rma2"],
                     ["rmi7"], ["rmi3"], ["rma6"], ["rma3"], ["rmi6"]})
```

Figure 3.9: The IsClassAOrBPINC algorithm.

```
IsClassCPINC(PINC)
1    return (PINC ∈ {["ra2"], ["rd7"], ["rd3"], ["ra6"], ["rd4"], ["ra5"], ["ra4"], ["rd5"]})
```

Figure 3.10: The IsClassCPINC algorithm.

3.2.5 Time and space complexities of Cam03

The 'while' loop in lines 6–19 of Cam03 executes once for each window and there are $O(|\textbf{MidiList}|/\mathit{WinSize})$ windows. For each of these windows, the function SpellWin03 is called in line 16 of Cam03. SpellWin03 computes $2^{1+2\,\mathit{WinSize}/3}$ different spellings for each window (apart from the first and possibly the last). For each spelling, ComputeSpellPen03 needs to calculate a penalty value for $O(\mathit{WinSize}^2)$ intervals and add these values together to get an overall penalty score for the window spelling. The overall worst-case time complexity of Cam03 is therefore

$$O\left(\frac{|\textbf{MidiList}|}{\mathit{WinSize}} \times 2^{1+2\,\mathit{WinSize}/3} \times \mathit{WinSize}^2\right) = O\left(|\textbf{MidiList}| \times \mathit{WinSize} \times 2^{2\,\mathit{WinSize}/3}\right). \tag{3.1}$$

If the Cam03 algorithm is implemented exactly as given in Figure 3.2, then it will use $O(|\textbf{MidiList}|)$ working memory as it eventually has all of **PNCList** stored in memory. However, if the retained segment for each window is written to output immediately after it has been computed in line 16 of Cam03 instead of being appended to **PNCList**, the algorithm would use only $O(\mathit{WinSize})$ working memory.

3.3 The algorithm described by Cambouropoulos (2001)

3.3.1 The Cam01 algorithm

In this section, I present my own implementation, expressed in pseudocode, of the algorithm described by Cambouropoulos (2001). The complete algorithm, which I call Cam01, is given in Figure 3.12.

Like Cam03, Cam01 takes as input a sequence of MIDI note numbers, **MidiList**, and outputs a sequence of pitch name classes, **PNCList** (see Figure 3.12). Also like Cam03, Cam01 uses a "shifting overlapping windowing technique" (Cambouropoulos, 2001, p. 4). However, in Cam01, each window does *not* contain a fixed number of contiguous elements in **MidiList**. Instead, each window is a segment of **MidiList** that contains *WinSize* distinct MIDI note numbers. This means that the length of the window varies as the algorithm executes. Cambouropoulos (2001, p. 5) explains that he employed such a "variable length window" in order to

> avoid unnecessary ambiguity that is introduced when too few different pitches appear in a given window (e.g. in a section that has just 3-4 repeating pitches as in the case of an alberti bass).

According to Cambouropoulos (2001, p. 5), using such a variable length window not only improves "transcription quality" by "avoiding misspellings due to lack of appropriate pitch context", but also improves efficiency "as larger windows can be used without adding to the computational complexity". Unfortunately, because a variable length window is used, controlling the windowing process itself is rather more involved in CAM01 than it is in CAM03.

In his 2001 paper, Cambouropoulos merely states that a variable length window is employed without giving any details as to how the windowing process is implemented. However, he provided me with the following fuller explanation in a personal communication (Cambouropoulos, 2004) (text in square brackets is mine):

> In each variable length window there are three sections - the prefix and suffix that contain three [i.e., *WinSize*/3] distinct MIDI note numbers and the middle section which may contain more than three [*WinSize*/3] distinct MIDI note numbers. The middle part is retained.
>
> The window is shifted to the next position - it now starts with the last three [*WinSize*/3] distinct MIDI pitch numbers of the previously spelt section and extends till a total of 9 [i.e., *WinSize*] distinct MIDI pitch numbers is reached.
>
> The prefix and suffix simply contain three [*WinSize*/3] distinct MIDI pitch numbers. The algorithm moves in such a way that the middle parts are consecutive.

He adds that

> for the last window one can keep more than three [*WinSize*/3] MIDI pitch numbers from the previous window so as to have a total of 9 [i.e., *WinSize*] distinct MIDI pitch numbers.
>
> (Cambouropoulos, 2004)

As an illustration, let us consider what happens when this process is carried out on a **MidiList** representation of the theme from Bach's *Musical Offering* (BWV 1079) (see Figure 3.11). Let's assume that $WinSize = 9$.

The first window must contain 9 distinct MIDI note numbers, therefore it extends up to the end of note 10 (i.e., it consists of $\mathbf{MidiList}[0, 11]$) (see Figure 3.11). The prefix segment must be empty for the first window, otherwise the first few notes would never get spelt. The suffix segment for the first window must contain $WinSize/3 = 3$ distinct MIDI note numbers, therefore it consists of notes 8–10 (i.e., $\mathbf{MidiList}[8, 11]$). I shall assume that both the prefix and suffix for each window are no longer than necessary and that the retained segment is as long as possible. The retained segment for each window extends from the note following the prefix segment to the note just before the suffix segment, inclusive. Therefore, for the first window, the retained segment consists of notes 0–7 (i.e., $\mathbf{MidiList}[0, 8]$), as indicated in Figure 3.11.

Cambouropoulos (2004) states that "the algorithm moves in such a way that the middle parts [i.e., the retained segments] are consecutive". Therefore the retained segment for the second window must begin at note 8. This implies that the prefix segment for the second window must begin at note 5 since the prefix must contain 3 distinct MIDI note numbers. The second window as a whole must contain 9 distinct MIDI note numbers. Therefore, the second window must consist of notes 5–13 (i.e., $\mathbf{MidiList}[5, 14]$). The suffix segment must contain 3 distinct MIDI

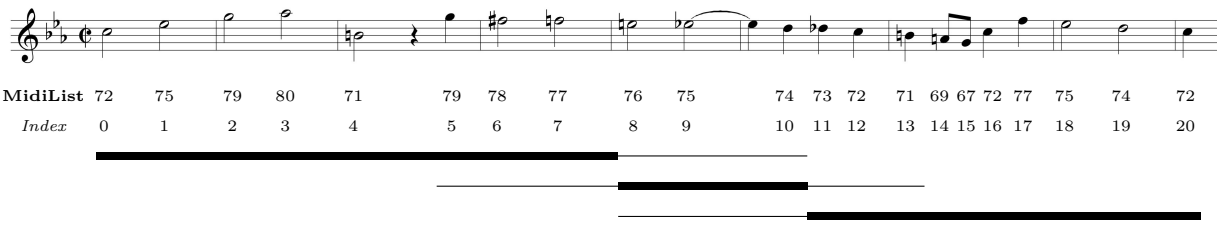


Figure 3.11: The theme from J. S. Bach's *Musical Offering* (BWV 1079) showing the MIDI note number and index within **MidiList** of each note. The locations of all windows and retained segments are also indicated.

note numbers, be as short as possible and, of course, end at the end of the window. Therefore the suffix segment for the second window consists of notes 11–13 (**MidiList**$[11, 14]$). The retained segment for the second window must therefore consist of notes 8–10 (**MidiList**$[8, 11]$).

Since the retained segment for the second window ended with note 10, the retained segment for the third window must start with note 11. The prefix segment must contain three distinct MIDI note numbers so the window must start with note 8. The window must contain 9 distinct note numbers and the retained segment must be as long as possible, therefore the window extends to the end of the theme. When a window extends to the end of a **MidiList**, the retained segment for that window must also extend to the end of the **MidiList** because it makes no sense for such a window to have a non-empty suffix segment. For example, if the third window in Figure 3.11 had to have a suffix segment, this suffix segment would contain notes 18–20. This would mean that a fourth window would have to be spelt in which the retained segment started at note 18. But Cambouropoulos (2004) states that "for the last window one can keep more than three MIDI pitch numbers from the previous window so as to have a total of 9". In other words, in such a situation, the prefix must be extended backwards until the window as a whole contains 9 distinct MIDI note numbers. This would mean that the putative fourth window would, in fact, cover the same region as the third window which would imply that the last three notes would be spelt in exactly the same way as they would have been had they been spelt in the third window. Thus, if a window extends to the end of a **MidiList**, it makes no sense for the retained segment not to do the same.

The CAM01 algorithm in Figure 3.12 implements the windowing process just described. This algorithm takes the same arguments, **MidiList** and *WinSize*, as CAM03. The variables *MidiListSize*, *WinStart*, *WinEnd* and **PNCList** also serve the same functions in CAM01 as they do in CAM03.

For each window, the first value to be determined is the position in **MidiList** where the retained segment starts. In CAM01, this position is stored in the variable *RetSegStart*, which is initialised to 0 in line 2 (see Figure 3.12). Once *RetSegStart* has been computed, the starting position of the current window can be computed by starting at position $RetSegStart - 1$ and moving back towards the beginning of **MidiList** until $WinSize/3$ distinct MIDI note numbers have been encountered. This is carried out in lines 5–11 of CAM01. This establishes the position of the window prefix segment as **MidiList**$[WinStart, RetSegStart]$. The set of distinct MIDI note numbers that occur within this prefix segment is stored in the variable **PreMIDISet**.

```
CAM01(MidiList, WinSize)
    MidiListSize ← |MidiList|
    RetSegStart ← 0
    PNCList ← ⟨ ⟩
    while |PNCList| < MidiListSize
      ▶ Find WinStart and PreMIDISet.
      WinStart ← RetSegStart
      PreMIDISet ← ⟨ ⟩
      while (WinStart > 0) ∧ (|PreMIDISet| < WinSize/3)
        WinStart ← WinStart − 1
        if MidiList[WinStart] ∉ PreMIDISet
          PreMIDISet ← ⟨MidiList[WinStart]⟩ ⊕ PreMIDISet
      ▶ Now find WinEnd and WinMIDISet.
      WinEnd ← RetSegStart
      WinMIDISet ← PreMIDISet
      while (WinEnd < MidiListSize) ∧ (|WinMIDISet| ≤ WinSize)
        if MidiList[WinEnd] ∉ WinMIDISet
          WinMIDISet ← WinMIDISet ⊕ ⟨MidiList[WinEnd]⟩
        WinEnd ← WinEnd + 1
      if |WinMIDISet| > WinSize
        WinEnd ← WinEnd − 1
        WinMIDISet ← WinMIDISet[0, WinSize]
      ▶ If WinMIDISet is too small, extend PreMIDISet and WinStart backwards,
      ▶ remembering to update both PreMIDISet and WinMIDISet.
      while (|WinMIDISet| < WinSize) ∧ (WinStart > 0)
        WinStart ← WinStart − 1
        if MidiList[WinStart] ∉ PreMIDISet
          PreMIDISet ← ⟨MidiList[WinStart]⟩ ⊕ PreMIDISet
        if MidiList[WinStart] ∉ WinMIDISet
          WinMIDISet ← ⟨MidiList[WinStart]⟩ ⊕ WinMIDISet
      ▶ Now find RetSegEnd and PostMIDISet.
      RetSegEnd ← WinEnd
      PostMIDISet ← ⟨ ⟩
      if WinEnd ≠ MidiListSize
        while (RetSegEnd > 0) ∧ (|PostMIDISet| < WinSize/3)
          RetSegEnd ← RetSegEnd − 1
          if MidiList[RetSegEnd] ∉ PostMIDISet
            PostMIDISet ← ⟨MidiList[RetSegEnd]⟩ ⊕ PostMIDISet
      ▶ Now find PrePNCList.
      PrePNCList ← ⟨ ⟩
      LocalPreMidiSet ← ⟨ ⟩
      i ← RetSegStart − 1
      while |PrePNCList| ≠ |PreMIDISet|
        if MidiList[i] ∉ LocalPreMidiSet
          PrePNCList ← ⟨PNCList[i]⟩ ⊕ PrePNCList
          LocalPreMidiSet ← ⟨MidiList[i]⟩ ⊕ LocalPreMidiSet
        i ← i − 1
      ▶ Now remove PreMIDISet from WinMIDISet to get NewWinMIDISet.
      NewWinMIDISet ← ⟨ ⟩
      for i ← 0 to |WinMIDISet| − 1
        if WinMIDISet[i] ∉ PreMIDISet
          NewWinMIDISet ← NewWinMIDISet ⊕ ⟨WinMIDISet[i]⟩
      ▶ Now append best spelling for this retained segment to PNCList.
      RetSeg ← MidiList[RetSegStart, RetSegEnd]
      PNCList ← PNCList ⊕ SPELLWIN01(PrePNCList, PreMIDISet, NewWinMIDISet, RetSeg)
      ▶ Finally, set RetSegStart to equal start position of next retained segment.
      RetSegStart ← RetSegEnd
    return PNCList
```

Figure 3.12: Cambouropoulos's (2001) pitch spelling algorithm expressed in pseudocode.

Having established the starting positions of the window and the retained segment, the next task is to determine where the current window ends (stored in *WinEnd*) and the set of distinct MIDI note numbers in the current window (stored in **WinMIDISet**). This is done in lines 12–21 of CAM01. *WinEnd* is initially set to equal the position where the retained segment starts (line 13) and **WinMIDISet** is initialised to **PreMIDISet** (line 14). *WinEnd* is then advanced towards the end of **MidiList** and **WinMIDISet** is correspondingly updated until either **WinMIDISet** contains $WinSize+1$ distinct MIDI note numbers or the end of **MidiList** is reached (lines 15–18). Note that *WinEnd* must be advanced until **WinMIDISet** contains $WinSize+1$ and not *WinSize* MIDI note numbers. If it were advanced until **WinMIDISet** contained only *WinSize* MIDI note numbers, the window and retained segment would not be as long as possible. This means, however, that when the 'while' loop in lines 15–18 terminates, **WinMIDISet** typically contains $WinSize+1$ MIDI note numbers and *WinEnd* is one greater than it should be. If this is the case, it is corrected in lines 19–21.

In this process, it is possible for *WinEnd* to reach the end of **MidiList** before **WinMIDISet** contains *WinSize* distinct MIDI note numbers. In this case, the prefix segment must be extended back towards the beginning of **MidiList** until either **WinMIDISet** contains *WinSize* MIDI note numbers or the beginning of **MidiList** is reached. Each time a new MIDI note number is encountered during this prefix extension process, both **PreMIDISet** and **WinMIDISet** must be updated. This task is carried out in lines 22–29 of CAM01.

When the starting and ending positions of the window and the starting position of the retained segment have been fixed, it is then possible to compute the ending position of the retained segment (stored in *RetSegEnd*) together with the set of MIDI note numbers that occur within the suffix segment (stored in **PostMIDISet**). This is done in lines 30–37 of CAM01. *RetSegEnd* is first initialised to equal *WinEnd* in line 31 and **PostMIDISet** is set to equal the empty sequence in line 32. As explained above, if the current window extends right to the end of **MidiList**, then the retained segment must do likewise and *RetSegEnd* and **PostMIDISet** can be left unchanged. If the window does not extend to the end of **MidiList**, then the ending position of the retained segment is decremented back towards the beginning of **MidiList** until $WinSize/3$ distinct MIDI note numbers have been encountered, updating **PostMIDISet** accordingly on each step (see lines 33–37).

In CAM01 I have assumed that the pitch name class assigned to each MIDI note number in the prefix segment of the current window is fixed by the way that it was spelt in the previous window (even though Cambouropoulos (2001, 2004) does not explicitly say that this is the case.) This corresponds to the way in which the spelling of the first third of each window is determined by the way it was spelt in the previous window in CAM03 (see section 3.2 above). Before computing the spelling of the current window, it is therefore necessary to find the pitch name classes for each of the MIDI note numbers in **PreMIDISet** that were assigned in the previous window. This is done in lines 38–46 of CAM01 simply by inspecting the last few elements in **PNCList**. The ordered set of prefix segment pitch name classes is stored in the variable **PrePNCList** (cf. **WinPNCList** variable in CAM03).

In CAM01, each distinct MIDI note number is spelt in the same way every time it occurs within the current window. This means that, when the spelling for a given window is being

```
SpellWin01(PrePNCList, PreMIDISet, NewWinMIDISet, RetSeg)
1    NewWinMIDISetSize ← |NewWinMIDISet|
2    MaxBitVecInt ← 2^NewWinMIDISetSize − 1
3    BestSpelling ← ⟨ ⟩
4    for i ← 0 to MaxBitVecInt
5       BitVec ← BitVector(i, NewWinMIDISetSize)
6       Spelling ← ComputeSpelling(NewWinMIDISet, BitVec, flatside)
7       SpellingPenalty ← ComputeSpellPen01(PrePNCList ⊕ Spelling)
8       if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
9          BestSpelling ← Spelling
10         BestPenalty ← SpellingPenalty
11      Spelling ← ComputeSpelling(NewWinMIDISet, BitVec, sharpside)
12      SpellingPenalty ← ComputeSpellPen01(PrePNCList ⊕ Spelling)
13      if SpellingPenalty < BestPenalty
14         BestSpelling ← Spelling
15         BestPenalty ← SpellingPenalty
16   ▶ Now spell pitches in retained segment and return it.
17   SpeltRetSeg ← ⟨ ⟩
18   for i ← 0 to |RetSeg| − 1
19      PNC ← (PrePNCList ⊕ BestSpelling)[Pos(RetSeg[i], PreMIDISet ⊕ NewWinMIDISet)]
20      SpeltRetSeg ← SpeltRetSeg ⊕ ⟨PNC⟩
21   return SpeltRetSeg
```

Figure 3.13: The SpellWin01 algorithm.

found, it is only necessary to find pitch name classes for those MIDI note numbers in **WinMIDISet** that do not occur within the prefix segment. In lines 47–51, therefore, the variable **NewWinMIDISet** is set to contain only those MIDI note numbers in **WinMIDISet** that do not occur in **PreMIDISet**.

In line 53, the retained segment for the current window is stored in the variable **RetSeg** as a sequence of MIDI note numbers and then, in line 54, a spelling for this retained segment is computed using the function SpellWin01 (defined in Figure 3.13) and appended onto **PNCList**.

Since consecutive retained segments must be contiguous, the starting position of the next retained segment (*RetSegStart*) is set to equal the ending position of the current retained segment (*RetSegEnd*) in line 56.

The 'while' loop in lines 4–56 continues to iterate until all the MIDI note numbers have been spelt and **PNCList** is the same length as **MidiList**, at which point, **PNCList** is returned (line 57).

3.3.2 The SpellWin01 algorithm

Let us now look more closely at the SpellWin01 function, called in line 54 of Cam01 and defined in Figure 3.13. In lines 1–15, SpellWin01 uses a method that is essentially identical to that used in SpellWin03 (see Figure 3.4 and section 3.2.2 above) to find the best spelling (**BestSpelling**) for the MIDI note numbers in its argument, **NewWinMIDISet**. Recall that **NewWinMIDISet** contains the MIDI note numbers in the current window that are not in **PreMIDISet**. SpellWin01 then uses the set of pitch name classes stored in **BestSpelling** (together with those in **PrePNCList** retained from the previous window) to assign a pitch name class to each of the MIDI note numbers in the retained segment, **RetSeg** (lines 17–20). The sequence of pitch name classes assigned to **RetSeg** is stored in **SpeltRetSeg** which is returned

```
ComputeSpellPen01(Spelling)
1    n ← |Spelling|
2    SpellingPenalty ← 0
3    for i ← 0 to n − 2
4       for j ← i + 1 to n − 1
5          if IsEnharmonicSpelling(Spelling[i])
6             SpellingPenalty ← SpellingPenalty + 4
7          if IsEnharmonicSpelling(Spelling[j])
8             SpellingPenalty ← SpellingPenalty + 4
9          PINC ← PNC2PINC(Spelling[i], Spelling[j])
10         if not IsClassAOrBPINC(PINC)
11            if IsClassCPINC(PINC)
12               SpellingPenalty ← SpellingPenalty + 1
13            else
14               SpellingPenalty ← SpellingPenalty + 3
15   return SpellingPenalty
```

Figure 3.14: The ComputeSpellPen01 algorithm.

at the end of the function (line 21).

Lines 1–15 in SpellWin01 and SpellWin03 are identical, except that

1. the arguments **WinPNCList** and **WinMidiList** in SpellWin03 are replaced in SpellWin01 by **PrePNCList** and **NewWinMIDISet**, respectively;

2. the variable *WinMidiListSize* in SpellWin03 is replaced by *NewWinMIDISetSize* in SpellWin01; and

3. the calls to ComputeSpellPen03 in lines 7 and 12 of SpellWin03 are replaced in SpellWin01 by corresponding calls to the function ComputeSpellPen01 which is defined in Figure 3.14.

3.3.3 The ComputeSpellPen01 algorithm

In SpellWin01, the total penalty score for each window spelling is computed using the function ComputeSpellPen01 defined in Figure 3.14. ComputeSpellPen01 is identical to ComputeSpellPen03, except that the penalty applied to each pitch name class pair is increased by

- 4 (instead of 2) for each pitch name class that is 'enharmonic'; and
- 3 (instead of 2) if the interval between the two pitch name classes is in modality class D.

These values are as specified by Cambouropoulos (2001, p. 5).

3.3.4 Time and space complexities of Cam01

The 'while' loop in lines 4–56 in Cam01 executes once for each window and, in the worst case, the number of windows is $O(|\textbf{MidiList}|/\mathit{WinSize})$. If the number of windows is worst case, then each window spans *WinSize* contiguous elements of **MidiList** and lines 5–53 take $O(\mathit{WinSize}\log_2(\mathit{WinSize}))$ time for each window (assuming each membership check in lines 10,

16, 26, 28, 36, 43 and 50 can be done in $O(\log_2(\mathit{WinSize}))$ time). For each window apart from the first (and possibly the last), SpellWin01, called in line 54 of Cam01, computes $2^{1+2\,\mathit{WinSize}/3}$ different spellings. For each of these window spellings, ComputeSpellPen01 needs to calculate a penalty value for $O(\mathit{WinSize}^2)$ intervals and add these values together to get an overall penalty score for the window spelling. The overall worst-case time complexity of Cam01 is therefore

$$O\left(\frac{|\mathbf{MidiList}|}{\mathit{WinSize}} \times 2^{1+2\,\mathit{WinSize}/3} \times \mathit{WinSize}^2\right) = O\left(|\mathbf{MidiList}| \times \mathit{WinSize} \times 2^{2\,\mathit{WinSize}/3}\right). \quad (3.2)$$

That is, in the worst case, the time complexity of Cam01 is the same as that of Cam03. However, the worst case only occurs when each window spans *WinSize* elements in **MidiList**. Usually, each window spans more than *WinSize* elements because of the presence of repeated notes.

If Cam01 is implemented as shown in Figure 3.12, then it will use $O(|\mathbf{MidiList}|)$ working memory because the whole of **PNCList** is stored before it is written to output in the last line. However, if each retained segment is written to output in line 54 of Cam01 instead of appended to **PNCList**, the algorithm would require only $O(\mathit{WinSize})$ space.

3.4 The algorithm described by Cambouropoulos (1996, 1998)

3.4.1 The Cam9698 algorithm

In this section I present my own implementation, expressed in pseudocode, of the earliest published pitch spelling method described by Cambouropoulos (1996, 1998). This implementation, which I call Cam9698, is defined in Figure 3.15.

Like Cam03 and Cam01, Cam9698 takes as input a sequence of MIDI note numbers, **MidiList**, and outputs a sequence of pitch name classes **PNCList** (see Figure 3.15). Also, like Cam03, Cam9698 employs a shifting overlapping windowing technique in which each window contains a fixed number (determined by the argument *WinSize*) of contiguous elements from **MidiList** (Cambouropoulos, 1996, p. 245).

However, unlike Cam03 and Cam01, Cam9698 was specifically designed for processing monophonic music. Also, it incorporates a special rule, based on Krumhansl's (1990, pp. 150–151) principle of contextual asymmetry, that is applied as a 'tie-breaker' when two or more spellings for a given window achieve the least penalty score. This rule will be discussed in more detail in section 3.4.6 below.

Cam9698 also differs from the two more recent versions of the algorithm discussed above in that the modality class of each interval is not simply looked up in Table 3.1, but calculated directly from the frequency of occurrence of the interval in the major and minor scales.

Also, in Cam9698, for each window *all* spellings are considered that are possible given the restricted range of pitch name classes that Cambouropoulos allows for each pitch class (i.e., three pitch name classes for each 'white note' pitch class and two for each 'black note' pitch class). In other words, each window spelling does not have to be completely contained within either the flatside region or sharpside region shown in Figure 3.3. This obviously results in Cam9698 having a higher time complexity than the two more recent versions of the algorithm. The running time of the algorithm is further increased by the fact that the pitch name classes for the first third of each window are not retained from the previous window as they are in Cam03 and Cam01.

```
CAM9698(MidiList, WinSize)
1   MidiListSize ← |MidiList|
2   LastWindow ← false
3   FirstWindow ← true
4   WinStart ← 0
5   PNCList ← ⟨ ⟩
6   MajorScale ← ⟨2, 2, 1, 2, 2, 2, 1⟩
7   NatMinScale ← ⟨2, 1, 2, 2, 1, 2, 2⟩
8   DescMelMinScale ← ⟨2, 1, 2, 2, 1, 2, 2⟩
9   AscMelMinScale ← ⟨2, 1, 2, 2, 2, 2, 1⟩
10  HarmMinScale ← ⟨2, 1, 2, 2, 1, 3, 1⟩
11  BlendedModTable ← COMPUTEBLENDEDMODTABLE(0.25,
                                              ⟨⟨MajorScale, 4⟩,
                                               ⟨NatMinScale, 1⟩,
                                               ⟨DescMelMinScale, 1⟩,
                                               ⟨AscMelMinScale, 1⟩,
                                               ⟨HarmMinScale, 2⟩⟩)
12  while |PNCList| < MidiListSize
13      WinEnd ← MIN({WinStart + WinSize, MidiListSize})
14      if WinEnd = MidiListSize
15          LastWindow ← true
16      WinMidiList ← MidiList[WinStart, WinEnd]
17      PNCList ← PNCList ⊕ SPELLWIN9698(WinMidiList, FirstWindow,
18                                       LastWindow, BlendedModTable, WinSize)
19      FirstWindow ← false
20      WinStart ← WinStart + (WinSize/3)
21  return PNCList
```

Figure 3.15: Cambouropoulos's (1996, 1998) pitch spelling algorithm expressed in pseudocode.

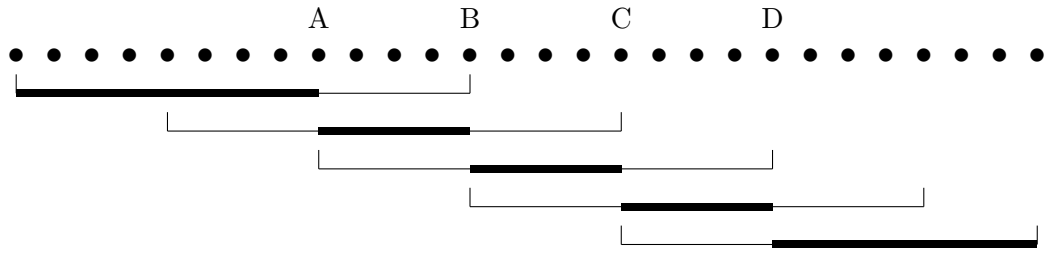


Figure 3.16: Illustration of the windowing process proposed by Cambouropoulos (1996). (Reproduced from Cambouropoulos, 1996, p. 245, Fig. 8.)

CAM9698 takes the same arguments, **MidiList** and *WinSize*, as CAM03 (see Figure 3.15). The variables *MidiListSize*, *LastWindow*, *FirstWindow*, *WinStart*, **PNCList**, *WinEnd* and **WinMidiList** also serve the same functions in CAM9698 as they do in CAM03.

The first five lines of CAM9698 and CAM03 are the same. The 'while' loop in lines 12–20 in CAM9698 works in essentially the same way as the 'while' loop in lines 6–19 of CAM03. However, because the pitch name classes of the first third of each window are not retained from the previous window in CAM9698, **WinMidiList** always contains the MIDI note numbers for the entire window (see line 16 in Figure 3.15) and there is no **WinPNCList** variable. Also, the method used for computing the best spelling for each window differs from that used in CAM03 so the SPELLWIN03 algorithm called in line 16 of CAM03 is replaced by the SPELLWIN9698 function called in line 17 of CAM9698.

As mentioned above, in CAM9698, the modality class of each pitch interval is computed directly from the frequency with which the interval occurs in the major and minor scales. To avoid calculating the modality class for the same interval several times, the modality classes of all possible pitch interval name classes are computed in lines 6–11 of CAM9698 and stored in the look-up table, **BlendedModTable**. This table is passed as an argument to SPELLWIN9698 and used in the calculation of the penalty score for each window spelling.

In CAM9698, the windowing process is identical to that in CAM03. That is, it works in the way illustrated in Figure 3.1. However, in his description of this earliest version of the algorithm, Cambouropoulos (1996, p. 245) proposes that the windowing process should follow the scheme illustrated in Figure 3.16. Note that in Figure 3.16 each window and each retained segment within each window appears to begin and end *on* a note rather than *between* two notes. However, Cambouropoulos (1996, p. 245) states that each window contains 13 elements, therefore we have to assume that each of the four notes marked A–D occurs in the retained segments of *two* windows. For example, note A would appear to be the last note in the retained segment for the first window and the first note in the retained segment for the second window. This causes a problem because it is not clear whether the pitch name class for note A should be fixed in the first window or the second window. Because this windowing process is ambiguous and, strictly speaking, unworkable, CAM9698 employs the very similar but more clearly specified process described by Cambouropoulos (2003, p. 420) and implemented in CAM03.

3.4.2 The ComputeModTable algorithm

In line 11 of Cam9698, the look-up table, **BlendedModTable**, which gives the modality class for every pitch interval, is computed using the function ComputeBlendedModTable defined in Figure 3.20. However, the ComputeBlendedModTable function cannot be understood before understanding the ComputeModTable function (called in line 3 of ComputeBlendedModTable and defined in Figure 3.17) and the ComputeModTable function cannot be understood before understanding certain aspects of Cambouropoulos's (1996, pp. 233–240) *General Pitch Interval Representation* (*GPIR*).

Cambouropoulos (1998, p. 67) states that

> "in the *GPIR* every pitch is represented by an array of the sort [nc, mdf, pc, oct] where *nc* (name-class) takes values from {0, 1, 2, ..., M} for an M-tone scale, *mdf* (modifier) takes values from {-u, ..., -1, 0, 1, ..., u} and u is the number of pitch interval units in the largest scale-step interval, *pc* (pitch-class) takes values from {0, 1, 2, ..., N} for an N-tone discrete equal-tempered pitch space and *oct* is octave range (middle C octave is 4). For example, in the diatonic system D4 is [1, 0, 2, 4], D♯4 is [1, 1, 3, 4], Eb5 is [2, -1, 3, 5], Gb3 is [4, -1, 6, 3]."

There are a few problems with this definition. First, the name-class of a pitch should actually take values from the set {0, 1, 2, ..., M-1} and the pitch-class of a pitch should take values from the set {0, 1, 2, ..., N-1}.

Second, Cambouropoulos stipulates that *mdf* can only take integer values between −u and u where "u is the number of pitch interval units in the largest scale-step interval". However, he does not specify *which* scale or scales this "largest scale-step interval" is permitted to occur in. If we assume, for example, that, for Western tonal pitch names, the "largest scale-step" is the largest interval that occurs between two consecutive elements in a major diatonic scale, then u must be 2. However, if this is the case, it will be impossible to represent pitch names with 3 or more sharps or flats in the *GPIR*. I suppose one might argue that, for Western tonal pitch names, the "largest scale-step" is the largest interval that occurs between two consecutive elements in any of the major and minor scales, including the harmonic minor scale. In this case, u would be 3 because of the augmented second between the submediant and leading note in the harmonic minor scale. However, this still means that any pitch name with four or more sharps or flats cannot be represented in the *GPIR*. Indeed, it is not clear why Cambouropoulos does not simply allow *mdf* to take any integer value.

Third, it is not clear whether *oct* represents the chromatic or diatonic (i.e., morphetic) octave of the pitch. For example, if *oct* represents diatonic octave then B♯3 would be represented by [6, 1, 0, 3]. However, if *oct* represents chromatic octave, then B♯3 would be represented by [6, 1, 0, 4].

Having specified the way that pitches are represented in *GPIR*, Cambouropoulos (1998, p. 69) goes on to consider the representation of pitch intervals. He defines the *name-class interval* (*nci*) of a pitch interval to be "the number of scale steps that an interval consists of" and states that it "is calculated as the modulo M difference between the name-class integers (for an M-tone scale)." For example, for a rising major second and rising augmented ninth, the *nci* is 1; and for

Pitch interval name	"p1"	"fd2"	"rma2"	"fmi3"	"raaa4"	"rma10"	"fma10"
nci	0	6	1	5	3	2	5
pci	0	0	2	9	8	4	8

Table 3.2: Some pitch interval names and their corresponding *nci*s and *pci*s.

a falling diminished second and falling minor ninth, the *nci* is 6. Some more examples are given in Table 3.2.

Cambouropoulos (1998, pp. 73, 75) uses the term *pitch class interval* (*pci*) to mean the number of steps that an interval spans modulo N in the underlying "N-tone discrete equal-tempered pitch space". For example, a rising major third and a rising diminished eleventh both have a *pci* of 4; and a falling augmented second and a falling minor tenth both have a *pci* of 9. Some more examples are given in Table 3.2.

Cambouropoulos (1998, p. 69) defines the *modality* of a pitch interval to be "determined by the *frequency of occurrence* of each member of the subset of intervals that relate to the *nci* integer". This rather vague 'definition' is made a little clearer by the following passage:

> If we calculate the number of times that all the different modalities of a specific name-class interval occur within a scale (taking as its lower note each degree of the scale), we can classify intervals depending on their frequency of occurrence. For example, the interval of a fourth in the diatonic genre occurs 6 times at the size of 5 semitones (frequency of occurrence F=6/7=0.86) and once at the size of 6 semitones (F=1/7=0.14).
>
> (Cambouropoulos, 1998, p. 69)

It is fairly clear from this that the modality of an interval i with respect to a particular scale S is supposed to be the number of times that i occurs in S divided by the number of notes in S. However, in his *GPIR*, Cambouropoulos uses only the *nci* and *pci* of an interval to calculate its modality. This means that any two intervals that have the same *nci* and *pci* will have the same modality in Cambouropoulos's *GPIR*. For example, "rma2" and "rddddddddddd2" both have $nci = 1$ and $pci = 2$. Therefore, according to Cambouropoulos's *GPIR*, they both have a modality of $\frac{5}{7} \simeq 0.71$ with respect to the diatonic major scale. In other words, when the modality of an interval with respect to a scale is calculated using just the *nci* and *pci* of the interval, many intervals that never occur within the scale will be assigned non-zero modalities. Fortunately, in the case of the Western tonal scales, the non-scale intervals that are assigned non-zero modalities are very exotic intervals that one probably never has to deal with when studying Western tonal music.

To make this clearer and to prepare for describing the ComputeModTable function, I shall now define more precisely the way that the modality of an interval is calculated in Cambouropoulos's *GPIR*.

We can represent an M_{Cam}-tone scale in an N_{Cam}-tone equal-tempered pitch system as a sequence of *pci*s, S, satisfying the following three conditions:

1. $|S| = M_{\text{Cam}}$;

```
ComputeModTable(Scale)
1    M_Cam ← |Scale|
2    N_Cam ← Σ_{i=0}^{M_Cam−1} Scale[i]
3    WithinScaleInts ← ⟨ ⟩
4    for nci ← 0 to M_Cam − 1
5       IntsForThisNCI ← ⟨ ⟩
6       for i ← 0 to M_Cam − 1
7          pci ← 0
8          for j ← i to i + nci − 1
9             pci ← pci + Scale[j mod M_Cam]
10         IntsForThisNCI ← IntsForThisNCI ⊕ ⟨⟨nci, pci⟩⟩
11      WithinScaleInts ← WithinScaleInts ⊕ IntsForThisNCI
12   ModalityTable ← ⟨ ⟩
13   for nci ← 0 to M_Cam − 1
14      ModTableRow ← ⟨ ⟩
15      for pci ← 0 to N_Cam − 1
16         f ← Count(⟨nci, pci⟩, WithinScaleInts)/M_Cam
17         ModTableElement ← ⟨nci, f, pci⟩
18         ModTableRow ← ModTableRow ⊕ ⟨ModTableElement⟩
19      ModalityTable ← ModalityTable ⊕ ⟨ModTableRow⟩
20   return ModalityTable
```

Figure 3.17: The ComputeModTable function.

2. $\sum_{i=0}^{M_{\text{Cam}}-1} S[i] = N_{\text{Cam}}$;

3. the ith element of S is the pci of the interval from the ith degree of the scale to the $(i+1)$th degree of the scale for all $1 \leq i \leq M_{\text{Cam}}$.

For example, in the 12-tone equal-tempered pitch system, the major diatonic scale would be represented by $\langle 2, 2, 1, 2, 2, 2, 1\rangle$ and the harmonic minor scale would be represented by $\langle 2, 1, 2, 2, 1, 3, 1\rangle$.

If i is a pitch interval, then let $nci(i)$ and $pci(i)$ denote the nci and pci of i, respectively. The first step in calculating the modality of an interval i with respect to a scale S (represented as an ordered set of M_{Cam} pcis as described in the previous paragraph), is to count the number n of values of j for which

$$S[j] + S[(j+1) \bmod M_{\text{Cam}}] + \ldots + S[(j + nci(i) - 1) \bmod M_{\text{Cam}}] = pci(i).$$

n is then simply divided by M_{Cam} to give the modality of i with respect to S.

The ComputeModTable function defined in Figure 3.17 implements the strategy just described. It takes a single argument, **Scale**, which must be a sequence of pcis representing a scale in the way described above. It then computes an $M_{\text{Cam}} \times N_{\text{Cam}}$ table, **ModalityTable**, in which **ModalityTable**$[n][p]$ gives the modality with respect to **Scale** for any interval whose nci is n and pci is p.

In lines 1 and 2 of ComputeModTable, the values of M_{Cam} and N_{Cam} are computed from **Scale** using the first two conditions for the pci sequence scale representation given above. Then, in lines 3–11, a list, **WithinScaleInts**, is constructed. If we denote by nci_{ij} and pci_{ij} the nci and pci respectively of the interval from the ith degree to the jth degree in the scale represented

$$\begin{array}{lccccccc} \langle & \langle 0,0\rangle, & \langle 0,0\rangle, & \langle 0,0\rangle, & \langle 0,0\rangle, & \langle 0,0\rangle, & \langle 0,0\rangle, & \langle 0,0\rangle, \\ & \langle 1,2\rangle, & \langle 1,2\rangle, & \langle 1,1\rangle, & \langle 1,2\rangle, & \langle 1,2\rangle, & \langle 1,2\rangle, & \langle 1,1\rangle, \\ & \langle 2,4\rangle, & \langle 2,3\rangle, & \langle 2,3\rangle, & \langle 2,4\rangle, & \langle 2,4\rangle, & \langle 2,3\rangle, & \langle 2,3\rangle, \\ & \langle 3,5\rangle, & \langle 3,5\rangle, & \langle 3,5\rangle, & \langle 3,6\rangle, & \langle 3,5\rangle, & \langle 3,5\rangle, & \langle 3,5\rangle, \\ & \langle 4,7\rangle, & \langle 4,7\rangle, & \langle 4,7\rangle, & \langle 4,7\rangle, & \langle 4,7\rangle, & \langle 4,7\rangle, & \langle 4,6\rangle, \\ & \langle 5,9\rangle, & \langle 5,9\rangle, & \langle 5,8\rangle, & \langle 5,9\rangle, & \langle 5,9\rangle, & \langle 5,8\rangle, & \langle 5,8\rangle, \\ & \langle 6,11\rangle, & \langle 6,10\rangle, & \langle 6,10\rangle, & \langle 6,11\rangle, & \langle 6,10\rangle, & \langle 6,10\rangle, & \langle 6,10\rangle \quad \rangle \end{array}$$

Figure 3.18: The value of **WithinScaleInts** after line 11 of ComputeModTable has executed when **Scale** is $\langle 2,2,1,2,2,2,1\rangle$.

by **Scale**, then

$$\textbf{WithinScaleInts}[M_{\text{Cam}}((j-i) \bmod M_{\text{Cam}}) + i - 1] = \langle nci_{ij}, pci_{ij}\rangle$$

for all $1 \leq i, j \leq M_{\text{Cam}}$ and all elements of **WithinScaleInts**. In other words, each element of **WithinScaleInts** gives the *nci* and *pci* of the interval between two degrees of the scale represented by **Scale**. For example, Figure 3.18 gives the value of **WithinScaleInts** after line 11 of ComputeModTable has executed when **Scale** represents the major diatonic scale (i.e., **Scale** $= \langle 2,2,1,2,2,2,1\rangle$).

In lines 12–20 of ComputeModTable, the two-dimensional array, **ModalityTable**, is constructed a row at a time and then returned. For each value of $0 \leq nci < M_{\text{Cam}}$ and $0 \leq pci < N_{\text{Cam}}$, the number of times that an interval with particular values of *nci* and *pci* occurs in the scale represented by **Scale** is determined in line 16 by counting the number of times that the ordered pair $\langle nci, pci\rangle$ occurs in the list **WithinScaleInts**. This value is then divided by M_{Cam} to give the modality for all intervals with name-class interval *nci* and pitch-class interval *pci*. This modality is stored in the variable f and then the ordered triple $\langle nci, f, pci\rangle$ is placed in the appropriate position in **ModalityTable**. The function Count(x, s), called in line 16, simply returns the number of times that item x occurs in sequence s. Figure 3.19 shows the value of **ModalityTable** returned by ComputeModTable for the diatonic major scale.

3.4.3 The ComputeBlendedModTable function

A tonal work is rarely constructed exclusively from diatonic major scales—at the very least, various forms of minor scales are also usually used. Cambouropoulos (1998, p. 73) points out that the modalities of certain common intervals with respect to the major scale are quite different from their modalities with respect to the minor scales. For example, the interval `"rp4"` has a modality of 4/7 with respect to the harmonic minor scale and a modality of 6/7 with respect to the major scale. In order to take all the normal tonal scales into account, Cambouropoulos decided to calculate a 'blended modality' for each interval which is equal to a weighted average of the various modalities that the interval has with respect to the major scale and the various forms of the minor scales.

In the earliest published version of his pitch spelling algorithm, the modality value that Cambouropoulos uses to calculate the modality class of an interval, i, is actually the blended

$$
\begin{array}{lllllllllll}
\langle\langle 0,1,0\rangle, & \langle 0,0,1\rangle, & \langle 0,0,2\rangle, & \langle 0,0,3\rangle, & \langle 0,0,4\rangle, & \langle 0,0,5\rangle, & \langle 0,0,6\rangle, & \langle 0,0,7\rangle, & \langle 0,0,8\rangle, & \langle 0,0,9\rangle, & \langle 0,0,10\rangle, & \langle 0,0,11\rangle\rangle, \\
\langle\langle 1,0,0\rangle, & \langle 1,2/7,1\rangle, & \langle 1,5/7,2\rangle, & \langle 1,0,3\rangle, & \langle 1,0,4\rangle, & \langle 1,0,5\rangle, & \langle 1,0,6\rangle, & \langle 1,0,7\rangle, & \langle 1,0,8\rangle, & \langle 1,0,9\rangle, & \langle 1,0,10\rangle, & \langle 1,0,11\rangle\rangle, \\
\langle\langle 2,0,0\rangle, & \langle 2,0,1\rangle, & \langle 2,0,2\rangle, & \langle 2,4/7,3\rangle, & \langle 2,3/7,4\rangle, & \langle 2,0,5\rangle, & \langle 2,0,6\rangle, & \langle 2,0,7\rangle, & \langle 2,0,8\rangle, & \langle 2,0,9\rangle, & \langle 2,0,10\rangle, & \langle 2,0,11\rangle\rangle, \\
\langle\langle 3,0,0\rangle, & \langle 3,0,1\rangle, & \langle 3,0,2\rangle, & \langle 3,0,3\rangle, & \langle 3,0,4\rangle, & \langle 3,6/7,5\rangle, & \langle 3,1/7,6\rangle, & \langle 3,0,7\rangle, & \langle 3,0,8\rangle, & \langle 3,0,9\rangle, & \langle 3,0,10\rangle, & \langle 3,0,11\rangle\rangle, \\
\langle\langle 4,0,0\rangle, & \langle 4,0,1\rangle, & \langle 4,0,2\rangle, & \langle 4,0,3\rangle, & \langle 4,0,4\rangle, & \langle 4,0,5\rangle, & \langle 4,1/7,6\rangle, & \langle 4,6/7,7\rangle, & \langle 4,0,8\rangle, & \langle 4,0,9\rangle, & \langle 4,0,10\rangle, & \langle 4,0,11\rangle\rangle, \\
\langle\langle 5,0,0\rangle, & \langle 5,0,1\rangle, & \langle 5,0,2\rangle, & \langle 5,0,3\rangle, & \langle 5,0,4\rangle, & \langle 5,0,5\rangle, & \langle 5,0,6\rangle, & \langle 5,0,7\rangle, & \langle 5,3/7,8\rangle, & \langle 5,4/7,9\rangle, & \langle 5,0,10\rangle, & \langle 5,0,11\rangle\rangle, \\
\langle\langle 6,0,0\rangle, & \langle 6,0,1\rangle, & \langle 6,0,2\rangle, & \langle 6,0,3\rangle, & \langle 6,0,4\rangle, & \langle 6,0,5\rangle, & \langle 6,0,6\rangle, & \langle 6,0,7\rangle, & \langle 6,0,8\rangle, & \langle 6,0,9\rangle, & \langle 6,5/7,10\rangle, & \langle 6,2/7,11\rangle\rangle
\end{array}
$$

Figure 3.19: The value of **ModalityTable** returned by ComputeModTable when **Scale** = $\langle 2,2,1,2,2,2,1\rangle$.

modality

$$\text{MDL}_{\text{blended}}(i) = \frac{4 \times \text{MDL}_{\text{ma}}(i) + \text{MDL}_{\text{n.mi}}(i) + \text{MDL}_{\text{d.me.mi}}(i) + \text{MDL}_{\text{a.me.mi}}(i) + 2 \times \text{MDL}_{\text{h.mi}}(i)}{9} \tag{3.3}$$

where

- $\text{MDL}_{\text{ma}}(i)$ is the modality of i with respect to the major scale ($\langle 2,2,1,2,2,2,1\rangle$);
- $\text{MDL}_{\text{n.mi}}(i)$ is the modality of i with respect to the natural minor scale ($\langle 2,1,2,2,1,2,2\rangle$);
- $\text{MDL}_{\text{d.me.mi}}(i)$ is the modality of i with respect to the descending melodic minor scale ($\langle 2,1,2,2,1,2,2\rangle$);
- $\text{MDL}_{\text{a.me.mi}}(i)$ is the modality of i with respect to the ascending melodic minor scale ($\langle 2,1,2,2,2,2,1\rangle$); and
- $\text{MDL}_{\text{h.mi}}(i)$ is the modality of i with respect to the harmonic minor scale ($\langle 2,1,2,2,1,3,1\rangle$).

Note that the natural minor scale is the same as the descending melodic minor scale and both of these are cyclically equivalent to the diatonic major scale. It is therefore not clear why Cambouropoulos did not simply give a weight of 6 to the major diatonic scale instead of weighting each of the three scales separately.

Cambouropoulos (1998, p. 70) then defines the modality class of an interval, i, to be

1. A, if $1 - x \leq \text{MDL}_{\text{blended}}(i) \leq 1$;
2. B, if $x < \text{MDL}_{\text{blended}}(i) < 1 - x$;
3. C, if $0 < \text{MDL}_{\text{blended}}(i) \leq x$; and
4. D, if $\text{MDL}_{\text{blended}}(i) = 0$;

where x is a threshold that he proposes should be made equal to 0.25.[2] If B is to be non-empty, then x may take any value provided that $x < 1 - x$ (i.e., $x < 0.5$).[3]

In line 11 of Cam9698 (see Figure 3.15), the function ComputeBlendedModTable, defined in Figure 3.20, is used to compute a table, **BlendedModTable**, which gives, for every possible interval, i, the blended modality, $\text{MDL}_{\text{blended}}(i)$, as defined in Eq. 3.3, together with the modality class for i. If $nci(i)$ and $pci(i)$ are n and p, respectively, then, when line 11 in Cam9698 has executed, $\textbf{BlendedModTable}[n][p] = \langle n, f, p, c\rangle$ where f is $\text{MDL}_{\text{blended}}(i)$ and c is the modality class of i.

ComputeBlendedModTable takes two arguments, x and **ScaleWtPairList**. x is simply the threshold that defines the boundaries of the modality classes, as described above. In Cam9698, x is set to 0.25 following Cambouropoulos's (1998, p. 70) suggestion (see line 11 of Cam9698 in Figure 3.15). **ScaleWtPairList** is an ordered set in which each element is an ordered pair, $\langle \textbf{Scale}, w\rangle$. In each of these ordered pairs, **Scale** is an ordered set of *pci*s representing a scale and w is the factor by which the modality of an interval with respect to **Scale** is

[2] Cambouropoulos (1998, p. 70) actually defines i to be in modality class C if $0 \leq \text{MDL}_{\text{blended}}(i) \leq x$ but this is obviously an error since it conflicts with the definition of modality class D.

[3] Cambouropoulos (1998, p. 77, Footnote 47) claims, incorrectly, that x must be less than or equal to 0.25.

```
ComputeBlendedModTable(x, ScaleWtPairList)
1   M_Cam ← |ScaleWtPairList[0][0]|
2   N_Cam ← Σ_{i=0}^{M_Cam−1} ScaleWtPairList[0][0][i]
3   n ← |ScaleWtPairList|
4   SumOfWeights ← Σ_{i=0}^{n−1} ScaleWtPairList[i][1]
5   ModTableList ← ⊕_{i=0}^{n−1} ⟨ComputeModTable(ScaleWtPairList[i][0])⟩
6   for i ← 0 to n − 1
7       for nci ← 0 to M_Cam − 1
8           for pci ← 0 to N_Cam − 1
9               ModTableList[i][nci][pci][1] ← ScaleWtPairList[i][1] × ModTableList[i][nci][pci][1]
10  BlendedModTable ← ModTableList[0]
11  for nci ← 0 to M_Cam − 1
12      for pci ← 0 to N_Cam − 1
13          BlendedModTable[nci][pci][1] ← 1/SumOfWeights Σ_{j=0}^{n−1} ModTableList[j][nci][pci][1]
14          f ← BlendedModTable[nci][pci][1]
15          if f ≥ 1 − x
16              ModClass ← A
17          else
18              if f > x
19                  ModClass ← B
20              else
21                  if f > 0
22                      ModClass ← C
23                  else
24                      ModClass ← D
25          BlendedModTable[nci][pci] ← BlendedModTable[nci][pci] ⊕ ⟨ModClass⟩
26  return BlendedModTable
```

Figure 3.20: The ComputeBlendedModTable function.

multiplied in the process of calculating the blended modality for the interval with respect to all the scales in **ScaleWtPairList** (see Eq. 3.3). In accordance with Eq. 3.3, the second argument to ComputeBlendedModTable in line 11 of Cam9698 is therefore set to equal

$$\langle\langle \textbf{MajorScale}, 4\rangle, \langle \textbf{NatMinScale}, 1\rangle, \langle \textbf{DescMelMinScale}, 1\rangle, \langle \textbf{AscMelMinScale}, 1\rangle, \langle \textbf{HarmMinScale}, 2\rangle\rangle$$

where

- **MajorScale** is the *pci* sequence representation of a major scale, $\langle 2, 2, 1, 2, 2, 2, 1\rangle$, (set in line 6 of Cam9698);
- **NatMinScale** is $\langle 2, 1, 2, 2, 1, 2, 2\rangle$, representing the natural minor scale (set in line 7 of Cam9698);
- **DescMelMinScale** is $\langle 2, 1, 2, 2, 1, 2, 2\rangle$, representing the descending melodic minor scale (set in line 8 of Cam9698);
- **AscMelMinScale** is $\langle 2, 1, 2, 2, 2, 2, 1\rangle$, representing the ascending melodic minor scale (set in line 9 of Cam9698); and
- **HarmMinScale** is $\langle 2, 1, 2, 2, 1, 3, 1\rangle$, representing the harmonic minor scale (set in line 10 of Cam9698).

ComputeBlendedModTable will only compute a meaningful blended modality table if all the scales that are being blended have the same values of M_{Cam} and N_{Cam}, which are calculated in lines 1 and 2 of ComputeBlendedModTable. For all the scales being blended in the call to ComputeBlendedModTable in line 11 of Cam9698, $M_{\text{Cam}} = 7$ and $N_{\text{Cam}} = 12$.

In line 3 of ComputeBlendedModTable, the variable n is set to equal the number of scales that are being blended. Then, in line 4, the variable *SumOfWeights* is set to equal the sum of the weighting factors for all the scales in **ScaleWtPairList**. When ComputeBlendedModTable is called in line 11 of Cam9698, *SumOfWeights* becomes equal to 9, in accordance with Eq. 3.3. In line 5 of ComputeBlendedModTable, the function ComputeModTable, discussed above, is used to compute a table of modalities for each of the scales being blended. These modality tables are stored in the list, **ModTableList**.

In lines 6 to 9 of ComputeBlendedModTable, the modality of each interval in each modality table in **ModTableList** is multiplied by the appropriate weighting factor. For example, when ComputeBlendedModTable is called in line 11 of Cam9698, the modality of each interval in the modality table for the major scale is multiplied by 4, the modality of each interval in the modality table for the harmonic minor scale is multiplied by 2 and the modalities in the other tables are left unchanged.

The variable **BlendedModTable**, which is eventually returned in line 26 of ComputeBlendedModTable, is initialised in line 10 to equal the first modality table in **ModTableList**. The nested 'for' loops in lines 11–25 compute the blended modality and modality class for each value of $0 \leq nci < M_{\text{Cam}}$ and $0 \leq pci < N_{\text{Cam}}$ and store these in the appropriate positions in **BlendedModTable**. The blended modality for each value of *nci* and *pci* is calculated in line 13 and this is then used in lines 14–24 to compute the modality class, which is appended to **BlendedModTable**[*nci*][*pci*] in line 25.

Modality class	A		B				C		
Blended modality	1.000	0.762	0.651	0.571	0.429	0.317	0.190	0.048	0.032
Intervals	["p1"]	["rp4"]	["rma2"]	["rmi3"]	["rma3"]	["rmi2"]	["ra4"]	["rd4"]	["ra2"]
		["rp5"]	["rmi7"]	["rma6"]	["rmi6"]	["rma7"]	["rd5"]	["ra5"]	["rd7"]

Table 3.3: The intervals with non-zero modalities in Figure 3.21.

Figure 3.21 shows the value of **BlendedModTable** returned by ComputeBlendedModTable in line 11 of Cam9698. Note that the blended modality for $nci = 6, pci = 9$ is $2/63 \simeq 0.03$ and not 0.33 as given by Cambouropoulos (1996, p. 240, Fig. 5a).

Table 3.3 shows just the intervals that have non-zero modalities in Figure 3.21. Note that setting the modality class boundary constant x to 0.25 causes class A to contain perfect intervals, class B to contain major and minor intervals and class C to contain augmented and diminished intervals that occur in the major and minor scales. This classification agrees with that specified by Cambouropoulos (2003, p. 416) (see page 100).

3.4.4 The SpellWin9698 algorithm

In Cam9698, the best spelling for each window is computed in line 17 using the function SpellWin9698 which is defined in Figure 3.22.

Recall that in SpellWin03 (Figure 3.4) and SpellWin01 (Figure 3.13), only those spellings that are wholly contained within either the flatside or sharpside regions are considered (see Figure 3.3). In SpellWin9698, however, for each window, *every* possible window spelling is evaluated (given that there are three permitted pitch name classes for each 'white note' and two for each 'black note').

Recall also that, in SpellWin03 and SpellWin01, the spelling returned for a particular window is arbitrarily chosen to be the first spelling generated that achieves the lowest penalty score for that window. However, the method implemented in Cam9698 was designed for processing monophonic music and incorporates a 'tie breaker' rule for deciding between two spellings that have the same overall penalty score. This 'tie breaker' rule states that, if the sum of the notational parsimony and interval modality optimization penalties for two window spellings are the same, then the spelling "in which the higher 'quality' intervals appear last" should be preferred (Cambouropoulos, 1996, pp. 242–243). By the "higher 'quality' interval", Cambouropoulos means the one with a higher blended modality. This rule is implemented in the TieBreaker function which is called in line 23 of SpellWin9698 and defined in Figure 3.26.

Note also that SpellWin9698 has no argument corresponding to **PrePNCList** in SpellWin01 or **WinPNCList** in SpellWin03. This is because, in Cam9698, the pitch name classes for the first third of each window are not fixed to be the same as the pitch name classes for the last third of the previous window.

SpellWin9698 takes five arguments, **WinMidiList**, *FirstWindow*, *LastWindow*, **BlendedModTable** and *WinSize*. **WinMidiList** is simply the ordered set of MIDI note numbers for the window currently being spelt. *FirstWindow* is a boolean value that is true iff the window currently being spelt is the first window; and *LastWindow* is a boolean value that is true iff the window currently being spelt is the last window. **BlendedModTable** is equal to the value of

⟨⟨⟨0, 1, 0, A⟩, ⟨0, 0, 1, D⟩, ⟨0, 0, 2, D⟩, ⟨0, 0, 3, D⟩, ⟨0, 0, 4, D⟩, ⟨0, 0, 5, D⟩, ⟨0, 0, 6, D⟩, ⟨0, 0, 7, D⟩, ⟨0, 0, 8, D⟩, ⟨0, 0, 9, D⟩, ⟨0, 0, 10, D⟩, ⟨0, 0, 11, D⟩⟩,
⟨⟨1, 0, 0, D⟩, ⟨1, 20/63, 1, B⟩, ⟨1, 41/63, 2, B⟩, ⟨1, 2/63, 3, C⟩, ⟨1, 0, 4, D⟩, ⟨1, 0, 5, D⟩, ⟨1, 0, 6, D⟩, ⟨1, 0, 7, D⟩, ⟨1, 0, 8, D⟩, ⟨1, 0, 9, D⟩, ⟨1, 0, 10, D⟩, ⟨1, 0, 11, D⟩⟩,
⟨⟨2, 0, 0, D⟩, ⟨2, 0, 1, D⟩, ⟨2, 0, 2, D⟩, ⟨2, 4/7, 3, B⟩, ⟨2, 3/7, 4, B⟩, ⟨2, 0, 5, D⟩, ⟨2, 0, 6, D⟩, ⟨2, 0, 7, D⟩, ⟨2, 0, 8, D⟩, ⟨2, 0, 9, D⟩, ⟨2, 0, 10, D⟩, ⟨2, 0, 11, D⟩⟩,
⟨⟨3, 0, 0, D⟩, ⟨3, 0, 1, D⟩, ⟨3, 0, 2, D⟩, ⟨3, 0, 3, D⟩, ⟨3, 1/21, 4, C⟩, ⟨3, 16/21, 5, A⟩, ⟨3, 4/21, 6, C⟩, ⟨3, 0, 7, D⟩, ⟨3, 0, 8, D⟩, ⟨3, 0, 9, D⟩, ⟨3, 0, 10, D⟩, ⟨3, 0, 11, D⟩⟩,
⟨⟨4, 0, 0, D⟩, ⟨4, 0, 1, D⟩, ⟨4, 0, 2, D⟩, ⟨4, 0, 3, D⟩, ⟨4, 0, 4, D⟩, ⟨4, 0, 5, D⟩, ⟨4, 4/21, 6, C⟩, ⟨4, 16/21, 7, A⟩, ⟨4, 1/21, 8, C⟩, ⟨4, 0, 9, D⟩, ⟨4, 0, 10, D⟩, ⟨4, 0, 11, D⟩⟩,
⟨⟨5, 0, 0, D⟩, ⟨5, 0, 1, D⟩, ⟨5, 0, 2, D⟩, ⟨5, 0, 3, D⟩, ⟨5, 0, 4, D⟩, ⟨5, 0, 5, D⟩, ⟨5, 0, 6, D⟩, ⟨5, 0, 7, D⟩, ⟨5, 3/7, 8, B⟩, ⟨5, 4/7, 9, B⟩, ⟨5, 0, 10, D⟩, ⟨5, 0, 11, D⟩⟩,
⟨⟨6, 0, 0, D⟩, ⟨6, 0, 1, D⟩, ⟨6, 0, 2, D⟩, ⟨6, 0, 3, D⟩, ⟨6, 0, 4, D⟩, ⟨6, 0, 5, D⟩, ⟨6, 0, 6, D⟩, ⟨6, 0, 7, D⟩, ⟨6, 0, 8, D⟩, ⟨6, 2/63, 9, C⟩, ⟨6, 41/63, 10, B⟩, ⟨6, 20/63, 11, B⟩⟩⟩

Figure 3.21: The blended modality table returned by ComputeBlendedModTable in line 11 of Cam9698.

```
SpellWin9698(WinMidiList, FirstWindow, LastWindow, BlendedModTable, WinSize)
1   PNCs ← ⟨⟨"Bs", "Cn", "Dff"⟩, ⟨"Cs", "Df"⟩, ⟨"Css", "Dn", "Eff"⟩, ⟨"Ds", "Ef"⟩,
           ⟨"Dss", "En", "Ff"⟩, ⟨"Es", "Fn", "Gff"⟩, ⟨"Fs", "Gf"⟩, ⟨"Fss", "Gn", "Aff"⟩,
           ⟨"Gs", "Af"⟩, ⟨"Gss", "An", "Bff"⟩, ⟨"As", "Bf"⟩, ⟨"Ass", "Bn", "Cf"⟩⟩
2   n ← |WinMidiList|
3   PCList ← ⨁_{i=0}^{n−1} ⟨WinMidiList[i] mod 12⟩
4   PNCIndexList ← ⨁_{i=0}^{n−1} ⟨0⟩
5   BestSpelling ← ⨁_{i=0}^{n−1} ⟨PNCs[PCList[i]][0]⟩
6   BestPenalty ← ComputeSpellPen9698(BestSpelling, BlendedModTable)
7   NumberOfSpellings ← ∏_{i=0}^{n−1} |PNCs[PCList[i]]|
8   for s ← 1 to NumberOfSpellings − 1
9     ▶ Increment PNCIndexList.
10    i ← 0
11    PNCIndexList[i] ← (PNCIndexList[i] + 1) mod |PNCs[PCList[i]]|
12    while (PNCIndexList[i] = 0) ∧ (i < n − 1)
13      i ← i + 1
14      PNCIndexList[i] ← (PNCIndexList[i] + 1) mod |PNCs[PCList[i]]|

15    ▶ Find new Spelling and SpellingPenalty.
16    Spelling ← ⨁_{i=0}^{n−1} ⟨PNCs[PCList[i]][PNCIndexList[i]]⟩
17    SpellingPenalty ← ComputeSpellPen9698(Spelling, BlendedModTable)

18    ▶ Decide whether BestSpelling should become equal to Spelling.
19    if (SpellingPenalty < BestPenalty)
20      BestSpelling ← Spelling
21      BestPenalty ← SpellingPenalty
22    else
23      if (SpellingPenalty = BestPenalty) ∧ (TieBreaker(Spelling, BestSpelling, BlendedModTable))
24        BestSpelling ← Spelling
25        BestPenalty ← SpellingPenalty

26  ▶ Now return appropriate part of BestSpelling.
27  BSStart ← WinSize/3
28  BSEnd ← 2 × BSStart
29  if FirstWindow
30    BSStart ← 0
31  if LastWindow
32    BSEnd ← n
33  return BestSpelling[BSStart, BSEnd]
```

Figure 3.22: The SpellWin9698 algorithm.

BlendedModTable computed in line 11 of CAM9698. *WinSize* is always equal to the value of the *WinSize* argument in CAM9698 provided by the user.

The first step in SPELLWIN9698 is to define a table, **PNCs**, that gives the set of possible pitch name classes for each different pitch class (cf. variable **PNCs** in COMPUTESPELLING function defined in Figure 3.5). In SPELLWIN9698, **PNCs** is defined so that **PNCs**[*PitchClass*] is an ordered set that contains the possible pitch name classes for pitch class *PitchClass*, sorted so that the pitch name classes decrease in 'sharpness' (i.e., **PNCs**[*PitchClass*][0] is always the sharpest pitch name class that can be assigned to pitch class *PitchClass*).

In line 2 of SPELLWIN9698, the length of **WinMidiList** is stored in the variable n for convenience and then, in line 3, an ordered set, **PCList**, of length n is computed in which **PCList**[i] is the pitch class of **WinMidiList**[i] for all values of $0 \leq i < n$.

Having computed **PCList** and **PNCs**, it is now possible to represent any possible spelling, S, of the current window as an ordered set of indices, **PNCIndexList**, of length n, such that the pitch name class assigned to **WinMidiList**[i] in S is given by **PNCs**[**PCList**[i]][**PNCIndexList**[i]] for all $0 \leq i < n$. For example, if **WinMidiList** is equal to

$$\langle 60, 63, 67, 68, 59, 67, 66, 65, 64 \rangle$$

then the spelling

$$\langle \texttt{"Dff"}, \texttt{"Ef"}, \texttt{"Aff"}, \texttt{"Af"}, \texttt{"Bn"}, \texttt{"Gn"}, \texttt{"Fs"}, \texttt{"Es"}, \texttt{"Dss"} \rangle$$

would be represented by the **PNCIndexList**

$$\langle 2, 1, 2, 1, 1, 1, 0, 0, 0 \rangle .$$

PNCIndexList is therefore essentially a vector representation of a mixed base number in which **PNCIndexList**[0] may be considered the least significant digit (i.e., it is a "little-endian" representation). The base of the ith element (i.e., 'digit') in **PNCIndexList** is |**PNCs**[**PCList**[$i-1$]]|.

In line 4 of SPELLWIN9698, **PNCIndexList** is initialised to be a sequence of n zeros and, in line 5, the variable **BestSpelling** is initialised to equal the spelling for the current window that corresponds to this sequence of zeros. In line 6, the penalty score for this spelling is computed using the function COMPUTESPELLPEN9698 (defined in Figure 3.23) and stored in the variable *BestPenalty*. In line 7, the total number of different spellings that have to be considered for the current window is calculated and stored in the variable *NumberOfSpellings*.

The 'for' loop in lines 8–25 of SPELLWIN9698 iterates through all the remaining spellings for **WinMidiList**, calculating the overall penalty score for each spelling and updating **BestSpelling** and *BestPenalty* when a spelling is found which is the best spelling so far. For each spelling, the first task is to increment **PNCIndexList**. This is done in lines 9–14. If we interpret **PNCIndexList** as being a little-endian, mixed base number, then the first step in incrementing **PNCIndexList** is to add 1 (mod |**PNCs**[**PCList**[0]]|) to **PNCIndexList**[0]. If this makes **PNCIndexList**[0] equal to 0, then we have to then add 1 (mod |**PNCs**[**PCList**[1]]|) to **PNCIndexList**[1]. If this makes **PNCIndexList**[1] equal to zero, then we add 1

(mod |**PNCs**[**PCList**[2]]|) to **PNCIndexList**[2] and so on, until incrementing an element of **PNCIndexList** fails to make it zero. As an example, let us assume, as above, that **WinMidiList** is

$$\langle 60, 63, 67, 68, 59, 67, 66, 65, 64 \rangle$$

so **PCList** must be

$$\langle 0, 3, 7, 8, 11, 7, 6, 5, 4 \rangle .$$

Let us further assume that **PNCIndexList** is

$$\langle 2, 1, 2, 1, 1, 1, 0, 0, 0 \rangle ,$$

corresponding to the spelling

$$\langle \texttt{"Dff"}, \texttt{"Ef"}, \texttt{"Aff"}, \texttt{"Af"}, \texttt{"Bn"}, \texttt{"Gn"}, \texttt{"Fs"}, \texttt{"Es"}, \texttt{"Dss"} \rangle .$$

To increment **PNCIndexList**, we begin by adding 1 (mod |**PNCs**[**PCList**[0]]|) to **PNCIndexList**[0]. **PCList**[0] = 0, therefore |**PNCs**[**PCList**[0]]| = 3, therefore **PNCIndexList**[0] becomes 0, making **PNCIndexList** equal to

$$\langle 0, 1, 2, 1, 1, 1, 0, 0, 0 \rangle .$$

But, because **PNCIndexList**[0] is now 0, we have to add 1 (mod |**PNCs**[**PCList**[1]]|) to **PNCIndexList**[1]. **PCList**[1] = 3, therefore |**PNCs**[**PCList**[1]]| = 2, therefore **PNCIndexList**[1] becomes 0, making **PNCIndexList** equal to

$$\langle 0, 0, 2, 1, 1, 1, 0, 0, 0 \rangle .$$

But, because **PNCIndexList**[1] is now 0, we have to add 1 (mod |**PNCs**[**PCList**[2]]|) to **PNCIndexList**[2] and the process continues until **PNCIndexList** is equal to

$$\langle 0, 0, 0, 0, 2, 1, 0, 0, 0 \rangle .$$

Having incremented **PNCIndexList**, the next step is to compute the spelling that corresponds to the new value of **PNCIndexList** and the overall spelling penalty for this new spelling. This is done in lines 16 and 17 of SpellWin9698. The new spelling and its penalty score are stored in the variables **Spelling** and *SpellingPenalty*, respectively. Then, in lines 18–25, **BestSpelling** and *BestPenalty* are made equal to **Spelling** and *SpellingPenalty*, respectively, but only if *SpellingPenalty* is less than *BestPenalty* or if *SpellingPenalty* = *BestPenalty* and the call to the TieBreaker function in line 23 returns `true`.

Finally, in lines 26–32, the algorithm uses the values of *FirstWindow* and *LastWindow* to determine the appropriate portion of **BestSpelling** to return. If the window being spelt is both the first and the last window, then the whole of **BestSpelling** is returned. Otherwise, if the current window is the first window, the first two thirds of **BestSpelling** are returned. Otherwise, if the current window is the last window, all of **BestSpelling** is returned apart from

```
ComputeSpellPen9698(Spelling, BlendedModTable)
1    SpellingPenalty ← 0
2    for i ← 0 to |Spelling| − 2
3        Enh1 ← IsEnharmonicSpelling(Spelling[i])
4        Enh2 ← IsEnharmonicSpelling(Spelling[i + 1])
5        if Enh1 ⩒ Enh2
6            SpellingPenalty ← SpellingPenalty + 2
7        if Enh1 ∧ Enh2
8            SpellingPenalty ← SpellingPenalty + 6
9        PINC ← PNC2PINC(Spelling[i], Spelling[i + 1])
10       ModClass ← ComputeModClass(PINC, BlendedModTable)
11       if ModClass = C
12           SpellingPenalty ← SpellingPenalty + 1
13       if ModClass = D
14           SpellingPenalty ← SpellingPenalty + 4
15   return SpellingPenalty
```

Figure 3.23: The ComputeSpellPen9698 algorithm.

the first *WinSize*/3 elements. Otherwise, the middle third of **BestSpelling** is returned.

3.4.5 The ComputeSpellPen9698 function

In Cam9698, the penalty score for each spelling in each window is calculated using the ComputeSpellPen9698 function which is called in lines 6 and 17 of SpellWin9698 and defined in Figure 3.23. Like ComputeSpellPen03 and ComputeSpellPen01, ComputeSpellPen9698 takes an argument, **Spelling**, which is an ordered set of pitch name classes representing one possible spelling for a window. However, ComputeSpellPen9698 also takes a second argument, **BlendedModTable**, which it uses to compute the modality class of the intervals in **Spelling**. Every time ComputeSpellPen9698 is called in Cam9698, the value of its argument, **BlendedModTable**, is equal to that of the variable **BlendedModTable** calculated in line 11 of Cam9698.

Recall that in ComputeSpellPen03 and ComputeSpellPen01, the total penalty score for a spelling, **Spelling**, of length n is the sum of the penalty values for each of the $n(n-1)/2$ intervals between (unordered) pairs of pitch name classes in **Spelling**. However, in his 1996 paper, Cambouropoulos (1996, p. 243) only suggests the possibility of applying his "optimisation method... to intervals between noncontiguous notes, e.g., every other note." This indicates that, in the earliest version of this method, the total penalty score for a window spelling was calculated by summing the notational parsimony and interval modality penalties only for intervals between consecutive notes in the window. ComputeSpellPen9698 therefore also only sums the penalties for pairs of consecutive elements in its argument, **Spelling**.

In this earliest version of the method, Cambouropoulos (1996, p. 242) proposes that, for each pair of consecutive pitch name classes in **Spelling**, the total penalty score should be incremented by

- 2, if one of the pitch name classes is enharmonic;
- 6, if both of the pitch name classes are enharmonic;

```
ComputeModClass(PINC, BlendedModTable)
1   NCIPCI ← PINC2NCIPCI(PINC)
2   nci ← NCIPCI[0]
3   pci ← NCIPCI[1]
4   return BlendedModTable[nci][pci][3]
```

Figure 3.24: The ComputeModClass function.

```
PINC2NCIPCI(PINC)
1   CMPI ← PIN2PI(PINC2PIN(PINC))
2   return ⟨(CMPI[1] mod 7, CMPI[0] mod 12)⟩
```

Figure 3.25: The PINC2NCIPCI function.

- 1, if the modality class of the interval between the two pitch name classes is C; and
- 4, if the modality class of the interval between the two pitch name classes is D.

ComputeSpellPen9698 implements this proposed penalty system.

As in ComputeSpellPen03 and ComputeSpellPen01, the function IsEnharmonicSpelling, defined in Figure 3.7 is used in ComputeSpellPen9698 to determine whether or not a pitch name class is enharmonic (see lines 3 and 4 in Figure 3.23). Also, as in ComputeSpellPen03 and ComputeSpellPen01, the function PNC2PINC, defined in Figure 1.25 above, is used in line 9 of ComputeSpellPen9698 to compute the pitch interval name class of the interval between each pair of consecutive pitch name classes in **Spelling**.

In line 10 of ComputeSpellPen9698, the modality class of the interval between each pair of consecutive pitch name classes is computed using the function ComputeModClass which is defined in Figure 3.24. This function takes two arguments, a pitch interval name class, *PINC*, and a blended modality table, **BlendedModTable**, whose value is that of the variable **BlendedModTable** computed in line 11 of Cam9698. In line 1 of ComputeModClass, the function PINC2NCIPCI, defined in Figure 3.25, is used to calculate the ordered pair $\langle nci(PINC), pci(PINC)\rangle$ which is assigned to the variable *NCIPCI*. In lines 2 and 3, the variables *nci* and *pci* are set to equal $nci(PINC)$ and $pci(PINC)$, respectively. Finally, in line 4, the modality class corresponding to *nci* and *pci* is simply looked up in **BlendedModTable**.

The function PINC2NCIPCI, called in line 1 of ComputeModClass, is defined in Figure 3.25. This function takes a pitch interval name class, *PINC*, as its single argument and uses the PIN2PI function, defined in Figure 1.19, and the PINC2PIN function to compute a chromamorphetic pitch interval, *CMPI*, from which the *nci* and *pci* of *PINC* are derived. The function PINC2PIN simply takes a pitch interval name class of the form [*PIN*] and returns *PIN*. For example,

$$\begin{aligned} \text{PINC2PIN}([\texttt{"rma2"}]) &= \texttt{"rma2"}, \\ \text{PINC2PIN}([\texttt{"fma10"}]) &= \texttt{"fma10"}, \text{ and} \\ \text{PINC2PIN}([\texttt{"a1"}]) &= \texttt{"a1"}. \end{aligned}$$

3.4.6 The TieBreaker function

In SpellWin9698 (see Figure 3.22), if the overall penalty value for the current spelling being evaluated (stored in *SpellingPenalty*) is equal to the best penalty score obtained so far (stored in *BestPenalty*), then the function TieBreaker is called in line 23 to decide whether or not the spelling stored in **Spelling** is preferable to the one stored in **BestSpelling**.

The TieBreaker function is defined in Figure 3.26 and implements my interpretation of Cambouropoulos's (1996, p. 243) 'tie-breaker' rule which states that, if two spellings for a given window both achieve the lowest overall penalty score, then the spelling is chosen "in which the *higher "quality" intervals appear last*" (the italics are Cambouropoulos's). Cambouropoulos goes on to explain that

> when there are two alternative spellings of two intervals the system prefers the sequence in which the last interval belongs to a "better" modality class.

Modality class A is considered "better than" class B, which is "better than" class C, which is "better than" class D. As an example, Cambouropoulos states that

> this rule gives precedence e.g., to the sequence G – G♯ – A over the equivalent G – A♭ – A (they both have a total value of 4).

The statement, "they both have a total value of 4", means that, for each sequence, the sum of the notational parsimony and interval modality penalties is 4.

From this explanation, it is fairly clear that if one has two three-note sequences with the same overall penalty score, then one should choose the sequence in which the interval between the second and third notes belongs to the better modality class. However, it is not clear how the rule applies to sequences containing more than three notes. I therefore asked Cambouropoulos if he could give me more details as to how he had implemented the rule. In his reply, he stated that the

> initial melodic implementation [i.e., the version of the algorithm described by Cambouropoulos (1996, 1998)] could produce more than one highest equal rating spellings. Within these few 'winners' only one would be selected that fulfilled the requirement of the third rule. E.g. between the equal scoring sequences
>
> A - F - F♯ - G - A
> A - F - G♭ - G - A
>
> the system would choose the first possibility.
>
> (Cambouropoulos, 2002)

Let S_1 and S_2 be defined as follows,

$$S_1 = \langle \texttt{"An"}, \texttt{"Fn"}, \texttt{"Fs"}, \texttt{"Gn"}, \texttt{"An"} \rangle ,$$
$$S_2 = \langle \texttt{"An"}, \texttt{"Fn"}, \texttt{"Gf"}, \texttt{"Gn"}, \texttt{"An"} \rangle .$$

Now let M_1 be the *modality class sequence* for S_1, that is, the sequence in which $M_1[i]$ is the modality class of the interval from $S_1[i]$ to $S_1[i+1]$. Also, let M_2 be the modality class sequence for S_2. That is,

$$\begin{aligned} M_1 &= \langle \mathsf{B}, \mathsf{D}, \mathsf{B}, \mathsf{B} \rangle , \\ M_2 &= \langle \mathsf{B}, \mathsf{B}, \mathsf{D}, \mathsf{B} \rangle . \end{aligned}$$

For convenience, let us also define an *HL-pair* (standing for 'high quality–low quality interval pair') to be a sequence of three pitch names for which the modality class sequence is a member of the following set:

$$\{ \langle \mathsf{A}, \mathsf{C} \rangle , \langle \mathsf{A}, \mathsf{D} \rangle , \langle \mathsf{B}, \mathsf{C} \rangle , \langle \mathsf{B}, \mathsf{D} \rangle , \langle \mathsf{C}, \mathsf{D} \rangle \} .$$

Note that for the purposes of the tie-breaker rule, I am assuming that Cambouropoulos would consider the difference in 'quality' between modality class A and B intervals to be too small for the sequence $\langle \mathsf{A}, \mathsf{B} \rangle$ to be considered an HL-pair. I am also assuming that Cambouropoulos would consider the difference in 'quality' between a rare interval that *does* occur in a major or minor scale (i.e., modality class C) and an interval that does *not* occur in any of the tonal scales (i.e., modality class D) to be sufficiently large for the sequence $\langle \mathsf{C}, \mathsf{D} \rangle$ to be considered an HL-pair.

Cambouropoulos states that the tie-breaker rule chooses the sequence in which the "highest quality interval appears last", but the last interval in S_1 has the same modality class as the last interval in S_2 (i.e, B), therefore, on this interpretation, the rule would be incapable of deciding between S_1 and S_2. However, Cambouropoulos claims that the tie-breaker would select S_1 in favour of S_2, therefore this interpretation of the rule must be incorrect.

Both M_1 and M_2 contain one HL-pair (i.e., $M_1[0,2]$ and $M_2[1,3]$). It would seem that the tie-breaker rule selects S_1 in favour of S_2 because the HL-pair in S_1 occurs earlier in the sequence than the HL-pair in S_2. Now, if the sequence of notes to be spelt constitutes a *group* in Lerdahl and Jackendoff's (1983, pp. 12–17, 36–67) sense, then it might be possible to argue that a spelling in which the last pair of intervals is not an HL-pair is preferable (e.g., by invoking a form of Krumhansl's (1990, pp. 150–151) "contextual asymmetry" principle). However, if the HL-pairs in two equally scoring spellings occur in positions other than at the ends of groups, it is hard to see why one should prefer the spelling in which the HL-pairs simply happen to occur earlier in some arbitrary musical segment such as a window that happens to contain *WinSize* contiguous notes in a melody. There is therefore no clear theoretical justification for preferring a spelling for some arbitrary window just because the HL-pairs occur earlier in the spelling than they do in some other spelling that incurs the same overall penalty value as calculated by ComputeSpellPen9698.

Nevertheless, according to Cambouropoulos, the tie-breaker rule should prefer S_1 to S_2. Therefore, in my TieBreaker function, if two window spellings have the same overall penalty value and contain the same number of HL-pairs, the one in which the HL-pairs occur earlier is preferred.

The TieBreaker function, defined in Figure 3.26 takes three arguments: two window spellings, $\mathbf{S}_1$ and $\mathbf{S}_2$; and the blended modality table, **BlendedModTable**. Every time

```
TieBreaker(S1, S2, BlendedModTable)
1    n ← |S1|
2    M1 ← ⨁_{i=0}^{n−2} ⟨ComputeModClass(PNC2PINC(S1[i], S1[i + 1]), BlendedModTable)⟩
3    M2 ← ⨁_{i=0}^{n−2} ⟨ComputeModClass(PNC2PINC(S2[i], S2[i + 1]), BlendedModTable)⟩
4    H1 ← ⟨ ⟩
5    H2 ← ⟨ ⟩
6    for i ← 0 to n − 3
7        if (M1[i] ≠ M1[i + 1]) ∧ (M1[i] ∈ {A, B, C}) ∧ (M1[i + 1] ∈ {C, D})
8            H1 ← H1 ⊕ ⟨i⟩
9        if (M2[i] ≠ M2[i + 1]) ∧ (M2[i] ∈ {A, B, C}) ∧ (M2[i + 1] ∈ {C, D})
10           H2 ← H2 ⊕ ⟨i⟩
11   if |H1| < |H2|
12       return true
13   if |H1| > |H2|
14       return false
15   P1 ← Σ_{p∈H1} p
16   P2 ← Σ_{p∈H2} p
17   return P1 < P2
```

Figure 3.26: The TieBreaker function.

TieBreaker is called in Cam9698, **BlendedModTable** is equal to the value of the **BlendedModTable** variable computed in line 11 of Cam9698. TieBreaker returns true iff it determines that $\mathbf{S}_1$ is preferable to $\mathbf{S}_2$. It is also assumed throughout TieBreaker that $\mathbf{S}_1$ and $\mathbf{S}_2$ are alternative spellings for the same sequence of MIDI note numbers.

For convenience, in line 1 of TieBreaker, the variable n is set to equal the length of $\mathbf{S}_1$. Then, in lines 2 and 3, the modality class sequences are computed for $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively, and stored in the variables $\mathbf{M}_1$ and $\mathbf{M}_2$, respectively. This is done using the ComputeModClass function, defined in Figure 3.24 above. In lines 4–10, the sequences $\mathbf{H}_1$ and $\mathbf{H}_2$ are computed. When the 'for' loop in lines 6–10 has terminated, $\mathbf{H}_1$ and $\mathbf{H}_2$ contain the positions at which HL-pairs occur in $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively. If $\mathbf{S}_1$ contains fewer HL-pairs than $\mathbf{S}_2$, then $|\mathbf{H}_1| < |\mathbf{H}_2|$ in line 11 and TieBreaker returns true in line 12. Otherwise, if $\mathbf{S}_1$ contains more HL-pairs than $\mathbf{S}_2$, then $|\mathbf{H}_1| > |\mathbf{H}_2|$ in line 13 and TieBreaker returns false in line 14. Execution only reaches line 15 if both spellings contain the same number of HL-pairs. In this case, TieBreaker must determine the spelling in which the HL-pairs "occur earlier". This is done in lines 15–17 by summing, for each spelling, the positions at which HL-pairs occur in that spelling, and then choosing the spelling for which this sum is least. This can be done simply by summing the elements of $\mathbf{H}_1$ to get a penalty value P_1 (line 15) and summing the elements of $\mathbf{H}_2$ to get a penalty value P_2 (line 16). The function then returns true iff $P_1 < P_2$ (line 17).

3.4.7 Time and space complexities of Cam9698

The running time of lines 1–11 in Cam9698 is independent of the input size. However, line 11 can take a significant amount of time to run. In ComputeBlendedModTable, the worst case running time of lines 6–25 is $O(nM_{\text{Cam}}N_{\text{Cam}})$. However, the time complexity of line 5 is

$O(nT)$ where n is the number of scales being blended and T is the time complexity of ComputeModTable.

In ComputeModTable, the worst case running time of lines 4–11 is $O(M_{\text{Cam}}^3)$. The running time of line 16 is proportional to the size of **WithinScaleInts** which is $O(M_{\text{Cam}}^2)$. However, if **WithinScaleInts** is stored as a data-structure such as a hash table which allows Count($\langle nci, pci \rangle$, **WithinScaleInts**) to be retrieved in constant time, then the running time of line 16 can be reduced to $O(1)$. The overall worst-case running time of ComputeModTable is therefore $O(M_{\text{Cam}}^3 + M_{\text{Cam}}N_{\text{Cam}})$ if line 16 can be executed in $O(1)$ time. However, it is $O(N_{\text{Cam}}M_{\text{Cam}}^3)$ if **WithinScaleInts** is stored as a list and Count has to traverse this list each time it runs.

This means that the overall running time of ComputeBlendedModTable is $O(nN_{\text{Cam}}M_{\text{Cam}}^3)$ if line 16 of ComputeModTable is implemented so that it runs in $O(M_{\text{Cam}}^2)$ time. However, this can be reduced to $O(n(M_{\text{Cam}}^3 + M_{\text{Cam}}N_{\text{Cam}}))$ if line 16 of ComputeModTable runs in constant time. The worst case running time of line 11 of Cam9698 could therefore be as bad as $O(nN_{\text{Cam}}M_{\text{Cam}}^3)$ where n is the number of scales blended (i.e., 5), M_{Cam} is the number of elements in each scale representation (i.e., 7) and N_{Cam} is the sum of the elements in each scale (i.e., 12).

Nevertheless, as already pointed out, the running time of lines 1–11 of Cam9698 is independent of the input size and therefore becomes insignificant in the asymptotic worst-case running time of the algorithm. As in Cam03 and Cam01, the number of windows processed is $O(|\textbf{MidiList}|/\mathit{WinSize})$ and the 'while' loop in lines 12–20 executes once for each window. All lines in this loop run in constant time apart from line 17 in which SpellWin9698 is called. In SpellWin9698, $|\textbf{WinMidiList}| = \mathit{WinSize}$ for every window apart from possibly the last which may be shorter. The 'for' loop in lines 8–25 executes once for each window and, in the worst case, the total number of spellings for each window is $O(3^{\mathit{WinSize}})$. On each of these iterations, lines 10–16 have a running time of $O(\mathit{WinSize})$. The functions ComputeSpellPen9698 and TieBreaker, called in lines 17 and 23 of SpellWin9698, respectively, both have worst-case running times of $O(\mathit{WinSize})$. Therefore, in the worst case, lines 9–25 of SpellWin9698 run in $O(\mathit{WinSize})$ time. This implies that the overall worst-case running time of SpellWin9698 is $O(3^{\mathit{WinSize}} \times \mathit{WinSize})$ which implies that the overall worst-case running time of Cam9698 is

$$O\left(\frac{|\textbf{MidiList}|}{\mathit{WinSize}} \times \mathit{WinSize} \times 3^{\mathit{WinSize}}\right) = O\left(|\textbf{MidiList}| \times 3^{\mathit{WinSize}}\right). \tag{3.4}$$

$O(M_{\text{Cam}}N_{\text{Cam}})$ space is required to store the variable **BlendedModTable** which is computed in line 11 of Cam9698. If the algorithm is implemented directly as shown in Figure 3.15, then the whole of **PNCList** is stored in memory before being returned in line 21. In this case, the algorithm would use $O(|\textbf{MidiList}|)$ space. However, if the pitch name classes for the retained segment of each window are written to ouput in line 17 instead of appended to **PNCList**, the overall space used by the algorithm would be reduced to $O(\mathit{WinSize})$.

Penalty	0	1	2	3	4	5	6	7	8	9	10	11	12
PINC	["p1"]	["rp5"]	["rma2"]	["rma6"]	["rma3"]	["rma7"]	["ra4"]	["a1"]	["ra5"]	["ra2"]	["ra6"]	["ra3"]	["ra7"]
		["rp4"]	["rmi7"]	["rmi3"]	["rmi6"]	["rmi2"]	["rd5"]	["d1"]	["rd4"]	["rd7"]	["rd3"]	["rd6"]	["rd2"]

Table 3.4: Table of interval optimisation penalties used in CAM03B. The penalty for each pitch interval name class, *PINC*, is equal to the distance along the line of fifths corresponding to *PINC*. (Reproduced from Cambouropoulos (2003, p. 416, Table 1), with Cambouropoulos's interval symbols converted to my pitch interval name class notation.)

Penalty	0	1	2	3	4	5	6	7	8	9	10	11	12
PINC	["p1"]	["rp5"]	["rma2"]	["rma6"]	["rma3"]	["rma7"]	["ra4"]	["ra5"]	["ra2"]	["ra6"]	["ra3"]	["ra7"]	["a1"]
		["rp4"]	["rmi7"]	["rmi3"]	["rmi6"]	["rmi2"]	["rd5"]	["rd4"]	["rd7"]	["rd3"]	["rd6"]	["rd2"]	["d1"]

Table 3.5: Table of interval optimisation penalties used in CAM03C.

3.5 Cambouropoulos's own evaluations of his pitch spelling algorithms

3.5.1 Cambouropoulos's (2003) own evaluation of CAM03

Cambouropoulos (2003, pp. 422–426) tested three versions of CAM03 on two test corpora.

The two corpora he used were

1. ten Mozart piano sonatas (K279–K284, K330–K333), containing, in total, 54418 notes; and

2. three Chopin waltzes (Op. 64, Nos. 1–3), containing, in total, 4876 notes.

The first version of the algorithm Cambouropoulos tested, which I shall denote by CAM03B, was essentially the same as CAM03, except that the 'interval optimization' penalty for each interval within a window was defined to be the distance along the line of fifths corresponding to the interval (Cambouropoulos, 2003, p. 423). For example, in CAM03B, a `"rma3"` would incur an interval optimization penalty of 4 whilst an `"a1"` (a rising chromatic semitone) would incur an interval optimization penalty of 7. Although the distance along the line of fifths for an interval can be computed directly (e.g., by using the algorithm in Figure 1.22), Cambouropoulos (2003, p. 423) seems to have actually computed the interval optimization penalties in CAM03B using a table as shown in Table 3.4. In CAM03B, Cambouropoulos (2003, p. 423) set the notational parsimony penalty to 13, because it was "larger by one than the highest distance value" in Table 3.4.

The second version of CAM03 tested by Cambouropoulos, which I shall denote by CAM03C, was the same as CAM03B, except that the interval optimization penalty incurred by [`"a1"`] and [`"d1"`] was increased from 7 to 12 and the intervals with penalties in Table 3.4 between 8 and 12 had their penalties reduced by 1 (Cambouropoulos, 2003, p. 423). In other words, the interval optimization penalties in CAM03C were looked up in Table 3.5 instead of Table 3.4.

The third version of CAM03 tested by Cambouropoulos was essentially the same as CAM03 and I shall denote this version of the algorithm by CAM03A. It appears that Cambouropoulos used a window size of 9 for all three versions of the algorithm in his evaluation, therefore

(a) Percentage of notes spelt correctly.

	CAM03B	CAM03C	CAM03A
Mozart (54418 notes)	98.57%	98.71%	98.83%
Chopin (4876 notes)	94.05%	95.84%	95.86%

(b) Number of notes spelt incorrectly.

	CAM03B	CAM03C	CAM03A
Mozart (54418 notes)	778	701	634
Chopin (4876 notes)	290	203	202

Table 3.6: The results obtained by Cambouropoulos (2003) when CAM03A, CAM03B and CAM03C were run on two test corpora consisting of 10 Mozart sonatas and 3 Chopin waltzes. Table (a) shows the percentage of notes spelt correctly by each algorithm in each test corpus. Table (b) shows the number of notes spelt incorrectly by each algorithm in each test corpus. (From Cambouropoulos, 2003, pp. 424–425, Tables 4–6.)

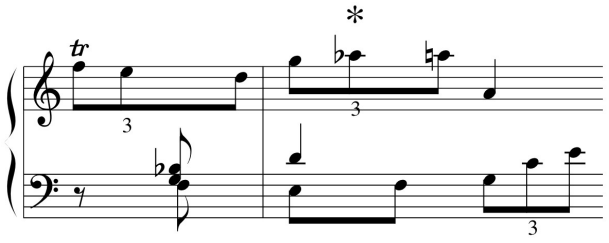


Figure 3.27: Excerpt from Mozart's Sonata in C major, K279, as spelt by CAM03A. The `"Af5"` marked by an asterisk is spelt as a `"Gs5"` in the original. (Reproduced from Cambouropoulos, 2003, p. 425, Fig. 10.)

'CAM03A' signifies specifically the CAM03 algorithm as defined above, run with the *WinSize* parameter set to 9.

Tables 3.6(a) and (b) show the results Cambouropoulos obtained when these three versions of CAM03 were run on his two test corpora. As can be seen in Tables 3.6(a) and (b), the version of the algorithm that used the (incorrect) interval modality class categorization in Table 3.1 performed better on both test corpora than the versions of the algorithm based on the line of fifths. Of the two versions of the algorithm based on the line of fifths, CAM03C, the one in which [`"a1"`] and [`"d1"`] are 'demoted', performed better. Note, however, that on the Chopin corpus, the number of notes spelt correctly by CAM03A and CAM03C differed by only one.

Figures 3.27 and 3.28 illustrate three of the errors made by CAM03A. Cambouropoulos (2003, pp. 425–6) claims that these errors arise because the algorithm fails to take voice-leading into account. This claim will be investigated in more depth in my own evaluation of the algorithms described below.

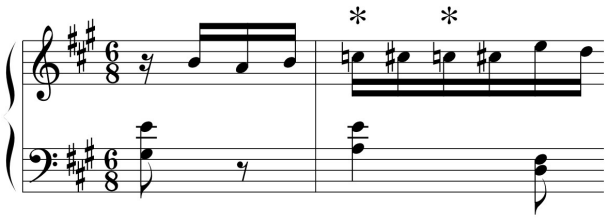


Figure 3.28: Excerpt from Mozart's Sonata in A major, K331 (Var. I), as spelt by Cam03A. The "Cn5"s marked by asterisks are spelt as "Bs4"s in the original. (Reproduced from Cambouropoulos, 2003, p. 426, Fig. 11.)

(a) Percentage of notes spelt correctly.

	Cam01B	Cam01C	Cam01A
Mozart (40058 notes)	98.38%	98.75%	98.94%

(b) Number of notes spelt incorrectly.

	Cam01B	Cam01C	Cam01A
Mozart (40058 notes)	649	501	424

Table 3.7: The results obtained by Cambouropoulos (2001) when Cam01A, Cam01B and Cam01C were run on eight Mozart sonatas. Table (a) shows the percentage of notes spelt correctly by each algorithm. Table (b) shows the number of notes spelt incorrectly by each algorithm. (From Cambouropoulos, 2001, p. 6, Tables 4–6.)

3.5.2 Cambouropoulos's (2001) own evaluation of Cam01

Cambouropoulos (2001, pp. 5–8) tested three versions of Cam01 on a test corpus consisting of eight Mozart piano sonatas (K279–K283, K331–K333) and containing, in total, 40058 notes.

The first version of Cam01 that Cambouropoulos tested, which I shall denote by Cam01B, was essentially the same as Cam01, except that the penalties for each interval were calculated in the same way as in Cam03B, mentioned above (Cambouropoulos, 2001, p. 6). That is, the interval optimization penalty was looked up in Table 3.4 and the notational parsimony penalty was defined to be 13.

The second version of Cam01 tested by Cambouropoulos, which I shall denote by Cam01C, was the same as Cam01B, except that the interval optimization penalties were derived in the same way as in Cam03C—that is, they were looked up in Table 3.5 (Cambouropoulos, 2001, p. 6).

The third version of Cam01 tested by Cambouropoulos, which I shall denote by Cam01A, was essentially the same as Cam01. It appears that Cambouropoulos used a window size of 9 for all three versions of the algorithm in his evaluation, therefore 'Cam01A' signifies specifically the Cam01 algorithm as defined above, run with the *WinSize* parameter set to 9.

Tables 3.7(a) and (b) show the results obtained when Cambouropoulos ran these three versions of Cam01 on the 8 Mozart sonatas. As in his evaluation of the Cam03 algorithm, Cam-

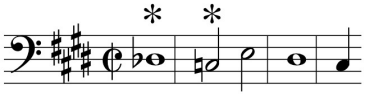


Figure 3.29: The subject of Bach's Fugue in C♯ minor (BWV 849) as spelt by Cambouropoulos's implementation of CAM9698, CAM9698A. The `"Df3"` and `"Cn3"` at the beginning of the subject (marked with asterisks) are spelt in the original as `"Cs3"` and `"Bs2"`, respectively. (Reproduced from Cambouropoulos, 1998, p. 79.)

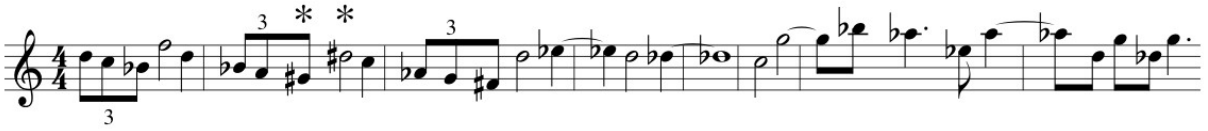


Figure 3.30: Excerpt from the cor anglais solo from Act III of Wagner's *Tristan and Isolde* as spelt by Cambouropoulos's implementation of CAM9698, CAM9698A. The `"Gs4"` and `"Ds5"` marked by asterisks are spelt as `"Af4"` and `"Ef5"` in the original. (Reproduced from Cambouropoulos, 1998, p. 80.)

bouropoulos found

1. that the version which performed best (CAM01A) was the one that used the (incorrect) modality class categorisation given in Table 3.1; and

2. that, of the two versions based on the line of fifths, the one that performed better (CAM01C) was the one in which [`"a1"`] and [`"d1"`] had been 'demoted'.

The percentages given in Table 3.7 differ from those given by Cambouropoulos (2001, p. 6, Tables 4–6) because Cambouropoulos's percentages were calculated by dividing the number of mis-spelled notes by the number of *notes with accidentals* in the corpus. However, there is no guarantee that every mis-spelled note is a note that has an accidental in the original score. It is therefore possible that the results given by Cambouropoulos (2001) do not represent all the errors made by the algorithms on his test corpus.

3.5.3 Cambouropoulos's (1996, 1998) own evaluation of CAM9698

Cambouropoulos (1998, pp. 78–80) tested his implementation of the CAM9698 algorithm (which I shall denote by CAM9698A) on a set of melodies including the theme of J. S. Bach's *Musical Offering* (BWV 1079), the 24 fugue subjects from the first book of J. S. Bach's *Das Wohltemperirte Clavier* (BWV 846–869), and several other melodies from later periods such as the opening of Chopin's *Ballade*, Op. 23, and an extract from the cor anglais solo from Act III of Wagner's *Tristan and Isolde*.

CAM9698A spelt the subject of Bach's Fugue in C♯ minor (BWV 849) in the way shown in Figure 3.29 and the Wagner excerpt in the way shown in Figure 3.30. In both cases, the incorrectly spelt notes are marked with asterisks. Cambouropoulos (1998, pp. 79–80) suggests that the errors made by CAM9698A in these examples could be avoided

if additional rules are applied such as 'avoid enharmonic spellings of a tone in a single passage', or if the optimisation method is additionally applied to intervals between non-contiguous notes, e.g. every other note.

As explained above, in his later versions of the algorithm, Cambouropoulos did indeed apply his optimization method to "intervals between non-contiguous notes". Cambouropoulos (2004) has confirmed that when he suggested "avoid[ing] enharmonic spellings of a tone in a single passage", what he actually meant was to avoid different enharmonic spellings of the same MIDI note number within a single window. In CAM01 it is impossible for the same MIDI note number to be spelt in two different ways in the same window, however there are no controls against this happening in CAM03 and CAM9698. I shall explore in more depth the effect of implementing this rule in my own evaluation described below.

3.6 A more thorough evaluation of Cambouropoulos's pitch spelling algorithms

3.6.1 Introduction

Although the three pitch spelling algorithms described by Cambouropoulos in his publications are distinct, they all have certain basic features in common. To be more specific,

1. each uses an overlapping windowing technique;
2. each one searches through a set of spellings for each window and chooses the one that achieves the least overall penalty score;
3. in each algorithm, the overall penalty score for a given window spelling is determined by adding together the individual penalty values for the intervals between the pitch name classes in the window spelling; and
4. each algorithm uses implementations of Cambouropoulos's principles of 'interval optimization' and 'notational parsimony' to compute the penalty value for each interval in a window spelling.

In a sense, any algorithm that had these four basic features would be a version of Cambouropoulos's pitch spelling algorithm. However, there are many ways in which two such versions of his algorithm could differ in their detailed features. For example,

- one might use a variable length window and the other a fixed length window;
- one might use the TIEBREAKER function and the other might not;
- the two algorithms might use different specific values for the interval optimization penalties;

and so on. Ideally, one would want to run *every* possible version of Cambouropoulos's algorithm on a single, varied, large test corpus in order to determine the version with the particular combination of detailed features that achieves the highest spelling accuracy. However, this would take an unreasonable amount of time. Instead, I have carried out a more modest evaluation in

which I compared the performance of twenty-six versions of Cambouropoulos's algorithm on the test corpus $\mathcal{C}$ defined in Table 1.4 above. The versions of the algorithm evaluated included those tested by Cambouropoulos himself together with other versions which were carefully selected so that an estimate could be obtained as quickly as possible of that combination of detailed features which results in the best performance.

3.6.2 The ways in which two versions of Cambouropoulos's algorithm may differ

In Table 3.8, I have attempted to list all the important ways in which two runs of Cambouropoulos's algorithm on the same data may differ. From this point on, I shall call each of these "ways in which two runs of the algorithm may differ" a *variable feature* of the algorithm. The first column in Table 3.8 gives a summary description of the variable feature. Each entry in the second column gives information about the values that the variable feature in that row may take. For example, the expression $3n$ in the first row indicates that the window size may be any positive integer multiple of 3. Each entry in columns 3–9 of Table 3.8 gives the specific value taken for a particular variable feature by a particular version of the algorithm evaluated by Cambouropoulos. For example, the value 'sca' in the fifth row of the column headed CAM9698A indicates that in the version of CAM9698 evaluated by Cambouropoulos, the modality class of each interval was computed directly by calculating the frequency of occurrence of the intervals in the tonal **sca**les, rather than by looking them up in **Tab**le 3.1. A blank entry implies that the variable feature does not apply to that version of the algorithm. The meanings of the other entries in Table 3.8 will be explained below.

3.6.2.1 Window size (Table 3.8, row 1)

Perhaps the most obvious variable feature of the algorithm is the size of the window used. This is controlled by the value assigned by the user to the *WinSize* parameter in CAM03, CAM01 and CAM9698 (see Table 3.8, row 1). Cambouropoulos (2003, p. 420) states that

> allowing a larger section to be spelled in each step... gives greater stability to the pitch-spelling process because a larger pitch context is taken into account and abrupt changes at the edges of the window are avoided.

This suggests that assigning larger values to the parameter *WinSize* might lead to the algorithm making fewer spelling errors. Unfortunately, if the window size is increased from w to $w + k$ then the worst case running time will increase by a factor of

- $(1 + k/w) \times 2^{2k/3}$ for CAM01 and CAM03 (see Eqs. 3.1 and 3.2); and
- 3^k for CAM9698(see Eq. 3.4).

This sets a fairly low practical upper limit on the size of window that can be used (typically 12). Cambouropoulos appears to have used a window size of 9 in all the evaluations described in his publications, except perhaps in CAM9698A. As discussed in section 3.4.1 above, Cambouropoulos (1996, p. 245) proposes an unworkable windowing procedure in which each window contains

			CAM03A	CAM01A	CAM9698A	CAM03B	CAM03C	CAM01B	CAM01C
1	Window size	$3n$	9	9	12	9	9	9	9
2	Fixed or variable length window	(fix,var)	fix	var	fix	fix	fix	var	var
3	How the notes are sorted in **MidiList** (voice-leading)	(int,end)	int	int	end	int	int	int	int
4	Method of computing interval optimisation penalties (IOPs)	(mod,lof)	mod	mod	mod	lof	lof	lof	lof
5	If modality-based IOP calculation, how is modality class determined?	(tab,sca)	tab	tab	sca				
6	If modality-based IOP calculation, what penalties applied to modality classes?	A,B,C,D	0,0,1,2	0,0,1,3	0,0,1,4				
7	If line-of-fifths-based IOP calculation, are ["a1"] and ["d1"] 'demoted'?	(yes,no)				no	yes	no	yes
8	Penalty values for notational parsimony	one,both	2,4	4,8	2,6	13,26	13,26	13,26	13,26
9	Spellings constrained to being within either flatside or sharpside region	(yes,no)	yes	yes	no	yes	yes	yes	yes
10	Retain pitch name classes for first third of window from previous window	(yes,no)	yes	yes	no	yes	yes	yes	yes
11	Include interval penalties for non-contiguous notes in window	(yes,no)	yes	yes	no	yes	yes	yes	yes
12	If fixed-length window and voices end-to-end, is TIEBREAKER used?	(yes,no)			yes				
13	If modality-based IOP derived directly from scales, how are scales weighted?	dia,amm,mh			6,1,2				
14	If modality-based IOP derived from scales, what are values of modality class boundaries?	$0 < x < 0.5$			0.25				
15	If variable-length window used, does each window contain fixed number of MIDI note numbers or pitch classes?	(mid,pc)		mid				mid	mid
16	For each MIDI note number or pitch class within single window, pitch name class is constant	(mid,pc,no)	no	mid	no	no	no	mid	mid
17	Uses "flat/natural-or-sharp/natural-first" heuristic	(yes,no)	no	no	no	no	no	no	no
18	Includes method for analysing metric structure	(yes,no)	no	no	no	no	no	no	no
19	Includes method for identifying accented notes	(yes,no)	no	no	no	no	no	no	no
20	Weight each interval by durations of notes involved in it	(yes,no)	no	no	no	no	no	no	no
21	Weight each note by distance in pitch from note to middle pitch range	(yes,no)	no	no	no	no	no	no	no
22	Includes method for analysing hierarchical structure	(yes,no)	no	no	no	no	no	no	no
23	Includes method for detecting secondary ornamental notes	(yes,no)	no	no	no	no	no	no	no
24	Uses timing information for weighting interval penalties	(yes,no)	no	no	no	no	no	no	no
25	If spellings constrained to be within flatside or sharpside, use first third of window to eliminate one side	(yes,no)	no	no	no	no	no	no	no
26	Rejects window spellings that contain more than one enharmonic pitch name class	(yes,no)	no	no	no	no	no	no	no
27	Rejects window spellings that contain consecutive enharmonic pitch name classes	(yes,no)	no	no	no	no	no	no	no
28	If modality-based IOP calculation, uses *PINC*s instead of $\langle nci, pci \rangle$ pairs	(yes,no)	no	no	no	no	no	no	no
29	Rejects spellings that contain modality class D intervals (except ["a1"] or ["d1"])	(yes,no)	no	no	no	no	no	no	no

Table 3.8: The variable features of Cambouropoulos's pitch spelling algorithm, showing the value taken for each variable feature by each version of the algorithm tested by Cambouropoulos. See text for explanation.

13 elements. This could most closely be approximated by running CAM9698 with *WinSize* set to 12.

3.6.2.2 Fixed or variable length window (Table 3.8, row 2)

A given version of Cambouropoulos's algorithm may use either fixed length windows as in CAM03 and CAM9698, or variable length windows as used in CAM01. The motivation for using a variable length window was discussed in section 3.3.1 above. In row 2 of Table 3.8, the values 'fix' and 'var' indicate that a particular version of the algorithm uses fixed length and variable length windows, respectively.

3.6.2.3 How the notes are sorted in MidiList (voice-leading) (Table 3.8, row 3)

One would expect the performance of Cambouropoulos's algorithm to depend heavily on the way that the notes are sorted in the list of MIDI note numbers, **MidiList**, that each version of the algorithm takes as input. If the piece to be processed is polyphonic, then there are essentially two sensible ways in which the notes in **MidiList** could be sorted: with the voices arranged 'end-to-end' or with the voices 'interleaved', as described in section 2.4.2.3 above. If there is no voice information in the input data from which **MidiList** is derived, then the notes will be in the order in which they occur in the music with notes beginning simultaneously being ordered either arbitrarily or sorted by MIDI note number. In this case, the resulting order will usually be very similar to that obtained when the voices are interleaved. If the music to be processed is monophonic, then **MidiList** will always be sorted so that the notes are in the order in which they occur in the music. In row 3 of Table 3.8, I have therefore made the simplifying assumption that the notes in **MidiList** will either be ordered so that the voices are arranged end-to-end (in which case the value entered in the table is 'end') or interleaved (in which case the value in the table is 'int').

In CAM03 and CAM01, Cambouropoulos assumes that **MidiList** is sorted so that the voices are (approximately) interleaved. Therefore one would not expect these algorithms to perform better when **MidiList** is sorted so that the voices are arranged end-to-end. On the other hand, the TIEBREAKER function used in CAM9698 only makes any sense at all if the music to be processed is monophonic. One would therefore expect versions of CAM9698 to perform better when **MidiList** is sorted so that the voices are arranged end-to-end. Of course, when the voices are arranged end-to-end, the interval between the last note in each voice and the first note in the next voice will not be melodic and, in such cases, the TIEBREAKER function would not be expected to be effective. Nevertheless, in a four-voiced polyphonic work containing 1000 notes, only 3 of the intervals between consecutive notes in **MidiList** would be between notes in different voices which means that, in such a case, TIEBREAKER would be expected to operate effectively for 99.7% of the piece.

Cambouropoulos (2003, p. 427) claims that "voice leading is also an important component of pitch spelling" and proposes that "if the various melodic streams are predetermined, additional rules can cater to voice-leading effects". In particular, he suggests that the 'tie-breaker' rule used in CAM9698 could be incorporated into CAM03 and CAM01 (Cambouropoulos, 2001, p. 8, Cambouropoulos, 2003, p. 427). The effect of voice-leading on the performance of Cam-

bouropoulos's algorithms can be tested to some extent by comparing the performance of a given version of the algorithm when it is run on data in which the voices are arranged end-to-end with its performance on data in which the voices are interleaved.

3.6.2.4 Method of computing interval optimization penalties (IOPs) (Table 3.8, row 4)

In his own evaluations of his algorithms, Cambouropoulos uses essentially two different methods to calculate the interval optimization penalty (IOP) for each interval within a window. In CAM03A, CAM01A and CAM9698A, the penalty incurred by an interval depends on its modality class; whereas, in CAM03B, CAM03C, CAM01B and CAM01C, the penalty depends upon the distance along the line of fifths that corresponds to the interval. An entry of 'mod' in row 4 of Table 3.8 indicates that a version of the algorithm uses the modality class to determine the IOPs; and an entry of 'lof' indicates that the IOPs are based on the line of fifths.

3.6.2.5 If modality-based IOP calculation, how is modality class determined? (Table 3.8, row 5)

CAM03A, CAM01A and CAM9698A all use the modality class of an interval to determine its IOP. However, as described above, CAM03A and CAM01A determine the modality class of each interval by effectively looking it up in Table 3.1; whereas, in CAM9698, the modality class of each interval is determined by directly calculating the frequency with which the interval occurs in the major and minor scales. As explained in section 3.2.4, there is an error in Table 3.1 which implies that these two methods of calculating the modality class of an interval are not strictly equivalent. An entry of 'tab' in row 5 of Table 3.8 indicates that a version of the algorithm uses Table 3.1 to determine interval modality classes. An entry of 'sca' indicates that the modality classes are determined by calculating the frequency of occurrence of intervals in the tonal scales, as in CAM9698.

3.6.2.6 If modality-based IOP calculation, what penalties applied to modality classes? (Table 3.8, row 6)

CAM03A, CAM01A and CAM9698A all use the modality class of an interval to determine its IOP. However, the specific penalty applied to an interval for belonging to modality class D is different in each version of the algorithm. Each entry in row 6 of Table 3.8 indicates the specific penalty value applied to an interval for belonging to each of the four modality classes.

3.6.2.7 If line-of-fifths-based IOP calculation, are ["a1"] and ["d1"] 'demoted'? (Table 3.8, row 7)

In CAM03B, CAM03C, CAM01B and CAM01C, the calculation of the IOP for a given interval is based on the distance along the line of fifths that corresponds to the interval. However, as explained in section 3.5 above, the penalty applied to ["a1"] and ["d1"] in CAM03C and CAM01C was increased from 7 to 12 so that these intervals were effectively 'demoted'. In row 7 of Table 3.8, an entry of 'yes' means that ["a1"] and ["d1"] are demoted in this way and

the algorithm determines the IOPs using Table 3.5. An entry of 'no' means that the IOPs are determined directly from the line-of-fifths distance using Table 3.4.

3.6.2.8 Penalty values for notational parsimony (Table 3.8, row 8)

Every version of the algorithm described and tested by Cambouropoulos incorporates an implementation of his principle of 'notational parsimony'. However, the specific notational parsimony penalties applied to an interval differ in the different versions that he evaluates. In each entry in row 8 of Table 3.8, the first number is the penalty applied to an interval when just one of the two pitch name classes forming the interval is enharmonic; and the second number is the penalty applied when both the pitch name classes forming the interval are enharmonic.

3.6.2.9 Spellings constrained to being within either flatside or sharpside region (Table 3.8, row 9)

As explained above, in CAM03 and CAM01, only those spellings for a window are evaluated that fall wholly within either the flatside or sharpside regions shown in Figure 3.3. However, in CAM9698, every possible spelling for each window is considered, given that there are two possible pitch name classes for each 'black-note' pitch class and three possible pitch name classes for each 'white-note' pitch class. An entry of 'yes' in row 9 of Table 3.8 indicates that each window spelling is constrained to being within either the flatside or sharpside regions and an entry of 'no' indicates that it is not.

3.6.2.10 Retain pitch name classes for first third of window from previous window (Table 3.8, row 10)

In CAM03 and CAM01, the pitch name classes for the first $WinSize/3$ elements in each window (apart from the first) are fixed by the retained segment from the previous window (the motivation for doing this was discussed in section 3.2.2 above). However, in CAM9698 every possible spelling for each window is evaluated, not just those that begin with the last $WinSize/3$ pitch name classes from the retained segment of the previous window.

3.6.2.11 Include interval penalties for non-contiguous notes in window (Table 3.8, row 11)

In CAM03 and CAM01, the total penalty score for each window spelling is computed by adding together the individual notational parsimony and interval optimization penalties for each of the $WinSize(WinSize-1)/2$ unordered pairs of notes within each window. In other words, in these algorithms, intervals between contiguous and non-contiguous notes are taken into account. However, in CAM9698, only intervals between contiguous notes in each window are considered. Another variable feature of Cambouropoulos's algorithm is therefore whether or not to consider the intervals between non-contiguous notes in each window.

3.6.2.12 If fixed length window and voices end-to-end, is TieBreaker used? (Table 3.8, row 12)

A given version of Cambouropoulos's algorithm may or may not use the TieBreaker function discussed in section 3.4.6 above. As has already been pointed out, it only really makes any sense to use the TieBreaker function if the data is sorted so that the voices are arranged end-to-end. It also does not really make sense to use the TieBreaker function on a variable length window, since consecutive pitch name classes in a spelling for such a window do not necessarily correspond to consecutive notes within a voice in the music.

3.6.2.13 If modality-based IOP derived directly from scales, how are scales weighted? (Table 3.8, row 13)

In Cam9698, the modality class of each interval is calculated by taking a weighted average of the modalities of the interval in the major and minor scales (see section 3.4.3 above). Since the purpose of the algorithm is to determine the correct pitch names in Western tonal music, it only makes sense to consider the modalities of an interval in the major and minor scales. The modality of an interval in the major scale is the same as its modality in any other scale that is cyclically equivalent to the major scale, such as the descending melodic minor scale or the natural minor scale. One therefore only needs to consider the modality of each interval in a diatonic scale (i.e., a major, ascending melodic minor or natural minor scale), the ascending melodic minor scale and the harmonic minor scale. In Cam9698, Cambouropoulos effectively weights the diatonic scales by a factor of 6, the ascending melodic minor scale by a factor of 1 and the harmonic minor scale by a factor of 2. In row 13 of Table 3.8, the weights assigned to the diatonic scales, the ascending melodic minor scale and the harmonic minor scale (indicated by the abbreviations 'dia', 'amm' and 'hm', respectively, in the second column), are listed in that order—hence the entry '6,1,2' for Cam9698A.

Clearly, there are various other reasonable ways in which one might weight the scales. For example, one could assign an equal weighting to the ascending melodic minor scale and the harmonic minor scale and twice this weighting to the diatonic scales (i.e., 2,1,1).

3.6.2.14 If modality-based IOP derived from scales, what are values of modality class boundaries? (Table 3.8, row 14)

As explained in section 3.4.3 above, the values of the modality class boundaries are determined by a threshold x that Cambouropoulos sets to 0.25 in Cam9698A (Cambouropoulos, 1998, p. 70). Cambouropoulos (1998, p. 77, Footnote 47) claims, incorrectly, that x must be less than or equal to 0.25, whereas, in fact, it must be less than 0.50. Cambouropoulos (1998, p. 70) acknowledges, however, that "this is an arbitrary selection of a limit" and that "further research may define a better value or range of values for limit x". This suggests that it may be worth experimenting with various different values of x. The value of x used in Cam9698A is given in the appropriate column in row 14 of Table 3.8.

3.6.2.15 If variable length window used, does each window contain fixed number of MIDI note numbers or pitch classes? (Table 3.8, row 15)

Cambouropoulos (2001, p. 5) states that his implementation of CAM01 "employs a *variable* length window that always contains 9 distinct pitches". However, it is not clear from this whether he means that each window contains 9 distinct *pitch classes* or that each window contains 9 distinct *MIDI note numbers*. When I asked Cambouropoulos about this (Meredith, 2002a), he stated that he "had tried both approaches" but that he "currently use[d] MIDI note numbers" because

> this way we avoid too much repetition (which causes various problems) but it does allow a small amount of repetition (with octave-equivalent MIDI numbers) which implicitly has the important effect of taking into account relative prominence of notes (i.e., repeating notes in the window force a solution that gives them a more stable spelling contributing less to the overall penalty value).
>
> (Cambouropoulos, 2002)

More recently, Cambouropoulos (2004) informed me that, when he used a window containing 9 distinct pitch classes instead of 9 distinct MIDI note numbers, the algorithm was faster but his "impression was that it gave worse results". Again, he attempts to explain this as follows:

> The reason is that allowing a certain amount of pitch class repetition (in terms of MIDI pitch numbers e.g. 48, 60, 72) indirectly incorporates pitch prominence into the algorithm—i.e., notes that repeat more will contribute more to the penalty value and hence a more 'stable' spelling for them will be preferred.
>
> (Cambouropoulos, 2004)

In other words, in Cambouropoulos's opinion, using a variable length window containing a fixed number of distinct MIDI note numbers resolves the problems that arise from excessive pitch repetition without totally ignoring the perceptual prominence that a particular pitch class achieves through this repetition (Krumhansl, 1990, pp. 66–74).

Nevertheless, it seems feasible that a version of the algorithm that uses variable length windows containing a fixed number of pitch classes (indicated by 'pc' in row 15 of Table 3.8) may have certain advantages (e.g., speed) over one that uses variable length windows containing the same number of MIDI note numbers (indicated by 'mid' in row 15 of Table 3.8).

3.6.2.16 For each MIDI note number or pitch class within single window, pitch name class is constant (Table 3.8, row 16)

As discussed in section 3.5.3 above, Cambouropoulos (1998, pp. 79–80) suggests that certain errors made by CAM9698A could be avoided by not allowing different pitch name classes to be assigned to different occurrences of the same MIDI note number within a single window. Similarly, one could incorporate a rule that disallowed different pitch name classes from being assigned to different occurrences of the same *pitch class* within a single window. An entry of 'mid' in row 16 of Table 3.8 indicates that no two occurrences of the same MIDI note number within a single window are permitted to have different pitch name classes. An entry of 'pc' in

this row indicates that all occurrences of a single pitch class within any given window must be assigned the same pitch name class. An entry of 'no' in this row indicates that there are no constraints on the number of different but enharmonically equivalent pitch name classes that may be assigned within any given window.

3.6.2.17 Uses "flat/natural-or-sharp/natural-first" heuristic (Table 3.8, row 17)

Cambouropoulos (2003, p. 427) suggests that the running time of CAM03 could be improved by employing heuristics to limit the number of alternative spellings that have to be evaluated for each window. In particular, he suggests that

> for a given window, the pitch-spelling process can start from the two simplest options: sharp and natural spellings, and flat and natural spellings–if the penalty value is zero for one of these, then select this spelling and stop further search for this window.

In row 17 of Table 3.8, each entry indicates whether or not a particular version of the algorithm employs this 'flat/natural-or-sharp/natural-first" heuristic.

3.6.2.18 Includes method for analysing metric structure (Table 3.8, row 18)

Cambouropoulos (2001, p. 8; 2003, p. 427) suggests that

> intervals between notes that appear on metrically stronger positions... should contribute more to the overall spelling penalty value.

Implementing this suggestion would involve carrying out the non-trivial task of incorporating into Cambouropoulos's algorithm a system for analysing metric structure that can determine the metric "strength" of the onset of each note. Cambouropoulos has not implemented such a system in any of the published versions of his algorithm, as indicated in row 18 of Table 3.8.

3.6.2.19 Includes method for identifying accented notes (Table 3.8, row 19)

Cambouropoulos (2001, p. 8; 2003, p. 427) suggests that

> intervals between notes that... are more accented (e.g. longer duration, extreme pitch register, etc.) should contribute more to the overall spelling penalty value.

In order to implement this suggestion, it would be necessary for the duration of each note to be encoded in the input data—that is, a list of durations would have to be provided to the algorithm in addition to the list of MIDI note numbers, **MidiList**. This could make the algorithm less robust to expressive temporal deviation and pedalling effects in MIDI data that has been generated from a performance. For example, the use of the sustain pedal in a keyboard performance could result in the performed duration of a note being many times longer than its notated duration, which would lead to an incorrect weighting for the note. Cambouropoulos has not implemented such a system in any of the published versions of his algorithm, as indicated in row 19 of Table 3.8.

3.6.2.20 Weight each interval by durations of notes involved in it (Table 3.8, row 20)

Instead of implementing a full-blown system for detecting accented notes or metrical structure, one could simply incorporate into the algorithm a mechanism that weights the penalty assigned to each interval within a window spelling by some combination of the durations of the notes involved in the interval. In a number of successful computational models of metrical structure perception, it has been assumed that longer notes begin on stronger beats (see Lee (1991, pp. 63–67) for an overview and critique of this idea).

Krumhansl (1990, p. 66–73) showed that there are strong correlations between the relative frequencies of occurrence of tones and perceived tonal hierarchies. However, her analysis of results obtained by Hughes (1977) suggests that even stronger correlations exist between perceived tonal hierarchies and tone distributions in which each occurrence of a tone is weighted by its duration.

These findings suggest that it may be possible to take into account (albeit indirectly) the tonal stability of each tone and the metric strength of its onset location by weighting the penalty applied to each interval by some combination of the durations of the notes forming the interval. However, as pointed out in section 3.6.2.19 above, this would involve providing the algorithm with a list of durations as well as a list of MIDI note numbers. It could also make the algorithm less robust to expressive temporal deviation and pedalling effects in MIDI data that has been generated from a performance.

3.6.2.21 Weight each note by distance in pitch from note to middle pitch range (Table 3.8, row 21)

As mentioned in section 3.6.2.19 above, Cambouropoulos (2001, p. 8; 2003, p. 427) suggests that, other things being equal, the more "extreme" the pitch of a note, the more accented it will be perceived to be. This implies the possibility of incorporating a mechanism into his pitch spelling algorithm that weights the penalty applied to each interval, i, by the absolute size of the pitch interval between the notes forming i and the middle of the pitch range (e.g., `"Cn4"`). Cambouropoulos has not implemented such a system in any of the published versions of his algorithm, as indicated in row 21 of Table 3.8.

3.6.2.22 Includes method for analysing hierarchical structure (Table 3.8, row 22)

Cambouropoulos (2003, p. 427) suggests that

> structural relationships between notes may contribute to establishing a more refined hierarchic organization of pitches and intervals that in turn can improve the spelling method.

Presumably, what he means by this is that one might expect the algorithm to perform better if it took into account the perceived hierarchical structural relationships between notes such as those that are identified in a Schenkerian analysis (Forte and Gilbert, 1982; Schenker, 1979) or Lerdahl and Jackendoff's (1983, p. 105–277) 'time-span' and 'prolongational' reductions. Incorporating a mechanism into Cambouropoulos's algorithm for identifying such relationships would be a

very substantial undertaking that would considerably increase the complexity of the algorithm. Cambouropoulos has not implemented such a system in any of the published versions of his algorithm, as indicated in row 22 of Table 3.8.

3.6.2.23 Includes method for detecting secondary ornamental notes (Table 3.8, row 23)

Cambouropoulos (2003, p. 427) proposes that

> notes such as secondary ornamental notes (e.g., passing and neighbor notes) should affect less the tonal core of a given musical section.

In order to explore this hypothesis fully, one would need to incorporate into the algorithm a fairly complex mechanism for identifying what Cambouropoulos calls "secondary ornamental notes". Cambouropoulos has not implemented such a system in any of the published versions of his algorithm, as indicated in row 23 of Table 3.8. However, it might be possible to produce a crude approximation to such a modification simply by weighting each note by its duration (as described in section 3.6.2.20 above) on the assumption that "secondary ornamental notes" tend to be relatively short. As discussed above, this would involve providing the algorithm with a list of durations as well as a list of MIDI note numbers and it could also make the algorithm less robust to expressive temporal deviation and pedalling effects in MIDI data that has been generated from a performance.

3.6.2.24 Uses timing information for weighting interval penalties (Table 3.8, row 24)

Cambouropoulos (2003, p. 427) proposes that

> timing information can be used to calculate the distance of pitches from the center of a window and then allow pitches that are closer together to have a stronger effect on the spelling process, that is, pitches that are further away should contribute less to the overall spelling penalty.

I interpret this to mean that one could weight the penalty assigned to each interval in a window spelling by a factor that decreases as the inter-onset interval between the two notes forming the interval increases. This would be in accord with the intuition that the way in which a note is spelt depends more on the notes in its immediate temporal vicinity than it does on notes that are further away in time. This would be relatively straightforward to implement but would require providing the algorithm with the onset time of each note. Unlike weighting interval penalties according to note duration (as discussed in section 3.6.2.20), weighting by inter-onset interval would probably be fairly robust to expressive temporal deviation and pedalling effects in MIDI data generated from a performance. Cambouropoulos has not implemented such a system in any of the published versions of his algorithm, as indicated in row 24 of Table 3.8.

3.6.2.25 If using flatside/sharpside regions, use first third of window to eliminate one side (Table 3.8, row 25)

In CAM03 and CAM01, every window spelling is constrained to being entirely contained within either the flatside region or sharpside region in Figure 3.3 (see row 9 in Table 3.8). Also, in both of these versions of the algorithm, the pitch name classes for the first third of each window are retained from the previous window (see row 10 in Table 3.8). If the first third of a window in one of these versions contains a pitch name class, *PNC*, that only occurs in one of the two pitch name class regions, there is no point in evaluating any of the spellings for the last two thirds of the window that are in the pitch name class region that does not contain *PNC*. In this way, it might be possible to improve the running time of the algorithm. Although this would be a fairly straightforward modification to make, Cambouropoulos does not appear to have implemented it in any of the versions of the algorithm that he has tested, as indicated in row 25 of Table 3.8.

3.6.2.26 Rejects window spellings that contain more than one enharmonic pitch name class (Table 3.8, row 26)

Cambouropoulos (2001, p. 5) suggests that

> the number of sequences [i.e., the number of spellings to be evaluated for each window] can be reduced by... rejecting sequences that contain more than one enharmonic spelling (e.g., not more than one double-sharp in a sequence).

However, this would seem to be a rather illogical thing to do, since an enharmonic pitch name class is quite commonly preceded or followed immediately by another enharmonic pitch class. For example, in a correctly spelt ascending G♯ major scale, `"Es"` is immediately followed by `"Fss"`. When I questioned Cambouropoulos about this suggestion, he admitted that such a rule would be "wrong in general" (Cambouropoulos, 2004). This rule does not seem to have been implemented in any of the versions of the algorithm tested by Cambouropoulos, as indicated in row 26 of Table 3.8.

3.6.2.27 Rejects window spellings that contain consecutive enharmonic pitch name classes (Table 3.8, row 27)

Cambouropoulos (1996, p. 244; 1998, p. 80) suggests that the running time of CAM9698 could be improved by "disallowing altogether... two successive enharmonic notes" in a single window spelling. As already discussed in section 3.6.2.26, this would seem to be an illogical thing to do since an enharmonic pitch name class is quite commonly preceded or followed immediately by another enharmonic pitch class (e.g., in an ascending G♯ major scale). This rule does not seem to have been implemented in any of the versions of the algorithm tested by Cambouropoulos, as indicated in row 27 of Table 3.8.

3.6.2.28 If modality-based IOP calculation, uses *PINC*s instead of $\langle nci, pci \rangle$ pairs (Table 3.8, row 28)

In section 3.4.2 above, it was explained that, when the modality of an interval is calculated using only its name class interval and pitch class interval, as suggested by Cambouropoulos

(1996, 1998), certain exotic non-scale intervals are assigned non-zero modalities. This could be remedied by calculating the frequencies of occurrence of intervals within scales using my pitch interval name class representation instead of Cambouropolous's *GPIR*.

3.6.2.29 Rejects spellings that contain modality class D intervals (except ["a1"] or ["d1"]) (Table 3.8, row 29)

Cambouropoulos (1996, p. 244; 1998, p. 80) suggests that the running time of CAM9698 could be improved by "disallowing altogether... all class D intervals with the exception of chromatic semitones". However, such a rule would not significantly improve the running time, since one would not be able to reject a window spelling containing disallowed class D intervals before actually generating the spelling and computing all the intervals in the spelling up to the one that is disallowed. In the worst case, the offending interval might be the last one to be calculated, in which case the running time would not be improved. In other words, such a modification would not change the worst case running time of the algorithm.

3.6.3 The versions of the algorithm tested here

It was impossible within the scope of this project to carry out an exhaustive evaluation of all 29 variable features identified in section 3.6.2 above. So, instead, the focus in my own evaluation was on exploring those variable features that required only relatively small modifications to be made to the three 'basic' versions of the algorithm described in Cambouropoulos's publications (i.e., CAM03, CAM01 and CAM9698). The main goal of this exploration was to identify the combination of values for these variable features that achieves the best spelling accuracy in a reasonable time.

Tables 3.9 and 3.10 show the 18 variable features investigated together with the values for these features for each of the 26 versions of the algorithm tested here. The first entry in each row in these tables gives the number of the row in Table 3.8 that corresponds to that particular feature. Thus the features investigated in this evaluation are those in rows 1–17 and 25 of Table 3.8. For every other variable feature in Table 3.8, the value is 'no' for all versions of the algorithm tested here.

Ideally, to identify the combination of values for these 18 variable features that leads to the best spelling accuracy, one would need to run several hundred thousand different versions of the algorithm on the test corpus. Using my implementations, this would have been impractical since, over the 26 versions of the algorithm tested in this evaluation, the average time taken for one algorithm to process the test corpus was about 29 hours.

In this evaluation, I therefore adopted a much quicker but less thorough strategy for estimating the best combination of variable features. For each variable feature, I estimated the individual effect on spelling accuracy of changing the value of that feature by comparing the spelling accuracies of two versions of the algorithm that differed only with respect to that particular feature. For most of the features studied, CAM03A was used as a standard against which the other versions could be compared. For example, an estimate of the individual effect on spelling accuracy of increasing window size can be gained by comparing the performance of CAM03D with that of CAM03A (see Table 3.9), since the only difference between CAM03D and

CAM03A is that *WinSize* is set to 12 in CAM03D whereas it is set to 9 in CAM03A. Similarly, the individual effect on spelling accuracy of changing from a fixed length window to a variable length one can be measured by comparing the performance of CAM01D, which uses a variable length window, with that of CAM03A which is as similar to it as possible given that it uses a fixed length window. For certain features, however, CAM03A could not be used as a standard. For example, to measure the effect of changing from a variable length window containing a fixed number of MIDI note numbers to one containing a fixed number of pitch classes, the performance of CAM01D (Table 3.9) was compared with that of CAM01E (see Table 3.10). The versions of the algorithm tested here were therefore selected so that, for each variable feature, there were two algorithms that differed only with respect to that feature. In addition, the versions of the algorithm tested by Cambouropoulos himself were included.

This strategy certainly provides useful information about the individual effect on spelling accuracy of a change in each variable feature. However, if two algorithms differ with respect to two or more variable features, the combined effect of these differences on spelling accuracy may not be easily predictable from the individual effects of each difference. For example, the only difference between CAM03A and CAM03E is that the voices are interleaved in CAM03A whereas they are arranged end-to-end in CAM03E. Changing from CAM03A to CAM03E increased the note error rate by 12.61%. Similarly, CAM03K and CAM03L also only differ with respect to the way the notes are ordered in the input, with the voices being interleaved in CAM03K and end-to-end in CAM03L. However, changing from CAM03K to CAM03L, *reduced* the note error rate by 0.29%. Fortunately, for most of the variable features examined here, there is more than one pair of tested versions of the algorithm that differ only with respect to that feature. This allows for the inter-dependence of the features to be investigated to some extent.

Each version of the algorithm evaluated here will now be described. I shall start with the versions in Table 3.9 and then describe the versions in Table 3.10.

CAM03A is simply CAM03 (Figure 3.2) run with the *WinSize* parameter set to 9.

In order to test the effect of increasing the window size, a version of CAM03 was run with a window size of 12. This version was denoted CAM03D.

CAM01D is effectively the same as CAM03A, except that it uses variable length windows that each contain 9 distinct MIDI note numbers. CAM01D is CAM01 (Figure 3.12) run with *WinSize* set to 9 and with all calls to COMPUTESPELLPEN01 in SPELLWIN01 (Figure 3.13) replaced with corresponding calls to COMPUTESPELLPEN03 (Figure 3.6).

CAM03E is identical to CAM03A, except that it is run on data in which the voices are arranged end-to-end rather than interleaved. Note that it does not incorporate the TIEBREAKER function.

CAM03B is one of the versions evaluated by Cambouropoulos himself (see section 3.5.1 above). It is the same as CAM03A, except that the interval optimization penalty for each interval is equal to the distance along the line of fifths to which the interval corresponds, as given in Table 3.4. Also, the notational parsimony penalty for an interval is set to 13 for each note in the interval that is enharmonic. CAM03B is the same as CAM03A, except that every call to COMPUTESPELLPEN03 in SPELLWIN03 (Figure 3.4) is replaced with a corresponding call to COMPUTESPELLPEN03B, defined in Figure 3.31. Note that, as specified by Cambouropoulos

			CAM03A	CAM03D	CAM01D	CAM03E	CAM03B	CAM03F	CAM03G	CAM03C	CAM03H	CAM03I	CAM03J	CAM03K	CAM03L
1	Window size	$3n$	9	12	9	9	9	9	9	9	9	9	9	9	9
2	Fixed or variable length window	(fix,var)	fix	fix	var	fix	fix	fix	fix	fix	fix	fix	fix	fix	fix
3	How the notes are sorted in **MidiList** (voice-leading)	(int,end)	int	int	int	end	int	int	int	int	int	int	int	int	end
4	Method of computing interval optimisation penalties (IOPs)	(mod,lof)	mod	mod	mod	mod	lof	mod	mod	lof	mod	mod	mod	mod	mod
5	If modality-based IOP calculation, how is modality class determined?	(tab,sca)	tab	tab	tab	tab		sca	tab		tab	tab	tab	tab	tab
6	If modality-based IOP calculation, what penalties applied to modality classes?	A,B,C,D	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2		0,0,1,2	0,0,1,4		0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2
7	If line-of-fifths-based IOP calculation, are ["a1"] and ["d1"] 'demoted'?	(yes,no)					no			yes					
8	Penalty values for notational parsimony	one,both	2,4	2,4	2,4	2,4	13,26	2,4	2,4	13,26	4,8	2,4	2,4	2,4	2,4
9	Spellings constrained to being within either flatside or sharpside region	(yes,no)	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes	yes
10	Retain pitch name classes for first third of window from previous window	(yes,no)	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
11	Include interval penalties for non-contiguous notes in window	(yes,no)	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	no
12	If fixed-length window and voices end-to-end, is TIEBREAKER used?	(yes,no)				no									no
13	If modality-based IOP derived directly from scales, how are scales weighted?	dia,amm,mh						6,1,2							
14	If modality-based IOP derived from scales, what are values of modality class boundaries?	$0 < x < 0.5$						0.25							
15	If variable-length window used, does each window contain fixed number of MIDI note numbers or pitch classes?	(mid,pc)			mid										
16	For each MIDI note number or pitch class within single window, pitch name class is constant	(mid,pc,no)	no	no	mid	no	no	no	no	no	no	no	no	no	no
17	Uses "flat/natural-or-sharp/natural-first" heuristic	(yes,no)	no	no	no	no	no	no	no	no	no	no	no	no	no
25	If spellings constrained to be within flatside or sharpside, use first third of window to eliminate one side	(yes,no)	no	no	no	no	no	no	no	no	no	no	no	no	no

Table 3.9: The values of the variable features for each version of Cambouropoulos's algorithm tested here. All values different from those for CAM03A are highlighted. See text for explanation.

```
ComputeSpellPen03B(Spelling)
1   PINCs ← ⟨["p1"], ["rp5"], ["rp4"], ["rma2"], ["rmi7"],
             ["rma6"], ["rmi3"], ["rma3"], ["rmi6"], ["rma7"],
             ["rmi2"], ["ra4"], ["rd5"], ["a1"], ["d1"],
             ["ra5"], ["rd4"], ["ra2"], ["rd7"], ["ra6"],
             ["rd3"], ["ra3"], ["rd6"], ["ra7"], ["rd2"]⟩
2   LOFPenalties ← ⟨0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12⟩
3   n ← |Spelling|
4   SpellingPenalty ← 0
5   for i ← 0 to n − 2
6      for j ← i + 1 to n − 1
7         if IsEnharmonicSpelling(Spelling[i])
8            SpellingPenalty ← SpellingPenalty + 13
9         if IsEnharmonicSpelling(Spelling[j])
10           SpellingPenalty ← SpellingPenalty + 13
11        PINC ← PNC2PINC(Spelling[i], Spelling[j])
12        p ← Pos(PINC, PINCs)
13        if p
14           IOP ← LOFPenalties[p]
15        else
16           IOP ← 13
17        SpellingPenalty ← SpellingPenalty + IOP
18  return SpellingPenalty
```

Figure 3.31: The ComputeSpellPen03B algorithm.

(2003, p. 423), the interval optimization penalties in ComputeSpellPen03B are not computed directly but looked up in Table 3.4, which is implemented in the two lists, **PINCs** and **LOFPenalties**, defined in lines 1 and 2, respectively. One problem with this strategy is that, if an interval is encountered that does not occur in Table 3.4, no penalty is specified. I have therefore added the rule (implemented in lines 12–16) that an interval that does not occur in Table 3.4 incurs a penalty of 13 (i.e., one more than the largest penalty incurred by an interval that *does* occur in Table 3.4).

Cam03F is the same as Cam03A, except that the modality class of each interval is computed directly in the manner of Cam9698 (Figure 3.15) rather than looked up in Table 3.1. Note that the weightings assigned to each scale type and the values of the modality class boundaries in Cam03F are identical to those in Cam9698. The Cam03F algorithm is given in Figure 3.32. Lines 1–11 of this algorithm are identical to lines 1–11 in Cam9698. Lines 12–26 in Cam03F are the same as lines 6–20 of Cam03 (Figure 3.2), except that the call to SpellWin03 in line 16–17 of Cam03 is replaced with a call to SpellWin03F in line 22–23 of Cam03F. The function SpellWin03F is defined in Figure 3.33. SpellWin03F is the same as SpellWin03 (Figure 3.4), except that

1. SpellWin03F takes an extra argument, **BlendedModTable**, which is the blended modality table computed in line 11 of Cam03F; and

2. the calls to ComputeSpellPen03 in lines 7 and 12 of SpellWin03 are replaced with calls to ComputeSpellPen03F in the same lines of SpellWin03F.

ComputeSpellPen03F is defined in Figure 3.34. ComputeSpellPen03F computes the notational parsimony penalty in the same way as ComputeSpellPen03 (Figure 3.6), therefore

```
CAM03F(MidiList)
1   MidiListSize ← |MidiList|
2   LastWindow ← false
3   FirstWindow ← true
4   WinStart ← 0
5   PNCList ← ⟨ ⟩
6   MajorScale ← ⟨2, 2, 1, 2, 2, 2, 1⟩
7   NatMinScale ← ⟨2, 1, 2, 2, 1, 2, 2⟩
8   DescMelMinScale ← ⟨2, 1, 2, 2, 1, 2, 2⟩
9   AscMelMinScale ← ⟨2, 1, 2, 2, 2, 2, 1⟩
10  HarmMinScale ← ⟨2, 1, 2, 2, 1, 3, 1⟩
11  BlendedModTable ← COMPUTEBLENDEDMODTABLE(0.25,
                                              ⟨⟨MajorScale, 4⟩,
                                               ⟨NatMinScale, 1⟩,
                                               ⟨DescMelMinScale, 1⟩,
                                               ⟨AscMelMinScale, 1⟩,
                                               ⟨HarmMinScale, 2⟩⟩)
12  while |PNCList| < MidiListSize
13      WinEnd ← MIN({WinStart + 9, MidiListSize})
14      if WinEnd = MidiListSize
15          LastWindow ← true
16      if FirstWindow
17          WinPNCList ← ⟨ ⟩
18          WinMidiList ← MidiList[WinStart, WinEnd]
19      else
20          WinPNCList ← PNCList[WinStart, WinStart + 3]
21          WinMidiList ← MidiList[WinStart + 3, WinEnd]
22      PNCList ← PNCList ⊕ SPELLWIN03F(WinPNCList, WinMidiList,
23                                      FirstWindow, LastWindow, BlendedModTable)
24      FirstWindow ← false
25      WinStart ← WinStart + 3
26  return PNCList
```

Figure 3.32: The CAM03F algorithm.

```
SpellWin03F(WinPNCList, WinMidiList, FirstWindow, LastWindow, BlendedModTable)
1   WinMidiListSize ← |WinMidiList|
2   MaxBitVecInt ← 2^WinMidiListSize − 1
3   BestSpelling ← ⟨ ⟩
4   for i ← 0 to MaxBitVecInt
5      BitVec ← BitVector(i, WinMidiListSize)
6      Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
7      SpellingPenalty ← ComputeSpellPen03F(WinPNCList ⊕ Spelling, BlendedModTable)
8      if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
9         BestSpelling ← Spelling
10        BestPenalty ← SpellingPenalty
11     Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
12     SpellingPenalty ← ComputeSpellPen03F(WinPNCList ⊕ Spelling, BlendedModTable)
13     if SpellingPenalty < BestPenalty
14        BestSpelling ← Spelling
15        BestPenalty ← SpellingPenalty
16  ▶ Now return appropriate part of BestSpelling.
17  BSEnd ← WinMidiListSize/2
18  if FirstWindow
19     BSEnd ← 2 WinMidiListSize/3
20  if LastWindow
21     BSEnd ← WinMidiListSize
22  return BestSpelling[0, BSEnd]
```

Figure 3.33: The SpellWin03F algorithm.

```
ComputeSpellPen03F(Spelling, BlendedModTable)
1   n ← |Spelling|
2   SpellingPenalty ← 0
3   for i ← 0 to n − 2
4      for j ← i + 1 to n − 1
5         if IsEnharmonicSpelling(Spelling[i])
6            SpellingPenalty ← SpellingPenalty + 2
7         if IsEnharmonicSpelling(Spelling[j])
8            SpellingPenalty ← SpellingPenalty + 2
9         PINC ← PNC2PINC(Spelling[i], Spelling[j])
10        ModClass ← ComputeModClass(PINC, BlendedModTable)
11        if ModClass = C
12           SpellingPenalty ← SpellingPenalty + 1
13        if ModClass = D
14           SpellingPenalty ← SpellingPenalty + 2
15  return SpellingPenalty
```

Figure 3.34: The ComputeSpellPen03F algorithm.

lines 1–9 in these two functions are the same. However, ComputeSpellPen03F computes IOPs in essentially the same way as ComputeSpellPen9698 (Figure 3.23), except that, in ComputeSpellPen03F, an interval in modality class D incurs a penalty of 2 instead of 4. Therefore, lines 10–15 of ComputeSpellPen03F are the same as lines 10–15 of ComputeSpellPen9698, except that, in line 14 of ComputeSpellPen03F, *SpellingPenalty* is increased by 2 instead of 4.

Cam03G is identical to Cam03A, except that the interval optimization penalty for intervals in modality class D is raised from 2 to 4. Cam03G is therefore the same as Cam03 (Figure 3.2) run with a window size of 9, except that line 14 of ComputeSpellPen03 (Figure 3.6) is replaced by the line

14 $SpellingPenalty \leftarrow SpellingPenalty + 4$

Cam03C is another of the versions evaluated by Cambouropoulos himself (see section 3.5.1 above). It is identical to Cam03B, except that the interval optimization penalty assigned to ["a1"] and ["d1"] is increased from 7 to 12. In other words, Cam03C is the same as Cam03B, except that line 1 in ComputeSpellPen03B (Figure 3.31) is replaced with

1 **PINCs** $\leftarrow$ ⟨["p1"], ["rp5"], ["rp4"], ["rma2"], ["rmi7"],
["rma6"], ["rmi3"], ["rma3"], ["rmi6"], ["rma7"],
["rmi2"], ["ra4"], ["rd5"], ["ra5"], ["rd4"],
["ra2"], ["rd7"], ["ra6"], ["rd3"], ["ra3"],
["rd6"], ["ra7"], ["rd2"], ["a1"], ["d1"]⟩

Cam03H is identical to Cam03A, except that the notational parsimony penalties are doubled. In other words, enharmonic spellings are avoided more strongly in Cam03H than in Cam03A. Cam03H is the same as Cam03 (Figure 3.2) run with *WinSize* equal to 9 and with lines 5 to 8 of ComputeSpellPen03 (Figure 3.6) replaced with

5 **if** IsEnharmonicSpelling(**Spelling**[i])
6 $SpellingPenalty \leftarrow SpellingPenalty + 4$
7 **if** IsEnharmonicSpelling(**Spelling**[j])
8 $SpellingPenalty \leftarrow SpellingPenalty + 4$

Cam03I is the same as Cam03A, except that the spelling for each window is not restricted to being completely contained within either the flatside region or the sharpside region. This means that SpellWin03 (Figure 3.4) in Cam03 (Figure 3.2) has to be replaced in Cam03I by the function SpellWin03I which is defined in Figure 3.35. SpellWin03I is based on SpellWin9698 (Figure 3.22), however there are the following differences between the two functions.

1. In Cam03I, the pitch name classes for the first third of each window are retained from the previous window, therefore SpellWin03I takes the argument **WinPNCList** containing these pitch name classes whereas SpellWin9698 does not.

```
SpellWin03I(WinPNCList, WinMidiList, FirstWindow, LastWindow)
1   PNCs ← ⟨⟨"Bs", "Cn", "Dff"⟩, ⟨"Cs", "Df"⟩, ⟨"Css", "Dn", "Eff"⟩, ⟨"Ds", "Ef"⟩,
           ⟨"Dss", "En", "Ff"⟩, ⟨"Es", "Fn", "Gff"⟩, ⟨"Fs", "Gf"⟩, ⟨"Fss", "Gn", "Aff"⟩,
           ⟨"Gs", "Af"⟩, ⟨"Gss", "An", "Bff"⟩, ⟨"As", "Bf"⟩, ⟨"Ass", "Bn", "Cf"⟩⟩
2   n ← |WinMidiList|
3   PCList ← ⨁_{i=0}^{n−1} ⟨WinMidiList[i] mod 12⟩
4   PNCIndexList ← ⨁_{i=0}^{n−1} ⟨0⟩
5   BestSpelling ← ⨁_{i=0}^{n−1} ⟨PNCs[PCList[i]][0]⟩
6   BestPenalty ← ComputeSpellPen03(WinPNCList ⊕ BestSpelling)
7   NumberOfSpellings ← ∏_{i=0}^{n−1} |PNCs[PCList[i]]|
8   for s ← 1 to NumberOfSpellings − 1
9     ▶ Increment PNCIndexList.
10    i ← 0
11    PNCIndexList[i] ← (PNCIndexList[i] + 1) mod |PNCs[PCList[i]]|
12    while (PNCIndexList[i] = 0) ∧ (i < n − 1)
13      i ← i + 1
14      PNCIndexList[i] ← (PNCIndexList[i] + 1) mod |PNCs[PCList[i]]|

15    ▶ Find new Spelling and SpellingPenalty.
16    Spelling ← ⨁_{i=0}^{n−1} ⟨PNCs[PCList[i]][PNCIndexList[i]]⟩
17    SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)

18    ▶ Decide whether BestSpelling should become equal to Spelling.
19    if (SpellingPenalty < BestPenalty)
20      BestSpelling ← Spelling
21      BestPenalty ← SpellingPenalty

22  ▶ Now return appropriate part of BestSpelling.
23  BSEnd ← n/2
24  if FirstWindow
25    BSEnd ← 2n/3
26  if LastWindow
27    BSEnd ← n
28  return BestSpelling[0, BSEnd]
```

Figure 3.35: The SpellWin03I algorithm.

```
CAM03J(MidiList)
1    MidiListSize ← |MidiList|
2    LastWindow ← false
3    FirstWindow ← true
4    WinStart ← 0
5    PNCList ← ⟨ ⟩
6    while |PNCList| < MidiListSize
7        WinEnd ← MIN({ WinStart + 9, MidiListSize})
8        if WinEnd = MidiListSize
9            LastWindow ← true
10       WinMidiList ← MidiList[ WinStart, WinEnd ]
11       PNCList ← PNCList ⊕ SPELLWIN03J(WinMidiList, FirstWindow,
12                                                    LastWindow)
13       FirstWindow ← false
14       WinStart ← WinStart + 3
15   return PNCList
```

Figure 3.36: The CAM03J algorithm.

2. In CAM03I, the penalty score for each window spelling is computed using COMPUTESPELLPEN03 (Figure 3.6), which does not use the blended modality table computed in line 11 of CAM9698 (Figure 3.15). Therefore SPELLWIN03I does not require the **BlendedModTable** argument of SPELLWIN9698.

3. The calls to COMPUTESPELLPEN9698 in SPELLWIN9698 are replaced in SPELLWIN03I with corresponding calls to COMPUTESPELLPEN03 (Figure 3.6).

4. There are no lines in SPELLWIN03I that correspond to lines 22–25 in SPELLWIN9698 because the TIEBREAKER function is not used in CAM03I.

5. Lines 23–28 of SPELLWIN03I are the same as lines 17–22 of SPELLWIN03 (Figure 3.4), except that the variable *WinMidiListSize* in SPELLWIN03 is renamed n in SPELLWIN03I.

CAM03J is the same as CAM03A, except that, for each window, the pitch name classes for the first third of the window are not fixed to be the same as in the retained segment for the previous window. CAM03J is defined in Figure 3.36. Lines 1–9 in CAM03J are the same as lines 1–9 in CAM03 (Figure 3.2). Lines 10–15 in CAM03J are the same as lines 16–21 in CAM9698 (Figure 3.15), except that the call to SPELLWIN9698 in line 17 of CAM9698 is replaced in line 11 of CAM03J with a corresponding call to the function SPELLWIN03J, which is defined in Figure 3.37. SPELLWIN03J is based on SPELLWIN03 (Figure 3.4), however SPELLWIN03J does not take the argument **WinPNCList**. Therefore, each time COMPUTESPELLPEN03 is called in SPELLWIN03J, its argument is not prepended by **WinPNCList** as it is in SPELLWIN03. Also, because **WinMidiList** spans the whole window in SPELLWIN03J but only the latter two-thirds of the window in SPELLWIN03, lines 17–23 in SPELLWIN03J are an appropriately modified version of lines 17–22 in SPELLWIN03.

CAM03K is the same as CAM03A, except that the total penalty score for each window spelling is computed by summing the individual IOPs and notational parsimony penalties only for intervals between consecutive notes in the window. This means that the COMPUTESPELLPEN03

```
SpellWin03J(WinMidiList, FirstWindow, LastWindow)
1   WinMidiListSize ← |WinMidiList|
2   MaxBitVecInt ← 2^WinMidiListSize − 1
3   BestSpelling ← ⟨⟩
4   for i ← 0 to MaxBitVecInt
5       BitVec ← BitVector(i, WinMidiListSize)
6       Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
7       SpellingPenalty ← ComputeSpellPen03(Spelling)
8       if (BestSpelling = ⟨⟩) ∨ (SpellingPenalty < BestPenalty)
9           BestSpelling ← Spelling
10          BestPenalty ← SpellingPenalty
11      Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
12      SpellingPenalty ← ComputeSpellPen03(Spelling)
13      if SpellingPenalty < BestPenalty
14          BestSpelling ← Spelling
15          BestPenalty ← SpellingPenalty
16  ▶ Now return appropriate part of BestSpelling.
17  BSStart ← 3
18  BSEnd ← 2 × BSStart
19  if FirstWindow
20      BSStart ← 0
21  if LastWindow
22      BSEnd ← WinMidiListSize
23  return BestSpelling[BSStart, BSEnd]
```

Figure 3.37: The SpellWin03J algorithm.

```
ComputeSpellPen03K(Spelling)
1   n ← |Spelling|
2   SpellingPenalty ← 0
3   for i ← 0 to n − 2
4       if IsEnharmonicSpelling(Spelling[i])
5           SpellingPenalty ← SpellingPenalty + 2
6       if IsEnharmonicSpelling(Spelling[i + 1])
7           SpellingPenalty ← SpellingPenalty + 2
8       PINC ← PNC2PINC(Spelling[i], Spelling[i + 1])
9       if not IsClassAOrBPINC(PINC)
10          if IsClassCPINC(PINC)
11              SpellingPenalty ← SpellingPenalty + 1
12          else
13              SpellingPenalty ← SpellingPenalty + 2
14  return SpellingPenalty
```

Figure 3.38: The ComputeSpellPen03K algorithm.

```
SpellWin03M(WinPNCList, WinMidiList, FirstWindow, LastWindow)
1    WinMidiListSize ← |WinMidiList|
2    MaxBitVecInt ← 2^WinMidiListSize − 1
3    BestSpelling ← ⟨ ⟩
4    for i ← 0 to MaxBitVecInt
5       BitVec ← BitVector(i, WinMidiListSize)
6       Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
7       SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
8       if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
9          BestSpelling ← Spelling
10         BestPenalty ← SpellingPenalty
11      else
12         if (SpellingPenalty = BestPenalty) ∧ (TieBreakerM(Spelling, BestSpelling))
13            BestSpelling ← Spelling
14            BestPenalty ← SpellingPenalty
15      Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
16      SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
17      if SpellingPenalty < BestPenalty
18         BestSpelling ← Spelling
19         BestPenalty ← SpellingPenalty
20      else
21         if (SpellingPenalty = BestPenalty) ∧ (TieBreakerM(Spelling, BestSpelling))
22            BestSpelling ← Spelling
23            BestPenalty ← SpellingPenalty
24   ▶ Now return appropriate part of BestSpelling.
25   BSEnd ← WinMidiListSize/2
26   if FirstWindow
27      BSEnd ← 2WinMidiListSize/3
28   if LastWindow
29      BSEnd ← WinMidiListSize
30   return BestSpelling[0, BSEnd]
```

Figure 3.39: The SpellWin03M algorithm.

function in Cam03 (Figure 3.2) has to be replaced in Cam03K by the function ComputeSpellPen03K, defined in Figure 3.38. ComputeSpellPen03K is formed from ComputeSpellPen03 (Figure 3.6) by deleting line 4 in ComputeSpellPen03 and changing all occurrences of j to $i + 1$.

Cam03L is identical to Cam03K, except that it is run on data in which the voices are arranged end-to-end instead of interleaved.

Cam03M is the same as Cam03E, except that Cam03M uses the TieBreakerM function which is a version of the TieBreaker function adapted so that the modality class of each interval is looked up in Table 3.1 rather than calculated directly as in Cam9698 (Figure 3.15). This means that Cam03M is the same as Cam03A, except that

1. Cam03M is run on data in which the voices are arranged end-to-end; and
2. the call to SpellWin03 in line 16 of Cam03 (Figure 3.2) is replaced in Cam03M with a call to the function SpellWin03M defined in Figure 3.39.

SpellWin03M can be formed from SpellWin03 (Figure 3.4) by inserting the lines

			CAM03M	CAM03N	CAM03O	CAM03P	CAM01E	CAM03R	CAM03S	CAM03T	CAM03U	CAM01A	CAM01B	CAM01C	CAM9698B
1	Window size	$3n$	9	9	9	9	9	9	9	9	9	9	9	9	9
2	Fixed or variable length window	(fix,var)	fix	fix	fix	fix	var	fix	fix	fix	fix	var	var	var	fix
3	How the notes are sorted in **MidiList** (voice-leading)	(int,end)	end	end	int	int	int	int	int	int	int	int	int	int	end
4	Method of computing interval optimisation penalties (IOPs)	(mod,lof)	mod	mod	mod	mod	mod	mod	mod	mod	mod	mod	lof	lof	mod
5	If modality-based IOP calculation, how is modality class determined?	(tab,sca)	tab	tab	sca	sca	tab	tab	tab	tab	tab	tab			sca
6	If modality-based IOP calculation, what penalties applied to modality classes?	A,B,C,D	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,2	0,0,1,3			0,0,1,4
7	If line-of-fifths-based IOP calculation, are ["a1"] and ["d1"] 'demoted'?	(yes,no)											no	yes	
8	Penalty values for notational parsimony	one,both	2,4	2,4	2,4	2,4	2,4	2,4	2,4	2,4	2,4	4,8	13,26	13,26	2,6
9	Spellings constrained to being within either flatside or sharpside region	(yes,no)	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
10	Retain pitch name classes for first third of window from previous window	(yes,no)	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
11	Include interval penalties for non-contiguous notes in window	(yes,no)	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
12	If fixed-length window and voices end-to-end, is TIEBREAKER used?	(yes,no)	yes	yes											yes
13	If modality-based IOP derived directly from scales, how are scales weighted?	dia,amm,mh			2,1,1	6,1,2									6,1,2
14	If modality-based IOP derived from scales, what are values of modality class boundaries?	$0 < x < 0.5$			0.25	0.4									0.25
15	If variable-length window used, does each window contain fixed number of MIDI note numbers or pitch classes?	(mid,pc)					pc					mid	mid	mid	
16	For each MIDI note number or pitch class within single window, pitch name class is constant	(mid,pc,no)	no	no	no	no	pc	mid	pc	no	no	mid	mid	mid	no
17	Uses "flat/natural-or-sharp/natural-first" heuristic	(yes,no)	no	no	no	no	no	no	no	yes	no	no	no	no	no
25	If spellings constrained to be within flatside or sharpside, use first third of window to eliminate one side	(yes,no)	no	no	no	no	no	no	no	no	yes	no	no	no	no

Table 3.10: The values of the variable features for each version of Cambouropoulos's algorithm tested here. All values different from those for CAM03A are highlighted. See text for explanation.

```
TieBreakerM(S1, S2)
1   n ← |S1|
2   M1 ← ⨁_{i=0}^{n−2} ⟨LookUpModClass(PNC2PINC(S1[i], S1[i + 1]))⟩
3   M2 ← ⨁_{i=0}^{n−2} ⟨LookUpModClass(PNC2PINC(S2[i], S2[i + 1]))⟩
4   H1 ← ⟨ ⟩
5   H2 ← ⟨ ⟩
6   for i ← 0 to n − 3
7       if (M1[i] ≠ M1[i + 1]) ∧ (M1[i] ∈ {A, B, C}) ∧ (M1[i + 1] ∈ {C, D})
8           H1 ← H1 ⊕ ⟨i⟩
9       if (M2[i] ≠ M2[i + 1]) ∧ (M2[i] ∈ {A, B, C}) ∧ (M2[i + 1] ∈ {C, D})
10          H2 ← H2 ⊕ ⟨i⟩
11  if |H1| < |H2|
12      return true
13  if |H1| > |H2|
14      return false
15  P1 ← Σ_{p∈H1} p
16  P2 ← Σ_{p∈H2} p
17  return P1 < P2
```

Figure 3.40: The TieBreakerM function.

```
LookUpModClass(PINC)
1   PINCs ← ⟨["p1"], ["rp4"], ["rp5"], ["rmi2"], ["rma7"], ["rma2"], ["rmi7"],
             ["rmi3"], ["rma6"], ["rma3"], ["rmi6"], ["ra2"], ["rd7"],
             ["rd3"], ["ra6"], ["rd4"], ["ra5"], ["ra4"], ["rd5"]⟩
2   ModClasses ← ⟨A, A, A, B, B, B, B, B, B, B, B, C, C, C, C, C, C, C, C⟩
3   p ← Pos(PINC, PINCs)
4   if p
5       return ModClasses[p]
6   else
7       return D
```

Figure 3.41: The LookUpModClass function.

```
SpellWin03N(WinPNCList, WinMidiList, FirstWindow, LastWindow)
1    WinMidiListSize ← |WinMidiList|
2    MaxBitVecInt ← 2^WinMidiListSize − 1
3    BestSpelling ← ⟨ ⟩
4    for i ← 0 to MaxBitVecInt
5       BitVec ← BitVector(i, WinMidiListSize)
6       Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
7       SpellingPenalty ← ComputeSpellPen03K(WinPNCList ⊕ Spelling)
8       if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
9          BestSpelling ← Spelling
10         BestPenalty ← SpellingPenalty
11      else
12         if (SpellingPenalty = BestPenalty) ∧ (TieBreakerM(Spelling, BestSpelling))
13            BestSpelling ← Spelling
14            BestPenalty ← SpellingPenalty
15      Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
16      SpellingPenalty ← ComputeSpellPen03K(WinPNCList ⊕ Spelling)
17      if SpellingPenalty < BestPenalty
18         BestSpelling ← Spelling
19         BestPenalty ← SpellingPenalty
20      else
21         if (SpellingPenalty = BestPenalty) ∧ (TieBreakerM(Spelling, BestSpelling))
22            BestSpelling ← Spelling
23            BestPenalty ← SpellingPenalty
24   ▶ Now return appropriate part of BestSpelling.
25   BSEnd ← WinMidiListSize/2
26   if FirstWindow
27      BSEnd ← 2WinMidiListSize/3
28   if LastWindow
29      BSEnd ← WinMidiListSize
30   return BestSpelling[0, BSEnd]
```

Figure 3.42: The SpellWin03N algorithm.

```
else
   if (SpellingPenalty = BestPenalty) ∧ (TieBreakerM(Spelling, BestSpelling))
      BestSpelling ← Spelling
      BestPenalty ← SpellingPenalty
```

between lines 10 and 11 and again between lines 15 and 16 in SpellWin03. The TieBreakerM function is defined in Figure 3.40. TieBreakerM is the same as TieBreaker (Figure 3.26), except that

1. TieBreakerM does not take the argument **BlendedModTable**; and
2. the calls to ComputeModClass in lines 2 and 3 of TieBreaker are replaced in TieBreakerM with corresponding calls to the function LookUpModClass, defined in Figure 3.41.

The function LookUpModClass simply looks up the modality class of a pitch interval name class in a representation of Table 3.1.

Cam03N is the same as Cam03L, except that Cam03N uses TieBreakerM. This means that Cam03N is the same as Cam03A except that,

1. in Cam03N, the input data is sorted so that the voices are arranged end-to-end;
2. in Cam03N, only intervals between contiguous notes are taken into account when calculating the overall penalty score for a particular spelling; and
3. the TieBreakerM function is used to decide between window spellings that have the least overall penalty score.

This means that Cam03N can be formed from Cam03A by sorting the input data so that the voices are arranged end-to-end rather than interleaved and replacing the call to SpellWin03 in line 16 of Cam03 (Figure 3.2) with a call to SpellWin03N, defined in Figure 3.42. SpellWin03N is the same as SpellWin03M (Figure 3.39), except that the calls to ComputeSpellPen03 in lines 7 and 16 of SpellWin03M are replaced in SpellWin03N with calls to ComputeSpellPen03K (Figure 3.38).

Cam03O is the same as Cam03F (Figure 3.32), except that the scales are weighted differently in calculating the blended interval modality table. That is, Cam03O is the same as Cam03F, except that line 11 in Cam03F is replaced in Cam03O with the line

$$\begin{aligned}\textbf{BlendedModTable} \leftarrow \textsc{ComputeBlendedModTable}(&0.25,\\ &\langle\langle \textbf{MajorScale}, 2\rangle,\\ &\ \langle \textbf{AscMelMinScale}, 1\rangle,\\ &\ \langle \textbf{HarmMinScale}, 1\rangle\rangle)\end{aligned}$$

This also means that lines 7 and 8 in Cam03F can be omitted from Cam03O.

Cam03P is the same as Cam03F (Figure 3.32), except that the modality class boundaries are changed by using a different value for the variable that Cambouropoulos (1998, p. 70) denotes by x. That is, Cam03P is the same as Cam03F, except that line 11 in Cam03F is replaced in Cam03P by the line

$$\begin{aligned}11\quad \textbf{BlendedModTable} \leftarrow \textsc{ComputeBlendedModTable}(&0.4,\\ &\langle\langle \textbf{MajorScale}, 4\rangle,\\ &\ \langle \textbf{NatMinScale}, 1\rangle,\\ &\ \langle \textbf{DescMelMinScale}, 1\rangle,\\ &\ \langle \textbf{AscMelMinScale}, 1\rangle,\\ &\ \langle \textbf{HarmMinScale}, 2\rangle\rangle)\end{aligned}$$

Changing the value of the parameter x from 0.25 to 0.40 has the effect of:

1. shifting the boundary between modality classes A and B one step to the right in Table 3.3; and
2. shifting the boundary between modality classes B and C one step to the left in Table 3.3 (see page 120).

This results in ["rma2"] and ["rmi7"] being moved to class A and ["rmi2"] and ["rma7"] being moved to class C.

```
CAM01E(MidiList)
1   MidiListSize ← |MidiList|
2   PCList ← ⨁_{i=0}^{MidiListSize−1} ⟨MidiList[i] mod 12⟩
3   RetSegStart ← 0
4   PNCList ← ⟨ ⟩
5   while |PNCList| < MidiListSize
6     ▶ Find WinStart and PrePCSet.
7     WinStart ← RetSegStart
8     PrePCSet ← ⟨ ⟩
9     while (WinStart > 0) ∧ (|PrePCSet| < 3)
10      WinStart ← WinStart − 1
11      if PCList[WinStart] ∉ PrePCSet
12        PrePCSet ← ⟨PCList[WinStart]⟩ ⊕ PrePCSet
13    ▶ Now find WinEnd and WinPCSet.
14    WinEnd ← RetSegStart
15    WinPCSet ← PrePCSet
16    while (WinEnd < MidiListSize) ∧ (|WinPCSet| ≤ 9)
17      if PCList[WinEnd] ∉ WinPCSet
18        WinPCSet ← WinPCSet ⊕ ⟨PCList[WinEnd]⟩
19      WinEnd ← WinEnd + 1
20    if |WinPCSet| > 9
21      WinEnd ← WinEnd − 1
22      WinPCSet ← WinPCSet[0, 9]
23    ▶ If WinPCSet is too small, extend PrePCSet and WinStart backwards,
24    ▶ remembering to update both PrePCSet and WinPCSet.
25    while (|WinPCSet| < 9) ∧ (WinStart > 0)
26      WinStart ← WinStart − 1
27      if PCList[WinStart] ∉ PrePCSet
28        PrePCSet ← ⟨PCList[WinStart]⟩ ⊕ PrePCSet
29      if PCList[WinStart] ∉ WinPCSet
30        WinPCSet ← ⟨PCList[WinStart]⟩ ⊕ WinPCSet
31    ▶ Now find RetSegEnd and PostPCSet.
32    RetSegEnd ← WinEnd
33    PostPCSet ← ⟨ ⟩
34    if WinEnd ≠ MidiListSize
35      while (RetSegEnd > 0) ∧ (|PostPCSet| < 3)
36        RetSegEnd ← RetSegEnd − 1
37        if PCList[RetSegEnd] ∉ PostPCSet
38          PostPCSet ← ⟨PCList[RetSegEnd]⟩ ⊕ PostPCSet
39    ▶ Now find PrePNCList.
40    PrePNCList ← ⟨ ⟩
41    LocalPrePCSet ← ⟨ ⟩
42    i ← RetSegStart − 1
43    while |PrePNCList| ≠ |PrePCSet|
44      if PCList[i] ∉ LocalPrePCSet
45        PrePNCList ← ⟨PNCList[i]⟩ ⊕ PrePNCList
46        LocalPrePCSet ← ⟨PCList[i]⟩ ⊕ LocalPrePCSet
47      i ← i − 1
48    ▶ Now remove PrePCSet from WinPCSet to get NewWinPCSet.
49    NewWinPCSet ← ⟨ ⟩
50    for i ← 0 to |WinPCSet| − 1
51      if WinPCSet[i] ∉ PrePCSet
52        NewWinPCSet ← NewWinPCSet ⊕ ⟨WinPCSet[i]⟩
53    ▶ Now append best spelling for this retained segment to PNCList.
54    RetSeg ← PCList[RetSegStart, RetSegEnd]
55    PNCList ← PNCList ⊕ SPELLWIN01E(PrePNCList, PrePCSet, NewWinPCSet, RetSeg)
56    ▶ Finally, set RetSegStart to equal start position of next retained segment.
57    RetSegStart ← RetSegEnd
58  return PNCList
```

Figure 3.43: The CAM01E algorithm

```
SpellWin01E(PrePNCList, PrePCSet, NewWinPCSet, RetSeg)
1    NewWinPCSetSize ← |NewWinPCSet|
2    MaxBitVecInt ← 2^NewWinPCSetSize − 1
3    BestSpelling ← ⟨ ⟩
4    for i ← 0 to MaxBitVecInt
5       BitVec ← BitVector(i, NewWinPCSetSize)
6       Spelling ← ComputeSpelling(NewWinPCSet, BitVec, flatside)
7       SpellingPenalty ← ComputeSpellPen03(PrePNCList ⊕ Spelling)
8       if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
9          BestSpelling ← Spelling
10         BestPenalty ← SpellingPenalty
11      Spelling ← ComputeSpelling(NewWinPCSet, BitVec, sharpside)
12      SpellingPenalty ← ComputeSpellPen03(PrePNCList ⊕ Spelling)
13      if SpellingPenalty < BestPenalty
14         BestSpelling ← Spelling
15         BestPenalty ← SpellingPenalty
16   ▶ Now spell pitches in retained segment and return it.
17   SpeltRetSeg ← ⟨ ⟩
18   for i ← 0 to |RetSeg| − 1
19      PNC ← (PrePNCList ⊕ BestSpelling)[Pos(RetSeg[i], PrePCSet ⊕ NewWinPCSet)]
20      SpeltRetSeg ← SpeltRetSeg ⊕ ⟨PNC⟩
21   return SpeltRetSeg
```

Figure 3.44: The SpellWin01E algorithm.

Cam01E is the same as Cam01D, except that each variable length window in Cam01E contains 9 distinct pitch classes; whereas, in Cam01D, each window contains 9 distinct MIDI note numbers. Cam01E is defined in Figure 3.43. Cam01E is formed from Cam01 (Figure 3.12) by:

1. changing all references to *WinSize* to 9;
2. inserting a line between lines 1 and 2 of Cam01 in which a variable **PCList** is made equal to the list of pitch classes such that **PCList**$[i]$ is the pitch class of **MidiList**$[i]$ for all $0 \leq i < |\mathbf{MidiList}|$;
3. replacing all references to **MidiList** after line 1 in Cam01 with references to **PCList**;
4. changing the name of the variable **WinMIDISet** to **WinPCSet**;
5. changing the name of the variable **NewWinMIDISet** to **NewWinPCSet**;
6. changing the name of the variable **PreMIDISet** to **PrePCSet**;
7. changing the name of the variable **LocalPreMidiSet** to **LocalPrePCSet**;
8. replacing the call to SpellWin01 in line 54 of Cam01 with the call to SpellWin01E in line 55 of Cam01E.

SpellWin01E is defined in Figure 3.44. SpellWin01E is the same as SpellWin01 (Figure 3.13), except that

1. the variables **PreMIDISet** **NewWinMIDISet** and *NewWinMIDISetSize* in SpellWin01 are renamed **PrePCSet**, **NewWinPCSet** and *NewWinPCSetSize*, respectively, in SpellWin01E;

```
CAM03R(MidiList)
1    MidiListSize ← |MidiList|
2    LastWindow ← false
3    FirstWindow ← true
4    WinStart ← 0
5    PNCList ← ⟨ ⟩
6    while |PNCList| < MidiListSize
7        WinEnd ← MIN({ WinStart + 9, MidiListSize})
8        if WinEnd = MidiListSize
9            LastWindow ← true
10       if FirstWindow
11           WinPNCList ← ⟨ ⟩
12           WinMidiList ← MidiList[WinStart, WinEnd]
13           PreMIDIList ← ⟨ ⟩
14       else
15           WinPNCList ← PNCList[WinStart, WinStart + 3]
16           WinMidiList ← MidiList[WinStart + 3, WinEnd]
17           PreMIDIList ← MidiList[WinStart, WinStart + 3]
18       PNCList ← PNCList ⊕ SPELLWIN03R(WinPNCList, PreMIDIList, WinMidiList,
19                                       FirstWindow, LastWindow)
20       FirstWindow ← false
21       WinStart ← WinStart + 3
22   return PNCList
```

Figure 3.45: The CAM03R algorithm.

2. the calls to COMPUTESPELLPEN01 in lines 7 and 12 of SPELLWIN01 are replaced in SPELLWIN01E with calls to COMPUTESPELLPEN03 (Figure 3.6).

CAM03R is the same as CAM03A, except that, within each window, every occurrence of a given MIDI note number must always have the same pitch name class. CAM03R is defined in Figure 3.45. It is essentially the same as CAM03 (Figure 3.2), run with a window size of 9, except that the call to SPELLWIN03 in line 16 of CAM03 is replaced with a call to SPELLWIN03R in line 18 of CAM03R. SPELLWIN03R has to be provided with a list of the MIDI note numbers occurring in the first third of each window, therefore it is given an argument, **PreMIDIList**, which contains these MIDI note numbers. The value of **PreMIDIList** is set in either line 13 or 17 of CAM03R, depending on whether or not the current window being processed is the first window. The function SPELLWIN03R called in line 18 of CAM03R is defined in Figure 3.46.

CAM03S is the same as CAM03A, except that, within each window, every occurrence of a given pitch class must always be assigned the same pitch name class. CAM03S is the same as CAM03R (Figure 3.45), except that lines 18–19 of CAM03R are replaced in CAM03S with the lines

```
18       PNCList ← PNCList ⊕ SPELLWIN03S(WinPNCList, PreMIDIList, WinMidiList,
19                                       FirstWindow, LastWindow)
```

The function SPELLWIN03S is defined in Figure 3.47. It is the same as SPELLWIN03R (Figure 3.46), except that

1. the variables *PreMIDISet*, *WinMIDISet* and *MidiSet* in SPELLWIN03R are replaced in SPELLWIN03S with the variables *PrePCSet*, *WinPCSet* and *PCSet*;

```
SpellWin03R(WinPNCList, PreMIDIList, WinMidiList, FirstWindow, LastWindow)
1   WinMidiListSize ← |WinMidiList|
2   WinPNCSet ← ⋃_{k=0}^{|WinPNCList|−1} {WinPNCList[k]}
3   PreMIDISet ← ⋃_{k=0}^{|PreMIDIList|−1} {PreMIDIList[k]}
4   WinMIDISet ← ⋃_{k=0}^{|WinMidiList|−1} {WinMidiList[k]}
5   MidiSet ← PreMIDISet ∪ WinMIDISet
6   MaxBitVecInt ← 2^WinMidiListSize − 1
7   BestSpelling ← ⟨ ⟩
8   for i ← 0 to MaxBitVecInt
9       BitVec ← BitVector(i, WinMidiListSize)
10      Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
11      SpellingSet ← ⋃_{k=0}^{WinMidiListSize−1} {Spelling[k]}
12      if (|MidiSet| ≥ |WinPNCSet ∪ SpellingSet|)
13          SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
14          if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
15              BestSpelling ← Spelling
16              BestPenalty ← SpellingPenalty
17      Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
18      SpellingSet ← ⋃_{k=0}^{WinMidiListSize−1} {Spelling[k]}
19      if (|MidiSet| ≥ |WinPNCSet ∪ SpellingSet|)
20          SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
21          if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
22              BestSpelling ← Spelling
23              BestPenalty ← SpellingPenalty
24  ▶ Now return appropriate part of BestSpelling.
25  BSEnd ← WinMidiListSize/2
26  if FirstWindow
27      BSEnd ← 2WinMidiListSize/3
28  if LastWindow
29      BSEnd ← WinMidiListSize
30  return BestSpelling[0, BSEnd]
```

Figure 3.46: The SpellWin03R algorithm.

```
SpellWin03S(WinPNCList, PreMIDIList, WinMidiList, FirstWindow, LastWindow)
1   WinMidiListSize ← |WinMidiList|
2   WinPNCSet ← ⋃_{k=0}^{|WinPNCList|−1} {WinPNCList[k]}
3   PrePCSet ← ⋃_{k=0}^{|PreMIDIList|−1} {PreMIDIList[k] mod 12}
4   WinPCSet ← ⋃_{k=0}^{|WinMidiList|−1} {WinMidiList[k] mod 12}
5   PCSet ← PrePCSet ∪ WinPCSet
6   MaxBitVecInt ← 2^WinMidiListSize − 1
7   BestSpelling ← ⟨ ⟩
8   for i ← 0 to MaxBitVecInt
9       BitVec ← BitVector(i, WinMidiListSize)
10      Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
11      SpellingSet ← ⋃_{k=0}^{WinMidiListSize−1} {Spelling[k]}
12      if (|PCSet| = |WinPNCSet ∪ SpellingSet|)
13          SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
14          if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
15              BestSpelling ← Spelling
16              BestPenalty ← SpellingPenalty
17      Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
18      SpellingSet ← ⋃_{k=0}^{WinMidiListSize−1} {Spelling[k]}
19      if (|PCSet| = |WinPNCSet ∪ SpellingSet|)
20          SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
21          if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
22              BestSpelling ← Spelling
23              BestPenalty ← SpellingPenalty
24  ▶ Now return appropriate part of BestSpelling.
25  BSEnd ← WinMidiListSize/2
26  if FirstWindow
27      BSEnd ← 2WinMidiListSize/3
28  if LastWindow
29      BSEnd ← WinMidiListSize
30  return BestSpelling[0, BSEnd]
```

Figure 3.47: The SpellWin03S algorithm.

```
SpellWin03T(WinPNCList, WinMidiList, FirstWindow, LastWindow)
1    WinMidiListSize ← |WinMidiList|
2    FlatNatPNCs ← ⟨"Cn", "Df", "Dn", "Ef", "En", "Fn", "Gf", "Gn", "Af", "An", "Bf", "Bn"⟩
3    BestSpelling ← ⊕_{k=0}^{|WinMidiList|−1} ⟨FlatNatPNCs[WinMidiList[k] mod 12]⟩
4    if (ComputeSpellPen03(WinPNCList ⊕ BestSpelling) ≠ 0)
5       SharpNatPNCs ← ⟨"Cn", "Cs", "Dn", "Ds", "En", "Fn", "Fs", "Gn", "Gs", "An", "As", "Bn"⟩
6       BestSpelling ← ⊕_{k=0}^{|WinMidiList|−1} ⟨SharpNatPNCs[WinMidiList[k] mod 12]⟩
7       if (ComputeSpellPen03(WinPNCList ⊕ BestSpelling) ≠ 0)
8          MaxBitVecInt ← 2^WinMidiListSize − 1
9          BestSpelling ← ⟨ ⟩
10         for i ← 0 to MaxBitVecInt
11            BitVec ← BitVector(i, WinMidiListSize)
12            Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
13            SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
14            if (BestSpelling = ⟨ ⟩) ∨ (SpellingPenalty < BestPenalty)
15               BestSpelling ← Spelling
16               BestPenalty ← SpellingPenalty
17            Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
18            SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
19            if SpellingPenalty < BestPenalty
20               BestSpelling ← Spelling
21               BestPenalty ← SpellingPenalty
22   ▶ Now return appropriate part of BestSpelling.
23   BSEnd ← WinMidiListSize/2
24   if FirstWindow
25      BSEnd ← 2WinMidiListSize/3
26   if LastWindow
27      BSEnd ← WinMidiListSize
28   return BestSpelling[0, BSEnd]
```

Figure 3.48: The SpellWin03T algorithm.

2. the variable *PrePCSet* is set in line 3 of SpellWin03S to be the set of pitch classes in **PreMIDIList**, whereas the variable *PreMIDISet* is set in line 3 of SpellWin03R to be the set of MIDI note numbers in **PreMIDIList**; and

3. the variable *WinPCSet* is set in line 4 of SpellWin03S to be the set of pitch classes in **WinMidiList**, whereas the variable *WinMIDISet* is set in line 4 of SpellWin03R to be the set of MIDI note numbers in **WinMidiList**.

Cam03T is the same as Cam03A, except that Cam03T uses what I have called the "flat/natural-or-sharp/natural-first" heuristic suggested by Cambouropoulos (2003, p. 427). Cam03T is the same as Cam03 (Figure 3.2) run with *WinSize* equal to 9, except that lines 16–17 of Cam03 are replaced in Cam03T with the lines

```
16      PNCList ← PNCList ⊕ SpellWin03T(WinPNCList, WinMidiList,
17                                      FirstWindow, LastWindow)
```

The function SpellWin03T is defined in Figure 3.48. As suggested by Cambouropoulos, SpellWin03T first checks to see if the window spelling that only uses flats and naturals has an overall penalty score of 0, and, if it does, it returns the appropriate part of this spelling (see lines 2–4).

```
SpellWin03U(WinPNCList, WinMidiList, FirstWindow, LastWindow)
1    CheckSharpSide ← true
2    CheckFlatSide ← true
3    for i ← 0 to |WinPNCList| − 1
4       if WinPNCList[i] ∈ {"Dff", "Eff", "Ff", "Gff", "Aff", "Bff", "Cf"}
5          CheckSharpSide ← false
6       if WinPNCList[i] ∈ {"Bs", "Css", "Dss", "Es", "Fss", "Gss", "Ass"}
7          CheckFlatSide ← false
8    WinMidiListSize ← |WinMidiList|
9    MaxBitVecInt ← 2^WinMidiListSize − 1
10   BestSpelling ← ⟨⟩
11   for i ← 0 to MaxBitVecInt
12      BitVec ← BitVector(i, WinMidiListSize)
13      if CheckFlatSide
14         Spelling ← ComputeSpelling(WinMidiList, BitVec, flatside)
15         SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
16         if (BestSpelling = ⟨⟩) ∨ (SpellingPenalty < BestPenalty)
17            BestSpelling ← Spelling
18            BestPenalty ← SpellingPenalty
19      if CheckSharpSide
20         Spelling ← ComputeSpelling(WinMidiList, BitVec, sharpside)
21         SpellingPenalty ← ComputeSpellPen03(WinPNCList ⊕ Spelling)
22         if (BestSpelling = ⟨⟩) ∨ (SpellingPenalty < BestPenalty)
23            BestSpelling ← Spelling
24            BestPenalty ← SpellingPenalty
25   ▶ Now return appropriate part of BestSpelling.
26   BSEnd ← WinMidiListSize/2
27   if FirstWindow
28      BSEnd ← 2WinMidiListSize/3
29   if LastWindow
30      BSEnd ← WinMidiListSize
31   return BestSpelling[0, BSEnd]
```

Figure 3.49: The SpellWin03U algorithm.

If this "flat/natural" spelling has a non-zero overall penalty score in line 4, then the algorithm checks if the "sharp/natural" spelling has an overall penalty of zero and, if it does, it returns the appropriate part of this spelling (see lines 5–7). If both the "flat/natural" and the "sharp/natural" spellings have non-zero penalties, then, in lines 8–21, SpellWin03T simply searches through all the possible spellings for the window in the same way as SpellWin03 (Figure 3.4).

Cam03U is the same as Cam03A, except that it uses the pitch name classes in the first third of each window to eliminate, if possible, either all the flatside spellings or all the sharpside spellings for the window. Cam03U is Cam03 (Figure 3.2) run with *WinSize* equal to 9 and with the call to SpellWin03 in line 16 of Cam03 replaced with a corresponding call to the function SpellWin03U, defined in Figure 3.49. In lines 1–7 of SpellWin03U, **WinPNCList** is examined to determine whether either of the pitch name class regions can be eliminated, and, if one of them can, then either *CheckSharpSide* or *CheckFlatSide* is set to false. The rest of the algorithm is the same as SpellWin03 (Figure 3.4), except that a flatside spelling is only evaluated if *CheckFlatSide* is true and a sharpside spelling is only evaluated if *CheckSharpSide* is true.

The last four versions of the algorithm given in Table 3.10 were only included in the evaluation

OPNDV2MIDILIST$_1$(*OPNDV*)
1 **OPNDV** ← SORT$_{\text{onset}}$(*OPNDV*)
2 **return** $\bigoplus_{i=0}^{|\mathbf{OPNDV}|-1}$ ⟨PN2P(**OPNDV**[*i*][1])[0] + 21⟩

Figure 3.50: The OPNDV2MIDILIST$_1$ algorithm.

OPNDV2MIDILIST$_2$(*OPNDV*)
1 **OPNDV** ← SORT$_{\text{voice}}$(*OPNDV*)
2 **return** $\bigoplus_{i=0}^{|\mathbf{OPNDV}|-1}$ ⟨PN2P(**OPNDV**[*i*][1])[0] + 21⟩

Figure 3.51: The OPNDV2MIDILIST$_2$ algorithm.

because they were evaluated by Cambouropoulos himself.

As discussed in section 3.5.2 above, CAM01A is CAM01 (Figure 3.12) run with *WinSize* set to 9.

As discussed in section 3.5.2, CAM01B is the same as CAM01A, except that the penalties for each interval are calculated in the same way as in CAM03B. That is, the interval optimization penalty is looked up in Table 3.4 and the notational parsimony penalty is defined to be 13. This means that CAM01B is the same as CAM01 (Figure 3.12) run with *WinSize* equal to 9, except that every call to COMPUTESPELLPEN01 in SPELLWIN01 (Figure 3.13) is replaced with a corresponding call to COMPUTESPELLPEN03B, defined in Figure 3.31.

As mentioned in section 3.5.2, CAM01C is the same as CAM01B, except that the interval optimization penalties were derived in the same way as in CAM03C—that is, they were looked up in Table 3.5 (Cambouropoulos, 2001, p. 6). This means that CAM01C is identical to CAM01B except that line 1 in COMPUTESPELLPEN03B (Figure 3.31) is replaced with

1 **PINCs** ← ⟨["p1"], ["rp5"], ["rp4"], ["rma2"], ["rmi7"],
["rma6"], ["rmi3"], ["rma3"], ["rmi6"], ["rma7"],
["rmi2"], ["ra4"], ["rd5"], ["ra5"], ["rd4"],
["ra2"], ["rd7"], ["ra6"], ["rd3"], ["ra3"],
["rd6"], ["ra7"], ["rd2"], ["a1"], ["d1"]⟩

In section 3.5.3 above, CAM9698A was defined to be simply CAM9698 (Figure 3.15) run with *WinSize* set to 12 in order to approximate as closely as possible to the version of the algorithm tested by Cambouropoulos (1996, 1998) (see discussion in sections 3.4.1 and 3.6.2.1 above). However, with this value of *WinSize*, my implementation of CAM9698A would have taken several months to run on the entire test corpus, because it is not uncommon for the algorithm to have to evaluate over 100000 different spellings for a single window. I therefore ran CAM9698 (Figure 3.15) with *WinSize* set to 9 instead of 12. I call this version of the algorithm CAM9698B.

3.6.4 Generating MidiLists from OPNDV files

In CAM9698B, CAM03E, CAM03L, CAM03M and CAM03N, the input data has to be sorted so that the voices are arranged end-to-end. In the other versions of the algorithm tested, the data is sorted so that the voices are interleaved. This means that each OPNDV dataset in the test corpus had to be converted into two **MidiList**s. The function OPNDV2MIDILIST$_1$, defined in Figure 3.50, takes an OPNDV dataset and returns a **MidiList** in which the MIDI note numbers are arranged so that the voices are interleaved. The function OPNDV2MIDILIST$_2$, defined in Figure 3.51, takes an OPNDV dataset and returns a **MidiList** in which the MIDI note numbers are arranged so that the voices are arranged end-to-end. The functions SORT$_{\text{onset}}$ and SORT$_{\text{voice}}$ used in the first line of each of these functions were defined in section 2.4.2.3 above.

3.6.5 Results and discussion

Tables 3.11, 3.12 and 3.13 summarise the results obtained when the 26 versions of Cambouropoulos's algorithm described in section 3.6.3 were run on the test corpus, $\mathcal{C}$, defined in Table 1.4 above.

Tables 3.11 and 3.12 show the note error counts and note accuracies for the various versions of the algorithm tested, for the complete test corpus, $\mathcal{C}$, and for each subset of $\mathcal{C}$ containing movements by one of the eight composers. In the first column of each of these tables, an entry of the form "C1x" represents the algorithm CAM01x. Similarly, an entry of the form "C3x" indicates the algorithm CAM03x. For example, "C1A" represents CAM01A and "C3F" indicates CAM03F. The algorithm CAM9698B is represented by the abbreviation "C96B".

As can be seen in Table 3.11, the versions of the algorithm that made the fewest number of errors were CAM01A and CAM01D. Both of these algorithms made 1822 errors on the test corpus. However, they did not make the same errors and, as can be seen in Table 3.12, the style dependence of CAM01A was slightly less than that of CAM01D. In terms of spelling accuracy and style dependence, the best version of the algorithm tested was therefore CAM01A.

Another basic observation to be made is that all the versions of the algorithm tested that used a variable length window (apart from CAM01B which used the basic line-of-fifths method for calculating IOPs) performed better than the versions that used a fixed length window of the same size. However, by using a larger fixed length window of 12 notes, CAM03D succeeded in matching the spelling accuracy of CAM01C and CAM01E which used variable length windows containing 9 notes.

In Table 3.12, it can be seen that, in CAM03L, CAM03K and CAM03N, ignoring intervals between non-contiguous notes when calculating the penalty for a spelling caused both a relatively large drop in spelling accuracy and a relatively large increase in style dependence. This effect was compounded further in CAM9698B which both ignores intervals between non-contiguous intervals and fails to retain the pitch names for the first third of each window from the previous window.

Comparing the results in Table 3.12 with those for Longuet-Higgins's algorithm in Table 2.3 reveals that nearly all the versions of Cambouropoulos's algorithm tested achieved higher spelling accuracies than the best version of Longuet-Higgins's algorithm. Also, the versions of Cambouropoulos's algorithm tested were markedly less dependent on style than those of Longuet-

Algorithm	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
C1A	283	232	141	128	476	200	172	190	1822
C1D	303	295	131	125	476	161	168	163	1822
C1C	299	312	148	123	462	173	176	179	1872
C1E	286	171	172	161	483	183	262	159	1877
C3D	267	280	103	169	476	159	216	212	1882
C3P	264	347	105	228	479	182	208	230	2043
C1B	350	331	164	143	482	230	178	183	2061
C3C	267	334	108	262	490	180	196	254	2091
C3A	291	331	109	273	446	172	284	236	2142
C3H	291	331	109	273	446	172	284	236	2142
C3I	291	331	109	273	446	172	284	236	2142
C3T	291	331	109	273	446	172	284	236	2142
C3U	291	331	109	273	446	172	284	236	2142
C3S	335	350	113	181	495	178	219	274	2145
C3R	294	331	109	308	448	174	272	231	2167
C3F	292	324	105	244	523	167	272	241	2168
C3O	292	324	105	244	523	167	272	241	2168
C3G	291	359	108	234	498	172	239	317	2218
C3B	316	378	115	301	497	230	189	262	2288
C3E	326	315	168	186	569	204	284	360	2412
C3M	332	317	167	192	577	213	286	361	2445
C3J	424	423	179	234	542	179	382	355	2718
C3L	540	492	261	260	735	325	416	416	3445
C3K	589	396	240	288	820	297	427	398	3455
C3N	638	472	310	303	750	331	511	457	3772
C96B	525	721	249	303	794	401	564	838	4395

Table 3.11: Note error counts for algorithms in column labelled "Algorithm" for the complete test corpus (last column), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus. The algorithms are sorted into increasing order of overall note error count. See text and sections 1.3.4 and 1.3.6 for further details.

Algorithm	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\overline{\text{NA}}$	SD_{Sty}
C1A	98.85	99.05	99.42	99.48	98.06	99.18	99.30	99.22	99.07	99.07	0.46
C1D	98.76	98.80	99.47	99.49	98.06	99.34	99.31	99.33	99.07	99.07	0.50
C1C	98.78	98.73	99.40	99.50	98.11	99.29	99.28	99.27	99.04	99.04	0.47
C1E	98.83	99.30	99.30	99.34	98.03	99.25	98.93	99.35	99.04	99.04	0.45
C3D	98.91	98.86	99.58	99.31	98.06	99.35	99.12	99.13	99.04	99.04	0.46
C3P	98.92	98.58	99.57	99.07	98.04	99.26	99.15	99.06	98.96	98.96	0.46
C1B	98.57	98.65	99.33	99.42	98.03	99.06	99.27	99.25	98.95	98.95	0.49
C3C	98.91	98.64	99.56	98.93	98.00	99.27	99.20	98.96	98.93	98.93	0.47
C3A	98.81	98.65	99.55	98.89	98.18	99.30	98.84	99.04	98.91	98.91	0.41
C3H	98.81	98.65	99.55	98.89	98.18	99.30	98.84	99.04	98.91	98.91	0.41
C3I	98.81	98.65	99.55	98.89	98.18	99.30	98.84	99.04	98.91	98.91	0.41
C3T	98.81	98.65	99.55	98.89	98.18	99.30	98.84	99.04	98.91	98.91	0.41
C3U	98.81	98.65	99.55	98.89	98.18	99.30	98.84	99.04	98.91	98.91	0.41
C3S	98.63	98.57	99.54	99.26	97.98	99.27	99.11	98.88	98.91	98.91	0.50
C3R	98.80	98.65	99.55	98.74	98.17	99.29	98.89	99.06	98.89	98.89	0.42
C3F	98.81	98.68	99.57	99.00	97.86	99.32	98.89	99.02	98.89	98.89	0.50
C3O	98.81	98.68	99.57	99.00	97.86	99.32	98.89	99.02	98.89	98.89	0.50
C3G	98.81	98.53	99.56	99.04	97.97	99.30	99.02	98.71	98.87	98.87	0.49
C3B	98.71	98.46	99.53	98.77	97.97	99.06	99.23	98.93	98.83	98.83	0.48
C3E	98.67	98.71	99.31	99.24	97.68	99.17	98.84	98.53	98.77	98.77	0.53
C3M	98.65	98.71	99.32	99.22	97.64	99.13	98.83	98.53	98.75	98.75	0.53
C3J	98.27	98.27	99.27	99.04	97.79	99.27	98.44	98.55	98.61	98.61	0.53
C3L	97.80	97.99	98.93	98.94	97.00	98.67	98.30	98.30	98.24	98.24	0.65
C3K	97.60	98.38	99.02	98.82	96.65	98.79	98.26	98.38	98.24	98.24	0.78
C3N	97.40	98.07	98.73	98.76	96.94	98.65	97.91	98.13	98.08	98.08	0.66
C96B	97.86	97.06	98.98	98.76	96.76	98.36	97.70	96.58	97.76	97.76	0.91

Table 3.12: Note accuracies expressed as percentages for each algorithm in the column headed "Algorithm" for the complete test corpus (column 10), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The columns headed $\overline{\text{NA}}$ and SD_{Sty} give the mean and standard deviation, respectively, of the values in columns 2 to 9. The algorithms are sorted into decreasing order of overall note accuracy. See text and sections 1.3.4 and 1.3.6 for further details.

	C1A	C1D	C1C	C1E	C3D	C3P	C1B	C3C	C3A	C3H	C3I	C3T	C3U	C3S	C3R	C3F	C3O	C3G	C3B	C3E	C3M	C3J	C3L	C3K	C3N	C96B
C1A	0.00	0.00	2.74	3.02	3.29	12.13	13.12	14.76	17.56	17.56	17.56	17.56	17.56	17.73	18.94	18.99	18.99	21.73	25.58	32.38	34.19	49.18	89.08	89.63	107.03	141.22
C1D	0.00	0.00	2.74	3.02	3.29	12.13	13.12	14.76	17.56	17.56	17.56	17.56	17.56	17.73	18.94	18.99	18.99	21.73	25.58	32.38	34.19	49.18	89.08	89.63	107.03	141.22
C1C	-2.67	-2.67	0.00	0.27	0.53	9.13	10.10	11.70	14.42	14.42	14.42	14.42	14.42	14.58	15.76	15.81	15.81	18.48	22.22	28.85	30.61	45.19	84.03	84.56	101.50	134.78
C1E	-2.93	-2.93	-0.27	0.00	0.27	8.84	9.80	11.40	14.12	14.42	14.12	14.12	14.12	14.28	15.45	15.50	15.50	18.17	21.90	28.50	30.26	44.81	83.54	84.07	100.96	134.15
C3D	-3.19	-3.19	-0.53	-0.27	0.00	8.55	9.51	11.11	13.82	13.82	13.82	13.82	13.82	13.97	15.14	15.20	15.20	17.85	21.57	28.16	29.91	44.42	83.05	83.58	100.43	133.53
C3P	-10.82	-10.82	-8.37	-8.13	-7.88	0.00	0.88	2.35	4.85	4.85	4.85	4.85	4.85	4.99	6.07	6.12	6.12	8.57	11.99	18.06	19.68	33.04	68.62	69.11	84.63	115.12
C1B	-11.60	-11.60	-9.17	-8.93	-8.69	-0.87	0.00	1.46	3.93	3.93	3.93	3.93	3.93	4.08	5.14	5.19	5.19	7.62	11.01	17.03	18.63	31.88	67.15	67.64	83.02	113.25
C3C	-12.86	-12.86	-10.47	-10.23	-10.00	-2.30	-1.43	0.00	2.44	2.44	2.44	2.44	2.44	2.58	3.63	3.68	3.68	6.07	9.42	15.35	16.93	29.99	64.75	65.23	80.39	110.19
C3A	-14.94	-14.94	-12.61	-12.37	-12.14	-4.62	-3.78	-2.38	0.00	0.00	0.00	0.00	0.00	0.14	1.17	1.21	1.21	3.55	6.82	12.61	14.15	26.89	60.83	61.30	76.10	105.18
C3H	-14.94	-14.94	-12.61	-12.37	-12.14	-4.62	-3.78	-2.38	0.00	0.00	0.00	0.00	0.00	0.14	1.17	1.21	1.21	3.55	6.82	12.61	14.15	26.89	60.83	61.30	76.10	105.18
C3I	-14.94	-14.94	-12.61	-12.37	-12.14	-4.62	-3.78	-2.38	0.00	0.00	0.00	0.00	0.00	0.14	1.17	1.21	1.21	3.55	6.82	12.61	14.15	26.89	60.83	61.30	76.10	105.18
C3T	-14.94	-14.94	-12.61	-12.37	-12.14	-4.62	-3.78	-2.38	0.00	0.00	0.00	0.00	0.00	0.14	1.17	1.21	1.21	3.55	6.82	12.61	14.15	26.89	60.83	61.30	76.10	105.18
C3U	-14.94	-14.94	-12.61	-12.37	-12.14	-4.62	-3.78	-2.38	0.00	-0.14	-0.14	-0.14	0.00	0.00	1.03	1.07	1.07	3.40	6.67	12.45	13.99	26.71	60.61	61.07	75.85	104.90
C3S	-15.06	-15.06	-12.73	-12.49	-12.26	-4.76	-3.92	-2.52	-0.14	-0.14	-0.14	-0.14	-0.14	0.00	0.00	0.05	0.05	2.35	5.58	11.31	12.83	25.43	58.98	59.44	74.07	102.81
C3R	-15.92	-15.92	-13.61	-13.38	-13.15	-5.72	-4.89	-3.51	-1.15	-1.15	-1.15	-1.15	-1.15	-1.02	0.00	0.05	0.05	2.35	5.58	11.31	12.83	25.43	58.98	59.44	74.07	102.81
C3F	-15.96	-15.96	-13.65	-13.42	-13.19	-5.77	-4.94	-3.55	-1.20	-1.20	-1.20	-1.20	-1.20	-1.06	-0.05	0.00	0.00	2.31	5.54	11.25	12.78	25.37	58.90	59.36	73.99	102.72
C3O	-15.96	-15.96	-13.65	-13.42	-13.19	-5.77	-4.94	-3.55	-1.20	-1.20	-1.20	-1.20	-1.20	-1.06	-0.05	0.00	0.00	2.31	5.54	11.25	12.78	25.37	58.90	59.36	73.99	102.72
C3G	-17.85	-17.85	-15.60	-15.37	-15.15	-7.89	-7.08	-5.73	-3.43	-3.43	-3.43	-3.43	-3.43	-3.29	-2.30	-2.25	0.00	3.16	8.75	10.23	22.54	55.32	55.77	70.06	98.15
C3B	-20.37	-20.37	-18.18	-17.96	-17.74	-10.71	-9.92	-8.61	-6.38	-6.38	-6.38	-6.38	-6.38	-6.25	-5.29	-5.24	-5.24	-3.06	0.00	5.42	6.86	18.79	50.57	51.01	64.86	92.09
C3E	-24.46	-24.46	-22.39	-22.18	-21.97	-15.30	-14.55	-13.31	-11.19	-11.19	-11.19	-11.19	-11.19	-11.07	-10.16	-10.12	-10.12	-8.04	-5.14	0.00	1.37	12.69	42.83	43.24	56.38	82.21
C3M	-25.48	-25.48	-23.44	-23.23	-23.03	-16.44	-15.71	-14.48	-12.39	-12.39	-12.39	-12.39	-12.39	-12.27	-11.37	-11.33	-11.33	-9.28	-6.42	-1.35	0.00	11.17	40.90	41.31	54.27	79.75
C3J	-32.97	-32.97	-31.13	-30.94	-30.76	-24.83	-24.17	-23.07	-21.19	-21.19	-21.19	-21.19	-21.19	-21.08	-20.27	-20.24	-20.24	-18.40	-15.82	-11.26	-10.04	0.00	26.75	27.12	38.78	61.70
C3L	-47.11	-47.11	-45.66	-45.52	-45.37	-40.70	-40.17	-39.30	-37.82	-37.82	-37.82	-37.82	-37.82	-37.74	-37.10	-37.07	-37.07	-35.62	-33.58	-29.99	-29.03	-21.10	0.00	0.29	9.49	27.58
C3K	-47.26	-47.26	-45.82	-45.67	-45.53	-40.87	-40.35	-39.48	-38.00	-38.00	-38.00	-38.00	-38.00	-37.92	-37.28	-37.25	-37.25	-35.80	-33.78	-30.19	-29.23	-21.33	-0.29	0.00	9.18	27.21
C3N	-51.70	-51.70	-50.37	-50.24	-50.11	-45.84	-45.36	-44.57	-43.21	-43.21	-43.21	-43.21	-43.21	-43.13	-42.55	-42.52	-42.52	-41.20	-39.34	-36.06	-35.18	-27.94	-8.67	-8.40	0.00	16.52
C96B	-58.54	-58.54	-57.41	-57.29	-57.18	-53.52	-53.11	-52.42	-51.26	-51.26	-51.26	-51.26	-51.26	-51.19	-50.69	-50.67	-50.67	-49.53	-47.94	-45.12	-44.37	-38.16	-21.62	-21.39	-14.18	0.00

Table 3.13: Each entry in this table gives the percentage increase in the note error rate caused by changing from the algorithm at the head of the entry's row to the algorithm at the head of the entry's column. A negative value means that the note error rate was reduced by the change in algorithm. See text and section 1.3.4 for further details.

Higgins.

Each entry in Table 3.13 gives the percentage increase in the note error rate caused by changing from the algorithm at the head of the entry's row to the algorithm at the head of the entry's column. For example, changing from CAM01A to CAM9698B increased the note error rate by 141.22% (i.e., by almost $2\frac{1}{2}$ times). More details are given in the table captions. The algorithms are arranged in these tables in non-increasing order of note accuracy.

Recall from section 1.3.4.3 that, because of the sudden enharmonic change at bar 166 in the fourth movement of Haydn's Symphony No. 100 in G ('Military') (Hob. I:100), I compared the output of each algorithm with two "correct" versions of this movement: one in which the notes are spelt as they are in the original score; and a second, modified version, in which all the notes in the original score up to bar 165 are transposed down a diminished second. Every version of Cambouropoulos's algorithm tested here generated a spelling for this movement that was more similar to the original score than the modified spelling, typically assigning pitch names to about 60 notes in this movement that were different from those in the original score (i.e., 99.06% correct).

Table 3.14 shows the actual time taken for each algorithm to process the test corpus, $\mathcal{C}$. Each algorithm was implemented in the same version of Lisp (Digitool Macintosh Common Lisp 4.3.5) and run on the same computer (an Apple Macintosh Dual 2GHz PowerPC G5 with 1GB DDR SDRAM). Also, the implementations were straightforward, direct translations into Lisp of the pseudocode given above. Therefore, the times given in Table 3.14 may be considered comparable. Table 3.14 also shows, for each algorithm, the average time taken per note and the average speed in notes per second over $\mathcal{C}$. Each entry in Table 3.15 gives the percentage increase in speed caused by changing from the algorithm at the head of the entry's row to the algorithm at the head of the entry's column.

Every version of Cambouropoulos's algorithm has a worst-case running time which is linear in the size of the input data. However, as can be seen in Tables 3.14 and 3.15, the actual time taken for the algorithms to process the test corpus used in this evaluation varied by a factor of 631 from about two weeks for CAM9698B down to about half an hour for CAM01E.

Table 3.16 gives a summary of the individual effects of changes in the values of variable features on overall note error rate and average speed. Columns 5 and 6 in this table show the percentage increase in note error rate and speed, respectively, that result from the change in algorithm indicated in column 3. Column 4 of each row shows the changes in the values of the variable features implied by the change in algorithm given in the third column. The first column gives, for each variable feature, the number of the row in Table 3.8 that corresponds to that variable feature.

From Table 3.16 it is possible to predict the combinations of values for the variable features that result in the fastest and the most accurate versions of Cambouropoulos's algorithm. I shall now consider what these results tell us about each of the variable features tested in this evaluation.

Algorithm	Total time (hours)	Mean time per note (ms)	Mean speed (notes/sec)
C1E	0.532	9.774	102.317
C3S	1.488	27.337	36.580
C3N	2.836	52.090	19.197
C3K	2.836	52.105	19.192
C3L	2.970	54.563	18.328
C1A	6.180	113.535	8.808
C1D	6.293	115.605	8.650
C3T	6.353	116.713	8.568
C1B	6.420	117.943	8.479
C1C	6.805	125.008	7.999
C3R	9.918	182.198	5.489
C3A	11.957	219.655	4.553
C3B	11.984	220.145	4.542
C3G	12.310	226.135	4.422
C3U	12.384	227.503	4.396
C3M	12.464	228.957	4.368
C3E	12.677	232.883	4.294
C3C	13.046	239.660	4.173
C3H	14.009	257.352	3.886
C3P	16.591	304.774	3.281
C3F	17.177	315.534	3.169
C3O	17.314	318.056	3.144
C3I	48.287	887.039	1.127
C3D	71.428	1312.129	0.762
C3J	99.659	1830.733	0.546
C96B	335.270	6158.897	0.162

Table 3.14: The column headed 'Total time (hours)' gives the total time in hours taken by each algorithm to process the test corpus, $\mathcal{C}$, defined in Table 1.4. Columns 3 and 4 give, respectively, the average time taken per note in ms and the average speed in notes per second over $\mathcal{C}$ for each algorithm. The algorithms are sorted by speed with the fastest at the top.

	C1A	C1B	C1C	C1D	C1E	C3A	C3B	C3C	C3D	C3E	C3F	C3G	C3H	C3I	C3J	C3K	C3L	C3M	C3N	C3O	C3P	C3R	C3S	C3T	C3U	C96B
C1A	0.00	-3.74	-9.18	-1.79	1061.64	-48.31	-48.43	-52.62	-91.35	-51.25	-64.02	-49.80	-55.88	-87.20	-93.80	117.89	108.08	-50.41	117.95	-64.31	-62.75	-37.68	315.30	-2.72	-50.09	-98.16
C1B	3.88	0.00	-5.66	2.02	1106.71	-46.30	-46.43	-50.78	-91.01	-49.36	-62.63	-47.85	-54.17	-86.71	-93.56	126.35	116.16	-48.48	126.41	-62.92	-61.30	-35.26	331.42	1.05	-48.15	-98.09
C1C	10.11	6.00	0.00	8.14	1179.12	-43.08	-43.22	-47.83	-90.47	-46.32	-60.38	-44.72	-51.42	-85.91	-93.17	139.93	129.13	-45.39	139.99	-60.70	-58.98	-31.38	357.31	7.11	-45.04	-97.97
C1D	1.83	-1.98	-7.53	0.00	1082.86	-47.36	-47.49	-51.76	-91.19	-50.36	-63.36	-48.88	-55.08	-86.97	-93.69	121.87	111.88	-49.50	121.93	-63.65	-62.07	-36.54	322.89	-0.95	-49.18	-98.13
C1E	-91.39	-91.71	-92.18	-91.55	0.00	-95.55	-95.56	-95.92	-99.26	-95.80	-96.90	-95.68	-96.20	-98.90	-99.47	-81.24	-82.09	-95.73	-81.24	-96.93	-96.79	-94.64	-64.25	-91.63	-95.70	-99.84
C3A	93.45	86.23	75.69	89.98	2147.24	0.00	-0.24	-8.35	-83.26	-5.69	-30.40	-2.88	-14.65	-75.25	-88.01	321.52	302.55	-4.06	321.63	-30.95	-27.94	20.56	703.43	88.18	-3.45	-96.44
C3B	93.92	86.68	76.11	90.44	2152.69	0.24	0.00	-8.12	-83.22	-5.46	-30.23	-2.64	-14.44	-75.19	-87.98	322.55	303.52	-3.83	322.66	-30.78	-27.76	20.85	705.37	88.64	-3.21	-96.43
C3C	111.07	103.19	91.68	107.28	2351.88	9.11	8.84	0.00	-81.74	2.90	-24.06	5.97	-6.88	-72.99	-86.92	359.91	339.20	4.67	360.03	-24.66	-21.38	31.54	776.59	105.32	5.34	-96.12
C3D	1055.91	1012.73	949.74	1035.17	13327.43	497.51	496.06	447.64	0.00	463.52	315.88	480.31	409.97	47.90	-28.35	2418.64	2305.25	473.23	2419.29	312.60	330.58	620.34	4700.52	1024.41	476.90	-78.74
C3E	105.12	97.46	86.28	101.44	2282.79	6.03	5.78	-2.82	-82.25	0.00	-26.20	2.98	-9.50	-73.75	-87.28	346.95	326.83	1.72	347.07	-26.78	-23.59	27.83	751.89	99.53	2.38	-96.23
C3F	177.94	167.56	152.41	172.96	3128.68	43.67	43.33	31.68	-75.95	35.50	0.00	39.54	22.63	-64.44	-82.77	505.62	478.35	37.84	505.77	-0.79	3.53	73.21	1054.31	170.37	38.72	-94.89
C3G	99.19	91.75	80.89	95.61	2213.82	2.96	2.71	-5.63	-82.77	-2.89	-28.34	0.00	-12.12	-74.51	-87.65	334.01	314.47	-1.22	334.12	-28.90	-25.80	24.13	727.23	93.76	-0.59	-96.34
C3H	126.66	118.19	105.84	122.59	2532.96	17.16	16.88	7.39	-80.39	10.50	-18.45	13.79	0.00	-71.00	-85.95	393.88	371.64	12.40	394.00	-19.09	-15.57	41.25	841.33	120.48	13.12	-95.83
C3I	681.54	652.35	609.76	667.52	8978.70	303.99	303.02	270.28	-32.39	281.01	181.19	292.37	244.81	0.00	-51.55	1602.93	1526.26	287.58	1603.37	178.97	191.13	387.05	3145.79	660.25	290.06	-85.63
C3J	1513.19	1452.93	1365.02	1484.25	18639.38	733.88	731.87	664.29	39.56	686.45	480.40	709.89	611.72	106.41	0.00	3415.02	3256.78	700.00	3415.93	475.82	500.92	905.31	6599.63	1469.23	705.13	-70.33
C3K	-54.11	-55.82	-58.32	-54.93	433.12	-76.28	-76.33	-78.26	-96.03	-77.63	-83.49	-76.96	-79.75	-94.13	-97.16	0.00	-4.50	-77.24	0.03	-83.62	-82.90	-71.40	90.60	-55.36	-77.09	-99.16
C3L	-51.94	-53.74	-56.36	-52.80	458.26	-75.16	-75.22	-77.23	-95.84	-76.57	-82.71	-75.87	-78.80	-93.85	-97.02	4.71	0.00	-76.17	4.74	-82.85	-82.10	-70.05	99.59	-53.25	-76.01	-99.12
C3M	101.65	94.12	83.13	98.03	2242.42	4.24	3.98	-4.46	-82.55	-1.69	-27.45	1.24	-11.03	-74.20	-87.50	339.38	319.60	0.00	339.49	-28.02	-24.89	25.66	737.45	96.15	0.64	-96.29
C3N	-54.12	-55.83	-58.33	-54.94	432.98	-76.28	-76.34	-78.26	-96.03	-77.63	-83.49	-76.97	-79.76	-94.13	-97.16	-0.03	-4.53	-77.25	0.00	-83.62	-82.91	-71.41	90.55	-55.37	-77.10	-99.16
C3O	180.15	169.69	154.42	175.13	3154.36	44.82	44.47	32.73	-75.76	36.58	0.80	40.65	23.60	-64.15	-82.63	510.43	482.95	38.93	510.59	0.00	4.36	74.59	1063.49	172.52	39.82	-94.85
C3P	168.45	158.43	143.80	163.64	3018.47	38.77	38.43	27.19	-76.78	30.87	-3.41	34.78	18.44	-65.65	-83.36	484.94	458.61	33.13	485.10	-4.18	0.00	67.30	1014.90	161.14	33.98	-95.06
C3R	60.47	54.47	45.73	57.59	1764.04	-17.05	-17.25	-23.98	-86.12	-21.77	-42.27	-19.44	-29.20	-79.47	-90.05	249.64	233.90	-20.42	249.74	-42.72	-40.23	0.00	566.42	56.09	-19.91	-97.05
C3S	-75.92	-76.82	-78.13	-76.35	179.71	-87.55	-87.58	-88.59	-97.92	-88.26	-91.34	-87.91	-89.38	-96.92	-98.51	-47.53	-49.90	-88.06	-47.52	-91.41	-91.03	-84.99	0.00	-76.58	-87.98	-99.56
C3T	2.80	-1.04	-6.64	0.96	1094.18	-46.86	-46.99	-51.30	-91.11	-49.88	-63.01	-48.39	-54.65	-86.85	-93.63	124.00	113.91	-49.02	124.05	-63.31	-61.71	-35.94	326.94	0.00	-48.69	-98.11
C3U	100.36	92.88	81.96	96.77	2227.50	3.57	3.32	-5.07	-82.67	-2.32	-27.91	0.59	-11.60	-74.36	-87.58	336.58	316.92	-0.64	336.69	-28.48	-25.36	24.86	732.12	94.90	0.00	-96.31
C96B	5337.04	5133.95	4837.65	5239.51	63058.64	2710.49	2703.70	2475.93	370.37	2550.62	1856.17	2629.63	2298.77	595.68	237.04	11746.91	11213.58	2596.30	11750.00	1840.74	1925.31	3288.27	22480.25	5188.89	2613.58	0.00

Table 3.15: Each entry in this table gives the percentage increase in speed caused by changing from the algorithm at the head of the entry's row to the algorithm at the head of the entry's column.

Row	Variable feature	From → To	From	→ To	% inc. in NER	% inc. in speed
1	Window size	C3A → C3D	9	→ 12	-12.14	-83.26
2	Fixed or variable length window	C3A → C1D	fix	→ var	-14.94	89.98
		C3R → C1D	fix	→ var	-15.92	57.59
		C3S → C1E	fix	→ var	-12.49	179.71
3	How the notes are sorted in **MidiList** (voice-leading)	C3A → C3E	int	→ end	12.61	-5.69
		C3K → C3L	int	→ end	-0.29	-4.50
4	Method of computing interval optimisation penalties (IOPs)	C3A → C3B	mod/tab/0,0,1,2/2,4	→ lof/no	6.82	-0.24
		C3A → C3C	mod/tab/0,0,1,2/2,4	→ lof/yes	-2.38	-8.35
		C3G → C3B	mod/tab/0,0,1,4/2,4	→ lof/no	3.16	2.71
		C3G → C3C	mod/tab/0,0,1,4/2,4	→ lof/yes	-5.73	-5.63
		C3H → C3B	mod/tab/0,0,1,2/4,8	→ lof/no	6.82	16.88
		C3H → C3C	mod/tab/0,0,1,2/4,8	→ lof/yes	-2.38	7.39
		C3F → C3B	mod/sca/0,0,1,2/2,4/6,1,2/0.25	→ lof/no	5.54	43.33
		C3F → C3C	mod/sca/0,0,1,2/2,4/6,1,2/0.25	→ lof/yes	-3.55	31.68
		C3O → C3B	mod/sca/0,0,1,2/2,4/2,1,1/0.25	→ lof/no	5.54	44.47
		C3O → C3C	mod/sca/0,0,1,2/2,4/2,1,1/0.25	→ lof/yes	-3.55	32.73
		C3P → C3B	mod/sca/0,0,1,2/2,4/6,1,2/0.40	→ lof/no	11.99	38.43
		C3P → C3C	mod/sca/0,0,1,2/2,4/6,1,2/0.40	→ lof/yes	2.35	27.19
		C1A → C1B	mod/tab/0,0,1,3/4,8	→ lof/no	13.12	-3.74
		C1A → C1C	mod/tab/0,0,1,3/4,8	→ lof/yes	2.74	-9.18
		C1D → C1B	mod/tab/0,0,1,2/2,4	→ lof/no	13.12	-1.98
		C1D → C1C	mod/tab/0,0,1,2/2,4	→ lof/yes	2.74	-7.53
5	If modality-based IOP calculation, how is modality class determined?	C3A → C3F	tab	→ sca	1.21	-30.40
		C3A → C3O	tab	→ sca	1.21	-30.95
		C3A → C3P	tab	→ sca	-4.62	-27.94
6	If modality-based IOP calculation, what penalties applied to modality classes?	C3A → C3G	0,0,1,2	→ 0,0,1,4	3.55	-2.88
7	If line-of-fifths-based IOP calculation, are ["a1"] and ["d1"] 'demoted'?	C3B → C3C	no	→ yes	-8.61	-8.12
		C1B → C1C	no	→ yes	-9.17	-5.66
8	Penalty values for notational parsimony	C3A → C3H	2,4	→ 4,8	0.00	-14.65
9	Spellings constrained to being within either flatside or sharpside region	C3A → C3I	yes	→ no	0.00	-75.25
10	Retain pitch name classes for first third of window from previous window	C3A → C3J	yes	→ no	26.89	-88.01
11	Include interval penalties for non-contiguous notes in window	C3A → C3K	yes	→ no	61.30	321.52
		C3E → C3L	yes	→ no	42.83	326.83
		C3M → C3N	yes	→ no	54.27	339.49
12	If fixed-length window and voices end-to-end, is TieBreaker used?	C3E → C3M	no	→ yes	1.37	1.72
		C3L → C3N	no	→ yes	9.49	4.74
13	If modality-based IOP derived directly from scales, how are scales weighted?	C3F → C3O	6,1,2	→ 2,1,1	0.00	-0.79
14	If modality-based IOP derived from scales, what are values of modality class boundaries?	C3F → C3P	0.25	→ 0.40	-5.77	3.53
15	If variable-length window used, does each window contain fixed number of MIDI note numbers or pitch classes?	C1D → C1E	mid	→ pc	3.02	1082.86
16	For each MIDI note number or pitch class within single window, pitch name class is constant	C3A → C3R	no	→ mid	1.17	20.56
		C3A → C3S	no	→ pc	0.14	703.43
		C3R → C3S	mid	→ pc	-1.02	566.42
17	Uses "flat/natural-or-sharp/natural-first" heuristic	C3A → C3T	no	→ yes	0.00	88.18
25	If spellings constrained to be within flatside or sharpside, use first third of window to eliminate one side	C3A → C3U	no	→ yes	0.00	-3.45

Table 3.16: Summary of individual effects on overall note error rate and average speed of changes in the values of variable features. Columns 5 and 6 show the percentage increase in note error rate and speed, respectively, that result from the change in algorithm indicated in column 3. Column 4 of each row shows the change in value for the variable feature implied by the change in algorithm given in the third column. The first column gives, for each variable feature, the number of the row in Table 3.8 that corresponds to that variable feature.

3.6.5.1 Window size (Tables 3.8 and 3.16, row 1)

Increasing the window size from 9 in CAM03A to 12 in CAM03D reduced the note error rate by over 12% but also increased the running time by nearly 6 times. A larger window size should therefore be combined with some speed enhancing features (e.g., a variable length window containing a fixed number of pitch classes) in order to achieve both high spelling accuracy and an acceptable running time.

These results support Cambouropoulos's (2003, p. 420) prediction, discussed in section 3.6.2.1 above, that increasing the window size should improve spelling accuracy. In section 3.6.2.1, I pointed out that increasing the window size from w to $w+k$ in CAM03 multiplies the worst-case running time by $(1 + k/w) \times 2^{2k/3}$. This implies that increasing from a window size of 9 in CAM03A to 12 in CAM03D multiplies the worst-case running time by $5\frac{1}{3}$ which agrees well with the observed increase in running time of 5.97 times.

3.6.5.2 Fixed or variable length window (Tables 3.8 and 3.16, row 2)

Changing from a fixed length window in CAM03A to a variable length one in CAM01D reduced the note error rate by nearly 15% *and* almost halved the running time. Similar improvements in note error rate and running time were observed by changing from CAM03R to CAM01D and from CAM03S to CAM01E. These results therefore suggest that using a variable length window instead of a fixed length one improves spelling accuracy by up to 15% and increases the speed by 2 to 3 times. These results support Cambouropoulos's (2001, p. 5) prediction that using a variable length window would improve both transcription quality and speed (see discussion in section 3.3.1 above).

3.6.5.3 How the notes are sorted in MidiList (voice-leading) (Tables 3.8 and 3.16, row 3)

Changing from interleaved **MidiList**s in CAM03A to end-to-end ones in CAM03E increased the note error rate by nearly 13% and also very slightly slowed down the algorithm. The only difference between CAM03K and CAM03L was also the ordering of the notes in the input **MidiList**s, with the voices being interleaved in CAM03K and arranged end-to-end in CAM03L. Changing from interleaved voices in CAM03K to end-to-end ones in CAM03L resulted in a small decrease in speed and an almost imperceptible improvement in spelling accuracy. Therefore, on balance, it would seem that Cambouropoulos's algorithm generally performs better when the voices are interleaved than when they are arranged end-to-end. This means that the algorithm can be used effectively on data in which the voicing structure is not given (e.g., one-channel MIDI format 0 files).

Neither CAM03E nor CAM03L attempt to use information about voice structure to improve performance. Therefore, the fact that they perform no better than CAM03A and CAM03K, respectively, does not imply that it would not be worth incorporating into the algorithm "additional rules" to "cater for voice-leading effects", as suggested by Cambouropoulos (2003, p. 427). However, when such a rule—the 'tie-breaker' rule—was incorporated into CAM03E and CAM03L, it actually *increased* the note error rate (see row headed 12 in Table 3.16). It would therefore seem

		Fixed or variable length window?	
		Fixed	Variable
Are ["a1"] and ["d1"] demoted?	No	CAM03B	CAM01B
	Yes	CAM03C	CAM01C

Table 3.17: Summary of relationships between CAM03B, CAM03C, CAM01B and CAM01C.

that, without incorporating more sophisticated rules to take voice-leading effects into account, Cambouropoulos's algorithm is better suited to processing data in which the notes are presented in the order in which they occur in the music, rather than one voice at a time.

3.6.5.4 Method of computing interval optimization penalties (IOPs) (Tables 3.8 and 3.16, row 4)

In CAM03B, CAM03C, CAM01B and CAM01C, the IOP assigned to each interval depends on its line-of-fifths distance; whereas, in the other versions of the algorithm tested, it depends on its modality class. CAM03B and CAM03C use fixed length windows whereas CAM01B and CAM01C use variable length windows. In CAM03C and CAM01C, ["a1"] and ["d1"] are 'demoted' (see section 3.6.2.7), whereas they are not in CAM03B and CAM01B. Table 3.17 summarises the relationships between these four algorithms.

The entries in the fourth column of the row headed 4 in Table 3.16 may need some explanation. Each entry on the left-hand side of this column indicates a combination of values for variable features relating to the modality-based method of calculating the IOPs. Each entry takes the form $val1/val2/\ldots/val4$ or $val1/val2/\ldots/val6$ in which

1. $val1$ indicates that the method is modality-based;
2. $val2$ indicates whether the modality class is determined from Table 3.1 ('tab') or the frequency with which the interval occurs within the tonal scales ('sca');
3. $val3$ gives the penalties applied to the modality classes;
4. $val4$ gives the notational parsimony penalties;
5. $val5$ gives the scale weighting for those algorithms in which the modality class of each interval is determined from the frequency with which it occurs in the tonal scales; and
6. $val6$ gives the value of the modality class boundary constant x (see section 3.4.3 above).

Each entry on the right-hand side of the fourth column in this row indicates a combination of values for a pair of variable features relating to the line-of-fifths-based method of calculating the IOPs. Each of these entries takes the form $val1/val2$ where

1. $val1$ indicates that the method is based on the line of fifths; and
2. $val2$ indicates whether or not ["a1"] and ["d1"] are 'demoted' (i.e., penalties increased from 7 to 12).

From Table 3.16 it can be seen in the row headed 4 that changing from a modality-based IOP calculation method to one in which the IOP is equal to the line-of-fifths distance (i.e., "lof/no" on the right-hand side of column 4), always increased the note error count. This agrees with the results that Cambouropoulos obtained in his own evaluations (see sections 3.5.2 and 3.5.1). However, for the fixed length window algorithms, changing from a modality-based IOP calculation method to one based on the line-of-fifths in which ["a1"] and ["d1"] were 'demoted' ("lof/yes"), actually *improved* the spelling accuracy in 5 out of 6 cases. Nevertheless, the spelling accuracy achieved using the modality-based method could be made to surpass that of this adjusted line-of-fifths method by increasing the value of the modality boundary constant x from 0.25 to 0.40 (in CAM03P). For the variable length window algorithms, the modality-based methods of calculating IOPs always resulted in better spelling accuracy than the line-of-fifths methods.

These results suggest that, when a fixed length window is used, Cambouropoulos's modality-based methods of calculating the IOPs are better than the basic line-of-fifths method but not clearly better than a line-of-fifths method in which ["a1"] and ["d1"] are demoted. Nevertheless, when a variable length window is used, it seems that using a modality-based IOP calculation method is always better than one based on the line of fifths.

3.6.5.5 If modality-based IOP calculation, how is modality class determined? (Tables 3.8 and 3.16, row 5)

In CAM03A, the modality class of each interval is determined using a 'table-based' method in which it is simply looked up in Table 3.1. CAM03A is the same as CAM03F, CAM03O and CAM03P, except that, in the latter three algorithms, the modality class of each interval is determined using a 'scale-based' method in which it is calculated from the frequency with which it occurs in the tonal scales. As shown in row 5 of Table 3.16, changing from the table-based method in CAM03A to the scale-based method used in CAM03F, CAM03O and CAM03P reduces the speed of the algorithm by around 30%. Also, when the modality class boundary constant x is set to 0.25 as suggested by Cambouropoulos, changing from the table-based method to the scale-based method slightly increases the note error rate. In other words, when $x = 0.25$, determining the modality classes of the intervals using the theoretically *incorrect* values in Table 3.1 actually leads to slightly better accuracy than when the modality classes are determined 'correctly' by directly calculating the frequencies of occurrence of the intervals in the tonal scales. However, raising the modality class boundary constant to 0.4 in CAM03P, causes this scale-based algorithm to achieve a note error rate which is almost 5% lower than that achieved by the table-based CAM03A.

Of the methods tested for determining the modality classes of the intervals, the one that performed best was therefore the scale-based one implemented in CAM03P. However, using a table-based method is about 40% faster than using a scale-based one. So if a scale-based method is used, this should be combined with speed-enhancing features that either improve or do not affect spelling accuracy.

3.6.5.6 If modality-based IOP calculation, what penalties applied to modality classes? (Tables 3.8 and 3.16, row 6)

CAM03A and CAM03G are identical, except that the penalty valued assigned to an interval for belonging to modality class D is 2 in CAM03A and 4 in CAM03G. The note error rate for CAM03G was slightly higher than that for CAM03A, suggesting that nothing is gained by using the modality class penalties used in CAM03G.

3.6.5.7 If line-of-fifths-based IOP calculation, are ["a1"] and ["d1"] 'demoted'? (Tables 3.8 and 3.16, row 7)

Defining the IOP of an interval to be its line-of-fifths distance is roughly equivalent to spelling notes so that they are as close together as possible on the line of fifths—a principle employed in Temperley and Sleator's system (see discussion of Temperley's (2001, p. 125) TPR 1 in section 4.3). It is also roughly equivalent to spelling notes so that they are as close as possible on the line of fifths to the local tonic. This principle underlies Stage 1 of my *ps13* algorithm (see section 6.1 and Eq. 6.1 below) and it is also the idea expressed in Longuet-Higgins's Rule 1 (Longuet-Higgins, 1987a, pp, 112–113—see also section 2.2).

By increasing the line-of-fifths-based IOP assigned to ["a1"] and ["d1"] from 7 to 12, Cambouropoulos effectively implements the idea that diatonic semitones are generally preferable to chromatic ones. This principle is stated more explicitly in Longuet-Higgins's theory as his Rule 4 (see section 2.2) and in Temperley's theory as his TPR 2 (see section 4.3). This is also the basic principle underlying Stage 2 of my *ps13* algorithm (see section 6.2.2 below).

It is therefore not surprising that the spelling accuracies of the line-of-fifths-based versions of Cambouropoulos's algorithm were improved by 'demoting' ["a1"] and ["d1"]: increasing the penalty assigned to ["a1"] and ["d1"] from 7 in CAM03B and CAM01B to 12 in CAM03C and CAM01C reduced the note error rate by about 9%.

3.6.5.8 Penalty values for notational parsimony (Table 3.8 and 3.16, row 8)

CAM03A and CAM03H are identical, except that the notational parsimony penalties in CAM03H are twice those in CAM03A. This means that, in CAM03H, notational parsimony plays a bigger part relative to interval optimization than it does in CAM03A. Changing from CAM03A to CAM03H made no difference to the note accuracy—indeed, CAM03H made exactly the same errors as CAM03A. Therefore increasing the relative importance of notational parsimony does not seem to improve spelling accuracy.

3.6.5.9 Spellings constrained to being within either flatside or sharpside region (Table 3.8 and 3.16, row 9)

The only difference between CAM03I and CAM03A is that the spelling of each window is not constrained to being within either the flatside region or the sharpside region in CAM03I. Removing this constraint from CAM03A had no effect on spelling accuracy but multiplied the running time by just over 4 times. Constraining the spelling of each window to being within either the flatside

region or the sharpside region therefore significantly enhances the speed of the algorithm without compromising accuracy.

3.6.5.10 Retain pitch name classes for first third of window from previous window (Table 3.8 and 3.16, row 10)

CAM03A and CAM03J are identical, except that, in CAM03J, the pitch name classes for the first third of each window are not constrained to being the same as they were in the retained section of the previous window. Removing this constraint from CAM03A increased the number of errors by about 27% and multiplied the running time by just over 8 times. For a window size of 9, as used in CAM03A and CAM03J, failing to fix the pitch names for the first third of the window effectively increases the number of notes in each window spelling by 3, which, in turn, results in 2^3 times as many spellings having to be considered for each window. This explains why the observed running time for CAM03J was approximately 8 times that of CAM03A. These results support Cambouropoulos's (2003, p. 422) claim that retaining the spellings for the first third of each window from the previous window improves both the spelling accuracy and the running time.

3.6.5.11 Include interval penalties for non-contiguous notes in window (Table 3.8 and 3.16, row 11)

CAM03A, CAM03E and CAM03M are identical to CAM03K, CAM03L and CAM03N, respectively, except that, in the latter three algorithms, intervals between non-contiguous notes are not considered when calculating the penalty for each window spelling. As shown in row 11 of Table 3.16, failing to consider intervals between non-contiguous notes increases the number of errors by between 40 and 60%. However, ignoring intervals between non-contiguous notes also makes the algorithm about 4 times faster when the window size is 9.

The best solution would therefore seem to be to consider intervals between non-contiguous notes but also incorporate speed-enhancing features that do not reduce spelling accuracy.

3.6.5.12 If fixed length window and voices end-to-end, is TIEBREAKER used? (Table 3.8 and 3.16, row 12)

Row 12 in Table 3.16 shows that incorporating the TIEBREAKER function into CAM03E and CAM03L to produce CAM03M and CAM03N, respectively, actually increased the note error rate. This suggests that the tie-breaker rule does not help to improve spelling accuracy even when the notes are ordered so that the voices are arranged end-to-end. Clearly, some other rule needs to be used if one wishes to "cater to voice-leading effects" properly (Cambouropoulos, 2003, p. 427).

3.6.5.13 If modality-based IOP derived directly from scales, how are scales weighted? (Table 3.8 and 3.16, row 13)

The effect of changing the weighting of the tonal scales in the scale-based method of calculating IOPs can be explored to some small extent by comparing the performance of CAM03F with

that of CAM03O. These algorithms are identical, except for the weighting of the scales used. Changing from a weighting of '6,1,2' in CAM03F to one of '2,1,1' in CAM03P made no difference whatsoever to spelling accuracy. The resulting decrease in speed was also almost imperceptible. We may conclude that this particular change in the scale weighting had no effect. Note that, in both CAM03F and CAM03O, the diatonic scale was weighted more heavily than the other scales. Perhaps a more radical change in the pattern of weights would have had an observable effect on performance.

3.6.5.14 If modality-based IOP derived from scales, what are values of modality class boundaries? (Table 3.8 and 3.16, row 14)

Raising the value of the modality class boundary constant x from 0.25 in CAM03F to 0.4 in CAM03P reduced the note error rate by almost 6%. Recall that this change in the value of x simply moved ["rma2"] and ["rmi7"] to modality class A and ["rmi2"] and ["rma7"] to modality class C (see Table 3.3 on page 120). In every other respect, CAM03F was identical to CAM03P, therefore it would seem that reducing the range of modality values in modality class B and increasing the ranges of modality classes A and C improves spelling accuracy. This suggests that eliminating modality class B altogether by making x equal to 0.5 might result in a further improvement in spelling accuracy. As can be seen in Table 3.3 on page 120, setting x to 0.5 would cause

1. modality class A to contain ["p1"], ["rp4"], ["rma2"] and ["rmi3"] and their inversions;
2. modality class B to be empty; and
3. modality class C to contain all other intervals contained within the major and minor scales (i.e., ["rma3"], ["rmi2"], ["ra4"], ["rd4"], ["ra2"] and their inversions).

More generally, there does not seem to be any reason why modality class boundaries should be restricted to being definable by using just a single parameter x such that the boundaries occur at x and $1 - x$. In principle, one could place the modality class boundaries between any of the columns in Table 3.3. It may be that exhaustively testing all such possible boundary position combinations might reveal a better interval categorisation scheme than either of the ones tested in this study.

3.6.5.15 If variable length window used, does each window contain fixed number of MIDI note numbers or pitch classes? (Table 3.8 and 3.16, row 15)

Changing from windows containing a fixed number of MIDI note numbers in CAM01D to ones containing the same number of pitch classes in CAM01E increased the note error rate by 3% but caused almost a twelve-fold increase in speed. This result is in general agreement with Cambouropoulos's (2004) intuition that using a fixed number of pitch classes instead of a fixed number of MIDI note numbers in each window was faster but less accurate (see section 3.6.2.15 above). Nevertheless, the reduction in accuracy is very small and the increase in speed is rather large, suggesting that, for most applications, it would probably be advisable to use a fixed number of pitch classes rather than a fixed number of MIDI note numbers.

3.6.5.16 For each MIDI note number or pitch class within single window, pitch name class is constant (Table 3.8 and 3.16, row 16)

In CAM03R, no two occurrences of the same MIDI note number within a single window may be assigned different pitch name classes. Similarly, in CAM03S, no two occurrences of the same pitch class within a single window may be assigned different pitch name classes. In all other respects, CAM03R and CAM03S are identical to CAM03A. The results summarised in row 16 of Table 3.16 reveal that changing from CAM03A to CAM03R slightly increased the note error rate but also increased the speed by over 20%. On the other hand, changing from CAM03A to CAM03S had hardly any effect on the note error rate and also made the algorithm over 8 times faster. It would therefore seem that, at least when a fixed length window is used, disallowing different pitch name classes to be assigned to different occurrences of the same pitch class within a single window makes the algorithm much faster without compromising spelling accuracy.

3.6.5.17 Uses "flat/natural-or-sharp/natural-first" heuristic (Table 3.8 and 3.16, row 17)

Comparing the results obtained using CAM03A with those obtained using CAM03T shows that, by using Cambouropoulos's (2003, p. 427) "flat/natural-or-sharp/natural-first" heuristic (see section 3.6.2.17), the speed of the algorithm can be almost doubled without affecting the note error rate.

3.6.5.18 If using flatside/sharpside regions, use first third of window to eliminate one side (Table 3.8 and 3.16, row 25)

In section 3.6.2.25 above, I suggested that the speed of the algorithm could be improved by using the pitch name classes assigned to the first third of each window to eliminate either the flatside or the sharpside region from consideration in certain cases. Comparison of the results obtained with CAM03U (which incorporates this heuristic), with those of CAM03A (which does not) reveals that this heuristic actually very slightly reduced the speed of the algorithm and had no effect on spelling accuracy. It would therefore seem that including this heuristic in the algorithm would not be worthwhile.

3.7 Towards an optimal version of Cambouropoulos's algorithm

3.7.1 Optimal variable-feature values

The results summarised in Table 3.16 and discussed in section 3.6.5 can be used to estimate the combination of values for the variable features that achieves the best spelling accuracy in a reasonable time. I therefore designed one final version of Cambouropoulos's algorithm in which I attempted to combine those features of the tested versions that resulted in the best spelling accuracy. This version is called CAMOPT and Table 3.18 shows the values that CAMOPT takes for the 18 variable features explored in the evaluation described above. I shall now explain the logic behind the design of CAMOPT.

			CamOpt
1	Window size	$3n$	12
2	Fixed or variable length window	(fix,var)	var
3	How the notes are sorted in **MidiList** (voice-leading)	(int,end)	int
4	Method of computing interval optimisation penalties (IOPs)	(mod,lof)	mod
5	If modality-based IOP calculation, how is modality class determined?	(tab,sca)	sca
6	If modality-based IOP calculation, what penalties applied to modality classes?	A,B,C,D	0,0,1,2
7	If line-of-fifths-based IOP calculation, are ["a1"] and ["d1"] 'demoted'?	(yes,no)	
8	Penalty values for notational parsimony	one,both	2,4
9	Spellings constrained to being within either flatside or sharpside region	(yes,no)	yes
10	Retain pitch name classes for first third of window from previous window	(yes,no)	yes
11	Include interval penalties for non-contiguous notes in window	(yes,no)	yes
12	If fixed-length window and voices end-to-end, is TieBreaker used?	(yes,no)	
13	If modality-based IOP derived directly from scales, how are scales weighted?	dia,amm,mh	6,1,2
14	If modality-based IOP derived from scales, what are values of modality class boundaries?	$0 < x < 0.5$	0.4
15	If variable-length window used, does each window contain fixed number of MIDI note numbers or pitch classes?	(mid,pc)	mid
16	For each MIDI note number or pitch class within single window, pitch name class is constant	(mid,pc,no)	mid
17	Uses "flat/natural-or-sharp/natural-first" heuristic	(yes,no)	yes
25	If spellings constrained to be within flatside or sharpside, use first third of window to eliminate one side	(yes,no)	no

Table 3.18: The values of the variable features for CamOpt. See text for explanation.

Comparison of the results obtained for Cam03A and Cam03D suggested that a larger window size leads to better spelling accuracy. However, a window containing more than 12 notes is not practical—even when a variable length window is used. I therefore set *WinSize* to 12 in CamOpt. (See row 1 of Tables 3.16 and 3.18 and section 3.6.5.1.)

In the tested versions of the algorithm, changing from a fixed length window to a variable length window of the same size improved both running time and spelling accuracy. Therefore CamOpt uses a variable length window. (See row 2 of Tables 3.16 and 3.18 and section 3.6.5.2.)

In section 3.6.5.3 above, it was concluded that Cambouropoulos's algorithm was better suited for processing data in which the notes are presented in the order in which they occur in the music, rather than one voice at a time. The **MidiList**s used as input to CamOpt are therefore sorted so that the voices are interleaved. (See row 3 of Tables 3.16 and 3.18.)

In section 3.6.5.4 above, it was shown that, when a variable length window is used, the algorithm achieves a higher note accuracy when the IOP of an interval is based on its modality class rather than its line-of-fifths distances. CamOpt therefore uses a modality-class-based method for calculating the IOPs. (See row 4 of Tables 3.16 and 3.18.)

In section 3.6.5.5 above, it was shown that when the IOP of an interval is based on its modality class, the method of calculating the modality class that led to the best spelling accuracy was the one implemented in Cam03P. Therefore, in CamOpt, the modality class of each interval is computed by calculating the frequency with which the interval occurs in the tonal scales with the scales weighted and the modality class boundaries defined as they are in Cam03P. (See rows 5, 13 and 14 in Tables 3.16, 3.18 and 3.10.)

In CamOpt, the penalties assigned to the modality classes are as in Cam03A (i.e., 0, 0, 1 and 2 for modality classes A, B, C and D, respectively), since it was found that changing these

penalties slightly increased the note error rate. A considerable amount of extra work would have to be done in order to determine the optimal values for these four parameters. (See row 6 in Tables 3.16 and 3.18.)

In section 3.6.5.8 above, it was pointed out that doubling the notational parsimony penalties by changing from CAM03A to CAM03H did not improve spelling accuracy. Therefore, in CAMOPT, the notational parsimony penalties were set to be the same as those in CAM03A. (See row 8 in Tables 3.16, 3.18, and 3.9.)

In section 3.6.5.9 above, it was concluded that constraining the spelling of each window to being within either the flatside region or the sharpside region markedly improved running time without affecting spelling accuracy. Therefore, this constraint was incorporated into CAMOPT. (See row 9 in Tables 3.16 and 3.18.)

In CAMOPT, the pitch names for the first third of each window are retained from the previous window, as this strategy was shown to improve both running time and spelling accuracy (see section 3.6.5.10 and row 10 in Tables 3.16 and 3.18.)

In section 3.6.5.11, it was concluded that ignoring the intervals between non-contiguous notes within a window when computing spelling penalties resulted in a severe increase in the number of errors. Therefore, in CAMOPT, the IOPs for intervals between both contiguous and non-contiguous notes within a window contribute to the overall penalty for each spelling. (See row 11 in Tables 3.16 and 3.18.)

In section 3.6.5.15 above, it was shown that, when a variable length window is used, changing from a window containing a fixed number of MIDI note numbers to one containing the same number of pitch classes slightly increased the note error rate. In CAMOPT, each window therefore contains 12 distinct MIDI note numbers (see row 15 in Tables 3.16 and 3.18). This implies that no two occurrences of the same MIDI note number within a single window may be assigned distinct pitch name classes (see row 16 in Tables 3.16 and 3.18). This does not preclude the possibility of disallowing two occurrences of the same pitch class within a single window from having distinct pitch name classes. However, this option was not tested here.

In section 3.6.5.17 above, it was concluded that Cambouropoulos's proposed "flat/natural-or-sharp/natural-first" heuristic improves running time without compromising spelling accuracy. This heuristic was therefore incorporated into CAMOPT (see row 17 in Tables 3.16 and 3.18).

Finally, in section 3.6.5.18 above, I showed that my proposed heuristic for improving running time by eliminating either the flatside or sharpside spellings for certain windows on the basis of the pitch names of the first third of the window did not improve the speed of the algorithm. This heuristic was therefore omitted from CAMOPT. (See row 25 in Tables 3.16 and 3.18.)

3.7.2 The CAMOPT algorithm

Figure 3.52 gives pseudocode for the CAMOPT algorithm. CAMOPT is the same as CAM01 run with *WinSize* set to 12 (see Figure 3.12), except that

1. CAMOPT incorporates the "flat/natural-or-sharp/natural-first" heuristic used in CAM03T; and

2. in CAMOPT, the penalty for each window spelling is calculated in the same way as it is in

CAM03P.

Consequently, lines 1–3 of CAMOPT are the same as lines 1–3 of CAM01 and lines 10–62 of CAMOPT are the same as lines 4–57 of CAM01 run with *WinSize* set to 12, except that the call to SPELLWIN01 in line 54 of CAM01 is replaced in CAMOPT with a corresponding call to the SPELLWINOPT function defined in Figure 3.53. Because the IOP calculation in CAMOPT is scale-based, a blended modality table has to be computed in the same way as it is in CAM9698 (see Figure 3.15). This is done in lines 4–9 of CAMOPT which are the same as lines 6–11 of CAM9698, except that the first argument to COMPUTEBLENDEDMODTABLE in line 9 of CAMOPT is 0.4 as it is in line 11 of CAM03P, instead of 0.25 as it is in CAM9698 (see description of CAM03P in section 3.6.3 above).

The SPELLWINOPT function defined in Figure 3.53 is based on the SPELLWIN01 function used in CAM01 (see Figure 3.13). Thus, line 1 in SPELLWINOPT is the same as line 1 in SPELLWIN01, and lines 8–27 in SPELLWINOPT are the same as lines 2–21 in SPELLWIN01, except that the calls to COMPUTESPELLPEN01 in lines 7 and 12 in SPELLWIN01 are replaced in SPELLWINOPT with corresponding calls to the COMPUTESPELLPEN03F function defined in Figure 3.34. COMPUTESPELLPEN03F is used here because it is the function used in CAM03P to compute the overall penalty for each window spelling.

However, unlike SPELLWIN01, SPELLWINOPT incorporates the "flat/natural-or-sharp/natural-first" heuristic used in CAM03T. This heuristic is implemented in lines 2–7 of SPELLWINOPT in exactly the same way as in lines 2–7 of SPELLWIN03T (see Figure 3.48), except that

1. the calls to COMPUTESPELLPEN03 in lines 4 and 7 of SPELLWIN03T are replaced in lines 4 and 7 of SPELLWINOPT with corresponding calls to COMPUTESPELLPEN03F;

2. the variables **WinMidiList** and **WinPNCList** in SPELLWIN03T are replaced with the variables **NewWinMIDISet** and **PrePNCList**, respectively, in SPELLWINOPT.

Note also that, because IOP calculation is scale-based in SPELLWINOPT, this function takes one more argument than SPELLWIN01, namely, **BlendedModTable**, which is always equal to the value of the **BlendedModTable** variable computed in line 9 of CAMOPT. **BlendedModTable** is then used by COMPUTESPELLPEN03F to compute the overall penalty for each window spelling.

3.7.3 Results of running CAMOPT on the test corpus $\mathcal{C}$

Tables 3.19, 3.20 and 3.21 summarise the results obtained when CAMOPT was run on the test corpus, $\mathcal{C}$, defined in Table 1.4. In these tables, the abbreviation 'CO' is used to denote the CAMOPT algorithm. Of the 26 versions of the algorithm tested in the evaluation described in section 3.6 above, the ones that achieved the best spelling accuracy were CAM01A and CAM01D (see Tables 3.11 and 3.12 above). The results for these algorithms are therefore repeated in Tables 3.19 and 3.20 in order to aid comparison with those for CAMOPT.

As can be seen in Table 3.21, CAMOPT made about 8% fewer errors than CAM01A and CAM01D over the test corpus $\mathcal{C}$. In Table 3.20 it can be seen that the style dependence of

```
CamOpt(MidiList)
1   MidiListSize ← |MidiList|
2   RetSegStart ← 0
3   PNCList ← ⟨ ⟩
4   MajorScale ← ⟨2, 2, 1, 2, 2, 2, 1⟩
5   NatMinScale ← ⟨2, 1, 2, 2, 1, 2, 2⟩
6   DescMelMinScale ← ⟨2, 1, 2, 2, 1, 2, 2⟩
7   AscMelMinScale ← ⟨2, 1, 2, 2, 2, 2, 1⟩
8   HarmMinScale ← ⟨2, 1, 2, 2, 1, 3, 1⟩
9   BlendedModTable ← ComputeBlendedModTable( 0.4,
                                               ⟨ ⟨MajorScale, 4⟩ ,
                                                 ⟨NatMinScale, 1⟩ ,
                                                 ⟨DescMelMinScale, 1⟩ ,
                                                 ⟨AscMelMinScale, 1⟩ ,
                                                 ⟨HarmMinScale, 2⟩⟩)
10  while |PNCList| < MidiListSize
11      ▶ Find WinStart and PreMIDISet.
12      WinStart ← RetSegStart
13      PreMIDISet ← ⟨ ⟩
14      while (WinStart > 0) ∧ (|PreMIDISet| < 4)
15          WinStart ← WinStart − 1
16          if MidiList[WinStart] ∉ PreMIDISet
17              PreMIDISet ← ⟨MidiList[WinStart]⟩ ⊕ PreMIDISet
18      ▶ Now find WinEnd and WinMIDISet.
19      WinEnd ← RetSegStart
20      WinMIDISet ← PreMIDISet
21      while (WinEnd < MidiListSize) ∧ (|WinMIDISet| ≤ 12)
22          if MidiList[WinEnd] ∉ WinMIDISet
23              WinMIDISet ← WinMIDISet ⊕ ⟨MidiList[WinEnd]⟩
24          WinEnd ← WinEnd + 1
25      if |WinMIDISet| > 12
26          WinEnd ← WinEnd − 1
27          WinMIDISet ← WinMIDISet[0, 12]
28      ▶ If WinMIDISet is too small, extend PreMIDISet and WinStart backwards,
        ▶ remembering to update both PreMIDISet and WinMIDISet.
29      while (|WinMIDISet| < 12) ∧ (WinStart > 0)
30          WinStart ← WinStart − 1
31          if MidiList[WinStart] ∉ PreMIDISet
32              PreMIDISet ← ⟨MidiList[WinStart]⟩ ⊕ PreMIDISet
33          if MidiList[WinStart] ∉ WinMIDISet
34              WinMIDISet ← ⟨MidiList[WinStart]⟩ ⊕ WinMIDISet
35      ▶ Now find RetSegEnd and PostMIDISet.
36      RetSegEnd ← WinEnd
37      PostMIDISet ← ⟨ ⟩
38      if WinEnd ≠ MidiListSize
39          while (RetSegEnd > 0) ∧ (|PostMIDISet| < 4)
40              RetSegEnd ← RetSegEnd − 1
41              if MidiList[RetSegEnd] ∉ PostMIDISet
42                  PostMIDISet ← ⟨MidiList[RetSegEnd]⟩ ⊕ PostMIDISet
43      ▶ Now find PrePNCList.
44      PrePNCList ← ⟨ ⟩
45      LocalPreMidiSet ← ⟨ ⟩
46      i ← RetSegStart − 1
47      while |PrePNCList| ≠ |PreMIDISet|
48          if MidiList[i] ∉ LocalPreMidiSet
49              PrePNCList ← ⟨PNCList[i]⟩ ⊕ PrePNCList
50              LocalPreMidiSet ← ⟨MidiList[i]⟩ ⊕ LocalPreMidiSet
51          i ← i − 1
52      ▶ Now remove PreMIDISet from WinMIDISet to get NewWinMIDISet.
53      NewWinMIDISet ← ⟨ ⟩
54      for i ← 0 to |WinMIDISet| − 1
55          if WinMIDISet[i] ∉ PreMIDISet
56              NewWinMIDISet ← NewWinMIDISet ⊕ ⟨WinMIDISet[i]⟩
57      ▶ Now append best spelling for this retained segment to PNCList.
58      RetSeg ← MidiList[RetSegStart, RetSegEnd]
59      PNCList ← PNCList ⊕ SpellWinOpt(PrePNCList, PreMIDISet, NewWinMIDISet,
                                        RetSeg, BlendedModTable)
60      ▶ Finally, set RetSegStart to equal start position of next retained segment.
61      RetSegStart ← RetSegEnd
62  return PNCList
```

Figure 3.52: The CamOpt algorithm.

```
SpellWinOpt(PrePNCList, PreMIDISet, NewWinMIDISet, RetSeg, BlendedModTable)
1   NewWinMIDISetSize ← |NewWinMIDISet|
2   FlatNatPNCs ← ⟨"Cn", "Df", "Dn", "Ef", "En", "Fn", "Gf", "Gn", "Af", "An", "Bf", "Bn"⟩
3   BestSpelling ← ⊕_{k=0}^{|NewWinMIDISet|−1} ⟨FlatNatPNCs[NewWinMIDISet[k] mod 12]⟩
4   if (ComputeSpellPen03F(PrePNCList ⊕ BestSpelling, BlendedModTable) ≠ 0)
5      SharpNatPNCs ← ⟨"Cn", "Cs", "Dn", "Ds", "En", "Fn", "Fs", "Gn", "Gs", "An", "As", "Bn"⟩
6      BestSpelling ← ⊕_{k=0}^{|NewWinMIDISet|−1} ⟨SharpNatPNCs[NewWinMIDISet[k] mod 12]⟩
7      if (ComputeSpellPen03F(PrePNCList ⊕ BestSpelling, BlendedModTable) ≠ 0)
8         MaxBitVecInt ← 2^NewWinMIDISetSize − 1
9         BestSpelling ← ⟨⟩
10        for i ← 0 to MaxBitVecInt
11           BitVec ← BitVector(i, NewWinMIDISetSize)
12           Spelling ← ComputeSpelling(NewWinMIDISet, BitVec, flatside)
13           SpellingPenalty ← ComputeSpellPen03F(PrePNCList ⊕ Spelling, BlendedModTable)
14           if (BestSpelling = ⟨⟩) ∨ (SpellingPenalty < BestPenalty)
15              BestSpelling ← Spelling
16              BestPenalty ← SpellingPenalty
17           Spelling ← ComputeSpelling(NewWinMIDISet, BitVec, sharpside)
18           SpellingPenalty ← ComputeSpellPen03F(PrePNCList ⊕ Spelling, BlendedModTable)
19           if SpellingPenalty < BestPenalty
20              BestSpelling ← Spelling
21              BestPenalty ← SpellingPenalty
22  ▶ Now spell pitches in retained segment and return it.
23  SpeltRetSeg ← ⟨⟩
24  for i ← 0 to |RetSeg| − 1
25     PNC ← (PrePNCList ⊕ BestSpelling)[Pos(RetSeg[i], PreMIDISet ⊕ NewWinMIDISet)]
26     SpeltRetSeg ← SpeltRetSeg ⊕ ⟨PNC⟩
27  return SpeltRetSeg
```

Figure 3.53: The SpellWinOpt algorithm.

CamOpt (0.47) was only very slightly more than that of Cam01A (0.46) and less than that of Cam01D (0.50).

Table 3.22 shows the time taken by my implementation of CamOpt to process the test corpus, $\mathcal{C}$. The running times of Cam01A and Cam01D are also shown for comparison. As can be seen in this table and Table 3.23, changing from Cam01A and Cam01D to CamOpt reduced the speed of the algorithm by around 85%. This figure is very close to the 83.26% decrease in speed observed when changing from Cam03A to Cam03D (see Row 1 in Table 3.16). This suggests that most of the decrease in speed observed when changing from Cam01A and Cam01D to CamOpt is due to the increase in window size from 9 to 12.

Algorithm	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
CO	225	242	138	126	473	156	176	139	1675
C1A	283	232	141	128	476	200	172	190	1822
C1D	303	295	131	125	476	161	168	163	1822

Table 3.19: Note error counts for algorithms in column labelled "Algorithm" for the complete test corpus (last column), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus. The algorithms are sorted into increasing order of overall note error count. See text and sections 1.3.4 and 1.3.6 for further details.

Algorithm	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\overline{\text{NA}}$	SD_{Sty}
CO	99.08	99.01	99.44	99.49	98.07	99.36	99.28	99.43	99.15	99.15	0.47
C1A	98.85	99.05	99.42	99.48	98.06	99.18	99.30	99.22	99.07	99.07	0.46
C1D	98.76	98.80	99.47	99.49	98.06	99.34	99.31	99.33	99.07	99.07	0.50

Table 3.20: Note accuracies expressed as percentages for each algorithm in the column headed "Algorithm" for the complete test corpus (column 10), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The columns headed $\overline{\text{NA}}$ and SD_{Sty} give the mean and standard deviation, respectively, of the values in columns 2 to 9. The algorithms are sorted into decreasing order of overall note accuracy. See text and sections 1.3.4 and 1.3.6 for further details.

	CO	C1A	C1D
CO	0.00	8.78	8.78
C1A	-8.07	0.00	0.00
C1D	-8.07	0.00	0.00

Table 3.21: Each entry in this table gives the percentage increase in the note error rate caused by changing from the algorithm at the head of the entry's row to the algorithm at the head of the entry's column. A negative value means that the note error rate was reduced by the change in algorithm. See text and section 1.3.4 for further details.

Algorithm	Total time (hours)	Mean time per note (ms)	Mean speed (notes/sec)
C1A	6.180	113.535	8.808
C1D	6.293	115.605	8.650
CO	41.703	766.085	1.305

Table 3.22: The column headed 'Total time (hours)' gives the total time in hours taken by each algorithm to process the test corpus, $\mathcal{C}$, defined in Table 1.4. Columns 3 and 4 give, respectively, the average time taken per note in ms and the average speed in notes per second over $\mathcal{C}$ for each algorithm. The algorithms are sorted by speed with the fastest at the top.

	C1A	C1D	CO
C1A	0.00	-1.79	-85.18
C1D	1.83	0.00	-84.91
CO	574.94	562.84	0.00

Table 3.23: Each entry in this table gives the percentage increase in speed caused by changing from the algorithm at the head of the entry's row to the algorithm at the head of the entry's column.

3.8 Summary and conclusions

This chapter has been devoted to the analysis, evaluation and improvement of the pitch spelling algorithms proposed by Cambouropoulos (1996, 1998, 2001, 2003). In his publications, Cambouropoulos has described three versions of his pitch spelling algorithm, which, although different in detail, have certain basic features in common. In particular,

1. each uses an overlapping windowing technique;
2. each one searches through a set of spellings for each window and chooses the spelling that achieves the least overall penalty score;
3. in each algorithm, the overall penalty score for a given window spelling is determined by adding together the individual penalty values for the intervals between the pitch name classes in the window spelling; and
4. each algorithm uses implementations of Cambouropoulos's principles of 'interval optimization' and 'notational parsimony' to compute the penalty value for each interval in a window spelling.

As no source code was available for these algorithms, I developed my own implementations based on Cambouropoulos's published descriptions and on information obtained directly from Cambouropoulos in personal communications. The complete pseudocode for these implementations has been provided in this chapter and analysed in detail. This process of analysis and implementation uncovered a number of errors and problems. First, in the version of the algorithm described by Cambouropoulos (2001, 2003), the interval optimization penalty for each interval is determined by looking up the modality class of the interval in Table 3.1 which contains an error (d3 and A6 should be in class D, not class C). Second, the windowing process described by Cambouropoulos (1996, p. 245) was shown to be unworkable and therefore replaced in my implementation by the process described by Cambouropoulos (2003, p. 420). Third, a number of problems were identified in Cambouropoulos's (1996, pp. 233–240) *General Pitch Interval Representation* (*GPIR*) which is used in the earliest version of his algorithm. In particular, it was shown that, in *GPIR*, the representations of certain exotic intervals are the same as those for certain much more common intervals. For example, `"rma2"` and `"rddddddddddd2"` have the same representation within *GPIR*. In other words, the way that intervals are represented in *GPIR* is not isomorphic to the tonal pitch interval naming system. Expressing the algorithms as pseudocode also allowed for the worst-case running time and space complexity of each algorithm to be determined.

I then identified 29 *variable features* of Cambouropoulos's algorithm—that is, 29 ways in which two runs of Cambouropoulos's algorithm on the same data can differ (see Table 3.8). Some of these features are relevant to all versions of the algorithm (e.g., window size, how the notes are sorted in the input data, whether the spelling for each window is constrained to being within either the flatside or sharpside region). However, some variable features are only relevant if other variable features take certain values. For example, one only specifies the penalties assigned to the modality classes if the interval optimisation penalty for each interval depends on the modality class to which it belongs. Most of the 29 variable features are implied

by the differences between the versions of the algorithm described by Cambouropoulos in his publications. Others are modifications to the algorithm that Cambouropoulos suggests but does not implement (e.g., incorporating a method for analysing metric structure). Others are features that I myself have proposed (e.g., using a pitch interval representation that is strictly isomorphic to the tonal pitch interval naming system).

I then carried out an evaluation of these variable features with the goal of identifying the combination of values for these features that achieves the best spelling accuracy in a reasonable time. There was insufficient time for me to explore thoroughly all 29 variable features. I therefore focused on 18 features that required only relatively small modifications to be made to the three versions of the algorithm described by Cambouropoulos (1996, 1998, 2001, 2003). In this evaluation, I ran 26 versions of Cambouropoulos's algorithm on the test corpus, $\mathcal{C}$, defined in Table 1.4. The individual effects on spelling accuracy and running time caused by changing the value of a particular variable feature were then explored by comparing the spelling accuracies and running times achieved by pairs of algorithms that differed only with respect to that variable feature.

The results obtained in this evaluation were then used to predict the combination of values for the 18 variable features explored that would achieve the best spelling accuracy in a reasonable running time. This combination of values is shown in Table 3.18 and was used to design a new version of Cambouropoulos's algorithm, CamOpt. CamOpt is the same as my implementation of the algorithm described by Cambouropoulos (2001), except that

1. it uses a window containing 12 distinct MIDI note numbers rather than 9;

2. it uses the interval optimization and notational parsimony penalty values used in my implementation of the algorithm described by Cambouropoulos (2003); and

3. the modality class of each interval is determined in the same way as in my implementation of the algorithm described by Cambouropoulos (1996, 1998), except that the boundaries of the modality classes are moved so that modality classes A and C cover a wider range of values and B covers a narrower range.

When CamOpt was run on the test corpus $\mathcal{C}$, it made 8% fewer errors than the most accurate of the other versions of the algorithm tested by myself and Cambouropoulos. CamOpt spelt 99.15% of the notes in the test corpus correctly. Also, the style dependence of CamOpt (0.47) was only 0.01 greater than the value achieved by the most accurate of the other versions tested.

Chapter 4

Temperley and Sleator's pitch spelling algorithm

4.1 Introduction

Temperley (2001) presents a computational theory of music cognition that is deeply influenced by Lerdahl and Jackendoff's (1983) *A Generative Theory of Tonal Music* (GTTM). Like Lerdahl and Jackendoff, Temperley attempts to explain the cognition of common-practice music by means of a system that generates structural descriptions from musical "surfaces". As in GTTM, the hypothesis underlying Temperley's theory is that the analysis it generates for a passage of common-practice music correctly describes certain aspects of how the passage is interpreted by listeners who are experienced in the idiom.

Like GTTM, Temperley's theory consists of a number of *preference rule systems*, each containing *well-formedness rules* that define a class of structural descriptions, and *preference rules* that specify an optimal structural description for a given input. Temperley presents preference rule systems for six aspects of musical structure: metre, phrasing, counterpoint, harmony, key and pitch spelling.

In collaboration with Daniel Sleator, Temperley has implemented most of his theory in a suite of computer programs called *Melisma*.[1] These programs take "note list" representations as input (Temperley, 2001, pp. 9–12) in which the pitch of each note (or sequence of tied notes) is represented by its MIDI note number and its onset-time and offset-time are given in milliseconds.

In Temperley's theory, the *tonal pitch class* (TPC) of a note is an integer that indicates the position of the pitch name class of the note on the line of fifths (Temperley, 2001, pp. 118, 123–125). Temperley defines the TPC of `"Cn"` to be 2, so the TPC of `"Gn"` is 3, the TPC of `"Bf"` is 0 and so on (see Figure 4.1). Temperley's concept of tonal pitch class is therefore essentially the same as Longuet-Higgins's (1987a, p. 111) concept of *sharpness* and Regener's (1973, p. 33) concept of *quint* (see section 2.1). Temperley also defines the *neutral pitch class* of a note to be the least positive residue modulo 12 of its MIDI note number (Temperley, 2001, p. 125). Temperley's term "neutral pitch class" is therefore synonymous with the term "pitch class" defined in section 1.4.5 above.

[1] Available online at <`http://www.link.cs.cmu.edu/music-analysis/`>.

Pitch name class	···	F♭	C♭	G♭	D♭	A♭	E♭	B♭	F	C	G	D	A	E	B	F♯	C♯	G♯	D♯	A♯	E♯	B♯	···
TPC	···	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	···
NPC	···	4	11	6	1	8	3	10	5	0	7	2	9	4	11	6	1	8	3	10	5	0	···

Figure 4.1: A segment of the 'line of fifths' showing the tonal pitch class (TPC) and neutral pitch class (NPC) associated with each pitch name class.

Temperley therefore sees the problem of pitch spelling to be that of determining the tonal pitch classes (TPCs) of the notes in a passage when given the MIDI note number, onset time and offset time of each note. The full pitch name of each note can then be computed from its MIDI note number and TPC using the algorithm in Figure 1.17.

Temperley's preference rule system for pitch spelling consists of just three preference rules (Temperley, 2001, pp. 124–132). However, the third of these rules (TPR 3)—the so-called "Harmonic Feedback Rule"—states that the system should "prefer TPC representations which result in good harmonic representations" (Temperley, 2001, p. 131). Roughly speaking, a "good" harmonic representation of a chord is one in which the notes are spelt as they are in the common chord types found in tonal music (i.e., the major triad, minor triad, dominant seventh, diminished seventh etc.) For example, if the three neutral pitch classes 6, 10 and 1 occur simultaneously, then spelling these as `"Fs"`, `"As"` and `"Cs"`, respectively, would result in a "good" harmonic representation; whereas spelling them as `"Fs"`, `"Bf"` and `"Cs"` would not.

Temperley formally defines the concept of a good harmonic representation in his first harmonic preference rule (HPR 1)—the so-called "Compatibility Rule"—which states that the root of a chord should, if possible, be chosen so that the pitch interval name classes from the root to the other notes in the chord are, in decreasing order of preference, [`"p1"`], [`"rp5"`], [`"rma3"`], [`"rmi3"`], [`"rmi7"`], [`"rd5"`] and [`"rmi2"`] (Temperley, 2001, p. 149). If the interval from the root to a note in the chord is not in one of these pitch interval name classes, then it is considered to be "ornamental".

The Harmonic Feedback Rule (TPR 3) implies that the TPC assigned to a note in a chord depends on the root of the chord, while the Compatibility Rule (HPR 1) implies that the root of a chord depends on the TPCs assigned to the notes in the chord. Strictly speaking, therefore, the roots of chords and the TPCs of the notes have to be determined together in order to avoid circularity. Temperley's theories of pitch spelling and harmony are therefore interdependent and this is reflected in the fact that they are both implemented in the `harmony` program in his and Sleator's *Melisma* system.

The complexity of Temperley and Sleator's pitch spelling algorithm is increased still further by the fact that the preference rule system for harmonic structure depends upon that for metrical structure. Specifically, the second harmonic preference rule states that the system should "prefer chord spans that start on strong beats of the meter" (Temperley, 2001, pp. 151, 359) and, according to the fourth harmonic preference rule, a note is more likely to be an ornamental dissonance if it starts on a weak beat (Temperley, 2001, p. 152–154, 359). This implies that the system for generating a harmonic analysis in Temperley's theory requires information about the metrical structure of the passage to be analysed. This is reflected in the fact that the `harmony` program requires not only a "note list" as input but also a representation of the metrical structure

of the passage to be analysed in the form of a "beat list" of the type generated by Temperley and Sleator's `meter` program (another component of *Melisma*).

Consequently, if one wishes to use Temperley's theory to determine the pitch names of the notes in a passage of tonal music, given only the MIDI note number, onset time and offset time of each note, then, strictly speaking, one has to use not only his theory of pitch spelling but also his theories of metrical structure and harmonic structure. More concretely, given a "note list" in which the MIDI note number, onset time and offset time of each note is given, one must first process this note list using the `meter` program to generate a beat list and then process both the note list and this beat list using the `harmony` program to compute the pitch names of the notes in the passage.

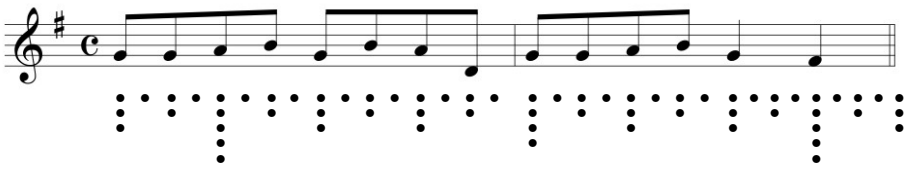
In the next three sections, I review Temperley's preference rule systems for metrical structure, pitch spelling and harmonic structure. Then, in section 4.5, I present an overview and discussion of Sleator and Temperley's `meter` and `harmony` programs. In section 4.6, I briefly discuss the results of Temperley's own evaluation of his pitch spelling method. Then, in section 4.7, I give an account of a more thorough evaluation in which various versions of Sleator and Temperley's own implementation of Temperley's theory are compared with my own implementations of parts of the theory. Finally, in section 4.8, I summarise the main results and conclusions of this chapter.

4.2 Temperley's preference rule system for metrical structure

In Temperley's preference rule system for metrical structure, it is assumed that the passage of music to be analysed has been represented in the form of a *note list* in which the MIDI note number, onset time and offset time of each note (or sequence of tied notes) in the passage is given. It is also assumed that the onset times and offset times are given in milliseconds (Temperley, 2001, pp. 9–12, 30). Such a note list can be automatically derived from an encoding of either a score or a performance. If it is derived from an encoding of a score, then the note onset and offset times will be strictly proportional to their notated values. However, if it is derived from an encoding of a performance (e.g., a MIDI file generated from a performance on a MIDI-enabled instrument), then the note onset and offset times will not, in general, be strictly proportional to their notated values, as a result of unintentional and expressive temporal deviations. Temperley's preference rule systems are designed to accept note lists derived from both scores and performances.

When given such a note list as input, Temperley's preference rule system for metrical structure generates a *beat list* representing a certain predetermined number, N_{Levels}, of contiguous levels in a metrical structure. Specifically, Temperley (2001, p. 37) assumes that his system will generate 5 contiguous metrical levels including the tactus level together with two levels above and below the tactus. The hypothesis underlying the system is that the beat list that it generates for any given note list correctly represents the metrical structure that an experienced listener perceives when he or she hears the music represented by the note list. This beat list gives, for each beat in the lowest metrical level represented, the time at which the beat occurs together with an integer between 0 and $N_{\text{Levels}} - 1$, which I shall call the *beat strength*, indicating the highest level in which the beat occurs (with 0 indicating the lowest metrical level) (Sleator and Temperley, ndb). The higher the highest level in which a beat occurs, the stronger it is. For

(a)



(b)

Note	0	250	67
Note	250	500	67
Note	500	750	69
Note	750	1000	71
Note	1000	1250	67
Note	1250	1500	71
Note	1500	1750	69
Note	1750	2000	62
Note	2000	2250	67
Note	2250	2500	67
Note	2500	2750	69
Note	2750	3000	71
Note	3000	3500	67
Note	3500	4000	66

(c)

Beat	0	2
Beat	105	0
Beat	245	1
Beat	350	0
Beat	490	4
Beat	595	0
Beat	735	1
Beat	875	0
Beat	1015	2
Beat	1120	0
Beat	1260	1
Beat	1365	0
Beat	1505	2
Beat	1610	0
Beat	1750	1
Beat	1855	0
Beat	1995	3
Beat	2100	0
Beat	2240	1
Beat	2345	0
Beat	2485	2
Beat	2625	0
Beat	2765	1
Beat	2870	0
Beat	3010	2
Beat	3115	0
Beat	3255	1
Beat	3360	0
Beat	3500	4
Beat	3605	0
Beat	3745	1
Beat	3850	0
Beat	3990	2

Figure 4.2: (c) shows the beat list generated by the `meter` program from the note list in (b) which represents the melody in (a). The beat list in (c) represents the metrical structure indicated by the rows of dots under the staff in (a). The times at which the beats occur in (c) do not correspond exactly to the onset times of the notes in (b) because the `meter` program quantizes the onset times and offset times of the notes to the nearest integer multiple of 35ms. The beat list in (c) differs from the one given by Sleator and Temperley (ndb) because their beat list was generated by running the `meter` program on the complete melody of which (a) is only the opening phrase.

MWFR 1 Every beat at a given level must be a beat at all lower levels.

MWFR 2 Exactly one or two beats at a given level must elapse between each pair of beats at the next level up.

MPR 1 (Event Rule) Prefer a structure that aligns strong beats with event-onsets.

MPR 2 (Length Rule) Prefer a structure that aligns strong beats with onsets of longer events.

MPR 3 (Regularity Rule) Prefer beats at each level to be maximally evenly spaced.

MPR 4 (Grouping Rule) Prefer to locate strong beats near the beginning of groups.

MPR 5 (Duple Bias Rule) Prefer duple over triple relationships between levels.

MPR 6 (Harmony Rule) Prefer to align strong beats with changes in harmony.

MPR 7 (Stress Rule) Prefer to align strong beats with onsets of louder events.

MPR 8 (Linguistic Stress Rule) Prefer to align strong beats with stressed syllables of text.

MPR 9 (Parallelism Rule) Prefer to assign parallel metrical structures to parallel segments. In cases where a pattern is immediately repeated, prefer to place the stronger beat on the first instance of the pattern rather than the second.

Figure 4.3: Temperley's preference rule system for metrical structure. (From Temperley, 2001, pp. 30–39, 48–51, 357–358.)

example, Figure 4.2(c) shows the beat list generated by the `meter` program from the note list in Figure 4.2(b) which represents the melody in Figure 4.2(a). The beat list in Figure 4.2(c) represents the metrical structure indicated by the rows of dots under the staff in Figure 4.2(a). Note that, in this particular example, the metrical structure predicted by the `meter` program is different from that indicated by the time-signature in the score.

As shown in Figure 4.3, Temperley's preference rule system for metrical structure consists of two well-formedness rules, MWFRs 1–2, and nine preference rules, MPRs 1–9 (Temperley, 2001, pp. 30–39, 48–51, 357–358). Most of these rules are borrowed from GTTM (Lerdahl and Jackendoff, 1983, pp. 68–104, 347–348), as shown in Table 4.1. Only MWFRs 1–2 and MPRs 1–5 are implemented in Temperley and Sleator's `meter` program, therefore I shall only consider these rules here.

MWFRs 1–2 and MPRs 1, 2 and 5 are self-explanatory (see Figure 4.3). Temperley's MPR 3 is interesting because it is based on Lerdahl and Jackendoff's (1983, pp. 72, 347) MWFR 4 which states that beats at the tactus level and above must be "equally spaced". Clearly, one cannot expect the beats in any metrical level to be exactly "equally spaced" in a note list derived from a performance. Therefore, in order that his theory of metrical structure could deal with performance-derived note lists, Temperley re-expressed Lerdahl and Jackendoff's MWFR 4 as the preference rule MPR 3 which specifies only that beats at every level should be "maximally evenly spaced" (Temperley, 2001, p. 34–35). Strictly speaking, MPR 4 (Temperley, 2001, p. 38) would seem to imply that Temperley's theory of metrical structure requires information about the grouping structure of the passage to be analysed. However, Temperley (2001, p. 38) claims that he and Sleator implement this rule in the `meter` program in a way that avoids the need for information about grouping structure. Specifically, he claims that they implement this rule by forcing the system to ignore the Length Rule (MPR 2) at the highest metrical level and instead prefer "beat locations which hit the maximum number of event-onsets". This reduces the

Temperley		*GTTM*
MWFR 1	=	MWFR 2
MWFR 2	=	MWFR 3
MPR 1	=	MPR 3
MPR 2	=	MPR 5a
MPR 3	≃	MWFR 4
MPR 4	≃	MPR 2
MPR 5	=	MPR 10
MPR 6	≃	MPR 5f
MPR 7	=	MPR 4
MPR 8		–
MPR 9	≃	MPR 1

Table 4.1: Correspondence between rules in Temperley's theory of metrical structure and those in the metrical structure component of GTTM (see Lerdahl and Jackendoff, 1983, pp. 345–347 and Temperley, 2001, pp. 357–358). The symbol "=" indicates that the two rules are the same; the symbol "≃" indicates that Temperley's rule is a modification of the rule in GTTM; the symbol "–" indicates that there is no equivalent of Temperley's rule in GTTM. (From Meredith, 2002b, p. 289.)

TPR 1 (Pitch Variance Rule) Prefer to label nearby events so that they are close together on the line of fifths.

TPR 2 (Voice-Leading Rule) Given two events that are adjacent in time and a half-step apart in pitch height: if the first event is remote from the current center of gravity, it should be spelled so that it is five steps away from the second on the line of fifths.

TPR 3 (Harmonic Feedback Rule) Prefer TPC representations which result in good harmonic representations.

Figure 4.4: Temperley's preference rule system for pitch spelling. (From Temperley, 2001, pp. 124–132, 359.)

likelihood of the onsets of long notes that typically end phrases being interpreted as downbeats at the highest metrical level. The system is also programmed to place a highest-level beat as early as possible in the music.

4.3 Temperley's preference rule system for pitch spelling

Figure 4.4 shows the three rules that make up Temperley's preference rule system for pitch spelling or, as he calls it, "tonal-pitch-class labeling" (Temperley, 2001, pp. 123–132).

The first of these "tonal-pitch-class preference rules" (TPRs) (Temperley, 2001, p. 125) simply states that notes that are "nearby" in the music should be assigned pitch names that are "close together" on the line of fifths. Temperley (2001, p. 125) claims that this is "the most important" of the TPRs and that "in many cases, this rule is sufficient to ensure the correct spelling of passages". In practice, TPR 1 is implemented by preferring to spell each note so that

HPR 1 (Compatibility Rule) In choosing roots for chord-spans, prefer certain TPC-root relationships over others, in the following order: $\hat{1}$, $\hat{5}$, $\hat{3}$, $\flat\hat{3}$, $\flat\hat{7}$, $\flat\hat{5}$, $\flat\hat{9}$, ornamental. (An ornamental relationship is any relationship besides those listed.)

HPR 2 (Strong Beat Rule) Prefer chord-spans that start on strong beats of the meter.

HPR 3 (Harmonic Variance Rule) Prefer roots that are close to the roots of nearby segments on the line of fifths.

HPR 4 (Ornamental Dissonance Rule) An event is an ornamental dissonance if it does not have a chord-tone relationship to the chosen root. Prefer ornamental dissonances that are (a) closely followed by an event a step or half-step away in pitch-height, and (b) metrically weak.

Figure 4.5: Temperley's preference rule system for harmonic analysis. (From Temperley, 2001, pp. 147–154, 359.)

it is as close as possible to the current "center of gravity" (COG) of the music on the line of fifths (Temperley, 2001, pp. 125–126, 132–133). The COG at a given point in the music is the weighted average of the TPCs of the notes up to that point, each note being weighted for its duration and recency. Note that the basic principle of spelling notes that are nearby in the music so that they are also nearby on the line of fifths is closely related to both the principle expressed in Longuet-Higgins's (1987a, pp. 112–113) Rule 1 (see section 2.2 above) and the line-of-fifths method of computing interval optimization penalties used in the CAM03B and CAM01B versions of Cambouropoulos's algorithm, discussed in sections 3.5.1 and 3.5.2 above (Cambouropoulos, 2001, p. 6; Cambouropoulos, 2003, p. 423).

TPR 2 (Temperley, 2001, pp. 127–130) is designed to account for the way that notes are typically spelt in chromatic scale segments. This rule states that, if two consecutive notes are separated by a semitone and the first is "remote" from the current COG, then the notes should be spelt so that they are a diatonic semitone apart (i.e., 5 steps apart on the line of fifths). Temperley (2001, pp. 130, 133) states that a note should be considered remote from the current COG if it is four or more steps away from it along the line of fifths.

Note that, like Temperley, Longuet-Higgins and Cambouropoulos also felt the need to include in their theories special rules for spelling notes a semitone apart. Longuet-Higgins's Rule 4 is, in effect, a special case of Temperley's TPR 2 which applies only to ascending semitones (see section 2.2). Cambouropoulos's (1996, p. 243) 'tie-breaker' rule also seems to have been primarily motivated by the need to control the spelling of notes separated by a semitone.

Temperley's (2001, pp. 130–132) TPR 3 has already been discussed in section 4.1 above.

4.4 Temperley's preference rule system for harmonic structure

Temperley's preference rule system for harmonic analysis takes as input a note list representing the passage to be analysed, together with a beat list of the type generated by his theory of metrical structure. As output, the system generates a segmentation of the passage into *chord spans*, each chord span being labelled with the TPC of a root. No two consecutive chord spans may have the same root and the root must be constant within each chord span (Temperley, 2001, pp. 147, 150).

Temperley's theory of harmonic structure consists of the four preference rules listed in Fig-

ure 4.5 (Temperley, 2001, pp. 147–154, 359). The first of these rules has already been discussed in section 4.1 above. The second and third rules are self-explanatory. In HPR 4, a note is an ornamental dissonance if the interval from the root to the note's pitch name is not in any of the following pitch interval name classes

$$\{[\texttt{"p1"}], [\texttt{"rp5"}], [\texttt{"rma3"}], [\texttt{"rmi3"}], [\texttt{"rmi7"}], [\texttt{"rd5"}], [\texttt{"rmi2"}]\}.$$

4.5 Sleator and Temperley's `meter` and `harmony` programs

Sleator and Temperley's `meter` program is an implementation of MPRs 1–5 in Temperley's theory of metrical structure (Temperley, 2001, pp. 39–42). As discussed in section 4.2 above, the `meter` program takes a note list as input and computes a beat list (see Figure 4.2). In the `meter` program, the time interval between consecutive tactus-level beats must be within a user-specified range, which Sleator and Temperley propose should be from 400 to 1600 ms (Temperley, 2001, p. 40). This implies that the metrical structure generated by the `meter` program for any given passage of music depends on its tempo.

The `harmony` program is an implementation of Temperley's preference rule systems for harmonic analysis and pitch spelling. As input, this program requires both a note list and a beat list of the type computed by the `meter` program. As output, it generates a segmentation of the passage into chord spans, each chord span being labelled with the TPC of a root. The output of the `harmony` program also predicts the TPC of each note in the passage. Because the output of the `harmony` program depends on the beat list with which it is provided as input, the pitch names predicted by the program may depend on the tempo of the input passage. The extent to which this is so was explored in my own evaluation of the program, described in section 4.7 below.

Temperley's (2001, pp. 51, 358) MPR 6 states that strong beats should preferably be aligned with changes in harmony (see Figure 4.3) and he found that a failure to take harmonic rhythm into account caused the `meter` program to generate "out of phase" metrical structures in certain cases (Temperley, 2001, p. 45). In an attempt to remedy this, Temperley and Sleator experimented with a "two-pass" method (Sleator and Temperley, nda), in which the `meter` program is first used to generate the tactus and lower levels for a passage. This partial metrical structure is then fed to the `harmony` program, run in a special "prechord" mode in which it only estimates the time points at which chord changes occur instead of generating a full-blown harmonic analysis. This list of estimated chord-change time points is then fed back into the `meter` program which uses the information to generate a full metrical structure in the form of a 5-level beat list (Temperley, 2001, pp. 46–47). Temperley found that, in some cases, the metrical structure generated by this two-pass method was more accurate than that generated by the `meter` program alone. Since the harmonic structure and therefore the pitch names predicted by the `harmony` program depend on the metrical structure generated by the `meter` program, one might expect the `harmony` program to predict pitch names more accurately when fed with the output of this "two-pass" method of metrical analysis than when it is provided with the output generated by the `meter` program alone. The degree to which this is so was explored in my own evaluation described in section 4.7 below.

4.6 Temperley and Sleator's own evaluation of their pitch spelling method

Temperley and Sleator tested their pitch spelling method by running the `meter` and `harmony` programs on a test corpus consisting of 46 excerpts from Kostka and Payne's (1995) theory workbook (see section 1.3.3.2 and Table 1.2 above for details). Temperley (2001, p. 136) reports that the system spelt 98.8% of the 8747 notes in this corpus correctly (that is, it made 103 errors). Temperley (2001, p. 135) found sudden enharmonic changes (see section 1.3.4.3) in three of the excerpts in his test corpus and chose to compare the output of his system with modified versions of these excerpts in which the notation aims to represent the tonal relationships heard by the listener even if this implies using extreme keys. It would have been interesting if he had also measured the note accuracy by comparing the output of his system with the spellings in the original scores of these excerpts.

4.7 A more thorough evaluation of Temperley and Sleator's pitch spelling method

In this section, I describe the procedure that I used to evaluate Temperley and Sleator's pitch spelling method and present the results obtained.

4.7.1 Evaluation of the `meter` and `harmony` programs

As described in section 1.3.3.3, in order to run the `meter` and `harmony` programs on the test corpus $\mathcal{C}$, each OPNDV file in this corpus had to be converted into a note list of the form accepted by the `meter` program. To do this, a suitable tempo had to be selected for each movement in the test corpus.

In fact, in order to test the effect of tempo on note accuracy, six different note list representations were generated for each OPNDV file in the test corpus $\mathcal{C}$:

1. one at what I considered to be a "natural" tempo for that particular movement;
2. one at twice the natural tempo (i.e., all onsets and durations in the "natural" version divided by 2);
3. one at four times the natural tempo;
4. one at half the natural tempo (i.e., all onsets and durations in the "natural" version multiplied by 2);
5. one at one quarter of the natural tempo; and
6. one at one sixth of the natural tempo.

This resulted in six versions of the test corpus, one containing all the movements at their natural tempi, one containing all the movements at twice their natural tempi, and so on.

In order to test the effect on pitch spelling of implementing MPR 6 using the "two-pass" method described in section 4.5 above, two versions of the `meter-harmony` system were run on

all six versions of the test corpus just described. In the first version of the system, the `meter` program was first run on the input data using the default values for all parameters supplied internally by the program. The output of this step was then fed into the `harmony` program, run with the parameter values defined in the `parameters` file for the `harmony` program supplied in the 2003 edition of *Melisma*.[2] This version of the system is the one that is run when one uses the `met-harm` script supplied with *Melisma*. The second version of the `meter-harmony` system tested was the one that is run when one uses the `met-harm-two-pass` script supplied with *Melisma*. This version of the system consists of four steps, as follows.

1. The input data is fed into the `meter` program run with the `parameters.prechord` file which restricts the program to generating only the tactus metric level and below.

2. The output of the first step is fed into the `harmony` program run with its `parameters.prechord` file which restricts the program to estimating the chord-change time points.

3. The output of the second step is fed back into the `meter` program, run with default parameter values which generates a full 5-level beat list.

4. The output of the third step is fed into the `harmony` program, run with default parameter values, which generates the full harmonic analysis including a predicted pitch name for each note.

Tables 4.2 and 4.3 summarise the results obtained when the `met-harm` and `met-harm-two-pass` scripts were run on the 6 versions of the test corpus described above.

Each entry in the first column of Tables 4.2 and 4.3 is a code which indicates the version of the `meter-harmony` system used (i.e., either the `met-harm` script or the `met-harm-two-pass` script) and the version of the test corpus on which it was run. Each of these codes either has the format "MH*ab*" or the format "MH2P*ab*" where *a* is empty, 'X' or 'D' and *b* is empty, '2', '4' or '6'. A code in the format "MH*ab*" indicates that the `met-harm` script was used; whereas, a code in the format "MH2P*ab*" indicates that the `met-harm-two-pass` script was used. If *ab* is empty, this indicates the system being run on the version of the test corpus in which each movement is at its "natural" tempo. If *a* is 'X', this indicates the version of the corpus in which the onsets and durations are *b* times those in the "natural" tempo version of the corpus. If *a* is 'D', this indicates the version of the corpus in which the onsets and durations are $\frac{1}{b}$ times those in the natural tempo version. For example, "MH2PX4" signifies the `met-harm-two-pass` script, run on the version of the corpus in which each movement is at $\frac{1}{4}$ of its "natural" tempo (i.e., the onsets and the durations are 4 times those in the natural tempo version). Similarly, "MHD2" signifies the `met-harm` script run on the version of the corpus in which each movement is at twice its natural tempo (i.e., the onsets and durations have been obtained by dividing the corresponding values in the natural tempo version by 2).

Figure 4.6 represents graphically the information in the last column of Table 4.2 (ignore the values in parentheses in this column—they will be discussed below). This graph clearly

[2]The version of *Melisma* used was that supplied in the file <`ftp://ftp.cs.cmu.edu/usr/ftp/usr/sleator/melisma2003.tar.gz`>.

Algorithm	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
MH2PX2	30	815	5	17	3229 (267)	170	26	33	4325 (1363)
MHX2	31	831	4	21	3252 (290)	163	23	27	4352 (1390)
MH	24	940	2	15	3105 (136)	127	26	149	4388 (1419)
MH2P	29	951	2	16	3110 (141)	140	26	118	4392 (1423)
MH2PX4	76	792	12	34	3573 (629)	301	90	51	4929 (1985)
MHX4	1277	2575	7	216	3563 (622)	279	353	55	8325 (5384)
MHD2	144	2401	311	310	4413 (1659)	989	484	1677	10729 (7975)
MH2PD2	1951	2366	1273	322	4393 (1651)	963	1252	1629	14149 (11407)
MHX6	2112	7406	13	354	4066 (1562)	296	377	1049	15673 (13169)
MH2PX6	2109	5738	21	45	4065 (1561)	10481	110	2048	24617 (22113)
MHD4	1159	9584	2096	1787	8562 (6599)	3168	2396	5082	33834 (31871)
MH2PD4	3905	12679	3584	3405	12170 (10209)	4495	2740	6843	49821 (47860)

Table 4.2: Note error counts for algorithms in column labelled "Algorithm" for the complete test corpus (last column), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus. The algorithms are sorted into increasing order of overall note error count. See text and sections 1.3.4 and 1.3.6 for further details.

Algorithm	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\overline{\text{NA}}$	SD_{Sty}
MH2PX2	99.88	96.67	99.98	99.93	86.82 (98.91)	99.31	99.89	99.87	97.79 (99.30)	97.79 (99.31)	4.57 (1.13)
MHX2	99.87	96.61	99.98	99.91	86.72 (98.82)	99.33	99.91	99.89	97.78 (99.29)	97.78 (99.29)	4.61 (1.16)
MH	99.90	96.16	99.99	99.94	87.32 (99.44)	99.48	99.89	99.39	97.76 (99.28)	97.76 (99.27)	4.41 (1.28)
MH2P	99.88	96.12	99.99	99.93	87.30 (99.42)	99.43	99.89	99.52	97.76 (99.27)	97.76 (99.27)	4.42 (1.30)
MH2PX4	99.69	96.77	99.95	99.86	85.41 (97.43)	98.77	99.63	99.79	97.48 (98.99)	97.48 (98.99)	4.99 (1.23)
MHX4	94.79	89.49	99.97	99.12	85.45 (97.46)	98.86	98.56	99.78	95.75 (97.25)	95.75 (97.25)	5.47 (3.54)
MHD2	99.41	90.20	98.73	98.73	81.98 (93.23)	95.96	98.02	93.15	94.53 (95.93)	94.52 (95.93)	5.99 (3.38)
MH2PD2	92.04	90.34	94.80	98.69	82.06 (93.26)	96.07	94.89	93.35	92.78 (94.18)	92.78 (94.18)	5.01 (2.55)
MHX6	91.38	69.76	99.95	98.56	83.40 (93.62)	98.79	98.46	95.72	92.00 (93.28)	92.00 (93.28)	10.53 (9.95)
MH2PX6	91.39	76.57	99.91	99.82	83.40 (93.63)	57.21	99.55	91.64	87.44 (88.72)	87.44 (88.72)	14.82 (14.86)
MHD4	95.27	60.87	91.44	92.71	65.04 (73.05)	87.07	90.22	79.25	82.74 (83.74)	82.73 (83.73)	13.15 (11.85)
MH2PD4	84.06	48.23	85.37	86.10	50.31 (58.31)	81.65	88.82	72.07	74.58 (75.58)	74.58 (75.58)	16.39 (14.88)

Table 4.3: Note accuracies expressed as percentages for each algorithm in the column headed "Algorithm" for the complete test corpus (column 10), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The columns headed $\overline{\text{NA}}$ and SD_{Sty} give the mean and standard deviation, respectively, of the values in columns 2 to 9. The algorithms are sorted into decreasing order of overall note accuracy. See text and sections 1.3.4 and 1.3.6 for further details.

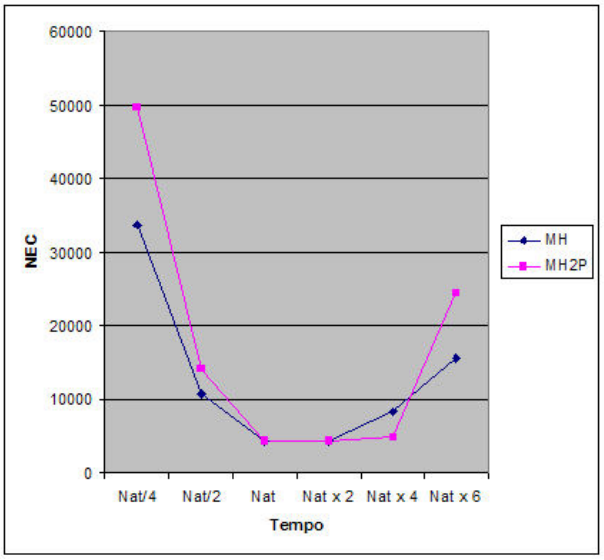


Figure 4.6: This graph shows the note error count (NEC) for the `met-harm` (MH) and `met-harm-two-pass` (MH2P) scripts achieved over the test corpus $\mathcal{C}$, defined in Table 1.4, for each of the six versions of the test corpus. On the horizontal axis, "Nat" signifies the corpus in which each movement is at its natural tempo. "Nat x y" indicates the test corpus in which each duration and onset time is y times its corresponding value in the natural tempo version of the corpus. "Nat/y" indicates the corpus in which each duration and onset time is $1/y$ times its corresponding value in the natural tempo version of the corpus. The tempo therefore decreases from left to right along the horizontal axis.

shows that the spelling accuracies achieved by the `met-harm` and `met-harm-two-pass` scripts are strongly dependent on the tempo of the music being analysed. If the music is more than twice the natural tempo or less than a quarter of the natural tempo, the spelling accuracy is severely reduced for both scripts. The graph suggests that both scripts are affected in a very similar way by tempo. However, while the `met-harm-two-pass` script performs well over a wider range of tempi than the `met-harm` script, its performance also drops off more quickly outside of this range. Nevertheless, when the movements were analysed at their natural tempo and at half-speed, the performance of the `met-harm` and `met-harm-two-pass` systems was almost indistinguishable.

The next observation to be made is that even the most accurate version tested (the `met-harm-two-pass` script run on the corpus in which the movements were all at half speed) only achieved an overall note accuracy of 97.79% on this test corpus. This result is less than that achieved by the most accurate version of Longuet-Higgins's algorithm tested (98.21%) (see Tables 2.2 and 2.3). Moreover, it is almost identical to the note accuracy achieved by the *least* accurate version of Cambouropoulos's algorithm tested (97.76%) (see Tables 3.11 and 3.12).

Figure 4.7 represents graphically the information in the last column of Table 4.3 (again, ignore for the moment the values in parentheses in this column) and shows how the style dependence of the `met-harm` and `met-harm-two-pass` scripts changes with tempo. The fact that the curves in Figure 4.7 have roughly the same 'U' shape as those in Figure 4.6 indicates that both the style dependence and the note error rate of these algorithms depend on tempo in a similar

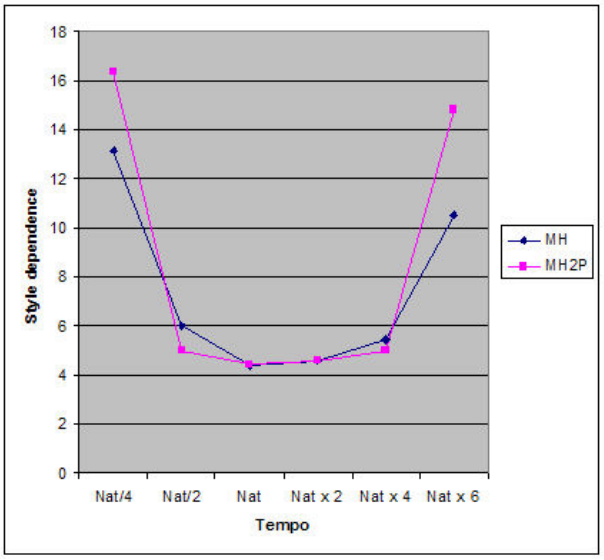


Figure 4.7: This graph shows the style dependence for the `met-harm` (MH) and `met-harm-two-pass` (MH2P) scripts over the test corpus $\mathcal{C}$, defined in Table 1.4, for each of the six versions of the test corpus. The horizontal axis labels have the same meaning as in Figure 4.6.

way. Specifically, the tempi at which the scripts achieve the highest overall note accuracy are also those at which they perform most consistently across the different styles in the test corpus. However, the lowest (i.e., best) value of style dependence achieved by Sleator and Temperley's system was 4.41, which is considerably higher than the best value achieved by Longuet-Higgins's algorithm (1.79) and very much higher than even the worst value obtained using Cambouropoulos's algorithms (0.91) (see Table 3.12).

As discussed in section 1.3.4.3, because of the sudden enharmonic change at bar 166 in the fourth movement of Haydn's Symphony No. 100 in G ('Military') (Hob. I:100), I compare the output of each algorithm with two "correct" versions of this movement: one in which the notes are spelt as they are in the original score; and a second, modified version, in which all the notes in the original score up to bar 165 are transposed down a diminished second. The results obtained when the output of each algorithm was compared with the modified version of this movement are shown in parentheses in Tables 4.2 and 4.3. As is evident from these results, the spellings generated by both the `met-harm` and `met-harm-two-pass` scripts were much more similar to the modified version of the fourth movement of Haydn's 'Military' Symphony than the original score for all six versions of the test corpus used in this evaluation. For example, when the `met-harm` script was run on this movement at its natural tempo, 2978 notes were assigned pitch names different from those in the original score (i.e., 53.35% correct) but only 9 notes were assigned different pitch names from those in the modified version (i.e., 99.86% correct). These results suggest that Temperley and Sleator's system is much more successful at assigning pitch names that correctly represent the perceived tonal relationships between notes than it is at predicting the occurrence of sudden enharmonic changes for notational convenience. Indeed, when the spelling accuracy of the system is measured using the modified spelling of the fourth movement of Haydn's 'Military' Symphony, the overall note accuracy achieved by the four most

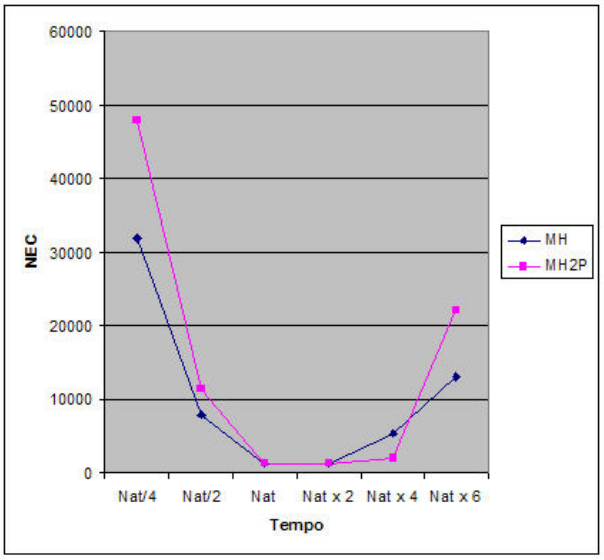


Figure 4.8: This graph shows the note error count (NEC) for the `met-harm` (MH) and `met-harm-two-pass` (MH2P) scripts over the test corpus $\mathcal{C}$, defined in Table 1.4, for each of the six versions of the test corpus using the modified version of the fourth movement of Haydn's 'Military' Symphony as a ground truth instead of the pitch names in the original score. The horizontal axis labels have the same meaning as in Figure 4.6.

accurate versions of Temperley and Sleator's system tested (99.27–99.30%) was better even than that achieved by the best version of Cambouropoulos's algorithm (99.15% for CamOpt) (see Tables 4.3 and 3.20). Also, when the modified spelling of the fourth movement of the 'Military' Symphony was used as a ground truth, the style dependence of the most accurate versions of the algorithm tested were markedly improved. For example, for the `met-harm-two-pass` script, run on the half-speed version of the test corpus (MH2PX2), the style dependence was reduced to 1.13 which is better than the best value achieved using Longuet-Higgins's algorithm (1.79) and comparable with that achieved by the least accurate version of Cambouropoulos's algorithm (0.91). Note also that, when the modified version of the fourth movement of Haydn's 'Military' Symphony was used, the most accurate version of the system tested (MH2PX2) was also the least dependent on style. When the graphs in Figures 4.8 and 4.9 are compared with those in Figures 4.6 and 4.7, respectively, it can be seen that both the spelling accuracy and style dependence of the two scripts, when the modified version of the fourth movement of Haydn's 'Military' Symphony is used, depend on tempo in essentially the same way as when the original score of this movement is used.

4.7.2 How much does pitch spelling depend on metrical structure?

As already discussed in sections 4.2 and 4.5, the `meter` program takes a note list of the type shown in Figure 4.2(b) as input and generates a beat list in the form of Figure 4.2(c). The `harmony` program then uses the metrical structure represented by this beat list to implement HPRs 2 and 4 (see Figure 4.5). Of the three TPRs, only TPR 3 depends in any way on harmonic structure (see Figure 4.4), and only two of the four harmonic preference rules depend on metrical structure. This suggests that pitch spelling is not heavily dependent on metrical structure within

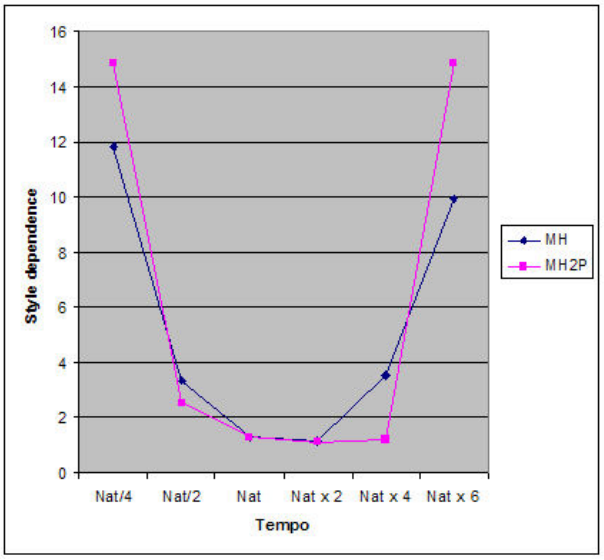


Figure 4.9: This graph shows the style dependence (SD) for the `met-harm` (MH) and `met-harm-two-pass` (MH2P) scripts over the test corpus $\mathcal{C}$, defined in Table 1.4, for each of the six versions of the test corpus using the modified version of the fourth movement of Haydn's 'Military' Symphony as a ground truth instead of the pitch names in the original score. The horizontal axis labels have the same meaning as in Figure 4.6.

Temperley and Sleator's system.

In the evaluation described in the previous section, the `met-harm` and `met-harm-two-pass` scripts both performed best when they were run on the half-speed version of the test corpus (i.e., MHX2 and MH2PX2). To explore the effect of metrical structure on spelling performance in Temperley and Sleator's system, I therefore re-ran the system on the half-speed version of the test corpus with metrical information omitted. As the results for `met-harm` and `met-harm-two-pass` on the half-speed version of the test corpus were very similar (see Table 4.3), only the `met-harm` script was re-run on the data without metrical information in this experiment. To be specific, the half-speed version of the test corpus was run through the `meter` program and then the beat list generated for each movement was modified so that every beat had the same strength (see section 4.2). The `harmony` program was then run on the half-speed version of the test corpus using these modified beat lists in which every beat had the same strength. The procedure was carried out twice: first, with the strength of every beat set to 2; and then again with the strength of every beat set to 0. The results were identical in both cases, indicating that the actual beat-strength assigned is immaterial, provided that all the beats have the same strength. The rows labelled 'HNM' (Harmony-No-Meter) in Tables 4.4 and 4.5 summarise the results obtained. The results for the `met-harm` and `met-harm-two-pass` scripts run on the half-speed version of the corpus are also given in these tables for comparison.

By comparing the results in the row labelled HNM in Table 4.4 with those in the row labelled MHX2, it can be seen that making all the beats have an equal strength (and thereby effectively ignoring metrical information) increased the overall note error count by 24% from 4352 to 5399 when the original score of the fourth movement of Haydn's 'Military' Symphony was used. However, when the modified spelling of this movement was used, the overall note error count

Algorithm	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
MH2PX2	30	815	5	17	3229 (267)	170	26	33	4325 (1363)
MHX2	31	831	4	21	3252 (290)	163	23	27	4352 (1390)
HNM	71	804	9	31	3196 (234)	151	30	1107	5399 (2437)

Table 4.4: Note error counts for algorithms in column labelled "Algorithm" for the complete test corpus (last column), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus. The algorithms are sorted into increasing order of overall note error count. See text and sections 1.3.4 and 1.3.6 for further details.

Algorithm	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\overline{\text{NA}}$	SD_{Sty}
MH2PX2	99.88	96.67	99.98	99.93	86.82 (98.91)	99.31	99.89	99.87	97.79 (99.30)	97.79 (99.31)	4.57 (1.13)
MHX2	99.87	96.61	99.98	99.91	86.72 (98.82)	99.33	99.91	99.89	97.78 (99.29)	97.78 (99.29)	4.61 (1.16)
HNM	99.71	96.72	99.96	99.87	86.95 (99.04)	99.38	99.88	95.48	97.25 (98.76)	97.24 (98.76)	4.49 (1.70)

Table 4.5: Note accuracies expressed as percentages for each algorithm in the column headed "Algorithm" for the complete test corpus (column 10), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The columns headed $\overline{\text{NA}}$ and SD_{Sty} give the mean and standard deviation, respectively, of the values in columns 2 to 9. The algorithms are sorted into decreasing order of overall note accuracy. See text and sections 1.3.4 and 1.3.6 for further details.

was raised by 75% from 1390 to 2437. This suggests that metrical structure plays an important rôle in determining pitch names in Temperley and Sleator's system.

However, a closer examination of the results reveals that most of this increase in the overall note error count is due to the rather large increase in the note error count for Vivaldi from 27 for MHX2 to 1107 for HNM. Moreover, a more detailed examination of the results reveals that 1067 (96%) of the 1107 errors made by the HNM system on the music of Vivaldi occurred in the third movement of his Concerto in G minor ('L'estate') from 'Le Quattro Stagioni' (Op. 8, No. 2, RV 315). Figure 4.10 shows the locations of the notes in this movement spelt incorrectly by the HNM system. In fact, most of the errors in this movement result from the fact that HNM spells all the notes from bar 91 to the end of the movement a diminished second higher than in the original score (see Figure 4.11). This mis-spelling of the passage from bar 91 to the end seems to result from errors made in spelling the long descending chromatic scale that extends from bar 85 to bar 96. Specifically, the HNM system spells the Bs in bar 87 as C♭s and the As in bar 89 as B𝄫s (see Figure 4.11).

It therefore seems that the observed increase in note error count on the test corpus when metrical information was ignored resulted primarily from the system's inability to cope with one particular descending chromatic scale passage which, in turn, caused a whole segment of music to be spelt a diminished second away from its correct spelling. We can therefore tentatively conclude that the use of metrical structure in Temperley and Sleator's system makes it more able to cope with certain types of chromatic passage, which, in turn, makes it less likely to spell whole segments in the wrong key.

4.7.3 Using TPR 1 alone to compute pitch names

4.7.3.1 Introduction

Recall from section 4.3 above that TPR 1 simply states that notes that are "nearby" in the music should be assigned pitch names that are "close together" on the line of fifths (Temperley, 2001, p. 125). Note that TPR 1 depends on neither harmonic nor metrical structure. Temperley (2001, p. 125) claims that this is "the most important" of the TPRs and that "in many cases, this rule is sufficient to ensure the correct spelling of passages". To test this claim, I designed an algorithm which implements just Temperley's TPR 1 alone. This algorithm, which I call TPROne, is shown in Figure 4.13. In practice, TPR 1 is implemented by preferring to spell each note so that it is as close as possible to the current "center of gravity" (COG) of the music on the line of fifths (Temperley, 2001, pp. 125–126, 132–133). The COG at a given point in the music is the weighted average of the TPCs of the notes up to that point, each note being weighted for its duration and recency—that is, the more recent the onset time of a note and the longer it lasts, the more it contributes to the COG. Temperley (2001, p. 126) claims that the "pressure to locate two events close together" on the line of fifths "decays as the events get further apart in time". He goes on to suggest that "roughly speaking, it seems that the pressure is significant for intervals of a few seconds, but decays for longer intervals". Note, however, that, in my implementation of TPR 1, the onset times and durations are assumed to be expressed in relative rather than absolute time units. In the `harmony` program, notes are "weighted for their recency according to an exponential function" (Temperley, 2001, p. 132).

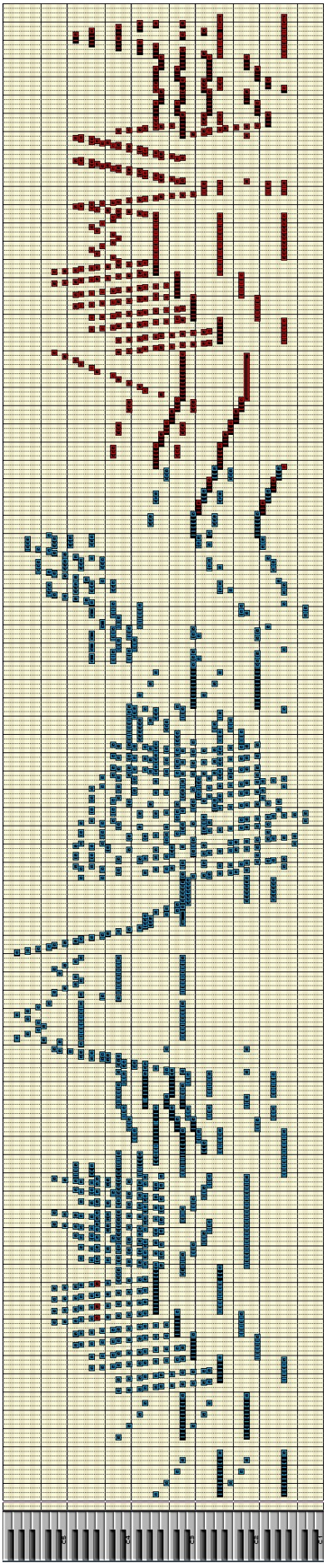


Figure 4.10: Sequencer screenshot showing the notes spelt incorrectly in the third movement of Vivaldi's Concerto in G minor ('L'estate') from 'Le Quattro Stagioni' (Op. 8, No. 2, RV 315) by the HNM system when the music is at half its natural tempo. The incorrectly spelt notes are red, the correctly spelt notes are blue.

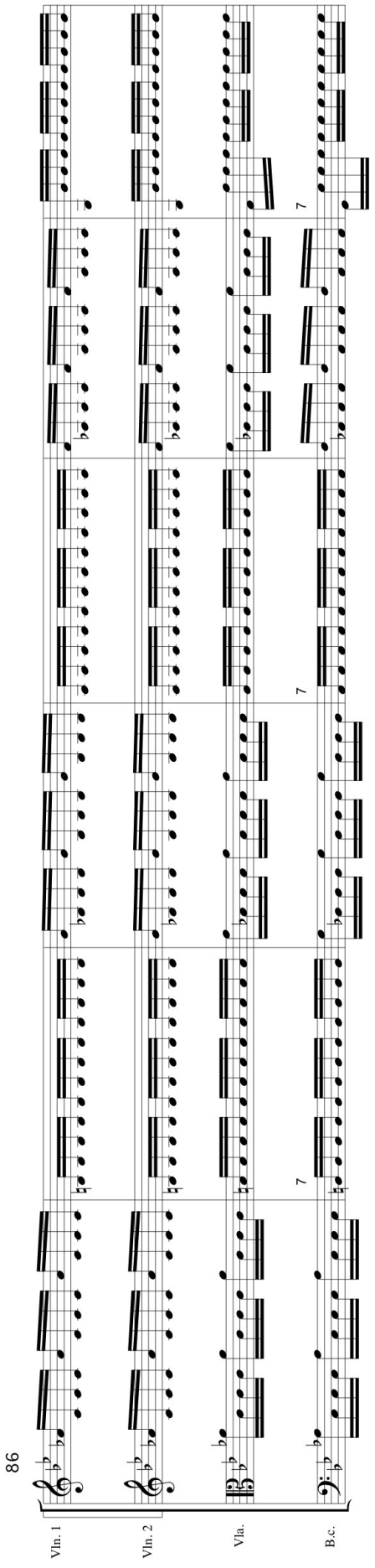


Figure 4.11: Bars 86 to 91 from the third movement of Vivaldi's Concerto in G minor ('L'estate') from 'Le Quattro Stagioni' (Op. 8, No. 2, RV 315).

However, in my implementation, the recency weight of a note is set to be inversely proportional to the difference between the onset time of the note and the time point for which the current COG is being calculated. Figure 4.12 compares various exponential decay functions with the reciprocal function used in TPROne. As can be seen, the exponential weighting function used in the *Melisma* `harmony` program can be made to decay more or less quickly than the reciprocal function used in TPROne simply by adjusting the values of the constants in the formula for the exponential function.

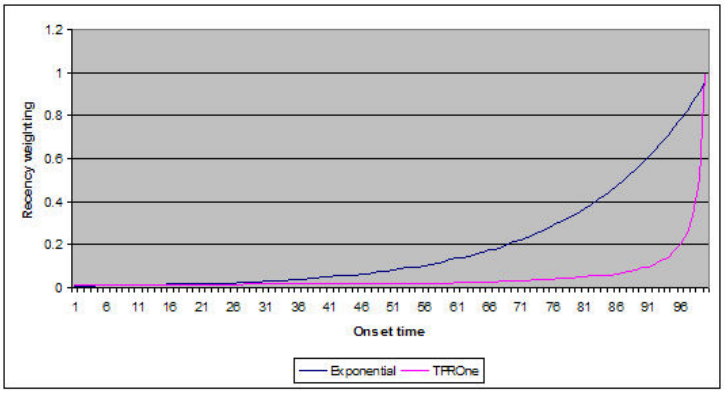
It should be noted that, in their implementation of TPR 1, Temperley and Sleator "arbitrarily limit the possible spellings for pitches to a range of four [actually five] cycles on the line of fifths" (Temperley, 2001, p. 132). In fact, their `harmony` program is limited to generating TPCs between -24 (`"Dffff"`) and 34 (`"Essss"`). In my implementation, this range of permitted TPCs can be set by the user.

Temperley and Sleator also "make a heuristic assumption that the spellings of simultaneous pitches are always within a twelve-step 'window' on the line of fifths" (Temperley, 2001, p. 132). For example, they "assume that A♭ and G♯ will never be present simultaneously". In practice, if a note is always spelt as close as possible to the current COG, and the COG is only updated once for each set of simultaneously starting notes, then two simultaneously starting notes with the same pitch class could only be assigned different TPCs if they were both exactly 6 steps away from the current COG on the line of fifths. This problem can therefore be solved by specifying that notes 6 steps from the current COG should be spelt either always a [`"ra4"`] away from the COG or always a [`"rd5"`] away from the COG. In my implementation, the user may choose between these two possibilities.

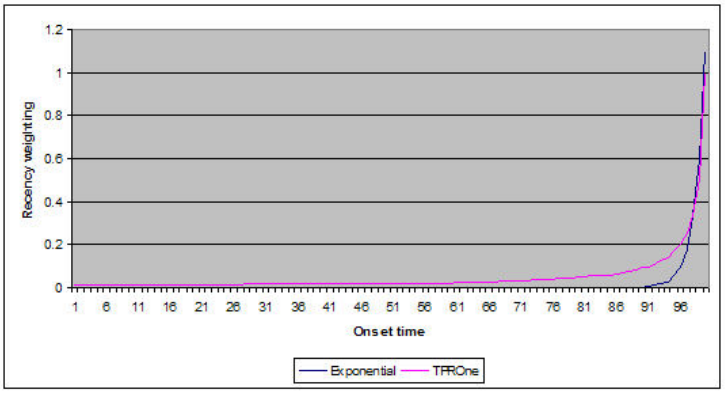
4.7.3.2 The TPROne algorithm

Figure 4.13 shows the TPROne algorithm which is my own implementation of Temperley's TPR 1. This algorithm takes five arguments: **SortedOMDList**, *RA4OrRD5*, *MaxTPC*, *MinTPC* and *WindowSize*. **SortedOMDList** is an ordered set of triples in which each triple, $\langle o, m, d\rangle$, gives the onset time, o, the MIDI note number, m, and the duration, d, of a note or sequence of tied notes in the movement or passage to be analysed. The values of o and d for a note are assumed to be equal to its onset time and duration, respectively, divided by the greatest common divisor of all the note onset times and durations in the movement or passage being processed. *RA4OrRD5* must be equal to either ra4 or rd5. If *RA4OrRD5* = ra4, every note which is six steps away from the current COG on the line of fifths is assigned a pitch name which is a [`"ra4"`] away from the current COG—that is, it is given a TPC which is 6 greater than the COG. If *RA4OrRD5* = rd5, every note which is six steps away from the current COG is assigned a pitch name which is a [`"rd5"`] away from the current COG—that is, it is given a TPC which is 6 less than the COG. It is assumed that **SortedOMDList** has been sorted by increasing onset time, MIDI note number and duration, in that order of preference. The values of the arguments *MaxTPC* and *MinTPC* define the range of TPCs that the algorithm is permitted to assign. The value of *WindowSize* must be either nil or a positive integer. If *WindowSize* = nil, all notes with onset times less than that of the note currently being spelt contribute to the current COG. If *WindowSize* is a positive integer, only the *WindowSize* most recent notes with onset times less

(a) $A = 1$ and $k = 0.05$



(b) $A = 2$ and $k = 0.6$



(c) $A = 1$ and $k = 0.8$

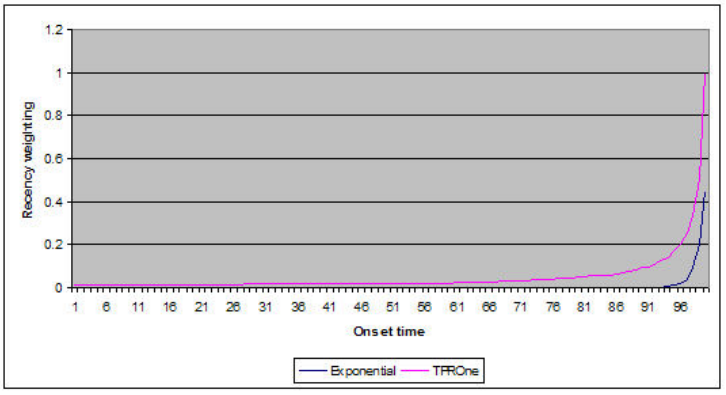


Figure 4.12: A comparison of various exponential decay recency weighting functions with the reciprocal function used in TPROne when the note to be spelt has an onset of 100. If w is recency weighting and t is onset time, then the curve labelled "TPROne" in each graph shows the weighting function used in TPROne. This curve has the formula $w = \frac{1}{100-t}$. Each curve labelled "Exponential" has the form $w = Ae^{-kt}$. However, in (a), $A = 1$ and $k = 0.05$; in (b), $A = 2$ and $k = 0.6$; and, in (c), $A = 1$ and $k = 0.8$. As can be seen, the exponential weighting function used in the *Melisma* `harmony` program can be made to decay more or less quickly than the reciprocal function used in TPROne by adjusting the values of the constants, A and k.

```
TPROne(SortedOMDList, RA4OrRD5, MaxTPC, MinTPC, WindowSize)
1   ChordList ← OMDList2ChordList(SortedOMDList)
2   N_Ch ← |ChordList|
3   OutputOMDTList ← ⟨ ⟩
4   for i ← 0 to N_Ch − 1
5      OMDTListForThisChord ← SpellChord(ChordList[i], OutputOMDTList,
                                         RA4OrRD5, MaxTPC, MinTPC, WindowSize)
6      OutputOMDTList ← OutputOMDTList ⊕ OMDTListForThisChord
7   return OutputOMDTList
```

Figure 4.13: The TPROne algorithm.

```
OMDList2ChordList(SortedOMDList)
1   NumberOfOMDs ← |SortedOMDList|
2   ChordList ← ⟨ ⟩
3   CurrentChord ← ⟨SortedOMDList[0]⟩
4   i ← 1
5   while i < NumberOfOMDs
6      if CurrentChord[0][0] = SortedOMDList[i][0]
7         CurrentChord ← CurrentChord ⊕ ⟨SortedOMDList[i]⟩
8      else
9         ChordList ← ChordList ⊕ ⟨CurrentChord⟩
10        CurrentChord ← ⟨SortedOMDList[i]⟩
11     i ← i + 1
12  ChordList ← ChordList ⊕ ⟨CurrentChord⟩
13  return ChordList
```

Figure 4.14: The OMDList2ChordList algorithm.

than that of the current note contribute to the current COG.

In line 1 of TPROne (see Figure 4.13), **SortedOMDList** is partitioned into ordered sets called *chords* such that

1. all the $\langle o, m, d\rangle$ triples within a chord have the same onset time;
2. no two $\langle o, m, d\rangle$ triples in different chords have the same onset time; and
3. the $\langle o, m, d\rangle$ triples within a chord are sorted by increasing onset time, MIDI note number and duration, in that order of preference.

This step is accomplished using the OMDList2ChordList function defined in Figure 4.14. This function generates an ordered set of chords which is stored in the variable **ChordList**.

Each of the chords in **ChordList** is then spelt in turn using the SpellChord function which is called in line 5 of TPROne and defined in Figure 4.15. In TPROne, the output of SpellChord is first stored in the variable **OMDTListForThisChord** which is an ordered set of 4-tuples in which the ith 4-tuple, $\langle o, m, d, t\rangle$, gives the onset time, o, the MIDI note number, m, the duration, d, and the assigned TPC, t, of the ith $\langle o, m, d\rangle$ triple in the chord given to SpellChord as input. Once **OMDTListForThisChord** has been computed for a chord, it is appended to **OutputOMDTList** in line 6. Once all the chords have been processed, the function returns **OutputOMDTList** which is an ordered set of 4-tuples in which each 4-tuple $\langle o, m, d, t\rangle$ gives the onset time, o, the MIDI note number, m, the duration, d, and the TPC, t, of a note in the input passage. The order of the elements in **OutputOMDTList** corresponds to that in **SortedOMDList**. The pitch name of each note in the input passage can be computed from its MIDI note number and TPC using the algorithm in Figure 1.17.

```
SpellChord(Chord, OMDTList, RA4OrRD5, MaxTPC, MinTPC, WindowSize)
1   COG ← ComputeCOG(Chord, OMDTList, WindowSize)
2   OMDTListForThisChord ← ⟨ ⟩
3   for i ← 0 to |Chord| − 1
4       TPC ← ComputeTPC(Chord[i][1], COG, RA4OrRD5, MaxTPC, MinTPC)
5       OMDTForThisOMD ← Chord[i] ⊕ ⟨TPC⟩
6       OMDTListForThisChord ← OMDTListForThisChord ⊕ ⟨OMDTForThisOMD⟩
7   return OMDTListForThisChord
```

Figure 4.15: The SpellChord algorithm.

```
ComputeCOG(Chord, OMDTList, WindowSize)
1   if OMDTList = ⟨ ⟩
2       COG ← 4
3   else
4       n ← |OMDTList|
5       ThisOnset ← Chord[0][0]
6       if WindowSize ≠ nil
7           WinStart ← max({0, n − WindowSize})
8           Window ← OMDTList[WinStart, n]
9       else
10          Window ← OMDTList
11      w ← |Window|
12      RecencyWeights ← ⨁_{i=0}^{w−1} ⟨1/(ThisOnset − Window[i][0])⟩
13      DurationWeights ← ⨁_{i=0}^{w−1} ⟨Window[i][2]⟩
14      TPCs ← ⨁_{i=0}^{w−1} ⟨Window[i][3]⟩
15      COG ← (Σ_{i=0}^{w−1} (TPCs[i] × DurationWeights[i] × RecencyWeights[i])) / (Σ_{i=0}^{w−1} (DurationWeights[i] × RecencyWeights[i]))
16  return COG
```

Figure 4.16: The ComputeCOG algorithm.

The SpellChord function, which is called in line 5 of TPROne, is defined in Figure 4.15. This function takes six arguments: **Chord**, **OMDTList**, *RA4OrRD5*, *MaxTPC*, *MinTPC* and *WindowSize*. The values of *RA4OrRD5*, *MaxTPC*, *MinTPC* and *WindowSize* are the same as those given by the user as arguments to TPROne (see Figure 4.13). SpellChord computes a TPC for each of the notes in the chord **Chord** and outputs an ordered set of 4-tuples, **OMDTListForThisChord**, in which each of the $\langle o, m, d\rangle$ triples in **Chord** has a TPC appended to it. The first step in SpellChord is to compute the COG, *COG*, for the current chord, **Chord** (line 1). This is done using the function ComputeCOG which is defined in Figure 4.16. Then, for each of the $\langle o, m, d\rangle$ triples in **Chord**, a TPC, *TPC*, is computed in line 4 using the ComputeTPC function which is defined in Figure 4.17. Next, in line 5, an $\langle o, m, d, t\rangle$ 4-tuple is constructed by appending the value of *TPC* to the current $\langle o, m, d\rangle$ triple and this 4-tuple is stored in the variable **OMDTForThisOMD**. In line 6, this new 4-tuple is appended to **OMDTListForThisChord**.

The ComputeCOG function defined in Figure 4.16 is called in line 1 of SpellChord and takes three arguments: **Chord**, **OMDTList**, and *WindowSize*. This function computes the COG at the onset time of the notes in the chord **Chord**. The argument **OMDTList** is a list of $\langle o, m, d, t\rangle$ 4-tuples giving the onset time, o, MIDI note number, m, duration, d, and TPC, t,

of all the notes starting before the onset time of the notes in **Chord**. The value of *WindowSize* is the same as that given by the user as the fifth argument of TPROne (see Figure 4.13). If **Chord** is the first chord, then **OMDTList** will be empty in line 1 and the *COG* will be set to 4, the TPC of `"Dn"`. Temperley (2001, p. 127) claims that "for actual musical passages in the major, the COG is generally about two steps in the sharp direction from the tonic" along the line of fifths. He supports this claim by observing that the mean line of fifths position of the pitches in a large corpus of works used by Krumhansl (1990, p. 67) was 3.82 (i.e., close to 4) when the pitches were represented as scale degrees (i.e., given relative to the tonic) and only pitches diatonically related to the tonic were considered (Temperley, 2001, p. 367, Chapter 5, endnote 5). Setting the initial COG to 4 is therefore tantamount to setting the key initially to C major.

If **OMDTList** is not empty then it is used to compute the current COG in lines 4–15 of ComputeCOG. In lines 4 and 5, the variable n is set for convenience to equal the number of elements in **OMDTList**, and the variable *ThisOnset* is set to equal the onset time of the notes in the chord **Chord**. As discussed above, the value of *WindowSize* must be either nil or a positive integer. If *WindowSize* = nil, all notes with onset times less than that of the note currently being spelt contribute to the current COG. If *WindowSize* is a positive integer, only the *WindowSize* most recent notes with onset times less than that of the current note contribute to the current COG. This is implemented in lines 6–10 of ComputeCOG. If *WindowSize* is not nil in line 6, then the variable **Window** is set in lines 7–8 to equal just that portion of **OMDTList** which contains the $\langle o, m, d, t\rangle$ 4-tuples for the *WindowSize* most recent notes starting before *ThisOnset*. If *WindowSize* is nil, then **Window** is set to equal **OMDTList** in line 10. In line 11, the variable w is set for convenience to equal the number of elements in **Window**. The COG is then defined to be the weighted mean (Borowski and Borwein, 1989, p. 634) of the TPCs of the notes in **Window**, the TPC for each note being weighted for recency and duration. Let $n_i = \langle o_i, m_i, d_i, t_i\rangle$ be one of the $\langle o, m, d, t\rangle$ 4-tuples in **Window**. Let δ_i be the duration weight of n_i and ρ_i be the recency weight of n_i. The COG at time point τ, COG_τ, is then given by

$$COG_\tau = \frac{\sum_{n_i \in \mathbf{Window}} t_i \delta_i \rho_i}{\sum_{n_i \in \mathbf{Window}} \delta_i \rho_i}.$$

In TPROne, δ_i is defined to be equal to d_i and ρ_i is defined to be equal to $1/(\tau - o_i)$. Therefore,

$$COG_\tau = \frac{\sum_{n_i \in \mathbf{Window}} \left(\frac{t_i d_i}{\tau - o_i}\right)}{\sum_{n_i \in \mathbf{Window}} \left(\frac{d_i}{\tau - o_i}\right)}. \tag{4.1}$$

The COG is computed in accordance with this definition in lines 12–15 of ComputeCOG. First, the recency weights ρ_i for all the notes in **Window** are computed in line 12 and stored in the ordered set **RecencyWeights**. Then the duration weights δ_i for all the notes in **Window** are computed in line 13 and stored in the ordered set **DurationWeights**. Then, in line 14, the TPCs of all the notes in **Window** are stored for convenience in the ordered set **TPCs**. Finally, the COG is computed in line 15 in accordance with the definition in Eq. 4.1 and stored in the variable *COG* which is then returned in line 16.

```
ComputeTPC(MIDINoteNumber, COG, RA4OrRD5, MaxTPC, MinTPC)
1    TPCClass ← (2 + 7 (MIDINoteNumber mod 12)) mod 12
2    COGTPCClass ← COG mod 12
3    COGToNoteLOFDisp ← TPCClass − COGTPCClass
4    if COGToNoteLOFDisp > 6
5        TPC ← Round(COG + COGToNoteLOFDisp − 12)
6    else
7        if −6 < COGToNoteLOFDisp < 6
8            TPC ← Round(COG + COGToNoteLOFDisp)
9        else
10           if COGToNoteLOFDisp < −6
11               TPC ← Round(COG + COGToNoteLOFDisp + 12)
12           else
13               if RA4OrRD5 = ra4
14                   TPC ← Round(COG + 6)
15               else
16                   TPC ← Round(COG − 6)
17   while TPC < MinTPC
18       TPC ← TPC + 12
19   while TPC > MaxTPC
20       TPC ← TPC − 12
21   return TPC
```

Figure 4.17: The ComputeTPC algorithm.

The ComputeTPC function defined in Figure 4.17 is called in line 4 of SpellChord (Figure 4.15) and takes five arguments: *MIDINoteNumber*, *COG*, *RA4OrRD5*, *MaxTPC* and *MinTPC*. This function assigns a TPC to the current note, given that its MIDI note number is *MIDINoteNumber* and the current COG is *COG*. The values of *RA4OrRD5*, *MaxTPC* and *MinTPC* are the same as those given by the user as input to TPROne (see Figure 4.13). Two TPCs have the same neutral pitch class if the difference between them is an integer multiple of 12. For example, TPCs 0 and 12 (i.e., `"Bf"` and `"As"`) have a neutral pitch class of 10, TPCs 1 and 13 (`"Fn"` and `"Es"`) have a neutral pitch class of 5, and so on. Let the *chromatic TPC class* of a TPC, *TPC*, be *TPC* mod 12. The chromatic TPC classes of two TPCs are equal iff they have the same neutral pitch class. The first step in ComputeTPC is to calculate the chromatic TPC classes of the current note to be spelt and the current COG and store these values in the variables *TPCClass* and *COGTPCClass*, respectively. This is done in lines 1–2 (see Figure 4.17). Next, in line 3, the variable *COGToNoteLOFDisp* is set to equal the value that has to be added to the chromatic TPC class of the COG to get the chromatic TPC class of the current note to be spelt. This value will always be between −11 and 11. Every note has to be assigned a TPC which is as close as possible to the current COG, therefore the absolute difference between the TPC and the current COG must always be less than or equal to 6. If the chromatic TPC class of the current note is more than 6 greater than that of the current COG (i.e., *COGToNoteLOFDisp* > 6) in line 4 of ComputeTPC, then the TPC assigned to the current note is set to be 12 less than the sum of the current COG and *COGToNoteLOFDisp*, rounded to the nearest integer (see line 5 in Figure 4.17). Otherwise, if the absolute difference between the chromatic TPC classes of the COG and the current note is less than 6 in line 7, the TPC assigned to the note is just the sum of the current COG and *COGToNoteLOFDisp*, rounded to the nearest integer (line 8). Otherwise, if the chromatic TPC class of the current note is more than 6 less than that of the current COG in line 10, then the TPC of the current note

is set to be 12 greater than the sum of the current COG and *COGToNoteLOFDisp*, rounded to the nearest integer (see line 11). Finally, if the absolute difference between the chromatic TPC classes of the COG and the current note is exactly 6, then the TPC is set to $COG + 6$ if *RA4OrRD5* = ra4 and $COG - 6$ if *RA4OrRD5* = rd5 (see lines 13–16 in Figure 4.17). If the TPC assigned to the note in lines 4–16 of ComputeTPC is outside the range permitted by the values of *MaxTPC* and *MinTPC* set by the user, then, in lines 17–20, it is repeatedly either reduced or increased by 12 until it is between *MaxTPC* and *MinTPC*.

4.7.3.3 Time and space complexities of TPROne

Let n denote the number of $\langle o, m, d \rangle$ triples in the ordered set **SortedOMDList** given to TPROne as input. The worst-case time complexity of OMDList2ChordList (see Figure 4.14) is $O(n)$ which means that line 1 of TPROne executes in $O(n)$ time in the worst-case. Lines 2–3 of TPROne can be assumed to execute in constant time (assuming the implementation stores the size of **ChordList** as it is constructed). In the worst case, N_{Ch} is equal to n and the 'for' loop in lines 4–6 iterates n times. On each of these iterations, the SpellChord function is called in line 5. In line 1 of SpellChord, the function ComputeCOG is called. The worst-case time complexity of ComputeCOG is $O(\mathit{WindowSize})$ if *WindowSize* ≠ nil and $O(n)$ otherwise. Consequently, the worst-case time complexity of SpellChord is also $O(\mathit{WindowSize})$ if *WindowSize* ≠ nil and $O(n)$ otherwise. It follows that the overall worst-case time complexity of TPROne is $O(n \times \mathit{WindowSize})$ if *WindowSize* ≠ nil and $O(n^2)$ if *WindowSize* = nil. The algorithm uses $O(n)$ space in the worst case.

4.7.3.4 Results obtained when TPROne was run on the test corpus $\mathcal{C}$

The TPROne algorithm was run on the test corpus $\mathcal{C}$ defined in Table 1.4 four times, each time with a different combination of parameter values. Table 4.6 shows the four combinations of parameter values used in this evaluation, together with the codes (TPR1A–TPR1D) that will be used throughout this section to denote these parameter value combinations. The combinations of parameter values used were chosen so as to test (in a very rudimentary way):

1. whether or not the linear-time version of the algorithm in which *WindowSize* ≠ nil can be made to perform as well as the quadratic-time version in which *WindowSize* = nil;

2. whether the value of the parameter *RA4OrRD5* makes any difference to the performance of the algorithm.

The results are summarised in Tables 4.7 and 4.8. The best results obtained with the `met-harm` and `met-harm-two-pass` scripts are also included in these tables for comparison. As in Tables 4.2 and 4.3, when an algorithm performed better on the modified version of the fourth movement of Haydn's 'Military' Symphony than on the original version containing the sudden enharmonic change, the results obtained using the modified version are given in parentheses after the results obtained with the original version of this movement (see section 1.3.4.3).

When the original version of the fourth movement of Haydn's 'Military' Symphony was used, the versions of TPROne in which *WindowSize* was set to nil (TPR1A and TPR1D) spelt more

Code	*RA4OrRD5*	*MaxTPC*	*MinTPC*	*WindowSize*
TPR1A	ra4	34	-24	nil
TPR1B	ra4	34	-24	100
TPR1C	ra4	34	-24	1000
TPR1D	rd5	34	-24	nil

Table 4.6: The combinations of parameter values used to evaluate TPROne.

Algorithm	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
TPR1A	151	532	119	166	428	236	150	105	1887
TPR1D	155	532	129	167	428	236	151	107	1905
MH2PX2	30	815	5	17	3229 (267)	170	26	33	4325 (1363)
MHX2	31	831	4	21	3252 (290)	163	23	27	4352 (1390)
TPR1C	148	549	119	165	3332 (376)	229	150	104	4796 (1840)
TPR1B	152	512	130	119	3425 (483)	254	131	158	4881 (1939)

Table 4.7: Note error counts for algorithms in column labelled "Algorithm" for the complete test corpus (last column), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus. The algorithms are sorted into increasing order of overall note error count. See text and sections 1.3.4 and 1.3.6 for further details.

Algorithm	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\overline{\text{NA}}$	SD_{Sty}
TPR1A	99.38	97.83	99.51	99.32	98.25	99.04	99.39	99.57	99.04	99.04	0.65
TPR1D	99.37	97.83	99.47	99.32	98.25	99.04	99.38	99.56	99.03	99.03	0.64
MH2PX2	99.88	96.67	99.98	99.93	86.82 (98.91)	99.31	99.89	99.87	97.79 (99.30)	97.79 (99.31)	4.57 (1.13)
MHX2	99.87	96.61	99.98	99.91	86.72 (98.82)	99.33	99.91	99.89	97.78 (99.29)	97.78 (99.29)	4.61 (1.16)
TPR1C	99.40	97.76	99.51	99.33	86.39 (98.46)	99.07	99.39	99.58	97.55 (99.06)	97.55 (99.06)	4.55 (0.63)
TPR1B	99.38	97.91	99.47	99.51	86.01 (98.03)	98.96	99.47	99.36	97.51 (99.01)	97.51 (99.01)	4.68 (0.67)

Table 4.8: Note accuracies expressed as percentages for each algorithm in the column headed "Algorithm" for the complete test corpus (column 10), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The columns headed $\overline{\text{NA}}$ and SD_{Sty} give the mean and standard deviation, respectively, of the values in columns 2 to 9. The algorithms are sorted into decreasing order of overall note accuracy. See text and sections 1.3.4 and 1.3.6 for further details.

than 99% of the notes in the test corpus correctly, making fewer than half as many errors as the best versions of Sleator and Temperley's implementation of the complete theory. It should also be remembered that, unlike Sleator and Temperley's implementation, TPROne's performance is independent of tempo, metrical structure and harmonic structure. However, the linear-time versions of TPROne (TPR1C and TPR1B) made approximately 10% more errors than the best versions of Sleator and Temperley's implementation when the original version of the fourth movement of Haydn's 'Military' Symphony was used. Note that increasing *WindowSize* from 100 in TPR1B to 1000 in TPR1C reduced the total number of errors by less than 2% from 4881 to 4796. Changing from $RA4OrRD5 = \mathsf{ra4}$ in TPR1A to $RA4OrRD5 = \mathsf{rd5}$ in TPR1D also only had a relatively small effect, increasing the overall note error count by less than 1% from 1887 to 1906.

However, nearly 3000 of the errors made by TPR1B and TPR1C were due to the sudden enharmonic change in the fourth movement of Haydn's 'Military' Symphony: when this change was omitted, the number of errors made on this movement fell from 3036 to 94 for TPR1B and from 3034 to 78 for TPR1C. Indeed, when this sudden enharmonic change was omitted, the overall number of errors made by TPR1C on the corpus (1840) was less than that made by TPR1A when the original version of the fourth movement of Haydn's 'Military' Symphony was used (1887). Nevertheless, when the sudden enharmonic change in the Haydn movement was omitted, even the best version of TPROne tested (TPR1C, $NA = 99.06\%$) made 35% more errors than the best version of Sleator and Temperley's algorithm ($NA = 99.30\%$). Again, when the sudden enharmonic change in the Haydn movement was omitted, increasing *WindowSize* from 100 in TPR1B to 1000 in TPR1C only had a relatively small effect on the total number of errors, reducing it by 5.4% from 1939 to 1840.

In TPROne, the recency weight associated with a note is inversely proportional to the interval between it and the note to be spelt (see Figure 4.12). One therefore might not expect notes that occur more than 1000 notes before a note to have any significant effect on its spelling. Consequently, one might expect the results for TPR1C, which uses a window containing the 1000 notes preceding that to be spelt, to be almost identical to those for TPR1A, which uses a window containing *all* the notes preceding that to be spelt. This expectation seems to be contradicted by the fact that TPR1C made over $2\frac{1}{2}$ times as many errors over $\mathcal{C}$ as TPR1A when the sudden enharmonic change in the fourth movement of Haydn's 'Military' Symphony was retained in the ground truth (see Table 4.7). However, a closer examination of the results reveals that the total number of errors over all of $\mathcal{C}$ *except* the fourth movement of Haydn's 'Military' Symphony are almost the same for TPR1A and TPR1C: 1778 errors for TPR1A and 1762 errors for TPR1C. Moreover, as can be seen in Table 4.7, the results obtained by the two algorithms are very similar for all the composers apart from Haydn. It is therefore only the results obtained for the two algorithms on the fourth movement of Haydn's 'Military' Symphony that seem to contradict our expectation that TPR1A and TPR1C would perform very similarly.

Figure 4.18 shows the COGs calculated by TPR1A and TPR1C at each onset position in the fourth movement of Haydn's 'Military' Symphony. The figure also shows the difference between these COGs at each onset position. As expected, the COGs computed by the two algorithms are identical for the first 1000 notes (i.e., up to the onset at 638 semiquavers), since, for this

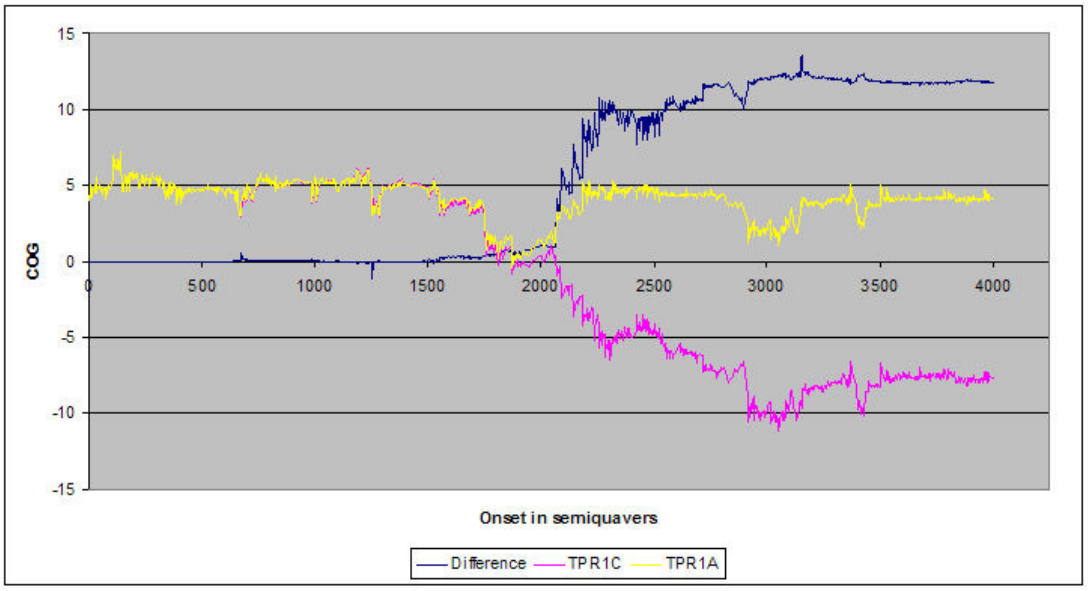


Figure 4.18: Graph showing the COG at each onset in the fourth movement of Haydn's 'Military' Symphony, as calculated by TPR1A and TPR1C. The third line shows the difference between the COGs calculated by the two algorithms at each onset in the movement.

initial segment of the movement, the contexts used by the two algorithms to compute each COG are identical. After this point, the contexts used to compute the COGs become more and more different as the piece progresses. Specifically, in TPR1C, a longer and longer segment from the beginning of the movement (including a substantial segment in the home key) fails to have any influence over the spellings of the notes. Nevertheless, the COGs computed by the two algorithms remain very similar (no more than about 1 step apart on the line of fifths) until the onset at 2072 semiquavers when the COG for TPR1C moves from 0.26 to -0.98 and the COG for TPR1A moves from 1.28 to 3.22. The onset at 2072 semiquavers corresponds to half-way through bar 172 in the score in Figure 1.5. Note that this is just $6\frac{1}{2}$ bars after the sudden enharmonic change. Thereafter, the COGs for the two algorithms diverge, with that of TPR1A returning to around 5 and that of TPR1C decreasing to around -8. Note that from just before 3000 semiquavers to the end of the piece, the COGs move approximately in parallel with that for TPR1C being approximately 12 steps flatter than (i.e., a diminished second above) that for TPR1A. At 2072 semiquavers in bar 172, the context for computing the COG in TPR1C consists of the notes in the segment between 1240 and 2072 semiquavers and thus excludes the long initial segment up to 1240 semiquavers over which the COG is consistently around 5 (see Figure 4.18). Clearly, this initial segment in the home key has enough of an effect on the computation of the COG at 2072 semiquavers in TPR1A to make it move in the opposite direction along the line of fifths to that in which the COG moves at this point in TPR1C. Once this COG movement in the negative direction along the line of fifths starts in bar 172 in TPR1C, more and more notes become spelt in a way that reinforces this negative tonal region, so the spellings of the two algorithms never again converge.

To sum up, the results obtained with TPROne clearly support Temperley's (2001, p. 125) claim that TPR 1 is "in many cases... sufficient to ensure the correct spelling of passages".

Indeed, when the algorithms were required to identify and correctly spell sudden enharmonic changes, my simple implementation of TPR 1 outperformed a system that implements the whole of Temperley's theories of metrical structure, harmonic structure and pitch spelling. However, only the quadratic-time version of TPROne was able to do this: even placing an upper limit of 1000 on the number of preceding notes to take into account when calculating the COG made TPROne perform less well than Sleator and Temperley's system. It seems that sudden enharmonic changes can only be predicted by TPROne when it is allowed to take into account all the notes in the movement that precede the change. In the case of the fourth movement of Haydn's 'Military' Symphony, this was because the movement began with a substantial segment in which the COG was fairly stable, followed by a drift in the negative direction along the line of fifths which could only be halted when the stable opening section was permitted to influence the spelling. Taking all the notes preceding the one to be spelt into account therefore seems to allow the quadratic-time version of TPROne to prevent the COG from straying too far from the position corresponding to the home key on the line of fifths.

When the algorithms were not required to deal with sudden enharmonic changes, the linear-time versions of TPROne made fewer errors than the quadratic-time versions but none of the tested versions of TPROne were as accurate as the best versions of Sleator and Temperley's implementation. Nonetheless, even when enharmonic changes were ignored, the best versions of TPROne spelt over 99% of the notes in the corpus correctly.

4.8 Summary and conclusions

Temperley's (2001) theory of music cognition consists of *preference rule systems* for six aspects of musical structure: metre, phrasing, counterpoint, harmony, key and pitch spelling. Most of this theory has been implemented by Daniel Sleator in a suite of computer programs called *Melisma*.[3] These programs take "note list" representations as input (Temperley, 2001, pp. 9–12) in which the pitch of each note (or sequence of tied notes) is represented by its MIDI note number and its onset-time and offset-time are given in milliseconds.

Temperley's theory of pitch spelling—or, as he calls it, "tonal-pitch-class labeling" (Temperley, 2001, pp. 123–132)—consists of the three *tonal-pitch-class preference rules* (TPRs) shown in Figure 4.4. In Temperley's theory, the *tonal pitch class* (TPC) of a note is an integer that indicates the position of the pitch name class of the note on the line of fifths (Temperley, 2001, pp. 118, 123–125). Temperley (2001, p. 125) claims that TPR 1 (see Figure 4.4) is "the most important" TPR and that "in many cases, this rule is sufficient to ensure the correct spelling of passages". TPR 2 is designed to account for the way that notes are typically spelt in chromatic scale segments (Temperley, 2001, pp. 127–130). TPR 3 states that the system should "prefer TPC representations which result in good harmonic representations" (Temperley, 2001, p. 131). Temperley formally defines the concept of a "good harmonic representation" in the first rule in his theory of harmony, HPR 1 (Temperley, 2001, p. 149), which states that, in choosing the roots for chords, certain specified TPC-root relationships should be preferred over others. Temperley's theories of pitch spelling and harmony are therefore interdependent and this is reflected in the

[3] Available online at
<http://www.link.cs.cmu.edu/music-analysis/>.

fact that they are both implemented in the `harmony` program in *Melisma*.

The complexity of Temperley's pitch spelling algorithm is increased still further by the fact that his theory of harmony depends on his theory of metrical structure. For example, the second harmonic preference rule states that the system should "prefer chord spans that start on strong beats of the meter" (Temperley, 2001, pp. 151, 359). The `harmony` program therefore requires as input both a "note list" and a representation of the metrical structure of the passage in the form of a "beat list" of the type generated by the *Melisma* `meter` program. Consequently, if one wishes to use Temperley's theory to determine the pitch names of the notes in a passage, one must carry out a process, which I denote by MH, in which one first uses the `meter` program to generate a beat list from a note-list and then processes both the note list and the beat list using the `harmony` program to compute the pitch names of the notes in the passage.

In an attempt to take harmonic rhythm into account when computing metrical structure, Temperley and Sleator also experimented with a "two-pass" method (Temperley, 2001, pp. 46–47), in which the `meter` and `harmony` programs are both run twice, once in a special "prechord" mode and then again in "normal" mode. I denote this "two-pass" method by MH2P.

The output of the `meter` program depends on tempo. Therefore, an evaluation was carried out in which both MH and MH2P were run on six different versions of the test corpus, one in which the music was at a "natural" tempo and five other versions in which the tempo was multiplied by 2, 4, $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{6}$. To test the extent to which pitch spelling depends on metrical structure, the half-speed version of the test corpus was run through the `meter` program and then the beat list generated for each movement was modified so that every beat had the same strength. These beat lists were then used to generate pitch names using the `harmony` program. I denote this procedure by HNM (for "Harmony-No-Meter"). Finally, a simple implementation of TPR 1, which I call TPROne, was run on the dataset to test Temperley's claim that "in many cases, this rule is sufficient to ensure the correct spelling of passages" (Temperley, 2001, p. 125). TPROne spells each note so that its pitch name is as close as possible on the line of fifths to the weighted average of the TPCs of the w notes immediately preceding the note to be spelt. w is either a fixed, user-supplied value (in which case TPROne runs in linear time) or a variable value that is always equal to the number of notes preceding the note to be spelt (in which case TPROne runs in quadratic time). Both linear-time and quadratic-time versions of TPROne were run on the test corpus.

It was found that both the MH and MH2P versions of Temperley and Sleator's algorithm performed very similarly and did best when the music was either at its natural tempo or at half speed. At these tempi, both MH and MH2P spelt 97.8% of the notes in the corpus correctly. Both systems were highly sensitive to tempo: if the music was more than twice the natural tempo or less than a quarter of the natural tempo, the spelling accuracy was severely reduced for both scripts. The note accuracy achieved by the best versions of MH and MH2P was less than that of the best version of Longuet-Higgins's algorithm (98.2%) (see Tables 2.2 and 2.3) and about equal to that achieved by the *worst* version of Cambouropoulos's algorithm tested (see Tables 3.11 and 3.12). However, 70% of the errors made by MH and MH2P on the half-speed version of the corpus were caused by the sudden enharmonic change in the fourth movement of Haydn's 'Military' Symphony discussed in section 1.3.4.3 above. When the outputs of MH and

MH2P were compared with the version of this movement with the enharmonic change omitted, their overall note accuracies on the test corpus rose to 99.3% which was greater than the 99.15% accuracy achieved by my optimized version of Cambouropoulos's algorithm (with the enharmonic change in the Haydn movement included) (see Table 3.20).

It was found that the style dependence and note accuracy of MH and MH2P depended on tempo in a similar way, so that the tempi at which the scripts achieved the highest overall note accuracy were also those at which they performed most consistently across the different styles in the test corpus. When the sudden enharmonic change in the Haydn movement was included in the ground truth, the best value of style dependence achieved by MH and MH2P was 4.41, which is considerably worse than the best value achieved by Longuet-Higgins's algorithm (1.79) and very much worse than even the worst value obtained using Cambouropoulos's algorithms (0.91) (see Table 3.12). When the sudden enharmonic change was omitted from the ground truth, the style dependences of the most accurate versions of MH and MH2P tested were markedly improved. For example, the style dependence of MH2P when it was run on the half-speed corpus was 1.13 which is better than the best value achieved using Longuet-Higgins's algorithm (1.79) and comparable with that achieved by the least consistent version of Cambouropoulos's algorithm (0.91).

Ignoring metrical information in the HNM procedure caused the overall number of errors to rise by 24% when the original score of the fourth movement of Haydn's 'Military' Symphony was used as a ground truth and by 75% when the modified version was used. However, this fall in note accuracy was primarily due to HNM's inability to cope with one particular chromatic passage in bars 85–96 of the third movement of Vivaldi's Concerto in G minor ('L'estate') from 'Le Quattro Stagioni' (Op. 8, No. 2, RV 315). This suggests that the use of metrical structure in Temperley and Sleator's system helps it to cope with certain types of chromatic passage, which, in turn, makes it less likely to spell whole segments in the wrong key.

When the sudden enharmonic change in the Haydn movement was included, the best quadratic version of TPROne tested spelt 99.04% of the notes in the corpus correctly whereas the best linear version (in which w was set to 1000) spelt only 97.55% of the notes correctly. However, when the enharmonic change was omitted from the ground truth, the best linear version performed better than the quadratic version, spelling 99.06% of the notes correctly. With the enharmonic change included, therefore, my quadratic implementation of TPR 1 made less than half as many errors as the best versions of MH and MH2P tested. With the enharmonic change omitted, however, even the best version of TPROne made 35% more errors than MH2P and MH. Nevertheless, all versions of TPROne were less dependent on style than any of the versions of MH and MH2P tested. These results support Temperley's (2001, p. 125) claim that TPR 1 is "in many cases... sufficient to ensure the correct spelling of passages".

N.B. Much of the material included in this chapter has been published in the following paper: Meredith, D. (2007a). Optimizing Chew and Chen's pitch-spelling algorithm. Computer Music Journal, 31:2, 54-72. http://dx.doi.org/10.1162/comj.2007.31.2.54

Chapter 5

Chew and Chen's pitch spelling algorithms

5.1 Introduction

5.1.1 The spiral array

Chew and Chen (2003a,b, 2005) describe several variants of a real-time pitch spelling algorithm based on Chew's (2000) "Spiral Array Model", which is a geometric model of tonal pitch relations. In the spiral array, the pitch name classes are arranged on a helix so that adjacent pitch name classes along this helix are a perfect fifth apart and adjacent pitch name classes along the length of the cylinder in which the helix is embedded are a major third apart (see Figure 5.1). As Chew and Chen (2005, p. 67) point out, the spiral array is "a spiral configuration of the line of fifths". That is, the spiral array can be constructed by 'coiling up' the line of fifths. Like Longuet-Higgins's (1987b, p. 66) two-dimensional "map of notes" and his three-dimensional "tonal space" (Longuet-Higgins, 1987a, p. 110), Chew's spiral array is a geometric representation of the idea that, in tonal music, pitches a major third and a perfect fifth apart are perceived to be particularly closely related. Chew and Chen (2005, p. 66) claim that the "depth added by going from one to three dimensions [i.e., from the line of fifths to the spiral array] allows the modeling of more complex hierarchical relations".

5.1.2 Center of effect

Let's suppose that S is a set of notes in a piece of tonal music, that $\mathbf{p}(n)$ denotes the vector representing the position in the spiral array of the pitch name class of the note, n, and that $d(n)$ is the duration of note, n. Chew and Chen (2005, p. 67) define the *center of effect* (CE) of S, denoted by $CE(S)$, to be

$$CE(S) = \frac{\sum_{n \in S} d(n).\mathbf{p}(n)}{\sum_{n \in S} d(n)}. \tag{5.1}$$

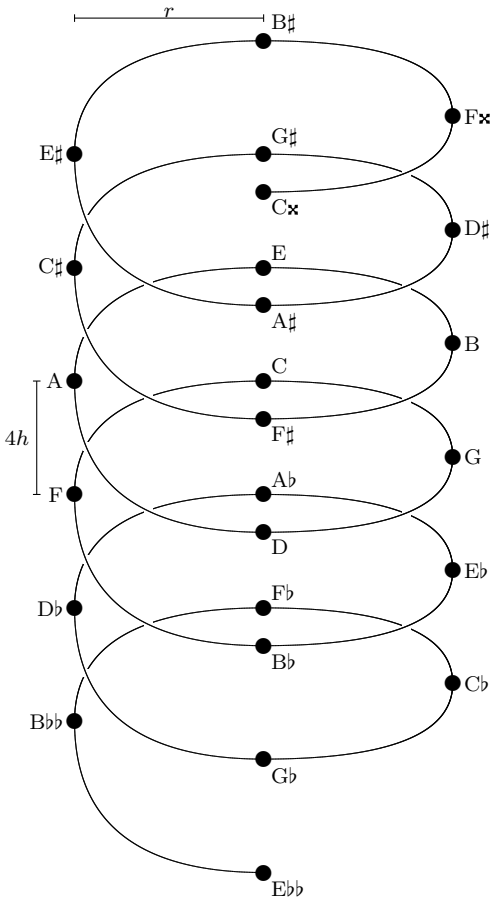


Figure 5.1: Part of Chew's (2000) spiral array model representing the perceived tonal relations between pitch name classes. r is the radius of the cylinder in which the helix is embedded and h is the distance parallel to the axis of this cylinder between two points one step apart along the helix. r/h is the aspect ratio of the spiral array (Chew and Chen, 2005, p. 67).

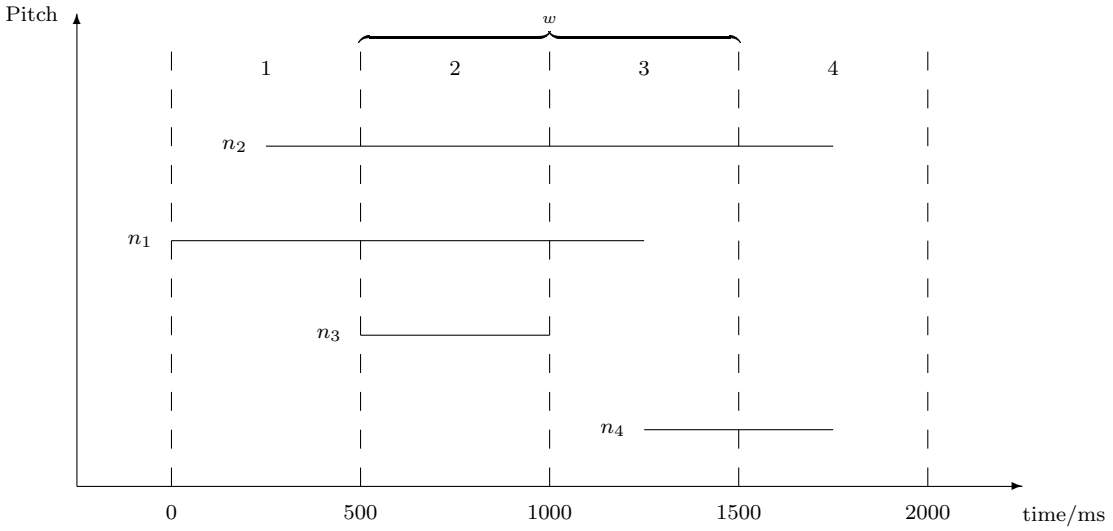


Figure 5.2: Piano-roll representation of four notes in a passage of music. The horizontal axis represents time and the vertical axis represents pitch. Each horizontal line segment, n_1–n_4, represents a note. The vertical dashed lines represent chunk boundaries.

That is, the CE of a set of notes is the weighted centroid[1] of the position vectors of the pitch name classes of the notes in the spiral array, each note being weighted by its duration. In Chew and Chen's (2005, p. 63) algorithm, the CE for a set of notes is used to represent (or "act as a proxy for") the key. The basic principle underlying Chew and Chen's algorithms is that each note should be spelt so that it is as close as possible in the spiral array to the CE of the notes that precede it. This principle is almost identical to that expressed in Temperley's (2001, p. 125) TPR 1 and implemented in the TPRONE algorithm described in the previous chapter (see section 4.7.3). The main differences between the two approaches are that Chew and Chen use the spiral array where Temperley uses the line of fifths and that Chew and Chen use a rather more complex windowing mechanism (described below) than that used in Temperley and Sleator's system.

In Chew and Chen's (2005) algorithm, it is assumed that the input data gives the MIDI note number, together with the onset and duration in milliseconds of each note. The data is then divided into "equal time slices" called *chunks* (Chew and Chen, 2005, p. 67) and the algorithm spells the notes a chunk at a time.

Let $W_{\text{sound}}(i, j)$ denote the set of notes that are *sounding* in a window consisting of chunks i to j; and let $W_{\text{start}}(i, j)$ denote the set of notes that *start* in a window consisting of chunks i to j. Chew (2004) has confirmed that, in her and Chen's implementation, the CE for a window is calculated by considering the notes that *sound* in the window, not the notes that *start* in it. To understand the distinction between these two possibilities, consider Figure 5.2 which shows a 'piano-roll' representation of four notes. In this representation, the horizontal axis represents time and the vertical axis represents pitch. Each note is represented as a horizontal line segment whose vertical position indicates the pitch. The horizontal co-ordinate values of the left and right end-points of each line segment represent the onset and offset times of the note, respectively.

[1] "A point whose coordinates are the mean values of the coordinates of the points in a given set" (Borowski and Borwein, 1989, p. 76, **centroid,** *n.*, sense **2**).

Each note in this diagram is given a label, n_1–n_4. Let's suppose that the music in Figure 5.2 is divided into the four 500ms chunks labelled 1–4 and indicated by the vertical dashed lines in the diagram. Let's further suppose that we wish to calculate $CE(w)$, the CE for the window w consisting of chunks 2–3. Let's define that a note is deemed to start in a chunk if its onset is greater than or equal to the start time of the chunk and strictly less than the end time of the chunk. Let's further define that a note is deemed to sound in a chunk if its onset is strictly less than the end time of the chunk and its offset is strictly greater than the start time of the chunk. If we only consider the notes that *start* in w, then

$$CE(w) = CE(W_{\text{start}}(2,3)) = CE(\{n_3, n_4\}).$$

From Eq. 5.1, it follows that, in this case,

$$CE(w) = \frac{500.\mathbf{p}(n_3) + 500.\mathbf{p}(n_4)}{1000}. \tag{5.2}$$

On the other hand, if we consider the notes that *sound* in w, then

$$CE(w) = CE(W_{\text{sound}}(2,3)) = CE(\{n_1, n_2, n_3, n_4\})$$

which would seem to imply that

$$CE(w) = \frac{1250.\mathbf{p}(n_1) + 1500.\mathbf{p}(n_2) + 500.\mathbf{p}(n_3) + 500.\mathbf{p}(n_4)}{3750}. \tag{5.3}$$

Note first that the expression for $CE(w)$ in Eq. 5.3 is quite different from that in Eq. 5.2. This implies that, in general, the results obtained when considering the notes that start in a window will be different from those obtained when considering the notes that sound in a window. However, there is a problem with the expression in Eq. 5.3: the spiral array position vector of each note is multiplied by its complete duration even though three of the notes extend outside of w. For example, in Eq. 5.3, the spiral array position vector of n_1 is multiplied by 1250 even though n_1 only sounds for 750ms in w. Chew (2004) has confirmed that, in her and Chen's implementation, each note is weighted by the duration for which it sounds *within the window over which the CE is being calculated*. Thus, in Chew and Chen's implementation, the CE for the window w in Figure 5.2 would actually be

$$CE(w) = \frac{750.\mathbf{p}(n_1) + 1000.\mathbf{p}(n_2) + 500.\mathbf{p}(n_3) + 250.\mathbf{p}(n_4)}{2500}. \tag{5.4}$$

Note that the expression in Eq. 5.4 is still different from that in Eq. 5.2 which demonstrates the importance of specifying in detail the way that the CE is calculated.

5.1.3 An overview of Chew and Chen's pitch spelling algorithm

Let's suppose that the algorithm is about to spell the notes in chunk j. According to Chew and Chen (2005, pp. 70–71), the following steps are executed.

1. First, the algorithm computes the *global CE*, $CE_{\text{global},j}$, which is the CE of the set of notes

in a sliding *global context window* that consists of the w_s chunks immediately preceding the jth chunk.[2] In other words, the algorithm computes the value

$$CE_{\text{global},j} = CE(W_{\text{sound}}(j - w_s, j - 1)), \tag{5.5}$$

where w_s, the number of chunks in the sliding global context window, is a parameter of the algorithm.

2. Next, the algorithm names each note in chunk j so that its pitch name is as close as possible to $CE_{\text{global},j}$ in the spiral array.

3. Then the algorithm computes a *local CE*, $CE_{\text{local},j}$, which is the CE of the set of notes in a *local context window* that consists of the chunk j that has just been spelt, together with the $(w_r - 1)$ chunks immediately preceding the jth chunk. That is, it computes the value of

$$CE_{\text{local},j} = CE(W_{\text{sound}}(j - w_r + 1, j)), \tag{5.6}$$

where w_r, the number of chunks in the local context window, is another parameter of the algorithm.

4. Next the algorithm computes the *cumulative CE*, $CE_{\text{cum},j}$, which is the CE of the notes in a *cumulative window* that consists of all the chunks preceding the jth chunk. That is, it computes the value of

$$CE_{\text{cum},j} = CE(W_{\text{sound}}(1, j - 1)). \tag{5.7}$$

5. Finally, the notes in chunk j are re-spelt so that their pitch names are as close as possible to the *hybrid CE*,

$$CE_{\text{hybrid},j} = f.CE_{\text{local},j} + (1 - f).CE_{\text{cum},j}, \tag{5.8}$$

where f is a parameter with a value between 0 and 1 which determines the relative weight given to the local and cumulative CEs.

Steps 1 and 2 constitute what Chew and Chen (2005, pp. 70–71) call "Phase I" of the algorithm and steps 3 to 5 make up "Phase II".

Chew and Chen (2005, p. 67) define the *index* of a note to be an integer that indicates the position of the pitch name class of the note on the line of fifths. The index of `"Cn"` is defined to be 0 and the index increases by 1 for each step in the 'sharp' direction.

The procedure for spelling the notes in the first chunk of the music to be processed is different from that described above. The notes in the first chunk are first spelt so that their indices are as close as possible to some specified *initial index* which Chew and Chen (2005, p. 71) set to 2 (corresponding to `"Dn"`), on the grounds that this biases "the notation towards fewer sharps and flats". Note that this strategy for initializing the algorithm is essentially identical to that

[2] In everyday parlance, taking into account the 'global context' of an event usually implies taking into account everything that is directly or indirectly relevant to it; whereas the 'global context window' in Chew and Chen's model contains only a finite number of chunks preceding that to be spelt. It might therefore have made more sense to call this window a 'regional' or 'mid-scale' context window. Nevertheless, I shall use Chew and Chen's own terminology throughout this text in order to be consistent with their publications and avoid the possibility of confusion arising from terms relating to their model having different meanings in different studies.

Algorithm	*Publication*	w_s	w_r	f	r/h	*Initial index*
Two-phase boot-strapping algorithm	Chew and Chen (2005) Chew (2004)	Any	$< w_s$	$0 \leq f \leq 1$	$\sqrt{15/2}$	2
Algorithm 1: Cumulative CE	Chew and Chen (2003a)	0	0	0	?	0
Algorithm 2: Sliding window	Chew and Chen (2003a)	Any	0	1	?	0
Algorithm 3: Two-phase assignment	Chew and Chen (2003a)	Any	$< w_s$	$0 \leq f \leq 1$	?	0
Cumulative CE	Chew and Chen (2003b)	0	0	0	?	0

Table 5.1: The various versions of Chew and Chen's pitch spelling algorithm described in their publications.

used by Temperley (2001, pp. 127; 367, Chapter 5, endnote 5). Temperley initializes the COG to 4 (i.e., `"Dn"`) because "the COG is generally about two steps in the sharp direction from the tonic". It is therefore not surprising that, when Chew and Chen (2003a,b) set the initial index to 0 (corresponding to `"Cn"`), they found that "there was a bias towards the flatted keys" (Chew, 2004). Having assigned pitch names to the notes in the first chunk, a CE for this first chunk is then computed and the notes in the first chunk are respelt so that they are as close as possible on the spiral array to this newly computed CE.

Another parameter of the algorithm is the *aspect ratio* of the spiral array, r/h, where r is the radius of the cylinder in which the pitch name class helix is embedded and h is the distance parallel to the axis of the helix corresponding to one step along the spiral array (i.e., two pitch name classes a perfect fifth apart) (see Figure 5.1). The value of the aspect ratio determines the ratio of the distances in the spiral array between notes a major third apart and notes a perfect fifth apart. Chew and Chen set r/h to $\sqrt{15/2}$ in order to ensure that "the major thirds and [perfect] fifths are equidistant" (Chew, 2004).

Given r and h, the spiral array position vector of a note can be computed from its index, k, using the following formula (Chew and Chen, 2005, p. 67):

$$\mathbf{p}(k) = \langle r \sin(k\pi/2), r \cos(k\pi/2), kh \rangle \,. \tag{5.9}$$

5.1.4 Versions of the algorithm described in Chew and Chen's publications

As Chew and Chen (2005, p. 71) point out, most of the variants of their algorithm described by them in their publications are either specific instances or classes of instances of the algorithm described in section 5.1.3, in which some of the parameters w_s, w_r, f, r/h and the initial index are set to particular values.

Table 5.1 gives the combinations of parameter values in the algorithm described in section 5.1.3 that correspond to the versions of the algorithm described by Chew and Chen in their publications. The first row in Table 5.1 gives the parameter values for the "two-phase boot-strapping algorithm" described by Chew and Chen (2005) and Chew (2004). This version is simply the algorithm described in section 5.1.3, with r/h set to $\sqrt{15/2}$.

The second and fifth rows of Table 5.1 indicate that Chew and Chen's (2003a) "Algorithm 1" and the algorithm described by Chew and Chen (2003b), which both use just the cumulative CE to assign pitch names, can be implemented by setting both w_s and f to 0 (Chew and Chen,

2005, p. 71). If $w_s = 0$, then, from Eq. 5.5,

$$CE_{\text{global},j} = CE(W_{\text{sound}}(j, j-1)).$$

Chew and Chen (2005, p. 71) state that, when the window over which a CE is to be computed is empty, "a CE is not generated, and the algorithm defaults to a do-nothing step". Therefore, setting $w_s = 0$ causes steps 1 and 2 of the algorithm described in section 5.1.3 to be skipped. Since $f = 0$, there is also no point in computing $CE_{\text{local},j}$ in step 3 since this value will be multiplied by 0 when computing $CE_{\text{hybrid},j}$ in step 5. When $f = 0$ we should therefore assume that the term $f.CE_{\text{local},j}$ in Eq. 5.8 is ignored. To summarise, when $f = 0$ and $w_s = 0$, only steps 4 and 5 of the algorithm in section 5.1.3 are executed and, in step 5, $CE_{\text{hybrid},j}$ is set to equal $CE_{\text{cum},j}$.

Chew and Chen (2005, p. 71) also state that the sliding window algorithm described by Chew and Chen (2003a) can be implemented using the algorithm in section 5.1.3 by "setting w_s to the desired window size and by setting $w_r = 0$ and $f = 1$". This is indicated in the third row of Table 5.1. Setting w_r to 0 implies that the window over which the local CE would be calculated will be empty, which, in turn, implies that "a CE is not generated, and the algorithm defaults to a do-nothing step" (Chew and Chen, 2005, p. 71). Setting w_r to 0 therefore causes step 3 of the algorithm in section 5.1.3 to be skipped. It also causes the term $f.CE_{\text{local},j}$ in Eq. 5.8 to be ignored. Setting f to 1 means that the cumulative CE is multiplied by 0 in step 5 of the algorithm, therefore there is no point in calculating the cumulative CE in step 4 and the term $(1-f).CE_{\text{cum},j}$ in step 5 is also ignored. To summarise, setting w_r to 0 and f to 1 causes only steps 1 and 2 of the algorithm in section 5.1.3 to be executed.

As indicated in the fourth row of Table 5.1, the "two-phase assignment" algorithm described by Chew and Chen (2003a) is essentially the same as the algorithm in section 5.1.3 with the initial index set to 0.

5.2 The ChewChen algorithm

Figure 5.4 shows the ChewChen algorithm which is my own implementation of the algorithm in section 5.1.3. Although Chew and Chen's algorithm can be implemented as a real-time algorithm, it was decided that the ChewChen algorithm should not be real-time as this would have made it harder to understand and describe.

5.2.1 The parameters of the ChewChen algorithm

The ChewChen algorithm has eleven parameters: **CCNoteList**, w_s, w_r, f, *AspectRatio*, *ChunkSize*, *InitialIndex*, *MinSAIndex*, *MaxSAIndex*, *StartOrSound* and *SAOrLOF*. The w_s, w_r and f parameters have the same meanings in ChewChen as they do in Eqs. 5.5, 5.6 and 5.8 above; *AspectRatio* gives the spiral array aspect ratio, r/h, as defined above; and the *ChunkSize* parameter simply specifies the duration of each chunk in ms.

CCNoteList is an ordered set of CCNote records representing the passage of music to be spelt. Each CCNote in **CCNoteList** represents a single note or sequence of tied notes in the music and contains seven fields as shown in Figure 5.3. Initially, each CCNote in **CCNoteList**

CCNote
Onset
Duration
MIDI
Index
SAVector
PitchName
ParentCCNote

Figure 5.3: The CCNote record type used in ChewChen.

has its Onset, Duration and MIDI fields set to the onset time (in ms), duration (in ms) and MIDI note number, respectively, of the note which it represents. The other four fields in each CCNote are initially set to nil.

As explained above, the notes in the first chunk are initially spelt so that their indices are as close as possible to some specified initial index which Chew and Chen (2005, p. 71) set to 2. The *InitialIndex* parameter is used to specify this initial index.

As explained by Chew and Chen (2003b, pp. 7–8; 2005, pp. 68–69), their algorithm is actually restricted to assigning pitch name classes within a specified range on the spiral array. Specifically, their own implementation can only assign pitch name classes whose indices are between -15 (F𝄫) and 19 (B𝄪), inclusive. In ChewChen, this range is specified by providing the minimum and maximum permitted spiral array indices in the parameters *MinSAIndex* and *MaxSAIndex*, respectively.

As explained above, in their own implementation, Chew and Chen calculate the CE for a window by considering the notes that sound in it, not the notes that start in it. They also weight each note by the duration for which it sounds within the window over which the CE is being calculated. However, it seems possible that good results might be obtainable by adopting the simpler strategy of considering the notes that start in a window and weighting each note by its entire duration. In the ChewChen algorithm, it is possible to adopt this latter strategy instead of that used by Chew and Chen, by setting the *StartOrSound* parameter to Starting instead of Sounding.

Finally, as pointed out above, although the spiral array is just "a spiral configuration of the line of fifths", Chew and Chen (2005, pp. 66-7) claim that the "depth added by going from one to three dimensions [i.e., from the line of fifths to the spiral array] allows the modeling of more complex hierarchical relations". However, in fact, it can be proved that replacing the spiral array with the line of fifths in Chew and Chen's pitch spelling algorithm never makes any difference to the spelling generated by the algorithm (see Appendix C). Nevertheless, in the ChewChen algorithm, the user can choose to use the line of fifths instead of the spiral array by setting the *SAOrLOF* parameter to LOF instead of SA.

5.2.2 The ChewChen algorithm in detail

In this section, I explain in some detail how the ChewChen algorithm works.

In line 1, the variable *NumberOfNotes* is set, for convenience, to equal the number of CCNotes in **CCNoteList**.

```
ChewChen(CCNoteList, w_s, w_r, f, AspectRatio, ChunkSize, InitialIndex,
        MinSAIndex, MaxSAIndex, StartOrSound, SAOrLOF)
1   NumberOfNotes ← |CCNoteList|
2   SA ← ConstructSpiralArray(MinSAIndex, MaxSAIndex, AspectRatio)
3   ChunkCCNoteLists ← ConstructChunkCCNoteLists(CCNoteList, ChunkSize, StartOrSound)
4   NumberOfChunks ← |ChunkCCNoteLists|
5   ChunkDurations ← ⨁_{i=0}^{NumberOfChunks−1} ⟨ Σ_{j=0}^{|ChunkCCNoteLists[i]|−1} ChunkCCNoteLists[i][j].Duration ⟩
6   Chunks ← ⟨ ⟩
7   for i ← 0 to NumberOfChunks − 1
8       Chunk ← MakeChunk
9       Chunk.CCNoteList ← ChunkCCNoteLists[i]
10      Chunk.Duration ← ChunkDurations[i]
11      Chunks ← Chunks ⊕ ⟨Chunk⟩
12  for j ← 0 to NumberOfChunks − 1
13      for Phase ← 0 to 1
14          if j > 0
15              if Phase = 0
16                  CE ← ComputeCE(Chunks[Max({j − w_s, 0}), j], SAOrLOF)
17              else
18                  CE_local ← ComputeCE(Chunks[Max({j − w_r + 1, 0}), j + 1], SAOrLOF)
19                  CE_cum ← ComputeCE(Chunks[0, j], SAOrLOF)
20                  CE ← nil
21                  if CE_local ∨ CE_cum
22                      if CE_local
23                          CE′_local ← f.CE_local
24                      else
25                          if SAOrLOF = SA
26                              CE′_local ← ⟨0, 0, 0⟩
27                          else
28                              CE′_local ← 0
29                      if CE_cum
30                          CE′_cum ← (1 − f).CE_cum
31                      else
32                          if SAOrLOF = SA
33                              CE′_cum ← ⟨0, 0, 0⟩
34                          else
35                              CE′_cum ← 0
36                      CE ← CE′_local + CE′_cum
37          if (j = 0) ∨ ((Phase = 0) ∧ (w_s ≠ 0)) ∨ ((Phase = 1) ∧ (not ((f = 1) ∧ (w_r = 0))))
38              for i ← 0 to |Chunks[j].CCNoteList| − 1
39                  PitchClass ← Chunks[j].CCNoteList[i].MIDI mod 12
40                  PossSANodes ← ⟨ ⟩
41                  for k ← 0 to |SA| − 1
42                      if PitchClass = SA[k].PitchClass
43                          PossSANodes ← PossSANodes ⊕ ⟨SA[k]⟩
44                  DistancesFromCE ← ⟨ ⟩
45                  if ((j = 0) ∧ (Phase = 0)) ∨ ((Phase = 0) ∧ (CE = nil))
46                      for k ← 0 to |PossSANodes| − 1
47                          DistancesFromCE ← DistancesFromCE ⊕ ⟨Abs(PossSANodes[k].Index − InitialIndex)⟩
48                  else
49                      if SAOrLOF = SA
50                          for k ← 0 to |PossSANodes| − 1
51                              DistancesFromCE ← DistancesFromCE ⊕ ⟨EuclideanDistance(PossSANodes[k].Vector, CE)⟩
52                      else
53                          for k ← 0 to |PossSANodes| − 1
54                              DistancesFromCE ← DistancesFromCE ⊕ ⟨Abs(PossSANodes[k].Index − CE)⟩
55                  MinDistFromCE ← Min(DistancesFromCE)
56                  BestPos ← Pos(MinDistFromCE, DistancesFromCE)
57                  BestSANode ← PossSANodes[BestPos]
58                  Chunks[j].CCNoteList[i].Index ← BestSANode.Index
59                  Chunks[j].CCNoteList[i].SAVector ← BestSANode.Vector
60              Chunks[j].SumPD ← ComputeChunkSumPD(Chunks[j], SAOrLOF)
61              if (j = 0) ∧ (Phase = 0)
62                  CE ← ComputeCE(Chunks[j, j + 1], SAOrLOF)
63  OutputList ← ⟨ ⟩
64  for j ← 0 to NumberOfChunks − 1
65      OutputList ← OutputList ⊕ Chunks[j].CCNoteList
66  if (StartOrSound = Sounding)
67      OutputList ← ComputeSoundingOutputList(OutputList, CCNoteList)
68  for i ← 0 to NumberOfNotes − 1
69      OutputList[i].PitchName ← MIDIIndex2PN(OutputList[i].MIDI, OutputList[i].Index)
70  return OutputList
```

Figure 5.4: The ChewChen algorithm.

SANode
  INDEX
  PITCHCLASS
  VECTOR

Figure 5.5: The SANode record type used in CHEWCHEN.

```
CONSTRUCTSPIRALARRAY(MinSAIndex, MaxSAIndex, AspectRatio)
1    SA ← ⟨ ⟩
2    for k ← MinSAIndex to MaxSAIndex
3        SANode ← MAKESANODE
4        SANode.INDEX ← k
5        SANode.PITCHCLASS ← (7k) mod 12
6        SANode.VECTOR ← ⟨AspectRatio. sin(kπ/2), AspectRatio. cos(kπ/2), k⟩
7        SA ← SA ⊕ ⟨SANode⟩
8    return SA
```

Figure 5.6: The CONSTRUCTSPIRALARRAY function.

In line 2, the function CONSTRUCTSPIRALARRAY (defined in Figure 5.6) is used to construct a representation of a segment of the spiral array. This representation is stored in the variable, **SA**, which is an ordered set of SANode records. Each SANode in **SA** represents one pitch name class in the spiral array and these SANodes are in increasing order of index within **SA**. Each SANode contains three fields: INDEX, PITCHCLASS and VECTOR, as shown in Figure 5.5. These fields give the spiral array index, the pitch class and the spiral array position vector, respectively, of the pitch name class represented by the SANode. The value of the VECTOR field within an SANode depends on the value of *AspectRatio* which is passed as an argument to CONSTRUCTSPIRALARRAY. The other two parameters of CONSTRUCTSPIRALARRAY are *MinSAIndex* and *MaxSAIndex* which determine the particular segment of the spiral array which is constructed.

The CONSTRUCTSPIRALARRAY function (see Figure 5.6) is fairly self-explanatory. The function MAKESANODE called in line 3 of CONSTRUCTSPIRALARRAY simply makes a new SANode and sets all its fields to nil. The value of the VECTOR field of each SANode is set in line 6 to equal the value on the right-hand side of Eq. 5.9 divided by h, that is,

$$\mathbf{p}'(k) = \left\langle \frac{r}{h}\sin(k\pi/2), \frac{r}{h}\cos(k\pi/2), k \right\rangle . \tag{5.10}$$

Note that the relative distances between the nodes in the spiral array are the same when the position vectors are calculated using Eq. 5.10 as they are when they are calculated using Eq. 5.9.

In lines 3–11 of CHEWCHEN, the music is divided up into chunks, each chunk being *ChunkSize*

Chunk
  CCNOTELIST
  DURATION
  SUMPD

Figure 5.7: The Chunk record type used in CHEWCHEN.

ms in duration. In a real-time implementation of the algorithm, this chunking process would be carried out "on the fly" with each new chunk being constructed just before the notes in it are spelt. However, in the ChewChen algorithm, the whole passage to be analysed is divided into chunks before any notes are spelt. Each chunk is represented within ChewChen by a Chunk record and the complete sequence of chunks is represented by a chronologically ordered set of Chunks which is stored in the variable **Chunks** (see lines 6–11 in Figure 5.4). Each Chunk contains three fields: CCNoteList, Duration and SumPD, as shown in Figure 5.7.

The CCNoteList field of a Chunk, C, is used to store an ordered set of CCNotes representing the notes either starting or sounding in the chunk represented by C. If *StartOrSound* is equal to Starting, then the CCNoteList field in each Chunk, C, in **Chunks** contains a CCNote record for each note that starts in the chunk represented by C and the Duration field of this CCNote gives the complete duration of the note it represents. However, if *StartOrSound* = Sounding, then the CCNoteList field in each Chunk, C, in **Chunks** contains a CCNote for each note that sounds in the chunk represented by C and the Duration field of this CCNote is set to equal the duration for which the note sounds within the chunk represented by C. The ParentCCNote field in each CCNote record, n', in the CCNoteList of each Chunk in **Chunks** is set to equal the position in **CCNoteList** of the CCNote from which n' was derived. The values of the CCNoteList fields of the Chunks in **Chunks** are computed in line 3 of ChewChen using a function ConstructChunkCCNoteLists (which will not be described in detail here) and stored temporarily in the variable **ChunkCCNoteLists**.

The Duration field of a Chunk, C, (see Figure 5.7) is used to store the sum of the values in the Duration fields of all the CCNotes in the CCNoteList field of C. When *StartOrSound* = Starting, this is simply the sum of the total durations of all the notes starting in C. However, when *StartOrSound* = Sounding, the value in the Duration field of a Chunk, C, should be the sum over all the notes that sound in C of the durations for which these notes sound within C. The values of the Duration fields of the Chunks in **Chunks** are computed in line 5 of ChewChen and stored temporarily in the variable **ChunkDurations**.

In lines 6–11 of ChewChen, the ordered sets **ChunkDurations** and **ChunkCCNoteLists** are used to construct the sequence of Chunks, **Chunks**. The function MakeChunk called in line 8 simply makes a new Chunk record and sets all its fields to nil. In **Chunks**, the CCNoteList and Duration fields of each Chunk are set to their correct values and the SumPD field of each Chunk is set to nil. The purpose of the SumPD field in a Chunk will be explained below.

The purpose of the 'for' loop in lines 12–62 of ChewChen is to compute the index of each note in the input passage using the algorithm described in section 5.1.3. This is done by assigning a value to the Index field of each CCNote in the CCNoteList field of each Chunk in **Chunks**. The Chunks in **Chunks** are processed in chronological order, one Chunk being processed on each iteration of the 'for' loop in lines 12–62. For each Chunk, lines 14–62 are executed twice: once with *Phase* = 0 and a second time with *Phase* = 1. When *Phase* = 0, lines 14–62 implement "Phase I" (i.e., steps 1 and 2) of the algorithm in section 5.1.3. When *Phase* = 1, lines 14–62 implement "Phase II" (i.e., steps 3–5) of the algorithm in section 5.1.3. For each of the two phases and each chunk, **Chunks**[j], the appropriate center of effect, **CE**, is computed in lines 15–36 and then, in lines 38–59, the Index and SAVector fields of each CCNote in the

CCNoteList field of **Chunks**[j] are made equal to the index and spiral array position vector, respectively, of the pitch name to be assigned to the note represented by the CCNote.

As explained in section 5.1.3, the first chunk has to be processed in a different way from the others. Let's suppose that lines 14–62 are about to be executed for the first time, so $j = 0$ and *Phase* $= 0$ in line 14. Lines 15–36 are skipped since j is not greater than 0 in line 14: no notes have been spelt yet, so no CE can be calculated. However, as $j = 0$ in line 37, lines 38–62 *are* executed. The 'for' loop in lines 38–59 iterates once for each CCNote in the CCNoteList field of **Chunks**[0], the Chunk that represents the first chunk. In line 39, the pitch class of the current CCNote, **Chunks**[0].**CCNoteList**[i], is stored in the variable *PitchClass*. Then, in lines 40–43, the spiral array segment, **SA**, is searched for SANodes that have the pitch class, *PitchClass*, and these SANodes are stored in the variable **PossSANodes**. At this point, therefore, **PossSANodes** contains all and only those SANodes in **SA** whose indices could possibly be assigned to **Chunks**[0].**CCNoteList**[i]. As $j = 0$, the next step is to find the SANode in **PossSANodes** whose index is closest to the value of *InitialIndex*. This is done by calculating the absolute distance along the line of fifths from *InitialIndex* to each of the SANodes in **PossSANodes** (lines 45–47) and then choosing the SANode, *BestSANode*, which is closest to *InitialIndex* (lines 55–57). The Index and SAVector fields of **Chunks**[0].**CCNoteList**[i] are then set to equal the values in the Index and SAVector fields, respectively, of *BestSANode* (lines 58–59). Once all the notes in the first chunk have been assigned indices that are as close as possible to *InitialIndex*, the SumPD field of the first chunk is set to equal the weighted sum of the spiral array position vectors (or indices if *SAOrLOF* = LOF) of the notes in the first chunk, each note being weighted by its duration (line 60).

As explained in section 5.1.3, once indices as close as possible to *InitialIndex* have been assigned to the notes in the first chunk, a CE for this first chunk is then computed. This is done in line 62 and the result is stored in the variable **CE**. The notes in the first chunk are then respelt so that they are as close as possible to **CE**. This respelling is done in lines 38–59 on the second iteration of lines 14–62 for the first chunk, during which *Phase* $= 1$. On this second iteration, the values of *PitchClass* and **PossSANodes** calculated in lines 39–43 are the same as on the first iteration. However, as *Phase* is now equal to 1, lines 49–54 are executed instead of lines 46–47. If the spiral array is being used (i.e., *SAOrLOF* = SA), then the Euclidean distance from each SANode in **PossSANodes** to **CE** is calculated in lines 50–51 using the function EuclideanDistance. These distances are stored in **DistancesFromCE**. Otherwise, if the line of fifths is used in place of the spiral array, **DistancesFromCE** is used to store the absolute difference between the index of each SANode in **PossSANodes** and the weighted mean line-of-fifths position for the first chunk stored in **CE** (lines 53–54). In lines 55–57, *BestSANode* is set to equal the SANode in **PossSANodes** that is closest to **CE** and then, in lines 58–59, the Index and SAVector of the current CCNote are set to equal those in *BestSANode*. In line 60, the value of **Chunks**[0].SumPD is updated to reflect the reassigned pitch names.

Let's suppose now that the ChewChen algorithm is about to process the $(j + 1)$th chunk (i.e., **Chunks**[j]) where $j > 0$. In other words, lines 14–62 are about to be executed for the $(2j + 1)$th time ($j > 0$) and *Phase* $= 0$. Let's further suppose for the moment that $w_s \neq 0$, $w_r \neq 0$ and $0 < f < 1$. Since *Phase* $= 0$, lines 14–62 must implement Phase I (i.e., steps 1 and

2) of the algorithm in section 5.1.3. When $Phase = 0$, line 16 implements step 1 by setting the variable **CE** to equal the global CE of the w_s chunks preceding **Chunks**[j]. This CE is computed using a function, ComputeCE, that implements the method described in section 5.1.2 above. If $j < w_s$ in line 16, then the CE of all the chunks preceding **Chunks**[j] is computed. The value returned by the ComputeCE function depends on whether the line of fifths or the spiral array is being used to compute the CE. If the line of fifths is being used (i.e., $SAOrLOF = \mathsf{LOF}$), then ComputeCE returns the weighted mean of the indices of the notes in the window (i.e., the segment of **Chunks**) given to it as its first argument. However, if the spiral array is being used (i.e., $SAOrLOF = \mathsf{SA}$), then the value returned by ComputeCE is the weighted centroid of the spiral array position vectors of the notes in the window given to the function as its first argument. If the window given to ComputeCE as its first argument contains no notes, then it returns nil.

Having computed the global CE, step 2 of the algorithm in section 5.1.3 is then implemented in lines 38–62 of ChewChen when $Phase = 0$. For each note, **Chunks**[j].CCNoteList[i], in the Chunk, **Chunks**[j], the algorithm finds the SANode in **SA** closest to **CE** that has the same pitch class as **Chunks**[j].CCNoteList[i] (lines 39–56), stores this SANode in *BestSANode* (line 57) and then sets the values of the Index and SAVector of **Chunks**[j].CCNoteList[i] to equal those of *BestSANode* (lines 58–59). If **CE** is nil in line 45, then the notes in the chunk are spelt so that their indices are as close as possible to *InitialIndex*, as is done for the first chunk when $Phase = 0$ (see above). In line 60, having assigned pitch name classes as close as possible to **CE** to all the notes in the chunk represented by **Chunks**[j], the SumPD field of **Chunks**[j] is set to equal the weighted sum of either the indices (if $SAOrLOF = \mathsf{LOF}$) or the spiral array vectors (if $SAOrLOF = \mathsf{SA}$) of the notes in **Chunks**[j].CCNoteList. This is used later by the ComputeCE function to simplify the process of computing the CE of any window that contains **Chunks**[j].

Having completed Phase I of the algorithm in section 5.1.3 for the chunk represented by **Chunks**[j], Phase II (i.e., steps 3–5) is then carried out by executing lines 14–62 with $Phase = 1$. When $Phase = 1$, lines 18–36 of ChewChen compute the hybrid CE defined in Eq. 5.8 and then lines 38–59 spell the notes in the chunk so that they are as close as possible to this hybrid CE.

The first step in Phase II of the algorithm is to compute the local CE (see step 3 of the algorithm in section 5.1.3). This local CE is computed in line 18 and stored in the variable $\mathbf{CE}_{\text{local}}$. Then, line 19 implements step 4 of the algorithm by computing the cumulative CE and storing it in the variable $\mathbf{CE}_{\text{cum}}$. If both $\mathbf{CE}_{\text{local}}$ and $\mathbf{CE}_{\text{cum}}$ are nil, then the hybrid CE will also be nil, so the variable **CE**, which is used to store this hybrid CE, is set to nil in line 20. If $\mathbf{CE}_{\text{local}}$ and/or $\mathbf{CE}_{\text{cum}}$ are not nil, then lines 22–36 are executed and the value of **CE** is updated. If the local CE, $\mathbf{CE}_{\text{local}}$, is not nil, then the first term, "$f.CE_{\text{local},j}$", in the hybrid CE equation, Eq. 5.8, is computed in line 23 and stored in the variable $\mathbf{CE}'_{\text{local}}$. If $\mathbf{CE}_{\text{local}}$ *is* nil, then $\mathbf{CE}'_{\text{local}}$ is set to equal either the zero spiral array position vector if $SAOrLOF = \mathsf{SA}$ (line 26), or simply 0 if $SAOrLOF = \mathsf{LOF}$ (line 28). This ensures that $\mathbf{CE}_{\text{local}}$ makes no contribution to the hybrid CE if it is nil. Similarly, if the cumulative CE, $\mathbf{CE}_{\text{cum}}$ is not nil, then the second term, "$(1-f).CE_{\text{cum},j}$", in the hybrid CE equation, Eq. 5.8, is computed in line 30 and stored in the

variable $\mathbf{CE}'_{\text{cum}}$. If $\mathbf{CE}_{\text{cum}}$ *is* nil, then $\mathbf{CE}'_{\text{cum}}$ is set to equal either the zero spiral array position vector if $SAOrLOF = \mathsf{SA}$ (line 33), or simply 0 if $SAOrLOF = \mathsf{LOF}$ (line 35). The hybrid CE is then computed in line 36 and stored in the variable **CE**. To complete Phase II, the notes in the chunk represented by **Chunks**$[j]$ are spelt in lines 38–59 so that they are as close as possible to the hybrid CE stored in **CE**. Finally, the SumPD field of **Chunks**$[j]$ is updated in line 60.

Setting w_s to 0 in the algorithm in section 5.1.3 causes Phase I (i.e., steps 1 and 2) to be skipped. This means that no local CE can be calculated since the notes in the chunk currently being spelt have not been assigned preliminary pitch names in Phase I. This means that f must be less than 1 so that the pitch names can be assigned in accordance with the cumulative CE. Therefore, if $w_s = 0$, f must be less than 1 and the net effect is for the notes to be spelt so that they are as close as possible to the cumulative CE.

Let us therefore consider what happens in lines 13–62 of ChewChen when we process the Chunk, **Chunks**$[j]$, and $w_s = w_r = 0$. For the first chunk, the spelling process is the same, regardless of the values of w_s, w_r and f, because lines 15–36 are omitted. If $j > 0$ and $Phase = 0$, then the variable **CE** is set to nil in line 16 because the window over which the CE is computed is empty and lines 38–62 are skipped because both *Phase* and w_s are zero in line 37. If $j > 0$ and $Phase = 1$, then $\mathbf{CE}_{\text{local}}$ becomes nil in line 18 so $\mathbf{CE}'_{\text{local}}$ becomes either 0 or the zero spiral array position vector in lines 25–28 and the hybrid CE, **CE**, becomes $\mathbf{CE}'_{\text{cum}}$ which is equal to the cumulative CE, $(1 - f).\mathbf{CE}_{\text{cum}}$. However, the cumulative CE, $\mathbf{CE}_{\text{cum}}$, computed in line 19, will not be nil (assuming the first chunk contains notes), so, if $f < 1$, $\mathbf{CE}'_{\text{cum}}$ will be set to equal some non-zero scalar multiple of $\mathbf{CE}_{\text{cum}}$ in line 30 and the effect will be the same as if the CE, **CE**, is set to equal $\mathbf{CE}_{\text{cum}}$ in line 36. Consequently, if $w_s = w_r = 0$ and $f < 1$, the notes in each chunk will be spelt so that they are as close as possible to the cumulative CE, as in the algorithm described by Chew and Chen (2003b) and the "Cumulative CE" algorithm ("Algorithm 1") described by Chew and Chen (2003a) (see Table 5.1). If f is set to 1 when $w_s = w_r = 0$ then lines 38–62 are never executed and no notes are spelt.

If $w_s > 0$ and $w_r = 0$ in the algorithm described in section 5.1.3, then Phase I (i.e., steps 1 and 2) of the algorithm is carried out, but no local CE is computed in step 3. If $f < 1$ when $w_r = 0$, then in Phase II the notes are spelt so that they are as close as possible to the cumulative CE. However, if $f = 1$ when $w_r = 0$, then Phase II is omitted altogether and the notes in each chunk are spelt so that they are as close as possible to the global CE, making the algorithm equivalent to the "sliding window" algorithm ("Algorithm 2") described by Chew and Chen (2003a).

Let's therefore consider what happens in lines 13–62 of ChewChen when we process the Chunk, **Chunks**$[j]$, with $w_r = 0$, $f < 1$ and $w_s > 0$. The spelling for the first chunk is independent of w_s, w_r and f, so let's assume that the first chunk is not empty and that $j > 0$. Phase I of the algorithm, implemented in lines 14–62 when $Phase = 0$, is carried out as described above for the case when $w_s \neq 0$, $w_r \neq 0$ and $0 < f < 1$. However, when $Phase = 1$, $\mathbf{CE}_{\text{local}}$ is set to nil in line 18, corresponding to the fact that step 3 is omitted from the algorithm in section 5.1.3 when $w_r = 0$. However, $\mathbf{CE}_{\text{cum}}$ will not be nil so, since $f < 1$, **CE** will be set to equal $\mathbf{CE}_{\text{cum}}$ in line 36. Consequently, when $w_r = 0$, $f < 1$ and $w_s > 0$ in ChewChen, the notes in each chunk are first spelt so that they are as close as possible to the global CE and then

respelt so that they are as close as possible to the cumulative CE. In other words, if $w_r = 0$, $f < 1$ and $w_s > 0$ in ChewChen, the algorithm effectively implements the cumulative CE version of Chew and Chen's algorithm (i.e., the version described by Chew and Chen (2003b) and the "Cumulative CE" algorithm ("Algorithm 1") of Chew and Chen (2003a)).

When $w_r = 0$, $w_s > 0$ and $f = 1$, Phase I of the algorithm, implemented in lines 14–62 when $Phase = 0$, is carried out as described above for the case when $w_s \neq 0$, $w_r \neq 0$ and $0 < f < 1$. And then, when $Phase = 1$, lines 38–62 are omitted because $Phase = 1$, $w_r = 0$, and $f = 1$ in line 37. Consequently, the notes in each chunk are spelt so that they are as close as possible to the global CE calculated when $Phase = 0$ for each chunk, thus implementing the "sliding window" algorithm ("Algorithm 2") described by Chew and Chen (2003a).

If $f = 0$ and both w_s and w_r are greater than 0 in the algorithm in section 5.1.3, then the notes in each chunk are spelt in Phase I so that they are as close as possible to the global CE. Then, in Phase II, the notes are spelt so that they are as close as possible to the cumulative CE and the spellings assigned to the notes in Phase I are overridden. Similarly, in ChewChen, if $f = 0$, **CE** is set to equal the cumulative CE, $\mathbf{CE}_{\text{cum}}$, when $Phase = 1$ and the notes within each chunk are spelt so that they are as close as possible to this value of **CE**. Therefore, with $f = 0$ and both w_s and w_r greater than 0, ChewChen implements the cumulative CE version of the algorithm (i.e., the version described by Chew and Chen (2003b) and the "Cumulative CE" algorithm ("Algorithm 1") of Chew and Chen (2003a)).

Finally, if $f = 1$ and both w_s and w_r are greater than 0 ($w_s \geq w_r$) in the algorithm in section 5.1.3, then in Phase I the notes are spelt so that they are as close as possible to the global CE. Then, in Phase II, the notes are respelt so that they are as close as possible to the local CE. Similarly, in ChewChen, when $Phase = 0$, the notes in each chunk are spelt so that they are as close as possible to the global CE calculated in line 16. Then, when $Phase = 1$, **CE** is set in line 36 to equal the local CE, $\mathbf{CE}_{\text{local}}$, calculated in line 18, and the notes within each chunk are respelt so that they are as close as possible to this CE.

By the time execution reaches line 63 in ChewChen, every note in every chunk has been assigned an index and a spiral array position vector. In lines 63–70, this information is used to assign a pitch name to each note in the input data. As a first step in doing this, the CCNoteList fields of the Chunks in **Chunks** are concatenated in lines 64–65 to create a list of CCNotes which is stored in the variable **OutputList**. If *StartOrSound* = Starting, each CCNote in **OutputList** will correspond to the CCNote in the same position in the input list, **CCNoteList**. Therefore, the required output can be generated by using the values in the MIDI and Index fields of each CCNote in **OutputList** to compute a pitch name for that CCNote which is stored in its PitchName field. This is done in lines 68–69 using the function MIDIIndex2PN defined in Figure 5.8.

However, if *StartOrSound* = Sounding, then **OutputList** will, in general, contain *at least* one CCNote for each CCNote in the input **CCNoteList**. However, the ParentCCNote field in each CCNote, n', in **OutputList** gives the position of the CCNote in **CCNoteList** from which n' was derived, and this information is used by the ComputeSoundingOutputList function in line 67 to generate a new version of **OutputList** in which each CCNote corresponds to the CCNote in the same position in **CCNoteList**. If two or more CCNotes in **OutputList** that

```
MIDIIndex2PN(MIDI, Index)
1   i ← ⌊(1 + Index)/7⌋
2   p′ ← MIDI − i
3   o ← ⌊p′/12⌋ − 1
4   if i > 0
5       c ← "s"
6   else
7       c ← "f"
8   i_str ← ""
9   for j ← 1 to Abs(i)
10      i_str ← i_str ⊕ c
11  if i = 0
12      i_str ← "n"
13  o_str ← Num2Str(o)
14  l_str ← ⟨"C", "D", "E", "F", "G", "A", "B"⟩ [(4 × Index) mod 7]
15  return l_str ⊕ i_str ⊕ o_str
```

Figure 5.8: The MIDIIndex2PN function.

are derived from the same CCNote, n, in **CCNoteList** are assigned different indices, then the index of n is assumed (arbitrarily) to be that assigned to the chronologically first CCNote in **OutputList** derived from n.

5.3 Computational complexity of Chew and Chen's algorithm

Let C_i denote the ith chunk and let $|C_i|$ denote the number of notes (either sounding or starting) in C_i. If we want to denote that a note n is "in" chunk C_i then we can write $n \in C_i$, regardless of whether we mean that the note sounds in C_i or starts in it. Let's suppose that there are N_C chunks in total. It is possible to implement the algorithm so that it runs in $O\left(\sum_{i=1}^{N_C} |C_i|\right)$ time in the worst case, as I shall now show.

If we assume that the algorithm has been implemented as a real-time program, then the total time taken by the algorithm to process a passage is $\sum_{i=1}^{N_C} t_i$ where t_i is the time taken to process the ith chunk, C_i. Processing C_i involves carrying out the 5 steps described in section 5.1.3. However, if the algorithm is to run in linear time, we also have to calculate a number of values for each chunk and store these values in such a way that they can be accessed in constant time.

Let $CE(a, b)$ denote the center of effect of the segment consisting of chunks C_a to C_b, inclusive. From the definition of center of effect given in Eq. 5.1, it is clear that

$$CE(a, b) = \frac{S(a, b)}{D(a, b)} \tag{5.11}$$

where

$$S(a, b) = \sum_{i=a}^{b} \left(\sum_{n \in C_i} d(n).\mathbf{p}(n) \right) \tag{5.12}$$

and

$$D(a,b) = \sum_{i=a}^{b} \left(\sum_{n \in C_i} d(n) \right). \tag{5.13}$$

It follows that

$$\begin{aligned} S(a+1,b+1) &= S(a,b) - S(a,a) + S(b+1,b+1), & (5.14)\\ S(a,b+1) &= S(a,b) + S(b+1,b+1), & (5.15)\\ D(a+1,b+1) &= D(a,b) - D(a,a) + D(b+1,b+1), & (5.16)\\ D(a,b+1) &= D(a,b) + D(b+1,b+1). & (5.17) \end{aligned}$$

Let us further suppose that, for all $1 \leq j < i$ we can access the following eight values in constant time:

$$\begin{aligned} S_{\text{glob},j} &= S(j - w'_s, j-1), & (5.18)\\ D_{\text{glob},j} &= D(j - w'_s, j-1), & (5.19)\\ S_{\text{cum},j} &= S(1, j-1), & (5.20)\\ D_{\text{cum},j} &= D(1, j-1), & (5.21)\\ S_{\text{local},j} &= S(j - w'_r + 1, j), & (5.22)\\ D_{\text{local},j} &= D(j - w'_r + 1, j), & (5.23)\\ S_j &= S(j,j), & (5.24)\\ D_j &= D(j,j). & (5.25) \end{aligned}$$

where $w'_s = \textsc{Min}(\{w_s, j-1\})$ and $w'_r = \textsc{Min}(\{w_r, j\})$. For each j, all these values can be calculated and stored as soon as the notes in chunk C_j have been spelt.

Processing C_i involves carrying out the 5 steps described in section 5.1.3 and calculating the eight values $S_{\text{glob},i}$, $D_{\text{glob},i}$, $S_{\text{cum},i}$, $D_{\text{cum},i}$, $S_{\text{local},i}$, $D_{\text{local},i}$, S_i and D_i. We therefore need to calculate the time t_i required to do this.

Step 1 of the algorithm in section 5.1.3 involves calculating the global CE for chunk C_i which is equal to $CE(i - w'_s, i-1)$ where $w'_s = \textsc{Min}(\{w_s, i-1\})$. From Eqs. 5.11, 5.18 and 5.19 it follows that

$$CE(i - w'_s, i-1) = \frac{S(i - w'_s, i-1)}{D(i - w'_s, i-1)} = \frac{S_{\text{glob},i}}{D_{\text{glob},i}}.$$

If $i - 1 > w_s$, then, from Eq. 5.14,

$$\begin{aligned} S_{\text{glob},i} &= S(i - w'_s - 1, i-2) - S(i - w'_s - 1, i - w'_s - 1) + S(i-1, i-1)\\ &= S_{\text{glob},i-1} - S_{i-w'_s-1} + S_{i-1} \end{aligned}$$

which can be computed in constant time. Also, if $i - 1 \leq w_s$, then, from Eq. 5.15,

$$\begin{aligned} S_{\text{glob},i} &= S(i - w'_s - 1, i-2) + S(i-1, i-1)\\ &= S_{\text{glob},i-1} + S_{i-1} \end{aligned}$$

which can also be computed in constant time. By an almost identical argument, it can be shown that $D_{\text{glob},i}$ can be computed in constant time which implies that only $O(1)$ time is required to execute step 1 for C_i and calculate the values of $S_{\text{glob},i}$ and $D_{\text{glob},i}$.

Applying step 2 of the algorithm in section 5.1.3 to chunk C_i involves assigning pitch name classes to the notes in C_i that are as close as possible to the global CE calculated in step 1. This takes $O(|C_i|)$ time in the worst case.

Applying step 3 of the algorithm to chunk C_i involves calculating the local CE which is given by

$$CE(i - w'_r + 1, i) = \frac{S(i - w'_r + 1, i)}{D(i - w'_r + 1, i)} = \frac{S_{\text{local},i}}{D_{\text{local},i}}$$

where $w'_r = \textsc{Min}(\{w_r, i\})$. If $i > w_r$, then, from Eq. 5.14,

$$\begin{aligned} S_{\text{local},i} &= S(i - w'_r, i - 1) - S(i - w'_r, i - w'_r) + S(i, i) \\ &= S_{\text{local},i-1} - S_{i-w'_r} + S_i \end{aligned}$$

which can be computed in constant time. Also, if $i \leq w_r$, then, from Eq. 5.15,

$$\begin{aligned} S_{\text{local},i} &= S(i - w'_r, i - 1) + S(i, i) \\ &= S_{\text{local},i-1} + S_i \end{aligned}$$

which can also be computed in constant time. By an almost identical argument, it can be shown that $D_{\text{local},i}$ can be computed in constant time which implies that only $O(1)$ time is required to execute step 3 for C_i and calculate the values of $S_{\text{local},i}$ and $D_{\text{local},i}$.

Applying step 4 of the algorithm in section 5.1.3 to chunk C_i involves calculating the cumulative CE which is given by

$$CE(1, i - 1) = \frac{S(1, i - 1)}{D(1, i - 1)} = \frac{S_{\text{cum},i}}{D_{\text{cum},i}}.$$

From Eq. 5.15 it follows that

$$\begin{aligned} S_{\text{cum},i} &= S(1, i - 2) + S(i - 1, i - 1) \\ &= S_{\text{cum},i-1} + S_{i-1} \end{aligned}$$

which can be computed in constant time. By an almost identical argument, it can be shown that $D_{\text{cum},i}$ can be computed in constant time which implies that only $O(1)$ time is required to execute step 4 for C_i and calculate the values of $S_{\text{cum},i}$ and $D_{\text{cum},i}$.

Applying step 5 of the algorithm in section 5.1.3 to chunk C_i involves spelling the $|C_i|$ notes in chunk C_i so that they are as close as possible to a hybrid CE which is a weighted sum of the local and global CEs already calculated. This can clearly be done in $O(|C_i|)$ time.

Finally, S_i and D_i have to be computed. This can also be done in $O(|C_i|)$ time.

Every step involved in processing chunk C_i can therefore be carried out in at worst $O(|C_i|)$ time. It follows that the worst case time complexity of the complete algorithm is $O\left(\sum_{i=1}^{N_C} |C_i|\right)$. If the CEs are calculated by considering only the notes that start within each chunk and each note

CEs to switch off	*Parameter values*	*Net effect*
Global	$w_s = 0$	Cumulative
Local	$w_r = 0 \vee f = 0$	Cumulative
Cumulative	$f = 1$	Global & Local
Global & Local	$w_s = 0$	Cumulative
Global & Cumulative	$w_s = 0 \wedge f = 1$	Nothing
Local & Cumulative	$w_r = 0 \wedge f = 1$	Global
Global, Local & Cumulative	$w_s = 0 \wedge f = 1$	Nothing

Table 5.2: Combination of values for w_s, w_r and f required for switching off each combination of the CE types (global, local and cumulative).

is weighted by its complete duration, then $\sum_{i=1}^{N_C} |C_i|$ is equal to the total number of notes, N_N, and the time complexity is linear in the number of notes. However, if the CEs are calculated by considering the notes that sound within each chunk and each note within each chunk is weighted by the duration for which it sounds within that chunk, then $\sum_{i=1}^{N_C} |C_i|$ may be as large as $N_C \times N_N$ in the pathological case where every note sounds in every chunk.

The algorithm requires $O(N_C)$ space since it has to store at least 8 values for every chunk in such a way that these values can be accessed in constant time.

5.4 Switching on and off the local, global and cumulative CEs

As should by now be apparent, the parameters w_s, w_r and f can be used to "switch off" one or more of the three types of CE (global, local or cumulative) that Chew and Chen's algorithm uses to spell the notes. Table 5.2 gives the parameter values required to switch off each combination of these three types of CE as well as the net effect of doing so.

Thus, if the global CE is switched off by setting w_s to 0, no names are assigned in Phase I, so no local CE can be computed in Phase II. Consequently, switching off the global CE has the net effect of causing the notes to be spelt as close as possible to the cumulative CE (see first row in Table 5.2).

The local CE can be switched off by setting either w_r or f to 0. Either way, the notes are spelt as close as possible to the cumulative CE in Phase II, overriding the global CE spellings assigned in Phase I (see second row in Table 5.2).

The cumulative CE can be switched off by setting f to 1. This causes the global CE spellings assigned in Phase I to be modified in Phase II using the local CE (see third row of Table 5.2).

Both the global and local CEs are effectively switched off by setting w_s to 0, since switching off the global CE in Phase I means that the local CE cannot be computed in Phase II. In this case, therefore, the notes are spelt so that they are as close as possible to the cumulative CE in Phase II (see fourth row in Table 5.2).

The fact that the global CE cannot be switched off without also switching off the local CE means that, if both the global and cumulative CEs are switched off, no notes are spelt whatsoever (see fifth row in Table 5.2).

To switch off both the local and cumulative CEs (i.e., omit Phase II), w_r must be set to 0 and f must be set to 1 (see sixth row in Table 5.2). The net result is that the notes are spelt so that they are as close as possible to the global CE.

Finally, all the CEs can be switched off by setting w_s to 0 and f to 1. Setting w_s to 0 switches off the global CE and therefore the local CE; setting f to 1 switches off the cumulative CE (see seventh row in Table 5.2).

There are therefore just four possible CE combinations in practice:

1. all three CEs have an effect ($w_s \neq 0 \wedge w_r \neq 0 \wedge 0 < f < 1$);
2. just the cumulative CE is used ($f = 0 \vee (0 < f < 1 \wedge (w_r = 0 \vee w_s = 0))$);
3. just the global CE is used ($f = 1 \wedge w_r = 0 \wedge w_s > 0$); or
4. only the global and local CEs have an effect ($f = 1 \wedge w_r > 0 \wedge w_s > 0$).

5.5 Chew and Chen's own evaluations of their algorithms

Table 5.3 summarises the results reported by Chew and Chen in their publications. As discussed in section 1.3.3.2 above, the test corpus used by Chew and Chen (2003a,b) consisted of just two movements from Beethoven's piano sonatas, while Chew and Chen (2005) ran the algorithm on the *Song of Ali-Shan*, a set of variations on a Taiwanese folksong by You-Di Huang, in addition to the two Beethoven piano sonata movements. It is hard to see how one could justify generalising from the results obtained on such a small corpus to some interesting larger population of tonal works.

Chew and Chen (2005, p. 75) claim that, over all three pieces, the algorithm makes 28 errors when $\langle w_s, w_r, f\rangle$ is either $\langle 4, 3, 0.8\rangle$ or $\langle 8, 6, 0.9\rangle$. However, it seems that only the cumulative CE version of their algorithm was run on the third movement of Beethoven's Sonata in G major, Op. 79. If this is indeed the case, then the results for this movement cannot meaningfully be combined with the results obtained when the two-phase version of the algorithm was run on the other two movements in the test corpus. Therefore, Chew and Chen's claim that their algorithm made 28 errors over all three movements may be invalid.

Chew and Chen (2005, p. 71) state that they "set the chunk size to one beat" and that "the beat size was read from the MIDI files". It is therefore possible that the chunks did not all have the same duration in ms in all three of the movements in Chew and Chen's test corpus.

As can be seen in Table 5.3, in Chew and Chen's experiments, the least number of errors on the first movement of Beethoven's Op. 109 were made when the two-phase algorithm was used and $\langle w_s, w_r, f\rangle$ were set to $\langle 4, 3, 0.8\rangle$, $\langle 4, 3, 0.7\rangle$ or $\langle 8, 6, 0.9\rangle$. With these settings, the algorithm made 27 errors on this movement. Chew and Chen (2005, p. 74) state that "the next best result had 30 errors using the parameters $(8, 6, 0.8)$ and $(16, 6, 0.8)$" and claim that "since the best results were achieved with high values for f, we can deduce that the local context is more important than the global [i.e., cumulative] context". However, the second lowest note error count on the first movement of Beethoven's Op. 109 was actually 28 errors and was made when the parameters $\langle w_s, w_r, f\rangle$ were set to $\langle 4, 2, 0.6\rangle$—that is, with f set to the lowest value tested

Publication	*Algorithm*	w_s	w_r	f	*Note error count*	*Note accuracy (%)*	*Test corpus*	*Number of notes*
Chew and Chen (2003b)	Cumulative CE	0	0	0	1	99.93	Beethoven, Piano Sonata in G major, Op. 79, 3rd. mvt.	1375
					73	95.18	Beethoven, Piano Sonata in E major, Op. 109, 1st. mvt.	1516
					74	97.44	Beethoven, Piano Sonata in E major, Op. 109, 1st. mvt. Beethoven, Piano Sonata in G major, Op. 79, 3rd. mvt.	2891
Chew and Chen (2003a)	Sliding window	4		1	31	98.00	Beethoven, Piano Sonata in E major, Op. 109, 1st. mvt.	1516
		8			47	96.90		
		16			40	97.36		
	Two-phase	4	2	0.6	28	98.15		
			3	0.8	27	98.22		
				0.7				
		8	2	0.9	31	97.96		
			4		40	97.36		
			6		27	98.22		
		16	4	0.8	40	97.36		
			6		30	98.02		
			8	0.9	37	97.56		
Chew and Chen (2005)		4	3		32	97.89		
		8	2	0.8	40	97.36		
				0.7				
			4	0.8				
				0.7	43	97.16		
			6	0.8	30	98.02		
				0.7	47	96.90		
		16	4		40	97.36		
			6	0.9	31	97.96		
			8	0.7	41	97.30		
	Cumulative CE	0	0	0	0	100.00	You-Di Huang, "The Song of Ali-Shan"	1571
	Sliding window	4		1				
		8						
		16						
	Two-phase	4	3	0.8				
		8	6	0.9				
		16		0.8				

Table 5.3: Summary of results obtained by Chew and Chen in their own evaluations of their algorithms.

Parameter	*Set of values used*
w_s	$\{0, 4, 8, 16\}$
w_r	$\{0, 2, 4, 6\}$
f	$\{0, 0.25, 0.5, 0.75, 1\}$
AspectRatio	$\left\{\sqrt{2/15}, \sqrt{15/2}\right\}$
ChunkSize	$\{500, 1000, 2000\}$
StartOrSound	$\{\mathsf{Sounding}, \mathsf{Starting}\}$
SAOrLOF	$\{\mathsf{SA}, \mathsf{LOF}\}$
$\langle MinSAIndex, MaxSAIndex\rangle$	$\{\langle -15, 19\rangle, \langle -22, 26\rangle\}$

Table 5.4: The sets of parameter values used to evaluate Chew and Chen's algorithm.

in Chew and Chen's experiments. This clearly casts some doubt on Chew and Chen's (2005, p. 74) claim that the local context is more important than the cumulative context.

However, the main point to be made here is that, because of the very small size of the test corpus and the small range of parameter values tested in Chew and Chen's experiments, any conclusions drawn from their results must be considered tentative at best.

5.6 A more thorough evaluation of Chew and Chen's algorithm

5.6.1 Method

A more thorough evaluation of Chew and Chen's algorithm was therefore carried out in which the ChewChen algorithm described in section 5.2 was run 1296 times on the test corpus $\mathcal{C}$ defined in Table 1.4, each time with a different combination of parameter values. Table 5.4 shows, for each parameter, the set of values used in the evaluation.

First, all parameter value combinations in the cross product of the sets in the right-hand column of Table 5.4 were generated to give a set, $\mathcal{P}$. Then a reduced set, $\mathcal{P}'$, of parameter value combinations was generated by removing all elements from $\mathcal{P}$ except

1. those in which the global, local and cumulative CEs all have an effect (i.e., those for which $w_s \neq 0 \wedge w_r \neq 0 \wedge 0 < f < 1$);

2. those that have one particular combination of values for w_s, w_r and f that causes only the cumulative CE to be used (the combination $f = 0$, $w_r = 2$ and $w_s = 4$ was used as the results for this combination had already been obtained in an earlier experiment);

3. those in which only the global CE is used (i.e., those in which $f = 1 \wedge w_r = 0 \wedge w_s > 0$); and

4. those in which only the global and local CEs have an effect ($f = 1 \wedge w_r > 0 \wedge w_s > 0$)

(see discussion in section 5.4).

A subset of $\mathcal{P}'$, which I shall denote by $\mathcal{P}''$ was then generated containing all the parameter value combinations in $\mathcal{P}'$ except

1. those in which $w_r > w_s + 1$ (except for the chunk currently being spelt, the local context window should not contain chunks that are not in the global context window); and

2. those in which $SAOrLOF = \mathsf{LOF}$ and $AspectRatio = \sqrt{15/2}$ (when $SAOrLOF = \mathsf{LOF}$, the *AspectRatio* parameter has no effect so there is no point in using more than one "dummy" value of *AspectRatio* in this case).

The ChewChen algorithm was run on the test corpus, $\mathcal{C}$, defined in Table 1.4 with each of the 1296 parameter value combinations in $\mathcal{P}''$.

5.6.2 Results and discussion

5.6.2.1 Parameter value combinations achieving highest note accuracy

The highest overall note accuracy achieved by the ChewChen algorithm in this experiment on the test corpus $\mathcal{C}$ was 99.15%. The algorithm achieved this note accuracy with 12 of the 1296 parameter value combinations in $\mathcal{P}''$. The 12 parameter value combinations with which ChewChen achieved this highest note accuracy were those in which $w_s = 8$, $w_r = 2$, $f = 0.5$ and *ChunkSize* was set to 500 milliseconds. That is, ChewChen performed best when

1. the local, global and cumulative CEs all had an effect;

2. the local context window was as small as possible;

3. the global context window was a moderate size;

4. the local and cumulative CEs were given equal weighting in Phase II; and

5. the chunks were small, leading to a frequent updating of the CEs.

The parameters that were critical for achieving the highest note accuracy were therefore those controlling the duration of the windows used (determined by w_r, w_s and *ChunkSize*) and the relative weighting given to the local and cumulative CEs in Phase II (determined by f). Note that it can be proved that replacing the spiral array with the line of fifths makes no difference to the output of the algorithm (see Appendix C). Note also that, for the 12 best parameter value combinations, changing the range of permitted indices from $\langle MinSAIndex, MaxSAIndex\rangle = \langle -15, 19\rangle$ to $\langle MinSAIndex, MaxSAIndex\rangle = \langle -22, 26\rangle$ made no difference to the results.

Table 5.5 gives the parameter values for the 12 most accurate versions of the ChewChen algorithm tested, together with an identification code, CCOP01–CCOP12, for each one. The results obtained using these 12 most accurate versions of the ChewChen algorithm are summarised in Tables 5.6 and 5.7. As can be seen in Table 5.7, of these 12 best versions of the algorithm, the six which took into account all the notes *sounding* within each window (i.e., CCOP01–06) were less dependent on style than those which only took into account the notes *starting* in each window (CCOP07–12): the style dependence was 0.35 when $StartOrSound = \mathsf{Sounding}$ and 0.42 when $StartOrSound = \mathsf{Starting}$. However, all 12 of the algorithms CCOP01–12 were less dependent on style than any of the other algorithms considered in this study. Note also, from Table 5.6, that, although CCOP07–12 only made one more error overall than CCOP01–06, the errors made by CCOP01–06 were different in general from those made by CCOP07–12, as is evident from the differing note error counts for the individual composers.

Code	w_s	w_r	f	*AspectRatio*	*ChunkSize*	*StartOrSound*	*SAOrLOF*	*MinSAIndex*	*MaxSAIndex*
CCOP01	8	2	0.50	$\sqrt{15/2}$	500	Sounding	SA	-22	26
CCOP02	8	2	0.50	$\sqrt{15/2}$	500	Sounding	SA	-15	19
CCOP03	8	2	0.50	$\sqrt{2/15}$	500	Sounding	LOF	-22	26
CCOP04	8	2	0.50	$\sqrt{2/15}$	500	Sounding	LOF	-15	19
CCOP05	8	2	0.50	$\sqrt{2/15}$	500	Sounding	SA	-22	26
CCOP06	8	2	0.50	$\sqrt{2/15}$	500	Sounding	SA	-15	19
CCOP07	8	2	0.50	$\sqrt{15/2}$	500	Starting	SA	-22	26
CCOP08	8	2	0.50	$\sqrt{15/2}$	500	Starting	SA	-15	19
CCOP09	8	2	0.50	$\sqrt{2/15}$	500	Starting	LOF	-22	26
CCOP10	8	2	0.50	$\sqrt{2/15}$	500	Starting	LOF	-15	19
CCOP11	8	2	0.50	$\sqrt{2/15}$	500	Starting	SA	-22	26
CCOP12	8	2	0.50	$\sqrt{2/15}$	500	Starting	SA	-15	19

Table 5.5: Parameter values for the best performing versions of the ChewChen algorithm tested. The first column gives an identification code for each parameter value combination.

Algorithm	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
CCOP01–06	175	311	152	136	365	230	149	147	1665
CCOP07–12	150	295	138	132	421	220	155	155	1666

Table 5.6: Note error counts for algorithms in column labelled "Algorithm" for the complete test corpus (last column), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus. The algorithms are sorted into increasing order of overall note error count. See text and sections 1.3.4 and 1.3.6 for further details.

Algorithm	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\overline{\text{NA}}$	SD_{Sty}
CCOP01–06	99.29	98.73	99.38	99.44	98.51	99.06	99.39	99.40	99.15	99.15	0.35
CCOP07–12	99.39	98.80	99.44	99.46	98.28	99.10	99.37	99.37	99.15	99.15	0.42

Table 5.7: Note accuracies expressed as percentages for each algorithm in the column headed "Algorithm" for the complete test corpus (column 10), and for each subset of the test corpus containing movements by one of the eight composers (columns 2 to 9). The columns headed $\overline{\text{NA}}$ and SD_{Sty} give the mean and standard deviation, respectively, of the values in columns 2 to 9. The algorithms are sorted into decreasing order of overall note accuracy. See text and sections 1.3.4 and 1.3.6 for further details.

Eq. class	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	Variance	St. dev.
[C]	98.63	98.64	98.64	98.65	98.67	98.67	0.0003	0.0180
[G]	93.54	97.13	97.15	97.07	97.17	98.10	0.6159	0.7848
[GL]	93.38	97.28	97.43	97.24	97.54	98.22	0.6760	0.8222
[GLC]	98.38	98.68	98.88	98.84	99.00	99.15	0.0398	0.1995
[C] ∪ [GLC]	98.38	98.66	98.86	98.83	99.00	99.15	0.0396	0.1990
[G] ∪ [GL]	93.38	97.16	97.35	97.19	97.52	98.22	0.6634	0.8145

Table 5.8: Descriptive statistics for note accuracy (%) for the four equivalence classes, [C], [G], [GL] and [GLC].

5.6.2.2 Frequency distribution of note accuracies over $\mathcal{P}''$

The histogram in Figure 5.9(a) has two peaks suggesting that the note accuracies over the whole of $\mathcal{P}''$ may be derived from two separate populations, one with a mode at just above 97% and the other with a mode at around 99%. Note also that the distribution is both badly negatively skewed (skewness = -2.08) and leptokurtic (kurtosis = 6.324). The Shapiro-Wilk test (Royston, 1982a,b, 1995; Shapiro and Wilk, 1965) returns a p value of less than 2.2×10^{-16} ($W = 0.7558$) implying that this distribution departs severely from normality.

The set of parameter value combinations $\mathcal{P}''$ can be partitioned into four classes according to the combination of CEs that is used (i.e., cumulative, global alone, global and local, or all three). If p is a parameter value combination in $\mathcal{P}''$, then let $\mathrm{GLC}(p)$ denote the combination of CEs employed when the ChewChen algorithm is run with the parameter value combination p, where

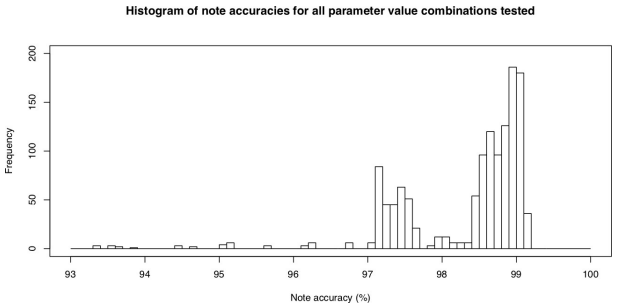
1. $\mathrm{GLC}(p) = \mathsf{GLC}$ iff the global, local and cumulative CEs are all used;
2. $\mathrm{GLC}(p) = \mathsf{GL}$ iff only the global and local CEs are used;
3. $\mathrm{GLC}(p) = \mathsf{G}$ iff only the global CE is used; and
4. $\mathrm{GLC}(p) = \mathsf{C}$ iff only the cumulative CE is used.

If $[x]$ denotes the set $\{p \mid p \in \mathcal{P}'' \wedge \mathrm{GLC}(p) = x\}$, then $\mathcal{P}''$ can be partitioned into the four equivalence classes, [GLC], [GL], [G] and [C]. The four histograms in Figure 5.9(b)–(e) show the distributions of note accuracies over each of these four equivalence classes and the box plot (Tukey, 1977) in Figure 5.10 represents the same information in a different form. From this box plot and the four histograms in Figure 5.9(b)–(e), it is clear that the distribution in Figure 5.9(a) has two peaks because the note accuracies for [GL] and [G] are mostly between 97 and 98% whereas those for [GLC] and [C] lie above 98%.

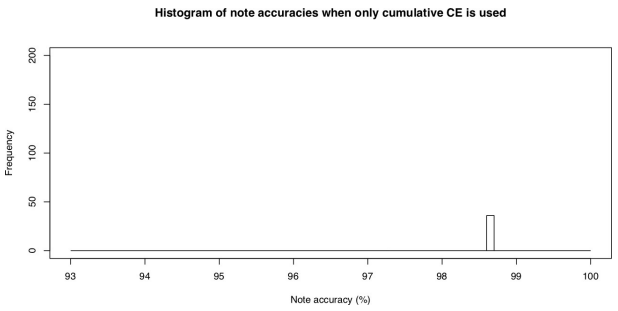
An examination of the box plot in Figure 5.10 and the descriptive statistics in Table 5.8 reveals that all the versions of the algorithm that used the cumulative CE were more accurate than those that did not, with those that used the cumulative CE achieving a median note accuracy of 98.86% and those that did not achieving a median note accuracy of 97.35%.

Furthermore, observe that the note accuracies achieved by the versions in which the cumulative CE was *not* used are much more widely dispersed than those achieved by the versions in

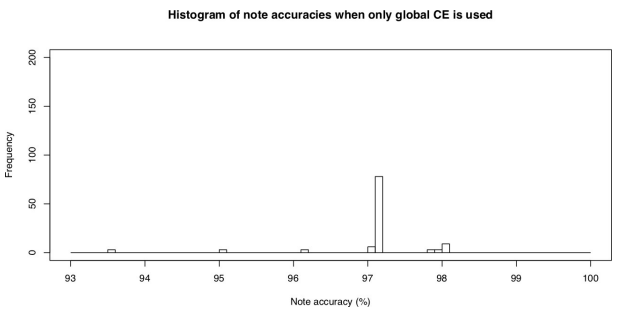
(a)



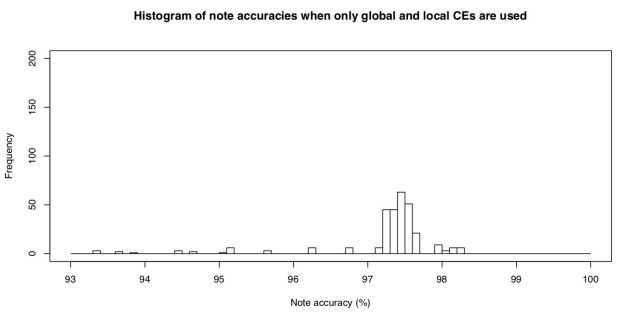
(b)



(c)



(d)



(e)

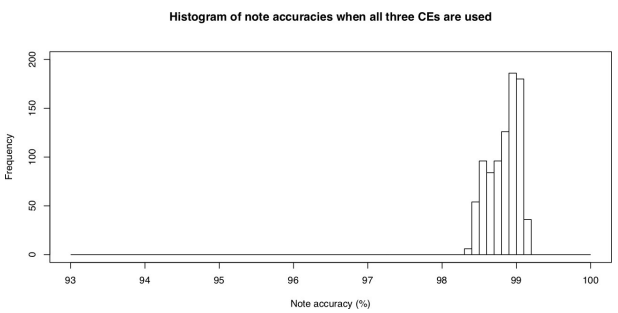


Figure 5.9: Histograms showing the distribution of note accuracies over: (a) the 1296 parameter value combinations in $\mathcal{P}''$; (b) the 36 parameter value combinations in [C]; (c) the 108 parameter value combinations in [G]; (d) the 288 parameter value combinations in [GL]; and (e) the 864 parameter value combinations in [GLC].

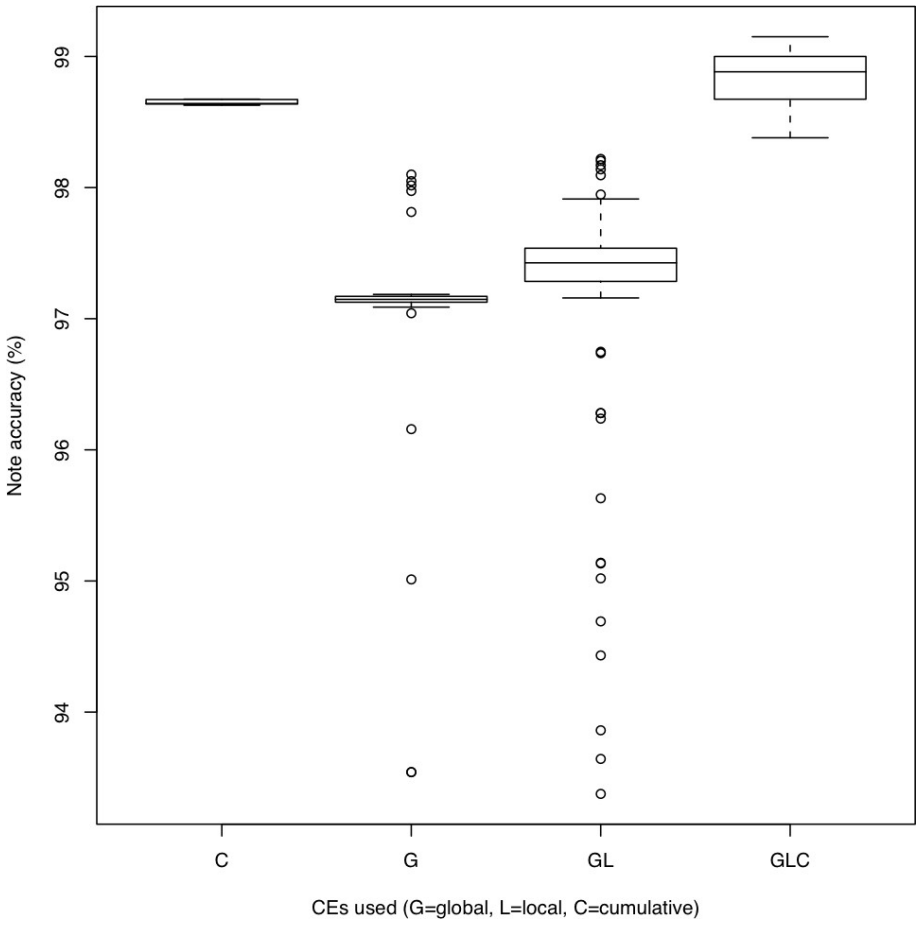


Figure 5.10: Box plot of note accuracy against GLC(p) for all $p \in \mathcal{P}''$. For a clear description of how to interpret box plots, see Dancey and Reidy (2002, pp. 52–55).

Sample	*Size*	*Skewness*	*Kurtosis*	*Shapiro-Wilk*	
				W	p
$\mathcal{P}''$	1296	-2.083	6.324	0.756	$< 2.2 \times 10^{-16}$
[C]	36	0.454	-1.544	0.792	$= 1.112 \times 10^{-5}$
[G]	108	-2.846	10.028	0.551	$< 2.2 \times 10^{-16}$
[GL]	288	-2.936	9.017	0.591	$< 2.2 \times 10^{-16}$
[GLC]	864	-0.429	-1.018	0.939	$< 2.2 \times 10^{-16}$
[C] ∪ [GLC]	900	-0.337	-1.102	0.945	$< 2.2 \times 10^{-16}$
[G] ∪ [GL]	396	-2.880	9.108	0.633	$< 2.2 \times 10^{-16}$

Table 5.9: The sample size, skewness, kurtosis and Shapiro-Wilk Normality Test results for $\mathcal{P}''$, [GLC], [GL], [G] and [C].

Class 1	*Class 2*	*Wilcoxon Rank Sum*		*t-test*		
		W	p	t	df	p
[C]	[G]	3888	$< 2.2 \times 10^{-16}$	20.853	107.338	$< 2.2 \times 10^{-16}$
[C]	[GL]	10368	$< 2.2 \times 10^{-16}$	29.082	289.177	$< 2.2 \times 10^{-16}$
[C]	[GLC]	7272	6.028×10^{-8}	−25.345	633.939	$< 2.2 \times 10^{-16}$
[G]	[GL]	7017	$< 2.2 \times 10^{-16}$	−1.831	200.546	0.0685
[G]	[GLC]	0	$< 2.2 \times 10^{-16}$	−23.267	108.735	$< 2.2 \times 10^{-16}$
[GL]	[GLC]	0	$< 2.2 \times 10^{-16}$	−32.702	298.339	$< 2.2 \times 10^{-16}$
[G] ∪ [GL]	[C] ∪ [GLC]	0	$< 2.2 \times 10^{-16}$	−39.484	415.889	$< 2.2 \times 10^{-16}$

Table 5.10: The results of using the Wilcoxon Rank Sum test (with continuity correction) and the Welch Two Sample t test to measure the significance of the differences between equivalence classes [C], [G], [GL] and [GLC] with respect to note accuracy. Note that, according to the Shapiro-Wilk test, the distributions are sufficiently non-normal for the t test results to be considered inappropriate for this data (see Table 5.9).

which the cumulative CE *was* used: the standard deviation of the note accuracies for [G] ∪ [GL] was 0.8145 whereas it was only 0.1990 for [C] ∪ [GLC]. This suggests that using the cumulative CE makes the algorithm much less sensitive to changes in its parameter values. Indeed, by using the cumulative CE alone, it was possible to achieve almost as high a note accuracy as it was by combining it with the local and global CEs. These results contradict Chew and Chen's (2005, p. 74) suggestion that "a purely sliding-window method should perform better than a purely cumulative-window method" and suggest that the opposite is in fact the case.

Table 5.9 shows the sample size, skewness, kurtosis and Shapiro-Wilk Normality Test results for each of the five histograms in Figure 5.9. The values in Table 5.9 suggest that none of the distributions in Figure 5.9 are even approximately normal. Therefore it would not be appropriate to compare these distributions using parametric tests such as the Student's t test or *ANOVA*.

A non-parametric, distribution-free test called the Wilcoxon Rank Sum test for two independent samples (which is equivalent to the Mann-Whitney test) (Howell, 1982, pp. 499–504;

Dancey and Reidy, 2002, pp. 517–523) was therefore used to determine whether there were any statistically significant differences with respect to note accuracy between the four equivalence classes [C], [G], [GL] and [GLC]. The results of this analysis are shown in Table 5.10 along with those obtained using the t test. Note that, according to the Wilcoxon Rank Sum test, all four classes were very significantly different from each other with respect to note accuracy. Also, the note accuracies achieved when the cumulative CE was used (i.e., $[\mathsf{C}] \cup [\mathsf{GLC}]$) were very significantly different from those achieved when it was not ($[\mathsf{G}] \cup [\mathsf{GL}]$). The p values returned by the t test were in close agreement with those returned by the Wilcoxon Rank Sum test except for the difference between [G] and [GL] which the t test judged to be insignificant at the 0.05 level. However, for the reasons discussed above, the results of the Wilcoxon Rank Sum Test should be considered more reliable than those of the t test in this case.

5.6.2.3 Effect of *SAOrLOF* and *AspectRatio*

It can be proved that the spelling generated by Chew and Chen's algorithm when the line of fifths is used is always exactly the same as that generated when the spiral array is used, regardless of the aspect ratio of the spiral array (see Appendix C). It follows that changing the aspect ratio of the spiral array also makes no difference to the output generated by the algorithm.

5.6.2.4 Effect of *StartOrSound* and the least accurate parameter value combinations

The set of parameter value combinations $\mathcal{P}''$ can be partitioned into 648 equivalence classes, $\mathcal{F}_i$, such that each $\mathcal{F}_i$ contains two parameter value combinations that are identical except that, in one, *StartOrSound* = Starting; whereas, in the other, *StartOrSound* = Sounding. The effect of the *StartOrSound* parameter can be studied by comparing the note accuracies of the two parameter value combinations in each of the equivalence classes, $\mathcal{F}_i$.

Let $\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i)$ denote the note accuracy (expressed as a percentage) achieved with the parameter value combination in $\mathcal{F}_i$ in which *StartOrSound* = Starting; and let $\mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)$ denote the note accuracy (again, expressed as a percentage) achieved with the parameter value combination in $\mathcal{F}_i$ in which *StartOrSound* = Sounding.

It was found that $\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i) \neq \mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)$ for every $\mathcal{F}_i$. In other words, changing from *StartOrSound* = Starting to *StartOrSound* = Sounding without changing any of the other parameter values always had an effect on the overall note accuracy. In 510 (78.7%) of the 648 cases tested, changing from *StartOrSound* = Sounding to *StartOrSound* = Starting *increased* the overall note accuracy; while in the other 138 cases, it reduced the overall note accuracy. The histogram in Figure 5.11 shows the frequency distribution of $\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i) - \mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)$ over the 648 classes $\mathcal{F}_i$. This histogram clearly shows that, in the vast majority of cases, the value of $\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i) - \mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)$ is close to 0. However, the distribution is quite strongly negatively skewed (skewness = -4.96), so, although the median value of $\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i) - \mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)$ is slightly greater than 0 (0.010), the mean value is slightly negative (-0.086).

However, the most important observation to be made with respect to the effect of the *StartOrSound* parameter is that in 621 (95.83%) of the 648 cases tested, changing the value of this parameter while keeping the other parameters constant caused the overall note accu-

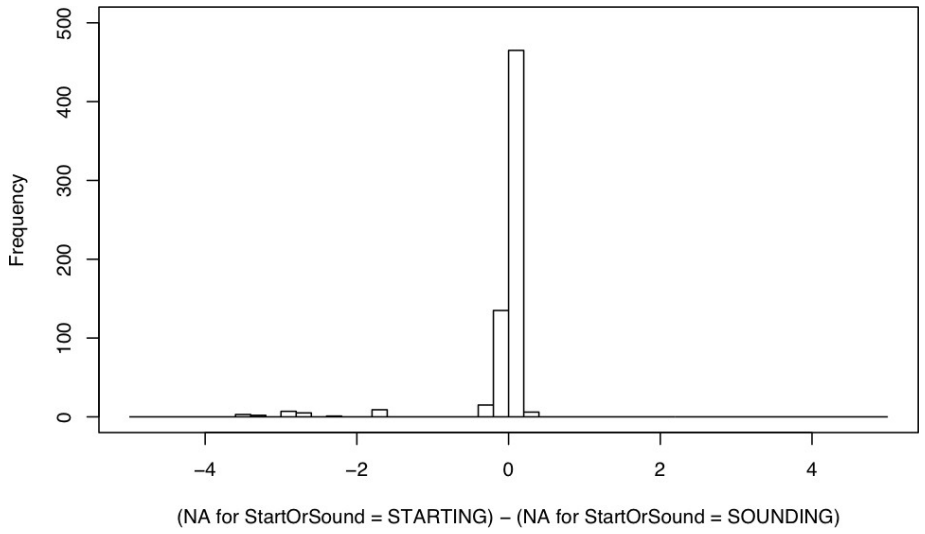


Figure 5.11: Frequency distribution of $\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i) - \mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)$ over all 648 $\mathcal{F}_i$ (see text for further explanation).

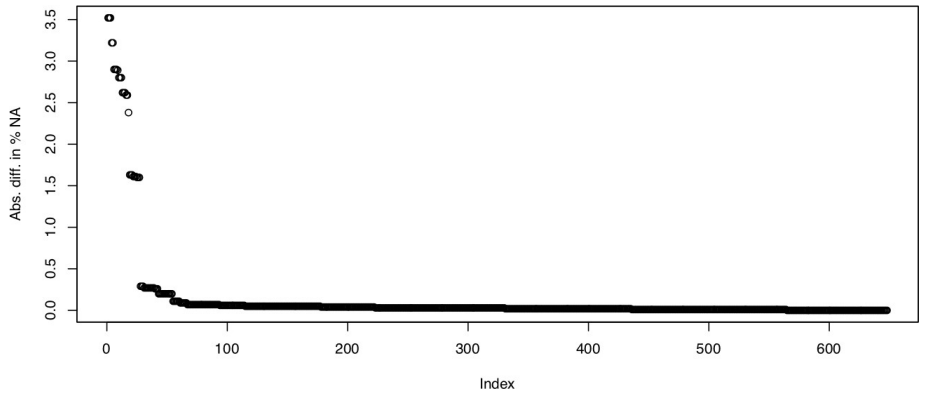


Figure 5.12: Graph of $|d_i| = |\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i) - \mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)|$ against position of $\mathcal{F}_i$ in a list in which the $\mathcal{F}_i$ are sorted into decreasing order of $|d_i|$.

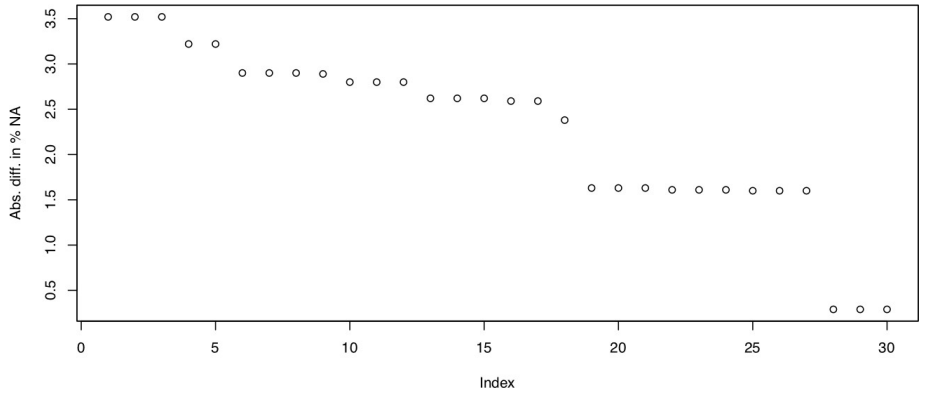


Figure 5.13: Exploded view of graph in Figure 5.12 showing points for 30 $\mathcal{F}_i$ most strongly affected by value of *StartOrSound* parameter.

racy expressed as a percentage to change by less than 0.3. In the remaining 27 cases (4.17%), changing the value of *StartOrSound* from Sounding to Starting *reduced* the note accuracy expressed as a percentage by 1.6 or more. The graph in Figure 5.12 was constructed by plotting $|d_i| = |\mathrm{NA}_{\mathrm{Start}}(\mathcal{F}_i) - \mathrm{NA}_{\mathrm{Sound}}(\mathcal{F}_i)|$ against the position of $\mathcal{F}_i$ in a list in which the equivalence classes had been sorted by decreasing value of $|d_i|$. Figure 5.13 gives an exploded view of this graph for the first 30 $\mathcal{F}_i$ in this list, that is, the 30 classes $\mathcal{F}_i$ that were most affected by the value of *StartOrSound*.

As can be seen in Figure 5.13, the value of $|d_i|$ drops suddenly from 1.60 for the 27th most affected $\mathcal{F}_i$ to just 0.29 for the 28th most affected $\mathcal{F}_i$. It is worth noting that the parameter value combinations in which *StartOrSound* = Starting in the 27 most affected $\mathcal{F}_i$ (i.e., the ones for which $|d_i| > 0.29$) are precisely the 27 worst performing (i.e., least accurate) combinations of the 1296 tested.

Note also that there is a relatively sudden drop in $|d_i|$ from 2.38 for the 18th most strongly affected $\mathcal{F}_i$ to 1.63 for the 19th most affected $\mathcal{F}_i$. Again, the parameter value combinations in which *StartOrSound* = Starting in the 18 most affected $\mathcal{F}_i$ (i.e., the ones for which $|d_i| > 1.63$) are precisely the 18 worst performing (i.e., least accurate) combinations of the 1296 tested. Moreover, these 18 most strongly affected $\mathcal{F}_i$ are precisely those in which $f = 1$, $w_s = 4$ and *ChunkSize* = 500ms (that is, those in which the smallest global window was used and the cumulative CE was not used).

Given that all those parameter value combinations in which the cumulative CE was used were more accurate than those in which it was not (see section 5.6.2.2), it is hardly surprising that the worst-performing parameter value combinations are those in which no cumulative CE and the smallest global context are used.

Figure 5.14 illustrates why the difference between considering the notes sounding in a window and considering only the notes starting in a window is greatest when the smallest windows are used. Figure 5.14 shows a note, N, 8 chunks in duration, represented by the horizontal line segment in the centre of the figure. The top half of the figure shows what happens when the duration of each window, w, is 2 chunks; and the bottom half of the figure shows what happens when each window is 5 chunks long (i.e., $w = 5$). When *StartOrSound* = Sounding and $w = 2$, N makes a contribution to the note spellings in 9 chunks, one for each of the windows A–I. For two of these chunks, N's weighting is 1 because each of windows A and I overlaps N by 1 chunk length. For the other 7 windows, B–H, N's weighting is 2. The weighting of N in a window when *StartOrSound* = Sounding is indicated to the left of the horizontal bracket representing that window. The sum of N's weightings over the whole input passage is therefore 16, but this total contribution is made up of relatively small influences over the spellings in 9 chunks. However, when *StartOrSound* = Starting and $w = 2$, N has a weighting of 8 in just two windows, A and B. The weighting of N in a window when *StartOrSound* = Starting is indicated to the right of the horizontal bracket representing that window. This means that the sum of N's weightings over the whole input passage is again 16, but this time this contribution is made up of relatively large influences over the spellings in just 2 chunks. In other words, when $w = 2$, N's influence is 'spread' over 4.5 times as many windows when *StartOrSound* = Sounding as when *StartOrSound* = Starting.

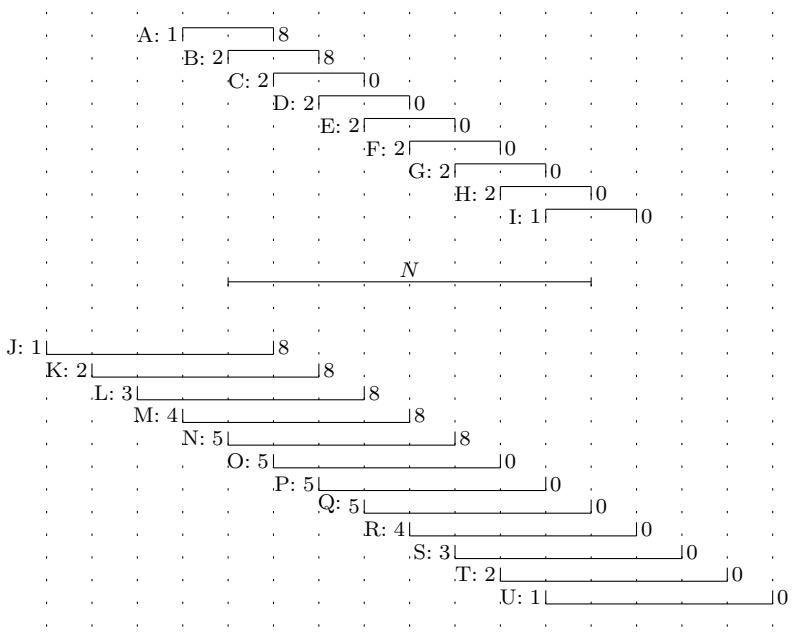


Figure 5.14: Illustration of the effect of window size on the proportional increase in the number of windows affected by each note when *StartOrSound* is changed from Starting to Sounding. The horizontal line segment labelled N represents a note 8 chunks long. The brackets A–I represent the positions of the window that overlap N when each window is 2 chunks long and the brackets J–U represent the positions of the window that overlap N when each window is 5 chunks long. The number to the left of each bracket gives the weighting of N when *StartOrSound* = Sounding for the window represented by the bracket. The number to the right of each bracket gives the weighting of N when *StartOrSound* = Starting for the window represented by the bracket. The vertical dotted lines represent chunk boundaries. Note that, when the window length is 2, the number of windows affected by N increases from 2 when *StartOrSound* = Starting to 9 when *StartOrSound* = Sounding—an increase of 4.5 times. However, when each window is 5 chunks long, the number of windows affected by N increases by only 2.4 times from 5 when *StartOrSound* = Starting to 12 when *StartOrSound* = Sounding.

Now consider what happens when $w = 5$, as in the bottom half of Figure 5.14. Regardless of the value of *StartOrSound*, the sum of N's weightings over the whole input passage is 40. However, when *StartOrSound* = Sounding, N influences the spellings in 12 chunks; and, when *StartOrSound* = Starting, it influences the spellings in 5 chunks. In other words, when $w = 5$, N's influence is 'spread' over 2.4 times as many windows when *StartOrSound* = Sounding as when *StartOrSound* = Starting. Thus, changing *StartOrSound* from Starting to Sounding causes the number of chunks affected by each note to increase by 4.5 times when $w = 2$ but only by 2.4 times when $w = 5$. This illustrates that the smaller the window size, the larger the proportional increase in the number of chunks influenced by each note when *StartOrSound* is changed from Starting to Sounding. This could explain why the parameter value combinations which were most affected by the value of *StartOrSound* were those in which the smallest global window was used and the cumulative window was omitted.

To summarise, for all but the most poorly performing parameter value combinations, changing the value of *StartOrSound* resulted in a relatively small change in the overall percentage note accuracy of less than 0.3, with most of the versions performing slightly better with *StartOrSound* = Starting than with *StartOrSound* = Sounding. From a practical point of view, this result suggests that not much is gained by using Chew and Chen's more complex method of calculating the centers of effect (corresponding to *StartOrSound* = Sounding) instead of the simpler method which I have proposed above (i.e., with *StartOrSound* = Starting). In particular, for the most accurate versions tested, using the simpler method resulted in just one more error over the entire test corpus. However, in a real-time situation where one needs to compute the pitch names of notes as they arrive in a MIDI stream, setting *StartOrSound* to Starting may not be practical since, in this case, each note is weighted by its full duration and one cannot know the duration of a note in a real-time situation until it has finished sounding. Consequently, if one were to set *StartOrSound* to Starting and use the algorithm to process a MIDI stream in real time, one could only assign pitch names at time points that do not occur in the middle of sounding notes. In general, this would cause "buffering"-type delays in the generation of the pitch names that would be worst in those cases where a passage contains long notes played against parts containing many short notes.

5.6.2.5 Effect of w_s

The effect that w_s has on the performance of the ChewChen algorithm can be studied by examining the effect that changing the value of w_s has on the overall note accuracy when the values of all the other parameters are kept constant. To do this, each of the equivalence classes [GLC], [GL] and [G] (see section 5.6.2.2 above) can itself be partitioned into smaller equivalence classes $\mathcal{G}_i$ such that the parameter value combinations in each $\mathcal{G}_i$ are identical except for the values of w_s. Since w_s must be greater than or equal to $w_r - 1$, each $\mathcal{G}_i$ will contain three parameter value combinations when $w_r < 6$: one for $w_s = 4$, one for $w_s = 8$ and one for $w_s = 16$. However, an equivalence class $\mathcal{G}_i$ in which $w_r = 6$ will contain only two parameter value combinations: one in which $w_s = 8$ and another in which $w_s = 16$.

Let $\mathrm{NA}_n(\mathcal{G}_i)$ denote the note accuracy (expressed as a percentage) achieved with the parameter value combination in $\mathcal{G}_i$ in which $w_s = n$. We need to be able to determine whether

S	w_s		N	*Shapiro-Wilk*		*Kurtosis*	*Skewness*	*One sample t-test*						*Wilcoxon signed rank test with continuity correction*				
	n_1	n_2		W	p			t	df	p	*Mean*	*95% CI*		V	p	*(Pseudo)median* (μ)	*95% CI*	
[G]	4	8	36	0.729	$\mathbf{8.45 \times 10^{-7}}$	0.403	1.30	2.13	35	**0.0399**	0.481	0.0234	0.939	456	0.0541	0.0285	-0.00100	0.522
	8	16	36	0.551	$\mathbf{2.49 \times 10^{-9}}$	0.900	-1.67	-1.98	35	0.0556	-0.109	-0.220	0.00274	450	0.0670	0.0159	-0.00132	0.0286
	4	16	36	0.766	$\mathbf{3.63 \times 10^{-6}}$	0.967	1.38	1.74	35	0.0912	0.373	-0.0629	0.808	459	**0.0485**	0.0276	0.00764	0.520
[GL]	4	8	72	0.803	$\mathbf{2.05 \times 10^{-8}}$	0.676	1.26	4.50	71	$\mathbf{2.59 \times 10^{-5}}$	0.571	0.318	0.823	2100	$\mathbf{1.04 \times 10^{-5}}$	0.289	0.112	0.672
	8	16	108	0.616	$\mathbf{2.18 \times 10^{-15}}$	7.20	2.24	2.41	107	**0.0175**	0.132	0.0236	0.241	4479	$\mathbf{2.52 \times 10^{-6}}$	0.0984	0.0827	0.109
	4	16	72	0.799	$\mathbf{1.62 \times 10^{-8}}$	0.210	1.19	4.82	71	$\mathbf{7.96 \times 10^{-6}}$	0.789	0.462	1.12	2112	$\mathbf{7.60 \times 10^{-6}}$	0.484	0.205	1.02
[GLC]	4	8	216	0.892	$\mathbf{2.46 \times 10^{-11}}$	0.653	1.05	4.48	215	$\mathbf{1.24 \times 10^{-5}}$	0.0311	0.0174	0.0448	14250	**0.00590**	0.0171	0.00758	0.0327
	8	16	324	0.974	$\mathbf{1.22 \times 10^{-5}}$	-0.107	0.172	-1.37	323	0.173	-0.00488	-0.0119	0.00215	23034	0.0512	-0.00611	-0.0118	0.000210
	4	16	216	0.867	$\mathbf{8.51 \times 10^{-13}}$	0.457	1.13	1.98	215	**0.0487**	0.0172	0.000101	0.0342	11742	0.980	-2.876×10^{-5}	-0.0161	0.0186
[G] ∪ [GL] ∪ [GLC]	4	8	324	0.533	$< \mathbf{2.2 \times 10^{-16}}$	10.4	3.11	5.03	323	$\mathbf{8.32 \times 10^{-7}}$	0.201	0.122	0.280	36282	$\mathbf{3.62 \times 10^{-9}}$	0.0452	0.0293	0.0584
	8	16	468	0.472	$< \mathbf{2.2 \times 10^{-16}}$	34.5	4.18	1.35	467	0.177	0.0187	-0.00847	0.0460	64809	**0.000689**	0.0138	0.00565	0.0222
	4	16	324	0.561	$< \mathbf{2.2 \times 10^{-16}}$	9.63	3.05	4.85	323	$\mathbf{1.92 \times 10^{-6}}$	0.228	0.136	0.321	32475	**0.000268**	0.0465	0.0209	0.0699

Table 5.11: Results of analysis to determine effect of w_s parameter. Significant p values (i.e., $p \leq 0.05$) are in bold. See text for explanation.

changing w_s from n_1 to n_2 has a significant effect on the note accuracy over some set, S, of parameter value combinations. To do this, we calculate $\delta_{\mathcal{G}}(i, n_1, n_2) = \mathrm{NA}_{n_2}(\mathcal{G}_i) - \mathrm{NA}_{n_1}(\mathcal{G}_i)$ for all $\mathcal{G}_i \subseteq S$. Then we examine the frequency distribution of $\delta_{\mathcal{G}}(i, n_1, n_2)$ to see if it is centred around some value other than 0.

Table 5.11 shows the results of this analysis. The first three columns give S, n_1 and n_2. The fourth column, N, gives the total number of classes $\mathcal{G}_i \subseteq S$ for which $\delta_{\mathcal{G}}(i, n_1, n_2)$ was calculated. The fifth and sixth columns give the results of the Shapiro-Wilk normality test on the distribution of $\delta_{\mathcal{G}}(i, n_1, n_2)$ over the set, S, and the seventh and eighth columns give the kurtosis and skewness of this distribution, respectively. For a distribution that approximates closely to normality, the p value returned by the Shapiro-Wilk test should be greater than 0.05 and the kurtosis and skewness should both be close to 0. The ninth, tenth and eleventh columns give the results of carrying out a one-sample t test to measure the statistical significance of the difference between 0 and the mean value of $\delta_{\mathcal{G}}(i, n_1, n_2)$ (which is given in the twelth column). The thirteenth and fourteenth columns give the 95% confidence interval around this mean. If the distribution of $\delta_{\mathcal{G}}(i, n_1, n_2)$ is not approximately normal, then the results of the t test may not be valid. In this case, more attention should be paid to the results of the non-parametric Wilcoxon signed rank test which are given in the last five columns of the table.

The Shapiro-Wilk test indicates that all the distributions in Table 5.11 depart significantly from normality, and, in most cases, this is supported by the values of kurtosis and skewness. For this data, the results of the Wilcoxon signed rank test should therefore be considered to carry more weight than those of the t test. However, the distribution of $\delta_{\mathcal{G}}(i, 8, 16)$ for [GLC] is probably sufficiently close to normal for the assumptions of the t test to be satisfied.

The results of the Wilcoxon test lead to the following conclusions.

1. Increasing w_s from 4 to 8 significantly increases the percentage note accuracy when the local window is used, but has a larger effect ($\mu = 0.29$) when the cumulative window is not used than when it is ($\mu = 0.02$).

2. Increasing w_s from 8 to 16 significantly increases the percentage note accuracy by around 0.10 when only the global and local windows are used.

3. Increasing w_s from 4 to 16 significantly increases the percentage note accuracy when the cumulative window is not used, but has a larger effect ($\mu = 0.48$) when the local window is used than when it is not ($\mu = 0.03$).

4. When all the tested parameter value combinations were considered together, increasing the global window size from 4 to either 8 or 16 significantly increased the percentage note accuracy by around 0.05, whereas increasing it from 8 to 16 significantly increased the percentage note accuracy but only by about 0.01.

5. In all other cases, changing the global window size did not result in a significant change in note accuracy.

S	w_r		N	*Shapiro-Wilk*		*Kurtosis*	*Skewness*	*One sample t-test*						*Wilcoxon signed rank test with continuity correction*				
	n_1	n_2		W	p			t	df	p	*Mean*	*95% CI*		V	p	*(Pseudo)median* (μ)	*95% CI*	
[GL]	2	4	108	0.636	$\mathbf{5.55 \times 10^{-15}}$	11.3	3.04	-0.532	107	0.596	-0.0236	-0.112	0.0644	1635	$\mathbf{6.12 \times 10^{-5}}$	-0.116	-0.134	-0.0899
	4	6	72	0.477	$\mathbf{1.35 \times 10^{-14}}$	10.8	3.30	1.504	71	0.137	0.0824	-0.0269	0.192	819	**0.00551**	-0.0705	-0.0824	-0.0242
	2	6	72	0.345	$\mathbf{2.68 \times 10^{-16}}$	15.7	4.07	-2.60	71	**0.0113**	-0.112	-0.198	-0.0262	381	$\mathbf{1.66 \times 10^{-7}}$	-0.202	-0.207	-0.196
[GLC]	2	4	324	0.923	$\mathbf{7.29 \times 10^{-12}}$	-0.550	0.679	-3.20	323	**0.00151**	-0.0172	-0.0277	-0.00661	20502	**0.000559**	-0.0242	-0.0372	-0.00924
	4	6	216	0.867	$\mathbf{8.49 \times 10^{-13}}$	2.49	1.46	-17.1	215	$< \mathbf{2.2 \times 10^{-16}}$	-0.0540	-0.0602	-0.0477	2100	$< \mathbf{2.2 \times 10^{-16}}$	-0.0600	-0.0648	-0.0546
	2	6	216	0.931	$\mathbf{1.40 \times 10^{-8}}$	-0.126	0.754	-10.3	215	$< \mathbf{2.2 \times 10^{-16}}$	-0.0712	-0.0848	-0.0575	4005	$< \mathbf{2.2 \times 10^{-16}}$	-0.0796	-0.0957	-0.0622
[GL]∪[GLC]	2	4	432	0.59	$< \mathbf{2.2 \times 10^{-16}}$	41.9	5.08	-1.60	431	0.111	-0.0188	-0.0419	0.00436	30849	$\mathbf{8.83 \times 10^{-10}}$	-0.0457	-0.0579	-0.0296
	4	6	288	0.306	$< \mathbf{2.2 \times 10^{-16}}$	54.8	7.00	-1.39	287	0.164	-0.0199	-0.0480	0.00818	6141	$< \mathbf{2.2 \times 10^{-16}}$	-0.0621	-0.0670	-0.0569
	2	6	288	0.534	$< \mathbf{2.2 \times 10^{-16}}$	42.9	5.69	-6.82	287	$\mathbf{5.46 \times 10^{-11}}$	-0.0813	-0.105	-0.0578	5880	$< \mathbf{2.2 \times 10^{-16}}$	-0.107	-0.120	-0.0927

Table 5.12: Results of analysis to determine effect of w_r parameter. Significant p values (i.e., $p \leq 0.05$) are in bold. See text for explanation.

5.6.2.6 Effect of w_r

The effect that w_r has on the performance of the ChewChen algorithm can be studied by examining the effect that changing the value of w_r has on the overall note accuracy when the values of all the other parameters are kept constant. To do this, each of the equivalence classes [GLC] and [GL] (see section 5.6.2.2 above) can itself be partitioned into smaller equivalence classes $\mathcal{H}_i$ such that the parameter value combinations in each $\mathcal{H}_i$ are identical except for the values of w_r. Since w_s must be greater than or equal to $w_r - 1$, each $\mathcal{H}_i$ will contain three parameter value combinations when $w_s \geq 6$: one for $w_r = 2$, one for $w_r = 4$ and one for $w_r = 6$ (see Table 5.4). However, an equivalence class $\mathcal{H}_i$ in which $w_s = 4$ will contain only two parameter value combinations: one in which $w_r = 2$ and another in which $w_r = 4$.

Let $\mathrm{NA}_n(\mathcal{H}_i)$ denote the note accuracy (expressed as a percentage) achieved with the parameter value combination in $\mathcal{H}_i$ in which $w_r = n$. We need to be able to determine whether changing w_r from n_1 to n_2 has a significant effect on the note accuracy over some set, S, of parameter value combinations. To do this, we calculate $\delta_{\mathcal{H}}(i, n_1, n_2) = \mathrm{NA}_{n_2}(\mathcal{H}_i) - \mathrm{NA}_{n_1}(\mathcal{H}_i)$ for all $\mathcal{H}_i \subseteq S$. Then we examine the frequency distribution of $\delta_{\mathcal{H}}(i, n_1, n_2)$ to see if it is centred around some value other than 0.

Table 5.12 shows the results of this analysis. This table has the same structure as Table 5.11 except that n_1 and n_2 now refer to values of w_r not w_s. The values returned by the Shapiro-Wilk test and the measures of kurtosis and skewness suggest that all the distributions considered in Table 5.12 depart significantly from normality. This means that more weight should be given to the measures of significance returned by the non-parameteric Wilcoxon signed rank test than to those returned by the t test. According to the Wilcoxon test, every distribution in Table 5.11 is centred around a value that is significantly different from 0. All the values of μ are negative. Therefore, these results suggest that increasing the size of the local window generally leads to a reduction in overall note accuracy. Note that the pseudomedian values indicate that, under most conditions, increasing the local window size by 2 or 4 chunks generally leads to a reduction in the percentage note accuracy of less than 0.2, with slightly larger decreases occurring when the cumulative CE is not used.

5.6.2.7 Effect of f

The effect that f has on the performance of the ChewChen algorithm can be studied by measuring the change in overall note accuracy that results when f is changed and the values of the other parameters are held constant.

When $w_r = 0$ and $f = 1$, only the global CE calculated in Phase I of the algorithm has any effect. When $w_r = 0$ and $f < 1$, only the cumulative CE has any effect, regardless of the specific value of f. Therefore, for each case in which $w_r = 0$, we only need to compare the result obtained when $f = 1$ with that obtained when $f = 0$. Furthermore, when $f = 0$, the values of w_r and w_s are immaterial. Therefore, to investigate the effect on note accuracy of changing f when $w_r = 0$, we have to compare the result for each parameter value combination, p, in which $w_r = 0$ and $f = 1$, with that for the parameter value combination in which $f = 0$ and all the other parameters (except possibly w_s and w_r) have the same values as in p. The total set of parameter value combinations over which this comparison is carried out will be denoted by

S	f			*Shapiro-Wilk*				*One sample t-test*						*Wilcoxon signed rank test with continuity correction*				
	n_1	n_2	f	W	p	*Kurtosis*	*Skewness*	t	df	p	*Mean*	*95% CI*		V	p	*(Pseudo)median* (μ)	*95% CI*	
$S_{w_r=0}$	0	1	108	0.549	$\mathbf{< 2.2 \times 10^{-16}}$	10.105	-2.879	-20.745	107	$\mathbf{< 2.2 \times 10^{-16}}$	-1.576	-1.727	-1.425	0	$\mathbf{< 2.2 \times 10^{-16}}$	-1.495	-1.502	-1.488
$S_{w_r\neq 0}$	0	0.25	288	0.966	$\mathbf{2.998 \times 10^{-6}}$	-0.737	-0.329	50.990	287	$\mathbf{< 2.2 \times 10^{-16}}$	0.251	0.241	0.260	41616	$\mathbf{< 2.2 \times 10^{-16}}$	0.254	0.243	0.264
	0	0.50	288	0.938	$\mathbf{1.299 \times 10^{-9}}$	-0.115	-0.717	78.796	287	$\mathbf{< 2.2 \times 10^{-16}}$	0.366	0.357	0.375	41616	$\mathbf{< 2.2 \times 10^{-16}}$	0.372	0.363	0.381
	0	0.75	288	0.978	$\mathbf{0.000210}$	-0.516	-0.160	-9.395	287	$\mathbf{< 2.2 \times 10^{-16}}$	-0.0522	-0.0632	-0.0413	9495	$\mathbf{1.269 \times 10^{-15}}$	-0.0505	-0.0622	-0.0398
	0	1.00	288	0.589	$\mathbf{< 2.2 \times 10^{-16}}$	9.024	-2.947	-28.933	287	$\mathbf{< 2.2 \times 10^{-16}}$	-1.412	-1.508	-1.316	0	$\mathbf{< 2.2 \times 10^{-16}}$	-1.231	-1.254	-1.205
	0.25	0.50	288	0.947	$\mathbf{1.1 \times 10^{-8}}$	0.312	-0.756	38.052	287	$\mathbf{< 2.2 \times 10^{-16}}$	0.115	0.109	0.121	41445	$\mathbf{< 2.2 \times 10^{-16}}$	0.119	0.113	0.125
	0.25	0.75	288	0.955	$\mathbf{8.965 \times 10^{-8}}$	-0.423	-0.383	-41.836	287	$\mathbf{< 2.2 \times 10^{-16}}$	-0.303	-0.317	-0.289	0	$\mathbf{< 2.2 \times 10^{-16}}$	-0.296	-0.318	-0.279
	0.25	1.00	288	0.587	$\mathbf{< 2.2 \times 10^{-16}}$	8.889	-2.953	-33.143	287	$\mathbf{< 2.2 \times 10^{-16}}$	-1.662	-1.761	-1.564	0	$\mathbf{< 2.2 \times 10^{-16}}$	-1.468	-1.495	-1.440
	0.50	0.75	288	0.956	$\mathbf{1.323 \times 10^{-7}}$	-0.875	-0.187	-62.560	287	$\mathbf{< 2.2 \times 10^{-16}}$	-0.418	-0.431	-0.405	0	$\mathbf{< 2.2 \times 10^{-16}}$	-0.415	-0.431	-0.400
	0.50	1.00	288	0.595	$\mathbf{1.323 \times 10^{-7}}$	8.876	-2.936	-36.404	287	$\mathbf{< 2.2 \times 10^{-16}}$	-1.777	-1.874	-1.681	0	$\mathbf{< 2.2 \times 10^{-16}}$	-1.593	-1.622	-1.567
	0.75	1.00	288	0.629	$\mathbf{< 2.2 \times 10^{-16}}$	8.426	-2.808	-28.357	287	$\mathbf{< 2.2 \times 10^{-16}}$	-1.359	-1.454	-1.265	0	$\mathbf{< 2.2 \times 10^{-16}}$	-1.207	-1.240	-1.167

Table 5.13: Results of analysis to determine effect of f parameter. Significant p values (i.e., $p \leq 0.05$) are in bold. See text for explanation.

$S_{w_r=0}$. $S_{w_r=0}$ can be partitioned into 288 *pairs* of parameter value combinations $\mathcal{I}_i$, such that each equivalence class $\mathcal{I}_i$ contains one parameter value combination, p, in which $w_r = 0$ and $f = 1$; and a second combination in which $f = 0$ and the other parameter values (apart from possibly w_r and w_s) have the same values as in p.

Let $\mathrm{NA}_n(\mathcal{I}_i)$ denote the note accuracy (expressed as a percentage) achieved with the parameter value combination in $\mathcal{I}_i$ in which $f = n$ ($n \in \{0, 1\}$). We need to be able to determine whether changing f from 0 to 1 has a significant effect on the note accuracy over the set $S_{w_r=0}$. To do this, we calculate $\delta_{\mathcal{I}}(i, 0, 1) = \mathrm{NA}_1(\mathcal{I}_i) - \mathrm{NA}_0(\mathcal{I}_i)$ for all $\mathcal{I}_i \subseteq S_{w_r=0}$. Then we examine the frequency distribution of $\delta_{\mathcal{I}}(i, 0, 1)$ to see if it is centred around some value other than 0.

The results of this analysis are shown in the first row of Table 5.13. This table has essentially the same structure as Tables 5.12 and 5.11, except that the values of n_1 and n_2 in the second and third columns are values of f. The results of the Shapiro-Wilk test together with the measures of kurtosis and skewness suggest that the distribution of $\delta_{\mathcal{I}}(i, 0, 1)$ departs significantly from normality, which implies that more weight should be given to the results of the non-parametric Wilcoxon signed rank test than to those of the one-sample t test. The Wilcoxon test indicates that changing f from 0 to 1 when $w_r = 0$ results in a highly significant reduction in the percentage note accuracy of about 1.5. Note that, when $w_r = 0$, changing f from 0 to 1 is equivalent to changing from using only the cumulative CE to using only the global CE (see section 5.6.2.2 above).

When $w_r \neq 0$, then,

1. when $f = 1$, only the global and local CEs have an effect;
2. when $f = 0.75$, all three CEs have an effect, but, in Phase II, the local CE has a greater influence than the cumulative CE;
3. when $f = 0.5$, all three CEs have an effect and, in Phase II, the local and cumulative CEs have the same degree of influence;
4. when $f = 0.25$, all three CEs have an effect, but, in Phase II, the cumulative CE has a greater influence than the local CE; and
5. when $f = 0$, only the cumulative CE has an effect.

To investigate the effect of changing f when $w_r \neq 0$, the set $S_{w_r \neq 0}$ containing all parameter value combinations in $\mathcal{P}''$ in which $w_r \neq 0$, can be partitioned into 288 equivalence classes, $\mathcal{J}_i$, such that each $\mathcal{J}_i$ contains parameter value combinations that are identical except for their values of f. Each $\mathcal{J}_i$ therefore contains 5 parameter value combinations, one for each of the 5 values of f used in the evaluation.

Let $\mathrm{NA}_n(\mathcal{J}_i)$ denote the note accuracy (expressed as a percentage) achieved with the parameter value combination in $\mathcal{J}_i$ in which $f = n$. We need to be able to determine whether changing f from n_1 to n_2 has a significant effect on the note accuracy over the set $S_{w_r \neq 0}$ of parameter value combinations. To do this, we calculate $\delta_{\mathcal{J}}(i, n_1, n_2) = \mathrm{NA}_{n_2}(\mathcal{J}_i) - \mathrm{NA}_{n_1}(\mathcal{J}_i)$ for all $\mathcal{J}_i \subseteq S_{w_r \neq 0}$. Then we examine the frequency distribution of $\delta_{\mathcal{J}}(i, n_1, n_2)$ to see if it is centred around some value other than 0.

The second and subsequent rows in Table 5.13 show the results of this analysis. According to both the t test and the Wilcoxon test, changing f from n_1 to n_2 resulted in a highly significant change in overall note accuracy for all values of n_1 and n_2 in $\{0, 0.25, 0.50, 0.75, 1.00\}$. The value of f that led to the highest accuracy was 0.5, followed, in order, by 0.25, 0, 0.75 and 1. The best accuracy was therefore achieved when the cumulative and local CEs make equal contributions in Phase II. Changing f from 0.5 to 0.25 (i.e., making the contribution of the cumulative CE greater than that of the local CE) reduced the overall percentage note accuracy by about 0.12. Changing f from 0.25 to 0 (i.e., eliminating the local CE altogether) reduced the overall percentage note accuracy by a further 0.25. However, omitting the local CE altogether was marginally better than weighting it more heavily than the cumulative CE in Phase II: changing f from 0 to 0.75 reduced the percentage note accuracy by a small, but apparently significant, 0.05. Finally, eliminating the cumulative CE altogether produced the worst results: changing f from 0.75 to 1 caused a relatively large drop of around 1.2 in the percentage note accuracy.

These results clearly contradict Chew and Chen's (2005, p. 74) claims that the local context is more important than the cumulative context[3] and that "a purely sliding-window method should perform better than a purely cumulative-window method". In fact, it seems that, first, a "purely cumulative-window method" is more accurate than one that only uses the sliding windows; and, second, that the cumulative context is just as important as the local context in Phase II of the algorithm.

5.6.2.8 The effect of *ChunkSize*

The effect that *ChunkSize* has on the performance of the ChewChen algorithm can be studied by examining the effect that changing the value of *ChunkSize* has on the overall note accuracy when the values of all the other parameters are kept constant. To do this, each of the equivalence classes [GLC], [GL], [G] and [C] (see section 5.6.2.2 above) can itself be partitioned into smaller equivalence classes $\mathcal{K}_i$ such that the parameter value combinations in each $\mathcal{K}_i$ are identical except for the values of *ChunkSize*.

Let $\mathrm{NA}_n(\mathcal{K}_i)$ denote the note accuracy (expressed as a percentage) achieved with the parameter value combination in $\mathcal{K}_i$ in which $\mathit{ChunkSize} = n$. We need to be able to determine whether changing *ChunkSize* from n_1 to n_2 has a significant effect on the note accuracy over some set, S, of parameter value combinations. To do this, we calculate $\delta_{\mathcal{K}}(i, n_1, n_2) = \mathrm{NA}_{n_2}(\mathcal{K}_i) - \mathrm{NA}_{n_1}(\mathcal{K}_i)$ for all $\mathcal{K}_i \subseteq S$. Then we examine the frequency distribution of $\delta_{\mathcal{K}}(i, n_1, n_2)$ to see if it is centred around some value other than 0.

The three box plots in Figure 5.15 give a general impression of the way that the note accuracy is affected over each of the sets [GLC], [GL], [G] and [C] when *ChunkSize* is changed from 500 to 1000 ms (Figure 5.15(a)), 1000 to 2000 ms (Figure 5.15(b)) and 500 to 2000 ms (Figure 5.15(c)). These box plots suggest that

1. when the cumulative CE is not used,

[3] What Chew and Chen (2005, p. 74) actually state is that "[b]ecause the best results were achieved with high values of f, we can deduce that the local context is more important than the *global* [sic] context in pitch spelling" (my emphasis). However, this is clearly a mistake: what they meant to claim was that the local context is more important than the *cumulative* context.

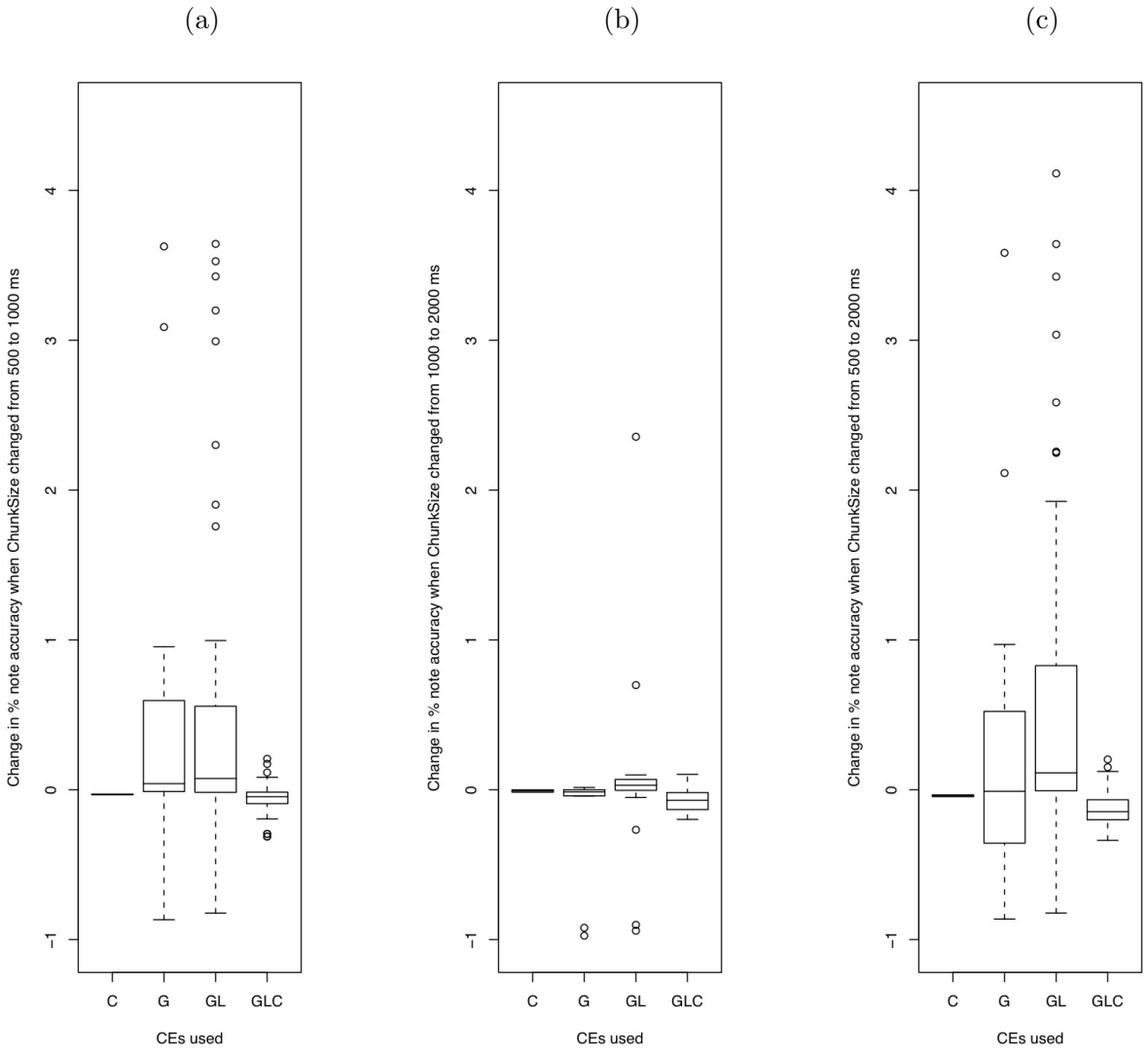


Figure 5.15: Box plots showing the effect for each of the sets [GLC], [GL], [G] and [C] on percentage note accuracy of changing *ChunkSize* from (a) 500 to 1000 ms, (b) 1000 to 2000 ms and (c) 500 to 2000 ms.

S	*ChunkSize*		N	*Shapiro-Wilk*		*Kurt.*	*Skew.*	*One sample t-test*						*Wilcoxon signed rank test with continuity correction*				
	n_1	n_2		W	p			t	df	p	*Mean*	*95% CI*		V	p	*(Pseudo)median*	*95% CI*	
$\mathcal{P}''$	500	1000	432	0.490	$< \mathbf{2.2 \times 10^{-16}}$	12.953	3.528	3.534	431	**0.000454**	0.122	0.0540	0.189	37827	**0.000578**	-0.0243	-0.0344	-0.0125
	1000	2000	432	0.458	$< \mathbf{2.2 \times 10^{-16}}$	45.975	4.188	-4.014	431	$\mathbf{7.048 \times 10^{-5}}$	-0.0518	-0.0771	-0.0264	19113	$< \mathbf{2.2 \times 10^{-16}}$	-0.0512	-0.0610	-0.0426
	500	2000	432	0.534	$< \mathbf{2.2 \times 10^{-16}}$	12.836	3.470	1.926	431	0.0548	0.0700	-0.00144	0.141	28161	$\mathbf{7.80 \times 10^{-13}}$	-0.0856	-0.105	-0.0633
[C]	500	1000	12	0.650	**0.000288**	-2.160	0	-46.032	11	$\mathbf{6.22 \times 10^{-14}}$	-0.0319	-0.0334	-0.0304	0	***0.00190**	-0.0318	*-0.0341	*-0.0296
	1000	2000	12	0.650	**0.000288**	-2.160	0	-3.524	11	**0.00477**	-0.00868	-0.0141	-0.00326	0	***0.00190**	-0.00865	*-0.0168	*-0.000510
	500	2000	12	0.650	**0.000288**	-2.160	0	-22.928	11	$\mathbf{1.23 \times 10^{-10}}$	-0.0406	-0.0445	-0.0367	0	***0.00190**	-0.0405	*-0.0464	*-0.0347
[G]	500	1000	36	0.730	$\mathbf{8.75 \times 10^{-7}}$	0.414	1.322	2.320	35	**0.0263**	0.528	0.0660	0.991	465	***0.0387**	0.0964	*0.00431	*0.595
	1000	2000	36	0.501	$\mathbf{6.57 \times 10^{-10}}$	0.973	-1.710	-2.837	35	**0.00753**	-0.168	-0.288	-0.0477	90	***0.000138**	-0.0188	*-0.0411	*-0.00978
	500	2000	36	0.765	$\mathbf{3.49 \times 10^{-6}}$	1.022	1.406	1.713	35	0.0957	0.361	-0.0669	0.788	381	*0.455	0.0273	*-0.0266	*0.523
[GL]	500	1000	96	0.772	$\mathbf{6.80 \times 10^{-11}}$	1.176	1.472	4.315	95	$\mathbf{3.90 \times 10^{-5}}$	0.503	0.272	0.735	3423	$\mathbf{6.33 \times 10^{-5}}$	0.148	0.0717	0.423
	1000	2000	96	0.517	$\mathbf{3.13 \times 10^{-16}}$	12.532	2.700	1.102	95	0.273	0.0558	-0.0447	0.156	3375	**0.000131**	0.0281	0.0155	0.0403
	500	2000	96	0.792	$\mathbf{2.53 \times 10^{-10}}$	1.075	1.419	4.375	95	$\mathbf{3.11 \times 10^{-5}}$	0.559	0.306	0.813	3357	**0.000171**	0.177	0.107	0.577
[GLC]	500	1000	288	0.948	$\mathbf{1.53 \times 10^{-8}}$	1.311	-0.037	-8.798	287	$< \mathbf{2.2 \times 10^{-16}}$	-0.0500	-0.0611	-0.0388	8703	$< \mathbf{2.2 \times 10^{-16}}$	-0.0497	-0.0585	-0.0419
	1000	2000	288	0.960	$\mathbf{4.07 \times 10^{-7}}$	-0.600	0.413	-17.929	287	$< \mathbf{2.2 \times 10^{-16}}$	-0.0749	-0.0831	-0.0667	3156	$< \mathbf{2.2 \times 10^{-16}}$	-0.0791	-0.0896	-0.0699
	500	2000	288	0.938	$\mathbf{1.12 \times 10^{-9}}$	0.751	0.855	-18.619	287	$< \mathbf{2.2 \times 10^{-16}}$	-0.125	-0.138	-0.112	3525	$< \mathbf{2.2 \times 10^{-16}}$	-0.135	-0.147	-0.123
[G] ∪ [GL]	500	1000	132	0.763	$\mathbf{2.52 \times 10^{-13}}$	1.041	1.452	4.874	131	$\mathbf{3.10 \times 10^{-6}}$	0.510	0.303	0.717	6354	$\mathbf{8.11 \times 10^{-6}}$	0.133	0.0674	0.352
	1000	2000	132	0.539	$< \mathbf{2.2 \times 10^{-16}}$	12.869	2.236	-0.126	131	0.9	-0.00516	-0.0862	0.0759	5301	**0.0384**	0.0117	0.000573	0.0227
	500	2000	132	0.795	$\mathbf{2.61 \times 10^{-12}}$	1.109	1.421	4.629	131	$\mathbf{8.75 \times 10^{-6}}$	0.505	0.289	0.721	6078	**0.000125**	0.136	0.0760	0.481
[C] ∪ [GLC]	500	1000	300	0.943	$\mathbf{2.02 \times 10^{-9}}$	1.480	-0.060	-9.026	299	$< \mathbf{2.2 \times 10^{-16}}$	-0.0492	-0.0600	-0.0385	9279	$< \mathbf{2.2 \times 10^{-16}}$	-0.0488	-0.0569	-0.0408
	1000	2000	300	0.963	$\mathbf{5.97 \times 10^{-7}}$	-0.692	0.322	-17.704	299	$< \mathbf{2.2 \times 10^{-16}}$	-0.0723	-0.0803	-0.0642	3624	$< \mathbf{2.2 \times 10^{-16}}$	-0.0747	-0.0842	-0.0676
	500	2000	300	0.946	$\mathbf{4.74 \times 10^{-9}}$	0.656	0.772	-18.700	299	$< \mathbf{2.2 \times 10^{-16}}$	-0.122	-0.134	-0.109	3813	$< \mathbf{2.2 \times 10^{-16}}$	-0.130	-0.140	-0.118

Table 5.14: Results of analysis to determine effect of *ChunkSize* parameter. Significant p values (i.e., $p \leq 0.05$) are in bold. p values and confidence intervals determined by Wilcoxon test that are marked with an asterisk are not reliable due to ties in the data. See text for explanation.

(a) increasing *ChunkSize* from 500 to either 1000 or 2000 ms often causes an increase in note accuracy;

(b) increasing *ChunkSize* from 500 to either 1000 or 2000 ms has a much more varied effect than when it is increased from 1000 to 2000 ms;

(c) increasing *ChunkSize* from 1000 to 2000 ms generally has only a very small effect on note accuracy;

2. when the cumulative CE is used,

(a) increasing *ChunkSize* generally decreases the overall note accuracy;

(b) increasing *ChunkSize* generally has a larger and more varied effect when the global and local CEs are also used.

These observations are confirmed by the more detailed results given in Table 5.14. This table has the same structure as Tables 5.11–5.13, except that, in Table 5.14, n_1 and n_2 are values of *ChunkSize*. The results obtained by the Shapiro-Wilk test and the measures of kurtosis and skewness given in Table 5.14 suggest that all the frequency distributions of $\delta_{\mathcal{K}}(i, n_1, n_2)$ considered here depart significantly from normality. This implies that the p values returned by the Wilcoxon signed rank test may be more reliable than those returned by the t test. Note, however, that some of the p values and confidence intervals returned by the Wilcoxon test are unreliable because of ties in the data.

One general trend that is suggested by the results in Table 5.14 is that increasing the value of *ChunkSize* tends to reduce note accuracy when the cumulative CE is used (i.e., when $S = [\mathsf{C}]$ or $S = [\mathsf{GLC}]$). Increasing the *ChunkSize* reduces the frequency with which the CEs are updated but also increases the size of the local and global windows. As discussed in section 5.6.2.2, all the parameter value combinations in which the cumulative CE is used achieved higher note accuracies than those that did not. Using the cumulative CE also seems to reduce the sensitivity of the algorithm to changes in the parameter value combinations. It therefore seems that the cumulative CE dominates over the other two CEs when it is used. In those cases where the cumulative CE is used, one might therefore expect the overall effect of changing *ChunkSize* to be dominated by the effect that this has on the cumulative CE. Since the cumulative CE is always calculated over the entire segment of music that has already passed, the only effect that increasing *ChunkSize* has on the cumulative CE is to reduce the frequency with which it is updated, which one would expect to lead to lower note accuracy, as observed in Table 5.14.

On the other hand, when the cumulative CE is not used, increasing *ChunkSize* increases the sizes of the global and local windows as well as reducing the frequency with which the CEs are updated. Intuitively, one would expect there to be some optimal window sizes for calculating the global and local CEs: if these windows are too big, the algorithm will not be sufficiently sensitive to changes in key; if they are too small, the context may not contain enough information to determine the key accurately. This intuition is in agreement with the results in Table 5.14 which suggest that, over the set $[\mathsf{G}]$, increasing *ChunkSize* from 500ms to 1000ms generally improves note accuracy, but that increasing it from 1000ms to 2000ms slightly reduces note accuracy. This seems to imply that, when the global CE is used alone, there is an optimal value for *ChunkSize*

which is somewhere between 500 and 2000ms. Similarly, over the set [GL], increasing *ChunkSize* from 500 to 1000ms increases the percentage note accuracy by a pseudomedian value of around 0.15, but increasing *ChunkSize* by a further 1000ms to 2000ms only increases the percentage note accuracy by around 0.03.

Note that the 12 best-performing parameter value combinations (see Table 5.5) are in [GLC] and that, in these cases, *ChunkSize* is set to 500ms.

5.6.2.9 Effect of ⟨*MinSAIndex*, *MaxSAIndex*⟩

To investigate the effect of changing the value of ⟨*MinSAIndex*, *MaxSAIndex*⟩, the complete set of parameter value combinations tested, $\mathcal{P}''$, was partitioned into 648 equivalence classes, $\mathcal{L}_i$, such that each $\mathcal{L}_i$ contained two parameter value combinations that were the same, except that, in one, ⟨*MinSAIndex*, *MaxSAIndex*⟩ was set to $\langle -15, 19\rangle$; whereas, in the other, it was set to $\langle -22, 26\rangle$. Recall (from section 5.2.1) that, when ⟨*MinSAIndex*, *MaxSAIndex*⟩ = $\langle -15, 19\rangle$, the algorithm is restricted to choosing pitch name classes that lie between F𝄫 and B𝄪 (inclusive) on the line of fifths; whereas, when ⟨*MinSAIndex*, *MaxSAIndex*⟩ = $\langle -22, 26\rangle$, the algorithm may choose from a wider range of pitch name classes extending from F𝄫𝄫 to B𝄪♯, inclusive.

Let $\mathrm{NA}_x(\mathcal{L}_i)$ denote the note accuracy (expressed as a percentage) achieved with the parameter value combination in $\mathcal{L}_i$ in which ⟨*MinSAIndex*, *MaxSAIndex*⟩ = x where $x \in \{\langle -15, 19\rangle, \langle -22, 26\rangle\}$. Let $\delta_i = \mathrm{NA}_{\langle -22,26\rangle}(\mathcal{L}_i) - \mathrm{NA}_{\langle -15,19\rangle}(\mathcal{L}_i)$.

For 198 of the 648 $\mathcal{L}_i$, changing from ⟨*MinSAIndex*, *MaxSAIndex*⟩ = $\langle -15, 19\rangle$ to ⟨*MinSAIndex*, *MaxSAIndex*⟩ = $\langle -22, 26\rangle$ caused a change in overall note accuracy (i.e., $\delta_i \neq 0$ for 198 of the 648 $\mathcal{L}_i$). In other words, in 69.4% of the cases tested, changing the value of ⟨*MinSAIndex*, *MaxSAIndex*⟩ made no difference to the overall note accuracy achieved.

In exactly half of the 198 cases for which $\delta_i \neq 0$, the note accuracy was improved by changing from ⟨*MinSAIndex*, *MaxSAIndex*⟩ = $\langle -15, 19\rangle$ to ⟨*MinSAIndex*, *MaxSAIndex*⟩ = $\langle -22, 26\rangle$ (i.e., $\delta_i > 0$ in 99 of the 198 $\mathcal{L}_i$ for which $\delta_i \neq 0$). The four histograms in Figure 5.16 show the distribution of δ_i over (a) all 648 $\mathcal{L}_i$, (b) the 198 $\mathcal{L}_i$ for which $\delta_i \neq 0$, (c) the 99 $\mathcal{L}_i$ for which $\delta_i > 0$, and (d) the 99 $\mathcal{L}_i$ for which $\delta_i < 0$. As can be seen in Figure 5.16(c), for the 99 $\mathcal{L}_i$ in which $\delta_i > 0$, the magnitude of the change in percentage note accuracy was very small (mean of δ_i: 0.012) and the range of δ_i was also very small (min: 0.001; max: 0.020; sd: 0.005). However, for the 99 $\mathcal{L}_i$ in which $\delta_i < 0$, the magnitude of the change in percentage note accuracy was often considerably larger (mean: -0.784) and varied over a much wider range of values (min: -2.292; max: -0.003; sd: 0.738).

There are 198 $\mathcal{L}_i$ in which $f = 1$ (i.e., the cumulative CE is not used), and these 198 $\mathcal{L}_i$ contain the 396 least accurate parameter value combinations tested—as discussed in section 5.6.2.2 above, all those versions in which the cumulative CE was used achieved higher note accuracies than those in which it was not.

The set of 198 classes $\mathcal{L}_i$ for which $\delta_i \neq 0$ is *almost* identical to the set of 198 $\mathcal{L}_i$ in which $f = 1$: the two sets have 195 elements in common. In other words, it is *very nearly* true to say that changing ⟨*MinSAIndex*, *MaxSAIndex*⟩ from $\langle -15, 19\rangle$ to $\langle -22, 26\rangle$ only affected the overall note accuracy when the cumulative CE was not used.

There were just 3 cases in which the cumulative CE was not used that were unaffected by the

(a)

(b)

(c)

(d)

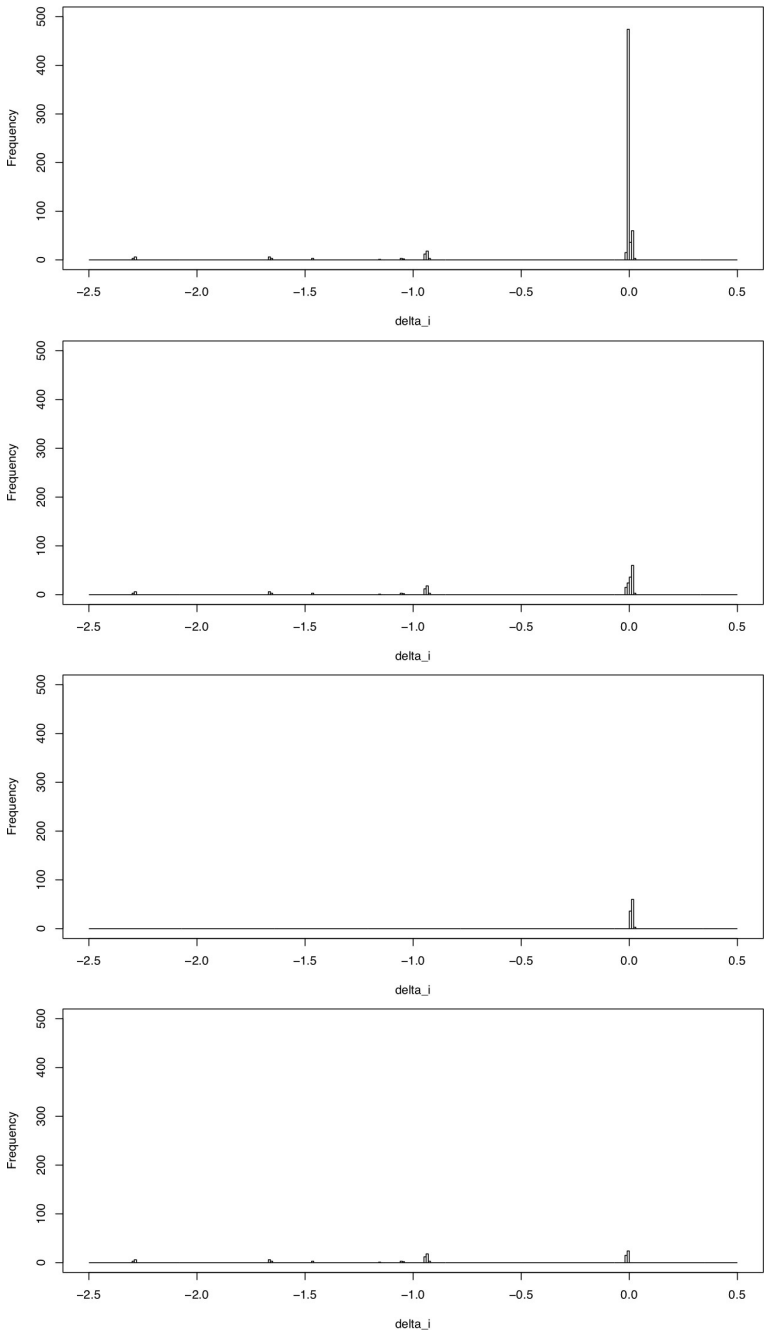


Figure 5.16: Histograms showing the distribution of δ_i over (a) all 648 $\mathcal{L}_i$, (b) the 198 $\mathcal{L}_i$ for which $\delta_i \neq 0$, (c) the 99 $\mathcal{L}_i$ for which $\delta_i > 0$, and (d) the 99 $\mathcal{L}_i$ for which $\delta_i < 0$.

change in $\langle MinSAIndex, MaxSAIndex\rangle$: when $w_s = 4$, $w_r = 0$, $f = 1$, $ChunkSize = 2000$ms and $StartOrSound = \mathsf{Sounding}$, the overall percentage note accuracy was always 97.13% regardless of the value of $\langle MinSAIndex, MaxSAIndex\rangle$. Correspondingly, there were only 3 cases where the cumulative CE *was* used that *were* affected by the change in $\langle MinSAIndex, MaxSAIndex\rangle$: when $w_s = 4$, $w_r = 2$, $f = 0.75$, $ChunkSize = 500$ms and $StartOrSound = \mathsf{Starting}$, changing $\langle MinSAIndex, MaxSAIndex\rangle$ from $\langle -15, 19\rangle$ to $\langle -22, 26\rangle$ reduced the overall percentage note accuracy from 98.54% by just 0.01 to 98.53%.

Unfortunately, it is not so easy to see how the set of 99 $\mathcal{L}_i$ in which $\delta_i > 0$ can be succinctly characterized.

In sum, for over 99% of the parameter value combinations tested, changing $\langle MinSAIndex, MaxSAIndex\rangle$ from $\langle -15, 19\rangle$ to $\langle -22, 26\rangle$ had no effect when the cumulative CE was used and changed the overall note accuracy when the cumulative CE was not used. When the cumulative CE was not used, changing $\langle MinSAIndex, MaxSAIndex\rangle$ from $\langle -15, 19\rangle$ to $\langle -22, 26\rangle$ increased the percentage note accuracy by an average of 0.01 in about half of the cases and reduced it by an average of 0.8 in the other half of the cases. Changing $\langle MinSAIndex, MaxSAIndex\rangle$ had no effect on the best-performing parameter value combinations.

5.7 Summary and conclusions

In this chapter, I have presented a detailed analysis and evaluation of the real-time pitch spelling algorithms described by Chew and Chen (2003a,b, 2005). These algorithms are based on Chew's (2000) "Spiral Array Model" which is a geometric model of tonal pitch relations. In the spiral array, the pitch name classes are arranged on a helix so that adjacent pitch name classes along this helix are a perfect fifth apart and adjacent pitch name classes along the length of the cylinder in which the helix is embedded are a major third apart (see Figure 5.1). As Chew and Chen (2005, p. 67) point out, the spiral array is "a spiral configuration of the line of fifths". However, they claim that the "depth added by going from one to three dimensions [i.e., from the line of fifths to the spiral array] allows the modeling of more complex hierarchical relations".

Chew and Chen define the *center of effect* (CE) of a set of notes to be the weighted centroid of the position vectors of the pitch name classes of the notes in the spiral array, each note being weighted by its duration. The basic principle underlying Chew and Chen's algorithms is that each note should be spelt so that it is as close as possible in the spiral array to the CE of the notes in a window preceding the note to be spelt.

In section 5.1.2, I showed that there are at least two reasonable methods that can be used to calculate the CE for a window. The simpler method is to consider only the notes *starting* in each window and weight each note by its total duration. However, in their own implementations, Chew and Chen use a more complex method in which they consider all the notes that *sound* in a window and weight each note by the duration for which it sounds within the window.

In section 5.1.3, I described Chew and Chen's two-phase boot-strapping algorithm (Chew and Chen, 2005, pp. 70–71). In this algorithm, the notes in a chunk are first spelt so that they are as close as possible to a *global CE* calculated over a window containing a certain number, w_s, of chunks preceding the chunk currently being spelt. Then the notes in the chunk are respelt so that they are as close as possible to a weighted average of two CEs: a *local CE* calculated over

a short window containing w_r chunks including the one currently being spelt; and a *cumulative CE* calculated over all the music preceding the current chunk. The relative weighting of the local and cumulative CEs is determined by the value of a parameter f: when $f = 1$, the cumulative CE is omitted altogether; and when $f = 0$, the local CE is omitted. The various versions of the algorithm described by Chew and Chen in their publications are all specific cases of this two-phase boot-strapping algorithm.

In section 5.2, I presented my own implementation of Chew and Chen's pitch spelling algorithm which I call ChewChen. This algorithm has various parameters which allow the user to control

1. the sizes of the global and local windows (w_s and w_r, respectively);
2. the relative weighting of the local and cumulative CEs (f);
3. the aspect ratio of the spiral array used;
4. the size of each chunk used in milliseconds;
5. the range of permitted pitch name classes along the line of fifths;
6. whether the line of fifths or the spiral array is used to calculate the CEs; and
7. whether the notes starting or sounding in a window are considered when calculating the CE of the window.

The ChewChen algorithm is not a real-time algorithm. However, in section 5.3, I showed that Chew and Chen's algorithm can be implemented as a real-time system whose worst-case running time is $O\left(\sum_{i=1}^{N_C} |C_i|\right)$ where N_C is the number of chunks in the input passage and $|C_i|$ is the number of notes in chunk C_i.

In section 5.4, I showed that the user can choose to "switch off" one or more of the global, local and cumulative CEs in Chew and Chen's boot-strapping algorithm by assigning particular values to the parameters w_s, w_r and f. In particular, I showed that there are four possible CE combinations in practice:

1. all three CEs have an effect ($w_s \neq 0 \wedge w_r \neq 0 \wedge 0 < f < 1$);
2. just the cumulative CE is used ($f = 0 \vee (0 < f < 1 \wedge (w_r = 0 \vee w_s = 0))$);
3. just the global CE is used ($f = 1 \wedge w_r = 0 \wedge w_s > 0$); or
4. only the global and local CEs have an effect ($f = 1 \wedge w_r > 0 \wedge w_s > 0$).

In section 5.5, I reviewed the results of the tests carried out by Chew and Chen to evaluate their algorithms. I concluded that the test corpus they used was too small and the range of parameter values explored too narrow for any conclusions drawn from these results to be considered anything more than tentative.

This motivated me to carry out a more thorough evaluation, described in section 5.6, in which the ChewChen algorithm was run 1296 times on the test corpus $\mathcal{C}$ defined in Table 1.4,

each time with a different combination of parameter values chosen from the sets in Table 5.4. An analysis of the results of this evaluation revealed a number of interesting facts.

First, the parameters that were critical for achieving the highest note accuracy were those controlling the duration of the windows used and the relative weighting given to the local and cumulative CEs. Specifically, the highest note accuracy was achieved when $w_s = 8$, $w_r = 2$, $f = 0.5$ and each chunk was 500ms. With these parameter values, the ChewChen algorithm spelt 99.15% of the notes in $\mathcal{C}$ correctly. Note that the best performance was achieved with $f = 0.5$—that is, with the local and cumulative CEs equally weighted. This contradicts Chew and Chen's (2005, p. 74) claim that the local context is more important than the cumulative context. Note also that it can be proved that the algorithm generates the same results with the line of fifths as it does with the spiral array, regardless of the aspect ratio of the latter (see Appendix C). There is therefore no real advantage in using the more complex spiral array instead of the line of fifths for pitch spelling.

It was found that, when $w_s = 8$, $w_r = 2$, $f = 0.5$ and each chunk was 500ms, considering the notes sounding in each window instead of just the notes starting improved the style dependence from 0.42 to 0.35. Also, in a real-time situation, where one does not know the total duration of a note until it has finished sounding, Chew and Chen's CE calculation method, in which the notes sounding in each window are considered and each note is weighted by its duration within the window, might be more practical.

Next, an examination of the frequency distribution of note accuracies over the 1296 parameter value combinations tested revealed that all those parameter value combinations in which the cumulative CE was used achieved higher note accuracies than those in which it was not, with those that used the cumulative CE achieving a mean note accuracy of 98.83% and those that did not achieving a mean note accuracy of 97.19%. It was also found that, when the cumulative CE was not used, the performance of the algorithm varied much more in response to changes in the parameter values (sd. of % NA with cum. CE: 0.2; sd of % NA without cum. CE: 0.8). Moreover, using the cumulative CE alone worked almost as well as using it in combination with the global and local CEs (cum. CE alone: max. = 98.67%, mean = 98.65%; cum., glob. and loc. CEs together: max. = 99.15%, mean = 98.84%). These results contradict Chew and Chen's (2005, p. 74) claim that "a purely sliding-window method should perform better than a purely cumulative-window method". In fact, it seems that using the cumulative CE alone consistently results in higher note accuracy than omitting it.

It can be proved that the spelling generated by Chew and Chen's algorithm when the line of fifths is used is always exactly the same as that generated when the spiral array is used, regardless of the aspect ratio of the spiral array (see Appendix C). It follows that changing the aspect ratio of the spiral array also makes no difference to the output generated by the algorithm.

Over all the parameter value combinations tested, it was found that changing from considering the notes sounding within a window to considering only the notes starting within a window while keeping the other parameters constant *always* resulted in a change in overall note accuracy. In 79% of cases, this change *improved* note accuracy. However, in 96% of cases, the resulting change in percentage note accuracy was less than 0.3, a greater change only being observed in

the 27 worst performing parameter value combinations.

Changing the size of the global window while keeping the other parameter values constant only had a noteworthy effect on note accuracy when the local CE was used and the cumulative CE was not. When this was the case, changing w_s from 4 to 8 added about 0.3 to the percentage note accuracy whereas changing it from 8 to 16 added about 0.1.

An analysis of the effect of changing the size of the local window while keeping the other parameter values constant revealed that increasing the size of this window generally reduced the overall percentage note accuracy, but usually by less than 0.2.

An analysis of the effect of changing f while keeping the other parameter values constant revealed that, when the local CE was not used (i.e., $w_r = 0$), changing f from 0 to 1 (i.e., changing from using only the cumulative CE to using only the global CE) reduced the percentage note accuracy by about 1.5. When the local CE *was* used (i.e., when $w_r \neq 0$), it was found that the best accuracy was achieved when f was set to 0.5—that is, when the local and cumulative CEs were equally weighted. These results clearly contradict Chew and Chen's (2005, p. 74) claim that the local context is more important than the cumulative context, and, in fact, suggest that the local and cumulative contexts are equally important in Phase II of the algorithm.

An analysis of the effect of changing the chunk size while keeping the other parameter values constant showed that increasing the chunk size from 500 to 1000 ms or from 1000 to 2000 ms typically reduced the overall percentage note accuracy by up to about 0.1 when the cumulative CE was used. When the cumulative CE was not used, increasing the chunk size from 500 to 1000 ms typically increased the percentage note accuracy by about 0.1; but increasing it from 1000 to 2000 ms typically caused either a much smaller improvement or a reduction in note accuracy.

Finally, increasing the range of permitted pitch name classes from F𝄫–B𝄪 on the line of fifths to F𝄫𝄫–B𝄪♯ almost never had an effect when the cumulative CE was used and nearly always did have an effect when it was not used. In about half of the cases where the cumulative CE was not used, increasing the range of permitted pitch name classes increased note accuracy by about 0.01. In the other cases where the cumulative CE was not used, this change caused a reduction in note accuracy of about 0.8. Increasing the range of permitted pitch name classes had no effect on the best-performing parameter value combinations.

To sum up, the results of this investigation suggest that Chew and Chen's algorithm works best when

1. the local, global and cumulative CEs all have an effect;
2. the local context window is relatively small (about 1s);
3. the global context window is a moderate size (about 4s);
4. the local and cumulative CEs are given equal weighting; and
5. the chunks are small (about 0.5s), leading to a frequent updating of the CEs.

Provided these conditions are satisfied, it makes no difference whether one uses the line of fifths or the spiral array and it makes little difference whether one considers only the notes starting in each window or all the notes sounding in each window.

N.B. The material included in this chapter has been published in the followng article published by Taylor & Francis: Meredith, D. (2006). The ps13 pitch spelling algorithm. Journal of New Music Research, 35:2, 121-159. http://dx.doi.org/10.1080/09298210600834961

Chapter 6

ps13

6.1 Introduction

Most experts seem to agree that the pitch name of a note in a passage of tonal music is primarily a function of

1. the key at the point where the note occurs; and
2. the voice-leading structure of the music in the note's immediate context.

For example, in Longuet-Higgins's (1987a, p. 115) "theory of tonality" (see section 2.2), Rule 1 ensures that each note is spelt in accordance with the key, while Rules 2 to 4 are primarily designed to ensure good voice-leading. Similarly, in Temperley's (2001, pp. 124–132) theory of pitch spelling, TPR 1 (Temperley, 2001, p. 125) effectively captures the influence of key on pitch spelling, while TPR 2 (Temperley, 2001, pp. 127–130) takes voice-leading into account.

Chew and Chen (2005, p. 61) state that pitch-spelling is "dependent on the key context, and to a lesser extent, the voice-leading tendencies in the music". In Chew and Chen's algorithm, the center of effect acts "as a proxy for" the key, and notes are spelt so that they are as close as possible to this center of effect (Chew and Chen, 2005, p. 63) (see discussion in section 5.1.2 above). Thus, although Chew and Chen's algorithm models the effect of key on pitch spelling, it does not take voice-leading into account, and they claim that this is one of the main causes of the errors made by their algorithm (Chew and Chen, 2005, pp. 73, 75).

Unlike the other algorithms discussed above, Cambouropoulos's method is based on the principles of interval optimisation and notational parsimony (Cambouropoulos, 2003, p. 421) (see section 3.1 above) and does not explicitly take key into account. However, interval optimisation involves pitches being spelt so that the intervals between them are preferably either (a) ones that occur frequently within the major and minor scales or (b) intervals that correspond to short distances along the line of fifths (Cambouropoulos, 2003, p. 423). Preferring to use intervals that correspond to short distances along the line of fifths tends to keep the pitch names close together on the line of fifths, which indirectly models the effect of key on pitch spelling to some extent. Similarly, preferring to use intervals that occur frequently within the major and minor scales would tend to lead to the notes being spelt so that the implied key does not change too frequently—again, indirectly modelling the effect of key on pitch spelling.

The earliest published version of Cambouropoulos's method (Cambouropoulos, 1996, 1998) incorporated a 'tie-breaker' rule, based on Krumhansl's (1990, pp. 150–151) principle of contextual asymmetry, that aimed to ensure good voice-leading in certain situations. However, the two more recent published versions of Cambouropoulos's algorithm do not take voice-leading into account. Nevertheless, Cambouropoulos (2003, p. 427) acknowledges that "voice leading is also an important component of pitch spelling" and proposes that "if the various melodic streams are predetermined, additional rules can cater to voice-leading effects".

When determining the pitch name of a note, the *ps13* algorithm (Meredith, 2003, 2005; Meredith and Wiggins, 2005) explicitly takes into account both the key at the point where the note occurs and the voice-leading structure of the music in the note's immediate vicinity. *ps13* is a two-stage algorithm: in Stage 1, each note is assigned a pitch name in accordance with the local key; in Stage 2, pitch names assigned in Stage 1 that lead to poor voice-leading are corrected.

In the algorithms of Temperley and Longuet-Higgins, the influence of key on pitch spelling is taken into account by assigning pitch names that are as close as possible on the line of fifths to either each other (in Temperley's theory) or the tonic (in Longuet-Higgins's theory). In Chew and Chen's algorithm, each pitch is spelt so that it is as close as possible to a "center of effect" in the spiral array which acts "as a proxy for" the key (Chew and Chen, 2005, p. 63) (see section 5.1.2) and it has been proved that the output generated by Chew and Chen's algorithm remains unchanged when the spiral array is replaced with the line of fifths (see Appendix C). This suggests that the effect of key on pitch spelling can be successfully modelled by spelling each note so that it is as close as possible to some "tonal centre" on the line of fifths. This has therefore been adopted as the basic principle underlying Stage 1 of *ps13*. However, in *ps13*, the tonal centre chosen is the *local tonic*, not a "center of gravity" or "center of effect" corresponding to the weighted average position on the line of fifths (or spiral array) of the pitch names in a passage. The decision to use the local tonic rather than any other possible "tonal centre" was motivated by the fact that traditional tonal music theory seems to endorse the idea of spelling notes so that they are as close as possible to the tonic on the line of fifths. For example, spelling notes in accordance with the conventional harmonic chromatic scale on a particular tonic (Associated Board of the Royal Schools of Music, 1958, p. 78) is the same as spelling them so that they are as close as possible to that tonic on the line of fifths, with notes 6 semitones from the tonic being preferably spelt as sharpened subdominants rather than flattened dominants. In other words, tonal theory seems to suggest that the line-of-fifths position, ℓ, of a note should satisfy the inequality $t-5 \leq \ell \leq t+6$ where t is the line-of-fifths position of the tonic. This was therefore the rule adopted in the *ps13* algorithm (see also the discussion of Longuet-Higgins's Rule 1 (Longuet-Higgins, 1987a, pp. 112–113) in section 2.2 and the discussion of the *RA4OrRD5* parameter of the TPRONE algorithm in section 4.7.3.2).

A number of authors have claimed that key is more important than voice-leading for determining pitch names. For example, Chew and Chen (2005, p. 61) claim that "the spelling of a given pitch number is primarily dependent on the key context, and to a lesser extent, the voice-leading tendencies in the music". Krumhansl (1990, p. 79) states that "once a key (or key region) has been determined, the correct spellings of the tones [i.e., pitch names] will be able to

be determined in most cases". Temperley (2001, p. 125) claims that his TPR 1, which states that notes nearby in the music should be assigned pitch names that are "close together" on the line of fifths, is "the most important" of the TPRs and that, "in many cases, this rule is sufficient to ensure the correct spelling of passages". Moreover, the high note accuracies obtained in my evaluations using Chew and Chen's algorithm (which ignores voice-leading) and the TPROne algorithm (see section 4.7.3) support the claim that the vast majority of notes in a passage of tonal music can be spelt correctly by taking only key into account and ignoring voice-leading.

In the algorithms of Longuet-Higgins, Temperley and Chew and Chen, the tonal centre at each time point in a passage of music is represented by *a single point* on either the line of fifths or the spiral array. However, the results of an experiment carried out by Krumhansl and Kessler (1982) suggest that two or more keys may be more or less strongly implied at any given point in a passage of tonal music (Krumhansl and Kessler, 1982; Krumhansl, 1990, pp. 214–226). In this experiment, Krumhansl and Kessler (1982) used the probe tone method to derive rating profiles for all the prefixes of ten chord sequences, each containing nine chords. In each trial of the experiment, a listener had to judge how well one of the 12 pitch classes fit with a particular sequence of chords forming a prefix of one of the ten chord sequences. The listeners showed strong agreement with each other, so their probe tone rating profiles were averaged. The correlation was then calculated between each of these 90 averaged rating profiles and each of the 24 tonal hierarchies for the major and minor keys obtained by Krumhansl and Kessler (1982) (see Krumhansl, 1990, pp. 25–31). Each of these calculated correlations was taken to represent the strength with which a particular key was implied at a particular point in one of the chord sequences. The results showed that, even in simple, non-modulating sequences of major and minor triads, there may be two or more keys that are more or less strongly implied at any given location in the music.

This suggests that it may be inappropriate to represent the perceived tonic at a given point in a passage by means of a single point on the line of fifths. This is effectively the same as using an "average" over all the tonics that may be implied to various extents at a given location in the music as a point estimate of the tonic. Such a representation would only be appropriate if the strengths with which the different tonics were implied were 'normally' distributed around some point on the line of fifths. If the most strongly implied tonics at a given point in the music are not typically adjacent on the line of fifths, then representing the tonic as an average, point estimate on the line of fifths would be inappropriate. For example, it might be that the two most likely tonics at a given point are the tonic of a major key and the tonic of that key's relative minor which would be a minor third below. In this situation, the two most strongly implied tonics would be three steps apart on the line of fifths, possibly giving a bi-modal distribution of strength of implication along the line of fifths. It may therefore be more appropriate to represent the sense of key at a point in the music by a *spectrum* indicating the strength with which each major and minor key is implied at a given location. Such a "key spectrum" might correspond to the set of 24 correlations with the tonal hierarchies for the major and minor keys obtained by Krumhansl and Kessler (1982) for each of the 90 probe-tone rating profiles obtained in their experiment to trace the developing and changing sense of key in chord sequences.

However, if we want to use such a key spectrum to represent the sense of key in a pitch spelling algorithm, we clearly cannot obtain the probe-tone rating profiles experimentally for every location in every piece of music. Fortunately, Krumhansl (1990, pp. 66–70) discovered that there are highly significant correlations between the major and minor tonal hierarchies, as measured experimentally by Krumhansl and Kessler (1982), and the frequency distributions of pitch classes in major and minor tonal works.

Krumhansl and Schmuckler used this result as the basis of a key-finding algorithm (Krumhansl, 1990, pp. 77–110). This algorithm takes as input a 12-vector, $\mathbf{I} = \langle d_1, d_2, \ldots d_{12} \rangle$, giving the total durations of the 12 pitch classes in the passage whose key is to be determined. Then, for each of the 12 major and 12 minor key probe-tone rating profiles $\mathbf{K}_i$ (Krumhansl and Kessler, 1982), the correlation r_i is calculated between $\mathbf{K}_i$ and $\mathbf{I}$. This gives a 24-vector, $\mathbf{R}$, representing a key spectrum, in which each entry gives the strength with which a particular key is implied in the passage being analysed.

Note that, in Krumhansl and Schmuckler's algorithm, the durations of the notes are taken into account. However, note duration was not taken into account in the data obtained by Youngblood (1958) and Knopoff and Hutchinson (1983) that Krumhansl (1990, pp. 66–70) used to show that there is a strong correlation between pitch class frequency distributions in tonal works and the probe-tone rating profiles for major and minor keys. This suggests that it may be possible to predict the sense of key accurately by simply counting the number of notes with each pitch class in a passage, giving equal weight to each note regardless of its duration.

Krumhansl and Kessler's (1982) probe tone rating profiles for the major and minor keys were obtained by asking musically experienced participants to judge how well each of the 12 pitch classes "fit with" various unambigous key-defining contexts (Krumhansl, 1990, pp. 25–31). For both major and minor keys, the highest rating was given to the tonic, followed by the other notes in the tonic triad, then the other scale tones and finally the non-scale tones (Krumhansl, 1990, pp. 29–31). Krumhansl (1990, p. 30) observes that the ordering of the probe-tone ratings for the 12 pitch classes "corresponds to the musical dimension variously known as relative stability, structural significance, priority, resolution, and rest". It therefore seems reasonable to interpret the probe-tone rating of a pitch class within one of these key profiles to be an indicator of how *tonic-like* that pitch class is perceived to be within that particular key context. This interpretation is supported by the fact that the major-key profile, obtained by Krumhansl and Shepard (1979) when musically trained listeners were asked to rate how well each pitch class *completed* an incomplete scale, was very similar to that obtained by Krumhansl and Kessler (1982) when listeners were asked to judge how well each pitch class "fit with" the key context.

As discussed above, Krumhansl (1990, pp. 66–70) observed that the probe tone rating profiles for the major and minor keys, as obtained by Krumhansl and Kessler (1982), correlate very well with the frequency distributions of pitch classes in major and minor tonal works. This implies that the frequency with which a pitch class occurs within a particular tonal context is typically a good indicator of how well it "fits with" that context—a result which seems almost self-evident, since one would hardly expect composers of tonal music to make frequent use of pitch classes that sound out of place. But, as discussed in the previous paragraph, it seems reasonable to

interpret the probe-tone rating of a pitch class within one of Krumhansl and Kessler's (1982) key profiles to be an indicator of how *tonic-like* that pitch class is perceived to be. Therefore, it seems likely that the frequency with which a pitch class occurs within a particular tonal context should typically be a good indicator of how tonic-like that pitch class is within that context. This, in turn, suggests that the frequency with which a pitch class occurs within a context surrounding some point in a piece of tonal music should provide a good measure of the likelihood of that pitch class being perceived to be the tonic at that point in the music. Therefore, in Stage 1 of *ps13*, the likelihood of a particular pitch class being that of the tonic at a given location in a passage is assumed to be proportional to the frequency with which that pitch class occurs within a region surrounding that location.

In Stage 1 of *ps13*, it is assumed that the notes are spelt so that they are as close as possible to the tonic on the line of fifths, with notes 6 semitones away from the tonic being spelt as sharpened subdominants rather than flattened dominants. This is the same as saying that, in *ps13*, it is assumed that notes are spelt as they are in the harmonic chromatic scale starting on the local tonic (Associated Board of the Royal Schools of Music, 1958, p. 78). Given this assumption, the pitch name, p, of a note, N, can be inferred if we know the pitch class of N and the pitch class, c_{t}, of the local tonic. In other words, for each note, N, (whose pitch class we know), each of the 12 possible local tonic pitch classes, c_{t}, implies a specific pitch name, p. It is assumed that the strength with which a particular local tonic pitch class, c_{t}, implies a pitch name, p, is proportional to the frequency with which c_{t} occurs within a region of the music surrounding N which I call the *context* of N. That is, the strength with which a particular pitch name, p, is implied by a particular local tonic pitch class, c_{t}, is assumed to be directly related to the likelihood of c_{t} being perceived to be the local tonic pitch class, which, as discussed in the previous paragraph, is, in turn, assumed to be proportional to the frequency with which c_{t} occurs in the context surrounding N. In general, a particular pitch name, p, may be implied for a note, N, whose pitch class we know, by more than one local tonic pitch class, c_{t}. In Stage 1 of *ps13*, the total strength with which a particular pitch name, p, is implied for a note, N, is therefore taken to be proportional to the sum of the frequencies of occurrence within the context of N of the local tonic pitch classes, c_{t}, that imply the pitch name, p. The most strongly implied pitch name, p, is then assigned to the note, N.

It can therefore be seen that Stage 1 of *ps13* only takes the local sense of key into account when assigning pitch names to the notes in a passage. As discussed earlier in this section, voice-leading also plays a part in determining pitch names. Stage 2 of *ps13* takes voice-leading into account by correcting those instances in the output of Stage 1 where a neighbour note or passing note is erroneously predicted to have the same letter name as either the note preceding it or the note following it (see Figure 6.5).

6.2 PS13: an implementation of *ps13*

Figure 6.1 shows an implementation called PS13 of the complete *ps13* algorithm just described in the previous section.As shown in Figure 6.1, lines 1–4 implement Stage 1 of the algorithm and lines 5–12 implement Stage 2. PS13 has three parameters: **SortedOCPList**, K_{pre} and K_{post}. **SortedOCPList** is an ordered set of ordered pairs in which each ordered pair, $\langle t_{\mathrm{on}}, p_{\mathrm{c}} \rangle$, gives

```
PS13(SortedOCPList, K_pre, K_post)
     ▶ Stage 1 begins here.
1    n ← |SortedOCPList|
2    ChromaList ← ComputeChromaList(SortedOCPList, n)
3    ChromaVectorList ← ComputeChromaVectorList(ChromaList, K_pre, K_post, n)
4    MorphList ← ComputeMorphList(ChromaList, ChromaVectorList, n)
     ▶ Stage 2 begins here.
5    OCMChordList ← ComputeOCMChordList(SortedOCPList, ChromaList, MorphList, n)
6    N_Ch ← |OCMChordList|
7    OCMChordList ← CorrectNeighbourNotes(OCMChordList, N_Ch)
8    OCMChordList ← CorrectDownwardPassingNotes(OCMChordList, N_Ch)
9    OCMChordList ← CorrectUpwardPassingNotes(OCMChordList, N_Ch)
10   MorphList ← ComputeNewMorphListFromOCMChordList(OCMChordList, N_Ch)
11   MorpheticPitchList ← ComputeMorpheticPitchList(SortedOCPList, MorphList, n)
12   return ComputeOPNList(SortedOCPList, MorpheticPitchList, n)
```

Figure 6.1: The PS13 algorithm.

```
ComputeChromaList(SortedOCPList, n)
1    ChromaList ← ⟨ ⟩
2    for i ← 0 to n − 1
3        Chroma ← SortedOCPList[i][1] mod 12
4        ChromaList ← ChromaList ⊕ ⟨Chroma⟩
5    return ChromaList
```

Figure 6.2: The ComputeChromaList function.

the onset time, t_{on}, and the chromatic pitch, p_{c}, of a single note or sequence of tied notes in the passage to be analysed. It is assumed that the elements of **SortedOCPList** have been sorted by increasing onset time and chromatic pitch, with priority given to onset time.

As explained in the previous section, in Stage 1 of *ps13*, the likelihood of a pitch class, c_{t}, being that of the tonic at the point where a note, N, occurs is assumed to be proportional to the frequency with which c_{t} occurs within a context surrounding N. The parameters, K_{pre} and K_{post}, of PS13 determine the size of this context. Specifically, the context for the note, N, represented by the ordered pair, **SortedOCPList**$[i]$, is **SortedOCPList**$[\mathit{ContextStart}, \mathit{ContextEnd}]$ where $\mathit{ContextStart} = \textsc{Max}(\{0, i - K_{\text{pre}}\})$ and $\mathit{ContextEnd} = \textsc{Min}(\{n, i + K_{\text{post}}\})$ where $n =$ |**SortedOCPList**|. K_{pre} must therefore be an integer greater than or equal to zero and K_{post} must be an integer greater than zero.

6.2.1 Implementing Stage 1 of *ps13*: Lines 1–4 of PS13

In line 1 of PS13 (see Figure 6.1), the variable, n, is set for convenience to equal the length of **SortedOCPList**. Then, in line 2, an ordered set of chromas, **ChromaList**, is derived from **SortedOCPList** such that |**ChromaList**| = |**SortedOCPList**| and **ChromaList**$[i]$ is the chroma of the note represented by **SortedOCPList**$[i]$ for all $0 \leq i < n$. This can be accomplished using the simple function, ComputeChromaList, defined in Figure 6.2. Note that the parameter, n, in ComputeChromaList must be equal to |**SortedOCPList**|.

Next, in line 3 of PS13 (see Figure 6.1), the function ComputeChromaVectorList, defined in Figure 6.3, is used to construct an ordered set of 12-vectors, **ChromaVectorList**, such that |**ChromaVectorList**| = |**SortedOCPList**| = n and

```
ComputeChromaVectorList(ChromaList, K_pre, K_post, n)
1   ThisVector ← ⟨0,0,0,0,0,0,0,0,0,0,0,0⟩
2   for i ← 0 to Min({n, K_post}) − 1
3      ThisVector[ChromaList[i]] ← 1 + ThisVector[ChromaList[i]]
4   ChromaVectorList ← ⟨ThisVector⟩
5   for i ← 1 to n − 1
6      if i + K_post ≤ n
7         ThisVector[ChromaList[i + K_post − 1]] ← 1 + ThisVector[ChromaList[i + K_post − 1]]
8      if i − K_pre > 0
9         ThisVector[ChromaList[i − K_pre − 1]] ← ThisVector[ChromaList[i − K_pre − 1]] − 1
10     ChromaVectorList ← ChromaVectorList ⊕ ⟨ThisVector⟩
11  return ChromaVectorList
```

Figure 6.3: The ComputeChromaVectorList function.

ChromaVectorList$[i]$ represents the chroma frequency distribution within the context surrounding **SortedOCPList**$[i]$ defined by K_{pre} and K_{post} (i.e., the chroma frequency distribution within **SortedOCPList**$[\text{Max}(\{0, i - K_{\text{pre}}\}), \text{Min}(\{n, i + K_{\text{post}}\})]$). Note that the parameter, n, in ComputeChromaVectorList must be equal to |**ChromaList**| and the parameters, K_{pre} and K_{post}, in ComputeChromaVectorList are equal to those given by the user as arguments to PS13. Therefore, when line 3 of PS13 has completed, **ChromaVectorList**$[i][c]$ gives the frequency with which chroma, c, occurs within the context, **SortedOCPList**$[\text{Max}(\{0, i - K_{\text{pre}}\}), \text{Min}(\{n, i + K_{\text{post}}\})]$, surrounding the note represented by **SortedOCPList**$[i]$.

The function ComputeChromaVectorList, defined in Figure 6.3, accomplishes its task efficiently. It first computes the chroma frequency distribution for the context surrounding the note represented by the first element in **SortedOCPList** and stores this distribution in the 12-vector, **ThisVector**, (lines 1–3). The variable, **ChromaVectorList**, is then initialized in line 4 so that it contains just **ThisVector**. Each subsequent chroma frequency distribution, **ChromaVectorList**$[i]$ $(1 \leq i < n)$, is calculated from the preceding one by adding 1 to the element in **ThisVector** representing the frequency of the chroma **ChromaList**$[i + K_{\text{post}} - 1]$ (if $i + K_{\text{post}} \leq n$) and/or subtracting 1 from the element in **ThisVector** representing the frequency of the chroma, **ChromaList**$[i - K_{\text{pre}} - 1]$, (if $i - K_{\text{pre}} > 0$). This is done in lines 5–9 of ComputeChromaVectorList and then the updated value of **ThisVector** is appended to **ChromaVectorList** in line 10.

The ultimate purpose of Stage 1 (i.e., lines 1–4) of PS13 is to assign to each element of **SortedOCPList** the morph which is most strongly implied by the context around that element. Given the morph and chromatic pitch of a note, its pitch name can be computed directly (given certain reasonable assumptions which will be discussed below in connection with the ComputeMorpheticPitchList function, defined in Figure 6.12).

As explained above (section 6.1), in Stage 1 of *ps13*, each note, N, (whose pitch class we know) is assigned the most strongly implied pitch name, where the strength with which a given pitch name, p, is implied is assumed to be proportional to the sum of the frequencies of occurrence within the context of N of the local tonic pitch classes that imply the pitch name, p. It was also stated that, if we assume that notes are spelt as they are in the harmonic chromatic scale and we know the pitch classes of the local tonic and the note to be spelt, then the pitch name of

the note can be inferred. However, this is a simplification: strictly speaking, we can only infer the pitch name of the note if we also know the pitch name class assigned to the tonic where the note occurs. Unfortunately, we do not, in general, know the pitch name class to be assigned to each tonic chroma at each point in a passage. Therefore, in the PS13 implementation of *ps13*, I make the simplifying assumption that the pitch name class associated with a given tonic chroma remains constant throughout the passage to be analysed. For example, if the pitch name class associated with the tonic chroma, 3, at the beginning of the passage to be analysed is `"Cn"`, then it is assumed that the pitch name class, `"Cn"`, is associated with the tonic chroma, 3, throughout the passage.

In order to explain how PS13 implements Stage 1 of *ps13*, the points made in the previous paragraph need to be re-considered more formally. Let's suppose that the variable, **ChromaList**, computed in line 2 of PS13, is equal to $\langle c_0, c_1, \ldots c_{n-1}\rangle$, so that c_j denotes the $(j+1)$th element of **ChromaList**. The task to be accomplished in lines 1–4 of PS13 is to construct a list of morphs,

$$\textbf{MorphList} = \langle m_0, m_1, \ldots m_{n-1}\rangle\,,$$

such that m_j is the morph that is most strongly implied by the context around chroma, c_j. Let $m(c_\mathrm{t}, j)$ denote the morph that should be assigned to the chroma, c_j, if the chroma of the tonic where c_j occurs is c_t; and let $m_\mathrm{t}(c_\mathrm{t}, j)$ denote the tonic morph associated with c_t at the point where c_j occurs. If c_j is spelt in accordance with the harmonic chromatic scale, then

$$m(c_\mathrm{t}, j) = (\textbf{MorphInt}[(c_j - c_\mathrm{t}) \bmod 12] + m_\mathrm{t}(c_\mathrm{t}, j)) \bmod 7, \tag{6.1}$$

where

$$\textbf{MorphInt} = \langle 0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6\rangle\,. \tag{6.2}$$

For example, if the local tonic is C natural, then $c_\mathrm{t} = 3$, $m_\mathrm{t}(c_\mathrm{t}, j) = 2$ and the definition of **MorphInt** ensures that the pitch name classes assigned will be between D♭ and F♯, inclusive, on the line of fifths—that is, spelt as in the harmonic chromatic scale on C.

Let $C_\mathrm{t}(m, j)$ denote the set of tonic chromas that imply that the morph, m, should be assigned to c_j. That is,

$$C_\mathrm{t}(m, j) = \{c_\mathrm{t} \mid m(c_\mathrm{t}, j) = m\}\,. \tag{6.3}$$

It follows that the strength, $S(m, j)$, with which morph, m, is implied as the spelling for c_j is given by

$$S(m, j) = \sum_{c_\mathrm{t} \in C_\mathrm{t}(m,j)} \textbf{ChromaVectorList}[j][c_\mathrm{t}], \tag{6.4}$$

where **ChromaVectorList** is the list of 12-vectors computed in line 3 of PS13. Therefore, the morph, m_j, that should be assigned to c_j in Stage 1 of PS13 is given by

$$m_j = \textsc{Pos}(\textsc{Max}(\langle S(0, j), S(1, j), \ldots S(6, j)\rangle), \langle S(0, j), S(1, j), \ldots S(6, j)\rangle). \tag{6.5}$$

The foregoing paragraph suggests that each m_j in **MorphList** can be computed by

1. computing $m(c_\mathrm{t}, j)$ for all $0 \leq c_\mathrm{t} \leq 11$ using Eq. 6.1;

2. computing $C_t(m, j)$ for all $0 \leq m \leq 6$ using Eq. 6.3;

3. using **ChromaVectorList** to compute $S(m, j)$ for all $0 \leq m \leq 6$ in accordance with Eq. 6.4; and

4. using Eq. 6.5 to compute m_j.

However, in order to carry out the first of these four steps, we need to know $m_t(c_t, j)$ for all $0 \leq c_t \leq 11$ and all $0 \leq j < n$, which, unfortunately, we do not. We therefore have to make an assumption about each value of $m_t(c_t, j)$ and the assumption made in PS13 is that $m_t(c_t, j) = m_t(c_t, 0)$ for all $0 \leq j < n$. In other words, it is assumed in PS13 that the morph associated with a given c_t is constant throughout the passage to be analysed.

This still leaves the problem, of course, of determining $m_t(c_t, 0)$ for all $0 \leq c_t \leq 11$. However, this problem can be easily solved by assigning a reasonable but otherwise arbitrary morph to c_0. This can be done, for example, by setting

$$m_0 = \textbf{InitMorph}[c_0], \tag{6.6}$$

where

$$\textbf{InitMorph} = \langle 0, 1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 6 \rangle . \tag{6.7}$$

$m_t(c_t, 0)$ is then given by

$$m_t(c_t, 0) = (m_0 - \textbf{MorphInt}[(c_0 - c_t) \bmod 12]) \bmod 7. \tag{6.8}$$

It should be stressed that the choice of values in the definition of **InitMorph** is, strictly speaking, arbitrary. However, the values were chosen here so that the initial note would be assigned a pitch name class between E♭ and G♯ on the line of fifths, as this was found to be the range of 12 consecutive steps on the line of fifths that contained the pitch name classes of the largest proportion of notes in the test corpus used here (see section 7.1 and Figure 7.1).

It follows that **MorphList** can therefore be computed by, first, computing $m_t(c_t, 0)$ for all $0 \leq c_t \leq 11$ using Eq. 6.8; and, then, carrying out the four steps given in the previous paragraph for each value of j between 0 and $n - 1$. This strategy is implemented directly in the function ComputeMorphList, defined in Figure 6.4.

In line 1 of ComputeMorphList, the variable, **MorphList**, is initialized to equal a list of length $n = |\textbf{ChromaList}|$, in which every element is equal to nil. In lines 2–4 of ComputeMorphList, **InitMorph** and m_0 are set in accordance with Eqs. 6.7 and 6.6, respectively, and the variable, c_0, is set for convenience to equal **ChromaList**[0]. In line 5, the variable, **MorphInt**, is set in accordance with Eq. 6.2. Then, in lines 6–8 of ComputeMorphList, the variable, **TonicMorphForTonicChroma**, is set to equal

$$\langle m_t(0, 0), m_t(1, 0), \ldots m_t(11, 0) \rangle$$

in accordance with Eq. 6.8. In lines 9–11, the variables, **MorphForTonicChroma**, **TonicChromaSetForMorph** and **MorphStrength**, are initialized to be lists of appropriate length in which all elements are equal to nil. On each iteration of the 'for' loop starting in line 12, the

```
ComputeMorphList(ChromaList, ChromaVectorList, n)
1   MorphList ← ⊕_{j=1}^{n} ⟨nil⟩
    ▶ First compute m_0.
2   InitMorph ← ⟨0, 1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 6⟩
3   c_0 ← ChromaList[0]
4   m_0 ← InitMorph[c_0]
5   MorphInt ← ⟨0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6⟩
    ▶ Compute Eq. 6.8 for 0 ≤ c_t ≤ 11.
6   TonicMorphForTonicChroma ← ⟨nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil⟩
7   for c_t ← 0 to 11
8       TonicMorphForTonicChroma[c_t] ← (m_0 − MorphInt[(c_0 − c_t) mod 12]) mod 7
9   MorphForTonicChroma ← ⟨nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil⟩
10  TonicChromaSetForMorph ← ⟨nil, nil, nil, nil, nil, nil, nil⟩
11  MorphStrength ← ⟨nil, nil, nil, nil, nil, nil, nil⟩
12  for j ← 0 to n − 1
        ▶ Compute Eq. 6.1 for 0 ≤ c_t ≤ 11.
13      for c_t ← 0 to 11
14          c ← ChromaList[j]
15          MorphForTonicChroma[c_t] ← (MorphInt[(c − c_t) mod 12] + TonicMorphForTonicChroma[c_t]) mod 7
        ▶ Compute Eq. 6.3 for 0 ≤ m ≤ 6.
16      for m ← 0 to 6
17          TonicChromaSetForMorph[m] ← nil
18      for m ← 0 to 6
19          for c_t ← 0 to 11
20              if MorphForTonicChroma[c_t] = m
21                  TonicChromaSetForMorph[m] ← ⟨c_t⟩ ⊕ TonicChromaSetForMorph[m]
        ▶ Compute Eq. 6.4 for 0 ≤ m ≤ 6.
22      for m ← 0 to 6
23          MorphStrength[m] ← Σ_{c_t ∈ TonicChromaSetForMorph[m]} ChromaVectorList[j][c_t]
        ▶ Compute Eq. 6.5.
24      MorphList[j] ← Pos(Max(MorphStrength), MorphStrength)
25  return MorphList
```

Figure 6.4: The ComputeMorphList function.

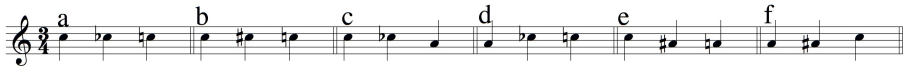


Figure 6.5: Examples of the types of passing and neighbour note errors corrected in Stage 2 of *ps13*.

morph is computed for a single element of **ChromaList**. For each chroma, **ChromaList**$[j]$, Eq. 6.1 is first computed for all $0 \leq c_{\mathrm{t}} \leq 11$ in lines 13–15 and the variable, **MorphForTonicChroma**, becomes equal to

$$\langle m(0,j), m(1,j), \ldots m(11,j) \rangle .$$

Then, in lines 16–21, Eq. 6.3 is computed for all $0 \leq m \leq 6$ and the variable, **TonicChromaSetForMorph**, becomes equal to

$$\langle C_{\mathrm{t}}(0,j), C_{\mathrm{t}}(1,j), \ldots C_{\mathrm{t}}(6,j) \rangle .$$

Next, in lines 22–23 of ComputeMorphList, Eq. 6.4 is computed for all $0 \leq m \leq 6$ and the variable, **MorphStrength**, becomes equal to

$$\langle S(0,j), S(1,j), \ldots S(6,j) \rangle .$$

Finally, in line 24 of ComputeMorphList, Eq. 6.5 is computed and **MorphList**$[j]$ is set to the appropriate value. Note that, at line 24 in ComputeMorphList, if **MorphStrength**$[k]$ = Max(**MorphStrength**) for two or more values of k, then Pos(Max(**MorphStrength**), **MorphStrength**) arbitrarily returns the least value of k for which **MorphStrength**$[k]$ = Max(**MorphStrength**). One way of improving this implementation of *ps13* might therefore be to incorporate rules for choosing, in a principled way, the best value of k in this type of situation (see section 6.6.1 below). Once the morph for each element of **ChromaList** has been calculated and stored in **MorphList**, **MorphList** is returned in line 25 of ComputeMorphList.

6.2.2 Implementing Stage 2 of *ps13*: Lines 5–12 of PS13

As explained near the end of section 6.1, the purpose of Stage 2 of *ps13* is to take voice-leading into account by correcting those cases in the output of Stage 1 where a neighbour note or passing note is erroneously predicted to have the same letter name (i.e., morph) as either the note preceding it or the note following it (see Figure 6.5).

Stage 2 of *ps13* is implemented in lines 5–12 of PS13 (see Figure 6.1). In lines 5–10, the variable, **MorphList**, is modified to produce a new list of morphs in which cases like the ones in Figure 6.5 have been corrected. In line 11, this new value of **MorphList** is used in conjunction with the chromatic pitches in **SortedOCPList** to compute a morphetic pitch for each element in **SortedOCPList** which is stored in the list, **MorpheticPitchList**. Finally, in line 12, the morphetic pitches in **MorpheticPitchList** are used in conjunction with the chromatic pitches in **SortedOCPList** to compute pitch names for all the notes in the passage being analysed and

```
ComputeOCMChordList(SortedOCPList, ChromaList, MorphList, n)
1   OCMList ← ⟨⟩
2   for i ← 0 to n − 1
3      OCMList ← OCMList ⊕ ⟨⟨SortedOCPList[i][0], ChromaList[i], MorphList[i]⟩⟩
4   OCMChordList ← ⟨⟨OCMList[0]⟩⟩
5   for i ← 1 to n − 1
6      if OCMList[i][0] = OCMList[i − 1][0]
7         OCMChordList[|OCMChordList| − 1] ← OCMChordList[|OCMChordList| − 1] ⊕ ⟨OCMList[i]⟩
8      else
9         OCMChordList ← OCMChordList ⊕ ⟨⟨OCMList[i]⟩⟩
10  return OCMChordList
```

Figure 6.6: The ComputeOCMChordList function.

the algorithm outputs a list of ordered pairs in which each ordered pair gives the onset time and pitch name of a single note or sequence of tied notes.

As just explained, the purpose of lines 5–10 of PS13 is to take the list of morphs, **MorphList**, generated by the ComputeMorphList function in line 4, and correct errors like the ones in Figure 6.5, producing a new, corrected value for **MorphList**. The first step in this process is carried out by the function ComputeOCMChordList, which is called in line 5 of PS13 and defined in Figure 6.6. In lines 1–3 of ComputeOCMChordList, the value of **SortedOCPList** provided by the user to PS13,

$$\textbf{SortedOCPList} = \langle\langle t_{\text{on},0}, p_{\text{c},0}\rangle \langle t_{\text{on},1}, p_{\text{c},1}\rangle \ldots \langle t_{\text{on},n-1}, p_{\text{c},n-1}\rangle\rangle\,,$$

together with the value of **ChromaList** computed in line 2 of PS13,

$$\textbf{ChromaList} = \langle c_0, c_1, \ldots c_{n-1}\rangle\,,$$

and **MorphList**, computed in line 4 of PS13,

$$\textbf{MorphList} = \langle m_0, m_1, \ldots m_{n-1}\rangle\,,$$

are used to compute a list of triples,

$$\textbf{OCMList} = \langle\langle t_{\text{on},0}, c_0, m_0\rangle, \langle t_{\text{on},1}, c_1, m_1\rangle \ldots \langle t_{\text{on},n-1}, c_{n-1}, m_{n-1}\rangle\rangle\,. \tag{6.9}$$

Then, in lines 4–10 of ComputeOCMChordList, this list of triples, **OCMList**, is partitioned into classes, which I shall call *chords*, such that each chord contains all and only those triples, $\langle t_{\text{on},j}, c_j, m_j\rangle \in \textbf{OCMList}$, for which the onset time, $t_{\text{on},j}$, is equal to a particular value. In other words, no two chords contain triples whose onset times are the same, and all the triples within a particular chord have the same onset time. These chords are stored, in increasing order of onset time, in the ordered set, **OCMChordList**, which is returned in line 10 of ComputeOCMChordList.

In line 6 of PS13, the number of elements (i.e., chords) in the list, **OCMChordList**, computed by ComputeOCMChordList in line 5, is stored, for convenience, in the variable, N_{Ch}. Then, in line 7 of PS13, the function CorrectNeighbourNotes, defined in Figure 6.7,

```
CorrectNeighbourNotes(OCMChordList, N_Ch)
1    i ← 0
2    while i < N_Ch − 2
3        for j ← 0 to |OCMChordList[i]| − 1
4            Note_1 ← OCMChordList[i][j]
5            if Note_1[1, 3] ∈ ⋃_{k=0}^{|OCMChordList[i+2]|−1} {OCMChordList[i + 2][k][1, 3]}
6                for ℓ ← 0 to |OCMChordList[i + 1]| − 1
7                    Note_2 ← OCMChordList[i + 1][ℓ]
8                    if Note_1[2] = Note_2[2]
9                        if (Note_2[1] − Note_1[1]) mod 12 ∈ {1, 2}
10                           OCMChordList[i + 1][ℓ][2] ← (Note_2[2] + 1) mod 7
11                       if (Note_1[1] − Note_2[1]) mod 12 ∈ {1, 2}
12                           OCMChordList[i + 1][ℓ][2] ← (Note_2[2] − 1) mod 7
13       i ← i + 1
14   return OCMChordList
```

Figure 6.7: The CorrectNeighbourNotes function.

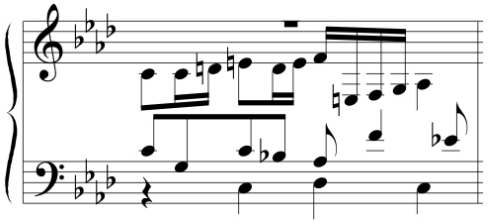


Figure 6.8: Bar 7 from J. S. Bach's Fugue in F minor (BWV 857), chosen to illustrate problems with CorrectNeighbourNotes function.

takes **OCMChordList** as input and corrects neighbour-note voice-leading errors such as the ones in Figure 6.5 (a) and (b). Let's suppose that

$$\mathbf{OCMChordList} = \langle H_0, H_1, \ldots H_{N_{\mathrm{Ch}}-1} \rangle .$$

CorrectNeighbourNotes considers, in turn, each sequence of three chords, $\langle H_i, H_{i+1}, H_{i+2} \rangle$, $(0 \leq i \leq N_{\mathrm{Ch}} - 3)$. For each three-chord sequence, $\langle H_i, H_{i+1}, H_{i+2} \rangle$, the function takes each note in H_i, stores this note in the variable, $\mathbf{Note}_1$, (line 4) and determines whether there is a note in H_{i+2} that has the same morph and chroma (line 5). If there is, each note in H_{i+1} (stored in $\mathbf{Note}_2$) is checked to see if it has the same morph as $\mathbf{Note}_1$ (line 8). If $\mathbf{Note}_1$ and $\mathbf{Note}_2$ have the same morph, then line 9 determines whether $\mathbf{Note}_2$ is 1 or 2 semitones above $\mathbf{Note}_1$ (i.e., $(\mathbf{Note}_2[1] - \mathbf{Note}_1[1]) \bmod 12 \in \{1, 2\}$). If $\mathbf{Note}_2$ has the same morph as $\mathbf{Note}_1$ and it is 1 or 2 semitones above $\mathbf{Note}_1$, then it is assumed to be an incorrectly spelt upper neighbour note and its morph is increased by 1 (mod 7) in line 10. Also, if $\mathbf{Note}_1$ and $\mathbf{Note}_2$ have the same morph, then line 11 determines whether $\mathbf{Note}_2$ is 1 or 2 semitones below $\mathbf{Note}_1$ (i.e., $(\mathbf{Note}_1[1] - \mathbf{Note}_2[1]) \bmod 12 \in \{1, 2\}$). If $\mathbf{Note}_2$ has the same morph as $\mathbf{Note}_1$ and it is 1 or 2 semitones below $\mathbf{Note}_1$, then it is assumed to be an incorrectly spelt lower neighbour note and its morph is decreased by 1 (mod 7) in line 12. When all the three-chord sequences have been checked and, if necessary, modified in this way, the new value of **OCMChordList** is returned in line 14.

The function CorrectNeighbourNotes implements what is clearly only a very crude

```
CorrectDownwardPassingNotes(OCMChordList, N_Ch)
1   i ← 0
2   while i < N_Ch − 2
3     for j ← 0 to |OCMChordList[i]| − 1
4       Note_1 ← OCMChordList[i][j]
5       for k ← 0 to |OCMChordList[i + 2]| − 1
6         Note_3 ← OCMChordList[i + 2][k]
7         if Note_3[2] = (Note_1[2] − 2) mod 7
8           for ℓ ← 0 to |OCMChordList[i + 1]| − 1
9             Note_2 ← OCMChordList[i + 1][ℓ]
10            if (Note_2[2] ∈ {Note_1[2], Note_3[2]})
                    ∧(0 < (Note_1[1] − Note_2[1]) mod 12 < (Note_1[1] − Note_3[1]) mod 12)
11              CanChange ← true
12              for m ← 0 to |OCMChordList[i + 1]| − 1
13                Note ← OCMChordList[i + 1][m]
14                if (Note[2] = (Note_1[2] − 1) mod 7) ∧ (Note[1] ≠ Note_2[1])
15                  CanChange ← false
16              if CanChange
17                OCMChordList[i + 1][ℓ][2] ← (Note_1[2] − 1) mod 7
18    i ← i + 1
19  return OCMChordList
```

Figure 6.9: The CorrectDownwardPassingNotes function.

method for correcting incorrectly spelt neighbour notes. For example, if the input data encoded in **SortedOCPList** is derived from a MIDI file generated from a performance of the passage to be analysed, then notes that are notated as starting simultaneously may well have slightly different onset times in **SortedOCPList**. If this is the case, then notes that should be members of the same chord may become members of different chords in **OCMChordList**. In this situation, a note that is actually an incorrectly spelt neighbour note may not occur in the chord that immediately follows the chord containing the note that precedes it in the neighbour note pattern. Such an incorrectly spelt neighbour note would therefore not be corrected by the CorrectNeighbourNotes function. For example, if the `"Dn4"` semiquaver in the alto voice, 3/8 of the way through the bar in Figure 6.8, were spelt as an `"Eff4"` and the tenor `"Bf3"` quaver were played very slightly before this incorrectly spelt alto neighbour note, then the CorrectNeighbourNotes function would fail to correct this error. Also, CorrectNeighbourNotes will fail to correct a neighbour note if another note, not involved in the neighbour note pattern, starts between the incorrectly spelt neighbour note and the note that precedes it in the pattern. For example, if the `"Df3"` crotchet in the bass part in Figure 6.8 were spelt as a `"Cs3"`, CorrectNeighbourNotes would fail to correct it, even if all the note onsets were strictly proportional to their notated values. This is because the bass `"Cn3"` crotchet which precedes this neighbour note would not be contained within the chord in **OCMChordList** preceding that containing the neighbour note—in fact, there would be two intervening chords between the one containing the first bass `"Cn3"` and the chord containing the bass neighbour note `"Df3"`, incorrectly spelt as `"Cs3"`.

Having corrected (some of) the incorrectly spelt neighbour notes, the function CorrectDownwardPassingNotes, defined in Figure 6.9, is then used in line 8 of PS13 to correct (some of the) passing note errors like examples (c) and (e) in Figure 6.5. Again, let's suppose that **OCMChordList** $= \langle H_0, H_1, \ldots H_{N_{\text{Ch}}-1} \rangle$. Like CorrectNeighbourNotes, CorrectDownwardPassingNotes considers, in turn, each sequence of three chords, $\langle H_i, H_{i+1}, H_{i+2} \rangle$,

($0 \leq i \leq N_{\text{Ch}} - 3$). For each three-chord sequence, $\langle H_i, H_{i+1}, H_{i+2} \rangle$, the function takes each note in H_i, stores this note in the variable, $\mathbf{Note}_1$, (line 4) and then checks each note in H_{i+2} (stored in $\mathbf{Note}_3$ in line 6) to determine if its morph is two less than that of $\mathbf{Note}_1$ (mod 7) (see line 7). If this is the case for a particular value of $\mathbf{Note}_3$, this means that the pitch interval between $\mathbf{Note}_1$ and $\mathbf{Note}_3$ is octave-equivalent to a falling third. In this case, the function checks each note in H_{i+1} (stored in $\mathbf{Note}_2$ in line 9) to see if

1. it has the same morph as either $\mathbf{Note}_1$ or $\mathbf{Note}_3$ (hence the expression,

$$\mathbf{Note}_2[2] \in \{\mathbf{Note}_1[2], \mathbf{Note}_3[2]\},$$

   in line 10); and

2. the chroma interval from $\mathbf{Note}_2$ to $\mathbf{Note}_1$ is greater than zero and less than that from $\mathbf{Note}_3$ to $\mathbf{Note}_1$ (hence the expression,

$$0 < (\mathbf{Note}_1[1] - \mathbf{Note}_2[1]) \bmod 12 < (\mathbf{Note}_1[1] - \mathbf{Note}_3[1]) \bmod 12,$$

   in line 10).

If both these conditions are satisfied, then the note stored in $\mathbf{Note}_2$ is a candidate for correction. However, this note only has its morph changed to being one less (mod 7) than that of $\mathbf{Note}_1$ (line 17), if there is not already a note in H_{i+1} with a morph one less (mod 7) than that of $\mathbf{Note}_1$ and a chroma that is not equal to that of $\mathbf{Note}_2$ (see line 14). This ensures that the 'correction' does not lead to two or more notes with different chromas having the same morph in the same chord.

Like the CorrectNeighbourNotes function, CorrectDownwardPassingNotes is only a crude implementation of a method for correcting downward passing note errors and suffers from similar short-comings to those of the CorrectNeighbourNotes function described above.

Having corrected (some of) the incorrectly spelt downward passing notes, the function CorrectUpwardPassingNotes, defined in Figure 6.10, is then used in line 9 of PS13 to correct (some of the) passing note errors like examples (d) and (f) in Figure 6.5. The CorrectUpwardPassingNotes function works in essentially the same way as the CorrectDownwardPassingNotes function just described.

Having completed the process of correcting various types of voice-leading error, a modified **MorphList** is produced from **OCMChordList** in line 10 of PS13, using the trivial function ComputeNewMorphListFromOCMChordList, defined in Figure 6.11.

Then, in line 11 of PS13, this corrected **MorphList** is given, together with **SortedOCPList**, as input to the function ComputeMorpheticPitchList, defined in Figure 6.12, which returns a list of morphetic pitches which is stored in the variable **MorpheticPitchList**. ComputeMorpheticPitchList uses the morph and the chromatic pitch of each note to predict the most likely morphetic pitch for that note. Let's suppose that a note has a chromatic pitch, p_{c}, and a morph, m. Now, it is possible for a note's morphetic octave to be different from its chromatic octave. For example, the morphetic octave of `"Af4"` is 4 whereas its chromatic octave is 3.

```
CorrectUpwardPassingNotes(OCMChordList, N_Ch)
1   i ← 0
2   while i < N_Ch − 2
3      for j ← 0 to |OCMChordList[i]| − 1
4         Note_1 ← OCMChordList[i][j]
5         for k ← 0 to |OCMChordList[i + 2]| − 1
6            Note_3 ← OCMChordList[i + 2][k]
7            if Note_3[2] = (Note_1[2] + 2) mod 7
8               for ℓ ← 0 to |OCMChordList[i + 1]| − 1
9                  Note_2 ← OCMChordList[i + 1][ℓ]
10                 if (Note_2[2] ∈ {Note_1[2], Note_3[2]})
                          ∧(0 < (Note_3[1] − Note_2[1]) mod 12 < (Note_3[1] − Note_1[1]) mod 12)
11                    CanChange ← true
12                    for m ← 0 to |OCMChordList[i + 1]| − 1
13                       Note ← OCMChordList[i + 1][m]
14                       if (Note[2] = (Note_1[2] + 1) mod 7) ∧ (Note[1] ≠ Note_2[1])
15                          CanChange ← false
16                    if CanChange
17                       OCMChordList[i + 1][ℓ][2] ← (Note_1[2] + 1) mod 7
18     i ← i + 1
19  return OCMChordList
```

Figure 6.10: The CorrectUpwardPassingNotes function.

```
ComputeNewMorphListFromOCMChordList(OCMChordList, N_Ch)
1   OCMList ← ⟨ ⟩
2   for i ← 0 to N_Ch − 1
3      OCMList ← OCMList ⊕ OCMChordList[i]
4   return ⨁_{j=0}^{|OCMList|−1} ⟨OCMList[j][2]⟩
```

Figure 6.11: The ComputeNewMorphListFromOCMChordList function.

```
ComputeMorpheticPitchList(SortedOCPList, MorphList, n)
1   MorpheticPitchList ← ⟨ ⟩
2   for i ← 0 to n − 1
3      ChromaticPitch ← SortedOCPList[i][1]
4      Morph ← MorphList[i]
5      MorphOct_1 ← ⌊ChromaticPitch/12⌋
6      MorphOct_2 ← MorphOct_1 + 1
7      MorphOct_3 ← MorphOct_1 − 1
8      MP_1 ← MorphOct_1 + (Morph/7)
9      MP_2 ← MorphOct_2 + (Morph/7)
10     MP_3 ← MorphOct_3 + (Morph/7)
11     Chroma ← ChromaticPitch mod 12
12     CP ← MorphOct_1 + (Chroma/12)
13     DiffList ← ⟨|CP − MP_1|, |CP − MP_2|, |CP − MP_3|⟩
14     MorphOctList ← ⟨MorphOct_1, MorphOct_2, MorphOct_3⟩
15     BestMorphOct ← MorphOctList[Pos(Min(DiffList), DiffList)]
16     BestMorpheticPitch ← Morph + (7 × BestMorphOct)
17     MorpheticPitchList ← MorpheticPitchList ⊕ ⟨BestMorpheticPitch⟩
18  return MorpheticPitchList
```

Figure 6.12: The ComputeMorpheticPitchList function.

```
ComputeOPNList(SortedOCPList, MorpheticPitchList, n)
1    OPNList ← ⟨ ⟩
2    for i ← 0 to n − 1
3        PN ← P2PN(⟨SortedOCPList[i][1], MorpheticPitchList[i]⟩)
4        OPNList ← OPNList ⊕ ⟨⟨SortedOCPList[i][0], PN⟩⟩
5    return OPNList
```

Figure 6.13: The ComputeOPNList function.

However, if the difference between the morphetic octave and the chromatic octave is more than 1, the pitch name of the note must have at least 13 flats or sharps. Therefore we may safely assume in PS13 that the difference between the chromatic and morphetic octaves of a note will never be more than 1. Therefore, if we're given the morph, m, and chromatic pitch, p_c, of a note, we may safely assume that its morphetic octave, o_m, will be a member of the set, $\{o_c, o_c + 1, o_c - 1\}$, where o_c, the chromatic octave, is equal to $\lfloor p_c/12 \rfloor$. We are therefore left with the problem of deciding for each note whether its morphetic octave should be o_c, $o_c + 1$ or $o_c - 1$. Fortunately, this can easily be solved correctly in almost every case by choosing the value of o_m for which $|o_m + (m/7) - (o_c + (c/12))|$ is a minimum, where $c = p_c \bmod 12$. Having found an appropriate morphetic octave, o_m, for a note, its morphetic pitch, $p_m = m + 7o_m$, can be computed directly. This is precisely the strategy implemented in the function ComputeMorpheticPitchList.

Finally, in line 12 of PS13, the ComputeOPNList function, defined in Figure 6.13, uses the list of morphetic pitches in **MorpheticPitchList** in conjunction with the chromatic pitches given in **SortedOCPList** to compute a pitch name for each note. The pitch name for each note is computed from its chromatic and morphetic pitches in line 3 of ComputeOPNList using the P2PN function defined in Figure 1.13.

6.3 Computational complexity of PS13

As already mentioned near the beginning of section 6.2, it is assumed in PS13 that the elements in **SortedOCPList** have been sorted by increasing onset time and chromatic pitch, with priority given to onset time. If this is not the case, then this data has to be sorted in a pre-processing phase which would require $O(n \log n)$ time in the worst case where n is the number of notes in the input passage.

Lines 1 and 6 of PS13 can each be executed in constant time, provided that **SortedOCPList** and **OCMChordList** are stored appropriately.

It is straightforward to see that lines 2, 3, 4 and 5 of PS13 each run in $O(n)$ time in the worst case, where $n = |\mathbf{SortedOCPList}|$. The worst-case time complexity of Stage 1 of PS13 is therefore $O(n)$.

It is slightly less straightforward to see that the worst-case running time of the CorrectNeighbourNotes function, called in line 7 of PS13 and defined in Figure 6.7, is also linear in the number of notes in the input passage. Each execution of each line apart from line 5 in CorrectNeighbourNotes takes $O(1)$ time. Each execution of line 5 takes $O(|\mathbf{OCMChordList}[i + 2]|)$ time. The total number of times that line 5 executes during one

run of CorrectNeighbourNotes is

$$\sum_{i=0}^{N_{\text{Ch}}-3} (|\textbf{OCMChordList}[i]|).$$

Therefore, the total time spent executing line 5 during one run of CorrectNeighbourNotes is

$$T_5 = \sum_{i=0}^{N_{\text{Ch}}-3} (|\textbf{OCMChordList}[i]| \times O(|\textbf{OCMChordList}[i+2]|)). \tag{6.10}$$

We know that

$$\sum_{i=0}^{N_{\text{Ch}}-1} (|\textbf{OCMChordList}[i]|) = n, \tag{6.11}$$

where $n = |\textbf{SortedOCPList}|$ (i.e., n is the total number of notes in the input passage). Therefore, to simplify the analysis, let's assume that the notes are distributed evenly between all the chords, so that each chord in **OCMChordList** contains n/N_{Ch} notes, that is

$$\textbf{OCMChordList}[i] = n/N_{\text{Ch}}, \text{ for all } 0 \leq i < N_{\text{Ch}}. \tag{6.12}$$

If we substitute Eq. 6.12 into Eq. 6.10, we find that the total amount of time spent on executing line 5 during one execution of CorrectNeighbourNotes is

$$\begin{aligned} T_5 &= \sum_{i=0}^{N_{\text{Ch}}-3} \left(\frac{n}{N_{\text{Ch}}} \times O\left(\frac{n}{N_{\text{Ch}}} \right) \right) \\ &= (N_{\text{Ch}} - 2)\, O\left(\frac{n^2}{{N_{\text{Ch}}}^2} \right) \\ &= O\left(\frac{n^2}{N_{\text{Ch}}} \right). \end{aligned} \tag{6.13}$$

Since

$$(N_{\text{Ch}} - 2)\, O\left(\frac{n^2}{{N_{\text{Ch}}}^2} \right) < N_{\text{Ch}} O\left(\frac{n^2}{{N_{\text{Ch}}}^2} \right).$$

In the worst case, the line that is executed the most number of times in CorrectNeighbourNotes is line 6. Each execution of line 6 takes $O(1)$ time and, in the worst case, the number of times this line is executed is given by

$$\sum_{i=0}^{N_{\text{Ch}}-3} \left(\sum_{j=0}^{|\textbf{OCMChordList}[i]|-1} (|\textbf{OCMChordList}[i+1]| + 1) \right).$$

Therefore, the total amount of time spent on executing line 6 during one run of CorrectNeighbourNotes is

$$T_6 = \sum_{i=0}^{N_{\text{Ch}}-3} \left(\sum_{j=0}^{|\textbf{OCMChordList}[i]|-1} (|\textbf{OCMChordList}[i+1]| + 1) \times O(1) \right). \tag{6.14}$$

If we again make the assumption in Eq. 6.12, then it follows that

$$\begin{aligned} T_6 &= \sum_{i=0}^{N_{\text{Ch}}-3}\left(\sum_{j=0}^{(n/N_{\text{Ch}})-1}\left(\frac{n}{N_{\text{Ch}}}+1\right)\times O(1)\right) \\ &= O\Big((N_{\text{Ch}}-2)\left(\frac{n}{N_{\text{Ch}}}\right)\left(\frac{n}{N_{\text{Ch}}}+1\right)\Big) \\ &= O\Big(N_{\text{Ch}}\left(\frac{n^2}{{N_{\text{Ch}}}^2}+\frac{n}{N_{\text{Ch}}}\right)\Big) \\ &= O\Big(\frac{n^2}{N_{\text{Ch}}}+n\Big). \end{aligned} \tag{6.15}$$

By a similar argument, it can be shown that $O\Big(\frac{n^2}{N_{\text{Ch}}}\Big)$ time is spent in the worst case on executing each of lines 7 to 12 during one complete run of CorrectNeighbourNotes. Therefore, the worst case running time of CorrectNeighbourNotes is $O\Big(\frac{n^2}{N_{\text{Ch}}}+n\Big)$. However, if we assume that the number of notes in any given single chord is never going to be greater than some reasonable constant, C, (which we can set to be a value as high as, say, 100), then we can assume that $n/N_{\text{Ch}} = C$ and the worst-case time complexity of CorrectNeighbourNotes becomes $O(Cn+n) = O(n)$.

The worst-case time complexities of the CorrectDownwardPassingNotes (Figure 6.9) and CorrectUpwardPassingNotes (Figure 6.10) functions called in lines 8 and 9 of PS13 are easier to derive than that of CorrectNeighbourNotes. Each execution of each line in CorrectDownwardPassingNotes takes $O(1)$ time. It can readily be seen that, in the worst case, the line that executes the most number of times in CorrectDownwardPassingNotes is line 12. In the worst case, the total number of times that this line is executed during one run of CorrectDownwardPassingNotes is

$$N_{12} = \sum_{i=0}^{N_{\text{Ch}}-3}\left(\sum_{j=0}^{|\textbf{OCMChordList}[i]|-1}\left(\sum_{k=0}^{|\textbf{OCMChordList}[i+2]|-1}\left(\sum_{\ell=0}^{|\textbf{OCMChordList}[i+1]|-1}(|\textbf{OCMChordList}[i+1]|+1)\right)\right)\right).$$

However, if we make the assumption in Eq. 6.12, then, in the worst case,

$$N_{12} < N_{\text{Ch}} \times \frac{n^3}{{N_{\text{Ch}}}^3} \times \left(\frac{n}{N_{\text{Ch}}}+1\right) = \frac{n^4}{{N_{\text{Ch}}}^3}+\frac{n^3}{{N_{\text{Ch}}}^2}.$$

Furthermore, if we assume, reasonably, that n/N_{Ch} is always less than some appropriately chosen constant, C, then

$$N_{12} < C^3n + C^2n = (C^3+C^2)n.$$

Since the time taken for one execution of line 12 is $O(1)$, the worst case running time of CorrectDownwardPassingNotes is therefore $O(C^3n)$, that is, linear in the number of notes in the input passage, but cubic in the maximum number of notes per chord. This suggests that Stage 2 of PS13 may become impractically slow if the maximum number of notes per chord in a passage is extremely high. It is clear that the worst-case running time of CorrectUpwardPassingNotes is the same as that of CorrectDownwardPassingNotes, that is, $O(C^3n)$, where C is the maximum number of notes per chord.

Each of the functions ComputeNewMorphListFromOCMChordList (Figure 6.11), ComputeMorpheticPitchList (Figure 6.12) and ComputeOPNList (Figure 6.13) clearly has a worst-case running time of $O(n)$. Therefore, the overall worst-case running time of PS13 (and of Stage 2 of PS13 in particular) is $O(C^3n)$, where n is the number of notes in the input passage and C is the maximum number of notes per chord.

It can readily be shown that the worst-case space complexity of PS13 is $O(n)$.

6.4 Results obtained in previous published evaluations of *ps13*

6.4.1 Results reported by Meredith (2003)

In an initial pilot study, I ran an implementation of *ps13* on a test corpus containing 41544 notes and consisting of all 48 movements in the first book of J. S. Bach's *Das Wohltemperirte Clavier* (BWV 846–869) (Meredith, 2003). This implementation was run on this test corpus 2500 times, each time using a different pair of values for the parameters K_{pre} and K_{post}, chosen so that both were between 1 and 50, inclusive (Meredith, 2003, p. 207). The image plot in Figure 6.14 summarises the results obtained. As can be seen in this figure, *ps13* performed best on this test corpus when K_{pre} was set to 33 and K_{post} was set to either 23 or 25. With these settings, *ps13* spelt 99.81% of the notes in this test corpus correctly. On the same test corpus, an implementation of Longuet-Higgins's algorithm spelt 99.36% of the notes correctly, an implementation of Cambouropoulos's algorithm spelt 93.74% of the notes correctly and Temperley's algorithm spelt 99.71% of the notes correctly.

ps13 was therefore the most accurate of the algorithms tested in this pilot study (Meredith, 2003). It was found that *ps13* made fewer errors than Temperley's algorithm on this test corpus for over 80% of the 2500 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ pairs tested. Moreover, the average note accuracy achieved by *ps13* on this test corpus over all 2500 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ pairs was 99.74% which was higher than the note accuracy achieved by any of the other algorithms tested. The worst result achieved by *ps13* in this evaluation was when both K_{pre} and K_{post} were set to 1. With these settings, *ps13* spelt 97.31% of the notes correctly.

Note that all the $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations that led to note accuracies greater than 99.80% on this test corpus lay within the rectangle, marked on Figure 6.14, which contains those combinations for which $25 \leq K_{\text{pre}} \leq 37$ and $20 \leq K_{\text{post}} \leq 28$.

6.4.2 Results reported by Meredith (2005)

In a later, larger-scale study, I ran an implementation of *ps13*, together with implementations of the algorithms of Longuet-Higgins, Temperley and Cambouropoulos, on a large corpus containing 1729886 notes and consisting of 1655 movements from works by 9 baroque and classical composers (Meredith, 2005, pp. 184–190). In this evaluation, *ps13* was run with $K_{\text{pre}} = 33$ and $K_{\text{post}} = 23$, these being the values that produced the best results in the pilot study just discussed (Meredith, 2003). Again, of the four algorithms, *ps13* achieved the highest note accuracy, spelling 99.33% of the notes in this test corpus correctly. The note accuracies achieved by all the algorithms in this evaluation over the complete test corpus and over each subset containing works by a particular composer are given in Table 6.1 and summarised graphically in Figure 6.15. This

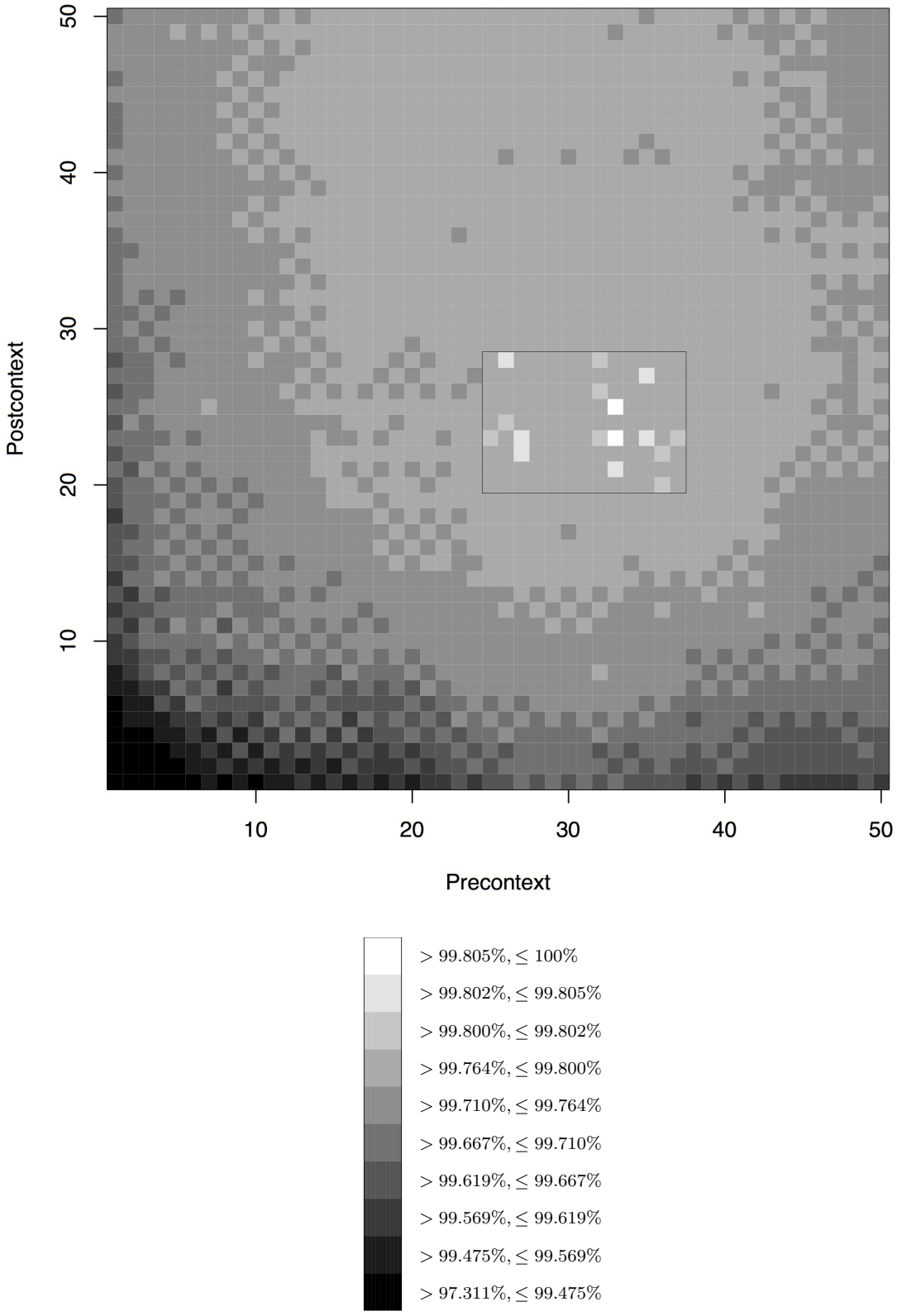


Figure 6.14: Image plot showing percentage of notes spelt correctly by *ps13* for all values of K_{pre} and K_{post} between 1 and 50, on a test corpus consisting of all movements in the first book J. S. Bach's *Das Wohltemperirte Clavier* (BWV 846–869). (Reproduced from Meredith, 2005, p. 183).

	Cam	LH	ps13	Tem
Corelli	99.53%	99.62%	99.90%	99.98%
Vivaldi	99.19%	97.53%	99.33%	97.81%
Telemann	99.33%	99.26%	99.46%	99.88%
Bach	97.87%	97.92%	99.45%	99.81%
Handel	99.57%	99.26%	99.48%	99.71%
Marcello	99.97%	99.86%	99.53%	99.83%
Haydn	98.23%	92.71%	99.03%	92.45%
Mozart	98.66%	94.10%	98.69%	88.48%
Beethoven	98.77%	96.50%	98.51%	86.59%
Complete test corpus	98.71%	97.65%	99.33%	97.67%

Table 6.1: Note accuracies achieved by algorithms in evaluation described by Meredith (2005, pp. 184–189). (Reproduced from Meredith, 2005, p. 185.)

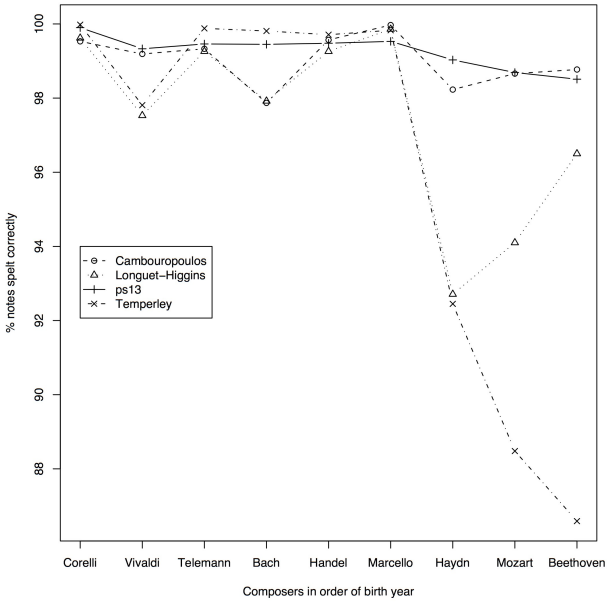


Figure 6.15: Graph showing the percentage of notes spelt correctly by each algorithm for each composer in the evaluation reported by Meredith (2005), with the composers arranged along the horizontal axis in increasing chronological order of birth. (Reproduced from Meredith, 2005, p. 187.)

Composer	*Number of notes*	*% of notes*	*Cum. %*
Corelli	31390	1.81%	1.81%
Vivaldi	223678	12.93%	14.74%
Telemann	89542	5.18%	19.92%
Bach	627083	36.25%	56.17%
Handel	449793	26.00%	82.17%
Marcello	2962	0.17%	82.34%
Haydn	84682	4.90%	87.24%
Mozart	172097	9.95%	97.19%
Beethoven	48659	2.81%	100.00%
Total	1729886		

Table 6.2: Number and percentage of notes in works by each composer in corpus used by Meredith (2005). (Reproduced from Meredith, 2005, p. 184.)

	Cam	LH	ps13	Tem
Corelli	99.08%	99.71%	99.81%	99.96%
Vivaldi	99.08%	99.43%	99.13%	99.58%
Telemann	99.19%	99.50%	99.12%	99.77%
Bach	97.97%	99.07%	99.28%	99.67%
Handel	99.44%	99.62%	99.55%	99.82%
Marcello	99.93%	99.73%	99.12%	99.73%
Haydn	97.93%	97.99%	98.47%	98.13%
Mozart	98.49%	98.41%	98.28%	98.33%
Beethoven	98.49%	98.41%	98.57%	98.08%
Complete test corpus	98.65%	99.16%	99.17%	99.45%

Table 6.3: Percentage of intervals between consecutive notes in the input representations spelt correctly by each algorithm for each composer in the evaluation reported by Meredith (2005). (Reproduced from Meredith, 2005, p. 189.)

graph suggests that the algorithms of Temperley and Longuet-Higgins performed significantly worse on the classical composers (Haydn, Mozart and Beethoven) in this evaluation than they did on the baroque composers (Meredith, 2005, p. 186). However, the note accuracies for the different composers in this evaluation are not really comparable, since the number of notes in this test corpus for each composer varied from 2962 notes from works by B. Marcello to 627083 notes from works by J. S. Bach (see Table 6.2).

When I examined the errors made by the algorithms in this larger-scale study, I observed that, in some cases, many errors were the result of large segments of the music simply being transposed up or down by a diminished second. In other words, a single incorrect interval between two consecutive notes in the input representation sometimes resulted in a whole segment of notes following the incorrect interval being spelt incorrectly. The algorithms were therefore also compared with respect to the number of *intervals* spelt correctly between consecutive notes in the input representations and the results are shown in Table 6.3 (Meredith, 2005, pp. 186–189). These results are also summarised graphically in Figure 6.16. Comparing this graph with that in Figure 6.15 reveals that most of the note spelling errors made by the algorithms of Longuet-Higgins and Temperley on the music of Haydn, Mozart and Beethoven were due to whole segments being spelt a diminished second away from the correct spelling (Meredith, 2005, p. 187). Indeed, as can be seen in Table 6.3, Temperley's algorithm actually performed better

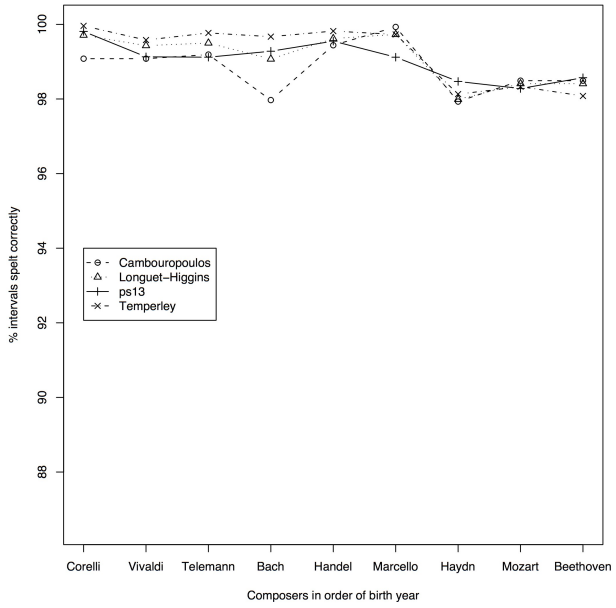


Figure 6.16: Graph showing the percentage of intervals between consecutive notes spelt correctly by each algorithm for each composer in the evaluation reported by Meredith (2005), with the composers arranged along the horizontal axis in increasing chronological order of birth. (Reproduced from Meredith, 2005, p. 188.)

than *ps13* in this evaluation in terms of the number of intervals between consecutive notes in the input representation spelt correctly.

6.4.3 Results reported by Meredith and Wiggins (2005)

Meredith and Wiggins (2005) summarise some of the results discussed in Chapters 2–5 above. Meredith and Wiggins (2005) also give note accuracy and style dependence values for two versions of the *ps13* algorithm:

1. an implementation of *ps13* which I shall call $\text{PS13}_{\text{ISMIR}}$; and
2. an implementation of an algorithm called *ps1303*.

The $\text{PS13}_{\text{ISMIR}}$ implementation of *ps13* is almost identical to the PS13 implementation described in section 6.2 above. However, there is one slight difference. Recall that, in line 24 of the ComputeMorphList function in Figure 6.4, if $\textbf{MorphStrength}[k] = \text{Max}(\textbf{MorphStrength})$ for two or more values of k, then $\text{Pos}(\text{Max}(\textbf{MorphStrength}), \textbf{MorphStrength})$ arbitrarily returns the least value of k for which $\textbf{MorphStrength}[k] = \text{Max}(\textbf{MorphStrength})$. This means that, in PS13, the morph assigned to a note is the *least* of the values that are implied most strongly for that note. However, in $\text{PS13}_{\text{ISMIR}}$, if the set of most strongly implied morphs for a given note is $\{m'_1, m'_2, \ldots m'_i, \ldots\}$, then the morph that is assigned is the one for which $(m'_i - m_0) \bmod 7$ is a minimum, where m_0 is the morph assigned to the first note (i.e., the note represented by $\textbf{SortedOCPList}[0]$). PS13 can therefore be made to generate exactly the same output as $\text{PS13}_{\text{ISMIR}}$ by replacing the call to ComputeMorphList in line 4 of PS13 (see Figure 6.1) with the line

4 **MorphList** ← ComputeMorphList$_{\text{ISMIR}}$(**ChromaList**, **ChromaVectorList**, n)

where the function ComputeMorphList$_{\text{ISMIR}}$ is as defined in Figure 6.17.

Lines 1–23 in ComputeMorphList$_{\text{ISMIR}}$ are identical to lines 1–23 in the ComputeMorphList function defined in Figure 6.4. However, line 24 in ComputeMorphList is replaced with lines 24–29 in ComputeMorphList$_{\text{ISMIR}}$. The purpose of lines 24–26 of ComputeMorphList$_{\text{ISMIR}}$ is to construct a list, **M**, of triples, in which each triple, $\langle m, i, s\rangle$, gives a morph, m, the morph interval, $i = (m - m_0) \bmod 7$, and the strength, s, with which the morph, m, is implied for **SortedOCPList**[j]. Then, in line 27 of ComputeMorphList$_{\text{ISMIR}}$, this list of triples is sorted by the second element of each triple into ascending order. In line 28, the list **S** is constructed so that **S**[i] gives the strength with which the morph **M**[i][0] is implied for **SortedOCPList**[j]. Then, in line 29, this list **S** is used to assign the morph to **SortedOCPList**[j] instead of the list **MorphStrength** used in line 24 of ComputeMorphList.

With $K_{\text{pre}} = 33$ and $K_{\text{post}} = 23$, PS13$_{\text{ISMIR}}$ spelt 1355 of the 195972 notes in the test corpus $\mathcal{C}$ defined in Table 1.4 incorrectly, achieving a note accuracy of 99.31% and a style dependence of 0.56. Note that, with the same values for the parameters K_{pre} and K_{post}, PS13$_{\text{ISMIR}}$ actually made 14 fewer errors than PS13 on the test corpus $\mathcal{C}$ (see Table 6.5). However, this difference was due simply to the different ways in which the algorithms arbitrarily choose between the most strongly implied morphs for each note.

ps1303 first predicts the pitch names using *ps13*. Then, for each note, it determines whether the pitch name predicted by *ps13* is relatively distant from the pitch names in its context along the line of fifths. If the pitch name assigned to a note is relatively distant from its neighbouring notes along the line of fifths, and it can be made closer to these neighbours by transposing it either up or down a diminished second, then it is transposed by the appropriate interval. This algorithm will be discussed in more detail in section 6.6.2 below. The implementation of *ps1303* used in the experiment reported by Meredith and Wiggins (2005) used the PS13$_{\text{ISMIR}}$ implementation rather than PS13. With $K_{\text{pre}} = 33$ and $K_{\text{post}} = 23$, this implementation of *ps1303* spelt 99.43% of the notes in the test corpus $\mathcal{C}$ correctly with a style dependence of 0.54.

6.5 Results of running PS13 on the test corpus $\mathcal{C}$

As discussed in section 6.4.1 above, it was found that *ps13* performed best on the first book of J. S. Bach's *Das Wohltemperirte Clavier* (BWV 846–869) when K_{pre} was set to 33 and K_{post} was set to either 23 or 25. However, these values of K_{pre} and K_{post} may only be optimal for processing music specifically in the style of Bach's keyboard music—other values of K_{pre} and K_{post} may give better results on a stylistically more varied test corpus such as the test corpus $\mathcal{C}$ used in this study.

To test this, PS13 was run 2500 times on a small subset of the test corpus $\mathcal{C}$, each time using a different pair of values for the parameters K_{pre} and K_{post}, chosen so that both were between 1 and 50, inclusive. The subset of $\mathcal{C}$ used contained exactly 24000 notes, consisting of exactly

```
ComputeMorphListISMIR(ChromaList, ChromaVectorList, n)
1   MorphList ← ⨁_{j=1}^{n} ⟨nil⟩
    ▶ First compute m_0.
2   InitMorph ← ⟨0, 1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 6⟩
3   c_0 ← ChromaList[0]
4   m_0 ← InitMorph[c_0]
5   MorphInt ← ⟨0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6⟩
    ▶ Compute Eq. 6.8 for 0 ≤ c_t ≤ 11.
6   TonicMorphForTonicChroma ← ⟨nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil⟩
7   for c_t ← 0 to 11
8       TonicMorphForTonicChroma[c_t] ← (m_0 − MorphInt[(c_0 − c_t) mod 12]) mod 7
9   MorphForTonicChroma ← ⟨nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil⟩
10  TonicChromaSetForMorph ← ⟨nil, nil, nil, nil, nil, nil, nil⟩
11  MorphStrength ← ⟨nil, nil, nil, nil, nil, nil, nil⟩
12  for j ← 0 to n − 1
        ▶ Compute Eq. 6.1 for 0 ≤ c_t ≤ 11.
13      for c_t ← 0 to 11
14          c ← ChromaList[j]
15          MorphForTonicChroma[c_t] ← (MorphInt[(c − c_t) mod 12] + TonicMorphForTonicChroma[c_t]) mod 7
        ▶ Compute Eq. 6.3 for 0 ≤ m ≤ 6.
16      for m ← 0 to 6
17          TonicChromaSetForMorph[m] ← nil
18      for m ← 0 to 6
19          for c_t ← 0 to 11
20              if MorphForTonicChroma[c_t] = m
21                  TonicChromaSetForMorph[m] ← ⟨c_t⟩ ⊕ TonicChromaSetForMorph[m]
        ▶ Compute Eq. 6.4 for 0 ≤ m ≤ 6.
22      for m ← 0 to 6
23          MorphStrength[m] ← Σ_{c_t ∈ TonicChromaSetForMorph[m]} ChromaVectorList[j][c_t]
        ▶ Following changes ordering of strengths so that output is the same as that
        ▶ generated by PS13ISMIR.
24      M ← ⟨ ⟩
25      for m ← 0 to 6
26          M ← M ⊕ ⟨⟨m, (m − m_0) mod 7, MorphStrength[m]⟩⟩
27      M ← SortByMorphInt(M)
28      S ← ⨁_{i=0}^{6} ⟨M[i][2]⟩
29      MorphList[j] ← M[Pos(Max(S), S)][0]
30  return MorphList
```

Figure 6.17: The ComputeMorphListISMIR function.

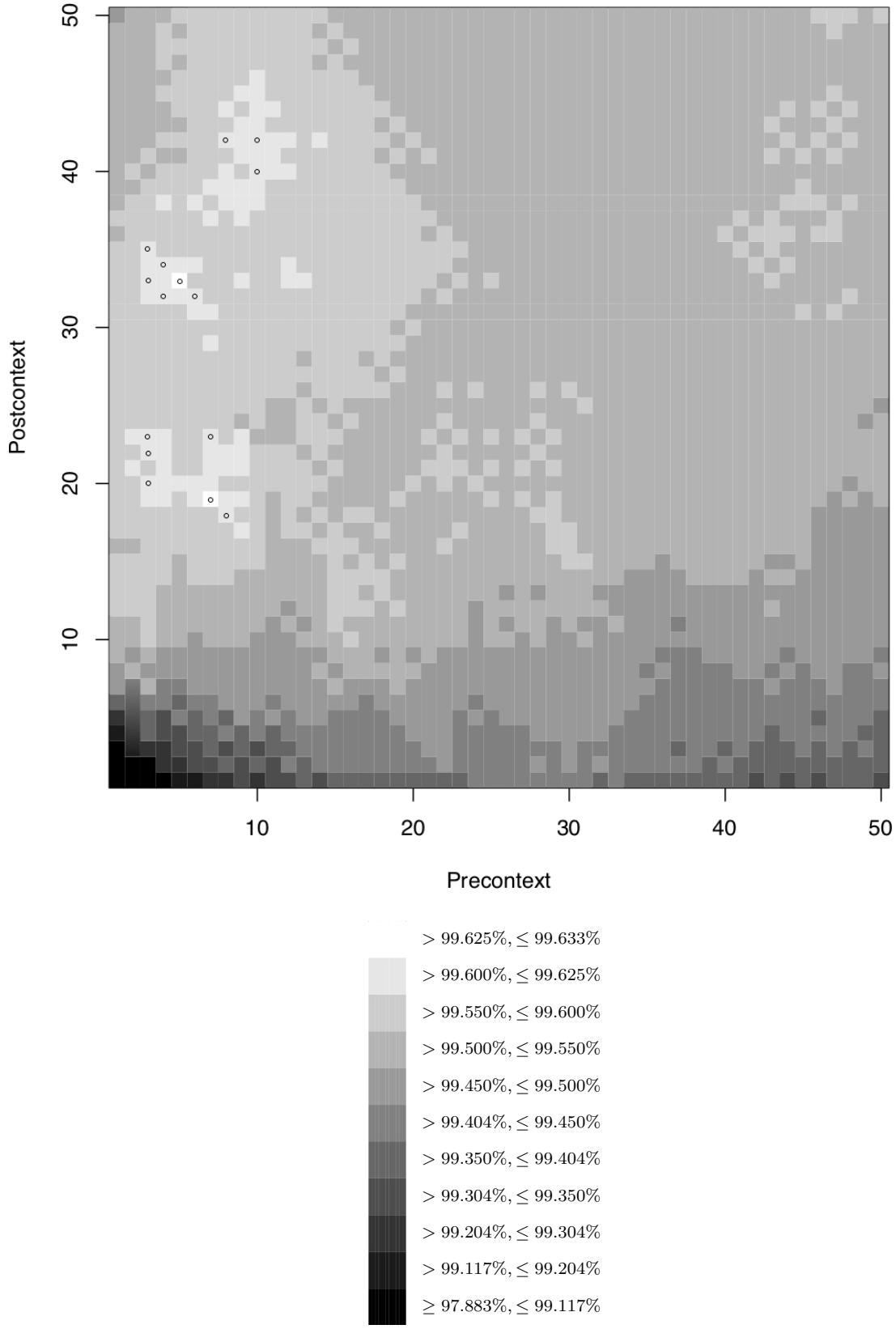


Figure 6.18: Image plot showing percentage of notes spelt correctly by PS13 for all values of K_{pre} and K_{post} between 1 and 50, on a test corpus containing 24000, consisting of 3000 notes from each of the eight composers represented in the test corpus $\mathcal{C}$.

	K_{pre}	K_{post}	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	SD
1	5	33	99.933	98.900	99.933	99.900	99.633	99.200	99.900	99.667	99.633	0.387
2	7	19	99.933	98.933	99.933	99.900	99.467	99.133	99.967	99.800	99.633	0.407
3	3	22	99.933	98.967	99.900	99.900	99.567	99.133	99.900	99.700	99.625	0.379
4	3	20	99.867	98.967	99.900	99.900	99.533	99.133	99.933	99.733	99.621	0.378
5	3	33	99.933	98.900	99.900	99.900	99.667	99.133	99.900	99.633	99.621	0.395
6	4	34	99.933	98.900	99.933	99.867	99.667	99.133	99.900	99.633	99.621	0.395
7	6	32	99.933	98.900	99.933	99.900	99.600	99.133	99.900	99.667	99.621	0.398
8	8	18	99.933	98.833	99.933	99.900	99.267	99.333	99.967	99.800	99.621	0.423
9	4	32	99.933	98.933	99.933	99.867	99.633	99.133	99.900	99.600	99.617	0.386
10	3	35	99.933	98.900	99.900	99.900	99.700	99.067	99.900	99.633	99.617	0.408
11	3	23	99.933	98.800	99.900	99.900	99.567	99.200	99.933	99.700	99.617	0.416
12	10	40	99.933	98.867	99.933	99.900	99.533	99.067	99.933	99.767	99.617	0.427
13	10	42	99.933	98.867	99.967	99.900	99.533	99.067	99.933	99.733	99.617	0.429
14	7	23	99.933	98.767	99.967	99.867	99.533	99.200	99.933	99.733	99.617	0.431
15	8	42	99.933	98.867	99.967	99.933	99.533	99.067	99.933	99.700	99.617	0.431

Table 6.4: Note accuracies achieved by PS13 expressed as percentages over $\mathcal{C}_{\text{Samp}}$ ("Complete") and each subset of $\mathcal{C}_{\text{Samp}}$ containing music by a particular composer. Results are given for the 15 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations that produced the highest note accuracies. The last column gives the style dependence for each of these $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations, measured over $\mathcal{C}_{\text{Samp}}$.

3000 notes for each of the 8 composers represented. I shall denote this 24000 note subset of $\mathcal{C}$ by $\mathcal{C}_{\text{Samp}}$. The results are summarised graphically in the image plot shown in Figure 6.18. The mean and standard deviation of the note accuracies achieved by PS13 on $\mathcal{C}_{\text{Samp}}$ over all 2500 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations tested were 99.52% and 0.076, respectively. The lowest note accuracy, 97.88%, resulted when $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ was set to $\langle 1, 1\rangle$; and the highest note accuracy, 99.63%, was achieved when $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ was set to either $\langle 7, 19\rangle$ or $\langle 5, 33\rangle$. Table 6.4 shows the note accuracy and style dependence values achieved by PS13 over $\mathcal{C}_{\text{Samp}}$ for the best 15 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations tested.

Figure 6.18 looks quite different from Figure 6.14, implying that the way in which the note accuracy of PS13 depends on the values of K_{pre} and K_{post} when it is run on $\mathcal{C}_{\text{Samp}}$ is quite different from when it is run on the first book of Bach's *Das Wohltemperirte Clavier*. In Figure 6.18, the best 15 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations are marked by small circles. Note that, in Figure 6.18, the best parameter value combinations seem to be spread between three separated clusters; whereas, in Figure 6.14, the $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations giving the best results were concentrated in one fairly compact region. Note also that PS13 performed best on $\mathcal{C}_{\text{Samp}}$ when K_{pre} was less than 10; whereas a value between 25 and 37 worked best when the algorithm was run on the first book of *Das Wohltemperirte Clavier*. The algorithm worked best on $\mathcal{C}_{\text{Samp}}$ when K_{post} was between about 18 and 42; and it worked best on the first book of Bach's *Das Wohltemperirte Clavier* when K_{post} was between 20 and 28. This seems to suggest that, in Bach's *Das Wohltemperirte Clavier*, the pitch name of a note typically depends on a larger context than is usually required to determine the pitch name of a note in a passage of 18th century music.

PS13 was then run 17 times on the full test corpus, $\mathcal{C}$, each time using a different combination of values for the parameters K_{pre} and K_{post}, chosen from the 15 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations that resulted in the highest note accuracy when the algorithm was run on $\mathcal{C}_{\text{Samp}}$, together with the two combinations that resulted in the highest note accuracy when the algorithm was run on

Rank	K_{pre}	K_{post}	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
1	10	42	34	423	17	66	274	231	113	194	1352
2	33	25	39	407	22	68	281	238	108	200	1363
3	8	42	36	425	17	62	276	237	116	199	1368
4	33	23	37	410	20	73	277	239	116	197	1369
5	10	40	35	433	19	65	275	235	118	189	1369
6	6	32	40	429	21	69	279	233	122	222	1415
7	5	33	43	431	21	72	281	230	120	222	1420
8	4	32	40	446	21	75	271	229	121	232	1435
9	7	23	32	430	21	75	270	241	126	243	1438
10	4	34	49	438	21	79	283	230	118	225	1443
11	8	18	35	425	19	76	298	232	127	232	1444
12	7	19	33	420	19	76	291	242	129	236	1446
13	3	33	44	441	22	78	282	228	124	227	1446
14	3	35	47	447	22	80	278	234	118	231	1457
15	3	23	45	436	24	78	282	239	142	255	1501
16	3	22	45	429	25	71	279	247	141	270	1507
17	3	20	48	436	23	75	285	239	142	275	1523

Table 6.5: Note error counts obtained for PS13 for 17 different $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations over the test corpus $\mathcal{C}$ and each subset of this corpus containing the movements by a particular composer. The results are sorted in ascending order by the overall note error count (given in the final column). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus.

the first book of Bach's *Das Wohltemperirte Clavier*. Tables 6.5 and 6.6 show the note error counts and note accuracies, respectively, achieved by PS13 for each of these 17 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations over $\mathcal{C}$ and each of the 8 subsets of this test corpus containing works by a particular composer. The last column of Table 6.6 gives the style dependence for each parameter value combination.

Note that the $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations $\langle 33, 25\rangle$ and $\langle 33, 23\rangle$ were, respectively, the second and fourth best-performing over the complete test corpus $\mathcal{C}$ of the 17 combinations tested, despite the fact that they did not perform particularly well over $\mathcal{C}_{\text{Samp}}$: $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 33, 25\rangle$ achieved a note accuracy of 99.54% over $\mathcal{C}_{\text{Samp}}$ and 888 of the 2500 combinations tested performed better than it; $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 33, 23\rangle$ spelt 99.53% of the notes in $\mathcal{C}_{\text{Samp}}$ correctly and 1155 of the 2500 combinations tested performed better than it. Also, note that, out of the 17 combinations tested over $\mathcal{C}$,

1. the combinations for which $K_{\text{pre}} + K_{\text{post}} \geq 50$ performed best (see fourth column in Table 6.6);
2. the combinations with the smallest K_{pre} (3) performed worst;
3. there seems to be a general trend for the overall note accuracy to fall as the size of the context becomes smaller.

These results suggest that $\mathcal{C}_{\text{Samp}}$ might not have been a particularly representative sample of $\mathcal{C}$. Nevertheless, there is no specific evidence to suggest that the note accuracy of PS13 can be raised to much higher than 99.31% over $\mathcal{C}$ by adjusting the values of K_{pre} and K_{post}. It can

Rank	K_{pre}	K_{post}	$K_{\text{pre}} + K_{\text{post}}$	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	SD_{Sty}
1	10	42	52	99.86	98.27	99.93	99.73	98.88	99.06	99.54	99.21	99.31	0.57
2	33	25	58	99.84	98.34	99.91	99.72	98.85	99.03	99.56	99.18	99.30	0.55
3	8	42	50	99.85	98.26	99.93	99.75	98.87	99.03	99.53	99.19	99.30	0.57
4	33	23	56	99.85	98.33	99.92	99.70	98.87	99.02	99.53	99.20	99.30	0.55
5	10	40	50	99.86	98.23	99.92	99.73	98.88	99.04	99.52	99.23	99.30	0.58
6	6	32	38	99.84	98.25	99.91	99.72	98.86	99.05	99.50	99.09	99.28	0.57
7	5	33	38	99.82	98.24	99.91	99.71	98.85	99.06	99.51	99.09	99.28	0.57
8	4	32	36	99.84	98.18	99.91	99.69	98.89	99.07	99.51	99.05	99.27	0.58
9	7	23	30	99.87	98.24	99.91	99.69	98.90	99.02	99.49	99.01	99.27	0.58
10	4	34	38	99.80	98.21	99.91	99.68	98.84	99.06	99.52	99.08	99.26	0.57
11	8	18	26	99.86	98.26	99.92	99.69	98.78	99.05	99.48	99.05	99.26	0.58
12	7	19	26	99.87	98.29	99.92	99.69	98.81	99.01	99.47	99.04	99.26	0.57
13	3	33	36	99.82	98.20	99.91	99.68	98.85	99.07	99.49	99.07	99.26	0.58
14	3	35	38	99.81	98.17	99.91	99.67	98.86	99.04	99.52	99.06	99.26	0.58
15	3	23	26	99.82	98.22	99.90	99.68	98.85	99.02	99.42	98.96	99.23	0.58
16	3	22	25	99.82	98.25	99.90	99.71	98.86	98.99	99.42	98.90	99.23	0.58
17	3	20	23	99.80	98.22	99.91	99.69	98.84	99.02	99.42	98.88	99.22	0.58

Table 6.6: Note accuracies achieved by PS13 for 17 different $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ combinations over $\mathcal{C}$ and each of the 8 subsets of this test corpus containing the music by a particular composer. The results are sorted into descending order by note accuracy over $\mathcal{C}$ (column headed "Complete") and then in ascending order by style dependence (column headed "SD_{Sty}").

therefore be reported that, in this evaluation, when $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was set to $\langle 10, 42 \rangle$, PS13 spelt 99.31% of the notes in $\mathcal{C}$ correctly (corresponding to 1352 incorrectly spelt notes out of 195972) with a style dependence of 0.57.

6.6 Improving PS13

6.6.1 Choosing between the most strongly implied morphs for a note

As discussed near the end of section 6.2.1 above in relation to the ComputeMorphList function, the way in which PS13 chooses between the most strongly implied morphs for a note is arbitrary. Then, in section 6.4.3, it was pointed out that $\text{PS13}_{\text{ISMIR}}$ made 14 fewer errors on $\mathcal{C}$ than PS13 with $\langle K_{\text{pre}}, K_{\text{post}} \rangle = \langle 33, 23 \rangle$, even though the only difference between the two algorithms is the way that they choose between the most strongly implied morphs for a note. This suggests that it might be possible to improve the performance of PS13 by modifying it so that the choice between the most strongly implied morphs for each note is made in a more principled way.

When PS13 was run on $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to $\langle 10, 42 \rangle$ (i.e., the parameter value combination that led to the best result in the evaluation described in section 6.5 above), there were 161 notes for which more than one morph was most strongly implied. In each of these 161 cases, two consecutive morphs (mod 7) were most strongly implied. These 161 cases were distributed over 46 movements, including several movements by each of the 8 composers represented in $\mathcal{C}$. In 53 of these 161 cases, the note was spelt incorrectly by PS13. If all 53 of these errors could be corrected by choosing between the most strongly implied morphs for a note in a principled way, then the overall note accuracy for PS13 over $\mathcal{C}$ could be raised to 99.34% (i.e., 1299 errors

out of 195972 notes).

It was noticed that, in some of the 161 cases in which two morphs were most strongly implied for a note, PS13 chose an incorrect pitch name that was relatively distant from the other pitch names around it on the line of fifths. This suggested that it might be possible to choose correctly between two most strongly implied morphs by selecting the one that results in the pitch name that is closer on average on the line of fifths to the pitch names of the notes in the vicinity of the note to be spelt. However, in practice, it was found that in 53 of the 161 cases where two morphs were most strongly implied for a note, the correct pitch name was *not* the one that was closest on average on the line of fifths to the pitch names of the notes around it. Moreover, of the 53 of these 161 cases that were mis-spelt by PS13, only 31 could be corrected by choosing the pitch name that was closest on the line of fifths to the other notes around it. This suggests that it is not, in general, a good idea to choose between the two most strongly implied morphs for a note by selecting the one that results in the pitch name that is closer on the line of fifths to the pitch names of the other notes in the vicinity of the note to be spelt.

Another possible way of deciding between two or more most (and equally) strongly implied morphs for a note is based on the idea that two notes influence each other less the further apart they are in time. As discussed in section 6.2 above, the ultimate purpose of Stage 1 (i.e., lines 1–4) of PS13 is to assign to each element of **SortedOCPList** the morph which is most strongly implied by the context around that element. Let t_j denote the onset time of **SortedOCPList**$[j]$ and let c_j denote **ChromaList**$[j]$ where **ChromaList** is the variable whose value is computed in line 2 of PS13 (see Figure 6.1). As already discussed, the task accomplished in lines 1–4 of PS13 is to construct a list of morphs,

$$\textbf{MorphList} = \langle m_0, m_1, \ldots m_{n-1} \rangle \,,$$

such that m_j is the morph that is most strongly implied by the context around **SortedOCPList**$[j]$, consisting of the notes represented by the ordered set of elements,

$$\textbf{Context}_j = \textbf{SortedOCPList}[\textsc{Max}(\{0, j - K_{\text{pre}}\}), \textsc{Min}(\{n, j + K_{\text{post}}\})]. \tag{6.16}$$

In PS13, this is done using the ComputeMorphList function, defined in Figure 6.4, which implements the strategy summarised in Eqs. 6.1–6.8. Let $\mathbf{M}_j$ denote the ordered set of distinct values of m (sorted into ascending order) for which $S(m, j) = \textsc{Max}(\langle S(0, j), S(1, j), \ldots S(6, j) \rangle)$ (see Eq. 6.4). In general, $\mathbf{M}_j$ may contain more than one value and our problem is to decide which of these values should be assigned to m_j.

In PS13, it is assumed that the strength with which a given tonic chroma, c_{t}, implies a particular morph, $m(c_{\text{t}}, j)$, (see Eq. 6.1), for a chroma, c_j, is equal to the frequency with which c_{t} occurs within **Context**$_j$, this frequency being stored in **ChromaVectorList**$[j][c_{\text{t}}]$, where **ChromaVectorList** is the list of 12-vectors computed in line 3 of PS13. Let's suppose that the strength with which a particular occurrence of c_{t} within **Context**$_j$ implies the morph, $m(c_{\text{t}}, j)$, for c_j decreases the further away this occurrence of c_{t} is from c_j. This suggests that, if we have two tonic chromas, c_{t}, that occur with equal frequency within **Context**$_j$, then the one

(a) Note error counts

Algorithm	$\langle K_{\text{pre}}, K_{\text{post}}\rangle$	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
PS13B	$\langle 10, 42\rangle$	32	426	17	64	280	240	112	206	1377
PS13	$\langle 10, 42\rangle$	34	423	17	66	274	231	113	194	1352

(b) Note accuracies (%)

Algorithm	$\langle K_{\text{pre}}, K_{\text{post}}\rangle$	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	SD_{Sty}
PS13B	$\langle 10, 42\rangle$	99.87	98.26	99.93	99.74	98.86	99.02	99.54	99.16	99.30	0.58
PS13	$\langle 10, 42\rangle$	99.86	98.27	99.93	99.73	98.88	99.06	99.54	99.21	99.31	0.57

Table 6.7: Tables (a) and (b) show the note error counts and note accuracies, respectively, for the PS13 and PS13B algorithms when they were run on the test corpus $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$. The number in parentheses underneath each column heading in table (a) gives the number of notes in that subset of the test corpus.

that influences the spelling of c_j the most will be the one for which

$$T(c_{\text{t}}, j) = \sum_{k \in K(c_{\text{t}}, j)} \textsc{Abs}(t_k - t_j) \tag{6.17}$$

is a minimum, where

$$K(c_{\text{t}}, j) = \{k \mid c_k = c_{\text{t}} \wedge \textsc{Max}(\{0, j - K_{\text{pre}}\}) \leq k < \textsc{Min}(\{n, j + K_{\text{post}}\})\}\,. \tag{6.18}$$

In turn, this suggests that the best value, $m \in \mathbf{M}_j$, to assign to m_j may be the one for which

$$\mathcal{T}(m, j) = \sum_{c_{\text{t}} \in C_{\text{t}}(m,j)} T(c_{\text{t}}, j) \tag{6.19}$$

is a minimum. This strategy was implemented in a modified version of PS13, which I call PS13B. PS13B was run on the test corpus, $\mathcal{C}$, with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$ and the results are shown in Table 6.7. As can be seen in Table 6.7, PS13B actually made 25 more errors over $\mathcal{C}$ than PS13. Moreover, of the 161 notes in $\mathcal{C}$ for which more than one morph was most strongly implied, PS13B spelt 69 incorrectly, whereas PS13 only spelt 53 incorrectly. This suggests that the strategy just described for deciding between the most strongly implied morphs for a note does not, in general, lead to a higher note accuracy than when the choice between the most strongly implied morphs for a note is made arbitrarily.

Although only two plausible strategies were tested for deciding on a principled basis between the most strongly implied morphs for a note, the foregoing discussion does suggest that it may be hard to do this in a way that results in a higher note accuracy than that achieved when the selection from the most strongly implied morphs for a note is made arbitrarily, as it is in PS13.

6.6.2 PS1303

In the previous section, it was shown that choosing a pitch name on the basis of how close it is on the line of fifths to the pitch names of the notes around it was not effective as a way of deciding in Stage 1 of PS13 between the most strongly implied morphs for a note.

However, in section 4.7.3 above, it was shown that a simple implementation of Temperley's TPR 1 (which simply states that notes that are "nearby" in the music should be assigned pitch names that are "close together" on the line of fifths (Temperley, 2001, p. 125)) was capable of spelling more than 99% of the notes in $\mathcal{C}$ correctly. Moreover, an analysis of the 1352 errors made by PS13 when it was run on $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$ showed that, for 620 of these 1352 notes, the correct pitch name was the one that was closest on the line of fifths to the pitch names of the notes in its context. This suggested the possibility of improving the note accuracy of PS13 by adding a post-processing phase in which the pitch names computed by PS13 are transposed up or down by a diminished second if this brings them closer on the line of fifths to the pitch names of the notes around them in the music. This idea was implemented in the PS1303 algorithm, defined in Figure 6.19.

PS1303 takes the same three parameters as PS13 (see section 6.2). In line 1 of PS1303, PS13 is used to compute an ordered set, **SortedOPNList**, in which each element is an ordered pair, $\langle t, p\rangle$, giving the onset time, t, and the predicted pitch name, p, of a note. The elements of **SortedOPNList** are sorted in ascending order by onset time and chromatic pitch, with preference given to onset time. In line 2 of PS1303, the total number of notes is stored for convenience in the variable, n. Again for convenience, the onset times of the notes are stored in the ordered set, **OnsetList**, in line 3. In line 4 of PS1303, the line-of-fifths positions of the pitch names in **SortedOPNList** are stored in the ordered set variable, **LOFList**, so that **LOFList**$[j]$ is the line-of-fifths position of the pitch name, **SortedOPNList**$[j][1]$, for all $0 \leq j < n$. This is accomplished using the functions PNC2LOF and PN2PNC, defined in Figures 1.15 and 1.14, respectively. In line 5 of PS1303, the variable, **NewLOFList**, is initialized. This variable is used to store the new line-of-fifths positions of the notes after some of them have been transposed to bring them closer on the line of fifths to the other notes around them in the music.

The 'for' loop in lines 6–25 of PS1303 iterates once for each note in the input passage, assigning a new line-of-fifths position to the note. If $\ell(p)$ is the line-of-fifths position of a pitch name, p, then $\ell(p) + 12$ is the line-of-fifths position of the pitch name that results when p is transposed by a falling diminished second; and $\ell(p) - 12$ is the line-of-fifths position of the pitch name that results when p is transposed by a rising diminished second. Let p be the pitch name assigned by PS13 to the note, n, being processed on a given iteration of the 'for' loop in lines 6–25 of PS1303. The new line-of-fifths position assigned to n will be either $\ell(p)$, $\ell(p) + 12$ or $\ell(p) - 12$, depending on which of the three leads to the pitch name of n being closest on the line of fifths to the notes in the context surrounding n.

In line 7 of PS1303, the variable, *LOF*, is set to equal the line-of-fifths position of the current note to be processed on this iteration of the 'for' loop—that is, *LOF* is set to equal **LOFList**$[j]$. Then, in lines 8 and 9, the variables, *LOFRD2* and *LOFFD2*, are set to equal the line-of-fifths positions of the pitch names that result when the pitch name assigned to the current note by PS13 is transposed by a rising and falling diminished second, respectively.

Next, in lines 10–13, the context for the current note is computed. The size of the context used in PS1303 is determined by the parameters, K_{pre} and K_{post}, in the same way as in PS13. For the untransposed line-of-fifths position, *LOF*, the context, stored in the variable, **Context**, in line 12, is set to equal **LOFList**$[$*ContextStart*, *ContextEnd*$]$, where *ContextStart* $= \textsc{Max}(\{0, j - K_{\text{pre}}\})$

PS1303($\mathbf{SortedOCPList}, K_{\text{pre}}, K_{\text{post}}$)
1 $\mathbf{SortedOPNList} \leftarrow$ PS13($\mathbf{SortedOCPList}, K_{\text{pre}}, K_{\text{post}}$)
2 $n \leftarrow |\mathbf{SortedOCPList}|$
3 $\mathbf{OnsetList} \leftarrow \bigoplus_{j=0}^{n-1} \langle \mathbf{SortedOPNList}[j][0] \rangle$
4 $\mathbf{LOFList} \leftarrow \bigoplus_{j=0}^{n-1} \langle \text{PNC2LOF}(\text{PN2PNC}(\mathbf{SortedOPNList}[j][1])) \rangle$
5 $\mathbf{NewLOFList} \leftarrow \bigoplus_{j=0}^{n-1} \langle \mathsf{nil} \rangle$
6 **for** $j \leftarrow 0$ **to** $n-1$
7 $\quad LOF \leftarrow \mathbf{LOFList}[j]$
8 $\quad LOFRD2 \leftarrow LOF - 12$
9 $\quad LOFFD2 \leftarrow LOF + 12$
10 $\quad ContextStart \leftarrow \text{MAX}(\{0, j - K_{\text{pre}}\})$
11 $\quad ContextEnd \leftarrow \text{MIN}(\{n, j + K_{\text{post}}\})$
12 $\quad \mathbf{Context} \leftarrow \mathbf{LOFList}[ContextStart, ContextEnd]$
13 $\quad \mathbf{ContextForRD2AndFD2} \leftarrow \mathbf{LOFList}[ContextStart, j] \oplus \langle 0 \rangle \oplus \mathbf{LOFList}[j+1, ContextEnd]$
14 $\quad \mathbf{OnsetContext} \leftarrow \mathbf{OnsetList}[ContextStart, ContextEnd]$
15 $\quad s \leftarrow |\mathbf{Context}|$
16 $\quad LOFDist \leftarrow \sum_{k=0}^{s-1} \frac{\text{ABS}(\mathbf{Context}[k] - LOF)}{\text{MAX}(\{1, \text{ABS}(\mathbf{OnsetList}[j] - \mathbf{OnsetContext}[k])\})}$
17 $\quad LOFDistRD2 \leftarrow \sum_{k=0}^{s-1} \frac{\text{ABS}(\mathbf{ContextForRD2AndFD2}[k] - LOFRD2)}{\text{MAX}(\{1, \text{ABS}(\mathbf{OnsetList}[j] - \mathbf{OnsetContext}[k])\})}$
18 $\quad LOFDistFD2 \leftarrow \sum_{k=0}^{s-1} \frac{\text{ABS}(\mathbf{ContextForRD2AndFD2}[k] - LOFFD2)}{\text{MAX}(\{1, \text{ABS}(\mathbf{OnsetList}[j] - \mathbf{OnsetContext}[k])\})}$
19 $\quad$ **if** $LOFDistFD2 < LOFDist$
20 $\quad\quad \mathbf{NewLOFList}[j] \leftarrow LOFFD2$
21 $\quad$ **else**
22 $\quad\quad$ **if** $LOFDistRD2 < LOFDist$
23 $\quad\quad\quad \mathbf{NewLOFList}[j] \leftarrow LOFRD2$
24 $\quad\quad$ **else**
25 $\quad\quad\quad \mathbf{NewLOFList}[j] \leftarrow LOF$
26 **return** $\bigoplus_{j=0}^{n-1} \langle \langle \mathbf{OnsetList}[j], \text{LOFCP2PN}(\mathbf{NewLOFList}[j], \mathbf{SortedOCPList}[j][1]) \rangle \rangle$

Figure 6.19: The PS1303 algorithm.

and $ContextEnd = \text{MIN}(\{n, j + K_{\text{post}}\})$, as assigned in lines 10 and 11.

However, for the transposed line-of-fifths positions, *LOFRD2* and *LOFFD2*, the context, stored in **ContextForRD2AndFD2** in line 13, is defined to be the same as that for *LOF except* that the untransposed line-of-fifths position in the context for the current note is replaced with a zero (i.e., the line-of-fifths position for the pitch name class `"Fn"`). This might seem like a somewhat arbitrary design decision. However, the effect of replacing the line-of-fifths position for the current note with 0 in **ContextForRD2AndFD2** is to slightly bias the algorithm towards spelling the notes so that they are not too extreme on the line of fifths. It was found that doing this resulted in higher note accuracy than when other, perhaps more obvious, strategies were adopted, such as

1. leaving the line-of-fifths position of the current note unchanged in the context;
2. omitting the line-of-fifths position for the current note from the context; or
3. changing the line-of-fifths position for the current note to *LOFRD2* for *LOFRD2* and to *LOFFD2* for *LOFFD2*.

Perhaps the obvious next step would be to choose the line-of-fifths position for the current note that minimises the sum of the absolute line-of-fifths distances between the current note and the context notes. However, in PS1303, it was found that a higher note accuracy could be obtained by weighting the line-of-fifths distance between the current note and each context note by the reciprocal of the time difference between the two note onsets. To facilitate this, the onset times of the context notes are stored for convenience in the variable, **OnsetContext**, in line 14 and the number of notes in the context is stored in the variable, s, in line 15. Then, in lines 16–18, the weighted sum of the absolute line-of-fifths distances between the current note and the context notes is calculated for *LOF*, *LOFRD2* and *LOFFD2* and stored in the variables, *LOFDist*, *LOFDistRD2* and *LOFDistFD2*, respectively. In lines 19–25, the new line-of-fifths position is stored in the appropriate position in **NewLOFList**. Note that, when $LOFDistRD2 = LOFDistFD2$ and both are less than *LOFDist*, then *LOFFD2* is arbitrarily assigned to the note in favour of *LOFRD2*.

Finally, in line 26, a new ordered set of $\langle t, p \rangle$ pairs is returned giving the onset time, t, and the newly assigned pitch name, p, for each note. The pitch name of each note is computed from its chromatic pitch (stored in **SortedOCPList**) and its new line-of-fifths position (stored in **NewLOFList**) using the function LOFCP2PN, defined in Figure 1.18.

PS1303 was run on the test corpus, $\mathcal{C}$, with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to $\langle 10, 42 \rangle$ and the results are shown in Table 6.8, together with those for PS13 for comparison. The results in this table show that adding the post-processing phase in PS1303 to PS13 decreases the overall note error count by over 17% from 1352 for PS13 to 1119 for PS1303. Moreover, note that this modification increased the note accuracy for all of the 8 composers in the test corpus. The results also seem to indicate that PS1303 is less dependent on style than PS13, with PS1303 achieving a style dependence value of 0.53, compared with the value of 0.57 obtained for PS13.

(a) Note error counts

Algorithm	$\langle K_{\text{pre}}, K_{\text{post}} \rangle$	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
PS1303	$\langle 10, 42 \rangle$	21	402	1	61	208	178	80	168	1119
PS13	$\langle 10, 42 \rangle$	34	423	17	66	274	231	113	194	1352

(b) Note accuracies (%)

Algorithm	$\langle K_{\text{pre}}, K_{\text{post}} \rangle$	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	SD_{Sty}
PS1303	$\langle 10, 42 \rangle$	99.91	98.36	100.00	99.75	99.15	99.27	99.67	99.31	99.43	0.53
PS13	$\langle 10, 42 \rangle$	99.86	98.27	99.93	99.73	98.88	99.06	99.54	99.21	99.31	0.57

Table 6.8: Tables (a) and (b) show the note error counts and note accuracies, respectively, for the PS13 and PS1303 algorithms when they were run on the test corpus $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to $\langle 10, 42 \rangle$. The number in parentheses underneath each column heading in table (a) gives the number of notes in that subset of the test corpus.

6.6.3 Removing Stage 2 from *ps13*

As discussed in section 6.2.2 above, Stage 2 of *ps13* is implemented in a rather crude way in PS13. In particular, the CorrectNeighbourNotes, CorrectUpwardPassingNotes and CorrectDownwardPassingNotes functions can only be expected to perform properly if the onset times of the notes are strictly proportional to their notated values—which, of course, they typically are not in MIDI files generated from performances. Also, as discussed in section 6.2.2 in relation to the CorrectNeighbourNotes function, there are certain situations where the implementation of Stage 2 in PS13 will fail to correct passing-note and neighbour-note errors even if all the note onsets *are* strictly proportional to their notated values. Unfortunately, it is hard to see how these problems could be corrected without using the durations of the notes to help with determining which notes occur within the same voice. However, modifying PS13 so that it requires and depends on note durations could make it less robust to the types of temporal deviations that typically occur in MIDI files derived from performances. For example, in MIDI files derived from keyboard performances, the relative durations of sustained notes may differ greatly from their notated values.

Moreover, as explained in section 6.3, the worst-case time complexity of Stage 1 of PS13 is $O(n)$, whereas the worst-case time complexity of Stage 2 of PS13 is $O(C^3 n)$ where C is the maximum number of notes that occur within any single chord in the input passage. This implies that Stage 2 of PS13 may become impractically slow if the maximum number of notes per chord in a passage is extremely high.

The foregoing arguments imply that omitting Stage 2 altogether from PS13 would

1. improve the overall time complexity of the algorithm and make the algorithm faster in absolute terms since less processing is required;

2. make the algorithm more robust to the types of temporal deviations that typically occur in data derived from performances;

3. simplify the algorithm considerably, thereby making it easier to implement.

This suggests that it would be a good idea to omit Stage 2 from PS13, provided that this does not reduce note accuracy.

```
PS13s1(SortedOCPList, K_pre, K_post)
1   n ← |SortedOCPList|
2   ChromaList ← ComputeChromaList(SortedOCPList, n)
3   ChromaVectorList ← ComputeChromaVectorList(ChromaList, K_pre, K_post, n)
4   MorphList ← ComputeMorphList(ChromaList, ChromaVectorList, n)
5   MorpheticPitchList ← ComputeMorpheticPitchList(SortedOCPList, MorphList, n)
6   return ComputeOPNList(SortedOCPList, MorpheticPitchList, n)
```

Figure 6.20: The PS13s1 algorithm.

(a) Note error counts

Algorithm	$\langle K_{\text{pre}}, K_{\text{post}} \rangle$	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
PS13s1	$\langle 10, 42 \rangle$	40	382	3	65	205	144	96	165	1100
PS13s103	$\langle 10, 42 \rangle$	31	407	1	61	219	166	78	162	1125
PS1303	$\langle 10, 42 \rangle$	21	402	1	61	208	178	80	168	1119
PS13	$\langle 10, 42 \rangle$	34	423	17	66	274	231	113	194	1352

(b) Note accuracies (%)

Algorithm	$\langle K_{\text{pre}}, K_{\text{post}} \rangle$	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	SD_{Sty}
PS13s1	$\langle 10, 42 \rangle$	99.84	98.44	99.99	99.73	99.16	99.41	99.61	99.33	99.44	0.49
PS13s103	$\langle 10, 42 \rangle$	99.87	98.34	100.00	99.75	99.11	99.32	99.68	99.34	99.43	0.53
PS1303	$\langle 10, 42 \rangle$	99.91	98.36	100.00	99.75	99.15	99.27	99.67	99.31	99.43	0.53
PS13	$\langle 10, 42 \rangle$	99.86	98.27	99.93	99.73	98.88	99.06	99.54	99.21	99.31	0.57

Table 6.9: Tables (a) and (b) show the note error counts and note accuracies, respectively, for the PS13s1 and PS13s103 algorithms when they were run on the test corpus $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to $\langle 10, 42 \rangle$. The results for PS13 and PS1303 are also given for comparison. The number in parentheses underneath each column heading in table (a) gives the number of notes in that subset of the test corpus.

Rank	K_{pre}	K_{post}	Bach (24505)	Beethoven (24493)	Corelli (24493)	Handel (24500)	Haydn (24490)	Mozart (24494)	Telemann (24500)	Vivaldi (24497)	Complete (195972)
1	10	42	40	382	3	65	205	144	96	165	1100
2	33	25	42	355	7	68	201	156	97	179	1105
3	10	40	41	386	5	64	207	145	100	164	1112
4	8	42	43	386	4	61	207	143	101	171	1116
5	33	23	41	360	5	73	201	159	104	177	1120
6	6	32	48	382	7	69	207	148	111	197	1169
7	5	33	47	391	8	73	213	150	107	191	1180
8	4	34	53	394	9	78	212	151	105	193	1195
9	4	32	48	396	8	75	200	149	111	208	1195
10	7	19	38	379	7	79	214	157	120	211	1205
11	7	23	41	389	7	75	209	155	116	214	1206
12	8	18	42	381	5	81	219	148	119	212	1207
13	3	33	48	398	11	76	213	150	112	199	1207
14	3	35	50	402	11	78	210	153	106	199	1209
15	3	23	51	396	14	77	208	153	130	226	1255
16	3	22	52	391	16	73	202	159	132	241	1266
17	3	20	55	403	11	78	217	156	134	245	1299

Table 6.10: Note error counts obtained for PS13s1 over $\mathcal{C}$ and each subset of $\mathcal{C}$ containing the movements by a particular composer, for the same 17 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations used to test PS13 (see Table 6.5). The results are sorted in ascending order of overall note error count (given in the final column). The number in parentheses underneath each column heading gives the number of notes in that subset of the test corpus.

The lines implementing Stage 2 were therefore deleted from PS13 to produce the PS13s1 algorithm, defined in Figure 6.20. A new version of PS1303 was also produced in which the call to PS13 in the first line (see Figure 6.19) was replaced with a call to PS13s1. I call this version of the algorithm PS13s103. Both PS13s1 and PS13s103 were run on the test corpus, $\mathcal{C}$, with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$ and the results obtained are shown in Table 6.9, together with the results for PS13 and PS1303 for comparison. These results show that omitting Stage 2 from PS13 actually *improved* its performance over the test corpus, $\mathcal{C}$, both in terms of note accuracy and style dependence: with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ equal to $\langle 10, 42\rangle$, PS13s1 made nearly 19% fewer errors than PS13 achieving a note accuracy of 99.44% and a style dependence of 0.49. However, omitting Stage 2 from PS1303, by changing the call to PS13 in the first line of this algorithm to a call to PS13s1, very slightly increased the total note error count.

PS13s1 was then run on the test corpus, $\mathcal{C}$, with each of the 17 values of $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ used to test PS13, shown in Tables 6.5 and 6.6. This was done in order to test whether PS13s1 is more accurate than PS13 in general or only for very specific values of $\langle K_{\text{pre}}, K_{\text{post}}\rangle$. Comparing Tables 6.5 and 6.10 reveals that, for each of the 17 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations tested, PS13s1 made about 15–19% fewer errors than PS13. Similarly, comparing Tables 6.6 and 6.11 reveals that, for each of the 17 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations tested, the style dependence of PS13s1 is 12–18% lower than that of PS13. Note that the ranking of the $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations for PS13s1 is also very similar to that for PS13, again suggesting that removing Stage 2 from PS13 consistently reduces the note error count over a wide range of values for $\langle K_{\text{pre}}, K_{\text{post}}\rangle$.

It may therefore be concluded that PS13s1 is superior to the other versions of the *ps13* algorithm tested here in terms of note accuracy, style dependence, time complexity and ease of

Rank	K_{pre}	K_{post}	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	SD_{Sty}
1	10	42	99.84	98.44	99.99	99.73	99.16	99.41	99.61	99.33	99.44	0.49
2	33	25	99.83	98.55	99.97	99.72	99.18	99.36	99.60	99.27	99.44	0.45
3	10	40	99.83	98.42	99.98	99.74	99.15	99.41	99.59	99.33	99.43	0.49
4	8	42	99.82	98.42	99.98	99.75	99.15	99.42	99.59	99.30	99.43	0.49
5	33	23	99.83	98.53	99.98	99.70	99.18	99.35	99.58	99.28	99.43	0.46
6	6	32	99.80	98.44	99.97	99.72	99.15	99.40	99.55	99.20	99.40	0.48
7	5	33	99.81	98.40	99.97	99.70	99.13	99.39	99.56	99.22	99.40	0.49
8	4	34	99.78	98.39	99.96	99.68	99.13	99.38	99.57	99.21	99.39	0.49
9	4	32	99.80	98.38	99.97	99.69	99.18	99.39	99.55	99.15	99.39	0.50
10	7	19	99.84	98.45	99.97	99.68	99.13	99.36	99.51	99.14	99.39	0.49
11	7	23	99.83	98.41	99.97	99.69	99.15	99.37	99.53	99.13	99.38	0.50
12	8	18	99.83	98.44	99.98	99.67	99.11	99.40	99.51	99.13	99.38	0.49
13	3	33	99.80	98.38	99.96	99.69	99.13	99.39	99.54	99.19	99.38	0.50
14	3	35	99.80	98.36	99.96	99.68	99.14	99.38	99.57	99.19	99.38	0.50
15	3	23	99.79	98.38	99.94	99.69	99.15	99.38	99.47	99.08	99.36	0.50
16	3	22	99.79	98.40	99.93	99.70	99.18	99.35	99.46	99.02	99.35	0.49
17	3	20	99.78	98.35	99.96	99.68	99.11	99.36	99.45	99.00	99.34	0.51

Table 6.11: Note accuracies achieved by PS13s1 over $\mathcal{C}$ and each subset of $\mathcal{C}$ containing the movements by a particular composer, for the same 17 $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ combinations used to test PS13 (see Table 6.6). The results are sorted into descending order by note accuracy over $\mathcal{C}$ (column headed "Complete") and then in ascending order by style dependence (column headed "SD_{Sty}").

implementation.

6.7 Summary and conclusions

Most experts seem to agree that the pitch name of a note in a passage of tonal music is primarily a function of the key at the point where the note occurs and the voice-leading structure of the music in the note's immediate context. However, most authors also seem to agree that the local key at the point where the note occurs is more important than voice-leading considerations when attempting to determine the pitch name of a note.

The *ps13* pitch spelling algorithm explicitly takes into account both key and voice-leading. In Stage 1 of *ps13*, the local sense of key at each point in a passage is represented by the frequency distribution of chromas within a context surrounding that point. This method of representing the sense of key relates closely to the way that key is represented in Krumhansl and Schmuckler's key-finding algorithm (Krumhansl, 1990, pp. 77–110) and contrasts with the way that a key is represented as a single 'average' point on the line of fifths (or spiral array) in the algorithms of Temperley, Longuet-Higgins and Chew and Chen.

In *ps13*, the frequency with which a chroma occurs within the context surrounding a point in the music is used as a measure of the likelihood of that chroma being the chroma of the tonic at that point. It is then assumed that the pitch name implied for a note by a particular tonic is the one that has the same chroma as the note and lies closest to the tonic on the line of fifths. The strength with which a particular pitch name is implied for a note is then taken to be the sum of the frequencies of occurrence within the context surrounding the note of the tonic chromas that imply that pitch name. The size of the context surrounding each note over which the chroma

frequency distribution is calculated is defined by two parameters, K_{pre} and K_{post}, which specify the number of notes preceding and following the note to be spelt that are to be included in the context.

In Stage 2 of *ps13*, voice-leading is taken into account by correcting those instances in the output of Stage 1 where a neighbour note or passing note is erroneously predicted to have the same letter name as either the note preceding it or the note following it in the neighbour-note or passing-note pattern.

Complete pseudocode was presented for an implementation of *ps13*, called PS13. Unfortunately, the implementation of Stage 2 of *ps13* in PS13 is rather crude and cannot be expected to perform well when the data being processed is derived from a performance. Also, because the algorithm does not have access to voicing information and incorporates no mechanism for determining the voice to which each note belongs, there are certain situations in which Stage 2 of PS13 fails to correct the neighbour-note and passing-note errors that it is designed to catch.

The worst case time complexity of Stage 1 of PS13 is $O(n)$. The worst-case time complexity of Stage 2 of PS13 is $O(C^3n)$ where C is the maximum number of notes per chord. Stage 2 may therefore become impractically slow if a passage contains an extremely high number of notes per chord. The overall worst-case time complexity of PS13 is therefore $O(C^3n)$ and its space complexity is $O(n)$.

PS13 was run 2500 times on a representative subset containing 24000 notes of the test corpus $\mathcal{C}$, each time using a different pair of values for the parameters K_{pre} and K_{post}, chosen so that both were between 1 and 50, inclusive. PS13 was then run 17 times on the full test corpus $\mathcal{C}$, each time using a different combination of values for the parameters K_{pre} and K_{post}, chosen from the 15 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations that resulted in the highest note accuracy when PS13 was run on the 24000 note subset of $\mathcal{C}$, together with the two combinations that gave the best results in an earlier pilot study. PS13 performed best on $\mathcal{C}$ when $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ was set to $\langle 10, 42\rangle$. With these settings, 99.31% of the notes were spelt correctly and the style dependence was 0.57.

In Stage 1 of PS13, it is possible for two or more morphs to be equally and most strongly implied for a given note. When this happens, the least morph is arbitrarily assigned to the note. An attempt was made to improve PS13 by modifying it so that the choice between the most strongly implied morphs for a note was made in a more principled way. Two different methods were tried for choosing between the most strongly implied morphs for a note. In the first, the morph was chosen which led to the note having a pitch name that was closest on the line of fifths to the pitch names of the notes around it. In the second method, the morph was chosen which was implied by tonic chromas occurring closest in time to the note to be spelt. Neither method led to an increase in note accuracy. This suggests that there may not be a principled way of selecting from the most strongly implied morphs for a note that results in a higher note accuracy than that which can be achieved by making the selection arbitrarily as in PS13.

An analysis of the 1352 errors made by PS13 when it was run on $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$ showed that, for 620 of these 1352 notes, the correct pitch name was the one that was closest on the line of fifths to the pitch names of the notes in its context. This suggested the possibility of improving the note accuracy of PS13 by adding a post-processing phase in which the pitch names computed by PS13 are transposed up or down by a diminished second if this

brings them closer on the line of fifths to the pitch names of the notes around them in the music. This idea was implemented in the PS1303 algorithm defined in Figure 6.19. When PS1303 was run on the test corpus, $\mathcal{C}$, with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to $\langle 10, 42 \rangle$, it achieved both a higher note accuracy and a lower style dependence than PS13, spelling 99.43% of the notes in the corpus correctly with a style dependence of 0.53.

Finally, I showed that removing Stage 2 from PS13 would have the beneficial effects of

1. improving the time complexity from $O(C^3 n)$ to $O(n)$;
2. making the algorithm more robust to the types of temporal deviations that typically occur in data derived from performances; and
3. simplifying the algorithm, thus making it easier to implement.

A new version of the algorithm was therefore constructed, called PS13s1, consisting of just Stage 1 of PS13. When PS13s1 was run on $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ equal to $\langle 10, 42 \rangle$, it spelt 99.44% of the notes correctly (corresponding to 1100 note errors) and its style dependence was 0.49. When $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was changed to $\langle 33, 25 \rangle$, the note accuracy was again 99.44% (1105 note errors) and the style dependence was reduced to 0.45. When PS13s1 was run on $\mathcal{C}$ with the 17 different values of $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ used to test PS13, it was found that, for each value of $\langle K_{\text{pre}}, K_{\text{post}} \rangle$, PS13s1 made between 15 and 19% fewer errors than PS13. It was also found that, for each value of $\langle K_{\text{pre}}, K_{\text{post}} \rangle$, the style dependence of PS13s1 was between 12 and 18% lower than that of PS13.

PS13s1 was therefore superior to all the other versions of *ps13* tested here in terms of note accuracy, style dependence, time complexity, tolerance to temporal deviation and ease of implementation. Also, the fact that PS13s1 performed better than PS13 would seem to support the idea that the local sense of key is more important than voice-leading considerations when determining pitch names in tonal music.

N.B. Some of the material included in this chapter has been published in the followng article published by Taylor & Francis: Meredith, D. (2006). The ps13 pitch spelling algorithm. Journal of New Music Research, 35:2, 121-159. http://dx.doi.org/10.1080/09298210600834961

Chapter 7

Comparing the best versions of the algorithms

7.1 Baseline algorithms

The results reported above for *ps13* and the algorithms of Longuet-Higgins, Temperley and Sleator, Cambouropoulos and Chew and Chen need to be set in context by considering what can be achieved using various trivial or chance-based *baseline* algorithms. Four baseline algorithms were considered. The first, BlackNotesSharp, spells all black notes (i.e., notes with pitch classes in the set $\{1, 3, 6, 8, 10\}$) as sharps and all white notes as naturals. The second, BlackNotesFlat, spells all black notes as flats and all white notes as naturals. The third, RandomFlatSharp, spells all white notes as naturals and assigns either a flat or a sharp with equal probability to each black note. The fourth baseline algorithm, FixedLOFRange, spells each note so that its pitch name class is within a user-specified range containing 12 consecutive positions on the line of fifths.

Table 7.1 shows the results that would be obtained if these algorithms were run on the test corpus $\mathcal{C}$ with the algorithms sorted in descending order of overall note accuracy. Note that the algorithms were not actually run on $\mathcal{C}$ as the results can be calculated directly from the ground-truth OPNDV files themselves (see section 1.3.3.3). Thus, the number of note errors made by BlackNotesSharp will be the sum of the number of white notes that are not spelt as naturals and the number of black notes that are not spelt as sharps. Similarly, in the case of BlackNotesFlat, the note error count will be the sum of the number of white notes not spelt as naturals and the number of black notes not spelt as flats. In the case of RandomFlatSharp, we are interested in the *expected* number of errors rather than the actual number of errors made on any particular run. The expected number of errors made by RandomFlatSharp is $e_w + e_b + (b - e_b)/2$, where e_w is the number of non-natural white notes, e_b is the number of black notes not spelt as single flats or single sharps and b is the number of black notes.[1] FixedLOFRange takes a single parameter, $\ell_{\min}$, which specifies the least line-of-fifths position permitted. The note error count for FixedLOFRange is therefore the number of pitch names with line-of-fifths positions that are either greater than $\ell_{\min} + 11$ or less than $\ell_{\min}$. BlackNotesSharp

[1] There was, in fact, just one occurrence in $\mathcal{C}$ of a black note not spelt as either a single flat or a single sharp. This was a single B𝄪 in the first movement of Haydn's String Quartet in A major, Hob. III:60 (Op. 55, No. 1).

Algorithm	*Note error counts*									*Note accuracies (%)*									
	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	SD_{Sty}
	24505	24493	24493	24500	24490	24494	24500	24497	195972										
FixedLOFRange ($\ell_{min} = -2$: E♭ to G♯)	1431	1324	716	761	1511	646	863	1333	8585	94.16	94.59	97.08	96.89	93.83	97.36	96.48	94.56	95.62	1.47
FixedLOFRange ($\ell_{min} = -3$: A♭ to C♯)	2154	487	1225	1421	1469	787	1553	1954	11050	91.21	98.01	95.00	94.20	94.00	96.79	93.66	92.02	94.36	2.26
FixedLOFRange ($\ell_{min} = -1$: B♭ to D♯)	1801	2417	1021	1198	1936	969	992	1480	11814	92.65	90.13	95.83	95.11	92.09	96.04	95.95	93.96	93.97	2.17
FixedLOFRange ($\ell_{min} = -4$: D♭ to F♯)	3651	390	2500	2649	2213	1710	3044	3031	19188	85.10	98.41	89.79	89.19	90.96	93.02	87.58	87.63	90.21	4.08
BlackNotesSharp	2829	3403	2067	2948	2679	2124	1884	2012	19946	88.46	86.11	91.56	87.97	89.06	91.33	92.31	91.79	89.82	2.24
FixedLOFRange ($\ell_{min} = 0$: F to A♯)	2829	3403	2067	2948	2679	2124	1884	2012	19946	88.46	86.11	91.56	87.97	89.06	91.33	92.31	91.79	89.82	2.24
RandomFlatSharp	4215.0	2081.5	3257.5	3570.5	3262.5	2668.0	3531.5	3437.0	26023.5	82.80	91.50	86.70	85.43	86.68	89.11	85.59	85.97	86.72	2.60
BlackNotesFlat	5601	760	4448	4193	3846	3212	5179	4862	32101	77.14	96.90	81.84	82.89	84.30	86.89	78.86	80.15	83.62	6.18
FixedLOFRange ($\ell_{min} = -5$: G♭ to B)	5601	760	4448	4193	3846	3212	5179	4862	32101	77.14	96.90	81.84	82.89	84.30	86.89	78.86	80.15	83.62	6.18
FixedLOFRange ($\ell_{min} = 1$: C to E♯)	4207	5988	3673	5070	4055	4155	3135	3131	33414	82.83	75.55	85.00	79.31	83.44	83.04	87.20	87.22	82.95	3.94
FixedLOFRange ($\ell_{min} = -6$: C♭ to E)	7752	2375	6820	6030	6065	4848	7685	7527	49102	68.37	90.30	72.16	75.39	75.23	80.21	68.63	69.27	74.94	7.43

Table 7.1: Results obtained when baseline algorithms applied to the test corpus $\mathcal{C}$.

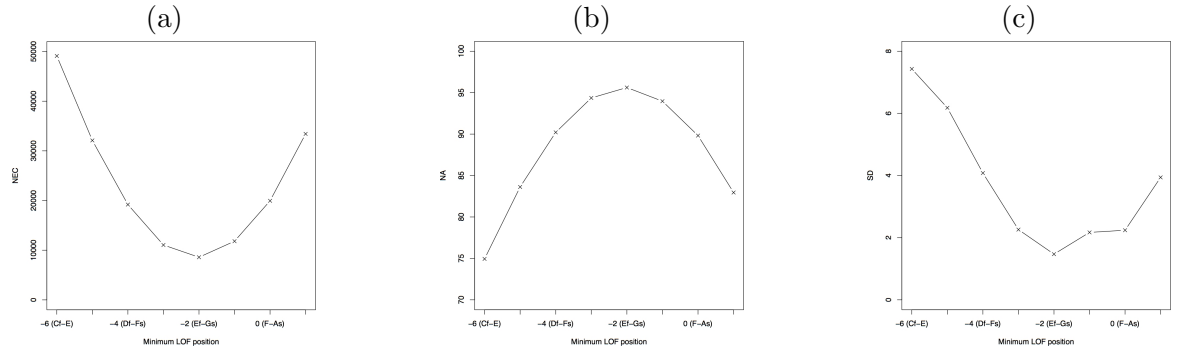


Figure 7.1: Graphs showing how (a) total note error count, (b) note accuracy and (c) style dependence varied with ℓ_{min} for FixedLOFRange when run on the test corpus $\mathcal{C}$.

is exactly equivalent to FixedLOFRange with $\ell_{\text{min}} = 0$ and BlackNotesFlat is exactly equivalent to FixedLOFRange with $\ell_{\text{min}} = -5$. Table 7.1 shows the results that would be obtained with FixedLOFRange on $\mathcal{C}$ for all values of ℓ_{min} between -6 and 1, inclusive. Note that FixedLOFRange performed best when ℓ_{min} was set to -2, that is, when all the notes were spelt so that they were between E♭ and G♯ on the line of fifths. With ℓ_{min} set to -2, FixedLOFRange performed best of all the baseline algorithms, spelling 95.62% of the notes in $\mathcal{C}$ correctly with a style dependence of 1.47. The graphs in Figure 7.1 show how the total note error count, the total note accuracy and the style dependence for FixedLOFRange over $\mathcal{C}$ varied with ℓ_{min}. Note that both the note error count and the style dependence increase steadily as ℓ_{min} moves further away from -2.[2]

7.2 The best versions of the algorithms

Table 7.2 summarises the results obtained for selected versions of the algorithms tested in Chapters 2 to 6, sorted into ascending order of overall note error count achieved over the test corpus, $\mathcal{C}$. The table includes the results for the best version of each algorithm, as well as those for certain baseline and poorly-performing algorithms for comparison. The results in parentheses in this table give the note accuracies obtained when the output for the fourth movement of Haydn's 'Military' Symphony was compared with the version in which the sudden enharmonic change was omitted (see section 1.3.4.3). These parenthesized values are only given for those algorithms that performed better when this enharmonic change was omitted from the ground truth.

7.2.1 Algorithms based on *ps13*

As can be seen in Table 7.2, the algorithm in this study that achieved the highest note accuracy over the test corpus, $\mathcal{C}$, was PS13s1. This algorithm made 1100 errors when processing $\mathcal{C}$, achieving a note accuracy of 99.44%, with a style dependence of 0.49 when $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was set to $\langle 10, 42 \rangle$. When $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was set to $\langle 33, 25 \rangle$, the note error count rose very slightly

[2]The fact that FixedLOFRange performed best over $\mathcal{C}$ with $\ell_{\text{min}} = -2$ is not particularly surprising, since, during the period over which the music in $\mathcal{C}$ was composed, the 'wolf fifth' in any regular mean-tone temperament (Oldham and Lindley, 2006) was commonly placed between G♯ and E♭—that is, the two pitch name classes at the limits of the range permitted by FixedLOFRange when $\ell_{\text{min}} = -2$.

Algorithm	*Note error counts*									*Note accuracies (%)*									
	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	SD_{Sty}
	24505	24493	24493	24500	24490	24494	24500	24497	195972										
PS13s1 ($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$)	40	382	3	65	205	144	96	165	1100	99.84	98.44	99.99	99.73	99.16	99.41	99.61	99.33	99.44	0.49
PS13s1 ($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 33, 25\rangle$)	42	355	7	68	201	156	97	179	1105	99.83	98.55	99.97	99.72	99.18	99.36	99.60	99.27	99.44	0.45
PS1303 ($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$)	21	402	1	61	208	178	80	168	1119	99.91	98.36	100.00	99.75	99.15	99.27	99.67	99.31	99.43	0.53
PS13 ($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$)	34	423	17	66	274	231	113	194	1352	99.86	98.27	99.93	99.73	98.88	99.06	99.54	99.21	99.31	0.57
MH2PX2	30	815	5	17	3229 (267)	170	26	33	4325 (1363)	99.88	96.67	99.98	99.93	86.82 (98.91)	99.31	99.89	99.87	97.79 (99.30)	4.57 (1.13)
MHX2	31	831	4	21	3252 (290)	163	23	27	4352 (1390)	99.87	96.61	99.98	99.91	86.72 (98.82)	99.33	99.91	99.89	97.78 (99.29)	4.61 (1.16)
MH	24	940	2	15	3105 (136)	127	26	149	4388 (1419)	99.90	96.16	99.99	99.94	87.32 (99.44)	99.48	99.89	99.39	97.76 (99.28)	4.41 (1.28)
MH2P	29	951	2	16	3110 (141)	140	26	118	4392 (1423)	99.88	96.12	99.99	99.93	87.30 (99.42)	99.43	99.89	99.52	97.76 (99.27)	4.42 (1.30)
PS13s1 ($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 40, 1\rangle$)	78	413	53	107	286	211	157	292	1597	99.68	98.31	99.78	99.56	98.83	99.14	99.36	98.81	99.19	0.51
CCOP01–06	175	311	152	136	365	230	149	147	1665	99.29	98.73	99.38	99.44	98.51	99.06	99.39	99.40	99.15	0.35
CCOP07–12	150	295	138	132	421	220	155	155	1666	99.39	98.80	99.44	99.46	98.28	99.10	99.37	99.37	99.15	0.42
CamOpt	225	242	138	126	473	156	176	139	1675	99.08	99.01	99.44	99.49	98.07	99.36	99.28	99.43	99.15	0.47
Cam01A	283	232	141	128	476	200	172	190	1822	98.85	99.05	99.42	99.48	98.06	99.18	99.30	99.22	99.07	0.46
TPR1C	148	549	119	165	3332 (376)	229	150	104	4796 (1840)	99.40	97.76	99.51	99.33	86.39 (98.46)	99.07	99.39	99.58	97.55 (99.06)	4.55 (0.63)
TPR1A	151	532	119	166	428	236	150	105	1887	99.38	97.83	99.51	99.32	98.25	99.04	99.39	99.57	99.04	0.65
LH1V	1045	334	74	41	1202	263	210	339	3508	95.74	98.64	99.70	99.83	95.09	98.93	99.14	98.62	98.21	1.79
FixedLOFRange ($\ell_{\text{min}} = -2$)	1431	1324	716	761	1511	646	863	1333	8585	94.16	94.59	97.08	96.89	93.83	97.36	96.48	94.56	95.62	1.47
LH1	116	890	102	681	2662	3689	418	1024	9582	99.53	96.37	99.58	97.22	89.13	84.94	98.29	95.82	95.11	5.28
BlackNotesSharp	2829	3403	2067	2948	2679	2124	1884	2012	19946	88.46	86.11	91.56	87.97	89.06	91.33	92.31	91.79	89.82	2.24
RandomFlatSharp	4215.0	2081.5	3257.5	3570.5	3262.5	2668.0	3531.5	3437.0	26023.5	82.80	91.50	86.70	85.43	86.68	89.11	85.59	85.97	86.72	2.60
MHD4	1159	9584	2096	1787	8562 (6599)	3168	2396	5082	33834 (31871)	95.27	60.87	91.44	92.71	65.04 (73.05)	87.07	90.22	79.25	82.74 (83.74)	13.15 (11.85)
BlackNotesFlat	5601	760	4448	4193	3846	3212	5179	4862	32101	77.14	96.90	81.84	82.89	84.30	86.89	78.86	80.15	83.62	6.18
FixedLOFRange ($\ell_{\text{min}} = 1$)	4207	5988	3673	5070	4055	4155	3135	3131	33414	82.83	75.55	85.00	79.31	83.44	83.04	87.20	87.22	82.95	3.94
MH2PD4	3905	12679	3584	3405	12170 (10209)	4495	2740	6843	49821 (47860)	84.06	48.23	85.37	86.10	50.31 (58.31)	81.65	88.82	72.07	74.58 (75.58)	16.39 (14.88)
FixedLOFRange ($\ell_{\text{min}} = -6$)	7752	2375	6820	6030	6065	4848	7685	7527	49102	68.37	90.30	72.16	75.39	75.23	80.21	68.63	69.27	74.94	7.43

Table 7.2: Results obtained for the best versions of the algorithms evaluated in Chapters 2 to 6. The algorithms are sorted into ascending order of overall note error count.

to 1105 but the style dependence dropped to 0.45. PS13s1 consists of just Stage 1 of PS13, but performed consistently better than PS13 in terms of note accuracy and style dependence. PS13s1 was also marginally more accurate than the PS1303 algorithm which consists of PS13, followed by a post-processing phase in which the pitch names assigned by PS1303 are transposed so that they are closer to their neighbours on the line of fifths. PS13s1 was found to be superior to the other versions of *ps13* tested in terms of note accuracy, style dependence, time complexity, robustness to temporal deviation and ease of implementation.

7.2.2 Temperley and Sleator's *Melisma* programs

When Temperley and Sleator's algorithm was evaluated on the test corpus, $\mathcal{C}$, the best results were obtained when the 'two-pass' version of the algorithm, MH2P, was run on the half-speed version of the corpus (see entry labelled "MH2PX2" in Table 7.2). MH2P spelt 99.30% of the notes in this version of the corpus correctly with a style dependence of 1.13. However, this was only the case when the sudden enharmonic change in the fourth movement of Haydn's 'Military' Symphony was omitted from the ground truth. When this enharmonic change was included, the note accuracy of MH2P on the half-speed version of the corpus dropped to 97.79% and the style dependence rose to 4.57. The note accuracy and style dependence achieved by MH2P on the half-speed version of the test corpus were only marginally better than those achieved by the simpler MH algorithm, which only involved running the `meter` and `harmony` programs once each on the data (see entry labelled MHX2 in Table 7.2). Also, the MH and MH2P algorithms only performed marginally better on the half-speed version of the corpus than they did on the version in which the music was at a natural tempo.

The note accuracies achieved by the best versions of Temperley and Sleator's algorithm on $\mathcal{C}$ were only slightly less than that achieved by PS13 when $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was set to $\langle 10, 42 \rangle$. However, Temperley and Sleator's algorithm performed considerably less consistently across the 8 composers in the test corpus than PS13 (compare SD_{Sty} values for MH2PX2 and PS13 in Table 7.2). Also, whereas the algorithms based on PS13 are unaffected by tempo, the note accuracies over $\mathcal{C}$ of MH and MH2P were highly sensitive to tempo, dropping to 83.74% and 75.58%, respectively, (i.e., worse than simply spelling all the black notes as flats), when the music was at 4 times its 'natural' tempo (see entries for MHD4 and MH2PD4 in Table 7.2). Furthermore, the performance of Temperley and Sleator's algorithm depends on the durations of the notes in the input passage being close to proportional to their notated values. This makes the algorithm less robust to temporal deviations of the type that typically occurs in data derived from performances. The fact that PS13s1 does not use note durations makes it immune to this type of 'noise' in the input data.

Also, it should be remembered that Temperley and Sleator's algorithms are by far the most complicated of the algorithms considered in this study, each involving carrying out a complete harmonic and metrical analysis of the passage to be processed. This complexity contrasts sharply with the simplicity of the PS13s1 algorithm which consistently made nearly 20% fewer errors than the best version of Temperley and Sleator's algorithm.

It should perhaps also be pointed out that running Temperley and Sleator's algorithms on a representation of a passage of music often causes information in the input representation to be

lost. For example, some notes in the input are deleted and the relative ordering of the notes is sometimes changed in an arbitrary way because of quantization effects.

The PS13s1 algorithm is therefore superior for pitch spelling to Temperley and Sleator's algorithms in terms of note accuracy, style dependence, robustness to temporal deviations and ease of implementation, as well as in terms of preserving information in the input data.

7.2.3 Chew and Chen's algorithms

The twelve most accurate versions of Chew and Chen's algorithm tested were those for which the global window was 8 chunks long, the local window was 2 chunks long, the chunk size was half a second and the local and cumulative CEs were equally weighted. These 12 best versions of the algorithm spelt 99.15% of the notes in $\mathcal{C}$ correctly (see entries for CCOP01–06 and CCOP07–12 in Table 7.2). It has also been proved (see Appendix C) that using the line of fifths in Chew and Chen's algorithm instead of the spiral array makes no difference to its output. Of these 12 best versions, the 6 in which all the notes *sounding* in each window were considered in calculating the CEs (CCOP01–06) were slightly less dependent on style than those which only used the notes *starting* in each window (CCOP07–12) (see Table 7.2). Indeed, CCOP01–06 were less affected by the stylistic differences between the composers in the test corpus than any of the other algorithms tested in this study. Moreover, Chew and Chen's algorithm is relatively fast, relatively easy to implement and well-suited to being used in real-time applications, as the choice of pitch name for each note only depends on the notes that precede it in the music.

Nevertheless, the best versions of Chew and Chen's algorithm made over 50% more errors than PS13s1. Also, Chew and Chen's algorithm, unlike PS13s1, uses the duration of each note to weight its contribution when calculating a center of effect, which potentially makes the algorithm less robust to the problem of note durations not being strictly proportional to their notated values in data derived from performances. Moreover, PS13s1 is simpler than Chew and Chen's algorithm and can easily be implemented as a real-time algorithm, provided that the post-context is empty, which is achieved by setting K_{post} to 1. When PS13s1 was run on the test corpus with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to $\langle 40, 1 \rangle$, it spelt 99.19% of the notes in the test corpus correctly—that is, it made over 4% *fewer* errors than Chew and Chen's algorithm (see Table 7.2).

Chew and Chen's algorithm and PS13s1 are similar insofar as

1. they are both based on the idea that each note should be spelt so that it is as close as possible to the tonic on the line of fifths (or, in Chew and Chen's algorithm, the line of fifths coiled up to form the spiral array);

2. they both assume that the strength with which a tonic is implied at a point increases with the frequency with which the tonic occurs within a window surrounding (or, in Chew and Chen's algorithm, preceding) the note to be spelt.

However, there are some important differences between the best versions of Chew and Chen's algorithm and PS13s1. First, Chew and Chen's algorithm uses a complex windowing process involving combining windows at three different scales of structure containing only notes that precede the note to be spelt; whereas PS13s1 uses a single window which may contain notes preceding and following the note to be spelt. Second, Chew and Chen model the local tonic

as a single point on the spiral array or line of fifths; whereas, in PS13s1, the sense of key at a point is modelled as a frequency distribution which allows each potential tonic pitch class to make a contribution to the choice of pitch name in proportion to the frequency with which it occurs within the context window. Third, Chew and Chen weight the contribution of each note by its duration; whereas, in PS13s1, duration is ignored (as it was in the data produced by Youngblood (1958) and Knopoff and Hutchinson (1983) that Krumhansl (1990, pp. 66–70) used to show that there is a strong correlation between pitch class frequency distributions in tonal works and the probe-tone rating profiles for major and minor keys).

7.2.4 Using the CE or COG as a proxy for the tonic

Another important difference between PS13s1 and Chew and Chen's algorithm concerns the way in which the algorithms find the local tonic at each point in the music. In the 12 best versions of Chew and Chen's algorithm tested, the center of effect (CE) of a set of notes on the line of fifths or spiral array is used as a "proxy" for the key (Chew and Chen, 2005, p. 63) and then the notes are spelt so that they are as close as possible to the CE on the line of fifths or in the spiral array (as proved in Appendix C, the line of fifths works just as well as the spiral array in Chew and Chen's algorithm). The CE is simply the weighted average position on the line of fifths (or spiral array) of the pitch names of the notes within a particular window. It is therefore essentially identical to Temperley's COG (Temperley, 2001, pp. 125–126, 132–133). However, Temperley (2001, p. 127) found that "for actual musical passages in the major, the COG is generally about two steps in the sharp direction from the tonic" along the line of fifths. This suggests that using the line-of-fifths CE or COG of a set of notes as a proxy for the tonic is not justified. If Temperley is right, then a better estimate of the location of the tonic on the line of fifths could be achieved by subtracting 2 from the COG or CE.

Spelling the notes in a passage so that they are as close as possible on the line of fifths to the current COG or CE is therefore *not* equivalent to spelling them so that they are as close as possible to the local tonic on the line of fifths. However, traditional tonal music theory seems to endorse the idea of spelling notes so that they are as close as possible to the *tonic* on the line of fifths. For example, spelling notes in accordance with the conventional harmonic chromatic scale on a particular tonic (Associated Board of the Royal Schools of Music, 1958, p. 78) is the same as spelling them so that they are as close as possible to that tonic on the line of fifths, with notes 6 semitones from the tonic being preferably spelt as sharpened subdominants rather than flattened dominants. In other words, tonal theory seems to suggest that the line-of-fifths position, ℓ, of a note should satisfy the inequality $t - 5 \leq \ell \leq t + 6$ where t is the line-of-fifths position of the tonic. This is also the idea underlying Longuet-Higgins's Rule 1 (Longuet-Higgins, 1987a, pp. 112–113) (see section 2.2). The fact that PS13s1 is more accurate than Chew and Chen's algorithm may therefore be partly due to the CE not being a good estimate of the local tonic.

PS13s1 therefore manages to be more accurate than Chew and Chen's algorithm while also being simpler and more robust, in that it uses a simpler windowing system and ignores note durations. Also, PS13s1, like Chew and Chen's algorithm, can be implemented as a real-time process. Therefore, even though the performance of Chew and Chen's algorithm appears to vary slightly less with musical style than that of PS13s1, it is hard to imagine a situation in which

one would choose to use Chew and Chen's algorithm in preference to PS13s1.

7.2.5 Cambouropoulos's algorithms

The most accurate versions of Cambouropoulos's algorithm tested differ from the algorithms of Temperley, Longuet-Higgins, Chew and Chen and myself in that they do not explicitly use the line of fifths. They are instead based on Cambouropoulos's principles of "notational parsimony" and "interval optimization" (Cambouropoulos, 2003, p. 421), where the latter involves penalising spellings for containing intervals that occur infrequently in the major and minor scales. However, the principle of notational parsimony involves penalising a spelling if it involves using more than one sharp or flat, and this is equivalent to penalising a pitch name class if its line-of-fifths position is greater than 13 (i.e., 'sharper' than B♯) or less than −7 (i.e., 'flatter' than F♭). This principle, therefore, (perhaps unintentionally) implies the use of the line of fifths.

The most accurate version of Cambouropoulos's algorithm tested was the CamOpt algorithm, which I developed by combining the best-performing features of the other versions of the algorithm tested in my evaluation. More specifically, the CamOpt algorithm is the same as my implementation of the algorithm described by Cambouropoulos (2001), except that

1. it uses a window containing 12 distinct MIDI note numbers rather than 9;
2. it uses the interval optimization and notational parsimony penalty values used in my implementation of the algorithm described by Cambouropoulos (2003); and
3. the modality class of each interval is determined in the same way as in my implementation of the algorithm described by Cambouropoulos (1996, 1998), except that the boundaries of the modality classes are moved so that modality classes A and C cover a wider range of values and B covers a narrower range.

When CamOpt was run on the test corpus, $\mathcal{C}$, it made 8% fewer errors than the most accurate of the other versions of the algorithm tested. CamOpt spelt 99.15% of the notes in the test corpus correctly with a style dependence of 0.47 (see Table 7.2). The next most accurate version of Cambouropoulos's algorithm tested was Cam01A, which was also the most accurate of the versions described and tested by Cambouropoulos himself (Cambouropoulos, 2001) (see Table 7.2). Cam01A spelt 99.07% of the notes in $\mathcal{C}$ correctly with a style dependence of 0.46 (i.e., it was very slightly less dependent on style than CamOpt). Cambouropoulos's algorithms do not use note durations and do not rely heavily on the note onsets being strictly in proportion to their notated values. This makes them robust to the types of temporal deviations that one typically finds in data derived from performances.

However, the running times of the most accurate versions of Cambouropoulos's algorithm are exponential in the size of the window used, which places a relatively low upper limit of about 12 on the size of the window that can be used in practice. Even with window sizes less than 12, the versions of Cambouropoulos's algorithm tested in this study were (with the quadratic version of TPROne) amongst the slowest of the algorithms tested—my implementation of CamOpt took over 40 hours to process the test corpus. Cam01A took about 6 hours, but this was still extremely slow in comparison with PS13s1, which took around 20 minutes.

Also, even CamOpt, the most accurate version of Cambouropoulos's algorithm tested, made over 50% more errors than PS13s1 over $\mathcal{C}$. And Cam01A, the most accurate of the versions described by Cambouropoulos himself, made around 66% more errors over $\mathcal{C}$ than PS13s1. Furthermore, the style dependence achieved by CamOpt was only marginally better than that of PS13s1, and, with a minute sacrifice in note accuracy (1105 errors instead of 1100), the style dependence of PS13s1 could actually be made marginally better than both CamOpt and Cam01A by setting $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ to $\langle 33, 25 \rangle$ instead of $\langle 10, 42 \rangle$. Therefore it is hard to imagine a situation in which Cambouropoulos's algorithm would be preferable to PS13s1.

7.2.6 TPROne

The TPROne algorithm, my implementation of Temperley's TPR 1 (Temperley, 2001, p. 125), performed surprisingly well over $\mathcal{C}$, considering that it simply assigns pitch names so that notes that are nearby in the music are spelt so that they are close together on the line of fifths. The versions of TPROne tested over $\mathcal{C}$ included a quadratic-time version, that takes into account all the notes preceding the note to be spelt; and a linear-time version, that considers the w notes preceding the note to be spelt, where w is a user-supplied value. The best quadratic-time version tested here made 1887 errors, spelling 99.04% of the notes in $\mathcal{C}$ correctly with a style dependence of 0.65 (see entry labelled TPR1A in Table 7.2). The best version of the linear-time algorithm spelt 99.06% of the notes in $\mathcal{C}$ correctly with a style dependence of 0.63, when the enharmonic change in the fourth movement of Haydn's 'Military' Symphony was ignored. Therefore, when the enharmonic change in the 'Military' Symphony was included, the quadratic-time version of TPROne actually made over 56% fewer errors than the best version of Temperley and Sleator's *Melisma* system (MH2PX2). However, with this enharmonic change ignored, the best version of TPROne (TPR1C) made nearly 35% more errors than the best system involving the `harmony` and `meter` programs (MH2PX2).

TPROne has several advantages over MH and MH2P, however. First, the performance of TPROne does not depend on tempo, nor does it rely on the note onsets or durations being strictly proportional to their notated values. This makes it potentially more robust to the types of temporal deviations that typically occur in data derived from performances. Second, TPROne is much easier to implement than MH or MH2P. Third, the two best versions of TPROne (TPR1A and TPR1C) were both considerably less dependent on style than MH and MH2P.

Nevertheless, the quadratic-time version of TPROne (TPR1A) would be impractically slow for processing very large movements. Also, it is hard to imagine why one would want to use TPROne rather than PS13s1, since the most accurate version of TPROne tested (TPR1C) made 67% more errors on $\mathcal{C}$ than PS13s1. Also, PS13s1 was less dependent on style than TPROne. Moreover, PS13s1 was faster, no harder to implement and no less robust to temporal deviation than TPROne.

Like Chew and Chen's algorithm, TPROne spells each note so that it is as close as possible on the line of fifths to the weighted average position on the line of fifths of the notes in a window preceding the note to be spelt. Unlike Chew and Chen, Temperley does not claim that this average position is a good estimate of the local tonic. However, as discussed above, the

structure of the harmonic chromatic scale and the greater note accuracy achieved by PS13s1 suggest that spelling notes so that they are as close as possible to the local tonic on the line of fifths may be a more successful strategy than spelling them so that they are close on the line of fifths to the notes nearby in the music. This suggests that Temperley's TPR 1 should perhaps be changed to

New TPR 1 Prefer to label events so that they are as close as possible on the line of fifths to the local tonic.

It may be possible to implement this change fairly simply by, for example, using Temperley's (2001, p. 127) observation that "for actual musical passages in the major, the COG is generally about two steps in the sharp direction from the tonic" along the line of fifths. However, to do so would imply that a knowledge of the local tonic is necessary for determining the pitch name of a note and this would contradict "one of the main claims" of Temperley's model, which "is that spelling can be accomplished without relying on 'top-down' key information" (Temperley, 2001, p. 126) (see also Meredith, 2002b, p. 294).

7.2.7 Longuet-Higgins's algorithm

The original version of Longuet-Higgins's algorithm, as implemented in his `music.p` program, performed better than those versions in which the implementation of his Rule 2 had been corrected. This algorithm, like PS13s1, is based on the idea of spelling the notes so that they are as close as possible to the local tonic on the line of fifths. However, in Longuet-Higgins's algorithm, it is assumed that there is one unambiguous local tonic; whereas, in PS13s1, all possible tonics are taken into account and the strength with which each tonic is implied is assumed to be proportional to the frequency with which it occurs within the context surrounding the note. Also, in PS13s1, the tonic chroma frequency distribution is updated for each note; whereas, in Longuet-Higgins's algorithm, the local tonic is assumed to remain constant unless certain configurations of chromatic intervals occur between consecutive notes in the input passage.

Unlike PS13s1, which does not take voice-leading into account, three of the six rules in the theory of tonality implemented in Longuet-Higgins's algorithm are concerned with chromatic intervals between consecutive notes in the input passage, which is assumed to be a monophonic melody. It was therefore not surprising that Longuet-Higgins's algorithm was considerably more accurate when the notes in the data were sorted so that the voices were arranged 'end-to-end', than when the notes in the input were sorted by onset time so that the voices were approximately 'interleaved'. When the voices were end-to-end, the best version of Longuet-Higgins's algorithm spelt 98.21% of the notes in $\mathcal{C}$ correctly with a style dependence of 1.79 (see entry for LH1V in Table 7.2). However, when the voices were 'interleaved', the note accuracy dropped to 95.11% and the style dependence rose to 5.28 (see entry for LH1 in Table 7.2). Note that, when the voices were 'interleaved', both the note accuracy and style dependence of Longuet-Higgins's algorithm were worse than those of the baseline algorithm FixedLOFRange when $\ell_{\min}$ was set to -2.

Longuet-Higgins's algorithm places relatively more emphasis on voice-leading as a factor in pitch spelling than the other algorithms tested in this study. As can be seen in Table 7.2,

this emphasis does not seem to have improved its note accuracy. PS13s1 ignores voice-leading altogether but Longuet-Higgins's algorithm made over 3 times as many errors as PS13s1 when the voices were end-to-end, and nearly 9 times as many errors when the voices were 'interleaved'. Indeed, several results obtained in this study suggest that voice-leading is of relatively minor importance in pitch spelling. In particular,

1. removing Stage 2 from PS13 reduced the overall note error count over $\mathcal{C}$ by 19%;
2. incorporating the 'tie-breaker' voice-leading rule into versions of Cambouropoulos's algorithm increased the overall note error count (see row 12 in Table 3.16);
3. versions of Cambouropoulos's algorithm performed less well when the voices were presented 'end-to-end' than when they were 'interleaved' (see row 3 in Table 3.16);
4. TPROne made 56% fewer errors than MH2P (which includes an implementation of Temperley's TPR 2, the 'Voice-leading Rule' (Temperley, 2001, pp. 127–130)) when the enharmonic change in Haydn's 'Military' Symphony was included in the ground truth.

Of course, it is possible that accuracy could be improved by using a more sophisticated mechanism for modelling the effect of voice-leading on pitch spelling than the rather crude methods used in the algorithms considered in this study.

7.2.8 The perceptual and cognitive implications of the context windows used in the best versions of the *ps13*-based algorithms

Unlike any of the other algorithms tested in this study, the best versions of PS13 and PS13s1 employ a substantial "post-context" containing between 23 and 42 notes *following* the note to be spelt. In Temperley and Sleator's system, Chew and Chen's algorithm and TPROne, the spelling of a note is assumed to depend only on the notes that precede it. In Longuet-Higgins's algorithm, the context used to spell a note includes one or two notes following the note to be spelt. In Cambouropoulos's algorithm, each window typically contains 9 or 12 notes and the final third of each window influences the spelling of the notes in the middle third. But the post-contexts used in the algorithms of Longuet-Higgins and Cambouropoulos are very small compared to the post-contexts used in the best versions of PS13 and PS13s1. Also, when the post-context was made empty in PS13s1, its accuracy dropped considerably (for PS13s1, compare the results when $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 40, 1\rangle$ with those when $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$). Moreover, the image plots in Figures 6.14 and 6.18 on pages 295 and 301, respectively, suggest that decreasing the post-context size to less than about 13 notes decreases accuracy whilst increasing it even up to 50 notes causes no decrease in accuracy provided the pre-context is in an appropriate range. On the other hand, both image plots suggest that increasing the pre-context beyond a certain limit causes a decrease in accuracy.

In the *ps13*-based algorithms, the context is only used to provide a chroma frequency distribution and the frequency with which each chroma occurs within the context is used as a predictor of the likelihood of that chroma being perceived to be the local tonic at the point where the note to be spelt occurs. The most accurate algorithms in this study used a substantial post-context; therefore, whether or not a pitch is perceived to be the tonic at a particular

time point would seem to depend not only on the music that immediately precedes that time point but also on the music that immediately follows it. In fact, the results obtained here suggest that the post-context used to determine the local tonic may even be typically *larger* than the pre-context. Moreover, the fact that the *ps13*-based algorithms, which use only a relatively small pre-context, were more accurate than those algorithms that use *all* the music preceding the note to be spelt suggests that whether or not a pitch is perceived to be a tonic at a time point depends on only a relatively local context and not on the large-scale, long-range structure of the music. The latter result seems to be more in agreement with what Tillmann and Bigand (2004) call the "concatenationist view of music perception" (e.g., Gurney, 1966; Levinson, 1997) than those theories that emphasize the importance of large-scale, hierarchical structure (e.g., Lerdahl and Jackendoff, 1983).

All these inferences are based on the assumption that the frequency with which a chroma occurs within a context is the principal factor in determining whether or not it is perceived to be the tonic. However, whilst Krumhansl (1990, pp. 66–70) showed that chroma frequency is highly correlated with tonal stability (see discussion on page 278 above), this correlation does not imply any causal relationship between the two. For example, it might be that chroma frequency and perceived tonal stability are both consequences of composers using particular characteristic note configurations (e.g., melodic motifs and chord progressions) that the listener associates with particular classes of tonal event such as modulations and cadences that are then used to identify the local key at each point in the music. Alternatively, it could be that the intervals that occur rarely within the diatonic scales (e.g., the tritone) provide the most important cues for identifying the local tonic (Brown, 1988; Butler, 1989; Butler and Brown, 1984).

It should also be remembered that the pre- and post-context sizes identified as optimal for PS13 and PS13s1 in this study were those that allowed these algorithms to perform most accurately over the test corpus *as a whole*. The best context to use for identifying the tonic at any particular time point could be quite different from the optimal values found here and could, indeed, vary considerably from one point to another in a passage of music.

From Table 6.6 on page 304, it can be seen that, for the five best-performing $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ combinations, the size of the context was between 50 and 58 consecutive notes (the notes being sorted first by onset time and then by chromatic pitch). In the version of the test corpus in which the music was at its "natural" tempo, the mean inter-onset interval between notes 50 notes apart was 5.03 seconds and that between notes 58 notes apart was 5.81 seconds. These values correspond remarkably well with estimates of the maximum duration of the psychological (or perceptual) present, which, according to Fraisse (1982, pp. 157–8), is around 5 seconds. The psychological present is that time interval ending with the current instant in which events are perceived as occurring in the present. Clarke (1999, p. 476) defines the maximum duration of the perceptual present as being "the boundary between direct perception and the memory-dependent processes of construction and estimation" and states that "a value somewhere around 3–8 seconds is in agreement with a good deal of the available evidence". Note that the mean durations of the best context sizes found here lie right in the middle of Clarke's range of 3–8 seconds. Because all events within the perceptual present are "directly perceived", it is particularly easy for events occurring early during the perceptual present to be retrospectively re-intepreted in the light of

events occurring later on during this period. It is therefore not particularly surprising that notes occurring up to around 4 seconds after the note to be spelt can influence its interpretation and therefore its spelling.

7.3 Significance of differences in note accuracy for best versions of algorithms

Each entry in Table 7.3 gives an estimate of the p value for the difference between the note accuracies achieved over $\mathcal{C}$ by the algorithms at the heads of the column and row containing the entry. Each of these p values was calculated using the method described in section 1.3.4.4. That is, each p value is the mean of the four p values returned by the matched-pairs t test when it is computed over the test corpus partitions G_4, G_8, G_{16} and G_{Comp} (see section 1.3.4.4). From Table 7.3, it can be seen that, if we set α, the probability of Type I error (Howell, 1982, p. 62), to its usual value of 0.05, then this method of estimating the statistical significance of the difference in note accuracy between two algorithms tells us that

1. PS13s1 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to either $\langle 10, 42\rangle$ or $\langle 33, 25\rangle$ and PS1303 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$ were significantly more accurate than PS13 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$;

2. PS13s1 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to either $\langle 10, 42\rangle$ or $\langle 33, 25\rangle$ and both PS1303 and PS13 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$ were significantly more accurate than

   (a) PS13s1 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 40, 1\rangle$,

   (b) both the quadratic and linear versions of TPROne (TPR1A and TPR1C),

   (c) the best version of Longuet-Higgins's algorithm when the voices were 'interleaved' (LH1) and

   (d) the baseline algorithms tested;

3. the best versions of Temperley and Sleator's algorithm (MH2PX2, MHX2, MH and MH2P), the 'real-time' version of PS13s1 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 40, 1\rangle$, the best versions of Chew and Chen's algorithm (CCOP01–12), the best versions of Cambouropoulos's algorithm (CamOpt and Cam01A) and the best versions of TPROne (TPR1A and TPR1C) were all significantly more accurate than the best version of Longuet-Higgins's algorithm with the voices interleaved (LH1) and the baseline algorithms;

4. the best version of Longuet-Higgins's algorithm with the voices end-to-end (LH1V) was significantly more accurate than the baseline algorithms;

5. the best version of Longuet-Higgins's algorithm with the voices interleaved (LH1) was significantly more accurate than all the baseline algorithms apart from FixedLOFRange with ℓ_{min} set to -2 (from which it differed insignificantly in accuracy);

6. all the other differences in note accuracy between the best versions of the algorithms tested were not statistically significant.

	PS13s1 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 33, 25\rangle\rangle$)	PS1303 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle\rangle$)	PS13 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle\rangle$)	MH2PX2	MHX2	MH	MH2P	PS13s1 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 40, 1\rangle\rangle$)	CCOP01–06	CCOP07–12	CamOpt	Cam01A	TPR1C	TPR1A	LH1V	LH1	BlackNotesSharp	FixedLOfRange ($\ell_{\text{min}} = -2$)	BlackNotesFlat	RandomFlatSharp
PS13s1 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle\rangle$)	0.89310	0.70436	*0.00739*	0.44882	0.41588	0.47743	0.48434	*0.00054*	0.12218	0.14263	0.22721	0.17968	*0.00546*	*0.00595*	0.11381	*0.02416*	*0.00009*	*0.00053*	*0.00083*	*0.00022*
PS13s1 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 33, 25\rangle\rangle$)		0.71672	*0.01361*	0.45718	0.42054	0.48591	0.49180	*0.00063*	0.12108	0.14433	0.22573	0.17658	*0.00697*	*0.00743*	0.11572	*0.02380*	*0.00008*	*0.00051*	*0.00082*	*0.00021*
PS1303 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle\rangle$)			*0.00856*	0.47140	0.43160	0.49674	0.50220	*0.00136*	0.12526	0.15008	0.23361	0.18231	*0.00482*	*0.00567*	0.11685	*0.02330*	*0.00009*	*0.00048*	*0.00082*	*0.00021*
PS13 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle\rangle$)				0.97273	0.90526	0.87289	0.86951	*0.02737*	0.35285	0.37545	0.48289	0.36344	*0.02752*	*0.02325*	0.14904	*0.02571*	*0.00009*	*0.00062*	*0.00086*	*0.00023*
MH2PX2					0.49525	0.69584	0.67342	0.52914	0.58391	0.59964	0.64679	0.53427	0.16876	0.15782	0.18736	*0.02854*	*0.00007*	*0.00105*	*0.00102*	*0.00028*
MHX2						0.83994	0.81694	0.57582	0.61676	0.63257	0.67413	0.55617	0.19370	0.18003	0.18964	*0.02874*	*0.00007*	*0.00097*	*0.00100*	*0.00027*
MH							0.90138	0.69370	0.68777	0.69838	0.73847	0.62706	0.32268	0.29792	0.21391	*0.03219*	*0.00008*	*0.00137*	*0.00108*	*0.00032*
MH2P								0.71034	0.69959	0.71028	0.74601	0.63446	0.34059	0.31523	0.21822	*0.03208*	*0.00008*	*0.00140*	*0.00109*	*0.00032*
PS13s1 ($\langle\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 40, 1\rangle\rangle$)									0.81913	0.82644	0.85624	0.63814	0.19405	0.14326	0.18648	*0.02939*	*0.00010*	*0.00064*	*0.00087*	*0.00023*
CCOP01–06										0.98715	0.96727	0.61792	0.50353	0.39236	0.16568	*0.03032*	*0.00020*	*0.00082*	*0.00081*	*0.00024*
CCOP07–12											0.96962	0.61868	0.53901	0.41916	0.15768	*0.03032*	*0.00023*	*0.00095*	*0.00083*	*0.00026*
CamOpt												0.24581	0.68429	0.59723	0.14142	*0.02794*	*0.00023*	*0.00037*	*0.00063*	*0.00015*
Cam01A													0.96915	0.88731	0.15979	*0.02861*	*0.00022*	*0.00019*	*0.00056*	*0.00011*
TPR1C														0.26901	0.22987	*0.03327*	*0.00012*	*0.00082*	*0.00091*	*0.00025*
TPR1A															0.23460	*0.03368*	*0.00013*	*0.00086*	*0.00092*	*0.00026*
LH1V																0.08573	*0.00102*	*0.00567*	*0.00052*	*0.00017*
LH1																	*0.02815*	0.68783	*0.00284*	*0.00266*
BlackNotesSharp																		*0.00068*	*0.03964*	*0.03964*
FixedLOfRange ($\ell_{\text{min}} = -2$)																			*0.00119*	*0.00015*
BlackNotesFlat																				*0.03963*

Table 7.3: Each entry in this table gives the p value, calculated using the method described in section 1.3.4.4, for the difference between the note accuracies achieved by the algorithms at the heads of the row and column of the entry. p values less than or equal to 0.05 are printed in italic.

Chunk	PS13s1 (X1)	PS13 (X2)	LH1V (X3)	X1-X2	X1-X3
1	99.60	99.42	99.16	0.18	0.44
2	99.64	99.59	95.14	0.05	4.50
3	99.67	99.56	99.34	0.11	0.33
4	99.73	99.58	95.59	0.16	4.14
5	98.57	98.40	99.31	0.17	-0.74
6	99.36	99.27	98.69	0.09	0.67
7	99.71	99.56	99.12	0.15	0.59
8	99.23	99.11	99.33	0.12	-0.09
Mean (m)	99.44	99.31	98.21	0.13	1.23
Standard deviation (s)	0.39	0.41	1.77	0.04	1.96
m/s				2.92	0.63
t statistic				8.26	1.77
p value				0.00007	0.11964

Table 7.4: t test analysis of differences in chunk note accuracies achieved over G_8 by PS13s1 and PS13 (with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$) and Longuet-Higgins' algorithm, run on the version of the test corpus in which the voices were arranged end-to-end (LH1V). The column headed "X1-X2" gives the difference in chunk note accuracy for each chunk between PS13s1 and PS13. The column headed "X1-X3" gives the difference in chunk note accuracy for each chunk between PS13s1 and LH1V.

Note that this method of measuring the significance of the difference in note accuracy between two algorithms tells us that the 99.44% note accuracy achieved by the best versions of PS13s1 over $\mathcal{C}$ was significantly better than the 99.31% achieved by the best version of PS13 but *not* significantly different from the *lower* note accuracies achieved by the best algorithms of Temperley and Sleator (99.27–99.30%), Chew and Chen (99.15%), Cambouropoulos (99.07%, 99.15%) or even Longuet-Higgins (98.21%). This may at first seem surprising, but it is a consequence of the fact that the significance of the difference in performance between two algorithms over a particular test corpus partition depends, not directly on the difference in the mean chunk note accuracies, but on the *ratio* of this mean difference to the standard deviation of the chunk note accuracy differences. Consequently, it is possible for a large mean difference in chunk note accuracy to be insignificant if the variance in the chunk note accuracy differences is sufficiently high. This can happen if the variance in the chunk note accuracies of just one of the algorithms being compared is very high—as it is, for example, when the best version of PS13s1 (which has a low chunk note accuracy variance) is compared with the best version of Longuet-Higgins's algorithm (which has a rather high chunk note accuracy variance).

This can perhaps best be understood by examining precisely how the estimates of significance are derived for two pairs of algorithms: one pair for which a relatively small difference in accuracy is deemed significant and another pair for which a relatively large difference in accuracy is deemed insignificant. Consider, therefore, the data given in Table 7.4. The second column in this table gives the chunk note accuracy achieved by PS13s1 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$ for each of the eight chunks in the test corpus partition G_8. Similarly, the columns headed PS13 and

LH1V give the chunk note accuracies over G_8 for PS13 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$ and the original version of Longuet-Higgins' algorithm run on the version of the test corpus in which the voices were arranged end-to-end. As can be seen in this table, the difference in the mean chunk note accuracies achieved over G_8 by PS13s1 and PS13 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$ is highly significant ($p = 0.00007$) even though the mean chunk note accuracy difference for these two algorithms is rather low (0.13). On the other hand, the difference in the mean chunk note accuracies achieved by PS13s1 ($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$) and LH1V is not statistically significant ($p = 0.11964$) even though the mean difference in chunk note accuracy (1.23) is nearly ten times larger than that between PS13s1 and PS13 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 10, 42\rangle$. The difference between the mean chunk note accuracies for two algorithms over G_8 is deemed significant by the matched-pairs t test at the 0.05 level if the t statistic is greater than some particular value that can be looked up in a table or computed using software. The t statistic is given by

$$t = \frac{\bar{D}}{s_{\bar{D}}} \tag{7.1}$$

where $\bar{D}$ is the mean of the chunk note accuracy differences and $s_{\bar{D}}$ is the standard error in $\bar{D}$ (see Howell, 1982, p. 127). In Table 7.4, the values of $\bar{D}$ are given in the row headed "Mean (m)" and the columns headed "X1-X2" and "X1-X3". By the Central Limit Theorem (Howell, 1982, p. 117), it can be shown that

$$s_{\bar{D}} = \frac{s_D}{\sqrt{N}} \tag{7.2}$$

where N is, here, the number of chunks in the partition and s_D is the standard deviation in the chunk note accuracy differences. In Table 7.4, the values of s_D are given in the row headed "Standard deviation (s)" and the columns headed "X1-X2" and "X1-X3". From Equations 7.1 and 7.2, it follows that, for the partition G_8,

$$t = \frac{\bar{D}\sqrt{N}}{s_D}. \tag{7.3}$$

When there are 8 chunks, there are 7 degrees of freedom and it can readily be confirmed that the least value of t for which the difference is significant at the 0.05 level is 2.36462. Therefore, in order for the mean chunk note accuracy difference to be significant, it follows from Equation 7.3 that

$$\frac{\bar{D}\sqrt{8}}{s_D} \geq 2.36462 \tag{7.4}$$

which implies that

$$\frac{\bar{D}}{s_D} \geq 0.836. \tag{7.5}$$

That is, the ratio of the mean chunk note accuracy difference to the standard deviation of the chunk note accuracy differences must be greater than 0.836 for the difference to be deemed significant at the 0.05 level. As can be seen in Table 7.4, for the statistically insignificant difference between PS13s1 and LH1V, this ratio is only 0.63 because of the large value of s_D. It can also be seen that this large value of s_D is primarily due to the large standard deviation in the chunk note accuracies achieved by LH1V (1.77). On the other hand, for the highly statistically significant difference between PS13s1 and PS13, the ratio $\frac{\bar{D}}{s_D}$ is 2.92 which is well above the

0.836 threshold for significance calculated above.

7.4 Testing the best algorithms on temporally noisy data

Several claims were made in section 7.2 concerning the robustness of the algorithms to the types of temporal deviation that typically occur in data derived from performances. Specifically, it was suggested that

1. PS13s1 should be more robust to temporal deviation than the other versions of *ps13* tested;
2. Temperley and Sleator's *Melisma* algorithms would not be very robust to the type of temporal deviations that typically occur in performance-derived data;
3. Cambouropoulos's algorithms would be robust to temporal deviations in performance-derived data; and
4. TPROne should be more robust to performance-derived temporal deviations than MH or MH2P.

These claims will now be substantiated by presenting the results obtained when the best versions of the algorithms tested in this study were run on the 'noisy' corpus, $\mathcal{C}'$, described in section 1.3.7 above. These results are shown in Table 7.5. The rightmost column in this table gives, for each algorithm, A, the proportional difference, $\Delta\mathrm{NER}_{\mathrm{Pr}}(\mathrm{NER}(A,\mathcal{C}),\mathrm{NER}(A,\mathcal{C}'))$, as discussed in section 1.3.7 and defined in Eq. 1.3. A higher positive value of $\Delta\mathrm{NER}_{\mathrm{Pr}}(\mathrm{NER}(A,\mathcal{C}),\mathrm{NER}(A,\mathcal{C}'))$ indicates that the algorithm's spelling accuracy is more strongly reduced by the introduction of noise into the data. Specifically, the values in the rightmost column of Table 7.5 give, for each algorithm, the percentage increase in the number of errors, caused by changing from the 'clean' test corpus, $\mathcal{C}$, to the 'noisy' test corpus, $\mathcal{C}'$.

From Table 7.5 it can be seen that the best versions of the *ps13*-based algorithms performed more accurately on the noisy corpus, $\mathcal{C}'$, than any of the other algorithms considered in this study. In particular, when $\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle$ was set to $\langle 33, 25\rangle$, PS13s1 spelt 99.41% of the notes in the noisy corpus correctly with a style dependence of 0.5. As expected, PS13s1 was more robust to the temporal deviations in $\mathcal{C}'$ than PS1303, whose note error count increased by 23.3%. However, PS13 actually made 5.6% fewer errors on the 'noisy' corpus $\mathcal{C}'$ than it did on the 'clean' corpus $\mathcal{C}$. This improvement in accuracy is probably due to the fact that Stage 2 of PS13 makes fewer "corrections" (and therefore introduces fewer new errors) when there are many temporal deviations in the data than when the data is temporally 'clean'. This is because the CorrectNeighbourNotes, CorrectDownwardPassingNotes and CorrectUpwardPassingNotes functions can only be expected to find neighbour-note and passing-note figures when the note onsets are strictly proportional to their notated values (see section 6.2.2). Note also that the 'real-time' version of PS13s1, in which K_{post} was set to 1, was only slightly affected by the introduction of noise and had the lowest note error count over the noisy corpus of the real-time algorithms. These results therefore support the claims made in section 7.2 that PS13s1 would be robust to the types of temporal noise typically present in data derived from performances.

Algorithm	*Note error counts*									*Note accuracies (%)*										$\Delta\text{NER}_{\text{Pr}}$ (%)
	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	SD_{Sty}	
	24505	24493	24493	24500	24490	24494	24500	24497	195972											
PS13s1($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 33, 25\rangle$)	42	345	11	68	220	282	82	114	1164	99.83	98.59	99.96	99.72	99.10	98.85	99.67	99.53	99.41	0.50	5.3
PS13s1($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$)	39	367	8	64	230	301	83	95	1187	99.84	98.50	99.97	99.74	99.06	98.77	99.66	99.61	99.39	0.54	7.9
PS13 ($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$)	42	389	12	64	244	321	95	109	1276	99.83	98.41	99.95	99.74	99.00	98.69	99.61	99.56	99.35	0.57	-5.6
PS1303($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$)	61	416	14	67	271	340	103	108	1380	99.75	98.30	99.94	99.73	98.89	98.61	99.58	99.56	99.30	0.61	23.3
PS13s1($\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 40, 1\rangle$)	79	413	55	109	286	336	137	222	1637	99.68	98.31	99.78	99.56	98.83	98.63	99.44	99.09	99.16	0.53	2.5
TPR1A	173	323	95	114	441	273	118	107	1644	99.29	98.68	99.61	99.53	98.20	98.89	99.52	99.56	99.16	0.52	-12.9
TPR1C	171	342	95	111	3383 (435)	275	117	109	4603 (1655)	99.30	98.60	99.61	99.55	86.19 (98.22)	98.88	99.52	99.56	97.65 (99.16)	4.65 (0.52)	-4.0 (-10.1)
Cam01A	269	210	135	167	426	150	161	179	1697	98.90	99.14	99.45	99.32	98.26	99.39	99.34	99.27	99.13	0.39	-6.9
CCOP07–12	147	291	143	109	447	277	159	153	1726	99.40	98.81	99.42	99.56	98.17	98.87	99.35	99.38	99.12	0.47	3.6
CamOpt	302	269	142	126	509	167	177	137	1829	98.77	98.90	99.42	99.49	97.92	99.32	99.28	99.44	99.07	0.53	9.2
CCOP01–06	175	517	148	125	369	254	181	166	1935	99.29	97.89	99.40	99.49	98.49	98.96	99.26	99.32	99.01	0.55	16.2
LH1V	1045	334	74	39	1124	270	210	337	3433	95.74	98.64	99.70	99.84	95.41	98.90	99.14	98.62	98.25	1.71	-2.1
LH1	88	1044	75	27	1487	556	406	993	4676	99.64	95.74	99.69	99.89	93.93	97.73	98.34	95.95	97.61	2.21	-51.2
MH2P	338	1223	240	259	3778 (919)	539	347	1172	7896 (5037)	98.62	95.01	99.02	98.94	84.57 (96.25)	97.80	98.58	95.22	95.97 (97.43)	4.88 (1.69)	79.8 (254.0)
MH	430	1914	286	355	3890 (1049)	789	404	1234	9302 (6461)	98.25	92.19	98.83	98.55	84.12 (95.72)	96.78	98.35	94.96	95.25 (96.70)	5.04 (2.31)	112.0 (355.3)
MH2PX2	138	2289	146	132	3581 (684)	2364	179	579	9408 (6511)	99.44	90.65	99.40	99.46	85.38 (97.21)	90.35	99.27	97.64	95.20 (96.68)	5.57 (3.91)	117.5 (377.7)
MHX2	175	2523	171	811	3608 (721)	2536	215	663	10702 (7815)	99.29	89.70	99.30	96.69	85.27 (97.06)	89.65	99.12	97.29	94.54 (96.01)	5.50 (4.05)	145.9 (462.2)
FixedLOFRange ($\ell_{\text{min}} = -2$)	1431	1324	716	761	1511	646	863	1333	8585	94.16	94.59	97.08	96.89	93.83	97.36	96.48	94.56	95.62	1.47	0
BlackNotesSharp	2829	3403	2067	2948	2679	2124	1884	2012	19946	88.46	86.11	91.56	87.97	89.06	91.33	92.31	91.79	89.82	2.24	0
RandomFlatSharp	4215.0	2081.5	3257.5	3570.5	3262.5	2668.0	3531.5	3437.0	26023.5	82.80	91.50	86.70	85.43	86.68	89.11	85.59	85.97	86.72	2.60	0
BlackNotesFlat	5601	760	4448	4193	3846	3212	5179	4862	32101	77.14	96.90	81.84	82.89	84.30	86.89	78.86	80.15	83.62	6.18	0
FixedLOFRange ($\ell_{\text{min}} = 1$)	4207	5988	3673	5070	4055	4155	3135	3131	33414	82.83	75.55	85.00	79.31	83.44	83.04	87.20	87.22	82.95	3.94	0
FixedLOFRange ($\ell_{\text{min}} = -6$)	7752	2375	6820	6030	6065	4848	7685	7527	49102	68.37	90.30	72.16	75.39	75.23	80.21	68.63	69.27	74.94	7.43	0

Table 7.5: Results obtained when the best versions of the algorithms evaluated in Chapters 2 to 6 were run on the noisy test corpus, $\mathcal{C}'$. The algorithms are sorted into ascending order of overall note error count.

Of the algorithms tested, Temperley and Sleator's programs were by far the worst affected by the introduction of temporal noise into the data. When the sudden enharmonic change in the fourth movement of Haydn's 'Military' Symphony was included, the best versions of Temperley and Sleator's algorithm achieved about the same note accuracy over the noisy corpus, $\mathcal{C}'$, as the best baseline algorithm (FIXEDLOFRANGE with $l_{\min} = -2$). When the enharmonic change in the 'Military Symphony' was omitted from the ground truth, changing from the clean corpus, $\mathcal{C}$, to the noisy corpus, $\mathcal{C}'$, increased the note error rate for the best versions of Temperley and Sleator's algorithm by between 3.54 and 5.62 *times* (i.e., by between 254 and 462%), making these algorithms the least accurate of the algorithms tested on the noisy corpus. These results support the prediction made in section 7.2 that the *Melisma* algorithms would not be very robust to the type of temporal deviations that typically occur in performance-derived data.

Of the 12 best versions of Chew and Chen's algorithm, CCOP01–12, those in which only the notes *starting* in each window are considered when calculating CEs (CCOP07–12) achieved a higher note accuracy than those in which the notes *sounding* in each window are considered (CCOP01–06). This suggests that, when using Chew and Chen's algorithm to process data derived from a performance, better results can be obtained by considering only the notes that start in each window when calculating a CE (as proposed in section 5.1.2) than by using the method adopted by Chew and Chen in their own implementation, which involves considering all the notes that sound in each window.

Changing from the clean corpus, $\mathcal{C}$, to the noisy corpus, $\mathcal{C}'$, *reduced* the number of errors made by CAM01A by 6.9%, increasing its overall note accuracy from 99.07% to 99.13% and making it the least dependent on style ($\mathrm{SD}_{\mathrm{Sty}} = 0.39$) of all the algorithms tested on the noisy corpus. However, the note accuracy of CAMOPT was reduced from 99.15% to 99.07% by the introduction of temporal deviations into the data, which, coupled with its slower running time, makes it a less attractive option than CAM01A for processing performance-derived data. Nevertheless, these results support the claim, made in section 7.2, that Cambouropoulos's algorithms would be robust to temporal deviations in performance-derived data.

Changing from the clean corpus, $\mathcal{C}$, to the noisy one, $\mathcal{C}'$, reduced the note error rates of the best versions of the TPRONE algorithm (TPR1A and TPR1C) by over 10%, making them more accurate over $\mathcal{C}'$ than the algorithms of Chew and Chen, Temperley and Sleator, Cambouropoulos and Longuet-Higgins. Notes that are notated as having simultaneous onsets (i.e., 'simultaneities') have simultaneous onsets in $\mathcal{C}$ but not necessarily in $\mathcal{C}'$. Since TPRONE processes the data a 'simultaneity' at a time, the set of notes in $\mathcal{C}'$ that are taken into account when computing the COG for a note typically contains some notes that are notated as starting simultaneously with the note to be spelt. However, in $\mathcal{C}$, notes with the same notated onset time as the note being spelt are not included when computing the COG. Therefore, some of the preceding notes that have the strongest effect on the spelling of a note in $\mathcal{C}'$ have no effect in $\mathcal{C}$. This may be part of the reason why TPRONE performs more accurately on $\mathcal{C}'$ than it does on $\mathcal{C}$. As predicted in section 7.2, therefore, TPRONE is far more robust to performance-derived temporal deviations than the *Melisma* algorithms. Nevertheless, although TPRONE performs relatively well on $\mathcal{C}'$, it should be remembered that it is slower and no easier to implement in practice than PS13S1.

The original and best version of Longuet-Higgins's algorithm (LH1V and LH1) also performed better on the noisy corpus $\mathcal{C}'$ than it did on the clean corpus $\mathcal{C}$. Indeed, when the voices were interleaved, changing from $\mathcal{C}$ to $\mathcal{C}'$ reduced the note error count for this algorithm by over 50%. Nevertheless, even on the noisy corpus, Longuet-Higgins's algorithm was out-performed by the *ps13*-based algorithms, TPRONE, and the algorithms of Cambouropoulos and Chew and Chen.

7.5 Summary and conclusions

In this dissertation I have analysed, evaluated and compared most of the pitch spelling algorithms that have been described in the literature. This has resulted in an up-to-date and detailed picture of the state-of-the-art in this field.

For most of the algorithms considered, I have provided detailed pseudocode that can be translated straightforwardly into working computer programs. In particular, detailed pseudocode has been given for certain algorithms for which only fairly sketchy descriptions have been published previously (e.g., *ps13* and the algorithms of Cambouropoulos and Chew and Chen).

All the algorithms considered were evaluated for spelling accuracy and style dependence by running them on a relatively large and varied test corpus of baroque and classical music, denoted by $\mathcal{C}$. The most accurate versions of the algorithms were then run again on a 'noisy' version of this corpus, denoted by $\mathcal{C}'$, in which the onset times and durations of the notes had been randomly modified so as to simulate the temporal deviations that typically occur in music data derived from human performances. I have also analysed the time and space complexities of most of the algorithms.

Longuet-Higgins's (1987a, pp. 112–114) theory of tonality, which forms the basis of his pitch spelling algorithm, was re-expressed as six rules and it was shown that one of these rules had been incorrectly implemented in Longuet-Higgins's `music.p` program. The implementation of this rule was corrected in a new version of the algorithm, which, however, proved to be less accurate than the original "incorrect" version implemented in `music.p`. It was also shown that removing the restriction in Longuet-Higgins's algorithm that pitch names must be between G𝄫 and A𝄪 on the line of fifths more than doubled the note error count. Longuet-Higgins's (1987a, p. 114) claim that his algorithm was not appropriate for processing polyphonic music was tested by running his algorithm on two versions of the test corpus, $\mathcal{C}$, one in which the voices were interleaved and another in which the voices were arranged end-to-end. The algorithm made roughly half as many errors when the voices were end-to-end as when they were interleaved, supporting Longuet-Higgins's claim. The best version of Longuet-Higgins's algorithm spelt 98.21% of the notes in $\mathcal{C}$ correctly with a style dependence of 1.79, when the voices were arranged end-to-end.

I presented detailed pseudocode for my own implementations of the pitch spelling algorithms described by Cambouropoulos (1996, 1998, 2001, 2003) and analysed the time and space complexities of these algorithms. Certain minor errors in Cambouropoulos's descriptions of his algorithms were identified and corrected. I then identified 29 variable features of Cambouropoulos's algorithms, most of which were implied by the differences between the versions of the algorithm described by Cambouropoulos in his publications. I then attempted to find an optimal combination of values for 18 of these variable features by running 26 versions of Cambouropoulos's algorithm on the test corpus, $\mathcal{C}$. The resulting optimal combination of variable feature values

was then implemented in a new version of Cambouropoulos's algorithm, CamOpt, which made 8% fewer errors over $\mathcal{C}$ than the most accurate of the other versions tested. CamOpt is the same as my implementation of the algorithm described by Cambouropoulos (2001), except that

1. it uses a window containing 12 distinct MIDI note numbers rather than 9;
2. it uses the interval optimization and notational parsimony penalty values used in my implementation of the algorithm described by Cambouropoulos (2003); and
3. the modality class of each interval is determined in the same way as in my implementation of the algorithm described by Cambouropoulos (1996, 1998), except that the boundaries of the modality classes are moved so that modality classes A and C cover a wider range of values and B covers a narrower range.

CamOpt spelt 99.15% of the notes in $\mathcal{C}$ correctly with a style dependence of 0.47. However, even with a window size of 12, my implementation of CamOpt was rather slow, taking over 40 hours to process $\mathcal{C}$. The second most accurate version of Cambouropoulos's algorithm tested was my implementation of the algorithm described by Cambouropoulos (2001), which I denote by Cam01A. This algorithm spelt 99.07% of the notes in $\mathcal{C}$ correctly, with a style dependence of 0.46 and took just over 6 hours to process the test corpus.

To determine the pitch names of the notes in a passage using Temperley and Sleator's *Melisma* system, one must either run the `harmony` program on the output of the `meter` program, a process which I denote by MH; or use a 'two-pass' method, which I denote by MH2P and which involves running the `harmony` and `meter` programs twice on the input data. It was found that MH and MH2P performed very similarly over the test corpus $\mathcal{C}$ and that both were very sensitive to tempo. When the music was at its natural tempo or at half speed and the sudden enharmonic change in the fourth movement of Haydn's 'Military' Symphony was omitted, the algorithms spelt 99.27–99.30% of the notes in $\mathcal{C}$ correctly with a style dependence between 1.13 and 1.30. However, when the music was more than twice or less than a quarter of its natural tempo, the accuracy of the algorithms was severely reduced. Also, if the music was at its natural tempo or at half speed and the sudden enharmonic change in the 'Military' Symphony was included in the ground truth, then the note accuracy fell to 97.76–97.79% and the style dependence rose to between 4.41 and 4.61. Results obtained when the `harmony` program was run on $\mathcal{C}$ without using metrical information suggest that, in particular, metrical information helps this program to process certain types of chromatic passages. I also presented a relatively simple implementation of Temperley's (2001, p. 125) TPR 1 which, in its linear-time form with a window containing 1000 notes, spelt 99.06% of the notes in $\mathcal{C}$ correctly when the sudden enharmonic change in the 'Military' Symphony was omitted. A quadratic-time version of this algorithm spelt 99.04% of the notes in $\mathcal{C}$ correctly when the sudden enharmonic change was included. These results support Temperley's (2001, p. 125) claim that his TPR 1 is "in many cases... sufficient to ensure the correct spelling of passages".

I presented pseudocode for my own implementation of Chew and Chen's pitch spelling algorithm and ran it on the test corpus, $\mathcal{C}$, 1296 times, each time with a different combination of parameter values. I showed that this algorithm works best when the local context window is small (2 chunks), the global context window is moderate in size (8 chunks), the local and

cumulative CEs are given equal weighting and the chunks themselves are small (500ms), leading to frequent updating of the CEs. With these parameter values, the algorithm spelt 99.15% of the notes in $\mathcal{C}$ correctly with a style dependence of 0.35–0.42, making it the least dependent on style of all the algorithms tested in this study. These results seem to contradict Chew and Chen's (2005, p. 74) claim that the local context is more important than the cumulative context. I also proposed an alternative method of calculating the center of effect to that used by Chew and Chen, in which only the notes *starting* in each window are considered, rather than the notes *sounding* in each window. My evaluation showed that, for the most accurate versions of the algorithm tested, considering only the notes starting in each window instead of the notes sounding in each window made no difference to the note accuracy but slightly increased the style dependence from 0.35 to 0.42. I have also proved that the spelling generated by Chew and Chen's algorithm when the line of fifths is used is always exactly the same as that generated when the spiral array is used, regardless of the aspect ratio of the spiral array (see Appendix C). It follows that changing the aspect ratio of the spiral array also makes no difference to the output generated by the algorithm.

My *ps13* algorithm is based on the assumption, which seems to have been accepted by most experts in this field, that the pitch name of a note depends on the local key and voice-leading, with the local key being the more important of the two factors. I presented detailed pseudocode for an implementation of *ps13* called PS13 and analysed its time and space complexities.

In Stage 1 of *ps13*, the sense of local key is represented by the frequency distribution of chromas within a context surrounding the note to be spelt. This contrasts with the way that key is represented as a single 'average' point on the line of fifths (or spiral array) in the algorithms of Temperley, Longuet-Higgins and Chew and Chen. In *ps13*, the frequency with which a chroma occurs within the context surrounding a point in the music is used as a measure of the likelihood of that chroma being the chroma of the tonic at that point. It is then assumed that the pitch name implied for a note by a particular tonic is the one that has the same chroma as the note and lies closest to the tonic on the line of fifths. The strength with which a particular pitch name is implied for a note is then taken to be the sum of the frequencies of occurrence, within a context surrounding the note, of the tonic chromas that imply that pitch name. In effect, the contribution that each possible tonic makes to determining the pitch name of a note is proportional to the strength with which that tonic is implied by the context surrounding the note.

Stage 2 of *ps13* attempts to correct certain passing-note and neighbour-note errors in the output of Stage 1.

In Stage 1 of *ps13*, the size of the context surrounding each note over which the chroma frequency distribution is calculated is defined by two parameters, K_{pre} and K_{post}, which specify the number of notes preceding and following the note to be spelt that are to be included in the context. An attempt was made to find optimal values for K_{pre} and K_{post} and it was found that PS13 performed best when $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was set to $\langle 10, 42 \rangle$. With these settings, PS13 spelt 99.31% of the notes in $\mathcal{C}$ correctly with a style dependence of 0.57. PS13 was then improved by adding a post-processing phase in which each of the pitch names computed by PS13 is transposed up or down by a diminished second if this brings it closer on the line of fifths to the

pitch names of the notes around it in the music. When $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was set to $\langle 10, 42 \rangle$, this new algorithm, which I call PS1303, achieved a spelling accuracy of 99.43% over $\mathcal{C}$ with a style dependence of 0.53.

I showed that removing Stage 2 altogether from PS13, leaving just Stage 1, would improve the time complexity of the algorithm, make it more robust to the type of temporal deviations that occur in performance-derived data and make it easier to implement. A new version was therefore defined, called PS13s1, consisting of just Stage 1 of PS13. When PS13s1 was run on $\mathcal{C}$ with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to either $\langle 10, 42 \rangle$ or $\langle 33, 25 \rangle$, it achieved a note accuracy of 99.44% with a style dependence of 0.49 when $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was $\langle 10, 42 \rangle$ and 0.45 when $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ was $\langle 33, 25 \rangle$. Indeed, when PS13s1 was run on $\mathcal{C}$ with various different values of $\langle K_{\text{pre}}, K_{\text{post}} \rangle$, it was found that it generally made 15–19% fewer errors than PS13 and that its style dependence was generally 12–18% lower than that of PS13. PS13s1 was therefore found to be superior to all the other versions of *ps13* tested in terms of note accuracy, style dependence, time complexity, tolerance to temporal deviation and ease of implementation. Also, the fact that PS13s1 performed better than PS13 seems to support the view that the local sense of key is more important than voice-leading when determining pitch names.

Over $\mathcal{C}$, the *ps13*-based algorithms were the most accurate of the algorithms tested, followed by Temperley and Sleator's *Melisma* programs, then the algorithms of Chew and Chen and Cambouropoulos, then TPROne, and finally Longuet-Higgins's algorithm. All the best versions of the algorithms were more accurate than the best baseline algorithm (FixedLOFRange with $l_{\text{min}} = -2$), apart from Longuet-Higgins's algorithm when it was run with the voices interleaved. The most accurate algorithm over $\mathcal{C}$ was PS13s1 with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to either $\langle 10, 42 \rangle$ or $\langle 33, 25 \rangle$. The *ps13*-based algorithms were also the most accurate over the 'noisy' corpus, $\mathcal{C}'$: with $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ set to $\langle 33, 25 \rangle$, PS13s1 spelt 99.41% of the notes in $\mathcal{C}'$ correctly with a style dependence of 0.50.

Temperley and Sleator's programs were only slightly less accurate over $\mathcal{C}$ than the *ps13*-based algorithms under certain conditions. However, when they were run on $\mathcal{C}'$, the most accurate version of Temperley and Sleator's system made over 4 times as many errors as the best version of PS13s1. Also, MH and MH2P

- were the most sensitive to tempo;
- were the most complicated algorithms tested and therefore the hardest to implement;
- were the worst affected of all the algorithms by the introduction of temporal deviations into the data;
- deleted and arbitrarily re-ordered notes in the input data, thereby losing information;
- were relatively inaccurate when required to detect sudden enharmonic changes; and
- were considerably more dependent on style than the *ps13*-based algorithms.

The best versions of Chew and Chen's algorithms were relatively fast, easy to implement and well-suited to being used in real-time applications. However, they made more than 50% more errors over $\mathcal{C}$ than the best version of PS13s1. When all the notes sounding in each window were

used to compute the CEs, as in Chew and Chen's implementation, the best versions of Chew and Chen's algorithm made 18% more errors over the 'noisy' test corpus $\mathcal{C}'$ than a real-time version of PS13s1 in which $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ was set to $\langle 40, 1\rangle$ (and 66% more errors than PS13s1 with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 33, 25\rangle$). Chew and Chen's algorithm is also no easier to implement than PS13s1.

I suggested that the fact that PS13s1 is more accurate than the superficially similar TPROne and the algorithms of Chew and Chen and Temperley and Sleator may be partly due to the differences in the way that these algorithms model the sense of key. In *ps13*, every tonic contributes to the pitch name in proportion to its strength of implication and the notes are spelt so that they are close to the local tonic on the line of fifths. However, in the other similar algorithms, the notes are spelt so that they are as close as possible on the line of fifths not to the local tonic but to a moving average of the preceding pitch names.

The best versions of Cambouropoulos's algorithms achieved high note accuracies with low style dependences over both $\mathcal{C}$ and $\mathcal{C}'$. However, even the best versions of this algorithm made over 50% more errors than the best version of PS13s1 on $\mathcal{C}$ and 46% more errors over $\mathcal{C}'$. Also, because the running times of Cambouropoulos's algorithms are exponential in the size of the window used, they were much slower in practice than PS13s1.

TPROne performed surprisingly well, considering how simple it is, proving to be much more robust to temporal deviations and less dependent on style than Temperley and Sleator's full-blown implementation of Temperley's theory. Indeed, on the 'noisy' test corpus, $\mathcal{C}'$, TPROne was more accurate than all the algorithms tested apart from those based on *ps13*. However, the quadratic-time version of TPROne is impractically slow for processing very large movements and even the best versions of TPROne made 67% more errors over $\mathcal{C}$ (and 41% more errors over $\mathcal{C}'$) than PS13s1, which was faster, no harder to implement and less dependent on style.

Longuet-Higgins's algorithm was less accurate than the more modern algorithms over the 'clean' test corpus $\mathcal{C}$. Indeed, when the voices were interleaved, its note accuracy was below that of the best baseline algorithm (FixedLOFRange with $l_{\text{min}} = -2$). However, on the noisy corpus, $\mathcal{C}'$, it performed better, achieving a higher note accuracy than the best version of Temperley and Sleator's algorithm.

None of the attempts to take voice-leading into account in the algorithms considered in this study resulted in an increase in note accuracy and the most accurate algorithm, PS13s1, ignores voice-leading altogether. This suggests that correctly modelling the sense of key and the effect that key has on pitch-names is much more important in pitch-spelling than taking voice-leading considerations into account.

It was noted that, unlike any of the other algorithms tested in this study, the best versions of PS13 and PS13s1 employ a substantial post-context containing between 23 and 42 notes *following* the one to be spelt. This context is used to compute the frequency of occurrence of each chroma which is then used as an indicator of how likely that chroma is to be the tonic. The fact that the most accurate algorithms tested used a substantial post-context therefore suggests that the notes occurring up to a few seconds *after* a time point can influence the interpretation and therefore the spelling of notes occurring at that time point. In particular, it suggests that whether or not a pitch is perceived to be the local tonic at a particular time point may depend

not only on the music that immediately precedes that time point but also on the music that immediately follows it. This result is not really surprising since the sizes of the contexts used in the best versions of the *ps13*-based algorithms were shown to be between 5 and 6 seconds which corresponds remarkably well with estimates of the upper limit on the duration of the perceptual present; and, clearly, it is particularly easy for events occurring early during the perceptual present to be retrospectively re-interpreted in the light of events occurring later on during this period.

In conclusion, on the evidence provided in this study, it is hard to imagine any situation in which PS13s1 would not be the best of the currently available algorithms to choose for determining the pitch names in a passage of tonal music. It has been shown to be more accurate than the other algorithms considered here over both 'clean' score-derived music representations and 'noisy' data containing temporal deviations of the type that one might expect to be present in performance-derived representations. Moreover, by setting its K_{post} parameter to 1, it can be implemented as a real-time algorithm which has been shown to be more accurate than the other real-time methods considered in this study. It is also at least as fast and easy to implement as any of the algorithms considered here and its style dependence is very low (around 0.5). Also, it does not use note durations so it can be used when only the onset time and MIDI note number of each note are available.

7.6 Suggestions for further work

Both *ps13* and Longuet-Higgins's algorithm are based on the idea that notes should be spelt so that they are as close as possible to the local tonic on the line of fifths. In Temperley and Sleator's `harmony` program and the TPROne algorithm, notes are spelt so that they are as close as possible on the line of fifths to the 'center of gravity' of the notes in a window preceding the note to be spelt. In Chew and Chen's algorithm, notes are spelt so that they are as close as possible on either the line of fifths or the spiral array (which is just the line of fifths coiled up) to the 'center of effect' of the notes in various windows that precede the note to be spelt. In Cambouropoulos's algorithm, notes are spelt so that the intervals between them are either ones that occur commonly in the major and minor scales or ones that correspond to short distances along the line of fifths. Also, in Cambouropoulos's method, a pitch name is penalised if it is a double-sharp or a double-flat—i.e., if it is more than 10 steps away from `"Dn"` on the line of fifths. It can therefore be seen that all the algorithms considered in this study use the line of fifths in one way or another.

Would it be possible to improve on existing algorithms by using some pitch space other than the line of fifths? In Chew and Chen's algorithm, changing from the line of fifths to the spiral array was shown to make no difference to performance (see Appendix C). In Cambouropoulos's algorithm, changing from a modality-based method of calculating the interval optimisation penalties to one based on the line of fifths generally slightly reduced the note error rate when augmented and diminished seconds were 'demoted', but slightly increased it when they were not (see row 4 of Table 3.16). Also, the best-performing algorithm in this study, PS13s1, uses the line of fifths. These results do not suggest that much is to be gained by using a pitch space other than the line of fifths.

Furthermore, it seems likely that most of the pitch spaces that have been proposed in the literature for accounting for the perception and cognition of tonal structure would give essentially the same results as the line of fifths when employed in a pitch spelling algorithm. As Temperley (2001, p. 117) observes, all the models that he considers in his survey of spatial models for representing tonal relations have "one feature in common", namely, "an axis of fifths, such that adjacent elements are a fifth apart". Nevertheless, I think it would be interesting and worthwhile to carry out a study to test rigorously whether or not using spaces other than the line of fifths in pitch spelling algorithms leads to an improvement in performance.

None of the algorithms considered in this study are machine-learning algorithms. However, it seems plausible that machine-learning techniques could be used to perform pitch spelling effectively. Indeed, Stoddard et al. (2004) have described a pitch-spelling algorithm that uses machine-learning techniques which achieved a note accuracy of 99.69% over a small test corpus of eighteenth and nineteenth century music. Unfortunately, Stoddard et al.'s (2004) method requires metrical information and only works on 'clean' input data in which the onsets and durations are strictly proportional to their notated values. It also needs to carry out a complete harmonic parse of the data as a pre-processing step, using Raphael and Stoddard's (2003) harmonic analysis system. The system was not considered here because the authors were able to provide me with neither a working system nor a description of it that was sufficiently detailed for me to implement it myself.

As pointed out above, none of the attempts to take voice-leading into account in the algorithms considered in this study resulted in an increase in note accuracy. However, it would be interesting and worthwhile to carry out a study to explore the possibility of improving the spelling accuracy of existing algorithms by using a more sophisticated mechanism for modelling the effect of voice-leading on pitch spelling than the rather crude methods used in the algorithms considered here.

PS13S1 determines pitch names by modelling the sense of local key at each point in a passage of music. The success with which PS13S1 computes pitch names suggests that it might also be successful as the basis of a key-tracking algorithm—that is, an algorithm that can determine the perceived key at each point in a passage of tonal music. Also, if a successful key-tracking algorithm can be developed, then this could form the basis of a harmonic analysis algorithm. It would be interesting to develop a key-tracking algorithm and a harmonic analysis algorithm based on PS13S1 and compare these algorithms with other such algorithms that have been proposed in the literature (e.g., Holtzmann, 1977; Krumhansl, 1990; Longuet-Higgins and Steedman, 1987; Maxwell, 1992; Raphael and Stoddard, 2003; Temperley, 1997, 1999, 2001; Temperley and Sleator, 1999; Vos and van Geenen, 1996; Winograd, 1968, 1993). Unfortunately, evaluating key-finding and harmonic analysis algorithms is more problematic than evaluating pitch-spelling algorithms, because expert listeners often do not agree about the perceived key and harmony at each point in a passage of music, whereas they *do* agree in the vast majority of cases on how a note should be spelt in a tonal context. Several difficult methodological problems therefore need to be solved, including the building of large and representative collections of expert key and harmonic analyses, before key-tracking and harmonic analysis algorithms can be evaluated and compared in a meaningful way.

Appendix A

The effect on algorithm performance of removing from the input data duplicate notes with the same onset and pitch

A.1 Introduction

Let's suppose that:

1. D and D' are two OPNDV datasets;
2. D has been derived from a Musedata encoding and therefore may contain duplicate notes with the same onset and pitch; and
3. D' is the result of removing such duplicate notes from D as described in section 1.3.3.3.

We therefore know that $D' \subseteq D$. In all the algorithms considered in this study, the pitch name of a note depends in some way on the properties of the notes near it in the music. Most of the algorithms will therefore be affected by removing duplicate notes with the same onset and pitch from the input data. However, the precise effect that this has on an algorithm's performance depends in detail on the way that the algorithm uses the context around a note to determine the note's pitch name, as will now be discussed.

A.2 The effect on *ps13*

In my *ps13* algorithm (see Chapter 6), the context upon which the pitch name of a note depends is defined to contain the note to be spelt along with a number of notes immediately preceding it and a number of notes immediately following it in a list generated by sorting the datapoints in the input OPNDV dataset by onset and then sounding pitch. Removing duplicate notes with the same onset and pitch could have a number of consequences on the performance of *ps13*, as will now be explained.

First, the set of OPNDV datapoints on which the pitch name of a note depends in D' will not always be the same as the set of OPNDV datapoints on which the pitch name of the same note depends in D. Therefore, the pitch name assigned by *ps13* to a note in D' will not necessarily be the same as the pitch name it assigns to the same note in D.

Second, if S is a set of simultaneously-starting notes with the same pitch in D, then all the notes in S will not necessarily be assigned the same pitch name because the contexts used to spell the notes in S will typically not all be the same. This could lead to simultaneously-starting notes with the same sounding pitch being assigned different pitch names—a situation that, to my knowledge, never occurs in tonal music.

Third, let S be a set of simultaneously-starting notes in D with the same pitch and let N be the longest note in S. The only member of S that is also in D' will therefore be N. In *ps13*, the context on which the pitch name of N depends in D will contain a higher proportion of notes with the same sounding pitch as N than the context on which N depends in D'. This will cause the pitch class of N to be considered to be more likely to be the local tonic in D than in D'. This, in turn, means that it is more likely in D than in D' for *ps13* to assign to N the pitch name implied by N being the tonic. In other words, leaving multiple, simultaneously-starting notes with the same pitch in the input data would cause *ps13* to consider such notes to be more likely to be the tonic. Typically, the notes in S will all be in different voices or parts, so leaving such duplicate notes in data to be processed by *ps13* would only be justified if we believe that the number of parts or voices in which a particular pitch simultaneously occurs is positively correlated with the degree to which the note is perceived to be consonant, stable or tonic-like. In orchestral and chamber music, it is not uncommon for several instruments to play the same line for sections of the music (Piston, 1978, p. 284). This is typically done either to emphasise a particular voice or line in the music, such as a melodic line or a bass line, or to cause it to have a particular timbre. In such cases, it is clearly not just the most tonally stable notes in a key that are doubled at the unison. However, notes are also doubled at the unison when there are more parts or instruments than there are notes in a chord; and, according to Piston (1978, p. 67), in traditional four-part harmony, "tones are doubled which are important to solidity of the key". Specifically, in triads in the root position, it is usually the root which is doubled. In first-inversion triads, the bass is doubled if it is a tonal degree and some tonal degree of the triad is doubled if the bass is not a tonal degree (Piston, 1978, p. 67). There may therefore be some grounds for believing that notes doubled at the unison for harmonic purposes tend to be tonally more stable. Nevertheless, when S contains more than two or three notes, this will typically be due to instrumental doubling of voices rather than doubling due to harmonic reasons and, in such cases, I see no justification for assuming that the number of parts playing a note is positively correlated with its tonal stability.

On balance, therefore, one might expect *ps13* to perform better on data in which duplicate simultaneously-starting notes with the same pitch had been removed. However, this would have to be confirmed by running the algorithm on a version of the test corpus in which the duplicate notes had been left in and comparing the results with those reported in this study.

A.3 The effect on Longuet-Higgins's algorithm

In Longuet-Higgins's algorithm (see Chapter 2), the pitch name assigned to a note depends on a context of three or four consecutive notes including the one to be spelt. In this study, Longuet-Higgins's algorithm was run on two versions of the test corpus, $\mathcal{C}$, one in which the notes were sorted by voice and then onset time; and a second version in which the notes were sorted by onset time and then pitch. That is, in one version, the algorithm processes the notes "a voice at a time"; and, in the other version, the algorithm processes the notes (approximately) "a chord at a time" (see section 2.4.2.3).

Removing duplicate notes with the same onset and pitch will mean that some voices will have notes missing which could adversely affect the performance of the algorithm when processing the notes a voice at a time. To confirm this, one would have to run the algorithm on a version of $\mathcal{C}$ in which no notes had been removed and compare the results with those reported in this study.

When the notes are sorted by onset and then pitch, each set of simultaneously-starting notes with the same pitch will occur as a contiguous sequence of OPNDV datapoints in the input data file in $\mathcal{C}$. However, Longuet-Higgins's algorithm incorporates a rule which states that, if two consecutive notes have the same pitch class, then the second note should be ignored when determining the pitch name and should be assigned the same pitch name as the first (see Longuet-Higgins's Rule 5 in section 2.2) (Longuet-Higgins, 1987a, p. 114). Therefore, if such a "doubled" note is assigned an incorrect pitch name by Longuet-Higgins's algorithm, all duplicates of the note will also be assigned the same incorrect pitch name. Similarly, if such a note is correctly spelt, then all duplicates of the note with the same pitch and onset will be correctly spelt. Therefore, when the notes in each movement are sorted by onset and then pitch, leaving in duplicate notes with the same pitch and onset will cause such "doubled" notes to be weighted more heavily in the calculation of note accuracy. It is hard to see how one could justify such heavier weighting of doubled notes since all instances of such a note are automatically assigned the same name and each doubled note is typically heard as being a single note even if it is played by several instruments.

Notes that have equal onset times in $\mathcal{C}$ will not necessarily have equal onset times in the "noisy" test corpus, $\mathcal{C}'$, (see section 1.3.7). Therefore, when processing $\mathcal{C}'$, even when the notes are sorted by onset first and then pitch, Longuet-Higgins's algorithm might not ignore all but one note in each set of unison notes that start simultaneously in $\mathcal{C}$. It is therefore possible that Longuet-Higgins's algorithm would have generated slightly different results on $\mathcal{C}'$ if the duplicate simultaneously-starting unison notes had not been removed from the data.

A.4 The effect on Cambouropoulos's algorithm

Cambouropoulos's algorithm (see Chapter 3) processes a sequence of MIDI note numbers using a "shifting overlapping windowing technique" (Cambouropoulos, 2003, p. 420) in which pitch names are predicted for the MIDI note numbers in the middle third of each window based primarily on the intervals between the pitches in the window. Typically, each window contains 9 or 12 notes. In the versions tested in this study, the notes in the input sequence are sorted

either by onset and then pitch (giving a representation in which the voices are approximately "interleaved") or by voice and then onset (giving a representation in which the voices are "end-to-end"). In the algorithms described by Cambouropoulos (1996, 1998, 2003), each window contains a fixed number of consecutive MIDI note numbers in the input sequence (see sections 3.2 and 3.4). However, the algorithm described by Cambouropoulos (2001) uses a "variable length" windowing method (Cambouropoulos, 2001, p. 5) in which each window is a segment of the input sequence that contains a fixed number of *distinct* MIDI note numbers or pitch classes. The latter algorithm therefore ignores duplicate notes with either the same MIDI note number or the same pitch class in the context upon which the pitch name of a note depends. Cambouropoulos (2001, p. 5) explains that he adopted such a variable-length window in order to "avoid unnecessary ambiguity that is introduced when too few different pitches appear in a given window".

The effect that removing such duplicate notes has on the results obtained using Cambouropoulos's algorithm depends on whether the algorithm uses a fixed-length window (as in the versions described by Cambouropoulos (1996, 1998, 2003)) or a variable-length window (as in the version described by Cambouropoulos (2001)). The effect also depends on whether the input data is sorted so that the voices are end-to-end or interleaved.

In Cambouropoulos's algorithm, the best spelling for a window is determined primarily on the basis of the intervals between the notes in the window. When a fixed-length window is used and the voices are interleaved, leaving in doubled notes will tend to cause a greater proportion of the intervals within a window to be intervals with instances of the doubled note, which will effectively increase the influence which that note has on the spelling of notes within the window. Leaving duplicate notes with the same onset and pitch (i.e., doubled notes) in the data will also tend to cause the windows around notes near such doubled notes to span shorter time periods in the music. There is no obvious reason why each instance of a doubled note should be treated as a separate note when determining the pitch names of notes near such doubled notes. Doing this would make it possible in principle for two instances of the same doubled note to be assigned different pitch names, which never seems to happen in tonal music. Also, when heard, a set of simultaneously-starting notes with the same pitch played on different instruments sounds like a single note with a particular timbre. This suggests that, when a fixed-length window is used and the voices are interleaved, each doubled note should be treated as a single note by removing duplicate notes with the same onset and pitch.

However, when the voices in the input data are end-to-end, removing duplicate notes with the same onset and pitch will cause some voices to have no instances of certain doubled notes, which will cause the context windows around certain notes to be incorrect. Removing such duplicates might therefore adversely affect the performance of those versions of Cambouropoulos's algorithm in which the music is processed a voice at a time.

When a variable-length window is used and voices are interleaved, Cambouropoulos's algorithm will itself ignore multiple occurrences of notes with the same onset and pitch when calculating the best spelling for a window. All instances of a doubled note will then be assigned the same pitch name automatically. As already discussed in relation to Longuet-Higgins's algorithm (see section A.3), this effectively weights doubled notes more heavily in the calculation of note accuracy and there do not seem to be any firm psychological or theoretical grounds for

doing this.

A.5 The effect on Temperley and Sleator's algorithm

As discussed in section 1.3.3.3, the *Melisma* programs used to evaluate Temperley and Sleator's algorithm in this study (see Chapter 4) were incapable of processing data containing duplicate notes with the same onset and pitch. Such duplicates were therefore removed from the data in order to ensure that exactly the same test corpus data could be used with all versions of all algorithms tested in this study. If duplicate simultaneously-starting notes with the same pitch had been left in the test corpus data, it would have been necessary to remove such notes before running the data through the *Melisma* programs and then restore them in a post-processing step, before calculating note accuracies, ensuring that all instances of a given doubled note are assigned the same pitch name. As already discussed, this would cause doubled notes to be weighted more heavily in the calculation of note accuracy and there do not seem to be firm psychological or theoretical grounds for applying such weighting.

A.6 The effect on Chew and Chen's algorithm

The basic idea underlying Chew and Chen's algorithm (see Chapter 5) is that each note should be spelt so that it is as close as possible within a geometric model of tonal pitch relations called the "spiral array" (Chew, 2000) to an estimate of the local tonic called the "center of effect" (Chew and Chen, 2005, p. 67) (see section 5.1.2). A center of effect is the weighted centroid of the position vectors within the spiral array of the pitch names of notes in a time window preceding the note to be spelt, each note being weighted by its duration.

As defined above, let us suppose that D is an OPNDV dataset, S is a set of simultaneously-starting notes in D with the same pitch and N is one of the longest notes in S. Let's further suppose that D' is the OPNDV dataset that results from removing from D duplicate notes with the same onset and pitch in the manner described in section 1.3.3.3. In Chew and Chen's algorithm, each context window on which the pitch name of a note depends is defined to span some specified time period. Let $W(T, D)$ and $W(T, D')$ be the context windows in D and D', respectively, that span the time period T and let's suppose that T contains the onset of the notes in S. $W(T, D)$ will contain all the notes in $W(T, D')$. $W(T, D)$ will also contain all the notes in S, whereas $W(T, D')$ will contain one of the longest notes in S but none of the other notes in S. In $W(T, D)$, each note in S will contribute to the calculation of the center of effect for the time span T in proportion to its duration, despite the fact that the combined effect of all the notes in S will be that of a single note with a possibly changing timbre which is as long as the longest note in S. Leaving duplicate notes with the same onset and pitch in the data will therefore cause such doubled notes to have greater influence on determining the center of effect in the context windows in which they occur. As discussed above, there do not seem to be firm psychological or theoretical grounds for allowing a doubled note (i.e., one that is played on more than one instrument or that appears in more than one part) to have a greater influence on the spelling of notes in its vicinity than one that is not doubled. For example, we cannot assume that a doubled note is, in general, louder, more salient or more stable than one that is

not doubled. It would therefore seem reasonable to remove simultaneously-starting notes with the same pitch from the input data before processing it with Chew and Chen's algorithm.

A.7 Conclusions

Duplicate, simultaneously-starting notes with the same pitch were removed from the test corpus encodings in order to ensure that exactly the same test corpus data could be used with all versions of all algorithms tested in this study. However, the foregoing discussion suggests that not all versions of the algorithms studied here would be affected in the same way by removing duplicate notes in this way. On the one hand, there do not seem to be any firm psychological or theoretical grounds for allowing each member of a set of simultaneous notes with the same pitch to have the same influence on the spelling of neighbouring notes as a non-doubled note. On the other hand, when the voices are arranged end-to-end, removing duplicate instances of a doubled note removes the note from all but one of the voices or parts in which it occurs, which will result in notes being omitted from certain voices. This, in turn, will cause certain notes to have incorrect contexts. This suggests that it might have been wiser to leave such doubled notes in the test corpus data and add parameters to each algorithm implementation that would allow the user to decide whether or not duplicate simultaneously-starting notes with the same pitch should be taken into account when determining pitch names and/or calculating note accuracies.

Appendix B

Instances where a movement is spelt in a different key from the original

As explained in section 1.3.4.2, it is possible for the spelling of a movement generated by a particular algorithm to be in a key that is different from (but enharmonically equivalent to) that in which the movement was originally notated. For example, an algorithm might spell a movement in G♯ minor when the original is spelt in A♭ minor.

Consequently, in this study, for every movement and every algorithm, three spellings were generated:

1. the spelling s generated directly by the algorithm;
2. another spelling generated by transposing s up a diminished second; and
3. a third spelling generated by transposing s down a diminished second.

The note error counts were then calculated for all three of these spellings and the spelling with the smallest number of errors was defined to be the spelling generated by the algorithm for that movement for the purpose of the results reported here.

However, it is possible in general for the spelling generated by an algorithm for a movement to be in a key that is neither a perfect prime nor a diminished second away from the original spelling. For example, the spelling generated by an algorithm for a movement could be a falling triply diminished third or a rising quintuply diminished fourth away from the original spelling. In such cases, the procedure just described, which only allows for a generated spelling to be a perfect prime or diminished second away from the original spelling, would produce an unfairly high note error count.

In general, let's suppose that an algorithm A generates a spelling s for a movement m, such that i is the pitch interval by which s needs to be transposed in order to give the least possible number of errors. In general, i could be any interval whose associated chromatic pitch interval is 0—that is, i could be `"p1"`, `"rd2"`, `"fd2"`, `"rddd3"`, `"fddd3"`, `"rddddd4"`, `"fddddd4"`, `"rddddddd5"`, `"fddddddd5"` and so on.[1] Therefore, in order to ensure that no unfairly high note error counts have been given in this study, it would be necessary to check that

[1] See section 1.4.5 for definitions of the terminology and notation used here for representing pitch and pitch interval information.

$i \in \{\texttt{"p1"}, \texttt{"rd2"}, \texttt{"fd2"}\}$ for the spelling s generated by each algorithm A for each movement m in $\mathcal{C}$ and $\mathcal{C}'$. Unfortunately, the spellings generated by the ChewChen algorithm over $\mathcal{C}$ were only saved for the 12 best parameter-value combinations, CCOP01-12, given in Table 5.5. To confirm that i is always in $\{\texttt{"p1"}, \texttt{"rd2"}, \texttt{"fd2"}\}$ for each of the other 1284 parameter value combinations tested with this algorithm would therefore involve re-running almost the complete ChewChen experiment, which has not been done. Similarly, the 2500 spellings generated by running PS13 on $\mathcal{C}_{\text{Samp}}$ with different values of K_{pre} and K_{post} were not saved. Confirming that i is always in $\{\texttt{"p1"}, \texttt{"rd2"}, \texttt{"fd2"}\}$ for each movement in $\mathcal{C}_{\text{Samp}}$ and each of these 2500 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ pairs would therefore involve re-running PS13 on $\mathcal{C}_{\text{Samp}}$ 2500 times, which, again, has not been done.

However, it was possible within the time available to determine the value of i for every movement m in $\mathcal{C}$ for:

1. the 12 tested versions of Longuet-Higgins's algorithm in Table 2.2 on page 83;
2. the 26 tested versions of Cambouropoulos's algorithm in Table 3.11 on page 171;
3. the CamOpt algorithm for which the note error counts are given in Table 3.19 on page 189;
4. the 12 tested versions of Temperley and Sleator's *Melisma* programs for which the note error counts are given in Table 4.2 on page 203;
5. the 'Harmony-No-Meter' version of Temperley and Sleator's system (see Table 4.4 on page 209);
6. the 4 tested versions of the TPROne algorithm in Table 4.6 on page 220;
7. the ChewChen algorithm with the 12 best parameter-value combinations given in Table 5.5 on page 249;
8. the PS13 algorithm with the 17 best $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ values given in Table 6.5 on page 303;
9. the PS13B algorithm with $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$ (see Table 6.7 on page 306);
10. the PS1303 algorithm with $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$ (see Table 6.8 on 310);
11. the PS13s103 algorithm with $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 10, 42\rangle$ (see Table 6.9 on page 311);
12. the PS13s1 algorithm with the 17 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ values given in Table 6.10 on page 312; and
13. the PS13s1 algorithm with $\langle K_{\text{pre}}, K_{\text{post}}\rangle = \langle 40, 1\rangle$ (see Table 7.2 on page 319).

The value of i was also determined for every movement in $\mathcal{C}'$ for the best versions of the algorithms tested, as given in Table 7.5 on page 333.

It was found that i was in the set $\{\texttt{"p1"}, \texttt{"rd2"}, \texttt{"fd2"}\}$ in all cases except the following.

1. When LHPitchSpell$_{4\text{–}6}$ (see Figures 2.24, 2.25 and 2.26) were run on the second movement of Haydn's Symphony No. 103 in E♭ major ('Drumroll') (Hob. I:103) (haydndoversyms-10302m, see Table 1.4) with the voices interleaved, i was actually

`"fddddddd5"`, not `"fd2"`, and the note error count for this movement should have been 2220, not 3943.

2. When LHPitchSpell$_{4–6}$ (see Figures 2.24, 2.25 and 2.26) were run on the first movement of Mozart's Symphony No. 38 in D major ('Prague') (K 504) (mozartbhsymk50401m, see Table 1.4) with the voices interleaved, i was actually `"fddddd4"`, not `"fd2"`, and the note error count for this movement should have been 3953, not 5961.

3. When Temperley and Sleator's `met-harm` script (see section 4.7.1) was run on the version of $\mathcal{C}$ at one sixth of the natural tempo (i.e., MHX6), the value of i for the second movement from Bach's Church Cantata, "Ich geh und suche mit Verlangen" (BWV 49), (bachbgcant004902m, see Table 1.4), was actually `"fddd3"`, not `"fd2"`, and the note error count should have been 258, not 1730.

4. When Temperley and Sleator's `met-harm` and `met-harm-two-pass` scripts (see section 4.7.1) were run on the version of $\mathcal{C}$ at one sixth of the natural tempo (i.e., MHX6 and MH2PX6), the value of i for the ninth movement of Telemann's "Germania mit ihrem Chor" (TWV 12:1c) (telemamagdebgerman09m, see Table 1.4), was actually `"fddd3"`, not `"fd2"`, and the note error count should have been 41 not 74.

5. When Temperley and Sleator's `met-harm-two-pass` script (see section 4.7.1) was run on the version of $\mathcal{C}$ at one sixth of the natural tempo (i.e., MH2PX6), the value of i for the second movement from Bach's Church Cantata, "Ich geh und suche mit Verlangen" (BWV 49), (bachbgcant004902m, see Table 1.4), was actually `"fddd3"`, not `"fd2"`, and the note error count should have been 298, not 1689.

The discovery of these errors implied that various corrections had to be made to the results presented in Tables 2.2, 2.3, 2.4, 4.2 and 4.3 and Figures 4.6, 4.7, 4.8 and 4.9. Various small changes also had to be made to the text in sections 2.4.2.4 and 2.5. All of these corrections have been made in the text presented above. Note that these errors implied no important changes to the main conclusions of the investigation.

N.B. Much of the material included in this Appendix was published in the following editorial letter: Meredith, D. (2007b). Letter. Computer Music Journal, 31:4, pp.5-6. http://dx.doi.org/10.1162/comj.2007.31.4.5

Appendix C

Proof of equivalence of spiral array and line of fifths in Chew and Chen's algorithm

Let S be a set of notes, let $\mathbf{CE}_S$ be the center of effect of S in the spiral array and let CE_L be the center of effect of S on the line of fifths. $\mathbf{CE}_S$ is a three-dimensional position vector, given by the following equation:

$$\mathbf{CE}_S = \frac{\sum_{n \in S} (d(n).\mathbf{p}(n))}{\sum_{n \in S} d(n)} \tag{C.1}$$

where $d(n)$ is the duration of the note n and $\mathbf{p}(n)$ is the position vector within the spiral array of the pitch name class already assigned to n. CE_L is a real number given by the following equation:

$$CE_L = \frac{\sum_{n \in S} (d(n)k(n))}{\sum_{n \in S} d(n)} \tag{C.2}$$

where $k(n)$ is the index of the pitch name class already assigned to the note n.

Let's suppose that we wish to use Chew and Chen's algorithm to assign to a note N, whose pitch class is c, the pitch name class that is closest to the center of effect of S. Let $K_S(N)$ be the set that contains the indices of the pitch name classes that can be assigned to N that are closest to $\mathbf{CE}_S$ in the spiral array; and let $K_L(N)$ be the set that contains the indices of the pitch name classes that can be assigned to N that are closest to CE_L on the line of fifths. I shall now prove the following theorem:

Theorem 1 *For a given set of notes S and a given note N, the set $K_L(N)$ is always equal to $K_S(N)$.*

Proof

For any note, n, whose pitch name class has been assigned,

$$\mathbf{p}(n) = \left\langle \frac{r}{h} \sin(k(n)\pi/2), \frac{r}{h} \cos(k(n)\pi/2), k(n) \right\rangle \tag{C.3}$$

where $\frac{r}{h}$ is the aspect ratio of the spiral array and $k(n)$ is the index of the pitch name class of n. Let's

further define that $\mathbf{p}(k)$ denotes the spiral array position vector associated with the index k and that therefore,

$$\mathbf{p}(k) = \left\langle \frac{r}{h}\sin(k\pi/2), \frac{r}{h}\cos(k\pi/2), k \right\rangle. \tag{C.4}$$

Equations C.3 and C.1 imply that

$$\mathbf{CE}_S = \left\langle \frac{\sum_{n\in S}\left(d(n)\frac{r}{h}\sin(k(n)\pi/2)\right)}{\sum_{n\in S} d(n)}, \frac{\sum_{n\in S}\left(d(n)\frac{r}{h}\cos(k(n)\pi/2)\right)}{\sum_{n\in S} d(n)}, \frac{\sum_{n\in S}(d(n)k(n))}{\sum_{n\in S} d(n)} \right\rangle. \tag{C.5}$$

Equations C.2 and C.5 together imply that

$$\mathbf{CE}_S = \left\langle \frac{\sum_{n\in S}\left(d(n)\frac{r}{h}\sin(k(n)\pi/2)\right)}{\sum_{n\in S} d(n)}, \frac{\sum_{n\in S}\left(d(n)\frac{r}{h}\cos(k(n)\pi/2)\right)}{\sum_{n\in S} d(n)}, CE_L \right\rangle. \tag{C.6}$$

In other words, the z component of the position vector of $\mathbf{CE}_S$ in the spiral array is CE_L, the center of effect of S on the line of fifths. It can readily be shown that

$$K_L(N) = \{k \mid (k = 12i + (7c \bmod 12)) \wedge (i \text{ is an integer}) \wedge (\textsc{Abs}(k - CE_L) \text{ is a minimum})\}. \tag{C.7}$$

Similarly, it is clear that

$$K_S(N) = \{k \mid (k = 12i + (7c \bmod 12)) \wedge (i \text{ is an integer}) \wedge (|\mathbf{p}(k) - \mathbf{CE}_S| \text{ is a minimum})\} \tag{C.8}$$

where $|\mathbf{x}|$ is the length of the vector $\mathbf{x}$ and $\mathbf{p}(k)$ is as defined in Equation C.4. From Equations C.7 and C.8, it follows that $K_L(N)$ is always equal to $K_S(N)$ iff

$$(\textsc{Abs}(12i + (7c \bmod 12) - CE_L) \text{ is a minimum}) \iff (|\mathbf{p}(12i + (7c \bmod 12)) - \mathbf{CE}_S| \text{ is a minimum}). \tag{C.9}$$

Let's define that $\mathbf{CE}_{S,x}$ and $\mathbf{CE}_{S,y}$ denote the x and y components of $\mathbf{CE}_S$, respectively. That is, from Equation C.5,

$$\mathbf{CE}_{S,x} = \frac{\sum_{n\in S}\left(d(n)\frac{r}{h}\sin(k(n)\pi/2)\right)}{\sum_{n\in S} d(n)} \tag{C.10}$$

and

$$\mathbf{CE}_{S,y} = \frac{\sum_{n\in S}\left(d(n)\frac{r}{h}\cos(k(n)\pi/2)\right)}{\sum_{n\in S} d(n)}. \tag{C.11}$$

From Equations C.4, C.6, C.10 and C.11, it follows that

$$\begin{aligned} |\mathbf{p}(12i + (7c \bmod 12)) - \mathbf{CE}_S| = |\langle & \frac{r}{h}\sin((12i + (7c \bmod 12))\pi/2) - \mathbf{CE}_{S,x}, \\ & \frac{r}{h}\cos((12i + (7c \bmod 12))\pi/2) - \mathbf{CE}_{S,y}, \\ & (12i + (7c \bmod 12)) - CE_L\rangle| \end{aligned}$$

and therefore

$$\begin{aligned} |\mathbf{p}(12i + (7c \bmod 12)) - \mathbf{CE}_S| = |\langle & \frac{r}{h}\sin(6\pi i + (7c \bmod 12)\pi/2) - \mathbf{CE}_{S,x}, \\ & \frac{r}{h}\cos(6\pi i + (7c \bmod 12)\pi/2) - \mathbf{CE}_{S,y}, \\ & 12i + (7c \bmod 12) - CE_L\rangle|. \end{aligned} \tag{C.12}$$

But $\sin(2\pi j + x) = \sin(x)$ and $\cos(2\pi j + x) = \cos(x)$ for all integers j. Therefore,

$$\begin{aligned} |\mathbf{p}(12i + (7c \bmod 12)) - \mathbf{CE}_S| = |\langle & \frac{r}{h}\sin((7c \bmod 12)\pi/2) - \mathbf{CE}_{S,x}, \\ & \frac{r}{h}\cos((7c \bmod 12)\pi/2) - \mathbf{CE}_{S,y}, \\ & 12i + (7c \bmod 12) - CE_L\rangle|. \end{aligned} \tag{C.13}$$

This implies that the x and y components of $|\mathbf{p}(12i + (7c \bmod 12)) - \mathbf{CE}_S|$ are constant for all values of i. This corresponds to the geometrical fact that all the possible spellings of a given note N lie on a straight line parallel with the central axis of the spiral array. Therefore, for a given note N with a pitch class c and a given context set of notes S, $|\mathbf{p}(12i + (7c \bmod 12)) - \mathbf{CE}_S|$ is a minimum if and only if the absolute value of its z component is a minimum. That is,

$$(|\mathbf{p}(12i + (7c \bmod 12)) - \mathbf{CE}_S| \text{ is a minimum}) \iff (\textsc{Abs}(12i + (7c \bmod 12) - CE_L) \text{ is a minimum})$$

which, as stated above (see Eq. C.9), implies that $K_L(N)$ is always equal to $K_S(N)$. ■

Theorem 1 implies that, for a given note N and a given context set of notes S, the set of pitch name classes that can be assigned to N that are closest to the center of effect of S in the spiral array is always the same as the set of pitch name classes assignable to N that are closest to the center of effect of S on the line of fifths. Note that $K_L(N)$ has cardinality 2 if the minimum value of $\textsc{Abs}(12i + (7c \bmod 12) - CE_L)$ is 6; otherwise, $K_L(N)$ contains a single value.

References

Abdallah, S. A. and Plumbley, M. D. (2004). Polyphonic transcription by non-negative sparse coding of power spectra. In *Proceedings of the Fifth International Conference on Music Information Retrieval (ISMIR 2004)*, Universitat Pompeu Fabra, Barcelona, Spain.

Arnold, D., editor (1983). *The New Oxford Companion to Music*. Oxford University Press, Oxford.

Associated Board of the Royal Schools of Music (1958). *Rudiments and Theory of Music*. Associated Board of the Royal Schools of Music, 14 Bedford Square, London, WC1B 3JG.

Babbitt, M. (1965). The structure and function of musical theory: I. In *College Music Symposium*, volume 5, pages 49–60.

Backus, J. (1977). *The Acoustical Foundations of Music*. Norton, New York, 2nd. edition. First edition published in 1969.

Boden, M. A. (1988). *Computer models of mind: Computational approaches in theoretical psychology*. Cambridge University Press, Cambridge.

Boden, M. A., editor (1990). *The Philosophy of Artificial Intelligence*. Oxford University Press, Oxford.

Borowski, E. J. and Borwein, J. M. (1989). *Dictionary of Mathematics*. Collins.

Boyd, M. (1983). *Bach*. The Master Musicians. J. M. Dent, London.

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772.

Brinkman, A. R. (1990). *PASCAL Programming for Music Research*. The University of Chicago Press, Chicago and London.

Brown, H. (1988). The interplay of set content and temporal context in a functional theory of tonality perception. *Music Perception*, 5(3):219–250.

Butler, D. (1989). Describing the perception of tonality in music: A critique of the tonal hierarchy theory and a proposal for a theory of intervallic rivalry. *Music Perception*, 6(3):219–242.

Butler, D. and Brown, H. (1984). Tonal structure versus function: Studies of the recognition of harmonic motion. *Music Perception*, 2(1):6–24.

Cambouropoulos, E. (1996). A general pitch interval representation: Theory and applications. *Journal of New Music Research*, 25(3):231–251.

Cambouropoulos, E. (1998). *Towards a General Computational Theory of Musical Structure*. PhD thesis, University of Edinburgh.

Cambouropoulos, E. (2001). Automatic pitch spelling: From numbers to sharps and flats. In *VIII Brazilian Symposium on Computer Music (SBC&M 2001)*, Fortaleza, Brazil.

Cambouropoulos, E. (2002). Re: your comments. E-mail sent Fri, 25 Oct 2002 18:35:53 +0300 from emilios@mus.auth.gr to dave@titanmusic.com.

Cambouropoulos, E. (2003). Pitch spelling: A computational model. *Music Perception*, 20(4):411–429.

Cambouropoulos, E. (2004). Some replies to many questions. E-mail sent Mon, 17 May 2004 22:52:22 +0300 from emilios@mus.auth.gr to dave@titanmusic.com.

Cambouropoulos, E., Crochemore, M., Iliopoulos, C. S., Mouchard, L., and Pinzon, Y. J. (2002). Computing approximate repetitions in musical sequences. *International Journal of Computer Mathematics*, 79(11):1135–1148.

Chew, E. (2000). *Towards a Mathematical Model of Tonality*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA.

Chew, E. (2004). Re: Implementing your algorithms. E-mail message of Wed, 29 Dec 2004 08:55:00 -0800 sent from Elaine Chew <echew@usc.edu> to Dave Meredith <dave@titanmusic.com>.

Chew, E. and Chen, Y.-C. (2003a). Determining context-defining windows: Pitch spelling using the spiral array. In *Fourth International Conference on Music Information Retrieval (ISMIR 2003) (October 26–30)*, Baltimore, MD.

Chew, E. and Chen, Y.-C. (2003b). Mapping MIDI to the Spiral Array: Disambiguating Pitch Spellings. In Bhargava, H. K. and Ye, N., editors, *Computational Modeling and Problem Solving in the Networked World - Proceedings of the 8th INFORMS Computer Society Conference*, pages 259–275, Chandler, AZ. Kluwer.

Chew, E. and Chen, Y.-C. (2005). Real-time pitch spelling using the Spiral Array. *Computer Music Journal*, 29(2):61–76.

Clarke, E. F. (1999). Rhythm and timing in music. In Deutsch, D., editor, *The Psychology of Music*, pages 473–500. Academic Press, San Diego.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. M.I.T. Press, Cambridge, Mass.

Crochemore, M., Iliopoulos, C. S., Lecroq, T., and Pinzon, Y. J. (2001). Approximate string matching in musical sequences. In Balik, M. and Simanek, M., editors, *Proceedings of the*

Prague Stringology Club Workshop (PSCW 01), pages 26–36, Czech Technical University, Prague, Czech Republic.

Dancey, C. P. and Reidy, J. (2002). *Statistics Without Maths for Psychology: Using SPSS for Windows*. Prentice-Hall/Pearson Education, Harlow, Essex, second edition.

Davy, M. and Godsill, S. J. (2003). Bayesian harmonic models for musical signal analysis (with discussion). In Bernardo, J. M., Berger, J. O., Dawid, A. P., and Smith, A. F. M., editors, *Bayesian Statistics*, volume VII. Oxford University Press.

Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1924.

Everitt, B. S. (1992). *The Analysis of Contingency Tables*. Chapman & Hall, London, second edition.

Fleiss, J. L. (1973). *Statistical Methods for Rates and Proportions*. John Wiley & Sons, New York.

Forte, A. (1973). *The Structure of Atonal Music*. Yale University Press, New Haven and London.

Forte, A. and Gilbert, S. E. (1982). *Introduction to Schenkerian Analysis*. Norton, New York.

Fraisse, P. (1982). Rhythm and tempo. In Deutsch, D., editor, *The Psychology of Music*, pages 149–180. Academic Press.

Gabrielsson, A. (1999). The performance of music. In Deutsch, D., editor, *The Psychology of Music*, pages 501–602. Academic Press, San Diego, 2 edition.

Galil, Z. (1979). On improving the worst case running time of the Boyer-Moore string matching algorithm. *Communications of the ACM*, 22(9):505–508.

Gurney, E. (1880/1966). *The Power of Sound*. Basic Books.

Hewlett, W. B. (1997). *MuseData*: Multipurpose representation. In Selfridge-Field, E., editor, *Beyond MIDI: The Handbook of Musical Codes*, pages 402–447. MIT Press, Cambridge, MA.

Holtzmann, S. R. (1977). A program for key determination. *Interface*, 6:26–56.

Howell, D. C. (1982). *Statistical Methods for Psychology*. Duxbury Press, Boston, MA.

Hughes, M. (1977). A quantitative analysis. In Yeston, M., editor, *Readings in Schenker Analysis and Other Approaches*. Yale University Press, New Haven, CT.

Kerman, J., Tyson, A., and Burnham, S. G. (2004). Beethoven, Ludwig van. In Macy, L., editor, *Grove Music Online*. Oxford University Press, Oxford. (Accessed Sunday 9 January 2005), <http://www.grovemusic.com>.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice-Hall International, London.

Knopoff, L. and Hutchinson, W. (1983). Entropy as a measure of style: The influence of sample length. *Journal of Music Theory*, 27:75–97.

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350.

Kostka, S. and Payne, D. (1995). *Workbook for Tonal Harmony*. McGraw-Hill, New York.

Krumhansl, C. L. (1990). *Cognitive Foundations of Musical Pitch*, volume 17 of *Oxford Psychology Series*. Oxford University Press, New York and Oxford.

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

Krumhansl, C. L. and Shepard, R. N. (1979). Quantification of the hierarchy of tonal functions within a diatonic context. *Journal of Experimental Psychology: Human Perception and Performance*, 5(4):579–594.

Lee, C. S. (1991). The perception of metrical structure: Experimental evidence and a model. In Howell, P., West, R., and Cross, I., editors, *Representing Musical Structure*, pages 59–127. Academic Press, London.

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

Levinson, J. (1997). *Music in the Moment*. Cornell University Press.

Longuet-Higgins, H. C. (1962a). Letter to a musical friend. *The Music Review*, 23:244–248. Republished as Longuet-Higgins (1987b).

Longuet-Higgins, H. C. (1962b). Letter to a musical friend. *The Music Review*, 23:271–280. Republished as Longuet-Higgins (1987b).

Longuet-Higgins, H. C. (1976). The perception of melodies. *Nature*, 263(5579):646–653. Republished as Longuet-Higgins (1987a, 1993).

Longuet-Higgins, H. C. (1987a). The perception of melodies. In Longuet-Higgins, H. C., editor, *Mental Processes: Studies in Cognitive Science*, pages 105–129. British Psychological Society/MIT Press, London, England and Cambridge, Mass. Based on Longuet-Higgins (1976) and re-published as Longuet-Higgins (1993).

Longuet-Higgins, H. C. (1987b). Two letters to a musical friend. In Longuet-Higgins, H. C., editor, *Mental Processes: Studies in Cognitive Science*, pages 64–81. British Psychological Society/MIT Press, London, England and Cambridge, Mass. Published earlier as Longuet-Higgins (1962a,b).

Longuet-Higgins, H. C. (1993). The perception of melodies. In Schwanauer, S. M. and Levitt, D. A., editors, *Machine Models of Music*, pages 471–495. M.I.T. Press, Cambridge, Mass. Published earlier as Longuet-Higgins (1976, 1987a).

Longuet-Higgins, H. C. and Steedman, M. J. (1971). On interpreting Bach. In Meltzer, B. and Michie, D., editors, *Machine Intelligence*, volume 6, pages 221–241. University of Edinburgh Press. Republished as Longuet-Higgins and Steedman (1987).

Longuet-Higgins, H. C. and Steedman, M. J. (1987). On interpreting Bach. In Longuet-Higgins, H. C., editor, *Mental Processes: Studies in Cognitive Science*, pages 82–104. British Psychological Society/MIT Press, London, England and Cambridge, Mass. Published earlier as Longuet-Higgins and Steedman (1971).

Maxwell, H. J. (1992). An expert system for harmonic analysis of tonal music. In Balaban, M., Ebcioğlu, K., and Laske, O., editors, *Understanding Music with AI: Perspectives on Music Cognition*, pages 335–353. AAAI Press/MIT Press, Cambridge, MA.

McNemar, Q. (1969). *Psychological Statistics*. John Wiley and Sons, New York, 4th edition.

Meredith, D. (2002a). Pitch spelling questions. E-mail sent Thu, 24 Oct 2002 11:09:51 +0100 from dave@titanmusic.com to emilios@mus.auth.gr.

Meredith, D. (2002b). Review of *The Cognition of Basic Musical Structures* by David Temperley. *Musicae Scientiae*, 6(2):pp. 287–302. Draft available online at <http://www.titanmusic.com/papers/public/temperley_review_3.pdf>.

Meredith, D. (2003). Pitch spelling algorithms. In Kopiez, R., Lehmann, A. C., Wolther, I., and Wolf, C., editors, *Proceedings of the Fifth Triennial ESCOM Conference (ESCOM5) (8-13 September 2003)*, pages pp. 204–207, Hanover University of Music and Drama, Hanover, Germany. Available online at <http://www.epos.uos.de/music/books/k/klww003/pdfs/080_Meredith_Proc.pdf>.

Meredith, D. (2005). Comparing pitch spelling algorithms on a large corpus of tonal music. In Wiil, U. K., editor, *Computer Music Modeling and Retrieval, Second International Symposium, CMMR 2004, Esbjerg, Denmark, May 26–29, 2004, Revised Papers*, volume 3310 of *Lecture Notes in Computer Science (LNCS)*, pages 173–192, Berlin. Springer. Available online at <http://www.springerlink.com/link.asp?id=b3vd6fpumrmpkqww>. (Draft and slides available at <http://www.titanmusic.com/papers.html>).

Meredith, D., Lemström, K., and Wiggins, G. A. (2002). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345. Available online at <http://taylorandfrancis.metapress.com/link.asp?id=yql23xw0177lt4jd>.

Meredith, D. and Wiggins, G. A. (2005). Comparing pitch spelling algorithms. In *Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR 2005)*, pages 280–287, Queen Mary, University of London. Available online at <http://ismir2005.ismir.net/proceedings/1004.pdf>.

Morris, R. D. (1987). *Composition with Pitch-Classes: A Theory of Compositional Design*. Yale University Press, New Haven and London.

Oldham, G. and Lindley, M. (2006). Wolf. In Macy, L., editor, *Grove Music Online*. Oxford University Press. <http://www.grovemusic.com>, accessed 29 May 2006.

Piperno, F. (1987). Corelli: Trio sonatas. Programme notes to CD 419 614-2, Arcangelo Corelli: Trio Sonatas, performed by Trevor Pinnock with members of the English Concert, Archiv Produktion.

Piston, W. (1978). *Harmony*. Victor Gollancz Ltd., London. Revised and expanded by Mark DeVoto.

Plumbley, M., Abdallah, S., Bello, J., Davies, M. E., Monti, G., and Sandler, M. (2002). Automatic music transcription and audio source separation. *Cybernetics and Systems*, 33(6):603–627.

Popper, K. R. (1983). *Realism and the Aim of Science (Postscript to* The Logic of Scientific Discovery, Volume 1*)*. Hutchinson, London.

Rahn, J. (1980). *Basic Atonal Theory*. Longman, New York.

Raphael, C. and Stoddard, J. (2003). Harmonic analysis with probabilistic graphical models. In *Proceedings of the Fourth International Conference on Music Information Retrieval (ISMIR 2003)*, Baltimore, Maryland, USA. Available online at <http://ismir2003.ismir.net/papers/Raphael.PDF>.

Regener, E. (1973). *Pitch Notation and Equal Temperament: A Formal Study*. University of California Press, Berkeley, CA.

Royston, P. (1982a). Algorithm AS 181: The W test for normality. *Applied Statistics*, 31:176–180.

Royston, P. (1982b). An extension of Shapiro and Wilk's W test for normality to large samples. *Applied Statistics*, 31:115–124.

Royston, P. (1995). A remark on algorithm AS 181: The W test for normality. *Applied Statistics*, 44:547–551.

Schenker, H. (1979). *Free Composition (Der Freie Satz): Volume III of New Musical Theories and Fantasies*. Schirmer, New York. Translated and edited by Ernst Oster.

Searle, J. R. (1980). Minds, Brains, and Programs. *The Behavioral and Brain Sciences*, 3:417–424. Republished as Searle (1990).

Searle, J. R. (1990). Minds, Brains, and Programs. In Boden, M. A., editor, *The Philosophy of Artificial Intelligence*, pages 67–88. Oxford University Press, Oxford. Published earlier as Searle (1980).

Shapiro, S. S. and Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 52(3 and 4):591–611.

Sleator, D. and Temperley, D. (n.d.a). The harmony program. <http://www.link.cs.cmu.edu/music-analysis/harmony.html> accessed on 8 March 2005.

Sleator, D. and Temperley, D. (n.d.b). The meter program. <http://www.link.cs.cmu.edu/music-analysis/meter.html> accessed on 21 January 2005.

Stoddard, J., Raphael, C., and Utgoff, P. E. (2004). Well-tempered spelling: A key-invariant pitch spelling algorithm. In *Proceedings of the Fifth International Conference on Music Information Retrieval (ISMIR 2004) (October 10–14)*, Universitat Pompeu Fabra, Barcelona, Spain.

Temperley, D. (1997). An algorithm for harmonic analysis. *Music Perception*, 15(1):31–68.

Temperley, D. (1999). What's key for key? The Krumhansl-Schmuckler key-finding algorithm reconsidered. *Music Perception*, 17(1):65–100.

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

Temperley, D. and Sleator, D. (1999). Modeling meter and harmony: A preference rule approach. *Computer Music Journal*, 23(1):10–27.

The MIDI Manufacturers' Association (1996). Midi 1.0 detailed specification (document version 4.2, revised february 1996). In *The Complete MIDI 1.0 Detailed Specification (Version 96.1)*, chapter 2. The MIDI Manufacturers' Association, P.O. Box 3173, La Habra, CA., 90632-3173.

Tillmann, B. and Bigand, E. (2004). The relative importance of local and global structures in music perception. *The Journal of Aesthetics and Art Criticism*, 62(2):211–222.

Tukey, J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley, Reading, MA.

Vos, P. G. and van Geenen, E. W. (1996). A parallel-processing key-finding model. *Music Perception*, 14:185–224.

Walmsley, P. J. (2000). *Signal Separation of Musical Instruments*. PhD thesis, Signal Processing Group, Department of Engineering, University of Cambridge.

Winograd, T. (1968). Linguistics and the computer analysis of tonal harmony. *Journal of Music Theory*, 12:2–49. Republished as Winograd (1993).

Winograd, T. (1993). Linguistics and the computer analysis of tonal harmony. In Schwanauer, S. M. and Levitt, D. A., editors, *Machine Models of Music*, pages 113–153. M.I.T. Press, Cambridge, Mass. Published earlier as Winograd (1968).

Young, R. W. (1939). Terminology for logarithmic frequency units. *Journal of the Acoustical Society of America*, 11:134–139.

Youngblood, J. E. (1958). Style as information. *Journal of Music Theory*, 2:24–35.