Journal of New Music Research
2006, Vol. 35, No. 2, pp. 121 – 159

The *ps13* Pitch Spelling Algorithm

David Meredith

Centre for Cognition, Computation and Culture, Department of Computing, Goldsmiths College, University of London, UK

Abstract

In the *ps13* pitch spelling algorithm, the pitch name of a note is assumed to depend on the local key and voice-leading. In the first stage of *ps13*, the pitch name implied for a note by a tonic is assumed to be the one that lies closest to that tonic on the line of fifths. The strength with which a pitch name is implied for a note is assumed to be proportional to the sum of the frequencies of occurrence, within a context around the note, of the tonics that imply that pitch name. In the second stage of *ps13*, certain neighbour-note and passing-note errors in the output of the first stage are corrected. An implementation of *ps13*, called PS13, spelt correctly 99.31% of the notes in a 195972 note test corpus, $\mathcal{C}$. A post-processing phase was added to PS13 in which the pitch names computed by PS13 are transposed by a diminished second if this brings them closer on the line of fifths to the pitch names of the notes in their vicinity. This version of the algorithm spelt 99.43% of the notes in $\mathcal{C}$ correctly. When the second stage was removed altogether from PS13, 99.44% of the notes in $\mathcal{C}$ were spelt correctly. The *ps13*-based algorithms achieved higher note accuracies than the algorithms of Temperley, Longuet-Higgins, Cambouropoulos and Chew and Chen on both $\mathcal{C}$ and a "noisy" version of $\mathcal{C}$ containing temporal deviations similar to those that occur in MIDI files derived from human performances.

1. Introduction

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, such an algorithm is an essential component of a 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 (Walmsley, 2000; Plumbley et al. 2002; Davy & Godsill, 2003; Abdallah & Plumbley, 2004). 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, 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. 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

Correspondence: David Meredith, The Barn, Middle Broad Drove, Tydd St. Giles, Wisbech, Cambs., PE13 5PA, UK.
E-mail: dave@titanmusic.com

DOI: 10.1080/09298210600834961 © 2006 Taylor & Francis

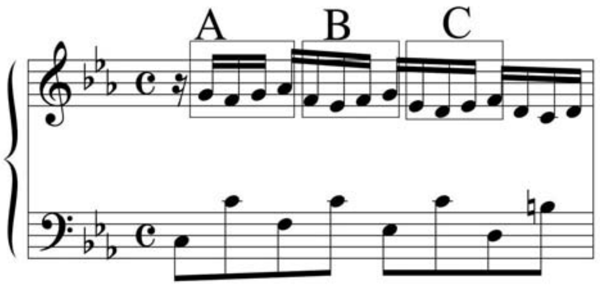

Fig. 1. Three perceptually similar patterns with different chromatic pitch interval structures (from the first bar of the Prelude in C minor (BWV 871/1) from Book 2 of J. S. Bach's *Das Wohltemperirte Clavier*).

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 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 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".

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".

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 for transcription and music information retrieval, but also with "useful insights into possible cognitive principles and mechanisms... that may be at work during the pitch transcription task" (Cambouropoulos, 2003, pp. 413–414).

In this paper, Section 2 is a discussion of the theory underlying the *ps13* pitch spelling algorithm, which several published evaluations have shown to be one of the most accurate pitch spelling algorithms currently available (Meredith 2003, 2005, 2006; Meredith & Wiggins, 2005). In Section 3, various basic concepts, notation, terminology and functions are defined that will be used without further comment throughout the remainder of this paper. Then, in Section 4, pseudocode will be presented and discussed for an implementation of *ps13* called PS13. In Section 5, the computational complexity of PS13 will be discussed. In Section 6, the methodology used here to evaluate and compare the algorithms will be briefly described. Then, in Section, 7, I present the results obtained when PS13 was run on a test corpus, denoted by $\mathcal{C}$, containing 195972 notes equally divided between eight baroque and classical composers. In Section 8, possible ways of improving PS13 will be discussed and the results obtained when these improvements are implemented will be presented. The results achieved by *ps13* on the test corpus $\mathcal{C}$ need to be set in context by considering what can be achieved using various trivial or chance-based *baseline* algorithms. This is done in Section 9. Then, in Section 10, the best versions of *ps13* will be compared with optimized versions of the algorithms proposed by Cambouropoulos (1996, 1998, 2001, 2003), Longuet-Higgins (1976, 1987, 1993), Temperley (2001) and Chew and Chen (2003a,b, 2005). In Section 11, I present and discuss the results obtained when these same algorithms were run on a "noisy" version of $\mathcal{C}$ in order to measure how robust they are to temporal deviations of the type that typically occurs in MIDI files derived from human performances. Finally, in Section 12, the main results of this paper are summarized and some directions for future research are suggested.

2. The theory underlying *ps13*

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. For example, in the "theory of tonality" (Longuet-Higgins, 1987 p. 115) implemented in the pitch spelling algorithm incorporated in Longuet-Higgins's (1976, 1987, 1993) `music.p` program, one rule ensures that each note is spelt in accordance with the local key (Longuet-Higgins, 1987, pp. 112–113), while three other rules (Longuet-Higgins, 1987, pp. 113–114) are primarily designed to ensure good voice-leading. Similarly, in Temperley's (2001, pp. 124–132) theory of pitch spelling, the first "tonal-pitch-class preference rule" (TPR 1) (Temperley, 2001, p. 125) effectively captures the influence of key on pitch spelling, while the second preference rule in this theory (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 (2005, p. 67) pitch spelling algorithm, the "centre of effect" of a set of notes is defined to be the weighted centroid of the position vectors of the pitch names of the notes in Chew's (2000) "Spiral Array Model", which is a geometric model of tonal pitch relations. In the spiral array, the pitch names are arranged on a helix so that adjacent pitch names along this helix are a perfect fifth apart and adjacent pitch names along the length of the cylinder in which the helix is embedded are a major third apart. The centre of effect of the notes preceding the note to be spelt acts "as a proxy for" the local key, and notes are spelt so that they are as close as possible to this centre of effect (Chew and Chen, 2005, p. 63). However, although Chew and Chen's algorithm models the effect of key on pitch spelling, it does not take voice-leading into account, and Chew and Chen (2005, pp. 73, 75) claim that this is one of the main causes of the errors made by their algorithm.

Unlike most of the other pitch spelling algorithms that have been proposed in the literature, Cambouropoulos's method does not explicitly take key into account. Instead, it is based on what he calls the principles of *interval optimization* and *notational parsimony* (Cambouropoulos, 2003, p. 421). Interval optimization 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, i.e.

... F♭ C♭ G♭ D♭ A♭ E♭ B♭ F C G D A E B F♯ C♯ G♯ D♯ A♯ E♯ B♯ ...

(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 may indirectly model 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. Furthermore, the principle of notational parsimony involves a pitch name being penalized if it is a double-sharp or a double-flat – i.e. if it is more than 10 steps away from D♮ on the line of fifths. This, again, tends to keep the pitch names assigned to notes within a fairly compact region on the line of fifths, which, indirectly, may serve to model the effect of key on pitch spelling.

The earliest published version of Cambouropoulos's (1996, 1998) method 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) 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".

When determining the pitch name of a note, the *ps13* algorithm (Meredith 2003, 2005, 2006; Meredith & 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 "centre of effect" which acts "as a proxy for" the key (Chew and Chen 2005, p. 63). The results of a thorough evaluation of Chew and Chen's algorithm (Meredith & Wiggins, 2005, p. 286; Meredith, 2006, Chapter 5) showed that it performed equally well when the spiral array was replaced with the line of fifths. All 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 the local tonic on the line of fifths. This has therefore been adopted as the basic principle underlying Stage 1 of *ps13*. If this principle is adopted, however, one needs to decide whether notes six semitones away from the tonic are spelt as sharpened subdominants or flattened dominants. In *ps13*, notes six semitones away from the tonic are spelt so that they are six steps *sharp* of the tonic rather than six steps flat of it along the line of fifths. This implies that, in Stage 1 of *ps13*, each note is spelt in the same way as it is in the harmonic chromatic scale beginning on the local tonic (Associated Board of the Royal Schools of Music, 1958, p. 78).

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" rule in his theory of pitch spelling and that, "in many cases, this rule is sufficient to ensure the correct spelling of passages". Moreover, in a thorough evaluation carried out by Meredith (2006, Chapters 4–5) and summarized by Meredith and Wiggins (2005, p. 286), Chew and Chen's algorithm (which ignores voice-leading) and a straightforward implementation of Temperley's TPR 1 both achieved high note accuracies, supporting the claim that the vast majority of notes in a passage of tonal music can be spelt correctly simply by keeping the pitch names close together on the line of fifths.

In the algorithms of Longuet-Higgins, Temperley and Chew and Chen, the local tonic at each time point in a passage of tonal 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 & 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) which 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 unambiguous 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 six 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 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 12, Section 4.2).

3. Preliminary definitions

3.1 Ordered sets and strings

The names of ordered sets will be printed in **bold-face** font. When printed in full, the elements of an ordered set will be enclosed between angle brackets and separated by commas (e.g. $\mathbf{A} = \langle 1, 2, 3, 1\rangle$ contains the elements 1, 2, 3 and 1 in that order). The *empty ordered set* will be denoted by $\langle\rangle$. The ith element of an ordered set, $\mathbf{A}$, will be denoted by $\mathbf{A}[i-1]$. This "square-bracket" notation can be used when an ordered set is printed out in full: $\langle 1, 2, 3, 4\rangle[2] = 3$. It can also be extended for accessing elements of nested ordered sets. For example, 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 $\mathbf{A}[0][1] = 2$ and $\mathbf{A}[2][1][1][0] = 3$. The *length* of an ordered set, $\mathbf{A}$, denoted by $|\mathbf{A}|$, is the number of elements in $\mathbf{A}$ (e.g. $|\langle 1, 1, 2, 3\rangle| = 4$). If $\mathbf{A} = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ then $\mathbf{A}[i, j] = \langle a_i, a_{i+1}, \ldots, a_{j-1}\rangle$. If $\mathbf{A} = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ and $\mathbf{B} = \langle b_0, b_1, \ldots, b_{m-1}\rangle$, then the *concatenation of* **B** *onto* **A**, denoted by $\mathbf{A} \oplus \mathbf{B}$, is $\langle a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{m-1}\rangle$. If $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$ are all ordered sets, then $\bigoplus_{i=1}^{n} \mathbf{A}_i = \mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \cdots \oplus \mathbf{A}_n$. An ordered set in which every element is a character may be called a *string*. When printed out in full, strings and characters will be printed in `teletype` font. Characters will be enclosed between single quotes (e.g. `'a'`). When printed out in full, the characters in a string may be printed adjacent to each other between double quotes (e.g. `"abcdefg"` = ⟨`'a'`, `'b'`, `'c'`, `'d'`, `'e'`, `'f'`, `'g'`⟩). The empty string (which is equal to the empty ordered set) will be denoted by `""`. All notation relating to ordered sets may be used for strings. If **Y** is an ordered set, then the function Pos(x, **Y**) returns

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

3.2 Pseudocode conventions

The pseudocode used in this paper should be easy to read for anyone who is familiar with a procedural programming language (e.g. C or Java). 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 "ComputeChromaList" on the first line of Figure 9, Section 4.1). The name of the function is followed on the same line by a list of parameters. For example, the function ComputeChromaList in Figure 9 takes two parameters, **SortedOCPList** and n. The lines following the header line of a function define the instructions carried out by the function on its parameters. 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. Everything occurring on a printed line after the symbol "▶" is a comment and is not executed. 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. 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). A **return** statement has the same meaning here as it does in C (Kernighan and Ritchie, 1988, pp. 25–26).

3.3 Some basic mathematical functions and algorithms

The function 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: $x \bmod y = x - y\lfloor x/y \rfloor$.

3.4 Pitch and pitch interval representations

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

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

{`"A"`, `"B"`, `"C"`, `"D"`, `"E"`, `"F"`, `"G"`, `"a"`, `"b"`, `"c"`, `"d"`, `"e"`, `"f"`, `"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. $A =$ `""`;
2. $A \in$ {`"n"`, `"N"`};
3. $|A| \geq 1 \wedge A[i] \in$ {`'b'`, `'B'`} for all $0 \leq i < |A|$;
4. $|A| \geq 1 \wedge A[i] \in$ {`'f'`, `'F'`} for all $0 \leq i < |A|$;
5. $|A| \geq 1 \wedge A[i] \in$ {`'s'`, `'S'`} for all $0 \leq i < |A|$; or
6. $|A| \geq 1 \wedge A[i] =$ `'#'` for all $0 \leq i < |A|$.

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

If x is a rational number, 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"`.

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''`, `''Af0''` 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, $\langle p_c, p_m \rangle$, 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 $\langle p_c, p_m \rangle$, then its *continuous binomial representation* in Brinkman's (1990, pp. 133–135) system is $\langle p_c + 9, p_m + 5 \rangle$.

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

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 $(m-2) \bmod 7$ and its *diatonic note class* in Regener's (1973, p. 34) theory is $(m+2) \bmod 7$.

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

If p_c is the chromatic pitch of a note, then its *chromatic octave* is $\lfloor p_c/12 \rfloor$. If p_m is the morphetic pitch of a note, then its *morphetic octave* is $\lfloor p_m/7 \rfloor$. 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. $o_m + 1$ 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.

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.

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 2 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

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	···

Fig. 2. 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.

Temperley's (2001, p. 118) theory is $\ell + 1$ and its *sharpness* in Longuet-Higgins's (1987, 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.

3.5 Algorithms for converting between different pitch and pitch interval representations

A selection of algorithms for converting between different types of pitch and pitch interval representations will now be presented. The following is not intended to constitute a complete and exhaustive conversion system. Only those functions that are used in this paper will be presented here (for a larger selection of conversion algorithms, see Meredith, 2006, pp. 41–50).

The pitch name that corresponds to a chromamorphetic pitch, *CMP*, can be determined using the function P2PN, defined in Figure 3. This function has one parameter, *CMP*, which must be a chromamorphetic pitch.

The pitch name class of a note can be derived from its pitch name using the trivial function, PN2PNC, defined in Figure 4.

The line-of-fifths position of a pitch name class can be computed using the function PNC2LOF, defined in Figure 5. PNC2LOF has a single parameter, *PNC*, which must be a valid pitch name class. The function UpCase(x), called in the first line of PNC2LOF, 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.

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 6. 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

```
P2PN($CMP$)
$Morph \leftarrow CMP[1] \bmod 7$
$LetterName \leftarrow \langle$"A", "B", "C", "D", "E", "F", "G"$\rangle\,[Morph]$
$UndisplacedChroma \leftarrow \langle 0, 2, 3, 5, 7, 8, 10\rangle\,[Morph]$
$Displacement \leftarrow CMP[0] - 12\lfloor CMP[1]/7\rfloor - UndisplacedChroma$
$Accidental \leftarrow$ ""
if $Displacement \neq 0$
    if $Displacement < 0$
        $AccidentalChar \leftarrow$ "f"
    else
        $AccidentalChar \leftarrow$ "s"
    for $i \leftarrow 0$ to ABS($Displacement$) $- 1$
        $Accidental \leftarrow Accidental \oplus AccidentalChar$
else
    $Accidental \leftarrow$ "n"
$ASAOctaveNumber \leftarrow \lfloor CMP[1]/7\rfloor$
if $Morph > 1$
    $ASAOctaveNumber \leftarrow ASAOctaveNumber + 1$
return $LetterName \oplus Accidental \oplus$ NUM2STR($ASAOctaveNumber$)
```

Fig. 3. The P2PN algorithm.

```
PN2PNC($PN$)
$i \leftarrow |PN| - 1$
while $PN[i] \in \{$'-', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'$\}$
    $i \leftarrow i - 1$
return $PN[0, i+1]$
```

Fig. 4. The PN2PNC algorithm.

```
PNC2LOF($PNC$)
$MorphenticLOFClass \leftarrow$ POS(UPCASE($PNC[0]$), "FCGDAEB")
if $(|PNC| = 1) \vee$ (UPCASE($PNC[1]$) = 'N')
    $MorphenticLOFCycle \leftarrow 0$
else
    if UPCASE($PNC[1]$) $\in \{$'S', '#'$\}$
        $MorphenticLOFCycle \leftarrow |PNC| - 1$
    else
        if UPCASE($PNC[1]$) $\in \{$'B', 'F'$\}$
            $MorphenticLOFCycle \leftarrow 1 - |PNC|$
return $MorphenticLOFClass + 7\,MorphenticLOFCycle$
```

Fig. 5. The PNC2LOF algorithm.

```
TPCMIDI2PN($TPC$, $MIDINoteNumber$)
$ChromaticPitch \leftarrow MIDINoteNumber - 21$
$Morph \leftarrow (4\,TPC + 1) \bmod 7$
$Displacement \leftarrow \lfloor (TPC - 1)/7\rfloor$
$UndispChromPitch \leftarrow ChromaticPitch - Displacement$
$MorphenticOctave \leftarrow \lfloor UndispChromPitch/12\rfloor$
$MorphenticPitch \leftarrow Morph + 7\,MorphenticOctave$
return P2PN($\langle ChromaticPitch, MorphenticPitch\rangle$)
```

Fig. 6. The TPCMIDI2PN algorithm.

function P2PN, defined in Figure 3, to compute the pitch name from this chromamorphetic pitch.

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 7. This function has two parameters: *LOF*, which must be a line-of-fifths position; and *CP*, which must be a chromatic

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

Fig. 7. The LOFCP2PN algorithm.

pitch. The MIDI note number is derived by adding 21 to *CP* and the tonal pitch class is derived by adding 1 to *LOF*.

4. PS13: an implementation of *ps13*

Figure 8 shows an implementation called PS13 of the complete *ps13* algorithm introduced in Section 2 above. As shown in Figure 8, 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, $\langle t_{\text{on}}, p_{\text{c}}\rangle$, gives 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 Section 2, 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** $[ContextStart, ContextEnd]$ where $ContextStart = \text{MAX}(\{0, i - K_{\text{pre}}\})$ and $ContextEnd = \text{MIN}(\{n, i + K_{\text{post}}\})$ 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.

```
PS13(SortedOCPList, $K_{pre}$, $K_{post}$)
      ▶ Stage 1 begins here.
1     $n \leftarrow |$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     $N_{Ch} \leftarrow |$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, $n$)
```

Fig. 8. The PS13 algorithm.

4.1 Implementing Stage 1 of *ps13*: lines 1 – 4 of PS13

In line 2 of PS13 (see Figure 8), an ordered set of chromas, **ChromaList**, is derived from **SortedOCPList** such that $|\textbf{ChromaList}| = |\textbf{SortedOCPList}|$ and **ChromaList**[i] is the chroma of the note represented by **SortedOCPList**[i] for all $0 \le i < n$. This can be accomplished using the simple function defined in Figure 9. Next, in line 3 of PS13 (see Figure 8), the function ComputeChromaVectorList, defined in Figure 10, is used to construct an ordered set of 12-vectors, **ChromaVectorList**, such that $|\textbf{ChromaVectorList}| = |\textbf{SortedOCPList}| = n$ and **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**[Max($\{0,\ i - K_{pre}\}$), Min($\{n,\ i + K_{post}\}$)]). Therefore, when line 3 of PS13 has completed, **ChromaVectorList**[i][c] gives the frequency with which chroma, c, occurs within the context, **SortedOCPList**[Max($\{0,\ i - K_{pre}\}$), Min($\{n,\ i + K_{post}\}$)], surrounding the note represented by **SortedOCPList**[i].

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 19, Section 4.2).

As explained in Section 2, 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.

```
ComputeChromaList(SortedOCPList, $n$)
    ChromaList ← ⟨ ⟩
    for $i \leftarrow 0$ to $n - 1$
        Chroma ← SortedOCPList[$i$][1] mod 12
        ChromaList ← ChromaList ⊕ ⟨Chroma⟩
    return ChromaList
```

Fig. 9. The ComputeChromaList function.

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, 9, at the beginning of the passage to be analysed is ``Fs'', then it is assumed that the pitch name class, ``Fs'', is associated with the tonic chroma, 9, 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 $m(c_t, j)$ 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 $m_t(c_t, j)$ 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

$$m(c_t, j) = (\textbf{MorphInt}[(c_j - c_t) \bmod 12] + m_t(c_t, j)) \bmod 7, \tag{1}$$

where

$$\textbf{MorphInt} = \langle 0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6 \rangle. \tag{2}$$

```
ComputeChromaVectorList(ChromaList, $K_{\text{pre}}$, $K_{\text{post}}$, $n$)
    ThisVector ← ⟨0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0⟩
    for $i \leftarrow 0$ to Min({$n$, $K_{\text{post}}$}) − 1
        ThisVector[ChromaList[$i$]] ← 1 + ThisVector[ChromaList[$i$]]
    ChromaVectorList ← ⟨ThisVector⟩
    for $i \leftarrow 1$ to $n - 1$
        if $i + K_{\text{post}} \leq n$
            ThisVector[ChromaList[$i + K_{\text{post}} - 1$]] ← 1 + ThisVector[ChromaList[$i + K_{\text{post}} - 1$]]
        if $i - K_{\text{pre}} > 0$
            ThisVector[ChromaList[$i - K_{\text{pre}} - 1$]] ← ThisVector[ChromaList[$i - K_{\text{pre}} - 1$]] − 1
        ChromaVectorList ← ChromaVectorList ⊕ ⟨ThisVector⟩
    return ChromaVectorList
```

Fig. 10. The ComputeChromaVectorList function.

Let $C_t(m, j)$ denote the set of tonic chromas that imply that the morph, m, should be assigned to c_j. That is,

$$C_t(m,j) = \{c_t | m(c_t, j) = m\}. \tag{3}$$

It follows that the strength, $S(m, j)$, with which morph, m, is implied as the spelling for c_j is given by

$$S(m,j) = \sum_{c_t \in C_t(m,j)} \textbf{ChromaVectorList}[j][c_t], \tag{4}$$

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

$$m_j = \text{Pos}(\text{Max}(\langle S(0,j), S(1,j), \ldots S(6,j)\rangle), \langle S(0,j), S(1,j), \ldots S(6,j)\rangle). \tag{5}$$

The foregoing paragraph suggests that each m_j in **MorphList** can be computed by

1. computing $m(c_t, j)$ for all $0 \leq c_t \leq 11$ using equation (1);
2. computing $C_t(m, j)$ for all $0 \leq m \leq 6$ using equation (3);
3. using **ChromaVectorList** to compute $S(m, j)$ for all $0 \leq m \leq 6$ in accordance with equation (4); and
4. using equation (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

$$m_0 = \textbf{InitMorph}[c_0], \tag{6}$$

where

$$\textbf{InitMorph} = \langle 0, 1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 6\rangle. \tag{7}$$

$m_t(c_t, 0)$ is then given by

$$m_t(c_t, 0) = (m_0 - \textbf{MorphInt}[(c_0 - c_t) \bmod 12]) \bmod 7. \tag{8}$$

MorphList can therefore be computed by, first, computing $m_t(c_t, 0)$ for all $0 \leq c_t \leq 11$ using equation (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 11.

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.

4.2 Implementing Stage 2 of *ps13*: lines 5–12 of PS13

As explained near the end of Section 2, 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 12).

Stage 2 of *ps13* is implemented in lines 5–12 of PS13 (see Figure 8). In lines 5–10, the variable, **MorphList**, is modified to produce a new list of morphs in which cases like the ones in Figure 12 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 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 12, 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 13. 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,

$$\textbf{OCMList} = \langle\langle t_{\text{on},0}, c_0, m_0\rangle, \langle t_{\text{on},1}, c_1, m_1\rangle \cdots \langle t_{\text{on},n-1}, c_{n-1}, m_{n-1}\rangle\rangle. \qquad (9)$$

Then, in lines 4 – 10 of ComputeOCMChordList, this list of triples, **OCMList**, is partitioned into classes, called *chords*, such that each chord contains all and only those triples, $\langle t_{\text{on},\,j}, c_j, m_j\rangle \in$ **OCMList**, for which the onset time, $t_{\text{on},\,j}$, is equal to a particular value. In other words, no two

```
ComputeMorphList(ChromaList, ChromaVectorList, n)
1   MorphList ← $\bigoplus_{j=1}^{n} \langle \text{nil} \rangle$
    ▶ 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 equation 8 for $0 \le c_t \le 11$.
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 equation 1 for $0 \le c_t \le 11$.
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 equation 3 for $0 \le m \le 6$.
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 equation 4 for $0 \le m \le 6$.
22      for $m$ ← 0 to 6
23          MorphStrength[$m$] ← $\sum_{c_t \in \text{TonicChromaSetForMorph}[m]}$ ChromaVectorList[$j$][$c_t$]
        ▶ Compute equation 5.
24      MorphList[$j$] ← Pos(Max(MorphStrength), MorphStrength)
25  return MorphList
```

Fig. 11. The ComputeMorphList function.

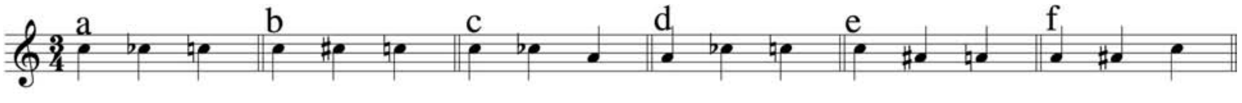

Fig. 12. Examples of the types of passing and neighbour note errors corrected in Stage 2 of *ps13*.

```
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
```

Fig. 13. The ComputeOCMChordList function.

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 14, takes **OCMChordList** as input and corrects neighbour-note voice-leading errors such as the ones in Figures 12(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 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 15, 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 15 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 16, is then used in line 8 of PS13 to correct (some of the) passing note errors like examples (c) and (e) in Figure 12. Again, let's suppose that $\mathbf{OCMChordList} = \langle H_0, H_1, \ldots, H_{N_{\mathrm{Ch}}-1} \rangle$. Like CorrectNeighbourNotes, CorrectDownwardPassingNotes

```
CorrectNeighbourNotes(OCMChordList, $N_{\mathrm{Ch}}$)
1   $i \leftarrow 0$
2   while $i < N_{\mathrm{Ch}} - 2$
3       for $j \leftarrow 0$ to $|\mathbf{OCMChordList}[i]| - 1$
4           $\mathbf{Note}_1 \leftarrow \mathbf{OCMChordList}[i][j]$
5           if $\mathbf{Note}_1[1,3] \in \bigcup_{k=0}^{|\mathbf{OCMChordList}[i+2]|-1} \{\mathbf{OCMChordList}[i+2][k][1,3]\}$
6               for $\ell \leftarrow 0$ to $|\mathbf{OCMChordList}[i+1]| - 1$
7                   $\mathbf{Note}_2 \leftarrow \mathbf{OCMChordList}[i+1][\ell]$
8                   if $\mathbf{Note}_1[2] = \mathbf{Note}_2[2]$
9                       if $(\mathbf{Note}_2[1] - \mathbf{Note}_1[1]) \bmod 12 \in \{1,2\}$
10                          $\mathbf{OCMChordList}[i+1][\ell][2] \leftarrow (\mathbf{Note}_2[2] + 1) \bmod 7$
11                      if $(\mathbf{Note}_1[1] - \mathbf{Note}_2[1]) \bmod 12 \in \{1,2\}$
12                          $\mathbf{OCMChordList}[i+1][\ell][2] \leftarrow (\mathbf{Note}_2[2] - 1) \bmod 7$
13      $i \leftarrow i + 1$
14  return OCMChordList
```

Fig. 14. The CorrectNeighbourNotes function.

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 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).

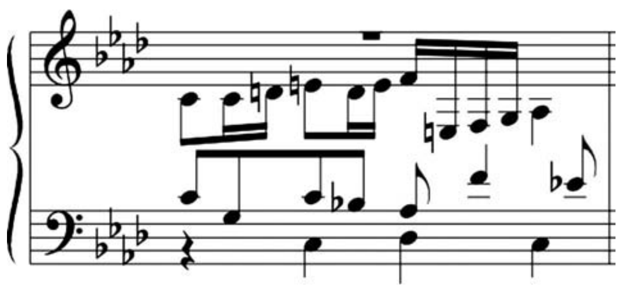

Fig. 15. Bar 7 from J. S. Bach's Fugue in F minor (BWV 857), chosen to illustrate problems with CorrectNeighbourNotes function.

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 17, is then used in line 9 of PS13 to correct (some of the) passing note errors like examples (d) and (f) in Figure 12. 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 18.

Then, in line 11 of PS13, this corrected **MorphList** is given, together with **SortedOCPList**, as input to the function ComputeMorpheticPitchList, defined in Figure 19, 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. However, if the difference between the morphetic octave and the chromatic octave is more

```
CorrectDownwardPassingNotes($\mathbf{OCMChordList}$, $N_{\mathrm{Ch}}$)
1   $i \leftarrow 0$
2   while $i < N_{\mathrm{Ch}} - 2$
3       for $j \leftarrow 0$ to $|\mathbf{OCMChordList}[i]| - 1$
4           $\mathbf{Note}_1 \leftarrow \mathbf{OCMChordList}[i][j]$
5           for $k \leftarrow 0$ to $|\mathbf{OCMChordList}[i+2]| - 1$
6               $\mathbf{Note}_3 \leftarrow \mathbf{OCMChordList}[i+2][k]$
7               if $\mathbf{Note}_3[2] = (\mathbf{Note}_1[2] - 2) \bmod 7$
8                   for $\ell \leftarrow 0$ to $|\mathbf{OCMChordList}[i+1]| - 1$
9                       $\mathbf{Note}_2 \leftarrow \mathbf{OCMChordList}[i+1][\ell]$
10                      if $(\mathbf{Note}_2[2] \in \{\mathbf{Note}_1[2], \mathbf{Note}_3[2]\})$
                              $\wedge (0 < (\mathbf{Note}_1[1] - \mathbf{Note}_2[1]) \bmod 12 < (\mathbf{Note}_1[1] - \mathbf{Note}_3[1]) \bmod 12)$
11                          $CanChange \leftarrow$ true
12                          for $m \leftarrow 0$ to $|\mathbf{OCMChordList}[i+1]| - 1$
13                              $\mathbf{Note} \leftarrow \mathbf{OCMChordList}[i+1][m]$
14                              if $(\mathbf{Note}[2] = (\mathbf{Note}_1[2] - 1) \bmod 7) \wedge (\mathbf{Note}[1] \neq \mathbf{Note}_2[1])$
15                                  $CanChange \leftarrow$ false
16                          if $CanChange$
17                              $\mathbf{OCMChordList}[i+1][\ell][2] \leftarrow (\mathbf{Note}_1[2] - 1) \bmod 7$
18      $i \leftarrow i + 1$
19  return $\mathbf{OCMChordList}$
```

Fig. 16. The CorrectDownwardPassingNotes function.

```
CorrectUpwardPassingNotes(OCMChordList, $N_{\text{Ch}}$)
$i \leftarrow 0$
while $i < N_{\text{Ch}} - 2$
    for $j \leftarrow 0$ to $|\textbf{OCMChordList}[i]| - 1$
        $\textbf{Note}_1 \leftarrow \textbf{OCMChordList}[i][j]$
        for $k \leftarrow 0$ to $|\textbf{OCMChordList}[i+2]| - 1$
            $\textbf{Note}_3 \leftarrow \textbf{OCMChordList}[i+2][k]$
            if $\textbf{Note}_3[2] = (\textbf{Note}_1[2] + 2) \bmod 7$
                for $\ell \leftarrow 0$ to $|\textbf{OCMChordList}[i+1]| - 1$
                    $\textbf{Note}_2 \leftarrow \textbf{OCMChordList}[i+1][\ell]$
                    if $(\textbf{Note}_2[2] \in \{\textbf{Note}_1[2], \textbf{Note}_3[2]\})$
                            $\wedge (0 < (\textbf{Note}_3[1] - \textbf{Note}_2[1]) \bmod 12 < (\textbf{Note}_3[1] - \textbf{Note}_1[1]) \bmod 12)$
                        $CanChange \leftarrow$ true
                        for $m \leftarrow 0$ to $|\textbf{OCMChordList}[i+1]| - 1$
                            $\textbf{Note} \leftarrow \textbf{OCMChordList}[i+1][m]$
                            if $(\textbf{Note}[2] = (\textbf{Note}_1[2] + 1) \bmod 7) \wedge (\textbf{Note}[1] \neq \textbf{Note}_2[1])$
                                $CanChange \leftarrow$ false
                        if $CanChange$
                            $\textbf{OCMChordList}[i+1][\ell][2] \leftarrow (\textbf{Note}_1[2] + 1) \bmod 7$
    $i \leftarrow i + 1$
return OCMChordList
```

Fig. 17. The CorrectUpwardPassingNotes function.

```
ComputeNewMorphListFromOCMChordList(OCMChordList, $N_{\text{Ch}}$)
$\textbf{OCMList} \leftarrow \langle\,\rangle$
for $i \leftarrow 0$ to $N_{\text{Ch}} - 1$
    $\textbf{OCMList} \leftarrow \textbf{OCMList} \oplus \textbf{OCMChordList}[i]$
return $\bigoplus_{j=0}^{|\textbf{OCMList}|-1} \langle \textbf{OCMList}[j][2] \rangle$
```

Fig. 18. The ComputeNewMorphListFromOCMChordList function.

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 are 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, $\{o_{\text{c}}, o_{\text{c}}+1, o_{\text{c}}-1\}$, where o_{c}, the chromatic octave, is equal to $\lfloor p_{\text{c}}/12 \rfloor$. We are therefore left with the problem of deciding for each note whether its morphetic octave should be o_{c}, $o_{\text{c}}+1$ or $o_{\text{c}}-1$. Fortunately, this can easily be solved correctly in almost every case by choosing the value of o_{m} for which $|o_{\text{m}} + (m/7) - (o_{\text{c}} + (c/12))|$ is a minimum, where $c = p_{\text{c}} \bmod 12$. Having found an appropriate morphetic octave, o_{m}, for a note, its morphetic pitch, $p_{\text{m}} = m + 7o_{\text{m}}$, 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 20, 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 3.

5. Computational complexity of PS13

As already mentioned near the beginning of Section 4, 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; and 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 $n = |\textbf{SortedOCPList}|$. The worst-case time complexity of Stage 1 of PS13 is therefore $O(n)$.

It can be shown that the worst-case running time of the CorrectNeighbourNotes function, called in line 7 of PS13 and defined in Figure 14, is $O(Cn)$ where C is the maximum number of notes that occur simultaneously in a single chord. It can also be shown that the worst case running time of CorrectDownwardPassingNotes is $O(C^3n)$, that is, linear in the number of notes in the input passage, but cubic in the maximum number of notes that occur simultaneously in a chord. This suggests that Stage 2 of PS13 may become impractically slow if the number of notes that occur simultaneously in the music is extremely high. CorrectUpwardPassingNotes clearly has the same worst-case running time as CorrectDownwardPassingNotes–that is, $O(C^3n)$ (Meredith, 2006).

Each of the functions ComputeNewMorphListFrom OCMChordList (Figure 18), ComputeMorpheticPitchList (Figure 19) and ComputeOPNList (Figure 20) 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 that occur simultaneously in the music.

It can readily be shown that the worst-case space complexity of PS13 is $O(n)$.

```
ComputeMorpheticPitchList(SortedOCPList, MorphList, $n$)
    MorpheticPitchList $\leftarrow \langle\,\rangle$
    for $i \leftarrow 0$ to $n-1$
        $ChromaticPitch \leftarrow$ SortedOCPList$[i][1]$
        $Morph \leftarrow$ MorphList$[i]$
        $MorphOct_1 \leftarrow \lfloor ChromaticPitch/12 \rfloor$
        $MorphOct_2 \leftarrow MorphOct_1 + 1$
        $MorphOct_3 \leftarrow MorphOct_1 - 1$
        $MP_1 \leftarrow MorphOct_1 + (Morph/7)$
        $MP_2 \leftarrow MorphOct_2 + (Morph/7)$
        $MP_3 \leftarrow MorphOct_3 + (Morph/7)$
        $Chroma \leftarrow ChromaticPitch$ mod 12
        $CP \leftarrow MorphOct_1 + (Chroma/12)$
        DiffList $\leftarrow \langle |CP - MP_1|, |CP - MP_2|, |CP - MP_3| \rangle$
        MorphOctList $\leftarrow \langle MorphOct_1, MorphOct_2, MorphOct_3 \rangle$
        $BestMorphOct \leftarrow$ MorphOctList[Pos(Min(DiffList), DiffList)]
        $BestMorpheticPitch \leftarrow Morph + (7 \times BestMorphOct)$
        MorpheticPitchList $\leftarrow$ MorpheticPitchList $\oplus \langle BestMorpheticPitch \rangle$
    return MorpheticPitchList
```

Fig. 19. The ComputeMorpheticPitchList function.

```
ComputeOPNList(SortedOCPList, MorpheticPitchList, $n$)
    OPNList $\leftarrow \langle\,\rangle$
    for $i \leftarrow 0$ to $n-1$
        $PN \leftarrow$ P2PN($\langle$SortedOCPList$[i][1]$, MorpheticPitchList$[i]\rangle$)
        OPNList $\leftarrow$ OPNList $\oplus \langle\langle$SortedOCPList$[i][0]$, $PN\rangle\rangle$
    return OPNList
```

Fig. 20. The ComputeOPNList function.

6. Methodology used for evaluating and comparing algorithms

When comparing algorithms, one must first 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. When comparing pitch spelling algorithms, these performance metrics are used to measure how well an algorithm performs on some specified test corpus of works. In the study described here, the test corpus, which I denote by $\mathcal{C}$, contained 195972 notes, consisting of 216 movements from works by eight baroque and classical composers (Corelli, Vivaldi, Telemann, J. S. Bach, Handel, Haydn, Mozart and Beethoven). This corpus was chosen so that it contained almost exactly 24500 notes for each of the eight composers represented. This corpus was derived by automatic conversion from files in the MuseData collection of encoded scores (Hewlett, 1997).[1]

In this paper, I use two principal evaluation criteria: *spelling accuracy*, that is, how well an algorithm predicts the pitch names of the notes; and *style dependence*, that is, how much the spelling accuracy of an algorithm is affected by the style of the music being processed. The performance metrics used here to measure spelling accuracy are *note error count* and *note accuracy*. The note error count, NEC(A, S), of an algorithm A over a set of movements S is defined to be the total number of notes in S spelt incorrectly by A. The note accuracy, NA(A, S), of an algorithm A over a set of movements S is defined to be the proportion of notes in S spelt correctly by A. In this study, the note accuracies were measured over the complete test corpus and over each of the eight subsets of this corpus containing the works by a particular composer. The standard deviation of the note accuracies over these eight composer subsets was used as a measure of style dependence (SD_{Sty}).

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.

Occasionally in tonal music, a modulation occurs that results in a passage being in an extremely flat or extremely sharp key that requires many double flats or double sharps. Composers often choose to notate such passages in enharmonically equivalent keys that require

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

fewer accidentals because this usually makes the music easier to read. When this happens, a sudden enharmonic change occurs in the score in which the notated key suddenly changes to a very distant key even though no such modulation is heard by the listener (Temperley, 2001, p. 135). The test corpus used here contained just one example of such a sudden enharmonic change. This occurs at bar 166 in 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, even though no change in key is perceived by the listener. As no modulation is perceived by the listener, it can be argued that spelling this movement without the enharmonic change (i.e. staying in D♭ major) would also be correct. It was therefore decided that the output of each algorithm for this movement should be compared 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 the enharmonic change is omitted. When an algorithm performed better on the modified version than on the original, an alternative value for its note accuracy will be given in the results.

Most of the various versions of *ps13* tested in this study achieved note accuracies higher than 99% on $\mathcal{C}$ which prompts one to question whether the differences between these values are statistically significant. To date, only Meredith (2005) has attempted to measure the statistical significance of the difference between the spelling accuracies achieved by pitch spelling algorithms. However, he used McNemar's test for this purpose (McNemar, 1969, pp. 54–8) and this test is not strictly appropriate in this situation because its validity depends on the correctness of any particular pitch name being independent of the correctness of the pitch names assigned to the notes around it – which is usually not the case since many pitch spelling algorithms (including *ps13*) typically use the context surrounding a note to determine its pitch name. In fact, it seems that there is no straightforward statistical method that is entirely appropriate for giving a reliable estimate of the significance of the difference between two spelling accuracies achieved over $\mathcal{C}$. In order to avoid the risk of giving misleading estimates of significance, I have therefore decided not to provide such estimates in this paper (for more discussion on this issue and some estimates of significance, see Meredith, 2006, pp. 24–28, 321–323).

The algorithms were also evaluated for their robustness to temporal deviations of the type that typically occurs in MIDI files derived from human performances. This was done by running the algorithms on a version of $\mathcal{C}$ in which the onset times and durations had been randomly adjusted by small amounts. This will be discussed in more detail in Section 11 below.

7. Results of running PS13 on the test corpus $\mathcal{C}$

Meredith (2003) 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 3000 notes for each of the 8 composers represented. This 24000 note subset of $\mathcal{C}$ will be denoted by $\mathcal{C}_{\text{Samp}}$. 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 1 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.

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 the first book of Bach's *Das Wohltemperirte Clavier*. Tables 2 and 3 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 3 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 3);
2. the combinations with the smallest K_{pre} (3) performed worst;

Table 1. 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 $\mathcal{C}_{\text{Samp}}$.

	K_{pre}	K_{post}	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	SD_{Sty}
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 2. 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.

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

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 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.

Table 3. 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}").

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

8. Improving PS13

8.1 Choosing between the most strongly implied morphs for a note

As discussed near the end of Section 4.1 above in relation to the ComputeMorphList function, the way in which PS13 chooses between the most strongly implied morphs for a note is arbitrary. 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 7 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 4 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 8). 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,

$$\mathbf{Context}_j = \mathbf{SortedOCPList}[\mathrm{Max}(\{0, j - K_{\mathrm{pre}}\}), \mathrm{MIN}(\{n, j + K_{\mathrm{post}}\})].$$

In PS13, this is done using the COMPUTEMORPHLIST function, defined in Figure 11, which implements the strategy summarized in equations (1)–(8). Let $\mathbf{M}_j$ denote the ordered set of distinct values of m (sorted into ascending order) for which $S(m,j) = \mathrm{MAX}(\langle S(0,j), S(1,j), \ldots, S(6,j)\rangle)$ (see equation (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, $m(c_t, j)$, (see equation (1)), for a chroma, c_j, is equal to the frequency with which c_t occurs within $\mathbf{Context}_j$, this frequency being stored in **ChromaVectorList**[j][c_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 $\mathbf{Context}_j$ implies the morph, $m(c_t, j)$, 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 $\mathbf{Context}_j$, then the one that influences the spelling of c_j the most will be the one for which

$$T(c_t, j) = \sum_{k \in K(c_t, j)} \mathrm{ABS}(t_k - t_j) \qquad (11)$$

is a minimum, where

$$K(c_t, j) = \{k | c_k = c_t \wedge \mathrm{MAX}(\{0, j - K_{\mathrm{pre}}\}) \leq k < \mathrm{MIN}(\{n, j + K_{\mathrm{post}}\})\}. \qquad (12)$$

In turn, this suggests that the best value, $m \in \mathbf{M}_j$, to assign to m_j may be the one for which

$$\mathcal{T}(m, j) = \sum_{c_t \in C_t(m,j)} T(c_t, j) \qquad (13)$$

is a minimum. This strategy was implemented in a modified version of PS13, called PS13B. PS13B was run on the test corpus, $\mathcal{C}$, with $\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle$ set to $\langle 10, 42\rangle$ and the results are shown in Table 4. As can be seen in Table 4, 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.

8.2 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

Table 4. 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 $\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle$ 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.

(a) Note error counts

Algorithm	$\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle$	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	$\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle$	Bach	Beethoven	Corelli	Handel	Haydn	Mozart	Telemann	Vivaldi	Complete	$\mathrm{SD}_{\mathrm{Sty}}$
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

PS13 between the most strongly implied morphs for a note.

However, Meredith (2006, pp. 210–218) showed 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 (see also Meredith and Wiggins, 2005, p. 286). 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 21.

PS1303 takes the same three parameters as PS13 (see Section 4). 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 '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 $\ell(p)+12$ and $\ell(p)-12$ are the line-of-fifths positions of the pitch names that result when p is transposed by a falling and rising diminished second, respectively. 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 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.

For the untransposed line-of-fifths position, stored in *LOF*, the context, stored in the variable, **Context**, in line 12, is set to equal **LOFList**[*ContextStart*, *ContextEnd*], where $\mathit{ContextStart} = \textsc{Max}(\{0, j-K_{\text{pre}}\})$ and $\mathit{ContextEnd} = \textsc{Min}(\{n, j+K_{\text{post}}\})$, 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

```
PS1303(SortedOCPList, $K_{\text{pre}}$, $K_{\text{post}}$)
1    SortedOPNList ← PS13(SortedOCPList, $K_{\text{pre}}$, $K_{\text{post}}$)
2    $n$ ← |SortedOCPList|
3    OnsetList ← $\bigoplus_{j=0}^{n-1} \langle \mathbf{SortedOPNList}[j][0]\rangle$
4    LOFList ← $\bigoplus_{j=0}^{n-1} \langle \mathrm{PNC2LOF}(\mathrm{PN2PNC}(\mathbf{SortedOPNList}[j][1]))\rangle$
5    NewLOFList ← $\bigoplus_{j=0}^{n-1} \langle \mathsf{nil}\rangle$
6    for $j \leftarrow 0$ to $n-1$
7        $LOF \leftarrow \mathbf{LOFList}[j]$
8        $LOFRD2 \leftarrow LOF - 12$
9        $LOFFD2 \leftarrow LOF + 12$
10       $ContextStart \leftarrow \textsc{Max}(\{0, j - K_{\text{pre}}\})$
11       $ContextEnd \leftarrow \textsc{Min}(\{n, j + K_{\text{post}}\})$
12       Context ← LOFList[$ContextStart$, $ContextEnd$]
13       ContextForRD2AndFD2 ← LOFList[$ContextStart, j$] $\oplus \langle 0\rangle \oplus$ LOFList[$j+1$, $ContextEnd$
14       OnsetContext ← OnsetList[$ContextStart$, $ContextEnd$]
15       $s$ ← |Context|
16       $LOFDist \leftarrow \sum_{k=0}^{s-1} \frac{\textsc{Abs}(\mathbf{Context}[k]-LOF)}{\textsc{Max}(\{1,\textsc{Abs}(\mathbf{OnsetList}[j]-\mathbf{OnsetContext}[k])\})}$
17       $LOFDistRD2 \leftarrow \sum_{k=0}^{s-1} \frac{\textsc{Abs}(\mathbf{ContextForRD2AndFD2}[k]-LOFRD2)}{\textsc{Max}(\{1,\textsc{Abs}(\mathbf{OnsetList}[j]-\mathbf{OnsetContext}[k])\})}$
18       $LOFDistFD2 \leftarrow \sum_{k=0}^{s-1} \frac{\textsc{Abs}(\mathbf{ContextForRD2AndFD2}[k]-LOFFD2)}{\textsc{Max}(\{1,\textsc{Abs}(\mathbf{OnsetList}[j]-\mathbf{OnsetContext}[k])\})}$
19       if $LOFDistFD2 < LOFDist$
20           NewLOFList[$j$] ← $LOFFD2$
21       else
22           if $LOFDistRD2 < LOFDist$
23               NewLOFList[$j$] ← $LOFRD2$
24           else
25               NewLOFList[$j$] ← $LOF$
26   return $\bigoplus_{j=0}^{n-1} \langle\langle \mathbf{OnsetList}[j], \mathrm{LOFCP2PN}(\mathbf{NewLOFList}[j], \mathbf{SortedOCPList}[j][1])\rangle\rangle$
```

Fig. 21. The PS1303 algorithm.

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 minimizes 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 (see lines 16 – 18 in Figure 21).

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 5, 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.

8.3 Removing Stage 2 from *ps13*

As discussed in Section 4.2 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 4.2 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 5, 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 $O(C^3n)$ 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.

Table 5. 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.

(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

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.

The lines implementing Stage 2 were therefore deleted from PS13 to produce the PS13s1 algorithm, defined in Figure 22. A new version of PS1303 was also produced in which the call to PS13 in the first line (see Figure 21) was replaced with a call to PS13s1. This version of the algorithm is called 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, 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 2 and 3. 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 2 and 7 reveals that, for each of the 17 $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ combinations tested, PS13s1 made 15–19% fewer errors than PS13. Similarly, comparing Tables 3 and 8 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, tolerance to temporal deviation and ease of implementation.

```
PS13s1(SortedOCPList, $K_{pre}$, $K_{post}$)
    $n \leftarrow$ |SortedOCPList|
    ChromaList $\leftarrow$ ComputeChromaList(SortedOCPList, $n$)
    ChromaVectorList $\leftarrow$ ComputeChromaVectorList(ChromaList, $K_{pre}$, $K_{post}$, $n$)
    MorphList $\leftarrow$ ComputeMorphList(ChromaList, ChromaVectorList, $n$)
    MorpheticPitchList $\leftarrow$ ComputeMorpheticPitchList(SortedOCPList, MorphList, $n$)
    return ComputeOPNList(SortedOCPList, MorpheticPitchList, $n$)
```

Fig. 22. The PS13s1 algorithm.

Table 6. 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.

(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 7. 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 2). 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.

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 8. 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 3). 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}").

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

9. Baseline algorithms

The results reported above 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 9 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. Thus, the number of note errors made by BlackNotesSharp is 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 is 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 Random FlatSharp 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.[2] FixedLOFRange takes a single parameter, $\ell_{\min}$, which specifies the least line-of-fifths position permitted. The note error count for FixedLO-FRange 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 is exactly equivalent to FixedLOFRange with $\ell_{\min} = 0$ and Black NotesFlat is exactly equivalent to FixedLOFRange with $\ell_{\min} = -5$. Table 9 shows the results that would be obtained with FixedLOFRange on $\mathcal{C}$ for all values of $\ell_{\min}$ between -6 and 1, inclusive. Note that Fixed LOFRange performed best when $\ell_{\min}$ 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 $\ell_{\min}$ 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 23 show how the total note error count, the total note accuracy and the style dependence for FixedLOFRange over $\mathcal{C}$ varied with $\ell_{\min}$. Note that both the note error count and the style dependence increase steadily as $\ell_{\min}$ moves further away from -2.[3]

[2]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 Bx in the first movement of Haydn's String Quartet in A major, Hob. III:60 (Op. 55, No. 1).

[3]I am grateful to Alan Marsden for pointing out that the fact that FixedLOFRange performed best over $\mathcal{C}$ with $\ell_{\min} = -2$ is not 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 $\ell_{\min} = -2$.

10. Comparing *ps13* with other algorithms

Table 10 summarizes the results obtained on $\mathcal{C}$ for the best versions of *ps13* and selected versions of the algorithms proposed by Cambouropoulos (1996, 1998, 2001, 2003), Longuet-Higgins (1976, 1987a, 1993), Temperley (2001) and Chew and Chen (2003a,b, 2005). The results are sorted into ascending order of overall note error count. This table gives the results for those versions of each algorithm that performed best in a comparison carried out by Meredith (2006). The table also gives results 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 6). These parenthesized values are only given for those algorithms that performed better when this enharmonic change was omitted from the ground truth.

As can be seen in Table 10, the algorithm in Meredith's (2006) 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 to 1105 but the style dependence dropped to 0.45. 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.

Pitch spelling can be achieved by using the `meter` and `harmony` programs in Temperley and Sleator's *Melisma* music analysis system, which implements Temperley's (2001) computational theory of music cognition.[4] The *Melisma* system can be used in two different ways for pitch spelling. First, pitch names can be computed by first running the `meter` program on the input passage and then running the `harmony` program on the output of the `meter` program. I denote this procedure by "MH". 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". Meredith (2006) ran both MH and MH2P 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

[4]The *Melisma* system is available online at <http://www.link.cs.cmu.edu/music-analysis/>

Table 9. Results obtained when baseline algorithms applied to the test corpus $\mathcal{C}$.

Algorithm	*Note error counts*									*Note accuracies (%)*									
	Bach 24505	Beet 24493	Core 24493	Hand 24500	Hayd 24490	Moza 24494	Tele 24500	Viva 24497	Comp 195972	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	$\mathrm{SD}_{\mathrm{Sty}}$
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

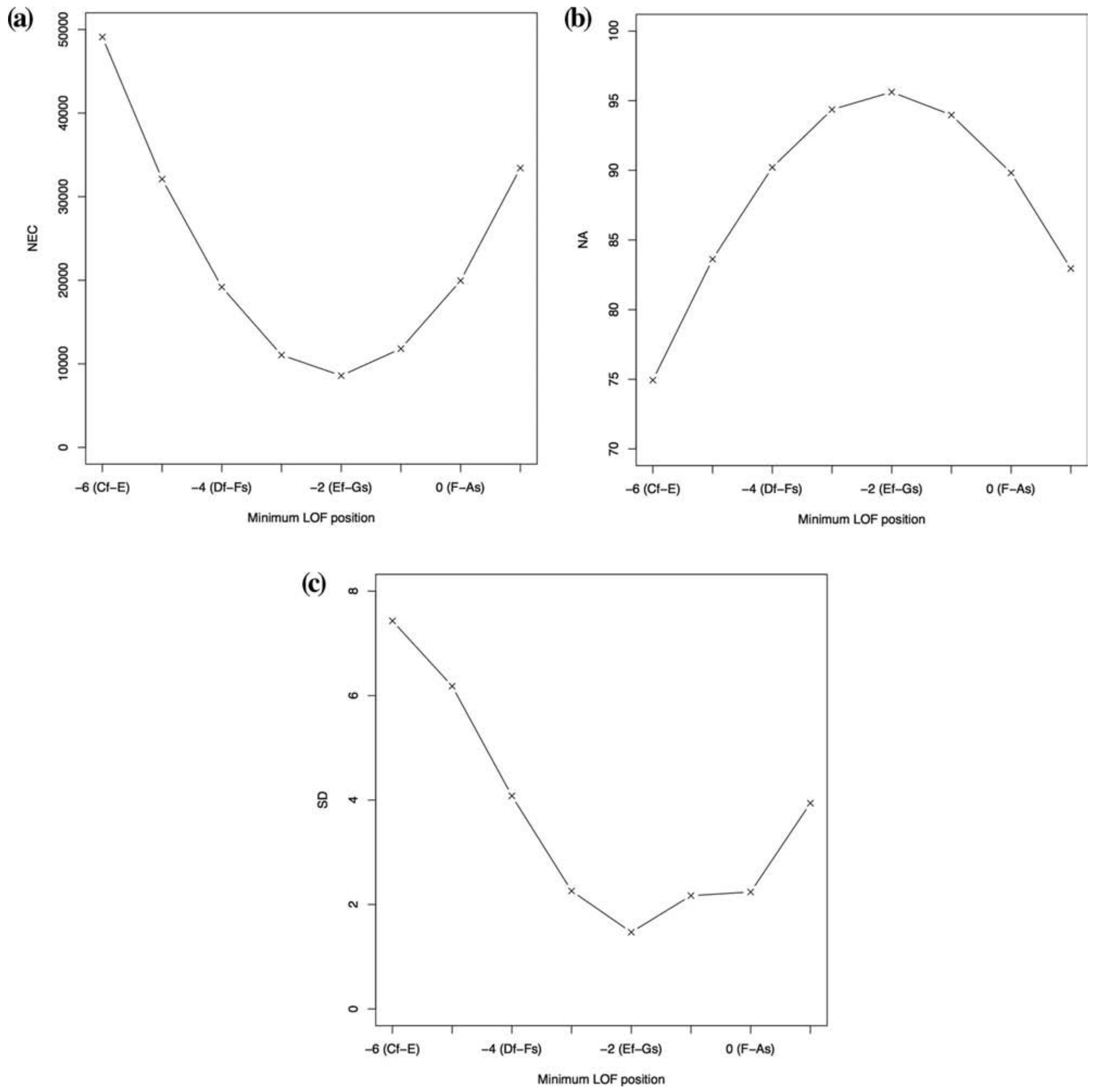

Fig. 23. Graphs showing how (a) total note error count, (b) note accuracy and (c) style dependence varied with ℓ_{min} for FIXEDLOFRANGE when run on the test corpus $\mathcal{C}$.

tempo was multiplied by 2, 4, $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{6}$. The best results were obtained when MH and MH2P were run on either the natural tempo version of $\mathcal{C}$ (see entries for MH and MH2P in Table 10) or the version in which all the music was at half speed (see entries labelled MHX2 and MH2PX2 in Table 10). Under these conditions, the *Melisma* system achieved a note accuracy of up to 99.30% and a style dependence as low as 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 accuracies achieved by the best versions of Temperley and Sleator's system on $\mathcal{C}$ were only slightly less than that achieved by PS13 when $\langle K_{\mathrm{pre}}, K_{\mathrm{post}} \rangle$ 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 $\mathrm{SD}_{\mathrm{Sty}}$ values for MH2PX2 and PS13 in Table 10). Also,

Table 10. Results obtained over the test corpus, $\mathcal{C}$, for the best versions of *ps13* and optimized versions of the algorithms proposed by Cambouropoulos (1996, 1998, 2001, 2003), Longuet-Higgins (1976, 1987, 1993), Temperley (2001) and Chew and Chen (2003a,b, 2005). The algorithms are sorted into ascending order of overall note error count. The final column, headed $\mathrm{SD_{Sty}}$, gives the style dependence for each algorithm.

	Note error counts									*Note accuracies (%)*									
Algorithm	Bach 24505	Beet 24493	Core 24493	Hand 24500	Hayd 24490	Moza 24494	Tele 24500	Viva 24497	Comp 195972	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	$\mathrm{SD_{Sty}}$
PS13s1 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 10, 42\rangle$)	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 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 33, 25\rangle$)	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 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 10, 42\rangle$)	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 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 10, 42\rangle$)	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 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 40, 1\rangle$)	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 ($\ell_{\mathrm{min}} = -2$)	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 ($\ell_{\mathrm{min}} = 1$)	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 ($\ell_{\mathrm{min}} = -6$)	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

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 10). 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 (see Section 11 below).

Also, Temperley and Sleator's system is by far the most complicated method for pitch spelling that has been proposed in the literature, 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 system.

The twelve most accurate versions of Chew and Chen's (2003a,b, 2005) algorithm tested by Meredith (2006) spelt 99.15% of the notes in $\mathcal{C}$ correctly (see entries for CCOP01 – 06 and CCOP07 – 12 in Table 10). For these best versions of the algorithm, the line of fifths worked just as well as the spiral array. Of these 12 best versions, the 6 in which all the notes *sounding* in each window were considered when calculating the centres of effect (CCOP01 – 06) were slightly less dependent on style than those which only used the notes *starting* in each window (CCOP07 – 12) (see Table 10). 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 Meredith's (2006) study. Moreover, Chew and Chen's algorithm is 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 over $\mathcal{C}$ than PS13s1. Also, Chew and Chen's algorithm, unlike PS13s1, uses the duration of each note to weight its contribution when calculating a centre 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 made sufficiently small. Indeed, it seems that even with K_{post} set to 1, PS13s1 is more accurate than Chew and Chen's algorithm: with $\langle K_{\text{pre}}, K_{\text{post}}\rangle$ set to $\langle 40, 1\rangle$, PS13s1 spelt 99.19% of the notes in the test corpus correctly, making over 4% fewer errors than Chew and Chen's algorithm (see Table 10).

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.

The fourth 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 centre 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 pointed out above, the line of fifths worked just as well as the spiral array in Meredith's (2006) evaluation). 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 "centre of gravity" (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 line-of-fifths 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 one of the principles underlying Longuet-Higgins's pitch spelling algorithm (Longuet-Higgins, 1987, pp. 112–113). 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.

The most accurate versions of Cambouropoulos's (1996, 1998, 2001, 2003) algorithm tested by Meredith (2006) differ from *ps13* and the algorithms of Temperley, Longuet-Higgins and Chew and Chen 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 penalizing spellings for containing intervals that occur infrequently in the major and minor scales. However, the principle of notational parsimony involves penalizing a spelling if it involves using more than one sharp or flat, and this is equivalent to penalizing 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 by Meredith (2006) was the CamOpt algorithm, which Meredith developed by combining the best-performing features of the other versions of the algorithm tested in his evaluation. When CamOpt was run on $\mathcal{C}$, it made 8% fewer errors than the most accurate of the other versions of the algorithm tested by Meredith (2006) and Cambouropoulos (1996, 1998, 2001, 2003). CamOpt spelt 99.15% of the notes in the test corpus correctly with a style dependence of 0.47 (see Table 10). 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 (2001) himself (see Table 10). 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 Meredith's (2006) study were amongst the slowest of the algorithms tested – Meredith's (2006) implementation of CamOpt took over 40 h to process $\mathcal{C}$. Cam01A took about 6 h, but this was still extremely slow in comparison with PS13s1, which took around 20 min.

Also, even CamOpt, the most accurate version of Cambouropoulos's algorithm tested by Meredith (2006), 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 $\langle 10, 42 \rangle$.

A straightforward implementation of Temperley's (2001, p. 125) TPR 1, which I call TPROne, 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 by Meredith (2006) 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 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 10). 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).

Unfortunately, the quadratic-time version of TPROne (TPR1A) would be impractically slow for processing very large movements. Also, the most accurate version of TPROne tested (TPR1C) made 67% more errors on $\mathcal{C}$ than PS13s1 and 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.

This change could be implemented 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, 2002, p. 294).

Longuet-Higgins's (1976, 1987, 1993) algorithm, like PS13s1, is based on the idea of spelling 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 (Longuet-Higgins, 1987, pp. 112–114). 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 10). 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 10). 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 10, 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". The hypothesis that voice-leading is of relatively minor importance in pitch spelling is also supported by the fact that removing Stage 2 from PS13 reduced the overall note error count over $\mathcal{C}$ by 19%. 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 *ps13* and the other algorithms considered in Meredith's (2006) study.

11. Testing the algorithms for robustness to temporal deviations

The 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. An "artificially noisy" version of the test corpus $\mathcal{C}$ was therefore generated by automatically 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 denote by $\mathcal{C}'$, was generated, by

1. assigning a "natural" tempo to each 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.

The best versions of the algorithms evaluated by Meredith (2006) were run on $\mathcal{C}'$ and the results are shown in Table 11. The rightmost column in this table gives, for each algorithm, A, the proportional difference in note error count,

$$\begin{aligned}\Delta\mathrm{NEC}_{\mathrm{Pr}}(\mathrm{NEC}(A,\mathcal{C}),\mathrm{NEC}(A,\mathcal{C}'))\\ =\frac{\mathrm{NEC}(A,\mathcal{C}')-\mathrm{NEC}(A,\mathcal{C})}{\mathrm{NEC}(A,\mathcal{C})}.\end{aligned} \quad (14)$$

A higher positive value of $\Delta\mathrm{NEC}_{\mathrm{Pr}}(\mathrm{NEC}(A,\mathcal{C}), \mathrm{NEC}(A,\mathcal{C}'))$ 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 11 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, $\mathcal{C}'$. As in Table 10, the results in parentheses in Table 11 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 6).

From Table 11 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 Meredith's (2006) 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 4.2). 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 10 that PS13s1 would be robust to the types of temporal noise typically present in data derived from performances.

Of the algorithms tested, Temperley and Sleator's *Melisma* 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 $\ell_{\min} = -2$). 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 count 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 10 that the *Melisma* programs 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 over $\mathcal{C}'$ 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 by Meredith, 2006, Chapter 5) 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 (Chew, 2004).

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

Table 11. Results obtained when the best versions of the algorithms evaluated by Meredith (2006) were run on the noisy test corpus, $\mathcal{C}'$. The algorithms are sorted into ascending order of overall note error count. The rightmost column gives, for each algorithm, A, the proportional difference, $\Delta\mathrm{NEC}_{\mathrm{Pr}}(\mathrm{NEC}(A,\mathcal{C}),\mathrm{NEC}(A,\mathcal{C}'))$, as defined in equation (14).

	Note error counts									*Note accuracies (%)*										
Algorithm	Bach 24505	Beet 24493	Core 24493	Hand 24500	Hayd 24490	Moza 24494	Tele 24500	Viva 24497	Comp 195972	Bach	Beet	Core	Hand	Hayd	Moza	Tele	Viva	Comp	$\mathrm{SD}_{\mathrm{Sty}}$	$\Delta\mathrm{NEC}_{\mathrm{Pr}}$ (%)
PS13s1 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 33, 25\rangle$)	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 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 10, 42\rangle$)	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 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 10, 42\rangle$)	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 ($\langle K_{\mathrm{pre}}, K_{\mathrm{post}}\rangle = \langle 10, 42\rangle$)	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 ($\langle \mathrm{K}_{\mathrm{pre}}, \mathrm{K}_{\mathrm{post}}\rangle = \langle 40, 1\rangle$)	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 ($\ell_{\mathrm{min}} = -2$)	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 ($\ell_{\mathrm{min}} = 1$)	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 ($\ell_{\mathrm{min}} = -6$)	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

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 10, 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 counts 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 10, therefore, TPROne is far more robust to performance-derived temporal deviations than the *Melisma* programs. 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.

12. Conclusions and suggestions for further work

The *ps13* pitch spelling 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.

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 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. 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.

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.

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, called PS1303, achieved a spelling accuracy of 99.43% over $\mathcal{C}$ with a style dependence of 0.53.

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 $O(n)$.

It was therefore shown that removing Stage 2 altogether from PS13, leaving just Stage 1, would improve the time complexity of the algorithm to $O(n)$, 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 by Meredith (2006), 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 $\ell_{\min} = -2$), 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.

It was 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.

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.

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. In this study, it achieved the best results on 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.

All the pitch spelling algorithms proposed to date in the literature use the line of fifths in one way or another. One might therefore reasonably wonder whether it would 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 (Meredith, 2006). 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, it would be interesting 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 by Meredith (2006) because the authors were able to provide neither a working system nor a description of it that was sufficiently detailed for it to be re-implemented.

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. Winograd, 1968, 1993; Holtzmann, 1977; Longuet-Higgins & Steedman, 1987; Krumhansl, 1990; Maxwell, 1992,; Vos & van Geenen, 1996; Temperley, 1997, 1999, 2001; Temperley & Sleator, 1999; Raphael & Stoddard, 2003). 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.

Acknowledgements

The work reported here was funded by EPSRC grant number GR/S17253/02. I am grateful to Geraint Wiggins and the other members of the ISMS group at Goldsmiths College, University of London, for their support during this project. I would also like to thank Alan Marsden for providing detailed feedback on an earlier draft of this paper.

References

Abdallah, S.A. & Plumbley, M.D. (2004). Polyphonic transcription by non-negative sparse coding of power spectra. *Proceedings of the Fifth International Conference on Music Information Retrieval (ISMIR 2004)*, Barcelona: Universitat Pompeu Fabra.

Associated Board of the Royal Schools of Music (1958). *Rudiments and Theory of Music*. London: Associated Board of the Royal Schools of Music.

Babbitt, M. (1965). The structure and function of musical theory: I. *College Music Symposium*, *5*, 49–60.

Backus, J. (1977). *The Acoustical Foundations of Music*, 2nd Edn. New York: Norton (1st Edn published in 1969).

Boyer, R.S. & 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*. Chicago, London: The University of Chicago Press.

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. Paper presented at VIII Brazilian Symposium on Computer Music (SBC&M 2001), Fortaleza, Brazil.

Cambouropoulos, E. (2003). Pitch spelling: a computational model. *Music Perception*, *20*(4), 411–429.

Cambouropoulos, E., Crochemore, M., Iliopoulos, C.S., Mouchard, L., & 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 December 2004 08:55:00 -0800 sent from Elaine Chew <echew@usc.edu> to Dave Meredith <dave@titanmusic.com>.

Chew, E. & Chen, Y.-C. (2003a). Determining context-defining windows: Pitch spelling using the spiral array. Paper presented at *Fourth International Conference on Music Information Retrieval (ISMIR 2003)*, 26–30 October, Baltimore, MD.

Chew, E. & Chen, Y.-C. (2003b). Mapping MIDI to the Spiral Array: disambiguating pitch spellings. In H. K. Bhargava & N. Ye (Eds.), *Computational Modeling and Problem Solving in the Networked World–Proceedings of the 8th INFORMS Computer Society Conference*, pp. 259–275. Chandler, AZ: Kluwer.

Chew, E. & Chen, Y.-C. (2005). Real-time pitch spelling using the Spiral Array. *Computer Music Journal*, *29*(2), 61–76.

Cormen, T.H., Leiserson, C.E., & Rivest, R.L. (1990). *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Crochemore, M., Iliopoulos, C. S., Lecroq, T., & Pinzon, Y. J. (2001). Approximate string matching in musical sequences. In M. Balik & M. Simanek (Eds.), *Proceedings of the Prague Stringology Club Workshop (PSCW 01)*, pp. 26–36. Prague, Czech Republic: Czech Technical University.

Davy, M. & Godsill, S.J. (2003). Bayesian harmonic models for musical signal analysis (with discussion). In J.M. Bernardo, J.O. Berger, A.P. Dawid & A.F.M. Smith (Eds.), *Bayesian Statistics*, Vol. VII. Oxford: Oxford University Press.

Forte, A. (1973). *The Structure of Atonal Music*. New Haven, London: Yale University Press.

Gabrielsson, A. (1999). The performance of music. In D. Deutsch (Ed.), *The Psychology of Music,* 2nd Edn. pp. 501–602. San Diego: Academic Press.

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.

Hewlett, W. B. (1997). *MuseData*: multipurpose representation. In E. Selfridge-Field (Ed.), *Beyond MIDI: The Handbook of Musical Codes*. pp. 402–447. Cambridge, MA: MIT Press.

Holtzmann, S.R. (1977). A program for key determination. *Interface*, *6*, 26–56.

Kernighan, B.W. & Ritchie, D.M. (1988). *The C Programming Language*. London: Prentice-Hall International.

Knopoff, L. & 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., & Pratt, V.R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, *6*, 323–350.

Krumhansl, C.L. (1990). *Cognitive Foundations of Musical Pitch*, Vol. 17, *Oxford Psychology Series*. New York, Oxford: Oxford University Press.

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

Krumhansl, C.L. & 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.

Longuet-Higgins, H.C. (1976). The perception of melodies. *Nature*, *263*(5579), 646–653. Republished as Longuet-Higgins (1993, 1987).

Longuet-Higgins, H.C. (Ed.) (1987). The perception of melodies. In *Mental Processes: Studies in Cognitive Science*, pp. 105–129. London, Cambridge, MA: British Psychological Society/MIT Press. Based on Longuet-Higgins (1976) and re-published as Longuet-Higgins (1993).

Longuet-Higgins, H.C. (1993). The perception of melodies. In S.M. Schwanauer & D.A. Levitt (Eds.), *Machine Models of Music*. pp. 471–495. Cambridge, MA: MIT Press. Published earlier as Longuet-Higgins (1987, 1976).

Longuet-Higgins, H.C. & Steedman, M.J. (1987). On interpreting Bach. In H.C. Longuet-Higgins (Ed.), *Mental Processes: Studies in Cognitive Science*. pp. 82–104. London and Cambridge, MA: British Psychological Society/MIT Press.

Maxwell, H. J. (1992). An expert system for harmonic analysis of tonal music. In M. Balaban, K. Ebcioğlu & O. Laske (Eds.), *Understanding Music with AI: Perspectives on Music Cognition*. pp. 335–353. Cambridge, MA: AAAI Press/MIT Press.

McNemar, Q. (1969). *Psychological Statistics*, 4th Edn. New York: John Wiley and Sons.

Meredith, D. (2002). Review of *The Cognition of Basic Musical Structures* by David Temperley. Musicae Scientiae, *6*(2), 287–302. Draft available online at <http://www.titanmusic.com/papers/public/temperley_review_3.pdf>

Meredith, D. (2003). Pitch spelling algorithms. In R. Kopiez, A.C. Lehmann, I. Wolther & C. Wolf (Eds.), *Proceedings of the Fifth Triennial ESCOM Conference (ESCOM5)*, 8–13 September, pp. 204–207. Hanover, Germany: Hanover University of Music and Drama. 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 U.K. Wiil (Ed.), *Computer Music Modeling and Retrieval, Second International Symposium, CMMR 2004, Esbjerg, Denmark, May 26–29, 2004, Revised Papers*, Vol. 3310, *Lecture Notes in Computer Science (LNCS)*, pp. 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. (2006). *Computing pitch names in tonal music: a comparative analysis of pitch spelling algorithms*. DPhil dissertation, Faculty of Music, University of Oxford. Draft available online at <http://www.titanmusic.com/papers/private/meredith-dphil.pdf>. To obtain username and password, send request to <dave@titanmusic.com>

Meredith, D., Lemström, K., & 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. & Wiggins, G.A. (2005). Comparing pitch spelling algorithms. Paper presented at *Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR 2005)*, pp. 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*. New Haven, London: Yale University Press.

Oldham, G. & Lindley, M. (2006). Wolf. In L. Macy (Ed.), *Grove Music Online*. Oxford: Oxford University Press. <http://www.grovemusic.com>, accessed 29 May 2006.

Plumbley, M., Abdallah, S., Bello, J., Davies, M.E., Monti, G., & Sandler, M. (2002). Automatic music transcription and audio source separation. *Cybernetics and Systems*, *33*(6), 603–627.

Rahn, J. (1980). *Basic Atonal Theory*. New York: Longman.

Raphael, C. & Stoddard, J. (2003). Harmonic analysis with probabilistic graphical models. Paper presented at *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*. Berkeley, CA: University of California Press.

Stoddard, J., Raphael, C., & 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)*, 10–14 October. Barcelona: Universitat Pompeu Fabra.

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*. Cambridge, MA: MIT Press.

Temperley, D. & 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.

Vos, P.G. 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 S.M. Schwanauer & D.A. Levitt (Eds.), *Machine Models of Music*. pp. 113–153. Cambridge, MA: MIT Press. 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.