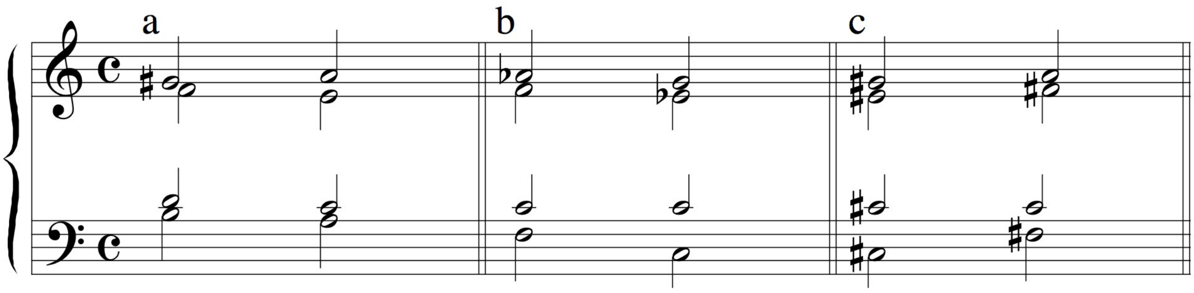


Figure 1: Examples of notes with identical MIDI note numbers being spelt differently in different tonal contexts (from Piston, 1978, p. 8).

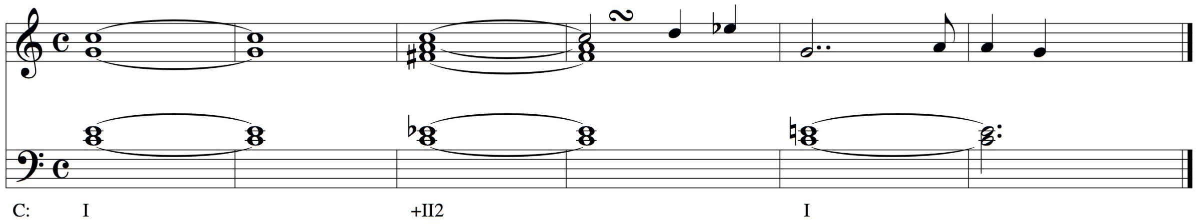
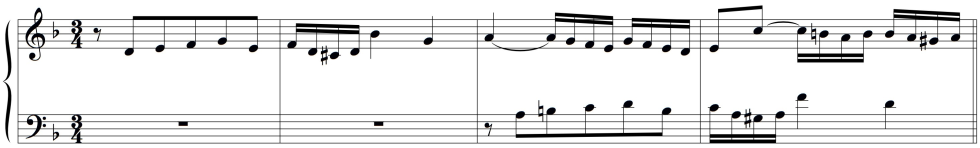


Figure 2: Should the E♭s be spelt as D♯s? (From Piston, 1978, p. 390.)

(a)



(b)

{ ⟨2, D4, 2⟩, ⟨4, E4, 2⟩, ⟨6, F4, 2⟩, ⟨8, G4, 2⟩, ⟨10, E4, 2⟩, ⟨12, F4, 1⟩,
⟨13, D4, 1⟩, ⟨14, CS4, 1⟩, ⟨15, D4, 1⟩, ⟨16, BF4, 4⟩, ⟨20, G4, 4⟩, ⟨24, A4, 5⟩,
⟨29, G4, 1⟩, ⟨30, F4, 1⟩, ⟨31, E4, 1⟩, ⟨32, G4, 1⟩, ⟨33, F4, 1⟩, ⟨34, E4, 1⟩,
⟨35, D4, 1⟩, ⟨36, E4, 2⟩, ⟨38, C5, 3⟩, ⟨41, B4, 1⟩, ⟨42, A4, 1⟩, ⟨43, B4, 1⟩,
⟨44, B4, 1⟩, ⟨45, A4, 1⟩, ⟨46, GS4, 1⟩, ⟨47, A4, 1⟩, ⟨26, A3, 2⟩, ⟨28, B3, 2⟩,
⟨30, C4, 2⟩, ⟨32, D4, 2, ⟩, ⟨34, B3, 2⟩, ⟨36, C4, 1⟩, ⟨37, A3, 1⟩, ⟨38, GS3, 1⟩,
⟨39, A3, 1⟩, ⟨40, F4, 4⟩, ⟨44, D4, 4⟩ }

Figure 3: (a) Bars 1 to 4 of Bach's Fugue in D minor from Book 1 of *Das Wohltemperirte Klavier* (BWV 851). (b) The *OPND* representation of the score in (a).

$S \leftarrow \langle\,\rangle$
$Q \leftarrow \langle 0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6 \rangle$
for $c \leftarrow 0$ **to** 11
  $m_0 \leftarrow 0$
  $s \leftarrow \langle\,\rangle$
  $s' \leftarrow \langle\,\rangle$
  for $i \leftarrow 0$ **to** $|C| - 1$
    $s \leftarrow s \oplus \langle Q[(C[i] - c) \bmod 12] \rangle$
  $m_0 \leftarrow s[0]$
  for $i \leftarrow 0$ **to** $|C| - 1$
    $s' \leftarrow s' \oplus \langle (s[i] - m_0) \bmod 7 \rangle$
  $S \leftarrow S \oplus \langle s' \rangle$
return S

Figure 4: Algorithm for computing the spelling table S.

$R \leftarrow \langle\,\rangle$
`for` $i \leftarrow 0$ `to` $|C| - 1$
    $V \leftarrow \langle 0, 0, 0, 0, 0, 0, 0 \rangle$
    $\mu \leftarrow \langle\,\rangle$
    `for` $k \leftarrow 0$ `to` 11
        $\mu \leftarrow \mu \oplus \langle S[k][i] \rangle$
    `for` $j \leftarrow 0$ `to` 11
        $s_{\text{prev}} \leftarrow V[\mu[j]]$
        $V[\mu[j]] \leftarrow s_{\text{prev}} + \Phi[i][j]$
    $R \leftarrow R \oplus \langle \mathit{POS}(\max(V), V) \rangle$
`return` R

Figure 5: Algorithm for computing the relative morph list R.

$H \leftarrow \langle\langle O[0]\rangle\rangle$
for $i \leftarrow 1$ **to** $|C| - 1$
    if $O[i][0] = O[i-1][0]$
        $H[|H|-1] \leftarrow H[|H|-1] \oplus \langle O[i]\rangle$
    else
        $H \leftarrow H \oplus \langle\langle O[i]\rangle\rangle$
return H

Figure 6: Algorithm for computing the chord list H.

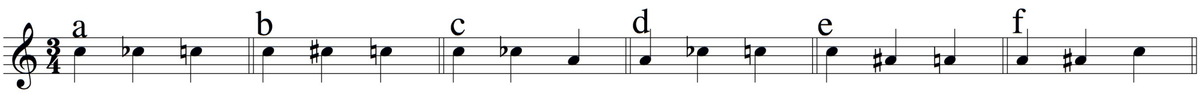


Figure 7: Neighbour note and passing note errors corrected by *ps13*.

```
for i ← 0 to |H| − 3
  ζ ← ⟨ ⟩
  for k ← 0 to |H[i + 2]| − 1
    ζ ← ζ ⊕ ⟨H[i + 2][k][1, 3]⟩
  for n_1 ← 0 to |H[i]| − 1
    if H[i][n_1][1, 3] ∈ ζ
      for n_2 ← 0 to |H[i + 1]| − 1
        if H[i + 1][n_2][2] = H[i][n_1][2]
          if (H[i + 1][n_2][1] − H[i][n_1][1]) mod 12 ∈ {1, 2}
            H[i + 1][n_2][2] ← (H[i + 1][n_2][2] + 1) mod 7
          if (H[i][n_1][1] − H[i + 1][n_2][1]) mod 12 ∈ {1, 2}
            H[i + 1][n_2][2] ← (H[i + 1][n_2][2] − 1) mod 7
return H
```

Figure 8: Algorithm for correcting neighbour note errors.

```
for i ← 0 to |H| − 3
  for n_1 ← 0 to |H[i]| − 1
    for n_3 ← 0 to |H[i + 2]| − 1
      if H[i + 2][n_3][2] = (H[i][n_1][2] − 2) mod 7
        for n_2 ← 0 to |H[i + 1]| − 1
          if H[i + 1][n_2][2] = H[i][n_1][2] or H[i + 1][n_2][2] = H[i + 2][n_3][2]
            if 0 < (H[i][n_1][1] − H[i + 1][n_2][1]) mod 12 < (H[i][n_1][1] − H[i + 2][n_3][1]) mod 12
              ζ ← ⟨ ⟩
              for j ← 0 to |H[i + 1]| − 1
                if H[i + 1][j][2] = (H[i][n_1][2] − 1) mod 7
                  ζ ← ζ ⊕ ⟨H[i + 1][j]⟩
              θ ← ⟨ ⟩
              for j ← 0 to |ζ| − 1
                if H[i + 1][n_2][1] ≠ ζ[j][1]
                  θ ← θ ⊕ ⟨ζ[j]⟩
              if θ = ⟨ ⟩
                H[i + 1][n_2][2] ← (H[i][n_1][2] − 1) mod 7
return H
```

Figure 9: Algorithm for correcting descending passing note errors.

```
for i ← 0 to |H| − 3
  for n_1 ← 0 to |H[i]| − 1
    for n_3 ← 0 to |H[i + 2]| − 1
      if H[i + 2][n_3][2] = (H[i][n_1][2] + 2) mod 7
        for n_2 ← 0 to |H[i + 1]| − 1
          if H[i + 1][n_2][2] = H[i][n_1][2] or H[i + 1][n_2][2] = H[i + 2][n_3][2]
            if 0 < (H[i + 2][n_3][1] − H[i + 1][n_2][1]) mod 12 < (H[i + 2][n_3][1] − H[i][n_1][1]) mod 12
              ζ ← ⟨ ⟩
              for j ← 0 to |H[i + 1]| − 1
                if H[i + 1][j][2] = (H[i][n_1][2] + 1) mod 7
                  ζ ← ζ ⊕ ⟨H[i + 1][j]⟩
              θ ← ⟨ ⟩
              for j ← 0 to |ζ| − 1
                if H[i + 1][n_2][1] ≠ ζ[j][1]
                  θ ← θ ⊕ ⟨ζ[j]⟩
              if θ = ⟨ ⟩
                H[i + 1][n_2][2] ← (H[i][n_1][2] + 1) mod 7
return H
```

Figure 10: Algorithm for correcting ascending passing note errors.

$O' \leftarrow \langle\, \rangle$
for $i \leftarrow 0$ **to** $|H| - 1$
  $O' \leftarrow O' \oplus H[i]$
$M' \leftarrow \langle\, \rangle$
for $i \leftarrow 0$ **to** $|O'| - 1$
  $M' \leftarrow M' \oplus \langle O'[i][2] \rangle$
return M'

Figure 11: Algorithm for computing M'.

$P \leftarrow \langle\,\rangle$
for $i \leftarrow 0$ **to** $|J| - 1$
    $o_1 \leftarrow \lfloor J[i][1]/12 \rfloor$
    $o_2 \leftarrow 1 + o_1$
    $o_3 \leftarrow o_1 - 1$
    $p_1 \leftarrow o_1 + M'[i]/7$
    $p_2 \leftarrow o_2 + M'[i]/7$
    $p_3 \leftarrow o_3 + M'[i]/7$
    $c \leftarrow J[i][1] \bmod 12$
    $p' \leftarrow o_1 + c/12$
    $D \leftarrow \langle |p' - p_1|, |p' - p_2|, |p' - p_3| \rangle$
    $\omega \leftarrow \langle o_1, o_2, o_3 \rangle$
    $o \leftarrow \omega[\mathit{POS}(\min(D), D)]$
    $P \leftarrow P \oplus \langle M'[i] + 7o \rangle$
return P

Figure 12: Algorithm for computing P.

```
PPN(p)
m ← p[1] mod 7
L ← ⟨"A", "B", "C", "D", "E", "F", "G"⟩
l ← L[m]
g_c ← p[0] − 12(⌊p[1]/7⌋)
A ← ⟨0, 2, 3, 5, 7, 8, 10⟩
c′ ← A[m]
e ← g_c − c′
i ← ""
if e < 0
   for j ← 0 to −e − 1
      i ← i ⊕ "f"
else
   if e > 0
      for j ← 0 to e − 1
         i ← i ⊕ "s"
   else
      i ← "n"
o_m ← ⌊p[1]/7⌋
if m = 0 or m = 1
   o ← o_m
else
   o ← 1 + o_m
o_str ← STR(o)
return l ⊕ i ⊕ o_str
```

Figure 13: *PPN* algorithm.

METHOD OF COMPUTING THE PITCH NAMES OF NOTES IN MIDI-LIKE MUSIC REPRESENTATIONS

Field of the invention

This invention relates to the problem of constructing a reliable *pitch spelling algorithm*—that is, an algorithm that reliably computes 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 (The MIDI Manufacturers' Association, 1996, p. 10) and possibly the duration of each note in the passage.

There are good practical and scientific reasons for attempting to develop a reliable pitch spelling algorithm. First, until such an algorithm is devised, it will be impossible to construct a reliable *MIDI-to-notation transcription algorithm*—that is, an algorithm that reliably computes a correctly notated score of a passage when given only a MIDI file of the passage as input. Commercial music notation programs (e.g., Sibelius (`www.sibelius.com`), Coda Finale (`www.codamusic.com`) and Nightingale (`www.ngale.com`)) typically use MIDI-to-notation transcription algorithms to allow the user to generate a notated score from a MIDI file encoding a performance of the passage to be notated. However, the MIDI-to-notation transcription algorithms that are currently used in commercial music notation programs are crude and unreliable. Also, existing audio transcription systems generate not notated scores but MIDI-like representations as output (see, for example, Davy and Godsill, 2003; Plumbley *et al.*, 2002; Walmsley, 2000). So if one wishes to produce a notated score from a digital audio recording, one typically needs a MIDI-to-notation transcription algorithm (incorporating a pitch spelling algorithm) in addition to an audio transcription system.

Knowing the letter-names of the pitch events in a passage is also indispensible in music information retrieval and musical pattern discovery (Meredith *et al.*, 2002). For example, the occurrence of a motive on a different degree of a scale (e.g., C-D-E-C restated as E-F-G-E) might be perceptually significant even if the corresponding chromatic intervals in the patterns differ. Such matches can be found using fast, exact-matching algorithms if the pitch names of the notes are encoded, but exact-matching algorithms cannot be used to find such matches if the pitches are represented using just MIDI note numbers. If a reliable pitch spelling algorithm existed, it could be used to compute the pitch names of the notes in the tens of thousands of MIDI files of works that are freely available online, allowing these files to be searched more effectively by a music information retrieval (MIR) system.

In the vast majority of cases, the correct pitch name for a note in a passage of tonal music can be determined by considering the rôles that the note is perceived to play in the perceived harmonic structure and voice-leading structure of the passage. For example, when played in isolation in an equal-tempered tuning system, the first soprano note in Figure 1a would sound the same as the first soprano note in Figure 1b. However, in Figure 1a, this note is spelt as a G♯4 because it is perceived to function as a leading note in A minor; whereas in Figure 1b, the first soprano note is spelt as an A♭4 because it functions as a submediant in C minor. Similarly, the first alto note in Figure 1b would sound the same as the first alto note in Figure 1c in an equal-tempered tuning system. However, in Figure 1b the first alto note is spelt as an F♮4 because it functions in this context as a subdominant in C minor; whereas, in Figure 1c, the first alto note functions as a leading note in F♯ minor so it is spelt as an E♯4.

Nevertheless, it is not always easy to determine the correct pitch name of a note by considering the harmonic structure and voice-leading structure of its context. For example, as Piston (1978, p. 390) observes, the tenor E♭4 in the third and fourth bars of Figure 2 should be spelt as a D♯4 if one perceives the harmonic progression here to be $^{+}\mathrm{II}^{2} - \mathrm{I}$ as shown. But spelling the soprano E♭5 in the fourth bar as D♯5 would result in a strange melodic line.

Such cases where it is difficult to determine the correct pitch name of a note in a tonal work are relatively rare—particularly in Western tonal music of the so-called 'common practice' period (roughly the 18th and 19th centuries). In the vast majority of cases, those who study and perform Western tonal music agree about how a note should be spelt in a given tonal context. Therefore a pitch spelling algorithm can be evaluated objectively by running it on tonal works and comparing the pitch names it predicts with those of the corresponding notes in authoritative published editions of scores of the works. However, this can only be done accurately and quickly if one has access to encodings of these authoritative scores in the form of computer files that can be compared automatically with the pitch spelling algorithm's output.

Related Art

In this section, the performance of three prior pitch spelling algorithms is compared on a single test corpus of works containing 41544 notes and consisting of all 48 pieces in the first book of J. S. Bach's *Das Wohltemperirte Klavier*. The algorithms compared are those of Cambouropoulos (1996, 1998, 2000, 2001, 2002), Longuet-Higgins (1976, 1987, 1993) and Temperley (1997, 2001).

When these algorithms were run on the test corpus, Cambouropoulos's algorithm made 2599 mistakes, Longuet-Higgins's algorithm made 265 mistakes and Temperley's algorithm made 122 mistakes.

The test corpus The test corpus used in the comparison consists of representations of the scores of all 24 Preludes and all 24 Fugues in the first book of J. S. Bach's *Das Wohltemperirte Klavier* encoded in the author's *OPND* format.[1] Each *OPND* representation is a set of triples, $\langle t, n, d \rangle$, each triple giving the onset time, t, the pitch name, n, and

[1] *OPND* stands for "onset, pitch-name, duration".

the duration, d, of a single note (or sequence of tied notes) in the score.[2] The onset time and duration of each note are expressed as integer multiples of the largest common divisor of all the notated onset times and note durations in the score. For example, Figure 3b gives the *OPND* representation of the score in Figure 3a. Note that within each pitch name in an element in an *OPND* representation, each flat symbol is represented by an 'F' character and each sharp symbol is represented by an 'S' character (double-sharps are denoted by two 'S' characters, so, for example, F𝄪4 is denoted by "FSS4".)

The test corpus was derived by automatic conversion from Hewlett's (1997) *MuseData* encodings of Bach's *Das Wohltemperirte Klavier*.[3] The *MuseData* encodings contained minor errors which were corrected in the *OPND* representations.

Also, Temperley's algorithm cannot deal with situations in which two or more notes with the same pitch begin at the same time. Thus, wherever two or more notes with the same pitch p began simultaneously at time t, all notes with pitch p and onset time t were removed from the *OPND* file except the one with the longest duration. This resulted in a test corpus containing 41544 notes.[4]

The "piano-roll" or MIDI-like input representations accepted by the algorithms compared in this study were derived automatically from the *OPND* representations of the pieces in the test corpus.

Longuet-Higgins's algorithm Pitch spelling is one of the tasks performed by Longuet-Higgins's (1976, 1987, 1993) *music.p* program. Longuet-Higgins has published the full POP-11 source code of this program (Longuet-Higgins, 1987, pp. 120–126; Longuet-Higgins, 1993, pp. 486–492). Just the pitch spelling portion of *music.p* was translated directly into Lisp in order to make it easier to perform the comparison. This was possible because the pitch spelling portion of *music.p* operates independently of the rhythmic

[2]The voice of each note is also given in the files used in the comparison, but this voice information is not used by any of the algorithms.

[3]Available online at `http://www.musedata.org/encodings/bach/bg/keybd/`.

[4]The complete test corpus is available online at `http://www.titanmusic.com/data.html`.

part.[5]

The input to *music.p* must be in the form of a list of triples, $\langle p, t_{\rm on}, t_{\rm off} \rangle$, each triple giving the "keyboard position" p together with the onset time $t_{\rm on}$ and the offset time $t_{\rm off}$ in centiseconds of each note. The keyboard position p is simply an integer indicating the key that would have to be pressed on a normal piano keyboard in order to perform the note, with C♮3 mapping onto 0, C♯3 and D♭3 mapping onto 1, C♮4 mapping onto 12 and so on (i.e., $p = \text{MIDI NOTE NUMBER} - 48$).

The algorithm then computes a value of "sharpness" q for each note in the input (Longuet-Higgins, 1987, p. 111). The sharpness of a pitch name indicates the position of the pitch name on the line of fifths (Temperley, 2001, p. 117) and is therefore essentially the same as Temperley's (2001, p. 118) concept of "tonal pitch class". In Longuet-Higgins's algorithm, if q_1 and q_2 are the sharpnesses of two notes then the interval between the notes is defined to have "degree" $\delta q = q_2 - q_1$. If $|\delta q| < 6$, the interval is defined to be "diatonic"; if $|\delta q| > 6$, it is defined to be "chromatic"; and if $|\delta q| = 6$, it is defined to be "diabolic" (Longuet-Higgins, 1987, p. 112). Longuet-Higgins's algorithm attempts to spell notes so that the degree between each note and the tonic at the point at which the note occurs is not chromatic (Longuet-Higgins, 1987, p. 113). The algorithm also incorporates various rules for dealing with chromatic passages (Longuet-Higgins, 1987, pp. 113–114).

Pitch spelling algorithms sometimes spell complete pieces in a different key from that in which they are notated in the original score. For example, Longuet-Higgins's algorithm spells the Prelude in G♯ minor from Book 1 of *Das Wohltemperirte Klavier* (BWV 863) in A♭ minor. This should not be considered an error. What matters is whether or not the pitch *interval* names (e.g., 'rising major third', 'falling augmented fourth', etc.) between corresponding pairs of notes are the same in the computed spelling as they are in the published score. The author's implementation of Longuet-Higgins's algorithm therefore actually generates three alternative spellings for each piece:

[5] The Lisp implementation of the pitch spelling portion of *music.p* is available online at `http://www.titanmusic.com/software.html`.

1. the spelling S for the whole input passage computed by the algorithm;

2. the spelling that results when S is transposed by a rising diminished second; and

3. the spelling that results when S is transposed by a falling diminished second.

Each of these three computed spellings is then compared with the pitch names in the original score and the number of errors made by the program is taken to be the number of errors in the best of the three alternative spellings.

When the author's implementation of Longuet-Higgins's algorithm was run on the test corpus described above, it made only 265 errors—that is, it predicted the correct pitch name for 99.36% of the notes.

Cambouropoulos's algorithm Unlike Longuet-Higgins, Cambouropoulos has not published an implementation of his pitch spelling algorithm, nor was he able to supply his own implementation when it was requested. A new implementation of his method was therefore made, based on his published descriptions of it (Cambouropoulos, 1996, 1998, 2000, 2001, 2002).[6]

Cambouropoulos's method involves first converting the input representation into a sequence of *chromas* or *pitch classes* in which the chromas are in the order in which they occur in the music (the chromas of notes that occur simultaneously being ordered arbitrarily). The chroma of a note can be computed from its MIDI note number using the formula

$$\text{CHROMA} = \text{MIDI NOTE NUMBER} \bmod 12.$$

The term *chroma* has been used in this sense since around 1950 (Bachem, 1950; Burns and Ward, 1982, pp. 246, 262–264; Cross *et al.*, 1991, pp. 212, 223–224; Deutsch, 1982, p. 272; Deutsch, 1999, p. 350; Shepard, 1964; Shepard, 1965; Shepard, 1982, p. 352; Dowling, 1991, p. 35; Ward and Burns, 1982, p. 432–433). The term *chroma* is essentially

[6] The Lisp code for the author's implementation of Cambouropoulos's pitch spelling algorithm is available online at `http://www.titanmusic.com/software.html`.

synonymous with the term *pitch class* as used in atonal theory (Babbitt, 1960; Babbitt, 1965; Forte, 1973; Morris, 1987; Rahn, 1980).

Having derived an ordered set of chromas from the input, Cambouropoulos's algorithm then processes the music a window at a time, each window containing a fixed number of notes (set to a value between 9 and 15). Each window is positioned so that the first third of the window overlaps the last third of the previous window. Cambouropoulos allows 'white note' chromas (i.e., 0, 2, 4, 5, 7, 9 and 11) to be spelt in three different ways (e.g., chroma 0 can be spelt as B♯, C♮ or D♭♭) and 'black note' chromas to be spelt in two different ways (e.g., chroma 6 can be spelt as F♯ or G♭). Given these restricted sets of possible pitch names for each chroma, the algorithm computes all possible spellings for each window. A penalty score is then computed for each of these possible window spellings. The penalty score for a given window spelling is found by computing a penalty value for the interval between each pair of notes in the window and summing these penalty values. A given interval in a particular window spelling is penalised more heavily if it is an interval that occurs less frequently in the major and minor scales. An interval is also penalised if either of the pitch names forming the interval is a double-sharp or a double-flat. For each window, the algorithm chooses the spelling that has the lowest penalty score.

When the author's implementation of Cambouropoulos's method was run on the test corpus, it made 2599 mistakes—that is, it predicted the correct pitch name for only 93.74% of the notes. When Cambouropoulos ran his own implementation of his method on the test corpus, he found that it made even more errors than the author's implementation. This may be due to the fact that the new implementation generates three alternative transpositions of the computed spelling and chooses the one that results in the least number of errors.

Temperley's algorithm Temperley's (1997, 2001) pitch spelling algorithm is implemented in his *harmony* program which forms one component of his and Sleator's *Melisma* system.[7] The input to the *harmony* program must be in the form of a "note-list" (Temperley, 2001, pp. 9–12) giving the MIDI note number of each note together with its onset time and duration in milliseconds. The *harmony* program also requires a specification of the metrical structure of the input passage which can be generated by first running the note-list through Temperley and Sleator's *meter* program (another component of the *Melisma* system).

Temperley's (2001, pp. 115–136) pitch spelling algorithm searches for the spelling that best satisfies three "preference rules". The first of these rules stipulates that the algorithm should "prefer to label nearby events so that they are close together on the line of fifths" (Temperley, 2001, p. 125). This rule bears some resemblance to the basic principle underlying Longuet-Higgins's algorithm (see above). The second rule expresses the principle that if two tones are separated by a semitone and the first tone is distant from the key centre, then the interval between them should preferably be spelt as a diatonic semitone rather than a chromatic one (Temperley, 2001, p. 129). The third preference rule underlying Temperley's algorithm steers the algorithm towards spelling the notes so that a "good harmonic representation" results (Temperley, 2001, p. 131), a "good harmonic representation" being one that allows Temperley's *harmony* program to generate a correct harmonic analysis of the passage.

Because the output of the *harmony* program depends on tempo, each *OPND* representation in the test corpus had to be supplemented with a file giving a tempo map for the piece represented. These two files were then automatically converted into a note-list in the form required by Temperley's programs. The output of the *harmony* program was then automatically converted into a format which would allow automatic comparison with the test corpus files. Again, for each piece in the test corpus, three transpositions of the spelling generated by Temperley's program were compared with the original spelling and the one with the least number of errors was selected. When Temperley's harmony

[7]The complete *Melisma* system together with documentation is available online at `http://www.link.cs.cmu.edu/music-analysis/`.

program was run on the test corpus, it made only 122 mistakes—that is, it predicted the correct pitch name for 99.71% of the notes.

Summary of the invention

The invention described here consists of an algorithmic method called *ps13* that reliably computes 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 and MIDI note number of each note in the passage.

The *ps13* algorithm has been shown to be more reliable than previous algorithms, correctly predicting the pitch names of 99.81% of the notes in a test corpus containing 41544 notes and consisting of all the pieces in the first book of J. S. Bach's *Das Wohltemperirte Klavier* (i.e., *ps13* incorrectly predicted the pitch names of only 81 notes in this test corpus).

Three previous algorithms (those of Cambouropoulos (1996, 1998, 2000, 2001, 2002), Longuet-Higgins (1976, 1987, 1993) and Temperley (1997, 2001)) were run on the same corpus of 41544 notes. On this corpus, Cambouropoulos's algorithm made 2599 mistakes, Longuet-Higgins's algorithm made 265 mistakes and Temperley's algorithm made 122 mistakes. As *ps13* made only 81 mistakes on the same corpus, this provides evidence in support of the claim that *ps13* is more reliable than previous algorithms that attempt to perform the same task.

The *ps13* algorithm is best understood to be in two parts, Part I and Part II. Part I consists of the following steps:

1. computing for each pitch class $0 \leq c \leq 11$ and each note n in the input, the pitch letter name $S(c, n) \in \{A, B, C, D, E, F, G\}$ that n would have if c were the tonic at the point in the piece where n occurs (assuming that the notes are spelt as they are in the harmonic chromatic scale on c);

2. computing for each note n in the input and each pitch class $0 \leq c \leq 11$, a value

$CNT(c,n)$ giving the number of times that c occurs within a context surrounding n that includes n, some specified number K_{pre} of notes immediately preceding n and some specified number K_{post} of notes immediately following n;

3. computing for each note n and each letter name l, the set of chromas $C(n,l) = \{c \mid S(c,n) = l\}$ (that is, the set of tonic chromas that would lead to n having the letter name l);

4. computing $N(l,n) = \sum_{c \in C(n,l)} CNT(c,n)$ for each note n and each pitch letter name l;

5. computing for each note n, the letter name l for which $N(l,n)$ is a maximum.

Part II of the algorithm corrects those instances in the output of Part I 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. Part II of *ps13*

1. lowers the letter name of every lower neighbour note for which the letter name predicted by Part I is the same as that of the preceding note;

2. raises the letter name of every upper neighbour note for which the letter name predicted by Part I is the same as that of the preceding note;

3. lowers the letter name of every descending passing note for which the letter name predicted by Part I is the same as that of the preceding note;

4. raises the letter name of every descending passing note for which the letter name predicted by Part I is the same as that of the following note;

5. lowers the letter name of every ascending passing note for which the letter name predicted by Part I is the same as that of the following note;

6. raises the letter name of every ascending passing note for which the letter name predicted by Part I is the same as that of the preceding note.

List of figures

Detailed description of preferred implementations

The invention consists of an algorithm, called *ps13*, that takes as input a representation of a musical passage (or work or set of works) in the form of a set I of triples $\langle t, p_c, d\rangle$, each of these triples giving the onset time t, the *chromatic pitch* p_c and the duration d of a single note or sequence of tied notes. The chromatic pitch of a note is an integer indicating the interval in semitones from A♮0 to the pitch of the note (i.e., $p_{\mathrm{c}} = \text{MIDI NOTE NUMBER} - 21$). The set I may be trivially derived from a MIDI file representation of the musical passage.

If $e_i = \langle t_i, p_{\mathrm{c},i}, d_i \rangle$ and $e_j = \langle t_j, p_{\mathrm{c},j}, d_j \rangle$ and $e_i, e_j \in I$ then e_i is defined to be *less than* e_j, denoted by $e_i < e_j$, if and only if $t_i < t_j$ or $(t_i = t_j \wedge p_{\mathrm{c},i} < p_{\mathrm{c},j})$ or $(t_i = t_j \wedge p_{\mathrm{c},i} = p_{\mathrm{c},j} \wedge d_i < d_j)$. Also, $e_i \leq e_j$ if and only if $e_i < e_j$ or $e_i = e_j$.

The first step in *ps13* is to sort the set I to give an ordered set

$$J = \left\langle \langle t_1, p_{\mathrm{c},1}, d_1 \rangle, \langle t_2, p_{\mathrm{c},2}, d_2 \rangle, \ldots \left\langle t_{|I|}, p_{\mathrm{c},|I|}, d_{|I|} \right\rangle \right\rangle \tag{1}$$

containing all and only the elements of I, sorted into increasing order so that $j > i \Rightarrow \langle t_i, p_{\mathrm{c},i}, d_i \rangle \leq \langle t_j, p_{\mathrm{c},j}, d_j \rangle$ for all $\langle t_i, p_{\mathrm{c},i}, d_i \rangle, \langle t_j, p_{\mathrm{c},j}, d_j \rangle \in J$.

The second step in *ps13* is to compute the ordered set

$$C = \left\langle c_1, c_2, \ldots c_k, \ldots c_{|J|} \right\rangle \tag{2}$$

where $c_k = p_{\mathrm{c},k} \bmod 12$, $p_{\mathrm{c},k}$ being the chromatic pitch of the kth element of J. c_k is the *chroma* of $p_{\mathrm{c},k}$.

If A is an ordered set of elements,

$$A = \left\langle a_1, a_2, \ldots a_k, \ldots a_{|A|} \right\rangle$$

then let $A[j]$ denote the $(j+1)$th element of A (e.g., $A[0] = a_1, A[1] = a_2$). Also, let $A[j,k]$ denote the ordered set that contains all the elements of A from $A[j]$ to $A[k-1]$, inclusive (e.g., $A[1,4] = \langle a_2, a_3, a_4 \rangle$).

In addition to the set I, *ps13* takes as input two numerical parameters, called the *precontext*, denoted by K_{pre}, and the *postcontext*, denoted by K_{post}. The precontext must be an integer greater than or equal to 0 and the postcontext must be an integer greater than 0. The third step in *ps13* is to compute the ordered set

$$\Phi = \langle \phi(C[0]), \phi(C[1]), \ldots \phi(C[k]), \ldots \phi(C[|C| - 1]) \rangle \tag{3}$$

where

$$\phi(C[k]) = \langle \mathit{CNT}(0, k), \mathit{CNT}(1, k), \ldots \mathit{CNT}(11, k) \rangle \tag{4}$$

and $\mathit{CNT}(c, k)$ returns the number of times that the chroma c occurs in the ordered set $C[\max(\{0, k - K_{\text{pre}}\}), \min(\{|C|, k + K_{\text{post}}\})]$ (the functions $\max(B)$ and $\min(B)$ return the greatest and least values, respectively, in the set B).

Every pitch name has three parts: a *letter name* which must be a member of the set {A,B,C,D,E,F,G}; an *inflection* which must be a member of the infinite set $\{\ldots 𝄪𝄪, \sharp𝄪, 𝄪, \sharp, \natural, \flat, \flat\flat, \flat\flat\flat, \flat\flat\flat\flat, \ldots\}$; and an *octave number* which must be an integer. By convention, if the inflection of a pitch name is equal to ♮ it may be omitted, for example, C♮4 may be written C4. The octave number of 'middle C' is 4 and the octave number of any other C is one greater than the next C below it *on the staff* and one less than the next C above it *on the staff.* The octave number of any pitch name whose letter name is not C is the same as that of the nearest C below it *on the staff.* Thus B𝄪3 sounds one semitone higher than C4 and C♭4 has the same sounding pitch as B♮3.

If N is a pitch name with letter name $l(N)$ and octave number $o(N)$, then the *morph* of N, denoted by $m(N)$, is given by the following table

$l(N)$	A	B	C	D	E	F	G
$m(N)$	0	1	2	3	4	5	6

The *morphetic octave* of N, denoted by $o_{\text{m}}(N)$, is given by

$$o_{\text{m}}(N) = \begin{cases} o(N), & \text{if } m = 0 \text{ or } m = 1, \\ o(N) - 1, & \text{otherwise.} \end{cases} \tag{5}$$

The *morphetic pitch* of N, denoted by $p_{\mathrm{m}}(N)$, is given by

$$p_{\mathrm{m}}(N) = m(N) + 7o_{\mathrm{m}}(N). \tag{6}$$

The fourth step in *ps13* is to compute the *spelling table*, S. This is done using the algorithm expressed in pseudo-code in Figure 4. In this pseudo-code, block structure is indicated by indentation and the symbol "$\leftarrow$" is used to denote assignment of a value to a variable (i.e., the expression "$x \leftarrow y$" means "the value of variable x becomes equal to the value of y"). The symbol "$\oplus$" is used to denote concatenation of ordered sets. Thus, if A and B are two ordered sets such that $A = \left\langle a_1, a_2, \ldots a_{|A|} \right\rangle$ and $B = \left\langle b_1, b_2, \ldots b_{|B|} \right\rangle$, then

$$A \oplus B = \left\langle a_1, a_2, \ldots a_{|A|}, b_1, b_2, \ldots b_{|B|} \right\rangle.$$

Also, $A \oplus \langle\,\rangle = \langle\,\rangle \oplus A = A$.

Having computed the spelling table S, the algorithm goes on to compute the *relative morph list*, R. This is computed using the algorithm in Figure 5. If A is an ordered set of ordered sets, then $A[i][j]$ denotes the $(j+1)$th element of the $(i+1)$th element of A. The expression $S[k][i]$ in line 6 of Figure 5 therefore denotes the $(i+1)$th element of the $(k+1)$th element of the spelling table S. If L is an ordered set and i is the value of at least one element of L, then the function $\mathit{POS}(i, L)$ called in line 10 of Figure 5 returns the least value of k for which $L[k] = i$.

The next step in *ps13* is to compute the *initial morph* m_{init} which is given by

$$m_{\mathrm{init}} = Q_{\mathrm{init}}[C[0]] \tag{7}$$

where $Q_{\mathrm{init}} = \langle 0, 1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 6 \rangle$.

Next, *ps13* computes the *morph list*, M, which satisfies the following condition

$$(|M| = |C|) \wedge (M[i] = (R[i] + m_{\mathrm{init}}) \bmod 7 \text{ for all } 0 \leq i < |C|). \tag{8}$$

Next, the ordered set O is computed which satisfies the following condition

$$(|O| = |C|) \wedge (O[i] = \langle J[i][0], C[i], M[i]\rangle \text{ for all } 0 \leq i < |C|). \tag{9}$$

Next, an ordered set H called the *chord list* is computed using the algorithm in Figure 6.

In the next three steps, the algorithm ensures that neighbour note and passing note figures are spelt correctly. Figure 7 illustrates the six types of error corrected by the algorithm. Figure 7a shows a lower neighbour note figure that is incorrectly spelt because the neighbour note (the middle note) has the same morphetic pitch as the two flanking notes. In this case, the morphetic pitch of the neighbour note is one greater than it should be. Figure 7b shows a similar error in the case of an upper neighbour note figure. In this case, the morphetic pitch of the neighbour note is one less than it should be. The passing note figures in Figure 7c and d are incorrect because the morphetic pitch of the passing note (the middle note) is one greater than it should be. Finally, the passing note figures in Figure 7e and f are incorrect because the morphetic pitch of the passing note is one less than it should be.

ps13 first corrects errors like the ones in Figure 7a and b using the algorithm in Figure 8. If A is an ordered set of ordered sets then the expression $A[i][j,k]$ denotes the ordered set $\langle A[i][j], A[i][j+1], \ldots A[i][k-1]\rangle$. Thus, the expression $H[i+2][k][1,3]$ in line 4 of Figure 8 denotes the ordered set $\langle H[i+2][k][1], H[i+2][k][2]\rangle$. In Figure 7a and b, the neighbour note is one semitone away from the flanking notes. The algorithm in Figure 8 also corrects instances where the neighbour note is 2 semitones above or below the flanking notes but has the same morphetic pitch as the flanking notes.

Next, *ps13* corrects errors like the ones in Figure 7c and e using the algorithm in Figure 9. Then it corrects errors like the ones in Figure 7d and f using the algorithm in Figure 10. In Figure 7c, d, e and f, the interval between the flanking notes is a minor third. The algorithms in Figures 9 and 10 also correct instances where the interval between the flanking notes is a major third.

Having corrected neighbour note and passing note errors, *ps13* then computes a new morph list M' using the algorithm in Figure 11.

Having computed M', it is now possible to compute a morphetic pitch for each note. This can be done using the algorithm in Figure 12 which computes the ordered set of morphetic pitches, P.

Finally, from J and P, *ps13* computes an *OPND* representation Z which satisfies the following condition

$$(|Z| = |J|) \wedge (Z[i] = \langle J[i][0], \mathit{PPN}(\langle J[i][1], P[i]\rangle), J[i][2]\rangle \text{ for all } 0 \leq i < |Z|). \qquad (10)$$

The function $PPN(\langle p_{\mathrm{c}}, p_{\mathrm{m}}\rangle)$ returns the unique pitch name whose chromatic pitch is p_{c} and whose morphetic pitch is p_{m}. This function can be computed using the algorithm in Figure 13. In this algorithm, anything in double inverted commas ("?") is a string—that is, an ordered set of characters. If A and B are strings such that $A =$ "*abcdef*" and $B =$ "*ghijkl*", then the concatenation of A onto B, denoted by $A \oplus B$, is "*abcdefghijkl*". In the pitch names generated by *PPN*, the sharp signs are represented by 's' characters and the flat signs are represented by 'f' characters. A double sharp is represented by the string "ss". For example, the pitch name C𝄪4 is represented in *PPN* by the string "Css4". The empty string is denoted by "" and the function $\mathit{STR}(n)$ called in line 23 returns a string representation of the number n. For example, $\mathit{STR}(-5) =$ "-5", $\mathit{STR}(105.6) =$ "105.6".

Results of running *ps13* on the test corpus As explained above, *ps13* takes as input a set I of triples, $\langle t, p_{\mathrm{c}}, d\rangle$, each one giving the onset time, chromatic pitch and duration of each note. In addition, *ps13* requires two numerical parameters, the precontext, K_{pre}, and the postcontext, K_{post}.

In order to determine the values of K_{pre} and K_{post} that give the best results, *ps13* was run on the test corpus 2500 times, each time using a different pair of values

$\langle K_{\text{pre}}, K_{\text{post}} \rangle$ chosen from the set $\{\langle K_{\text{pre}}, K_{\text{post}} \rangle \mid 1 \leq K_{\text{pre}}, K_{\text{post}} \leq 50\}$. For each pair of values $\langle K_{\text{pre}}, K_{\text{post}} \rangle$, the number of errors made by *ps13* on the test corpus was recorded.

ps13 made fewer than 122 mistakes (i.e., performed better than Temperley's algorithm) on the test corpus for 2004 of the 2500 $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ pairs tested (i.e., 80.160% of the $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ pairs).

ps13 performed best on the test corpus when K_{pre} was set to 33 and K_{post} was set to either 23 or 25. With these parameter values, *ps13* made only 81 errors on the test corpus—that is, it correctly predicted the pitch names of 99.805% of the notes in the test corpus.

The mean number of errors made by *ps13* over all 2500 $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ pairs was 109.082 (i.e., 99.737% of the notes were correctly spelt on average over all 2500 $\langle K_{\text{pre}}, K_{\text{post}} \rangle$ pairs). This average value is better than the result obtained by Temperley's algorithm for this test corpus.

The worst result was obtained when both K_{pre} and K_{post} were set to 1. In this case, *ps13* made 1117 errors (97.311% correct). However, provided K_{pre} is greater than about 14 and K_{post} is greater than about 21, *ps13* predicts the correct pitch name for over 99.75% of the notes in the test corpus.

A Lisp implementation of *ps13* The Lisp implementation of *ps13* given below assumes that the input representation I is represented as a list of sublists, each sublist taking the form $(t\ p_{\text{c}}\ d)$ where t, p_{c} and d are the onset time, chromatic pitch and duration, respectively, of a single note. For example, the passage in Figure 3a would be represented by the list

```
((2 41 2) (4 43 2) (6 44 2) (8 46 2) (10 43 2) (12 44 1) (13 41 1) (14 40 1)
 (15 41 1) (16 49 4) (20 46 4) (24 48 5) (26 36 2) (28 38 2) (29 46 1)
 (30 39 2) (30 44 1) (31 43 1) (32 41 2) (32 46 1) (33 44 1) (34 38 2)
 (34 43 1) (35 41 1) (36 39 1) (36 43 2) (37 36 1) (38 35 1) (38 51 3)
 (39 36 1) (40 44 4) (41 50 1) (42 48 1) (43 50 1) (44 41 4) (44 50 1)
 (45 48 1) (46 47 1) (47 48 1))
```

Similarly, the output of the Lisp implementation of *ps13* given below is represented as a list of sublists. For example, the output of this implementation of *ps13* for the passage given in Figure 3a is

```
((2 "Dn4" 2) (4 "En4" 2) (6 "Fn4" 2) (8 "Gn4" 2) (10 "En4" 2) (12 "Fn4" 1)
 (13 "Dn4" 1) (14 "Cs4" 1) (15 "Dn4" 1) (16 "Bf4" 4) (20 "Gn4" 4) (24 "An4" 5)
 (26 "An3" 2) (28 "Bn3" 2) (29 "Gn4" 1) (30 "Cn4" 2) (30 "Fn4" 1) (31 "En4" 1)
 (32 "Dn4" 2) (32 "Gn4" 1) (33 "Fn4" 1) (34 "Bn3" 2) (34 "En4" 1) (35 "Dn4" 1)
 (36 "Cn4" 1) (36 "En4" 2) (37 "An3" 1) (38 "Gs3" 1) (38 "Cn5" 3) (39 "An3" 1)
 (40 "Fn4" 4) (41 "Bn4" 1) (42 "An4" 1) (43 "Bn4" 1) (44 "Dn4" 4) (44 "Bn4" 1)
 (45 "An4" 1) (46 "Gs4" 1) (47 "An4" 1))
```

Here, then, is the Lisp code for an implementation of *ps13*:

```
(defun ps13 (&optional (input-filename (choose-file-dialog))
                       (pre-context 33)
                       (post-context 23))
  (let* ((sorted-input-representation
          (remove-duplicates
           (sort (with-open-file (input-file
                                  input-filename)
                   (read input-file))
                 #'vector-less-than)
           :test #'equalp))
         (onset-list (mapcar #'first sorted-input-representation))
         (chromatic-pitch-list
          (mapcar #'second sorted-input-representation))
         (chroma-list
          (mapcar #'chromatic-pitch-chroma chromatic-pitch-list))
         (n (list-length chroma-list))
         (chroma-vector-list
          (do* ((cvl nil)
                (i 0 (1+ i)))
               ((= i n)
                cvl)
            (setf cvl
                  (append cvl
                          (list
                           (do* ((context
                                  (subseq chroma-list
                                          (max 0 (- i pre-context))
                                          (min n (+ i post-context))))
                                 (cv (list 0 0 0 0 0 0 0 0 0 0 0 0))
                                 (c 0 (+ 1 c)))
                                ((= c 12)
                                 cv)
                             (setf (elt cv c)
```

```
                        (count c context))))))))
(chromamorph-table (list 0 1 1 2 2 3 3 4 5 5 6 6))
(spelling-table
 (do* ((first-morph nil nil)
       (spelling nil nil)
       (spelling2 nil nil)
       (st nil)
       (c 0 (1+ c)))
      ((= c 12)
       st)
   (setf spelling
         (mapcar #'(lambda (chroma-in-chroma-list)
                     (elt chromamorph-table
                          (mod (- chroma-in-chroma-list c) 12)))
                 chroma-list))
   (setf first-morph (first spelling))
   (setf spelling2
         (mapcar #'(lambda (morph-in-spelling)
                     (mod (- morph-in-spelling first-morph) 7))
                 spelling))
   (setf st (append st (list spelling2)))))
(relative-morph-list
 (do ((morph-vector (list 0 0 0 0 0 0 0)
                    (list 0 0 0 0 0 0 0))
      (rml nil)
      (i 0 (1+ i))
      (morphs-for-this-chroma nil
                              nil)
      )
     ((= i n)
      rml)
   (setf morphs-for-this-chroma
         (mapcar #'(lambda (spelling)
                     (elt spelling i))
                 spelling-table))
   (setf rml
         (do ((prev-score nil nil)
              (j 0 (1+ j)))
             ((= j 12)
              ;(pprint morph-vector)
              (append rml
                      (list (position
                             (apply #'max morph-vector)
                             morph-vector))))
           (setf prev-score
                 (elt morph-vector
                      (elt morphs-for-this-chroma j)))
           (setf (elt morph-vector
```

```
                    (elt morphs-for-this-chroma j))
               (+ prev-score
                  (elt (elt chroma-vector-list i) j)))))))
(initial-morph (elt '(0 1 1 2 2 3 4 4 5 5 6 6)
                  (mod (first chromatic-pitch-list) 12)))
(morph-list (mapcar #'(lambda (relative-morph)
                      (mod (+ relative-morph initial-morph) 7))
                  relative-morph-list))
(ocm (mapcar #'list onset-list chroma-list morph-list))
(ocm-chord-list (do* ((cl (list (list (first ocm))))
                    (i 1 (1+ i)))
                   ((= i n)
                    cl)
                (if (= (first (elt ocm i))
                       (first (elt ocm (1- i))))
                  (setf (first (last cl))
                        (append (first (last cl))
                                (list (elt ocm i))))
                  (setf cl
                        (append cl
                                (list (list (elt ocm i))))))))
(number-of-chords (list-length ocm-chord-list))
;neighbour notes
(ocm-chord-list
 (do* ((i 0 (1+ i)))
      ((= i (- number-of-chords 2))
       ocm-chord-list)
   (dolist (note1 (elt ocm-chord-list i))
     (if (member (cdr note1)
                 (mapcar #'cdr (elt ocm-chord-list (+ i 2)))
                 :test #'equalp)
       (dolist (note2 (elt ocm-chord-list (1+ i)))
         (if (= (third note2)
                (third note1))
           (progn
             (if (member (mod (- (second note2) (second note1))
                              12)
                         '(1 2))
               (setf (third note2)
                     (mod (+ 1 (third note2)) 7)))
             (if (member (mod (- (second note1) (second note2))
                              12)
                         '(1 2))
               (setf (third note2)
                     (mod (- (third note2) 1) 7))))))))))
;downward passing notes
(ocm-chord-list
 (do* ((i 0 (1+ i)))
```

```
      ((= i (- number-of-chords 2))
       ocm-chord-list)
    (dolist (note1 (elt ocm-chord-list i))
      (dolist (note3 (elt ocm-chord-list (+ i 2)))
        (if (= (third note3) (mod (- (third note1) 2) 7))
          (dolist (note2 (elt ocm-chord-list (1+ i)))
            (if (and (or (= (third note2)
                            (third note1))
                         (= (third note2)
                            (third note3)))
                     (< 0
                        (mod (- (second note1) (second note2))
                             12)
                        (mod (- (second note1) (second note3))
                             12)))
              (unless (remove-if
                        #'null
                        (mapcar
                         #'(lambda (note)
                             (/= (second note)
                                 (second note2)))
                         (remove-if
                          #'null
                          (mapcar
                           #'(lambda (note)
                               (if (= (third note)
                                      (mod (- (third note1) 1)
                                           7))
                                 note))
                           (elt ocm-chord-list (1+ i))))))
                (setf (third note2)
                      (mod (- (third note1) 1) 7))))))))))
;upward passing notes
(ocm-chord-list
 (do* ((i 0 (1+ i)))
      ((= i (- number-of-chords 2))
       ocm-chord-list)
    (dolist (note1 (elt ocm-chord-list i))
      (dolist (note3 (elt ocm-chord-list (+ i 2)))
        (if (= (third note3) (mod (+ (third note1) 2) 7))
          (dolist (note2 (elt ocm-chord-list (1+ i)))
            (if (and (or (= (third note2)
                            (third note1))
                         (= (third note2)
                            (third note3)))
                     (< 0
                        (mod (- (second note3) (second note2))
                             12)
```

```
                      (mod (- (second note3) (second note1))
                           12)))
            (unless (remove-if
                      #'null
                      (mapcar
                       #'(lambda (note)
                           (/= (second note)
                               (second note2)))
                       (remove-if
                        #'null
                        (mapcar #'(lambda (note)
                                    (if (= (third note)
                                           (mod (+ (third note1)
                                                   1)
                                                 7))
                                      note))
                                (elt ocm-chord-list (1+ i))))))
              (setf (third note2)
                    (mod (+ (third note1) 1) 7))))))))))
(morph-list (mapcar #'third (apply #'append ocm-chord-list)))
(morphetic-pitch-list
 (mapcar #'(lambda (chromatic-pitch morph)
             (let* ((morphetic-octave1 (floor chromatic-pitch 12))
                    (morphetic-octave2 (+ 1 morphetic-octave1))
                    (morphetic-octave3 (- morphetic-octave1 1))
                    (mp1 (+ morphetic-octave1 (/ morph 7)))
                    (mp2 (+ morphetic-octave2 (/ morph 7)))
                    (mp3 (+ morphetic-octave3 (/ morph 7)))
                    (chroma (mod chromatic-pitch 12))
                    (cp (+ morphetic-octave1 (/ chroma 12)))
                    (difference-list (list (abs (- cp mp1))
                                           (abs (- cp mp2))
                                           (abs (- cp mp3))))
                    (morphetic-octave-list (list morphetic-octave1
                                                 morphetic-octave2
                                                 morphetic-octave3))
                    (best-morphetic-octave
                     (elt morphetic-octave-list
                          (position (apply #'min difference-list)
                                    difference-list))))
               (+ (* 7 best-morphetic-octave) morph)))
         chromatic-pitch-list
         morph-list))
(opd (mapcar #'(lambda (tpcd-triple morphetic-pitch)
                 (list (first tpcd-triple)
                       (list (second tpcd-triple)
                             morphetic-pitch)
                       (third tpcd-triple)))
```

```
                      sorted-input-representation
                      morphetic-pitch-list))
         (opnd (mapcar #'(lambda (opd-datapoint)
                           (list (first opd-datapoint)
                                 (p-pn (second opd-datapoint))
                                 (third opd-datapoint)))
                       opd)))
    opnd))

(defun chromatic-pitch-chroma (chromatic-pitch)
  (mod chromatic-pitch 12))

(defun vector-less-than (v1 v2)
  (cond ((null v2) nil)
        ((null v1) t)
        ((< (first v1) (first v2)) t)
        ((> (first v1) (first v2)) nil)
        (t (vector-less-than (cdr v1) (cdr v2)))))

(defun p-pn (p)
  (let* ((m (p-m p))
         (l (elt '("A" "B" "C" "D" "E" "F" "G") m))
         (gc (p-gc p))
         (cdash (elt '(0 2 3 5 7 8 10) m))
         (e (- gc cdash))
         (i "")
         (i (cond ((< e 0)
                   (dotimes (j (- e) i)
                     (setf i (concatenate 'string i "f"))))
                  ((> e 0)
                   (dotimes (j e i)
                     (setf i (concatenate 'string i "s"))))
                  ((= e 0) "n")))
         (om (p-om p))
         (oasa (if (or (= m 0) (= m 1))
                   om
                   (+ 1 om)))
         (o (format nil "~D" oasa)))
    (concatenate 'string l i o)))

(defun p-om (p)
  (div (p-pm p) 7))

(defun p-pm (p)
  (second p))

(defun div (x y)
  (int (/ x y)))
```

```
(defun int (x)
  (values (floor x)))

(defun p-gc (p)
  (- (p-pc p)
     (* 12 (p-om p))))

(defun p-pc (p)
  (first p))

(defun p-m (p)
  (bmod (p-pm p) 7))

(defun bmod (x y)
  (- x
     (* y
        (int (/ x y)))))
```

References

Babbitt, M. (1960). Twelve-tone invariants as compositional determinants. *The Musical Quarterly*, **46**(2), 246–259.

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

Bachem, A. (1950). Tone height and tone chroma as two different pitch qualities. *Acta Psychologica*, **7**, 80–88.

Burns, E. M. and Ward, W. D. (1982). Intervals, scales and tuning. In D. Deutsch, editor, *The Psychology of Music*, chapter 8, pages 241–269. Academic Press, Orlando, FL.

Cambouropoulos, E. (1996). A general pitch interval representation: Theory and applications. *Journal of New Music Research*, **25**, 231–251.

Cambouropoulos, E. (1998). *Towards a General Computational Theory of Musical Structure*. Ph.D. thesis, University of Edinburgh.

Cambouropoulos, E. (2000). From MIDI to traditional musical notation. In *Proceedings of the AAAI 2000 Workshop on Artificial Intelligence and Music, 17th National Conference on Artificial Intelligence (AAAI'2000), 30 July–3 August*, Austin, TX. Available online at `ftp://ftp.ai.univie.ac.at/papers/oefai-tr-2000-15.pdf`.

Cambouropoulos, E. (2001). Automatic pitch spelling: From numbers to sharps and flats. In *VIII Brazilian Symposium on Computer Music (SBC&M 2001)*, Fortaleza, Brazil. Available online at `ftp://ftp.ai.univie.ac.at/papers/oefai-tr-2001-12.pdf`.

Cambouropoulos, E. (2002). Pitch spelling: A computational model. *Music Perception*. To appear.

Cross, I., West, R., and Howell, P. (1991). Cognitive correlates of tonality. In P. Howell,

R. West, and I. Cross, editors, *Representing Musical Structure*, pages 201–243. Academic Press, London.

Davy, M. and Godsill, S. J. (2003). Bayesian harmonic models for musical signal analysis (with discussion). In J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, editors, *Bayesian Statistics*, volume VII. Oxford University Press. Draft available online at `http://www-sigproc.eng.cam.ac.uk/~sjg/papers/02/harmonicfinal2.ps`.

Deutsch, D. (1982). The processing of pitch combinations. In D. Deutsch, editor, *The Psychology of Music*, chapter 9, pages 271–316. Academic Press, Orlando, FL.

Deutsch, D. (1999). The processing of pitch combinations. In D. Deutsch, editor, *The Psychology of Music*, pages 349–411. Academic Press, San Diego, CA.

Dowling, W. J. (1991). Pitch structure. In P. Howell, R. West, and I. Cross, editors, *Representing Musical Structure*, pages 33–57. Academic Press, London.

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

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

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

Longuet-Higgins, H. C. (1987). The perception of melodies. In H. C. Longuet-Higgins, editor, *Mental Processes: Studies in Cognitive Science*, pages 105–129. British Psychological Society/MIT Press, London, England and Cambridge, Mass.

Longuet-Higgins, H. C. (1993). The perception of melodies. In S. M. Schwanauer and D. A. Levitt, editors, *Machine Models of Music*, pages 471–495. M.I.T. Press, Cambridge, Mass.

Meredith, D., Lemström, K., and Wiggins, G. A. (2002). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, **31**(4), 321–345. Draft available online at `http://www.titanmusic.com/papers/public/siajnmr_submit_2.pdf`.

Morris, R. D. (1987). *Composition with Pitch-Classes: A Theory of Compositional Design*. Yale University Press, New Haven and London.

Piston, W. (1978). *Harmony*. Victor Gollancz Ltd., London. Revised and expanded by Mark DeVoto.

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

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

Shepard, R. N. (1964). Circularity in judgments of relative pitch. *Journal of the Acoustical Society of America*, **36**, 2346–2353.

Shepard, R. N. (1965). Approximation to uniform gradients of generalization by monotone transformations of scale. In D. Mostofsky, editor, *Stimulus Generalization*, pages 94–110. Stanford University Press, Stanford, CA.

Shepard, R. N. (1982). Structural representations of musical pitch. In D. Deutsch, editor, *The Psychology of Music*, pages 343–390. Academic Press, London.

Temperley, D. (1997). An algorithm for harmonic analysis. *Music Perception*, **15**(1), 31–68.

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

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.

Walmsley, P. J. (2000). *Signal Separation of Musical Instruments*. Ph.D. thesis, Signal Processing Group, Department of Engineering, University of Cambridge.

Ward, W. D. and Burns, E. M. (1982). Absolute pitch. In D. Deutsch, editor, *The Psychology of Music*, chapter 14, pages 431–451. Academic Press, Orlando, FL.

CLAIMS

1. A method for computing the pitch names (i.e., C♯4, B♭3, etc.) of notes in a representation of music in which at least the onset time and MIDI note number (or chromatic pitch) of each note is given, comprising the steps of

   (a) computing for each pitch class $0 \leq c \leq 11$ and each note n in the input, the pitch letter name $S(c, n) \in \{A, B, C, D, E, F, G\}$, that n would have if c were the tonic at the point in the piece where n occurs (assuming that the notes are spelt as they are in the harmonic chromatic scale on c);

   (b) computing for each note n in the input and each pitch class $0 \leq c \leq 11$ a value $CNT(c, n)$ giving the number of times that c occurs within a context surrounding n that includes n, some specified number K_{pre} of notes immediately preceding n and some specified number K_{post} of notes immediately following n;

   (c) computing for each note n and each letter name l, the set of chromas $C(n, l) = \{c \mid S(c, n) = l\}$ (that is, the set of tonic chromas that would lead to n having the letter name l);

   (d) computing $N(l, n) = \sum_{c \in C(n,l)} CNT(c, n)$ for each note n and each pitch letter name l;

   (e) computing for each note n, the letter name l for which $N(l, n)$ is a maximum.

2. The method of Claim 1, adapted to correct those errors in the output of the method of Claim 1 in which 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, comprising the further steps of

   (a) lowering the letter name of every lower neighbour note for which the letter name predicted by the method of Claim 1 is the same as that of the preceding note;

(b) raising the letter name of every upper neighbour note for which the letter name predicted by the method of Claim 1 is the same as that of the preceding note;

(c) lowering the letter name of every descending passing note for which the letter name predicted by the method of Claim 1 is the same as that of the preceding note;

(d) raising the letter name of every descending passing note for which the letter name predicted by the method of Claim 1 is the same as that of the following note;

(e) lowering the letter name of every ascending passing note for which the letter name predicted by the method of Claim 1 is the same as that of the following note;

(f) raising the letter name of every ascending passing note for which the letter name predicted by the method of Claim 1 is the same as that of the preceding note.

3. Computer software or hardware adapted for performing the method of any preceding Claim 1–2.

ABSTRACT

METHOD OF COMPUTING THE PITCH NAMES OF NOTES IN MIDI-LIKE MUSIC REPRESENTATIONS

The invention described here consists of an algorithmic method called *ps13* that reliably computes 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 and MIDI note number of each note in the passage.

The *ps13* algorithm has been shown to be more reliable than previous algorithms, correctly predicting the pitch names of 99.81% of the notes in a test corpus containing 41544 notes and consisting of all the pieces in the first book of J. S. Bach's *Das Wohltemperirte Klavier* (i.e., *ps13* incorrectly predicted the pitch names of only 81 notes in this test corpus).

Three previous algorithms (those of Cambouropoulos (1996, 1998, 2000, 2001, 2002), Longuet-Higgins (1976, 1987, 1993) and Temperley (1997, 2001)) were run on the same corpus of 41544 notes. On this corpus, Cambouropoulos's algorithm made 2599 mistakes, Longuet-Higgins's algorithm made 265 mistakes and Temperley's algorithm made 122 mistakes. As *ps13* made only 81 mistakes on the same corpus, this provides evidence in support of the claim that *ps13* is more reliable than previous algorithms that attempt to perform the same task.