SCORE PROCESSING FOR MIR

Donncha S. Ó Maidín
Centre for Computational Musicology and Computer Music
Department of Computer Science and Information Systems
University of Limerick
353(0)61202705
donncha.omaidin@ul.ie

Margaret Cahill
Centre for Computational Musicology and Computer Music
Department of Computer Science and Information Systems
University of Limerick
353(0)61202759
margaret.cahill@ul.ie

ABSTRACT
The focus of this paper is on the design and use of a music score representation. The structure of the representation is discussed and illustrated with sample algorithms, including some from music information retrieval. The score representation was designed for the development of general algorithms and applications. The common container-iterator paradigm is used, in which the score is modelled as a container of objects, such as clefs, key signatures, time signatures, notes, rests and barlines. Access to objects within the score is achieved through iterators. These iterators provide the developer with a mechanism for accessing the information content of the score. The iterators are designed to achieve a high level of data hiding, so that the user is shielded from the substantial underlying complexity of score representation, while at the same time, having access to the score's full information content.

1. INTRODUCTION
The focus of this paper is on representing music scores. The music score is the primary document for practically all music of the past. It holds a primary place in literacy, in education, in composition and performance and in music theory.

In the computer era, two main sources for digital versions of scores arise. The first of these is from digitising initiatives, such as those at Center for Computer Assisted Research in the Humanities at Stanford University. The second comes as a by-product of music publishing. These activities have resulted in the production quantities of machine-readable scores. Unfortunately, not all of these efforts are readily usable in IR research. Lack of agreed open standards and lack of openness on the part of notation software developers form some of the main barriers to more general use.

In practice MIDI has become the representation used in much of music information retrieval research. The MIDI standard was invented to capture the gestures of a keyboard player. Its ability to provide the pitch and duration content of music has resulted in its acceptability for music research. However basic MIDI representation is radically different from score representation. Rests, slurs, barlines, staccatos, trills, ornaments, triplets and chromaticisms, are examples of concepts that are not explicit in MIDI.

An alternate prospect to using MIDI is that of having an availability of music scores, encoded to professional editorial standards, together with appropriate tools. One consequence of this approach is that the music IR researcher may work with the full information content of a score, rather than a simplified view of pitch and duration. Additionally this approach will serve to facilitate cooperation between the music IR researcher and music theorists, who deal primarily with the score.

2. CONTAINER-ITERATOR
Some endeavours in Computer Science have been concerned with discovering useful ways for organising collections of data. One of the most basic ways of organising data is to conceive of a complex object as a collection of objects that are included in a **container**. Objects within the container are accessed by means of iterators. Iterators are often characterised as 'safe' pointers – that is that they can point to objects within a container, and can be safely manipulated, or moved about so as to make all of the internal objects accessible. The success of this approach has led to the development of many libraries, the most widely used one being the C++ Standard Template Library that was developed at Hewlett-Packard Labs by Alexander Stepanov and Meng Lee[1][2] and was adapted as part of ANS and ISO standards in the 90's. The containers in the STL allow the user to structure the data as vectors, lists, deques, sets, maps, stack and queues. Most of these data structures are one-dimensional. Their associated iterators are built so that all objects in the more general containers may be accessed in a left-to-right fashion, and possibly in a reverse order. Additionally some iterator/container combinations allow random access to the contained objects.

It is appropriate to consider how the music score may be modelled as a container. Items in a score may be represented as objects within the container. Objects are used to represent notes, barlines, key and time signature. Iterators allow the software developer access to the information content of a score.

A strict adherence to the S.T.L. model has proved inappropriate for score representation and manipulation. In S.T.L., one of the main functions of iterators is to allow access to container members. In C.P.N.View, the iterator is used to carry out the substantial scoping resolution. Automatic scorping resolution is essential in order to achieve an appropriate level of abstraction or complexity hiding. Iterators must be able to randomly access objects (e.g. the uppermost object half ways through bar no 22 in the second violin line). Iterators may be required to move vertically, to access harmony or horizontally to follow a melody. The iterator must, at the same time keep track of scoping information about aspects of the current context such as clef, key signature, time signature, metronome settings, tempo indications, accidental alterations and location within a bar. This is necessary in order to free the user from the considerable scoping complexities, that would otherwise tend to get in the way of the development.

3. COMMON PRACTICE NOTATION VIEW

C.P.N.View[3][4] is a score representation written in C++, that was developed in the mid-1990s. It implements a representation of scores as containers and provides iterators for use with algorithms. A score object is created either algorithmically or by using one of the components for converting from various file-based representations (ALMA, *kern, NIFF, EsAC).

Creating a score object:

Score s(filename);

One or more score iterators may be created in order to gain access to the internal objects in the score as

ScoreIterator si(s);

For the initial examples we will assume that the score is monophonic.

A number of functions exist to move the score iterator about the score. Random access is achieved using the **locate** member function. This moves the iterator to an arbitrary place in the score

si.locate(NOTE, 23);

will move the iterator **si** to the 23rd note of the score. This function returns a TRUE/FALSE value to signal the success or otherwise of the operation. For example if we call this function on a score that has only 22 notes, we get a FALSE result. Almost all of the functions in C.P.N.View return a TRUE/FALSE result, where appropriate.

si.locate(BAR, 20);

moves the iterator **si** to the start of the 20th bar of the score, if it exists.

Relative movement of the iterator is achieved by the **step** member function. This function may take a parameter, indicating the kind of object that it is required to move to. The following code fragment may be used to traverse all of the notes of the score contained in **filename**.

```
Score s(filename);
ScoreIterator si(s);
while (si.step(NOTE))
{
        doSomething(si);
}
```

Iterators in C.P.N.View are used to directly extract information about the objects in the score. C.P.N.View iterators carry out domain level processing. Much of this processing is involved with resolving contextual information. For example, where a score iterator points to a note, the member function **pitch12()** returns the chromatic note number (effectively the MIDI note number). C.P.N.View performs the following operations automatically. (1) key signature in use; (2) checks if any accidental alterations are present since the start of the bar in which the note in question resides, and (3) calculates relevant adjustments to the final pitch of the current note. Using all of this information the correct pitch12 value is returned.

The following fragment illustrates how the constructs discussed so far can be used to identify and print out the highest and lowest note of a piece and to calculate the pitch range of the piece.

```
Score s(filename);
ScoreIterator si(s);
si.step(NOTE);
ScoreIterator highest = si, lowest = si;
while (si.step(NOTE))
{
if ( si.getPitch12() < lowest.getPitch12()) lowest = si;
if (si.getPitch12() > highest.getPitch12()) highest = si;
}
std::cout << "\nHighest note is " << highest;
std::cout << "\nLowest note is " << lowest;
std::cout << "\nRange is " << highest.getPitch12() – lowest.getPitch12()
<< " semitones.\n";
```

Similarly, a program to locate the longest and shortest note in a piece, excluding grace notes,

```
Score s(filename);
ScoreIterator si(s);
si.step(NOTE);
ScoreIterator longest = si, shortest = si;
while (si.step(NOTE))
{
   if (!si.hasAttribute(GRACE_NOTE))
   {
            if ( si.getRDuration() < shortest.getRDuration()) shortest = si;
            if (si..getRDuration( ) > highest.getRDuration()) longest = si;
   }
}
std::cout << "\nLongest note is " << longest;
std::cout << "\nShortest note is " << shortest;
```

A question arises of what happens if we run these programs on a polyphonic score? The answer is that these will work and produce meaningful results.

To explain what happens it is necessary to consider two cases. The first one is where a score consist of a single stave, but has simultaneous notes. Some examples of this will be found in string music in which multiple stopping occurs. More complicated examples happen in scores where two lines occupy the same stave.

An iterator in C.P.N.View has an internal mode setting of MONO or POLY. In MONO mode the iterator traverses the uppermost notes on each stave only, and skips others. In POLY mode the iterator traverses all of the notes, moving vertically, from top to bottom, where possible, and moving to the next highest rightmost object otherwise.

Functions exist for setting and querying the scanning mode of an iterator.

si.putScanMode(MONO);

si.putScanMode(POLY);

getScanMode();

Figures 1 and 2 show iterators in MONO and POLY mode operating on a single stave piece.

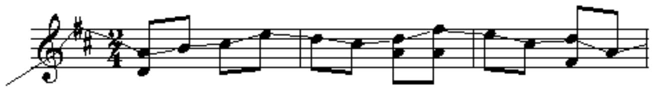

Figure.1 Single stave traversal in MONO mode

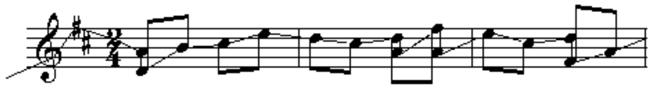

Figure 2 Single stave traversal in POLY mode

By default, the iterators created above will have MONO scan modes if the score contained in **filename** has one stave.

On the other hand, if the score is a multistave score, the scan mode will default to POLY, and the iterator will scan all of the objects in the score. To understand how the iterators scan across multiple staves, it is necessary to regard the score as being divided vertically into windows. Each window has an associated width, corresponding to a time span. The left and right borders of each window correspond to onsets or offsets of notes or rests. A window may not contain internal onsets or offsets.

Successive calls to the **step()** function moves the iterator vertically, wherever possible, from the uppermost object on the top stave in a window to objects on the lowermost stave in the same window. Where the score iterator points to the lowermost allowable object, a call made to **step()** moves the iterator to the uppermost object in the next adjacent window to the right. Figure 3 shows an such an iterator. Figure 4 demonstrates the concept of dividing the score into vertical windows.

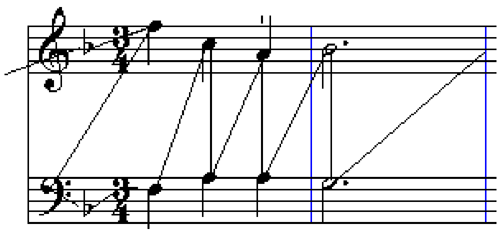

Figure 3 Multi-stave traversal in POLY mode

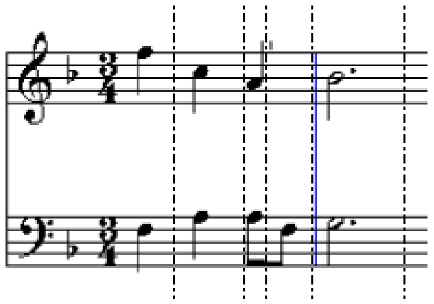

Figure 4 Multi-stave score divided into vertical windows

The program fragments above will work correctly with a polyphonic score and will search through all of the notes present. The occurrence of different clefs, changes in key and accidental changes in the score are all dealt automatically.

In cases where we want to scan individual staves of a polyphonic score, the constructor for the ScoreIterator takes an additional parameter corresponding to the stave number. The uppermost stave is numbered 0. An iterator created in this way will have a default scanning mode of MONO. To access objects on the last stave of a polyphonic score that has 10 staves, the following iterator could be used.

ScoreIterator si(s, 9);

In the previous examples, we have seen use of the member functions **locate**, **step**, **getPitch12**, **getRDuration**, **getScanMode** and **putScanMode**. A design strategy arises in creating such information retrieving functions. One could aim to design a minimal set of functions to retrieve the basic information content from the score. Such a minimal set will compromise convenience. The opposite approach of providing functions to retrieve every conceivable form will make things more difficult for the user, who will have a larger set to sift through and remember. The current set of 131 functions and operators might appear to lean towards the second approach. However many of these operators and functions cluster into families and thereby reduce the cognitive load in familiarization. Also many of these group into meaningful pairs. For example most member functions that start with 'get' have a counterpart that starts with 'put'. Additionally some of these are more frequently used than others. For example the **step** and **locate** functions are the main navigation mechanisms. There is very little additional to learn. The **getPitch12**, **getRDuration** deliver information similar to that available in a basic MIDI file. A short review of some of the main functions is given below.

The **getTag** member function give the type of object that is current.

If the current object is a note or rest, duration values may be retrieved

getHead() returns the head value,

getDots() returns the dot count,

getRDuration() retrieves the rational time value of the note or rest, including the resolution of groupette scoping.

Pitch information can be retrieved in many forms, including

getAlpha() returns the alphabetic note name,

getOctave() returns the octave number,

getAccid() returns details of any accidental placed directly on the note,

getPitch12() returns the MIDI pitch number of the note, with all necessary scoping resolved,

Many function return scoping information. These include the self obvious **getKeySig(), getTimeSig(), getClef()**, **getBarNo()**.

getBarDist() returns the distance of the current position from the start of the bar.

A facility exists for annotating any score object, and for querying these annotations.

4. IR EXAMPLES

The following illustrates the use of C.P.N.View in an IR context. It contains a complete implementation of the dynamic programming algorithm as documented in Sequence-Based Melodic Comparison: A Dynamic-Programming Approach[5]. The two encoded score fragments used in the algorithm have been encoded in two files that appear in lines 1 and 2. These are "Innsbruck ich muss dich lassen" and "Nun ruhen alle Waelder".

This algorithm is based on the concept of a minimal edit cost of transforming a source melody into a target melody, using operations of insert, delete and replace. The cost matrix d, represents the minimal cost of transforming the notes of the source tune, represented in rows, into notes of the target tune represented across the columns. The recurrence equations for generating the matrix are

$$d_{00} = 0 \qquad \text{a1}$$

$$d_{i0} = d_{i-1,0} + w(a_i, \phi), i \geq 1 \qquad \text{a2}$$

$$d_{0j} = d_{0,j-1} + w(\phi, b_j), j \geq 1 \qquad \text{a3}$$

$$d_{ij} = \min\{d_{i-1,j} + w(a_i, \phi), d_{i-1,j-1} + w(a_i, b_j), d_{i,j-1} + w(\phi, b_j)\} \qquad \text{a4}$$

Where

$w(a_i, \phi)$, is the cost of inserting note a_i,

$w(\phi, b_j)$, is the cost of deleting b_j,

both of these have value 4 in this example.

$w(a_i, b_j)$, is the cost of substituting a_i with b_j,

This cost is calculated by taking the absolute value of the difference in MIDI note number, then adding half the absolute value of the difference in duration measured in 16th note units.

```
Score s1("C:\\Mdb\\Others\\inns.alm");                              // 1
Score s2("C:\\Mdb\\Others\\nur.alm");                               // 2
ScoreIterator si1(s1);                                              // 3
ScoreIterator si2(s2);                                              // 4
ScoreIterator siAr1[100] = {ScoreIterator()},
              siAr2[100]= {ScoreIterator()};                        // 5
int length1 = 0;                                                    // 6
while ( si1.step(NOTE)) siAr1[++length1] = si1;                     // 7
int length2 = 0;                                                    // 8
while ( si2.step(NOTE)) siAr2[++length2] = si2;                     // 9
double diffMatrix[100][100];                                        // 10
int i, j;                                                           // 11
diffMatrix[0][0] = 0.0;


for ( i = 1; i <= length1; i++)                                     //12
        diffMatrix[i][0] = diffMatrix[i-1][0] + 4.0;                //13
for ( j = 1; j <= length2; j++)                                     //14
        diffMatrix[0][j] = diffMatrix[0][j-1] + 4.0;                // 15

for ( i = 1; i <= length1; i++)                                     // 16
{
 for ( j = 1; j <= length2; j++)                                    // 17
 {
  diffMatrix[i][j] =                                                // 18
   min3(
    diffMatrix[i-1][j] + 4.0,                                       // 19
    diffMatrix[i][j-1] + 4.0,                                       // 20
    diffMatrix[i-1][j-1] +                                          // 21
     fabs(siAr1[i].getPitch12() - siAr2[j].getPitch12()) +
     0.5*16*fabs(siAr1[i].getRDuration()- siAr2[j].getRDuration()));
 }
}
```

Lines 1 to 9 two arrays siAr1 and siAr2 are created, and contain iterators that point to each note of the score

Lines 12 to 15 correspond to equations a1, a2 and a3.

Lines 16 to 21 implement a4.

The function min3 is not documented here.

Table 1. The difference matrix diffMatrix.

		A	F	G	A	B	C	B	A
	0	4	8	12	16	20	24	28	32
F	4	6	6	10	14	18	22	26	30
F	8	8	6	8	12	16	20	24	28
G	12	10	10	6	10	14	18	22	26
A	16	14	14	10	9	13	17	19	22
C	20	18	18	14	13	14	15	19	22
B	24	22	22	18	17	16	18	15	19
A	28	26	26	22	21	20	21	19	15

Each cell in the matrix represents the difference at a particular point between the two fragments of “Innsbruck ich muss dich lassen” and “Nun ruhen alle Waelder”. The total distance between the two melodies is represented by the value in the bottom right corner, 15. The combination of edit operators that yield this result may also be determined by tracing the best path from the bottom, rightmost cell to the top left corner.

In this example, it will appear that little is to be gained from using the score representation instead of MIDI. However it is worth emphasising that the with score representation this algorithm may be refined since all of the information content of the score is available. Some examples of using score information, instead of MIDI may be gleaned from the following possibilities. Stressed notes may be distinguished by their position in the bar, using information for **getBarDist** and **getTimeSig** functions. Using these, one could allow greater weights for inserting and deleting stressed notes. Pitch information can be dealt with at a finer level by distinguishing between enharmonic versions of the same note. Hence a C sharp may be treated differently than a D flat. A policy on handling rests will be necessary, if this algorithm is to have general applicability. The algorithm is not explicit on how grace notes are handled. Are they to be treated as part of the pitch contour? Perhaps a researcher may wish to treat them as providing extra emphasis on the following note.

A final example will illustrate a partial implementation of this same basic algorithm, as it was designed by Mongeau and Sankoff[6]. They used a more sophisticated model for the calculation of replacement costs than in the previous example. The previous example was encoded above as part of line 21, by calculating the absolute value of the two MIDI note number.

fabs(siAr1[i].getPitch12() - siAr2[j].getPitch12())

Mongeau and Sankoff used the pitch differences between pairs of notes to calculate difference weights, based on consonances of their intervals. This involves a simple table look-up for diatonic notes. However for comparing pairs of notes in cases where one or other note involved is chromatic, an alternate table is used, as is reflected in the following algorithm. The array **deg** holds the difference weights for the diatonic table and **ton** holds the corresponding weights for chromatic intervals.

The algorithm checks if the notes involved are diatonic, and applies the appropriate transformation. The function **pitchD** is a simplified version of the production algorithm, that is applicable to music is in a major key only.

The function **noteInKey** was developed specifically for this application. Its coding is given below.

```
int ScoreIterator::noteInKey()
{
 int key = currentKs;
 int actual12 = getPitch12();
 int actual7 = getPitch7();
 int nrOctaves7 = actual7/7;
 int octaveDisp12 = nrOctaves7*12;
 int scaleStep7 = actual7%7;
 int diatonicSteps[] = {0,2,4,5,7,9,11};
 int unalteredPitch12 = octaveDisp12 + diatonicSteps[scaleStep7] +
                                                 getKeySigAdjust();
 if ( unalteredPitch12 == actual12 ) return TRUE;
 return FALSE;
}
```

```
double pitchD(ScoreIterator &si1, int major1, ScoreIterator & si2)
{
  double pitchDist = 0;
  if ( inKey(si1, major1 ))
   {
   double deg[] = {0.0, 0.9, 0.2, 0.5, 0.1, 0.35, 0.8};
   double ton[] = {0.6, 2.6, 2.3, 1.0, 1.0, 1.6, 1.8, 0.8,
                          1.3, 1.3, 2.2, 2.5};
   if ( si1.noteInKey() && si2.noteInKey() )
    {
    int diatonicSteps = fabs(si1.getPitch7() –
                                 si2.getPitch7());
     diatonicSteps = diatonicSteps % 7;
     pitchDist = deg[diatonicSteps];
    }
    else
    {
     int chromaticSteps = fabs(si1.getPitch12() –
                                  si2.getPitch12());
     chromaticSteps = chromaticSteps % 12;
     pitchDist = ton[chromaticSteps];
   }
  }
  return pitchDist;
}
```

5. CONCLUSION

Processing of music scores gives the prospect for accessing ever increasing corpora that have been created to high editorial standards. The Container/Iterator model gives an appropriate tool for algorithmic construction. Experience with C.P.N.View raises some interesting issues. An illustration of one such, that has been mentioned earlier in this paper is on devising an optimal set of operations to include in C.P.N.View. A minimal set, makes it easier for anyone to learn to use C.P.N.View. A more extensive set of operations, make it easier to write algorithms. A case in point is the **noteInKey** function above. This was not developed initially as part of C.P.N.View, but instead formed part of the implementation of the Mongeau and Sankoff algorithm. It was added to C.P.N.View, on the basis that it provided potential for reuse in other algorithms.

C.P.N.View provides a sufficiently abstract model of a score that it is potentially useable with a wide range of score representations, including some representations from notation packages. Currently C.P.N.View can accept input from score codings in ALMA, NIFF, *kern and EsAC. Some incomplete work has been done with SCORE and Enigma files.

6. REFERENCES

[1] Stepanov, A.A., and Lee, M. The Standard Template Library, Technical Report HPL-95911, Hewlett-Packard Laboratories, Palo Alto VA, February 1995.

[2] Plauger, P.J., Stepanov, A.A., Lee, M., and Musser, D.R. The C++ Stamdard Template Library, Prentice-Hall, NJ, 2001.

[3] Ó Maidín, D. Common Practice Notation View: a Score Representation for the Construction of Algorithms, Proceeding of the 1999 ICMC (Beijing,1999), ICMA, San Francisco, 248-251.

[4] Ó Maidín, D. “Common Practice Notation View User’ Manual” Technical Report UL-CSIS-98-02, University of Limerick, 1998.

[5] Smith, L., McNab, R., Witten, I. , Sequence-Based Melodic Comparison: A Dynamic-Programming Approach, in Melodic Similarity, Concepts, Procedures and Applications,Computing in Musicology 11. Hewlett W., Selfridge-Field, E. (eds). MIT Press 1998, 101-117.

[6] Mongeau, M.,Sankoff,D., Comparison of Musical Sequences, Computers and the Humanities 24,Kluwer Academic Publishers 1990, 161-175.