

MuM Tutorial

version 1.0.1

© 2016 NCM/UnB

Carlos Eduardo Mello
Núcleo de Computação Musical da Universidade de Brasília

Introduction

MuM is an algorithmic composition framework. This framework was developed to facilitate the manipulation of music information, when implementing music composition algorithms. The framework is designed around the concept of “Musical Material”, an open designation for any length of musical information, which can be manipulated in order to create more complex musical information. In this sense, music material can be a motive, a melody, a chord, a multi-voiced choral, a complete orchestral piece or a single musical interval. In another words, any data from which a composer could extract musical ideas for new compositions. MuM models this idea in a class of objects which can store any combination of notes and voices.

MuM is based on a discrete concept of sonic events, in which a composition is ultimately represented as organized groups of notes. Furthermore, even though there is support for manipulating and storing timbre information, the main focus is on pitch and rhythm organization. Thus, the framework is not particularly useful for music which is centered on continuous transformation of sound. The framework is built with standard (ANSI) C++. Therefore it is easily portable to various platforms in source code form. Its output may consist of either Csound score files or Standard MIDI Files (*TBD*).

MuM is free software. It may be copied, modified and reused in any way, as long as it retains the credits for the original developers (as noted in this document and in source code) in the derived work's source and documentation files. The original MuM implementation was written by Carlos Eduardo Mello, at the University of Brasilia. Continued maintenance and expansions are currently being implemented by NCM (*Núcleo de Computação Musial*) at UnB. The project is available as opensource through Git Hub.

Tutorial

This tutorial is an attempt to introduce basic ideas behind the design of the **MuM** algorithmic composition framework. As you will see down the road, using MuM is actually very simple and intuitive. You only need two things in order to employ MuM in your composition algorithms: (1) know how to program in C++ and (2) have a working knowledge of music theory. With these two pre-requisites, you can go through the lessons here and try to grasp the basic feeling for the framework. When you want to know more details about MuM's classes and their methods you can consult the MuM Reference Manual, which should be found in the same place you found this document.

MuM is small. Some may say it doesn't even deserve to be called a framework. In any case, it is a collection of interrelated classes, which help you define a common ground for developing composition algorithms. Some of these are internal classes which you'll probably never use directly. Others are only used sporadically, depending on your needs. But there are a couple of classes that most programs will use for everything: `MuNote` and `MuMaterial`. These are the classes we will concentrate on, in this tutorial.

Lesson 1 - Defining a Note

Declaring a note in MuM is as easy as declaring any other built in type like *int* or *float*:

```
MuNote aNote;
```

The next thing you need to know is how to define the parameters for that single note. As in many other classes in MuM, this is done by accessor methods. These usually come in pairs, one for *setting* and one for *getting* the data. The "get" methods take on the name of the parameter, starting in upper case; the "set" version simply adds "Set" to this name. Thus, for example, for dealing with the pitch of the note, you would use `Pitch()` and `SetPitch()`.

```
aNote.SetPitch(60);  
short thePitch = aNote.Pitch();
```

The code above defines the pitch of middle C (following MIDI pitch format) for the note and then retrieves this value from the note. Besides pitch, MuM uses 4 other fixed parameters: instrument, start time, duration and amplitude. (Beyond these five, extra parameters can be stored through a parameter block - see `MuParamBlock` for more details). So for a full definition of note we would use something like this:

```
MuNote theNote;  
theNote.SetInstr(1);  
theNote.SetStart(0.0);  
theNote.SetDur(3.0);  
theNote.SetPitch(67);  
theNote.SetAmp(0.5);
```

The lines above define all the built-in parameters for `theNote`. Instrument numbers are positive integers. 'Start' and 'Duration' are in seconds. 'Pitch' is a 7-bit integer in MIDI pitch format. 'Amp' is a value between 0.0 and 1.0 (For more details, check the **MuM Reference Manual**).

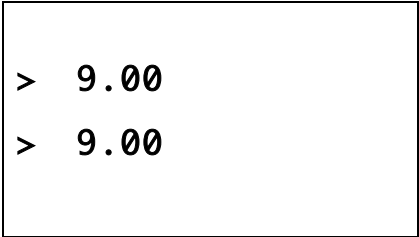
Lesson 2 - Note Output

The **MuNote** class exists mostly to carry note definitions to be modified by the **MuMaterial** class. It has no dynamically allocated members. Even the parameter block it contains is a regular member (which happens to be an object and handle its own internal memory allocation). Instances of the note class are always passed by value or reference. You will use this class whenever you need to generate isolated note data to place in a material object.

Other than the methods we've seen so far, MuNote has a couple of utility features which let us retrieve and debug pitch information in Csound format. A call to `CsPitch()` returns the note's octave and pitch class separated in the fields of a structure of type `cs_pitch`. `PitchString()` returns this same information in text form. `CsString()` returns the entire note definition in Csound format. Like this:

```
MuNote note;  
note.SetPitch(72);  
cs_pitch thePitch = note.CsPitch();  
cout << thePitch.octave << "." << thePitch.pitch << endl;  
cout << note.PitchString() << endl;
```

The last two lines of code produce exactly the same output to the screen:



```
> 9.00  
> 9.00
```

You would use the struct when you need to manipulate the octave/pitch class information separately (for example, to transpose the note and octave independently). MuNote also has a special `SetPitch()` method which takes a `cs_pitch` structure as input. This helps when we need to modify pitch class or octave, or both, without worrying about what the original pitch was. For example, to move a minor 15th down (Minor 7th plus octave), instead of trying to figure out the correct pitch number, you could do this:

```
cs_pitch csp = note.CsPitch();  
csp.pitch += 2;  
csp.octave -= 2;  
note.SetPitch(csp);
```

The text form, on the other hand can be useful for building debugging statements or when constructing a Csound score manually (as you will see further down the tutorial, in most situations MuM does it for you automatically).

Lesson 3 - Notes and Materials

Once you have a note definition there are various methods you can use to place it inside a material object. The simplest one is `AddNote()`. This is how you use it:

First declare an `MuMaterial` object - your musical material.

```
MuMaterial mat;
```

Then just call `AddNote()` and pass it the note object.

```
mat.AddNote(note);
```

Ok, but... so what? That doesn't do anything! We should try to do something more interesting. Let's say we want to build a sequenced pattern from a major scale. Something like this (**Figure 1**):



Here is one way to do it. First we define the melodic pattern. Then we write a loop to construct the notes for the pattern and place them in a material object.

```
MuMaterial mat, transp, seq;
MuNote note;
int i;
short pattern[8] = {60, 64, 62, 65, 64, 67, 65, 69};

for( i = 0; i < 8; i++)
{
    note.SetInstr(1);
    note.SetStart(i * 0.5);
    note.SetDur(0.5);
    note.SetPitch(pattern[i]);
    note.SetAmp(0.6);
    mat.AddNote(note);
}
```

At this point we have the pattern inside the material. Every note is an eighth-note (for $MM = 60$) with a dynamic range around *mezzoforte*. (By the way, this could be made a little more efficient if we left 'instrument', 'duration' and 'amplitude' out of the loop, since they never change.)

Now we need to sequence this melodic pattern to every scale degree by moving up the scale and transposing the material. It looks like we should use MuMaterial's transposition features. So let's try this:

```
short scale[7] = { 0, 2, 4, 5, 7, 9, 11 };

for( i = 0; i < 7; i++ )
{
    transp = mat;
    transp.Transpose( scale[i] );
    seq = seq + transp;
}
```

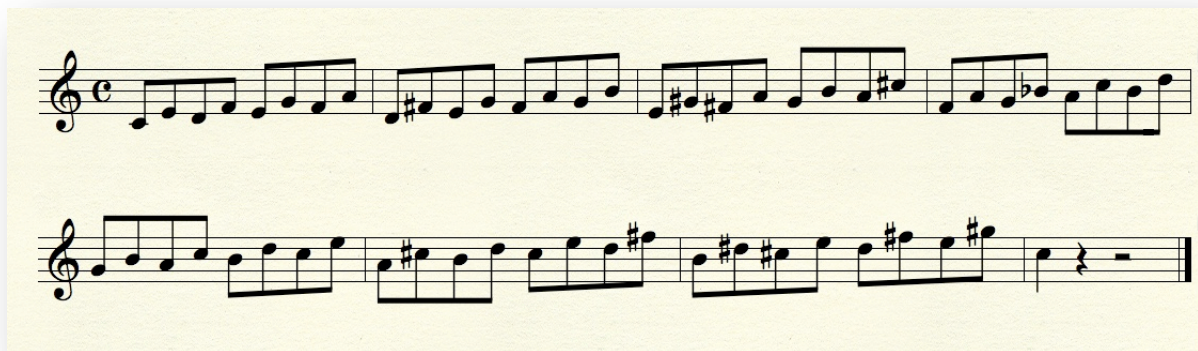
Here we got the intervals for transposition from the major scale structure. Every entry in the 'scale' array gives the exact interval between the corresponding degree (the array's index) and the first one (index zero). This is the interval to which the melodic pattern should be transposed, for each iteration of the loop. In order to keep the original pattern untouched, we copy it from 'mat' to 'transf', both instances of the MuMaterial class. This is

where OOP comes in handy - you can copy any material to any other material without having to deal with its internals, just like you assign an **int** to another **int**, or a **float** to another **float**. After we transpose the contents of our copied material ('transf'), we append it to the sequence by adding it to the 'seq' object, yet another MuMaterial instance.

This looks pretty easy doesn't it? But in fact it doesn't quite do the trick. Try listening to this by writing the Csound score and rendering the sound with a suitable instrument.

```
seq.SetFunctionTables("f1 0 4096 10 1");  
seq.Score("transposed.sco");
```

You will notice that the Transpose() method we used, performed a chromatic transposition. In other words, the intervals in the melodic pattern were transposed **exactly** like the original, without regard to the key. This is what it would look like in music notation (**Figure 2**).



In order to build the sequence as illustrated in **Figure 1**, we need to do a diatonic trasposition, that is, instead of moving each note by interval, we should move them by degree. To accomplish this in MuM we need to use the DiatonicTranspose() method. This method will take a key, a mode, a scale degree and a direction of transposition. The key is specified as a halfstep, with values between 0 (C) and 11 (B). The degree is an integer between 1 and 7. It tells DiatonicTranspose() to which scale degree the material should be transposed. For the mode choice MuM has a couple of constants: MAJOR_MODE and MINOR_MODE. Finally, the direction can be indicated as ASCENDING or DESCENDING, another pair of constants in the Library. This method works by calculating the degree distance between the first note in the material and the target degree -- the degree we pass to it. Then every note is treated as a degree and transposed by that same number of degrees.

Watch Out!

Currently, `DiatonicTranspose()` only works for materials which contain exclusively notes found in the declared key. If a given note's pitch is not part of that key/mode combination, the method returns and the material becomes invalid.

Figure 3 shows the complete code for our little exercise, using `DiatonicTranspose()`. The example ends with a long 'C', following the sequenced pattern.

```
// tutorial1.cpp
#include "MuMaterial.h"
int main(void)
{
    MuMaterial mat, transp, seq;
    MuNote note;
    int i;
    short pattern[8] = {60, 64, 62, 65, 64, 67, 65, 69};
    note.SetInstr(1);
    note.SetDur(0.5);
    note.SetAmp(0.7);

    for( i = 0; i < 8; i++)
    {
        note.SetStart(i * 0.5);
        note.SetPitch(pattern[i]);
        mat.AddNote(note);
    }

    for( i = 1; i < 8; i++ )
    {
        transp = mat;
        transp.DiatonicTranspose(0, MAJOR_MODE, i, ASCENDING);
        seq.Append(0, transp, 0);
    }

    note.SetDur(1.0);
    note.SetPitch(72);
    transp.Clear();
    transp.AddNote(note);
    seq = seq + transp;

    seq.SetDefaultFunctionTables();
    seq.Score("/Users/cem/Desktop/outFile");
    seq.Orchestra("/Users/cem/Desktop/outFile");

    return 0;
}
```

You may have noticed the call to `SetDefaultFunctionTables()` right before writing the score. These "function tables" are used by Csound to store predefined wave forms in memory, for use with its instruments. `MuMaterial` objects store and carry these around, so they can be placed in the score for playback. Depending on the instruments you play your score with, you may need other tables, which you can insert using `SetFunctionTables()`.

Lesson 4 - Random Melody

Very often, when trying to build a composition algorithm, we want to be able to produce results which are not always the same. For example, we may want to generate a melodic line that is fairly unpredictable, so that we may build original, interesting music from it. In general, computer programs utilize pseudo-random generators to that end. These are complex mathematical functions that generate, as output, a sequence of numbers which in turn *seem* not to present any particular pattern among them. In other words, the sequence appears to be chaotic. Such sequences only repeat themselves after thousands of iterations. Using them in strategic points in our code, you can build programs that produce interesting, varied patterns resulting in different music every time the program is run.

One of the most commonly used random generators is the Standard C Library function called `rand()`, which is readily available at any C/C++ implementation. Every time it is called, `rand()` produces a different integer between 0 and 32767 (`RAND_MAX`):

```
#define RAND_MAX 0x7FFF
```

In order to produce results which seem more random, `rand()` needs to be used in conjunction with `srand()` which "seeds" the generator's input, making the sequence start at different points. `srand()` takes as argument an integer which is used as seed for `rand()`. A good value to be used as seed is the output of another Standard C function: `time()`, which delivers a number representing the current clock time from the computer. Since this value is going to be different every time the program runs, `rand()` will always be started from a different point. Besides seeding the random generator, programmers usually have to control the random generator's output to suit each use they make of it. This is usually done with the "modulo" (%) operator, which restricts the random output to a certain range. For example, to get a number between 1 and 100 we can use:

```
srand(time(0));  
int x = (rand()%100 + 1);
```

In order to make life easier for the composer, MuM provides a very simple function that returns a random value between a lower and an upper limit. Not by chance, this function is called `Between()`. `Between()` is not a method of any of MuM's classes. It is a utility function which is available to any code that includes the Library. Here is a short example of how to use this function to achieve the same result as the previous lines.

```
int x = Between(1,100);
```

MuM uses `rand()` and `srand()` internally, so you don't have to bother with them. Instead, just insert a call to `Between()` every time you need to get changing values for something.

Now, so we can demonstrate some of the possibilities of random generators in composition algorithms, let's build a little phrase machine to generate quick, imitating, melodic patterns. Study the code bellow, compile, run it and listen to the results. To make things easier, the source code for this example is contained in the MuM Documentation you downloaded from GitHub. You can compile **tutorial2.cpp**, following the instructions in the documentation.

```
// tutorial2.cpp
#include "MuMaterial.h"

int main(void)
{
    MuMaterial mat1, mat2, matFinal;
    MuNote nota;
    srand(static_cast<unsigned int>(time(0)));
    nota.SetInstr(1);
    nota.SetStart(0.0);
    nota.SetDur(0.17);
    int turns = 10;
    while(turns--)
    {
        mat1.Clear();
        mat2.Clear();
        for(int i = 0; i < 7; i++)
        {
            if(i == 0)
                nota.SetAmp(0.75);
            else
                nota.SetAmp(0.4);
            nota.SetPitch(Between(48,60));
            mat1 += nota;
        } // end for()
        mat2 = mat1;
        int transp = 12; // 8va abaixo <==> 8va acima
    }
}
```

```

        if (rand()%2) transp *= -1;
        mat2.Transpose(transp);
        mat1 += mat2;
        matFinal += mat1;
    } // end while()

    matFinal.PlaybackWithCsound("/Users/cem/Desktop/melodia");
    return 0;
}

```

You'll probably notice that a simple run of fast changing notes, like the one in this example, doesn't always qualify as an interesting melody (although it may have its uses here and there...). Usually there needs to be structure and direction in the algorithm to produce a more musical result. The idea here was just to demonstrate the use of random values with MuM. For more involved algorithms, capable of delivering interesting materials, you need to structure your musical ideas so that they can be automated, and balance each choice you make in the structure, deciding which parts should be picked directly and which should be left to chance.

The code in this example also shows a few extra features of the MuM Library, which were added recently. The += operator has been overloaded to append entire materials or a single note to the receiving material. The latter allows us to build the melodic line directly in the receiving material, without having to calculate the notes' starting times, like in the previous example. The former makes it simpler and more intuitive to assemble the multiple patterns produced by the while loop into a single material (`matFinal`). Another very useful feature is the call to `PlaybackWithCsound()`, which makes a system call to start the Csound application, using the output from the calling material in csd format. This makes it possible for us to listen to the example by simply running the program we just compiled.