Automated Musical Part Writing

Kevin Deisz

03/26/2009

Abstract

This program writes music. Simply put, that is actually a relatively accurate summary of what this program does. Through a series of pseudo-randomly selected chords, user-generated constraints and constraint satisfaction code, this program writes music. The output is handled by a very handy programming language called lilypond, that engraves the generated musical progression into an outputted png and pdf. The output is also ported to a midi file so the user can hear what the program just wrote.

1 Introduction and Background

1.1 Variations and Themes

In 1680, a man named Johann Pachelbel came up with a theme written for three violins and a string bass. That theme was then arranged for a wide variety of other ensembles, but the most important part was the theme itself. Known as the Canon in D major, the sequence of notes became the basis for many pieces of the era. Handel used it for the main theme in the second movement of his Organ Concerto No. 11 in G minor, Mozart used it in Die Zauberflote, and even Haydn used it (albeit many years before) in his Opus 50 No. 2. This progression can be found in many modern pieces; unwitting artists perform this progression the unknowing crowds almost every day. Such artists as Matchbox 20, Avril Lavigne, Green Day, Blues Traveler and Aerosmith, to name a few. As demonstrated by this piece, it is not only the progression that separates pieces, but the overlying, more ornamented parts.

What this program attempts to do is to show that the same progressions can produce entirely different, random music. That even though the pieces may be quoting each other, the generated output is actually almost entirely unique. Therefore, this program writes its own music from user-specific variables, so as to allow the user to create unique music that may or may not be based off of others work.

1.2 Background

Many people have attempted (and succesfully too), to create almost this exact same project. Two examples are listed below.

1. Microsoft at one point created software that would take input from a microphone, and then create harmonies and a chordal progression that would play beneath the singer, as a sort of back-up music [7]. Their product was called "Songsmith", and it involved simply singing into the mic, and they advertised that it would do the rest for you. The product recieved mixed review, with some articles saying that the quality of the music was very poor, but that that was probably a good thing, because music being generated by a computer being as good as that of a human would be dangerous.

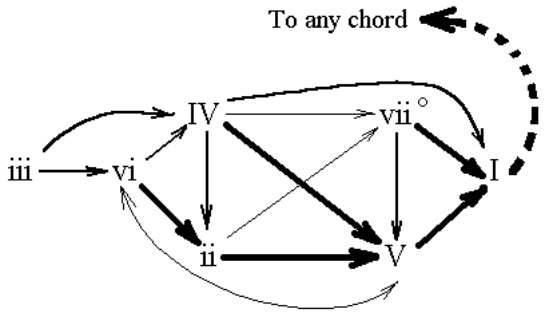
2. A man by the name of Bruce Jacob produced a product called "variations", which he used to write music for him. It is viewable on his webpage, along with all of its source code and the scant documentation he came up with [3]. The program goes by a set of constraints that he set up to model his own style of composition, which he then filters through a program he called the "ear". This ear program is actually a genetic algorithm designed to filter out music that he himself doesn't like. Obviously this program is very specifically designed to his tastes, but this is pretty much what I would be doing anyway. In reality, this is the embodiment of my project, simply put to a different genre. Jacob also came out with an essay on the subject of using algorithmic composition as a model for creativy, which ended up being very influencial in how I looked at this project [2].

In addition to the products, there are many people and organizations that have written numerous detailed reports on the subject of algorithmic composition. Florida International University came out with a great essay on the subject [1]. Stanford came out with a very helpful "Brief History of Algorithmic Composition" [6]. They also compiled an extensive bibliography in which can be found many useful resources on the subject [5]. In short, this is a subject that has been studied extensively, so I am only just brushing the tip of the iceberg with this project.

2 Process

2.1 Bass Progression

The first step in the development of this project was to create a working bass progression that would follow a general map of chords. This map of chords was taken from the Tonal Harmony textbook, and works only for major progressions. I then translated this map into a CSV file, where the first item of each line was the number of the chord (called scale degrees), and every other item in that line was a scale degree that that chord could progress to. All of these chords are then read in, processed, and added to a progressions array that is returned from the readin() method (viewable in Appendix A under "other_methods.py").



Major Key Chord Progressions

The next step, following the creation of the progression array, is to pass through the step() method x number of times, where x is the user-specified integer that was given at the very beginning. The incredibly simple step() method simply takes the array of possible neighbors from the progressions array and returns a random neighbor. This ensures that each and every composition is unique, so as not to make it a formula. However, this does pose some problems as to how to make the piece fit together as a whole, because at some point there needs to be a repeated motif, and currently returning a random chord does not lend itself well to this eventual goal. Obviously this is a place for further work to be done.

2.2 Secondary Dominants

I created a certain area of this project that added some difficulty for me. I added in secondary dominants, an area of music that makes everything slightly more advanced. These secondary dominant chords b̈orrowc̈hords from other keys and then place them in the regular key. Since my program outputs music in the key of C, the easiest example is a D major chord. Since G is the dominant in C major, and D is the dominant in G major (with an F#), D F# A is a secondary dominant of the dominant and C major. These chords added much depth to the music created.

2.3 Domains

Once the bass progression has been established, the domains for each chord for each voice can be established. For the most basic music, without deviating from the specific chordal tones, in each chord each voice can only be one of three notes. These notes are scale degrees one, three and five, but these can be in any octave. So, for each chord, each voice has an array of possible values, and those values are filled with the first, third and fifth scale degree, repeated for about three octaves. This gives each voice a relatively large domain, and the constraint satisfaction code I slightly larger area to work in, allowing the code a little more freedom, eventually resulting in slightly more random music.

The domains themselves are created differently for different genres. In jazz, for example, the scales are completely different, and the progressions are determined entirely differently. Obviously Bach and Benny Goodman composed with different approaches. In order to bridge this gap and make the code more general, I've created a command-line argument for the musicwriter.py file that takes the filename of a python file and pulls the methods from it (also in Appendix A). The methods are named the same in each file so that it will not have to change names, and this way multiple files can be created in the future. Hopefully in the future this could be a web-based program, and these files could be generated by the user. For now however, they are all created manually, and the only one created so far is voice.py, which writes for four voices.

2.4 Constraints

The constraints, which are the basic partwriting rules broken down into their own binary components, are relatively simple to implement. There is only one constraint function, and the output is a boolean on whether or not the tried chord was valid. The four inputs are voice A, note a, voice B, note b. The voices are simply integers from 0 to 3, signifying soprano through bass. The notes are numerical values which are indexes in the notes array (in other_methods.py).

The first, and probably most important constraint so far, is that a higher voice cannot have lower notes than a lower voice. Other constraints include parallel fifths, which mean that if two voices are four notes apart, the next chord cannot see the same two voices at the same interval. Parallel octaves are another, which is the exact same as parallel fifths, except that they are seven notes apart. A more interesting constraint, that resulted in some peculiar learned behavior, was that the voices could not be more than one octave apart. In the beginning (before the domains were properly implemented), because of this constraint, the entire piece would go as low as it possibly could. The last constraint, which is still not done being written and implemented, is that each chord must have a first degree (called the root), a third, and a fifth. This constraint is going to be the most difficult to implement, because not long is everything binary. In order for this to work, the note has to be linked to not only its neighbor, but also every other note in the chord. This involves ternary constraints, which will take quite a bit more research before they are implemented.

In order to link all of the voices together, and to make sure that they follow all of the rules created by the constraints, a map is created, and each voice is added to the map. Each voice is then given an array, that contains other mini-arrays that are simply pairs. Each pair contains a neighboring voice (which in this case, neighboring means any and all), and a pointer to the constraint function. This way, when it is time to choose the notes, the code need simply parse through this array, pull out the neighbor, and pass the neighbor and its attempted note to the constraint function dictated by the pointer.

2.5 Constraint Satisfaction

Once the domain and constaint maps are created, it is time to pass everything to the constraint satisfaction code. This code is actually entirely generic, and has many other uses. A new map is created, called assignments, which will be the output. This blank map is then passed to the main method in the csp file (Appendix A, under csp_solver_4.py), because the main method is a recursive method that will stop when all of the assignments have been made. The code itself implements its own relatively complex algorithms that are separate from the actual music writing. These include a least-constraining value function, which chooses the note that will leave the most options up to its neighbors, a function that sorts by the amount of unassigned neighbors, and a few other tricks to make it execute more quickly.

2.6 Ornamenting the Output

For now, this process is relatively simple, but in the future this is going to be the portion of the process that makes the music the most unique. This is the portion of the code that will add the non-chordal tones (NCTs), which are the notes not defined by the bass progression (i.e. not the root, third or fifth). Currently, the only NCTs that have been implemented are passing tones, which are notes that are added between other chordal tones to produce a scale-like effect. They are generated if a voice makes a jump over one note, C to E for example. The ornament() method would then make the C an eighth note, add in a D eighth note proceeding the C, and then end with an E quarter note. This makes the music far more interesting, as up until this point there was no variance in the rhythm. In the future, other NCTs will be added to make it more interesting, like appoggiatura, escape tones, and suspensions.

As of now, the only rhythmic deviation from straight quarter notes is described above, except for the very end. The add_ending() method takes the array and very simply makes the last chord a whole note. This adds to the music a sort of definitive ending that allows the listener to know that is the point at which to clap. In the future, other rhythmic devices will be implemented as well, such as syncopation and hemiola.

2.7 Output

Once all of the bass progression is decided, the constraints and domains established, everything passed through the constraint satisfaction and the output ornamented, it is finally time to print out the result. Thankfully, there is a programming language much like LaTeX for music, called lilypond. Lilypond has sections defined by back slashes, that allows the user to very simply output their music with minimal effort. As is, the printout method works wonders, which takes all four voices, and prints them out individually. The top two voices are printed out in the upper staff, which is housed in the treble clef, while the lower two voices are written in the bass clef, in the lower staff. This paper will not go

into the actual syntax of lilypond files, but you can see the output in Appendix B.

Lilypond thankfully has a helpful utility that allows the user to also output a midi file that contains the generated music. This means that every time this program is run, it generates the sheet music, and then a midi file that plays the sheet music. Since it is the same naming convention every time, it was very easy to make a webpage in which to view this output. Hence, a simple html page was written to house this project, the output and the midi, viewable at http://www.tjhsst.edu/ kdeisz/?page=musicwriter. The midi file and the generated png image will always be the latest files generated by the program.

3 Results

The outputted music was surprisingly complex. The program generated relatively good music, though there did lack a certain human touch. While one could discover general themes throughout the music, its very clear that these are completely circumstancial and not intended upon. The next major step in this research is going to be how to have the program generate a general theme for the piece and elaborate on that, because currently the randomness weighs on the listener. Additionally, because the triads (chords with a root, third and fifth) have not been fully implemented, at times the music seems empty in the middle parts.

4 Appendix A - Code

4.1 musicwriter.py

```
1 #!usr/bin/python
2 # Kevin Deisz
3 #
4
5 import sys
6 import other_methods
7 instrument_file = __import__( "instruments/" + sys.argv[1] )
8
9 num_chords = int( sys.argv[2] )-1
10 progressions = other_methods.readin( "major.csv" )
11
12 chords = [ '1' ]
13 for x in xrange( num_chords ):
14     chords.append( other_methods.step( progressions, chords[x] ) )
15
16 parts = [ [17,21,24,26] ]
17 for x in xrange( len( chords )-1 ):
18     parts.append( instrument_file.begin( parts[x], chords[x+1] ) )
19
20 final = []
21 for x in xrange( len( parts ) ):
22     line = []
```

```
23      for y in xrange( len( parts[x] ) ):
24          line.append( other_methods.getnote( parts[x][y] ) )
25      final.append( line )
26  final4 = other_methods.separate( final )
27
28  final3 = []
29  final3.append( final4[0] )
30  final3.append( final4[1] )
31  final3.append( other_methods.ornament( final4[2] ) )
32  final3.append( other_methods.ornament( final4[3] ) )
33
34  final2 = other_methods.add_ending( final3 )
35  for part in final2:
36      if part[0][len(part[0])-1] != "8" and part[0][len(part[0])-1]:
37          part[0] = part[0] + "4"
38
39  other_methods.printout( final2 )
```

4.2 other_methods.py

```
 1  import random, os, sys
 2
 3  def add_ending( parts ):
 4      for x in xrange( 4 ):
 5          if parts[x][len(parts[x])-1][-1] == "4" or parts[x][len(
                parts[x])-1][-1] == "8":
 6              parts[x][len(parts[x])-1] = parts[x][len(parts[x])
                    -1][0:len(parts[x][len(parts[x])-1])-1]
 7          parts[x][len(parts[x])-1] = parts[x][len(parts[x])-1] + "1"
 8      return parts
 9
10  def getnote( num ):
11      note_names = [ None, "a,,," , "b,,," ,
12          "c,,", "d,,", "e,,", "f,,", "g,,", "a,,", "b,,",
13          "c,", "d,", "e,", "f,", "g,", "a,", "b,",
14          "c", "d", "e", "f", "g", "a", "b",
15          "c'", "d'", "e'", "f'", "g'", "a'", "b'",
16          "c''", "d''", "e''", "f''", "g''", "a''", "b''",
17          "c'''", "d'''", "e'''", "f'''", "g'''", "a'''", "b'''",
18          "c''''" ]
19      return note_names[ num ]
20
21  def getnum( note ):
22      note_names = [ None, "a,,," , "b,,," ,
23          "c,,", "d,,", "e,,", "f,,", "g,,", "a,,", "b,,",
24          "c,", "d,", "e,", "f,", "g,", "a,", "b,",
25          "c", "d", "e", "f", "g", "a", "b",
26          "c'", "d'", "e'", "f'", "g'", "a'", "b'",
27          "c''", "d''", "e''", "f''", "g''", "a''", "b''",
28          "c'''", "d'''", "e'''", "f'''", "g'''", "a'''", "b'''",
29          "c''''" ]
30      for x in xrange( len( note_names ) ):
31          if note_names[x] == note:
32              return x
33      return -1
34
35  def ornament( part ):
```

```
36      retval = []
37      need4 = False
38
39      for note in xrange( len( part )-1 ):
40          first, second = getnum( part[note] ), getnum( part[note+1]
                )
41
42          if abs( first-second )==2:
43              retval.append( "%s8" % ( part[note]) )
44              if first > second:
45                  retval.append( "%s" % (getnote( getnum( part[note]
                        )-1 )) )
46              else:
47                  retval.append( "%s" % (getnote( getnum( part[note]
                        )+1 )) )
48              need4 = True
49          else:
50              if need4:
51                  retval.append( "%s4" % (part[note]) )
52                  need4 = False
53              else:
54                  retval.append( part[note] )
55      if need4:
56          retval.append( "%s4" % (part[len(part)-1] ) )
57          need4 = False
58      else:
59          retval.append( "%s" % (part[len(part)-1] ) )
60      return retval
61
62  def printout( parts ):
63      fh = open( "output.ly", "w" )
64      fh.write( "\\version \"2.12.1\"\n\n" )
65
66      def printvoice( voice ):
67          for x in voice:
68              fh.write( "%s " % (x) )
69          fh.write( "} " )
70
71      fh.write( "\\header{\n" )
72      fh.write( "\ttitle=\"Generated Music\"\n" )
73      fh.write( "\tcomposer=\"Kevin Deisz\"\n" )
74      fh.write( "}\n\n" )
75
76      fh.write( "\\score {<<\n\t\\new Staff <<\n" )
77      fh.write( "\t\t\\tempo 4=160\n" )
78      fh.write( "\t\t\\clef treble\n\t\t{ " )
79      printvoice( parts[3] )
80
81      fh.write( "\\\\\n\t\t{ " )
82      printvoice( parts[2] )
83      fh.write( "\n\t>>\n" )
84
85      fh.write( "\t\\new Staff <<\n" )
86      fh.write( "\t\t\\tempo 4=160\n" )
87      fh.write( "\t\t\\clef bass\n\t\t{ " )
88      printvoice( parts[1] )
89
```

```
90      fh.write( "\\\\\n\t\t{ " )
91      printvoice( parts[0] )
92      fh.write( "\n\t>>\n>>\n" )
93
94      fh.write( "\t\\midi{ }\n" )
95      fh.write( "\t\\layout{ }\n}\n" )
96
97      fh.close()
98      os.system( "lilypond --png output.ly" )
99
100 def readin( filename ):
101     file = open( filename, "r" )
102     temp = file.read().split('\n')[:-1]
103
104     progressions = []
105     for i in temp:
106         item = i.split( ',' )[1:]
107         progressions.append( item )
108     return progressions
109
110 def separate( parts ):
111     retval = []
112
113     for part in xrange( 4 ):
114         line = []
115         for chord in parts:
116             line.append( chord[part] )
117         retval.append( line )
118     return retval
119
120 def sharp_seventh( parts ):
121     for part in parts:
122         for x in range( 1, len( part ) ):
123             if part[x][0] == "g":
124                 part[x] = part[x][0] + "is" + part[x][1:len(part[x
                        ])]
125     return parts
126
127 def step( progressions, curchord ):
128     neighbors = progressions[ int( curchord )-1 ]
129     return neighbors[ random.randint( 0, len( neighbors )-1 ) ]
```

4.3 voice.py

```
1  #!usr/bin/python
2  # Kevin Deisz
3  #
4
5  import sys
6  import other_methods
7  import csp_solver_4
8
9  # ---------------------------------------- #
10
11 current_chord = []
12 def constraint_func( A, a, B, b ): # --- voice crossing, distancing
       , parallel 5ths and 8ths --- #
```

```
13      global current_chord
14      if ( (a<b and A<B) or (a>b and A>B) ):
15          if abs(A-B)==1 and ( abs(a-b)>7 ):
16              return False
17          if A!=0 and B!=0 and ( abs(current_chord[A]-current_chord[B
                ])==15 and abs(a-b)==15 ):
18              return False
19          if A!=0 and B!=0 and ( abs(current_chord[A]-current_chord[B
                ])==7 and abs(a-b)==7 ):
20              return False
21          if ( abs(current_chord[A]-current_chord[B])==4 and abs(a-b)
                ==4 ):
22              return False
23          return True
24      return False
25
26  def begin( current, next ):
27      next = int( next )
28      min, max = current[0], current[3]
29
30      possible, options = [ next+2, next+4, next+6 ], []
31      for x in xrange( len( possible ) ):
32          if possible[x] > 7:
33              possible[x] = possible[x]-7
34
35      for opt in possible:
36          for x in range( 1, 5 ):
37              val = opt+x*7
38              if val < 12 or val > 36:
39                  continue
40
41              if ( val ) > min and ( val ) < max:
42                  if val not in options:
43                      options.append( val )
44              elif abs( ( val ) - min ) < 4:
45                  if val not in options:
46                      options.append( val )
47              elif abs( ( val ) - max ) < 4:
48                  if val not in options:
49                      options.append( val )
50      return findvalues( current, options )
51
52  def findvalues( current, options ):
53      global current_chord
54      current_chord = current
55
56      domains = {}
57      for x in xrange( len( current ) ):
58          domain = []
59          for y in options:
60              if abs( current[x]-y ) < 4:
61                  domain.append( y )
62          domains[ x ] = domain
63      limit_domains( domains )
64
65      grid, constraints = {}, {}
66      for x in xrange( 4 ):
```

```
67          grid[ x ] = []
68          for y in xrange( 4 ):
69              if x!=y:
70                  grid[ x ].append( y )
71      for node in grid:
72          constraints[ node ] = []
73          for neighbor in grid[ node ]:
74              constraints[ node ].append( [ neighbor, constraint_func
                    ] )
75
76      assignments = {}
77      csp_solver_4.backtracking( domains, assignments, constraints )
78
79      retval = []
80      for x in xrange( len( assignments ) ):
81          retval.append( assignments[x] )
82      return retval
83
84  def limit_domains( domains ):
85      for x in domains[0]:
86          if x<13 or x>22:
87              domains[0].remove( x )
88      for x in domains[1]:
89          if x<17 or x>26:
90              domains[1].remove( x )
91      for x in domains[2]:
92          if x<20 or x>29:
93              domains[2].remove( x )
94      for x in domains[3]:
95          if x<25 or x>34:
96              domains[3].remove( x )
97
98  # ------------------------------------------------ #
```

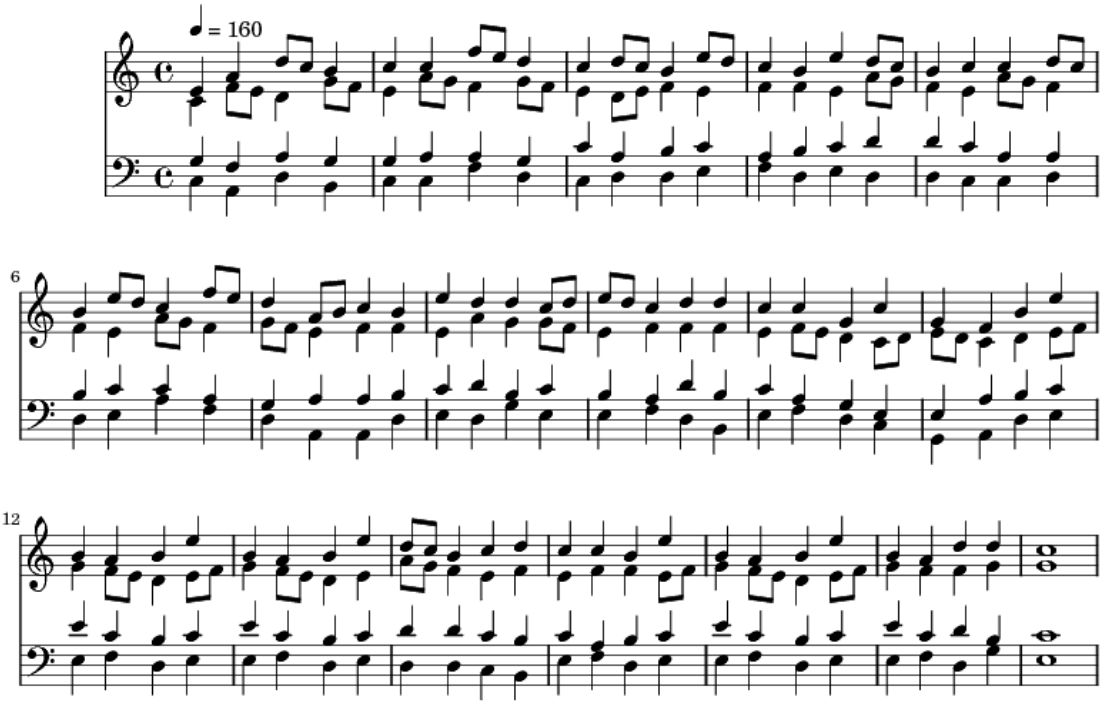
5 Appendix B - Output

5.1 Lilypond

```
1  \version "2.12.1"
2
3  \header{
4      title="Generated Music"
5      composer="Kevin Deisz"
6  }
7
8  \score {<<
9      \new Staff <<
10         \tempo 4=160
11         \clef treble
12         { e'4 a' d''8 c'' b'4 c'' c'' f''8 e'' d''4 c'' d''8 c'' b
               '4 e''8 d'' c''4 b' e'' d''8 c'' b'4 c'' c'' d''8 c'' b
               '4 e''8 d'' c''4 f''8 e'' d''4 a'8 b' c''4 b' e'' d'' d
               '' c''8 d'' e''8 d'' c''4 d'' d'' c'' c'' g' c'' g' f'
               b' e'' b' a' b' e'' b' a' b' e'' d''8 c'' b'4 c'' d'' c
               '' c'' b' e'' b' a' b' e'' b' a' d'' d'' c''1 } \\
```

```
13          { c'4 f'8 e' d'4 g'8 f' e'4 a'8 g' f'4 g'8 f' e'4 d'8 e' f
                '4 e' f' f' e' a'8 g' f'4 e' a'8 g' f'4 f' e' a'8 g' f
                '4 g'8 f' e'4 f' f' e' a' g' g'8 f' e'4 f' f' f' e' f'8
                 e' d'4 c'8 d' e'8 d' c'4 d' e'8 f' g'4 f'8 e' d'4 e'8
                f' g'4 f'8 e' d'4 e' a'8 g' f'4 e' f' e' f' f' e'8 f' g
                '4 f'8 e' d'4 e'8 f' g'4 f' f' g' g'1 }
14      >>
15      \new Staff <<
16          \tempo 4=160
17          \clef bass
18          { g4 f a g g a a g c' a b c' a b c' d' d' c' a a b c' c' a
                g a a b c' d' b c' b a d' b c' a g e e a b c' e' c' b c
                ' e' c' b c' d' d' c' b c' a b c' e' c' b c' e' c' d' b
                 c'1 } \\
19          { c4 a, d b, c c f d c d d e f d e d d c c d d e a f d a, a
                , d e d g e e f d b, e f d c g, a, d e e f d e e f d e
                d d c b, e f d e e f d e e f d g e1 }
20      >>
21  >>
22      \midi{ }
23      \layout{ }
24  }
```

5.2 Music



References

[1] Burns, Kristine H. "Algorithmic Composition". September 1996. Florida International University. 23 Jan. 2009 <http://eamusic.dartmouth.edu/~wowem/hardware/algorithmdefinition.html>.

[2] Jacob, Bruce. "Algorithmic Compositions As A Model Of Creativity". University of Maryland. 23 Jan. 2009 <http://www.ece.umd.edu/~blj/algorithmic_composition/algorithmicmodel.html>.

[3] Jacob, Bruce. "VARIATIONS: Algorithmic Compositions for Acoustic Instruments." University of Maryland. 20 Jan. 2009 <http://www.ece.umd.edu/~blj/algorithmic_composition/>.

[4] Kostka, Stefan and Payne, Dorothy (1996). *Tonal Harmony: With An Introduction To Twentieth Century Music*. 3rd ed. McGraw-Hill, Inc.: San Francisco.

[5] Kunze, Tobias. "Algorithmic Composition Bibliography". 1998. Stanford University. 16 Jan. 2009 <http://ccrma-www.stanford.edu/~tkunze/res/algobib.html>.

[6] Maurer, John A. "A Brief History of Algorithmic Composition". March 1999. Stanford University. 19 Jan. 2009 <http://ccrma.stanford.edu/~blackrse/algorithm.html>.

[7] "Microsoft Research Songsmith". 2009. Microsoft Corporation. 22 Jan. 2009 <http://research.microsoft.com/en-us/um/redmond/projects/songsmith/>.