David Cope
University of California at Santa Cruz
Music Center, 242
University of California
1156 High Street
Santa Cruz, CA 95064 U.S.A.
howell@ucsc.edu

A Musical Learning Algorithm

In this article I describe a computer program called Gradus (after Johann Joseph Fux's 1725 treatise *Gradus ad Parnassum*) that initially analyzes a set of model two-voice, one-against-one, first-species counterpoints in order to produce a series of compositional goals. Gradus then attempts to compose goal-correct counterpoints similar to the models using a given fixed voice called a *cantus firmus*. Initial notes for the new lines of these counterpoints are drawn from pre-compositional seed notes, the choice of which are optimized by consulting the program's previously saved successful compositions.

The article then describes how Gradus, in order to compose more quickly and mistake-free, backtracks from the ''dead ends'' it encounters, catalogs the conditions that led to these dead ends as rules, and avoids these conditions on subsequent runs with the same *cantus firmus* until backtracking is no longer necessary. Gradus then uses the rules it collects when encountering new *cantus firmi*, applying its accumulated experiences to increase the chances that it will succeed. Ultimately, the program learns to compose first-species counterpoint quickly, accurately, and without any need for backtracking. I have based the processes used in Gradus on those that I use when learning or re-learning species counterpoint.

This article includes a brief discussion of musical inference and a description of how the learning processes described may be seen as computationally inferring solutions to basic musical problems. The article then concludes with more elaborate examples created by the program, including a fugue exposition and counterpoint in a more dissonant, non-triadic style.

Background

Webster's College Dictionary defines ''learning'' as the ability ''to acquire knowledge of or skill in by study, instruction, or experience'' (Costello 1991, p. 772). Few computer programs actually learn, although the word ''learn'' is often freely used when describing programs such as spelling checkers and Internet search engines. The program I present in this article actually changes its behavior by modifying its own approaches to solving musical problems, becoming faster and more accurate over time. This program also chooses its own originating notes and produces specific and generic rules that can be instructive for teaching users themselves to create better results when working with two-voice, one-against-one, species counterpoint.

Although significant research in machine learning has taken place in the artificial intelligence community (see for example Mackintosh 1983; Michalski 1986; Mitchell 1997; Cherkassky 1998; Hinton and Sejnowski 1999; Baldi 2001; Cesa-Bianchi, Numao, and Reischuk 2002), very little similar research has taken place in the field of music. Gerhard Widmer's work (Widmer 1992; 1993) stands out among the few published examples of computational learning in music. In his 1992 article, Mr. Widmer argues the need for true learning programs in music and that music requires domain specificity. The program he describes involves user interaction to achieve its results, which are impressive because the program then alters itself to reduce backtracking. William Schottstaedt's counterpoint program (1989) creates effective output in all five species but without any learning involved. Cruz-Alcázar's and Vidal-Ruiz's work with grammatical inference (GI) algorithms (1997) holds potential but primarily applies to monophonic rather than polyphonic music.

David Evan Jones's CPA program (2000) provides an effective contrapuntal toolbox for composers but centers almost entirely on atonal music. Hõrnel and Menzel (1998) use feed-forward neural networks in their work, with learning accomplished through adjustment of nodal weights. However, learning with neural networks often requires extensive training and elaborate complementary algo-

Computer Music Journal, 28:3, pp. 12–27, Fall 2004
© 2004 Massachusetts Institute of Technology.

rithmic programming (see Miranda 2001, pp. 99–118). Other important work in the area of musical learning has been accomplished; however, the paucity of publications remains clear.

David Lewin's goal-oriented approach to species counterpoint (1983) clearly enhances one's ability to compose first-species counterpoint. However, this approach provides shortcuts that attempt to avoid the dead ends that rules cause rather than incorporating true learning processes. There are many such shortcuts to creating counterpoint. For example, in my book *Experiments in Musical Intelligence* (Cope 1996, p. 213), I describe an approach to creating algorithmic fugues that involves ''subtractive counterpoint,'' a technique that voids much of the need to backtrack. Again, however, such shortcuts do little to achieve learning, and hence I do not discuss them here.

The need exists therefore for music programs involving machine learning to produce effective tools for composers and to provide a better understanding of the ways in which humans learn musically. The program described here is available from my Web site at arts.ucsc.edu/faculty/cope and will run on any platform and requires only a version of the computer language Common Lisp.

The Goals

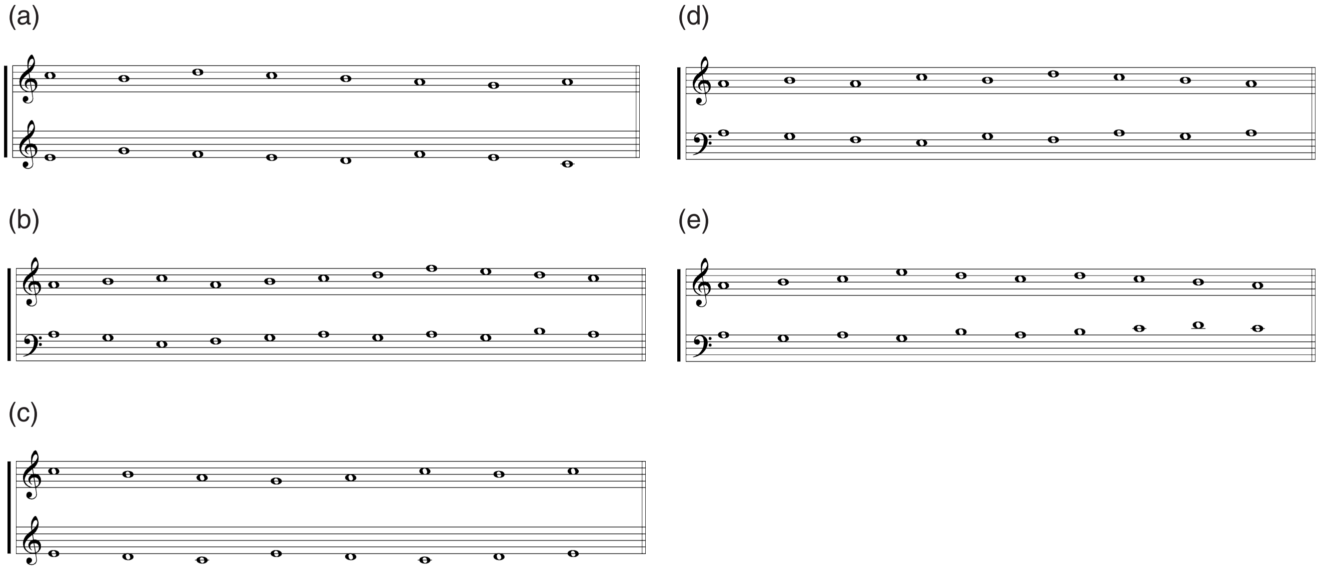
Species counterpoint poses particular composing problems that have challenged students for centuries. A single fixed line (called the *cantus firmus*) combines with a variable number of simultaneously sounding new lines that must adhere to a strict set of musical goals. Often, new lines lead to dead ends for which no correct choice is possible. Beginning students often restart only to find themselves in the same or similar predicaments. Many months of practice are typically required for even highly intelligent students to become facile at creating these new lines without backtracking. Although other kinds of music (such as canons, fugues, serial constructions, and so on) can cause similar conflicts and thus serve as interesting problems for computer programs to solve, I have chosen a rather straightforward type of species counterpoint known as two-voice, one-against-one, first-species counterpoint for the purposes of this demonstration. This is the type of counterpoint with which students usually begin and for which, I feel, the rules are most programmable. I will describe extensions of this software beyond its two-voice, one-against-one limitations later in this article.

Gradus uses six goal categories. Although these do not conform exactly to those of Fux and even omit certain details which many counterpoint instructors find important (climax, mode adherence, cadence, and so on) or further limit Fux's constraints (as in leaps of thirds only), most of Fux's general rules appear here in one form or another:

1. Simultaneously sounding notes must be consonant, forming intervals such as thirds, fifths, sixths, octaves, and so on
2. Voice motion must be stepwise (i.e., no repeated notes) with occasional leaps of thirds always followed by stepwise contrary motion
3. No simultaneous leaps in both voices are allowed
4. Parallel fifths and octaves must be avoided
5. Hidden fifths and octaves must be avoided
6. No more than two continuous same-direction motions are allowed

The more models used, the more accurately Gradus sets its goals. Therefore, I have chosen 50 two-part, one-against-one completed counterpoints to serve as models of music for Gradus to analyze. Each of these counterpoints incorporates the above goals. Figure 1 shows several of the 50 models from which the program—in its default state—sets its goals. In essence, Gradus assumes that all of the vertical intervals, parallel motions, skip numbers, and so on found in the models are allowable and places the vertical intervals, parallel motions, skip numbers, and so on it does not find—presumed to be illegal—in appropriate variables to serve as program goals. If the number of models to be used for goal analysis changes between runs of Gradus, the program can re-analyze the models, and users may re-set the corresponding goal's variable for further composition.

Figure 1. Five of the 50 models upon which the Gradus program bases its analysis of goals for creating good counterpoint.



As can be seen, the goals in Gradus are prohibitory rather than exemplary, informing the program of what not to do rather than what to do—a more standard approach to programming. Figure 2 provides musical examples for each of the six goals listed above as exemplified in the 50 models provided with the Gradus program. These goals help the two voices remain in consonant relationship and to retain their relative independence, which are important features of both 16th- and 18th-century contrapuntal composition.

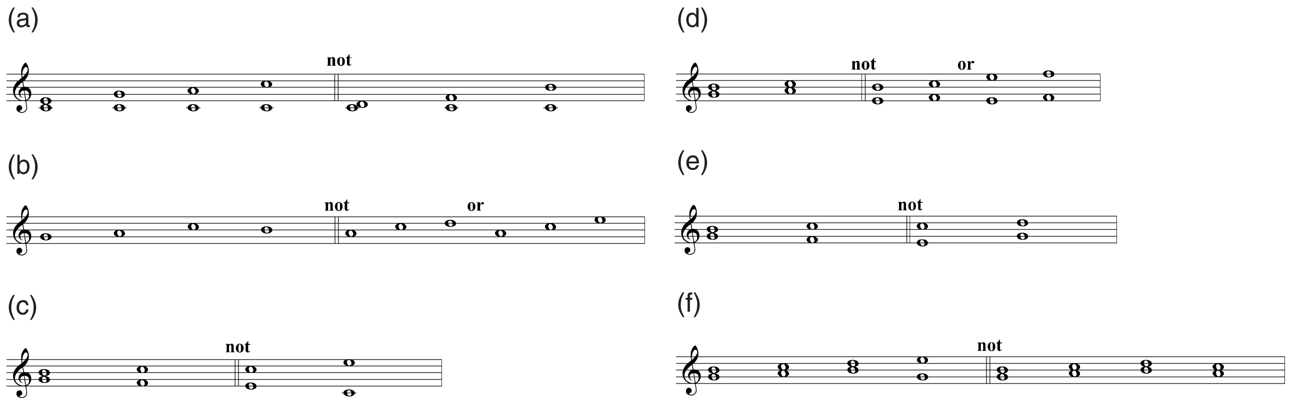
The Gradus program requires a seed note to begin its composition of a new accompanying line to the *cantus firmus*. This seed note acts as an origin for selecting a first note. Because composing a first note from this seed note follows the previously presented goals, the seed note generates a number of possibilities (four to be exact, as we will shortly see) from which the program chooses a logical initial pitch using stepwise motions and interval-of-a-third leaps only. Some seed notes will project several likely correct candidates, whereas other seed notes will project only one possible logical choice, a choice that occasionally may later prove incapable of creating a completed counterpoint. Therefore, selecting a good seed note is important for the composing process.

To make the best possible seed-note choice, Gradus saves its successful outputs with their generating seed notes and uses the seed notes of these successful lines to create new accompanying lines. To provide as broad a match for the comparison process, Gradus also saves *cantus firmus* templates. Templates (like rules discussed later) are measured in terms of diatonic steps rather than by exact intervals; in other words, the upward interval of a second equates to the number 1, the upward interval of a third to the number 2, and so on, with negative numbers referring to downward motion. A template takes the form of the number of vertical diatonic steps between the seed note and the beginning pitch of the *cantus firmus*, followed by a reduced, diatonic-step map of the *cantus firmus*, as in the following:

```
(-4 (5 0))
```

The first number here (–4) represents the number of diatonic steps between the seed note (not the first note of the successful accompanying line) and the first note of the *cantus firmus*. The list that follows this first number indicates respectively the number of diatonic steps between the note most removed (high or low) and the beginning note of the *cantus firmus,* and the number of diatonic steps be-

Figure 2. The program goals in musical notation where (a)–(f) relate to goals 1–6, respectively.



tween the last note of the *cantus firmus* and the first note of the *cantus firmus*. In the above case, the *cantus firmus* rises to a maximum five diatonic steps above the first note and ends with the same note as it began. With several of these successful *cantus firmus* templates saved in a global variable, the Gradus program can select the most similar map to the *cantus firmus* being used and then project the appropriate seed note. This process helps the program avoid false starts. If there are two (–4 (5 0)) templates stored but only one (–5 (5 0)) template stored, the program will select the (–4 (5 0)) template, because it has obviously proved more successful. The Gradus program thus uses previous experience to infer future choices.

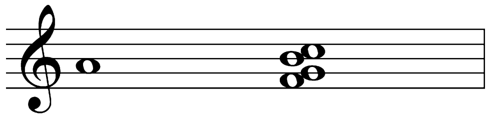
The Process

Gradus uses one of the *cantus firmi* included with the program or one provided by the user. *Cantus firmi* should follow the goals mentioned in the previous section. Failing to use a *cantus firmus* that follows these goals can make it impossible to compose an appropriate accompanying line or prevent the program from behaving as described. The default *cantus firmus* and all of the other *cantus firmi* provided with Gradus follow the first-species goals of the models provided with the program.

As just mentioned, Gradus can move only stepwise or leap a third up or down. Consequently, only four notes can be projected from a seed note, and each successive note in the new line as shown in Figure 3. Gradus initially chooses one of these notes randomly, which means that different results may occur each time the program is run, depending on whether the note chosen follows the program's goals. Gradus also requires a scale for diatonic tonal pitch selection, the default of which is the C-major scale, which allows the program to only select notes in the key of the provided *cantus firmi* (also in C major). Gradus is not currently designed to compose in other keys or use atonal scales and *cantus firmi*. However, only a relatively few small changes in a number of variables would be required to make this kind of variation possible.

The centerpiece of the Gradus program is its goal-testing cycle. This cycle contains the hoops through which a successful choice must pass: vertical consonance, avoidance of parallel fifths and octaves, leaps followed by contrary motion steps, no simultaneous leaps in both octaves, avoidance of direct fifths and octaves, and three or fewer consecutive and legal parallel-direction motions. The program looks ahead for its best choice by projecting forward one pitch from the currently chosen pitch. Comparing the results of this projection with the current rules (soon to be discussed) enables the program to avoid choosing a pitch that will conflict with the goals, which requires backtracking. Gradus then selects another pitch if the current choice creates a conflict. If no correct possibility among those available exists, then the program creates and saves a new rule and backtracks to find a different solution. I will discuss this process in more detail momentarily.

Figure 3. The four possible first-note choices for the seed note A.



Rules in the Gradus program consist of three or fewer motions and appear as follows:

```
(-2 (1 1)(-1 1))
```

A rule consists of three parts: (1) the number of vertical diatonic steps between the first affected note of the *cantus firmus* and the new line; (2) the number of diatonic steps between the affected notes of the *cantus firmus* itself; and (3) the number of diatonic steps between the related notes of the new line. The vertical difference is necessary because there may be diatonic step distances between the associated first notes of the two lines where the combination of voice motions will work correctly. For example, the counterpoint A-B-C/F-E-F separated by an octave resulting in a rule of (–2 (1 1)(–1 1)) is incorrect due to parallel fifths. However, the counterpoint A-B-C/A-G-A in the same register—equivalent to the rule (–7 (1 1)(–1 1))—is correct, but it would match the rule of (–2 (1 1)(–1 1)) if the vertical separation were not present to differentiate the two patterns. Voice motions alone simply do not provide enough information to be useful as rules. Using vertical distances and voice motions as rules also avoids the need to have rules that duplicate one another when diatonically transposed. For example, the pitch rules C-D-E/A-G-A and E-F-G/C-B-C in the same register are diatonically identical when transposed, but both would be necessary in a pitch-notated rules system, whereas the single diatonic-step rule of (–2 (1 1)(–1 1)) includes both of them as well as all other similar transpositions.

Composing in Gradus follows a straightforward process. At every point in choosing a new note, the program produces a set of four possible notes from which it can choose, seconds or thirds up or down from the current note. These four notes are tested using the previously described goals. If none of the four possibilities survives the tests, then the program backs up one note and begins again, this time with the offending note—the last note of the new line composed thus far—removed from the possible choices to avoid repeating the same mistake. If one or more of the four possibilities does survive the tests, then the program looks ahead with a kind of approximation. This approximation takes into account the next *cantus firmus* note but with a wildcard matching anything, and it compares this projection with the currently constituted rules. If a rule prevents all of the possible four projected new-line notes from occurring, then the program chooses another pitch from those currently available. The balance between inline tests and looking ahead to see if a rule exists to block a future move is precarious, but it ensures that back-tracking will only occur at times requiring new-rules creation and that wasted effort will be kept to a minimum.

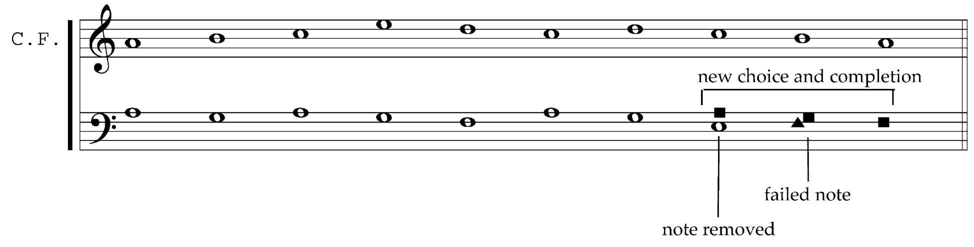
Figure 4 shows graphically how this process works. In this representation, the program has correctly completed all of the whole notes in the bass clef up to and including the E, the third note from the end. Because this E resulted from a leap downward, the program must—according to the goals—move stepwise in the opposite direction. Doing so, however, creates a dissonant tritone relation with the B, the second note from the end in the *cantus firmus*, causing an error (represented by the black triangle note). The program, therefore, creates a rule to cover this situation, backs up one note, subtracts the E from the possibilities available for a third note from the end, and chooses a more successful A (shown as a black square note). The program then continues to negotiate the remainder of the counterpoint successfully. In future attempts to compose with this *cantus firmus*, the program will look ahead and, upon encountering G as fourth note from the end, will see that selecting E produces a problem with the necessary F and will thus choose A as third note from the end to avoid back-tracking.

The look-ahead process in the Gradus program involves creating a rule on the fly—without adding it to the database—using an extension of the *cantus firmus* by one note beyond the current state of the new line along with the proposed new note, as well as a wildcard that will match anything found in the rules. The temporary rule of (–9 (1 –1 –1)(–1 –2 nil)), which is equivalent to C-D-C-B/A-G-E-? (fifth note from the end of Figure 4 and onward) where the last E represents the proposed new note

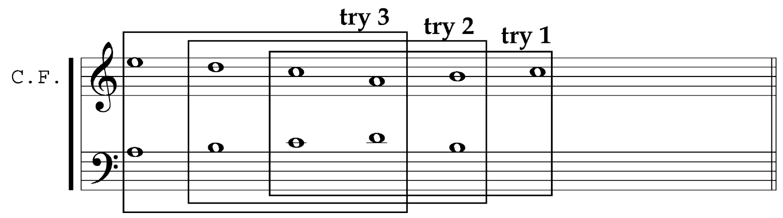
Figure 4. The Gradus backtracking process.

Figure 5. Backtracking to create new rules to avoid future backtracking.

(4)



(5)



and the "?" represents a wildcard, is a good example of such an on-the-fly rule. With the rule (–9 (1 –1 –1) (–1 –2 1)) already present in the rules variable, this example temporary rule will find it a match, and the program will seek another possibility as a new note. Because rules placed in the rules variable represent dead ends (i.e., all of the other three possible intervals substituted for the interval shown also create problems), the match correctly causes the program to choose another note for the one causing the problem. Hence, the look-ahead process avoids dead ends before they can occur.

When the look-ahead function finds a rule prohibiting the program from choosing a note, Gradus makes another note choice from those that pass the previously discussed tests. Often, however, there are no more notes to choose from, and the program must then backtrack. This may seem like wasted effort, as we have not reduced backtracking using this method. To solve this, the look-ahead function creates a new rule that prohibits the up-to-this-point correct note that led to the incorrect choice. This ensures that when this situation is encountered again with this *cantus firmus*, the program will choose another note (if possible) to avoid the collision ahead. This creation of rules prohibiting a correct choice may continue until it prohibits one of the possible correct notes projected from the seed note. This new rule, unlike normal rules because it disallows a pattern that by itself is perfectly correct, helps to eliminate backtracking entirely. This new rule is kept in a separate variable—in fact a temporary variable, since with another *cantus firmus* this correct choice may in fact lead to further correct choices. This process of creating rules for otherwise correct choices and storing them in temporary locations is extremely important to the Gradus program's learning processes.

Figure 5 demonstrates how this process works. Each of the rectangles here represents a rule. As can be seen in this instance, the process backs new rules up to the beginning of the composition with the result that the note A will be avoided in all subsequent runs of the program with this *cantus firmus*. In detail here, try 1 is foiled because the program has leapt to the last B in the lower voice, forcing a stepwise opposite direction C, which, if created, would produce parallel octaves. In try 2, the result of the program's looking ahead and discovering the problem with try 1, no other correct possibilities exist for the new line except the B, and thus it too fails. Try 3 does not work because no other choice exists here aside from the shown fourth-note D.

Figure 6 shows a typical set of Gradus program runs without any backtracking. The program identifies each attempt to extend the new line with the output text "working . . ." followed on the next line by the current state of the composition of the accompanying line in relation to the *cantus firmus*. Final output is presented in event-list format with

Figure 6. A typical set of runs without any backtracking.

Figure 7. The results of Figure 6 in standard musical notation.

Figure 8. Six outputs from the Gradus program showing different second lines to the same cantus firmus.

```
? (gradus)
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 b2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 b2 c3))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 b2 c3 d3))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 b2 c3 d3 c3))
((0 69 1000 1 90)(0 57 1000 2 90)(1000 71 1000 1 90)(1000 55 1000 2 90)
 (2000 72 1000 1 90) (2000 57 1000 2 90) (3000 76 1000 1 90)
 (3000 55 1000 2 90) (4000 74 1000 1 90) (4000 59 1000 2 90)
 (5000 72 1000 1 90) (5000 57 1000 2 90) (6000 74 1000 1 90)
 (6000 59 1000 2 90) (7000 72 1000 1 90) (7000 60 1000 2 90)
 (8000 71 1000 1 90) (8000 62 1000 2 90) (9000 69 1000 1 90)
 (9000 60 1000 2 90))
```

Figure 6

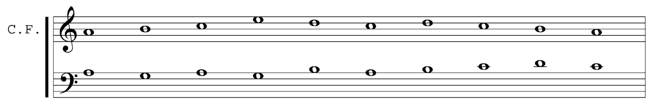
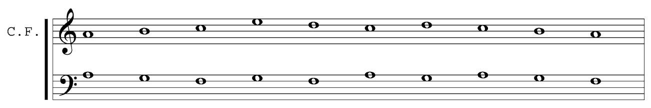
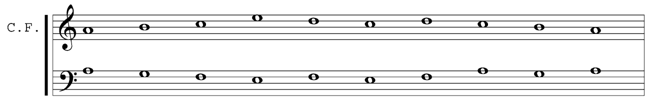


Figure 7

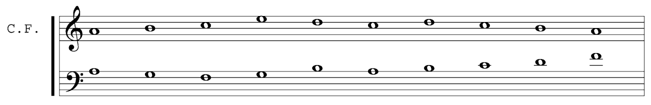
(a)



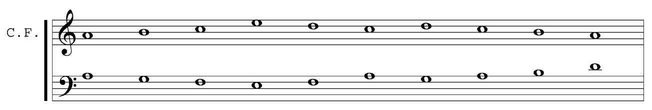
(b)



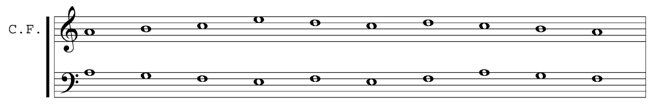
(c)



(d)



(e)



(f)

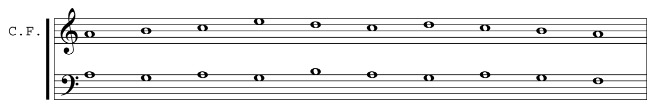


Figure 8

Figure 9. A typical set of runs with backtracking.

```
? (gradus)
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 e2))
backtracking.....there are now
1
rules.
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 f2))
backtracking.....there are now
2
rules.
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 b2))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 b2 c3))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 g2 a3 c3))
backtracking.....there are now
3
rules.
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 g2 a3 b3))
working.....
((a3 b3 c4 e4 d4 c4 d4 c4 b3 a3) (a2 g2 a2 g2 b2 a2 g2 a3 b3 d3))
((0 69 1000 1 90)(0 57 1000 2 90)(1000 71 1000 1 90)(1000 55 1000 2 90)
 (2000 72 1000 1 90) (2000 57 1000 2 90) (3000 76 1000 1 90)
 (3000 55 1000 2 90) (4000 74 1000 1 90) (4000 59 1000 2 90)
 (5000 72 1000 1 90) (5000 57 1000 2 90) (6000 74 1000 1 90)
 (6000 55 1000 2 90) (7000 72 1000 1 90) (7000 57 1000 2 90)
 (8000 71 1000 1 90) (8000 59 1000 2 90) (9000 69 1000 1 90)
 (9000 62 1000 2 90))
```

each element in each list referring to (1) runtime in thousands of a second; (2) pitch (with middle C equaling 60, C-sharp above that being 61, etc.); (3) duration in thousands of a second; (4) channel (1 for the *cantus firmus* and 2 for the newly created line); and finally (5) loudness (0–127). Note that the upper voice appears in channel 1 and the lower voice in channel 2 and that the two voices are spliced together here—they alternate by event—in order to make performance easier. Figure 7 shows the results of Figure 6 in musical notation.

As mentioned earlier, because there often exists more than one correct possible new note, the program can set the same *cantus firmus* in a variety of ways. Figure 8 shows six such settings to the *cantus firmus* of Figure 7. Although each of these settings follows the program's goals slightly differently, the results are quite similar, as will be the case whenever the seed note remains constant for a given *cantus firmus*. However, the slight modifications resulting from different note choices often produce further changes, creating different versions of accompanying lines.

Figure 9 shows a typical set of program runs that

Figure 10. The results of Figure 9 in musical notation.

Figure 11. (a) A dead-ended passage requiring backtracking. The dead end causes Gradus to produce (b) a new passage and (c) a new rule.

Figure 12. Rules are initially created by backtracking, but once created, they plateau in number with backtracking no longer necessary.

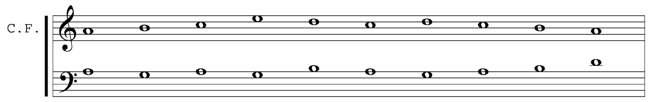
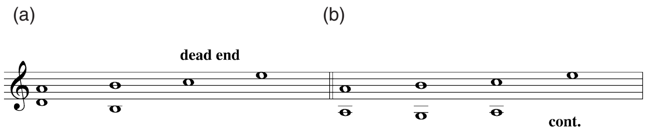


Figure 10



(c)

```
(-4 (1)(-2))
```

Figure 11

```
using default cantus firmus of '(69 71 72 76 74 72 74 72 71 69)


(2 3 4 4 4 5 6 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8)
```

Figure 12

include backtracking. Here the text ''backtracking . . . there are now 3 rules'' clearly delineates the process for the user. Figure 10 shows the results of Figure 9 in musical notation; Figure 11 then illustrates how the dead ends appearing in Figure 9 occur. Figure 11a presents an example of a dead end requiring backtracking. Figures 11b–c show the new rule resulting from the backtracking in musical and numerical notations. In Figure 11a (equivalent to the dead end–creating rule 1 of Figure 9), the music dead ends because each of the four possibilities cause conflicts with one or more of the program's goals. Leaping upward to a D creates both a double leap and dissonance with the C in the *cantus firmus*. Moving upward stepwise to a C creates parallel octaves. Moving downward to an A does not resolve the downward leap in the new line by stepwise opposite motion as required by the program's goals. Leaping downward to G causes the same conflict. Thus Gradus has no option but to backtrack and, in this case, because composition is so near the beginning, start again from a new pitch.

The results of the tests shown in Figure 12 show how rules increase in number by backtracking. However, rule numbers quickly plateau with backtracking no longer necessary. Each of the entries in these lists represents the length of the rules variable (`*rules*`) following one additional run of the Gradus program. In this case, the program was called one hundred successive times without hand-resetting the rules variable; in other words, the rules here accumulate rather than reinitiate with each new function call.

Using two short, transposed-from-one-another *cantus firmi* in multiple runs demonstrates the effectiveness of rules in the learning process and of using diatonic steps in rules rather than actual note names. For example, running the program with the upwards-stepping A-B-C-D-E as *cantus firmus* with middle C as seed note 50 times produces three backtracks and three rules. Running the program then with F-G-A-B-C as a *cantus firmus*, a simple transposition a third down, with A (an interval of a third below middle C) as the seed note, produces no new rules and no backtracking.

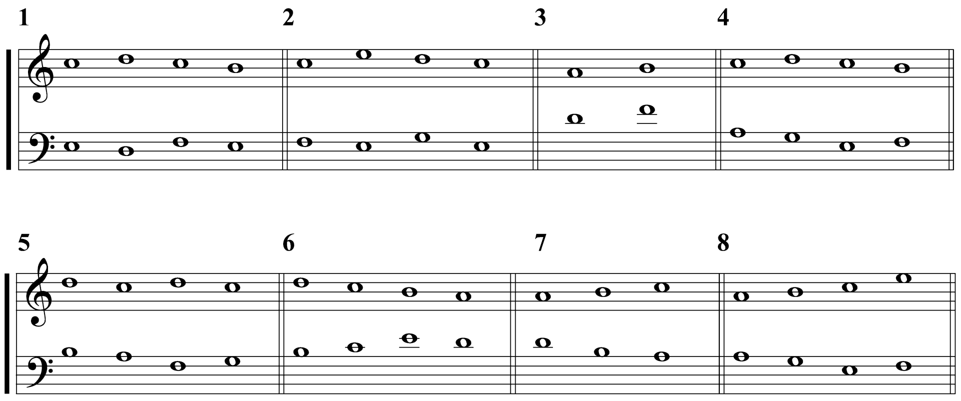
Figure 13 shows the contents of the rules variable in both rule format and in musical notation after the runs shown in Figure 12 were conducted. As can be seen, particularly in their musical versions, each rule is context-specific but at the same time not tied to a particular *cantus firmus*. Rules of fewer than four elements (the norm for rules) result from their occurring at the beginning of a counterpoint. (For example, compare the upper lines of rules 3 and 7 to the upper-line *cantus firmus* opening in Figure 9.) Note that each rule actually ends with the error that caused the backtracking. Note also that Gradus does not reduce the diatonic step-

Figure 13. The contents of the `*rules*` *variable in both (a) rule format and in (b) musical notation.*

(a)

```
((-12 (1 -1 -1) (-1 2 -1)) (-11 (2 -1 -1) (-1 2 -2)) (-4 (1) (2)) (-9
(1 -1 -1) (-1 -2 1))
 (-9 (-1 1 -1) (-1 -2 1)) (-9 (-1 -1 -1) (1 2 -1)) (-4 (1 1) (-2 -1))
(-7 (1 1 2) (-1 -2 1)))
```

(b)



wise vertical distances to within the octave, because occasionally lines separated by an octave will produce correct results while the same lines transposed by octave for closer proximity will not produce correct results (primarily due to voice-crossing problems).

The tests of 50 calls each to the Gradus program shown in Figure 14 immediately followed the runs shown in Figure 12, without clearing the rules variable and therefore using the rules created by different *cantus firmi*. As can be seen here, the first new *cantus firmus* requires only two more rules than the eight already created by the runs in Figure 12. The second run in Figure 14 requires two more new rules as well. The third new *cantus firmus*, however, requires no new rules at all. In essence, the program has ''learned'' how to compose two-voice, one-against-one counterpoint—at least it has learned those rules required to create appropriate accompanying lines to the *cantus firmus* shown here.

I have chosen these particular *cantus firmi* because they resemble one another somewhat and therefore suggest that fewer new rules will be necessary to create new logical counterpoints. Running the program with very different *cantus firmi* may require more or fewer new rules than shown here. In addition, choosing different seed notes—users may override the program's choice—may result in different numbers of new rules from those given here. Sooner or later, however, the program will find all the rules it requires to appropriately accompany almost any goal-correct *cantus firmus* to which the user wishes a second line.

User Interaction

The most frequent user interaction with the Gradus program involves changing the *cantus firmus* by redefining the `*cantus-firmus*` variable. As mentioned earlier, one should not create *cantus firmi* that themselves stray from the goals, because doing so can produce unsolvable problems to which the program may find itself in an infinite loop. As an example, the difference between the *cantus firmus*

```
(72 71 69 67 72 67 71 72 74 76)
```

and

```
(72 71 69 71 69 72 71 72)
```

Figure 14. Using rules created by the cantus firmus *of Figure 12 to help produce a second line to new* cantus firmi, *which then require less backtracking.*

```
using '(69 71 69 72 71 74 72 71 69) added only 2 rules

(9 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

10 10)

using '(69 71 72 76 74 72 71 72 74 72) added only 2 rules
(10 10 10 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 12)

using '(69 71 72 71 72 74 72) added no rules at all
((12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12)
```

is relatively small. However, because the first *cantus firmus* contains more than two motions in the same direction, double leaps, leaps not followed by contrary stepwise motion, and so on, it will cause excess backtracking. Usually, if the problems are few in number, the program will still successfully negotiate the learning process. If the differences are great, however, Gradus may lapse into backtracking loops for which the only escape is to abort the program. This is particularly true for *cantus firmi* that contain repeated notes. To keep the code relatively small and readable, I have not accounted for these anomalies. However, simply keeping the *cantus firmus* within the stated goals of the program nullifies any chance that such problems will occur.

Gradus can create its own *cantus firmus* by simply using one of its own counterpoint solutions transposed up one octave, thus eliminating the need for a user-supplied *cantus firmus*. However, because a provided *cantus firmus* is typically considered a part of the musical puzzle for which a solution is desired, eliminating this user-supplied element seems unnecessary. Ultimately, the program functions only with a set of examples upon which to base its own creations, a situation not unlike that facing human composers. In fact, as I mentioned earlier, Gradus composes counterpoint in much the same way that I, as a composer, compose counterpoint, particularly when attempting to solve problems created by working within explicit constraints.

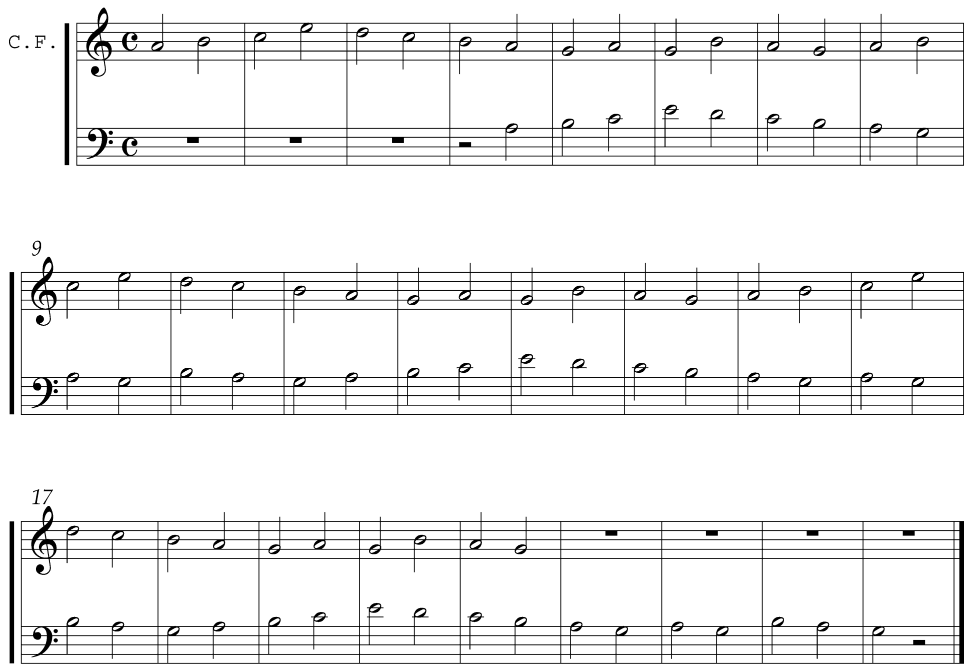
Rules are stored in the rules variable and may be accessed at any time when using the program. Setting the rules variable to nil (see the documentation accompanying the code to Gradus for more information on this and other alterable variables) allows users to reset the program to begin from scratch. A knowledge of Lisp will obviously provide users with many more ways to interact and to follow Gradus. However, even with access to the minimal number of variables given above, users can observe the ''learning'' processes the program uses.

Inference

I describe several forms of musical inference in my book *The Algorithmic Composer* (Cope 2000). Of interest here is my definition of inference as ''an ability to extrapolate basic principles from examples'' (p. 67; see also McCorduck 1979; Anderson 1964; Charniak and McDermott 1985). While centering on set theory as the primary basis for inference in my book, I also discuss a form of tonal inference (see pp. 73–79) using voice leading as a core. Although an extension of the program I describe here could use the same manner of inference as the program I discuss in *The Algorithmic Composer*, Gradus is too small in its current incarnation to incorporate that program's inferential techniques.

Interestingly, however, Gradus uses rule-making and look-ahead processes that, by their very nature, are inferential. For example, although rules 1, 2, 4, 5, 6, 7, and 8 in Figure 13 are context-specific, the look-ahead process will consider them universal. As such, the program will disallow any leap up a third in a new line it composes to a *cantus firmus* that simultaneously contains a downward scale of notes beginning two octaves above (as between notes 2 through 4 of Figure 13b1), because that clearly requires an incorrect diatonic stepwise downward motion to follow it creating parallel fifths. Contextually, when D is determined as a new note of an accompanying line to the *cantus firmus* as in Figure 13b1, an F as a following note is eliminated during the look-ahead process. In essence, the program infers that there are no correct possibilities with the choice of F as a third note (in this instance, as well as all other equivalent instances). Importantly, the program does not then disqualify the D as the second note of the counterpoint, because that note could be correctly followed by the note E.

Figure 15. A simple first-species canon created by five lines of added code.



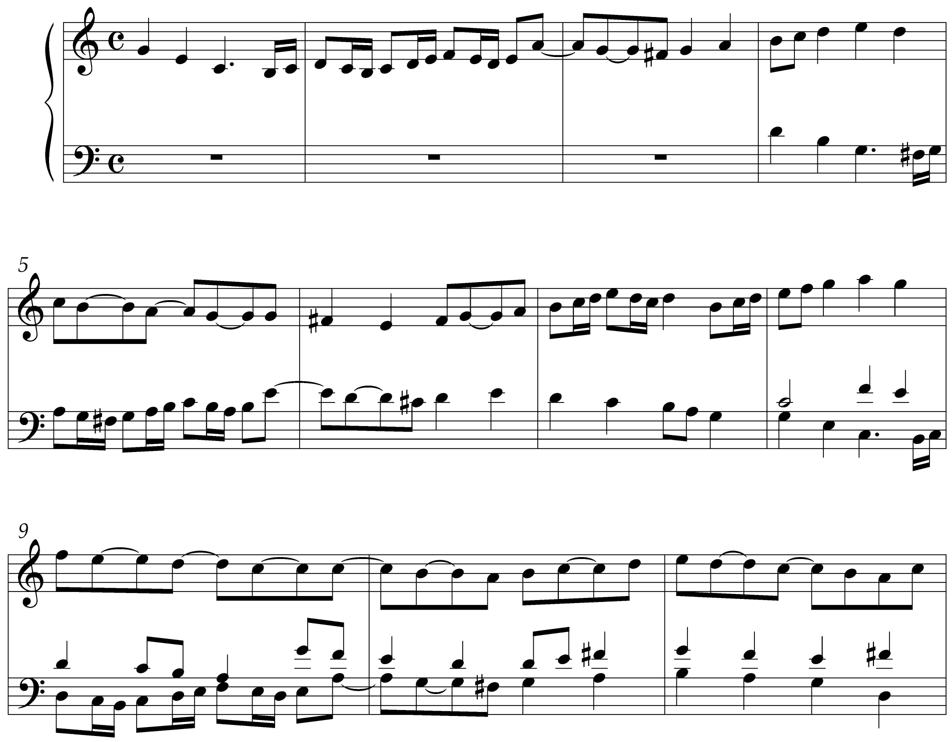
Extensibility

As an example of extensions possible with the Gradus program, I have included with the source code a simple canon-maker function along with a sample *cantus firmus*. Canons, at least canons at the octave as is the case here, require essentially two additional ingredients beyond the basic goals for first-species counterpoint: *invertability* and *offset repetition*. Invertability can be achieved by adding the perfect fifth to the invalid vertical interval list, because perfect fifths invert to invalid perfect fourths during imitation at the octave. Thirds, sixths, and octaves—the only remaining valid intervals—invert to correct intervals when inverted at the octave and thus remain acceptable. Adding the perfect fifth to the invalid vertical interval list creates an added constraint that then makes many *cantus firmi* impossible to accompany with a second line. Therefore, *cantus firmi* must be chosen carefully when creating canons so as not to force dead ends that no amount of backtracking can amend.

Figure 15 presents an example of a first-species two-voice canon created by Gradus's create-canon function. Note that the simple code for producing this canon does not account for the seams between repetitions of the theme. Here, these seams are acceptable. However, using other *cantus firmi* may produce repeated notes or double leaps at the intersections between the theme repetitions. This problem could easily be avoided with the addition of further code that I omitted in an effort to make the transition from simple one-against-one counterpoint to more formalistic canons as clear as possible.

I should note here that this canon program in no way extends the learning capabilities of the Gradus program. The program continues to create and use only the rules that enable it to compose better first-species non-canon counterpoint. However, the above-mentioned problems (encountering impossible-to-accompany *cantus firmi* and poor seams between theme entries) could themselves become learning processes. For example, one could code the seams of canons as part of an extended one-against-one counterpoint—unlike the simplistic manner in which I have strung them together here—with added rules accounting for the desired seamlessness.

Figure 16. The beginning of the first of 48 fugues and the accompanying 48 preludes from The Well-Programmed Clavier.

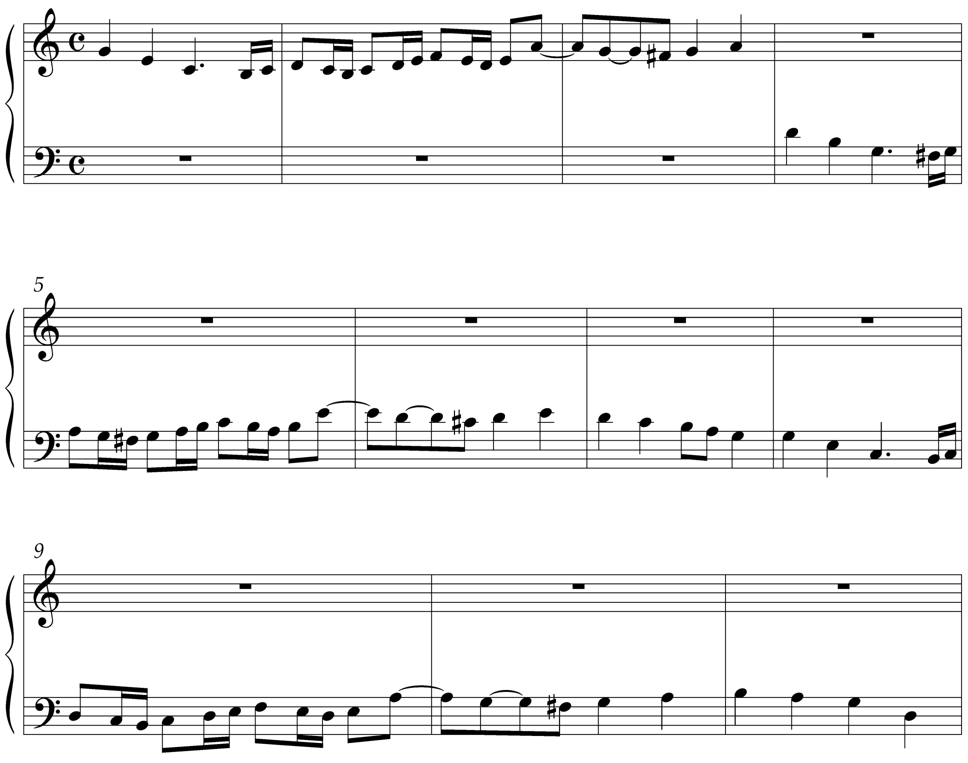


The Gradus program is domain-specific and will currently function only within the boundaries I have defined. However, with minimal programming, the program could be adapted to second-, third-, fourth-, and fifth-species counterpoint as well as to counterpoint in more than two voices. From the resulting four-voice, fifth-species counterpoint code, little change would be required for the program to ''learn'' to compose Bach-like four-voice chorales. Creating more elaborate canons (at intervals other than the octave for example) and fugue-like contrapuntal forms should also be possible by adapting the code. Because most composers use rules of some kind (set theory, serial constructions, and so on) that at least occasionally collide to produce cul-de-sacs, the program would seem a natural fit for a variety of machine-learning musical situations.

As an example, the entire four-voice fugue (of which only the opening three-voice texture appears in Figure 16) was created by a program different from, but not dissimilar to, Gradus. Whereas the basic imitative structure—repetition of the subject at the interval of a fifth, tonal versus real answers, and so on—was not learned (it was built into the code, as was the canon function just described), most of the basic counterpoint resulted from analyzed goals learned by self-created rules. As with the version of Gradus described previously, the program that created this fugue did not contain any code relating to consonance, dissonance, or voice leading, but rather it determined the detail of these goals from the models provided to the program.

The three-voice texture and instrumental counterpoint style presented in Figure 16 require altered versions of the goal types presented previously as well as new goal types added to cover the triadic implications of the thicker texture. In short, the program requires two extensions not discussed thus far: (1) the ability to detect goals in fifth-species (''florid'') counterpoint; and (2) the capability to compose two new voices in a three-voice texture. For the most part, these extensions involve expanding the numbers and types of goal variables so that

Figure 17. The cantus firmus *used to create the fugue shown in Figure 16.*



the program can collect and then follow goals more appropriate to expanded species and textural vocabularies. In a few cases, the creation of this example required the addition of specific code to account for the formalisms inherent in fugal composition (extending existing lines, repeating sequences, etc.), as will be seen shortly.

Figure 17 shows the single-line *cantus firmus* Gradus used to create the music shown in Figure 16. This template contains three iterations of the fugue theme in different transpositions and keys as expected in a fugue (mm. 1, 4, and 8). Built-in instructions require Gradus to begin its fifth-species counterpoint in measure 4 by attaching itself to the end of the subject's first statement. The program also includes instructions for repeating this counter-subject in transposition in measure 8 when the subject appears again in its original guise in a different register. Space limitations do not permit a detailed description in this article of the differences between the creation of the fifth-species three-voice composition here and the creation of the first-species two-voice counterpoint presented earlier. Suffice it to say, however, that the same basic processes—goal determination, rules developed through backtracking, and memory of past mistakes eliminating the need for future backtracking—follow accordingly. This fugue, as well as its allusions (see Cope 2003) to Bach's originals, is the first of 48 fugues and the accompanying 48 preludes from *The Well-Programmed Clavier* (Cope 2002).

To show the flexibility of Gradus and to demonstrate that the program is not limited to traditional concepts of consonance and dissonance nor to historical models of counterpoint alone, I here include three models (see Figure 18) with fifths, seconds, and sevenths acting as consistent vertical intervals instead of fifths, thirds, and sixths. As can be seen in the output in Figure 19, the result of the program's learning new goals, Gradus incorporates the rules of the new models without the need for revising existing code. Although contemporary composers may not use rules so rigorously nor as simply as

Figure 18. Three models that feature open fifths, seconds, and sevenths acting as consistent vertical intervals.

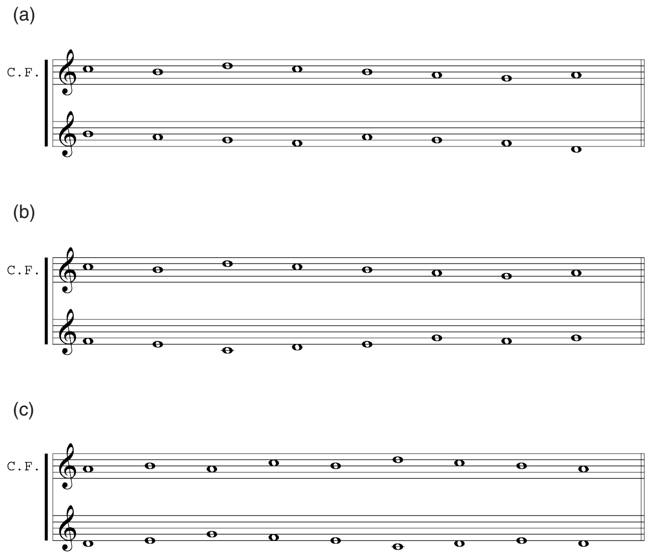
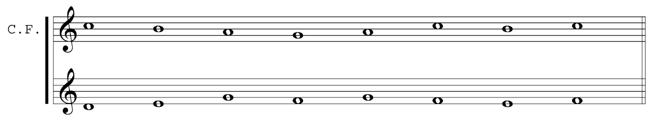


Figure 19. Output from Gradus using the three models of Figure 17.



these, most composers do apply rules of counterpoint in their compositions, whether such formalisms are conscious or not.

The Gradus program and ideas presented here, although simple, indicate a number of interesting if not important conclusions. For example, the notion that machine learning in music is not only possible but attainable using fairly rudimentary techniques can have significant consequences. Such an observation may in itself not seem particularly important were it not for the fact that so little other evidence exists in terms of the published literature. Because a ''learning'' algorithm can be implemented with so few lines of code and with such basic concepts, one can further imagine that extending this code to include more sophisticated musical problems would require less than heroic efforts. Whether such extensions would be simple tools for composing, orchestration, and so on, or composing programs that create new works in their entirety remains to be seen. However, the potential clearly exists within Gradus for a diversity of applications. The Gradus program's current ability to learn and to generalize its discoveries in ways that improve its ability to succeed in its intended purpose, however limited, can lay the foundation for more elegant applications to come.

Because my work with Gradus is a part of my ongoing research with Experiments in Musical Intelligence, a project centered on the discovery and modeling of creativity and intelligence in music, a word about the Gradus program's ability to learn and its potential role as an ''intelligent'' computational composing program seems in order here. In my book *The Algorithmic Composer*, I describe intelligence as based, at least in part, on analysis, association, and adaption. Gradus analyzes models to create its objectives (goals), associates its progress towards those objectives by associating its counterpoint with self-produced rules, and adapts by slowly decreasing its need to backtrack until backtracking is no longer necessary.

I do not feel, however, that Gradus is intelligent. First of all, the program is not ''alive,'' which is one important prerequisite for intelligence I argue for in *The Algorithmic Composer*. Aside from this obvious shortcoming, however, Gradus does not pose its own problems to solve or even initiate the solutions of problems that I pose. The concepts realized in Gradus, however, provide the basis for further work in this area and provide hope that one day we might better understand the elusive concepts of learning, creativity, and intelligence.

Whatever the future might hold for extending the Gradus program presented here, the modeling of these kinds of learning processes in music can, at the least, reveal a great deal about the ways in which humans succeed and fail in their own approaches to musical learning and can provide a means for improving those learning processes. I also believe that, while rudimentary, the basic concepts presented in Gradus can form the foundation for learning in a wide range of other important musical situations than those presented here.

I invite readers to download and use the Gradus program code and see and hear the ways in which the program increases its skills in creating simple counterpoints. The program is quite small, easy to use, and includes a simple guide to aid even those with minimal computer skills in taking advantage of the processes it offers.

References

Anderson, A. R., ed. 1964. *Minds and Machines*. Englewood Cliffs, New Jersey: Prentice-Hall.

Baldi, P. 2001. *Bioinformatics: The Machine Learning Approach*. Cambridge, Massachusetts: MIT Press.

Cesa-Bianchi, N., M. Numao, and R. Reischuk, eds. 2002. *Algorithmic Learning Theory: 13th International Conference*. New York: Springer Verlag.

Charniak, E., and D. McDermott. 1985. *Introduction to Artificial Intelligence*. Reading, Massachusetts: Addison-Wesley.

Cherkassky, V. S. 1998. *Learning from Data: Concepts, Theory, and Methods*. New York: Wiley.

Cope, D. 1996. *Experiments in Musical Intelligence.* Madison, Wisconsin: A-R Editions.

Cope, D. 2000. *The Algorithmic Composer*. Madison, Wisconsin: A-R Editions.

Cope, D. 2002. *The Well-Programmed Clavier* (after J. S. Bach). Paris: Spectrum Press.

Cope, D. 2003. ''Computer Analysis of Musical Allusions.'' *Computer Music Journal* 27(1):11–28.

Costello, R. B., ed. 1991. *Webster's College Dictionary*. New York: Random House.

Cruz-Alcázar, E., and E. Vidal-Ruiz. 1997. ''A Study of Grammatical Inference Algorithms in Automatic Music Composition and Music Style Recognition.'' *Proceedings of the 1997 Workshop on Automata Induction, Grammatical Inference, and Language Acquisition* (n. p., n.d.).

Fux, J. J. 1725. *Gradus ad Parnassum,* trans. A. Mann (*Steps to Parnassus, The Study of Counterpoint*, New York: Norton, 1943).

Hinton, G., and T. J. Sejnowski, eds. 1999. *Unsupervised Learning: Foundations of Neural Computation*. Cambridge, Massachusetts: MIT Press.

Hõrnel, D., and W. Menzel. 1998. ''Learning Musical Structure and Style with Neural Networks.'' *Computer Music Journal* 22(4):44–62.

Jones, D. E. 2000. ''Composer's Assistant for Atonal Counterpoint.'' *Computer Music Journal* 24(4):33–43.

Lewin, D. 1983. ''An Interesting Global Rule for Species Counterpoint,'' *In Theory Only* 6(6):19–44.

Mackintosh, N. 1983. *Conditioning and Associative Learning*. New York: Oxford University Press.

McCorduck, P. 1979. *Machines Who Think*. San Francisco: Morgan Kaufmann.

Michalski, R. 1986. *Machine Learning: An Artificial Intelligence Approach*. San Francisco: Morgan Kaufmann.

Miranda, E. R. 2001. *Composing Music with Computers*. Boston, Massachusetts: Focal Press.

Mitchell, T. 1997. *Machine Learning*. New York: McGraw-Hill.

Schottstaedt, W. 1989. ''Automatic Counterpoint.'' In M. Mathews and J. Pierce, eds. *Current Directions in Computer Music Research*. Cambridge, Massachusetts: MIT Press, pp. 199–214.

Widmer, G. 1992. ''The Importance of Basic Musical Knowledge for Effective Learning.'' In M. Balaban, K. Ebcioglu, and O. Laske, eds. *Understanding Music with AI*. Cambridge, Massachusetts: MIT Press, pp. 490–507.

Widmer, G. 1993. ''Understanding and Learning Musical Expression.'' *Proceedings of the 1993 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 268–275.