

Steven Kelly  
Christ's College

Automatic 4-part Harmonisation

Computer Science Tripos Part II  
1991

# Proforma

Name: Steven Kelly  
College: Christ's  
Project Title: Automatic 4-Part Harmonisation  
Examination: Computer Science Tripos Part II  
Year: 1991  
Number of Words<sup>1</sup>: 11600  
Project Originator: S.M. Kelly  
Project Supervisor: Mr. W.F. Clocksin

## Original Aims:

1. To design, write and test a piece of software, which, when given the bass singer's part, could compose three further parts for the soprano, alto and tenor singers, according to the rules of 4-part harmony.
2. To acquire and demonstrate good functional and modular programming practice, and become proficient in the ANSI C language.
3. To acquire and demonstrate good software engineering practice, and become proficient in using the tools of the UNIX<sup>TM</sup> programming environment.
4. To learn and explore techniques for formalising rules of music harmony as programs.
5. To acquire experience in programming Artificial Intelligence applications.

## Work Completed:

The program performs well, producing good musical output even for difficult input. It performs better than previously published systems.

## Special Difficulties:

None.

---

<sup>1</sup>This is the number of words in the body of the text (i.e. excluding appendices etc.)

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Choice of Project . . . . .	1
1.2	Melody and Harmony . . . . .	1
1.3	Computers and Composition . . . . .	3
<b>2</b>	<b>Description of the Problem</b>	<b>4</b>
2.1	Relevant Areas of Computer Science . . . . .	4
2.2	Breakdown of the Problem . . . . .	5
2.3	The Input Stage . . . . .	5
2.4	The Harmony Stage . . . . .	6
2.5	The Output Stage . . . . .	9
2.6	Areas Concentrated On . . . . .	10
<b>3</b>	<b>Design and Implementation</b>	<b>11</b>
3.1	Programming Environment . . . . .	11
3.2	Modules and Global Decisions . . . . .	13
3.3	Data Structures . . . . .	18
3.4	Tree Search Algorithms . . . . .	23
3.5	Music . . . . .	25
<b>4</b>	<b>Results</b>	<b>31</b>
4.1	Results Expected of a Harmony Student . . . . .	31
4.2	Results Obtained . . . . .	32
4.3	Tunes Produced . . . . .	33
<b>5</b>	<b>Conclusion</b>	<b>43</b>
5.1	Success of the Project . . . . .	43
5.2	Quotes from Musicians on the Harmoniser's Output . . . . .	43
5.3	Further Work . . . . .	44
<b>A</b>	<b>Project Proposal</b>	<b>45</b>
<b>B</b>	<b>Code Samples</b>	<b>49</b>
B.1	<code>input.c</code> . . . . .	49
B.2	<code>pickbest.c</code> . . . . .	52

<b>C</b>	<b>Elementary Music Theory</b>	<b>59</b>
C.1	Rhythm and Pitch . . . . .	59
C.2	Triads and Harmony . . . . .	62
<b>D</b>	<b>Glossary</b>	<b>63</b>

# List of Figures

3.1	Macros to Handle Errors in <code>input.c</code> . . . . .	16
3.2	Data Structure to Represent a Note . . . . .	20
3.3	Data Structure to Represent a Chord . . . . .	20
3.4	Implementation of Relative Scoring Functions . . . . .	22
3.5	Basic Structure of <code>makepes.c</code> . . . . .	25
3.6	Formal Definition of Input Syntax . . . . .	27
3.7	Extract from Start of Input File <code>handel130.in</code> . . . . .	28
3.8	Extract from Output File <code>handel130.out.98</code> . . . . .	28
4.1	Output for Corelli36 — p=98% . . . . .	34
4.2	Output for Handel30 — p=98% . . . . .	35
4.3	Output for Handel226 — p=98% . . . . .	36
4.4	Output for Handel227 — p=98% . . . . .	38
4.5	Output for Mendelssohn293 — p=98% . . . . .	40
4.6	Output for Tune44 — p=0% . . . . .	41
4.7	Output for Tune121 — p=98% . . . . .	42
C.1	Notation for Rhythm and Length . . . . .	60
C.2	Note Names and Intervals in the Major Scale . . . . .	60
C.3	Notation for Pitch, and Key and Time Signatures . . . . .	61

# Chapter 1

## Introduction

This chapter briefly covers the practical aspects of writing good 4-part harmony, from a mainly theoretical point of view. The latter part of the chapter discusses the use of computers to write harmony. Appendix C provides an introduction to music for readers who would like some background knowledge and explanation.

### 1.1 Choice of Project

I have always been interested in writing software that would solve problems with which I had difficulty, and, lacking much practical musical skill, have written several programs that enable the computer to play music I have written. Knowing little music theory, and having less musical intuition, I found that my compositions tended to lack structure, so I decided to try to use the power of the computer to compose, as well as perform, the music. As 4-part harmony has been around for centuries, the style has been analysed in many textbooks, and there are fairly well-defined, if convoluted, rules a composer must follow. The existence of this large corpus outweighed my own taste in music, and I decided to learn about 4-part harmony over the course of the project.

### 1.2 Melody and Harmony

According to the dictionary, *melody* is an “arrangement of single notes in a musically expressive succession”, and *harmony* is a sequence of simultaneous “combinations of notes forming chords”.

#### 1.2.1 Melody

The melody of a tune is normally the most immediately obvious part, often being the uppermost in pitch<sup>1</sup>. Hence, for a tune to be pleasing to the ear, it should have an interesting melody, but also one that does not leap around too much, so the brain

---

<sup>1</sup>The human ear distinguishes variations of pitch more easily in the higher part of its range.

can follow the path of the notes without their becoming lost in the lower parts of the tune.

Because of the peculiarities of the human ear and voice, certain interval jumps are unpleasant or difficult to sing, and certain notes within a key tend to lead on naturally to another note. If the melody does not mainly use the notes in the current key, there is no ‘sense of key’, and the tune seems to wander around vaguely. However, if the melody only uses the notes in the key, the tune can often sound boring. Different composers and different periods tend towards particular kinds of melody, while still paying homage to that which has gone before.

In short, a good melody line is difficult to write. In 4-part harmony, whilst the soprano officially has the melody, the other parts should also have good melodic progression, to give the music its characteristic richness of texture.

## 1.2.2 Harmony

Any combination of notes occurring together is called a *chord*; as with melody, some chords sound pleasant (*concord*s), whilst others are distinctly unpleasant (*discord*s).

The sequence of chords is important too, not only inasmuch as it affects the melody of each part. Given a chord, the choice of the succeeding chord is not free — for instance, to follow a ‘II’ chord with a ‘III’ chord nearly always sounds bad. Certain chord sequences can make it difficult to avoid breaking the rules of harmony — for instance, ‘V’ followed by ‘VI’ in a minor key tends to lead to consecutive octaves, which are regarded as poor harmony.

Fortunately, the figured bass notation frees the program from the choice of chord sequence, forcing it to obey that laid down by the composer. It must still try to obey the other rules of harmony, such as choosing which interval (if any) to double, keeping the separation between the upper parts small, and forbidding two parts to sing the same melody line, if they are separated by octaves or fifths.

For a more natural sound, *inessential* notes should be added, to prevent the monotony of continuous block chords. These come under two main categories, *suspensions*, where a part’s note in one chord is sustained for part of the duration of the next chord, before the essential note for that chord is played, and *passing notes*, where a leap of a third in a part is softened by the inclusion of a note between the chords, so the part progresses stepwise.

## 1.2.3 Figured Bass

Figured bass was widely used as a notation for music in the 17th and 18th centuries. It consists of the bass singer’s part, together with the *figure*, or intervals above the bass note that should be present in the other parts. It therefore always specifies which chord is to be played, and in which inversion, although this information is not immediately obvious. It does not specify which interval, if any, should be doubled, or how the intervals should be assigned to the voices<sup>2</sup>, or in which octave each voice

---

<sup>2</sup>The interval to appear in the melody, relative to the chord root (not the bass note), is occasionally specified.

should sing its note.

The figured bass was written to be an accompaniment of solo music (and string groups). It was intended to be read by a *continuo* player, who would then be able to play along on an organ, giving a fuller accompaniment. The figured bass merely gave the chordal structure of the tune; it was up to the continuo player to use his skill and judgement to play a reasonable accompaniment.

The figures are further complicated by the abbreviations used, which follow no easily-defined pattern. For instance, no figure at all is an abbreviation for the basic triad  $\begin{pmatrix} 5 \\ 3 \end{pmatrix}$ , and any accidental without a number is taken as referring to the third. Musicians differ on whether a figure should be read in numerical order, or from the top down; I will always refer to  $\begin{pmatrix} 6 \\ 4 \end{pmatrix}$  as ‘four-six’. The figure may have from nought to three digits, such as  $\begin{pmatrix} 7 \\ 5 \\ 3 \end{pmatrix}$ . Any of the digits may be preceded by accidentals, (e.g.  $\begin{pmatrix} \flat 6 \\ \sharp 4 \\ \times 2 \end{pmatrix}$ ), which then affect all notes of that interval appearing in the chord above.

## 1.3 Computers and Composition

The idea of using a mechanical means of composition dates back at least as far as Mozart, who reputedly used throws of the dice to determine the structure of one of his works. Most likely, the great composer would have used the dice only to choose one of a number of possibilities he had previously decided could come next; the use of computers to do the actual composition is a much more recent phenomenon. After an early program to generate simple serial music [Gill], virtually the entire output of computer composition programs has been of *atonal* music, which in general is liked by only a small select group, Stockhausen being one of the more famous composer/programmers. In this kind of music, the rôle of the computer is to generate novel music, using its ability to produce random numbers, which will be shaped and directed by the program.

Of the few past attempts to use a computer to compose in an existing style, many have been criticised as producing ‘unmusical’ output. I believe this is largely due to a lack of rules to control the path of the composition, and the lack of sufficiently powerful computers to make full searches of all the possible paths through the tune. Read the comments of two previous computer harmony programmers:

“Towards the end, the harmoniser tended to exceed CPU limits when asked to find harmonies that satisfied stringent stopping rules.”

Donald Bett, on Phoenix (IBM 3084) [Bett].

“In the VAX 11/780 version, it took 15–60 minutes of CPU time. In the present version, it takes 3–30 minutes of IBM 3081 CPU time, although a few chorales have required several hours.”

Kemal Ebcioglu [Kemal].



# Chapter 2

## Description of the Problem

### 2.1 Relevant Areas of Computer Science

Clearly, in requiring a computer to perform one of the more complicated functions of the human brain, the project concerns itself with the area of Artificial Intelligence. To be more specific, large portions of the project are similar to the work that is necessary to make an Expert System:

- There is a large knowledge base of rules;
- There is feedback between the rules, i.e. a previous decision can be overturned on the evidence of a later rule, so the logic is not monotonic;
- The program addresses only a very narrow field of problems, but is expected to perform well on input that falls into this domain;
- The program works best in conjunction with someone who has a good knowledge of its subject area<sup>1</sup>.

Another closely-related area of A.I. is that of game playing. Within a game, certain moves are legal depending on the position reached, and certain moves are advisable, in that they tend to lead to a satisfactory conclusion. The program must, to be successful, look at more than just one move ahead. Again, the logic is not monotonic, and the program must be able to back-track to a previous state as it plays out the game ‘in its head’.

However, there is only one player in the game of harmony, and there is no clearly-defined ‘win’ situation. As no move need ever be played until all subsequent moves have been considered, in theory the program should work out the whole game tree ‘in its head’, and then play its selected sequence of moves all at once. Constraints of memory and speed often prevent this ideal from occurring, and computers can generally solve only small examples of such game trees by exhaustive search.

---

<sup>1</sup>This is not to say that the program is only of use to someone who could perform its tasks himself; it is merely, like any good student, best *able* to show its mettle on being given intelligent input, and having a user who understands its output well.

## 2.2 Breakdown of the Problem

The problem of harmonising a figured bass can be broken down into 3 sub-problems:

**input:** There must be some method of entering the figured bass notation in a way the computer can make sense of.

**harmony:** The computer must be able to generate and test various completions of the figured bass entered, and choose the best one it finds.

**output:** There must be some output from the program that can be read and understood by a human.

The input and output stages are fairly easy problems algorithmically, requiring only the simple iterative manipulation of data, although if only the information given in a standard figured bass is input, the parsing and preprocessing of the input will require quite complex and bulky code.

The harmony stage, however, will be much more complicated, and will have to use some Artificial Intelligence techniques if it is to be successful. It must also contain a large body of rules that the harmony should obey; these will require considerable researching and testing, and it will certainly be difficult to produce code that emulates the often vague and exception-riddled guidelines in harmony manuals.

## 2.3 The Input Stage

### 2.3.1 The Problem

As explained in Section 1.2.3, figured bass is a notation intended to be read quickly by a skilled and experienced harmonist. It therefore contains the absolute minimum of information, and uses many abbreviations, which would be understood by someone with good musical intuition, but follow no clearly-defined pattern. The large number of special cases and context-sensitive information makes writing a conventional parser for the input a difficult exercise.

As the bass line must also be specified, some representation of music must be decided on. This must be useful to the computer and intelligible to the programmer and end-user. Most effort in this area will be concerned with maintaining the accidentals in force at any time, whether from the key signature or from preceding notes in the bar. There is also the problem that much of the arithmetic in music is modular, and thus complex to program and prone to errors.

### 2.3.2 Previous Attempts

In both of the previous attempts at chorale harmonisation, the input was just the melody, and so there was no need to be able to parse and preprocess a figure.

The CHORAL system [Kemal] used a purpose-written C program to read in and preprocess the melody of the chorale.

Donald Bett's Diploma project [Bett] used the syntax of an extant musical type-setting program [PMS] for its input, which was read in by a purpose-written BCPL program. A regular expression matcher from Martin Richard's BCPL course was used to check the syntax of the notes.

## 2.4 The Harmony Stage

### 2.4.1 The Problem

Given a bass note, and a 2-interval figure (e.g.  $\begin{pmatrix} 5 \\ 3 \end{pmatrix}$ ), there are 3 possible intervals to double (root, third or fifth), and then each interval must be assigned to one of the upper three voices, and each voice's note may then be sung in (on average) one of two octaves in that voices range, giving

$$(6 + 3 + 3) * 2^3 = 96$$

possible arrangements of that one chord. Given that an average chorale has about 30 chords, this gives about  $100^{30} = 10^{60}$  possible chords that the program must examine to check every possible sequence of chord completions for a tune. Sadly, at present CPU speeds, even with each chord taking only one cycle to assess on a specialised 'harmony CPU' running at 1MHz, this would take over  $3 * 10^{43}$  years. Even for a small (say 10-note) tune, and better-defined input, the hypothetical 'harmony CPU' would still take nearly three hours, and a realistic estimate for a DECstation would be around 300 years. Clearly the problem we are dealing with is NP-complete, and attempts to solve it by brute computing power will fail.

Apart from the computational complexity of the problem, there is also the area of representing musical rules in code. For instance, even the most basic rule of harmony, that of prohibiting implied consecutive octaves, is expressed thus in a primer on harmony [Morris]:

“...it is better not to approach a fifth or octave by similar motion at all (at any rate between the outer voices) unless one of the two parts concerned, preferably the upper, moves by step.

(The above rule, so far as fifths are concerned, does not apply when moving from one arrangement or position to another of *the same chord*, and consecutive fifths of which one is diminished are permitted between the upper parts, but not between one of the upper parts and the bass.)

An exception is made in the case of triads in root position when the bass falls a fifth and either of the upper parts falls from the third of the first chord to the fifth of the second.”

The language used to write the harmony section of the program can greatly reduce the amount of work to be done. Ideally, the language should be able to

support a database of rules, and back-tracking or, at the least, recursion should be built-in.

For a harmony student<sup>2</sup>, the composition of harmony is a process of finding completions that satisfy fixed rules, or *constraints*, and then selecting one of these completions on the basis of a number of preferences, or *heuristics*, which determine both in what order to test the completions, and how musically satisfying each completion is.

One of the problems facing a programmer is whether any given rule in a harmony manual should be regarded as a production rule which will generate a possible completion, a constraint to exclude forbidden completions, or an heuristic to express a preference for certain completions. If there are too many production rules or constraints, for all but the simplest of tunes the program may exit having found no solutions; if too few, the program may have so many possible completions for each chord that it takes an inordinately long time to run.

In theory, a harmonic style could be represented entirely in constraints, and indeed such an approach has been tried [Baroni]. However, to do this accurately would require that each constraint had a huge number of exceptions, and heuristics appear to be closer to the way a human composer would work.

Research on exhaustive search techniques has produced several different algorithms, each with its proponents in the field, but one appears to have stood the test of time. This process, called the *generate-and-test* paradigm [Stefik], can be modelled by a computer:

- Make all the possible completions for the current chord that satisfy the constraints;
- Score each completion according to the heuristics;
- Sort the list of completions by score;
- Write the best-scoring completion into the tune;
- Recurse onto the next note; if we encounter a dead-end and back-track to here, move on to the next best completion.

Another approach is to use *simulated annealing*, which takes a completion of the whole tune, scores it, and then changes a few notes and rescores the new completion. Initially, the search has a coarse grain, specified by the larger number of notes to change, and will be more likely to accept a solution with a lower score. As more completions are tried, both the number of notes changed at each completion and the probability of accepting a sub-optimal completion are reduced; this gives a finer grain of search.

---

<sup>2</sup>As distinct from an experienced composer, who can use his well-developed intuition to find completions that satisfy a broad range of heuristics, and can tell when it is better to break a rule.

The coarseness of the search is analogous to the temperature a metal is raised to in annealing<sup>3</sup>. By modelling the process by which a metal is annealed into a stable form, simulated annealing attempts to ‘home in’ on a good solution.

If the finished output is to sound less artificial, it must include inessential notes between chords, as well as the essential notes in the chords. These are added by a human composer generally after an otherwise complete tune is composed. Although it would be possible to add inessential notes whilst completing the main part of the tune, the temptation then is to make completions that are locally good because they include inessential notes, but globally bad, as the harmony of the piece as a whole suffers. From a computational point of view, it is better to keep as much work as possible out of the main part of the harmonising, as this makes the most effective use of resources.

## 2.4.2 Previous Attempts

The CHORAL harmoniser [Kemal] was written in BSL [BSL], a language designed specifically to implement CHORAL. BSL is a first order logic programming language, which naturally implements back-tracking, and is compiled into C code in an architecture-specific way. It can implement the formal analogue of a knowledge-based system based on the generate-and-test paradigm. BSL, unlike other logic-programming languages, has the concept of heuristics built in, and so exhaustive search algorithms are simple to write.

Over the course of a PhD., and an extra research year, the CHORAL program accumulated 350 production rules, constraints and heuristics, based mainly on empirical observation and personal intuition, and also traditional harmony manuals. The size of this knowledge base, coupled with the benefits of an application- and architecture-specific language running as sole process on a large mainframe, enabled the program to “seem to work, and produce musical results” [Kemal, p174].

Donald Bett’s Diploma project was written in BCPL. The basic algorithm used was simulated annealing, although this seems to have been a late addition, and the author says it was “never run long enough to have a beneficial effect on the final result.” The final output seems to have retained too much of the random element he used to make completions of chords to score, and despite his initial statement that “the harmonies were effective with the exception of the odd dubious chord,” he later admits in his conclusion that:

“The harmoniser as it stands is still relatively unconstrained; it tends to be more radical than conservative in its approach. . . Its choice of keys often tends to be novel.”

---

<sup>3</sup>A process whereby metals are strengthened by repeated heating and cooling, with the maximum temperature reached lowered each time.

## 2.5 The Output Stage

### 2.5.1 The Problem

To produce a full musical score as output would involve large amounts of extra work, most of which would be more concerned with music than with computer science. There are many good scoring programs available, and an end user would probably like the ability to format the output as he saw fit. Some lateral thinking raises the possibility of producing output in audio form, by directly linking the computer to a synthesiser *via* MIDI<sup>4</sup>. This, however, is not possible under University regulations, being more appropriate for a music project than one in computer science.

So, a purely textual output format for the project would be sensible. It should be possible to follow the chordal structure and individual parts reasonably easily, to facilitate debugging and testing without having to transcribe the text into a score. Musical conventions should be followed as much as possible, to facilitate transcription, and to enable users familiar with normal score format to understand the output as easily as possible. This requires a surprisingly large amount of work, to keep track of the bars, beats within bars, key signature, accidentals current in a bar etc. The output should contain enough information that separate manuals for its decoding are unnecessary.

### 2.5.2 Previous Attempts

The CHORAL system [Kemal] produced its output as a rudimentary music score, using purpose-written graphics routines written in C. The output could be considered unsatisfactory, however, as it was all in the key of C, irrespective of the key of the tune. Further, the music as displayed would not necessarily be within the range of the voices for which it was intended; the program only ensured that there existed some transposition of the tune which would bring all the parts within the range of the voices. This means that to hear the music, the score would need to be transcribed in the correct key, thus removing many of the advantages of having a score as output.

Donald Bett's Diploma project [Bett] sensibly re-used the PMS syntax [PMS] from the input stage for its output. Although this meant that the output was almost meaningless to one unfamiliar with PMS syntax, it allowed a reasonably easy method of transcription into score format. A small problem with this approach is that it leaves the program too dependent on PMS, and especially vulnerable to changes in that system's syntax.

---

<sup>4</sup>Musical Instrument Digital Interface — the industry standard for communication of music as data.

## 2.6 Areas Concentrated On

In this project, the following areas have been concentrated on; some because of the time constraints, some because of a desire to produce music that is aesthetically pleasing, some because of a desire to produce a project that would please the examiners (!), and some just to prove it could be done:

- The avoidance of technically bad harmony;
- The production of technically good melodies for each part;
- The selection of reasonable rules for the above, and good output as judged by those rules;
- A nominally Bachian style, with inessential notes, to pay homage to the master of harmony<sup>5</sup>;
- The use of true musical principles throughout, even where a much easier solution comes close to working, to facilitate later development and maintenance;
- A truly modular design, to keep input and output formats distinct from the work of the harmoniser;
- A truly functional design, to facilitate the addition of new scoring functions, and the testing of different algorithms;
- The ability to deal with tunes of reasonable size (i.e. at least 8 bars, the average size of a chorale) in a reasonable amount of time;
- The investigation of algorithms to deal with NP-complete problems.

An area it was felt best to ignore for the most part was that of rhythm, the effect of this being left more in the hands of the composer and performer, as is the tradition in music, and especially in figured bass.

---

<sup>5</sup>This was not carried through to the extent of allowing those harmonies of Bach that ‘broke rules’ — only a genius can do that successfully.

# Chapter 3

## Design and Implementation

This chapter concentrates on the actual work done over the year, and shows the growth of the project through design decisions and implementation of the modules, functions and data structures. Unless otherwise stated, I wrote all the code, and designed all the algorithms and data structures in the course of the project.

### 3.1 Programming Environment

#### 3.1.1 Policy Decisions

The project was originally intended to be written as a composite of C and Prolog code, using C for the input and output modules, and Prolog for the harmony module. The built-in backtracking and logic programming of Prolog would be most useful at the harmony stage, but its facilities for input and output are severely limited. After consultation with my project supervisor, Mr. W. F. Clocksin, it was decided that the support for Prolog on both Phoenix and the DECstations was insufficient for my project.

As the ANSI standard for the C language was now current, and adopted by most C programmers, it seemed sensible to use it for my project. This also enabled work to be done on my Amstrad PC, using Microsoft C version 5.1, which is ANSI-compatible. Much experience was gained in writing portable code by this approach, despite initial difficulties. The DECstations' architecture-specific C compiler, `cc`, is not ANSI compatible, which would have been a disappointing setback were it not for `gcc`. Although both compilers claim to conform to the ANSI standard, neither supported initialisation of arrays properly, and the ANSI standard itself is, in my view, flawed by its omission to specify the results of modular arithmetic on negative numbers.

With a project which would now be written entirely in C, the natural choice for a programming environment was UNIX<sup>TM</sup>. In the UNIX<sup>TM</sup> environment on the DECstations, there are many useful facilities for the C programmer:

**emacs:** An advanced multi-purpose full screen editor, which is easy to customise, fast and powerful. It has the ability to run other utilities from within the



editing session, using the same user interface. Many of the utilities below have custom interfaces with **gnuemacs**, the local version of **emacs**.

**rcs**: A revision control system, which automates the storing, retrieval, logging, identification, and merging of revisions of a document: in this case C source code. It maintains a history of all the code written, useful for backups, propagating and undoing changes, and tracing bugs.

**gdb**: A powerful source level debugger, which runs under **gnuemacs**. It allows the programmer to step through his code and see the contents of the data structures and the flow of control through the program. It is thus an invaluable aid in tracing bugs.

**gcc**: An ANSI compatible optimising C compiler, whose output can be interpreted by **gdb** and **prof**.

**prof**: A profiler, which shows the percentage of time spent in each function during a run of the program, and maintains counts of how many times a function is executed. Can also work at the finer granularity of ‘basic blocks’.

**nohup**, **nice**: These enable lengthy program runs to be done on a remote host, without the programmer having to be logged on.

There are also many other facilities, such as **time**, which measures how long a program takes to execute; **lint**, which performs careful checks on C source code for possible bugs, although the present version is not compatible with ANSI C; and various text processing commands which help quickly identify any changes in output caused by a change in the code.

### 3.1.2 Implementation

Initially, I had little experience of programming in C, and had used none of the programmers’ facilities provided by UNIX<sup>TM</sup>. I therefore spent some time at the start of the year familiarising myself with both the language and facilities, by writing small C programs to perform fairly simple text processing tasks. These were artificially made more useful by using all the facilities such as **rcs**, **prof**, **gdb**, modules and include files, despite the lack of need for their use. In this way, much useful experience was gained which saved time later on.

At first, large parts of the programming were done on my PC, as the DECstations were often too busy. However, the unreliable file transfer facilities between the PWF and DECstations eventually put a stop to this. I intended to be able to complete the harmony section at home over the Easter vacation, but secured permission to come back up two weeks early, in case I found working at home too distracting. This measure of protection proved its worth, as my PC broke the day I got it home!

## 3.2 Modules and Global Decisions

### 3.2.1 Policy Decisions

#### ‘Black Box’ Approach

Following the initial decision to split the code into 3 modules, I spent some time designing how these would interact, trying to make each module as independent of the others as possible. By having a formal interface definition, and keeping the module interfaces small and neat, it would be possible to substitute different input and output formats easily, without having to rewrite the main bulk of the harmony module.

To facilitate this ‘black box’ approach, the input to the harmony module would actually have to contain more information than that which could be simply divined from a textual figured bass input format. For instance, a figured bass chord does not immediately specify which inversion the chord is in, whereas other sensible input formats might. A common alternative to figured bass uses Roman numerals to represent the root of the chord, and a letter to indicate which inversion, so  $\text{V}^6$  below a ‘B’ in the key of C major is equivalent to Va, both representing the first inversion of the triad on the dominant, ‘G’. To write a new input module for the second input format would require functions to calculate the intervals in the triad relative to the bass note, as that information is present in the figured bass, but not immediately obvious from the second format.

In a similar way, as there are many possible input formats, there are also many possible algorithms to search a tree of scores for the best path from the root to the leaves. I therefore designed the algorithm to search the tree to be independent of the functions that produced the scores, and of the input and output formats. The algorithm could thus easily be replaced if it proved to be inefficient or incapable of searching the huge possibility space of tunes, as seemed likely given the complexity of the problem.

The scoring functions were also designed to present a consistent interface to the tree search algorithm, so new scoring functions could be added without having to change the code in the algorithm.

To support the links between modules, and allow the use of global variables, macros and compile time variables, I decided to use include files, to avoid having to write the same code at the start of each module, and change all modules if part of this global information changed. As C allows nested inclusion of text files through its `#include` preprocessor directive, a single include file at the top of each module was envisaged.

#### Error Handling

As the input was to be a text file typed in by the user, it was necessary to check it for errors, and report these clearly so the input file could be corrected. Unless the program could itself recover from an error and be sure it had corrected it properly, any error should cause the program to abort before entering the harmony stage,

rather than probably crashing untidily at a later stage. Some errors, such as being unable to find the input file, should clearly abort from the input stage, whereas others, such as an octave being entered as a letter rather than a number, should merely cause the user to be notified and allow the rest of the input to be processed, so all the errors in the input can be found in one run of the program.

Provision was made in the structure of the error handling for the program to make substantial attempts at recovering from errors on its own, although I thought it unlikely that there would be time to implement this feature.

## Debugging Statements

Whilst writing and testing code, it is always useful to have the code print out messages to the screen when it reaches certain major points of the execution, and also to print out the values of important variables. These functions are implemented by inserting ‘debugging statements’, whose effect is removed in the final version of the program. The information they provide can help identify simple bugs quickly, without requiring the use of a source-level debugger, and they are often better at showing how the values in a data structure changed over the course of the program.

I decided to try to implement these debugging statements without littering my code with preprocessor conditional-inclusion directives, which spoil the appearance of code, thus making it *harder* to debug. I also wanted the debugging output to be controlled by a command line option, and the debugging code to be completely removed by an argument to the compiler, so it would not affect the speed of the optimised code.

## Command Line Arguments

As the code was intended to be useful to a musician, and not just to a programmer, it was necessary that it be configurable from the command line. I hoped that I could implement this even to the extent of selecting which scoring functions should be called. In keeping with the standard UNIX<sup>TM</sup> philosophy, I decided that the command should produce no output to the shell for a successful run (unless debugging was turned on), and that the syntax of the command line arguments should be a ‘-’ sign, followed by a letter to specify which argument, possibly followed by a value for that argument.

### 3.2.2 Implementation

#### ‘Black Box’ Approach

In the end, the program was split into 6 modules, and 3 include files:

`harmony.c` was originally written as a simple test harness to call the various other modules’ functions in order. The testing code was later removed, and it became the command line argument interpreter, setting up global variables and calling the other main functions.

**input.c** takes the textual input format of the figured bass, and adds in other information such as the chord root and inversion. The code is quite complex and bulky, as it has to process an irregular input into a regular data structure. It also must contain a large framework of functions to deal with music, such as **str\_to\_acc**, which converts a string containing an accidental to a numeric value for that accidental, depending on what degree in what key the note is, and what other accidentals have already appeared in the bar. These accidentals can have been specified in the bass line, the figure, or the melody information. Some of its functions are used by other modules, but not enough to merit their collation into a separate module.

**makepes.c** takes the data structure filled in by **input.c** and makes and stores all the possible completions for each chord. This removes this work from the bottleneck of the recursive algorithm. Each possible chord completion is a 'possible event', and this is shortened to **pe** in the code, hence the module's name.

**pickbest.c** scores each **pe** absolutely, and then searches the tree of completions, scoring each **pe** relatively again each time it is encountered. When the final path has been chosen, the code runs through this path, adding inessential notes.

**output.c** runs through the path produced by **pickbest.c**, printing the textual output format of the chords to a file, along with a key to the format and other useful information.

**allfiles.c** contains the definitions of the global variables. **allfiles.h** contains the **extern** declarations of these, **preproc.h** contains compile time variables, and **note\_adt.h** contains the type definitions.

The time spent in designing the modules' interfaces to be narrow was worthwhile; there are few arguments passed between modules, the only overlap being global data structures such as **tune**, which contains all the chords and their possible completions, and is used in every module. The 'black box' approach has resulted in comprehensible, re-usable code; for instance, it would be simple to make the code into a tutor, which could mark a student's attempt at harmony and point out the faults, suggesting better alternatives.

## Error Handling

**input.c** is the only module which has to have a secure error recovery structure; errors in the other modules indicate mistakes in the code. Some time was spent designing a consistent system of response to errors in the input. These can be discovered at a depth of up to 4 function calls from the main **input** function, and may be discovered in a generic function, e.g. **str\_to\_num**.

There are three basic responses to errors that could be used:

```
#define raise_error(a) return (a)
#define cause_error(a) {fprintf (stderr, "Error "#a" occurred. Exiting.\n");\
                        exit (a);}
```

Figure 3.1: Macros to Handle Errors in `input.c`

1. Do not check for errors in the input at all, and thus allow the program to crash later.
2. On an error being found, immediately call an error-handling function, which prints an error message and exits from the program.
3. Report all errors to the calling function, and allow it to deal with them.

Approach (1) leaves it up to the user to enter correct input; this is clearly insufficient, as the user has no way of telling which line caused the error, or if a bug, rather than incorrect input, caused the program to crash.

Approach (2) can leave the user having to run the program on essentially the same input several times, correcting one error after each run. Also, an error in a generic function can only ever report the name of that function; the actual error may have occurred in any of the functions that call this generic function. It is, however, simple to implement, although it does give functions multiple exit points.

Approach (3) provides a consistent system of error passing, similar to that of raising exceptions, or of ‘aborts’ in a database system. It can provide a complete back trace of the functions called before the error. Further, it can be implemented without multiple exit points, and allows for error recovery as well as detection.

For these reasons, approach (3) was adopted, and implemented using macros, as seen in Figure 3.1. These allowed the insertion of extra code into the error-raising path, should this ever be desirable, and prevented having multiple hard-wired **return** statements or deeply-nested **if** statements in a function. Their implementation as macros allowed for their use in functions returning any type. The arguments to the macros were always compile time variables, and the top level error-handling function, `in.error`, searched the source file for the values of these constants, and thus determined the name of the variable for the error report.

## Debugging Statements

The ideas I had for debugging code were difficult to implement fully. One of my early thoughts was to make each debugging statement a call to a function which would contain preprocessor directives and also an **if** statement which would check a variable set up by the command line. This was abandoned, however, as it would not be optimised out; the function call would remain and take up time, despite it only being an empty function.

The final solution was to implement the debugging statements as macros, which were defined as empty if a ‘`NODEBUG`’ option was passed to the compiler by `make`. The normal expansion of the macros was to a **printf** statement conditional on a

variable set from the command line. This solution required different macros for debugging statements with different numbers of arguments; not a severe problem, as the maximum number of arguments used was 2.

These macros provided everything I needed from debugging statements in a simple and effective way; I shall certainly be using them in any C code I should write in the future.

```
#ifndef NODEBUG
    #define dbprint(a)
    #define dbprint1(a,b)
#else
    #define dbprint(a)    if (debug_print) printf (a)
    #define dbprint1(a,b) if (debug_print) printf (a,b)
#endif
```

## Command Line Arguments

The final set of command line arguments gave the user a thorough control of the actions of the harmoniser. The arguments are listed below; throughout,  $n$  represents a Boolean value, ‘0’ being off and ‘1’ being on.

- i** $\langle file\_name \rangle$ : This selected the file from which the input should be taken. If this argument was unspecified, input was taken from **data.txt**.
- o** $\langle file\_name \rangle$ : This selected the file into which the output should be sent. If this argument was unspecified, output was sent to **op.txt**.
- d** $[n]$ : This turned the debugging output on for non-optimised code. The default state was off, and if  $n$  was unspecified debugging was turned on.
- t** $\langle score \rangle$ : This allowed the user to specify an acceptable score for a tune. If this score was reached or beaten, the program would immediately stop the tree search and output the tune.
- p** $\langle real\_num \rangle$ : This allowed the user to select how much of the search tree should be pruned.  $real\_num$  is a decimal value between 0 and 1, 0 causing no pruning, and 1 causing only the best event to be selected at each level.
- s** $\langle rand\_seed \rangle$ : The selection of tunes with equal scores, and the placing of accented passing notes were controlled by the **rand** pseudo-random number function. This argument allowed the user to specify the seed (a whole number) for that function.
- x** $[n]$ : This controlled the insertion of inessential notes. The default state was on, and if  $n$  was unspecified insertion was turned off.
- a** $n_1 n_2 n_3 \dots$ : This allowed the user to select which absolute scoring functions should be used. Each  $n_i$  is a Boolean value (‘0’ or ‘1’) specifying whether to call the corresponding function. If the list of  $n_i$ s was shorter than the list of the functions, all remaining functions would be called.

`-rn1n2n3...`: As `-a` above, but for the relative scoring functions.

## 3.3 Data Structures

### 3.3.1 Design Decisions

The motivating forces behind the choice of data structures were:

- They should provide easy access to the information required by the harmoniser, even at the expense of redundancy. This may mean it takes longer to fill them in, but it should speed up the harmony section, which mostly reads already-stored information.
- They should embody established programming standards.
- They should be portable between different systems.
- They should allow multiple viewpoints, or knowledge models. For instance, the data structure for a note should contain not only the absolute pitch of that note, but also its degree in the scale, and any pertinent accidentals.
- They should be structured so that argument passing is simple to understand; i.e. a structure should not be passed to a function if only one part of it is accessed by the function.
- `typedef` and `struct` should be used to make the code more readable, and reduce the impact of changes to the implementation of the structures.
- Data structures in different modules should all conform to the same standards, e.g. if a ‘b’ is represented by setting the top bit in one data structure, it should be represented thus wherever it is bit-encoded.

Multiple viewpoints of data are often required in complex expert systems. The Hearsay-II speech understanding system [HSay] is a good example: there was a need to observe the interpretation of speech simultaneously as mutually-consistent streams of syllables, words and phrases. The CHORAL harmoniser used multiple viewpoints to good effect, having 6 viewpoints of a tune<sup>1</sup>.

Several data structures, such as the tune and those containing accidentals current in a bar, had to be global. I was careful to design these to be written and referenced in the same way in all the modules. In maintaining this consistency, I aimed to make the code clearer and hence easier to debug.

---

<sup>1</sup>Chord sequence, inessential note sequence, melody of each part, ditto with repeated pitches merged, time-quantised, and Schenkerian voice leading [Schenk].

### 3.3.2 Implementation

It was originally hoped that the main structures could be implemented as Abstract Data Types; the module `note_adt.h` was to have included the source code of the functions to deal with notes, chords and accidentals. Sadly, this was prevented by the poor support of C for ADTs, and the complexity of representing music in a data structure. However, the basic philosophy behind ADTs, i.e. abstraction, was followed wherever possible.

In general, data types were constructed so the minimum copying of information took place; pointers were used to the existing structures except where recursion prevented this.

Several types were bit-encoded, especially the representation of an interval. This allowed quick and easy access, whilst keeping the storage required down (important for the multiple instances created in recursion). Access to the fields in bit-encoded variables was by macros, which allowed the bit twiddling to be hidden from the programmer, as in an ADT.

#### CHORD and NOTE Types, and Viewpoints

The two main data types were `CHORD`, which contained all the information about a filled-in chord, and `NOTE`, which contained all the information about a given note in a chord. The definitions of these data structures can be seen in Figures 3.2 and 3.3. Together, they allowed the following viewpoints:

**Time-sliced:** where the position of each chord in time was visible;

**Chord structure:** where the tune was seen as a sequence of chords, each specified by its root and inversion;

**Raw notes:** where the absolute pitches of the notes were visible (useful for MIDI output and prevention of consecutives);

**Melodies:** where the movement of each part within the key was visible;

**Stacking:** where the relation of the notes within a chord was visible (for spacing, overlap and doubling considerations);

**Crossing:** where the paths of adjacent voices in adjacent chords were visible (to prevent parts crossing);

**Inessential:** where the distribution of inessential notes about the tune was visible (to prevent unharmonic clashes).

#### Global Data Structures

The global data structures that maintained the key, the accidentals current in a bar, and the bar and beat position within the tune were, as far as was possible,



```

typedef struct {
    char abs_pitch;    /* 0=C 3 octaves below middle C */
    char interval;     /* 0-7, from bass note (as in figure) */
    signed char modifier; /* semitones from 'white note' */
    char note;         /* value of note - A->0 etc. as output of char_to_note:*/
                        /* enables recovery of note name for output stage.    */
}
NOTE;

```

Figure 3.2: Data Structure to Represent a Note

```

typedef struct{
    char bar;          /* Number of current bar */
    char beat;         /* beat (actually clock pulse) within that bar */
    char bpbar;        /* beats per bar (so we can update time sig on output) */
    char key;          /* bit-encoded key */
    char length;       /* in clock pulses (i.e. as multiple of shortest note) */
    char chord_root;   /* relative to key, 1 - 7, or 'r' for rest */
    char figure[3];    /* encoded intervals */
    char melody;       /* encoded interval */
    char inversion;    /* Which inversion of chord */
    char doubled_int;  /* Which interval was doubled, or BAD_INTERVAL if none */
    char num_pes;      /* Number of pes for this event initially */
    union {
        char suspension; /* Bits 0-3 (vals 1-8) = voice numbers */
        char passing_note; /* Bits 4-7 (vals 16-128) = voice numbers */
    } aux;
    int abs_part_score; /* Absolute part of score of chosen pe */
    int rel_part_score; /* Relative part of score of chosen pe */
    PEP poss_event_list; /* linked list of all possible events (pes) */
    NOTE parts[4];
}
CHORD;

```

Figure 3.3: Data Structure to Represent a Chord

kept along with the chord to which they referred. This meant that the minimum of information would have to be updated when referencing the chords used in the tree search algorithm, once again removing work from the computation-intensive part of the program. The only exceptions were the accidentals current in the bar, which were stored as a global array. The reason for this is obscure, and relates to the differences between conventions in displaying accidentals in 4-part harmony. Suffice it to say that storing the current accidentals in the **CHORD** data structure would have meant large changes to the body of the code if the output format had been changed; this would be counter to the ‘black box’ modular principles employed elsewhere.

## Accidentals

I encountered significant difficulties in designing a data structure to maintain the accidentals current within a bar, and in writing the code to maintain this data structure. This was due partly to a misapprehension on my part when attempting to formalise the syntax of accidentals, and the rules by which they change, and partly to my insistence that true musical principles should be employed (see Section 2.6). The simple option would have been to make the input format specify accidentals for all notes, and have the output display a maximum of one accidental per note — e.g. ‘Fx’ (F double sharp) would have been output as ‘G’. Whilst this would have left the output correct in terms of the actual pitches of the notes, the degrees of the notes within in the key would often be wrong. This would be a severe notational inconvenience to a musician, in a program designed to be useful to musicians.

## Rests

There were also problems in deciding how to represent a rest. Again, the simple option was just to ignore the rest and make the preceding note longer. This would cause problems, as no harmony function could then be written which would take into account the change a rest makes to the harmonic structure of a piece. Rests were initially implemented as directives (**#rest**) in the input, which simply incremented the bar and beat counters. I later decided that a rest looking like a note of no pitch would make the input file look neater, as it could be split into bars whose notes’ lengths would sum to the correct value. Unfortunately, this was going to make the code for the tree search complicated, as it would have to skip over the rest when looking back at previous chords. This would have been bearable under the previously-envisaged system of passing pointers to preceding chords to the next instance of the algorithm, but was virtually impossible when this approach was abandoned for considerations of speed. Finally, therefore, rests were implemented as a directive which incremented the bar and beat counters. These values were stored in the **CHORD** data structures, and so the position and length of the rest could be recovered at the output stage.

```

typedef int RELFUN (char); /* relative scoring function */
RELFUN leaps, consecutives, tedo, crossing, same;
RELFUN *rel_score_funs[10] = {leaps, consecutives, tedo, crossing, same, NULL};

/* Accumulate all relative scoring functions' scores for this event */
for (curr_fun = rel_score_funs[0], fun_num=0;
     curr_fun != NULL;
     curr_fun = rel_score_funs[++fun_num]) {

    tune_path[curr_event]->rel_score += (*curr_fun)(curr_event);
}

```

Figure 3.4: Implementation of Relative Scoring Functions

## Relative Scoring Functions

The relative scoring functions were all designed to have the same type, so pointers to them could be placed in an array, and then called by a loop. This also allowed command line arguments to select which functions should be called, without multiple `if` statements. The benefits of this decision were proved again and again at the debugging stage. Figure 3.4 shows the code used to calculate all the scores of the relative scoring functions for a given `pe`. To add a new scoring function, all that is necessary is to include it in the two lists of such functions. The command line argument `-r11011` would disable the third function, namely `tedo`, so it does not contribute to the relative score for that event.

My initial vague ideas of passing the past few events chosen as arguments to `rel_score` quickly proved too inflexible, as any new scoring function which needed to look further back in the tune than was currently allowed required changing all the function definitions. Instead, the type of `rel_score` became:

```
char rel_score (char curr_event, int score_so_far);
```

so each instance of `rel_score` had an index, `curr_event` into the array of completions made in the best overall tune found so far, and into the array of completions made by preceding instances, and the cumulative score of those completions. Each relative scoring function could thus look as far back to the beginning of the tune as necessary for that function, and no time or memory was wasted passing a list of pointers to previous completions. The variables containing the best tune and the current tune were visible throughout the whole module, and had types<sup>2</sup>:

```

struct {PEP poss_event_pointer;          /* Pointer to the best completion */
        int rel_score; /* The relative score this completion had when it */
                          /* was considered with others in best_tune_path. */
        } best_tune_path [MAX_EVENT];

struct {PEP poss_event_pointer; /* Pointer to the currently chosen pe */
        } tune_path [MAX_EVENT];

```

---

<sup>2</sup>PEP is a pointer to a `pe`.

## 3.4 Tree Search Algorithms

### 3.4.1 Design Decisions

It is possible to produce a harmonised tune working purely iteratively down the tree of possible completions of each chord. At each stage, that chord is chosen which scores highest absolutely and relatively to the preceding chords. This approach suffers from the serious fault that a locally good choice of chord may prove to be globally bad; it may force the path of the tune down a musical dead end, where every possible completion breaks some major rule of harmony. Even should this not happen, the probability of the tune produced being the best path through the tree is tiny. Although it is possible to introduce some measure of back-tracking into an iterative algorithm, such an addition shows a poor design; it would have been better to write a recursive algorithm in the first place.

On a naïve level, each call of the harmony section would generate all possible completions of the current chord, score them absolutely (on the relations of the notes of the chord to each other) and relatively (on the relations of each note of the chord to the same voice's note in the previous chords, and the relations of those chords as whole entities), and then recurse on each completion to the next chord in the tune.

Each chord can be completed in a number of different ways, but the choice of `pes` available is only dependent on the information present in the input, and not on the path the tune may have taken so far. To make all the possible events for a chord each time that event was encountered would be to repeat computations needlessly. I therefore wrote `makepes.c` to create all the possible events before and outside the recursive tree search algorithm (see Figure 3.5). The absolute tests of a chord also rely only on the notes within that one chord, so I decided to complete all these tests before the tree search algorithm, to save repeating them every time a chord was produced. In these and other ways, the code for the recursive function was made as short and efficient as possible, enabling the program to search the greatest possible number of paths through the tune.

With the massive tree (about  $10^{60}$  nodes — see Section 2.4.1) to search, there was a large possibility that any recursive approach based on an exhaustive search would fail, even if the tree was substantially pruned at each node. I therefore spent some time calculating the complexity of various approaches, the effects of adding more scoring functions, and the ways in which a user could specify how thorough a search he wished the program to make.

Given that to recurse on every single `pe` is computationally infeasible, the code must select only the highest-scoring `pes`, and recurse on these. There are two ways of doing this; either the code can recurse on the  $k$  highest-scoring `pes`, or on all `pes` whose scores are within  $l\%$  of that of the highest-scoring `pe`. The latter method is better, for several reasons:

- To select the  $k$  best `pes` requires  $k$  passes through the list of scored `pes`, whilst to select all those within  $l\%$  of the highest score requires only 2 passes.

- For a tune with 30 chords, with<sup>3</sup>  $k = 2$ , the number of events to be examined is  $2^{30} \simeq 10^9$ , which I thought would be too long<sup>4</sup>.

If the scores were smoothly distributed, by changing  $l$  we could make the mean number of `pes` in the top  $l\%$  be any real number  $m$  greater than 1. This gives us a theoretical  $m^{30}$  e.g.  $1.4^{30} \simeq 24000$  event examinations, which would take about 4 minutes in the final version of the code.

- Often there is either only one completion worth looking at, and a student will just write it straight in, or several which are equally good, and each should be examined. Taking scores within  $l\%$  of the best score allows more flexibility to deal with this problem, and also more closely mimics the behaviour of a human student.

### 3.4.2 Implementation

With the design of the main recursive algorithm largely decided by complexity considerations, its implementation was reasonably straightforward, the main problems being in maintaining a record of the path taken so far, and the overall best path found so far.

The algorithm was implemented as a single function, `rel_score`, whose basic outline was:

- Have we filled in all the tune's events? If so, return, saying whether we beat the previous `best_tune_score`.
- Score all the possible completions of this chord, relative to those chosen in the preceding instances of `rel_score`.
- Find the best score, and recurse on all completions whose scores are within  $l\%$  of the best.
- After each recursion, if the returned value says we found a new best tune, fill in that completion in `best_tune_path`.
- When all the recursions have been done, return, saying whether any of our completions was part of a new best tune.

Originally, it was intended that some scoring functions should return a negative score for a transgression of a rule of harmony, and others should return a positive score for a good harmonic progression. This would give us a maximum score which could be amassed per event. Knowing this, and the number of events still uncompleted in the current tune, it is possible that we could see there was no chance of beating the best tune's score. This opened up the possibility of using 'branch-and-bound' techniques to prune the search tree considerably [AHU, p331].

---

<sup>3</sup>If we make  $k$  any smaller, i.e. 1, then we will in effect have a purely iterative algorithm, which selects the locally best chord at the expense of the global structure of the tune.

<sup>4</sup>In the final version of the program, it took about 30 hours to examine  $10^7$  events.

```

for (each event in input)
  for (each possible doubling of an interval)
    for (each assignment of intervals to voices, where
          melody interval, if specified, is in soprano)
      for (each occurrence of tenor's interval in tenor's range)
        for (each occurrence of alto's interval in alto's range)
          for (each occurrence of soprano's interval in soprano's range)

            Make and store a possible event with these attributes.

```

Figure 3.5: Basic Structure of `makepes.c`

As it turned out, it was possible to write the scoring functions so they only ever returned 0 or a negative number, by subtracting points when something good *failed* to happen.

Hence, any instance of `rel_score` called with a `score_so_far` less than `best_tune_score` could never produce a new best tune, and so could return immediately. If this happened at, say, the 10th event out of 30, (e.g. if the score for the current tune so far was  $-65$  and the best score for the whole tune was  $-60$ ) and there were 25 pes per event, then we have saved ourselves having to examine

$$25^{(30-10)} \simeq 9 * 10^{27}$$

possible completions of chords. With this hugely powerful advance, it now became feasible to consider much larger tunes. For instance, for a 30 note tune by Handel, 143,619 calls to `rel_score` were sufficient to find the best path through the tune, ruling out every single other path, amounting to nearly  $8 * 10^{51}$  paths.

I believe that the combination of

- constructing as much of the nodes as possible (i.e. everything apart from their relative scores) outside of the recursive tree search, and
- using the variation on branch-and-bound pruning described above

played a major part in the success of the harmoniser, by allowing it to search the tree exhaustively in a reasonable time.

## 3.5 Music

### 3.5.1 Policy Decisions

I decided at an early stage of the project to use the figured bass format as input, rather than the melody line:

- Using a melody as the input imposes too few constraints on the tune, and so makes it difficult to judge the output.

- Working from a melody requires far larger musical intuition, which takes years of experience to build up. I have not had this experience, and would be therefore unable to write code that simulated it.
- Previous attempts to use a melody as input have run into complexity difficulties; the problem is computationally explosive.

I would, however, allow the input to specify a melody line. It is conventional in figured bass notation to occasionally specify a melody note, where the composer has a particular progression or *motif* in mind.

As many of the harmony rules are different for major keys and minor keys, I decided to base my knowledge base on major tunes only. This was not a severely limiting decision, and prevented too much time being spent on the musical aspects of the program, rather than the Computer Science aspects.

Inessential notes help to take away some of the artificial feel of computer-generated music. Although the harmoniser was not intended to mimic the style of any one composer, Bach's use of passing notes and suspensions seemed particularly suitable. As the program was intended for use by a musician, it seemed better to represent inessential notes in the output by some sort of symbols, rather than outputting all the notes for the chord twice.

At the start of the year, my knowledge of harmony was minimal. I accordingly designed the project schedule so that the sections of coding and testing requiring the greatest musical knowledge were right at the end of the year. During the course of the year, I intended to build up my practical and theoretical ability, mainly by reading and doing the exercises in harmony manuals.

### 3.5.2 Implementation

#### Input - Figured Bass Syntax

Whilst the design of the input stage was underway, I began to build up a library of harmony manuals [Morris] [ManH] [FigH] [BnM]. Reading through these gave me a basic understanding of the principles of harmonising from a figured bass, and of the syntax and semantics of the figured bass format. As the syntaxes given by the various authors were not entirely in agreement, I had to spend some time deciding on which syntax my project should adopt. This involved some major work, as decisions at this stage could have a profound effect on the coherence and efficiency of the harmony stage. The initial format accepted by the input stage was the intersection of the various syntaxes, including the more common abbreviations of figures. As the code developed, more abbreviations were added. Some advice was received from my project supervisor, Mr. W. F. Clocksin, on which figured bass conventions to follow.

The symbols '#', 'b', 'n' and 'x' were used to represent the accidentals '♯', 'b' '♮' and 'x' (double sharp); the inclusion of a lower case 'b' meant that the letters used for note names must be upper case. To avoid having to deal with fractional or real-valued lengths in the input, a 'beat' was defined as the length of the shortest

```

<Input Line> = <Directive>|<Chord>|'newline'
<Directive>  = {'#beats ' <Num>} | {'#rest ' <Num>} | {'#key ' <Key>}
<Key>        = <Note Name> ['b']
<Note Name>  = 'A'|'B'|'C'|'D'|'E'|'F'|'G'

<Chord>       = <Bass Note>['', '<Figure>['', '<Melody>]] |
               <Bass Note>', '<Melody>'
<Bass Note>   = <Note Name><Octave><Accidental>
<Octave>      = '0'|'1'|'2'|'3'|'4'|'5'
<Accidental>  = {'#'|'b'|'x'|'n'}*

<Figure>      = <Interval>[' ', <Interval>[' ', <Interval>]]
<Interval>    = <Accidental><Digit>
<Digit>       = '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<Num>         = <Digit> {<Digit>|'0'}*

```

Figure 3.6: Formal Definition of Input Syntax

note or part of note<sup>5</sup> in the tune. Octaves were measured from C, with Middle C being C in octave 3, thus containing all the voices' ranges without using negative numbers. The accidental was placed after the octave, to emphasise that 'C3b' is one semitone below Middle C, not (as might be thought otherwise) the 'C flat' in the octave above Middle C. The final input format is shown in Figure 3.6, and an extract from an input file is in Figure 3.7.

## Output syntax

With the syntax for a note already defined for the bass note in the input stage, along with notation for length, key and accidentals, the major thought in designing the output format went into a clear, useful layout. The best representation seemed to be a tabbed format, with the chronological order of the tune going down the page, as in the input, and the four parts' notes being set out in columns across the page. This meant it was easy to read off an individual part, or the notes in a chord. The other information in the line was helpful at the debugging and commissioning stages, and has been left in to give the user an indication of the difficulty of harmonising each chord. Figure 3.8 is an extract from the output for the input file in Figure 3.7.

## Harmony Rules

In the current version of the program, there are 3 absolute and 5 relative scoring functions. Some of these contain more than one rule, to keep similar rules together and cut down on the function calling and looping overheads.

The absolute scoring functions were:

**part\_overlap** , which returns a score to indicate by how much two parts overlap within a chord. The penalty for an overlap between the alto and tenor is not

---

<sup>5</sup>A dotted note will force a beat to be the length of the 'dot'.



```

#beats 3
#key Bb

#rest 2
1 B2

1 F2,4 6
1 F2,3 5
1 B1

1 F2,4 6
1 F2,3 5
1 D3,6

2 C3,6

```

Figure 3.7: Extract from Start of Input File handel30.in

```

Reading input from file: handel30.in
Sending output to file : handel30.out.98.1.same

```

(Numbers after colon after notes are:

- (i) number of ways of harmonising input;
- (ii) score for chord considered alone;
- (iii) score for chord in relation to preceding chords.)

An S means suspend the above note; a P means insert a diatonic passing note.

	Soprano	Alto	Tenor	Bass	:(i)	Scores	;Original	Input
New time signature: 3								
New key signature: Bb								
Event 0	Bar	1:1	Len. 2	REST				
Event 0	Bar	1:3	Len. 1	F3	D3	B2	B2	:64 0 0 ;1 B2
Event 1	Bar	2:1	Len. 1	F3	D3	B2	F2	:54 0 -3 ;1 F2,4 6
Event 2	Bar	2:2	Len. 1	A3	F3	C3	F2	:54 0 -4 ;1 F2,3 5
Event 3	Bar	2:3	Len. 1	B3	F3	D3	B1	:64 0 -1 ;1 B1
Event 4	Bar	3:1	Len. 1	B3	F3	D3	F2	:54 0 -14 ;1 F2,4 6
Event 5	Bar	3:2	Len. 1	C4	A3	F3	F2	:54 0 -12 ;1 F2,3 5
Event 6	Bar	3:3	Len. 1	B3	B3	F3	D3	:49 0 -1 ;1 D3,6
Event 7	Bar	4:1	Len. 2	C4	A3	E3	C3	:49 0 0 ;2 C3,6
				S				
:								
:								

Total score for tune = -299

(Percentage recursion = 0.980000, number of recursions = 240528

number of paths checked = 8.37516e+38, out of 1.84571e+52)

Figure 3.8: Extract from Output File handel30.out.98

as severe.

**separation** , which returns a score to indicate by how much each of the distances between adjacent parts exceeds that deemed acceptable.

**doubling** , which returns a score to indicate how acceptable the doubling used is; this varies over the different inversions of chords — a different score is needed for each interval for each inversion.

The relative scoring functions were:

**leaps** , which computes first approximations to the first and second derivatives of the path of each voice over the last two or three notes, and deducts points for parts which leap around too much, as shown by these derivatives. Points are also deducted if a note repeats, moves disjointly (not stepwise), or leaps an interval which is hard to sing.

**consecutives** , which deducts points for any consecutive octave or fifth, explicit or implied, between any two parts. This is made more complicated to test by the possibility that the parts may overlap.

**tedo** , which deducts points whenever ‘Te’ (the leading note) does not rise to ‘Do’ (the tonic). The penalty is more severe in the soprano.

**crossing** , which deducts points if the current note of a voice overlaps the preceding note of an adjacent voice: a greater constraint than **part\_overlap**.

**same** , which deducts points if the same note occurs in a voice three times consecutively; many more points are deducted here than in **leaps** for a repeated note.

The functions to add inessential notes were an addition to the project, over and above what was stated in the Project Proposal. The functions required perhaps the most complicated code of the whole program, especially as far as considering multiple viewpoints is concerned. The constraints used to determine whether an inessential note should be inserted were:

**add\_suspensions** :

- A part falls by one step or rises by a step of a semitone, and
- The second note is not the fifth of the chord, and
- The second note is not in any of the upper parts.

**add\_passing\_notes** :

- A part moves by a simple third, and
- There is no suspension on the same note, and
- The second note is on an accented beat; if it is unaccented, the passing note is only added half of the time.

The basic theory behind these rules was gleaned from the harmony manuals mentioned at the start of this section. The actual coding of the functions required much thought, especially those like `tedo` and the inessential note functions, which required the consideration of the chords from several viewpoints.

The numeric values assigned to each of the faults found by the scoring functions were initially guesswork, based on how ‘bad’ a manual writer or musical friend thought a particular transgression was. The distinction between constraints and heuristics was blurred by their being represented merely by larger or smaller negative values respectively. This was useful, in that it allowed the harmoniser to produce output, even when that the tune was forced to break a major rule, e.g. consecutive octaves. Indeed, the first tune I invented to test the harmoniser forced such a progression, by making the melody move in octaves with the bass. A few worried minutes were spent searching for the bug in a harmoniser which appeared to have broken a basic rule, before I realised that the fault lay with the input! The values of the faults were gradually altered by repeatedly testing the code against a broad selection of tunes, changing apparently suspect values each time.

Much helpful advice was received at this stage from my brother, Paul, a student at Leeds College of Music, and grade 8 pianist. He painstakingly transcribed the output I read to him over the telephone, played it, and rang me back half an hour later with his verdict and advice. Suggestions for additional rules were also made by Stephen Turner, Joel Goldstein and Zoë Flint, all of whom are from Christ’s College, and sing in various choirs. Their comments, along with those of others whose musical ability exceeds mine, can be seen in Section 5.2.

# Chapter 4

## Results

This chapter looks at the results obtained by runs of the harmoniser on various figured basses, both true chorales and text book exercises. In judging these results, the reader should remember that the project's aim was to produce conservative harmonisations which break no rules, rather than original-sounding masterpieces.

It looks first at the results which would be expected of a student of harmony, then takes an overall view of the program's results, and finally includes some completed tunes with comments on their quality.

### 4.1 Results Expected of a Harmony Student

A student composing harmony must continually balance the desire to obtain good melodies in all parts, especially the soprano, against the requirement not to break any major rules of harmony.

- The soprano should have an interesting and pleasant melody.
- The inner two parts should have melody lines which are not too difficult to sing, i.e. contain no unusual interval leaps.
- The inner parts' rhythms are allowed to be quite difficult, to allow for suspensions.
- The inner parts need not be spectacular, and will normally stay in a fairly narrow range. Repeated notes will thus often have to occur, and these may well be merged into single notes.
- Inessential notes should be added in keeping with the style of the composer or period.

A reasonable indication of the quality of harmony is to look at the flow of the tune as a whole, and of each individual part. It should be possible to see each part's path distinctly through most of the tune. Complicated looking chords, with many accidentals or notes very close together, should generally only occur when the figure below the chord also looks complex.

## 4.2 Results Obtained

The results produced overall were impressive. In particular, when run on a real figured bass, or one of the earlier textbook exercises, the harmoniser produced very pleasant, singable tunes. Even on later more complicated exercises, which tend to have too many complex figures causing discords, the output was sound both harmonically, and to a slightly lesser extent, melodically. As the intention of these later exercises is to test a student, rather than produce pleasing music, I believe the program still produces output which would be considered good by the exercise writer.

The harmoniser can produce good results on medium-length (under 20 notes) tunes in one or two seconds; much faster than previous systems. On the longer chorales (about 30 notes), it takes about 20 seconds for fair output, and about 5 to 10 minutes for good output. Most runs for the project were done with a percentage recursion of 98%, to practically ensure that the best path through the tune was found. These runs averaged between 20 and 30 minutes in length. With a percentage recursion of 0%, every single path is accounted for; this can take hours, and only once has produced a better score than the runs with 98%.

Longer tunes tend to be about 80 notes in length, and I doubt that the current version of the program could cope with these in a reasonable time. Despite looking through 6 harmony manuals for exercises around 45 notes long, I could only find 1, a piece by Mendelssohn. It took 2 hours to harmonise at 98%, and the results were more than satisfactory. It is discussed further in the next section.

Running the program using the `prof` profiler shows that 99.89% of the time is spent in the relative scoring functions and algorithm. This proves that the effort spent making these efficient was worthwhile; even more encouraging is the fact that the algorithm only took up less than 14% of the time, enabling most effort to be spent scoring the possible completions.

My project supervisor, Mr. W. F. Clocksin, provided me with some fragments of a piece by Frescobaldi, which contained remote unprepared harmonies — technically very difficult. With this piece, *Sonetto Spirituale*, as input, the program produced, in his words,

“A convincing harmonisation, unlike other systems, which would probably have run out of time.”

The addition of the inessential notes code proved well worthwhile; these notes added life and interest to many pieces. Whilst some inessential notes sound odd, these only occurred where the figure was complex or the scale used unusual.

Although the program was written mainly for tunes in major keys, it produced surprisingly good output for minor keys. It seems that enough of the sense of the minor key is given by the figures to force the program to choose harmonies that sound good for the key.

Just about all of the output produced by the current version of the harmoniser would be considered good if written by a harmony student. Some tunes produced

are most impressive, sounding like the work of a proper composer, albeit a relatively minor one!

### 4.3 Tunes Produced

The tunes chosen for inclusion here have not been selected necessarily because they were the best produced. Some are included because they show up deficiencies in the harmoniser, to encourage further work, and some are included to show how the program coped with long or difficult input.

Tunes whose name begins with `tune` are exercises from a harmony manual [ManH]; those whose name begins with that of a composer are whole pieces or fragments by that composer, from a harmony manual which teaches using real works [BnM]. The numbers after the name are the numbers given by the authors of the respective books; more difficult tunes appear later and hence have higher numbers. The tunes are included in alphanumeric order.

The comments after each tune are intended to be useful to the reader without great musical knowledge, rather than to be scholarly music *critique*. The lower the value of ‘p’ in the caption, the greater the number of paths were recursed on.



Figure 4.1: Output for Corelli36 — p=98%

#### 4.3.1 Corelli36

Overall: The input contains many 7th chords, and has a monotonous rhythm. This makes it all the more remarkable that the harmoniser has produced a jolly tune. The inessential notes sound really good, especially the run around the start of bar 6, and the preparation for the end at the start of bar 8. The effect produced by the inessential notes moving down through the voices in bars 1 and 2 is very impressive. The tune has a sense of purpose throughout, and it is easy to follow the beat.

Soprano: The part is excellent melodically throughout.

Alto: Some notes are repeated, making the part hard to sing, but these are important in the overall effect.

Tenor: Stays between B $\flat$  and F the whole time, which is good for an inner part. Does not lose its sense of purpose as the alto sometimes does.

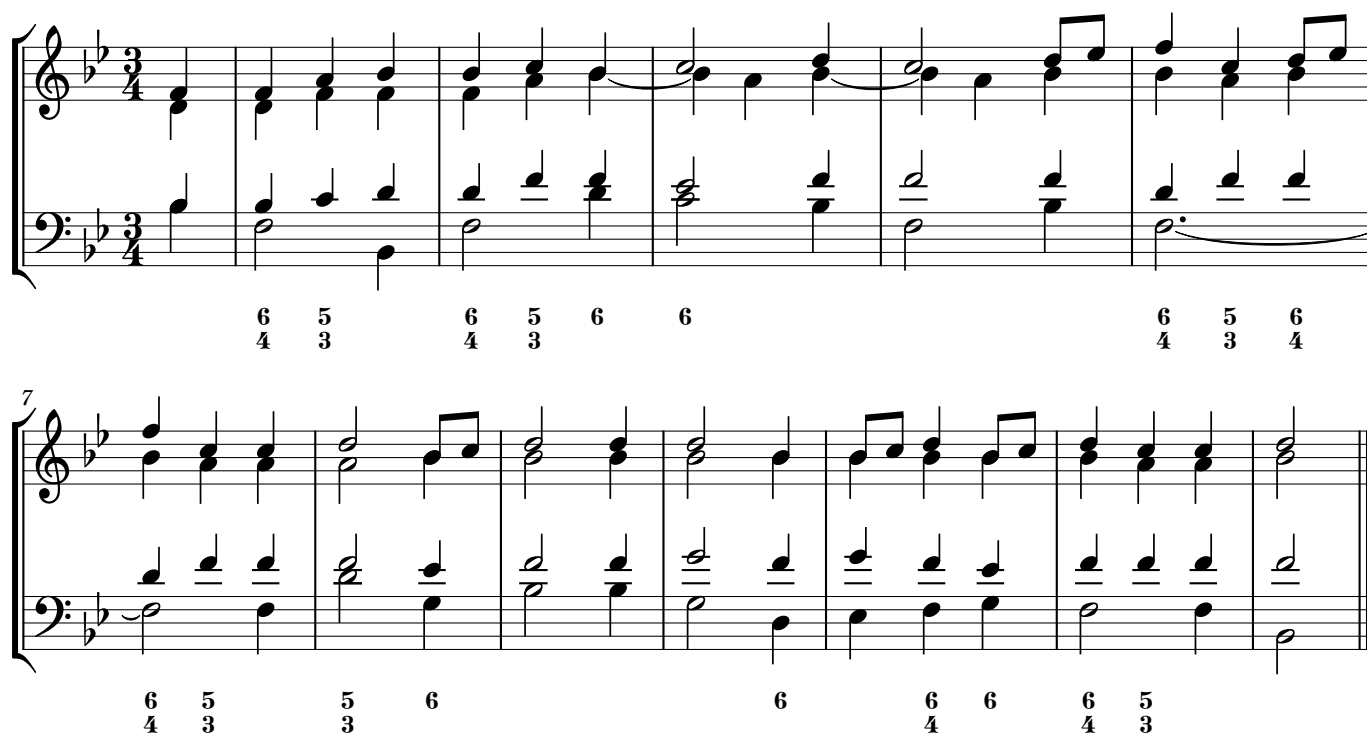


Figure 4.2: Output for Handel30 — p=98%

### 4.3.2 Handel30

Overall: A most impressive tune. The stately, almost majestic progression has flavour and interest added by well-chosen inessential notes. The repeated *motifs* in bars 4 to 6 are very good. The final chord is well-prepared and executed.

Soprano: An excellent, memorable melody.

Alto: There is some repetition, but this does not come across too badly, and is necessary for the effect of the whole tune. The suspensions in bars 4 and 5 sound particularly accomplished.

Tenor: Good voice leading, and stays in the range from B $\flat$  to F. The rhythm is easy to follow and sing.



Figured bass figures for the first system (6 measures):

6	6	7	—	6	—
4	—	5	—	4	—
2	—	4	—	2	—

Figured bass figures for the second system (7 measures):

7	—	6	5	7	7	5
4	—	4	3	5	5	3
3	—	—	—	3	4	3

Figure 4.3: Output for Handel226 — p=98%

### 4.3.3 Handel226

Overall: A very good tune which sounds interesting. It copes well with the complicated figures in the input, and overcomes the temptation to end the first phrase too early, at the start of bar 3. Maybe I should have allowed some inessential notes at the command line. The ends of bars 4 and 7 are excellent, and bars 10 and 11 sound superb.

Soprano: A good melody line, helped by keeping most of the unusual notes in the lower parts. It is eminently singable.

Alto: Possibly gets a few too many of the unusual notes near the start, but the second half is better.

Tenor: Contains some interval leaps which would be difficult to sing. Bars 9 to the end are, however, excellent.

#### 4.3.4 Handel227

Overall: The second half is superb, especially the two bars from from bar 7 beat 3, which are reminiscent of Beethoven's 'Moonlight Sonata'. The ending is very good. There are, however, some suspect inessential notes in bars 2 and 3; the figures are too complicated here, and they should probably have been omitted. This apart, the tune is most impressive, sounding more like the work of a real composer than a student.

Soprano: Good throughout; the figure prevents the expected 'E C C $\sharp$ ' ending. The passing note in bar 4 is particularly well-placed.

Alto: Fair, but not very memorable. There are some nice touches, but the multiple suspensions tend to make it lose its way as an individual part, although they work well with the other voices.

Tenor: The second half is good, but the first half lacks purpose when listened to alone. Keeps within a sensible range of G $\sharp$  to E.

Handwritten musical score for Handel 227, measures 1-9. The score is written in C major, common time, and features a treble and bass staff. The figured bass notation below the staves is as follows:

Measure	Figured Bass
1	$\flat$
2	$\flat 5$
3	$\flat 2$
4	$\flat 6$
5	$\flat 7$
6	$\flat 5$
7	$\flat 7$
8	$\flat 5$
9	$\flat 7$

Figure 4.4: Output for Handel227 — p=98%

### 4.3.5 Mendelssohn293

Overall: Loses its way a bit around bar 6, but very good from bar 10 to the end.

The ♮ in the last figure makes the tune sound unfinished.

Soprano: Good voice leading; the complicated figures make the melody sound a little odd in places.

Alto: Very good — an ideal inner part.

Tenor: Not quite so good, mostly because of repeated notes, which would be merged by a composer. The many possible suspensions would make it sound better, but are not compatible with the style of the composer.

Figure 4.5 displays a musical score for Mendelssohn293, showing measures 1 through 19. The score is presented in two systems, each with a treble and bass staff. The key signature is B-flat major (two flats). The time signature is common time (C). The figured bass notation is provided below the staves, indicating the harmonic structure of the music.

**Measure 1:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: — (line), 6, 6, 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 2:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 3:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 4:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 5:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 6:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 7:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 8:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 9:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 10:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 11:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 12:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 13:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 14:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 15:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 16:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 17:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 18:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

**Measure 19:** Treble staff has a whole note chord (F4, A4, C5). Bass staff has a whole note chord (F2, A2, C3). Figured bass: 6,  $\frac{6}{4}$   $\frac{2}{2}$ , 6, 7, 7,  $\flat 6$   $\frac{5}{3}$ ,  $\flat 5$   $\frac{4}{4}$ , #.

*bar missing from '91 thesis*

Figure 4.5: Output for Mendelssohn293 — p=98%

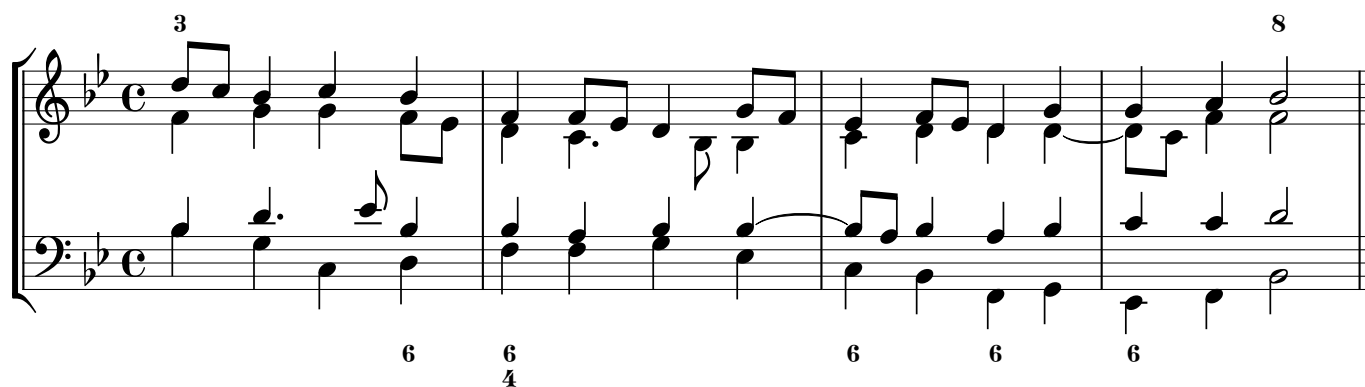


Figure 4.6: Output for Tune44 — p=0%

### 4.3.6 Tune44

Overall: A rolling little tune, with a very good overall pattern of pitches, especially the rise at the end. The many good inessential notes give it a ‘musical box’ flavour.

Soprano: The repeated passing note *motif* in bars 2 and 3 is excellent. Very much a singer’s melody.

Alto: Very singable; a typical alto part — good but nothing spectacular.

Tenor: Fair, loses its way a bit in the second half, but good as far as voice leading goes.



Figure 4.7: Output for Tune121 — p=98%

#### 4.3.7 Tune121

Overall: Sounds ‘muddy’ — perhaps there are too many inessential notes which clash rhythmically. A good end, although the rise is a bit sudden. It is possible to follow the beat, but the piece as a whole does not sound especially interesting.

Soprano: The end sounds a little unfinished. The suspensions take away the sense of rhythm, and the larger leaps are in the wrong places rhythmically.

Alto: Quite good, especially in the turn at the end. Again, the suspensions make it hard to follow the beat.

Tenor: By far the best part. Bars 4 to 6 are definitely good. The suspensions near the end make it slightly too hard to follow the beat, but the part still holds together.

# Chapter 5

## Conclusion

### 5.1 Success of the Project

Looking back at the original aims in the Project Proposal, I feel that the project has fulfilled all it set out to do, and more besides:

- The program can produce good 4-part harmony from a figured bass line;
- Over the course of the project, I have become increasingly adept at translating vague English sentences into logic;
- Reading around the subject, and looking at previous knowledge-based systems, e.g. CHORAL, has given me a better understanding of these;
- The UNIX<sup>TM</sup> environment, and especially X windows, is now second nature to me;
- The practical application of many lectures on good programming technique given under the guise of various courses has made much of the advice and rules stated more ‘real’ to me;
- I now have a thorough knowledge of the C programming language, especially as defined in the ANSI standard, and of writing portable code in this language;
- The output produced is conservative, as opposed to radical. A common complaint from choristers is that the tunes produced are “quite boring”; this is often followed by the explanation “but it’s very like what a student would write”.

### 5.2 Quotes from Musicians on the Harmoniser’s Output

“This is really very good.”

Mr. W. F. Clocksin, on seeing the first proper output.



“Very good harmony, no mistakes, good voice leading, and all sensibly done.”

Paul Kelly, on Handel30.

“Is that your harmoniser? ... That’s a lot better than it was before, isn’t it?”

Joel Goldstein, on the change after adding 2 scoring functions.

“And your program did this? That’s really good. It’s not very exciting, but it’s fine as far as the harmony goes.”

Zoë Flint; she walked away whistling the melody!

Mr. Clocksin is my project supervisor, but also an expert on *continuo* harmony, and plays in the Cambridge Early Music Orchestra. Paul Kelly, my brother, is a grade 8 pianist and a student at Leeds College of Music, having been accepted there on a matriculation offer. He has sung in several choirs and musicals. Joel Goldstein is a bass in the Cambridge University Madrigal Society. Zoë Flint is a soprano in the Christ’s College Chapel Choir, and also sings in other choirs and musicals.

## 5.3 Further Work

Whilst all the work stated in the Project Proposal has been completed, there are still areas which it would be interesting to explore further:

- Some further harmony rules could be added; for instance, if two voices sing the same pitch note, they should leave that note by contrary motion, and the addition of inessential notes should check that it will not introduce consecutives.
- The code at present chooses randomly between tunes with the same score; although the `-s<rand_seed>` command line argument will enable all tunes to be produced, there could be an option that would keep all of the best tunes found, and display them all.
- In a true generate-and-test paradigm, the paths are recursed on in order of score; this was not implemented as the sorting was felt to be too slow. Comparing the times taken to find the best score would be of some theoretical interest.
- The program has not been tested on very long input, but I believe it would have difficulty with tunes over 60 events long. The tune could be split into sections of approximately 30 notes, and each of these harmonised in turn. It would be useful if each sub-tune overlapped its successor by 3 notes; this would allow the harmony to flow well over the joins between sub-tunes.
- The code in `harmony.c` could be used as the ‘brains’ of a harmony tutor, which would score students’ harmonies and explain the faults, suggesting better progressions where necessary.

# Appendix A

## Project Proposal

Name: Steven Kelly  
College: Christ's

Project Title: Automatic 4-part Harmonisation.  
Project Originator: S.M. Kelly

Director of Studies: Dr. I.M. Leslie  
Project Overseers: Dr. A. Hopper  
Mr. W.F. Clocksin  
Project Supervisor: Mr. W.F. Clocksin

### Description of Work Planned:

#### Broad outline:

The project is concerned with the area of music composition known as '4-part harmony', where the intention is to produce a piece of music for 4 voices singing together. There are many pedagogal books written on the subject of harmony, which is usually taught formally.

#### Project aim:

To write a program that will produce such musical works by 'fleshing out' a basic input. The input will contain the bass line of the piece (i.e. the lowest of the 4 parts), and details of the chord progression desired by the composer.

An historically conventional method of teaching harmony is to ask students to harmonise a 'figured bass', which fits the constraints for input described above, and this would therefore be a sensible input format.

Several other goals must be met to accomplish in these aims:

- to learn more exactly the rules of harmony,
- to learn techniques for formalising rules into code,
- to learn more about knowledge-based systems,
- to learn to use the UNIX facilities for software development,
- to acquire and use good modular programming technique,
- to learn to program effectively in C and Prolog.

## Methods:

The method to be used is a knowledge-based heuristic search technique, which will contain various constraints (i.e. rules that the harmony must obey, e.g. ‘never allow parallel octaves’), and also heuristics (i.e. rules that would enhance the musicality of the harmony, e.g. ‘make the upper parts move by steps rather than leaps whenever possible’).

The intention is to make the approach of the harmoniser as conservative as possible, so it is more important that the harmony breaks no constraints, than that it sounds particularly brilliant.

## Implementation:

The ideal output from any music system is both sound and score — unfortunately, this cannot be achieved on standard computer facilities in Cambridge, as they are not designed for complex sound output.

Input will initially be in the form of a ‘figured bass’ score from a textbook. This will have to be entered into the computer as text, or by use of a music editing program such as that written by Mr Clocksin.

The processing will be done on a Unix based system, as only these have sufficient support for Artificial Intelligence languages. It is hoped to use one language throughout the development — a version of Prolog with extensions in C — failing this the input and output processing will be performed in C on files needed or produced by the Prolog harmoniser.

Output would be acceptable as pure text, as this would be sufficient to determine the project’s success at harmony. An extension to the project could be to produce output as a score using existing music notation software, and/or output as music using a PC with a MIDI card linked to a synthesiser.

The output for the dissertation will be a scored version of several pieces composed by the program, along with notes on how they conform to the rules of harmony, and comments on how they sound musically.

## Resources required:

The project can be carried out to completion using only standard resources.

However, for the additional output facilities described above, a musical type-setting program will be required — Mr Clocksin has offered the use of his — and a MIDI card on my PC would enable sound output to be produced. Both these are merely amendments to the project's output, which are not necessary for its full completion.

## Starting point:

An early part of the project will be acquiring a knowledge of the constraints and heuristics used by real composers in their works, and translating these into rules implementable by the computer. I have a basic knowledge of music and harmony already, but must investigate further the computer science methods of formalising sets of vaguely-expressed rules.

I have attended the CST lectures in Prolog and C, but am yet to attempt any sizable piece of software in these languages.

I have little experience of using the DEC workstations on which I intend to do my project.

## Plan of Work:

week ending:

Oct 27     \     Familiarisation with tools, languages and harmony.

Nov 3     |— Preliminary decisions on data structures.

10     /     Decisions on language use and modularity.

17     \\_     Write code for input and output (text format).

24     /     Finalise data structures.

Dec 1     >     Write basic rules and produce output for checking.

----- Christmas Vacation -----

Jan 19     \\_     Add more rules.

26     /     Start heuristics.

\*\*\*\*\* Jan 26-31 Produce preliminary results. Write progress report.

Feb 9     \     Add final few rules, amend existing ones as necessary.

16     |— Finish rest of heuristics.

23     /     Iterate these 2 depending on output quality.

week ending:

Mar 2     \\_    Acquire and format results, consider enhanced output.  
          /    Write dissertation structure.

----- Easter Vacation -----

Apr 27    \\_    Write body of dissertation.

May 4     /    Fine-tune results as time allows

11       >    Finish dissertation

\*\*\*\*\* May 16     Submit dissertation.

Steve Kelly 23/10/90.

# Appendix B

## Code Samples

The samples of code included in the section have been selected to give the reader an idea of the complexity of the code written, and also of the standards I adopted for writing C code. As far as possible, the code has been inserted exactly as it is in the program.

### B.1 `input.c`

This source file reads in a file of figured bass information for the 4-part harmoniser. It parses this text file, and preprocesses it, adding all the different viewpoints it can.

The `input` function manages all the parsing and processing of the input, feeding each line in turn to the relevant function, and handling any errors encountered.

The `str_to_interval` function converts an interval from its text form in a figure or melody into the byte-sized bit-encoded internal representation.

```

int input (void)
/* Aim: to manage whole of input from a file */

{
    char line[82];
    int line_ok=NO_LINES_READ;

    in_stream = set_up_stream (input_file_name, "r");

    /* Read in lines from file, encode input notation into machine representation*/
    while (event < MAX_EVENT && fgets (line, 80, in_stream) != NULL) {
        line_no++;          /* 1st line read is counted as 1 */

        /* action determined by 1st character of input line */
        switch (*line) {
            case '#' :          /* Directive line */
                line_ok = get_directive (line);
                break;
            case '\n':          /* Empty line - ignore it */
                line_ok = GOOD_NO_EVENT_LINE;
                break;
            default:            /* Music line, containing chord information */
                c_chord = tune[event] = make_new_chord();    /* make empty chord */
                line_ok = fill_in_chord (c_chord, line);      /* fill in from line */
        }

        /* Handle any errors */
        switch (line_ok) {
            case GOOD_EVENT_LINE:          /* Line read OK */
                event++;                    /* so move on to next event */
                break;
            case GOOD_NO_EVENT_LINE:       /* No event occurred, nothing to free*/
                break;
            default:                        /* Error occurred - call handler */
                in_error (line_ok, line);
        }
    }
    return (error_in_input);
}

```

```

char str_to_interval (char *int_str)
/* Aim: To return a bit-encoded representation of an interval of the form */
/*      "8b" or "3n" in the string. */
/* Encoding (from lsb up) is 3 bits for note, 3 bits for acc, 1 bit for # */
{ char encoded_interval, int_note,
  acc_str[80];
  signed char acc_val;

  dbprint ("Just entered function 'str_to_interval'\n");

  acc_str[0] = '\0'; /* To give null string if no accidentals */
  /* strcpy doesn't terminate with a '\0', so must use strncat which does */
  strncat (acc_str, int_str, strspn (int_str, "b#xn"));
  int_str += strlen (acc_str);

  /* If figure is just an accidental with no number, then number is assumed */
  /* to be 3; hence make int_str point to a string with '3' in it. */
  if (*int_str == '\0' && strlen (acc_str) != 0)
    int_str = "3";

  if (*int_str < '1' || *int_str > '9') /* 9 OK, gets read as 2 */
    raise_error (BAD_INTERVAL);

  encoded_interval = (*int_str - '1') % 7;
  int_note = (tune[event]->parts[BASS].note + encoded_interval) % 7;

  acc_val = str_to_acc (acc_str, int_note);

  /* Will accidental fit in the 3 bits reserved for its encoding? */
  if (iabs (acc_val) > 7)
    raise_error (BAD_INTERVAL);

  if (acc_val > 0)
    encoded_interval |= INT_SHARP_BIT;
  encoded_interval |= iabs (acc_val) * INT_ACC_MULT;

  /* If we have reached here, we have a valid bit-encoded interval. */
  dbprint1 ("Enc.intvl is oct.%.o.", encoded_interval);
  return (encoded_interval);
}

```



## B.2 pickbest.c

This source file takes a tune with all the possible chords for each event, and finds the best path through these possible events. It scores all the possible events absolutely, and then calls a recursive function, `rel_score`, which scores all the current possible completions and recurses on all scores within `percent%` of the best-scoring completion. Finally, it adds suspensions and passing notes between chords (if required).

The `pickbest` function manages the task of finding a good path through the tune, according to the functions requested by the user on the command line.

The `abs_score` function builds up the absolute score for a possible event, by calling the absolute scoring functions. It is called in a loop which runs through all the `pes`. Although it could do all the events itself, it only fills in one event, to keep it in line with `rel_score`.

`part_overlap` is one of three absolute scoring functions which exist at present. It penalises chords where a voice sings a note higher than the voice which is meant to be above it. `SOPRANO`, `ALTO`, `TENOR` and `BASS` are compile time variables with values from 0–3 respectively.

The tree search algorithm is implemented by `rel_score`, which is broken over two pages here. The value returned, `had_best`, indicates whether a new best path through the tune has been found. A value of 2 indicates that the new best path is better than the `terminate_score` specified on the command line, and causes all the instances of `rel_score` to return without doing any more searching. The function loops through all the possible events once, scoring them using the relative scoring functions, and then loops through again, recursing on any `pes` whose score is greater than `percent%` of the best score found this event. The values these recursions return determine whether the relevant `pe` becomes part of the `best_tune_path`.

`consecutives` is one of the 5 relative scoring functions which exist at present. It penalises any consecutive octaves or fifths between any pair of parts, searching first for explicit consecutives, and later for implied consecutives, which do not carry as high a penalty.

`add_suspensions` is one of the two inessential note adding functions. It runs through a completed tune, adding suspensions wherever possible, in keeping with the chosen Bachian *cliché*.

```

void pickbest (void)
/* Aim: To score each event alone, and then to call <fun> recursively to */
/* score events wrt possible neighbours and so find best path down tree.*/
{ char voice, pe_num, curr_fun, used_fun;
  PEP curr_pe;

  /* Make best_tune_score */
  best_tune_score = -32760;

  /***** Select relative & absolute scoring functions to be used *****/
  for (curr_fun=0, used_fun=0; curr_fun<9; curr_fun++)
    if (rel_fun_select[curr_fun] == '1')
      rel_score_funs[used_fun++] = rel_score_funs[curr_fun];
  rel_score_funs[used_fun] = NULL;

  for (curr_fun=0, used_fun=0; curr_fun<9; curr_fun++)
    if (abs_fun_select[curr_fun] == '1')
      abs_score_funs[used_fun++] = abs_score_funs[curr_fun];
  abs_score_funs[used_fun] = NULL;

  /***** Fill in abs_score for each poss_event considered alone *****/
  for (event=0; event<no_of_events; event++) {
    curr_pe = tune[event]->poss_event_list;
    while (curr_pe != NULL) {
      curr_pe->abs_score = 0; /* Can only go down from here */
      abs_score (curr_pe);
      curr_pe = curr_pe->next_pe;
    }
  }

  /***** Build up paths_omitted *****/
  paths_omitted[no_of_events]=1;
  for (event=no_of_events-1; event>=0; event--)
    paths_omitted[event] = paths_omitted[event+1] * tune[event]->num_pes;
  total_paths = paths_omitted[0];

  /***** Recurse on rel_score *****/
  rel_score (0, 0);

  /***** Fill in from best path into tune *****/
  for (event=0; event<no_of_events; event++) {
    curr_pe = best_tune_path[event].pep;
    for (voice=SOPRANO; voice<=TENOR; voice++)
      tune[event]->parts[voice] = curr_pe->parts[voice];
    tune[event]->abs_part_score = curr_pe->abs_score;
    tune[event]->rel_part_score = best_tune_path[event].rel_score -
      curr_pe->abs_score;
    tune[event]->doubled_int = curr_pe->doubled_int;
  }

  /***** Now add suspensions and passing notes, if required *****/
  if (do_aux) {
    add_suspensions ();
    add_passing_notes ();
  }

  return;
}

```

```

void abs_score (PEP curr_pe)
/* Aim: To score a possible event purely on its own, apart from neighbours */
{
    char    fun_num;
    ABSFUN *curr_fun;

    for (curr_fun=abs_score_funs[0], fun_num=0; curr_fun!=NULL;
         fun_num++, curr_fun=abs_score_funs[fun_num]) {
        curr_pe->abs_score += (*curr_fun) (curr_pe);
    }

    return;
}

```

```

int part_overlap (PEP curr)
/* Aim: To return a score depending on whether parts overlap. */
{
    char voice, overlap;
    int  overlap_penalty=-20, score=0;

    for (voice=SOPRANO; voice<BASS; voice++) {
        overlap = curr->parts[voice+1].abs_pitch - curr->parts[voice].abs_pitch;
        if (overlap > 0)
            if (voice == ALTO)
                score += overlap_penalty * overlap / 4;
            else
                score += overlap_penalty * overlap;
    }
    return (score);
}

```

```

char rel_score (char curr_event, int score_so_far)

/* Aim: To recurse down tree of poss_events, moving down an event for each */
/*      level of recursion, and trying a selection of poss_events for the */
/*      event at which it is called. */

{ char    fun_num, had_best, new_best;
  int     best_score_this_event, accept_equal;
  RELFUN *curr_fun;

  recs_so_far++;
  had_best = 0;
  accept_equal = rand() % 2;
  /* This enables a better statistical evaluation of the algorithm. Rather */
  /* than always accepting the first or last combination of completions */
  /* with equal scores, we choose one randomly. We also have equal_scores */
  /* to maintain a count of how many other completions of the tune were */
  /* found with the same score as the best found. */

  /* Want to return if there is no chance of getting a better score, and */
  /* so avoid doing unnecessary work. */
  if (score_so_far < best_tune_score) {
    paths_checked += paths_omitted[curr_event];
    return (had_best);
  }

  /* If we are at end, we have had a best score (or we would have returned */
  /* in line above). Whether or not we accept an equal best score, we */
  /* want to increment the counter equal_scores; accept_equal will make */
  /* the choice of whether to fill an equal scoring completion into */
  /* best_tune_path random. */

  if (curr_event >= no_of_events) {
    if (score_so_far >= terminate_score) {
      had_best = 2;
      equal_scores = 1;
    }
    else if (score_so_far == best_tune_score) {
      had_best = accept_equal;
      equal_scores++;
    }
    else {
      had_best = 1;
      equal_scores = 1;
    }

    paths_checked++;
    best_tune_score = score_so_far;
    return (had_best);
  }

  best_score_this_event = -32760;
  tune_path[curr_event] = tune[curr_event]->poss_event_list;

```

```

:
```

```

:
/* Score each pe, remember best score */
while (tune_path[curr_event] != NULL) {
    /* Relative scores include absolute scores (until output.c) */
    tune_path[curr_event]->rel_score = tune_path[curr_event]->abs_score;

    /* Only want to use relative scoring functions on events with */
    /* neighbours before them - i.e. don't score event==0 */
    if (curr_event > 0) {
        for (curr_fun = rel_score_funs[0], fun_num=0;
            curr_fun != NULL;
            curr_fun = rel_score_funs[++fun_num]) {
            tune_path[curr_event]->rel_score += (*curr_fun)(curr_event);
        }
    }
    if (tune_path[curr_event]->rel_score > best_score_this_event)
        best_score_this_event = tune_path[curr_event]->rel_score;
    tune_path[curr_event] = tune_path[curr_event]->next_pe;
}

/* Go through all pe's, recursing on all those within percent% of best */
/* NB 94% of -100 = -94 > -100, and so all scores > 0 */
tune_path[curr_event] = tune[curr_event]->poss_event_list;
while (tune_path[curr_event] != NULL) {
    if (tune_path[curr_event]->rel_score + score_offset >=
        (best_score_this_event + score_offset) * percent) {
        new_best = rel_score (curr_event+1,
                               score_so_far + tune_path[curr_event]->rel_score);
        if (new_best) {
            had_best = new_best;
            /* Write this event into best_tune_path - NB must write */
            /* rel_score here as it will be overwritten later. */
            best_tune_path[curr_event].pep = tune_path[curr_event];
            best_tune_path[curr_event].rel_score = tune_path[curr_event]
                                                    ->rel_score;
        }
    }
    tune_path[curr_event] = tune_path[curr_event]->next_pe;

    /* If we have reached terminate_score, just drop back out to event 0 */
    if (had_best == 2)
        break;
}

/* If we have had a new best total score, inform caller by returning 1 */
/* or returning 2 if we want to exit as we have beaten terminate_score */
return (had_best);
}

```

```

int consecutives (char curr_event)

/* Aim: To return a score depending on whether there are any prohibited */
/*      consecutives between this chord and the previous one.           */
/* Must take care when parts overlap, as  $-3\%12 = -3$ . We are concerned */
/* about any 2 intervals that are a 5th or oct, and are same, but if we */
/* apply iabs too early, we also catch consec. 4ths - hence we just need*/
/* to include -7 as a consecutive 5th, and everything is OK.             */

{ char out_voice, in_voice, this_int, last_int, out_drop, in_drop;
  int score=0;
  static int cons_oct=-180, cons_fifth=-180, impl_oct=-30, impl_fifth=-25;

  for (out_voice=SOPRANO; out_voice<=TENOR; out_voice++) {

    if (tune_path[ curr_event ]->parts[out_voice].abs_pitch ==
        tune_path[curr_event-1]->parts[out_voice].abs_pitch)
      continue;

    for (in_voice=out_voice+1; in_voice<=BASS; in_voice++) {

      if (tune_path[ curr_event ]->parts[in_voice].abs_pitch ==
          tune_path[curr_event-1]->parts[in_voice].abs_pitch)
        continue;

      this_int = (tune_path[ curr_event ]->parts[out_voice].abs_pitch -
                  tune_path[ curr_event ]->parts[in_voice ].abs_pitch) %
                  12;
      last_int = (tune_path[curr_event-1]->parts[out_voice].abs_pitch -
                  tune_path[curr_event-1]->parts[in_voice ].abs_pitch) %
                  12;
      if (this_int == last_int) {
        if (this_int == 0)
          /* Consecutive octaves */
          score += cons_oct;
        if (iabs (this_int) == 7)
          /* Consecutive fifths */
          score += cons_fifth;
      }

      else if (this_int == 0 || iabs (this_int) == 7) {
        /* Don't approach 5th or 8ve by similar motion , as this is */
        /* an implied consecutive.                                     */
        out_drop = tune_path[curr_event-1]->parts[out_voice].abs_pitch -
                   tune_path[ curr_event ]->parts[out_voice].abs_pitch;

        in_drop  = tune_path[curr_event-1]->parts[in_voice].abs_pitch -
                   tune_path[ curr_event ]->parts[in_voice].abs_pitch;

        if (out_drop * in_drop > 0 &&
            iabs (out_drop) > 2 && iabs (in_drop) > 2)
          /* Unless one part moves by step */
          if (this_int == 0)
            score += impl_oct;
          else
            score += impl_fifth;
      }
    }
  }

  return (score);
}

```

```

void add_suspensions (void)
/* Aim: To add suspensions to filled-in tune, to give it a Bachian flavour */
{ char voice;
  int do_sus;

  /* Wherever any part (not bass) falls by one step, we should use */
  /* a suspension, unless we end up on the 5th of a chord, or the */
  /* note we are delaying is in the other upper part already, */
  /* The random element means do all possible suspensions for this */
  /* chord; we prefer accented beats for suspensions. */
  /* NB parts[].interval is interval from bass, not chord_root. */

  for (event=1; event<no_of_events-1; event++) {
    /* Don't do if there is a rest in between our notes */
    if ((tune[event]->beat
        != ((tune[event-1]->beat + tune[event-1]->length) %
            tune[event-1]->bpbar)) ||
        (tune[event]->bar
        != (tune[event-1]->bar +
            (tune[event-1]->beat + tune[event-1]->length) /
            tune[event-1]->bpbar)))
      continue;

    if (tune[event]->beat == 0 ||
        tune[event]->beat == tune[event]->bpbar / 2.0)
      do_sus = 1;
    else
      do_sus = 0;
    for (voice = SOPRANO;
        voice <= TENOR && tune[event]->aux.suspension == 0;
        voice++)
      if (((((tune[event-1]->parts[voice].note -
              tune[event]->parts[voice].note + 7) % 7
              == 1) && /*((Down 1 step */
                  /* and */
                  ((tune[event-1]->parts[voice].abs_pitch -
                    tune[event]->parts[voice].abs_pitch)
                    < 12)) || /* not > 8ve) */
              ((tune[event-1]->parts[voice].note -
              tune[event]->parts[voice].note + 7) % 7
              == 6) && /*(Up 1 step */
                  /* and */
                  ((tune[event-1]->parts[voice].abs_pitch -
                    tune[event]->parts[voice].abs_pitch)
                    == -1))) && /* Up semitone))*/
              /* AND */
              ((tune[event]->parts[voice].interval +
              tune[event]->parts[BASS].note -
              tune[event]->chord_root + 7) % 7 + 1
              != 5) && /* AND */
              ((voice > ALTO) || /* if upper part, other */
              (tune[event-1]->parts[1-voice].note /* upper can't have our */
              != tune[event]->parts[voice].note)) && /*final note at start */
              (do_sus)) /* AND random says yes */
      tune[event]->aux.suspension |= (0x01 << voice);
  }

  return;
}

```

# Appendix C

## Elementary Music Theory

Whilst the discussion of the musical elements of the project is kept on as simple a level as possible, some basic understanding of the concepts and terms used is necessary. This chapter is intended to provide the reader without a musical background with that understanding — he may also care to look at the Glossary in Appendix D.

### C.1 Rhythm and Pitch

In music, a series of notes possesses both *rhythm* and *pitch*. Rhythm is the ‘beat’ of the music — how long the notes are in relation to each other. Pitch is a function of the frequency, or rate of vibration, of a note. Together, they form the basis of written music, and, along with varying performance characteristics such as *tempo* (the speed of the tune), *dynamics* (the rise and fall of the volume) and *timbre* (the sound or tone of the instrument), they define how the finished tune will sound.

#### C.1.1 Rhythm Notation and Time Signature

A tune is written as a series of *bars*, each of which generally has the same length in terms of beats or time. Bars are separated by single vertical lines (*bar-lines*), and contain varying numbers and lengths of notes and pauses (*rests*) to fill up the allotted time of the bar.

The notation for lengths in music is based on powers of 2, with a base which is now normally taken as the *semi-breve*, a note 4 beats<sup>1</sup> long. The shorter notes are each half the size of the next larger, right down to a *hemi-demi-semi-quaver*, which has  $\frac{1}{64}$  of the length of a semi-breve. Adding a dot after the round head of a note makes that note have  $1\frac{1}{2}$  times its normal length. A curved line connecting the heads of two notes of the same pitch is a *tie*, and makes them into one continuous note, whose length is the sum of the two notes’ lengths. Figure C.1 shows the notation for the lengths of notes and their equivalent rests.

Each tune also has a *time signature*, which specifies how many beats there are to a bar, and how long each beat is (again, relative to a semi-breve). Time signatures look

---

<sup>1</sup>The word ‘beat’ is normally taken as referring to crotchet beats.



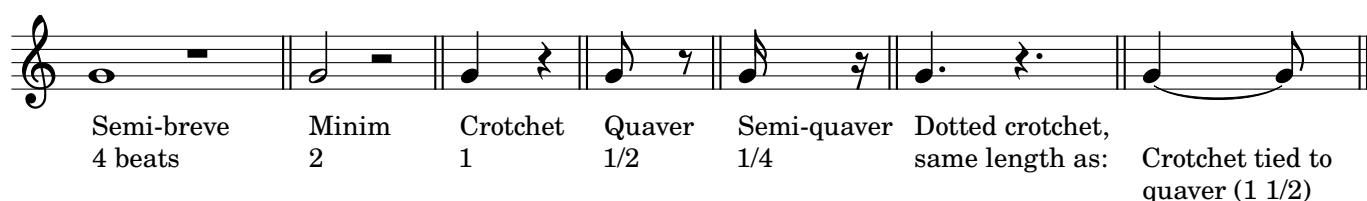


Figure C.1: Notation for Rhythm and Length<sup>2</sup>

like fractions, with the numerator being the number of beats, and the denominator the number of beats which would equal a semi-breve. The tune in Figure C.3 would be said to be in ‘four-four time’ — 4 crotchet (quarter-note) beats to a bar. The first bar of the tune only has one beat in it; this is called an *anacreusis*, and the full length of the bar is made up by the 3 beat long last bar.

## C.1.2 Scales and Interval Notation

Musical pitch is also based on powers of 2; in fact, pitch is the logarithm to base 2 of the frequency of a note. Each doubling of the frequency results in an increase of an *octave*. The octave is a natural division to the ear; ask a man and a woman to sing the same note, and most likely the man’s note will be an octave below the woman’s. A rising sequence of notes where the last note is an octave above the first is a *scale* (from the Latin *scale*, a ladder).

Each octave is divided into 12 equal intervals, called *semitones*. The *chromatic* scale ascends in just semitones, and is equivalent to playing 12 consecutive black and white notes on a piano. More important musically is the *diatonic* scale, which ascends in 7 unequal steps, or *degrees*. Two diatonic scales are in use today, the *major* and the *minor* scales. The degrees in the major scale correspond to the lines and spaces on the 5-lined musical *staff*; the white notes on a piano form a major scale. The minor scale has the third and sixth degrees of the major scale flattened (lowered by a semitone).

<sup>2</sup>The double bar-lines above indicate that the bars are not of regular length

<sup>3</sup>The triad formed on the leading note is diminished

Numeral	Name	Interval	Semitones Above Root	Triad Type
I	tonic	unison	0	primary
II	supertonic	second	2	secondary
III	mediant	third	4	secondary
IV	subdominant	fourth	5	primary
V	dominant	fifth	7	primary
VI	submediant	sixth	9	secondary
VII	leading note	seventh	11	secondary <sup>3</sup>


Figure C.2: Note Names and Intervals in the Major Scale

The degrees of the major scale can be written as Roman numerals, or as special names (see Figure C.2). If an interval is said to be *minor* or *diminished*, it is smaller; if *augmented* it is larger. *Perfect* or *major* emphasise that an interval is normal size. An *inversion* of an interval is simply one in which the lower note is moved up an octave, so an inverted fourth is a fifth, and an inverted major third is a minor sixth.

### C.1.3 Pitch Notation and Key Signature

Music is written on a *staff* (plural *staves*), consisting of 5 horizontal lines. Notes are written in the lines and spaces, which correspond to the degrees of the scale. The higher a note is on the staff, the higher its pitch; if a note is so high or low it ‘falls off’ the staff, further short horizontal lines (*ledger lines*) are drawn in above or below the regular 5 lines, and the note is placed on or touching one of these. The curly symbol at the start of a staff is the *clef*, and indicates the coarse pitch of the notes. The standard clef is the *treble clef*, which usefully represents the range of the female voice. The *bass clef* allows representation of the lower notes of the male voice without excessive use of ledger lines. ‘Middle C’ (roughly the middle note on the piano) falls just between the clefs, being one ledger line below the treble, and one above the bass.

The notes of the staff have single-letter names from ‘A’ to ‘G’, cycling back to the start once each octave. In the *key* of C, the base note of the scale is C, and so a 1-octave scale is CDEFGAB, as shown in Figure C.3. If we are in a different key, for example G, the semitone intervals in Figure C.2 must remain the same, making the seventh note of the scale be one semitone above an F. This note is an ‘F sharp’, indicated by a ‘#’ sign before the note. To save having to write a ‘#’ before every F in the tune, the key is indicated by a *key signature* after the clef; in this case a ‘#’ sign placed on the F line at the top of the staff. This means all Fs in the tune are to be read as if they were preceded by a sharp sign, unless otherwise indicated. The ‘#’ sign, and its relatives the flat (‘b’ — lower by a semitone) and natural (‘n’ — play the white note, irrespective of previous signs) are together called *accidentals*. They are still necessary in the main part of a tune, as the composer may wish to use notes not in the major scale. When they do occur, their effect lasts till the end of that bar, and affects all voices. Tunes in minor keys are written in the key signature of the relative major, which is 3 semitones higher (e.g. C minor has the key signature of E $\flat$  major), as this reduces the number of accidentals used mid-tune.



Key of G: one sharp. F at \* sharpened.

Note anacrusis of 1 beat.



C D E F G A B

C major scale

Figure C.3: Notation for Pitch, and Key and Time Signatures

## C.2 Triads and Harmony

The basic unit of harmony is the *triad*, which is composed of a root note, and the third and fifth above it. An *inversion* of a triad occurs when the root is moved up an octave: this can be looked at now as being a chord of a root, the third and the sixth above it. Indeed, in figured-bass notation, that is exactly the view taken. Similarly, inverting this chord again gives the second inversion — a root, and the fourth and sixth above it. A further inversion merely moves the chord back to root position, but an octave higher than it started. A triad is major or minor, according to the dimension of the third — generally primary triads are major, and secondary are minor. If we wish to split the 3 notes of a triad among four voices, we are forced to *double* one — that is, to let two parts sing the same interval of the chord (not necessarily the same note — they may be octaves apart). The choice of which interval to double is not free: some intervals are better suited to doubling for a given chord.

# Appendix D

## Glossary

**Cadence:** The function of cadence in music is, broadly speaking, identical with that of punctuation in literature. It is a sequence of (normally two) chords, which, because of their position in the tune and relation to the key, give an impression of the end of a musical phrase.

**Accented:** An accented beat is one which, because of its position in a bar, is played louder. The first beat is always accented; in even times, the beat halfway through the bar is normally accented; in 3/4 time, only the first beat is accented; in other odd times, the mid-bar accented beats vary.

**pe (‘Possible event’):** pe is an abbreviation of possible event, and describes an object which completely specifies the notes in a chord, including their note name (‘A’–‘G’) and their absolute pitch, measured in semitones.

**Inessential Notes:** Essential notes are those in the chords specified in the input. Inessential notes lie between these chords, and help break the monotony of consecutive block chords. Common inessential notes are suspensions, where a part moves by step, and a note is delayed, or suspended, by holding the previous note; and passing notes, where a part moves by a third, and the diatonic note between the essential notes is inserted between them.

**Stepwise Motion:** Also known as conjunct motion, this occurs when a part moves only a small amount — at most a degree of the scale — between consecutive notes. The opposite is called disjoint motion.

**Voice Leading:** This is a view of a tune considered by looking at the melodic progression of individual parts.

**Score:** A score is a piece of music written in standard musical notation. The term is commonly used to describe music where there is more than one note being

played at once. A score consists of groupings of staves, called systems; notes in line vertically in a system are played simultaneously.

**MIDI:** An abbreviation for Musical Instrument Digital Interface, the music industry standard for communicating note information as data between electronic instruments. There exist MIDI interfaces which enable computers to send and receive MIDI data, enabling them to play synthesisers etc.

# Bibliography

- [Gill]     *A Technique for the Composition of Music in a Computer* S. Gill, Computer Journal 6, No. 2, July 1963
- [Kemal]   *An Expert System for Harmonizing Chorales in the Style Of J. S. Bach* Kemal Ebcioglu, Journal of Logic Programming 1990:8:145–185
- [Bett]     *Chorale Harmonisation By Computer* Donald Bett, Diploma Project, 1988
- [Morris]   *The Oxford Harmony* R. O. Morris M.A., D.Mus. (Oxon.), F.R.C.M., Oxford University Press, 1946
- [PMS]     *Philip's Musical Scribe* Philip Hazel, 33 Metcalfe Road Cambridge, CB4 2DB. Manuals on `uk.ac.cam.phx` in `PH10.$PMS.SPEC`, Nov 1989
- [Baroni]   *Verso una Grammatica della Melodia* M. Baroni and C. Jacoboni, Universitadina Studi di Bologna, 1976.
- [BSL]     *An Expert System for Harmonizing Four-Part Chorales* Kemal Ebcioglu, Research Report RC12628, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., March 1987
- [Stefik]   *Inferring DNA Structures from Segmentation Data* M. Stefik, Artificial Intelligence 11, 1978
- [ManH]    *Manual of Harmony* T. Otterström, The University of Chicago Press, 1941
- [FigH]    *Figured Harmony at the Keyboard* R. O. Morris M.A., D.Mus. (Oxon.), F.R.C.M., Oxford University Press, 1933
- [BnM]     *Basses and Melodies* R. Dunstan D.Mus. (Cantab.), Novello, Ewer and Co., 1894
- [KnR]     *The C Programming Language, 2nd Edition* B. W. Kernighan and D. M. Ritchie, Prentice Hall, 1989
- [HSay]     *The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty* L. D. Erman et al., Computer Surveys 12, No. 2 (June 1980)

- [Schenk] *Free Composition (Der Freie Satz)* H. Schenker (E. Oster, trans. and ed.), Longman, 1979
- [AHU] *Data Structures and Algorithms* A. V. Aho, J. E. Hopcroft, J. D. Ullman, Addison-Wesley, 1983