Journal of Computer-Based Instruction
Spring 1989, Vol. 16, No. 2, 59-70.
0098-597X/89/16059-$1.00

AN INTEGRATED SOFTWARE SYSTEM FOR SET-THEORETIC AND SERIAL ANALYSIS OF CONTEMPORARY MUSIC

Craig R. Harris
Craig Harris Consulting, Ltd. - San Francisco

Alexander R. Brinkman
Eastman School of Music - University of Rochester

Contemporary Music Analysis Package, or CMAP, is a set of programs for dealing with atonal and serial music. The package includes a comprehensive set of main programs, a compiled data base of pitch-class set data, filters, and a large library of subroutines for bitwise manipulation of pitch-class sets that can be called by user-written programs. The program set is used for analysis, modeling, and composition of contemporary music in an interactive computing environment. The article describes the programs and techniques used to implement them.

Music Analysis and Computers

Any approach to theoretical analysis using computers raises important issues, not the least of which is the degree of control and spontaneity left to the analyst. The programs described here have been designed on the fundamental premise that in music analysis there needs to be a direct and continuous interaction at many levels between the musician, the music, and the analytical data. As an analytical tool the computer resides between these elements and facilitates their interaction.

Throughout the course of a theoretical analysis, the musician needs to investigate numerous hypotheses and make appropriate adjustments in the direction of the analytical approach. This ongoing process requires a very personal mix of the musician's knowledge base, musical instincts, and intuitive responses to the musical data. The computer can participate as an integral part of this relationship only if the operative software has been designed with this in mind, and provides an efficient means for exploring various hypotheses during the analytic process.

One difficulty in the design of a model for music analysis is the tendency for many analyses to deal with specific aspects of music, and not others. An objection to some techniques, such as set-theoretic analysis of music, is that the analysis is primarily concerned with the pitch material of the composition, to the exclusion of other critical parameters such as rhythm, timbre, or register. While this is due in part to the fact that analytical models of pitch are more highly developed, it may also result from the length of time required to thoroughly understand the pitch material alone.

Alexander R. Brinkman, Eastman School of Music, 26 Gibbs Street, Rochester, NY 14604. Telephone, Office: (716) 275-3824; Home: (716) 342-3922.

In the realm of set-theoretic and serial analysis, a well-designed software system can substantially reduce the time factor for the collection and investigation of data, and provide a tool which can enhance the creative and analytical process. Both of these aspects are important. Inordinate time required to perform this kind of analysis without appropriate tools often leads to the result that one either doesn't do it at all, or one doesn't do it to the extent necessary to make the process worth the time invested. Too many compromises result in an incomplete or invalid analysis. The use of computers can speed the process, but speed without flexibility is not sufficient. Appropriate tools allow the analyst to direct the activity and use the tool to assist in the process. If the software is cumbersome or inflexible, then the user will not be able to pursue as many possible interpretations as may seem important, if only to determine their viability.

Software tools that facilitate this kind of analysis make it possible not only to pursue a variety of directions within the realm of pitch material, but to extend the analysis into other areas in order to provide a more comprehensive theoretical perspective. Decisions at all levels have to be made solely on the basis of the musical requisites. Ultimately the usefulness of the system is measured by the degree to which it provides this flexibility.

Contemporary Music Analysis Package, or *CMAP*, is a set of programs for dealing with atonal and serial music. The programs in this software system utilize a well-developed model of pitch that has formed the basis of much theoretical literature dealing with twentieth century music. The pitches in the equally-tempered scale are grouped into equivalence classes based on octave equivalence. All instances of a given pitch, regardless of spelling or register belong to the same *pitch class* (pc). In the

integer model of this system the twelve pcs are represented by the integers 0, 1, 2, ..., 11 with 0 representing C or some other arbitrarily chosen pitch. The interval in semitones between any two pcs, calculated as the difference (*modulo* 12) between the pc integers, is also represented by integers 0 through 11. Frequently it is useful to represent inversionally equivalent intervals by the smaller of the pair; any interval x greater than six can be represented by its *modulo* 12 inverse, $12-x$. Thus, any pc interval can be represented by an *interval class* (ic) 0, 1, 2, ..., 6. For example, the perfect fifth (7 half steps) and the perfect fourth (5 half steps) both belong to ic 5. To minimize keystrokes in entering pitch data and conserve space on output, we use the single characters *a* and *b* to represent pitch classes 10 and 11 (B-flat and B). While an explanation of the theory of pitch-class sets and their relations is beyond the scope of this article, the references at the end will serve as an introduction to this literature, and we have included a glossary of terms that may be unfamiliar to the reader.

The Programs

The primary design objective of this system was to provide efficient and flexible tools for the analysis of contemporary music, with a focus on set-theoretic and serial applications of music theory. These software tools are useful in several contexts. In theoretical modeling, one is able to examine features of set classes or ordered pitch collections, searching for specific properties. For music analysis, the programs can help determine structural elements of the pitch material from actual compositions. A composer can use the programs to explore properties of prospective compositional material in the form of either abstract set classes or specific pitch classes.

The programs are written in the C programming language and were implemented initially for the UNIX† operating system environment. Flexibility is achieved through modular design, facilitating integration with the other programs in the set, operating system utilities, and user-written programs. Most programs are command-driven, and the user controls the flow of information between the database and program modules. One can design *ad hoc* shell scripts that manipulate data from files, terminal input, or other programs. While the programs are independent as they stand, additional features and program modules can be implemented and integrated with the system.

The programs fall into four categories: Library procedures, main programs, shell scripts, and filters. The flow diagram in Figure 1 shows the relationship between these components, the set-class data base, and optional user-written programs. The CMAP Function Library consists of subroutines that are called by CMAP main programs

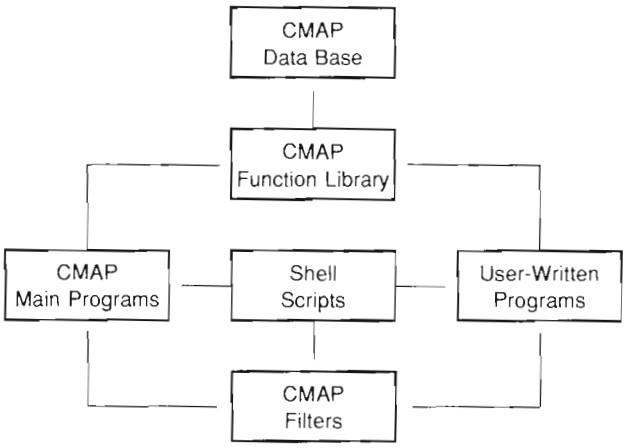

Figure 1. The CMAP system.

and are available to user-written programs as well. The library includes functions for efficient searching of the data base. The output from main programs can be filtered by programs that extract specific data and then be used as input to other main programs or "Shell Scripts" written in the command language of the operating system. Shell scripts can also invoke main programs, filters, and other shell scripts.

The CMAP library consists of C functions that perform operations such as transposition, inversion, deriving the set cardinality, calculating normal order, checking for subset inclusion, etc. The library contains over fifty functions that are called by the main programs and are

Table 1. CMAP Main Programs.

complement ...	calculate the complement of a set
getset	identify pc-sets
imatrix	construct invariance matrices
intersect	find the intersection between two or more sets
kh	generate K/Kh tables
matrix	construct a matrix
partition	partition a pitch collection into all possible n-note combinations
pccount	display pitch class count
pcmap	display pitch class mapping
prime	iterative set identification
printtab	print set-class table
rotarray	construct a rotational array matrix
rowcomb	test combinatorial properties of row or rows
rowop	generate specified row form
rowstruct	calculate imbricated subsets of row
search	search matrices for sets
setdif	find all elements of one set that are not in the other
setmap	show operations (Tn, TnI, TnM, TnMI) under which a set maps into itself or its complement
subset	examine supersets for appearance of subsets
symdif	calculate the symmetric difference between pc sets
transpose	transpose pitch classes sets [Tn(X), TnI(X), TnM(X), TnMI(X)]
union	find the union of two or more pc sets

available to programs designed by the user.

Main programs invoke library functions and control I/O (input/output) operations. These programs accept user requests to retrieve a specific set class or cardinality class, for example, and print the results. The primary programs in the system are listed in Table 1.

Shell scripts can be designed for various purposes, making calls to the operating system and C Programs as needed. The user can investigate multiple properties of a set or groups of sets, retrieving the information required for a specific application. Figure 2 shows an elementary shell script that automates testing for set classes that occur as subsets in their complements. This program is called with two arguments, the cardinality of the sets to be examined and the highest ordinal number in the list of sets of that size. For example:

% setcomp 3 6

examines subset relations between each of the six three-note sets and its complementary nine-note set. The script assigns the value 1 to the variable *i*, and 9 (the *modulo* 12 complement of 3) to *x*. The *while* loop then calls the CMAP program *subset* to calculate subset relations between each complementary pair of sets: 3-1 and 9-1; 3-2 and 9-2; 3-3 and 9-3; ... 3-6 and 9-6. (See the definition of *set names* in the glossary at the end of this article.)

```
#  C-Shell Script to test for subset relations
#  between set and complement.
#
#  Usage: setcomp $1 $2
#
#  Parameters:  $1: cardinality
#               $2: highest ordinal number for set
#
@ i = 1
@ x = (12 - $1)
while ($i <= $2)
echo "Checking in set class $x-$i for subset $1-$i"
echo " "
subset $x-$i $1-$i
echo " "
@ i += 1
end
```

Figure 2. Example of shell script.

Filters are programs that extract specific information from the output from other programs. Our filters:

cint – extract the adjacent interval pattern
ic – extract the interval class vector
inv – extract the invariance vector
no – extract the normal order
m – extract the M-related set class
z – extract the Z-set class

We have adopted a convention for differentiating set characteristics in the program output. By identifying different components in the output from main programs, these symbols facilitate parsing the data with text-scanning facilities available in the operating system. These conventions are shown below:

{ } Curly brackets surround the listings of unordered pitch class elements.
( ) Parentheses surround the invariance vector.
[ ] Square brackets are used for the interval class vector.
< > These brackets enclose the adjacent interval array.
z The letter *z* is used in front of the Z-related set class. If no Z-set exists, a *0* will follow.
m The letter *m* is used in front of the M-related set class.

We also use letters to identify the following:

Operators		*Relationships*	
p	Prime form	**e**	Equal to
i	Inversion	**l**	Less than
m	M operation	**g**	Greater than
w	MI operation	**rp**	Rp-relation
		i	Invariant
		c	Complement

Using the Programs

While it is impossible to illustrate all of the programs in this article, a few examples will illustrate the manner in which they are invoked and the types of information that can be obtained. The program package was designed to be both flexible and easy to use, and both novice users and experienced programmers will benefit from its use. However, the user should be familiar with some fundamental principles. We assume a basic knowledge of operating system utilities for file and data manipulation. With this limited background, the design and usage of the programs should be easy to understand. With more advanced knowledge of the "pipe" (using the output from one program directly as input to another), input and output redirection, shell command design, and macro capabilities of the operating system in use, the power of this system is extended greatly.

The design of commands and argument passing follow the UNIX/C model. Programs are called by typing the name of the program followed by arguments to the programs. The command line is parsed by the program, and the appropriate functions are executed. The examples below illustrate usage.

Figure 3 illustrates the use of program *getset*. In the first example *getset* is passed as an argument '294675b', a series of pitch classes in hexadecimal notation. *Getset*

parses the command line, reduces the argument to it "prime form," and locates this set class in a data table. It then formats and prints pertinent data. In the second example several arguments are passed to *getset* in different formats. Each argument is examined to determine if it is a set-class name as in the first two arguments or a pitch-class collection as in the third. The output of the program is in the same order as the arguments on the command line.

% getset 294675b

NAME	Z-SET	M-SET	IC VECTOR	INV VECTOR	ORDER	CYCLE
7-23	z0	m 2	[354351]	(10000000)	{0234579}	<2111223>

% getset 4-23 5-17 19467352a2

NAME	Z-SET	M-SET	IC VECTOR	INV VECTOR	ORDER	CYCLE
4-23	z0	m 1	[021030]	(11005511)	{0257}	<2325>
5-17	z37	m37	[212320]	(11001122)	{01348}	<12144>
9-3	z0	m11	[767763]	(10000000)	{012345689}	<111111213>

Figure 3. Calling program *getset*.

Some of the programs require specific argument structure, with each argument designating different meanings for that program. The program *rowstruct*, which takes as input a series of ordered pitch classes followed by an integer representing a sub-cardinality for the analysis of the imbricated sets, is one of these programs. Figure 4 demonstrates the principle.

% rowstruct 09ab78215436 3

```
***  Row:  [09ab78215436]  ***
Imbricated 3-note sets
  Rotation 0:  Subrow: (09a)  Set Class 3-2:  {013}
  Rotation 1:  Subrow: (9ab)  Set Class 3-1:  {012}
  Rotation 2:  Subrow: (ab7)  Set Class 3-3:  {014}
  Rotation 3:  Subrow: (b78)  Set Class 3-3:  {014}
  Rotation 4:  Subrow: (782)  Set Class 3-5:  {016}
  Rotation 5:  Subrow: (821)  Set Class 3-5:  {016}
  Rotation 6:  Subrow: (215)  Set Class 3-3:  {014}
  Rotation 7:  Subrow: (154)  Set Class 3-3:  {014}
  Rotation 8:  Subrow: (543)  Set Class 3-1:  {012}
  Rotation 9:  Subrow: (436)  Set Class 3-2:  {013}
***wrap-around***
  Rotation 10:  Subrow: (360)  Set Class 3-10:  {036}
  Rotation 11:  Subrow: (609)  Set Class 3-10:  {036}
```

Figure 4. Calling program *rowstruct*.

Many of the programs accept a series of command-line options, which designate specifications for searches and output format. These options are always preceded by a dash (–), which signals that the following character or characters are to be interpreted as a program option. Figure 5 illustrates some of the options available with program *getset*. The first example requests that the program show all three-note sets:

% getset –s 3

This particular program allows requests for specific information to be printed. For example, the user might request only columns 1, 4, and 6 of the above example: the set name (implicit), the interval-class vector (i), and the normal order (o) for all three-note set classes. This could be accomplished by

% getset –sio 3

or

% getset –s –i –o 3

The invariance-vector-search option for *getset* allows the user to make more specific inquiries. This option requires the input of a series of four characters, each of which represents information needed to satisfy the search routine. The command line appears as follows:

% getset –s –xcig6 3

The *–s* option for the cardinality search remains the same, but the *x* option signifies to the program that the next four characters will represent the request to search for self or complement mapping, the operator to view, the relation, and the frequency for the context search. In this case the program will examine all sets of cardinality 3 and print only those sets that satisfy the specifications of mapping into the complement under inversion with a frequency greater than 6.

% getset –s 3

NAME	Z-SET	M-SET	IC VECTOR	INV VECTOR	ORDER	CYCLE
3-1	z0	m 9	[210000]	(11007744)	{012}	<11a>
3-2	z0	m 7	[111000]	(10005655)	{013}	<129>
3-3	z0	m11	[101100]	(10005655)	{014}	<138>
3-4	z0	m 4	[100110]	(10105656)	{015}	<147>
3-5	z0	m 5	[100011]	(10016776)	{016}	<156>
3-6	z0	m 6	[020100]	(11117777)	{024}	<228>
3-7	z0	m 2	[011010]	(10005655)	{025}	<237>
3-8	z0	m 8	[010101]	(10016776)	{026}	<246>
3-9	z0	m 1	[010020]	(11007744)	{027}	<255>
3-10	z0	m10	[002001]	(11118888)	{036}	<336>
3-11	z0	m 3	[001110]	(10005655)	{037}	<345>
3-12	z0	m12	[000300]	(33339999)	{048}	<444>

% getset –s –xcig6 3

NAME	Z-SET	M-SET	IC VECTOR	INV VECTOR	ORDER	CYCLE
3-1	z0	m 9	[210000]	(11007744)	{012}	<11a>
3-5	z0	m 5	[100011]	(10016776)	{016}	<156>
3-6	z0	m 6	[020100]	(11117777)	{024}	<228>
3-8	z0	m 8	[010101]	(10016776)	{026}	<246>
3-9	z0	m 1	[010020]	(11007744)	{027}	<255>
3-10	z0	m10	[002001]	(11118888)	{036}	<336>
3-12	z0	m12	[000300]	(33339999)	{048}	<444>

Figure 5. Some other options with *getset*.

Knowledge of input/output pipe capabilities and of shell programming enable the user to take advantage of the full power of the CMAP programs. While this capability is somewhat dependent on the manner in which individual programs interpret command lines, it is possible to manipulate the data in many different ways. Figure 6 illustrates the principle, performing an analysis of the input set class 4–12. First, *setmap* is invoked to find common-tone relationships among the transpositions of the specified set class. The output from *setmap* is piped into the UNIX text searching utility *grep*, which filters the output for only those elements that are Rp-related (having one fewer than the cardinality of the original set class in common). The output from *grep* is processed again by a set-class filter that suppresses everything on each line except for the pitch classes. This filtered output is stored in a disk file. This file is used as input to *getset*, which examines these subsets for specific interval content and

```
% setmap 4-12 | grep rp | no > newfile
% getset -t3g0 -xcmg2 'cat newfile'
```

[**setmap 4-12** generates the following]

SET CLASS 4-12 {0236} INVARIANCE VECTOR: [10002432]
SET MAP RELATIONSHIPS FOR PRIME

T	CMN	MAP	SC	PC
t0p:	4	i	4-12	{0236}
t1p:	1		1-1	{0}
t2p:	1		1-1	{0}
t3p:	2		2-3	{03}
t4p:	1		1-1	{0}
t5p:	0	c		
t6p:	2		2-6	{06}
t7p:	0	c		
t8p:	1		1-1	{0}
t9p:	2		2-3	{03}
tap:	1		1-1	{0}
tbp:	1		1-1	{0}

[similar output for TnI, TnM, and TnMI omitted]

[| **grep rp** extracts rp related sets]

t6i:	3	rp	3-10	{036}
t0m:	3	rp	3-10	{036}
t0w:	3	rp	3-8	{026}
t6w:	3	rp	3-10	{036}

[| **no > newfile** filters out normal orders and places them in newfile:]

```
036
036
026
036
```

getset -t3g0 -xcmg2 'cat newfile'

NAME	Z-SET	M-SET	IC VECTOR	INV VECTOR	ORDER	CYCLE
3-10	z0	m10	[002001]	(11118888)	{036}	<336>

Figure 6. Using pipes and redirection.

invariance relationships, and prints only those sets that have the characteristics requested on the command line. In this case the search is for those set classes that have at least one occurrence of interval class 3, and map into their complement under the M-relation more than two times.

The CMAP program set includes a few programs that are self-contained and are not designed to work with the system of filters and pipes. Some of these programs, which loop until the user requests termination of the program run, provide a shell escape mechanism so that other programs can be run without leaving the program. An example is program *search*, an interactive screen-oriented matrix searching program. The output of *search* is an x/y matrix based on the specified row or row segment, or pair of rows or segments. The matrix is displayed on the screen with order and transposition numbers. The user specifies pitch collections or individual pitch classes, and the locations of contiguous occurrences of the pcs in the collection are highlighted on the displayed matrix. The program matches *unordered* segments in the matrix, i.e., the ordering of the pcs is not significant to the matching process. Alternately the user may specify that matching pcs be shown wherever they occur in the matrix, whether or not they are contiguous. The user can also specify the type of matrix (see glossary), and whether the row[s] are to be normalized by transposing them to zero before constructing the matrix. In the default mode, the program is useful as an aid to analyzing serial compositions, since the user can quickly determine all possible derivations of any pitch collection in the matrix. Since all of the common matrix types can be specified, the program also serves as an interactive invariance matrix program, and thus is useful in determining invariances between any row or segment and various twelve-tone transformations of another row or segment. Figure 7 shows that the pitch collection (03ab) occurs in the matrix as T_6P (ordinal 7-a) and T_bI (0-3).

```
% search 1029384756ab
```

T	0	b	1	8	2	7	3	6	4	5	9	a	
0	0	**b**	1	8	2	7	3	6	4	5	9	a	0
1	1	**0**	2	9	3	8	4	7	5	6	a	b	1
b	b	**a**	0	7	1	6	2	5	3	4	8	9	2
4	4	**3**	5	0	6	b	7	a	8	9	1	2	3
a	a	9	b	6	0	5	1	4	2	3	7	8	4
5	5	4	6	1	7	0	8	b	9	a	2	3	5
9	9	8	a	5	b	4	0	3	1	2	6	7	6
6	6	5	7	2	8	1	9	**0**	**a**	**b**	**3**	4	7
8	8	7	9	4	a	3	b	2	0	1	5	6	8
7	7	6	8	3	9	2	a	1	b	0	4	5	9
3	3	2	4	b	5	a	6	9	7	8	0	1	a
2	2	1	3	a	4	9	5	8	6	7	b	0	b
	0	1	2	3	4	5	6	7	8	9	a	b	

type set: **03ab**

Figure 7. Program *search*.

Program *kh* (Figure 8) accepts a group of set names and prints a table showing the K and KH relations among the group of sets (Forte, 1973). Thus this program shows the superset/subset relations among a group of sets. Options provide different formats for the printed table. The *–v* option results in a "vertical" format that makes it possible to display large tables that would not ordinarily fit on the computer screen. [This table replicates Figure 117 on p. 115 of Forte (1973), in a different format.]

```
% kh -v 3-1 3-3 3-4 3-5 3-9 4-8 4-9 4-15 8-15 4-18 8-28 7-4 5-6\
       7-6 5-7 5-19 6-6 6-38 6-13

                          8-15      7-6            6-38
          4-8  4-9  4-15 4-18 5-6  5-7  5-19 6-6  6-13 7-4  8-28
          ================================================
3-1       | K  | K  | K  | K  | Kh | Kh | K  | Kh | K* | Kh |    |
3-3       | K  | K  | Kh | Kh | Kh | K  | Kh | K  | Kh | Kh | K  |
3-4       | Kh | K  | K  | K  | Kh | Kh | K  | Kh | K* | Kh |    |
3-5       | Kh | Kh | Kh | Kh | Kh | Kh | Kh | Kh | Kh | Kh | K  |
3-9       | K  | K  | K  | K  | K  | Kh | K  | Kh |    | K  |    |
4-8       |    |    |    |    | Kh | Kh | K  | Kh |    | K  |    |
4-9       |    |    |    |    | K  | Kh | Kh | Kh | K* | K  |    |
4-15/8-15 |    |    |    |    | Kh | K  | Kh | K  | K* | K  |    |
4-18      |    |    |    |    | K  | K  | Kh |    | Kh | K  |    |
5-6 /7-6  | Kh | K  | Kh | K  |    |    |    | K  |    |    |    |
5-7       | Kh | Kh | K  | K  |    |    |    | Kh |    |    |    |
5-19      | K  | Kh | Kh | Kh |    |    |    |    | K* |    | K  |
6-6 /6-38 | Kh | Kh | K  |    | K  | Kh |    |    |    |    |    |
6-13      |    | K* | K* | Kh |    |    | K* |    |    | K* | K* |
7-4       | K  | K  | K  | K  |    |    |    |    | K* |    |    |
          ================================================
          4-8  4-9  4-15 4-18 5-6  5-7  5-19 6-6  6-13 7-4  8-28
                    8-15      7-6            6-38
```

Figure 8. Program *kh*.

Finally, the user can obtain a listing of the entire set table by invoking program *printtab* (Figure 9). Since two prime form algorithms are in common use, all programs are designed to accommodate either type. The user sets an environment variable called PRIMETYPE to *forte* or *rahn*, and thereafter all prime forms are of the desired type. The programs are designed to print a summary of the command-line syntax and options when the user types the program name with no arguments. Figure 10 illustrates the "usage statement" for one of the programs discussed above.

We will now examine aspects of the implementation of the programs, including the representation of sets as unsigned integers and data structures used for the set-class data base and for some of the programs in the set. Each implementation detail was designed for efficiency and flexibility.

```
% printtab

1-1   {0}    z 0   m 1    [000000]   0   (1111bbbb)   <c>

2-1   {01}   z 0   m 5    [100000]  40   (11009988)   <1b>
2-2   {02}   z 0   m 2    [010000]  20   (11119999)   <2a>
2-3   {03}   z 0   m 3    [001000]  10   (11119999)   <39>
2-4   {04}   z 0   m 4    [000100]   4   (11119999)   <48>
2-5   {05}   z 0   m 1    [000010]   2   (11009988)   <57>
2-6   {06}   z 0   m 6    [000001]   1   (2222aaaa)   <66>

3-1   {012}  z 0   m 9    [210000]  60   (11007744)   <11a>
3-2   {013}  z 0   m 7    [111000]  70   (10005655)   <129>
3-3   {014}  z 0   m11    [101100]  54   (10005655)   <138>
3-4   {015}  z 0   m 4    [100110]  46   (10105656)   <147>
3-5   {016}  z 0   m 5    [100011]  43   (10016776)   <156>
3-6   {024}  z 0   m 6    [020100]  24   (11117777)   <228>
3-7   {025}  z 0   m 2    [011010]  32   (10005655)   <237>
3-8   {026}  z 0   m 8    [010101]  25   (10016776)   <246>
3-9   {027}  z 0   m 1    [010020]  22   (11007744)   <255>
3-10  {036}  z 0   m10    [002001]  11   (11118888)   <336>
3-11  {037}  z 0   m 3    [001110]  16   (10005655)   <345>
3-12  {048}  z 0   m12    [000300]   4   (33339999)   <444>

4-1   {0123} z 0   m23    [321000]  70   (11005511)   <1119>
4-2   {0124} z 0   m22    [221100]  74   (10003411)   <1128>
4-3   {0134} z 0   m26    [212100]  74   (11003322)   <1218>
4-4   {0125} z 0   m14    [211110]  76   (10001323)   <1137>

                                                      etc. ...
```

Figure 9. Program *printtab*.

```
% search

                        SEARCH

Usage: search [ –pimwd ] [ –tn ] [ –sc ] row1 [ row2 ]

Options:
    p  - construct p-matrix
    i  - construct i-matrix
    m  - construct m-matrix
    w  - construct mi-matrix
    d  - default for single row: p-matrix with trans numbers
    t  - do not transpose rows to 0 (default for pimw-matrix)
    n  - normalize rows to 0 (default for d-matrix)
    s  - mode: match separate elements
    c  - mode: match contiguous elements (default)

during execution:
    type '&' to reset match mode
    type '$' to change matrix type
    type '*' to reset alternate chars. for 10 and 11
    type '?' or 'help' for help
    type '!' for shell escape
    type 'quit' to terminate
```

Figure 10. Usage statement for program *search*.

Bitwise Manipulation of Sets

We use a numeric representation of pitch-class sets. Each set is represented as an unsigned binary integer with the 2^i bit representing pc i. Thus, the rightmost bit (positional weight $2^0 = 1$) represents pc 0, the next bit (positional weight $2^1 = 2$) represents pc 1, and so on. The value of the bit signifies the presence (1) or absence (0) of the pitch class. Thus the pc set {025} is represented by the binary integer shown below:

```
000000100101
           |____ 0        [C]
         |______ 2        [D]
     |________ 5          [F]
```

We call this integer, which can also be represented by decimal 37 or octal 045, a *set-number*. Each of the 4096 possible pitch-class sets is represented by a unique set-number. This figure includes the null set (000000000000 = 0) and the twelve-tone set (111111111111 = 4095). This representation has two advantages. First, most twelve tone operations can be performed on sets rather than on elements, via bitwise operations. Second, the numeric representation facilitates design of efficient algorithms for searching the data base or program-specific data structures as explained below.

Set-numbers are particularly convenient for use with the C programming language since C provides primitives for bitwise operations. Although these operations can be simulated arithmetically, the bitwise operators are more efficient since they translate directly to machine level computer instructions.

&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

We combine these operations with appropriate bit masks, explained below, to implement the necessary twelve-tone operations. In each case, the high-order bits are set to zero, since we use only the twelve low-order bits. Each twelve-tone operation can be implemented as a combination of bitwise operations. A few examples will suffice to illustrate the technique.

The complement of a set is obtained by taking the set's one's complement, thus setting all of the 0 bits to 1 and vice versa:

S = 000000100101 {0,2,5}

$\overline{S}$ = ~000000100101
= 111111011010 {1,3,5,6,7,8,9,a,b}

In C this is accomplished by one's complement followed by masking out the high-order bits as shown in the function complement (Figure 11).

```
unsigned complement (x)          /* return complement of x */
unsigned x;
{
        return (~x & 07777);
}
```

Figure 11. Function *complement*.

A set is transposed one half step by shifting all bits to the left (with bit 12 rotating to position 0). We will refer to this operation as left-rotation.

$T_0(S)$ = 000000100101 {0,2,5}
$T_1(S)$ = 000001001010 {1,3,6}
$T_2(S)$ = 000010010100 {2,4,7}
$T_3(S)$ = 000100101000 {3,5,8}
. . .
$T_6(S)$ = 100101000000 {6,8,b}
$T_7(S)$ = 001010000001 {0,7,9}
etc.

This is implemented as the C function *rotate* (Figure 12). Note that bit masks are coded as octal numbers in C, thus the mask 07777 in the return statement represents the binary number with the twelve low-order bits on. Higher-level operations are built using these bitwise operations. For example function *transpose* (Figure 13) uses *rotate* to calculate $T_n(X)$, i.e., it transposes set X by n half steps.

```
unsigned rotate (x)              /* left-rotate first 12 bits */
unsigned x;
{
        x <<= 1;                 /* shift left          */
        if (x & 010000)          /* if 12 bit is on     */
                x |= 001;        /* turn on 0 bit       */
        return (x & 07777);      /* return bits 0-11    */
}
```

Figure 12. Function *rotate*.

```
unsigned transpose (x,n)         /* transpose by rotation */
unsigned x;                      /* set number            */
int n;                           /* transposition level   */
{
        int i;
        unsigned rotate ();

        for (i = 1; i <= n; i++)
                x = rotate (x);
        return (x);
}
```

Figure 13. Function *transpose*.

Inversion is obtained by mapping each bit i into bit $(12-i)$ *mod* 12, i.e., for each 1 in position i, set bit i to 0 and bit $(12-i \bmod 12)$ to 1. Note that under this operation bit 0 maps into itself:

$T_0(S)$ = 000000100101 {0,2,5}

$S' = T_0I(S)$ = 010010000001 {0,7,a}

A function to perform this operation is shown in Figure 14.

```
unsigned inverse (x)                    /*  return inverse of x    */
unsigned x;                             /*  where x is a setnum    */
{
        unsigned t, rotate ();
        int i;
        t = 0;

        for (i = 1; i <= 12; i++) {
                t |= (x & 001);         /*  set bit in t           */
                x = x >> 1;             /*  right-shift x          */
                t = rotate (t);         /*  left-rotate t          */
        }
        return (t);                     /*  return inverse         */
}
```

Figure 14. Function *inverse*.

Masking is used to set or test arbitrary bits in a binary number. For example, to set bit 3 in set-number S, we use a mask with bit 3 set to 1. The result of the expression S | MASK (inclusive OR) sets the bit.

S:		010001000011	{0,1,6,a}
MASK:	OR	000000001000	{3}
		010001001011	{0,1,3,6,a}

The bitwise AND operation is used to check for subset inclusion. For example, if we wish to see if set {0,1,4} occurs as a subset of another set *S*, we create a mask that represents {0,1,4}. If *S* & MASK (bitwise AND) equals the mask, then {0,1,4} occurs in the set.

S:		010001010111	{0,1,2,4,6,a}
MASK:	AND	000000010011	{0,1,4}
		000000010011	{0,1,4}

By repeating the operation after successive rotations of the mask, we can check for inclusion of each transposition of {014}. The inversion of the mask is used to check for inclusion of the inversion of the set. Using techniques such as this, we can quickly find all twelve-tone operators (TTOs) that map a set into itself, its complement, or into another set. If we merely want to know if one set is a subset of another, the set and its inverse are each rotated until a match is found or until all possibilities have been exhausted.

Most operations on pc sets are simplified by using techniques such as these. For example, the prime form of a set is found by taking a set *S* and its inverse *S'*. Each is rotated twelve times and the form that yields the least value (as a binary integer) represents the prime form. This prime-form algorithm was described by Starr (1978). A C function that implements this algorithm is shown in Figure 15.

```
unsigned primeform (pset)  /* returns prime form of set-number */
unsigned pset;
{
        int i;
        unsigned iset, least, inverse (), rotate ();

        least = pset;
        iset = inverse (pset);   /* inverse form for comparison */

        for (i=1; i <= 12; i++) {
                pset = rotate (pset);   /*  get T1 (pset)         */
                iset = rotate (iset);   /*  get T1 (iset)         */

                if (pset < least)       /*  save least value      */
                        least = pset;
                if (iset < least)
                        least = iset;
        }
        return (least);                 /*  return prime form     */
}
```

Figure 15. Function *primeform*.

Data Structures

A central feature of our programs is that set-class data are retained in memory in a data structure ordered according to the set-number representing the prime form of each set class. This design minimizes search time, and makes it possible to utilize bitwise operations for many processes, facilitating examination and comparisons between sets.

The set table, which is implemented as an array of structures, consists of two parts; one contains the data for the individual set classes and the other specifies the location of cardinality classes in the larger table (see Figure 16). The table includes, from left to right, a comment showing the position or index of each structure in the list, the index of the next element in order by set name, the set name as a character string, the set-number representing the prime form of the set, an integer representing the prime-form type (Forte, Rahn, or both), the ordinal number of the M and Z related set class, the interval-class vector, a bit map of the interval vector, the invariance vector, and the circular interval pattern. The octal map of the interval vector, which shows which interval classes occur in the vector but not their frequency, is useful when looking for sets with specific interval content, since this task is again reduced to a masking operation.

The table is arranged in numerical order by set-number, with all sets of the same cardinality contiguous. The column labeled "next" indicates the next set-class in the list ordered by set name. This field is used when it is desirable to traverse the list or part of it in the customary order, as when printing the entire table, or listing sets of a specified cardinality. Because of the various orderings explicit in the table, we can use binary search when

```
#include "settab.h"
struct tabinfo loc [] = {
      -1,   -1,  -1,     /*  0 */
       0,    0,   1,     /*  1 */   /* location of each cardinality */
       1,    6,   6,     /*  2 */   /* class in table               */
       7,   18,  12,     /*  3 */
      19,   47,  29,     /*  4 */
      48,   86,  38,     /*  5 */
      87,  138,  50,     /*  6 */
     139,  178,  38,     /*  7 */
     179,  208,  29,     /*  8 */
     209,  220,  12,     /*  9 */
     221,  226,   6,     /* 10 */
     227,  227,   1,     /* 11 */
     228,  228,   1      /* 12 */
};

struct set settab [] = {
/* index next name  setnum pft  M  Z  ic-vector    map  invarince     cint  */
/*  0*/  1, "1-1",    1,  0,  1,  0, "1000000",    0, "1111bbbb",    "c",
/*  1*/  2, "2-1",    3,  0,  5,  0, "2100000",   32, "11009988",    "1b",
/*  2*/  3, "2-2",    5,  0,  2,  0, "2010000",   16, "11119999",    "2a",
/*  3*/  4, "2-3",    9,  0,  3,  0, "2001000",    8, "11119999",    "39",
/*  4*/  5, "2-4",   17,  0,  4,  0, "2000100",    4, "11119999",    "48",
/*  5*/  6, "2-5",   33,  0,  1,  0, "2000010",    2, "11009988",    "57",
/*  6*/  7, "2-6",   65,  0,  6,  0, "2000001",    1, "2222aaaa",    "66",
/*  7*/  8, "3-1",    7,  0,  9,  0, "3210000",   48, "11007744",    "11a",
/*  8*/  9, "3-2",   11,  0,  7,  0, "3111000",   56, "10005655",    "129",
/*  9*/ 11, "3-3",   19,  0, 11,  0, "3101100",   44, "10005655",    "138",
/* 10*/ 12, "3-6",   21,  0,  6,  0, "3020100",   20, "11117777",    "228",
/* 11*/ 13, "3-4",   35,  0,  4,  0, "3100110",   38, "10105656",    "147",
/* 12*/ 14, "3-7",   37,  0,  2,  0, "3011010",   26, "10005655",    "237",
/* 13*/ 10, "3-5",   67,  0,  5,  0, "3100011",   35, "10016776",    "156",
/* 14*/ 16, "3-8",   69,  0,  8,  0, "3010101",   21, "10016776",    "246",
/* 15*/ 17, "3-10",  73,  0, 10,  0, "3002001",    9, "11118888",    "336",
/* 16*/ 15, "3-9",  133,  0,  1,  0, "3010020",   18, "11007744",    "255",
/* 17*/ 18, "3-11", 137,  0,  3,  0, "3001110",   14, "10005655",    "345",
/* 18*/ 19, "3-12", 273,  0, 12,  0, "3000300",    4, "33339999",    "444",
/* 19*/ 20, "4-1",   15,  0, 23,  0, "4321000",   56, "11005511",    "1119",
/* 20*/ 21, "4-2",   23,  0, 22,  0, "4221100",   60, "10003411",    "1128",
/* 21*/ 22, "4-3",   27,  0, 26,  0, "4212100",   60, "11003322",    "1218",
/* 22*/ 26, "4-4",   39,  0, 14,  0, "4211110",   62, "10001323",    "1137",
/* 23*/ 28, "4-11",  43,  0, 11,  0, "4121110",   62, "10101313",    "1227",
/* 24*/ 23, "4-10",  45,  0, 10,  0, "4122010",   58, "11113333",    "2127",
/* 25*/ 31, "4-7",   51,  0, 20,  0, "4201210",   46, "11003333",    "1317",
/* 26*/ 32, "4-5",   71,  0, 16,  0, "4210111",   55, "10002432",    "1146",
etc. . . .
```

Figure 16. The set-class table.

locating sets, or linear search when addressing sets by name or cardinality. In either case the length of the list is limited to the number of set classes of a given cardinality. Binary search consists of looking first in the middle of an ordered list. If the middle element is not the desired one, half of the list can be eliminated since it is either less than or greater than the target element, and the process is repeated using the new, shorter list. Using this searching method recursively, any set class can be located in at most six tries, since the largest cardinality class, six-note sets, contains 50 elements.

With the exception of the set-number (giving the prime form) and the set name, all of the data in the table were generated by functions in the CMAP Function Library. This approach has obvious advantages in terms of data entry and accuracy. The table was compiled separately, much like a function, and entered into the library. Thus, any function or main program that needs access to the data in the table can be linked to the table, as it is to other functions.

Table searching procedures return pointers to the appropriate record in the table, or a null pointer if the search fails. [A pointer is the address of an object in the computer memory.] This facilitates building data structures for implementing various algorithms. An example is the data structure used in program *kh*, which calculates K/Kh tables (Figure 17). This structure is an array of pointers to data nodes. Each data node contains, among other things, the set-name encoded as an integer and pointers to the records in the set table representing the prime form of the set and the prime form of its complement. As in most other programs, sets can be specified by name or by a pitch collection, which may or may not be the prime form of a set. Our command line interpreter extracts the set name or pcs. If pcs are specified, they are placed in prime form and the set is located in the table by a binary search on the set-numbers. If a set name is specified the location is found by linear search.

Once this structure has been built, the array of pointers is sorted on the set name, and duplicates are eliminated. During the reshuffling process, pointers to data nodes are moved rather than the data itself. Finally, subset relationships between sets are calculated using techniques described earlier.

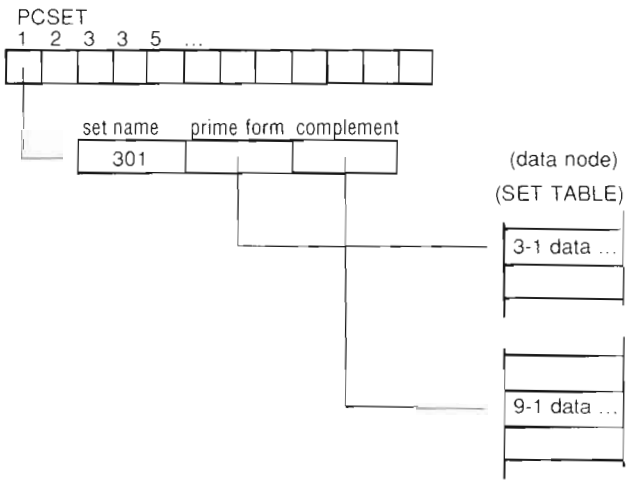

Figure 17. Data structure for program *kh*.

The data structure used for program *search*, which interactively locates and displays unordered sets in a matrix, provides another example of efficient searching based on the binary representation of sets. This data structure is a binary search tree ordered by set-numbers (S in Figure 18). The tree is built and searched recursively. Set-number X is found by following the rule: if X is less than S, search the left subtree, if it is greater, search the right subtree. At each juncture approximately half of the remaining list is eliminated. Thus the search tree has the same advantage as using binary search with numerically ordered lists. Separate trees are used for each cardinality class and trees are constructed once, when the subsets of the matrix are calculated. Since the algorithm for building the tree is essentially the same as for searching it, this data structure is an efficient method of storing and finding each unordered segment of the matrix.

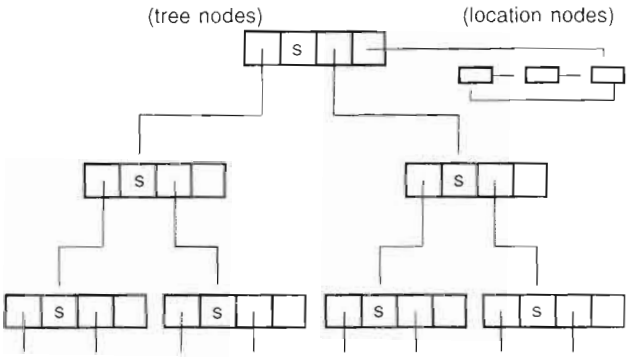

Figure 18. Data structure for program *search*.

Conclusion

The combined use of bitwise operations, in-memory storage of the database, and fast algorithms to search the data tables optimize program efficiency, while independent program modules, careful design of program output, and the use of output filters increase utility and flexibility.

Using this program set, the analyst is able to use CMAP Programs and operating system utilities to investigate properties of sets and the structure of tone rows, and to access information in the database. In addition, the more experienced programmer can utilize the CMAP Function Library and any of the standard C libraries in developing programs designed for more specific requirements.

When combined with a powerful shell programming language such as the C–Shell, the ability to control the flow of data through manipulation of input and output is further enhanced. The availability of job control, command histories, and command-line substitution allow the user to mold the programs around one's general interests in the analysis or composition of music, as well as the specific requirements of the moment.

Contemporary Music Analysis Package (CMAP) has been published by the authors, and is available to users under software license. The UNIX† version has been tested at several universities under a beta-test agreement, and the music faculty at University of California, Santa Barbara is developing a course around the software system. The MSDOS† version of the programs for IBM PC's and compatibles is currently available, and a MACINTOSH† version is in preparation.

Glossary

The following glossary is intended as an aid to the reader who is relatively unfamiliar with the technical terminology used in this paper. It is not intended to be comprehensive. The reader who is interested in the application of these concepts is directed to the works cited in the references listed below. Rahn (1980) contains extensive bibliographies. The bibliography in Morris (1987) is more selective but more current.

aggregate — The set of all twelve pitch classes taken together.

cardinal number — The number of elements in a set.

cardinality class — The set of all pc set classes of the same cardinality.

circular interval pattern — The array of intervals between the pitch classes in the prime form of a set. The pattern includes the interval from the last pc to the first. For example the circular interval pattern for the set {0,1,4} is 1 - 3 - 8. The final interval is from 4 to 12 (0).

circular permutation — The permutation of an ordered set obtained by placing the first element at the end. The first circular permutation of <1,5,2,a> is <5,2,a,1>.

combinatoriality — A property of a twelve-tone row that makes it possible to state two row forms simultaneously such that secondary aggregates result. For example, if a row is hexachordally combinatorial, there are two (or more) row forms for which the first six notes of each row combine to form an aggregate, as do the last six notes of each row.

complement — the literal complement of a pc set consists of the set of pitch classes that are *not* in the original set. For example the literal complement of {1,4,5} is {0,2,3,6,7,8,9,a,b}. The term is also used abstractly to refer to complementary set classes, i.e., the prime form of the literal complement of another prime form.

imbricated subsets — Segments of an ordered set. The three-note imbricated subsets of <0,2,5,7,a,b> are <0,2,5>, <2,5,7>, <5,7,a>, and <7,a,b>.

inclusion — A set may be literally included in another, when all pcs in one set occur also in the other, or abstractly when the relation holds for any transposition or transposition after inversion of one of the sets.

intersection — The elements that are common to two or more sets. The intersection of the sets {0,1,5,6} and {1,4,5} is {1,5}.

interval — The distance in half steps between two pitches. The interval between two pitch class integers A and B is calculated as B A (modulo 12). There are 12 pitch-class intervals, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, where a=10 and b=11.

interval class (ic) — A pair of inverse-related intervals, e.g. 1 and b, 2 and a, 3 and 9. The interval class is represented by the smaller of the two intervals. There are seven interval classes, 0,1,2,3,4,5,6.

Interval-class vector — An ordered array that indicates the total interval content of a pitch-class set, calculated as the interval class between each pair of pitch classes. The vector has six elements, indicating the occurrences of interval classes 1, 2, 3, 4, 5, 6. (Interval class 0 does not occur, since sets do not contain duplicate elements.)

invariance matrix — A matrix constructed from two row forms or two rows in such a way that invariant segments between the two rows can be easily detected. With the standard matrix, or p-matrix, constructed from the set and its inversion, the user can read invariant properties in transpositions of the prime form of the set. Other types of matrices (i-matrix, m-matrix, mi-matrix) enable the reader to find invariant properties in other transformations (Alphonce, 1974 and Morris, 1987).

invariance vector — An eight-position array showing in the first four positions the numbers of times a set S maps into itself under the operations $T_n(S)$, $T_nI(S)$, $T_nM(S)$, and $T_nMI(S)$; and in the last four positions the number of times it maps into its complement under the same four operations (See *twelve-tone operation.*) (Morris, 1987).

invariant segments — Contiguous segments of two transformations of the same row that have the same pitch-class content.

inverse — Each pitch class interval A has an inverse A', which is equal to $12 - A$ (modulo 12). This operation may be followed by transposition.

inversion — The inversion of a pc set is calculated by subtracting each element of the set from 12 (modulo 12).

k / kh relations — A model of relations among pc set-classes based on subset/superset relations among complexes of pairs of complementary set-classes (Forte, 1973).

m / mi — The operations M and MI are also called M5 and M7. Under these operations each element of a pc set is multiplied by 5 or 7 (modulo 12). The primary characteristic of this transformation is that ic 1 (m2/M7) maps into ic 5 (P4/P5) and vice versa. (Other intervals either map into themselves or their mod 12 inverse, thus preserving the interval class.) MI is the inverse operation of M, i.e., multiplying each element of a set by 7 yields the inversion of the set obtained by multiplying each element by 5. These operations are of particular interest since they map each of the twelve pitch-class integers into another. Other multiplicative operators with this property are M1 (unity) and M11 which produces the inversion.

matrix — In its most conventional meaning, a two-dimensional array constructed from a twelve-tone row such that all 48 transformations of the row can be easily seen. These transformations are the twelve transpositions of the row, twelve transpositions of the inversion of the row, and the retrograde (reverse order) of these twenty-four row forms. In fact, it is possible to construct several types of matrices from a single row or from two rows, which allow one to see many different relationships between the rows or row forms. (See also *invariance matrix.*)

modulo 12 — A modulo 12 system uses 12 integers 0,1,2,...11. Operations such as addition, subtraction, and multiplication are done in the normal way, then if the result is greater than or less than 12, 12 is subtracted or added until the result is in the range 0...11. Since there are 12 pitch classes, musical pitches can be modeled by this system.

normal order — The specific ordering of a pc set that is most closely related to the prime form, but without transposing or inverting the set. Alternately, the term is sometimes used as a synonym for prime form.

ordered set — A set of pitch classes in which the order is significant.

partition — A division of a set or row. A row may be segmented in a number of ways, for example into three note segments. It is also possible to extract non-adjacent elements that have some desired quality, and articulate this property through compositional means such as register, articulation, etc.

pitch class (pc) — A pitch class includes all pitches related by octave equivalence, i.e. all enharmonically equivalent pitches in any octave. There are twelve pitch classes, representing all pitches in the equally-tempered scale.

pitch-class collection — Unlike a set, a collection of pitch-classes may contain duplicate pcs.

pitch-class integer — The twelve pitch classes are represented by the integers 0, 1, 2, ..., 11. Usually 0 represents C, 1 C♯ or D♭, 2 D, etc., although this mapping is arbitrary and is sometimes transposed. Often *a* and *b* are used to represent 10 and 11.

pitch-class set — A collection of pitch classes with no duplicates.

prime form — A reference form of a pc set. The prime form is a normal order of the set transposed to zero. All sets that are related by transposition or transposition after inversion can be reduced to the same prime form, which is commonly used as the representative form of a set class. There are two common methods of calculating the prime form of a set, proposed by Forte (1973) and Rahn (1980). For all but six sets both methods result in the same prime form.

rotational array — A matrix constructed from all circular permutations of an ordered set. Rotational arrays were used compositionally by Igor Stravinsky and Ernst Krenek, among others.

Rp relation — One measure of similarity between two non-equivalent pc sets of the same cardinality. Two sets of cardinality n are Rp-related if they contain at least one common subset of cardinality $n - 1$ (Forte, 1973).

serial composition — A compositional idiom based on the use of an ordered series of pitch classes, or tone row.

set difference — The difference between two pc sets A and B is the set of elements of A that are not elements of B.

set name — A label for a set or set class that consists of the cardinal number of the set followed by the ordinal number, e.g. 3-4 is the fourth set in an ordered list of three-note set classes. The ordered lists of sets are arranged so that complementary set classes occur in the same position of their respective ordered lists. Thus set classes 3-3 and 9-3 are complements, as are 4-28 and 8-28 (Forte, 1973).

set-class — The set of all pitch-class sets that map into the same prime form under the operations of transposition or transposition after inversion. The 4095 possible sets in the twelve-tone system can be reduced to 223 set classes of cardinality 1 to 12, plus the null (empty) set.

subset — A portion of a set. Set X is a subset of set Y if each element of X is also an element of Y.

symmetric difference — The set of elements in the union of two sets that does not occur in the intersection of the sets.

transposition — The operation by which the same interval is added modulo 12 to each element of a set. For a set S, this is signified $T_n(S)$, where n is the transposition interval.

twelve-tone operation (TTO) — An operation on a set or ordered set in the twelve tone system. The standard operations are transposition, inversion, M, and MI. Each of these can be followed by transposition. For a set S, the operations are $T_n(S)$, $T_nI(S)$, $T_nM(S)$, and $T_nMI(S)$, which represent the transposition by interval n of the set, its inversion, and of the sets resulting from multiplication by 5 or 7. In CMAP, we use w (inverted m) to represent MI.

union — The set of elements that occur collectively in two or more sets. The union of {013} and {1357} is {01357}.

Z relation — Two sets or set classes are said to be Z-related if they have the same interval content (interval vector) but cannot be reduced to the same prime form (Forte, 1973).

Acknowledgments

An earlier version of this paper was presented at the annual conference of the Society for Music Theory, Indiana University, Bloomington, November 1986. Some material is excerpted from Harris and Brinkman (1987).

The work described and this article are the result of the combined and equal efforts of the authors. Their names have been listed in reverse alphabetical order and do not imply that either author is senior or subordinate. Some of the material in CMAP is based on Craig Harris's Ph.D. Dissertation (1986) at the Eastman School of Music, University of Rochester. The original concept — a set of analytic programs based on the UNIX† model of software tools that perform specific tasks well but can be interconnected to suit the user's purpose — was Harris's. The initial studies for this work began while Harris was studying with Robert Morris at the Eastman School of Music, and the information in the set tables and program requirements are based largely on material presented by Morris (1984). Aleck Brinkman was Harris's advisor for the research portion of his dissertation, and they have continued work on the program set as a joint research project. The data structure used for the data base was designed by Brinkman, who also wrote the program that generated the data table and many of the library functions. The main programs were written by the authors in equal proportion, and each made significant use of functions written by the other.

The authors gratefully acknowledge the influence of many people's theoretical work upon the development of this software. We are particularly indebted to work presented in Forte (1973) and Morris (1977). Material for these sources is used by permission of Yale University Press. Set-class names, interval vectors, normal order, prime form, and relations Rp, Z, and K/Kh are presented by Forte (1973) and in articles by the same author. Many of these are also discussed by Rahn (1980) and Morris (1987).

The bitwise prime-form algorithm, described by Starr (1978), yields the same prime form as the manual method described by Rahn (1980). These are the same as Forte's except for the six sets listed below. Our set table includes both types; when the Forte type is required, the set-number representing the appropriate prime form is substituted before searching the database.

Set Class	*Forte*	*Rahn*
5-20	01378	01568
6-29	013689	023679
6-31	013589	014579
7-18	0123589	0145679
7-20	0124789	0125679
8-26	0124579a	0134578a

Adjacent interval arrays are presented by Chrisman (1971). Since Chrisman follows Forte's normal order, we adjust the pattern for the six sets listed above when the Rahn form is used.

The multiplicative operators M and MI (sometimes called M5 and M7) are discussed by Howe (1965) and also by Rahn (1980) and Morris (1984 and 1987).

The invariance vector was presented by Morris (1984 and 1987).

The invariance matrix was introduced by Bo Alphonce (1974). The same source also explores the binary representation of pc sets. Morris (1987) explores a number of extensions to the invariance matrix concept.

Rotational arrays are discussed by Rogers (1968).

Special thanks are due to Robert Morris, whose inspired teaching was a prime mover in the inception of this project, and whose insightful criticism and suggestions have been an unwavering source of guidance throughout its fulfillment.

Finally, we would like to thank the Eastman School of Music, The Ohio State University, and University of California at Santa Barbara for serving as beta test sites for this package.

† UNIX is a trademark of Bell Laboratories

MS-DOS is a registered trademark of Microsoft.

MACINTOSH is a trademark of Apple Computers, Inc.

References

Alphonce, B. H. (1974). *The invariance matrix.* (Doctoral dissertation, Yale University). Ann Arbor: University Microfilms International.

Chrisman, R. (1971). The identification and correlation of pitch-sets. *Journal of Music Theory,* **15**, 58-83.

Forte, A. (1973). *The structure of atonal music.* New Haven: Yale University Press.

Harris, C. R. (1986). *Computer programs for set-theoretic and serial analysis of contemporary music; Composition — In such an hour: I. Down the rabbit hole, for orchestra and computer generated tape; II. Pool of tears, for orchestra.* (Doctoral dissertation, University of Rochester). Ann Arbor: University Microfilms International.

Harris, C. R., & Brinkman, A. R. (1986). A unified set of software tools for computer-assisted set-theoretic and serial analysis of contemporary music. *Proceedings of the 1986 International Computer Music Conference.* San Francisco: The Computer Music Association.

Harris, C. R., & Brinkman, A. R. (1987). *CMAP reference manual.* San Francisco: Craig Harris Consulting, Inc.

Howe, H. (1965). Some combinational properties of pitch structures. *Perspectives of New Music,* **4**(1), 45-61.

Morris, R. D. (1984-1985). *Class notes in atonal music theory.* Rochester, NY: University of Rochester, Eastman School of Music.

Morris, R. D. (1987). *Composition with pitch-classes: A theory of compositional design.* New Haven: Yale University Press.

Rahn, J. (1980). *Basic atonal theory.* New York: Longman.

Rogers, J. (1968). Some properties of non-duplicating rotational arrays. *Perspectives of New Music,* **7**(1), 80-102.

Starr, D. (1978). Sets, invariance and partitions. *Journal of Music Theory,* **22**(1), 1-42.