SIGMA, A NEW LANGUAGE FOR INTERACTIVE ARRAY-ORIENTED COMPUTING

R. Hagedorn, J. Reinfelds,
C. Vandoni and L. Van Hove

GENEVA
1973
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

© Copyright CERN, Genève. 1973

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
SIGMA, A NEW LANGUAGE FOR INTERACTIVE ARRAY-ORIENTED COMPUTING

R. Hagedorn, J. Reinfelds*),
C. Vandoni and L. Van Hove**)
ABSTRACT

This report describes the principles and the main facilities of SIGMA (System for Interactive Graphical Mathematical Applications), a programming language for scientific computing whose major characteristics are: automatic handling of multidimensional rectangular arrays as basic data units, interactive operation of the system, and graphical display facilities. After introducing the basic concepts and features of the language, it describes in some detail the methods and operators for the automatic handling of arrays and for their graphical display, the procedures for construction of programs by users, and other facilities of the system.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 General characteristics of the SIGMA language</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Notation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Summary of notation</td>
<td>3</td>
</tr>
<tr>
<td>2. ALPHABET</td>
<td>4</td>
</tr>
<tr>
<td>3. SYNTACTIC STRUCTURE</td>
<td>4</td>
</tr>
<tr>
<td>3.1 Basic sets</td>
<td>4</td>
</tr>
<tr>
<td>3.2 Assignment statements</td>
<td>6</td>
</tr>
<tr>
<td>3.3 Program statements</td>
<td>6</td>
</tr>
<tr>
<td>3.4 Other statements</td>
<td>7</td>
</tr>
<tr>
<td>4. BASIC CONSTRUCTS</td>
<td>8</td>
</tr>
<tr>
<td>4.1 Names</td>
<td>8</td>
</tr>
<tr>
<td>4.2 Types of operands</td>
<td>8</td>
</tr>
<tr>
<td>4.3 Numbers</td>
<td>9</td>
</tr>
<tr>
<td>4.3.1 Real numbers</td>
<td>9</td>
</tr>
<tr>
<td>4.3.2 Complex numbers</td>
<td>9</td>
</tr>
<tr>
<td>4.3.3 Double precision numbers</td>
<td>10</td>
</tr>
<tr>
<td>4.3.4 Boolean truth values</td>
<td>10</td>
</tr>
<tr>
<td>4.3.5 Rules for real, complex and double precision numbers</td>
<td>10</td>
</tr>
<tr>
<td>4.4 Strings</td>
<td>11</td>
</tr>
<tr>
<td>4.5 Labels</td>
<td>11</td>
</tr>
<tr>
<td>4.6 Comments</td>
<td>11</td>
</tr>
<tr>
<td>4.7 Basic data units and values</td>
<td>11</td>
</tr>
<tr>
<td>4.8 Array concept</td>
<td>12</td>
</tr>
<tr>
<td>4.9 Functional form</td>
<td>13</td>
</tr>
<tr>
<td>5. OPERATORS AND OPERANDS</td>
<td>14</td>
</tr>
<tr>
<td>5.1 Scope of operators</td>
<td>14</td>
</tr>
<tr>
<td>5.2 Binding power</td>
<td>14</td>
</tr>
<tr>
<td>5.3 Expression and statement operators</td>
<td>15</td>
</tr>
</tbody>
</table>
6. AUTOMATIC HANDLING OF ARRAYS
   6.1 Index order
   6.2 Concept of row
   6.3 Vector of numbers of components (NCO)
   6.4 Classes of operations on arrays
   6.5 Binary infix operators
   6.6 An extended definition of binary infix operators
   6.7 IF test
   6.8 Display of arrays
   6.9 Subscripting
   6.10 Array generation

7. SPECIALIZED ARRAY OPERATORS
   7.1 Concatenation operator &
   7.2 Projection operator PROJ
   7.3 Reduction operator DROP
   7.4 Shift operators LS and RS
   7.5 Reflect operator REFL
   7.6 Transpose operator TP
   7.7 Trace operator TRACE
   7.8 Diagonalization operator DIAG
   7.9 Topological product TOP
   7.10 Contracted topological products TRATOP and DIATOP
   7.11 Other array operators

8. STRING HANDLING

9. MANAGEMENT OF DATA AND NAMES
   9.1 User code name
   9.2 System library
   9.3 Program names
   9.4 Variable names
   9.5 Global variables
   9.6 Labels

10. USER-CONSTRUCTED SUBPROGRAMS
    10.1 Scope of program names
    10.2 Formal parameters
    10.3 Actual parameters
    10.4 Program call
    10.5 Suspension of execution
    10.6 Program editing
11. **INPUT-OUTPUT**

12. **GRAPHICAL DISPLAY OF ARRAYS**

13. **EXAMPLES**
   - 13.1 Some examples of the graphical facilities
   - 13.2 Discussion of a curve; find extrema
   - 13.3 Fourier analysis
   - 13.4 Quasi-three-dimensional display

14. **ACKNOWLEDGEMENTS**

   REFERENCES

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11.</td>
<td>INPUT-OUTPUT</td>
</tr>
<tr>
<td>12.</td>
<td>GRAPHICAL DISPLAY OF ARRAYS</td>
</tr>
<tr>
<td>13.</td>
<td>EXAMPLES</td>
</tr>
<tr>
<td>13.1</td>
<td>Some examples of the graphical facilities</td>
</tr>
<tr>
<td>13.2</td>
<td>Discussion of a curve; find extrema</td>
</tr>
<tr>
<td>13.3</td>
<td>Fourier analysis</td>
</tr>
<tr>
<td>13.4</td>
<td>Quasi-three-dimensional display</td>
</tr>
<tr>
<td>14.</td>
<td>ACKNOWLEDGEMENTS</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>REFERENCES</td>
</tr>
</tbody>
</table>
1. **INTRODUCTION**

SIGMA (System for Interactive Graphical Mathematical Applications) is a programming language for scientific computing whose major characteristics are the following:

i) The basic data units are scalars, one-dimensional arrays, and multi-dimensional rectangular arrays; SIGMA provides automatic handling of these arrays.

ii) The calculational operators of SIGMA closely resemble the operations of numerical mathematics; procedural operators are often analogous to those of FORTRAN.

iii) The system is designed to be used in interactive mode on terminals connected to a central computer; it provides convenient facilities for graphical display of arrays in the form of (sets of) curves.

iv) The user can construct his own programs within the system and has also access to a program library; he can store and retrieve his data and programs; he obtains on request hard copy of alphanumeric and graphical type.

v) The SIGMA implementation is laid out as a multi-access time-sharing system using the central processor only for actual computation; the implementation also provides for batch-processing use of SIGMA, the user-written code being the same for both interactive and batch modes.

The conceptual development of the SIGMA language resulted from several years of practical work in the field of interactive, array-oriented computing with graphical displays. The early part of this work used GAMMA\(^1\), an interactive system for one-dimensional array computation implemented and operated at CERN on computers of the CDC 3000 series. GAMMA was based on the Culler-Fried system\(^2\) as far as structure and language were concerned; to develop its graphical facilities one relied extensively on user experience. The array-handling methods of SIGMA were taken over from the AMTRAN language\(^3\) and were developed further for multi-dimensional arrays. The SIGMA syntax is similar to FORTRAN (avoiding, however, several of its inconveniences) but contains also elements of other programming languages. SIGMA was first implemented on CDC 6000 series computers at CERN, practical experience was gained, and the graphical facilities were developed by extending GAMMA graphics to multi-dimensional arrays.

The present report describes the principles and main facilities of the SIGMA system. A major part of the facilities are operative, others are being implemented on CDC 6000 series computers at CERN and at the University of Georgia in Athens, U.S.A. The report is not a user manual. It concentrates on the main features of the system rather than on details of operation and implementation; the latter will necessarily be different from one computer centre to the other, and will depend not only on hardware or software considerations but also on the specialization and wishes of the users. In fact, for various features of the system which are described in a specific way in this report, this is done only for purposes of clarity and brevity; other specifications are possible and may even turn out to be preferable depending on the main usage of the system.

Sections 1 to 5 introduce the general concepts and characteristics of the system. Sections 6 to 8 are devoted to the automatic handling of arrays. Section 9 deals with the
management of names and data, Section 10 with user-constructed programs, Section 11 with input-output, and Section 12 with the graphical display of arrays. Some examples are given in Section 13.

1.1 General characteristics of the SIGMA language

A programming language is easy to use if its structure is very simple, or if a beginner can start to use it by learning only a few new concepts and conventions, proceeding later on to more advanced features and capabilities. Very simple languages have the defect that the user, as he gains experience, does not find in them enough power and flexibility to develop his ability and deal with complex problems. The SIGMA language is designed to fall into the second category.

On the simplest level a user has to be familiar with only the concepts of "name", "expression", "statement", and "assignment", and he can calculate with single numbers in a desk calculator fashion. The concept of "vector" leads him to one-dimensional arrays without the need to master the intricacies of multi-dimensional array arithmetic. Finally, the experienced user may use the full language with multi-dimensional arrays, complex and double precision arithmetic, stored programs, and global and local variables. Since all stages are part of one and the same language, the user never has to "unlearn" anything that is no longer valid in his new environment.

1.2 Notation

A BNF (Backus-Naur Form)-like notation will be used in this report for a more concise and clear description of syntactic structures. Semantic notions will be described in English, and any special notation needed will be defined where used.

The basic structural elements of the language are names, numbers, labels, arrays, expressions, statements, lists of any of these, and so on.

The set of all names which may be legally constructed at any given moment may be concisely denoted by <name>, where the diamond brackets denote a set, and the set descriptor inside the diamond bracket is a mnemonic reminder of what the set is all about. Such a notion is not formally rigorous because the exact content of the set depends on the time and point of use; for example: a program name may be used, but it may not be redefined to denote a variable without prior deletion of its use as a program name. Hence the set of legally available names at definition time is the difference between the set of all possible names and the set of names already defined and not to be deleted.

Such subtle distinctions are mostly obvious to the user. They are difficult to formalize through a syntactic formalism such as BNF because they are essentially of semantic origin. Hence it seems advantageous to define only a few general syntactic elements with obvious structural differences. For example, a name differs from a number because a name must start with a letter while a number must start with a digit or the decimal point. The difference between a variable name and a program name is semantic because AAA may represent either, and it is not apparent from the name itself which is which. Such semantic distinctions are covered by the phrase "legally constructed", implying a semantic check at execution time, with an appropriate message to the user if something illegal transpires. Hence we do not subdivide the syntactically (i.e. in this case lexicographically) unique set <name> any further.
Similarly we assume \(<\text{expr}\rangle\) to be the set of all expressions which may legally be constructed in the language. The value of any such expression may be a scalar, or an array of one dimension or more, and its components may either be string characters or numbers. The syntactic structure of an expression does not uniquely determine its type and dimension; hence every operator has to make a semantic check of its operands at execution time. So again, for the purpose of this report, the set \(<\text{expr}\rangle\) is not subdivided further.

New sets may be defined using the symbol \(:=\) which does not belong to the alphabet of our language; \(:=\) simply denotes the words "is defined as a combination of the symbols and of elements of the known sets indicated on the right-hand side". For example:

\[
<\text{DO sta}> := <\text{DO head}> ; <\text{DO loop}>
\]
says that a complete DO statement consists of two parts; the DO head followed by the DO loop.

Two notations will be used to describe simple alternatives in the definition of a set:

i) Curly brackets enclosing the list of alternatives, with each alternative written below the other; for example:

\[
<\text{progr sta}> := <\text{progr head}> ; <\text{sta list}> ; \text{END}
\]

\[
<\text{progr head}> := \text{SUBROUTINE} \ <\text{name}> \ \{\ <\text{empty}> \ \} \ \{\ <\text{name list}> \ \}
\]

The first line says that a subroutine consists of a program head, a statement list, and the final statement END, and the second line that the program head contains the word SUBROUTINE followed either by a user-assigned name, or by a user-assigned name and a formal parameter list between parentheses.

ii) A vertical line separating alternatives; for example

\[
<\text{name}> \ := <\text{letter}> | <\text{name}> \ <\text{alphanumeric}>
\]

\[
<\text{alphanumeric} > := <\text{letter}> | <\text{digit}>
\]

In the second line we define an alphanumeric character to be either a letter or a digit. In the first line we recursively define a name to be either a letter or a name followed by an alphanumeric character. This definition permits names of any length whose first character must be a letter. In the actual implementation the name length is limited to six characters.

1.3 Summary of notation

\(<\text{mnemonic}\rangle\) denotes a set of objects of syntactically obvious distinction, described by mnemonic.

\(:=\) assigns to the set named on the left-hand side the structure described on the right-hand side in terms of known sets and symbols.

\(A | B\) or equivalently \(\{A\} \cup \{B\}\) mean that \(A\) or \(B\) are alternatives in the definition of a set.
2. ALPHABET

The following symbols compose the alphabet of the system:

<table>
<thead>
<tr>
<th>Digits</th>
<th>0 1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbols</td>
<td>+ - * / &amp; . , ; ! # $ ' ( ) $ &quot; : [ ]</td>
</tr>
<tr>
<td>Blank space</td>
<td>&lt;b&gt;</td>
</tr>
</tbody>
</table>

3. SYNTACTIC STRUCTURE

This section contains a simplified description of the syntactic structure of the language. Only those syntactic elements of concern to the user are defined and described. An implementor of the language will have to provide either an exact syntax of basic sets such as <expr>, or a semantic formalism to permit the determination of the legality and an exact specification of the execution sequence of any operator-operand combination constructed by the user. In this report we give the necessary information for such definitions in an informal way by describing each operator in some detail.

3.1 Basic sets

The following basic sets are essential for a syntactic description of the language (for most of them a detailed discussion is given in later sections):

<empty> := a set with no elements. This set is necessary in order to denote missing elements as well as options under which structural elements may be omitted.

<b> := a set with one element which is the alphanumeric blank. The blank is a delimiter in the language and functions as a separator of alphanumeric objects. Except inside a string (Section 4.4), any number of consecutive blanks are equivalent to one blank.

<name> := the set of all names permitted by the language. A legal name contains a letter as the first character, and up to five more letters or digits as subsequent characters. No other symbols and no blanks may appear in a name.

Examples of names are:

X X101 XXX53A ALPHA

The concept of name is slightly ambiguous because it denotes either a value or the storage location address where a value is stored. The simplest way to resolve this is to examine the context in which a name is used. If used
in an expression (value generating situation) the name denotes a value, otherwise it denotes the storage location of a value. In this respect, the set \texttt{name} will be used only in the latter sense to denote the storage location of a value. The value-generating situation will be covered by the set \texttt{expr}.

\texttt{expr} := the set of all legal expressions which may be formed from value generating operators and operands.

A scientist-user has a clear intuitive feeling of what a legal mathematical expression should be; hence he should not be burdened with excessive and artificial syntactic formalism, but the implementation should either accept an expression or provide an error message if it disagrees with its legality. The value of an expression may have scalar or array structure, and may be of real, complex, or string type. No attempt is made at compiling-time to produce all the code that is necessary for execution, but all checking of type and dimension of operands is done by each operator just before execution.

Hence the internal code produced by the compiler of this language must be completely data-independent. Examples of expressions are

\begin{align*}
5.6 & \quad \text{SIN} (\text{PI}/3) + \text{ALPHA}/X \\
A & \quad A \text{ GT } B \\
A + 2 & \quad A + 2 \text{ LT } 5
\end{align*}

\texttt{range} := \texttt{expr} \ # \ \texttt{expr}_2

This is a special concept which is accepted in connection with a few operators only (\texttt{ARRAY} and \texttt{DISPLAY}). It has no value as such and it must be interpreted in the context of the operator concerned. Both \texttt{expr}_1 and \texttt{expr}_2 are scalars. The range expresses the notion of an interval whose lower limit is the value of \texttt{expr}_1 and whose upper limit is the value of \texttt{expr}_2.

\texttt{pair} := \texttt{expr} \ % \ \texttt{expr}_2

This is a special concept used in the display mode for arrays. Its function is to set the abscissa of the scope display with the value of \texttt{expr}_2 and to display the value of \texttt{expr}_1 taken as ordinate against the value of \texttt{expr}_2 on the scope.

\texttt{label} := the set of all labels. A label may be any positive integer number of up to five digits. Its purpose is to label a statement.

\texttt{list} := the concept of list applies equally to names, expressions, or statements. No specification of the length of a list is given by the syntax, although in some cases there is a semantic requirement on its length. Again, an execution time check ensures compatibility or provides an error message if necessary. For example:

\begin{align*}
\texttt{name list} & := \texttt{name} \ | \ \texttt{name} , \ \texttt{name list} \\
\texttt{expr list} & := \texttt{expr} \ | \ \texttt{expr} , \ \texttt{expr list} \\
\texttt{sta list} & := \texttt{sta} \ | \ \texttt{sta} ; \ \texttt{sta list}
\end{align*}
The comma is used as separator, except in the statement list which uses the semicolon.

\[ \text{fun form} := \langle \text{name list} \rangle : \langle \text{expr} \rangle \]

The set of all functional forms. A functional form assigns a special interpretation to a sequence of names (bound variable names) followed by an expression.

\[ \text{sta} := \text{the set of all complete statements which may be constructed in the language.} \]

The notion of a complete statement is necessary because some statements are structured like a statement list. For example the DO head is syntactically defined as a statement but it is not a complete statement. Hence it is not in the set represented by \( \text{sta} \). It becomes a complete statement together with the DO loop, so that

\[ \langle \text{DO sta} \rangle := \langle \text{DO head} \rangle ; \langle \text{DO loop} \rangle \]

is a complete statement.

3.2 Assignment statements

The language utilizes various types of statements. The main ones are assignment statements which occur in two forms:

\[ \langle \text{name} \rangle = \{ \langle \text{expr} \rangle \} \]

\[ \langle \text{name} \rangle (\langle \text{subscr list} \rangle) = \{ \langle \text{expr} \rangle \} \]

Here the subscript list is of the following form:

\[ \langle \text{subscr list} \rangle := \langle \text{empty} \rangle | \langle \text{expr} \rangle | \{ \langle \text{empty} \rangle, \langle \text{subscr list} \rangle \} \]

3.3 Program statements

The definition of a program by a user may be regarded as a complete statement of the language. It has the following form:

\[ \langle \text{prog sta} \rangle := \langle \text{prog head} \rangle ; \langle \text{sta list} \rangle ; \text{END} \]

\[ \langle \text{prog head} \rangle := \text{SUBROUTINE} \langle \text{b} \rangle \langle \text{name} \rangle \{ \langle \text{empty} \rangle \} \]

\[ \text{FUNCTION} \langle \text{b} \rangle \langle \text{name} \rangle (\langle \text{name list} \rangle) \]

where \( \langle \text{name list} \rangle \) in \( \langle \text{prog head} \rangle \) is the list of the formal parameters.

Subroutine call:

\[ \text{CALL} \langle \text{b} \rangle \langle \text{name} \rangle \{ \langle \text{empty} \rangle \} ; \]

\[ \langle \text{act par list} \rangle := \langle \text{act par} \rangle | \langle \text{act par} \rangle , \langle \text{act par list} \rangle \]

\[ \langle \text{act par} \rangle := \langle \text{expr} \rangle | \langle \text{fun form} \rangle | "\langle \text{name} \rangle" \]
Hence an actual parameter may be either a value or a functional form or the name of a sub-
program (subroutine or function).

Function call:

\[ \text{name} \ (\text{act par list}) \]

where \(\text{(act par list)}\) is defined above.

3.4 Other statements

The system also uses IF and DO statements with the following structure:

IF statement

\[
\text{IF } \text{expr}\text{ THEN } \text{sta list} \text{ ELSE } \text{sta list} \text{ FI}
\]

\[
\text{IF } \text{expr}\text{ THEN } \text{sta list} \text{ FI}
\]

\[
\text{IF } \{\text{expr}\} \text{ sta}
\]

DO statement

\[
\text{DO sta} := \text{DO head} ; \text{DO loop}
\]

\[
\text{DO head} := \text{DO } b_1 \ (b) \ \text{name} = \text{expr} , \text{expr} \left\{ \text{expr} , \text{empty} \right\}
\]

\[
\text{DO loop} := \left\{ \text{sta list} , \text{empty} \right\} \ L_1 \ \text{sta}
\]

where \(L_1\) is a label. It marks the end of the \(\text{DO loop}\) and appears in the \(\text{DO head}\) to
bracket the loop uniquely.

The structure of all other statements is self-explanatory or will become clear from
the description of the corresponding operators. For example:

\[
\text{!AXES}
\]

\[
\text{GOTO } \text{expr}
\]

where \(\text{expr}\) is a (calculated) label.
4. **BASIC CONSTRUCTS**

This section describes the basic constructs of the language in an informal way. They form the syntactic building blocks of the language.

4.1 Names

A name is constructed by the user to identify a variable (scalar or array), a function, or a subroutine. There are, in addition, a number of system keywords common to all users. The following rules apply:

i) system keywords are not allowed as names for user-defined objects;

ii) each subprogram definition defines or redefines the name used for it; for user protection, the system asks the user to state explicitly his intention to replace an already existing program by a new one of the same name;

iii) the type and value of a variable are defined or redefined entirely or partly by every assignment statement in which the variable name appears on the left of the assignment statement.

4.2 Types of operands

Operands are the passive elements of the language upon which the operators of the language operate. The most common operands are scalars and arrays.

SIGMA distinguishes two operand types: numbers and string characters. These two types may not be mixed in a single array. Real, integer, and Boolean values are all represented by numbers; they can be mixed in a single array, and in general the user does not have to distinguish between them. Whenever such a distinction is relevant, the operator concerned makes it at execution time and gives an error message if necessary.

4.3 Numbers

Each number is characterized by one property of group A and one property of group B as listed below:

<table>
<thead>
<tr>
<th>Group A</th>
<th>Group B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>single precision</td>
</tr>
<tr>
<td>integer</td>
<td>double precision</td>
</tr>
<tr>
<td>real</td>
<td>multiple precision</td>
</tr>
<tr>
<td>complex</td>
<td></td>
</tr>
</tbody>
</table>

Group A contains semantic properties of numbers, and from top to bottom each lower group includes all higher ones as subsets. Group B contains structural information: it indicates the length of the mantissa of the floating-point representation of the numbers. For example, a number may be single precision real which is the most common case, or double precision integer, or single precision complex (where \( z = z_1 + iz_2 \) and \( z_1, z_2 \) are single precision real), and so on.

In the SIGMA implementation for the CDC 6000 series computers, multiple precision numbers and double precision complex numbers are not available.
4.3.1 Real numbers

A real number may be entered as a string of digits with or without sign, decimal point, and signed or unsigned exponent part. For example, valid numbers are:

\[
\begin{align*}
+123 & \quad \text{meaning } 123 \\
-1234.56E5 & \quad \text{meaning } -(1234.56 \times 10^5) \\
0.3E-2 & \quad \text{meaning } .003 \\
74. & \quad \text{meaning } 74 \\
.6 & \quad \text{meaning } .6
\end{align*}
\]

The "E-format" is identical to the E-format of FORTRAN, and the letter E represents the phrase "times ten to the power of". The general structure is

\[<N_1> \text{ E } <N_2>\]

where \(N_1\) is a signed or unsigned number with up to 15 significant decimal digits and at most one decimal point, while \(N_2\) must be a signed or unsigned integer \(-300 \leq N \leq 300\). No blanks may be embedded anywhere in the number, and \(<N_1>\) must not be omitted since E123 is syntactically indistinguishable from a legally constructed name.

Invalid numbers are:

\[
\begin{align*}
\text{E}5 & \quad \text{the lone exponent part is illegal;} \\
\text{-E}+76 & \\
6E2.3 & \quad \text{the exponent part must be an integer;} \\
2E.5 & \\
2E600 & \quad \text{the exponent is too large.}
\end{align*}
\]

12345678901234567890 too many digits. The system accepts only the first 15 digits and truncates the remaining ones, giving a non-fatal error message.

All numbers entered into the system are stored in floating-point form. Because the CDC floating-point notation assumes the decimal point on the right-hand end of the mantissa, all integer arithmetic is automatically exact provided no intermediate result exceeds 15 decimal digits.

In some cases such as in subscripts or in operators requiring integral arguments, real numbers will be rounded to the nearest positive or negative integer (half-integral numbers are rounded to the nearest integer of larger absolute value). Otherwise, there is no distinction between real and integer.

4.3.2 Complex numbers

A complex number is represented by a pair of real numbers \(n_1, n_2\) separated by the letters I or A where

\[n_1 \text{I}n_2 \quad \text{implies } n_1 + \text{in}_2\]
\[n_1 \text{A}n_2 \quad \text{implies } n_1 \exp (\text{in}_2)\]
The number \( n \) may not be omitted. The real numbers \( n \) are entered as in Section 4.3.1 and are signed or unsigned. No blanks may be embedded anywhere inside a complex number. For example:

\[
\begin{align*}
2.317 & \quad \text{is the complex number} \quad 2.3-7i \\
2.7E-3A4E1 & \quad \text{is the complex number} \quad .0027\exp(40i).
\end{align*}
\]

### 4.3.3 Double precision numbers

The numbers described in Section 4.3.1 are stored and handled as single precision floating-point numbers. This provides a precision of about 15 decimal digits.

To store numbers as double precision floating-point numbers they must be entered using the "D-format". The "D-format" is identical in form to the "E-format" of Section 4.3.1 except that \( E \) is replaced by \( D \), indicating to the system that a double precision number should be stored. Hence valid numbers are

\[-1234.56D5 \]
\[0.5D-2 \]
\[123456789012345678901234567890D0 \]

Double precision provides a mantissa of about 29 decimal digits. Double precision results are always output in the D-format to distinguish them from single precision results even when the user has restricted the automatic output format to a small number of significant digits.

### 4.3.4 Boolean truth values

No special symbols are used for logical truth values "true" and "false". The value "true" is represented by the integer 1 and the value "false" by the integer 0. Conversely, a Boolean operator interprets any integer value 1 as "true" and any zero as "false", regardless of how they were generated, and refuses as illegal any other value (e.g. .999999). Hence

\[ 2 \ \text{GT} \ 1 \]

creates the value 1, while the relation

\[ 2 \ \text{LT} \ 1 \]

creates the value 0. The IF statement

\[ \text{IF} \ 1 \ \text{THEN} \ X = 0 \ \text{ELSE} \ X = 100 \ \text{FI} \]

will always assign to \( X \) the value zero.

### 4.3.5 Rules for real, complex and double precision numbers

The following rules apply:

i) Boolean, integer, and real numbers can be freely mixed.

ii) Single precision numbers are extended to double precision by extending the mantissa with zeros.
iii) If a complex or a double precision number or array appears in an expression, the result of this expression will be complex or double precision, respectively.

iv) No operation may combine complex and double precision elements, otherwise an error message is produced at execution time.

v) Operators are provided (as in FORTRAN) to convert numbers from one form to another:
   - extract the real or the imaginary parts of a complex array to form a real array;
   - form a complex array, by combining two real arrays;
   - form a single precision array using the 15 most significant digits of a double precision array.

4.4 Strings

A string character is any character of the alphabet (see Section 2) except the single quote mark (''). Single quote marks are used to define a string. Any sequence of string characters, including blanks, enclosed in single quote marks constitutes a string, for example

'AB9 CD='

A string is regarded as an array with each string character occupying one element of the array in index order (see Section 6.1).

4.5 Labels

The user may label any statement by writing a positive unsigned integer of no more than five digits before the statement to be labelled, followed by a blank. For example:

101 X = 5
12  X = X + 3

A label is used for reference purposes only. The numerical magnitude of a label has meaning only when used in a computed GOTO statement. Numerical values of labels on sequentially written and executed statements may be in any arbitrary order.

4.6 Comments

The comment is a statement with the structure

$ <comment>

where $ is the "comment operator" saying to the system: "please do not execute the following statement". A comment may contain any characters except $. Comments can appear in subprograms and are stored as part of the subprogram.

4.7 Basic data units and values

Expressions deal either with numbers or with string characters. Integers and Boolean truth values are numbers and are recognized automatically by those execution routines whose results depend on such distinctions.

The basic data unit is called a value and it has the structure of a rectangular array. A value consists of one or more numbers, or string characters, and a specificator containing
the type of the value (real or complex or double precision number, or string character) as well as its dimension information (number of dimensions, and numbers of components for each dimension).

The dimension information of a value is contained in a vector of positive integers called the NCO vector; it indicates the number of components for each dimension of the value. The number of components of the NCO vector is equal to the number of dimensions of the value. Note that the NCO vector of a value is itself a value.

4.8 Array concept

The basic data structure of SIGMA is the rectangular array of \( N \geq 1 \) dimensions. It is a set of components, each of which is indexed by \( N \) subscripts. Each component is a number (number array), or a string character (string array). With the exception of Section 8 our treatment concerns number arrays.

An array is referred to by an unsubscripted name, for example \( A \). Any operation in which this name is an operand will be carried out on the whole array; for example \( \text{SIN}(A) \) computes the sine of each component of \( A \).

Single components of an array are referred to by attaching \( N \) subscripts to its name (where \( N \) is the number of dimensions), for example \( A(i_1, i_2, \ldots i_N) \). Each subscript \( i_k \) is a positive integer taking values \( 1_k = 1, 2, \ldots n_k \), where \( n_k \) is the number of components for the \( k \)th dimension. The NCO vector of the array has then the components \( n_1, n_2, \ldots n_N \).

Single numbers, also called scalars, are considered in SIGMA as equivalent to arrays of one dimension and one component. Arrays of one dimension are called vectors (which means that a scalar is considered as equivalent to a one-component vector).

For arrays of \( N \geq 2 \) dimension, two alternatives present themselves concerning the possible values of the numbers of components \( n_1, n_2, \ldots n_N \). Of greatest generality is the convention that all values \( n_k = 1, 2, \ldots \) are allowed. In addition to vectors as arrays of one dimension, one has then new vector-like structures of \( N \geq 2 \) dimension, e.g. arrays with numbers of components \( n_1 > 1, n_2 = 2, \ldots n_N = 1 \). For \( N = 2 \) they could be regarded as "column" vectors in contradistinction to ordinary or "row" vectors. Similarly, one has matrix-like structures not only for \( N = 2 \) but also for \( N \geq 3 \), many distinct classes being possible (e.g. for \( N = 3 \) arrays with \( n_1 > 1, n_2 > 1, n_3 = 1 \) or with \( n_1 > 1, n_2 = 1, n_3 > 1 \)). The price of generality is that such a rich variety of array structures tends to complicate the methods of automatic array handling.

The second alternative is simpler from a practical standpoint but logically more restrictive. It consists in ignoring every dimension of an array for which the number of components is one. Now all vector-like structures are arrays of one dimension, and all matrix-like structures have dimension two.

In the present report we adopt the second alternative. Concerning the first one, we only mention that we have examined its principles and methods of array handling; it appears that each alternative has specific advantages, the practical significance of which will depend on the main usage of the system.

The array concept is fundamental in SIGMA. Its purpose is to give obvious coherence to sets of numbers representing mathematical functions, vectors, matrices, and other objects
which often have to be handled as a whole. Sections 6 and 7 give a detailed description of array handling.

4.9 Functional form

The concept of functional form derives from the λ-expression of Church\(^1\). It is an unambiguous way of describing a function containing bound variables and parameters:

\[
\text{<fun form>} := "\text{name list} : \text{expr}"
\]

where

\[
\text{name list} \quad \text{contains the list of one or more bound variables which may (but need not) appear in the expression;}
\]

\[
\text{expr} \quad \text{may contain the bound variables defined in the preceding list, and as parameters any variables defined in the context of the use of the functional form. This expression defines the functional form concerned; for example}
\]

\[
"X, Y : (A \ast X + B \ast Y) \ast X"
\]

defines a functional form with two bound variables \(X,Y\) and two parameters \(A,B\) which should be defined in the context in which the functional form is executed.

A functional form is like an implicit subroutine -- it cannot be evaluated by itself. Before evaluation, values must be assigned to the bound variables. SIGMA uses the functional form in two situations:

i) as actual parameter in subroutine or function call (see Section 10.3);

ii) as argument of topological product (Section 7.9) and of contracted topological products (Section 7.10).
5. OPERATORS AND OPERANDS

Data units (see Section 4.7), names representing data units or their location, pieces of code, and complete expressions or statements are called operands. They are passive objects of the system. They are initial values or are the result of some completed action.

Actions upon operands are caused by operators. They are therefore the dynamic objects of the language. An operator is a symbol which denotes a well-defined change in the interpretation, value, or location of one or more operands. For example, let PI be an operand with the value 3.1415... SIN is an operator denoting a well-defined computation, and the completed action SIN(PI) is again an operand, in this case the value zero.

The operator concept may be extended to include objects normally classified as separators, such as the parenthesis, comma, IF, THEN, ELSE, and so on. These may be regarded as operators with extended and sometimes overlapping arguments because they alter the interpretation which their arguments would otherwise have. For example, the argument of the left parenthesis is everything to the right of it until the corresponding right parenthesis. The action of the parenthesis is to alter or suspend the natural binding power of operators in expressions, or to ascribe some special property to their arguments, such as membership of a formal parameter list, actual parameter list, etc.

Two concepts are necessary in order to understand operator-operand interactions in the SIGMA language. They are the scope and binding power of operators.

5.1 Scope of operators

The scope of an operator determines the extent of its influence. Three categories are distinguished:

i) **Immediate commands**: for example, backspace. These are executed immediately upon entry in all circumstances. Hence one may say they have no scope at all.

ii) **Complete statement operators**: This category comprises the bulk of operators of the language. Some operators (for example DO, IF) may extend their influence to the end of a complete statement, but not beyond (see Section 3.4). Most operators have simple operands (for example +, -, *, SIN, COS, etc.). All the user-defined operators belong to this category.

iii) **Modes**: Modes are operators whose scope extends beyond the statement in which they are used until the mode is again explicitly changed by the user. These operators are said to change the mode of the whole system, hence their name. To emphasize their long-term effect, the keywords for all modes start with !, the exclamation mark. For example !PRINT, !AXES, !FRAME are modes. !PRINT will provide a value trace for every subsequently executed statement until an explicit mode change !NOPRINT is executed. The modes !AXES, !FRAME influence the graphical display (see Section 12).

5.2 Binding power

Conventional mathematics assign a fairly well-defined sequence of priorities in the evaluation of expressions containing the traditional binary infix operators +, -, *, /, and exponentiation **. For example, in

\[ A + B/C ** 2 \]
C is first squared, B is divided by the result, and the new result is added to A, while A * B - C + D is evaluated from left to right. In order to provide an evaluation of expressions which is as close as possible to the traditional priorities of mathematical notation, the SIGMA language assigns a binding power to each operator. For the purpose of this report it is not necessary to describe the exact binding power ascribed to all operators of the language.

The concept is as follows. If explicit parentheses are not used by the user, the operands of any operator are determined by scanning the expression (to the left or right as the prefix, postfix, or infix type of the operator demands) until another operator of equal or lower binding power is found. For example, in the expression

$$\text{SIN}(A) + B/C ** 2 - 6$$

the operand of SIN is A, because SIN binds only to its argument in parentheses; the operands of / are B (because * binds less) and C ** 2 (because ** binds more strongly than /, but - binds less strongly).

In cases when the linearization of algebraic expressions could cause notational ambiguities in the interpretation of execution priorities, or if any doubt exists as to what interpretation will be provided by the language, explicit parentheses should be used.

5.3 Expression and statement operators

The set of all operators may be separated into two subsets: operators which serve to generate a value and thus may appear in an expression, named expression operators; and all other operators, named statement operators.

Typical expression operators are +, -, GT, LT, SIN, COS, or any user-generated function. GOTO or DISPLAY are examples of statement operators.

This approach allows the most general definition of <expr> as the set of all combinations of expression operators and operands in which each operator is provided with the correct number of arguments of the correct type and dimension. When this is the case, we may say that each operator is satisfied and the value can be generated. It implies that operator execution routines interpretatively control the number, type, dimension, etc., of the arguments they require, and the user is not forced to foresee all this detail through declarations or other assignment processes. In case of mistake, an error message is provided at execution time.
6. AUTOMATIC HANDLING OF ARRAYS

The general concept of array and the basic definition adopted in this report were discussed in Section 4.8. The present section brings together the key rules for automatic handling of arrays. They form a very central part of the SIGMA language and system. We begin by describing a standard ordering principle for array components, to be called index order, as well as the concept of row of a general array.

6.1 Index order

Computer memory is a linear device where data are stored into successive memory locations. Hence the elements of multi-dimensional arrays have to be arranged in a linear sequence. Such a sequence is called an arrangement in index order. There are many ways of defining such a sequence, but any definition should specify a unique and unambiguous decomposition of a multi-dimensional array into a one-dimensional one or vector. The ordering adopted by SIGMA is defined as follows. In a k-dimensional array with numbers of components \(n_1, n_2, \ldots, n_k\) and with components

\[
A_{i_1i_2\ldots i_p} \quad \text{where} \quad \begin{cases} 
1 \leq i_1 \leq n_1 \\
1 \leq j \leq n_2 \\
\vdots \\
1 \leq p \leq n_k
\end{cases}
\]

one starts with the element \(A_{11\ldots1}\) and allows the rightmost index to run through all its values. Then the next index to the left is run through its values, while, for each value of it, the rightmost index runs through all its values, and so on. For example, the array

\[
A_{ijk} \quad \begin{cases} 
1 \leq i \leq 2 \\
1 \leq j \leq 3 \\
1 \leq k \leq 2
\end{cases}
\]

which has dimension 3 and numbers of components 2, 3, 2 is stored as the vector

\[A_{111} A_{112} A_{121} A_{122} A_{131} A_{132} A_{211} A_{212} A_{221} A_{222} A_{231} A_{232}\]

6.2 Concept of row

Because in common matrix notation \(A_{ij}\) denotes an element of the \(i\)th row and \(j\)th column of the matrix \(A\), it is natural to define the one-dimensional vector \(A_{11}, A_{12}, \ldots\) as a row vector. More generally, one may regard each array as being built up of rows. The scalar and vector are trivial cases. A two-dimensional array of components \(A_{ij} (1 \leq i \leq n_1, 1 \leq j \leq n_2)\) consists of \(n_1\) rows each having \(n_2\) elements \(A_{i1}, A_{i2}, \ldots, A_{in_2}\). Higher dimensions may be visualized through the "library analogy".

Let \(A\) be a three-dimensional array of components \(A_{ijk} (1 \leq i \leq n_1, 1 \leq j \leq n_2, 1 \leq k \leq n_3)\). It consists of \(n_1\) "pages", each containing one matrix of \(n_2\) rows. The array \(A\) has \(n_1 \times n_2\) rows altogether, and each row has \(n_3\) elements. A four-dimensional array consists of \(n_1\) "books", each of \(n_2\) pages, each containing \(n_3\) rows, each row having \(n_4\) elements. There are a total \(n_1 \times n_2\) pages defined by the two left indices, each consisting of a two-dimensional array with numbers of components \(n_3\) and \(n_4\). More generally, if the indices of
an array are arbitrarily divided into a left part and right part, there is one object of
dimension defined by the right part for each legitimate combination of index values in the
left part.

6.3 Vector of numbers of components (NCO)

As mentioned in Section 4.7, each array contains in its specificator a vector whose
components are integers equal to the numbers of components of the array for its successive
dimensions. This vector is available to the user through the operator NCO. Let A be an
N-dimensional array of components $A_{ij...}$, $1 \leq i \leq n_1$, $1 \leq j \leq n_2$, .... Then

$$C = NCO(A)$$

is a vector of N components which are $n_1$, $n_2$, ..., $n_N$. Note that NCO(NCO(<expr>)) is always
a scalar; it is equal to the number of dimensions of the value of <expr>. For the array A
just mentioned, NCO(NCO(A)) = N. If S is a scalar, NCO(S) = 1. Whenever A is an array with
more than one component, all components of NCO(A) are integers ≥ 2.

6.4 Classes of operations on arrays

Operators which act on arrays may be divided into three classes. In order of increased
specialization of arguments these are: functors, binary operators, specialized operators.

i) Functors are single argument functions (such as SIN, COS, for example). They are
defined as operating independently on each component of the array. There is no
restriction on the shape of the array which may appear as an argument of these oper-
ators. For example, if A is an array of components $A_{ij...k}$, then the statement
$B = SIN(A)$ creates an array $B$ of components $B_{ij...k} = SIN(A_{ij...k})$.

ii) Binary infix operators combine two arrays in some way, and some compatibility between
the arrays to be combined is required. This is discussed in Sections 6.5 and 6.6
below. The power of array arithmetic depends largely upon the generality and con-
venience with which arrays may be combined in a logically consistent and non-
ambiguous way.

iii) Specialized operators. This group comprises all other operators defined on arrays.
Some of these may be quite general with few semantic requirements on their arguments,
but others have specific requirements concerning the type and structure of at least
some of their arguments. The specialized array operators will be described individu-
ally in Section 7.

6.5 Binary infix operators

The binary infix operators to be discussed here and in Section 6.6 are + - * / **
AND OR GT LT GE LE EQ NE. We denote by <op> any of these operators. We shall say that two
arrays have identical shape when they have the same NCO vector.

The operation A <op> B is readily defined in the following cases:

i) if A and B are scalars (i.e. single numbers, see Section 4.7);

ii) if A is a scalar and B an array; the operation is then executed on each component of
B and creates an array C of identical shape to B

$$C = A <op> B$$

$$C_{ij...} = A <op> B_{ij...}$$
iii) if A is an array and B a scalar; this case is similar to the previous one;
iv) if A and B are arrays of identical shape; the operation is executed componentwise
    and creates an array of identical shape:

    \[ C = A \langle op \rangle B \]

    \[ C_{ij...} = A_{ij...} \langle op \rangle B_{ij...} \]

Extension of these definitions to other cases is of great interest. Various extensions can
be considered, and a natural one is discussed in the next section. This extension as well
as others can also be achieved by the operator DIATOP of Section 7.10.

6.6 An extended definition of binary infix operators

A very natural extension of the definitions in the previous section concerns arrays A
and B of different dimensions but with the same numbers of components in the last, last but
one, ... dimensions, for example:

\[
A_{ijkl} \quad \text{and} \quad B_{kl} \begin{cases}
1 \leq i \leq n_1 \\
1 \leq j \leq n_2 \\
1 \leq k \leq n_3 \\
1 \leq l \leq n_4
\end{cases}
\]

The operation is executed componentwise with respect to indices k, l; it creates an array
of identical shape to the array of largest dimension

\[ C = A \langle op \rangle B \]

\[ C_{ijkl} = A_{ijkl} \langle op \rangle B_{kl} \]

By making use of the transpose operator TP (Section 7.6), the operation can also be
made to act with respect to others than the last, last but one, ... dimensions.

6.7 IF test

The IF test can take three forms:

\[ \text{IF } \langle\text{expr}\rangle \text{ THEN } \langle\text{sta list}\rangle \text{ ELSE } \langle\text{sta list}\rangle \text{ FI} \]

\[ \text{IF } \langle\text{expr}\rangle \text{ THEN } \langle\text{sta list}\rangle \text{ FI} \]

\[ \text{IF } (\langle\text{expr}\rangle) \text{ ) } \langle\text{statement}\rangle \]

The expression \( \langle\text{expr}\rangle \) must have Boolean value. If \( \langle\text{expr}\rangle \) has an array value, two possibilities exist. We discuss them for the first form of the test:

i) if any one component of the value of \( \langle\text{expr}\rangle \) is true (one), then execute the statement
    list after THEN, otherwise execute the statement list after ELSE;

ii) if all components of the value of \( \langle\text{expr}\rangle \) are true (one), then execute the statement
    list after THEN, otherwise execute the statement list after ELSE.

In SIGMA the second possibility was chosen so that the statement list after THEN is executed
if and only if all components of the \( \langle\text{expr}\rangle \) are true (one). In the second and third form of
the IF test, if all components of the \texttt{<expr>} are true (one), the \texttt{<stmt list>} or the \texttt{<statement>} is executed; otherwise the computer proceeds without executing them.

A logical function \texttt{ANY} is provided in SIGMA, which gives the result "true" if any component of its argument is "true", and the result "false" otherwise. Hence

\texttt{IF (ANY(<expr>)) <statement>}

will execute the statement when any one component of the value of \texttt{<expr>} is "true", while

\texttt{IF (<expr>) <statement>}

will execute the statement if and only if all components of the value \texttt{<expr>} are "true".

6.8 Display of arrays

The display of arrays is one of the major facilities provided by the SIGMA system. It will be described in detail in Section 12 and illustrated with examples in Section 13. Here we give the principles only. The display of a pair

\texttt{<expr1> \&<expr2>}

sets the value of \texttt{<expr2>} in the abscissa of the scope and the value of \texttt{<expr1>} in the ordinate.

In the simplest case, \texttt{<expr1>} and \texttt{<expr2>} are one-dimensional arrays; the 1\textsuperscript{st}, 2\textsuperscript{nd}, ... component of \texttt{<expr1>} is displayed against the 1\textsuperscript{st}, 2\textsuperscript{nd}, ... component of \texttt{<expr2>}, up to and including the last component of the array of smallest number of components (the further components of the other array, if any, are ignored).

If \texttt{<expr1>} is a multi-dimensional array and \texttt{<expr2>} a one-dimensional one, each row of \texttt{<expr1>} is displayed in succession against the common abscissa \texttt{<expr2>}. If \texttt{<expr1>} and \texttt{<expr2>} are multi-dimensional arrays, the 1\textsuperscript{st}, 2\textsuperscript{nd}, ... row of \texttt{<expr1>} is displayed against the 1\textsuperscript{st}, 2\textsuperscript{nd}, ... row of \texttt{<expr2>}, respectively. The ordering of rows follows the index order described in Section 6.1. If \texttt{<expr1>} has more rows than \texttt{<expr2>}, the last row of \texttt{<expr2>} is left in the abscissa for displaying all remaining rows of \texttt{<expr1>}.

Instead of displaying a pair \texttt{<expr1> \&<expr2>} one can display a single expression \texttt{<expr1>}; its rows are then set in the ordinate of the scope and displayed against the array left in the abscissa from the last display command.

As explained in Section 12, various options are foreseen for fixing or modifying the window (i.e. the numerical intervals) displayed on the screen, for selecting linear or logarithmic scales, for giving continuous or broken lines or other patterns, etc. On-line access can be provided to a hard-copy device.

The above-mentioned ability to display multi-dimensional arrays against each other, combined with the transpose operator \texttt{TP} (Section 7.6), permits the simple construction of display subroutines for displaying functions of two or more variables in quasi-three-dimensional perspective projections (see example Section 13.4).
6.9 Subscripting

Individual components of arrays as well as subarrays of arrays are addressed by attaching
a subscript list to the array name:

<name> (<subscr list>)

where

<subscr list> := <empty> | <expr> | <subscr list>, <expr>

i) The number of items in the subscript list, including the <empty>'s, should be equal to
   the number of dimensions of the array to which the subscript list is attached.

ii) Each item of a subscript list specifies one or more indices of the dimension cor-
    responding to this item. If necessary, the value of a subscript expression will be
    rounded by the system, and it should be either a scalar or a vector whose components
    must be positive and have a magnitude not exceeding the number of components in that
    dimension. Such vectors can be specified by means of the concatenation operator of
    Section 7.1, for example 1&2&3.

iii) A subscript list item of type <empty> specifies all values of the index of the corre-
     sponding dimension. It is a shorthand notation totally equivalent to the vector of
     integers

     1&2&3&...&n_j

     where n_j is the number of components in the corresponding dimension.

As an example, consider a three-dimensional array A with components:

\[
A_{ijk} \begin{cases} 
1 \leq i \leq n_1 \\
1 \leq j \leq n_2 \\
1 \leq k \leq n_3
\end{cases}
\]

The expression \( A(3,2, \) ) represents the row \( A_{32k} \), \( 1 \leq k \leq n_3 \), and the expression \( A(1&3,2,1&2) \)
represents the two-dimensional array

\[
A_{121} A_{122} \\
A_{321} A_{322}
\]

A subscripted name may only be used after the unsubscripted name has been defined and given
an array value. Subscripted names may be used on both sides of an assignment statement, for
example:

\[
A(4,2,3) = 2.7 \\
A(4,2,3) = A(4,2,2)
\]

These statements assign the value indicated on the right-hand side to the component \( A_{423} \) of
array \( A \); they do not affect the other components of \( A \). Assignment into single or
subscripted subarrays must agree with the type (real or complex or double precision number, 
or string character) of the original array; the type of an array may be changed only by an 
assignment involving the unsubscripted array name.

6.10 ARRAY generation

Scalars may be generated by direct assignment. Concatenated numbers generate a vector, 
i.e. a one-dimensional array. Any array may be generated by the ARRAY operator, whose first 
argument defines the NCO vector of the resulting array and the second argument specifies the 
values of its components:

$$\text{ARRAY } (\text{<expr}_1 \{, \text{<range}>\}, \text{<expr}_2)$$

<expr>_1 := an expression whose value is a scalar or vector. This value is rounded if it 
contains non-integers, and the (rounded) value must contain only positive 
integers. It forms the NCO vector of the newly created array. If some compo-

<range> := <expr>_1 \#<expr>_4 denotes a range of values. It is interpreted as defining a 
sequence of equally spaced values which are given as components to each row of 
the newly created array; <expr>_2 is the first, <expr>_4 the last element of each 
row; <expr>_3 can be smaller than, equal to, or larger than <expr>_4. In case 
the newly created array is a scalar, its value is taken to be <expr>_3.

<empty> := in this case, all components of the array are given the value one.
7. **SPECIALIZED ARRAY OPERATORS**

7.1 **Concatenation operator &**

Given two arrays $A$ and $B$ which have the same dimension and have the same number of components in all dimensions but the last one,

$$ C = A \& B $$

is the array obtained by concatenating each row of $A$ and the corresponding row of $B$. For example, if $A$ and $B$ are two-dimensional

$$ A_{ij} \begin{cases} 1 \leq i \leq n_1 \\ 1 \leq j \leq n_2 \end{cases} \quad B_{i'j'} \begin{cases} 1 \leq i' \leq n_1' \\ 1 \leq j' \leq n_1' \end{cases} $$

and if $n_1 = n_1'$, the array $C = A \& B$ is two-dimensional and has components

$$ C_{ij} \begin{cases} 1 \leq i \leq n_1 \\ 1 \leq j \leq n_2 + n_1' \end{cases} $$

where

$$ C_{ij} = A_{ij} \quad \text{for} \quad 1 \leq j \leq n_2 $$

$$ C_{ij} = B_{i'j'} \quad \text{for} \quad n_2 + 1 \leq j \leq n_2 + n_1' \quad , \quad j' = j - n_2 $$

Concatenation is particularly useful for constructing a vector (one-dimensional array) out of scalars, for example

$$ F = 1\&1\&2\&2 $$

or out of vectors $V, W, \ldots$, for example

$$ F = V \& W $$

Generalizations are possible. A simple one permits concatenation of an array $A$ and a scalar $S$

$$ G = A \& S $$

It consists in adding at the end of each row of $A$ an additional component of value $S$. Similarly

$$ H = S \& A $$

adds in front of each row of $A$ an additional component of value $S$. 
7.2 Projection operator PROJ

**Purpose:** to project out a set of elements from a multi-dimensional array in one operation.

**Usage:** \[ R = \text{PROJ} (<\text{expr}_1>, <\text{expr}_2>) \]

\(<\text{expr}_1>:=\) specifies the array from which components are to be projected.

\(<\text{expr}_2>:=\) must be a two-dimensional array whose rows specify the subscript lists of the elements which are to be projected. The value of \(<\text{expr}_2>\) is rounded if necessary by the system. The number of elements in each row of \(<\text{expr}_2>\) must be equal to the number of dimensions of \(<\text{expr}_1>\).

The result of this operation is a vector containing the projected elements in the sequence defined by the successive rows of \(<\text{expr}_2>\). More specifically, let

\[ A = \text{ARRAY} (n_1 \& n_2 \& \ldots \& n_N, \ldots) \]

\[ X = \text{ARRAY} (m \& N, \ldots) \]

where \(X\) is a two-dimensional array whose rows have components \(X_{11}, X_{12}, \ldots X_{1N}\) verifying \(1 \leq X_{ij} \leq n_j\). The system rounds the \(X_{ij}\) to integers if necessary.

The statement

\[ P = \text{PROJ} (A, X) \]

gives a vector \(P\) of \(m\) components \(P_i\) equal to the following components of array \(A\)

\[ P_i = A(X_{i1}, X_{i2}, \ldots X_{iN}), \quad 1 \leq i \leq m \]

It is allowed to have some of the rows of \(X\) equal to each other.

7.3 Reduction operator DROP

**Purpose:** To reduce an array by specifying the subarrays which are to be eliminated. This is in some sense the inverse operation to the subscripted array name.

**Usage:** \[ \text{DROP} (<\text{name}>, <\text{expr}_1>, <\text{expr}_2>, \ldots, <\text{expr}_N>) \]

\(<\text{name}>::=\) name of the \(N\)-dimensional array to be reduced.

The remainder of the argument list contains one item for each dimension of the named array.

\(<\text{expr}_1>::=\) a non-negative integer or a vector of positive integers, with value(s) not exceeding the number of components \(n_i\) of the named array in the \(i^{th}\) dimension (the value of \(<\text{expr}_1>\) is rounded by the system if necessary). The subarray corresponding to every component listed in \(<\text{expr}_1>\) is removed from the named array in the \(i^{th}\) dimension. Only whole rows, columns, etc., are removed in this way, always leaving a rectangular subarray. If \(<\text{expr}_1>\) is the scalar zero, nothing is removed in the \(i^{th}\) dimension.

For example, if \(A\) is a two-dimensional array of components

\[ A_{ij} \begin{cases} 1 \leq i \leq 3 \\ i \leq j \leq 4 \end{cases} \]
the statement

\[ B = \text{DROP} \ (A,1,1&3) \]

gives the array of components

\[ B_{11} = A_{22} \quad B_{12} = A_{24} \]
\[ B_{21} = A_{32} \quad B_{22} = A_{34} \]

The statement

\[ B = \text{DROP} \ (A,2&3,0) \]

gives the one-dimensional array of components

\[ B_i = A_{i1}, \quad 1 < i < 4 \]

Note: If all components of an array were dropped, the result would be a no-component array, hence an error message is given.

7.4 Shift operators LS and RS

Purpose: To shift circularly each row of a multi-dimensional array to the left or right a given number of steps.

Usage: \( \text{LS} \ (<\text{expr}_1>, <\text{expr}_2>) \)
\( \text{RS} \ (<\text{expr}_1>, <\text{expr}_2>) \)

\(<\text{expr}_1> := \) array to be shifted

\(<\text{expr}_2> := \) value must be a scalar (rounded if necessary by the system). Interpreted as the number of places the array has to be shifted to the right by RS and to the left by LS. The scalar \(<\text{expr}_2>\) can be negative, in which case RS shifts to the left and LS to the right a number of places equal to the absolute value of \(<\text{expr}_2>\).

Note: The shift is performed circularly modulo \(N\), where \(N\) is the number of components in the rows of the array to be shifted. Hence, \(\text{RS}(X, N+1)\) shifts the \(N\) component rows of \(X\) by 1 to the right, and \(\text{RS}(X,-1)\) shifts the rows by \(N-1\) to the right (or by 1 to the left).

7.5 Reflect operator REFL

Purpose: To reflect the components in every dimension specified. To reflect the components in the \(k\)th dimension means to reverse the component sequence such that the \(i\)th component exchanges place with the \((n_k - i + 1)\)th component (\(n_k\) is the number of components in the \(k\)th dimension).

Usage: \( \text{REFL} \ (<\text{expr}_1>, <\text{expr}_2>) \)

\(<\text{expr}_1> := \) any array

\(<\text{expr}_2> := \) must be scalar or vector, rounded if necessary by the system. It gives the dimensions in which reflection is to take place. All
components of \(<\text{expr}_2>\) should be different and none should be larger than the dimension of \(<\text{expr}_1>\).

Example: If \(A\) has components

\[ A_{ij}, \quad 1 \leq i \leq 3, \quad 1 \leq j \leq 2 \]

then

\[ B = \text{REFL} (A,2) \]

has components

\[ B_{1i} = A_{1i}, \quad B_{12} = A_{11}, \quad 1 \leq i \leq 3 \]

The abbreviated form \(\text{REFL} (\text{expr}_1>)\) is interpreted as reflecting the rows of the array \(<\text{expr}_1>\). It is equivalent to \(\text{REFL} (\text{expr}_1>_{}, N)\) where \(N\) is the dimension of \(<\text{expr}_1>\).

7.6 Transpose operator TP

Purpose: To permit the rearrangement of components of an array by exchanging the position of two or more dimensions in its structure, i.e. by permuting two or more indices.

Usage: \(\text{TP} \ (<\text{expr}_1>, <\text{expr}_2>)\)

\(<\text{expr}_1>\) := array to be transposed; it should have \(N \geq 2\) dimensions.

\(<\text{expr}_2>\) := a vector composed of \(N\) positive integers (rounded by the system if necessary), \(1 \leq N\) and all different from each other. The value of \(<\text{expr}_2>\) gives the permutation of indices to be achieved in the rearrangement of the components of \(<\text{expr}_1>\).

Example: A being a three-dimensional array with numbers of components 5, 6, and 7, the array

\[ B = \text{TP} \ (A,3&1&2) \]

is three-dimensional and has the components

\[ B_{ijk} = A_{kij}, \quad 1 \leq i \leq 6, \quad 1 \leq j \leq 7, \quad 1 \leq k \leq 5. \]

A simple mnemonic of the rule of index permutation is to remember that in the above case

\[ B_{123} = A_{312} \]

More generally, if \(a,b,c\) is a permutation of 1,2,3 and

\[ L = \text{TP} \ (A,a&b&c) \]

one has \(L_{1i12} = A_{j1j2j3}\), with \(j_1 = i_a, j_2 = i_b, j_3 = i_c\) and in particular \(L_{123} = A_{abc}\). The abbreviated form \(\text{TP} \ (<\text{expr}>\)\) is interpreted as cyclic transposition, hence \(\text{TP} \ (<\text{expr}>\)\) is equivalent to \(\text{TP} \ (<\text{expr}>, 2&3&...&N&1)\), \(N \geq 2\) being the dimension of the array \(<\text{expr}>\).
7.7 Trace operator TRACE

**Purpose**: To contract a multi-dimensional array in some of its dimensions by summing over certain indices, some of these being set equal to each other; this operation is a generalization of the trace operation of tensor calculus but permits also summation over single indices.

**Usage**: \( \text{TRACE} (\langle \text{expr}_1 \rangle, \langle \text{expr}_2 \rangle, \langle \text{expr}_3 \rangle, \ldots) \)

\( \langle \text{expr}_1 \rangle := \) array whose (generalized) trace is to be taken; the numbers of components in the dimensions to be contracted with each other should be equal.

\( \langle \text{expr}_2 \rangle, \langle \text{expr}_3 \rangle, \ldots := \) one or more integer scalars or vectors (rounded by the system if necessary) specifying the indices to be contracted and summed over; the components of \( \langle \text{expr}_2 \rangle, \langle \text{expr}_3 \rangle, \ldots \) must all be positive integers not larger than the dimension of \( \langle \text{expr}_1 \rangle \), and all must be different from each other.

The execution is as follows. Let \( A \) be an array of dimension 5, with numbers of components \( n_1, n_2, \ldots, n_5 \). The array

\[
B = \text{TRACE} (A, 4, 2)
\]

is defined if \( n_4 = n_2 \); it is then the three-dimensional array of components

\[
B_{ijk} = \sum_{a=1}^{n_2} A_{iajka} , \quad 1 \leq i < n_1 , \quad 1 \leq j < n_3 , \quad 1 \leq k < n_5 .
\]

The array

\[
C = \text{TRACE} (A, 2, 4, 3)
\]

is defined if \( n_2 = n_4 \); it is then the two-dimensional array of components

\[
C_{ij} = \sum_{a=1}^{n_2} \sum_{b=1}^{n_4} A_{iabaja} , \quad 1 \leq i < n_1 , \quad 1 \leq j < n_5 .
\]

Note that this operator also permits just to sum over certain indices; for example, the array

\[
D = \text{TRACE} (A, 2, 3, 4)
\]

is always defined and is the two-dimensional array of components

\[
D_{ij} = \sum_{a=1}^{n_2} \sum_{b=1}^{n_3} \sum_{c=1}^{n_4} A_{iabcj} .
\]

\( \text{TRACE} (A, 1, 2, 3, 4, 5) \) is a scalar equal to the sum of all components of \( A \). The abbreviated form

\[
\text{TRACE} (A)
\]
is interpreted as TRACE (A,(N-1)&N) if the dimension N of A is ≥ 2, and as TRACE (A,1) if A is a vector (N = 1). In the latter case TRACE (A) is a scalar equal to the sum of all components of the vector A.

7.8 Diagonalization operator DIAG

Purpose: To extract from a multi-dimensional array an array of lower dimension by setting some of its indices equal to each other.

Usage: DIAG (<expr_1>, <expr_2>, <expr_3>, ...)  
<expr_1> := array on which diagonalization is to be executed; it should have dimension N ≥ 2 and its numbers of components in the indices to be set equal should be the same.
<expr_2>, <expr_3>, ... := one or more integer vectors (rounded by the system if necessary) specifying which groups of indices are to be set equal; each of the vectors <expr_2>, <expr_3>, ... must have at least two components; all components must be positive integers ≤ N, and all must be different from each other.

The execution is as follows. Let A be a five-dimensional array with numbers of components n_1, n_2, ..., n_5. The array

\[ B = \text{DIAG} (A,4&1&2) \]

is defined if n_0 = n_1 = n_2. It has three dimensions and its components are

\[ B_{ijk} = A_{lijlk}, \quad 1 ≤ i ≤ n_1, \quad 1 ≤ j ≤ n_2, \quad 1 ≤ k ≤ n_5 \]

The array

\[ C = \text{DIAG} (A,1&2&4,3&5) \]

is defined if n_1 = n_2 = n_3 and n_5 = n_5. It has two dimensions and its components are

\[ C_{ij} = A_{lijij}, \quad 1 ≤ i ≤ n_1, \quad 1 ≤ j ≤ n_3 \]

The abbreviated form DIAG (A) is interpreted as DIAG (A,(N-1)&N) where N ≥ 2 is the dimension of the array A.

7.9 Topological product TOP

The topological product TOP and its contracted forms TRATOP and DIATOP are very important operations in the SIGMA system. The purpose of the TOP product is to generate higher dimensional arrays starting from lower dimensional ones, in particular from one-dimensional ones (vectors), just as one can generate functions of several variables from the various independent variables. It uses the functional form described in Section 4.9.

Consider a functional form

"<name_1>, <name_2>, ..., <name_N> : <expr>"
containing \( N \geq 2 \) bound variable names \(<\text{name_1}, \ldots, \text{name_N}>\). The expression \(<\text{expr}>\) may (but need not) contain some or all of these names. It may also contain other names \(<\text{name_{N+1}}, \ldots, \text{name_p}>\). All names \(<\text{name_1}, \ldots, \text{name_p}>\) should be different from each other and should refer to previously defined scalars or arrays. The \(<\text{expr}>\) should have the property that it becomes a scalar when

i) scalar quantities are substituted for the bound variable names \(<\text{name_1}, \ldots, \text{name_N}>\);

ii) their previously defined (scalar or array) values are substituted for the other names \(<\text{name_{N+1}}, \ldots, \text{name_p}>\).

Then the statement

\[
F = \text{TOP} ("<\text{name_1}, <\text{name_2}, \ldots, <\text{name_N}> : <\text{expr}>")
\]

defines an array with the following properties:

i) It has the structure of the topological product of the previously defined arrays of names \(<\text{name_1}, <\text{name_2}, \ldots, <\text{name_N}>\); in other words, its vector of numbers of components \( \text{NCO}(F) \) is obtained by forming the concatenation

\[
\text{NCO} (<\text{name_1}>) \& \text{NCO} (<\text{name_2}>) \& \ldots \& \text{NCO} (<\text{name_N}>)
\]

and by dropping from it any component of value one (i.e. the \( \text{NCO} \) of any \(<\text{name_i}>\) which has scalar value). If all \(<\text{name_i}>\) have scalar values, \( F \) is a scalar.

ii) Each component of \( F \) is the scalar value of \( <\text{expr}> \) obtained by replacing in \( <\text{expr}> \) the bound variable names \(<\text{name_1}, <\text{name_2}, \ldots, <\text{name_N}>\) by the corresponding components of the arrays which they represent.

This rule is clarified in the following examples: let \( X \) and \( Y \) be the names of vectors with components

\[
X_1, X_2, \ldots, X_n \quad \text{and} \quad Y_1, Y_2, \ldots, Y_p .
\]

Then

\[
F = \text{TOP} ("X,Y:X+Y**2")
\]

is a two-dimensional array with components

\[
F_{ij} = X_i + Y_j^2 , \quad 1 \leq i \leq n , \quad 1 \leq j \leq p .
\]

\( Z \) being the name of a vector of \( q \) components, the array

\[
G = \text{TOP} ("X,Y,Z:SIN(X)*COS(Y)")
\]

is three-dimensional and its components are

\[
G_{ijk} = \sin (X_i) \cos (Y_j) , \quad 1 \leq i \leq n , \quad 1 \leq j \leq p , \quad 1 \leq k \leq q .
\]

Note that the third bound variable does not occur in the \(<\text{expr}>\), but is used to determine the third dimension of \( G \).
An extreme case would be

\[ H = \text{TOP}(''X,Y:S'') \]

where \( S \) is a previously defined scalar. The array \( H \) has dimension 2 and the following components

\[ H_{ij} = S, \quad 1 \leq i \leq n, \quad 1 \leq j \leq p. \]

The topological product can also act on multi-dimensional arrays, as in

\[ M = \text{TOP}(''X,Y,Z:(X+2)*Y'') \]

where \( X \) and \( Y \) are the same vectors as before and \( Z \) is a two-dimensional array of components

\[ Z_{kl}, \quad 1 \leq k \leq r, \quad 1 \leq l \leq s. \]

The array \( M \) has dimension 4 and its components are

\[ M_{ijkl} = (X_iZ_{kl})Y_j \begin{cases} 1 \leq i \leq n, \quad 1 \leq j \leq p, \\ 1 \leq k \leq r, \quad 1 \leq l \leq s. \end{cases} \]

Note that the ordering of indices of \( M \) follows the order of the bound variable list \( X,Y,Z \).

The rule that all names occurring in \( \text{TOP} \) must be different implies that, if the user wants to combine topologically an array with itself, he must first create an identical array bearing another name. For example, \( X \) and \( Y \) being the same vectors as before, the two statements

\[ W = X \]
\[ A = \text{TOP}(''X,Y,W:X*W+Y'') \]

will create a three-dimensional array \( A \) with components

\[ A_{ijk} = X_iW_k + Y_j = X_iX_k + Y_j, \quad 1 \leq i \leq n, \quad 1 \leq j \leq p, \quad 1 \leq k \leq n. \]

7.10 **Contracted topological products TRATOP and DIATOP**

One is often led to combine arrays as in the topological product defined above, but without intending to use all components of the topological product. A typical example is the matrix multiplication of two square arrays \( A,B \)

\[ P_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk}, \quad 1 \leq i \leq n, \quad 1 \leq k \leq n. \]

This can be performed by combining the operators \( \text{TOP} \) and \( \text{TRACE} \)

\[ P = \text{TRACE}(\text{TOP}(''A,B:A*B''),2&3) \]

Such a procedure is wasteful of core space, however, because it generates the four-dimensional array \( \text{TOP}(''A,B:A*B'') \) which has \( n^4 \) components \( A_{ij} B_{kl} \), and does this for the sole purpose of extracting the contracted quantity \( P \) which has \( n^2 \) components only.
To avoid such wasteful procedures, SIGMA defines two forms of contracted topological products, equivalent to \textsc{trace} (TOP...) and \textsc{diag} (TOP...), respectively. They are

i) \textsc{tratop} ("name_1, \ldots, name_N : <expr>", <expr_1>, <expr_2>, \ldots) which follows the same rules and gives the same result as the combined operations

\textsc{trace} (TOP("name_1, \ldots, name_N : <expr>")), <expr_1>, <expr_2>, \ldots)

ii) \textsc{diatop} ("name_1, \ldots, name_N : <expr>", <expr_1>, <expr_2>, \ldots) which follows the same rules and produces the same result as

\textsc{diag} (TOP("name_1, \ldots, name_N : <expr>")), <expr_1>, <expr_2>, \ldots)

The matrix multiplication mentioned above will therefore be executed as

\[ P = \textsc{tratop} ("A,B:A*B",2&3) \]

On the other hand, the operations

\[ Q = \textsc{tratop} ("A,B:A*B",2&4) \]

\[ R = \textsc{diatop} ("A,B:A*B",1&4,2&3) \]

will respectively construct the two-dimensional arrays with components

\[ Q_{ij} = \sum_{k=1}^{n} A_{ik} B_{jk} \] \[ R_{ij} = A_{ij} B_{ji} \] \[ 1 \leq i \leq n \] \[ 1 \leq j \leq n \]

### 7.11 Other array operators

SIGMA provides additional array operators, some of which will now be described in general terms. Further such operators can be defined and implemented on user's request.

The operator \textsc{sum} (<expr>) takes every row of the array <expr>, say \( X_1, X_2, \ldots, X_1, \ldots, X_n \), and creates an array with components equal to the running sum of the components of \( X \):

\[ X_1, X_1+X_2, \ldots, X_1+X_2+\ldots+X_i, \ldots, X_1+X_2+\ldots+X_n \]

The operator \textsc{diff} (<expr>) is the inverse of \textsc{sum}; it replaces every row of <expr> by the row of backward differences and uses a simple extrapolation for the first element. Thus \( X_1, \ldots, X_n \) is replaced by

\[ X_1 - X_1, X_2 - X_1, X_3 - X_2, \ldots, X_n - X_{n-1} \]

\( X_1 \) being a fake zeroth component calculated by quadratic extrapolation to the left. It is clear that more sophisticated component calculated by quadratic extrapolation to the left. It is clear that more sophisticated integration and differentiation operators can be defined if the number of components of the rows is sufficient.

The operator \textsc{delta} (<expr>) is defined for real valued arrays only. It takes every row \( X_1, \ldots, X_n \) of the array <expr> and replaces every component \( X_i \) by zero except in three cases where it replaces \( X_i \) by one:

i) when the component \( X_i \) is itself zero;
ii) when \(X_{i-1}, X_i\) are of opposite sign and \(|X_i| < |X_{i-1}|\); if \(i = 1\), quadratic extrapolation to the left is used to calculate \(X_0\);

iii) when \(X_i, X_{i+1}\) are of opposite sign and \(|X_i| \leq |X_{i+1}|\); if \(i = n\), quadratic extrapolation to the right is used to calculate \(X_{n+1}\).

Also \(\max\) (maximum) and \(\min\) (minimum) operators are provided by \(\text{SIGMA}\). For each row of the array \(<\text{expr}>\) the operator \(\max\) \(<\text{expr}>\) replaces all elements by the maximum value found in that same row, without changing the structure of the array. Hence the vectors \(\text{NCO}(\max(A))\) and \(\text{NCO}(A)\) are identical. The operator \(\min\) acts correspondingly.

If one wishes to know only one single number, namely the value of the largest or smallest component of the whole array, one uses \(\text{SMAX}\) or \(\text{SMIN}\). The result of these operators is always a scalar.

The logical operator \(\text{ANY}\) has already been defined in Section 6.7. The \(\text{SIGMA}\) system has also an operator \(\text{EVAL}\) taken over from the Culler-Fried language\(^4\); it makes it possible to evaluate by interpolation a function of a variable for a new set of values of this variable.
8. STRING HANDLING

As mentioned earlier (Section 4.4), a string character may be any character of the alphabet of the language including a blank space, except for the quote mark (') which is the string-defining character.

Any string character occupies one component of an array, and string arrays can be given the same multi-dimensional array structure as number arrays. For input and output the following rules hold:

i) In array generation, quote marks must be used around the string to be input, blank spaces count as characters, and blank spaces fill the array components for which no input is specified. For example

\[ X = \text{ARRAY} (10, 'ABCD\ EF\GH!') \]

generates a vector of 10 components consisting of the string 'ABCD\ EF\GH!' (note the blank at the end), and

\[ \text{PRINT } X(11009868144) \]

types out the string

A HEAD

ii) Since each array component is occupied by one and only one character, and since blank is a character, string characters are delimited by themselves and they must not be separated by blank spaces for input or output. This is the difference in an otherwise identical output format for numbers. A vector will print horizontally and a multi-dimensional array will print row by row, each new row starting on a new line.

It is clear that a page of text can essentially be regarded as a two-dimensional string array, and conversely. A book can similarly be regarded as a three-dimensional array. The index order defined in Section 6.1 coincides with the natural writing order of a text. All this suggests the use of array operators for purposes of symbol and text manipulation. The relational and arithmetic operators defined for number arrays can be used for this purpose if string characters are assumed to be represented in a one-to-one way by real numbers (for example by integers). For example, A and B being string arrays of identical shape, the Boolean array

\[ C = A \ NE \ B \]

has components of value one (true) where the "texts" A and B differ, and value zero (false) where the "texts" are identical. Assume further that the blank space is represented by zero. The arrays or "texts"

\[ A1 = A * C, \quad B1 = B * C \]

will be identical to A and B, respectively, where A and B differ, and will contain blanks where A and B are identical. If N denotes the common dimension of A and B, the array

\[ T = \text{TRACE} (C_{\times N}) \]

has dimension \( N - 1 \); each component of T corresponds to a row of A and a row of B, and its value is the number of characters by which these two rows differ. If A and B
are "pages" (N = 2) and B is a copy of A, the vector T says how many "copying errors" have been made in each line. Specific lines and characters can be called and manipulated by subscripting.

We shall not pursue these matters further in the present report, nor shall we specify the (rather obvious) syntactical rules to be obeyed by relational and arithmetic operators when some of the operands are strings. All we wanted to show in this section is that, although SIGMA has been developed as a language for numerical array computation, it offers interesting possibilities in the field of symbol and text manipulation. Only further work will establish how useful these possibilities can be in practice.
9. **MANAGEMENT OF DATA AND NAMES**

This section describes the general principles followed by the SIGMA system for the handling and storage of user-defined data, subprograms, and their names. Each user has a number of facilities which are at his disposal without interference with other users, but communication between users as well as between user and program library is also possible when desired. The main facilities available to every SIGMA user are the following:

i) The user is able to define and construct variables and subprograms, assign a name to them, and use them later by referring to their name.

ii) The user is able to PROTECT names so that their contents cannot be changed by reassignment, and to DEPROTECT them when he wishes.

iii) He is able to change the contents of an unprotected name in full or in part by reassignment or editing. He is able to DELETE an unprotected name and its value in order to free memory space.

iv) The user has access to library programs. He can be given access to data and programs of other users.

v) The user is able to store his own data and programs by a log-out procedure, and to retrieve them by a log-in procedure.

vi) His data and programs cannot be modified by other users.

vii) By means of the mode operator !NAMES, the user obtains a full listing of all names defined by him and in existence in the system, with indication of type, numbers of components vector, protection status, etc.

9.1 **User code name**

To guarantee the above facilities, each user has his own user code name. The following rules apply:

i) The assignment of user code names is under control of the system supervisor, who checks the list of user code names on each assignment so as to avoid duplication.

ii) The log-in procedure asks the user to identify himself by typing in his user code name. Log-out is performed under the user code name.

iii) Library programs provided by the system have no user code name attached to them and are automatically available to all users. Names of library programs are all protected names.

9.2 **System library**

Programs are placed in the system library by the system supervisor, who assigns a unique name to each program. The deletion of any program from the library is also done externally by the system supervisor. The alteration or updating of the library is outside the scope of the language and not under user control.

9.3 **Program names**

Subprogram names defined by a user are global and unique to his entire working environment. Since the redefinition of a subprogram name (either by assignment to a new subprogram or an array, or in editing of the original subprogram) deletes this subprogram, the system
9.4 Variable names

The scope of variables defined by the user in interactive mode is local to the interactive mode, but they remain in storage until explicitly deleted by the user. Deletion is achieved by the operator DELETE followed by the names to be deleted:

\[ \text{DELETE <name list>} \]

The value of any name may be changed by an assignment statement. In this way valuable data or programs can be overwritten accidentally. To protect such valuable information, SIGMA provides the PROTECT and DEPROTECT statement; namely,

\[ \text{PROTECT <name list>} \]

assigns all names of the argument list to protected status. These names are not accepted by a program head or on the left-hand side of an assignment statement. They will not be accepted as arguments by the DELETE statement.

Names are released from protected status by the DEPROTECT statement; namely,

\[ \text{DEPROTECT <name list>} \]

will remove all names contained in the argument list from protected status, so that their value can again be altered or deleted.

Variables defined in a subprogram are local to that subprogram. They become undefined upon exit from the subprogram, hence they exist only during the execution of the subprogram in which they are defined.

As explained in the next section, variable names can be declared global.

9.5 Global variables

Any local variable (local to the interactive mode, or local to a subprogram) is made global by explicit declaration, e.g.

\[ \text{GLOBAL X,Y,Z} \]

where X, Y, Z are the names of the variables to be treated as global.

The SIGMA concept of GLOBAL is similar to that of the labelled COMMON of FORTRAN. A variable declared GLOBAL in interactive mode or in a subprogram becomes available to all those subsequent subprograms in the calling chain in which the same name is declared global.

i) Any subsequent subprogram down the calling chain can use and can alter the value of the global variable if the name is declared global in it; a local variable of the same name may be used quite independently in subprograms on any calling level in which the name concerned is not declared global.

ii) When a variable has been declared global for the first time in a subprogram, it becomes undefined upon exit from this subprogram. If a variable has been declared global in interactive mode, it remains global until it is deleted by a DELETE statement.
9.6 Labels

An unsigned integer label of no more than five digits may precede any statement. These labels are used for transfer of control only, and their magnitude has no influence upon the order of execution of the statements. The same label may not be used on two statements within its scope of definition.

A label defined in a subprogram is local to that subprogram. Hence the same label may be used by two different subprograms also when they call each other. No two labels can be the same within a subprogram (including DO loops, if any).

All labels defined in interactive mode become undefined upon terminating a session. Therefore the same labels can be employed again in the next session, even when the data of the previous session have been kept by log-out and retrieved by log-in.
10. USER-CONSTRUCTED SUBPROGRAMS

A subprogram may be constructed and stored in the system by the user, with a program statement referred to as <progr sta>. A program statement consists of a statement list bracketed by the program head <progr head> at the beginning and the keyword END at the end. The statement list may contain any number of complete statements, labelled or unlabelled:

<progr sta> ::= <progr head> ; <sta list> ; END

The program head describes the type of subprogram using the keywords FUNCTION or SUBROUTINE, defines a user-given name for the subprogram, and provides if necessary a list of formal parameters.

The program head of a function or subroutine is constructed as follows:

<progr head> ::= SUBROUTINE <b> <name> \{<name list>\} or FUNCTION <b> <name> \{<name list>\}

A function subprogram must always have a list of one or more formal parameters. A function subprogram should have a value assigned to its user-assigned name at least once in the subprogram body. The last value assigned to its name during execution is the value of the function program. Hence the execution of a function always results in a value, and functions may be used in expressions in order to generate further values.

A subroutine carries out a sequence of statements as specified in the subroutine body. It assigns no value to the name of the subroutine; hence it cannot be used in an expression. It has to be called using the CALL operator, and a subroutine call forms a complete statement by itself.

Programs are not executed while they are defined. The <progr head> automatically invokes the "definition" mode in the system, indicating that a new program has to be accepted and checked, but not executed. The END statement returns the system to the interactive mode (where execution follows each "end-of-message" command given by the user). During the definition of a program, each "end-of-message" command enters a statement into the system, and after some syntax checking the system returns control to the user for the entry of the next statement. When a program has been called, all statements in the <sta list> are executed until one of the keywords END or RETURN is encountered; the system then returns to the calling level (see, however, Section 10.5).

10.1 Scope of program names

A program name, once defined, is unique and global at all levels of execution. Hence any program may be called, on any level of nesting, by any other program in the system. We envisage that programs may be recursive, either directly or indirectly.

All variables defined inside a program are local to the program unless explicitly declared global (see Section 9.5). All other outside variables are unavailable during the execution of the program. The subprogram body communicates with the environment of its call through formal-actual parameter correspondence and by global variables.
10.2 Formal parameters

The user may assign any `<name>` to a formal parameter, but all formal parameter names must be listed in the program head.

At execution time, each formal parameter is matched with its corresponding actual parameter, and only at this time can a complete check of the correspondence be made.

10.3 Actual parameters

Two kinds of actual parameters are permitted in the call of a subprogram P (subroutine or function):

i) `<expr>`, which is evaluated at the time of the program call and may contain variables defined on the calling level. The value of `<expr>` is set up as an implicit local variable of the program P and used as any other local variable of P.

ii) A piece of code which should be evaluated on each use of the corresponding formal parameter during the execution of the program P. Such actual parameters are enclosed in double quotes (to emphasize that they do not represent a value as such) and are of two kinds:

   a) "<name>"

   This must be the name of a previously defined subprogram S. If S has a parameter list, the formal parameter replaced by "S" must have a matching parameter list wherever it occurs in the body of program P. This use of actual parameters is similar to the EXTERNAL concept of FORTRAN but without the need for an explicit declaration of external names.

   b) `<fun form>`

   This is similar to the Algol call by name with Jensen's device resolved directly. The functional form `<fun form>` is a λ-expression in the sense of Church\(^1\). For example,

   "X,Y:A*Y+B*X"

   is a functional form with two bound variables X,Y and two parameters A,B. When used as an actual parameter, the `<fun form>` is regarded as an implicit subprogram with two formal parameters X and Y, and with A and B taking their values from the calling level in the sense of the one-line function of FORTRAN. The actual parameters to this implicit subprogram are provided by the parameter list attached to every use of the formal parameter in the body of the program P. Hence the functional form is evaluated on every use of the corresponding formal parameter, using local variables of the program P for the bound variables (X,Y above) and local variables of the calling level for the parameters (A,B above).

10.4 Program call

A function subprogram generates a value. It may be used in any expression and is called by

    `<name> (<act par list>)`

where `<name>` is the user-assigned name of the function and `<act par list>` the list of actual parameters.
A subroutine is called by the subroutine call statement

\[ \text{CALL } \{ \text{<act par list>}, \text{<empty>}\} \]

10.5 Suspension of execution

The system provides an automatic safeguard against infinite loops which might occur in executing either user-defined subprograms, or DO loops and GOTO statements performed in interactive mode. It operates by means of an automatic safeguard counter. The counter is set to zero by each statement manually introduced by the user at the terminal keyboard and it counts all consecutive statements executed automatically by the system (a statement contained in a loop is counted as many times as it is actually executed). When the number counted reaches a preset value called the safeguard limit, the system automatically suspends execution, informs the user, and returns control to him.

At this stage the user can interact with the system in the context of all local and global variables as they are defined at the point of suspension. He can print or display their values and he can carry out any calculations not involving user-defined programs. To proceed further, the user has the following mode operators at his disposal:

i) `!QUIT` instructs the system that the user does not wish the execution of the suspended sequence of statements to be resumed. He is returned to normal interactive mode.

ii) `!RESUME` instructs the system to resume execution of the suspended sequence of statements, in the context of variables existing when the `!RESUME` command is given; the safeguard counter is set to zero and starts counting again.

iii) `!RESUME (<expr>),` where `<expr>` is a positive integer (rounded by the system if necessary), does the same as `!RESUME`, except that execution will proceed by a number of statements equal to the value of `<expr>` unless it encounters previously either the safeguard limit or the `!HALT` operator (see below). The operator `!RESUME (<expr>)` allows the user to proceed through long subprograms by steps of controlled length.

The system also provides a way to have suspension of execution at any preset point in a user-defined subprogram or in an interactive mode DO loop. This is done by means of the mode operator `!HALT`, which is inserted by the user as a statement when he constructs the subprogram or DO loop. The `!HALT` operator leads to immediate suspension of execution and returns control to the user within the context of the variables then defined. All further actions which the user can then take are the same as described before, including usage of `!QUIT`, `!RESUME`, and `!RESUME (<expr>)`.

10.6 Program editing

Any user-constructed subprogram may be modified by the user in interactive mode by invoking the mode operator `!EDIT`. The user then gives the name of the subprogram to be edited. This causes the statement list to be printed out with line numbers to facilitate the editing process. The latter is performed by use of commands for copy, delete, insert, and end-of-insertion. Also the name of a subprogram can be modified. Protected programs must be deprotected before editing.
11. INPUT-OUTPUT

The SIGMA system is primarily intended to be used in interactive mode on terminals connected to a central computer. The terminal may be regarded as the primary I/O device and all other I/O equipment attached to the central computer as optional secondary I/O devices.

The most common input of data is performed on the terminal keyboard through the assignment statement

\[
\text{name} = \text{expr}
\]

The input of programs through the program statement is described in Section 10. To facilitate the input of long lists of data on the keyboard, one can use a special INPUT statement. Input of data from an external file is done by means of a READ statement.

The most common output is performed on the terminal by a statement of form

\[
\text{PRINT} \ \text{expr list}
\]

\[
\text{expr list} := \text{expr} | \text{expr} , \text{expr list}
\]

The \text{expr} may also contain names of user-defined subprograms (subroutines and functions); the complete program statement (in the sense of Section 10) is then printed. Further output statements are:

\[
\text{WRITE} \ \text{expr list} \quad \text{which outputs on an external file, and}
\]

\[
\text{PUNCH} \ \text{expr list} \quad \text{which punches on cards.}
\]

By use of the mode operator \text{!PRINT} every expression evaluated by the system is automatically printed by the terminal. The mode operator \text{!NOPRINT} returns the system to the mode of operation where printing is done only on request.

Input and output can also be performed in the course of execution of a subprogram or an interactive mode DO loop, either by inclusion of the appropriate statements (INPUT, READ, PRINT, WRITE, PUNCH), or when the execution is suspended by \text{!HALT}, by \text{!RESUME} (\text{expr}) or by the safeguard limit (see Section 10.5).

To avoid the complicated concepts contained in the explicit formatting of numerical data output, an automatic format is provided by the system. Only the maximum number of digits typed is under the control of the user. The mode operator \text{!DIGITS} (\text{expr}) instructs the system as to how many significant digits the output should provide. The value of \text{expr} is rounded if necessary and is interpreted as the number of significant digits required by the user. This value should be between 1 and 15 for single precision numbers, and between 1 and 28 if double precision arithmetic is used.

Arrays are printed row by row.
12. **GRAPHICAL DISPLAY OF ARRAYS**

Section 6.8 described the general principles adopted in SIGMA for the graphical display of arrays. We here present more practical aspects of their implementation.

The use of the operator DISPLAY followed by appropriate arguments puts the system into the *curve display mode* in which successive rows of arrays are displayed as curves on the scope. This graphical output may be transferred to paper when desired by means of hard-copy devices. The arguments following DISPLAY specify the display window, the display pattern (solid, dashed or dotted line, other characters), the ordinate, and the abscissa. Some or all, except the ordinate, may be omitted, in which case the system adopts automatic options for the items omitted.

The usage is as follows:

\[
\text{DISPLAY} \left\{ \begin{array}{l}
\text{<empty>}
\end{array} \right\}, \text{<display list>}
\]

\[
\text{<range pair>} := \left\{ \begin{array}{l}
\text{<range>}
\end{array} \right\} \& \left\{ \begin{array}{l}
\text{<empty>}
\end{array} \right\}
\]

\[
\text{<range>} := \left\{ \begin{array}{l}
\text{<expr>}
\end{array} \right\} \# \left\{ \begin{array}{l}
\text{<empty>}
\end{array} \right\}
\]

\[
\text{<display list>} := \left\{ \begin{array}{l}
\text{<descr> <display item>}
\end{array} \right\}
\]

\[
\text{<display list>} \cdot \left\{ \begin{array}{l}
\text{<display list>} \& \text{<descr> <display item>}
\end{array} \right\}
\]

\[
\text{<descr>} := \left\{ \begin{array}{l}
\text{<empty>}
\end{array} \right\}
\]

\[
\text{<char pattern>} := \text{one or more characters of the alphabet except square brackets}
\]

\[
\text{<display item>} := \left\{ \begin{array}{l}
\text{<expr>}
\end{array} \right\}
\]

\[
\text{<pair>} := \text{<expr>} \& \text{<expr>}
\]

The meaning of the arguments of DISPLAY is the following:

- **<range pair>** determines the window (i.e. the intervals) in which the curves will be displayed. The range pair is interpreted as

\[
Y_{\text{min}} \leq Y \leq Y_{\text{max}} \& X_{\text{min}} \leq X \leq X_{\text{max}}
\]

where \(X\) refers to the abscissa and \(Y\) to the ordinate. If **<range pair>** is

...
absent, the window is automatically set to include all points to be displayed. When one range or one \texttt{expr} in a range are absent, they are chosen automatically so as to include all points compatible with the partial window setting given by the incomplete \texttt{range pair}. All \texttt{expr} in \texttt{range pair} are to be scalars.

\begin{description}
\item[] \texttt{if empty}, the curves of the display items are displayed as continuous lines (linear interpolation). Various choices of the character pattern lead to display in dashed or dotted lines or in lines constructed of alphabet characters.
\end{description}

\begin{description}
\item[] \texttt{if a pair }A \% B, the rows of A are taken as ordinate and those of B as abscissa. The 1\textsuperscript{st}, 2\textsuperscript{nd}, \ldots row of A is displayed against the 1\textsuperscript{st}, 2\textsuperscript{nd}, \ldots row of B. If A has more rows than B, the last row of B is left in the abscissa. The 1\textsuperscript{st} display item in the first \texttt{DISPLAY} command must always be a pair; subsequent items can be single expressions whose rows are then displayed as ordinates against the abscissa left by the last displayed pair.
\end{description}

The system provides a number of \emph{display mode operators} which allow the user to set the mode of display until another mode is selected by him. The main ones are:

\begin{itemize}
\item \texttt{LINX} \hspace{1cm} \texttt{LOGX} \hspace{1cm} \texttt{LINY} \hspace{1cm} \texttt{LOGY}
\end{itemize}

These operators select linear or logarithmic scales for the abscissa (X) and the ordinate (Y).

\begin{itemize}
\item \texttt{AXES} \hspace{0.5cm} \texttt{NOAXES} select whether axes are displayed or not.
\item \texttt{FRAME} \hspace{0.5cm} \texttt{NOFRAME} select whether the display is surrounded by a frame or not.
\item \texttt{GRID} \hspace{0.5cm} \texttt{NOGRID} select whether the display is covered or not be a grid of lines parallel to the axes.
\end{itemize}

Examples of displays are given in the next section.
13. **EXAMPLES**

We first illustrate the display facilities of SIGMA by showing two simple functions in various display modes. We then turn to examples of problems which are mathematically trivial but have enough computational complexity to show the power and compactness of the language. The reader may convince himself of these features by writing the corresponding FORTRAN programs.

13.1 **Some examples of the graphical facilities**

We compute the function

\[ f(x) = 4 + \sin x \cdot e^{x/2} \quad -\pi \leq x \leq \pi \]  

and wish to display it. We type

\begin{verbatim}
X = ARRAY (100, -PI # PI)
F = 4 + SIN(X) * EXP(X/2)
DISPLAY F $ X
\end{verbatim}

This results in the display shown in Fig. 1. Scale and position of the coordinate origin are automatically chosen: the system scans through all values of F and X and then sets the scale on the axes such that no point is lost while the curve appears in reasonable size. In the present case the coordinate axes cross at x = 0, y = 2; if we had chosen to display sin x for 1000 ≤ x ≤ 1000 + 2π, they would not cross at x = 0 but in the region of x = 1000, and the coordinate origin (x = y = 0) would lie on the left far outside the screen.

We might wish to have the same in logarithmic scale and we might also wish a grid. The commands are [see Sections 6.8, 12, 5.1 iii)]

```
:LOGY
:GRID
DISPLAY F
```

resulting in Fig. 2. Note that F is automatically displayed over X; the system remembers that the abscissa had previously been identified with X. If we would like to see F displayed over another, also previously defined variable Y, then we would type DISPLAY F $ Y.

Finally we wish to see F and its square together in one single figure without grid and in linear scale; F as full line, F² broken. We type

```
:NOGRID
:LINY
F2 = F ** 2
DISPLAY F, [.-5] F2
```

This results in Fig. 3 (remember that the abscissa is still X). The symbol [.-5] preceding F2 tells the system that F² should be displayed in the form of a broken line — — — — — —

*) All figures shown in this section are polaroid photographs of a Tektronix display screen. For technical reasons we reproduce them here as negatives.
Fig. 1 Automatic display of the function
\[ f(x) = 4 + \sin x \cdot \exp(x/2) \] for \(-\pi \leq x \leq \pi\)

Fig. 2 Automatic display of the function \(f(x)\) (as in Fig. 1) with grid and logarithmic scale on the ordinate
with length of period 5. Had we typed \([-\text{-}]\) the line would have a five times shorter period: \(-\text{---}\). Here the scale is automatically chosen such that both curves find a place on the screen.

![Graph](image)

**Fig. 3** Automatic display of the function \(f(x)\) (as in Fig. 1) together with its square (linear scale, no grid) \(f(x)\) as full line, \([f(x)]^2\) as broken line.

This example shows the necessity of the window facility: if a curve has great variations, then the display of the whole curve will make details invisible; one can then choose a window to display only parts of it. We illustrate this by typing

\[
X = ARRAY (300, 0 \neq 8)
\]

\[
G = \text{COSH} (X) + \text{SIN} (1/(.1 + X^X))
\]

DISPLAY G ¥ X

which results in Fig. 4.

The scale is chosen automatically such that no point of \(G\) is lost; in that scale one cannot see the sine oscillations. One could, of course, recompute \(G\) over a smaller interval, but we have the simpler solution of enlarging a part of \(G\) without recomputing it. We simply type a display command with window specification, namely

\[
\text{DISPLAY} (0 \neq 5 \% 0 \neq 3) G , [+] G \quad (\text{see Section 12}).
\]

The result is the display shown in Fig. 5. The full line is the linear interpolation between the points actually computed, the latter being shown by the crosses \(+\). Had we typed \([0] G\) we would have obtained small circles instead of crosses.
Fig. 4 Automatic display of the function 
\[ g(x) = \cosh x + \sin \left( \frac{1}{0.1 + x^2} \right) \] for \( 0 \leq x \leq 8 \).
In this scale the sine oscillations are invisible.

Fig. 5 Display with preset window 
\[ 0 \leq y \leq 5, \quad 0 \leq x \leq 3 \]
of the same function \( g(x) \) as in Fig. 4, without recomputing the function. Details lost in Fig. 4 become visible here. Crosses indicate computed points; full line obtained by automatic linear interpolation.
13.2 Discussion of a curve; find extrema

We choose a simple example

\[ g(x) = e^{-x} \sin ax + bx^2 \]  \hspace{1cm} (2)

and define accordingly

\begin{verbatim}
FUNCTION F(X)
  GLOBAL A,B
  F = EXP (-X) * SIN (A*X) + B * X * X
END
\end{verbatim}

To obtain an idea about \( f(x) \) we first compute it for \( 0 \leq x \leq 5 \) by typing, for example,

\begin{verbatim}
GLOBAL A,B
A = 8
B = .01
X = ARRAY (100, 0 \neq 5)
G = F(X)
DISPLAY G \& X
\end{verbatim}

The result is Fig. 6. The display shows clearly five maxima; it is not obvious whether there is a sixth one or not.

![Graph of the function](image)

*Fig. 6* The function

\[ g(x) = e^{-x} \sin ax + bx^2 \]

for \( 0 \leq x \leq 5 \); \( a = 8 \); \( b = 0.01 \)

Problem: value and position of fifth maximum; is there a sixth one?
Problem:  i) find position and value of fifth maximum;
   ii) is there a sixth one?

We solve this by a mixed technique involving curve display as well as alphanumeric display.

By inspection of Fig. 6 one sees that the fifth maximum lies somewhere in the interval $3.3 \leq x \leq 3.4$. Hence we type

\[
\begin{align*}
X &= \text{ARRAY (10, 3.3 & 3.4)} \\
G &= F(X) \\
\text{DISPLAY G & X}
\end{align*}
\]

and obtain Fig. 7. Note that we have reduced the number of points to 10 in order to save computing time. We could proceed in this way and look now at the interval $3.35 \leq x \leq 3.37$, display and then narrow further. Instead we may decide to use the following approach: we define a subroutine which prints the values of $G$ and $X$ of that computed point which lies nearest to the true $G_{\text{max}}^{(5)}$ and $X_{\text{max}}^{(5)}$. We type

\[
\begin{align*}
\text{SUBROUTINE FINDMAX (F,X)} & \quad \text{See Section} \\
F1 &= \text{DELTA (MAX(F) - F)} & 10, 10.1 \text{ to } 10.3 \\
FMAX &= F1 \times F & 7.11 \\
XMAX &= F1 \times X \\
\text{PRINT} &\, X, XMAX, FMAX & 11 \\
\text{END}
\end{align*}
\]

Fig. 7  The function $g(x)$ of Fig. 6 recomputed in 10 points over $3.3 \leq x \leq 3.4$; the maximum should lie in the region of $3.35 \leq x \leq 3.37$. Full line shows linear interpolation.
Explanation:

MAX(F) creates an array with the structure of F, all of its components being equal to the maximum value found in F; therefore MAX(F) - F has exactly one component equal to zero at the place of the maximum value of F, all others being > 0. DELTA (see Section 7.11) then creates an array F1 which is zero except at the place where F = MAX(F); there it has the value 1. Hence XMAX will be an array made out of zeros, except at the place of F = MAX(F), where it has the value of the corresponding X; similarly FMAX has one component different from zero, its value is MAX(F).

Now we type

\[
\begin{align*}
X &= \text{ARRAY (10, 3.35 \# 3.37)} \\
G &= F(X) \\
\text{CALL FINIMAX (G,X)}
\end{align*}
\]

See Section 6.10

\[
\begin{align*}
G &= 10.4
\end{align*}
\]

The display shows the numerical values of X, XMAX, FMAX as in Fig. 8. The maximum lies somewhere near

\[
X = 3.352, \quad F = .1471
\]

Fig. 8 Alphanumeric display of the array X together with XMAX and FMAX; the latter two have zeros everywhere except at the place where the maximum occurs; there they equal X_{(max)} and G_{(max)}, respectively.

Inspecting X now allows one to choose the next interval as 3.351 ≤ x ≤ 3.353 for instance. It is clear that in this way we arrive in a few steps at a determination of X_{(max)} and G_{(max)} within the precision limits of the computer.
The next question, whether there is a sixth maximum, is better decided graphically by typing

\[
X = \text{ARRAY} (10, 4.1 \neq 4.3) \\
G = F(X) \\
\text{DISPLAY } G \% X
\]

The resulting Fig. 9 shows that indeed there is a sixth maximum. If only B would have been slightly greater than .01, this maximum might have disappeared. In fact, typing

\[
B = .013 \\
X = \text{ARRAY} (100, 0 \neq 5) \\
G = F(X) \\
\text{DISPLAY } G \% X
\]

produces the display shown in Fig. 10 where it still is not clear whether there is a sixth maximum or not (compare with Fig. 6). But now

\[
X = \text{ARRAY} (10, 4.2 \neq 4.4) \\
G = F(X) \\
\text{DISPLAY } G \% X
\]

results in Fig. 11, which settles the question.

---

**Fig. 9** The function \( g(x) \) of Fig. 6 recomputed in 10 points over \( 4.1 \leq x \leq 4.3 \), showing that there is a sixth maximum (which was not clear from Fig. 6). Full line shows linear interpolation.
Fig. 10  The function $g(x)$ of Fig. 6 with $b = 0.013$ instead of 0.01. It is hard to decide whether there is still a sixth maximum or not.

Fig. 11  The function $g(x)$ of Fig. 10 ($b = 0.013$) recomputed over $4.2 \leq x \leq 4.4$, showing that there is no sixth maximum any more. Full line shows linear interpolation.
13.3 Fourier analysis

We construct a subroutine which calculates Fourier approximations to a function \( f(x) \) given over an interval \( x_0 \leq x \leq x_1 \). We call the subroutine \textsc{fourier} \((F,X)\) and use the formulae

\[
f(x) = \lim_{n \to \infty} f^{(n)}(x) \tag{3}
\]

\[
f^{(n)}(x) = \frac{a_n}{2} + \sum_{k=1}^{n} a_k \cos \frac{k \pi x}{L} + \sum_{k=1}^{n} b_k \sin \frac{k \pi x}{L} \tag{4}
\]

\[
a_k = \frac{1}{L} \int_{x_0}^{x_1} f(x) \cos \frac{k \pi x}{L} \, dx ; \quad \xi = \frac{1}{2} (x_1 - x_0) \tag{5}
\]

\[
b_k = \frac{1}{L} \int_{x_0}^{x_1} f(x) \sin \frac{k \pi x}{L} \, dx . \tag{6}
\]

The integrals will here be approximated by simple sums. As the function to be approximated is given only at a finite number of points, it is useless to push \( n \) to very high values; we limit the approximation to \( n_{\text{max}} = \text{NCO}(F)/4 \) so that for the highest wave there are four points per full oscillation. After each step the routine displays \( f(x) \) and \( f^{(n)}(x) \) and prints the number \( n \). Then it stops to allow the user to inspect the approximation. At this place the user disposes of all global and local variables presently defined. If the displayed curves do not suffice to decide whether the approximation is good enough, the user may compute in manual mode

i) the difference between the function and the approximation,

ii) the mean quadratic error,

or whatever quantity he likes. He may then continue to the next approximation (by typing \$\textsc{resume}\$) or exit from the subroutine (command \$\textsc{quit}\$). The subroutine requires, of course, that \( F \) and \( X \) have equal NCO vectors.
<table>
<thead>
<tr>
<th>SUBROUTINE FOURIER (F,X)</th>
<th>See Section</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBAL A,B,FF</td>
<td></td>
<td>9.5</td>
</tr>
<tr>
<td>A = ARRAY (NCO(X)/4) * 0</td>
<td></td>
<td>6.10, 6.2</td>
</tr>
<tr>
<td>B = A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L = (SMAX(X) - SMIN(X))/2</td>
<td></td>
<td>7.11</td>
</tr>
<tr>
<td>D = 2/(NCO(X) - 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 = D * TRACE (F)</td>
<td></td>
<td>7.7</td>
</tr>
<tr>
<td>FF = A0/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DO 1 K = 1, NCO(A)</td>
<td></td>
<td>3.4</td>
</tr>
<tr>
<td>T = K * PI * X/L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C = COS (T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S = SIN (T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(K) = D * TRACE (F * C)</td>
<td></td>
<td>6.9</td>
</tr>
<tr>
<td>B(K) = D * TRACE (F * S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FF = FF + C * A(K) + S * B(K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISPLAY F % X, [-.-] FF</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>PRINT K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!HALT</td>
<td></td>
<td>10.5</td>
</tr>
<tr>
<td>1 CONTINUE</td>
<td></td>
<td>3.4</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Remarks to SUBROUTINE FOURIER**

We make FOURIER a subroutine and not a function, because it not only returns a value but does several other things in addition.

1) A and B are arrays which will contain the computed coefficients $a_k$ and $b_k$; FF will be the current approximation. These variables should be available after exit from the subroutine, therefore they are declared global. They must also be declared global in manual mode before calling FOURIER.

2) We limit the analysis to $n \leq 1/4 \cdot (\text{number of points in } X)$. If NCO(X)/4 is not an integer, the system rounds it automatically.

3) $L = \lambda$ is the length of the half-interval.

4) D stands for the differential $dx/\lambda$.

5) This command calculates an approximation to $a_0 = \int_{x_0}^{x_1} f(x) \, dx/\lambda$.

6) Mean value of $f(x)$.

7) $T = k\pi x/\lambda$ is the argument of sine and cosine; it is, of course, an array.
8) Computes \( a_k = \int_{x_1}^{x_2} f(x) \cos (k \pi x / L) \, dx / L \).

9) Displays the original function together with its current approximation and prints the current value of \( k \); i.e., \( F \) equals \( f^{(n)} \) and \( k \) equals \( n \) in the sense of Eqs. (3) to (6).

10) The system stops and the user can inspect the result. He may, for instance, type PRINT A,B to see the numerical values of the Fourier coefficients computed thus far (those which are not yet computed are printed as zeros).

As an example we shall Fourier-analyse the function given by Eqs. (1) and shown in Fig. 1. We construct it once more (because in the meantime \( F \) and \( X \) have been assigned other values in computing the other examples):

\[
T = \text{ARRAY (100, -PI \, \pi)}
\]
\[
G = 4 + \sin(T) \times \text{EXP}(T/2)
\]

Note that we have chosen different names to illustrate the notion of formal and actual parameters (Sections 10.2, 10.3) and of global and local variables (Section 9.5.1); the appearance of a local variable \( T \) inside \textit{FOURIER} does not lead to confusion with the \( T \) which we will use as an actual parameter when calling \textit{FOURIER}:

\[
\text{GLOBAL A,B,FF}
\]
\[
\text{CALL FOURIER (G,T)}
\]

This results in displaying \( F \) (i.e., our \( G \)) together with the first approximation \( FF = F^{(1)} \) and then stopping. We type !RESUME and obtain the display shown in Fig. 12 (\( K = 2 \) is not

---

Fig. 12 Fourier analysis of the function (same as in Fig. 1)

\[ g(x) = 4 \times \sin x \times \exp(x/2) \quad \text{for} \quad \pi \leq x \leq \pi 
\]

- full line: original \( g(x) \)
- broken line: second order Fourier approximation.
Fig. 13 Same as in Fig. 12; the broken line shows here the 10th order Fourier approximation.

Fig. 14 Same as Figs. 12-13; broken line shows 25th order Fourier approximation.
printed in our figure). Going on a few more steps leads to Fig. 13 (here \( K = 10 \); not printed in our figure) and finally to Fig. 14 (\( K = 25 \)). As we had defined \( T \) as an array of 100 components, the subroutine will come to an end here. Better approximations would require either more points or a more refined subroutine using a better integration than just a sum (TRACE).

13.4 **Quasi-three-dimensional display**

Let \( z(x,y) \) be a function defined over a rectangular domain \( x \times y := (x_0 \leq x \leq x_1; y_0 \leq y \leq y_1) \) of the \( x-y \) plane. Suppose then

\[
X = \text{ARRAY} (N, X0 \neq X1) \\
Y = \text{ARRAY} (M, Y0 \neq Y1) \\
F = \text{TOP} ("Y,X : Z(X,Y)"
\]

where \( Z(X,Y) \) is some expression or function calculating \( z(x,y) \).

One way to inspect the result is to display \( F \), row by row, over abscissa \( X \), or column by column over the abscissa \( Y \):

\[
\text{DISPLAY } Z \times X \\
\text{DISPLAY } Z \times Y
\]

(see Sections 6.8, 12)

will display \( M \) curves \( z(x_i,y) \); \( i = 1 \ldots M \) as functions of \( x \), while

\[
\text{DISPLAY } Z \times Y
\]

(see Section 7.6)

will display \( N \) curves \( z(x,y_i) \); \( i = 1 \ldots N \) as functions of \( y \). These two methods correspond to projecting the sets of curves \( z(x,y_i) \) and \( z(x_i,y) \) onto the \( x-z \) and \( y-z \) planes, respectively.

This might not be sufficient to obtain an intuitive feeling for the function \( z(x,y) \), and we might wish to look at the surface \( z(x,y) \) in the direction of polar angles \( \theta, \phi \) (the above projections are then the special cases \( \theta = 90^\circ, \phi = -90^\circ \) and \( 0^\circ \), respectively).

Let \( \hat{n} \) then be the unit vector with components \( \sin \theta \cos \phi, \sin \theta \sin \phi, \cos \theta \); we wish to construct a projection of the surface \( z(x,y) \) onto a plane orthogonal to \( \hat{n} \), and to look at this projection from the direction \(-\hat{n}\); in fact we wish to project the two sets of curves \( z(x_i,y) \) and \( z(x,y_i) \), \( i = 1 \ldots M, \ k = 1 \ldots N \). In the plane of projection we chose two axes \( y' \) and \( z' \) such that \( z' \) is vertical and coincides with the projection of the original \( z \)-axis. The projection formulae are then

\[
y' = -x \sin \phi + y \cos \phi \\
z' = -\cos \theta (x \cos \phi + y \sin \phi) + z \cos \theta
\]

(7)

We thus transform our variables and display the projection. As a guide for the eye we wish to project also the three original axes \( x, y, z \) and place them such that \( z(x,y) \) seems to fall in the first octant \( (x, y, z \geq 0) \). [If we would place the axes at their true position, then a function \( z(x,y) \) defined over \( 1000 \leq x, y \leq 1001 \) would yield a display containing the origin and the domain \( x \times y \); the latter would shrink to a point.] We further give
these axes, before projecting them, the lengths \( x_1 - x_2, y_1 - y_2, z_{\text{max}} - z_{\text{min}} \) so that they define a rectangular box, each face of which touches the surface \( z(x, y) \). The curves \( z(x, y) \) shall be displayed as full lines, those of \( z(x, y) \) as broken lines. The user shall only give the angles \( \theta \) and \( \phi \) in degrees and call a subroutine resulting in the quasi-three-dimensional display.

The power of the SIGMA language is well illustrated by the shortness and transparency of the corresponding subroutine VIEW.

<table>
<thead>
<tr>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBROUTINE VIEW (X, Y, Z, THETA, PHI)</td>
</tr>
<tr>
<td>DEGR = 2 * PI/360</td>
</tr>
<tr>
<td>T = THETA/DEGR</td>
</tr>
<tr>
<td>P = PHI/DEGR</td>
</tr>
<tr>
<td>CT = COS (T)</td>
</tr>
<tr>
<td>ST = SIN (T)</td>
</tr>
<tr>
<td>CP = COS (P)</td>
</tr>
<tr>
<td>SP = SIN (P)</td>
</tr>
<tr>
<td>A = ARRAY (3&amp;2)</td>
</tr>
<tr>
<td>AX = SMIN (X) * A</td>
</tr>
<tr>
<td>AX (1,2) = SMAX (X)</td>
</tr>
<tr>
<td>AY = SMIN (Y) * A</td>
</tr>
<tr>
<td>AY (2,2) = SMAX (Y)</td>
</tr>
<tr>
<td>AZ = SMIN (Z) * A</td>
</tr>
<tr>
<td>AZ (3,2) = SMAX (Z)</td>
</tr>
<tr>
<td>AY1 = -AX * SP + AY * CP</td>
</tr>
<tr>
<td>AZ1 = -CT * (AX * CP + AY * SP) + AP * ST</td>
</tr>
<tr>
<td>Y1 = TOP (&quot;Y,Y : Y * CP - X * SP&quot;)</td>
</tr>
<tr>
<td>Z1 = TOP (&quot;Y,X : -CT * (Y * SP + X * CP)&quot;) + Z * ST</td>
</tr>
<tr>
<td>DISPLAY [.-, ( \text{Y1, AZ1, AY1} )</td>
</tr>
<tr>
<td>( \text{Y1 = TP (Y1)} )</td>
</tr>
<tr>
<td>( \text{Z1 = TP (Z1)} )</td>
</tr>
<tr>
<td>( \text{Z1 &amp; Y1} )</td>
</tr>
<tr>
<td>PRINT THETA, PHI</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>
Remarks:

1) Angles are transformed into radians and their sine and cosine calculated.

2) The three axes defining a rectangular box enclosing the surface \( z(x,y) \) are computed, transformed, and stored under the names AY1, A21 for later display. Note the use of 3 by 2 arrays to deal with all three axes simultaneously.

3) The coordinates \( x, y \) and \( z(x,y) \) are transformed; note the use of \( M \) by \( N \) arrays to deal simultaneously with all points representing the function \( z(x,y) \) together with the independent variables \( x \) and \( y \). As \( z(x,y) \) is represented by a two-dimensional array over the domain \( x \otimes y \), the TOP operator has to be used to properly combine \( z_{1k} = z(x_1,y_k) \) with \( x_1 \) and \( y_k \). The curves \( z(x_1,y) \) will be displayed as broken lines, the projected coordinate axes as full lines; note that one single display command causes the display of the first row of \( Z1 \) over the first row of \( Y1 \), then the second row of \( Z1 \) over the second of \( Y1 \), and so on, followed by the display of the array \( A21 \) over \( AY1 \) in the same way (see Section 6.8).

4) Transposing the arrays and displaying again results in curves \( z(x,y_k) \) drawn as full lines. Finally, \( \theta \) and \( \phi \) are printed (in degrees).

Utilization:

We take as an example the function

\[
\begin{align*}
f(x,y) &= 2 \exp \left[ -5 \left( x - \frac{1}{2} \right)^2 - 5 \left( y - \frac{1}{2} \right)^2 \right] \\
&\quad -2 \exp \left[ -5 \left( x + \frac{1}{2} \right)^2 - 5 \left( y + \frac{1}{2} \right)^2 \right] \\
&\quad -y^2 + x^2
\end{align*}
\]

and construct it by defining an auxiliary function

\begin{verbatim}
FUNCTION GAUSS (X,Y,C)
GAUSS = TOP ("Y,X : EXP (-C * X * X - C * Y * Y)")
END
\end{verbatim}

Note that here and in VIEW we first write \( Y \), then \( X \) in the name list of the topological product, in order to have \( X \) varying along rows and \( Y \) along columns.

Now we obtain our function \( f(x,y) \) by fixing the \( x \otimes y \) domain and computing as follows:

\[
\begin{align*}
X &= ARRAY (20, -1 \neq 1) \\
Y &= ARRAY (15, -1 \neq 1) \\
F &= TOP ("Y,X : Y * Y - X * X") \\
F &= 2 * GAUSS (X-.5 , Y-.5 , 5) - F \\
F &= F - 2 * GAUSS (X+.5 , Y+.5 , 5)
\end{align*}
\]

To show the surface \( f(x,y) \) for \( \theta = 15^\circ, \phi = 40^\circ \):

\begin{verbatim}
CALL VIEW (X, Y, F, 15, 40)
\end{verbatim}

Figures 15 to 18 show the result for various angles.
Fig. 15 Quasi-three-dimensional display of the surface given by Eq. (8) and seen under the angles $\theta = 15^\circ; \phi = 40^\circ$.

Fig. 16 Same as Fig. 15; $\theta = 15^\circ; \phi = 20^\circ$. 
Fig. 17  Same as Fig. 15;  $\theta = 15^\circ$;  $\phi = -20^\circ$

Fig. 18  Same as Fig. 15;  $\theta = 15^\circ$;  $\phi = -45^\circ$
14. ACKNOWLEDGEMENTS

The authors of this report are indebted to many people who contributed in various ways to the development of the GAMMA and SIGMA projects. C. Vandoni, as chief implementor of both systems, acknowledges, in particular, the important contributions made by F. Wittmann and J. Moszczynski. His and J. Reinfeld's thanks are also due to H. Jedlicka for contributing to the implementation work at the University of Georgia. R. Hagedorn and L. Van Hove, theoretical physicists and proponents of the two projects, wish to express their gratitude to M.G.N. Hine and G.R. Macleod for giving the necessary support at CERN, as well as to G.J. Culler and B.D. Fried for introducing them to the principles and applications of the Culler-Fried system for on-line scientific computing at the University of California. In the conceptual development of the systems, and especially in the definition of aims, data structures, and language of SIGMA, the four authors profited greatly from suggestions and remarks contributed by many users and based on their experience with concrete scientific applications; special mention should here be made of R. Chatwin, M.G. Doncel and M. Roos.

REFERENCES


