SIGMA is a language for interactive numerical mathematics. Dynamically managed multi-dimensional arrays are the basic data types of SIGMA. The concept of array compatibility in binary operations is extended to allow array combinations in the sense of a topological direct product. Subscripting is extended to allow arrays as subscripts. A simple but flexible display operator permits easy generation of graphic displays as an alternative to numerical printout of results.

(To be presented to the IFIP CONGRESS 77  
8-12 August 1977, Toronto, Canada)

*) Dept. of Computer Science, University of Wollongong, Wollongong, N.S.W., Australia.
SIGMA is a language for interactive numerical mathematics. Dynamically managed multi-dimensional arrays are the basic data types of SIGMA. The concept of array compatibility in binary operations is extended to allow array combinations in the sense of a topological direct product. Subscripting is extended to allow arrays as subscripts. A simple but flexible display operator permits easy generation of graphic displays as an alternative to numerical printout of results.

1. INTRODUCTION

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

(i) the basic data structure consists of automatically and dynamically managed n-dimensional rectangular arrays

(ii) the calculational operators of SIGMA closely resemble the operations of numerical mathematics. In particular array transformations available through the subscript notation of numerical mathematics are provided by analogous array-handling operators

(iii) there are convenient facilities for the graphical display of arrays in the form of curves or sets of curves.

SIGMA was defined in 1971[1] by joining the GAMMA development of CERN[2] and the AMTRAN effort of the University of Georgia.[3] Two essentially independent implementations of SIGMA exist at the moment. The CERN implementation for CDC 6000 series computers was developed by C.E. Vandoni and collaborators. The second implementation is by R. Miller at the University of Saskatchewan[4] for IBM 360/370 series computers. The Saskatchewan version was adopted for use on UNIVAC 1100 series computers by P. Castle at the University of Wollongong.

A language is simple to use and easy to learn if its use requires only a few general rules with no exceptional cases or special features. This was recognised in the development of AMTRAN[5] and it is encouraging to find that such ideas are supported by studies of natural languages.[6]

An algorithm is best built from well-defined programming mechanisms or action clusters in the terminology of Naur[7]. Similarly, a programming language must contain a reasonably well-defined set of language mechanisms or features. The description of SIGMA given in the subsequent sections shows how SIGMA satisfies such a set of essential language mechanisms and highlights the areas in which SIGMA extends the usual features or breaks new ground.

The language closest to SIGMA in spirit if not in syntax or semantics is APL. Also the BASIC language is undergoing active development and each new BASIC compiler often provides or extends another array handling feature.[8]

2. DATA STORAGE AND MANIPULATION

Data storage and manipulation is based on a conventional assignment statement

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

where the result of the expression is assigned to the name.

The expression may include arithmetical, relational or logical operators because Boolean truth values are represented by zero for FALSE and one for TRUE. The expression may also include any prefix operator provided by SIGMA, defined as a user-generated function subprogram or attached by the user as an other-than-SIGMA-language subprogram.

Since the basic data type of SIGMA is an n-dimensional rectangular array, the result of any expression will be an array without any explicit loop structures to process individual components.

3. CONTROL STRUCTURES

Control structures of SIGMA are quite conventional and FORTRAN-like in appearance. The conditional statement allows two forms

IF \( \text{expression} \) THEN \( \text{statement} \) ELSE \( \text{statement} \)
IF \( \text{expression} \) \( \text{statement} \)

The \( \text{statement} \) will be executed if the value of the expression is 1 (TRUE). If the value of the expression is an array, the IF condition is satisfied if all components of the value are 1 (TRUE). A SIGMA function ANY is provided which gives the scalar value TRUE if any component of its argument is TRUE so that

IF (ANY (\( \text{expression} \))) statement

will execute the statement if any component of the value of the expression is 1 (TRUE).

A loop structure is provided by a FORTRAN-like DO statement generalised to allow non-integer loop index and increments. A GOTO statement is provided for unconditional branching. Statement labels are 1 to 5 digit integers hence the following form of GOTO

GOTO \( \text{expression} \)

permits program controlled branching where the result of the expression is rounded and taken to denote a label. Array result is meaningless in this case.
4. DATA TYPES

Data types of SIGMA are arrays of real numbers, complex numbers or string characters. Except for input and output, string characters are identical to small numbers because string characters are represented by the numerical internal character code of the computer. Hence string character manipulation may involve any of the number handling operators.

The transition to complex numbers is automatic for the result of any operator which generates a complex result from a real argument. For example, the square root of any array with at least one negative component will generate an array of complex numbers as a result. The transition from a complex array to a real array is by explicit application of a conversion operator.

As a rule, an operator which needs an integer result will round a real result before use. Generalizing this rule to complex numbers: an operator which needs a real argument will ignore the imaginary part of a complex argument.

The basic idea which guided the definition of the validity of complex operands for all operators of SIGMA was smooth ("analytic") continuation from the real axis into the complex plane.

5. DATA STRUCTURE

SIGMA supports a single data structure: dynamically managed n-dimensional rectangular arrays. An array consists of a data-type-tag, a dimension vector and a sequence of elements stored in row-major order. The dimension vector is accessible to the user: the NOD (number of components) operator will give the dimension vector of its argument as its result; e.g., if A is a 5 * 3 * 7 (three dimensional) array, NOD (A) gives the vector 5 3 7 and NOD (NOD(A)) gives the scalar 3, i.e., the dimension of A.

Arrays are generated by assignment, concatenation or the ARRAY operator. Concatenation (represented by the ampersand &) is defined as a row extension; it affects only the column index of an array. For example:

X = 2
X = X & 3*X
Y = -5 & X & X

generates the five component vector Y such that

Y1 = -5  Y2 = 2  Y3 = 6  Y4 = -2  Y5 = -6

Arrays of more than one dimension are generated by the ARRAY operator which has two arguments. For example,

R = ARRAY( <arg1>,<arg2> )

generates and stores an array named R such that the NOD of R is given by the rounded result of arg1 and the elements of R in index order are supplied by the elements of the result of arg2 whose elements are taken in their index order. For equally spaced arrays the range operator (represented by #) permits the specification of the end points only. For example, an equally spaced 101 component vector from zero to 2π is generated by the statement

Y = ARRAY(101, 0 # 2 * PI)

Subscripting is generalized to allow subscripts which are one dimensional arrays. All components of a dimension are denoted by a "missing subscript" in that dimension. For example if

X = ARRAY (34, ARRAY(12, 1 # 2))
Y = X (1,3)
Z = X (382, 16261)

then Y is a scalar whose value is 3 while Z is a 2 * 3 array containing the elements.

\[
Z = \begin{pmatrix}
X_{34} & X_{52} & X_{31} \\
X_{52} & X_{22} & X_{21}
\end{pmatrix} = \begin{pmatrix}
9 & 10 & 9 \\
5 & 6 & 5
\end{pmatrix}
\]

Combined with the array operators, multicompont subscripting allows powerful array handling with concise commands. For example, an n x n Hilbert matrix is generated by the function

FUNCTION HILM (N)
H = ARRAY (N & N)
H (1,) = SLM (H(1,))
HILM = 1/SLM (TP(H))
RETURN
END

Some dynamic array manipulation is possible through matrix operations of some BASIC compilers. APL provides for general manipulation of rectangular arrays but it restricts arrays which can participate in a binary operation to arrays which have identical dimension vectors or situations where one of the arrays is a scalar. SIGMA has extended the conformability of arrays which may participate in binary operations to permit the automatic generation of topological products in the following way:

5.1 Topological arithmetic

Two arrays are conformable for binary operations if and only if corresponding elements of their NOD vectors are either equal or one of the elements of an unequal pair is equal to one.

The NOD of the result is found by taking the larger element of each pair of corresponding NOD elements.

The above two rules permit the formation of topological direct products in a very simple way. A function of two variables, z = f(x,y) may be formed by creating Y with NOD equal to ny and X with NOD equal to nx & 1 so that the result of any expression containing X and Y will generate the nx * ny array Z such that Z = f(X,Y). For example, the statements

Y = ARRAY (5, 1 # 5)
X = ARRAY (35, Y)
Z = 10 * X + Y

will generate the 3 x 5 array

\[
\begin{array}{cccccc}
11 & 12 & 13 & 14 & 15 \\
21 & 22 & 23 & 24 & 25 \\
31 & 32 & 33 & 34 & 35
\end{array}
\]

6. INPUT-OUTPUT HANDLING

Input from cards or user terminal is in free format. Output to printer or user terminal is in a fixed format provided by the SIGMA system.

Both data files and files with user-written SIGMA programs may be read or written in the above format. Data files may also contain binary data which are read directly into array elements, or written without conversion.

*) The TP operator serves to transpose the argument array - see Section 10.
For added convenience the system remembers each abscissa as it is entered until the abscissa is replaced by a new abscissa in a subsequent display statement. Hence the above statement could also be written more briefly as

DISPLAY SXXY, SX**2, COS(X)**2

On a Tektronix graphics terminal (but not on the printer or teletype) one or more curves can be added to an existing display by entering an incomplete statement consisting of a display list within the DISPLAY command. In the last example an entry

SX or SXIX

will retrace the sine curve and make it flash momentarily while

COS(X)

will add the cosine curve to the display using the abscissa determined by the last display pair entered and the scale factors established by the last explicit use of the DISPLAY command.

7.1 Scaling control

Automatic scaling may be suppressed by the command INOSCALE. Any DISPLAY statements following the execution of INOSCALE will draw the new picture according to the scale established by the last DISPLAY statement executed before INOSCALE. Automatic scaling may be resumed by executing a RESCAL command after which all subsequent DISPLAY statements will perform automatic scaling.

In any one DISPLAY statement scaling may be prescribed on one or both axes by including in parentheses between DISPLAY and the display list a pair of ranges separated by the operator and using % to separate ordinate and abscissa ranges. For example, both scales, ordinates only and abscissa only, are prescribed by including suitable numbers for the limits in the following

DISPLAY (YMIN# YMAX# XMIN# XMAX#) YXX
DISPLAY (YMIN# YMAX#) YXX
DISPLAY (% XMIN # XMAX #) YXX

With a prescribed scale only points within the prescribed range will be displayed. A geometrically true picture which displays all the points of the display list may be obtained by

DISPLAY (=) YXX

7.2 Texture control

The display normally connects consecutive points with an unbroken straight line. Reasonably rounded contours are obtained by choosing adjacent points sufficiently close together.

This default may be overridden for any display-list item by including a texture descriptor in square brackets before the display-list item concerned. For example,

DISPLAY [*] SXXY, [A]COS(X)

will plot a star * for each point of the sine curve and the letter A for each point of the cosine curve. Any symbol is allowed inside the square bracket except the square bracket itself. Hence if we execute

SX**2, [*] COS(X)**2

in addition to the above then the first curve will be added as a continuous line while each point of the second curve will be denoted by a *.

Various further facilities are available, for instance

DISPLAY [BAR] YXX includes error-bars at each point.

In the error-bar display Y and X must be two dimensional arrays where each first row specifies the points and each second row should include the corresponding errors to be drawn for y and x respectively.

A histogram display facility is also available:

DISPLAY [HIST] YXX displays Y as a histogram. Example:

X = ARRAY (21,9#11)
Y = EXP (-4*X-10)**2
DISPLAY [HIST] YXX

Fig. 4. Histogram resulting from the above three commands.

7.3 Multidimensional arrays

If both sides of a display pair contain more than one row, corresponding rows are displayed against each other in index order.

If only one side of a display pair contains more than one row, it determines the number of curves displayed. For example,

X = ARRAY(100, 0.04*PI)
Z = ARRAY(581, .24,.48,.68,.88 1)
M = SIN(X)*Y
DISPLAY MXX, -MXX
Graphic display of results is an essential feature of SIGMA and a lot of work was spent in the streamlining of the syntax and semantics to make our graphic DISPLAY command as user-friendly as possible without affecting the scope and flexibility of the available graphics.

7. GRAPHIC DISPLAYS

The simplest possible graphic display needs an ordinate \((y\)-coordinate) vector and an abscissa \((x\)-coordinate) vector. These must be specified by the user as a **display pair**

\[ y \% x \]

where the percent sign is called "the pair operator" and joins the two expressions which form the pair. In the display pair, each pair of corresponding elements \((y_i, x_i)\) describes a displayable point. All scaling and provision of axes is done automatically by the system so that a sine curve will be drawn showing all points and a pair of axes through \((0,0)\) if one executes

\[
X = \text{ARRAY} \left(100, 0.02 \times P1\right)
\]

\[
\text{DISPLAY} \ \text{SIN}(X) \% X
\]

**Fig. 1.** Graphical output resulting from the above two lines of SIGMA code.

More than one display pair may follow the DISPLAY operator. These pairs, separated by commas, constitute the display list. Abscissas as well as ordinates may be identical, overlapping or disjoint. A common scale is automatically chosen to include all points of all pairs entered in a DISPLAY statement. For example, using the above \(X\), two curves with disjoint abscissa: a sine followed by a cosine is obtained by

\[
\text{DISPLAY} \ \text{SIN}(X) \% X, \ \text{COS}(X) \% (X*2\times P1)
\]

**Fig. 2.** Result of the above display command.

while the following curves have overlapping or disjoint ordinates but the same abscissa

\[
X = \text{SIN}(X)
\]

\[
\text{DISPLAY} \ X, \ X**2\times X, \ \text{COS}(X)\times 2\times X
\]

**Fig. 3.** Result of the last display command.
will display a set of five successively larger sine curves and their complements.

![Graph of sine functions](image)

**Fig. 5.** The function $F(x,y) = y \sin x$ with $y$ as parameter.

8. SUBPROGRAMS

A MACRO is a named sequence of statements. The statements are assigned to a name and stored by the system during macro definition and they are executed whenever the macro name is called. The macro body is executed in the context of the macro call. All variable names used in the macro are taken to be variables defined at macro call time.

SUBROUTINES or FUNCTIONS communicate with the calling level through a conventional actual-formal argument mechanism which allows recursive definition of programs. For example, the Ackermann function may be defined by the following function subprogram:

```plaintext
FUNCTION ACK(M,N)
  IF(M EQ 0) ACK = N+1
  IF(M N EQ 0 AND N EQ 0) ACK = ACK(M-1,1)
  IF(M N EQ 0 AND N N EQ 0) ACK = ACK(M-1,ACK(M,N-1))
END
```

A variable defined inside a subprogram is a local variable of the subprogram. A local variable is automatically deleted upon exit from the subprogram in which it was defined. On entry to a subprogram all variables defined on the calling level become inaccessible except via the actual-formal argument correspondence.

A variable defined outside any subprogram is said to be defined "on the console level". Console level variables may be made accessible from any program by using the GLOBAL operator. This bypasses the formal-actual argument mechanism and permits shorter calling sequences of programs. For example, the statement

```plaintext
GLOBAL X,Y,Z
```

when executed in a program connects the so-far undefined or local variables named X, Y, and Z with the console level variables named X, Y, and Z. For example, one can define the following function

```plaintext
FUNCTION GAUSS(x)
  GLOBAL SIGMA, KSI;
  GAUSS = EXP(-((x-KSI)**2*0.5/SIGMA**2))
  SIGMA = GAUSS/SQRT(2*PI*SIGMA**2)
END
```

Now we must have SIGMA and KSI defined on the console level, but the function call will involve only one argument. Hence to evaluate for $-5 \leq z \leq 5$, the function

```plaintext
W(z) = \sin z * 1/18 * exp(-z^2/18)
```

one has to execute the following statements

```plaintext
SIGMA = 3; KSI = 0
Z = ARRAY(100, -5 #5)
W = SIN(Z)*GAUSS(Z)
```

9. SYSTEM DIRECTIVES AND EDITING FUNCTIONS

SIGMA uses a workspace concept to save and restore the state of a user's system. At any time in a console session the user may SAVE the status of the system and all its console level variables and user programs. Upon a subsequent LOAD the system is restored to the previously saved state. A user may also COPY information from another user's workspace.

A user may GET any file attaching it to SIGMA so that he can read or write on it and a user may PUT a file back into the operating system to release it from SIGMA use.

A user may EDIT a user program using the editor provided by SIGMA but if a user is partial to an editor of his own choice, he can write his user program to a file, use an editor outside of SIGMA and read the edited program back into the SIGMA system later.

10. SPECIAL ARRAY FUNCTIONS

Dynamic array handling requires a set of special array handling operators. Such operators are usually generalizations from tensor calculus such as, for example, the generalized transpose operator TP which permits any permutation of the component dimensions and the generalized trace operator TRMS which permits a summation of any dimension or any diagonal of an array. Other available operators include:

- **NNO**: Extract "number of components" vector
- **MAX**: Maximum
- **MIN**: Minimum
- **DEL**: Analogous to Dirac-delta function
- **DIFF**: Forward difference
- **SUM**: Running sum
- **PROD**: Running product
- **QUAD**: Running integral
- **LS**: Circular left shift in rows
- **ORDER**: Array re-ordering
- **HIST**: Use array to fill histogram bins
- **DROP**: Drop part of array
MULT Matrix multiplication
DET Determinant
EIGVAL Eigenvalues
EIGVEC Eigenvectors
INV Matrix inversion

The complete description of these operators is given in [9].

In addition, a subset of mathematical functions, provided by the CERN PROGRAM LIBRARY [10], is also available under SIGMA; the functions automatically act component-by-component when applied to arrays.

11. CONCLUSIONS

A system for interactive on-line numerical analysis problem-solving has been developed and its main features have been described. Some simple examples of the utilization of SIGMA have been given.

SIGMA is fully operational at CERN [9] on CDC 6600 Series computers and is used by a community of more than 50 users. It is also operational at several other computer centres, including: the University of Georgia in Athens (U.S.A.), the Free University of Brussels (Belgium); the University of Warsaw (Poland); the University of Saskatchewan (Canada); the University of Vienna (Austria); the Universities of Rome, Milan and Bologna (Italy), the Universities of Wollongong and Melbourne (Australia).

ACKNOWLEDGMENT

The above paper summarizes the work done on SIGMA development at CERN and at the University of Georgia from 1971 to 1975. Significant contributions to the development were made by all colleagues who participated in the SIGMA project in that time. Special mention should be made here to G. Barta (CERN), to J. Moszczynski (University of Warsaw) and to Dr. H. Jedlicka (University of Georgia). One of the authors (J.R.) is grateful to the Data Handling Division of CERN for their hospitality during his stay as Visiting Scientist.

In addition the authors wish to express their gratitude to Dr. M.G.N. Hine and Dr. G.R. MacLeod for giving the necessary support at CERN, as well as to Dr. R. Hagedorn and Prof. L. Van Hove, theoretical physicists, originators of the project.

The present status of SIGMA could not have been reached without constant feedback from users whom we thank here for their patience and their willingness to participate in discussions.

REFERENCES