CERN 78-08
Data Handling Division and
Theoretical Physics Division
4 September 1978

ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

SIGMA WITHOUT EFFORT

(An interactive tutorial for on-line SIGMA learning)

R. Hagedorn and J. Reinfelds

GENEVA

1978

CERN - Service d'information scientifique - RD/349 - 2000 - septembre 1978

CERN 78-08
Data Handling Division and
Theoretical Physics Division
4 September 1978

ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

SIGMA WITHOUT EFFORT

(An interactive tutorial for on-line SIGMA learning)

R. Hagedorn and J. Reinfelds*)

GENEVA

1978

*) The University of Wōllongong, Wōllongong N.S.W. 2500, Australia.

ABSTRACT

SIGMA (System for Interactive Graphical Analysis) is an interactive computing language with automatic array handling and graphical facilities. It is designed as a tool for mathematical problem solving. The SIGMA language is simple, almost obvious, yet flexible and powerful.

This tutorial introduces the beginner to SIGMA. It is supposed to be used at a graphics terminal having access to SIGMA. The user will learn the language in dialogue with the system in sixteen sessions of about one hour. The first session enables him already to compute and display functions of one or two variables.

Contents Page

INTRODUCTION 1

SESSION 1: SOME BASIC NOTIONS 4

1.1 Arrays and displays 4

1.2 Function definition 6

1.3 Functions of two independent variables 7

1.4 String arrays 9

SESSION 2: FUNCTIONS, SUBROUTINES AND MACROS 12

SESSION 3: STRUCTURAL ORGANIZATION 17

3.1 Levels of operation 17

3.2 Protection of names 17

3.3 Scope of names 19

SESSION 4: MORE ON STRUCTURAL ORGANIZATION 24

4.1 Control structures 24

4.2 Conditional statements, logical operators 24

4.3 Loops 28

4.4 Labels and jumps 29

INTERLUDE 32

SESSION 5: MULTIDIMENSIONAL ARRAYS 35

5.1 Concatenation 35

5.2 The NCO vector 35

5.3 Constructing multidimensional arrays 36

5.4 The transpose operator 39

5.5 Trace 39

5.6 Indexing 41

SESSION 6: PROGRAM EDITING 45

SESSION 7: SOME FUN WITH SPECIAL ARRAY OPERATORS 50

7.1 General operators 50

7.2 Random numbers and histograms, ordering 50

7.3 Integration 52

Contents Page

SESSION 8: MORE FUN WITH SPECIAL ARRAY OPERATORS: INTERPOLATION AND FUNCTION INVERSION 53

8.1 Interpolation 53

8.2 Function inversion 53

8.3 Finding zeros of functions 54

SESSION 9: MATRIX OPERATIONS 56

SESSION 10: DIALOGUE WITH SIGMA 59

10.1 Modes of operation 59

10.2 Copying information from another SIGMA user 63

10.3 The SIGMA Library Workspace 63

SESSION 11: MORE ON DIALOGUE WITH SIGMA; DISPLAY MODES 64

SESSION 12: TOPOLOGICAL ARITHMETIC 68

SESSION 13: COMPLEX NUMBERS AND FUNCTIONS 72

SESSION 14: MORE COMPLEX MAPPING 76

SESSION 15: INPUT AND OUTPUT OF DATA; COMMUNICATION WITH THE MAIN COMPUTING CENTRE VIA FILES. 79

15.1 Input from files 79

15.1.1 Communication with a running SIGMA program; (the terminal as file) 83

15.2 Output of data to a file 84

15.3 Output of a program to a file 86

15.4 Input of programs from files 87

15.5 General comments 88

SESSION 16: CONCLUDING SESSION 90

APPENDIX 1: AVAILABLE FUNCTIONS 92

Contents Page

APPENDIX 2: THE SIGMA LIBRARY WORKSPACE 96

APPENDIX 3: SOME USEFUL SIGMA PROGRAMS 98

APPENDIX 4: ORGANIZATION OF USER´S PRIVATE DATA AND PROGRAM LIBRARY 110

INTRODUCTION

This booklet is an on-line SIGMA teacher. SIGMA (= System for Interactive Graphical Mathematical Analysis) is designed to be a powerful, yet easy to learn, language for direct conversation with a computer.

Access to SIGMA, where it exists, is different from place to place and therefore not discussed here. This tutorial supposes that you can find somebody who is familiar with SIGMA and who teaches you how to use a terminal, how to get into SIGMA, and how to get out at the end of each session. Here are a few of the main features of this booklet:

- No experience in programming required.
- Sessions last from about 1 to about 2 hours.
- After the first session (60 min) you can already compute and display a function of one or two variables.
- Sessions proceed from simple to advanced; before !SAVE and !LOAD is introduced, each session is self-contained; after that, your data and programs are accumulated through all further sessions to be available at any later session.
- At any time during this course you can -- and should -- try to solve your own problems.

The present tutorial cannot replace the CERN SIGMA USER´S MANUAL as a source of further information.

If you work with a SIGMA version not identical with the CERN version, you may experience unpredictable difficulties in going through the sessions. This should, however, not sensibly diminish the usefulness of this tutorial because any version of SIGMA supplies many error messages to guide the user back to the correct statement after a mistake; furthermore, a mistake will, in general, not have any disastrous consequences, since commands are executed one by one as you type them in and most mistakes can be corrected by the next statement. Therefore this tutorial will help you to learn SIGMA even if the version available to you is not the one which we assumed in writing this text.

NOTE

In the following teaching sessions you are supposed to type in all those statements (=commands) which are marked by a vertical line on the left.

After each statement you must push the RETURN key in order to indicate to SIGMA that the statement is complete and should be executed. This will not be mentioned in the text any more.

The answers of SIGMA to your statements are not given here; you will see what happens and you should try to understand SIGMA´s responses before proceeding.

Often you will receive "error messages" telling you that you made a mistake. In most cases the error message gives you some information about the nature of the mistake; normally the erroneous statement is not executed and you can simply type its corrected version as the next statement.

Computer failures, if and when they occur, result in unintelligible messages and/or no answer at all (be patient; under heavy load the computer may give you an answer only after a time which may reach the order of a minute). If the word

COMMAND

appears, you have been thrown out of SIGMA and you must re-enter it by typing the word SIGMA. If you are in serious trouble consult an experienced SIGMA user or abandon the session and start again some time later. In case of endless printing or no answer, you may "kill SIGMA" by pushing the keys ESC A RETURN in this order and re-enter SIGMA by typing SIGMA.

SIGMA often gives the value Ø (zero) to mathematically ill-defined or infinite quantities. Practical experience has shown that this is convenient and without danger.

Keyboards of graphical terminals may differ from place to place. You will become familiar with yours by trial and error in the course of these sessions.

Frequently !ERASE is used to start a "new page" on the terminal; pressing L whilst holding down the CTRL key will do the same thing.

To recall in any situation the last few statements, you push the "add" key:

@

(followed by RETURN as usual)

This has no other effect then to erase the screen and print the last 20 statements you entered. You then proceed. The key

BACK
SPACE

may be used to correct typing errors while entering a statement; it is useless after RETURN. Pushing the CTRL key followed by X is equivalent to back-spacing to the beginning of the line.

LF

is line feed and may be used after BACK SPACE in order to continue on the next line.

SESSION 1: SOME BASIC NOTIONS
(60 minutes)

1.1 Arrays and displays

Any ordered set of numbers is an array. If the numbers are equally spaced along the real number line, they are said to span a range. A range is described by the "number sign" "#" so that

1#20

denotes a range of numbers (as yet not specified how many) from 1 to 20 inclusive. To generate an array over a range, we have to specify how many numbers are needed as well as what range is to be spanned. The statement (statement = command) (type in)

```
        PRINT ARRAY (20,1#20)
```

will generate the first 20 integers and print them on your terminal. Since PI = 3.1415926535898 is available with the name PI, a variable X consisting of 100 numbers over the range 0 ≤ X ≤ 6*PI is generated by the assignment statement- (type in)

```
        X=ARRAY (100, 0#6*PI)
```

Note that nothing is printed. X is generated and saved for further use: the name X and the array of 100 numbers are assigned to each other. An explicit print command is required to print any array (type in)

```
        !ERASE
        PRINT X
```

Automatic print-out after each assignment statement is switched on by the command !PRINT and switched off by !NOPRINT. Try this to see what happens.

```
        !PRINT
        X = ARRAY (100, 0#6*PI)
```

Most standard functions are available in SIGMA. Their mnemonic names are quite conventional and similar to FORTRAN. A few examples are (do not type)

EXP(X)	exponential function
LOG(X)	natural logarithm
LOG10(X)	obvious
SIN(X), COS(X), TAN(X)	obvious
TANH(X)	obvious
SQRT(X)	square root.

A full list of available functions is found in Appendix 1. The array $Y = \{y_i;\ y_i = x_i * \sin(x_i);\ 1 \leq i \leq 100\}$ will be generated by (type in)

```
Y = X * SIN(X)
```

Lists of numbers are not well suited to study functions. Hence type

```
!NOPRINT
```

Then try the following command:

```
DISPLAY Y : X
```

This command displays all points (x_i, y_i) $1 \leq i \leq 100$ and connects the points with straight lines. All scaling and generation of axes is done automatically. SIGMA remembers the scaling factors and the independent variable (here: X) used by the last DISPLAY command, so that the same curve will briefly flash if you enter simply the <u>"incomplete display command"</u>:

```
Y                          (or Y : X)
```

and a new curve is added for each of the entries

```
 COS(X)                    (or COS(X):X)
- COS(X)                   (or - COS(X):X)
```

The "colon" : is called the "pair-symbol", because each point is defined by a pair of coordinates. A pair of arrays defines a sequence of points which make up a curve. The points themselves may be made visible by including in the display commands any symbol in brackets as for instance in

```
        DISPLAY [*]COS(X):X             or else by
        DISPLAY COS(X):X, [O]COS(X)
```

Note that the use of the word DISPLAY erases the old picture and rescales the new picture. If we now add y = x sin (x) to this picture, most of the new curve will lie outside our picture area. To see this enter

```
        Y                          (or Y:X)
```

A common scale is established for all curves specified in one display statement. Try the following command:

```
        DISPLAY X*SIN(X)*SIN(X):X, COS(X), [S]SIN(X)
```

To summarize: The word DISPLAY starts a new picture and finds a common scale so that all parts of all curves are displayed. A pair without the word DISPLAY will add a curve to the existing picture according to scale factors established by the last use of DISPLAY. It may easily happen that the whole of the newly added curve lies outside the visible picture area. A character in square brackets before any pair will draw each point explicitly using the given character without any connecting lines.

1.2 Function definition

To study a function, such as, for example, $f(x) = e^{-x} \sin ax + bx^2$ with $0 \le X \le 5$ and various values of the parameters a and b, it is tedious to enter for a = 8, b = 0.01

```
        !ERASE
        X = ARRAY (101, 0#5)
        DISPLAY EXP(-X)*SIN(8*X)+.01*X**2:X
```

and to repeat the lengthy display statement for each new curve. Instead, define a function

```
FUNCTION F(A, B, X)
F = EXP(-X) * SIN(A*X) + B * X ** 2
END
```

and use it simply as one would expect to use a function f(x):

```
Z = SQRT(X)*F(3,.005,X)
DISPLAY F(8,.01,X):X,[.-]Z
```

Note the effect of [.-]; try also [.-2], [.-3] in a DISPLAY. Any statement which is used directly may also be used in the definition of a function. The result of the function is the value which is assigned to the function name; therefore the function name must appear inside the function body at least once on the left-hand side of an assignment statement. Type

```
FUNCTION G(X)
Z = SIN(X)*SQRT(X)
END

PRINT G(5)
```

Type the correct version of this function and try again. A user-defined function is very similar to any other function defined in SIGMA, such as SIN or COS for example.

To summarize: A function definition is started by the command FUNCTION followed by a user-given function name and argument list. A function definition is terminated by the END statement. The sequence of commands between FUNCTION and END is called the function body.

1.3 Functions of two independent variables

The arrays defined up to now may be regarded as one-dimensional row vectors. They span a single dimension and arithmetic operations combine them component by component. For example, type

```
X = ARRAY(10, 1 # 10)
Y = ARRAY(10, 10 # 100)
```

then

```
Z = X + Y
!ERASE
PRINT X,Y,Z
```

A column vector may be obtained as the transpose of the row vector

```
!ERASE
X = ARRAY(5,.1#.5)
Y = TP(X)                    [TP is the "transpose" operator]
Z = 10*Y+X
PRINT X,Y,Z
```

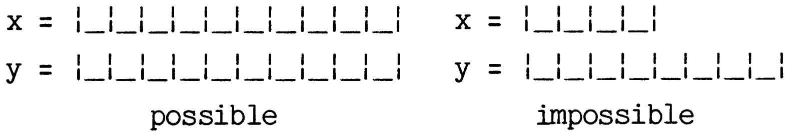
What is the difference between the first and the second example? In the first we have

$$Z_i = X_i + Y_i \quad \text{for} \quad i = 1,..,10$$

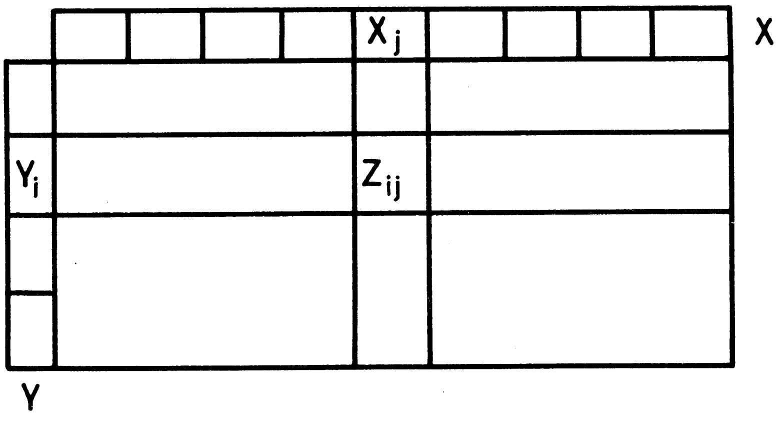
and in the second

$$Z_{ij} = 10Y_i + X_j \; .$$

Why? the general rule for arithmetical combination adopted for SIGMA, is "component by component" combination: think of having put the two arrays below each other and then carry out the required operation between corresponding components. Obviously this is possible if and only if both arrays have the same structure:



There is one exception, namely our second example, where y is a column vector and x a row vector; if we put them next to each other we get



and again "component by component" combination makes sense; x and y need not even have the same number of components. Of course, Z is now a two-dimensional array. We may thus say that z = f(y,x) is a function of two independent variables over the rectangle spanned by the topological (direct) product of the range of the row vector X with the range of the column vector Y. We postpone to a later session the generalization to more than two variables. An example is:

```
FUNCTION GAUSS(SIGMA,X)
GAUSS = EXP(-X*X/2/SIGMA**2)
GAUSS = GAUSS/SQRT(2*PI*SIGMA**2)
END

X = ARRAY(51,-1#1)
SIGMA = TP(ARRAY(10,0.1#1))
Z = GAUSS(SIGMA,X)
DISPLAY Z:X
```

Interpret the figure. Then

```
DISPLAY TP(Z):TP(SIGMA)
```

and interpret this also. Why TP?

1.4 <u>String arrays</u>

<u>String arrays</u> contain text instead of numerical values. To inform SIGMA that in the statement

```
A = GAUSS(1,5)
```

you do not wish A to be the numerical value of the function GAUSS with SIGMA = 1, X = 5, but instead the sentence (= string of characters) ´GAUSS(1,5)´, you put this string between quotation marks:

```
ASTRING = ´GAUSS(1,5)´
```

Now look at the result:

```
PRINT A, ASTRING
```

String statements are useful in printing out tables or in commenting displays, for instance by including a message in a program:

```
FUNCTION SADDLE(A,B)
X = ARRAY(21,-1#1)
Y = TP(X)
SADDLE = A*X*X-B*Y*Y
DISPLAY SADDLE:X
PRINT´PICTURE OF SADDLE´,´SADDLE=A*X**2-B*Y**2´
PRINT´A IS´,A,´B IS´,B
END
```

```
Z = SADDLE(2,3)
```

Further applications will come later. In the computer the string of characters is represented by numbers; it is possible to carry out arithmetic operations on strings and thereby create your own secret code or check the identity of two strings. If in a statement numerical and string variables appear mixed, the result may appear in numerical or string form:

```
!ERASE
A = ´THIS IS A STRING ARRAY´
PRINT A,A+1,1+A
```

Any numerical array can be translated into a string array and vice versa:

```
X = NUMBER(A)
PRINT X,A
B = STRING(X)                  (Compare with the previous print!)
PRINT X,B
!ERASE
```

Now as an example of coding:

```
!PRINT
SACODE = (A*3+2)**2        (in string form)
NACODE = (2+A*3)**2        (the same in numerical form)
```

Now we recover the original text from the coded one:

```
RAS = (SQRT(SACODE)-2)/3
RAN = (SQRT(NACODE)-2)/3
```

Both are numerical now, becaue SQRT does not give a string result. This does not worry us, because

```
RAS = STRING(RAS)
RAN = STRING(RAN)
```

result both in recovering the original text. We could as well have typed

```
RA = STRING((SQRT(SACODE)-2)/3)
!NOPRINT
!ERASE
```

We end the session with an example of comparing texts (proof reading):

```
B = ´THIS IS E STWING ARREY
PRINT A,B
PRINT A-A, A-B
```

as A is identical with itself, A-A is a string of blanks, while A-B shows blanks where both are equal and some other symbol where they differ.

```
C = ´12345´
PRINT C,C+1
```

So much about strings; we do not need them here, except for messages in programs. Further information can be found in the CERN SIGMA USER´S MANUAL.

To end the session, type

```
!STOP
```

and when you are asked whether you really mean it, type

```
YES
```

END OF SESSION 1

SESSION 2: FUNCTIONS, SUBROUTINES AND MACROS
(50 minutes)

Suppose that we try to shorten the argument list of the function F defined in Session 1 by defining it differently:

```
FUNCTION G(X)
G = EXP(-X)*SIN(A*X)+B*X*X
END
```

Now we try

```
X = ARRAY(101,0#5)
Z = G(X)
```

The function cannot be executed. So we type

```
A = 8
B = 0.01
Z = G(X)
```

The function can still not be executed because it contains two variables A and B which are by convention local to the function body and independent of any other variables of the same name which may occur outside the function body. (The function body is defined to be the sequence of statements between the command FUNCTION and the command END.). This convention avoids a clash between the intermediate variables used in programs and other variables with the same names used elsewhere.

The argument list is one way to connect variables used inside a function with variables used outside. Another way to connect variables is by using the GLOBAL command.

```
!ERASE

FUNCTION H(X)
GLOBAL A,B
H = EXP(-X)*SIN(A*X)+B*X**2
END
```

will perform correctly if we say

```
Z = ARRAY(101,0#5)
```

```
DISPLAY H(Z):Z
```

because the GLOBAL statement links the variables A and B in the function body to the A and B defined previously outside the function. Note that here the variable is called Z and not X as in the function definition: Z is the "actual argument" of the "function call" (where it may have any name including X) while X is the "formal argument" in the "function definition. For a more thorough discussion see Session 3.

Functions defined by a FUNCTION ... END "bracket" pair are stored for further use in the same session (permanent storage is explained in Session 4). They may be printed:

```
!ERASE
PRINT G,H
PRINT H(5),H(PI)
```

PRINT followed by a function name without argument(s) prints the text of the function; with argument(s) it prints the corresponding value(s).

The function G is no longer needed, so type

```
DELETE G
```

All names currently defined by the user are displayed by the command:

```
!NAMES
```

You see that G really got deleted and that H remains defined. Also note that several variables are still defined and stored. Some contain scalars such as A and B, some contain arrays such as X and Z. Variables which are no longer needed may also be deleted by the DELETE operator. All names defined and stored by the user should be periodically reviewed using !NAMES and all obsolete variables and programs deleted using DELETE. The interactive command !DELETE makes deletion easier. Try this command and see what happens (type A, N, or S as answer).

A FUNCTION is a special case of a program. A program is simply a named sequence of commands. The whole sequence may be executed by inserting the function name in an assignment statement. This saves a lot of typing in repetitive situations.

A function is a program which returns a value. A function is called by writing the function name plus argument list in any arithmetical expression. All variables defined in a function are local to the function unless declared global in the function or appearing in the argument list.

A subroutine is a program which performs a sequence of commands but does not necessarily return a result. A subroutine is simply a function without assignment of a value to the subroutine name. Hence a subroutine call may not appear in an arithmetical expression, but is executed by the command CALL. For example, suppose in the previous example one wishes to study the function H(X) by always displaying H(X) against X. Then one can define the subroutine

```
        SUBROUTINE SH1(X)
        GLOBAL A,B
        H = EXP(-X)*SIN(A*X)+B*X**2
        DISPLAY H:X
        PRINT ´THIS IS H:X´, ´PARAMETERS A,B´, A,B
        END
```

and execute it by the call command

```
        CALL SH1(Z)
```

Alternately one could use our previously-defined function H(X) and define the subroutine

```
        !ERASE

        SUBROUTINE SH2(X)
        DISPLAY H(X):X
        END
```

and execute this by the call

```
        CALL SH2(Z)
```

This is correctly executed, since A and B are still defined with the values 8 and 0.01, respectively, and the GLOBAL definition in H(X) communicates directly with these user-defined values.

Often it is convenient to disregard the distinction between local and global variables and arguments. For such situations a third kind of program, called macro, can be defined.

A *macro* is simply a named list of commands (*macro body*), which is executed by the command CALL followed by the name of the macro; it then behaves exactly as if the macro body had been inserted in place of the call statement; all variables it uses must be defined at call time and the variables it generates remain defined after execution. No variables are made local or global or inaccessible; a macro has no arguments. For example, our study of function H could also be performed by defining a macro

```
!ERASE

MACRO MH
H = EXP(-X)*SIN(A*X)+B*X**2
DISPLAY H:X
PRINT ´MACRO MH DID IT´, A,B
END
```

and execute this by

```
CALL MH
```

It could not work, because now X is no longer a "formal argument", the macro requires X to be defined as well as A and B. Hence

```
X = Z
CALL MH
```

executes correctly, because A, B, and X have values when the macro is called. In addition, the macro execution will leave a new variable H defined. Try PRINT H or !NAMES to check this. This is because variables defined by a macro are not local to the macro execution. Hence the main *advantage* of a macro is that all variables which are valid at call time may be used in the macro. The main *disadvantage* (sometimes an advantage) is that all variables generated in the macro body remain defined after the completion of the macro execution and overwrite any others of the same name.

<u>Remark</u>: In the following text we shall not always distinguish between FUNCTION, SUBROUTINE and MACRO; wherever no confusion is possible we simply use the common name "program".

```
!STOP
YES
```

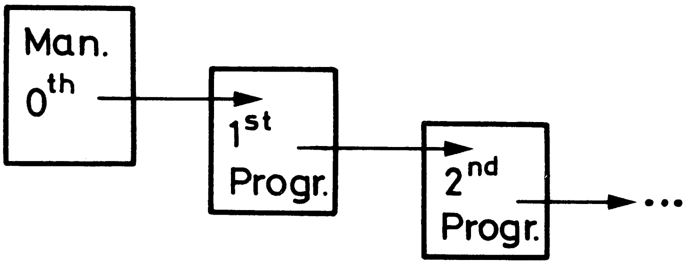
END OF SESSION 2

SESSION 3: STRUCTURAL ORGANIZATION

Remark: Read this session carefully before trying it on the terminal.

3.1 Levels of operation

We distinguish, at a given moment, different levels of operation: the manual level, first program level (program called from manual level), second program level (program called from first program level), etc. Relative to the n^{th} program level, the $(n-1)^{th}$ program level (0^{th} is manual) is the "calling level". Schematically



Type

```
MACRO FIRST
PRINT 'THIS IS THE FIRST LEVEL CALLED FROM MANUAL'
CALL SECOND
END

MACRO SECOND
PRINT 'SECOND LEVEL CALLED FROM FIRST'
END
```

```
CALL FIRST
```

3.2 Protection of names

As we have seen so far, any assignment statement or program definition can redefine any previously-defined name. The previously-defined object (variable or program) is overwritten and lost.

This is often convenient: new objects may be assigned to existing names without explicit deletion of obsolete objects which are already attached to these names.

This is also dangerous, however: one soon becomes careless and forgets what programs and variables one has defined earlier and before long one will destroy valuable objects by overwriting or redefining them unintentionally. To avoid this, use PROTECT. See what happens if you type

```
!ERASE
X = 1
A = 2
C = ARRAY(5)
PRINT C                   (an ARRAY command with missing second
                          argument produces an array filled
                          with 1's)
PROTECT X,A,C
X = 4
SUBROUTINE A(X)
DELETE C
PRINT X,A,C
```

Protected objects can be used, however:

```
!ERASE
XA = X*A
XX = X*X
PRINT XX,XA
```

but they are protected against any damage. The command

```
DEPROTECT A,X,C
```

gives back to A, X and C their normal unprotected status.

A way to get a completely fresh system where all names, including protected ones, have disappeared is

```
!CLEAR
```

check it:

```
!NAMES
```

Think twice before you use !CLEAR; it is very dangerous, because everything is lost.

3.3 Scope of names

Names and the objects which they represent must be considered as functions of time, the latter may be taken to increase by one unit for each executed statement. Particular times with respect to a program are $t = t_C$, the "call time" when control goes to the program and $t = t_R$, the "return time" when control goes back to the calling level.

All program names are universal, i.e. any program can be called from any level. We therefore need not discuss program names further.

All names needed for the execution of a statement or of a program must be defined at call time, otherwise the system will respond with an error message. Type

```
!ERASE

SUBROUTINE SSS
GLOBAL A
PRINT A
END

A = 5
CALL SSS
DELETE A
CALL SSS
Z = A*A
```

From now on we assume that all needed names are defined. Consider the following sets of names:

$\{MAN\}_t$ = Set of names defined at time t on the manual level (listed by !NAMES);

$\{CLL\}_t$ = Set of names defined at time t on the calling level;

$\{PRG\}_t$ = Set of names defined at time t in a program during execution; more specifically we write $\{FCT\}_t$, $\{SBR\}_t$, $\{MCR\}_t$ for function, subroutine, macro, respectively.

$\{ARG\}_t$ = Set of names appearing in the calling statement as actual arguments (of function or subroutine) and being defined at t;

$\{GLB\}_t$ = Set of global names defined at t;

$\{LOC\}_t$ = Set of local names defined at t;

$\{\ \}$ = Empty set.

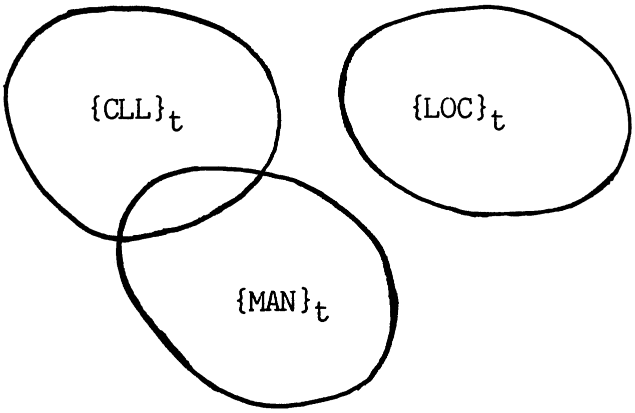
We now state the scope of names by a few set relations:

For subroutines and/or functions we have

1. $\{ARG\}_t \cup \{GLB\}_t \cup \{LOC\}_t = \{PRG\}_t$

2. $\{LOC\}_t = \{\ \}$ if $t < t_C$ or $t > t_R$

3. Local names do not interfere with names outside the program:

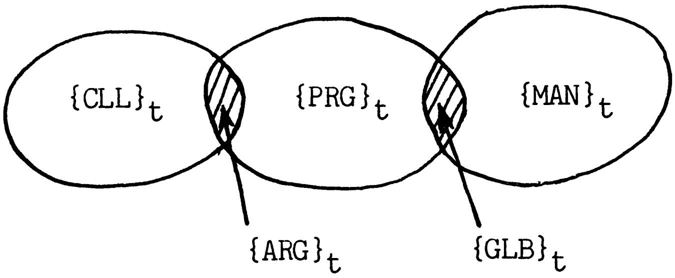


namely

$\{LOC\}_t \cap (\{MAN\}_t \cup \{CLL\}_t) = \{\ \}$

4. $\{GLB\}_t$ and $\{ARG\}_t$ communicate with different sets:

$\{GLB\}_t$ with $\{MAN\}_t$ and $\{ARG\}_t$ with $\{CLL\}_t$



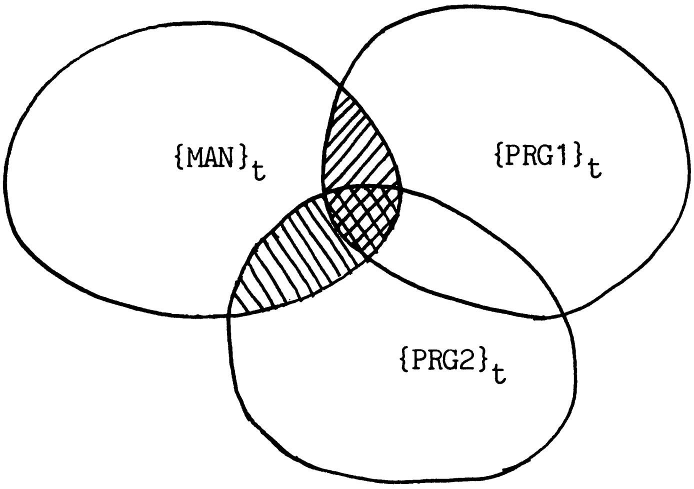
$\{ARG\}_t = \{CLL\}_t \cap \{PRG\}_t$

$\{GLB\}_t = \{MAN\}_t \cap \{PRG\}_t$.

5. Two subroutines and/or functions PRG1 and PGR2 which do not call one another can nevertheless communicate

- via their common global variables over the manual level,

- via their common actual arguments over the calling level.



$\{\text{▨}\}_t = \{GLB1\}_t = \{PRG1\}_t \cap \{MAN\}_t$

$\{\text{▧}\}_t = \{GLB2\}_t = \{PRG2\}_t \cap \{MAN\}_t$

$\{\text{▦}\}_t = \{GLB1\}_t \cap \{GLB2\}_t = \{PRG1\}_t \cap \{PRG2\}_t \cap \{MAN\}_t$

This rule remains true if {GLB} is replaced by {ARG} and {MAN} by {CLL}.

6. Warning: Any argument of a function or subroutine which is changed during execution, is changed for good: it takes its new value also on the calling level. (i.e. unlike the habit in mathematics, the argument can also be an output variable).

Try

```
!ERASE

FUNCTION ABC(X)
X = X**2+2*X+1                X is the formal argument
ABC = X
END

Z = 10
PRINT ABC(Z),Z                Z is the actual argument
```

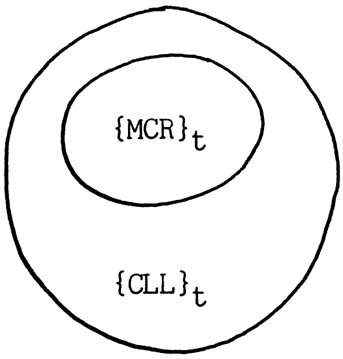
Z has changed its value. This often leads to undesired and unpredictable side effects; therefore avoid systematically any assignment statement with the argument of a function on the left-hand side; rather introduce a local dummy variable: try

```
!ERASE

FUNCTION ABC(X)
Y = X**2+2*X+1
ABC = Y
END

Z = 10
PRINT ABC(Z),Z
```

this was only an example of what can happen if one is careless about local variables. A good programmer would have used neither of these two examples but simply written

```
FUNCTION ABC(X)
ABC = X**2+2*X+1
END
```

7. Macros behave differently from subroutines and functions: all their names communicate with, and only with the names of the calling level:



$\{MCR\}_t \subset \{CLL\}_t$

```
!STOP
YES
```

END OF SESSION 3

SESSION 4: MORE ON STRUCTURAL ORGANIZATION
(60 minutes)

4.1 Control structures

Control structures are mainly used inside programs, although they can be used on the manual level. They allow three things:

- to make execution of a statement dependent on some condition checked inside the program;
- to carry out "loops" of repetitive operations;
- to jump over any number of statements and go to some specifically labelled one, from where execution proceeds as usual.

4.2 Conditional statements, logical operators

The conditional statement is of the form

IF <expression>   <statement>

the <expression> is evaluated and rounded to the nearest [positive or negative integer(s)]. If then all values of the result (which might be an array!) are equal to 1 (1 stands for "true"), <statement> is executed; if not, instead of the conditioned statement, the next following statement will be executed.

Type

```
        X = -1
        IF(X LT 0) Z = ABS(X)
        PRINT X,Z
        IF(X GT 0) Y = 0
        PRINT Y
```

Why the error message?

```
        X = 10
        IF(X GT 5 OR X LT -5) PRINT´ABS(X)EXCEEDS 5´
```

The following "order relation operators" are in use:

```
LT = less than
LE = less than or equal to
EQ = equal to
NE = not equal to
GE = greater than or equal to
GT = greater than.
```

To these add the "logical relation operators" AND, OR, NOT, IF, ANY. Logical operators require "truth values" ("true" and "false"); in SIGMA we identify

"true" with the value +1

"false" with the value 0

The mentioned logical operators have the following <u>truth tables</u>:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

$$\text{NOT} \begin{vmatrix} 0 \\ 1 \end{vmatrix} = \begin{vmatrix} 1 \\ 0 \end{vmatrix}$$

IF, when applied to an array, has the sense of "if all". In SIGMA all expressions, whether they do or do not contain logical operators and functions can be freely mixed in the same statement.

Whenever a truth-value interpretation is implied by the context, any number x: $-0.5 \leq x < 0.5$ is rounded to zero ("false") and any $0.5 \leq x < 1.5$ is rounded to one ("true") and then interpreted; any other number will be refused and cause an error message. The relational operators generate one or zero according to whether the relation is true or false:

```
!ERASE
X = ARRAY (4,-2#1)
PRINT X
PRINT X LT -1
PRINT X LE 0
PRINT X EQ 1
PRINT X NE -1
PRINT X GE 1
PRINT X GT 2
PRINT NOT(X GT 2)
```

One can mix logical or order relations with numbers:

```
!ERASE
X = ARRAY(21,-1#1)
Y = X**3+1
Y1 = Y*(X LT 0)+1
DISPLAY Y:X, [.-2]Y1
```

Note: To see what [.-2] means, try

```
DISPLAY [.-5]Y, [.-9]Y1
```

Another example is:

```
!ERASE

MACRO TELLME
IF X PRINT' X IS NEAR TO ONE'
IF NOT X PRINT' X IS NEAR TO ZERO'
END
```

Now try

```
X = 0
CALL TELLME
X = 1
CALL TELLME

X = ARRAY(4,-0.4999#0.4999)
CALL TELLME
X = ARRAY(4,0.5001#1.4999)
CALL TELLME
X = 3
CALL TELLME
```

Here X itself is interpreted as the truth value (rounded if necessary).

As the IF has the sense of "if all", a SIGMA function ANY is provided which gives the scalar value TRUE if any component of its argument is TRUE so that

```
IF (ANY (<expr>)) <statement>
```

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

```
!ERASE

MACRO ANYALL
IF (X LT 0) PRINT' ALL X NEGATIVE'
IF (X LT 0) PRINT X
IF  (X  LT  0)  RETURN          (which  means:  return  to the
                               calling level)
IF (X GE 0) PRINT 'NO NEGATIVE X'
IF ANY (X LT 3) PRINT 'SOME X LT THREE'
IF (X LE 100) PRINT 'NO X EXCEEDS 100'
IF ANY (Y EQ 0) PRINT 'AT LEAST ONE X = 0'
PRINT X
END
```

```
!ERASE
X = ARRAY (4,0#3)
CALL ANYALL
X = X-4
CALL ANYALL
X = X+3
CALL ANYALL
!ERASE
X = X+100
CALL ANYALL
```

One more example is:

```
!ERASE

SUBROUTINE SHOW (X)
Y = (X-1)*(X-2)*(X-3)
Z = -Y
IF ANY (X LE 0) Z = ABS(Y)
DISPLAY Y:X,[.-2]Z
END

R = ARRAY(101,0.5#3.5)
CALL SHOW(R)
S = ARRAY(101,3.001#4)
CALL SHOW(S)
```

It is important to realize that if in

IF <expression> <statement>

either the <expression> and/or the <statement> contain arrays, the IF statement does not function for each component individually. Evaluation proceeds from left to right and firstly the whole array <expr> is evaluated and then, if all components of the result equal 1, the <statement> is executed as usual.

If the result of <expr> contains at least one 0, control passes to the next statement following the IF statement.

Conditional component-by-component execution of statements can be achieved by combining logical and/or order relations with arithmetic or other operators as shown in one of the above examples.

4.3 Loops

Loops are of the general form (do not type)

DO n_1 $I = i_1, i_2, i_3$

____

____

________ "Do-loop-body" commands may use I but should not change it.

____

____

____

n_1 CONTINUE

In words this means: "do the statements of the do-loop-body once for each value of I, starting with $I = i_1$, then for $I = i_1 + i_3$, $I = i_1 + 2i_3$ ... until the next I would be greater than i_2; then continue". The do-loop-body is the set of statements embraced by the DO statement and the CONTINUE. n_1 must be an integer $1 \leq n_1 \leq 99999$, called <u>label</u>; inside a program the same label may not be used twice; note that the label is not the same as the number printed automatically by SIGMA which latter numbers the sequence of lines on which statements are written. i_1, i_2, i_3 must be either numbers or scalar names (not arrays) or expressions having a scalar result. i_3 and the comma preceding it may be omitted; in that case $i_3 = 1$ is assumed by SIGMA: i_1, i_2, i_3 need not be integer, but i_2 should be > i_1 and $i_3 > 0$ to really generate a loop. Type

```
!ERASE
DO 15 K = 1,5
PRINT K*K
15 CONTINUE

DO 3 J = 1.3, 1.75, 0.1
PRINT J
3 CONTINUE
!ERASE

MACRO SELECT
Z = 1
DO 1 I = 1,10
Z = RNDM(Z)                     (random number)
IF (Z GE 0.5) Z = -Z + 0.5
X(I) = Z
1 CONTINUE
END

X = ARRAY(10)
```

```
PRINT X
CALL SELECT
PRINT X
```

The loop body is performed at least once, because the test on I is performed after the execution of the loop body:

```
DO 5 J = 4,1,1
PRINT J
5 CONTINUE
```

4.4 Labels and jumps

Statement labels are one to five digit integers which may be attached to any statement by the user. Indiscriminate branching may be performed by the GOTO statement

GOTO <expression>

The expression is evaluated and rounded to the nearest integer (expression must, of course, result in a scalar). This integer is taken to denote a label and, if such a label exists, execution control jumps to the statement labelled by it.

Indiscriminate branching is useful in IF statements when the alternatives consist of several statements, e.g.

```
    ------

    ------

    ------
    IF <expr> GOTO 1

    ------

    ------

    ------

    ------
    GOTO 3

    ------
1

    ------

    ------

    ------

    ------

    ------
3

    ------

    ------
```

Example: type

```
!ERASE

MACRO ROOT
IF (X GE 0) GOTO 1
X1 = X*(X GE 0)
DISPLAY SQRT(X1):X
PRINT ´SOME X NEGATIVE´
GOTO 33
1 DISPLAY SQRT(X):X
PRINT ´NO NEGATIVE X´
33 PRINT ´b´, ´b´, ´b´    (insert 3 blanc lines*))
PRINT ´ROOT ACCEPTS NEGATIVE X´
PRINT ´BUT REPLACES THEM BY 0´, ´b´, ´b´
PRINT ´SMALLEST X IS´, SMIN(X)
END

X = ARRAY(101,-1#1)
CALL ROOT
X = X + 1
CALL ROOT
```

*) The letter b means: push the ¦SPACE¦ key ("blanc")

Note: Never direct a GOTO from outside a DO loop into it. Example:

```
!ERASE

MACRO DONEVER
PRINT X
DO 7 K = 1,10
X(K) = K*X(K)
13 PRINT X(K)
7 CONTINUE
IF ANY(X LT 0) GOTO 13
END

!ERASE
X = ARRAY(10,-4#5)
CALL DONEVER
!STOP
YES
```

END OF SESSION 4

INTERLUDE
(this is not a Session)

SAVING AND LOADING

By now you have acquired enough skill to solve simple problems and to write some programs. You will feel the need to preserve what you did, so that you can use it later in another session. For this you must have your own SIGMA workspace, which is uniquely identified by two words, e.g.

NAME, YOURID

the first, NAME, is the "filename", the second, YOURID is your "identifier"*). You may have several workspaces with different filenames and the same identifier, e.g. JACK, BOB, CARL, ..., all with the same YOURID. You chose these names and ask somebody who is familiar with the workspace creating procedure to create for you the workspace NAME. From now on, at the end of a session you type

```
!SAVE, NAME, YOURID
```

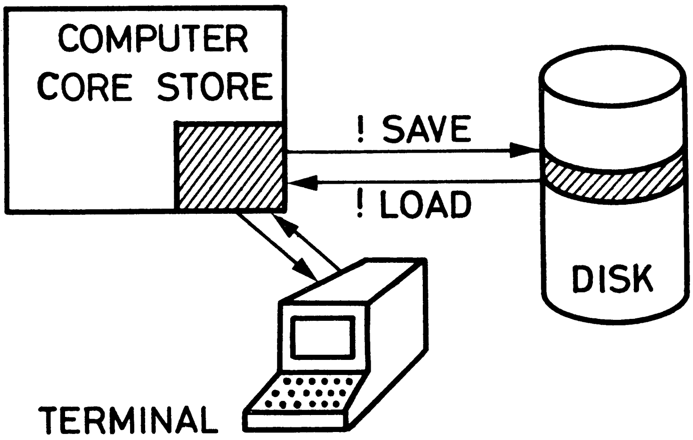
and the current status of your SIGMA Session (everthing visible by !NAMES) will be "saved", i.e. stored somewhere safely. Tomorrow or next week you can continue your work by

```
!LOAD, NAME, YOURID
```

at exactly the point where you saved it. The scheme is as follows:

---

*) "NAME" and "YOURID" stand here as symbols for the actual words you will choose and which will be different from "NAME" and "YOURID"



Every command that you type in will cause some action of the computer and, in general, it will imply some change in the data and/or programs stored in the core memory, where a small corner is reserved for you. Upon saving, a copy of this corner is transferred to a permanent store (disk) overwriting there the previous copy (if there was one). Upon loading, a copy of what was stored on disk (by the last !SAVE) is transferred to your corner of the memory, overwriting what was there (if anything).

You should update your saved copy on disk, not only at the end of a session, but also each time after creating some time-consuming valuable object (data and/or program) by !SAVE; if then afterwards the computer should break down, you can at least resume your work at the point of this last !SAVE.

You will now realize how dangerous the !CLEAR command is: if you use !CLEAR in a session, everything in your corner of core memory is erased; if you would !SAVE, all your previously saved work would be replaced by nothing; it is lost.

Get familiar with !LOAD and !SAVE; it is important. Note that !SAVE may take a long time (minutes!) if the workspace is very long and the computer busy.

From here on this tutorial supposes that you have a workspace of your own (called here: NAME, YOURID) and that you start each session by !LOAD and end it by !SAVE. Programs and numerical objects created in the following sessions will be assumed to exist in later sessions until redefined or deleted.

Do not continue before you have obtained your own workspace.

* END OF INTERLUDE *

SESSION 5: MULTIDIMENSIONAL ARRAYS
(90 minutes)

Remark: We do not begin here with !LOAD, since nothing was yet saved onto your workspace.

5.1 Concatenation

We use the sign & (ampersand) to concatenate two objects. Type

```
!PRINT                          (remain in !PRINT mode until 5.6)
X = 1&2
X = X&3&4
Y = 10*X
Z = 100*X
!ERASE
ALL = X&Y&Z
EACH = TP(X)&TP(Y)&TP(Z)
WRONG = EACH&TP(ALL)
OK = EACH&TP(10+Z)
KO = TP(10*Z)&EACH
```

Thus the rule is: arrays of any dimension can be concatenated if they have the same structure except for the last dimension, where they may differ; concatenation is done by joining rows in the sequence in which the objects to be concatenated appear in the statement.

5.2 The NCO vector

In all the printed objects you may have noticed a message like:

NCO(X) = 2 or NCO(EACH) = 4&3 etc.

NCO (= Number-of-COmponents-vector)

is a vector whose components indicate how many components are in each dimension of the array. Hence the NCO vector defines the structure of the array; NCO(EACH) = 4&3 says that there are four rows of three elements each, while NCO(OK) = NCO(KO) = 4&4 says that KO and OK are 4 by 4 structures. NCO is an operator:

```
!ERASE
NX = NCO(X)
NALL = NCO(ALL)
NOK = NCO(OK)
```

If NCO has n components, the array is n-dimensional:

```
DIMX = NCO(NCO(X))
DIMOK = NCO(NCO(OK))
```

the product of all components of NCO is the total number of elements of the array. The NCO operator plays a great role in constructing programs which must handle arguments of any, not predetermined, structure. It also is the most important tool in constructing arrays of a given structure.

5.3 Constructing multidimensional arrays

We have used statements like X = ARRAY(101,0#1). The first argument of the ARRAY operator is the NCO vector of the array; here NCO(X) = 101 is one single number, NCO(NCO(X)) = 1. Now try

```
!ERASE
X = ARRAY(24,1#24)
X = ARRAY(8&3,X)
```

In the second command the NCO vector is 8&3 and accordingly the ARRAY operator has constructed an 8 x 3 matrix using the elements of X.

Now try

```
!ERASE
X = ARRAY(6&4,0#3)
```

Why do we obtain six identical rows? It is a <u>convention</u>, adopted for the following reason:

To preserve the "orthogonality" of x and y in f(x,y). [See Session 1] the range specification (0#3 here) <u>by definition</u> extends over the "row-dimension" or the "right-most index" only. Hence if we specify an NCO of two components (6&4) we are asking for six identical rows of four numbers equally spaced over the range 0#3. Similarly

```
!ERASE
X = ARRAY(2&3&4,10#40)
```

generates two matrices each containing three identical rows of four numbers over (10#40).

One may be tempted to try to generate a column vector with the integers 1,...,5 as components by

```
        Y = ARRAY(5&1,1#5)
```

However the result is a column vector with each element equal to 1. This is because <u>by definition</u> the left-hand end point of a range is taken for a one-component row and the above statement asks for five one-component rows over the range 1#5.

So how can we create column vectors and arrays with different rows? By using the NCO vector to define the structure and using as the second argument not a range but an array, which

- either was previously generated or explicitly concatenated from other arrays or numbers
- or created on the spot; i.e. the second argument is an expression which generates an array as its result.

For example, the column vector (1,2,3,4,5) may be generated in one of four ways:

```
        !ERASE
        Z = ARRAY(5,1#5)
        X = SQRT(Z)-PI
```

Now construct Y by

```
        !ERASE
        Y = ARRAY(5&1,Z)
        Y = ARRAY(5&1,1&2&3&4&5)
        Y = ARRAY(5&1,ARRAY(5,1#5))
        Y = ARRAY(5&1,(PI+X)**2)
```

The user is urged to explore various ways to generate multidimensional arrays to deduce the rule how components from the array result of the second argument are transcribed to the newly created array. The rules may be stated thus:

i) Arrays are stored linearly such that through the sequence of components each index goes through all its permitted values before the next index to the left of it is increased by one.

This sequence of components is called index order.

ii) The ARRAY operator takes the elements of the result of the second argument in index order and fills the newly created array structure in its index order.

iii) If all components of the result of the second argument are exhausted, any unfilled components of the new array are set equal to 1; if there is no second argument, all are set equal to 1.

iv) If all components of the newly created array are filled, then the remaining components of the result of the second argument are discarded.

Try the following:

```
!ERASE
X = ARRAY(2&3)
X = ARRAY(12,1#12)
```

Note that although, because the display screen has finite dimensions, X looks like a 3 x 4 matrix when printed, it actually is a row of 12 numbers: the NCO vector is 12 and not 3&4 as it will be for

```
Y = ARRAY(3&4,X)
```

there Y not only looks like a 3 x 4 matrix, it really is one. Go on with

```
Y = ARRAY(4&3,X)
!ERASE
Y = ARRAY(6&2,Y)
Y = ARRAY(3&2&2,Y)
```

So far Y as well as X had 12 components; now:

```
!ERASE
Y = ARRAY(4&4,X)
Y = ARRAY(3&2,X)
Y = ARRAY(4&3)
Y = ARRAY(3&4&4)*0
Y = ARRAY(2&2)*PI
```

5.4 The transpose operator

The transpose operator (TP) performs a general permutation of the indices of an array. In its simplest form TP appeared with one argument in Session 1, where it simply transposed a row into a column. In general

$$R = TP\ (arg_1,\ arg_2)$$

rearranges the components of the array specified by arg_1 into a new index order specified by arg_2 as a permutation of the indices. For example, if A is a three-dimensional array
$A = \{A_{i1i2i3};\ 1 \leq i_1 \leq n_1;\ 1 \leq i_2 \leq n_2;\ 1 \leq i_3 \leq n_3\}$, then

$$B = TP(A,\ 3\&1\&2)$$

defines B to be a three-dimensional array such that

$$B_{i1i2i3} = A_{i3i1i2}$$

so that

$B = \{B_{j1j2j3};\ B_{j1j2j3} = A_{j3j1j2};\ 1 \leq j_1 \leq n_3;\ 1 \leq j_2 \leq n_1;$
$1 \leq j_3 \leq n_2\}.$

Try the following example and try to account for all components in the new index order:

```
!ERASE
A = ARRAY(3,1#3)+ARRAY(2&1,10&20)+ARRAY(3&1&1,100&200&300)
ATP1 = TP(A)
ATP2 = TP(A, 3&1&2)
```

Further extensions to the transpose operation are described in the CERN SIGMA USER´S MANUAL.

5.5 Trace

Another important array operator is the trace operator TRACE. The TRACE is a generalization of the trace operation of matrix calculus. It contracts an array over one or more dimensions in one or more summations.

$$R = \mathrm{TRACE}\ (\mathrm{arg},\ \mathrm{arg}_1,\ \mathrm{arg}_2,\ \ldots,\ \mathrm{arg}_n)$$

takes the array specified by arg and generates the array R contracted according to arg_1, arg_2, ..., arg_n. The simplest is a straightforward matrix calculus contraction over the right-most two dimensions. In that case arg_1, ..., arg_n may be omitted:

```
        !ERASE
        X = ARRAY (3&3&3, ARRAY(27, 10#270))
        Y = TRACE(X)
```

gives a three component row vector Y such that

$$Y = \{y_i = \sum_{j=1}^{3} x_{ijj};\ 1 \leq i \leq 3\}$$

In this case Y = TRACE(X) is therefore equivalent to

```
        Y = TRACE(X, 2&3)
```

Now try

```
        !ERASE
        X = X                            (to initiate printing of X)
        Y = TRACE(X,3)
```

the result is $Y = \{y_{ij};\ y_{ij} = \sum_{k=1}^{3} x_{ijk};\ 1 \leq i,j \leq 3\}$

```
        !ERASE
        X = X
        Y = TRACE(X, 1&3)
```

the result is $Y = \{y_i;\ y_i = \sum_{k=1}^{3} x_{kik};\ 1 \leq i \leq 3\}$

```
        !ERASE
        X = X
        Y = TRACE(X,1,2,3)
```

the result is $Y = \sum_{i=1}^{3} \sum_{j=1}^{3} \sum_{k=1}^{3} x_{ijk}$

```
        !ERASE
        X = X
        Y = TRACE(X, 1&2&3)
```

the result is $Y = \sum_{k=1}^{3} x_{kkk}$

Hence the concatenation operator & combines those indices which are to be put equal and summed over, while the comma separates different such groups of indices, each such group being summed over independently of the others. More sophisticated features of the trace operator are described in the CERN SIGMA USER´S MANUAL.

5.6 Indexing

```
        !NOPRINT
```

Array elements are addressed by a subscript list attached to an array name.For example, define

```
        X = ARRAY(100, 10#1000)
        Y = ARRAY(2&3&4, 5#28)
```

then the following values: 10, 50, 990, 5, 28 will be printed by

```
        PRINT X(1), X(5), X(99), Y(1,1,1), Y(2,3,4)
```

A one-dimensional array may be used as a subscript to specify more than one component in any dimension. Hence rectangular subarrays or extensions of subarrays may be specified by subscripting. For example, try

```
        !PRINT
        Z = X(1&5&99)
        Z = X(99&1&5&1&1)
        Z = X(ARRAY(10, 31#40))
```

This applies to arrays of any dimension so try

```
        Y = ARRAY(2&3&4, ARRAY(24,1#24))
        Z = Y(1, 2&3, 3&4)
        Z = Y(1, 3&2, 3&4)
        Z = Y(1, 2&3, 4&3)
        Z = Y(1, 3&2, 4&3)
```

In each case Z is a 2 x 2 matrix, but in each case the components appear in a different sequence. The rule for component sequencing on indexing is an extension of the basic idea of index order: go through all right-most index values before incrementing the next index to the left, but go through the indices of each dimension from left to right as specified in the subscript list. With subscripting not all index values may be present and some may be repeated. Thus

```
Z = Y(1, 2&3, 3&4)
```

is equivalent to

```
Z = ARRAY(2&2, Y(1,2,3) & Y(1,2,4) & Y(1,3,3) & Y(1,3,4))
```

while

```
!ERASE
Z = Y(1, 2&3, 1&1)
```

is equivalent to

```
Z = ARRAY(2&2, Y(1,2,1) & Y(1,2,1) & Y(1,3,1) & Y(1,3,1))
```

A missing index in a subscript list denotes all elements in that dimension. For example

```
!ERASE
Z = Y(1, 3, 1&2&3&4)
Z = Y(1, 3, )
```

denote the same third row of the first matrix. As another example, the same two-dimensional subarray is denoted by the two expressions

```
Z = Y( , , 2&3)
Z = Y(1&2, 1&2&3, 2&3).
```

A subscripted variable may appear on the left-hand side of an assignment statement, in which case the right-hand side is evaluated and taken in index order to fill the left-hand side in the extended index order defined by subscripting.

If there are too many components on the right-hand side, surplus components are discarded after all components specified on the left-hand side are filled.

If, however, there are too few components on the right-hand side, and all components specified on the left-hand side are not filled, then the right-hand sequence is repeated in index order until all components specified on the left-hand side are filled. This differs from the definition of the ARRAY operator (see Section 1.6) and allows for the filling of arrays with cyclic patterns. For example, the n x n unit matrix may be defined by

```
!ERASE
N = 10
M = ARRAY(N&N)
M(,) = 1 & ARRAY(N, 0#0).
```

A problem arises in the definition of the NCO of Z if one specified

```
Z = Y(1&2, 3, 3&4)
```

and SIGMA resolves this by defining that the NCO of an array generated by subscripting is obtained by

i) writing down the number of components specified for each subscript dimension (for the above example (2&1&2))

ii) striking out all 1´s from this vector [leaving (2&2) in our example] except that if no components remain, a single component is specified which is set equal to 1. You may find this rule strange, because it gives the new construct a sometimes unwanted structure; try for instance

```
!ERASE
Y = ARRAY(3&1&4, ARRAY(12, 1#12))
```

Y consists of three 1 x 4 matrices (to be distinguished from one 3 x 4 matrix). Now type

```
Z = Y(,,2)
```

which selects the second component of each of the three matrices; you might have wished that the result keeps a memory of the structure of

Y and should be a set of three 1 x 1 matrices (NCO = 3&1&1) while SIGMA has rearranged it into a 1&3 vector.

Even

```
        Y1 = Y(,,)
```

does not reproduce the old Y; it eliminates the 1 from the NCO(Y) and makes Y1 into a 3 x 4 matrix.

These conventions have not been invented to make life difficult for the user, but because in complex situations ambiguities can arise. In any case, if one wishes to obtain a particular structure containing 1´s in the NCO vector, then one can always do it by explicit array definition:

```
        Z1 = ARRAY(3&1&1, Y(,,2))
```

has the structure the user might have expected in the above example.

```
        !NOPRINT
```

Further special array operators:

- the diagonal element selector DIAG
- the reduction operator DROP
- the array element projector PROJ

are somewhat too sophisticated to be discussed here; they are also rarely used. Therefore we refer to the CERN SIGMA USER´S MANUAL.

```
        !SAVE ..., ...
        !STOP
```

END OF SESSION 5

SESSION 6: PROGRAM EDITING (20 minutes)

```
!LOAD ..., ...
```

With the saving facility, you can attempt to construct longer and more complicated programs. The editing facility allows one to correct programs line by line.

Let us construct a program which computes the polynomial $P(x) = a_1 + a_2x + \dots a_nx^{n-1}$ and let us put in some mistakes:

```
SUBROUTINE PO(X,A)
N = NCO(A)
DO 1 J = 0,N
1 PO = (PO + A(N-J))*X
END
```

It is useful to test a new program on a simple case, printing out everything:

```
!PRINT
X = 2
A = 1&2&3
CALL PO(X,A)
```

The message says that when the program comes to the statement PO = (PO + ...) it has no PO available, at least not a numerical entity with that name. Now we correct:

```
!EDIT PO
```

You see the program and a message asking you either to copy or to delete or to insert a new statement. We wish to insert PO = 0 just before the DO loop. To this end we type

```
C.3
```

which copies until statement No. 3 exclusively. Now we insert:

```
PO = 0
```

and are told that we are in "insert mode", which means, we can go on inserting statements. As we have nothing more to insert, we type

```
C.6
```

which copies until statement No. 6 exclusively, that is, until END inclusively. You simply can type C.99 with the same effect, but without having to look carefully what number you should type. Now we look and try again

```
PRINT PO
CALL PO(X,A)
```

The message tells us that there must be something wrong with our DO loop. Indeed, for J = N we try to obtain A(0), while the index 0 does not exist. Hence

```
!EDIT PO
```

now we copy until No. 4 exclusively by

```
C.4
```

and type the correct statement

```
DO 1 J = 0,N-2
```

which makes only A(2) ... A(N) appear in the DO loop; in fact A(1) should be added at the end without being once more multiplied by X. To achieve that, we copy until No. 6 (exclusively)

```
C.6
```

and put in the new statements

```
PO = PO + A(1)
END
```

Instead of typing C.99 we have typed END, which terminates the editing as well (without doing any copying!). Now the program should work:

```
        CALL PO(X,A)
```

How is that possible in spite of our correction? We had better look at the program:

```
        PRINT PO
```

We see that the new statement with J = 0,N-2 and the old one with J = 0,N are both there; we have forgotten to delete the old one. We do this now:

```
        !EDIT PO
        C.5
```

We delete from here until (exclusively) No. 6 by typing

```
        D.6
```

and copy the rest:

```
        C.99
        !ERASE
        CALL PO(X,A)
```

The result is correct. Hence we say

```
        !NOPRINT
        !ERASE
```

and try once more

```
        CALL PO(X,A)
        PRINT PO
```

The text of the program is printed, because although the <u>subroutine</u> PO is executed, the <u>variable</u> PO was local and cannot be printed. If we would declare it global, the program would destroy itself by replacing its text by the value 17. Thus we should make it a function and, furthermore, give it a longer name, which is more mnemonic and less likely to be chosen if working in another area (this is a general rule!).

```
!EDIT PO
FUNCTION POLYNOM(X,A)
```

now we must eliminate the old "program head":

```
D.2
```

Further we insert a descriptor of the function:

```
$
$
$ COMPUTES POLYNOM A(1)+A(2)*X+...+A(N)*X**(N-1)
$
$
C.99
```

The $ sign serves as defining a "comment", i.e. a piece of text which is stored in the program but not executed. We look at the new program:

```
!ERASE
PRINT POLYNOM
PRINT POLYNOM(X,A)
```

Indeed, the name POLYNOM does not appear in an assignment statement; hence

```
!EDIT POLYNOM
C.12
POLYNOM = PO
END
```

We try again:

```
PRINT POLYNOM(X,A)
```

The result is correct. We make one more check:

$y = (x-1)(x-2)(x-3) = -6 + 11x - 6x^2 + x^3$
has the roots 1,2,3. We type

```
        A = -6&11&-6&1
        X = ARRAY(101, 1#3)
        Y = POLYNOM(X,A)
        DISPLAY Y%X
```

It seems to be all right, hence we

```
        !ERASE
        PRINT POLYNOM (and take a copy)
        PROTECT POLYNOM
        !SAVE ..., ...
```

and have the useful function POLYNOM saved for later use. This is a sample of the technique to be employed in program construction. In constructing programs which call chains of subprograms it is useful to not only !PRINT before starting to check the program, but also to type !CHAIN; SIGMA will then send a message each time another subprogram is called (end this procedure by !NOCHAIN)

```
        !STOP
```

END OF SESSION 6

SESSION 7: SOME FUN WITH SPECIAL ARRAY OPERATORS
(30 minutes)

```
        !LOAD ..., . .
```

SIGMA has some special operators faciliating array handling; this and the next Session will give you a feeling of what they can do for you.

7.1 General operators

```
!PRINT
X = ARRAY(8,1#8)
D = DIFF(X)                 (forward difference)
S = SUM(X)                  (running sum)

!ERASE
DS = DIFF(S)                (DIFF is not the inverse of SUM!)
SD = SUM(D)
P = PROD(X)                 (running product)

!ERASE
Y = X-4.4
Z = DEL(Y)                  (Dirac " -function")
Y = X-4.6                   (find the rule for location of 1)
Z = DEL(Y)

!ERASE
X  = ARRAY(3&4,ARRAY(12,1#12))
MI = MIN(X)                 (row-wise   minimum,   structure
                            preserved)
SMI = SMIN(X)               (the smallest element of X)
MA = MAX(X)                 (row-wise   maximum,   structure
                            preserved)
SMA = SMAX(X)               (the largest element of X)

!ERASE
L1 = LS(X,1)
L2 = LS(X,2)                (row-confined  cyclic  left  and
                            right shift)
R2 = LS(X,-2)
```

7.2 Random numbers and histograms, ordering

```
!NOPRINT
!ERASE
X = ARRAY(100)
                            (each time another set of random
RX = RNDM(X)                numbers 0 ≤ z  ≤  1  in  an  array
                            having the structure of x)
RY = RNDM(X)
DISPLAY RY:RX,[*]RY
DISPLAY [HIST]RY

!ERASE
OX = ORDER(RX,RX)           (ordering operator)
OY = ORDER(RY,RY)
DISPLAY [*]OY:OX,OY
DISPLAY [HIST]OY
```

The rule for the ORDER operator is: R = ORDER(A,B) finds that permutation of elements of B which would bring B into a non-descending sequence (row-wise); it then constructs R by applying this permutation (row-wise) to A. (A and B are left unchanged).

```
!ERASE
N = ARRAY(10,1#10)
ON = ORDER(N,-N)
PRINT ON,N
```

Now we sort the random numbers RX into a histogram having 10 bins from 0 to 1:

```
BIN = ARRAY(11,0#1)
HX = HIST(RX,BIN)            (histogram operator)
PRINT HX, TRACE(HX)
```

Thus HIST(RX,BIN) counts how many elements of RX fall into each bin. There are, however, eleven numbers printed, while we have only ten bins. The last (eleventh) bin collects all those numbers which could not be sorted into the ten bins dividing up the range 0#1; we call it the "waste bin". Try

```
!ERASE
BIN1 = ARRAY(6,0#.5)
H = HIST(RNDM(X),BIN1)
PRINT H,TRACE(H)
```

About half of the numbers (repeat the last two statements a few times) lie outside the interval chosen and are collected in the waste bin.

We combine the HIST operator with the histogram display facility (these two are different things):

```
DISPLAY [HIST] HX:BIN
PRINT TRACE(RX)/NCO(RX), TRACE(BIN*HX)
```

What numbers did you expect?

7.3 Integration

The operator QUAD computes the definite running integral with approximately the precision of Simpson´s rule:

$$F(t) = \int_{t_0}^{t} y(t') \, dt' \Longleftrightarrow F = \mathrm{QUAD}(Y,DT)$$

Example:

```
!ERASE
T = ARRAY(101,0#2*PI)
Y = SIN(T)
DT = T(2)-T(1)
F = QUAD(Y,DT)                  (quadrature operator, with step
                                length DT)
ENDF = F(101)
DISPLAY Y:T,[.-2]F
```

The result should be

$$F(t) = \int_{0}^{t} \sin(t') \, dt' = 1 - \cos(t)$$

$$\mathrm{ENDF} = \int_{0}^{2\pi} \sin(t') \, dt' = 0 \; ;$$

Check by typing

```
[.-3] 1 - COS(T)
PRINT ENDF
```

The error is invisible but not zero (e.g. ENDF is not zero). We check the error by typing

```
DISPLAY F-(1-COS(T))
```

In the present case the maximal error is < 10^{-6}.

It is not useful to construct a differentiation operator, because it is much too sensitive to round-off errors and to extrapolations at the ends.

```
!SAVE ..., ...
!STOP
```

END OF SESSION 7

SESSION 8: MORE FUN WITH SPECIAL ARRAY OPERATORS: INTERPOLATION AND FUNCTION INVERSION (35 minutes)

```
!LOAD ..., ...
```

8.1 Interpolation

Given a range a#b and a function over it, for example: FX,X. Let Y be a vector of values

$$Y = \{y_1, y_2, \ldots, y_n\} = \{x_1, x_2, \ldots, x_k\}.$$

We have in SIGMA an operator EVAL (evaluation) which can compute F(Y) by linear interpolation.

```
!ERASE
X = ARRAY(11,0#10)
FX = X*X
Y = ARRAY(10,.5#9.5)
FY = EVAL(FX,X,Y)          (evaluation operator)

!ERASE
PRINT TP(X)&TP(FX),TP(Y)&TP(Y*Y)&TP(FY)
```

You see the linear interpolation result in the last column, the exact result in the middle.

```
DISPLAY FX:X, [.-2]FY:Y
```

Before proceeding do a few examples of your own with more complicated FX and also with a Y range only partly overlapping the X range.

8.2 Function inversion

EVAL is -- in many cases -- useful in function inversion. Suppose we have a function y = f(x) which we can easily cast in the form of a SIGMA FUNCTION, but what we really need is x as a function of y: x = g(y) and we cannot construct G explicitly. What we can do is to compute a table y = f(x) for equidistant x, but what we need is a table x = g(y) for equidistant y. Take a trivial example $f(x) = x^2$, where we can check the method. Type

```
FUNCTION F(X)
F = X*X
END

FX = F(X)
DISPLAY [HIST]X:FX,X
```

We have displayed X as function of FX, but the [HIST] display shows how much the points on the FX axis deviate from being equidistant. Suppose we want $x = g(y)$ (here y) for 11 equidistant points

$$0 \leq y_i \leq 100;$$

```
!ERASE
Y = ARRAY(11,0#100)
SQX = EVAL(X,FX,Y)
DISPLAY X:FX, [0]SQX:Y
```

The EVAL operator has found the values at the equidistant Y points by linear interpolation; the result can be improved by iteration if the FUNCTION F(X) is explicitly known as in our example:

```
!ERASE
SQX1 = EVAL(SQX,F(SQX),Y)
SQX2 = EVAL(SQX1,F(SQX1),Y) etc.
```

This can be continued until the precision equals the working precision of the computer. Compare the approximations with the exact results:

```
!ERASE
PRINT TP(SQX)&TP(SQX1)&TP(SQX2)&TP(SQRT(Y))
```

Now try to invert $\{y = \sin(x);\ 0 \leq x \leq 2\pi\}$ with this method and see where it fails. However, the display facility of SIGMA is just as useful in showing things which do not work (and indicating why) as for workable cases.

8.3 Finding zeros of functions

Let us find one of the zeros of $f(x) = x^3 - 6x^2 + 11x - 6$. We use our function POLYNOM(X,A) [Session 6]:

```
A = -6&11&-6&1
X = ARRAY(101,-100#100)
DISPLAY POLYNOM(X,A):X
```

Obviously the range was too large;

```
X = ARRAY(101,-20#20)
DISPLAY POLYNOM(X,A):X
X = ARRAY(101,-5#5)
DISPLAY POLYNOM(X,A):X
X = ARRAY(101,0#4)
DISPLAY POLYNOM(X,A):X
```

Where exactly lies the zero located near x = 2?

```
X = ARRAY(101,1.9#2.1)
DISPLAY POLYNOM(X,A):X
PRINT EVAL(X,POLYNOM(X,A),0)
```

What we have done is almost the same as function inversion: we have calculated the inverse function at one single point. With this method roots can be located with arbitrary precision, if the function is explicitly given (as a FUNCTION or otherwise).

```
!SAVE ..., ...
!STOP
```

END OF SESSION 8

SESSION 9: MATRIX OPERATIONS

```
!LOAD ..., ...
```

SIGMA has a few special matrix operators: C = MULT(A,B) gives $C_{ik} = \sum_j A_{ij}B_{jk}$ for any two compatible rectangular matrices: if A and B are vectors, C is the scalar product; if scalars, C is the ordinary product. The determinant of a quadratic matrix Q is D = DET(Q), its inverse I = INV(Q), its eigenvalues are E = EIGVAL(Q) and its eigenvectors F = EIGVEC(Q).

We turn to examples:

Let $x' = M(ALPHA)x$ be the new coordinates of a point P after the coordinate axes in the plane have been rotated by ALPHA degrees ("passive" rotation)

```
FUNCTION ROT(ALPHA)
$ ROT IS TWO DIMENSIONAL PASSIVE
$ ROTATION MATRIX FOR ALPHA (DEGREES)
A = 2*PI*ALPHA/360
C = COS(A)
S = SIN(A)
ROT = ARRAY(2&2,C&S&-S&C)
END

M15 = ROT(15)
M25 = ROT(25)
M40 = ROT(40)
!ERASE
PRINT MULT(M15,M25),M40    Compare!
IM25 = INV(M25)
PRINT IM25, ROT(-25)       Compare!
PRINT MULT(M25,INV(M25))   Unit matrix
!ERASE
PRINT DET(M25),DET(IM25),DET(M40)
```

Now construct a matrix

$$a = \begin{pmatrix} 5 & 0 \\ 0 & -3 \end{pmatrix}$$

by typing

```
A = ARRAY(2&2,5&0&0&-3)
```

and consider it in the rotated frame: $(ax)' = a'x'$ hence $a' = MaM^{-1}$;

take ALPHA = 40°:

```
        A1 = MULT(MULT(M40,A),INV(M40))
```

Determinants are invariant, hence

```
        PRINT DET(A),DET(A1)
```

Eigenvalues are invariant, too:

```
        EVLA = EIGVAL(A)
        EVLA1 = EIGVAL(A1)
        PRINT EVLA,EVLA1
```

Obviously the result of EIGVAL is a vector with the eigenvalues as elements. The order in which the eigenvalues are sorted into the resulting vector, is unpredictable:

```
        C = ARRAY (10&10)*0
        DO 1 K=1,10
        C(K,K) = K
        1 CONTINUE
        !ERASE
        PRINT EIGVAL(C),EIGVAL(C+1E-20),EIGVAL(-C)
```

Back to our two-dimensional example!

```
        !ERASE
        EVCA = EIGVEC(A)
        EVCA1 = EIGVEC(A1)
        PRINT EVCA,EVCA1
```

While the order of the eigenvalues is unpredictable, the order of the eigenvectors is -- at least -- the same as that of the eigenvalues: the first row of the result of EIGVEC is the eigenvector belonging to the eigenvalue which is the first element of the result of EIGVAL and so on.

While EVCA contains the vectors (0,1) and (1,0) properly normalized (within the precision of the computer), EVCA1 is not normalized. Hence, as a rule, normalize eigenvectors if necessary.

Note that the eigenvectors are the rows, not the columns; hence if one wishes to multiply the eigenvectors by a matrix, one has to transpose them first! And do not forget that they appear in an unpredictable order! Geometrically, the eigenvectors of A do not depend on the choice of the coordinate axes, hence the operator A and its eigenvectors are geometrically invariant; they only have different components in different coordinate systems: the eigenvectors of A1 must have the same components as those of A seen from the rotated coordinate frame:

```
!ERASE
EVCA11 = MULT(M40,TP(EIGVEC(A)))
PRINT EVCA1,EVCA11
```

Again arbitrary normalization makes it difficult to compare;

```
PRINT EVCA1/EVCA11
```

which shows that both differ only by normalization.

All this works for n x n matrices and for sets of matrices combined in higher dimensional arrays as well. Non-square matrices can be multiplied if compatible in the sense of matrix multiplication.

```
!SAVE ..., ...
!STOP
```

END OF SESSION 9

SESSION 10: DIALOGUE WITH SIGMA
(30 minutes)

10.1 Modes of operation

```
!LOAD ..., ...
```

You can, apart from computing interactively, communicate with SIGMA in another way; namely, by telling it to deliver some special information, to display in a particular way, or to operate in a mode different from usual.

You know already a few of the commands used for this communication; namely, commands which usually are preceded by an exclamation mark (!); for instance you know

```
!NAMES,        !EDIT,         !ERASE,      !PRINT
!NOPRINT,      !LOAD,         !SAVE,       !DELETE
!CHAIN,        !NOCHAIN,      !CLEAR
```

The !sign is necessary to distinguish the following word from a possible name; for instance try

```
X = ARRAY(101,0#1)
Y = X**2
EDIT = Y**X
DISPLAY Y:X,EDIT
```

It is seen that EDIT is taken as a name and is displayed.

We shall now try some other communication commands. Type

```
!ERASE
T = ARRAY(101,0#4*PI)
R = LOG(1+T)
X = R*COS(T)
Y = R*SIN(2*T)
DISPLAY Y:X
DISPLAY 2*Y:X/2
```

You see that the two figures are identical, while the scale of the axes has changed. If one wishes to see how figures are deformed by a change of parameters, one must make sure that SIGMA plots them all to the same scale instead of adjusting each figure to a scale for an optimized use of the screen. To do this, we type !NOSCALE, which fixes the scale as it was established by the last DISPLAY statement preceding this command. The scale remains fixed for all following displays until !SCALE restores automatic scaling. Type

```
DISPLAY Y:X
!NOSCALE
DISPLAY 2*Y:X/2
!SCALE
```

Now we come to logarithmic scaling. Type

```
Y = Y+3
X = X+3
!LOGY
DISPLAY Y:X
!LOGX
DISPLAY Y:X
```

Display remains double logarithmic until we type

```
!LINY
DISPLAY Y:X
!LINX
DISPLAY Y:X
```

We may put a grid over the figure for easier reading of coordinates

```
!GRID
DISPLAY Y:X
!LOGY
!LOGX
DISPLAY Y:X
```

Instead of removing grid and log, we can simply type

```
!NORMAL                    (axes, linear scale, no grid,
                            frame, automatic scaling)
DISPLAY Y:X
```

We also may wish to see the curve all alone:

```
!NOFRAME
!NOAX
DISPLAY Y:X
!NORMAL
```

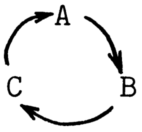
The labels on the axes may be too large, or you may wish to print a long program; type

```
!SMALL
DISPLAY Y:X
!ERASE
PRINT X
!LARGE
```

For testing nested programs we type !CHAIN and from now on SIGMA informs us which program is called from where and when its end is reached. Type

```
SUBROUTINE A
CALL B
END

SUBROUTINE B
CALL C
END

SUBROUTINE C
PRINT ´THIS IS HOW !CHAIN WORKS´
END

CALL A
!CHAIN
CALL A
```

Suppose in subroutine C we had, by mistake, put in a statement CALL A; then SIGMA would enter into the infinite loop



and only a certain stopping procedure(namely: ESC A RETURN) , which would destroy your current work and oblige you to reload, could return control to the terminal. To prevent that, SIGMA counts all automatically executed statements with a "limit counter" and stops, when a certain limit is reached (4000 statements). The counter is reset to zero each time a manual command from the terminal is executed. Let us try it; we can change the limit

```
!LIMIT 20
!EDIT C
C.2
PRINT ´LIMIT IS 20´
CALL A
END
```

Now, without the limit counter, CALL A would lead to an infinite loop; in reality:

```
        !ERASE
        CALL A
```

You should now type STOP. We reset

```
        !LIMIT 4000
        DELETE A,B,C
```

You may not always wish to have eight digits printed. Construct a matrix with elements a_{ik} = ik

```
        !ERASE
        M = 1&2&3&4
        MM = TP(10*M)+M
        PRINT MM
```

Not very nice; now

```
        !DIGITS 2
        PRINT MM
```

This looks better. We reset

```
        !DIGITS 8
```

Of course, playing around with the system, we may have forgotten which of the various options we chose. For information type

```
        !STATUS
```

There are a few entries in the printed status report which we do not discuss now, but there is one in line 8 from above:

NO OF FREE NAME SPACES (some number)

which means that you cannot define an arbitrary number of names, in total only about 440. If you go beyond that, you must first delete some old ones, or ask for another workspace with another name. If you disregard this advice, you will receive an error message telling you about OVERFLOW and there may be damage to your current work.

10.2 Copying information from another SIGMA user

You may have learned about some SIGMA program or a set of data, constructed by somebody else and stored on his workspace, which might be of use for you. You need not type all this into your terminal; you can simply copy it from your colleague´s workspace. Let XYZ, DATA, PROGRAM be required the items and HISWORK the name of his workspace (you do not need his identifier!). In this case you would type

```
!COPY HISWORK, XYZ, DATA, PROGRAM
```

If you have already yourself items with such names, they must be renamed or deleted beforehand, otherwise you will receive an error message (no damage is done).

10.3 The SIGMA Library Workspace

A number of useful SIGMA programs are collected in the SIGMA Library Workspace.

```
!COPY LIBRARY, CONTENT
!ERASE
CALL CONTENT
```

CONTENT informs you about the contents of the library and about the procedure to be followed in order to copy any of the library programs into your workspace. Copy a few of them and try them out. Delete them if not needed further; this keeps your workspace short (you can get them back at any time by copying again).

```
!SAVE ..., ...
!STOP
```

END OF SESSION 10

SESSION 11: MORE ON DIALOGUE WITH SIGMA; DISPLAY MODES (30 minutes)

```
!LOAD ..., ...
```

We have learned about some simple display modes. Here are a few more sophisticated ones.

!NOERA is used to suppress the automatic erasing of the screen before a new display command is executed; !ERA will restore automatic erase.

If in a program you have different displays following each other, the next display may start before there was the time to look at or copy the previous one. For this we put !WAIT just at the point where the program should wait indefinitely until you type GO (or STOP to leave the program and go back to manual); instead you can insert !PAUSE 25 (or any other number) which causes the program to idle 25 seconds and then continue. We can also define the size of the display by !SETW followed by a four-component vector giving the coordinates of Y_{min}, Y_{max}, X_{min}, X_{max}, in this order, called the "window coordinates". The maximal window will depend on the type of terminal screen; for example for the Tektronix T4012 the screen is limited by

$Y_{lim} = 0$

$Y^{lim} = 780$

$X_{lim} = 0$

$X^{lim} = 1023$

and SIGMA uses normally

$Y_{min} = 80$

$Y_{max} = 746$

$X_{min} = 338$

$X_{max} = 1000$

Let us define some windows; type

```
WMAX = 0&780&0&1023
WNORM = 80&746&338&1000
WLEFT = 0&768&20&500
WRIGHT = 0&768&510&1010
W11 = 400&780&0&500
W12 = 400&780&523&1023
W21 = 0&380&0&500
W22 = 0&380&523&1023

PROTECT WMAX, WNORM, WLEFT, WRIGHT, W11, W12, W21, W22
!SAVE ..., ...
```

With these windows and the !NOERA command, one may draw one, two, or four figures on the screen. We shall now study the use of all this by building them into a program via several editing steps. Type

```
SUBROUTINE SHOW
X = ARRAY(101,0#2*PI)
Y1 = X*X*SIN(2*X)
Y2 = 20*EXP(-X/2)*COS(2*X)
DISPLAY Y1:X
DISPLAY Y2
DISPLAY Y1,[.-]Y2
DISPLAY Y1:Y2
END

CALL SHOW
```

You hardly had the time to look at the figures. There are two ways out: either insert !WAIT or !PAUSE 20 or display all four on one screen. We define two new subroutines SHOSLO for slowly displaying and SHOW4 for four pictures:

```
!EDIT SHOW
SUBROUTINE SHOSLO       (new title)
D.2                     (delete old title)
C.6
!WAIT                   (waits for STOP or GO)
C.7
!WAIT
C.8
!PAUSE 30               (30 seconds pause)
C.99

CALL SHOSLO
```

This is already nice. Maybe we like the last curve so much that we display it once more without axes and frame:

```
!EDIT SHOSLO
C.12
!WAIT
!NOAX
!NOFRAME
DISPLAY Y1:Y2
!NORMAL
```

```
END
CALL SHOSLO
```

We can be satisfied with this. Now all four in one figure:

```
!EDIT SHOW
SUBROUTINE SHOW4
D.2
GLOBAL W11, W12, W21, W22, WNORM
C.5
!SETW W11
C.6
!SETW W12
C.7
!SETW W21
C.8
!SETW W22
C.9
END

CALL SHOW4
```

Obviously we now need the !NOERA to avoid automatic erasing. Also the large labels in the small figures are disturbing; we use !SMALL to cure this. Hence

```
!EDIT SHOW4
C.6
!ERASE

!NOERA
!SMALL
C.14
!ERA
!LARGE
!SETW WNORM
END

CALL SHOW4
```

(Note the difference between !ERA and !ERASE)

We may wish to have a figure without axes, but with four frames:

```
!NOAX
CALL SHOW4
!AXES
```

Still more sophisticated commands are available but seldom needed; some of them have already been used in earlier sessions. Here are a few examples:

```
X = ARRAY(300,0#8)
F = COSH(X) + SIN(1/(X*X+0.1))
DISPLAY F:X                  [The    sin    oscillations    remain
                             invisible]

DISPLAY(=) F:X               [equal  scale  on  both axes; does
                             not help in this case]

DISPLAY(0#5:0#3)F:X          [limits display  of  F  and  X  to
                             0 ≤ F ≤ 5, 0 ≤ X ≤ 3; oscillations
                             become visible]

F                            ["incomplete  display  statement";
                             lets F flash once]

SQRT(F)                      [adds √F to the display]

[*]F                         [adds *,S at the points computed;
[S] SQRT(F)                  the   joining   lines  are  merely
                             linear interpolations drawn]

[S]SQRT(F)

X = ARRAY(21,-1#1)
Y = X**2
DISPLAY[0]Y:X
DISPLAY[PARABOLA]Y
Y
DISPLAY[.-5]Y:X
DISPLAY[.-]Y
!SAVE ..., ...
!STOP
```

END OF SESSION 11

SESSION 12: TOPOLOGICAL ARITHMETIC
(40 minutes)

```
!LOAD ..., ...
```

Apart from automatic array handling and convenient graphical displays, topological arithmetic is one of the most powerful devices in SIGMA. You know its simplest aspect: construction of functions of two variables, from Session 1; the basic idea introduced there will now be generalized.

Remember that in arithmetical operations SIGMA combines arrays component-wise. If they have exactly the same structure (i.e. identical NCO vectors), this is uniquely possible. However, there is another situation, where this is also uniquely possible, although their NCO vectors are not identical.

Consider an example of what was said in Session 1.

```
!PRINT
X = ARRAY(4,1#4)
Y = ARRAY(5,0#4)
Z = X+Y
```

Of course, component-wise addition is impossible. As in Session 1 we transpose Y:

```
YT = TP(Y)
Z = X+YT
```

This works, because a unique "component by component" combination (in an extended sense) is possible (see figure, Session 1).

```
PRINT NCO(X), NCO(YT)
```

We have NCO(X) = 1&4 and NCO(YT) = 5&1. Write NCO(YT) below NCO(X):

1 4 (any number of 1´s to the left are

5 1 implied)

What SIGMA (in a way) does, is to make the two NCO´s equal by repeating those structures which appear only once (where the 1´s occur) identically as often as required to match the other NCO:

$$X = \{1 \quad 2 \quad 3 \quad 4\} \longrightarrow X' = \left\{\begin{matrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix}\right\}$$

$$YT = \left\{\begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix}\right\} \qquad YT' = \left\{\begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{matrix}\right\}$$

NCO(X) = 1&4 NCO(X´) = 5&4
NCO(YT) = 5&1 NCO(YT´) = 5&4

SIGMA combines the new compatible arrays X´ and YT´ component by component (in reality it does not expand X and YT, but only arranges its internal DO loops in a proper way to produce the same result as if they had been expanded). The above procedure neither loses nor arbitrarily adds any information. Obviously it can be applied to arrays of any dimension: if

$$NCO(A) = \{a_N, a_{N-1}, \ldots, a_i \ldots, a_1\}$$

$$NCO(B) = \{b_N, b_{N-1}, \ldots, b_i \ldots, b_1\}$$

and (OP) is any binary operator, the statement C = A(OP)B gives a result if and only if

i) either $a_i = b_i$ for all $i = 1 \ldots N$

ii) or, if some $a_i \neq b_i$, then either $a_i = 1$ or $b_i = 1$.

We then call "A and B compatible for topological arithmetic". Note that if these conditions are not fulfilled, they can always be enforced by restructuring arrays by inserting 1´s into their NCO.

```
!ERASE
X = ARRAY(3&3,ARRAY(9,1#9))
Y = ARRAY(2&2,ARRAY(4,1#4))
X1 = ARRAY(3&3&1&1,X)
Z1 = X1 + Y

!ERASE
X2 = ARRAY(3&1&1&3,X)
Y2 = ARRAY(2&2&1,Y)
Z2 = X2 + Y2
PRINT NCO(Z1), NCO(Z2)
```

Try a few more choices to make X and Y compatible.

A few examples are as follows:

```
!ERASE
X = ARRAY(4,1#4)
Y = ARRAY(4&1,X)
Z = ARRAY(4&1&1,X)
!DIGITS 3

!ERASE
MATRIX = 100*Z+10*Y+X
!DIGITS 8
!NOPRINT

!ERASE

FUNCTION LENGTH(X,Y,Z)
LENGTH = SQRT(X*X+Y*Y+Z*Z)
END

PRINT LENGTH(1,2,2)
A = ARRAY(4,0#3)
B = ARRAY(3&1,ARRAY(3,-1#1))
C = ARRAY(2&1&1,0&1)

!ERASE
PRINT A,B,C, LENGTH(A,B,C)
```

The rules for topological arithmetic allow construction of functions of several (up to 10) variables. Be aware of the rapid growth of the total number of components with the array dimension (type !STATUS to learn the maximum number of array components permitted).

Summary:

> Topological arithmetic extends the component-by-component rule to the combination of arrays over the topological product of their subscript spaces. In general, programs will automatically do the correct topological combination if the arrays match in the sense described above. It is up to you to give your arrays the required structure before calling programs or typing statements.

A last example:

```
!ERASE

FUNCTION F(X,Y)
F = X*SIN(Y)-Y*COS(X)
END

X = ARRAY(21,-PI#PI)
Y = X
Z = F(X,Y)
Z1 = F(X,TP(Y))
DISPLAY Z:X
DISPLAY Z1:X
DISPLAY TP(Z1):Y
```

Interpret the results!

Topological arithmetic is worth being learned well.

```
!SAVE ..., ...
!STOP
```

END OF SESSION 12

SESSION 13: COMPLEX NUMBERS AND FUNCTIONS
(25 minutes)

```
!LOAD ..., ...
```

SIGMA does not make a systematic distinction between real numbers and complex ones; there is, however, a distinction in practice: while, in principle, a real number is a complex number with zero imaginary part, it would be wasteful to carry this zero imaginary part through a computation which remains entirely in the real domain. Hence in ordinary, real computations, SIGMA drops the redundant imaginary part. SIGMA is, however, constantly watching whether a computation leads into the complex domain; if that happens, it "analytically continues" from the real axis into the complex plane and from then on carries both the real and the imaginary part of all elements in all arrays of which at least one element has become complex. Thus, in principle, one need not worry about a computation going complex, SIGMA will care. In practice, unfortunately, it is not so simple

- because arrays, having suddenly twice as many components, may become too large;
- because DISPLAY Y:X may not give the expected picture if Y and/or X is complex;
- because some array operators are "non-analytic" (e.g. MAX, ORDER, HIST and others) and because many of the available functions (see Appendix 1) and of your own user-written programs will only work for real arguments.

We shall not discuss all the possible pitfalls here; you will realize soon enough if something went wrong and your own growing experience, together with the CERN SIGMA USER´S MANUAL, will help you. Here we consider only a few cases which do work.

Complex numbers can be entered in two ways:

```
!ERASE
Z1 = 3I4
Z2 = 2A3
```

The first is interpreted as 3+4i, the second as 2 exp (3i).

```
PRINT Z1, Z2, ABS(Z1), ABS(Z2), Z1+Z2
```

Internally all complex numbers are stored as pairs of real numbers (real part; imaginary part) and the above 2A3 is converted to that form as you see from the printed Z2.

Now type

```
A = 3
B = 4
Z = AIB
```

It does not work, because AIB is interpreted as a name; had SIGMA been designed to interpret it as a complex number, all names containing the letter I would have to be banned.

How then do we form a complex array from two real ones? Type

```
!ERASE
Z = CPLX(A,B)
M = ABS(Z)
R = REAL(Z)
I = IMAG(Z)
CZ = CONJ(Z)
PRINT A,B,Z,CZ,R,I,M
```

Complex and real arrays may be mixed:

```
PRINT Z+A, Z+B, A*Z
```

Now let us see how SIGMA goes complex automatically:

```
!ERASE
PRINT SQRT(4), SQRT(-4)
X = -4&-1&1&4
Q = SQRT(X)
Q2 = Q*Q
PRINT Q,ABS(Q),Q2
```

Next we study complex functions. The best way to do this is to look at the mapping of the complex plane by constructing arcs in the original plane and see where they go in the image plane.

```
!ERASE
T = ARRAY(101,0#2*PI)

R = LOG(1+T)
X = R*COS(T)
```

```
        Y = R*SIN(T)
        Z = CPLX(X,Y)
```

This is a logarithmic spiral

$$Z(t) = e^{it} \log (1 + t) \qquad 0 \leq t \leq 2*PI$$

How do we plot it? Try

```
        DISPLAY Z:T
```

So this does not work. Whenever a display gives you some figure like this, you know that your computation went complex somewhere. The correct way to plot the arc Z(t) is, of course,

```
        DISPLAY IMAG(Z):REAL(Z)
```

Now we try some functions:

```
        !ERASE
        R = SQRT(Z)
        S = Z**2
        E = EXP(Z)
```

and plot them:

```
        DISPLAY IMAG(Z):REAL(Z), IMAG(R):REAL(R)
```

Where is the square root cut located? Next

```
        DISPLAY IMAG(Z):REAL(Z), IMAG(S):REAL(S)
```

finally

```
        DISPLAY IMAG(Z):REAL(Z), IMAG(E):REAL(E)
```

It is rather time-consuming to type all this each time; also we should have inserted (=) after DISPLAY to have equal scale on both axes. We write a small subroutine for complex plotting:

```
      !ERASE

  SUBROUTINE COMPLOT(W,Z)
  $
  $   ============================================
  $
  $   PLOTS CONTOURS OF W (FULL) AND Z (BROKEN)
  $   IN COMPLEX PLANE
  $
  $   ============================================
  $
  DISPLAY(=)IMAG(W):REAL(W),[.-2]IMAG(Z):REAL(Z)
  END
```

[DISPLAY(=) instructs SIGMA to use the same scale on the X and Y axis]

```
      PROTECT COMPLOT
      CALL COMPLOT(R,Z)
      CALL COMPLOT(S,Z)
      CALL COMPLOT(E,Z)
      CALL COMPLOT(Z/(1+Z),Z)
```

Try a few more examples. Then !SAVE so that COMPLOT becomes part of your private program library:

```
      !SAVE ..., ...
      !STOP
```

END OF SESSION 13

SESSION 14: MORE COMPLEX MAPPING
(60 minutes)

```
        !LOAD ..., ...
```

Instead of mapping one curve, it is much better to map a regular set of curves. The following subroutines provide you with

- a rectangular grid (COMGRID)
- a rectangular set of radii and circles (COMNET)
- a segment of the radii-circle set (COMSECT)

You may !COPY all three from the SIGMA library (see end of Session 10, and Appendix 2); if at your place there is no such library, type

```
SUBROUTINE COMGRID(NV,NH,SIZE,CENTER)
$
$   =============================================
$
$   DRAWS NV VERTICAL,NH HORIZONTAL LINES IN SQUARE
$   OF SIDELENGTH SIZE, CENTERED AT CENTER
$   AVAILABLE AS ZV,ZH
$
$   =============================================
$
GLOBAL ZV,ZH
XV=ARRAY(NV,-SIZE/2#SIZE/2)
YH=ARRAY(NH,-SIZE/2#SIZE/2)
X=ARRAY(51,-SIZE/2#SIZE/2)
Y=X
ZV=CPLX(TP(XV),Y)+CENTER
ZH=CPLX(X,TP(YH))+CENTER
CALL COMPLOT(ZV,ZH)
END
```

```
    PROTECT COMGRID
    CALL COMGRID(5,4,2,1I1)
```

From now on, under the name ZV, you have five vertical lines of 51 points each and under the name ZH four horizontal lines of 51 points each at your disposal as arguments of the complex function you wish to study in the neighbourhood of z = 1+i. Consider e^z and z+1/(1+z). You type

```
    WV = EXP(ZV)
    WH = EXP(ZH)
    CALL COMPLOT(WV,WH)
    WV1 = ZV+1/(1+ZV)
    WH1 = ZH+1/(1+ZH)
    CALL COMPLOT(WV1,WH1)
```

In many cases you wish to see what the mapping does in the vicinity of a singularity: consider w = 1/z. Make sure that no line of our grid actually goes through zero; type

```
        CALL COMGRID(10,10,1,0)
        CALL COMPLOT(1/ZV,1/ZH)
```

It might be more instructive to put a net of circles and radii; type (or !COPY from SIGMA library)

```
        !ERASE

     SUBROUTINE COMNET(NR,NC,RMIN,RMAX,CENTER)
     $
     $  ==========================================
     $
     $  DRAWS NR RADII,NC CIRCLES WITH RMIN < R < RMAX
     $  CENTERED AT CENTER,AVAILABLE AS ZR AND ZC
     $
     $  ==========================================
     $
     GLOBAL ZR,ZC
     RR=ARRAY(51,RMIN#RMAX)
     RC=ARRAY(NC,RMIN#RMAX)
     P=ARRAY(51,-PI#PI*.9999)
     PR=ARRAY(NR+1,-PI#PI*.9999)
     ZR=CPLX(RR*COS(TP(PR)),RR*SIN(TP(PR)))+CENTER
     ZC=CPLX(TP(RC)*COS(P),TP(RC)*SIN(P))+CENTER
     CALL COMPLOT(ZR,ZC)
     END
```

```
        PROTECT COMNET
         !SAVE   ...,  ...  (to  save  the two subroutines in case of
                                computer breakdown)
        CALL COMNET(10,10,0.2,1,1I1)
```

You see that the point 1+i is not touched if RMIN > 0; if you wish to see the mapping in the neighbourhood of a singularity, you may employ COMNET with Z put right on the singular point; consider log(z-(1+i)):

```
        CALL COMPLOT(LOG(ZR-1I1), LOG(ZC-1I1))
```

Finally one might wish to see only some sector of the set of radii and circles: type (or !COPY from SIGMA library)

```
        !ERASE
```

```
SUBROUTINE COMSECT(NR,NC,RMIN,RMAX,AMIN,AMAX,CENTER)
$
$  ===================================================
$
$  DRAWS NR RADII AND NC CIRCLES WITH RMIN < R < RMAX
$  AMIN < ANGLE(RADIAN) < AMAX
$  ENTERED AT CENTER. AVAILABLE AS ZR AND ZC
$
$  ===================================================
$
GLOBAL ZR,ZC
RR=ARRAY(51,RMIN#RMAX)
RC=ARRAY(NC,RMIN#RMAX)
P=ARRAY(51,AMIN#AMAX)
PR=ARRAY(NR,AMIN#AMAX)
ZR=CPLX(RR*COS(TP(PR)),RR*SIN(TP(PR)))+CENTER
ZC=CPLX(TP(RC)*COS(P),TP(RC)*SIN(P))+CENTER
CALL COMPLOT(ZR,ZC)
END
```

```
   PROTECT COMSECT
   CALL COMSECT(5.5,0.2,1,PI/5,PI/7,1I1)
   WR = SIN(1I1-ZR)
   WC = SIN(1I1-ZC)
   CALL COMPLOT(WR,WC)
   !SAVE ..., ...
```

Now try to understand the programs COMGRID, COMNET, COMSECT!

```
   !STOP
```

END OF SESSION 14

SESSION 15: INPUT AND OUTPUT OF DATA; COMMUNICATION WITH THE MAIN COMPUTING CENTRE VIA FILES.
(50 minutes)

LOGIN WITHOUT ENTERING SIGMA; REMAIN IN COMMAND MODE.

Remark: In what follows, commands within brackets [ ] are optional. Do not type the brackets!

15.1 Input from files

So far all data-input has been from your terminal. This is satisfactory in cases where a few parameters define a problem and most of the data required by a problem are generated by mathematical functions. There are, however, problems which require many multidigit numbers as input data so that retyping by hand is both time-consuming and error prone. This session will illustrate the simplest techniques to handle such problems. As an example, let us now create a "local" file (that will be destroyed after LOGOUT)*)

This time, after LOGIN, you did not enter SIGMA; you remain in INTERCOM in COMMAND mode: COMMAND - is displayed at your terminal. Type

```
|       ETL, 1111
```

We call the INTERCOM editor by typing

```
|       EDITOR
```

When two dots .. appear you are in EDIT mode.

*) If you need to keep data after LOGOUT and use it with SIGMA later on, you will need a "permanent" file (not to be confused with your workspace). In this case you will have to contact your CUAC (Computer Users Advisory Committee) representative to have such a permanenet file allocated.

Let us create some data. Please type

```
CREATE
```

The line number is displayed. After the line number please type, for example,

```
1
2
3
4
5
6
7
=
```

The = sign ends your data creation; now save these data on a local file, call it MYFILE, typing

```
SAVE MYFILE N
```

where N indicates that you do not want the line number saved. To exit the editor type

```
BYE
```

You have created a local file with the filename MYFILE; have a look for yourself by typing

```
FILES
```

The following procedure of using (in SIGMA) data on file applies as well to any existing permanent file:

If your permanent file was not created by CDC 6000, please make sure that the format of this file is SIGMA compatible. In case of doubt, please contact the PEO (Program Enquiry Office).

If you have an existing permanent file on the CDC 6000 series you can make your file local by typing

```
ATTACH, MYFILE, PFNAME, ID=YOURID*)
```

Your permanent file is now a local file with the temporary name MYFILE and you can use the procedure below.

You are still in COMMAND mode. Enter SIGMA the usual way. For our first exercise loading your workspace is not necessary. (In general you would load your workspace with programs which will analyse your data.) You are ready to copy the data from the file to a SIGMA array. This is done in three steps:

i) make the file known to SIGMA by typing

```
GET MYFILE
```

ii) if you want to copy the first N numbers from MYFILE and place them in the one-dimensional array A the command is (do not type yet)

```
A = READ (MYFILE, N)
```

where N is either an integer or a variable containing an integer, or an expression resulting in an integer.

Provided there were at least N numbers on the file MYFILE, the array A will now contain a vector of N numbers. SIGMA reads as many records as necessary (and existing) from the file to obtain N numbers. SIGMA also converts them from character to floating point, using the same conventions as for terminal entries.

----

*) Here PFNAME stands symbolically for the actual name of your permanent file, YOURID stands for your own identifier and MYFILE for the name you wish to give to your (permanent) file while it is local. Hence in an actual situation the above three names will be different from MYFILE, PFNAME, YOURID.

To make sure you get the first N numbers you must rewind your file to start reading from the beginning; type

```
        REWIND MYFILE
```

For example, you want the first four data placed in a one-dimensional array A: type

```
        A = READ (MYFILE,4)
```

If you do want the next three data placed in array B, DO NOT rewind; just type

```
        B = READ (MYFILE,3)
```

To place the whole file on an array C, type

```
        REWIND MYFILE
        C = READ (MYFILE,7)
```

To see what you got, type

```
        PRINT A,B,C
```

iii) Release the file MYFILE from the SIGMA system by executing (do not type yet)

```
         PUT MYFILE     - makes file unknown to SIGMA (delete) -
or
         RELEASE MYFILE - filename afterwards refers to a
                          non-existing file;
```

Type

```
    PUT MYFILE
    !STOP
    YES
```

If you are back in COMMAND mode of INTERCOM, look at your files:

```
    FILES
```

You see SIGMA with an asterisk; you had attched SIGMA; it is a permanent file. MYFILE is a local file; it does not have an asterisk.

Re-enter SIGMA. You do not have to attach SIGMA again, just type

```
SIGMA
```

15.1.1 Communication with a running SIGMA program; (the terminal as file)

The terminal itself is regarded by SIGMA as a file named TERMNL, which is always known to SIGMA (no need for GET) and which cannot be released (no need for PUT). Hence the READ operator can be used to communicate with a running SIGMA program as for example in the following program:

```
SUBROUTINE CHAT
PRINT 'HOW OLD ARE YOU?'
A = READ(TERMNL, 1)
PRINT 'YOUR AGE IS', A, 'YEARS'
END
CALL CHAT
```

To enter a string with or without quotes around it, the second argument of READ should take the form m & n where m specifies how many records should be read (each time a RETURN key is pressed on the terminal, one record is read from the terminal) n specifies how many characters from each record should be read.

Hence to accept a "yes" or "no" response one could write the following program

```
SUBROUTINE,YESNO
111 PRINT 'IF YOU WOULD LIKE ME TO ASK YOU AGAIN, ENTER YES'
S = READ(TERMNL, 1 & 3)
IF (S EQ 'YES') GOTO 111
END

CALL YESNO
```

Respond with YES a couple of times then with NO or anything else.

Here is a more realistic example, using numerical as well as string input from the terminal:

```
SUBROUTINE FCTRL
PRINT ´TO OBTAIN N!, TYPE N´, ´N=´
N = READ(TERMINL, 1)
N = ARRAY(N, 1#N)
PRINT ´IF YOU WISH LOG(N!), TYPE LFCT´
PRINT ´IF YOU WISH N! ITSELF, TYPE FACT´
A = READ(TERMNL, 1 & 4)
!ERASE
IF A EQ ´FACT´ R = PROD(N)
IF A EQ ´LFCT´ GOTO 10
PRINT ´N EQUALS´, NCO(N), ´N! EQUALS´, R(NCO(N))
RETURN
10 R = TRACE(LOG(N))
PRINT ´N EQUALS´, NCO(N), ´LOG(N!) EQUALS´, R
END

CALL FCTRL                  (call it a few times and try the
                            options it offers)
```

15.2 Output of data to a file

Any SIGMA array may be written to a file where it will appear in a format identical to the format used by PRINT (i.e. character file with blanks separating the numbers). This means, in particular, that each line of SIGMA printout corresponds to one record on the file (in most cases four numbers). This must be kept in mind when re-reading from the file into SIGMA. The procedure to write onto a file again proceeds in three steps

i) make the file known to SIGMA, except that in this case the file should be a new file to be created, type

```
GET MYOUT*)
```

For creation of a permanent file, you would have to supply the proper identifier; see your local CUAC representative.

ii) Copy one or more arrays in index order onto the file. Define, for example,

```
A1 = ARRAY (5, 1#5)
```

*) As before, MYOUT stands symbolically for any name you choose

```
A2 = ARRAY (3, 9#11)
```

Write the arrays on the file MYOUT by typing

```
WRITE MYOUT, A1, A2
PRINT A1, A2
```

Look at the print-out: in exactly this same format A1 and A2 will now be in your file, each line of print will be in one file record (only the numbers are written, not the NCO, etc.)

iii) Release the file from SIGMA and make it into a local [permanent] file

```
PUT MYOUT [,YOURID]
```

The optional ID in PUT would ensure that the file is registered as a permanent file in case MYOUT is a new name created in (i) above. RELEASE or PUT without ID should not be used for permanent files, since the file will disappear after you log out from the terminal. To see that your file MYOUT has been created as a local [permanent] file

```
!STOP
YES
FILES
```

Remember the asterisks; there are no asterisks if MYOUT is local. If you wish you may read this file MYOUT into your workspace. Let us read from MYOUT. First enter SIGMA by typing SIGMA and then type

```
GET MYOUT
REWIND MYOUT
!ERASE
X = READ (MYOUT,3)
Y = READ (MYOUT,5)
REWIND MYOUT
Z = READ (MYOUT,100)
```

See what you got

```
PRINT X,Y,Z
```

Y contains only the 5th to 8th element and not the 4th to 8th element as one might have expected. This is because of the WRITE format being equal to the PRINT format: remember, we had written A1 = ARRAY(5, 1#5) and AZ = ARRAY (3, 9#11) onto the file with the PRINT format. The file records will therefore contain:

```
1st record:  1.0000      2.0000      3.0000      4.0000
2nd record:  5.0000
3rd record:  9.0000     10.0000     11.0000
```

READ (MYOUT,N) will read the next N elements from MYOUT, but always starting at the beginning of the next file record.

The easiest way not to worry about format is to use

```
REWIND MYOUT
DUMMY = READ (MYOUT, 1000)
```

to read the whole file (or, if it contains > 1000 numbers, its first 1000 numbers) into DUMMY and then use the by now familiar SIGMA array structuring techniques to construct your desired arrays from DUMMY. Try it!

In case you wish to read into SIGMA a file containing more than the maximum number of numbers allowed in a SIGMA array (!STATUS informs you about this), you must read your file into several arrays; in that case be careful to choose N in the READ commands such that each READ ends with the last number of a record; if you do not know the length of the records, you can easily find out by trial and error, varying N and printing a few of the first and last numbers of each array.

15.3 Output of a program to a file

Since WRITE used the same format and philosophy as PRINT and PRINT can print the text of a program as well as its result, one can also WRITE the text of any one or several programs onto a file using the procedure described in Section 15.2 (ii). The only difference is that (do not type yet)

```
WRITE MYPRO, PROG1, PROG2 ... PROGN
```

should be made to reference one or more programs. We try that now.

```
!LOAD your workspace.
```

To create a local file named MYPRO type

```
GET MYPRO
!NAMES
```

You may have programs named POLYNOM and COMPLOT on your workspace which you may want to write on MYPRO; type

```
WRITE MYPRO, POLYNOM, COMPLOT
PUT MYPRO
```

To make MYPRO permanent, type after !STOP and before logout:

```
CATALOG,MYPRO,ID=YOURID
```

15.4 Input of programs from files

A program written to a file by WRITE cannot be read back by READ, because the READ operation always generates an array as a result and a user program is certainly not an array. Try, for example, to assign any subroutine to an array and see what happens (print it). Hence another operator is needed. This operator, called SYSIN, informs SIGMA that the following file contains programs.

Suppose that you have saved several programs on the file named MYPRO and some time later you would like to read them again. Let us simulate "sometime later" by first saving the current state of our SIGMA session and then clearing the system completely:

```
!SAVE
!CLEAR
YES
```

Test that the system contains no names by

```
!NAMES
```

Now go through the familiar procedure 15.2 (i), (ii), and (iii) using SYSIN in step (ii) because the file MYOUT contains programs

```
  i) |    GET MYPRO
 ii) |    SYSIN MYPRO
```

and when control returns to you at the terminal

```
iii) |    PUT MYPRO
```

Now have a look at what names the system contains:

```
|  !NAMES
```

and PROG1 ... PROGN should be present and ready for action.

NOTE that SYSIN will return control to the terminal when it reaches the end of the specified file. SYSIN simply deceives SIGMA into believing that the specified file is the terminal with a very fast and accurate typist. Hence SYSIN may be used to read programs or command sequences from any character file whether made by a WRITE operation or not. For example, INTERCOM experts may prefer their own favourite editor to SIGMA´s rather limited edit facilities. SYSIN permits a file generated by any editor to be read into SIGMA as if it were retyped character by character on your terminal. Type

```
|  !STOP
```

!do not save, since you want to keep your old workspace, not the new one that only contains COMPLOT and POLYNOM!

```
|  YES
|  FILES
|  LOGOUT
```

LOGIN again for next Session.

15.5 General comments

i) All statements and procedures described above may be incorporated in any of your programs.

ii) There may be any number of READ or REWIND requests between GET and PUT of a file; if N of a read request exceeds the number of numbers remaining on the file, reading stops at the last number without giving an error message. As a rule you should always inspect the array resulting from a read command to make sure that it correponds to your intentions: each new READ starts at the beginning of the next record and ignores those numbers of the previous record which have not been read.

There may be any number of WRITE requests, too.

iii) A GET request may in some unfortunate cases cause errors which cannot be detected by SIGMA; since this will cause the loss of all current work it is always advisable to !SAVE before any dialogue with the file system is initiated.

iv) To discard any permanent file e.g. MYFILE, you type, when you are in INTERCOM (not in SIGMA)

```
    DISCARD MYFILE
or
    PURGE MYFILE, MYOUT, ABC, ...
```

END OF SESSION 15

SESSION 16: CONCLUDING SESSION

With the end of SESSION 15 we have also come to the end of this tutorial.

At this point you should review all defined objects:

```
        !LOAD ..., ...
        !NAMES
```

Type in and protect (or !COPY from SIGMA library workspace) the programs of Appendix 3 (as far as frequently needed) and delete all programs and variables which you no longer need (remember that you may delete all seldom needed programs available from the SIGMA library; see end of SESSION 10):

```
        DELETE PROGR1, PROGR2, ...
```

(if protected, DEPROTECT before DELETE)

```
        !DELETE (for variables)
```

Inspect once more by

```
        !NAMES
```

and, if you are satisfied with what remains,

```
        !SAVE ..., ...
        !STOP
```

Now you are familiar with the fundamental concepts of SIGMA and have a workspace with several useful programs. For further reference consult the CERN SIGMA USER'S MANUAL and/or go back to one or the other of the Sessions of this tutorial.

Acknowledgements

Many people have contributed to this tutorial by suggestions and in discussions and -- above all -- by trying out parts or the whole of it in practice. We wish to thank all of them.

We are particularly grateful to K. S. Koelbig and H. E. Rafelski for a thorough critical check of all Sessions resulting in many improvements. Session 15 has been rewritten by H. E. Rafelski. The typing was done by M. Guarisco using the AUTHOR facility; we thank her as well as H. von Eicken and H. E. Rafelski for the creation of the final layout and the production of this beautiful computer printed text.

The present SIGMA is the end product of more than ten years of evolution, during which ideas and hard work of many people were synthesized. It owes its first conception to C. J. Culler and B. D. Fried [in: ON LINE COMPUTING; ed. W. Karplus; Mc Graw Hill Inc., New York 1967]; main contributions are due to G. Barta, H. Jedlicka and L. van Hove but above all to C. E. Vandoni, who was the chief implementor and supervisor of the prodject through its whole history.

Further Reading

CERN SIGMA USER´S MANUAL (available at CERN from C. E. Vandoni, DD, ext. 3355).

R. Hagedorn, J. Reinfelds, C.Vandoni and L. Van Hove, SIGMA. To be published as CERN Yellow Report in 1978/79.

APPENDIX 1: AVAILABLE FUNCTIONS

In this Appendix we list all operators and functions of the CERN SIGMA version available in Spring 1978. At later times and at other computer centers this list might not correspond to reality. Please consult the manual valid then and there.

Array Functions:

FUNCTION	USE	SEMANTICS OF ARGUMENTS AND RESULT	NOTES
ANY	R=ANY(arg)	Components of arg must be Boolean. R is a Boolean scalar.	R is 1 if and only if at least one component of arg is true (=1).
DEL	R=DEL(arg)		A component of R is 1 if arg has a zero in the half interval on either side of the corresponding element of arg.
DET	R=DET(arg)	arg must be an array of square matrices. R is a scalar or vector of determinants.	R is the determinant of arg.
DIFF	R=DIFF(arg)		Rows of R contain forward differences of rows of arg. Last differences by quadratic extrapolation.
DROP	R=DROP(arg,arg1...argN)	arg1...argN must be scalars or vectors.	Drop reduces arrays by specifying sub-arrays which are to be eliminated.
EIGVAL	R=EIGVAL(arg)	arg must be an array of square matrices.	Rows of R contain the eigenvalues of the matrices of arg.
EIGVEC	R=EIGVEC(arg)	arg must be an array of of square matrices.	The eigenvectors of 1st matrix of arg are stored row-wise into the 1st matrix of R and so on.
EVAL	R=EVAL(arg1,arg2,arg3)	arg1 and arg2 must have identical NCO vectors. Row length of R is equal to row length of arg3.	EVAL regards arg1 as a function of arg2 and R is arg1 interpolated at points given by arg3.
HIST	R=HIST(arg1,arg2)		Data points defined by rows of arg1 are counted into bins defined by rows of arg2.
INV	R=INV(arg)	arg must be an array of square matrices.	Matrix inversion.
LS	R=LS(arg1,arg2)	arg2 must be a scalar	The rows of arg1 are shifted circularly by arg2 positions to the left.

Array Functions continued:

FUNCTION	USE	SEMANTICS OF ARGUMENTS AND RESULTS	NOTES
MAX	R=MAX(arg)		Replaces each element in each row by the largest value in that row
MIN	R=MIN(arg)		Replaces each element in each row by the smallest value in that row
MULT	R=MULT(arg1,arg2)	Arg1 and Arg2 must be two dimensional arrays compatible in the matrix sense	$R_{ij}=\sum_{k=1}^{n}(\text{arg1})_{ik}(\text{arg2})_{kj}$
NCO	R=NCO(arg)		Obtains the NCO vector of arg
ORDER	R=ORDER(arg1,arg2)	arg1 and arg2 must have compatible dimensions	Rows of arg1 are re-ordered such that the same permutation of subscripts would re-order rows of arg2 in non-descending order
PROD	R=PROD(arg)		Rows of R are the running products of arg
PROJ	R=PROJ(arg1,arg2)	arg2 is a vector or matrix	R is a vector of elements projected from arg1 as specified by arg2
QUAD	R=QUAD(arg1,arg2)	arg2 must be a scalar arg1 must have 5 or more elements in each row	Rows of arg1 are numerically integrated to produce rows of R
SMAX	R=SMAX(arg)		R is a scalar equal to the largest element of arg
SMIN	R=SMIN(arg)		R is a scalar equal to the smallest element of arg
SUM	R=SUM(arg)		Rows of R are running sums of rows of arg
TP	R=TP(arg1,arg2)	arg2 must be a vector	R is obtained by transposition of the indices of arg1 according to arg2
TRACE	R=TRACE(arg,arg1, ...argN)	arg1...argN must be scalars or vectors	R is obtained by contracting as specified by arg1...argN

Library Functions:

These are a subset of mathematical functions provided by the CERN PROGRAM LIBRARY and are available under SIGMA, where they act automatically componentwise if applied to arrays.

NAME	CERN Prog.Lib.	PURPOSE	NOTES
ACOS	(B100)	Arcosine	If $\lvert arg \rvert > 1.$, R=0
ALOGAM	(C341)	Logarithm of the Gamma Function	If arg $\leq$ 0., R=0
ASIN	(B100)	Arcsine	If $\lvert arg \rvert > 1.$, R=0
BESCJ	(C331)	Complex Bessel Functions	R=BESCJ(A,N,X) $\{J_{A+N}(X)\}$,0$\leq$A$\leq$1 x real or complex
BESI0	(C313)	Modified Bessel Functions	If arg > 741.66, R=0
BESI1	(C313)	I_0, I_1	If arg > 741.66, R=0
BESJ0	(C312)	Bessel Function J_0	If arg > 2.0E14, R=0
BESJ1	(C312)	Bessel Function J_1	If arg > 2.0E14, R=0
BESK0	(C313)	Modified Bessel Functions	If arg > 741.66, or arg $\leq$ 0, R=0
BESK1	(C313)	K_0, K_1	If arg > 741,66, or arg $\leq$ 0.1, R=0
BESY0	(C312)	Bessel Function Y_0	If arg > 2.0E14, or arg $\leq$ 0, R=0
BESY1	(C312)	Bessel Function Y_1	If arg > 2.0E14, or arg $\leq$ 0, R=0
COSH	(B200)	Hyperbolic Cosine	If $\lvert arg \rvert > 741.66$, R=0
COSINT	(C336)	Cosine Integral	If arg =0, R=0
DAWSON	(C339)	Dawson's integral	
DILOG	(C304)	Dilogarithm Function	
EBESI0	(C313)	$e^{-\lvert x \rvert}\ I_0$	
EBESI1	(C313)	$e^{-\lvert x \rvert}\ I_1$	
EBESK0	(C313)	$e^x K_0$	If x$\leq$0., R=0
EBESK1	(C313)	$e^x K_1$	If x$\leq$0., R=0
ELLICK	(C308)	Complete Elliptic	
ELLICE	(C308)	Integral K and E	
ERF	(C300)	Error Function and	
ERFC	(C300)	Normal	
EXPINT	(C337)	Exponential Integral	
FREQ	(C300)	Frequency Function	
GAMMA	(C305)	Gamma Function	If -163.5 > arg > 176.5, R=0
RNDM	(V104)	Random Number Generator	Components of RNDM(X) are random numbers between 0 and 1 with NCO(RNDM(X))■NCO(X)
SINH	(B200)	Hyperbolic Sine	If $\lvert arg \rvert > 741.66$, R=0
SININT	(C336)	Sine Integral	
TAN	(B100)	Tangent	If $\lvert arg \rvert > 8.4E14$, R=0

Systems Functions:

FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	USE	NOTES
Absolute value	$\|arg\|$	1	ABS(arg)	for complex arg $\sqrt{(Re(arg))^2 + (Im(arg))^2}$

Arctangent	arctang(arg)
	arctan(arg_1/arg_2)
		1
		2
			ATAN(arg)
			ATAN2(arg_1,arg_2)
				ATAN(X)=θ, where tan θ=X $-\pi/2<\theta<\pi/2$
				ATAN2(Y,X)=arctan($^Y/_X$)=θ $-\pi\leq\theta<\pi$

Take complex conjugate	Re(arg)-Im(arg)*i	1	CONJ(arg)	
Trigonometric Cosine	cos(arg)	1	COS(arg)	Accuracy diminishes for large arguments. Returns zero if $|arg|\geq$2**47
Combine two real arrays into a complex array	arg_1+arg_2*i	2	CPLX(arg_1,arg_2)	The real parts are taken for complex arg_1 or arg_2
Exponential	e^{arg}	1	EXP(arg)	Valid range for arg is: $-675.8 \leq arg \leq 741.6$ else returns zero
Take imaginary part	Im(arg)	1	IMAG(arg)	The imaginary part of a real array is zero
Truncation	Sign of arg times largest integer<$\|arg\|$	1	INT(arg)	If $\|arg\|\geq 2^{48}$ result is unpredictable
Natural Logarithm	$\log_e$(arg)	1	LOG(arg)	
Common Logarithm	$\log_{10}$(arg)	1	LOG10(arg)	
Remaindering	arg_1(mod arg_2)	2	MOD(arg_1,arg_2)	The function MOD(arg_1,arg_2) is defined as $arg_1-[arg_1/arg_2]\ arg_2$ where [x] is the truncation function of x, namely INT(x)
Convert string array into a numerical array	values of array elements unchanged	1	NUMBER(arg)	
Take real part	Re(arg)	1	REAL(arg)	
Transfer of sign	$\|arg_1\|$ times sign of arg_2	2	SIGN(arg_1,arg_2)	
Trigonometric Sine	sin (arg)	1	SIN(arg)	Accuracy diminishes for large arguments. Returns zero if $|arg|\geq$2**47
Square root	$arg^{\frac{1}{2}}$	1	SQRT(arg)	
Convert numerical array into a string array	arg modulo 64	1	STRING(arg)	
Hyperbolic tangent	tanh (arg)	1	TANH(arg)	

APPENDIX 2: THE SIGMA LIBRARY WORKSPACE

As SIGMA offers facilities not common with usual computing (batch) situations, it cannot use without special interface the programs collected in your batch services library. Those programs, which have been made available under SIGMA, are found in Appendix 1. On the other hand, a number of SIGMA programs, exploiting typical SIGMA techniques, have been written by users and might be of interest to other users. Some are collected at CERN in a particular SIGMA workspace with the name LIBRARY. Using the !COPY command, you can copy any of these programs into your workspace(s). The procedure is as follows: type

```
!COPY LIBRARY, CONTENT
!ERASE
CALL CONTENT
```

The subroutine CONTENT will inform you about the content of the library and of what you must do in order to copy any of the programs available. You should follow this prescription and not try to copy immediately a library program; the given prescription ensures that you do not only obtain the wanted program, but with it also all those programs which it calls as subprograms.

The whole sequence of commands used for copying a library program to your workspace is then:

```
!LOAD NAME, YOURID        (symbolic names, use your own!)
!COPY LIBRARY, CONTENT
!ERASE
CALL CONTENT
```

From the printed list you select, for example, IMS3 (invariant momentum space for three particles):

```
!COPY LIBRARY, LIMS3
CALL LIMS3
DELETE CONTENT, LIMS3
!SAVE NAME, YOURID
```

From now on IMS3 will be part of your own workspace. It is recommended to leave library programs in your own workspace only as long as you need them frequently; the shorter your workspace, the faster is loading and saving and, therefore, it is less time-consuming to copy a seldom used library program each time it is needed and delete it afterwards, than to keep it forever in your workspace.

APPENDIX 3: SOME USEFUL SIGMA PROGRAMS

In this Appendix we collect a few useful programs written in SIGMA language:

1. Stereoscopic three-dimensional view
2. Complex mapping (from Sessions 13/14)
3. Polynomial (from Session 6)
4. Simpson definite integral
5. Polynomial least squares fit. (Needs polynomial)
6. Legendre polynomial approximation.
7. Fourier series approximation.

At CERN, all these programs are contained in the SIGMA Library workspace from where you may copy them using the description given in Appendix 2. It is a good exercise in SIGMA programming to analyse these programs, to understand exactly how they work and possibly to improve them. In any case try them out on a few examples.

For those SIGMA users who do not have access to the SIGMA Library at CERN, we give below the full text of the above seven programs so that you can type them in, test, correct, and !SAVE them.

1. STEREO (X,Y,Z, THETA, PHI, NUMBER)

This produces one (NUMBER=1) or two (NUMBER=2) pictures of the surface or trajectory Z(X,Y). X and Y may be vector arrays, transposed or not; the total number of components of Z must be $\leq$ 1280 (CERN 1977 version; ask !STATUS for up to date information). THETA and PHI are the view directions in degrees for MONO view (NUMBER=1). In STEREO view (NUMBER=2) we look at the left (right) figure with

$$\mathrm{PHI}_{\mathrm{left}} = \mathrm{PHI} - 4^{\circ} * \sin(\mathrm{THETA})$$

$$\mathrm{PHI}_{\mathrm{right}} = \mathrm{PHI} + 4^{\circ} * \sin(\mathrm{THETA})$$

which for $20^{\circ} < \mathrm{THETA} < 40^{\circ}$ corresponds to seeing the object from about 60 cm away with a distance between the eyes of about 6 cm; for THETA outside this range the stereoscopic effect is there but for THETA = 0 it vanishes, while for THETA = 90° it is somewhat exaggerated. Had we tried to keep the effect the same for all THETA, the z-axis could not be kept always vertical.

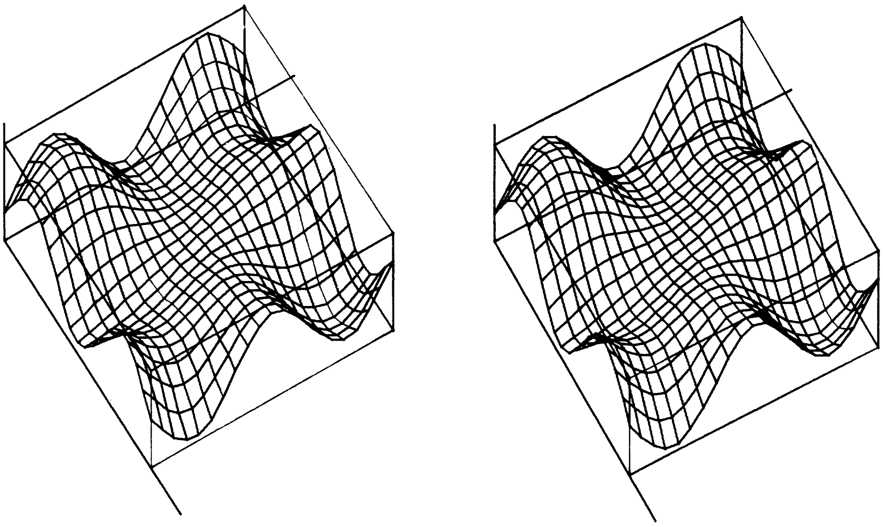
The user who cannot (even not with a thin piece of cardboard as separation wall) manage to merge the two slightly different figures into a single one (it may take a few minutes of patience the first time), puts simply NUMBER=1 in STEREO call.

The figures are not faithful in scale: all variables are scaled to lie between 0 and 1, so that the picture shows everything inscribed in a unit cube {X,Y,Z}.

```
SUBROUTINE STEREO(X,Y,Z,THETA,PHI,NUMBER)
$
$   ==============================================================
$
$   FOR NUMBER=2 PRODUCES TWO FIGURES FOR STEREO VIEW
$   NUMBER=1 GIVES SINGLE FIGURE
$   BEST RESULTS WITH THETA BETWEEN 20 AND 40
$   THETA IS POLAR,PHI AZIMUTHAL ANGLE OF VIEW (DEGREE)
$   WORKS FOR CURVE AND SURFACE IN SPACE
$
$   ===============================================================
$
N=SMAX(PROD(NCO(Z)))
NX=SMAX(PROD(NCO(X)))
NY=SMAX(PROD(NCO(Y)))
TRAJ=(NX EQ NY) AND (NX EQ N)
XMA=SMAX(X)
YMA=SMAX(Y)
ZMA=SMAX(Z)
XMI=SMIN(X)
YMI=SMIN(Y)
ZMI=SMIN(Z)
E=1.2
!ERASE
XN=(X-XMI)/(XMA-XMI)
YN=(Y-YMI)/(YMA-YMI)
ZN=(Z-ZMI)/(ZMA-ZMI)
!NOAX
!NOFRAME
!NOERA
RAD=2*PI/360
T=THETA*RAD
P=PHI*RAD
CT=COS(T)
ST=SIN(T)
A=8*ST*RAD
AX=0&0&0&0&0&0&0&1&1&1&1&1
AX=TP(AX)&TP(0&0&0&0&E&1&1&0&1&1&1&1)
AY=0&0&0&1&0&0&1&1&0&0&1&1
AY=TP(AY)&TP(0&E&1&1&0&0&1&1&0&1&0&1)
AZ=0&0&1&0&0&1&0&1&0&0&1&1
AZ=TP(AZ)&TP(E&0&1&1&0&1&0&1&1&0&1&0)
IF NUMBER EQ 1 GOTO 100
I=0
CP=COS(P-A/2)&COS(P+A/2)
SP=SIN(P-A/2)&SIN(P+A/2)
W=ARRAY(2&4,280&780&112&612&280&780&411&911)
201 I=I+1
```

```
3 0 AY1=-AX*SP(I)+AY*CP(I)
AZ1=-CT*(AX*CP(I)+AY*SP(I))+AZ*ST
YY=ARRAY(SMAX(NCO(Y))&1,YN)
IF TRAJ YY=TP(YY)
Y1=YY*CP(I)-XN*SP(I)
Z1=-CT*(YY*SP(I)+XN*CP(I))+ZN*ST
IF NUMBER EQ 1 GOTO 101
202 !SETW W(I,)
DISPLAY(=) AZ1:AY1
!NOSCALE
DISPLAY Z1:Y1
IF NOT(TRAJ) DISPLAY TP(Z1):TP(Y1)
IF I EQ 1 GOTO 201
DO 203 K=1,17
203 PRINT
ANGLE=A/RAD
PRINT,THETA,PHI,
PRINT´ , ,
PRINT´STEREOSCOPIC VIEW UNDER ANGLES THETA,PHI.´
PRINT´ ´, ´LOOK WITH LEFT EYE ON LEFT,RIGHT´EYE ON,RIGHT FIGURE´
PRINT´YOU MAY PUT SEPARATION WALL BETWEEN FIGURES´ ,
PRINT´PERPENDICULAR TO FIGURES AND TOUCHING YOUR NOSE´
PRINT ´VIEWING DISTANCE ABOUT 50 CENTIMETRES´ ,
PRINT´FIGURES MERGE INTO SINGLE ONE SEEN IN SPACE´ ,
PRINT´ ´, ´BEST RESULTS WITH THETA BETWEEN 20 AND 40´
301 !NORMAL
RETURN
100 I=1
CP=COS(P)
SP=SIN(P)
GOTO 300
101 DISPLAY(=) AZ1:AY1
!NOSCALE
DISPLAY Z1:Y1
IF NOT(TRAJ) DISPLAY,TP(Z1):TP(Y1)
PRINT,THETA,PHI,  ´,    ,
PRINT´THIS IS MONO´´
PRINT´FOR STEREO CALL WITH NUMBER=2´
GOTO 301
END
```

Example: $z = x \cos y + y \cos x$; $-5 \leq x,y \leq 5$



```
THETA= 20.000000
PHI= -30.000000


STEREOSCOPIC VIEW UNDER ANGLES THETA,PHI.

LOOK WITH LEFT EYE ON LEFT,RIGHT EYE ON RIGHT FIGURE
YOU MAY PUT SEPARATION WALL BETWEEN FIGURES
PERPENDICULAR TO FIGURES AND TOUCHING YOUR NOSE
VIEWING DISTANCE ABOUT 50 CENTIMETERS
FIGURES MERGE INTO SINGLE ONE SEEN IN SPACE

BEST RESULTS WITH THETA BETWEEN 20 AND 60
```

2. COMPLEX MAPPING

Known from Sessions 13 and 14

```
SUBROUTINE COMPLOT(W,Z)
$
$   ============================================
$
$   PLOTS CONTOURS OF W (FULL) AND Z (BROKEN)
$   IN COMPLEX PLANE
$
$   ===========================================
$
DISPLAY(=)IMAG(W):REAL(W),[.-2]IMAG(Z):REAL(Z)
END
```

```
SUBROUTINE COMGRID(NV,NH,SIZE,CENTER)
$
$   ==============================================
$
$   DRAWS NV VERTICAL,NH HORIZONTAL LINES IN SQUARE
$   OF SIDELENGTH SIZE, ENTERED AT CENTER
$   AVAILABLE AS ZV,ZH
$
$   ==============================================
$
GLOBAL ZV,ZH
XV=ARRAY(NV,-SIZE/2#SIZE/2)
YH=ARRAY(NH,-SIZE/2#SIZE/2)
X=ARRAY(51,-SIZE/2#SIZE/2)
Y=X
ZV=CPLX(TP(XV),Y)+CENTER
ZH=CPLX(X,TP(YH))+CENTER
CALL COMPLOT(ZV,ZH)
END
```

```
SUBROUTINE COMNET(NR,NC,RMIN,RMAX,CENTER)
$
$   =============================================
$
$   DRAWS NR RADII,NC CIRCLES WITH RMIN < R < RMAX
$   CENTERED AT CENTER,AVAILABLE AS ZR AND ZC
$
$   =============================================
$
GLOBAL ZR,ZC
RR=ARRAY(51,RMIN#RMAX)
RC=ARRAY(NC,RMIN#RMAX)
P=ARRAY(51,-PI#PI*.9999)
PR=ARRAY(NR+1,-PI#PI*.9999)
ZR=CPLX(RR*COS(TP(PR)),RR*SIN(TP(PR)))+CENTER
ZC=CPLX(TP(RC)*COS(P),TP(RC)*SIN(P))+CENTER
CALL COMPLOT(ZR,ZC)
END
```

```
SUBROUTINE COMSECT(NR,NC,RMIN,RMAX,AMIN,AMAX,CENTER)
$
$  ====================================================
$
$  DRAWS NR RADII AND NC CIRCLES WITH RMIN < R < RMAX
$  AMIN < ANGLE(RADIAN) < AMAX
$  ENTERED AT CENTER. AVAILABLE AS ZR AND ZC
$
$  ====================================================
$
GLOBAL ZR,ZC
RR=ARRAY(51,RMIN#RMAX)
RC=ARRAY(NC,RMIN#RMAX)
P=ARRAY(51,AMIN#AMAX)
PR=ARRAY(NR,AMIN#AMAX)
ZR=CPLX(RR*COS(TP(PR)),RR*SIN(TP(PR)))+CENTER
ZC=CPLX(TP(RC)*COS(P),TP(RC)*SIN(P))+CENTER
CALL COMPLOT(ZR,ZC)
END
```

3. POLYNOMIAL

Known from Session 6

```
FUNCTION POLYNOM(X,A)
$
$  ================================================
$
$  COMPUTES A(1)+A(2)*X+A(3)*X**2+....A(N)*X**(N-1)
$
$  ================================================
$
N=NCO(A)
P=0
DO 1 I=0,N-2
1 P=(P+A(N-I))*X
POLYNOM=P+A(1)
END
```

4. SIMPSON DEFINITE INTEGRAL

The definite integral

$$\int_{x_0}^{x_1} F(x)dx$$

can be obtained by means of QUAD, the integration operator; one simply picks up the last component of the result. Applied to multidimensional arrays, QUAD(Y,DX) assumes that each row of the array Y is to be integrated with the <u>same</u> step size DX; furthermore, QUAD needs ≥ 5 elements per row in Y.

SIMPSON has been written to allow more generality: each row may have another step length which may either be given (third argument DX; then set X = 0) or computed by SIMPSON from X(second argument; then set DX = 0); furthermore SIMPSON requires only ≥ 3 points. Apart from the restrictions ≥ 3 points and step length constant within a row, SIMPSON is very general; the number of points may be even or odd. Note that the result of integrating a rectangular array (N by K) is an N by 1 <u>column</u> vector; more generally, if

$$\mathrm{NCO(FX)} = N_1 \;\&\; N_2 \;\&\; \ldots \;\&\; N_{l-1} \;\&\; N_l$$

then

$$\mathrm{NCO(SIMPSON(FX,X,0))} = N_1 \;\&\; N_2 \;\&\; \ldots \;\&\; N_{l-1} \;\&\; 1$$

```
FUNCTION SIMPSON(FX,X,DX)
$
$   ==============================================
$
$   SIMPSON INTEGRAL( FX*DX ) FOR ARBITRARY ARRAYS
$
$   IF FX AND X ARE GIVEN,SET DX=0
$   IF FX AND DX ARE GIVEN,SET X=0
$   FX AND X MUST MATCH TOPOLOGICALLY
$   IF SO,SIMPSON WORKS ROW BY ROW INDEPENDENTLY
$   IF X IS GIVEN,EQUIDISTANCE IS CHECKED
$   IF DX IS GIVEN, EQUIDISTANCE IS ASSUMED
$
$   NUMBER OF SUPPORT POINTS MAY BE EVEN
$
$   ================================================
$
Y=FX
IF(DX NE 0) GOTO 51
IF(ABS(DIFF(DIFF(X))/DIFF(X)) GE 1E-10) GOTO 101
NX=NCO(X)
NX1=NX(NCO(NX))
NY=NCO(Y)
NY1=NY(NCO(NY))
IF(NX1 NE NY1) GOTO 100
NX(NCO(NCO(X)))=1
```

```
H=ARRAY(NX,X(NX1)-X(NX1-1))
GOTO 52
51 H=DX
NY=NCO(Y)
52 N=NY(NCO(NY))
N1=NY
N1(NCO(NY))=1
$  ======================================
$  SPLIT INTO EVEN(K=2) AND ODD(K=3) CASE
$  ======================================
K=MOD(N,2)+2
IF(K EQ 3) GOTO 10
GOTO 20
3 SIMPSON=S
RETURN
$  =======================================
$  SUPPRESS LAST POINT TO ACHIEVE ODD NUMBER
$  =======================================
20 YLAST=ARRAY(N1,Y(N))
N=N-1
NN=ARRAY(N,1#N)
N2=NY
N2(NCO(NY))=N
Y=ARRAY(N2,Y(NN))
GOTO 10
2 N=N+1
Y=Y&YLAST
S=S+H/12*ARRAY(N1,5*Y(N)+8*Y(N-1)-Y(N-2))
SIMPSON=S
RETURN
$  ===========================
$  HERE IS THE SIMPSON PROCEDURE
$  ===========================
10 MASK=(MOD(ARRAY(N,1#N),2) EQ 0)
S=ARRAY(N1,TRACE(Y*(2*MASK+(NOT MASK)),NCO(NY)))
S=H/3*(2*S-ARRAY(N1,Y(1)+Y(N)))
GOTO K
$  ========
$  MESSAGES
$  ========
100 PRINT´NOT DONE ; NO MATCH IN X AND Y´
RETURN
101 PRINT´NOT DONE ; SUPPORT POINTS NOT EQUIDISTANT´
END
```

A step length varying from row to row is useful in double integration with variable inner limits; an example is the moment of inertia of a unit sphere with unit density.

$$\Theta = 2\pi \int_{-1}^{1} dz \int_{0}^{\sqrt{1-z}} r^3 \, dr = 8\pi/15 .$$

Numerically

```
Z = ARRAY(21,-1#1)
R = ARRAY(21,0#1)*TP(SQRT(1-Z**2))
```

R is a 21-row array, each row having a different range and step length

```
      THETA = 2*PI*SIMPSON(TP(SIMPSON(R**3,R,0)),Z,0)
      PRINT THETA, 8*PI/15
```

5. POLYNOMIAL LEAST SQUARES FIT

Given MX = MY pairs of values $\{x_i, y_i\}$ ("data") (neither of the sets $\{x_i\}$, $\{y_i\}$ need be equidistant and/or ordered); then FIT computes the least squares fit by a polynomial of order ORDER; WEIGHT must be either a scalar (put it = 1) or have NCO(WEIGHT) = MX. For instance, WEIGHT might be the inverse of the error y_i attached to each "measured value" y_i:

WEIGHT = 1/ABS(DY), where NCO(DY) = NCO(Y)

or any function of DY or of Y or of X.

The program makes available and prints out the coefficients a_i of

$$\text{fit} = a_o + a_1x + \dots + a_nx^n \quad (n = \text{ORDER})$$

and makes available the fit-function pair YFIT, XFIT. Note that the program works rather slowly if NCO(X)*(ORDER+1)**2 GE 1280.

```
SUBROUTINE FIT(X,Y,WEIGHT,ORDER)
$
$   ================================================
$
$   COMPUTES LEAST SQUARE FIT OF GIVEN ORDER N
$   A0 + A1*X + A2*X**2 + ....+ AN*X**N
$   WEIGHT=1 OR NCO(WEIGHT=NCO(X)
$   COEFFICIENTS UNDER COEFF ; AK=COEFF(K+1)
$   101-POINT FIT FUNCTION AVAILABLE UNDER XFIT,YFIT
$   RELATIVE ERROR AVAILABLE UNDER RELERR
$
$   ==================================================
$
GLOBAL COEFF,XFIT,YFIT,RELERR
N=ORDER+1
IF N EQ 1 GO TO 5
L=ARRAY(N,1#N)
K=ARRAY(N&1,L)
MX=NCO(X)
MY=NCO(Y)
W=NCO(WEIGHT)
IF(W EQ 1) WEIGHT=1
IF((MX EQ MY)AND((MX EQ W)OR(W EQ 1))) GO TO 1
PRINT 'X,Y,WEIGHT  DO NOT MATCH ; STOP'
RETURN
1 MKL=K+L-2
IF(MX*N*N GE 1280) GO TO 2
XX=ARRAY(MX&1&1,X)
WW=ARRAY(MX&1&1,WEIGHT)
C=TRACE(XX**MKL*WW,1)
GO TO 4
2 WW=0*X+WEIGHT
C=0
```

```
DO 3 I=1,MX
3 C=C+X(I)**MKL*WW(I)
4 CONTINUE
D=TRACE(Y*WEIGHT*X**(K-1),2)
D=TP(D,2&1)
COEFF=MULT(INV(C),D)
COEFF=TP(COEFF,2&1)
XFIT=ARRAY(101,SMIN(X)#SMAX(X))
YFIT=POLYNOM(XFIT,COEFF)
DISPLAY [*]Y:X,YFIT:XFIT
PRINT´ORDER EQUAL´,N-1
Y1=POLYNOM(X,COEFF)
RELERR=(Y1-Y)/ABS(Y)
PRINT´ MEAN REL ERROR =´
PRINT TRACE(ABS(Y1-Y))/TRACE(ABS(Y))
GOTO 50
5 IF NCO(WEIGHT) EQ  1  WEIGHT=X/X
COEFF=TRACE(Y*WEIGHT,1)/TRACE(WEIGHT,1)
XFIT=ARRAY(101,SMIN(X)#SMAX(X))
YFIT=0*XFIT+COEFF
DISPLAY[.]Y:X,YFIT:XFIT
PRINT´ORDER EQUAL´,N-1
Y1=X*0+COEFF
RELERR=(Y1-Y)/ABS(Y)
PRINT ´MEAN REL ERR=´
PRINT TRACE(ABS(Y1-Y))/TRACE(ABS(Y))
50 PRINT´ ´,´ ´,´ ´,´COEFFICIENTS =´,´AO ON TOP´,´ ´
PRINT TP(COEFF)
END
```

6. LEGENDRE POLYNOMIAL APPROXIMATION

A function f(x) known for equidistant x is approximated by Legendre polynomials after being transformed to the interval -1 to +1.

```
SUBROUTINE FITLEG(F,X)
$
$   ==========================================================
$
$   FITLEG FIRST TRANSFORMS TO INTERVAL  -1 TO +1 AND THEN
$   MAKES LEGENDRE POLYNOMIAL FIT TO
$   F(X)=G(XX) = SUM(A(L)*(L+1/2)*P(L,XX))
$   WHERE XX=(X-(XMAX+XMIN)/2)*2/(XMAX-XMIN)
$   RESULT IN LGAPPR ; COEFFICIENTS A(L) IN COEFFL(L+1)
$
$   RESULT DISPLAYED IN FULL , ORIGINAL F AS ****
$
$   ==========================================================
$
N=0
IF NCO(NCO(F)) NE 1 GOTO 10
A=   ´TYPE IN THE ORDER DESIRED - 50 FOR INTERRUPTIBLE´,
A=A&´SEQUENCE OF APPROXIMATIONS, OR THE ORDER N WHERE´
A=ARRAY(3&48,A&´N IS GREATER THAN 1 - ´&ARRAY(30)*45)
N=INPUT(A,1)
GLOBAL COEFFL,LGAPPR
X1=SMAX(X)
X0=SMIN(X)
XX=(X-(X1+X0)/2)*2/(X1-X0)
COEFFL=ARRAY(N+1)
L0=ARRAY(NCO(X))
L1=XX
COEFFL(1)=SIMPSON(F,XX,0)
COEFFL(2)=SIMPSON(F*XX,XX,0)
LGAPPR=COEFFL(1)/2+3/2*XX*COEFFL(2)
DO 1 K=3,N+1
L=(L1*XX*(2*K-3)-L0*(K-2))/(K-1)
COEFFL(K)=SIMPSON(L*F,XX,0)
LGAPPR=LGAPPR+(K-.5)*COEFFL(K)*L
L0=L1
L1=L
IF(N NE 50) GO TO 3
!SMALL
DISPLAY LGAPPR:XX,[*]F
PRINT´  HIGHEST P(K,X) HAS K=´,K-1
PRINT´  MEAN REL.ERROR =
PRINT TRACE(ABS(LGAPPR-F))/TRACE(ABS(F))
CCC=INPUT(´TYPE C FOR CONTINUE, S FOR STOP -´,1&1) EQ ´Y´
!LARGE
IF(CCC EQ 0) RETURN
3 CONTINUE
1 CONTINUE
!SMALL
DISPLAY LGAPPR:XX,[*]F
!LARGE
PRINT´   HIGHEST P(K,X) HAS K=´,N
PRINT´  MEAN REL ERROR =
PRINT TRACE(ABS(LGAPPR-F))/TRACE(ABS(F))
RETURN
10 PRINT´NOT EXECUTED,WRONG DIMENSION OF ARGUMENT(S)´
END
```

7. FOURIER SERIES APPROXIMATION

If f(x) is given over an interval with equidistant spacing, FOURIER computes successive Fourier approximations to f(x) up to an order N = NCO(X)/4.

```
SUBROUTINE FOURIER(F,X)
$
$   ====================================================
$
$   COMPUTES SUCCESSIVE FOURIER APPR. TO F(X)
$   F IS NAME OF VECTOR ARRAY
$   RESULT AVAILABLE UNDER FOURAPP
$   COEFFICIENTS AVAILABLE UNDER MEANVAL,FOURCA,FOURSB
$
$   ====================================================
$
GLOBAL FOURAPP,MEANVAL,FOURCA,FOURSB
K=NCO(X)
DX=X(2)-X(1)
FOURCA=ARRAY(K/4)*0
FOURSB=FOURCA
L=(SMAX(X)-SMIN(X))/2
FOURAPP=QUAD(F,DX)/2/L
FOURAPP=FOURAPP(K)
MEANVAL=FOURAPP
DO 1 N=1,NCO(FOURCA)
T=N*PI*X/L
C=COS(T)
S=SIN(T)
A=QUAD(C*F,DX)/L
B=QUAD(S*F,DX)/L
FOURCA(N)=A(K)
FOURSB(N)=B(K)
FOURAPP=FOURAPP+C*FOURCA(N)+S*FOURSB(N)
DISPLAY [-.3]F:X,FOURAPP
PRINT N
!WAIT
1 CONTINUE
PRINT 'ORDER OF APPR. EQUAL TO 1/4 OF NCO; END REACHED'
END
```

APPENDIX 4: ORGANIZATION OF USER´S PRIVATE DATA AND PROGRAM LIBRARY

After some time of heavy SIGMA use the user´s workspace becomes long, mainly owing to programs which he might wish to conserve for a long time. In that case saving takes times which go easily into the order of several minutes. As frequent saving is an effective means against heavy losses of work, this situation is inconvenient. In that case it is useful to have several workspaces (with the same identifier), for example

```
JACK, EPLAB
JACKMAT, EPLAB
JACKDAT, EPLAB
JACKPHY, EPLAB
JACKDIV, EPLAB
```

where JACK serves as the main workspace in which actual computing is done and which is kept as short as possible; all permanent mathematical functions you have constructed, you put in JACKMAT, all valuable data, tables, etc., in JACKDAT, programs with physics application in JACKPHY, and all other useful programs in JACKDIV. Suppose you wish to add to JACKDIV the polynomial least squares fit subroutine FIT. Then there are two possibilities:

If you have access to the CERN SIGMA LIBRARY	If there is no SIGMA LIBRARY containing FIT
!LOAD JACKDIV, EPLAB	!LOAD JACKDIV, EPLAB
!COPY LIBRARY, CONTENT	Type in the program FIT without trying to correct your typing errors now; it is safer first to
!ERASE	!SAVE JACKDIV, EPLAB

```
CALL CONTENT
```

```
!COPY LIBRARY, LFIT
```

```
CALL LFIT
```

```
PROTECT FIT, POLYNOM
```

```
DELETE LFIT, CONTENT
```

```
!SAVE JACKDIV, EPLAB
```

As FIT needs POLYNOM as a subprogram, LFIT has copied both of them and both are now in JACKDIV

Now you !EDIT FIT to correct typing errors (if necessary) and test FIT in a realistic case. You may be informed that POLYNOM is unknown. In that case you type in the program POLYNOM and !SAVE again. Then !EDIT POLYNOM (if there are mistakes) and try again FIT on a few cases. If everything works,

```
!NAMES
```

delete all superfluous objects (there are arrays left from testing FIT)

```
!PROTECT FIT, POLYNOM
```

```
!SAVE JACKDIV, EPLAB
```