stichting
mathematisch
centrum

MC

AFDELING NUMERIEKE WISKUNDE NW 66/79 MEI
(DEPARTMENT OF NUMERICAL MATHEMATICS)

P.W. HEMKER & D.T. WINTER

A PRELIMINARY REPORT ON NUMERICAL OPERATORS IN ALGOL 68

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 65-00, 65-04, 68B99

A preliminary report on numerical operators in ALGOL 68
(A skeleton for a modern numerical software library)

by

P.W. Hemker & D.T. Winter

ABSTRACT

In this report a proposal for a set of numerical operators in ALGOL 68 is described. These operators enable ALGOL 68 users to program standard numerical computations in an easy but still flexible way (i.e. in a way resembling the usual mathematical notation).

The system of operators serves the same purpose as a numerical library in e.g. FORTRAN or ALGOL 60. Computations in numerical algebra, elementary numerical analysis (quadrature, root-finding) and initial value problems for ODEs are considered. In addition a flexible error message system is provided.

KEY WORDS & PHRASES: *Numerical software library, ALGOL 68 operators*

Introduction for the non-ALGOL 68 user.

Programming of numerical problems often is terribly opaque, even when the original problem is clear and can be formulated in a few lines. Why should one be worried with things as LOOPS when computing

$$\sum_{i=1}^{n} a_i \quad \text{or} \quad \sum_{n=10}^{1000} 1/n^2 \quad ?$$

Why should one be teased with WORKING SPACE and ERROR FLAGS if one wants the eigenvalues of a real square matrix? Why to compute

$$c = \int_0^1 e^x dx$$

by means of

```
EXTERNAL F
A = 0.0
B = 1.0
RERR = 0.0
AERR = 1.0E-5
C = ROUT(F,A,B, AERR, RERR, ERROR, IER)
.
.
.
END
FUNCTION F(x)
F = EXP(x)
RETURN
END
```

if it can be done by

```
c = range (0,1) integral ((real x) real: exp(x))
                abserr 1.0E-5;    ?
```

Why not should the programmer of a library take the burden of irrelevant details from the shoulders of a user?
With the aid of the language ALGOL 68 it is possible to construct a system of modes (e.g. range and real) and operators (integral and abserr in the above example) which enable the user to program his numerical problems in an easy and mathematical looking way.

ents

<u>Notation</u>

The generic calling sequence

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \quad \begin{matrix} [e] & \\ [f] & [i] \\ [g] & [j] \\ [h] & [k] \end{matrix} \quad \left\{\begin{array}{c} y \\ \hline z \end{array}\right\},$$

where letters denote a part of a calling sequence (e.g. an operator-operand-pair), denotes any calling sequence which consist of

- a choice of <u>at</u> <u>most</u> <u>one</u> from a, b, c and d
- a choice from e,f,...,k; any number of items and in any order
- <u>either</u> y <u>or</u> z, (exclusive or).

E.g. calling sequences generated by the above generic calling sequence are:

1) b f y
2) z
3) h k g i e z
4) c e f k j y

CHAPTER 1

INTRODUCTION

In this report we describe a proposal for a set of numerical operators. These numerical operators are ALGOL 68 operators that enable a programmer to perform (standard) numerical computations in an easy but still flexible way.

Usually such computations are done by routines, of which the calling sequence is more or less complex, depending on the kind of problem and the particular routine used. Often this complexity of a calling sequence is a nuisance for the programmer because it forces him to plough through a substantial piece of documentation and it confronts him with programming details, many of which are of no relevance to his original problem.

Fortunately, the structure of ALGOL 68 is such that the length of calling sequences can be reduced considerably as compared with the practice in some other languages (e.g. the dimension of an array does not need to appear in the calling sequence as a separate parameter) and the complexity of the calling can be decreased (or at least be made more surveyable) by structuring the data into a smaller number of new objects.

This idea of simplifying the calling sequences could be cultivated and it would lead to a numerical software library which would consist of a set of procedures and a set of data structures. However, pursuing the line of thought which led to smaller parameter lists, we come to the idea of using operators instead of procedures since an operator is a routine with at most two parameters. The use of operators, has the additional advantage that the same name (ALGOL 68 TAG) can be used with different meanings when the argument(s) is (are) of different modes. This last feature of ALGOL 68 yields the flexibility of taking together different routines which have essentially the same mathematical meaning. This makes the programming of a numerical problem very close to its mathematical description.

EXAMPLES. The ALGOL 68 clauses

(1.1) real r = interval (0,pi) integral ((real x) real: sin(x));

(1.2) mat a = ((int i,j) real: 1/(i+j-1)) into square (n);

(1.3) vec b = a[1,];

(1.4) real s = determ a;

(1.5) vec x = a sol b;

are readily recognizable as programmed equivalents of the mathematical formulas

(1.6) $r = \int_0^\pi \sin(x)\ dx$

(1.7) $A = (a_{ij}) = (1/i+j-1), \quad i,j = 1,\dots,n;$

(1.8) $b = (b_i) = (a_{1i}), \quad i = 1,\dots,n;$

(1.9) $s = \det(A)$

(1.10) $x = A^{-1}b \quad (\text{or } \{x \mid Ax = b\}).$

It will be clear that such a description which is close to the mathematical notation is easy to handle but often needs to be extended in order to give more details about requirements imposed on the numerical solutions. E.g. the right-hand side in the clause (1.1) will yield the result of (1.6) with a default accuracy of, say, 1.0E-10. If the user wants only an absolute accuracy of 1.0E-5, he might write

(1.11) real r = interval (0,p) integral ((real x) real:
sin(x)) absacc 1.0E-5;

It will also be clear that sometimes the computation cannot be executed in the straightforward way which the programmer might have hoped for. In that case the program will have to give messages to the user in such a way that the user will be able to take proper action in order to obtain the required result. E.g. this can be done by providing more information about the problem. In order to provide the means to have the proper messages delivered to the user, an error-message system is introduced in chapter 2.

For elementary operations on vectors and matrices we refer to the TORRIX system of VAN DER MEULEN and VELDHORST (1978). We notice that we restrict ourselves to the use of a part of this system only, viz. only the parts of TORRIX BASIS and TORRIX COMPLEX are used that are fully express-ible in ALGOL 68. Moreover, we use only the application of TORRIX with mode scal = real.

The set of numerical operators we give in this report is not complete. We consider only problems in numerical algebra (chapter 3), some elementary operators in numerical analysis, such as quadrature over an interval and the search of a zero of a real function of one real variable, in chapter 4 and the integration of an initial value problem for a system of ordinary differential equations (chapter 5).

We emphasize that the report is concerned with the operator structure of the library only. Except for the most simple algorithms, the pieces of the program that perform the true computation are supposed to be implemented in the form of a procedure (possibly with a quite complex calling sequence). Although the user has the opportunity to call these basic procedures, it is the intention of the library that the procedures are called indirectly through the system of numerical operators. In this way the programmer is not bothered by (for him) irrelevant details.

The basic procedures which truly perform the computation either may be programmed in ALGOL 68 or may rely (by means of a language interface; cf. H.J. BOS and D.T. WINTER (1978)) on routines in other programming languages (e.g. FORTRAN). The bodies of these routines, i.e. the implementations of the various algorithms are not described in this report. Their existence is assumed and the operators in this report rely on them. These routines, however, are supposed to be of a type such as they are available in existing software libraries such as NUMAL, NAG or ISML.

The numerical operators described in this report are provided in the form of an ALGOL 68 prelude. Therefore, expressed in ALGOL 68 themselves, they may be considered as an extension of the language ALGOL 68 and may serve - in a much more elegant (mathematical looking) way - the same purpose as a numerical library in FORTRAN and ALGOL 60. In view of this purpose, some of the aims in the development of the system of operators were:

- it should be easy to handle,
- it should be efficient,
- it should be safe (in the sense that it does what one "normally" should expect), and
- in the case of trouble, diagnosis should be made in terms that are understandable for the user.

We remark that some of the structures defined in this report are not completely defined in the sense that the user should not know its precise mode. Objects of these modes can be created by the tools that are made available for this purpose. The fields of the struct that are known to the user can be read and changed, but possibly there are other fields in the structs which the user should ignore or (even better) wouldn't know the existence of. This way of hiding the mode of a structure leaves the possibility for implementers to extend these structures in future -if necessary- without changing the external view of the prelude (i.e. without restricting the way the prelude can be used).

2. THE ERROR MESSAGE SYSTEM

2.1. Basic ideas

The basic idea behind the error message system is to provide a complete set of routines to issue error messages and to perform all actions needed when an error is encountered. If one is not interested in the error message system, one need not to be aware of its existence, except that proper error messages may appear. However, the error message system provides the user with the ability to alter the actions made when an error is encountered.

2.1.1. Different kinds of errors

Errors will be classified in the following four groups:

a. Programming errors, i.e. errors arising because of the improper structure of the calling program. In this group are e.g. wrong array bounds or an attempt to solve a set of linear equations by means of an improperly decomposed matrix.

b. Fatal errors, i.e. errors arising because there is no (proper) solution to the problem. Example: the attempt to invert a singular matrix.

c. Non-fatal errors. Non-fatal error messages will be given if the value of the result delivered is questionable. If a routine that is designed to calculate the decomposition of a symmetric positive definite matrix finds that its input matrix is not positive definite, an error condition exists. If the routine is not able to proceed with the decomposition (e.g. Cholesky) the error is fatal. Some routines, however, are able to deliver a symmetric decomposition of a non-positive-definite matrix (e.g. LDL^T), in that case the error is non-fatal, but the result is questionable. Another example of a non-fatal error appears when an argument is passed to a special function, for which the result should be greater than max real (overflow). In this case max real is delivered and a non-fatal error is issued.

d. Trivial errors. In some cases it may be useful to provide additional information about the computational process although there is no real error encountered. An example of a trivial error is underflow in a special function. I.e. the function result for the argument given is (although non-zero) in absolute value less than the smallest representable positive real number; nevertheless zero will be delivered. Trivial errors can also be used to issue general messages, e.g. information about iterative process. (Trivial error: the number of iteration steps exceeds n; n = $1,2,3,\ldots,10^6,\ldots$.)

2.1.2. The response of the error message system

Upon detection of a non-trivial error the following actions are taken:

a. If the error message file is not yet opened, it will be opened.

b. On the error message file the current position of the file standout (pagenumber, linenumber and charnumber) and a descriptions of the error is written.

c. If the error is a programming error or a fatal error, the error message file will be closed and copied to the file standout. Moreover, the program will be terminated. The way to terminate the program is not specified here, because it depends on the implementation. If possible, a method that triggers a traceback will be used, otherwise a jump to the label STOP could be performed.

On detection of a trivial error no action is taken (however see section 2.2). If at the end of program execution the error message file is open, it will be closed and copied to the file standout.

2.2. Changing the actions of the error message system

Because the default actions of the error message system may not always meet the requirements of the user, the user has the following possibilities to change the response of the error message system.

a. There is a global boolean variable: copy error file. If set to <u>false</u> the error file will not be copied to the file standout, but, if a fatal or programming error occurs, a short message is written to standout.

b. There is a global variable <u>ref</u> <u>file</u> err file (its mode is <u>ref</u> <u>ref</u> <u>file</u>) initialized at the start of the program at <u>nil</u>. At the moment the error message system needs an error file, a default file is created and assigned to err file, unless err file has no longer the value <u>nil</u>. If one wants to use an own file for error messages, this file should be assigned to err file. One may also assign the file standout to errfile, however in this case errors will not be preceded by standout pagenumber etc., and the error messages will not be copied at program termina-

tion, regardless the value of copy error file. If one wants to switch from one error file to another, one should use the proc close error file, to close the previous file. Close error file is a dummy routine if errfile is nil or standout, in all other cases the file is closed and (if copy error file is true) copied to standout.
Note: Both, open and closed files may be assigned to errfile.

c. There is a global variable: bool detailed errors, that indicates whether one wants to have full details with error messages. If this variable is false only a precise identification of the error is given. If it is true (the default value) more information is given, e.g. the values of relevant variables.

d. The response to programming errors can not be changed by other means than specified in a to c.

e. One may change the reaction to fatal errors by an assignation to the variable errproc fatal. One of the following three objects may be assigned to fatal: errproc harderror, softerror, noerror. The reaction will be as follows:
   1. harderror: the default reaction. I.e. a message is given and the program is terminated,
   2. softerror: the error message is given, but the program is not terminated,
   3. noerror: no message is written and the program continues.

   It is not defined here how the program continues. In general the routine that detected the fatal error will return control to the calling routine, indicating the error by means of an errorflag.

f. As with fatal errors one may change the reaction to non-fatal errors, by the assignation of harderror, softerror or noerror to the variable errproc warning.

g. The reaction on trivial errors is changed by an assignation of harderror, softerror or noerror to the variable errproc inform.

h. One may also define one's own error routine to assign to the variables fatal, warning or inform. Its mode should be a proc (int, string) errorreaction where errorreaction is a struct (bool write, stop). The first parameter is an errornumber, the second is the name of the routine that detected the error. If the routine delivers true for write, the program continues with error processing, otherwise no further action is taken and the program continues. If, moreover, stop is true the program will be terminated after error processing.

Examples:

1. A fatal error in the routine abc should be repressed:

```
errorproc ownerror1 = (int num, string rout) errorreaction:
     if rout = "abc"
     then softerror(num, rout)
     else harderror(num, rout)
     fi;
fatal := ownerror1;
```

2. Suppose only a fatal error should be written to standout, while other errors are to be written to the standard error file:

```
errorproc ownerror2 = (int num, string rout) errorreaction:
     (close error file; # if file is open, close it #
      err file := standout; #messages to standout #
     (true, true) # write and stop #
     );
copy error file := false; fatal := ownerror2;
```

Note: Because of scope restrictions an error routine should not use objects with scope local to the program, but only global objects from the standard prelude or from the error message system prelude.

i. One may assign values simultaneously to the triple fatal, warning and inform. This is done by the declaration of an object of the mode errorprocs (i.e. [1:3] errorproc) and assigning it to the global variable errorprocs errorprocs. The variable is defined by the following:

fatal is errorprocs[1]
warning is errorprocs[2] and
inform is errorprocs[3].

There are two standard objects of the mode errorprocs: errorprocs

```
defaulterrors = (harderror,softerror,noerror), # the default values #
noerrors      = (noerror,noerror,noerror),     # no action in all cases #
```

j. There is a variable int error count, that counts the number of messages written. Moreover, there exists a global variable int error limit, that specifies the maximal number of messages that will be written by the system. If this limit is exceeded, a programming error is issued. The default setting of error limit is 10, one may set it to any other integral number. If error limit ≤ 0 there is no limit to the number of messages.

2.3. How to use the error message system

In general the user will not interfere with the error message system. If the default actions taken by the system do not satisfy his needs, the user can modify the actions as described in section 2.2.

If one wants to generate error messages through the error message system, this can be done as follows. One should first classify the type the error which one wants to signalize and call the appropriate routine:

class	routine
programming error	program error
fatal error	fatal
non-fatal error	warning
trivial error	inform

The routine should be called with two parameters, the first is the error number, the second is the name (string) of the routine detecting the error. Thus, throughout the library and the calling program, any possible error is identified by this error number and routine name.
The result of the routine call (which is a <u>struct</u> (<u>bool</u> write, stop)) is used to determine further action:

If write is <u>true</u> the (global) routine errorheading should be called with the same two parameters as the error routine called. This routine initiates error processing, it writes the identifying information to the error file. After this one may write (by means of the standard routine PUT) a specification of the error to errfile (a newline at the beginning is not necessary). If the specification of the error is written and detailed errors is <u>true</u> one can write as much information as will be appropriate to errfile. Error processing is finished by a call of the routine errorend with one parameter: the value of stop of the result of the first routine called.

In the case of a fatal, non-fatal or trivial error, (i.e. any non-programming error), if error processing has been completed, one should determine the actions that are needed either to continue processing or (in the case of fatal errors) to jump out of the routine. Note: even after fatal errors it is possible that the routine will regain control!

As an illustration we give a skeleton of a call to the error message system:

```
if # error condition #
then errorreaction react = fatal(number, "routine")
       # or program error etc. #
    if write of react
    then errorheading(number, "routine");
      put(errfile,("kind of error", newline));
      if detailed errors
      then put(errfile,("more information", newline)) fi;
      errorend(stop of react)
    fi;
    # either code to correct the error or a jump to leave
      the routine (probably to a label after an
      exit near the end of the routine #
fi;
```

The writing of intermediate information about the computational processes (trivial error) is done in the same way as the issuing of an error-message. The routine inform is used in this case and there is of course no corrective code.

It is possible that an error message is issued that can not be interpreted by the user. For example this may happen if a differential equation solver calls a matrix decomposer to decompose the jacobian. If the jacobian is singular, the message "singular matrix" will be issued, where the differential equation solver might have given more relevant information. In those cases the following actions can be taken:

1. Declare at the beginning of the calling routine a variable of the mode errprocs, say errorprocs errorsave.
2. Let a routine call be preceded by

   errorsave := errorprocs; errorprocs := noerrors;

   and be followed by

   errorprocs := errorsave;

   After this last clause the calling routine should test for possible errors encountered by the routine just called.

2.4. Comments

It is possible to deviate from the standard described in section 2.3. This should be done with care as deviations may lead to the impairment of the error message system.

2.5. Description of the modes, constants, variables and routines in the error message system

Modes

```
mode errorreaction = struct(bool write, stop),
     errorproc     = proc(int, string) errorreaction,
     errorprocs    = [1:3] errorproc;
```

Objects

Name	Mode
copy error file	ref bool
errfile	ref ref file
close error file	proc void
detailed errors	ref bool
program error	errorproc
hard error	errorproc
soft error	errorproc
no error	errorproc
fatal	ref errorproc
warning	ref errorproc
inform	ref errorproc
errorprocs	ref errorprocs
default errors	errorprocs
no errors	errorprocs
error count	ref int
error limit	ref int
error heading	proc (int, string) void
error end	proc (bool) void

2.6. Source text

```
´BEGIN´
    ´MODE´ ´ERRORREACTION´ = ´STRUCT´ (´BOOL´ WRITE, STOP),
        ´ERRORPROC´ = ´PROC´ (´INT´, ´STRING´) ´ERRORREACTION´;

    ´ERRORPROC´ PROGRAM ERROR = (´INT´ ERROR, ´STRING´ ROUTINE)
        ´ERRORREACTION´: (´TRUE´, ´TRUE´);
    ´ERRORPROC´ HARD ERROR = PROGRAM ERROR;
    ´ERRORPROC´ SOFT ERROR = (´INT´ ERROR, ´STRING´ ROUTINE)
        ´ERRORREACTION´: (´TRUE´, ´FALSE´);
    ´ERRORPROC´ NO ERROR = (´INT´ ERROR, ´STRING´ ROUTINE)
        ´ERRORREACTION´: (´FALSE´, ´FALSE´);

    [] ´ERRORPROC´ DEFAULT ERRORS = (HARD ERROR, SOFT ERROR, NO ERROR),
                   NO ERRORS      = (NO ERROR,   NO ERROR,   NO ERROR);

    ´MODE´ ´ERRORPROCS´ = [1:3] ´ERRORPROC´;

    ´ERRORPROCS´ ERROR PROCS:= DEFAULT ERRORS;

    ´REF´ ´ERRORPROC´ FATAL   = ERROR PROCS[1],
                      WARNING = ERROR PROCS[2],
                      INFORM  = ERROR PROCS[3];

    ´REF´ ´FILE´ ERRFILE:= ´REF´ ´FILE´ (´NIL´);

    ´INT´ CHAR OF STANDOUT;

    ´PROC´ ERROR HEADING = (´INT´ ERROR, ´STRING´ ROUTINE) ´VOID´:
        ´BEGIN´ ´IF´ ´REF´ ´FILE´ (ERRFILE) ´IS´ ´REF´ ´FILE´ (´NIL´)
            ´THEN´ ESTABLISH(ERRFILE, "N68ERRS", ZTYPE CHANNEL,
                        1, 1, 131071)
            ´FI´;
            ´IF´ ´REF´ ´FILE´ (ERRFILE) ´IS´ STANDOUT
            ´THEN´ CHAR OF STANDOUT := CHAR NUMBER(STANDOUT);
                PUT(ERRFILE, (NEWLINE, NEWLINE, 136 * "*"))
            ´FI´;
            PUT(ERRFILE, (NEWLINE, "****ERROR FROM ", ROUTINE,
                " NUMBER ", WHOLE(ERROR, - 4), NEWLINE));
            ´IF´ ´REF´ ´FILE´ (ERRFILE) ´ISNT´ STANDOUT
            ´THEN´ PUT(ERRFILE, ("POSITION OF STANDOUT: PAGE ",
                WHOLE(PAGE NUMBER(STANDOUT), - 4), ", LINE ",
                WHOLE(LINE NUMBER(STANDOUT), - 3), ", CHAR ",
                WHOLE(CHAR NUMBER(STANDOUT), - 4), NEWLINE))
            ´FI´
        ´END´;

    ´PROC´ ERROR END = (´BOOL´ STOP) ´VOID´:
        ´BEGIN´ ´IF´ ´REF´ ´FILE´ (ERRFILE) ´IS´ STANDOUT
            ´THEN´ PUT(ERRFILE, (136 * "*", NEWLINE, NEWLINE,
                    CHAR OF STANDOUT * " "))
            ´FI´;
            ERROR COUNT +:= 1;
            ´IF´ ERROR COUNT >= ERROR LIMIT ´AND´ ERROR LIMIT /= 0
                ´OR´ STOP
            ´THEN´ CLOSE ERROR FILE;
                ´IF´ ´NOT´ STOP
                ´THEN´ PUT(STANDOUT, (NEWLINE,
```

```
                                               "ERROR MESSAGE LIMIT REACHED"))
                'FI';
                ERROR
            'FI'
        'END';

    'PROC' CLOSE ERROR FILE = 'VOID':
        'IF' ('REF' 'FILE' (ERRFILE) 'ISNT' STANDOUT) 'AND'
            ('REF' 'FILE' (ERRFILE) 'ISNT' 'REF' 'FILE' ('NIL'))
        'THEN' 'IF' COPY ERROR FILE
            'THEN' RESET(ERRFILE); [1:136] 'CHAR' LINE;
                ON LOGICAL FILE END(ERRFILE,
                    ('REF' 'FILE' F) 'BOOL': L);
                CHAR OF STANDOUT:= CHAR NUMBER(STANDOUT);
                PRINT((NEWLINE, NEWLINE, 136 * "*", NEWLINE));
                'DO' GET(ERRFILE, LINE); PRINT((LINE, NEWLINE)) 'OD';
            L:  PRINT((136 * "*", NEWLINE, NEWLINE, CHAR OF STANDOUT
                    * " "));
                SCRATCH(ERRFILE); ERRFILE := 'NIL'
            'ELSE' CLOSE(ERRFILE)
            'FI'
        'FI';

    'BOOL' COPY ERROR FILE:= 'TRUE', DETAILED ERRORS:= 'TRUE';

    'INT' ERROR COUNT:= 0, ERROR LIMIT:= 10;

    'PR' PROG 'PR' 'SKIP';

STOP: CLOSE ERROR FILE
'END'
```

3. OPERATORS FOR NUMERICAL ALGEBRA

3.1. Basic ideas and definitions

3.1.1. General ideas

In this chapter we present a collection of modes, operators and procedures for the solution of linear systems and eigenvalue problems. It is the purpose of this package that the user will be able to execute many computations in numerical algebra in a simple way. So there are operators for the following computations:

1. The solution of a linear system with one or more right-hand sides.
2. The calculation of the inverse of a matrix.
3. The calculation of the determinant of a matrix.
4. The calculation of an eigensystem.
5. The calculation of singular values and singular vectors.
6. The solution of linear least squares problems.

Each of these calculations is based upon a decomposition of the original matrix, which decomposition may be used for more, similar, calculations. For this reason the various decompositions take a central place in this package. The user can have at his disposal the decompositions which he can use more than once for different calculations. Besides the values on which to operate, additional information about the original problem may be important. The main information is the form or type of matrix (see section 3.1.2. for the distinction between the different types of matrices). The accuracy of the given data is important as well for the proper definition of problem in numerical algebra (for this see section 3.1.3.).

3.1.2. Matrix storage modes

In this section we define modes for matrix storage and operators and procedures to generate such objects. In this respect the main difference between matrices is their basic form or structure, as we may distinguish symmetric and asymmetric matrices, having a bandstructure etc.. Since various kinds of matrices can be stored in a more efficient way than in a two-dimensional row of reals, we use the following three basic matrix storage modes (cf. VAN DER MEULEN and VELDHORST (1978).

```
mode mat = ref [,] real,
     vec = ref [ ] real,
  vecrow = ref [ ] ref [ ] real;
```

An element a_{ij} of a matrix (a_{ij}) may be found in the following way for the different modes:

mode	selection	comments
mat	a[i,j]	for a general matrix
	a[i,j-i]	for a band matrix (this type of storage will not be used in this report).
vec	a[(i-1)*n+j]	for a general matrix with n columns (not used in this report).
	a[(i-1)*i over 2+j]	(j ≤ i) for a symmetric matrix or a lower triangular matrix
	a[(w+rw)*(i-1)+j]	for a bandmatrix with lw subdiagonals and rw superdiagonals
	a[(i-1)*w+j]	for a symmetric band matrix with w subdiagonals
vecrow	a[i] [j]	for all different kinds of matrices.

Note that here, in agreement with ALGOL 68 practice, matrices are stored rowwise.

It is clear that for every matrix stored in a mat or a vec one may construct by simple means a vecrow in such a way that the vecrow points to the same storage cells as the original matrix. For this we do not need to copy the matrix. The reverse is not true!

Except for the case where a general matrix is stored as a mat, we have constructed modes for the different types of matrices in such a way that there are always two references to the same matrix elements. The first reference is through a vecrow and the second is through a vec using indexing as indicated above. The reason to use two references is that some routines may be made much more efficient if indexing is performed in a vec, on the other side, indexing on a vecrow is more clear and simpler to write. We introduce the following different modes for different matrix types:

Kind of matrix	mode
general	mat = ref [,] real
symmetric	symmat = struct(vecrow, mat, vec symmat)
symmetric and positive definite	possymmat = struct(vecrow mat, vec possymat)
bandstructure	bndmat = struct(vecrow mat, vec bndmat, int(w,rw)
symmetric	symbndmat = struct(vecrow mat, vec symbndmat, int w)
symmetric bandstructure	possymbndmat = struct(vecrow mat, vec possymbndmat, int w)

For each type of matrix a generating routine exists that creates the storage cells and the correct structure:

```
proc gensymmat = (int order) symmat
proc genpossymmat = (int order) possymmat
proc genbndmat = (int order, lw, rw) bndmat
proc gensymbndmat = (int order, w) symbndmat
proc genpossymbndmat = (int order, w) possymbndmat.
```

Note: In the TORRIX-system procedures are available to generate general matrices of mode mat i.e.

```
proc genmat = (int n, m) mat
```

in order to generate a general n * m matrix and

```
proc gensquare = (int n) mat
```

to generate a square matrix of order n.

With these procedures we are able to create space for (e.g.) a symmetric matrix of order n by the following declaration:

```
symmat matrix = gensymmat(n)
```

The different elements are accessible with

```
(mat of matrix) [i] [j]
```

unless j > i. It may be convenient to append to the declaration above the following:

vecrow m = mat of matrix

because this enables us to replace (mat of matrix) [i] [j] by m[i][j].

Note: In the case that assignments to rows of the matrix are made the user should take care that the different elements of the structure always should point to the same storage cells. So one should not use the following assignment:

m[i] := # some ref [ ] real #

since after this assignment m[i] will, in general, no longer refer to the corresponding part of the vector representation. For assignment to a row of the matrix, one should write:

vec(m[i]) := # the right hand side #

There is an operator (incompatible) that tests for each type of matrix the correctness of the matrix structure (i.e. whether the fields of the structure are compatible):

op incompatible = (symmat m) bool

and similar ones for the other matrix storage modes. This operator delivers true if the structure is considered to be not correct (if one of the fields of the structure is nil the structure is considered to be not correct).

Note: by these operators only the structure of a matrix is checked(!), positive definiteness of a matrix is not checked.

In some cases it is useful to reshape a matrix to a different type. To make this possible we introduce a number of conversion routines that are listed below. The input matrix has the form of either a mat or a vecrow. The routines deliver a (newly generated) structure into which the part of the original matrix has been copied. No check is made whether the original matrix has already the special shape into which it will be formed.

proc mat to symmat = (mat a, bool use lower) symmat,

proc mat to bndmat = (mat a, int (w,rw) bndmat,

proc mat to symbndmat = (mat a, int w, bool use lower) symbndmat,

and similar routines for vecrow's and:

proc vecrow to mat = (vecrow a) mat.

The parameter use lower is introduced in some procedures to indicate whether the lower triangular part or the upper triangular part of the original matrix is to be used. The following routines are also defined:

proc symmat to mat = (symmat s) mat

and for the mode <u>symbndmat</u> to store such a matrix in a rectangular array, preserving symmetry.

The following four operators are also conversion routines, but they do not copy the original matrix. They deliver a structure of the appropriate mode that points to the same storage cells as the original matrix.

<u>op</u> <u>posdef</u> = (<u>symmat</u> a) <u>possymmat</u>,
<u>op</u> <u>posdef</u> = (<u>symbndmat</u> a) <u>possymbndmat</u>,
<u>op</u> <u>notposdef</u> = (<u>possymmat</u> a) <u>symmat</u>,
<u>op</u> <u>notposdef</u> = (<u>possymbndmat</u> a) <u>symbndmat</u>.

3.1.3. Tolerances, errors etc. (Matrix problem modes)

The way to indicate the accuracy of the data, the tolerances (i.e. required precision) and the resulting precision for a numerical routine depends on the type of routine used. With procedures this is done explicitly by means of the parameter list. For operators the following conventions are used:

1. If no tolerances etc. are indicated, default values are used.
2. If the user wants to give tolerances etc., he should make use of the following provisions. In addition to each matrix storage mode, there is a matrix problem mode. These new modes are structures with two fields, the first is the corresponding matrix storage mode and the second contains the additional information required for a matrix problem to be solved. So we obtain:

   <u>mode</u> <u>matprob</u> = <u>struct</u> (<u>mat</u> mat, <u>prob</u> prob)

   etc.

   The field prob has the following mode:

   <u>mode</u> <u>prob</u> = <u>ref</u> <u>struct</u> (<u>real</u> relacc, absacc, reltol,
   abstol, <u>int</u> maxit).

   The meaning of the different fields is:

   relacc relative accuracy, i.e. the precision with which the elements of the matrix are known,

   absacc absolute accuracy,

   reltol relative tolerance, i.e. the precision desired in the result.

   abstol absolute tolerance.

   maxit maximal number of iterations when the result is obtained by an iterative method.

Not all processes use all fields, however all fields may be filled in but the fields not used will simply be ignored.
To define a problem mode one may use one or more of the following operators:

relacc, absacc, reltol, abstol and maxit.

They all accept as left-hand parameter a matrix in matrix storage mode or a matrix problem mode. The right-hand parameter is a int for maxit and a real for the other operators. The result is the problem mode associated with the matrix, where the corresponding value has been filled in. Moreover, if the left-hand parameter was in matrix storage mode the other fields of prob are filled with default values. Examples (the mode of a is a matrix storage mode or a matrix problem mode):

1. a relacc 1.0E-10
2. a reltol 1.0E-10 abstol 1.0E-10
3. a abstol 1.0E-10 reltol 1.0E.10.

There is no difference in the result of examples 2 and 3. The operators relacc etc. are called problem descriptors. If some problem descriptor is used more than once in a problem defining sequence, the right-most appearance eill be the valid specification.

3.1.4. Decomposition modes

As mentioned before, the matrix decompositions play a central part in this chapter. For each decomposition and for some matrix storage modes we have defined decomposition modes.

decomposition	storage	decomposition mode
LU	mat	lud
LU	possymmat	possymlud
LU	bndmat	bndlud
LU	possymbndmat	possymbndlud
QR(orthogonalization)	mat	qrd
SVD(singular values)	mat	svd
EVD(eigenvalues)	mat	evd
EVD	symmat	symevd
Schur	mat	shd

The decomposition modes are structures of which the fields contain all information that may be required for further calculations.

The fields of these modes are not completely defined, as they may depend upon the actual implementation. This however, is no serious problem, as it is the intention that all fields that are pertinent to the user may be fetched either directly or by means of appropriate operators.

3.2. The general format of a calling sequence

3.2.1. Linear systems, matrix inversion and determinant

The basic decomposition here is the LU-decomposition, with the associated modes lud, possymlud, bndlud and possymbndlud. In the following subsections we give the generic calling sequences for the computation of the LU-decomposition, the solution of a linear system, the matrix inverse and the determinant of a matrix.

3.2.1.1. Decomposition of a matrix

```
┌ lud            lud = ┐  dec (a [relacc relacc])
│ possymlud      lud = │         [absacc absacc]
│ bndlud         lud = │
└ possymbndlud   lud = ┘
```

The mode of a should be mat, possymmat, bndmat or possymbndmat. The result is the associated decomposition mode as outlined in chapter 3.1.4. The operator dec calculates the LU decomposition of a. This decomposition may be used for the calculation of the solution of a linear system, the inverse of a matrix or the determinant of the matrix.

[bool ok =] check lud.

This operator tells whether the operator dec has completed the decomposition, or whether decomposing is stopped because of some error condition.

3.2.1.2. Solution of a linear system

```
┌ vec res = ┐ ┌ a [relacc relacc] ┐  sol rhs
│           │ │                   │
└ mat res = ┘ {  [absacc absacc]  }
              │ ----------------- │
              └       lud         ┘
```

The mode of a should be as in section 3.2.1.1. or lud should be an object delivered by dec (3.2.1.1). The operand rhs contains the right-hand side(s) of the system, and its mode may be either vec (one right-hand side) or mat (more than one right hand side). The mode of the result is the same as the mode of rhs.

3.2.1.3. Matrix inversion

```
┌mat        inv =┐  inv ┌(a [relacc relacc])┐
│                │      │   [absacc absacc] │
│possymmat  inv =│     {                     }
└                ┘      │-------------------│
                        └        lud        ┘
```

The mode of a or lud should be the same as with sol (section 3.2.1.2). The mode of the result is the same as the mode of the original matrix, if it was mat or possymmat. If the original matrix was bndmat, the result is mat and if it was possymbndmat, the result is possymmat (the bandstructure is lost on inversion).

3.2.1.4. Calculation of determinant

```
[real d =] determ ┌(a [relacc relacc])┐
                  │  [absacc absacc   │
                 {                     }
                  │-------------------│
                  └       lud         ┘
```

The mode of a or lud should be the same as with sol (section 3.2.1.2).

3.2.2. Linear least squares problems

The basic decomposition here is the QR-decomposition, with the associated mode qrd. In the following subsections we give the generic calling sequences for the computation of the QR-decomposition, the solution of a linear least squares problem and for the calculation of the inverse of $A^T A$.

3.2.2.1. Decomposition of a matrix

```
[qrd qrd =] qrdec (a [relacc relacc])
                     [absacc absacc]
```

The mode of a is mat. The operator qrdec calculates the QR decomposition of a. This decomposition may be used for the calculation of the solution of a linear least squares problem or of the inverse of $A^T A$.

3.2.2.2. Solution of a linear least squares problem

```
[vec res =] qrd sol rhs
[mat res =]
```

The left operand qrd is of mode qrd, the right-operand is either a vec (one right-hand side) or a mat (more than one right-hand side). The mode of the result is the same as the mode of the right-operand.

3.2.2.3. Calculation of the inverse of $A^T A$

```
[possymmat inv =] inv qrd
```

Applied to an operand of the mode qrd, the operator inv delivers $(A^T A)^{-1}$, where A is the matrix of which qrd is the QR-decomposition. The result is a possymmat.

3.2.3. Singular values decomposition

For symmetric matrices see section 3.2.4.: eigenvalue decomposition of symmetric matrices. In the following subsections we give the generic calling sequences for the computation of the singular value decomposition, the singular values and the pseudo-inverse of a matrix, and the solution of homogeneous, and over - or under - determined systems of linear equations.

3.2.3.1. Decomposition of a matrix

```
[svd svd =] svdec (a [relacc relacc])
                     [absacc absacc]
                     [reltol reltol]
                     [abstol abstol]
                     [maxit    it  ]
```

The mode of a should be mat. The operator svdec calculates the singular values decomposition of a.

```
[bool ok =] check svd.
```

This operator tells whether the operator svdec has completed the decomposition properly.

3.2.3.2. Calculation of singular values

$$[\underline{\text{vec}}\ \text{sngval} =]\ \underline{\text{sngval}} \left\{\begin{array}{l} (\text{a}\ [\underline{\text{relacc}}\ \text{relacc}]) \\ \quad \underline{\text{absacc}}\ \text{absacc} \\ \quad \underline{\text{reltol}}\ \text{reltol} \\ \quad \underline{\text{abstol}}\ \text{abstol} \\ \quad \underline{\text{maxit}}\quad \text{it} \\ \hline \qquad \text{svd} \end{array}\right\}$$

The mode of a should be mat or the mode of svd should be svd. The singular values are delivered in a vec. If this operator is used on a svd, no new decomposition is made, but sngval delivers one of the fields of svd. If it is used on a mat (or matprob) first a decomposition is made, however this is a faster decomposition than the full one made by svdec.

3.2.3.3. Singular vectors and rows

$$[\underline{\text{mat}}\ \text{res} =] \left\{\begin{array}{l} \underline{\text{sngvec}} \\ \underline{\text{sngrow}} \end{array}\right\} \text{svd}$$

The mode of svd should be svd. The result of sngvec is the matrix V in the decomposition $U\ \Sigma\ V^T$, while the result of sngrow is U.

3.2.3.4. Ordering of singular values and the corresponding vectors in the singular value decomposition, numerical rank

[svd svd2 =] [bound] trims svd1.

The mode of svd1 and svd2 should be svd, the mode of bound should be real. The operator trims orders the singular values in a non-increasing ordering. The vectors in the decomposition are ordered accordingly. Moreover, in the presence of the left operand bound, the singular values less than the value of bound are set to zero. The number of nonzero singular values may be called the numerical rank of the original matrix. This numerical rank can be computed by means of

[int rank =] rank [[bound] trims] svd12.

3.2.3.5. Solution of over- or under-determined linear systems

[<u>vec</u> res =] svd <u>sol</u> rhs
[<u>mat</u> res =]

The left operand of <u>sol</u> is of mode <u>svd</u>, the right-operand is either a <u>vec</u> or a <u>mat</u> (compare section 3.2.1.2.). The mode of the result is the same as of rhs. A solution (of minimal length) in the least squares sense is obtained.

3.2.3.6. Solution of a homogeneous system

$$[\underline{\text{mat}}\ \text{res} =]\ [\text{bound}] \begin{Bmatrix} \underline{\text{homsol}} \\ \underline{\text{solhom}} \end{Bmatrix}\ \text{svd}$$

The solution of the homogeneous system $Ax = \vec{0}$ or the system $x^T A = \vec{0}^T$ is computed by means of the operators <u>homsol</u> and <u>solhom</u> respectively. Here svd is the singular values decomposition of A, and of mode <u>svd</u>. The columns of the <u>mat</u> res contain the solutions of the system. The number of columns of res equals the order of the matrix A minus the numerical rank of A. The column index is from numerical rank of A + 1 to order of A. The real (non-negative) parameter bound is used to determine the numerical rank as in section 3.2.4.3.; its default value is zero.

3.2.3.7. Calculation of the pseudo-inverse

[<u>mat</u> inv =] [bound] <u>inv</u> svd.

Here svd is of mode <u>svd</u> and the result is of mode <u>mat</u>. The operator <u>inv</u> calculates the pseudo-inverse A^+ of the matrix A for which svd is the singular values decomposition. The real, nonnegative, parameter bound is used to determine the numerical rank as in section 3.2.4.3; its default value is zero.

3.2.4. Eigenvalue decomposition of symmetric matrices

For general matrices see section 3.2.5. and 3.2.6. In the following subsections we give the generic calling sequences for the calculation of the eigenvalue decomposition for symmetric matrices. Moreover, information is given about the use of this decomposition as a singular values decomposition.

3.2.4.1. Decomposition of a matrix

```
[symevd evd =] evdec (a [relacc relacc])
                        [absacc absacc]
                        [reltol reltol]
                        [abstol abstol]
                        [maxit    it  ]
```

The mode of a should be symmat. The operator evdec calculates the eigenvalue decomposition of a.

```
[bool ok =] check evd
```

This operator tells whether the operator evdec has completed the decomposition properly.

3.2.4.2. Calculation of eigenvalues

```
[vec eigval =] eigval ⎧(a [relacc relacc])⎫
                      ⎪   [absacc absacc] ⎪
                      ⎪   [reltol reltol] ⎪
                      ⎨   [abstol abstol] ⎬
                      ⎪   [maxit    it  ] ⎪
                      ⎪------------------ ⎪
                      ⎩        evd        ⎭
```

The mode of a should be symmat or the mode of evd should be symevd. The eigenvalues are delivered in a vec. If this operator is used on a symevd, no new decomposition is made, but eigval delivers one of the fields of evd. If it is used on a symmat (or symmatprob) first a decomposition is made, however this is a faster decomposition than the full one made by evdec.

3.2.4.3. Eigenvectors

```
[mat res =] eigvec evd.
```

The parameter evd should be of mode symevd, the result is the matrix of eigencolumns taken from the structure evd.

3.2.4.4. Relation with singular values decomposition

For symmetric matrices the eigenvalue decomposition may be used as a singular values decomposition. Hence the operators sngval, trims, sol, homsol, solhom and inv may be used here with the same meaning as for the standard singular values decomposition (section 3.2.4.). Generic calling sequences:

```
[vec sngval =] sngval ⎧( a [relacc relacc])⎫
                      ⎪    [absacc absacc] ⎪
                      ⎨    [reltol reltol] ⎬
                      ⎪    [abstol abstol] ⎪
                      ⎪    [maxit     it ] ⎪
                      ⎪ ------------------ ⎪
                      ⎩        evd         ⎭
```

The mode of a should be symmat or the mode of evd should be symevd, the result is of mode vec.

```
[symevd evd2 =] [bound] trims evd1.
```

The mode of evd1 should be symevd, the optional parameter bound is of mode real, the mode of the result is symevd

```
[int rank =] rank [[bound] trims] evd1
```

calculates the numerical rank.

```
[vec res =] evd sol rhs
[mat res =]
```

The mode of evd should be symevd and the mode of rhs should be vec or mat. The mode of the result is the same as the mode of rhs.

```
[mat res =] [bound]⎧homsol⎫ evd
                   ⎩solhom⎭
```

The mode of evd should be symmevd, the optional parameter bound is of mode real and the result is of mode mat.

```
[symmat inv =] [bound] inv evd
```

Evd should be of mode symevd, bound of mode real and the result is of mode symmat.

The operators sngvec and sngrow are not needed here as they would deliver (apart from signs) the matrix of eigenvectors.

3.2.5. Schur decomposition and the computation of eigenvalues for general (real square) matrices

For symmetric matrices the Schur decomposition is identical to the eigenvalue decomposition (see section 3.2.4.). For general matrices, the Schur decomposition is sufficient for the calculation of the eigenvalues, and may be seen as an intermediate stage for the calculation of eigenvectors. Below we give the generic calling sequences for the calculation of the Schur decomposition and for the calculation of eigenvectors.

3.2.5.1. Decomposition of a matrix

```
[shd shd =] shdec (a [relacc relacc])
                     [absacc absacc]
                     [reltol reltol]
                     [abstol abstol]
                     [maxit  maxit ]
```

The mode of a should be mat. The result is the Schur decomposition.

```
[bool ok =] check shd.
```

This operator reports whether the operator shdec has complete the decomposition properly.

3.2.5.2. Calculation of eigenvalues

```
[covec eigval =] eigval ⎧(a [relacc relacc])⎫
                        ⎪   [absacc absacc] ⎪
                        ⎨   [reltol reltol] ⎬
                        ⎪   [abstol abstol] ⎪
                        ⎪   [maxit     it ] ⎪
                        ⎪ ----------------- ⎪
                        ⎩        shd        ⎭
```

The mode of a should be mat or the mode of shd should be shd. The result is of mode **covec**, which is **ref** [ ] **compl**, cf. VAN DER MEULEN and VELDHORST (1978).

3.2.6. Eigenvalue decomposition for general matrices

In the following subsections we give generic calling sequences for the computation of the eigenvalue decomposition of a general (real square) matrix and for the calculation of its eigenvectors and eigenrows. See also section 3.2.5. for the calculation of eigenvalues.

3.2.6.1. Decomposition of a matrix

$$[\underline{\text{evd}}\ \text{evd} =]\left\{\begin{array}{l}\underline{\text{evdec}}\\ \underline{\text{fevdec}}\end{array}\right\}\left\{\begin{array}{c}(\text{a } [\underline{\text{relacc}}\ \text{relacc}])\\ [\underline{\text{absacc}}\ \text{absacc}]\\ [\underline{\text{reltol}}\ \text{reltol}]\\ [\underline{\text{abstol}}\ \text{abstol}]\\ [\underline{\text{maxit}}\quad \text{it}\quad]\\ \hline \text{shd}\end{array}\right\}$$

The mode of a should be **mat** or the mode of shd should be **shd**, the result is the eigenvalue decomposition. The operator **evdec** does not calculate the eigenrows, while **fevdec** calculates the full eigenvalue decomposition, including eigenrows.

$$[\underline{\text{bool}}\ \text{ok} =]\ \underline{\text{check}}\ \text{evd}$$

This operator reports whether the operator **evd** or the operator **fevd** has completed the decomposition properly.

3.2.6.2. Calculation of eigenvalues

$$[\underline{\text{covec}}\ \text{eigval} =]\ \underline{\text{eigval}}\ \text{evd}$$

The mode of evd should be **evd**. For other possible parameters of the operator **eigval** see section 3.2.5.2.

3.2.6.3. Calculation of eigenvectors or eigenrows

$$[\underline{\text{comat}}\ \text{res} =]\left\{\begin{array}{l}\underline{\text{eigval}}\\ \underline{\text{eigrow}}\end{array}\right\}\left\{\begin{array}{l}\text{shd}\\ \text{evd}\end{array}\right\}$$

The mode of shd and evd should be shd or evd respectively. The result is of mode comat, which is ref [,] compl, cf. VAN DER MEULEN and VELDHORST (1978). Note: The operator eigrow can not be used on an object of the mode evd, if this object has been calculated by evdec, but only on an operand calculated by fevdec.

3.3. Comments

3.3.0. The matrix storage modes

As a general comment we give here the motivation for the matrix storage modes as defined in section 3.1.2.

If the matrix has a special form as outlined in section 3.1.2. the following three (contradicting) points have to be considered:

1. Indexing of elements, columns and rows should be clear.
2. The matrix should be stored as efficient as possible.
3. Indexing of elements, columns and rows should be efficient.

Now we consider the three possible ways of storing a matrix:

As a mat. Here, in general, the matrix is not stored efficiently, however, indexing is both clear and fast.

As a vecrow. The matrix is stored efficiently. Indexing is clear. Indexing is also efficient if we consider elements and rows only (i.e., for symmetric matrices, if we consider the lower triangle only).

As a vec. Both storage and indexing are efficient. Indexing on columns is efficient as the stride between two elements of a column is either constant or (with symmetric matrices) increases with 1 on successive elements. Indexing is not clear in this case.

Except for the case where the mat representation is the obvious choice, these consideration led us to matrix storage modes as structured fields, containing both a vecrow and a vec pointing to the same storage calls. In this way we can choose at wish the clear but in some cases inefficient indexing on a vecrow or the opaque but efficient indexing on a vec.

3.3.1. Linear systems, matrix inversion and determinant

Four different kinds of matrices are recognized:

1. general matrices
2. positive definite symmetric matrices

3. band matrices
4. positive definite symmetric band matrices.

In this report no routines are defined for the solution of linear systems with general symmetric matrices. However, one can imagine an implementation that provides routines for these matrices.

3.3.2. Linear least squares problems

The only matrix storage mode allowed is mat. In order to avoid errors, there is no operator sol, accepting a non-square matrix as a left operand. For calculation of the least squares solution of the overdetermined system Ax = b one should use either

qrdec a sol b

(using the QR-decomposition) or

[bound trims] svdec a sol rhs

(using the singular value decomposition).

3.3.3. Singular value decomposition

This section does not contain special operators for symmetric matrices, because the singular value decomposition is identical to the eigenvalue decomposition in this case (apart from signs). So these operators are to be found in section 3.2.4.

3.3.4. Eigenvalue decomposition of symmetric matrices

For the eigenvalue decomposition, the symmetric matrices are separated from the general matrices because processing differs in many respects, e.g.:

1. For general matrices the Schur decomposition is used at an intermediate stage, while for symmetric matrices this decomposition is identical to the eigenvalue decomposition.
2. The eigenvalue decomposition of symmetric matrices may be used as a kind of singular values decomposition. This is not true for general matrices.
3. All eigenvalues and eigenvectors of symmetric matrices are real, This is not true for the general matrices.

3.3.5. Schur decomposition for general real matrices

In general the Schur decomposition is not real for real matrices. Hence, a slightly modified form is used, where the matrix in the decomposition may contain some non-zero subdiagonal elements. However, at least one of to successive subdiagonal elements is zero. The Schur decomposition is an intermediate step to the eigenvalue decomposition. The Schur decomposition itself yields the eigenvalues immediately.

3.3.6. Eigenvalue decomposition of general real matrices

This decomposition is calculated from the Schur decomposition. Eigenvalue decomposition is not always possible (defect matrices) and, if possible, not always stable. So, in general, no good results can be guaranteed.

3.4. Source text

```
'BEGIN'
    'MODE' 'VEC' = 'REF' [] 'REAL',
        'COVEC' = 'REF' [] 'COMPL',
        'MAT' = 'REF' [,] 'REAL',
        'COMAT' = 'REF' [,] 'COMPL',
        'VECROW' = 'REF' [] 'VEC';  # CF TORRIX #

    'MODE' 'SYMMAT' = 'STRUCT' ('VECROW' MAT,
                                'VEC' SYMMAT),
          'POSSYMMAT' = 'STRUCT' ('VECROW' MAT,
                                  'VEC' POSSYMMAT),
          'BNDMAT' = 'STRUCT' ('VECROW' MAT,
                               'VEC' BNDMAT,
                               'INT' LW, RW),
          'SYMBNDMAT' = 'STRUCT' ('VECROW' MAT,
                                  'VEC' SYMBNDMAT,
                                  'INT' W),
          'POSSYMBNDMAT' = 'STRUCT' ('VECROW' MAT,
                                     'VEC' POSSYMBNDMAT,
                                     'INT' W);

    'PROC' GENSYMMAT = ('INT' ORDER) 'SYMMAT': 'SKIP',
       GENPOSSYMMAT = ('INT' ORDER) 'POSSYMMAT': 'SKIP',
       GENBNDMAT = ('INT' ORDER, LW, RW) 'BNDMAT': 'SKIP',
       GENSYMBNDMAT = ('INT' ORDER, W) 'SYMBNDMAT': 'SKIP',
       GENPOSSYMBNDMAT = ('INT' ORDER, W) 'POSSYMBNDMAT': 'SKIP',

    'OP' 'INCOMPATIBLE' = ('SYMMAT' M) 'BOOL': 'SKIP',
        'INCOMPATIBLE' = ('POSSYMMAT' M) 'BOOL': 'SKIP',
        'INCOMPATIBLE' = ('BNDMAT' M) 'BOOL': 'SKIP',
        'INCOMPATIBLE' = ('SYMBNDMAT' M) 'BOOL': 'SKIP',
        'INCOMPATIBLE' = ('POSSYMBNDMAT' M) 'BOOL': 'SKIP';

    'PROC' MAT TO SYMMAT = ('MAT' M, 'BOOL' USE LOWER) 'SYMMAT': 'SKIP',
       MAT TO BNDMAT = ('MAT' M, 'INT' LW, RW) 'BNDMAT': 'SKIP',
       MAT TO SYMBNDMAT = ('MAT' M, 'INT' W, 'BOOL' USE LOWER)
           'SYMBNDMAT': 'SKIP',
       VECROW TO MAT = ('VECROW' M) 'MAT': 'SKIP',
       VECROW TO SYMMAT = ('MAT' M, 'BOOL' USE LOWER)
           'SYMMAT': 'SKIP',
       VECROW TO BNDMAT = ('MAT' M, 'INT' LW, RW)
           'BNDMAT': 'SKIP',
       VECROW TO SYMBNDMAT = ('MAT' M, 'INT' W, 'BOOL' USE LOWER)
           'SYMBNDMAT': 'SKIP',
       SYMMAT TO MAT = ('SYMMAT' M) 'MAT': 'SKIP',
       SYMBNDMAT TO MAT = ('SYMBNDMAT' M) 'MAT': 'SKIP';

    'OP' 'POSDEF' = ('SYMMAT' M) 'POSSYMMAT':
           (MAT 'OF' M, SYMMAT 'OF' M),
        'POSDEF' = ('SYMBNDMAT' M) 'POSSYMBNDMAT':
           (MAT 'OF' M, SYMBNDMAT 'OF' M, W 'OF' M),
        'NOTPOSDEF' = ('POSSYMMAT' M) 'SYMMAT':
           (MAT 'OF' M, POSSYMMAT 'OF' M),
        'NOTPOSDEF' = ('POSSYMBNDMAT' M) 'SYMBNDMAT':
           (MAT 'OF' M, POSSYMBNDMAT 'OF' M, W 'OF' M);

    'MODE' 'PRB' = 'STRUCT' ('REAL' RELACC, ABSACC, RELTOL, ABSTOL,
```

```
                         'INT' MAXIT);
'MODE' 'PROB' = 'REF' 'PRB';

'MODE' 'MATPROB' = 'STRUCT' ('MAT' MAT, 'PROB' PROB),
    'SYMMATPROB' = 'STRUCT' ('SYMMAT' MAT, 'PROB' PROB),
    'POSSYMMATPROB' = 'STRUCT' ('POSSYMMAT' MAT, 'PROB' PROB),
    'BNDMATPROB' = 'STRUCT' ('BNDMAT' MAT, 'PROB' PROB),
    'SYMBNDMATPROB' = 'STRUCT' ('SYMBNDMAT' MAT, 'PROB' PROB),
    'POSSYMBNDMATPROB' = 'STRUCT' ('POSSYMBNDMAT' MAT, 'PROB' PROB);

'PRIO' 'RELACC' = 8, 'ABSACC' = 8,
       'RELTOL' = 8, 'ABSTOL' = 8,
       'MAXIT'  = 8;

'OP' 'DEFPROB' = ('MAT' M) 'PROB':
    ('HEAP' 'PRB' PROB :=
        (SMALL REAL, SMALL REAL, SMALL REAL * 10, SMALL REAL * 10,
             'SIZE' M * 10
        );
    PROB
    ),
    'DEFPROB' = ('SYMMAT' M) 'PROB':
    ('HEAP' 'PRB' PROB :=
        (SMALL REAL, SMALL REAL, SMALL REAL * 10, SMALL REAL * 10,
             'SIZE' M * 10
        );
    PROB
    ),
    'DEFPROB' = ('POSSYMMAT' M) 'PROB':
    ('HEAP' 'PRB' PROB :=
        (SMALL REAL, SMALL REAL, SMALL REAL * 10, SMALL REAL * 10,
             'SIZE' M * 10
        );
    PROB
    ),
    'DEFPROB' = ('BNDMAT' M) 'PROB':
    ('HEAP' 'PRB' PROB :=
        (SMALL REAL, SMALL REAL, SMALL REAL * 10, SMALL REAL * 10,
             'SIZE' M * 10
        );
    PROB
    ),
    'DEFPROB' = ('SYMBNDMAT' M) 'PROB':
    ('HEAP' 'PRB' PROB :=
        (SMALL REAL, SMALL REAL, SMALL REAL * 10, SMALL REAL * 10,
             'SIZE' M * 10
        );
    PROB
    ),
    'DEFPROB' = ('POSSYMBNDMAT' M) 'PROB':
    ('HEAP' 'PRB' PROB :=
        (SMALL REAL, SMALL REAL, SMALL REAL * 10, SMALL REAL * 10,
             'SIZE' M * 10
        );
    PROB
    );
```

```
'OP' 'SIZE' = ('MAT' M) 'INT': 'UPB' M - 'LWB' M + 1,
    'SIZE' = ('SYMMAT' M) 'INT':
        'UPB' MAT 'OF' M - 'LWB' MAT 'OF' M + 1,
    'SIZE' = ('POSSYMMAT' M) 'INT':
        'UPB' MAT 'OF' M - 'LWB' MAT 'OF' M + 1,
    'SIZE' = ('BNDMAT' M) 'INT':
        'UPB' MAT 'OF' M - 'LWB' MAT 'OF' M + 1,
    'SIZE' = ('SYMBNDMAT' M) 'INT':
        'UPB' MAT 'OF' M - 'LWB' MAT 'OF' M + 1,
    'SIZE' = ('POSSYMBNDMAT' M) 'INT':
        'UPB' MAT 'OF' M - 'LWB' MAT 'OF' M + 1;

'OP' 'RELTOL' = ('PROB' P, 'REAL' R) 'PROB':
        (RELTOL 'OF' P := R; P),
    'ABSTOL' = ('PROB' P, 'REAL' R) 'PROB':
        (ABSTOL 'OF' P := R; P),
    'RELACC' = ('PROB' P, 'REAL' R) 'PROB':
        (RELACC 'OF' P := R; P),
    'ABSACC' = ('PROB' P, 'REAL' R) 'PROB':
        (ABSACC 'OF' P := R; P),
    'MAXIT' = ('PROB' P, 'INT' I) 'PROB':
        (MAXIT 'OF' P := I; P),

'OP' 'RELACC' = ('MAT' M, 'REAL' R) 'MATPROB':
        (M, 'DEFPROB' M 'RELACC' R),
    'ABSACC' = ('MAT' M, 'REAL' R) 'MATPROB':
        (M, 'DEFPROB' M 'ABSACC' R),
    'RELTOL' = ('MAT' M, 'REAL' R) 'MATPROB':
        (M, 'DEFPROB' M 'RELTOL' R),
    'ABSTOL' = ('MAT' M, 'REAL' R) 'MATPROB':
        (M, 'DEFPROB' M 'ABSTOL' R),
    'MAXIT' = ('MAT' M, 'INT' I) 'MATPROB':
        (M, 'DEFPROB' M 'MAXIT' I),

    'RELTOL' = ('MATPROB' M, 'REAL' R) 'MATPROB':
        (RELTOL 'OF' PROB 'OF' M := R; M),
    'ABSTOL' = ('MATPROB' M, 'REAL' R) 'MATPROB':
        (ABSTOL 'OF' PROB 'OF' M := R; M),
    'RELACC' = ('MATPROB' M, 'REAL' R) 'MATPROB':
        (RELACC 'OF' PROB 'OF' M := R; M),
    'ABSACC' = ('MATPROB' M, 'REAL' R) 'MATPROB':
        (ABSACC 'OF' PROB 'OF' M := R; M),
    'MAXIT' = ('MATPROB' M, 'INT' I) 'MATPROB':
        (MAXIT 'OF' PROB 'OF' M := I; M),

    'RELACC' = ('SYMMAT' M, 'REAL' R) 'SYMMATPROB':
        (M, 'DEFPROB' M 'RELACC' R),
    'ABSACC' = ('SYMMAT' M, 'REAL' R) 'SYMMATPROB':
        (M, 'DEFPROB' M 'ABSACC' R),
    'RELTOL' = ('SYMMAT' M, 'REAL' R) 'SYMMATPROB':
        (M, 'DEFPROB' M 'RELTOL' R),
    'ABSTOL' = ('SYMMAT' M, 'REAL' R) 'SYMMATPROB':
        (M, 'DEFPROB' M 'ABSTOL' R),
    'MAXIT' = ('SYMMAT' M, 'INT' I) 'SYMMATPROB':
        (M, 'DEFPROB' M 'MAXIT' I),

    'RELTOL' = ('SYMMATPROB' M, 'REAL' R) 'SYMMATPROB':
```

```
    (RELTOL 'OF' PROB 'OF' M := R; M),
'ABSTOL' = ('SYMMATPROB' M, 'REAL' R) 'SYMMATPROB':
    (ABSTOL 'OF' PROB 'OF' M := R; M),
'RELACC' = ('SYMMATPROB' M, 'REAL' R) 'SYMMATPROB':
    (RELACC 'OF' PROB 'OF' M := R; M),
'ABSACC' = ('SYMMATPROB' M, 'REAL' R) 'SYMMATPROB':
    (ABSACC 'OF' PROB 'OF' M := R; M),
'MAXIT' = ('SYMMATPROB' M, 'INT' I) 'SYMMATPROB':
    (MAXIT 'OF' PROB 'OF' M := I; M),

'RELACC' = ('POSSYMMAT' M, 'REAL' R) 'POSSYMMATPROB':
    (M, 'DEFPROB' M 'RELACC' R),
'ABSACC' = ('POSSYMMAT' M, 'REAL' R) 'POSSYMMATPROB':
    (M, 'DEFPROB' M 'ABSACC' R),
'RELTOL' = ('POSSYMMAT' M, 'REAL' R) 'POSSYMMATPROB':
    (M, 'DEFPROB' M 'RELTOL' R),
'ABSTOL' = ('POSSYMMAT' M, 'REAL' R) 'POSSYMMATPROB':
    (M, 'DEFPROB' M 'ABSTOL' R),
'MAXIT' = ('POSSYMMAT' M, 'INT' I) 'POSSYMMATPROB':
    (M, 'DEFPROB' M 'MAXIT' I),

'RELTOL' = ('POSSYMMATPROB' M, 'REAL' R) 'POSSYMMATPROB':
    (RELTOL 'OF' PROB 'OF' M := R; M),
'ABSTOL' = ('POSSYMMATPROB' M, 'REAL' R) 'POSSYMMATPROB':
    (ABSTOL 'OF' PROB 'OF' M := R; M),
'RELACC' = ('POSSYMMATPROB' M, 'REAL' R) 'POSSYMMATPROB':
    (RELACC 'OF' PROB 'OF' M := R; M),
'ABSACC' = ('POSSYMMATPROB' M, 'REAL' R) 'POSSYMMATPROB':
    (ABSACC 'OF' PROB 'OF' M := R; M),
'MAXIT' = ('POSSYMMATPROB' M, 'INT' I) 'POSSYMMATPROB':
    (MAXIT 'OF' PROB 'OF' M := I; M),

'RELACC' = ('BNDMAT' M, 'REAL' R) 'BNDMATPROB':
    (M, 'DEFPROB' M 'RELACC' R),
'ABSACC' = ('BNDMAT' M, 'REAL' R) 'BNDMATPROB':
    (M, 'DEFPROB' M 'ABSACC' R),
'RELTOL' = ('BNDMAT' M, 'REAL' R) 'BNDMATPROB':
    (M, 'DEFPROB' M 'RELTOL' R),
'ABSTOL' = ('BNDMAT' M, 'REAL' R) 'BNDMATPROB':
    (M, 'DEFPROB' M 'ABSTOL' R),
'MAXIT' = ('BNDMAT' M, 'INT' I) 'BNDMATPROB':
    (M, 'DEFPROB' M 'MAXIT' I),

'RELTOL' = ('BNDMATPROB' M, 'REAL' R) 'BNDMATPROB':
    (RELTOL 'OF' PROB 'OF' M := R; M),
'ABSTOL' = ('BNDMATPROB' M, 'REAL' R) 'BNDMATPROB':
    (ABSTOL 'OF' PROB 'OF' M := R; M),
'RELACC' = ('BNDMATPROB' M, 'REAL' R) 'BNDMATPROB':
    (RELACC 'OF' PROB 'OF' M := R; M),
'ABSACC' = ('BNDMATPROB' M, 'REAL' R) 'BNDMATPROB':
    (ABSACC 'OF' PROB 'OF' M := R; M),
'MAXIT' = ('BNDMATPROB' M, 'INT' I) 'BNDMATPROB':
    (MAXIT 'OF' PROB 'OF' M := I; M),

'RELACC' = ('SYMBNDMAT' M, 'REAL' R) 'SYMBNDMATPROB':
    (M, 'DEFPROB' M 'RELACC' R),
'ABSACC' = ('SYMBNDMAT' M, 'REAL' R) 'SYMBNDMATPROB':
```

```
        (M, 'DEFPROB' M 'ABSACC' R),
    'RELTOL' = ('SYMBNDMAT' M, 'REAL' R) 'SYMBNDMATPROB':
        (M, 'DEFPROB' M 'RELTOL' R),
    'ABSTOL' = ('SYMBNDMAT' M, 'REAL' R) 'SYMBNDMATPROB':
        (M, 'DEFPROB' M 'ABSTOL' R),
    'MAXIT' = ('SYMBNDMAT' M, 'INT' I) 'SYMBNDMATPROB':
        (M, 'DEFPROB' M 'MAXIT' I),

    'RELTOL' = ('SYMBNDMATPROB' M, 'REAL' R) 'SYMBNDMATPROB':
        (RELTOL 'OF' PROB 'OF' M := R; M),
    'ABSTOL' = ('SYMBNDMATPROB' M, 'REAL' R) 'SYMBNDMATPROB':
        (ABSTOL 'OF' PROB 'OF' M := R; M),
    'RELACC' = ('SYMBNDMATPROB' M, 'REAL' R) 'SYMBNDMATPROB':
        (RELACC 'OF' PROB 'OF' M := R; M),
    'ABSACC' = ('SYMBNDMATPROB' M, 'REAL' R) 'SYMBNDMATPROB':
        (ABSACC 'OF' PROB 'OF' M := R; M),
    'MAXIT' = ('SYMBNDMATPROB' M, 'INT' I) 'SYMBNDMATPROB':
        (MAXIT 'OF' PROB 'OF' M := I; M),

    'RELACC' = ('POSSYMBNDMAT' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (M, 'DEFPROB' M 'RELACC' R),
    'ABSACC' = ('POSSYMBNDMAT' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (M, 'DEFPROB' M 'ABSACC' R),
    'RELTOL' = ('POSSYMBNDMAT' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (M, 'DEFPROB' M 'RELTOL' R),
    'ABSTOL' = ('POSSYMBNDMAT' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (M, 'DEFPROB' M 'ABSTOL' R),
    'MAXIT' = ('POSSYMBNDMAT' M, 'INT' I) 'POSSYMBNDMATPROB':
        (M, 'DEFPROB' M 'MAXIT' I),

    'RELTOL' = ('POSSYMBNDMATPROB' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (RELTOL 'OF' PROB 'OF' M := R; M),
    'ABSTOL' = ('POSSYMBNDMATPROB' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (ABSTOL 'OF' PROB 'OF' M := R; M),
    'RELACC' = ('POSSYMBNDMATPROB' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (RELACC 'OF' PROB 'OF' M := R; M),
    'ABSACC' = ('POSSYMBNDMATPROB' M, 'REAL' R) 'POSSYMBNDMATPROB':
        (ABSACC 'OF' PROB 'OF' M := R; M),
    'MAXIT' = ('POSSYMBNDMATPROB' M, 'INT' I) 'POSSYMBNDMATPROB':
        (MAXIT 'OF' PROB 'OF' M := I; M);

'PRIO' 'SOL' = 2,
       'INV' = 2,
       'TRIMS' = 3,
       'HOMSOL' = 2,
       'SOLHOM' = 2;

'MODE' 'LUD' = 'STRUCT' ('MAT' LU # AND OTHER FIELDS #,
                         'BOOL' READY # SET BY 'DEC' #),
    'POSSYMLUD' = 'STRUCT' ('POSSYMMAT' LU # AND OTHER FIELDS #,
                            'BOOL' READY # SET BY 'DEC' #),
    'BNDLUD' = 'STRUCT' ('BNDMAT' LU # AND OTHER FIELDS #,
                         'BOOL' READY # SET BY 'DEC' #),
    'POSSYMBNDLUD' = 'STRUCT' ('POSSYMBNDMAT' LU # OTHER FIELDS #,
                               'BOOL' READY # SET BY 'DEC' #);

'OP' 'DEC' = ('MATPROB' M) 'LUD': 'SKIP';
```

```
'OP' 'DEC' = ('MAT' M) 'LUD':
        ('DEC' 'MATPROB' (M, 'DEFPROB' M));
'OP' 'DEC' = ('POSSYMMATPROB' M) 'POSSYMLUD': 'SKIP';
'OP' 'DEC' = ('POSSYMMAT' M) 'POSSYMLUD':
        ('DEC' 'POSSYMMATPROB' (M, 'DEFPROB' M));
'OP' 'DEC' = ('BNDMATPROB' M) 'BNDLUD': 'SKIP';
'OP' 'DEC' = ('BNDMAT' M) 'BNDLUD':
        ('DEC' 'BNDMATPROB' (M, 'DEFPROB' M));
'OP' 'DEC' = ('POSSYMBNDMATPROB' M) 'POSSYMBNDLUD': 'SKIP';
'OP' 'DEC' = ('POSSYMBNDMAT' M) 'POSSYMBNDLUD':
        ('DEC' 'POSSYMBNDMATPROB' (M, 'DEFPROB' M));

'OP' 'CHECK' = ('LUD' LUD) 'BOOL': READY 'OF' LUD,
    'CHECK' = ('POSSYMLUD' LUD) 'BOOL': READY 'OF' LUD,
    'CHECK' = ('BNDLUD' LUD) 'BOOL': READY 'OF' LUD,
    'CHECK' = ('POSSYMBNDLUD' LUD) 'BOOL': READY 'OF' LUD;

'OP' 'SOL' = ('LUD' LUD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('LUD' LUD, 'MAT' RHS) 'MAT':
       (['LWB' RHS:'UPB' RHS,2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
        'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
        'DO' SOL[,I] := LUD 'SOL' RHS[,I] 'OD';
        SOL
       ),
    'SOL' = ('MATPROB' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('MATPROB' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS,
    'SOL' = ('MAT' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('MAT' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS;
'OP' 'SOL' = ('POSSYMLUD' LUD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('POSSYMLUD' LUD, 'MAT' RHS) 'MAT':
       (['LWB' RHS:'UPB' RHS,2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
        'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
        'DO' SOL[,I] := LUD 'SOL' RHS[,I] 'OD';
        SOL
       ),
    'SOL' = ('POSSYMMATPROB' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMMATPROB' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMMAT' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMMAT' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS;
'OP' 'SOL' = ('BNDLUD' LUD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('BNDLUD' LUD, 'MAT' RHS) 'MAT':
       (['LWB' RHS:'UPB' RHS,2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
        'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
        'DO' SOL[,I] := LUD 'SOL' RHS[,I] 'OD';
        SOL
       ),
    'SOL' = ('BNDMATPROB' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('BNDMATPROB' M, 'MAT' RHS) 'MAT':
```

```
        'DEC' M 'SOL' RHS,
    'SOL' = ('MATPROB' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS,
    'SOL' = ('MAT' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('MAT' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS;
'OP' 'SOL' = ('POSSYMLUD' LUD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('POSSYMLUD' LUD, 'MAT' RHS) 'MAT':
        ([ 'LWB' RHS:'UPB' RHS,2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
         'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
         'DO' SOL[,I] := LUD 'SOL' RHS[,I] 'OD';
         SOL
        ),
    'SOL' = ('POSSYMMATPROB' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMMATPROB' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMMAT' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMMAT' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS;
'OP' 'SOL' = ('BNDLUD' LUD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('BNDLUD' LUD, 'MAT' RHS) 'MAT':
        ([ 'LWB' RHS:'UPB' RHS,2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
         'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
         'DO' SOL[,I] := LUD 'SOL' RHS[,I] 'OD';
         SOL
        ),
    'SOL' = ('BNDMATPROB' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('BNDMATPROB' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS,
    'SOL' = ('BNDMAT' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('BNDMAT' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS;
'OP' 'SOL' = ('POSSYMBNDLUD' LUD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('POSSYMBNDLUD' LUD, 'MAT' RHS) 'MAT':
        ([ 'LWB' RHS:'UPB' RHS,2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
         'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
         'DO' SOL[,I] := LUD 'SOL' RHS[,I] 'OD';
         SOL
        ),
    'SOL' = ('POSSYMBNDMATPROB' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMBNDMATPROB' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMBNDMAT' M, 'VEC' RHS) 'VEC':
        'DEC' M 'SOL' RHS,
    'SOL' = ('POSSYMBNDMAT' M, 'MAT' RHS) 'MAT':
        'DEC' M 'SOL' RHS;

'OP' 'INV' = ('LUD' LUD) 'MAT': 'SKIP';
'OP' 'INV' = ('MATPROB' M) 'MAT': 'INV' 'DEC' M,
    'INV' = ('MAT' M) 'MAT': 'INV' 'DEC' M;
'OP' 'INV' = ('POSSYMLUD' LUD) 'POSSYMMAT': 'SKIP';
```

```
'OP' 'INV' = ('POSSYMMATPROB' M) 'POSSYMMAT': 'INV' 'DEC' M,
    'INV' = ('POSSYMMAT' M) 'POSSYMMAT': 'INV' 'DEC' M;
'OP' 'INV' = ('BNDLUD' LUD) 'MAT': 'SKIP';
'OP' 'INV' = ('BNDMATPROB' M) 'MAT': 'INV' 'DEC' M,
    'INV' = ('BNDMAT' M) 'MAT': 'INV' 'DEC' M;
'OP' 'INV' = ('POSSYMBNDLUD' LUD) 'POSSYMMAT': 'SKIP';
'OP' 'INV' = ('POSSYMBNDMATPROB' M) 'POSSYMMAT': 'INV' 'DEC' M,
    'INV' = ('POSSYMBNDMAT' M) 'POSSYMMAT': 'INV' 'DEC' M;

'OP' 'DETERM' = ('LUD' LUD) 'REAL': 'SKIP';
'OP' 'DETERM' = ('MATPROB' M) 'REAL': 'DETERM' 'DEC' M,
    'DETERM' = ('MAT' M) 'REAL': 'DETERM' 'DEC' M;
'OP' 'DETERM' = ('POSSYMLUD' LUD) 'REAL': 'SKIP';
'OP' 'DETERM' = ('POSSYMMATPROB' M) 'REAL': 'DETERM' 'DEC' M,
    'DETERM' = ('POSSYMMAT' M) 'REAL': 'DETERM' 'DEC' M;
'OP' 'DETERM' = ('BNDLUD' LUD) 'REAL': 'SKIP';
'OP' 'DETERM' = ('BNDMATPROB' M) 'REAL': 'DETERM' 'DEC' M,
    'DETERM' = ('BNDMAT' M) 'REAL': 'DETERM' 'DEC' M;
'OP' 'DETERM' = ('POSSYMBNDLUD' LUD) 'REAL': 'SKIP';
'OP' 'DETERM' = ('POSSYMBNDMATPROB' M) 'REAL': 'DETERM' 'DEC' M,
    'DETERM' = ('POSSYMBNDMAT' M) 'REAL': 'DETERM' 'DEC' M;

'MODE' 'QRD' = 'STRUCT' ('MAT' QR # AND OTHER FIELDS #);

'OP' 'QRDEC' = ('MATPROB' M) 'QRD': 'SKIP';
'OP' 'QRDEC' = ('MAT' M) 'QRD':
        ('QRDEC' 'MATPROB' (M, 'DEFPROB' M));

'OP' 'SOL' = ('QRD' QRD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('QRD' QRD, 'MAT' RHS) 'MAT':
        (['LWB' (QR 'OF' QRD):'UPB' (QR 'OF' QRD),
            2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
         'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
         'DO' SOL[,I] := QRD 'SOL' RHS[,I] 'OD';
         SOL
        );

'OP' 'INV' = ('QRD' QRD) 'POSSYMMAT': 'SKIP';

'MODE' 'SVD' = 'STRUCT' ('MAT' SV # AND OTHER FIELDS #,
                         'BOOL' READY # SET BY SVDEC #,
                         'VEC' SNGVAL # DEFINED BY SVDEC #);

'OP' 'SVDEC' = ('MATPROB' M) 'SVD': 'SKIP';
'OP' 'SVDEC' = ('MAT' M) 'SVD':
        ('SVDEC' 'MATPROB' (M, 'DEFPROB' M));

'OP' 'CHECK' = ('SVD' SVD) 'BOOL': READY 'OF' SVD;

'OP' 'SNGVAL' = ('MATPROB' M) 'VEC': 'SKIP';
'OP' 'SNGVAL' = ('MAT' M) 'VEC':
        ('SNGVAL' 'MATPROB' (M, 'DEFPROB' M));
'OP' 'SNGVAL' = ('SVD' SVD) 'VEC': SNGVAL 'OF' SVD;

'OP' 'SNGVEC' = ('SVD' SVD) 'MAT': 'SKIP',
    'SNGROW' = ('SVD' SVD) 'MAT': 'SKIP';
```

```
'OP' 'TRIMS' = ('REAL' R, 'SVD' SVD) 'SVD': 'SKIP';
'OP' 'TRIMS' = ('SVD' SVD) 'SVD': 0.0 'TRIMS' SVD;

'OP' 'RANK' = ('SVD' SVD) 'INT': 'SKIP';

'OP' 'SOL' = ('SVD' SVD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('SVD' SVD, 'MAT' RHS) 'MAT':
        (['LWB' (SV 'OF' SVD):'UPB' (SV 'OF' SVD),
            2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
         'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
         'DO' SOL[,I] := SVD 'SOL' RHS[,I] 'OD';
         SOL
        );

'OP' 'HOMSOL' = ('REAL' R, 'SVD' SVD) 'MAT': 'SKIP',
    'SOLHOM' = ('REAL' R, 'SVD' SVD) 'MAT': 'SKIP';
'OP' 'HOMSOL' = ('SVD' SVD) 'MAT': 0.0 'HOMSOL' SVD,
    'SOLHOM' = ('SVD' SVD) 'MAT': 0.0 'SOLHOM' SVD;

'OP' 'INV' = ('REAL' R, 'SVD' SVD) 'MAT': 'SKIP';
'OP' 'INV' = ('SVD' SVD) 'MAT': 0.0 'INV' SVD;

'MODE' 'SYMEVD' = 'STRUCT' ('MAT' EV # AND OTHER FIELDS #,
                            'BOOL' READY # SET BY EVDEC #,
                            'VEC' EIGVAL # DEFINED BY EVDEC #);

'OP' 'EVDEC' = ('SYMMATPROB' M) 'SYMEVD': 'SKIP';
'OP' 'EVDEC' = ('SYMMAT' M) 'SYMEVD':
        ('EVDEC' 'SYMMATPROB' (M, 'DEFPROB' M));

'OP' 'CHECK' = ('SYMEVD' EVD) 'BOOL': READY 'OF' EVD;

'OP' 'EIGVAL' = ('SYMMATPROB' M) 'VEC': 'SKIP';
'OP' 'EIGVAL' = ('SYMMAT' M) 'VEC':
        ('EIGVAL' 'SYMMATPROB' (M, 'DEFPROB' M));
'OP' 'EIGVAL' = ('SYMEVD' EVD) 'VEC': EIGVAL 'OF' EVD;

'OP' 'EIGVEC' = ('SYMEVD' EVD) 'MAT': 'SKIP';

'OP' 'SNGVAL' = ('SYMMATPROB' M) 'VEC': 'SKIP';
'OP' 'SNGVAL' = ('SYMMAT' M) 'VEC':
        ('SNGVAL' 'SYMMATPROB' (M, 'DEFPROB' M));
'OP' 'SNGVAL' = ('SYMEVD' EVD) 'VEC': 'SKIP';

'OP' 'TRIMS' = ('REAL' R, 'SYMEVD' EVD) 'SYMEVD': 'SKIP';
'OP' 'TRIMS' = ('SYMEVD' EVD) 'SYMEVD': 0.0 'TRIMS' EVD;

'OP' 'RANK' = ('SYMEVD' EVD) 'INT': 'SKIP';

'OP' 'SOL' = ('SYMEVD' EVD, 'VEC' RHS) 'VEC': 'SKIP';
'OP' 'SOL' = ('SYMEVD' EVD, 'MAT' RHS) 'MAT':
        (['LWB' (EV 'OF' EVD):'UPB' (EV 'OF' EVD),
            2 'LWB' RHS:2 'UPB' RHS] 'REAL' SOL;
         'FOR' I 'FROM' 2 'LWB' RHS 'TO' 2 'UPB' RHS
         'DO' SOL[,I] := EVD 'SOL' RHS[,I] 'OD';
         SOL
        );
```

```
'OP' 'HOMSOL' = ('REAL' R, 'SYMEVD' EVD) 'MAT': 'SKIP',
    'SOLHOM' = ('REAL' R, 'SYMEVD' EVD) 'MAT': 'SKIP';
'OP' 'HOMSOL' = ('SYMEVD' EVD) 'MAT': 0.0 'HOMSOL' EVD,
    'SOLHOM' = ('SYMEVD' EVD) 'MAT': 0.0 'SOLHOM' EVD;

'OP' 'INV' = ('REAL' R, 'SYMEVD' EVD) 'SYMMAT': 'SKIP';
'OP' 'INV' = ('SYMEVD' EVD) 'SYMMAT': 0.0 'INV' EVD;

'MODE' 'SHD' = 'STRUCT' ('MAT' SH # AND OTHER FIELDS #,
                         'BOOL' READY # SET BY SHDEC #,
                         'COVEC' EIGVAL # DEFINED BY SHDEC #);

'OP' 'SHDEC' = ('MATPROB' M) 'SHD': 'SKIP';
'OP' 'SHDEC' = ('MAT' M) 'SHD':
        ('SHDEC' 'MATPROB' (M, 'DEFPROB' M));

'OP' 'CHECK' = ('SHD' SHD) 'BOOL': READY 'OF' SHD;

'OP' 'EIGVAL' = ('MATPROB' M) 'COVEC': 'SKIP';
'OP' 'EIGVAL' = ('MAT' M) 'COVEC':
        ('EIGVAL' 'MATPROB' (M, 'DEFPROB' M));
'OP' 'EIGVAL' = ('SHD' SHD) 'COVEC': EIGVAL 'OF' SHD;

'MODE' 'EVD' = 'STRUCT' ('COMAT' COL, ROW # AND OTHER FIELDS #,
                         'BOOL' READY # SET BY EVDEC #,
                         'COVEC' EIGVAL # DEFINED BY EVDEC #);

'OP' 'EVDEC' = ('SHD' SHD) 'EVD': 'SKIP',
    'FEVDEC' = ('SHD' SHD) 'EVD': 'SKIP';
'OP' 'EVDEC' = ('MATPROB' M) 'EVD':
        ('EVDEC' 'SHDEC' M),
    'FEVDEC' = ('MATPROB' M) 'EVD':
        ('FEVDEC' 'SHDEC' M),
    'EVDEC' = ('MAT' M) 'EVD':
        ('EVDEC' 'SHDEC' 'MATPROB' (M, 'DEFPROB' M)),
    'FEVDEC' = ('MAT' M) 'EVD':
        ('FEVDEC' 'SHDEC' 'MATPROB' (M, 'DEFPROB' M));

'OP' 'CHECK' = ('EVD' EVD) 'BOOL': READY 'OF' EVD;

'OP' 'EIGVAL' = ('EVD' EVD) 'COVEC': EIGVAL 'OF' EVD;

'OP' 'EIGVEC' = ('EVD' EVD) 'COMAT': COL 'OF' EVD,
    'EIGROW' = ('EVD' EVD) 'COMAT': ROW 'OF' EVD;

'SKIP'
'END'
```

4. ELEMENTARY NUMERICAL ANALYSIS

4.1. General remarks

This chapter describes some elementary operators for the calculation of:

a. a zero of a function on a given interval,

b. a minimum or a maximum of a function on a interval,

c. the definite integral of a function over a given interval.

4.4.1. Intervals

The interval over which the problem has to be solved is defined by the struct range. In order that "infinite" may be used in a way similar to the use of ∞ in the mathematical notation for infinite intervals, the following modes are defined:

```
mode infinite = struct (bool pos), # pos indicates whether
                                      +∞ or -∞ is meant #
mode point = union (real, infinite);
```

There is one standard object of the mode infinite: infinite. This objects indicates +∞. Further, the following operators are defined for objects of the mode infinite and real:

monadic + and -, and the comparison operators < ≤ = ≥ > and ≠.

All have their natural meaning. For objects of the mode point the following operators are defined:

lt, le, eq, ge, gt and ne.

To define intervals, the following mode is given:

```
mode range = struct (point low, upp);
```

(low ≥ upp is permitted).
To define an interval a cast is now sufficient e.g.:

```
range (1.0, infinite)
range (-infinite, +infinite)
```

are legal intervals.
Note. Open, closed and half-open intervals are not distinguished.
Note. If we would waive infinite intervals we can abandon the union and define: mode range = struct (real low, upp).

4.1.2. Functions

All main operators in this chapter deal with real functions of one real variable i.e.

mode function = proc (real) real.

4.1.3. Tolerances, errors etc.

To pass to the particular routines additional information, such as tolerances and required precision, the following construction is created.

Beside the mode function there is also a mode funcprob. This is a struct of which the first field contains the function and the other fields contain information about its accuracy. To the operators in this section one may pass a funcprob as well as a function. Objects of the mode funcprob are created and their fields are filled with the relevant information by the following operators (the left hand operand is a function or a funcprob while the mode of the right hand operand depends upon the kind of information):

relacc defines relative accuracy, its right hand operand is real,
absacc defines absolute accuracy, its right hand operand is real,
accx defines absolute accuracy as a function of an independent variable, its right hand operand is a function.

If a funcprob is defined by

f relacc r absacc s accx t,

the inaccuracy in the computation of f at the point x is considered to be

r*f(x) + s + t(x).

4.2. The general form of a calling sequence

4.2.1. The search for a zero, a maximum or a minimum of a function

```
[real res :=] range (low,upp) ⎧zero   ⎫ function [relacc relacc]
                              ⎨minimum⎬          [absacc absacc]
                              ⎩maximum⎭          [accx   accx  ]
```

The modes of function and accx are function, the modes of relacc and absacc are real. Low and upp can be of mode real, of mode infinite or of mode point. This clause calculates a zero, a minimum or a maximum of the given functions on the interval from low to upp. The accuracy of the function is defined by relacc, absacc and accx.

4.2.2. Adaptive quadrature

A definite integral, is computed by:

[real int :=] range (low, upp) integral function [relacc relacc]
[absacc absacc]
[accx accx]

Here the modes of function, relacc, absacc and accx are function, real, real and function respectively; low and upp may be real, infinite or point. The above clause calculates:

$$\text{int} := \int_{\text{low}}^{\text{upp}} \text{function}(x)\,dx$$

with the best possible accuracy, taking into account that the accuracy of the function is defined by relacc, absacc and accx. If a method is specified the following calculation is performed:

$$\text{int} := \int_{\text{low}}^{\text{upp}} \text{function}(x)\,w(x)\,dx$$

where the weighting function w(x) depend upon the method (see section 4.2.3.). If relacc, absacc and accx are not specified the following default values are taken:

relacc := small real;
absacc := small real;
accx := ((real x) real: 0.0).

4.2.3. Fixed method quadrature

For quadrature by means of a prescribed method one can use

[real int :=] range (low, upp) integral f method {gauss legendre (n)
gauss hermite (n,s)
gauss laguerre (a,n,s)
gauss jacobi (a,b,n)}

The modes of low and upp are real, infinite or point; the mode of f is function or funcprob; the mode of n is integer and the mode of a, b and s is real.

The clause calculates

$$\text{int} := \int_{\alpha}^{\beta} f(x)w(x)dx$$

by means of an n-point Gauss quadrature rule.

The values of w(x), α and β are restricted, depending on the method called, as given in the following table.

	α	β	w(x)	a	b	s
gauss legendre	low	upp	1			
gauss hermite	low	±inf	$\exp(-(\frac{x-\alpha}{s})^2)$			> 0
gauss laguere	low	+inf	$\exp(-(\frac{x-\alpha}{s}))$			> 0
gauss jacobi	low	upp	$(x-\alpha)^a(\beta-x)^b$	> - 1	> - 1	

REMARK. For quadrature by means of a prescribed method, in the prelude the following procedures are provided:

proc gausslegendre = (int n) methinf
proc gausshermite = (int n, real scale) methinf
proc gausslaguere = (real a, int n, real scale) methinf
proc gaussjacobi = (real a, b, int n) methinf.

These routines do not perform quadrature themselves but they deliver an object (the mode of which is methinf). If this object is combined with either a function or a funcprob by means of the operator method (e.g. func method gausslegendre(10)) the result is another object (the mode of which is methint). This object may be passed to the operator integral, which performs the integration by the method defined by the initial procedure call. Gauss Legendre and Gauss Jacobi quadrature can only be applied on finite intervals, Gauss Hermite and Gauss Laguerre only on half-infinite intervals. The intervals, are mapped onto the standard interval by an affine transformation, which is determined by low and s (i.e. scale).

EXAMPLE.

```
range (a,-infinite) integral ((real x) real: sin(x)/x)
                    method gauss hermite (10,s)
```

computes

$$\int_{a}^{-\infty} e^{-\left(\frac{x-a}{s}\right)^2} \frac{\sin(x)}{x}\,dx$$

by means of a 10-point Gauss-Hermite integration.

4.3. Comments

1. Although infinite intervals are defined, this does not imply that all operators can be applied to infinite intervals. This may depend on the particular implementation of the prelude text.
2. The list of procedures for non-standard integration is not limitative, morever, the definition of the modes methinf and methint is not strict. These modes depend upon the actual non-standard integrations that are implemented. This means that with the implementation of more non-standard methods of integration the number of fields in the struct methint may be increased.

4.4. Source text

```
'BEGIN'
   'MODE' 'INFINITE' = 'STRUCT' ('BOOL' POS);
   'MODE' 'POINT'    = 'UNION' ('REAL','INFINITE');
   'MODE' 'RANGE'    = 'STRUCT' ('POINT' FROM, TO);
   'MODE' 'FUNCTION' = 'PROC' ('REAL') 'REAL';
   'MODE' 'FUNCPROB' = 'STRUCT' ('FUNCTION' F, 'REAL' RELACC, ABSACC,
                                 'REF' 'FUNCTION' ACCX);

   'INFINITE' INFINITE = ('INFINITE' I; POS 'OF' I := 'TRUE'; I);

   'OP' + = ('INFINITE' INF) 'INFINITE':     INF;

   'OP' - = ('INFINITE' INF) 'INFINITE':
      ('INFINITE' I; POS 'OF' I := 'NOT' (POS 'OF' INF); I);

   'OP' < = ('INFINITE' I, J) 'BOOL':
      'NOT' (POS 'OF' I) 'AND' (POS 'OF' J),
   'OP' <= = ('INFINITE' I, J) 'BOOL':
      'NOT' (POS 'OF' I) 'OR' (POS 'OF' J),
   'OP' = = ('INFINITE' I, J) 'BOOL':
      (POS 'OF' I) = (POS 'OF' J),
   'OP' /= = ('INFINITE' I, J) 'BOOL':
      (POS 'OF' I) /= (POS 'OF' J),
   'OP' >= = ('INFINITE' I, J) 'BOOL':
      (POS 'OF' I) 'OR' 'NOT' (POS 'OF' J),
   'OP' > = ('INFINITE' I, J) 'BOOL':
      (POS 'OF' I) 'AND' 'NOT' (POS 'OF' J),

   'OP' < = ('INFINITE' I, 'REAL' R) 'BOOL':
      'NOT' (POS 'OF' I),
   'OP' <= = ('INFINITE' I, 'REAL' R) 'BOOL':
      'NOT' (POS 'OF' I),
   'OP' = = ('INFINITE' I, 'REAL' R) 'BOOL': 'FALSE',
   'OP' /= = ('INFINITE' I, 'REAL' R) 'BOOL': 'TRUE',
   'OP' >= = ('INFINITE' I, 'REAL' R) 'BOOL':
      POS 'OF' I,
   'OP' > = ('INFINITE' I, 'REAL' R) 'BOOL':
      POS 'OF' I,

   'OP' < = ('REAL' R, 'INFINITE' I) 'BOOL':
      POS 'OF' I,
   'OP' <= = ('REAL' R, 'INFINITE' I) 'BOOL':
      POS 'OF' I,
   'OP' = = ('REAL' R, 'INFINITE' I) 'BOOL': 'FALSE',
   'OP' /= = ('REAL' R, 'INFINITE' I) 'BOOL': 'TRUE',
   'OP' >= = ('REAL' R, 'INFINITE' I) 'BOOL':
      'NOT' (POS 'OF' I),
   'OP' > = ('REAL' R, 'INFINITE' I) 'BOOL':
      'NOT' (POS 'OF' I),

   'OP' 'LT' = ('POINT' P, Q) 'BOOL':
      'CASE' P
      'IN' ('REAL' R):    'CASE' Q
                          'IN' ('REAL' S): R < S,
                               ('INFINITE' J): POS 'OF' J
                          'ESAC',
         ('INFINITE' I):  'NOT' (POS 'OF' I) 'AND'
```

```
                           'CASE' Q
                           'IN' ('REAL' S): 'TRUE',
                               ('INFINITE' J): POS 'OF' J
                           'ESAC'
    'ESAC',
'OP' 'LE' = ('POINT' P, Q) 'BOOL':
    'CASE' P
    'IN' ('REAL' R):       'CASE' Q
                           'IN' ('REAL' S): R <= S,
                               ('INFINITE' J): POS 'OF' J
                           'ESAC',
        ('INFINITE' I):    'NOT' (POS 'OF' I) 'OR'
                           'CASE' Q
                           'IN' ('REAL' S): 'FALSE',
                               ('INFINITE' J): POS 'OF' J
                           'ESAC'
    'ESAC',
'OP' 'EQ' = ('POINT' P, Q) 'BOOL':
    'CASE' P
    'IN' ('REAL' R):       'CASE' Q
                           'IN' ('REAL' S): R = S,
                               ('INFINITE' J): 'FALSE'
                           'ESAC',
        ('INFINITE' I):    'CASE' Q
                           'IN' ('REAL' S): 'FALSE',
                               ('INFINITE' J): I = J
                           'ESAC'
    'ESAC',
'OP' 'NE' = ('POINT' P, Q) 'BOOL':
    'CASE' P
    'IN' ('REAL' R):       'CASE' Q
                           'IN' ('REAL' S): R /= S,
                               ('INFINITE' J): 'TRUE'
                           'ESAC',
        ('INFINITE' I):    'CASE' Q
                           'IN' ('REAL' S): 'TRUE',
                               ('INFINITE' J): I /= J
                           'ESAC'
    'ESAC',
'OP' 'GE' = ('POINT' P, Q) 'BOOL':
    'CASE' P
    'IN' ('REAL' R):       'CASE' Q
                           'IN' ('REAL' S): R >= S,
                               ('INFINITE' J): 'NOT' (POS 'OF' J)
                           'ESAC',
        ('INFINITE' I):    (POS 'OF' I) 'OR'
                           'CASE' Q
                           'IN' ('REAL' S): 'FALSE',
                               ('INFINITE' J): 'NOT' (POS 'OF' J)
                           'ESAC'
    'ESAC',
'OP' 'GT' = ('POINT' P, Q) 'BOOL':
    'CASE' P
    'IN' ('REAL' R):       'CASE' Q
                           'IN' ('REAL' S): R > S,
                               ('INFINITE' J): 'NOT' (POS 'OF' J)
                           'ESAC',
```

```
        ('INFINITE' I): (POS 'OF' I) 'AND'
                        'CASE' Q
                        'IN' ('REAL' S): 'TRUE',
                             ('INFINITE' J): 'NOT' (POS 'OF' J)
                        'ESAC'
    'ESAC',

'FUNCPROB' DEFAULTFUNCPROB = ('SKIP', SMALLREAL, SMALLREAL, 'NIL');

'OP' 'SETFUN' = ('FUNCTION' F) 'REF' 'FUNCPROB':
('HEAP' 'FUNCPROB' I := DEFAULT FUNCPROB; F 'OF' I := F; I);

'OP' 'ABSACC' = ('REF' 'FUNCPROB' F, 'REAL' TOL) 'REF' 'FUNCPROB':
(ABSACC 'OF' F := TOL; F);

'OP' 'RELACC' = ('REF' 'FUNCPROB' F, 'REAL' TOL) 'REF' 'FUNCPROB':
(RELACC 'OF' F := TOL; F);

'OP' 'ACCX' = ('REF' 'FUNCPROB' F, 'FUNCTION' ACC) 'REF' 'FUNCPROB':
(ACCX 'OF' F := 'HEAP' 'FUNCTION' := ACC; F);

'OP' 'RELACC' = ('FUNCTION' F, 'REAL' TOL) 'REF' 'FUNCPROB':
'SETFUN' F 'RELACC' TOL;

'OP' 'ABSACC' = ('FUNCTION' F, 'REAL' TOL) 'REF' 'FUNCPROB':
'SETFUN' F 'ABSACC' TOL;

'OP' 'ACCX' = ('FUNCTION' F, ACC) 'REF' 'FUNCPROB':
'SETFUN' F 'ACCX' ACC;

'MODE' 'METHINF' = 'STRUCT' (
         'PROC' ('RANGE', 'FUNCPROB', 'METHINF') 'REAL' INTEGRATOR
                              # AND PROBABLY SOME OTHER FIELDS #);
'MODE' 'METHINT' = 'STRUCT' ('FUNCPROB' F #THE PROBLEM#,
                   'METHINF' M #NON STANDARD INFORMATION #);

'PROC' GAUSSLEGENDRE = ('INT' N) 'METHINF': 'SKIP',
       GAUSSHERMITE = ('INT' N, 'REAL' SCALE) 'METHINF': 'SKIP',
       GAUSSLAGUERRE = ('REAL' A, 'INT' N, 'REAL' SCALE) 'METHINF':
                         'SKIP',
       GAUSSJACOBI = ('REAL' A, B, 'INT' N) 'METHINF': 'SKIP';

'OP' 'METHOD' = ('FUNCPROB' F, 'METHINF' M) 'METHINT': (F, M);

'OP' 'METHOD' = ('FUNCTION' F, 'METHINF' M) 'METHINT':
    'SETFUN' F 'METHOD' M;

'OP' 'ZERO' = ('RANGE' R, 'FUNCPROB' S) 'REAL': 'SKIP',
'OP' 'ZERO' = ('RANGE' R, 'FUNCTION' F) 'REAL':
    R 'ZERO' 'SETFUN' F,

'OP' 'MINIMUM' = ('RANGE' R, 'FUNCPROB' S) 'REAL': 'SKIP',
'OP' 'MINIMUM' = ('RANGE' R, 'FUNCTION' F) 'REAL':
    R 'MINIMUM' 'SETFUN' F,

'OP' 'MAXIMUM' = ('RANGE' R, 'FUNCPROB' S) 'REAL': 'SKIP',
'OP' 'MAXIMUM' = ('RANGE' R, 'FUNCTION' F) 'REAL':
```

```
        R 'MAXIMUM' 'SETFUN' F,

    'OP' 'INTEGRAL' = ('RANGE' R, 'FUNCPROB' S) 'REAL': 'SKIP',
         'INTEGRAL' = ('RANGE' R, 'METHINT' M) 'REAL':
            ((INTEGRATOR 'OF' M 'OF' M) (R, F 'OF' M, M 'OF' M));
    'OP' 'INTEGRAL' = ('RANGE' R, 'FUNCTION' F) 'REAL':
        R 'INTEGRAL' 'SETFUN' F,

    'PRIO' 'INTEGRAL' = 2, 'RELACC' = 3, 'ABSACC' = 3, 'ACCX' = 3,
          'METHOD' = 3, 'SETFUN' = 3, 'ZERO' = 2, 'MINIMUM' = 2,
          'MAXIMUM' = 2;

    'SKIP'
'END'
```

5. OPERATORS FOR THE SOLUTION OF O.D.E.'s

5.0. General remarks

In this chapter we describe operators for the numerical solution of initial value problems (i.v.p.) of the following form: given a differential equation

$$(5.0.1)\qquad \begin{cases} dy/dx = f(x,y), \\ y(a) \;\;\;\, = ya, \end{cases}$$

where y and f are vector-functions of dimension N, compute the value of y(x) at a sequence of points $x_1, x_1, \ldots, x_n, \ldots$.

Operators have been constructed in such a way that, given a value x_i, a vector $y(x_i)$ will be computed. If the process did already compute the solution of the differential equation over an interval $[a, x_{i-1}]$, the information that can be used for the continuation of the computational process is stored and will be used in the subsequent call when integration is continued from x_{i-1} to x_i.

The user can specify a relative and an absolute tolerance parameter to control the accuracy of the computational process. Beside these parameters, the user can provide a number of additional data to control the process (such as bounds for the steplength, a scaling vector or a jacobian matrix) in order to meet his particular purposes or to increase the efficiency of the computation.

If the problem cannot be solved according to the given specifications, an errormessage and a suggetion for other specifications will be returned.

5.1. Basic ideas

The general idea behind the operators that are described in section 5.3 is that almost all data concerning the numerical solution of the i.v.p. (5.0.1) and the specifications how to solve it, can be stored in a <u>struct</u> (the mode of this struct is <u>ivp</u>; an object that refers to such a struct we shall call ivp). As soon as the value x_i, the endpoint of integration, is known, the vector $y(x_i)$ can be computed. This computation is performed by means of the statement

(5.1.1) <u>solve</u> ivp <u>until</u> x_i.

After execution, the solution is found in

sol *of* ivp,

which is a *ref* *vec* pointing to a [1:N] *real*. After execution of (5.1.1), all other information concerning the i.v.p., contained in ivp has been updated up to the point x_i. A subsequent call will use this new information and will update again, etc.

If a particular method for the solution of (5.0.1) is to be specified, this can be done by

(5.1.2) *solve* ivp *until* xi *method* method1.

All specifications, except the particular method to use, are available in the *ivp* ivp. The clause (5.1.2) allows for solution of the problem by the method as specified by method1.

The initial value problem to be solved (eq.(5.0.1)) is specified by

(5.1.3.a) *ivp* ivp := *state* (a,ya) *integrates* f

or

(5.1.3.b) *ivp* ivp := f *integrates* *state* (a,ya).

Both statements have the same meaning: the initial conditions are given in *state* (a,ya), a is the starting value of the independent variable, the vector ya contains the initial values; and eq. (5.0.1) has to be solved for the right hand side f. The mode of a, ya and f should be *real*, *vec* and *proc* (*real*, *vec*, *vec*) *void* respectively. Execution of (5.1.1), where ivp is defined by (5.1.3) would result in solution of the problem by default specifications.

Each default specifications can be changed in two ways:

A.) either by application of the operator *spec* (with some proper right operand) to ivp (as a left operand):

(5.1.4) ivp *spec* spec.

Here spec is a *struct* containing a set of specifications,

B.) or by application of special operators to specify the specific specifications; e.g.

ivp *hmin* hmin.

Here hmin is a real number which specifies the minimal steplength to be used in the computational process. Table 5.1. gives a list of operators that are available to specify specifications which can be used in the same way as hmin. A description of these specifications is given at the end of this section.

operator	mode of the right hand operand	default value of specificator
reltol	real	0.001
abstol	real	small real
scale	real	1.0
scale	vec	nil
jac	mat	nil
jac	ref proc(real,vec,mat)void	nil
monitor	ref proc(ref ivp)void	nil
hmin	real	small real
hmax	real	max real
hstart	real	skip
stiff	bool	true
linear	bool	false
continuable	bool	true
restart	bool	false
maxevals	int	maxint
maxsteps	int	maxint

Table 5.1

Operators available to control the integration of an i.v.p. and default values of the specificators.

A sequence of operators from table 5.1 and corresponding right hand side operands can be given. If the same operator is used more than once in the same sequence, the rightmost appearance is the valid specification.

A sequence of specifications can also be put at the right hand side of a set of specifications as given by spec spec. The effect is now that the particular soecifications given in spec will be changed as specified.

EXAMPLES

<u>specif</u> defaultspecification = (...);

<u>specif</u> spec, spec1 , spec2; spec1 := ...;

(spec := defaultspecification) <u>reltol</u> 1.0E-6

here a set of specifications is generated deviating from the default values only by the fact that the relative tolerance is set to 1.0E-6 #.

ivp <u>spec</u> (spec2 := spec1) <u>hmin</u> 0.0001 <u>hmax</u> 0.1

here the specification as given in spec1 are used for the i.v.p. ivp except that the minimal steplength is set to 0.0001 and the maximal steplength to 0.1 #.

If the computation cannot be performed according to the given specifications, the given specifications are changed and a softerror (warning) message is given.

Sometimes it may be useful to have explicitly available as an identifier the vector containing the solution of the differential equation at xi. This can be obtained by

(5.1.5) <u>solve</u> ivp <u>until</u> <u>state</u> (xi, yend);

here yend is an identifier of the mode <u>vec</u>. After execution this <u>vec</u> points to the same [1:N] <u>real</u> as sol <u>of</u> ivp. Thus the solution vector can be overwritten on the initial values by

(5.1.6) f <u>integrates</u> <u>states</u> (x0,y) <u>until</u> <u>state</u> (xend, y).

<u>The</u> <u>specifications</u> (additional information about the inital value problem).

The user can specify how he wants the i.v.p. to be solved in the following ways:

The user may specify by means of a boolean value:

1) whether the problem is stiff or not (after the operator <u>stiff</u>),
2) whether the problem is linear of not (after the operator <u>linear</u>),
3) whether the differential equation can be continued after the specified final value of the independent variable (after the operator <u>continuable</u>). E.g. if the function f in eq. (5.0.1) is discontinuous at $x = x_n$, the differential equation is not continuable when it is solved until x_n.

4) whether the computational process has to be restarted with the given values of the dependent and independent variable or that additional information from previous steps can be used. (After the operator restart.)

The user may specify by means of an integer value:

1) the maximal number of evaluations of the right-hand side of the equation, i.e. the function f (after the operator maxevals)

2) the maximal number of integration steps (after the operator maxsteps).

The user may specify by means of a positive real number:

1)2)3) the minimal and maximal steplength with which the integration is performed and a suggestion for a first steplength. (After the operators hmin, hmax, and hstart respectively).

The user may specify by means of a mat (an approximation of) the jacobian matrix in matrix form or by means of a proc (real, vec, mat) void, he may give an analytical expression of the Jacobian matrix. This procedure proc jac = (real x, vec y, mat m) void, should deliver the Jacobian matrix $\left(\frac{\partial f_i(x,y)}{\partial y_j}\right)$ in the mat m, x and y being given by the real x and the vec y respectively. The specification concerning the jacobian matrix should be given after the operator jac.

The user may specify a routine for intermediate output by means of a proc (ref ivp) void. This routine of which the name should be given after the operator monitor, will be called after each succesful step of the integration process. This routine can be used to monitor continuously the variables during integration. At each call of the routine the ivp which is passed to this routine contains the information that is needed to interpolate the solution of the differential equation over the last step. When a call of the routine interrupts the integration, all information to continue integration from the last point reached is contained in the ivp.

The tolerance parameters (error control).

The operators give also several possibilities to specify bounds for the local error. A step in the integration process is considered to be succesful if it satisfies the requirement

$$\text{loc error}_i < \text{abstol} * \text{scale}_i + \text{reltol} * |y_i|$$

for each i-th component of the vector y.

Here loc error_i is an estimate of the local error in this step and abstol and reltol are given values (≥ 0) for the absolute and relative tolerance (not both = 0); scale_i is a given scaling factor for the i-th component. The scaling can be given by means of a vec of which the i-th component contains scale_i or by means of a real. In the latter case all values of scale_i are taken equal to the value of this real. The default value is $\text{scale}_i = 1.0$. Default values for abstol and reltol are smallreal and 0.001 respectively.

5.2. The general form of a calling sequence

The general form of the calling sequence of operators for the solution of an i.v.p. of the form (5.0.1) is given in fig.5.2.1. The modes of the various identifiers are given in table 5.2.2.

```
[ivp :=] ⎧state (x,y) integrates f⎫ [spec spec] [reltol reltol]    ⎡until ⎧[endpoint] xe        ⎫ [method ivproc]⎤
         ⎨f integrates state (x,y)⎬             [abstol abstol]    ⎢      ⎨state (xe, ⎧y ⎫ )    ⎬                ⎥
         ⎩solve ivp               ⎭             [scale {w}     ]   ⎣      ⎩           ⎩ye⎭      ⎭                ⎦
                                                       {c}
                                                [jac {matrix }]
                                                     {routine}
                                                [monitor outproc]
                                                [hmin hmin      ]
                                                [hmax hmax      ]
                                                [hstart h       ]
                                                [stiff stiff    ]
                                                [linear lin     ]
                                                [continuable cont ]
                                                [restart restart  ]
                                                [maxevals maxevals]
                                                [maxsteps maxsteps]
```

Fig. 5.2.1

The generic calling sequence for the solution of an i.v.p. The modes of the identifiers are given in table 5.2.2.

mode	identifier
bool	stiff, cont, lin, restart
int	maxevals, maxsteps
real	x, xe, hmin, hmax, h
	reltol, abstol, c
vec	y, ye, w
mat	matrix
ivproc	ivproc
proc (real, vec, vec) void	f
ref proc (real, vec, mat) void	routine
ref proc (ref ivp) void	outproc
ref ivp	ivp
ref specif	spec

Table 5.2.2.

The modes of the variables given in Fig.5.2.1.

5.3. Description of modes and operators

In this section we give a survey of the modes and operators that are used to allow for the solution of an initial value problem by means of a clause as described in section 5.2.

5.3.1. Modes

```
mode specif = struct (bool stiff, linear, continuable, restart,
                      int maxevals, maxsteps,
                      real reltol, abstol, scalescal,
                           hmin, hmax, hstart,
                      vec scalevec,
                      mat jacobian mat,
                      ref proc (real, vec, mat) void jac,
                      ref proc (ref ivp) void out);
```

```
mode ivp = struct (real x, vec y, proc (real, vec, vec) void f,
                   ref specif specs,
                   ref [ ] int flags, vec sol,
                   ref [ ] vec savevecs,
                   ref [ ] mat savemats,
                   ref lud ludec
                 # possibly hidden fields #);
mode state = struct (real x, vec y);
mode through = struct (state and, ivproc ivp solver);
mode ivproc = proc (ref ivp, state) void;
```

5.3.2. Operators

operator	left operand	right operand	result	prio	remark
until	ref ivp	real	ref ivp	1	performs integration
until	ref ivp	state	ref ivp	1	id.
until	ref ivp	through	ref ivp	1	solves the ivp by ivpsolver of through
method	real	ivproc	through	2	forms a through
method	state	ivproc	through	2	id.
endpoint		real	state		forms a state
integrates	proc(real,vec,vec)void	state	ivp	3	forms a ref ivp
integrates	state	proc(real,vec,vec)void	ivp	3	id.
solve		ref ivp	ref ivp	—	does nothing
spec	ref ivp	ref specif	ref ivp	2	puts a new specif into ivp
*reltol	ref ivp	real	ref ivp	2	puts a new reltol in specif of ivp
*reltol	ref specif	real	ref specif	2	puts a new reltol in specif

*

analogous operators for all other specifications as given in table 5.2.8.

The true integration of the initial value problem is executed by means of a routine of which the declaration should read:

```
proc ivpsolver = (ref ivp ivp, state xend) void;
     (# the integration routine # skip);.
```

5.4. Source text

```
'BEGIN'    # CHAPTER 5,    PWH781005    #

 'MODE'  'VEC'  =  'REF'  [ ]  'REAL';
 'MODE'  'MAT'  =  'REF'  [,]  'REAL';
 'MODE'  'LUD'  =  'STRUCT'  ('MAT' LU ,'REF'[]'INT' PIVS
                              #HIDDEN FIELDS# );

 'MODE'  'SPECIF'  =  'STRUCT'  (
                          'BOOL' STIFF,LINEAR,CONTINUABLE,RESTART,
                          'INT' MAXEVALS, MAXSTEPS,
                          'REAL' RELTOL,ABSTOL,SCALESCAL,
                                 HMIN,HMAX,HSTART,
                          'VEC' SCALEVEC,
                          'MAT' JACOBIAN MAT,
                          'REF' 'PROC'('REAL','VEC','MAT')'VOID' JAC,
                          'REF' 'PROC'('REF' 'IVP')'VOID' OUT );

 'SPECIF' DEFAULTSPECS:= ('TRUE','FALSE','TRUE','FALSE',
                           10000, 10000,
                           1.0E-3, SMALLREAL, 1.0,
                           1.0E-5, 1.0E+3, 1.0E-2,
                          'NIL','NIL','NIL','NIL');

 'MODE'  'IVP'  =  'STRUCT'  ('REAL' X, 'VEC' Y,
                              'PROC'('REAL','VEC','VEC')'VOID' F,
                              'REF' 'SPECIF' SPECS,
                              'VEC' SOL,
                              'REF' [] 'INT' FLAGS,
                              'REF' [] 'VEC' SAVEVECS,
                              'REF' [] 'MAT' SAVEMATS,
                              'REF' 'LUD' LUDEC
                          # POSSIBLE HIDDEN FIELDS #     );

 'MODE'  'STATE'   =  'STRUCT'  ('REAL' X, 'VEC' Y);

 'MODE'  'THROUGH'  =  'STRUCT'  ('STATE' END, 'IVPROC' IVPSOLVER );

 'MODE'  'IVPROC'   =  'PROC'  ('REF' 'IVP', 'REAL')  'VOID';


 'PRIO'  'INTEGRATES' = 3,
         'METHOD'     = 2,
         'SPECS'      = 2,
         'RELTOL'=2 ,'ABSTOL'=2,'SCALE'=2,'JAC'=2,'MONITOR'=2,
         'HMIN'=2,'HMAX'=2,'HSTART'=2,'STIFF'=2,'LINEAR'=2,
         'CONTINUABLE'=2,'RESTART'=2,'MAXEVALS'=2,'MAXSTEPS'= 2,
         'UNTIL'      = 1;

 'OP'  'ENDPOINT'  = ('REAL' XEND )  'STATE'  :(XEND,'NIL');
 'OP'  'SOLVE'     = ('REF'  'IVP' IVP )  'REF'  'IVP'  : IVP;

 'OP'  'INTEGRATES' = ('PROC'('REAL','VEC','VEC')'VOID' F, 'STATE' S )
                      'REF' 'IVP': 'HEAP' 'IVP':=
                     (X'OF'S, Y'OF'S, F,
                      'HEAP' 'SPECIF' := DEFAULTSPECS,
                      'NIL','NIL','NIL','NIL','NIL');
```

```
'OP' 'INTEGRATES' = ('STATE' S,'PROC'('REAL','VEC','VEC')'VOID' F)
                     'REF' 'IVP': F 'INTEGRATES' S;

'OP' 'METHOD' = ('STATE' STATE, 'IVPROC' IVPROC ) 'THROUGH':
                (STATE, IVPROC );

'OP' 'METHOD' = ('REAL' XEND, 'IVPROC' IVPROC ) 'THROUGH':
                'ENDPOINT' XEND 'METHOD' IVPROC ;

'OP' 'SPECS'  = ('REF' 'IVP' IVP, 'SPECIF' SP ) 'REF' 'IVP' :
                ( SPECS'OF'IVP := 'HEAP' 'SPECIF' := SP; IVP );

'OP' 'RELTOL' = ('REF' 'IVP' IVP, 'REAL' RELTOL ) 'REF' 'IVP' :
                ( RELTOL'OF'SPECS'OF'IVP := RELTOL; IVP );

'OP' 'RELTOL' = ('REF' 'SPECIF' SP, 'REAL' RELTOL ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; RELTOL'OF'SPC := RELTOL; SPC);

'OP' 'ABSTOL' = ('REF' 'IVP' IVP, 'REAL' ABSTOL ) 'REF' 'IVP' :
                ( ABSTOL'OF'SPECS'OF'IVP := ABSTOL; IVP );

'OP' 'ABSTOL' = ('REF' 'SPECIF' SP, 'REAL' ABSTOL ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; ABSTOL'OF'SPC := ABSTOL; SPC);

'OP' 'SCALE' = ('REF' 'IVP' IVP, 'REAL' SCALE ) 'REF' 'IVP' :
                ( SCALESCAL'OF'SPECS'OF'IVP := SCALE; IVP );

'OP' 'SCALE' = ('REF' 'SPECIF' SP, 'REAL' SCALE ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; SCALESCAL'OF'SPC := SCALE; SPC);

'OP' 'SCALE' = ('REF' 'IVP' IVP, 'VEC' SCALE ) 'REF' 'IVP' :
                ( SCALEVEC'OF'SPECS'OF'IVP := SCALE; IVP );

'OP' 'SCALE' = ('REF' 'SPECIF' SP, 'VEC' SCALE ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; SCALEVEC'OF'SPC := SCALE; SPC);

'OP' 'JAC' = ('REF' 'IVP' IVP, 'MAT' JAC ) 'REF' 'IVP' :
                ( JACOBIAN MAT'OF'SPECS'OF'IVP := JAC; IVP );

'OP' 'JAC' = ('REF' 'SPECIF' SP, 'MAT' JAC ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; JACOBIAN MAT'OF'SPC := JAC; SPC);

'OP' 'JAC' = ('REF' 'IVP' IVP,
        'REF''PROC'('REAL','VEC','MAT')'VOID' JAC ) 'REF' 'IVP' :
                              ( JAC'OF'SPECS'OF'IVP := JAC; IVP );

'OP' 'JAC' = ('REF' 'SPECIF' SP,
     'REF''PROC'('REAL','VEC','MAT')'VOID' JAC ) 'REF' 'SPECIF' :
             ('HEAP' 'SPECIF' SPC:= SP; JAC'OF'SPC := JAC; SPC);

'OP' 'MONITOR' = ('REF' 'IVP' IVP,
          'REF' 'PROC'('REF''IVP')'VOID' OUT) 'REF' 'IVP' :
                              ( OUT'OF'SPECS'OF'IVP := OUT; IVP );

'OP' 'MONITOR' = ('REF' 'SPECIF' SP,
          'REF' 'PROC'('REF''IVP')'VOID' OUT) 'REF' 'SPECIF' :
```

```
               ('HEAP' 'SPECIF' SPC:= SP; OUT'OF'SPC := OUT; SPC);

'OP' 'HMIN' = ('REF' 'IVP' IVP, 'REAL' HMIN ) 'REF' 'IVP' :
                ( HMIN'OF'SPECS'OF'IVP := HMIN; IVP );

'OP' 'HMIN' = ('REF' 'SPECIF' SP, 'REAL' HMIN ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; HMIN'OF'SPC := HMIN; SPC);

'OP' 'HMAX' = ('REF' 'IVP' IVP, 'REAL' HMAX ) 'REF' 'IVP' :
                ( HMAX'OF'SPECS'OF'IVP := HMAX; IVP );

'OP' 'HMAX' = ('REF' 'SPECIF' SP, 'REAL' HMAX ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; HMAX'OF'SPC := HMAX; SPC);

'OP' 'HSTART' = ('REF' 'IVP' IVP, 'REAL' HSTART ) 'REF' 'IVP' :
                ( HSTART'OF'SPECS'OF'IVP := HSTART; IVP );

'OP' 'HSTART' = ('REF' 'SPECIF' SP, 'REAL' HSTART ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; HSTART'OF'SPC := HSTART; SPC);

'OP' 'STIFF' = ('REF' 'IVP' IVP, 'BOOL' STIFF ) 'REF' 'IVP' :
                ( STIFF'OF'SPECS'OF'IVP := STIFF; IVP );

'OP' 'STIFF' = ('REF' 'SPECIF' SP, 'BOOL' STIFF ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; STIFF'OF'SPC := STIFF; SPC);

'OP' 'LINEAR' = ('REF' 'IVP' IVP, 'BOOL' LINEAR ) 'REF' 'IVP' :
                ( LINEAR'OF'SPECS'OF'IVP := LINEAR; IVP );

'OP' 'LINEAR' = ('REF' 'SPECIF' SP, 'BOOL' LINEAR ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; LINEAR'OF'SPC := LINEAR; SPC);

'OP' 'CONTINUABLE' = ('REF' 'IVP' IVP, 'BOOL' CONT ) 'REF' 'IVP' :
            ( CONTINUABLE'OF'SPECS'OF'IVP := CONT; IVP );

'OP' 'CONTINUABLE' = ('REF' 'SPECIF' SP, 'BOOL' CONT ) 'REF' 'SPECIF':
   ('HEAP' 'SPECIF' SPC:= SP; CONTINUABLE'OF'SPC := CONT; SPC);

'OP' 'RESTART' = ('REF' 'IVP' IVP, 'BOOL' RESTART ) 'REF' 'IVP' :
                ( RESTART'OF'SPECS'OF'IVP := RESTART; IVP );

'OP' 'RESTART' = ('REF' 'SPECIF' SP, 'BOOL' RESTART ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; RESTART'OF'SPC := RESTART; SPC);

'OP' 'MAXEVALS' = ('REF' 'IVP' IVP, 'INT' MAXEVALS ) 'REF' 'IVP' :
                ( MAXEVALS'OF'SPECS'OF'IVP := MAXEVALS; IVP );

'OP' 'MAXEVALS' = ('REF' 'SPECIF' SP, 'INT' MAXEVALS ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; MAXEVALS'OF'SPC := MAXEVALS; SPC);

'OP' 'MAXSTEPS' = ('REF' 'IVP' IVP, 'INT' MAXSTEPS ) 'REF' 'IVP' :
                ( MAXSTEPS'OF'SPECS'OF'IVP := MAXSTEPS; IVP );

'OP' 'MAXSTEPS' = ('REF' 'SPECIF' SP, 'INT' MAXSTEPS ) 'REF' 'SPECIF':
     ('HEAP' 'SPECIF' SPC:= SP; MAXSTEPS'OF'SPC := MAXSTEPS; SPC);
```

```
'OP' 'UNTIL' = ('REF' 'IVP' IVP, 'THROUGH' THROUGH ) 'REF' 'IVP':
               ( (IVPSOLVER 'OF' THROUGH)(IVP, X'OF'END'OF'THROUGH);
                 'VEC' SOLUT = Y'OF'END'OF'THROUGH ;
                 (SOLUT:/=:'VEC'('NIL') ! SOLUT:= 'COPY'(SOL'OF'IVP));
                 IVP );

'OP' 'UNTIL' = ('REF' 'IVP' IVP, 'STATE' STATE ) 'REF' 'IVP':
                IVP 'UNTIL' STATE 'METHOD' IVPSOLVER;

'OP' 'UNTIL' = ('REF' 'IVP' IVP, 'REAL' XEND ) 'REF' 'IVP':
                 IVP 'UNTIL' 'ENDPOINT' XEND;


'PROC' IVPSOLVER = ('REF' 'IVP' RIVP , 'REAL' XEND) 'VOID':
  'SKIP' ;

  'SKIP'
'END'
```

REFERENCES

H.J. BOS & D.T. WINTER, *AFLINK: A new ALGOL 68 - FORTRAN interface*, Report NN17/78, Mathematical Centre, Amsterdam, 1978.

S.G. VAN DER MEULEN & M. VELDHORST, *TORRIX, a programming system for operations on vectors and matrices over arbitrary fields and of variable size,* Volume 1, Mathematical Centre Tract 86, Mathematisch Centrum, Amsterdam, 1978.

A. VAN WYNGAARDEN et al. (eds.), *Revised report on the algorithmic language ALGOL 68,* Mathematisch Centre Tract 50, Mathematisch Centrum, Amsterdam, 1978.

NUMAL, A library of numerical procedures in ALGOL 68, 8 volumes, 2nd revision, May 1977, Mathematisch Centrum, Amsterdam, 1977.

NAG, Library Manual (1974), Numerical Algorithms Group, Oxford.

IMSL, Library 3, Reference Manual (1974), International Mathematical and Statistical Libraries, Inc., Houston, Texas.