DISPMODULE User Manual

1. INTRODUCTION

DISPMODULE is a standard Fortran 95 module for quick and easy displaying ("pretty-printing") of numbers, vectors or matrices using default or specified format. It can be useful for debugging, purposes, for preliminary display of numerical results, and even for final display of such results in cases when carefully formatted tables are not needed. It is comparable to the automatic matrix printing of Matlab, S and R, but offers substantially more control over the format used.

The module can handle the standard Fortran data types integer, single precision, double precision, complex, logical and character. Integer, real, complex and logical data of other than default kind are supported with add-on modules. The module contains the following public procedures:

`Subroutine DISP`	The main procedure used for displaying items
`Subroutine DISP_SET`	Used to change default settings for `DISP`
`Subroutine DISP_SET_FACTORY`	Restores `DISP`-settings to original (factory) default
`Function DISP_GET`	Returns a structure with current `DISP`-settings
`Function TOSTRING`	Returns a string representation of a scalar or vector
`Subroutine TOSTRING_SET`	Used to change default settings for `TOSTRING`
`Subroutine TOSTRING_SET_FACTORY`	Restores `TOSTRING`-settings to original default

In addition the module defines a public derived type, `DISP_SETTINGS`, used for saving and restoring settings for `DISP`. The procedures `DISP` and `TOSTRING` have a generic interface and optional arguments, so the same subroutine / function name, is used to display items of different data types and ranks, with or without labels, and using default or specified format. Similarly `DISP_SET` is generic and can be used both to change individual settings and to restore previously saved settings.

The most basic calling syntax for displaying is `CALL DISP(`*expression*`)` which will display the expression with default format. The format may be specified with `CALL DISP(`*expression*`,` *edit-descriptor*`)`, and `CALL DISP(`*title*`,` *expression*`)` will label the displayed item with a title. Examples are `CALL DISP(A)`, `CALL DISP(A,'F9.3')`, `CALL DISP('A=',A)` and `CALL DISP('A=',A,'F9.3')`, the last one specifying both title and format. If $a_{ij} = \exp(i + j - 1)$, $i, j = 1,\ldots,4$, then `CALL DISP('A = ', A)` writes out:

```
A =  2.72    7.39   20.09    54.60
     7.39   20.09   54.60   148.41
    20.09   54.60  148.41   403.43
    54.60  148.41  403.43  1096.63
```

and if $b_{ij} = \exp(ij)$ the result of `CALL DISP(B)` is:

```
2.71828E+0  7.38906E+0  2.00855E+1  5.45981E+1
7.38906E+0  5.45981E+1  4.03429E+2  2.98096E+3
2.00855E+1  4.03429E+2  8.10308E+3  1.62755E+5
5.45981E+1  2.98096E+3  1.62755E+5  8.88611E+6.
```

It is also possible to number the rows and columns: `CALL DISP(A, STYLE='NUMBER')` will give:

```
      1       2       3        4
1   2.72    7.39   20.09    54.60
2   7.39   20.09   54.60   148.41
3  20.09   54.60  148.41   403.43
4  54.60  148.41  403.43  1096.63.
```

The selection between F and E editing depends on the size of the largest displayed element as discussed in section 3.2 below. Among the settings that may be controlled is the spacing between columns, the number of significant digits, the placement of the label, and the file unit where the output goes. Items can in addition be displayed side by side, for example:

```
CALL DISP('X = ', X, ADVANCE='NO')
CALL DISP('Y = ', Y)
```

which might output:

```
X = 7  8  3   Y = 11
    4  0  2        2
    1  3  6        7
```

Complex numbers are formatted as illustrated by:

```
complex C(3,3)
forall(i=1:3, k=1:3) C(i,k)=log(cmplx(-i*k))**k
call disp('C = ', C, 'F0.3')
```

which will display

```
C = 0.000 + 3.142i   -9.389 +  4.355i   -31.203 - 19.631i
    0.693 + 3.142i   -7.948 +  8.710i   -47.300 -  0.749i
    1.099 + 3.142i   -6.659 + 11.258i   -54.449 + 14.495i
```

Infinite and *not-a-number* real values are supported and displayed as `NaN`, `+Inf` or `–Inf`.

The remaining sections in this user manual contain detailed information on using the module. Section 2 discusses the basics of using the module, including use statements, compiling and linking, and add-on modules supporting non-default kinds of data. Section 3 gives a detailed description of the generic subroutine `DISP`. All the possible arguments are listed and the purpose of each one described. Section 4 describes how to change various settings that control how items are displayed with `DISP`. Section 5 describes the function `TOSTRING` which may be used to change numbers to strings. Finally testing of the module is discussed in section 6.

2. USAGE

2.1 Overview of modules

The major part of the package is the module `DISPMODULE` in the file `dispmodule.f90`. This module should be referenced with a use-statement, `USE DISPMODULE`, and the file should be compiled and linked with other files. To display non-default kinds of data the add-on modules described in section 2.3 below must also be used. The file `dispmodule.f90` actually begins with two auxiliary modules, `PUTSTRMODULE` and `DISPMODULE_UTIL`. The first one contains two dummy subroutines, `PUTSTR` and `PUTNL`, which do nothing, but must be incorporated to avoid an “undefined symbol” link error. In addition it defines the named constant (parameter) `DEFAULT_UNIT` = −3, which makes the asterisk unit (usually the screen) the default to display on. The second auxiliary module has utility procedures which are not accessible to the user.

Alternatively the user can write his own `PUTSTRMODULE` as described in section 3.4 below. An example is near the beginning of `dispmodule.f90` (commented out). It may be used (by commenting it in, and commenting the default module out) to allow Matlab mex files to display in the Matlab command window.

2.2 An example program

Following is a short example program that uses the package:

```
PROGRAM EXAMPLE
  USE DISPMODULE
  REAL :: A(3) = (/ 1.2345, 2.3456, 3.4567 /)
  CALL DISP('A = ', A, SEP=', ', ORIENT = 'ROW')
END PROGRAM EXAMPLE
```

If this program is in a file `example.f90` and the Gnu Fortran compiler is used with Unix, the following commands will accomplish compiling, linking and executing (the $-symbols are operating system prompts):

```
$ gfortran -c dispmodule.f90
$ gfortran -o example example.f90 dispmodule.o
$ ./example
```

Alternatively the compiled modules may be placed in a library that is included when linking:

```
$ gfortran -c dispmodule.f90
$ ar -r libdispmodule.a dispmodule.o
$ gfortran -o example example.f90 -L. -ldispmodule
$ ./example
```

The program should write out “`A = 1.23450, 2.34560, 3.45670`”. The package also contains a little longer example program (in `dispdemo.f90`) with a makefile. On Unix with gfortran, the command:

```
$ make demo compiler=gfortran
```

should compile, link, and run `dispdemo`. The makefile’s default compiler name is `f95`.

2.3 Displaying data of non-default kind

The Fortran standard only guarantees the support of one kind of integers, logicals and characters, and two kinds of real and complex numbers (the standard mentions double precision complex numbers, and reading between the lines their support is almost deducible, though not directly guaranteed). The size of default kind integers, logicals and single precision reals is with most (if not all) computers / compilers either 4 or 8 bytes, and many (if not all) computers / compilers support other kinds of integers covering the range of byte lengths 1, 2, 4 and 8. The support of 1 byte logicals is also common, and some computers / compilers have quadruple precision reals.

Support of these “non-standard” kinds is provided by the add-on modules `DISP_I1MOD`, `DISP_I2MOD`, `DISP_I4MOD`, `DISP_I8MOD`, `DISP_L1MOD` and `DISP_R16MOD`. These assume that the following hold:

`SELECTED_INT_KIND(2)`	→ 1 byte integers
`SELECTED_INT_KIND(4)`	→ 2 byte integers
`SELECTED_INT_KIND(9)`	→ 4 byte integers
`SELECTED_INT_KIND(18)`	→ 8 byte integers
`SELECTED_REAL_KIND(25)`	→ 16 byte reals (quad precision)
`LOGICAL(1)`	→ 1 byte logicals

If this is not the case, or support for other kinds is wanted, then the files must be edited to obtain what is desired. To use any of these modules, reference them with use-statements: `USE DISP_xxMOD`, compile them and link with the program. With a little editing effort it is also possible to combine (a subset of) them with `DISPMODULE` in a single module − the details are left to the user.

3. SUBROUTINE DISP

This is the principal subroutine of the package. It has various control arguments that specify the exact format of the output. Most of these may also be used as arguments of the subroutine `DISP_SET`. When used with `DISP`, a control argument affects only the item being displayed with the current call, but when used with `DISP_SET`, the default settings for subsequent `DISP` calls are affected. The default values for individual arguments given below are used unless they have been changed by a call to `DISP_SET`. All character arguments should be of type default character. An overview of the arguments is in table 1, and further details are in sections 3.1−3.5.

3.1 Simple calls

```
CALL DISP
CALL DISP(X)
CALL DISP(TITLE, X)
CALL DISP(X, FMT)
CALL DISP(TITLE, X, FMT)
```

Table I. Arguments of subroutine DISP (all are optional).

Argument	Type	Length	Kind	Rank	Possible values	Default	Purpose
TITLE	Character	*	Default	0	Any	Empty	Label to use for X
X	Integer Real Complex Logical Character		Any	0, 1 or 2	Any		Item to display
FMT	Character	*	Default	0	Valid edit descriptor	Depends on X and DIGMAX	Edit descriptor for an element of X (real part if X is complex)
FMT_IMAG	Character	*	Default	0	Valid edit descriptor	Depends on X and DIGMAX	Edit descriptor for imaginary parts of elements of X
ADVANCE	Character	≤ 6	Default	0	NO, YES or DOUBLE	YES	Stay on same line, advance to next line, to next line but one?
DIGMAX	Integer		Default	0	1, 2,… , 90	6	Number of significant digits to show for X-element of largest absolute magnitude
LBOUND	Integer		Default	1	Any	[1] or [1,1]	Numbers of first row / column of X
ORIENT	Character	3	Default	0	ROW or COL	COL	Display row vector or column vector?
SEP	Character	≤ 9	Default	0	Any		String used to separate matrix columns
STYLE	Character	≤ 9	Default	0	LEFT, ABOVE, PAD, UNDERLINE or NUMBER — can be combined, e.g. PAD & NUMBER	LEFT	Placement and form of title; should rows / columns be numbered?
TRIM	Character	≤ 4	Default	0	YES, NO or AUTO	AUTO	Trim blanks from left of X? (AUTO means trim if FMT is absent)
UNIT	Integer		Default	0	≥ -3	ASTERISK_UNIT	External file unit to send output to
ZEROAS	Character	≤ 9	Default	0	Any		String to display instead of zeros

The first call advances to the next line, and the other calls display `X` on the default unit (the unit may be changed with the `UNIT` argument). The default `putstrmodule` (see section 2) sets the asterisk unit (usually the screen) to be default. The purpose of individual arguments is as follows:

`TITLE` Provides a label for `X`. The label prefixes `X` by default but this may be changed with the `STYLE` argument (see examples in section 3.2). When `X` is absent `TITLE` must also be absent.

`X` The item to be displayed. `X` may be scalar, vector or matrix (i.e. of rank ≤ 2) and the following kinds of data are supported:

default integer
default real (or single precision, `real(kind(1.0))`)
double precision real (or `real(kind(1d0))`)
default complex (or `complex(kind(1.0))`)
double precision complex (or `complex(kind(1d0))`)
default logical
default character

With the add-on modules described in section 2.3 other kinds may be displayed. Matrices are displayed in traditional mathematical order, so the rows displayed are `X(1,:)`, `X(2,:)` etc. Vectors are by default displayed as column vectors (but a row orientation may be specified with the `ORIENT` argument). An SS edit descriptor is applied automatically so positive elements are not prefixed with a + sign (the Fortran standard makes outputting a + sign optional).

FMT When present, `FMT` should contain an edit descriptor that will be used to format each element of `X` (or the real parts of `X` in case `X` is complex and `FMT_IMAG` is present; see below). The possible edit descriptors are:

F*w*.*d*, D*w*.*d*, E*w*.*d*E*e*, EN*w*.*d*E*e*, ES*w*.*d*E*e*: real data (the E*e* suffixes are optional)
I*w*, B*w*, O*w*, Z*w*: integer data (all may be suffixed with .*m*)
L*w*: logical data
A, A*w*: character data
G*w*.*d*, G*w*.*d*E*e*: any data

Example calls for numeric `X` are `CALL DISP(X,'ES11.4')` and `CALL DISP('X=',X,'F8.4')`. If `X` is a scalar string (i.e. of rank 0) and `TITLE` is absent `FMT` must be specified with a keyword (otherwise the call is taken to have `TITLE` and `X`): `CALL DISP('str',FMT='A4')` displays “ `str`” but `CALL DISP('str','A4')` displays “`strA4`”).

If `FMT` is absent, each element of `X` is formatted with a default edit descriptor. When `X` is of type logical the default is L1 and when it is of type character the default is A (which is equivalent to A*w* where *w* = `LEN(X)`). For integer data the default is I*w* where *w* is exactly big enough to accommodate both the largest positive and the largest negative values in `X`. For real and complex data the default also depends on the largest absolute values in `X`, as detailed in the `DIGMAX`-paragraph in section 3.2. The format used for complex numbers is demonstrated in the introduction above.

3.2 Call with complete list of arguments

```
CALL DISP (TITLE, X, FMT, FMT_IMAG, ADVANCE, DIGMAX, LBOUND, ORIENT, SEP,
           STYLE, TRIM, UNIT, ZEROAS)
```

All dummy arguments are optional and some of them are incompatible with some data types of `X`. The arguments control how `X` is displayed, as described in section 3.1 and below. For the character arguments `ADVANCE` and `ORIENT` the case of letters is ignored (so e.g. `ADVANCE = 'yes'` and `ADVANCE = 'YES'` are equivalent). Normally argument association for arguments after `FMT` (or `FMT_IMAG`) will be realized with argument keywords, e.g. `CALL DISP('X=', X, DIGMAX=3, ORIENT='row')`. When `X` is a scalar string `FMT` must also be associated with keyword, as mentioned in section 3.1. The most useful application of calling `DISP` with `X` absent is to advance to the next line or display an empty line. For this purpose, the only relevant arguments are `UNIT`, and `ADVANCE` with the value `'yes'` or `'double'`.

FMT_IMAG = *edit-descriptor-imag* An edit descriptor for imaginary parts of complex `X`. The statement `CALL DISP((1.31,2.47),'F0.1','F0.2')` will display “1.3 + 2.47i”. If `FMT_IMAG` is absent and `FMT` is present then both real and imaginary parts are edited with `FMT`. If both are absent, separate defaults are used, as explained in the `DIGMAX`-paragraph below. `FMT_IMAG` must be absent if `X` is not complex.

ADVANCE = *adv* The value for *adv* may be `'yes'`, `'no'` or `'double'`. If the value is `'yes'` then `X` is written out immediately, if it is `'double'` then `X` is written out followed by an empty line (thus giving double spacing), and if it is `'no'` then `X` is not written out until the next DISP call on the same unit with advancing turned on (either by default, via a call to `DISP_SET`, or via the `ADVANCE` keyword). When this occurs, all the items displayed with `DISP` since the last output occurred on the unit are written out side by side, separated by three spaces unless a different separation has been specified via the `MATSEP` argument of `DISP_SET`. Default value of `ADVANCE` is `'yes'`.

DIGMAX = *n* Controls the format used for real and complex data in the absence of FMT. For real items the format is chosen so that the displayed number of largest absolute magnitude (say $x_{\max}$) has n significant decimal digits. If $0.1 \le |x_{\max}| < 10^n$ an F edit descriptor is used, otherwise an E edit descriptor. For complex items these rules are applied separately to the real parts and imaginary parts, and thus two different formats are used. When X is not of real or complex type the argument DIGMAX is ignored. When DIGMAX is present FMT should be absent. The default is $n = 6$.

LBOUND = *lbound* This argument is a default integer vector with the numbers of the first row / column to show when displaying with numbered style. When calling subroutines in Fortran, only the shape of matrix arguments is passed with the arguments, but matrix lower bounds are assumed to be 1 unless declared explicitly in the routine. To compensate for this deficiency *lbound* may be set to the declared lower bound(s) of X. To take an example, let $a_{ij} = \exp(i + j - 1)$ as in section 1, but let A be declared with REAL::A(0:3,0:3). Then CALL DISP(A, STYLE = 'NUMBER', LBOUND = LBOUND(A)) will display:

```
        0        1         2         3
0   1.000    2.718     7.389    20.086
1   2.718    7.389    20.086    54.598
2   7.389   20.086    54.598   148.413
3  20.086   54.598   148.413   403.429.
```

In fact the call may be shortened to CALL DISP(A, LBOUND = LBOUND(A)) because numbering is default when LBOUND is present.

ORIENT = *ori* This argument can only be used when X is a vector (i.e. has rank 1). If *ori* is 'col' (the default) a column vector is displayed, and if *ori* is 'row' a row vector results.

SEP = *sep* Specifies a string which is written out between columns of displayed matrices. If X has rows (−1, 3) and (5, 10) and *sep* is ', ' then the output will be:

```
-1,  5
 5, 10
```

The length of the string must be at most 9. Default is ' ' (character string with two spaces).

STYLE = *style* There are five possible styles:

'left'	Title is immediately to the left of the first line of the displayed item.
'above'	Title is centred immediately above the item.
'pad'	Title is centred above the item, padded with hyphens (-).
'underline'	Title is centred above the item, underlined with hyphens.
'number'	Each matrix or vector row and / or column is numbered.

Any of the four title position styles can also be combined with the number style by specifying for example STYLE = 'pad & number'. Any character except space may be used instead of hyphen by prefixing it to the style. STYLE = '*underline' will thus underline the title with asterisks. Both row and column numbers appear for numbered matrices, but for vectors only row numbers appear (or column numbers when ORIENT is 'col'). The five styles are illustrated below, accompanied by an example of combined padded title and numbering.

```
Matr = 1.2   4.2       Matr      ---Matr--       Matr            1     2    ____Matr____
       5.6  18.3    1.2   4.2    1.2   4.2    ---------     1  1.2   4.2         1     2
                    5.6  18.3    5.6  18.3    1.2   4.2     2  5.6  18.3    1  1.2   4.2
                                              5.6  18.3                     2  5.6  18.3
```

The default value of *style* is 'left' if LBOUND is absent, 'number' if it is present, and 'left & number' if both TITLE and LBOUND are present.

TRIM = *trim* This argument can take three values, `'YES'`, `'NO'` and `'AUTO'`. When `YES` is specified, each column of displayed items is trimmed from the left, with `'NO'` the items are not trimmed and if *trim* is `'AUTO'` the items are trimmed when `FMT` is absent but not when it is present. In the following example, `X` and `U` are displayed with *trim* = `'yes'`, but `Y` and `V` with *trim* = `'no'`. In all cases the edit descriptor is the default (`I4`). The default is `TRIM` = `'AUTO'`.

```
----X----   -------Y------   -----U-----   -------V------
1  2    4      1    2    3   333 22 4444     333   22 4444
2 22   34      2   22   34
3 32 1234      3   32 1234
```

One application of trimming is to display matrices with a fixed number of fractional digits but variable effective field width. Then F*w*.*d* editing with *w* big enough is accompanied by `TRIM = 'yes'`. An example is the following display of a matrix with (*i*, *k*) element $\exp(k^i)$ using F20.2 and `'yes'`:

```
  power exponentials
2.72   7.39    20.09
2.72  54.60  8103.08
```

Similar output may be obtained using I and F edit descriptors with *w* = 0 as discussed in section 3.5. Apart from I and F edited displays, it is possible to trim A-edited displays as well as E-edited displays with some negative elements, but the first column all positive:

```
With TRIM='yes': X=1.2e+5 -4.1e-2    With TRIM='no':  X= 1.2e+5 -4.1e-2
     2.3e-3  8.6e+1          2.3e-3  8.6e+1
```

UNIT = *external-file-unit* The unit which the output is sent to. There are three special units, which may be referred to either with constants or parameters (named constants) as follows:

Constant	Parameter	Preconnected unit
−3	ASTERISK_UNIT	The asterisk unit (often the screen)
−2	PUTSTR_UNIT	The subroutines PUTSTR and PUTNL
−1	NULL_UNIT	Null device (all output to this is discarded)

These units are further described in sections 3.3 and 3.4. Other unit numbers correspond to external files that should have been connected with open-statements. The default unit depends on the named constant `DEFAULT_UNIT`, defined in `PUTSTRMODULE`. The default `PUTSTRMODULE` sets it to −3 (see sections 2 and 3.4).

ZEROAS = *zerostring* Supported for integer and real items (**not** complex). Any element that compares equal to 0 will be displayed as *zerostring*. If, for example, `A` is a 4 by 4 upper triangular matrix with $a_{ij} = 1/\max(0, j-i+1)$ then `call disp('A = ', A, 'F0.3', zeroas = '0', advance = 'no')` and `call disp('B = ', A, 'F0.3', zeroas='.')` will display:

```
A = 1.000  0.500  0.333  0.250   B = 1.000  0.500  0.333  0.250
        0  1.000  0.500  0.333         .    1.000  0.500  0.333
        0      0  1.000  0.500         .      .    1.000  0.500
        0      0      0  1.000         .      .      .    1.000
```

Notice that when *zerostring* contains a decimal point it is lined up with other decimal points in the column. If *zerostring* has length 0, the default behaviour of not treating zeros specially is re-established, in case an earlier `DISP_SET` call has been used to set `ZEROAS`.

3.3 ASTERISK_UNIT and NULL_UNIT

As already mentioned in section 3.2 there are three special units, `ASTERISK_UNIT` = −3, `PUTSTR_UNIT` = −2 and `NULL_UNIT` = −1. These public named constants (parameters) are defined by `DISPMODULE`.

Selecting `ASTERISK_UNIT` channels all output to the unit that `WRITE(*,...)` statements use. The `ISO_FORTRAN_ENV` intrinsic module of Fortran 2003 defines the named constant `OUTPUT_UNIT` and this may be used instead, unless its value is set to −2 by the compiler (which would clash with `PUTSTR_UNIT`).

Selecting `NULL_UNIT` causes all output via `DISP` to be discarded. This feature makes it simple to turn the output on and off, which may be useful for debugging and testing purposes. If `UNIT = U` is specified in all `DISP`-calls, it is enough to change the value of `U` to −1 to turn off output.

3.4 PUTSTR_UNIT: Output with user written subroutines

One of the purposes of the `PUTSTR_UNIT` is to make displaying possible in situations where ordinary print- and write-statements do not work. This is for example the case in Matlab mex-files (in fact the execution of a write statement on the asterisk unit crashes Matlab). To use the `PUTSTR_UNIT` it is necessary to write two subroutines with interfaces:

```
SUBROUTINE PUTSTR(S)
CHARACTER(*), INTENT(IN) :: S

SUBROUTINE PUTNL()
```

The first of these should output the string `S`, and the second one should advance output to the next line. These subroutines should be placed in a module `PUTSTRMODULE` as explained in section 2. The module should also define a named constant `DEFAULT_UNIT`, which could be set to −2 to make the `PUTSTR_UNIT` default. An example that works with g95 and Matlab mex-files is:

```
MODULE PUTSTRMODULE
  integer, parameter :: default_unit = -2

CONTAINS
  subroutine putstr(s)
    character(*), intent(in) :: s
    call mexprintf(s//char(0))
  end subroutine putstr

  subroutine putnl()
    call mexprintf(char(10)//char(0))
  end subroutine putnl

END MODULE PUTSTRMODULE
```

At the beginning of the file `dispmodule.f90` there is a slightly longer version which works with both g95 and gfortran. Testing this module is discussed in section 6.2 below.

3.5 Using *w*=0 editing

The Fortran standard stipulates that writing a single element with I0 editing results in the smallest field width that accommodates the value, and the same applies to B0, O0, Z0 and F0.*d* editing. With `DISP`, the width of a displayed column will be the width of the widest field in the column, and each element is right-adjusted in the column. This gives exactly the same output as using `TRIM='yes'` and a specified field width bigger than the largest occurring. Note that with F0.*d* editing, there is no limit on the width of a column, but with F*w*.*d* and `TRIM='yes'` any element wider than *w* will be displayed as *w* asterisks:

```
------------------F0.2------------------ -----F13.2, TRIM='yes'----
14.28  142857142857142857142857.14  0.47 14.28  *************  0.47
 1.42                1414213562.37  0.69  1.42  1414213562.37  0.69
```

3.6 Not-a-number and infinite values

If the compiler supports *not-a-number* and infinite values as defined by the IEEE exceptional values of Fortran 2003, these are displayed as NaN, +Inf or −Inf. A *not-a-number* value X is identified as being not equal to itself, and an infinite value is either greater than HUGE(X) or smaller than –HUGE(X). On all the compilers tried the sequence BIG=1E20; CALL DISP(EXP(BIG)) displays +Inf, and the program segment:

```
REAL :: Z = 0, BIG = 1E20
CALL DISP((/Z, Z/Z, BIG, -EXP(BIG)/))
```

displays

```
0.00000E+00
        NaN
1.00000E+20
       -Inf
```

4. CONTROLLING DEFAULT SETTINGS − DISP_SET AND DISP_GET

The subroutine DISP_SET may be used to change default values of all the arguments of DISP except TITLE, X, FMT and LBOUND. In addition the default separator between items that are displayed side-by-side (using ADVANCE='no') may be changed with the MATSEP argument.

4.1 The derived type DISP_SETTINGS

DISPMODULE contains the following definition of the data type DISP_SETTINGS.

```
TYPE DISP_SETTINGS
  character(3)  :: advance   = 'YES'
  character(9)  :: matsep    = '   '
  character(3)  :: orient    = 'COL'
  character(9)  :: sep       = '  '
  character(19) :: style     = 'LEFT'
  character(4)  :: trim      = 'AUTO'
  character(9)  :: zeroas    = ''
  integer       :: digmax    = 6
  integer       :: matseplen = 3
  integer       :: seplen    = 2
  integer       :: unit      = -3
  integer       :: zaslen    = 0
END TYPE DISP_SETTINGS
```

Structures of type DISP_SETTINGS may be used to save and later restore format control settings of DISP. As shown, new variables of this type will automatically have default values for all components.

4.2 Calling syntax for DISP_SET

There are two ways to call DISP_SET:

```
CALL DISP_SET(SETTINGS)
CALL DISP_SET(ADVANCE, DIGMAX, MATSEP, ORIENT, SEP, STYLE, UNIT, ZEROAS)
```

Both calls change the default format control used in subsequent calls to DISP. In the first call, SETTINGS is of type DISP_SETTINGS and the default values for all arguments is changed. In the second call all the arguments are optional. If an argument is absent the corresponding default setting is not changed. An example call is

```
CALL DISP_SET(STYLE = 'pad', SEP = ' ').
```

The effect is that titles will be written padded above matrices, and matrix column will be separated by one blank. The type and purpose of all the arguments except MATSEP has been described in section 3.2.

MATSEP = *ms* Specifies a character string of length ≤ 9 that is written out between items (matrices) when they are displayed side-by-side. An example is:

```
CALL DISP(X, ADVANCE='NO')
CALL DISP(Y, ADVANCE='NO')
CALL DISP_SET(MATSEP=' | ')
CALL DISP(Z, ADVANCE='YES')
```

The output from these calls might be:

```
12.2 |  1.3 | 1
 9.6 | 13.0 | 3
-2.0 |  4.0 | 4
```

Note that `MATSEP` affects the separation of all items that have been placed in the output queue of the unit being displayed on.

4.3 The function DISP_GET

The argumentless function `DISP_GET` returns the current default settings in a structure of type `DISP_SETTINGS`. If a subroutine changes the default settings with `DISP_SET` it is possible to save the settings that are in effect when the routine is entered, and restore these settings before returning from the routine. An example is:

```
subroutine disp_xy(x,y)
  use dispmodule
  real x(:,:), y(:,:)
  type(disp_settings) ds
  ds = disp_get()
  call disp_set(digmax=4, sep=',')
  call disp('x=',x)
  call disp('y=',y)
  call disp_set(ds)
end
```

5. NUMBERS TO STRINGS

5.1 Introduction

Many programming languages have built-in functions that change numbers to strings. It is for instance possible with Matlab to write

```
s = ['The square of ' num2str(x) ' is ' num2str(x*x)]; disp(s)
```

and in Java one may write

```
String s = "The square of " + Float.toString(x) + " is " Float.toString(x*x);
System.out.println(s)
```

(or in fact even shorter: `System.out.println("The square of " + x + " is " + x*x);`). Both program fragments will display “The square of 1.5 is 2.25”. It is possible to achieve a similar effect in Fortran using internal files and list-directed output:

```
character(100) s
real :: x = 1.5
write(s, *) 'The square of', x, 'is', x*x
print *, trim(s)
```

but this is cumbersome, and also there is the disadvantage that the result is compiler-dependent.

DISPMODULE has a function, TOSTRING, which overcomes this disadvantage and offers additional flexibility. With x = 1.5 the following statement will produce the same output as Matlab and Java give:

```
call disp('The square of '//tostring(x)//' is '//tostring(x*x))
```

Because Fortran 95 does not have variable length strings, using TOSTRING lacks some of the possibilities of the Java and Matlab functions. However, TOSTRING is compatible with the ISO_VARYING_STRING module, and if this is available one may for example obtain the same output as above with:

```
use iso_varying_string
type(varying_string) :: xs, x2s
...
x = 1.5
xs = tostring(x)
x2s = tostring(x**2)
print *, 'The square of'//xs//' is '//x2s
```

TOSTRING accepts integer, logical or real scalars or vectors. The subroutine TOSTRING_SET may be used to change settings for TOSTRING.

5.2 The function TOSTRING

Apart from the item to be turned into a string, an edit descriptor to use can optionally be supplied as the second argument to TOSTRING. The two ways to invoke TOSTRING are:

```
TOSTRING(X)
TOSTRING(X, FMT)
```

These invocations return a character string representing the value of the argument X. When X is a vector individual elements are separated by a string, with the original (or factory) default value ", ". By (original) default, G editing is used to convert real numbers, I editing integers, and blanks are trimmed from (each element of) X, both from the left and the right. In addition trailing zeroes are trimmed from the fractional part of real X-elements, as well as a trailing decimal point. The separating string, trimming behaviour, and default editing may be changed by calling TOSTRING_SET

X The item to be changed to a string. X may be a scalar or a vector (i.e. of rank 0 or 1) and of one of the following kinds:

default integer
default real (i.e. real(1.0), single precision)
double precision real (i.e. real(1d0))
default logical

FMT Character string with an edit descriptor used to format each element of X. The possible edit descriptors are given in section 3.1, except that A and A*w* can of course not be used. When FMT is absent, a default edit descriptor is used. The default may be set by calling TOSTRING_SET but the original (or factory) defaults are I0 for integers, L1 for logicals and 1PG12.5 for reals.

5.3 The subroutines TOSTRING_SET and TOSTRING_SET_FACTORY

The subroutine TOSTRING_SET has five arguments, all of which are optional. Argument association will normally be realized using argument keywords, e.g. CALL TOSTRING_SET(SEP='; '). The examples in section 5.4 clarify how to use this subroutine. The five arguments are:

SEP Character string used to separate elements of displayed vectors. Original default value is ', '.

RFMT Character string containing default edit descriptor to use to display real items. The original default value is `'1PG12.5'`

IFMT Character string containing default edit descriptor to use to display integer items. The original default value is `'I0'`.

TRIMB Controls whether leading and trailing blanks are trimmed from individual displayed elements. Possible values are `'YES'` (to trim blanks) and `'NO'` (for no trimming). Default is `'YES'`.

TRIMZ Controls whether trailing zeros are trimmed from the fractional part of displayed items. Possible values are `'NONE'` (for no zero trimming), `'G'` (to trim fractional trailing zeros only when G editing is used), and `'ALL'` (to trim zeros with all types of editing). Trailing decimal points are also removed when zero-trimming is active. Default value is `'G'`.

The subroutine `TOSTRING_SET_FACTORY` (which has no arguments) may be called to restore all settings of `TOSTRING` to the original default values (the factory defaults): `SEP=', '`, `RFMT='1PG12.5'`, `IFMT= 'I0'`, `TRIMB='YES'` and `TRIMZ='G'`.

5.4 TOSTRING examples

When the original (factory) defaults are in effect, the result of invoking `TOSTRING` will usually be as follows.

```
Invocation                         Returned string
tostring(atan(1.0))                '0.785398'
tostring(exp([-3.,-1.,0.,1.]))     '4.97871E-02, 0.36788, 1, 2.7183'
tostring(real([(i,i=1,5)])**8)     '1, 256, 6561, 65536, 3.90625E+05'
tostring([1.23456,1.2300,1.23456e6])  '1.2346, 1.23, 1.23456E+06'
tostring(real([(i,i=1,5)])**8,'f0.1') '1.0, 256.0, 6561.0, 65536.0, 390625.0'
tostring(real([(i,i=1,5)])**8,'f6.1') '1.0, 256.0, 6561.0, ******, ******'
tostring([1200000.,-1.2e-9])       '1.2E+06, -1.2E-09'
!
tostring(1.200d103)                '1.2+103'
tostring([1.1d0,2.2d10,3.3d20])     '1.1E+00, 2.2E+10, 3.3E+20'
!
tostring(-77)                      '-77'
tostring([(i,i=-3,3)]**11)         '-177147, -2048, -1, 0, 1, 2048, 177147'
tostring([(i,i=-3,3)]**11, 'i7')   '-177147, -2048, -1, 0, 1, 2048, 177147'
tostring([(i,i=-3,3)]**11, 'i4')   '****, ****, -1, 0, 1, 2048, ****'
!
tostring((1,3)/(4,2))              '0.5 + 0.5i'
tostring(cmplx((/-1,-2/))**0.25)   '0.70711 + 0.70711i, 0.8409 + 0.8409i'
!
tostring([.true., .false., .false.])  'T, F, F'
tostring(.true., 'L2')             'T'
```

The returned strings may be slightly different from the ones shown, because some compilers (at least some versions of g95) will produce one more decimal place in a few cases, and because the Fortran standard allows G editing to give exponent fields in the form ±0dd instead of ±Edd. The examples make use of brackets to construct vector constants (a Fortran 2003 feature). If the compiler being used does not support this, (/ and /) must be used instead. Notice that trimming is on by default so there is not much purpose in specifying the format for integers and logicals. Notice also that (usually) 5 significant digits are displayed when the default G editing results in F edited output, but 6 digits for the numbers of small or large magnitude, displayed with E editing. This discrepancy is present in the Fortran standard; the presence of the scale factor 1P in the edit descriptor increases the number of displayed significant digits.

Examples of using `TOSTRING_SET` follow (again the returned string may be slightly different).

```
Invocation                             Returned string
call tostring_set(sep=';')
tostring([1,2,30])                     '1;2;30'
!
call tostring_set(trimb='NO')
tostring(real([(i,i=1,5)])**8,'f6.1')  '   1.0; 256.0;6561.0;******;******'
tostring([1,2,30],'i3')                '  1;  2; 30'
tostring([(i,i=-3,3)]**11, 'i4')       '****;****;  -1;   0;   1;2048;****'
tostring([1,2,30],'i0')                '1;2;30'
tostring(.true.,'L3')                  '  T'
!
call tostring_set(trimz='NONE',sep=', ',trimb='YES')
tostring(real([(i,i=1,4)])**8)         '1.0000, 256.00, 6561.0, 65536.'
tostring([1.23456,1.2300,1.23456e6])   '1.2346, 1.2300, 1.23456E+06'
tostring(1.200d103)                    '1.20000+103'
!
call tostring_set(trimz='ALL')
tostring(real([(i,i=1,5)])**8,'f0.1')  '1, 256, 6561, 65536, 390625'
!
call tostring_set(rfmt='G12.4')
tostring(real([(i,i=0,5)])**8)         '1, 256, 6561, 0.6554E+05, 0.3906E+06'
tostring([1200000.,-1.2e-9])           '0.12E+07, -0.12E-08'
!
call tostring_set_factory()
```

6. TESTING THE MODULES

The package has been checked successfully with four different compilers, g95, gfortran, ifort from Intel and f95 from Nag. The g95 compiler has been used on Windows, Ubuntu Linux and Sun Solaris, gfortran on Windows and Ubuntu Linux, ifort on Windows and an AMD-64 computer with Suse Linux and the Nag compiler on Windows only.

There is one named constant at the beginning of the test programs which may be changed. This is the parameter `verbose`. When `verbose` is 2, the results of all tests are reported on the screen, when `verbose` is 1 (the default) the report is much more concise, and when it is 0 the report is minimal (and a few tests are bypassed). In all cases the report ends with “OK” if all tests are passed, otherwise it ends with assertion failure information.

6.1 Checking DISPMODULE only

The display modules are accompanied by a test program in the file `test_dispmodule.f90`. The test program is fairly comprehensive, and tries to test all the features of `DISPMODULE`. Thus all public procedures are called with and without all optional arguments, and for those arguments that have a limited range of allowed values (such as `ADVANCE`, `STYLE` and `TRIM`) all these values are tried. In addition the tests check all the examples that are given in this user manual. However, it is obviously impossible to test all possible combinations of arguments and argument values, so one must be content with a partial test, accompanied by examination of the source code of the modules and the test program.

To check only `DISPMODULE` (not the add-on modules), `test_dispmodule` should be compiled and linked with `dispmodule`. With the Gnu Fortran compiler and Unix give the commands:

```
$ gfortran -o test test_dispmodule.f90 dispmodule.f90
$ ./test
```

It is also possible to use Make, with the command (again assuming the Gnu compiler):

```
$ make check compiler=gfortran.
```

6.2 Complete checking

To check also the add-on modules and displaying of NaN-s and infinities it is easiest to use Make, which works if the compiler has a pre-processing option (most compilers do). Use one of the make-commands:

```
$ make check-all-kinds
$ make check-quadprec
$ make check-naninf
$ make check-naninf-ieee
```

possibly adding an option to select the compiler and/or editing the makefile to select the add-on modules to link with the program (needed if default integers are 8 bytes). The first listed make command checks displaying of 1, 2, 4 and 8 byte integers, single and double precision reals and 1 byte logicals, the second one checks displaying of quaruple precision reals, and the last two check displaying of nan-s and infinities (use check-naninf-ieee if the compiler supports Fortran 2003 ieee_arithmetic).

It is also possible to carry out complete checking manually, by un-commenting (some of) the "`use disp_...`" statements, changing the kind parameters (`irange` or `sik`, `logikind` and `srk`) near the beginning of `test_dispmodule.f90`, and linking with the needed add-on modules. With some compilers / computers it may be necessary to make changes to definitions of kind parameters in the add-on modules (and the test program).

6.3 Displaying from Matlab mex files

There is a separate example program of calling `DISP` from a Matlab mex file in `mexdispdemo.f90`. This has been tested with g95, gfortran and ifort on Windows. To run this example one must first set up the Matlab mex command for Fortran compilation, and for the testing this has been done with *gnumex* (see gnumex.sourceforge.net). Also one needs to use the `PUTSTRMODULE` that is commented out at the start of `DISPMODULE` (or with g95 the one given in section 3.4) instead of the dummy version. Having made these preparations one can issue from the Matlab prompt the commands:

```
>> mex -c dispmodule.f90
>> mex mexdispdemo.f90 dispmodule.obj
>> mexdispdemo(hilb(3), 1:4)
```

`Mexdispdemo` simply displays its arguments, which should be a matrix and a vector.