Appendix C. Usage of the *mathc90* Library

Table of Contents

1.1 Introduction

mathc90 is a conversion to the ANSI C language of the MATH77 library of Fortran 77 mathematical subprograms. We recommend organizing the *mathc90* library into two subdirectories: `libsrc` containing files of library source code and their associated header files, and `demo` a subdirectory of this containing source code files for demonstration drivers and their associated header files.

1.2 Usage documentation.

Usage of procedures in the (Fortran) MATH77 library is thoroughly documented in this manual. We do not provide similar documentation for the *mathc90* library. Rather, we have attempted to produce the *mathc90* code with a consistent correspondence between its functions and the subprograms of MATH77. Thus we hope a user can understand how to use *mathc90* by referring to the MATH77 (Fortran) documentation and applying the conventions given in this appendix for translating information about the declaration of variables. This approach can be supplemented by looking at the function prototypes in the header file `mathc90.h`, which is listed in Appendix D, and by studying the *mathc90* demonstration drivers.

Most of the user-callable C functions in *mathc90* have the same name, number of arguments, and functionality as the corresponding Fortran subprogram in MATH77. The types of arguments and results correspond according to conventions that will be described subsequently.

Exceptions as to the number of arguments occur for some procedures that have character arguments. For CSORT we have omitted the last argument which was work space of type CHARACTER. Also for CSORT, and for other procedures discussed under CHAR_INT on Page C–3, we have added a character length argument of type *int* immediately following an argument that was a CHARACTER array in the Fortran version.

Exceptions as to functionality occur in procedures DSVA/SSVA (Chapter 4.3), DNLxxx/SNLxxx (Chapter 9.3), MESS (Chapter 19.3), and I1MACH (Chapter 19.1). The Fortran subroutines DSVA/SSVA, DNLxxx/SNLxxx, and MESS allow the user to specify a Fortran output unit number to be written to. The *mathc90* function mess() emulates this functionality by writing to a file named `messf_nn`, where `nn` is the requested unit number. The *mathc90* functions dsva() and dnlxxx() simply write any requested output to *stdout*.

Calls to R1MACH, D1MACH, and I1MACH in the MATH77 library are replaced using functionality in the C language. Thus these subroutines are not included in the *mathc90* library.

A few codes are only in MATH77 or only in *mathc90*. These are discussed in Sections 1.8 and 1.9.

1.3 Header files.

There is one large header file `mathc90.h` which contains all the declarations required by any of the library files. This file is only provided with the entire mathc90 library, but in most cases we recommend using instead the header files corresponding to the individual routines. Each library routine except for `csort1.c` has a corresponding header file with the ".c" replaced by ".h". A user-written program that uses functions from *mathc90* should reference either `mathc90.h` in a `#include` directive or reference all the ".h" files with names corresponding to the names of the library files being referenced directly by the user's code. The library is not designed to work in a pre-ANSI C90 environment.

In addition there are header files for the library codes required by each of the demonstration drivers. A demonstration driver with a name of the form `drxxxx.c` uses a header file of the form `p_xxxx.h`. When calling only library routines that are called by a demonstration driver, one may want to include the appropriate one of these headers. Also a few of the drivers require a header file with the same name as the demonstration driver, but with the suffix ".c" replaced by ".h".

`fcrt.h` contains definitions of a number of macros whose

use is needed in some of the converted codes, due to the way the commercial conversion processor we use produces some of the conversions. A user code should not reference `fcrt.h`, as we may change or delete items in this file without notice. **If one is compiling single precision codes in *mathc90* changes may be required in `fcrt.h` as described in Section 1.5 below.**

1.3.a The library file, `fhelp.c`

The library file, `fhelp.c`, contains a number of functions supporting operations that are built-in in Fortran but not in C. Functions in this file are not intended to be called by users. They may be changed or deleted in the future without notice.

1.3.b A library archive file

We suggest that on UNIX systems the object files for all of the library be collected into a library archive file with the name `libmathc90.a`. This can easily be done using the UNIX `ar` command. Then this library can be referenced from a compile or link command using the option `-l mathc90`. We provide a Gnu make file that does this, as well as compiling all of the demos and running them. Analogous methods of creating and referencing a pre-compiled library file exist on other systems.

1.4 Correspondence between Fortran and C types.

Fortran 77 declaration in user code.	**C declaration in user code.**
double precision a	`double a;`
double precision a(5)	`double a[5];`
double precision a(5,10)	`double a[10][5];`
real a	`float a;`
real a(5)	`float a[5];`
real a(5,10)	`float a[10][5];`
integer i	`long i;`
integer i(5)	`long i[5];`
integer i(5,10)	`long i[10][5];`
logical p	LOGICAL32 p;
logical p(5)	LOGICAL32 p[5];
character ...	See: **CHARACTER arguments**, Page C–3.
external sname	Function declaration for sname.

In converting Fortran 77 code to ANSI C there are different choices one could make as to how to convert types. The specifications above relate specifically to the way we intend the *mathc90* library to be used. For subprogram arguments whose specification in the MATH77 (Fortran) documentation is as given in the first column below, we recommend a C declaration as specified in the second column.

A reference to an element of a 1-dimensional array, whose Fortran indexing begins at 1, has a corresponding reference in C with the index decremented by 1. Thus a(j) in Fortran 77 should be replaced by a[j-1] in C.

In a reference to an element of a 2-dimensional array, whose Fortran indexing begins at (1,1), the indices must be interchanged, and each decremented by 1. Thus a(i,j) in Fortran 77 should be replaced by a[j-1,i-1] in C.

If MATH77 documentation specifies an argument of a procedure to be an array of any type, or a CHARACTER variable with length that could be greater than one, the actual argument in referencing a *mathc90* function must be an address, *i.e.*, a pointer. (Recall that in C an array name is a pointer.) For an argument that is specified in the MATH77 documentation as numeric, LOGICAL, or CHARACTER*1, and not an array, it must be passed by address (*i.e.* a pointer) or by value depending on whether its value can or cannot be modified by the procedure. The MATH77 documentation identify each variable as having the intent *in* if it cannot be modified, and one of the intents *out* or *inout* (or *work* or *scratch*) if it can be modified. If in doubt about the correct typing of an argument look at the function prototype in the header file `mathc90.h` or at Appendix D.

Any argument to a library procedure that is specified in the MATH77 documentation as a 2-dimensional numeric array with the first dimension being adjustable will be declared within the corresponding *mathc90* function as a 1-dimensional array, and subscript computation will be expressed explicitly. (From the point of view of the MATH77 documentation, a 2-dimensional array has an adjustable first dimension if the size of the first dimension is one of the arguments in the same argument list as the array name.) Thus if the user declares the array to be used as the corresponding actual argument as 2-dimensional, as we recommend, the user should also use "cast" syntax in the function reference statement to force agreement of the type of the argument. For example, if an array specified to have an adjustable first dimension is declared as

```
double a[10][5];
```

it should appear in the function reference as

```
(double*)a
```

Some C compilers may not require this but some, *e.g.* HP-720, do.

On the other hand, if the first dimension of a 2-dimensional array is specified in the MATH77 documentation to be a fixed value, such as 2 in some subprograms dealing with double precision complex data and in Chapter 9.3, then the array name in the function reference should not be prefixed with `(double*)`. Check the func-

tion prototypes in `mathc90.h` (listed in Appendix D) or the associated header file when in doubt.

1.4.a CHARACTER arguments

In Fortran each CHARACTER variable has a declared length which may be specified using "*" notation or has the default length 1 otherwise. This declared length is automatically passed with the character variable whenever it is passed as a subprogram argument.

In C it is more common to think in terms of what could be called the operational length of a character string, *i.e.*, the length up to but not including a terminating null character. (The ASCII null character has the numeric value of zero and is written in C as $'\backslash 0'$.) This model of a character string is supported by a number of standard C library functions that do operations on null-terminated strings.

The notation $'A'$ denotes a constant of type *char* whose value is the integer, 65, associated with the ASCII character A. The notation $''A''$ denotes a pointer of type *char** pointing to an array of length 2 whose components are $'A'$ and $'\backslash 0'$. This later notation can also be used for longer strings, *e.g.*, $''Mars''$ denotes (a pointer to) an array of length 5 with components $'M'$, $'a'$, $'r'$, $'s'$, and $'\backslash 0'$.

Following conventions introduced by the commercial conversion processor that we use, we use the defined terms *byte*, STRING, and CHAR_INT to distinguish different usages of character variables. These three terms are synonyms for *char*, *char**, and *char*,int*. Note that *byte** is therefore another synonym for *char**.

We choose these different terms as follows:

STRING The translation of a non-array CHARACTER*k argument of intent *in* whose purpose is just to be printed. In the C version such an argument must be null-terminated. If the actual argument in C is written as a literal, it must be a string literal delimited by (double) quotes, *e.g.*, $''Venus''$. Occurrences in user-callable procedures are in the table at the top of the next column.

CHAR_INT The translation of an array of CHARACTER*k variables for which both the array size and CHARACTER length are user specified and the length k is not present in the same (Fortran) argument list. In this case the single argument in Fortran becomes two arguments in C. This pair of arguments must be of types *char** and *int*, respectively. The *int* parameter should be one greater than the k in the corresponding Fortran declaration to allow space for null terminators. Occurrences in user-callable procedures:

Chapter	Procedure Name	CHARACTER*k Argument
6.1	dmatp, dvecp, imatp, ivecp, smatp, svecp	text
6.2	dmatpr, dvecpr, imatpr, ivecpr, smatpr, svecpr	text
14.1	divadb, sivadb	text
16.2	dtcst, stcst	tcs, mode
16.3	dcft, scft	mode
16.3	dplot, splot	copt
19.2	ermsg,	subnam, mess
19.2	derm1, ierm1, serm1	subnam, mess, label
19.2	derv1, ierv1, serv1	label
19.2	ermor	mess
21.1	dprpl1, dprpl2 sprpl1, sprpl2	title, xname, yname

Chapter	Procedure Name	CHARACTER Array Argument
4.3	dsva, ssva	names
18.1	csort, csortp, csortq	c
19.2	dervn, servn	labels
19.3	dmess, mess, smess	text

byte The translation of a non-array CHARACTER*1 argument whose intent is *in*. If the actual argument in C is written as a literal, it must be a character literal delimited by apostrophes, *e.g.*, $'A'$. Occurrences in user-callable procedures:

Chapter	Procedure Name	CHARACTER*1 Argument
16.1	drft1, srft1	mode
16.4	drft, srft	mode
19.2	derm1, derv1, dervn, ermor, ermsg, ierm1, ierv1, serm1, serv1, servn	flag
21.2	dprpl, sprpl	symbol

***byte**, or **byte**[]* The translation of an array or non-array CHARACTER*k argument whose length k is known to the procedure by some means other than the implicit passing of the length in Fortran. The non-array argument of intent *in*, *i.e.*, symbol, is not required to be null-terminated, however if the actual argument is written as a literal it must be a string literal delimited by (double) quotes, *e.g.*, $''A''$.

In the procedures DSFITC/SSFITC of Chapter 11.5, the argument, CCODE, is an array of intent *in*, declared as

```
CHARACTER CCODE(mdim)*(4)
```

In C, ccode should be declared as

```
char ccode[mdim][5];
```

If the main program in Fortran would have made an assignment such as CCODE(I) = ′10∼a′, then the corresponding assignment in C could be ccode[i-1] = ″10∼a″.

In the Fortran subroutines DPRPL1/SPRPL1 and DPRPL2/SPRPL2, the declaration of the argument IMAGE depends on two other arguments, NLINES and NCHARS:

CHARACTER IMAGE(NLINES)*(NCHARS)

In C, *image* should be declared as

```
char image[nlines][nchars + 1];
```

and the actual argument should be written as `char* image` in the reference to dprpl1/sprpl1() or dprpl2/sprpl2(). On return, null terminators will have been stored in image[i][nchars], for i = 0, ..., $nlines - 1$.

In the Fortran subroutines DPRPL/SPRPL, the declaration of the argument IMAGE depends on another argument, NCHAR:

CHARACTER IMAGE*(NCHAR)

In C, *image* should be declared as

```
char image[nchar + 1];
```

On return there will be a null terminator in *image*[*nchar*].

Occurrences in user-callable procedures:

Chapter	Procedure Name	CHARACTER*k Argument	Intent
11.5	dsfitc, ssfitc	ccode	*in*
21.1	dprpl1, sprl1	image	*out*
21.1	dprpl2, sprl2	image	*out*
		symbol	*in*
21.2	dprpl,sprpl	image	*inout*

1.5 Using the type *float* in C.

The ANSI C language standard does not support the type *float* as completely as it supports the type *double*. In particular, there is a set of twenty-two elementary mathematical functions that are required by the ANSI C standard to be provided for operations on data of type *double*. The standard does not require these to be provided for type *float*, but does specify names these functions should have if they are provided.

The names of these functions for type *double* are *acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10, modf, pow, sqrt, ceil, fabs, floor*, and *fmod*. The corresponding names for type *float* are *acosf, asinf, atanf, atan2f, cosf, sinf, tanf, coshf, sinhf, tanhf, expf, frexpf, ldexpf, logf, log10f, modff, powf, sqrtf, ceilf, fabsf, floorf*, and *fmodf*. Function prototypes for these functions of type *float* are contained in the header file `mathf.h`.

The end of `fcrt.h` contains definitions for these single precision "intrinsic" functions. Three methods for defining these functions are provided, the method is selected by the value in the preprocessor variable `SINGLE_MATH_FUNCS`. If this variable has the value `intrinsic`, it is assumed that the same names can be used for the single as for the double precision versions. If this variable has the value `use_double` the double precision functions are used with the arguments cast to *double*, and the result cast back to type *float*. If `SINGLE_MATH_FUNCS` has neither of these values, the names used for these functions are the usual double names followed by an "f", thus for example `sin` becomes `sinf` in the single precision codes. **It is critical that the user choose the correct value for** `SINGLE_MATH_FUNCS` if the single precision routines are compiled.

Note that the following two function headers do not have the same meaning:

Example H1:
```
fname(x)
float x;
```

Example H2:
```
fname(float x)
```

The first example is in the K&R or pre-ANSI style. It implies that the argument x is expected to be passed as type *double* and will be coerced to type *float* before being used within the function. In the second example the argument x is expected to be passed as type *float*.

Similarly there are distinct pre-ANSI and ANSI styles for use in a code that references a function:

Example R1:
```
void fname();
float x;
...
fname(x);
```

Example R2:
```
void fname(float);
float x;
...
fname(x);
```

In Example R1, which is the pre-ANSI style, x will be coerced from *float* to *double* when it is passed to fname, whereas in Example R2, which is the ANSI style, x will be passed as type *float*. The ANSI standard allows usage of either pre-ANSI or ANSI style. Note, however, that R1 is compatible with H1, and R2 is compatible with H2, but mixing the styles gives erroneous results.

We have used the ANSI style, i.e., as in Example H2, in the codes of mathc90 that have arguments of type *float*. Thus a user must use the ANSI style, i.e., as in Example R2, in code that references a library function that has an argument of type *float*. The needed function prototypes can be easily obtained by simply including the header file *mathc90.h*.

1.6 Specification of functions to be passed to library procedures.

For some library procedures one or more of the arguments is the name of a user-coded procedure. As a documentation aid, we list prototypes for these procedures to the right, using the MATH77 names.

Chap.	Library Functions	Prototypes for user-coded functions
8.2	dnqsol()	`void dnqfj(long,double[],double[],double*,long*);`
	snqsol()	`void snqfj(long,float[],float[],float*,long*);`
9.2	dmlc01()	`void dmlcfg(long,double[],double*,double[], LOGICAL32*);`
	smlc01()	`void smlcfg(long,float[],float*,float[], LOGICAL32*);`
9.3	dnlafu() dnlagu() dnlafb() dnlagb()	`void dcalcr(long,long,double[],long*,double[]);` `void dcalcj(long,long,double[], long*,double*);`
	snlafu() snlagu() snlafb() snlagb()	`void scalcr(long,long,float[],long*,float[]);` `void scalcj(long,long,float[],long*,double*);`
9.3	dnlsfu() dnlsgu() dnlsfb() dnlsgb()	`void dcalca(long,long,long,double[],long*,double*);` `void dcalcb(long,long,long,double[],long*,double*);`
	snlsfu() snlsgu() snlsfb() snlsgb()	`void scalca(long,long,long,float[],long*, float*);` `void scalcb(long,long,long,float[],long*,float*);`
14.4	diva() divaa()	`void derivs(double[],double[],double[],long[]);` `void output(double[],double[],double[],long[]);`
	siva() sivaa()	`void derivs(float[],float[],float[],long[]);` `void output(float[],float[],float[],long[]);`
18.3	insort()	`long compar(long,long);`
18.4	exsort()	`void dataop(long,long,long,long*);`

1.7 User-accessible COMMON blocks.

We have generally avoided user-accessible COMMON blocks in MATH77, however in specialized usages of the code of Chapter 14.1 for Ordinary Differential Equations there is a need to access the COMMON block DIVASC.

In *mathc90* this is represented as an external data structure named *divasc* and having type *t_divasc*. See the file `diva.c` for the components of this structure. If you are working from the "mangled" code, the names will be different than those in the documentation, but the names in the structure appear in the same order as given in the documentation. For type *float*, replace "diva" with "siva" in the text above.

1.8 Codes in MATH77 but not in *mathc90*.

Since ANSI C does not have a complex data type, some codes using a complex data type are in MATH77 but not in *mathc90*. In particular codes in Chapters 4.1,"Square Nonsingular Systems of Equations", and 6.3, "Basic Linear Algebra Subprograms" which use complex arguments are not available in *mathc90*. When the Fortran codes implement a complex version by using real or double precision arrays, such versions are also available in C.

1.9 Codes in *mathc90* but not in MATH77.

There is one additional sorting function and a few additional single precision complex arithmetic functions in *mathc90*.

To the functions *csort*(), *csortp*(), and *csortq*() of Chapter 18.1, we have added a character string sorting function named *csort1*(). For *csort*(), *csortp*(), and *csortq*(), the character data to be sorted must be in an array declared in a form such as

```
char c[100][11];
```

which would provide space for 100 character strings each

having up to 10 meaningful character positions plus a null termination character. The second dimensioning value, *e.g.* 11 in this example, must be passed as the second argument to *csort*(), *csortp*(), or *csortq*(). Note that the operational length of each character string in the array is determined by the position of its null terminator, but a total of 11 character positions of storage are allocated for each string.

For *csort1*() the character data to be sorted are referenced via an array of pointers to *char*. Such an array, say for 100 character strings, can be declared as

```
char *pc[100];
```

With this method of storing character strings, each string need not occupy any more storage than is needed for its meaningful characters, its null terminator, and its pointer. The procedure *csort1*() operates by swapping C pointers, whereas *csortp*() and *csortq*() operate by swapping indices, and *csort*() operates by swapping the actual character strings.

The statement for referencing *csort1*() is

```
csort1( pc, m, n, k, l);
```

where `pc` must be declared as described above and the other four arguments are *long int*'s with the same interpretation as in the other character sorting procedures.

In MATH77, Chapter 17.2, there are subprograms for double precision complex arithmetic named ZSUM, ZDIF, ZPRO, ZQUO, ZSQRTX, and DZABS. These are provided because double precision complex arithmetic is not directly supported in standard Fortran 77. There is no need in Fortran for single precision versions of these since single precision complex arithmetic is directly supported in standard Fortran 77.

In C, however, there is no direct support for complex arithmetic, so there is potential use for both single precision and double precision versions of these complex operations. Thus, in *mathc90* we provide routines named *csum, cdif, cpro, cquo, csqrtx*, and *scabs*, with arguments (and the returned value from *scabs*) of type *float*, as well as the "*z*" procedures for type *double*.

1.10 Testing the portability of the *mathc90* library.

The *mathc90* library has been tested for portability on the same machines as the MATH77 library. This test involves running the set of demonstration drivers and comparing the results with those obtained in Fortran and with results in C on other machines.