AMD Core Math Library (ACML)

Version 6.1.0

Attributions

Some versions of the library are built with Intel(R) Visual Fortran Compiler Professional Edition for Windows `http://software.intel.com/en-us/intel-compilers`

Short Contents

Table of Contents

1 Introduction

The AMD Core Math Library (ACML) is a set of numerical routines tuned specifically for AMD64 platform processors (including Opteron™ and Athlon™64). The routines, which are available via both FORTRAN 77 and C interfaces, include:

- BLAS - Basic Linear Algebra Subprograms (including Sparse Level 1 BLAS);
- LAPACK - A comprehensive package of higher level linear algebra routines;
- FFT - a set of Fast Fourier Transform routines for real and complex data;
- RNG - a set of random number generators and statistical distribution functions.

The BLAS and LAPACK routines provide a portable and standard set of interfaces for common numerical linear algebra operations that allow code containing calls to these routines to be readily ported across platforms. Full documentation for the BLAS and LAPACK are available online. This manual will, therefore, be restricted to providing brief descriptions of the BLAS and LAPACK and providing links to their documentation and other materials (see Chapter 3 [The BLAS], page 15 and see Chapter 4 [LAPACK], page 16).

The FFT is an implementation of the Discrete Fourier Transform (DFT) that makes use of symmetries in the definition to reduce the number of operations required from O(n*n) to O(n*log n) when the sequence length, n, is the product of small prime factors; in particular, when n is a power of 2. Despite the popularity and widespread use of FFT algorithms, the definition of the DFT is not sufficiently precise to prescribe either the forward and backward directions (these are sometimes interchanged), or the scaling factor associated with the forward and backward transforms (the combined forward and backward transforms may only reproduce the original sequence by following a prescribed scaling).

Currently, there is no agreed standard API for FFT routines. Hardware vendors usually provide a set of high performance FFTs optimized for their systems: no two vendors employ the same interfaces for their FFT routines. The ACML provides a set of FFT routines, optimized for AMD64 processors, using an ACML-specific set of interfaces. Beginning with version 6.0, ACML library has adopted the popular FFTW library's interfaces. The functionality, interfaces and use of the ACML FFT routines are described below (see Chapter 5 [Fast Fourier Transforms], page 23).

The RNG is a comprehensive set of statistical distribution functions which are founded on various underlying uniform distribution generators *(base generators)* including Wichmann-Hill and an implementation of the Mersenne Twister. In addition there are hooks which allow you to supply your own preferred base generator if it is not already included in ACML. All RNG functionality and interfaces are described below (see Chapter 6 [Random Number Generators], page 100).

Chapter 2 [General Information], page 2 provides details on:

- how to link a user program to the ACML;
- FORTRAN and C interfaces to ACML routines;
- how to obtain the ACML version and build information;
- how to access the ACML documentation.

2 General Information

2.1 Determining the best ACML version for your system

ACML comes in versions for 64-bit processors, running both Linux and Microsoft Windows®operating systems. To use the following tables, you will need to know answers to these questions:

- Are you running a 64-bit operating system (on AMD64 hardware such as Opteron or Athlon64)?
- Is the operating system Linux or Microsoft Windows?
- Do you have the GNU compilers (gfortran/gcc) or compatible compilers (compilers such as Absoft that are interoperable with the GNU compilers) installed?
- Do you have the PGI compilers (pgf77/pgf90/pgcc) installed?
- Do you have the Open64 compilers (openf95/opencc) installed?
- Do you have the NAGWare compiler (f95) installed?
- Do you have a single processor system or a multiprocessor (SMP) system? The single processor version of ACML can be run on an SMP machine and vice versa, but (if you have the right compilers) it is more efficient to run the version appropriate to the machine.
- Does the machine support advanced instruction sets such as FMA4 or AVX?

The ACML installation includes a binary utility that can help you find an answer to the last question. The utility lies in directory util, and is named cpuid.exe. It interrogates the processor to determine what instruction families exist.

```
util/cpuid.exe
```

Under a Linux operating system, another way of finding out the answer to the last question is to look at the special file /proc/cpuinfo, and see what appears under the "flags" label. Try this command:

```
cat /proc/cpuinfo | grep flags
```

If the list of flags includes the flag "sse2" then your machine supports SSE2 instructions. If it also includes "fma4" then your machine supports FMA4 instructions. If your machine supports these instructions, it is better to use a version of ACML which was built to take advantage of them, for reasons of good performance.

The method of examining /proc/cpuinfo can also be used under Microsoft Windows if you have the Cygwin UNIX-like tools installed (see http://www.cygwin.com/) and run a bash shell. Note that AMD64 machines always support both SSE and SSE2 instructions, under both Linux and Windows. Other manufacturers' hardware may or may not support AVX or FMA4.

If you link to a version of ACML that was built to use AVX or FMA4 instructions, and your machine does not in fact support them, it is likely that your program will halt due to encountering an "illegal instruction" - you may or may not be notified of this by the operating system.

Once you have answered the questions above, use these tables to decide which version of ACML to link against.

Linux 64-bit

Number of threads	*Compilers*	*ACML install directory*
Single thread	PGI pgf77/pgf90/pgcc	`acml6.1.0/pgi64`
”	PGI pgf77/pgf90/pgcc fma4	`acml6.1.0/pgi64_fma4`
”	GNU gfortran/gcc or compat.	`acml6.1.0/gfortran64`
”	GNU gfortran/gcc fma4	`acml6.1.0/gfortran64_fma4`
”	Open64 openf95/opencc	`acml6.1.0/open64_64`
”	Open64 openf95/opencc fma4	`acml6.1.0/open64_64_fma4`
”	Intel Fortran	`acml6.1.0/ifort64`
”	Intel Fortran fma4	`acml6.1.0/ifort64_fma4`
”	NAGWare f95	`acml6.1.0/nag64`
”	Absoft (use gfortran ACML)	`acml6.1.0/gfortran64`
Multiple threads	PGI pgf77/pgf90/pgcc	`acml6.1.0/pgi64_mp`
”	PGI pgf77/pgf90/pgcc fma4	`acml6.1.0/pgi64_fma4_mp`
”	GNU gfortran/gcc or compat.	`acml6.1.0/gfortran64_mp`
”	GNU gfortran/gcc fma4	`acml6.1.0/gfortran64_fma4_mp`
”	Open64 openf95/opencc	`acml6.1.0/open64_64_mp`
”	Open64 openf95/opencc fma4	`acml6.1.0/open64_64_fma4_mp`
”	Intel Fortran	`acml6.1.0/ifort64_mp`
”	Intel Fortran fma4	`acml6.1.0/ifort64_fma4_mp`
”	Absoft (use gfortran ACML)	`acml6.1.0/gfortran64_mp`

Microsoft Windows 64-bit

Number of processors	*Compilers*	*ACML install directory*
Single processor	PGI pgf77/pgf90/pgcc/MSC	`acml6.1.0/win64`
”	Intel Fortran/Microsoft C	`acml6.1.0/ifort64`
Multi processor or core	PGI pgf77/pgf90/pgcc/MSC	`acml6.1.0/win64_mp`
”	Intel Fortran/Microsoft C	`acml6.1.0/ifort64_mp`

2.2 Accessing the Library (Linux)

2.2.1 Accessing the Library under Linux using GNU gfortran/gcc

If the Linux 64-bit gfortran version of ACML was installed in the default directory, /opt/acml6.1.0/gfortran64, then the command:

```
gfortran -m64 driver.f -L/opt/acml6.1.0/gfortran64/lib -lacml
```

can be used to compile the program driver.f and link it to the ACML.

The command

```
gfortran -m64 -mavx -mfma4 driver.f \
         -L/opt/acml6.1.0/gfortran64_fma4/lib -lacml
```

will compile and link a 64-bit program with the 64-bit FMA4 ACML. The Fortran module driver.f will be compiled using AVX and FMA4 instructions where possible in lieu of SSE/SSE2.

The ACML Library is supplied in both static and shareable versions, libacml.a and libacml.so, respectively. By default, the commands given above will link to the shareable version of the library, libacml.so, if that exists in the directory specified. Linking with the static library can be forced either by using the compiler flag `-static`, e.g.

```
gfortran -m64 driver.f \
         -L/opt/acml6.1.0/gfortran64/lib -static -lacml -lrt
```

or by inserting the name of the static library explicitly in the command line, e.g.

```
gfortran -m64 driver.f /opt/acml6.1.0/gfortran64/lib/libacml.a -lrt
```

Notice that if the application program has been linked to the shareable ACML Library, then before running the program, the environment variable `LD_LIBRARY_PATH` must be set. Assuming that libacml.so was installed in the directory /opt/acml6.1.0/gfortran64/lib, then `LD_LIBRARY_PATH` may be set by, for example, the C-shell command

```
setenv LD_LIBRARY_PATH /opt/acml6.1.0/gfortran64/lib
```

(See the man page for ld(1) for more information about `LD_LIBRARY_PATH`.)

If you have an SMP machine and want to take best advantage of it, link against the gfortran OpenMP version of ACML like this:

```
gfortran -fopenmp -m64 driver.f \
         -L/opt/acml6.1.0/gfortran64_mp/lib -lacml_mp
```

Note that the directories and library names involved now include the suffix *_mp*.

To compile and link a 64-bit C program with a 64-bit ACML, invoke

```
gcc -m64 -I/opt/acml6.1.0/gfortran64/include driver.c \
    -L/opt/acml6.1.0/gfortran64/lib -lacml -lgfortran
```

The switch "-I/opt/acml6.1.0/gfortran64/include" tells the compiler to search the directory /opt/acml6.1.0/gfortran64/include for the ACML C header file acml.h, which should be included by driver.c. Note that it is necessary to add the gfortran compiler run-time library -lgfortran when linking the program.

2.2.2 Accessing the Library under Linux using PGI compilers pgf77/pgf90/pgcc

Similar commands apply for the PGI versions of ACML. For example,

```
pgf77 -tp=k8-64 -Mcache_align driver.f -L/opt/acml6.1.0/pgi64/lib -lacml
```

will compile driver.f and link it to the ACML 64-bit version. In the example above we are linking with the single-processor PGI version of ACML.

The command

```
pgf77 -tp=bulldozer -Mcache_align -pgf90libs driver.f \
      -L/opt/acml6.1.0/pgi64_fma4/lib -lacml
```

will compile and link a 64-bit program with the 64-bit FMA4 ACML. The Fortran module driver.f will be compiled using AVX and FMA4 instructions where possible instead of SSE/SSE2.

If you have an SMP machine and want to take best advantage of it, link against the PGI OpenMP version of ACML like this:

```
pgf77 -tp=k8-64 -mp -Mcache_align driver.f \
      -L/opt/acml6.1.0/pgi64_mp/lib -lacml_mp
```

Note that the directories and library names involved now include the suffix *_mp*.

The -mp flag is important - it tells pgf77 to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time. The -Mcache_align flag is also important - it tells the compiler to align objects on cache-line boundaries.

The commands

```
pgcc -c -tp=k8-64 -mp -Mcache_align \
      -I/opt/acml6.1.0/pgi64_mp/include driver.c
pgcc -tp=k8-64 -mp -Mcache_align driver.o \
      -L/opt/acml6.1.0/pgi64_mp/lib -lacml_mp -lpgftnrtl -lm
```

will compile driver.c and link it to the 64-bit ACML. Again, the -mp flag is important if you are linking to the PGI OpenMP version of ACML. The C compiler is instructed to search the directory /opt/acml6.1.0/pgi64_mp/include for the ACML C header file acml.h, which should be included by driver.c, by using the switch "-I/opt/acml6.1.0/pgi64_mp/include". Note that in the example we add the libraries -lpgftnrtl and -lm to the link command, so that required PGI compiler run-time libraries are found.

Note that since ACML version 3.5.0, all PGI 64-bit variants are compiled with the PGI -Mlarge_arrays switch to allow use of larger data arrays (see PGI compiler documentation for more information). The special 'large array' variants that were distributed with earlier versions of ACML are therefore no longer required.

2.2.3 Accessing the Library under Linux using Open64 compilers openf95/opencc

Similar commands apply for the Open64 versions of ACML. For example,

```
openf95 driver.f -L/opt/acml6.1.0/open64_64/lib -lacml
```

will compile driver.f and link it to the ACML 64-bit version. In the example above we are linking with the single-processor open64 version of ACML.

The command

```
openf95 driver.f -L/opt/acml6.1.0/open64_64_fma4/lib -lacml
```

will compile and link a 64-bit program with the 64-bit FMA4 ACML. The Fortran module driver.f will be compiled using AVX and FMA4 instructions where possible in lieu of SSE/SSE2. The open64 compiler automatically detects AVX/FMA4 capability on the host CPU and will use these instructions if the CPU supports them.

If you have an SMP machine and want to take best advantage of it, link against the open64 OpenMP versions of ACML like this:

```
openf95 -mp driver.f \
         -L/opt/acml6.1.0/open64_64_mp/lib -lacml_mp
```

Note that the directories and library names involved now include the suffix *_mp*.

The commands

```
opencc -c -I/opt/acml6.1.0/open64_64/include driver.c
opencc driver.o -L/opt/acml6.1.0/open64_64/lib -lacml -lfortran
```

will compile driver.c and link it to the 64-bit ACML. The switch

```
-I/opt/acml6.1.0/open64_64/include
```

tells the C compiler to search directory /opt/acml6.1.0/open64_64/include for the ACML C header file acml.h, which should be included by driver.c. Note that in the example we add the library -lfortran to the link command, so that the required Open64 compiler run-time library is found.

2.2.4 Accessing the Library under Linux using the NAGWare f95 compiler

Similar commands apply for the NAGware f95 versions of ACML. For example,

```
f95 driver.f -L/opt/acml6.1.0/nag64/lib -lacml
```

will compile driver.f and link it to the ACML using the 64-bit version.

2.2.5 Accessing the Library under Linux using the Intel ifort compiler

Similar commands apply for the Intel ifort versions of ACML. For example,

```
ifort driver.f -L/opt/acml6.1.0/ifort64/lib -lacml
```

will compile driver.f and link it to the ACML using the 64-bit version.

The commands

```
gcc -c -I/opt/acml6.1.0/ifort64/include driver.c
ifort -nofor-main driver.o -L/opt/acml6.1.0/ifort64/lib -lacml
```

will compile driver.c and link it to the 64-bit ACML. The switch

```
-I/opt/acml6.1.0/ifort64/include
```

tells the C compiler to search directory /opt/acml6.1.0/ifort64/include for the ACML C header file acml.h, which should be included by driver.c. Note that in the example we link the C program using the ifort compiler with the -nofor-main switch, so that required ifort compiler run-time libraries are found.

If you have an SMP machine and want to take best advantage of it, link against the ifort OpenMP version of ACML like this:

```
ifort -openmp driver.f -L/opt/acml6.1.0/ifort64_mp/lib -lacml_mp
```

Note that the directories and library names involved now include the suffix *_mp*.

The -openmp flag is important - it tells ifort to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time.

2.2.6 Accessing the Library under Linux using Absoft af90

The Absoft compiler af90 is compatible with the GNU compiler gfortran version of ACML, so long as the appropriate gfortran run-time libraries are installed on your system. If the Linux 64-bit gfortran version of ACML was installed in the default directory, /opt/acml6.1.0/gfortran64, then the command:

```
af90 -m64 driver.f -L/opt/acml6.1.0/gfortran64/lib -lacml -lgfortran
```

can be used to compile the program driver.f and link it to the ACML. Note that -gfortran links to the gfortran run-time library, which must be installed on your system.

The ACML Library is supplied in both static and shareable versions, libacml.a and libacml.so, respectively. By default, the commands given above will link to the shareable version of the library, libacml.so, if that exists in the directory specified. Linking with the static library can be forced either by using the compiler flag `-static`, e.g.

```
af90 -m64 driver.f -L/opt/acml6.1.0/gfortran64/lib -static \
     -lacml -lgfortran -lrt
```

or by inserting the name of the static library explicitly in the command line, e.g.

```
af90 -m64 driver.f /opt/acml6.1.0/gfortran64/lib/libacml.a \
     -lgfortran -lrt
```

Notice that if the application program has been linked to the shareable ACML Library, then before running the program, the environment variable `LD_LIBRARY_PATH` must be set. Assuming that libacml.so was installed in the directory /opt/acml6.1.0/gfortran64/lib, then `LD_LIBRARY_PATH` may be set by, for example, the C-shell command

```
setenv LD_LIBRARY_PATH /opt/acml6.1.0/gfortran64/lib
```

(See the man page for ld(1) for more information about `LD_LIBRARY_PATH`.)

If you have an SMP machine and want to take best advantage of it, link against the gfortran OpenMP version of ACML like this:

```
af90 -m64 driver.f -L/opt/acml6.1.0/gfortran64_mp/lib \
     -lacml_mp -lgfortran -lgomp
```

Note that the directories and library names involved now include the suffix *_mp*. Also note that it is necessary to link to the gfortran run-time libraries -lgfortran -lgomp, both of which must be installed on your system.

2.2.7 Accessing the Library under Linux using compilers other than GNU, PGI, Open64, NAGWare, Intel or Absoft

It may be possible to link to some versions of ACML using compilers other than those already mentioned, if they are compatible with one of the other versions. If you do this, it may be necessary to link to the run-time library of the compiler used to build the ACML you link to, in order to satisfy run-time symbols. Since doing this is very compiler-specific, we give no further details here.

2.3 Accessing the Library (Microsoft Windows)

2.3.1 Accessing the Library under 64-bit Windows using PGI compilers pgf77/pgf90/pgcc

Under 64-bit versions of Windows, ACML 6.1.0 comes as a static (.LIB) library or a DLL.

To link with the 64-bit Windows DLL library PGI version of ACML, in a DOS command prompt use a command like

```
pgf77 -Mdll driver.f c:\acml6.1.0\win64\lib\libacml_dll.lib
```

where libacml_dll.lib is the import library for the DLL. In the example above we are linking with the single-processor WIN64 version of ACML.

If you have an SMP machine and want to take best advantage of it, link against the WIN64 OpenMP version of ACML like this:

```
pgf77 -Mdll -mp driver.f c:\acml6.1.0\win64_mp\lib\libacml_mp_dll.lib
```

Note that the directories and library names involved now include the suffix *_mp*.

For the OpenMP version of ACML, if you link to the static library libacml_mp.lib rather than the DLL import library libacml_mp_dll.lib, you will need to use the PGI compiler flag -mp in order to tell the compiler to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time. This

should not be necessary when linking to the ACML DLL because the DLL itself knows that it depends on the run-time library; but using the -mp flag in any case will do no harm.

Note that the performance of OpenMP code produced with the PGI WIN64 compilers depends on environment variables named *MP_BIND* and *MP_SPIN*, which control how multiple threads behave (see PGI compiler documentation for discussion of these variables). For ACML, empirical experiments show that higher values of *MP_SPIN* than the default are likely to give better performance. We recommend that users set *MP_BIND=yes* and *MP_SPIN=100000000*.

Under WIN64, to compile and link a C program, the commands

```
pgcc -Mdll driver.c -Ic:\acml6.1.0\win64\include
     c:\acml6.1.0\win64\lib\libacml_dll.lib
pgcc -Mdll -mp driver.c -Ic:\acml6.1.0\win64_mp\include
     c:\acml6.1.0\win64_mp\lib\libacml_mp_dll.lib
```

will link against the single-threaded DLL and multi-threaded versions of ACML respectively.

To use the Microsoft C command line compiler, *cl*, use commands like this:

```
cl driver.c -Ic:\acml6.1.0\win64\include
     c:\acml6.1.0\win64\lib\libacml_dll.lib
cl driver.c -Ic:\acml6.1.0\win64_mp\include
     c:\acml6.1.0\win64_mp\lib\libacml_mp_dll.lib
```

for single- and multi-threaded ACML variants respectively.

2.3.2 Accessing the Library under 64-bit Windows using Microsoft C or Intel Fortran

Under 64-bit versions of Windows, ACML 6.1.0 comes as a static (.LIB) library or a DLL.

To link with the 64-bit Windows DLL library Intel Fortran version of ACML, in a DOS command prompt use a command like

```
ifort /libs:dll driver.f c:\acml6.1.0\ifort64\lib\libacml_dll.lib
```

where libacml_dll.lib is the import library for the DLL. In the example above we are linking with the single-processor ifort version of ACML.

If you have an SMP machine and want to take best advantage of it, link against the ifort OpenMP version of ACML like this:

```
ifort /libs:dll -Qopenmp driver.f
  c:\acml6.1.0\win64_mp\lib\libacml_mp_dll.lib
```

Note that the directories and library names involved now include the suffix *_mp*.

For the OpenMP version of ACML, if you link to the static library libacml_mp.lib rather than the DLL import library libacml_mp_dll.lib, you will need to use the ifort compiler flag -Qopenmp in order to tell the compiler to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time. This should not be necessary when linking to the ACML DLL because the DLL itself knows that it depends on the run-time library; but using the -Qopenmp flag in any case will do no harm.

Under WIN64, to compile and link a C program using the Microsoft C command line compiler, *cl*, the commands

```
cl driver.c -Ic:\acml6.1.0\ifort64\include
     c:\acml6.1.0\ifort64\lib\libacml_dll.lib
cl driver.c -Ic:\acml6.1.0\ifort64_mp\include
     c:\acml6.1.0\ifort64_mp\lib\libacml_mp_dll.lib
```

will link against the single-threaded DLL and multi-threaded versions of ACML respectively.

2.4 ACML FORTRAN and C interfaces

All routines in ACML come with both FORTRAN and C interfaces. The FORTRAN interfaces typically follow the relevant standard (e.g. LAPACK, BLAS). Here we document how a C programmer should call ACML routines.

In C code that uses ACML routines, be sure to include the header file <acml.h>, which contains function prototypes for all ACML C interfaces. The header file also contains C prototypes for FORTRAN interfaces, thus the C programmer could call the FORTRAN interfaces from C, though there is little reason to do so.

C interfaces to ACML routines differ from FORTRAN interfaces in the following major respects:

- The FORTRAN interface names are appended by an underscore (except for the Windows 64-bit Microsoft C/Intel Fortran version of ACML, where FORTRAN interface names are distinguished from C by being upper case rather than lower case - this is the default for the Intel Fortran compiler)
- The C interfaces contain no workspace arguments; all workspace memory is allocated internally.
- Scalar input arguments are passed by value in C interfaces. FORTRAN interfaces pass all arguments (except for character string *length* arguments that are normally hidden from FORTRAN programmers) by reference.
- Most arguments that are passed as character string pointers to FORTRAN interfaces are passed by value as single characters to C interfaces. The character string *length* arguments of FORTRAN interfaces are not required in the C interfaces.
- Unlike FORTRAN, C has no native *complex* data type. ACML C routines which operate on complex data use the types *complex* and *doublecomplex* defined in <acml.h> for single and double precision computations respectively. Some of the programs in the ACML examples directory (see Section 2.8 [Examples], page 12) make use of these types.

It is important to note that in both the FORTRAN and C interfaces, 2-dimensional arrays are assumed to be stored in column-major order. e.g. the matrix

$$A = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

would be stored in memory as 1.0, 3.0, 2.0, 4.0. This storage order corresponds to a FORTRAN-style 2-D array declaration A(2,2), but not to an array declared as a[2][2] in C which would be stored in row-major order as 1.0, 2.0, 3.0, 4.0.

As an example, compare the FORTRAN and C interfaces of LAPACK routine dsytrf as implemented in ACML.

FORTRAN:

```
void dsytrf_(char *uplo, int *n, double *a, int *lda, int *ipiv,
             double *work, int *lwork, int *info, int uplo_len);
```

C:

```
void dsytrf(char uplo, int n, double *a, int lda, int *ipiv,
            int *info);
```

C code calling both the above variants might look like this:

```
double *a;
int *ipiv;
double *work;
int n, lda, lwork, info;

/* Assume that all arrays and variables are allocated and
   initialized as required by dsytrf. */

/* Call the FORTRAN version of dsytrf. The first argument
   is a character string, and the last argument is the
   length of that string. The input scalar arguments n, lda
   and lwork, as well as the output scalar argument info,
   are all passed by reference. */
dsytrf_("Upper", &n, a, &lda, ipiv, work, &lwork, &info, 5);

/* Call the C version of dsytrf. The first argument is a
   character, workspace is not required, and input scalar
   arguments n and lda are passed by value. Output scalar
   argument info is passed by reference. */
dsytrf('U', n, a, lda, ipiv, &info);
```

2.5 ACML variants using 64-bit integer (INTEGER*8) arguments

Where compilers support, through the use of switches, the automatic promotion of regular INTEGER (32-bit) arguments to INTEGER*8 (64-bit) arguments, ACML variants exist to use this facility. This means that if you have a 64-bit Fortran program using INTEGER*8 variables, or a 64-bit C program using 8-byte long variables, there is an ACML version that you can use. This applies to 64-bit ACML versions built with PGI, Open64, gfortran, Intel and NAG compilers.

The INTEGER*8 versions of these libraries are distinguished from the usual versions by having the string "_int64" as part of the name of the directory under which ACML is installed. Thus, for example, if the regular PGI 64-bit library is in a directory named pgi64, then the INTEGER*8 version will be installed in directory pgi64_int64.

For these ACML variants, all ACML documentation that mentions arguments of Fortran type *INTEGER* or C type *int* should be read as *INTEGER*8* or *long* respectively.

It is important to ensure that if you have INTEGER*8 variables in your code, you link to the int64 variant, and not otherwise. Unexpected program crashes are likely to occur if you link to the wrong version.

2.6 Library Version and Build Information

This document is applicable to version 6.1.0 of ACML. The utility routine `acmlversion` can be called to obtain the major, minor and patch version numbers of the installed ACML.

Prior to version 5.2.0, this routine returned three integers; the major, minor and patch version numbers, respectively.

Beginning with version 5.2.0, this routine returns four integers; the major, minor and patch version numbers, and a build number. The build number corresponds to an internal AMD build system numbering scheme and will be different for various versions of ACML.

The utility routine `acmlinfo` can be called to obtain information on the compiler used to build ACML, the version of the compiler, and the options used for building the Library. This subroutine takes no arguments and prints the information to the current standard output.

FORTRAN specifications:

`ACMLVERSION` (*MAJOR*, *MINOR*, *PATCH*, *BUILD*) [SUBROUTINE]

MAJOR, *MINOR*, *PATCH*, *BUILD* [INTEGER]

`ACMLINFO ()` [SUBROUTINE]

C specifications:

`void acmlversion` (*int* `*major`, *int* `*minor`, *int* `*patch`, *int* `*build`); [function]

`void acmlinfo` (*void*); [function]

2.7 Library Documentation

The /Doc subdirectory of the top ACML installation directory, (e.g. /opt/acml6.1.0/Doc under Linux, or c:\AMD\acml6.1.0\Doc under Windows), should contain this document in the following formats:

- Printed Manual / PDF format – acml.pdf
- Info Pages – acml.info (Linux only)
- Html – html/index.html
- Plain text – acml.txt

Under Linux the info file can be read using *info* after updating the environment variable INFOPATH to include the doc subdirectory of the ACML installation directory, e.g.

```
% setenv INFOPATH ${INFOPATH}:/opt/acml6.1.0/Doc

% info acml
```

or simply by using the full name of the file:

```
% info /opt/acml6.1.0/Doc/acml.info
```

2.8 Example programs calling ACML

The /examples subdirectory of the top ACML installation directory (for example, possible default locations are /opt/acml6.1.0/pgi64/examples under Linux, or, under windows, c:\acml6.1.0\win64\examples), contains example programs showing how to call the ACML, along with a GNUmakefile to build and run them. Examples of calling both FORTRAN and C interfaces are included. They may be used as an ACML installation test.

Depending on where your copy of the ACML is installed, and which compiler and flags you wish to use, it may be necessary to modify some variables in the GNUmakefile before using it.

For 64-bit Windows versions of ACML it may be useful to have the Cygwin UNIX-like tools installed, so that you can use the *make* command that comes with them to build the examples. However, it is not necessary to have the Cygwin tools. The examples directory contains a bat script, *acmlexample.bat*, which can be used to run one of the example programs. Another bat script, *acmlallexamples.bat*, builds and runs all the examples in the directory. Alternatively, if you do have the Cygwin tools installed, you can use the GNUmakefile to build the examples.

If you need more example programs showing how to call LAPACK routines from Fortran, we refer you to this web page:

`http://www.nag.com/lapack/`

Here you will find examples for all double precision LAPACK driver routines, and all of these should work when linked with ACML. Note that as well as the example programs themselves, it is necessary to download and compile a small amount of utility code used by the programs. See the web page for detailed instructions.

2.9 Example ACML programs demonstrating performance

The /examples/performance subdirectory of the top ACML installation directory (for example, possible default locations are /opt/acml6.1.0/pgi64/examples/performance under Linux, or c:\acml6.1.0\win64\examples\performance under windows) contains several timing programs designed to show the performance of ACML when running on your machine. Again, a GNUmakefile may be used to build and run them.

Depending on where your copy of the ACML is installed, and which compiler and flags you wish to use, it may be necessary to modify some variables in the GNUmakefile before using it.

The 64-bit Windows versions of ACML assume that you have the Cygwin UNIX-like tools installed, and can use the *make* command that comes with them to build the examples.

In addition, the GNUmakefile uses the gnuplot plotting program to display graphs of the timing results. If you do not have gnuplot installed, the timing programs will still run and show their results, but you will see no graph plots. Under linux, gnuplot may come with your linux distribution, but you may need to explicitly ask for it to be installed. Note that version 4.0 or later of gnuplot is required.

The gnuplot program is also available for Windows machines. See `http://www.gnuplot.info` for more information.

If you are on an SMP (multiprocessor) machine and have installed an OpenMP version of the ACML, then in the examples/performance directory a command such as

```
% make OMP_NUM_THREADS=5
```

will run the timing programs on P processors, where P = 1, 2, 4, 5; i.e., P equals an integer power of 2 and also equals *OMP_NUM_THREADS* if this value is not a power of 2. The results for a particular routine are concatenated into one file. gnuplot then shows on one graph for each routine the results of varying the number of processors for that routine.

Setting *OMP_NUM_THREADS* in this way is not useful if you are not on an SMP machine or are not using an OpenMP version of ACML. Neither is it useful to set *OMP_NUM_THREADS* to a value higher than the number of processors (or processor cores) on your machine. A way to find the number of processors (or cores) under linux is to examine the special file /proc/cpuinfo which has an entry for every core.

Not all routines in ACML are SMP parallelized, so in this context the *OMP_NUM_THREADS* setting only applies to those examples, including time_cfft2d.f90, time_dgemm.f90 and time_dgetrf.f90, which are for parallelized routines. The other timing programs run on one thread regardless of the setting of *OMP_NUM_THREADS*.

In all cases, timing graphs can be viewed without regenerating timing results by typing the command

```
% make plots
```

Note that all results generated by timing programs will vary depending on the load on your machine at run time.

2.10 ACML Memory Usage

Many ACML routines make use of allocated memory as temporary workspace during computation. Normally this workspace is freed as soon as it is no longer required, just before exit from the ACML routine that allocated it.

In general this allocatiom and freeing of memory does not make much difference to the performance of ACML. However, in some cases it can make a difference; for example, where a kernel routine which allocates memory is repeatedly called by a higher level routine to solve subproblems. This applies particularly to some LAPACK routines which make many calls to the matrix-matrix multiply routine DGEMM.

For this reason, we have introduced into ACML a system whereby the memory allocation mechanism used by most Level 3 and some Level 2 BLAS routines may avoid freeing memory, instead re-using memory allocated previously. This feature, called fast malloc, is now implemented in both Linux and Windows builds of ACML.

By default, beginning with ACML 5.3.0, the scheme is turned *on*. If you wish to turn if off, you will need to explicitly switch it off. This can be accomplished by means of an environment variable. The variable is named `ACML_FAST_MALLOC`, and can be turned off under Linux like this

```
export ACML_FAST_MALLOC=0
```

(if you use bash or a similar shell), or like this:

```
setenv ACML_FAST_MALLOC 0
```

(if you use csh, tcsh or similar). Any other value will leave the feature enabled.

At run-time, programs linked with ACML which use e.g. DGEMM, directly or indirectly, will use the new scheme. Level 2 routines that will use the fast malloc feature are CGEMV, CGERC, CGERU, CTRMV, CTRSV, DGEMV, DGER, DTRMV, DTRSV,

SGEMV, SGER, STRMV, STRSV, ZGEMV, ZGERC, ZGERU, ZTRMV, ZTRSV. Level 3 routines that will use the fast malloc feature are DGEMM, SGEMM, and ZGEMM.

Since with this scheme memory is not always freed, there is an obvious penalty to be paid - some of your machine's memory will be tied up while your program is running and will not be available for other use. The default amount of memory that will be assigned is 10 Megabytes (actually 10,000,000 bytes) for each thread. If the problem sizes require more workspace memory than is currently allocated, beginning with ACML 5.3.1, the previous buffer will be freed and a new buffer allocated that is large enough, and this new larger buffer will be retained. This means that the most memory that will be used may increase above the previous limit of 10 MB per thread, but it should reach a maximimum value that depends on the problem and the number of threads.

You can change the default values of the intial allocation size through the use of environment variables. These are `ACML_FAST_MALLOC_CHUNK_SIZE` which can be used to change the default chunk size from 10 Megabytes to any other positive value, given in bytes, and `ACML_FAST_MALLOC_MAX_CHUNKS` which sets the maximum number of chunks that may be used. You should use care when changing these values - if you accidentally set them too large and your program uses enough threads you may find that your program fails due to overallocation.

There is one final environment variable relevant to the fast memory allocation scheme. This is `ACML_FAST_MALLOC_DEBUG`. If you set this environment variable to any value, then ACML will print various messages showing the memory being used under the scheme. Note that other memory may still be allocated and freed in the normal way by ACML - such allocation will not lead to messages. This environment variable is designed to let you verify that the scheme is actually being used. In normal use you would definitely want `ACML_FAST_MALLOC_DEBUG` not to be set.

2.11 ACML Thread Binding

Beginning with ACML 5.3.1, a new CPU affinity method is used by the OpenMP versions of the library. This method is used for DGEMM, SGEMM, and ZGEMM. The first time one of these routines is called by a program, each OpenMP thread will be bound to a specific CPU core. Core binding starts with core 0 and increments up to a maximum of 64 threads. In the Windows builds of ACML, this binding is implemented using SetThreadGroupAffinity. In the Linux builds, pthread_setaffinity_np is used. The environment variable OMP_NUM_THREADS is used to control the total number of OpenMP threads.

3 BLAS: Basic Linear Algebra Subprograms

The BLAS are a set of well defined basic linear algebra operations ([1], [2], [3]). These operations are subdivided into three groups:

- Level 1: operations acting on vectors only (e.g. dot product)
- Level 2: matrix-vector operations (e.g. matrix-vector multiplication)
- Level 3: matrix-matrix operations (e.g. matrix-matrix multiplication)

Efficient machine-specific implementations of the BLAS are available for many modern high-performance computers. The implementation of higher level linear algebra algorithms on these systems depends critically on the use of the BLAS as building blocks. AMD provides, as part of the ACML, an implementation of the BLAS optimized for performance on AMD64 processors.

ACML 6 enables the heterogeneous computation of all BLAS Level 3 subroutines and two L2 subroutines (GEMV & SYMV) by dispatching work loads between the CPU and GPU. The GPU computation is done by calling the clBLAS library that ships with ACML6. The dispatching heuristic logic is controlled by ACML scripting language described in Chapter 7 [ACMLScript], page 188.

For any information relating to the BLAS please refer to the BLAS FAQ:

`http://www.netlib.org/blas/faq.html`

ACML also includes interfaces to the extensions to Level 1 BLAS known as the sparse BLAS. These routines perform operations on a sparse vector x which is stored in compressed form and a vector y in full storage form. See reference [4] for more information.

4 LAPACK: Package of Linear Algebra Subroutines

4.1 Introduction to LAPACK

LAPACK ([5]) is a library of FORTRAN 77 subroutines for solving commonly occurring problems in numerical linear algebra. LAPACK components can solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. Dense and banded matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices.

LAPACK routines are written so that as much as possible of the computations is performed by calls to the BLAS. The efficiency of LAPACK routines depends, in large part, on the efficiency of the BLAS being called. Block algorithms are employed wherever possible to maximize the use of calls to level 3 BLAS, which generally run faster than lower level BLAS due to the high number of operations per memory access.

The performance of some of the LAPACK routines has been further improved by reworking the computational algorithms. Some of the LAPACK routines contained in ACML are therefore based on code that is different from the LAPACK sources available in the public domain. In all these cases the algorithmic and numerical properties of the original LAPACK routines have been strictly preserved. Furthermore, key LAPACK routines have been treated using OpenMP to take advantage of multiple processors when running on SMP machines. Your application will automatically benefit when you link with the OpenMP versions of ACML.

4.2 Reference sources for LAPACK

The LAPACK homepage can be accessed on the World Wide Web via the URL address:

`http://www.netlib.org/lapack/`

The on-line version of the Lapack User's Guide, Third Edition ([5]) is available from this homepage, or directly using the URL:

`http://www.netlib.org/lapack/lug/index.html`

The standard source code is available for download from netlib, with separate distributions for UNIX/Linux and Windows® installations:

`http://www.netlib.org/lapack/lapack.tgz`
`http://www.netlib.org/lapack/lapack-pc.zip`

A list of known problems, bugs, and compiler errors for LAPACK, as well as an errata list for the LAPACK User's Guide ([5]), is maintained on netlib

`http://www.netlib.org/lapack/release_notes`

A LAPACK FAQ (Frequently Asked Questions) file can also be accessed via the LAPACK homepage

`http://www.netlib.org/lapack/faq.html`

4.3 LAPACK block sizes, ILAENV and ILAENVSET

As described in Section 6.2 of the LAPACK User's Guide, block sizes and other parameters used by various LAPACK routines are returned by the LAPACK inquiry function ILAENV. In ACML, values returned by ILAENV have been chosen to achieve very good performance on a wide variety of hardware and problem sizes.

In general it is unlikely that you will want or need to be concerned with these parameters. However, in some cases it may be that a default value returned by ILAENV is not optimal for your particular hardware and problem size. Following the advice in the LAPACK User's Guide may enable you to choose a better value in some circumstances.

For convenience, ACML includes a subroutine which allows you to override default values returned by ILAENV if you have superior knowledge. The routine is named ILAENVSET and has the following specification.

`ILAENVSET (`*`ISPEC,NAME,OPTS,N1,N2,N3,N4,NVALUE,INFO`*`)` [SUBROUTINE]

`INTEGER ISPEC` [Input]
On input: *ISPEC* specifies the parameter to be set (see Section 6.2 of the LAPACK User's Guide for details).

`CHARACTER*(*) NAME` [Input]
On input: *NAME* specifies the name of the LAPACK subroutine for which the parameter is to be set.

`CHARACTER*(*) OPTS` [Input]
On input: *OPTS* is a character string of options to the subroutine.

`INTEGER N1,` *N2, N3, N4* [Input]
On input: *N1*, *N2*, *N3* and *N4* are problem dimensions. A value of -1 means that the dimension is unused or irrelevant.

`INTEGER NVALUE` [Input]
On input: *NVALUE* is the value to be set for the parameter specified by *ISPEC*. This value will be retrieved by any future call of `ILAENV` with similar arguments, including the call of `ILAENV` coming directly from the routine specified by argument *NAME*. In most cases, but not all, the value set will apply irrespective of the values of arguments *OPTS*, *N1*, *N2*, *N3* and *N4*.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

All arguments of `ILAENVSET` apart from the last two, *NVALUE* and *INFO*, are identical to the arguments of `ILAENV`. `ILAENVSET` should be called *before* you call the LAPACK routine in question.

It should be noted that not all LAPACK routines make use of the `ILAENV` mechanism (because not all routines use blocked algorithms or require other tuning parameters). Calls of `ILAENVSET` with argument *NAME* set to the name of such a routine will fail with *INFO=0*. In addition, the ACML versions of some important routines that do use blocked algorithms, such as the QR factorization routine `DGEQRF`, bypass `ILAENV` because they make use of a different tuning system which is independent of standard LAPACK. For all such routines,

`ILAENVSET` can still be called with no error exit, but calls will have no effect on performance of the routine.

Below we give examples of how to call `ILAENVSET` in both FORTRAN and C.

Example (FORTRAN code):

```
      INTEGER ILO, IHI, INFO, N, NS
      CHARACTER COMPZ, JOB
      INTEGER ILAENV
      EXTERNAL ILAENV, ILAENVSET
      JOB = 'E'
      COMPZ = 'I'
      N = 512
      ILO = 1
      IHI = 512
C     Check the default shift parameter (ISPEC=4) used by DHSEQR
      NS = ILAENV(4, 'DHSEQR', JOB//COMPZ, N, ILO, IHI, -1)
      WRITE (*,*) 'Default NS = ', NS
C     Set a new value 5 for the shift parameter
      CALL ILAENVSET(4, 'DHSEQR', JOB//COMPZ, N, ILO, IHI, -1, 5, INFO)
C     Then check the shift parameter again
      NS = ILAENV(4, 'DHSEQR', JOB//COMPZ, N, ILO, IHI, -1)
      WRITE (*,*) 'Revised NS = ', NS
      END
```

Example (C code):

```
#include <acml.h>
#include <stdio.h>
int main(void)
{
  int n=512, ilo=1, ihi=512, ns, info;
  char compz = 'I', job = 'E', opts[3];
  opts[0] = job;
  opts[1] = compz;
  opts[2] = '\0';
/* Check the default shift parameter (ISPEC=4) used by DHSEQR */
  ns = ilaenv(4, "DHSEQR", opts, n, ilo, ihi, -1);
  printf("Default ns = %d\n", ns);
/* Set a new value 5 for the shift parameter */
  ilaenvset(4, "DHSEQR", opts, n, ilo, ihi, -1, 5, &info);
/* Then check the shift parameter again */
  ns = ilaenv(4, "DHSEQR", opts, n, ilo, ihi, -1);
  printf("Revised ns = %d\n", ns);
  return 0;
}
```

4.4 IEEE exceptions and LAPACK

Some LAPACK eigensystem routines (namely CHEEVR, DSTEVR, DSYEVR, SSTEVR, SSYEVR, ZHEEVR) are able to take advantage of a faster algorithm when the full eigenspectrum is requested on machines which conform to the IEEE-754 floating point standard [14].

Normal execution of the faster algorithm (implemented by LAPACK routines SSTEGR and DSTEGR, which are called by the routines mentioned above) may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner. This may depend upon the compiler flags used to compile and link the main program.

The LAPACK routine ILAENV, called with ISPEC = 10 or 11, states whether or not NaNs or infinities respectively will cause a trap. In ACML, by default ILAENV assumes that NaNs and infinities cause traps, even if this reduces the performance of the eigensystem routines. This is because it is not possible in general to reliably check whether they do trap or not at run-time. The intention is to ensure that these routines always function correctly, irrespective of how the main program calling ACML is compiled.

However, if your main program is compiled in such a way that NaNs and infinities do not cause traps, the ACML-specific routine ILAENVSET (see Section 4.3 [ILAENV-ILAENVSET], page 17) may be used to override the default operative mode of ILAENV, and allow the xxxEVR routines to use the faster xSTEGR algorithm when calculating the full eigenspectrum. When used for this purpose, ILAENVSET should be called as follows:

```
CALL ILAENVSET(10,'X','X',0,0,0,0,1,INFO)
CALL ILAENVSET(11,'X','X',0,0,0,0,1,INFO)
```

(or the C equivalent).

It is important to note that if you use ILAENVSET in this way before calling an xxxEVR routine, but your program *does* trap on IEEE exceptions, then there is a chance that your program will terminate unexpectedly. You should consult the documentation for the compiler you are using to find out whether there are compiler flags controlling this.

4.5 Progress monitoring function: ACML_PROGRESS

Sometimes when using LAPACK routines to solve very large problems it may be convenient to know how far through the solution ACML has got. For this purpose ACML uses the function `ACML_PROGRESS`.

A default version of `ACML_PROGRESS` is contained in ACML. You may override the default version by compiling and linking in your own version of the routine. (Note that this may not be possible with some versions of ACML, in particular with Microsoft Windows DLLs which do not allow a routine to be overridden in this manner).

`FUNCTION` *ACML_PROGRESS* [INTEGER]
(*THREAD*,*NTHREADS*,*STEP*,*PERCENT*,*STAGE*)

`INTEGER THREAD` [Input]
On input: the number of the thread that `ACML_PROGRESS` is called from. For sequential code this will be 0.

`INTEGER NTHREADS` [Input]
On input: the number of the threads in the parallel region. For sequential code this will be 1.

`INTEGER STEP` [Input]
On input: *STEP* is the linear progress indicator that shows the amount of work done. Increases from 0 to the linear size of the problem during the computation.

`INTEGER PERCENT` [Input]
On input: *PERCENT* is the (approximate) percentage of work done so far towards solving the whole problem.

`CHARACTER*(*) STAGE` [Input]
On input: *STAGE* is the name of the LAPACK routine from which `ACML_PROGRESS` is called.

`ACML_PROGRESS function` *return value.* [Output]
On output: set *ACML_PROGRESS* to 0 to continue, or set to any non-zero value to terminate execution of the LAPACK routine on this thread.

The default version of `ACML_PROGRESS` looks like this:

```
      INTEGER FUNCTION ACML_PROGRESS(THREAD,NTHREADS,STEP,PERCENT,STAGE)
C     .. Scalar Arguments ..
      INTEGER          STEP, THREAD, NTHREADS, PERCENT
      CHARACTER*(*)    STAGE
C
C   This function gets called from some ACML LAPACK routines to
C   allow the user's program to monitor progress.
C
C    THREAD: INTEGER
C      The number of the thread from which ACML_PROGRESS is called.
C      0 is passed for sequential code.
C
C    NTHREADS: INTEGER
C      The number of threads in the parallel region.
C
C    STEP: INTEGER
C      The linear progress indicator that shows the amount of work done.
C      Increases from 0 to the linear size of the problem during the
C      computation.
C
C    PERCENT: INTEGER
C      The (approximate) percentage of work done towards solving the
C      whole problem.
C
C    STAGE: CHARACTER*(*)
C      The name of the LAPACK routine from which ACML_PROGRESS
C      is called.
C
C    Function return value:
C      Set ACML_PROGRESS = 0 to continue, or set to any non-zero
C      value to terminate execution of the LAPACK routine.
C
C     This is the default version of ACML_PROGRESS. Users may override
C     it by supplying their own version.
C
C     .. Executable Statements ..
C
C     Sample usage of ACML_PROGRESS:
C
C      IF (THREAD.EQ.0) THEN
C         WRITE (*,99) 'ACML_PROGRESS: thread = ', THREAD, ' / ',
C     *      NTHREADS, ', stage:' // STAGE(:LEN_TRIM(STAGE)) //
C     *      ', step:', STEP, ', percent: ', percent
C  99     FORMAT(1X,A,I0,A,I0,A,I0,A,I0)
C      END IF
C
      ACML_PROGRESS = 0
      RETURN
      END
```

You may use this as a template for your own version of `ACML_PROGRESS`. An example of the kind of things you might do is given as comments in the code above.

Note that the interface shown above is Fortran-style. You may wish to write your version of the function in C rather than Fortran. If so, you must follow the rules of mixed-language programming on your system. A C version of `ACML_PROGRESS` for use with Linux versions of ACML might look like this:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int acml_progress_(int *thread, int *nthreads, int *step,
                   int *percent, char *stage, int len_stage)
{
  /* Sample usage of ACML_PROGRESS: */
  if (*thread == 0)
  {
    char *stagebuff = (char *)malloc(len_stage+1);
    strncpy(stagebuff, stage, len_stage);
    stagebuff[len_stage] = '\0';
    printf(
      "ACML_PROGRESS: thread = %d / %d, stage: %s, step: %d, percent: %d\n",
      *thread, *nthreads, stagebuff, *step, *percent);
    free(stagebuff);
  }
  return 0;
}
```

Note that the character string input *stage* is *not* NULL-terminated, and its length is passed by value as argument *len_stage*. All other arguments are passed by reference - this is the norm when being called from a Fortran program.

Currently, `ACML_PROGRESS` only works in conjunction with the LAPACK LU factorization routines CGETRF, DGETRF, SGETRF and ZGETRF.

5 Fast Fourier Transforms (FFTs)

5.1 FFTW Interface

There are no established FFT API standards in the library world as there are for BLAS and LAPACK interfaces, but FFTW has become what can be treated as a defacto-standard just based on its popularity. ACML is now providing the FFTW3 header file and our own implementation of the FFTW library that users can link with, `acml_fftw`. Through the use of scripting logic described in section Chapter 7 [ACMLScript], page 188 below, ACML will dispatch work between a user provided FFTW shared library and the clFFT library that ships with ACML 6.

The user provided FFTW implementation will be chosen to run host side computation, and the clFFT implementation will be chosen to run device (GPU) side computation. It is expected for the user to provide the host based shared library implementation, because the clFFT implementation only supports a limited set of features at this time, and if the host program calls a function with parameters that is not supported, `acml_fftw` needs to be able to call the host based FFTW or it will return an error message. `acml_fftw` is not using the traditional ACML FFT's to avoid bringing in Fortran dependencies to the users program. It is still possible to use only `acml_fftw` and perform FFTs if the transform is supported by the underlying clFFT library.

ACML expects to find the proper CPU based FFTW libraries in the `PATH` on windows based systems, or in the `LD_LIBRARY_PATH` on Linux based systems. ACML looks for both the single precision and the double precision version of the shared libraries, and loads them both dynamically at runtime if available.

For linux, ACML looks for the multi-threaded version of the FFTW shared libraries in addition to the main library

– Single Precision: `libfftw3f.so`, `libfftw3f_threads.so`
– Double Precision: `libfftw3.so`, `libfftw3_threads.so`

For windows, ACML is looking for libraries with the same naming convention as those precompiled libraries available from its website. For windows, the threading support is built straight into the

– Single Precision: `libfftw3f-3.dll`
– Double Precision: `libfftw3-3.dll`

ACML now ships with the FFTW header file packaged inside of the `ACML_ROOT/include` folder. Users may include that file and link with the `acml_fftw` library. ACML 6 does not support all of the available API's yet. The subset that are supported are documented with links below:

- Basic complex : Complex DFT's (`http://www.fftw.org/fftw3_doc/Complex-DFTs.html#Complex-DFTs`)
- Advanced complex : Advanced Complex DFT's (`http://www.fftw.org/fftw3_doc/Advanced-Complex-DFTs.html#Advanced-Complex-DFTs`)
- Basic real : `http://www.fftw.org/fftw3_doc/Real_002ddata-DFTs.html#Real_002ddata-DFTs`
- Advanced real : `http://www.fftw.org/fftw3_doc/Advanced-Real_002ddata-DFTs.html#Advanced-Real_002ddata-DFTs`

- Guru: not yet supported
- Threaded: Multi-threaded Complex DFT's (`http://www.fftw.org/fftw3_doc/Usage-of-Multi_002dthreaded-FFTW.html#Usage-of-Multi_002dthreaded-FFTW`), all API's
- Using Plans: Execute/Destroy/Cleanup Plans (`http://www.fftw.org/doc/Using-Plans.html#Using-Plans`)

5.2 Introduction to FFTs

There are two main types of Discrete Fourier Transform (DFT):

- routines for the transformation of complex data: in the ACML, these routines have names beginning with `ZFFT` or `CFFT`, for double and single precision, respectively;
- routines for the transformation of real to complex data and vice versa: in the ACML the names for the former begin with `DZFFT` or `SCFFT`, for double and single precision, respectively; the names for the latter begin with `ZDFFT` or `CSFFT`.

The following subsections provide definitions of the DFT for complex and real data types, and some guidelines on the efficient use of the ACML FFT routines.

5.2.1 Transform definitions and Storage for Complex Data

The simplest transforms to describe are those performed on sequences of complex data. Such data are stored as arrays of type complex. The result of a complex FFT is also a complex sequence of the same length and, for the simple interfaces, is written back to the original array. Where multiple (m, say), same-length sequences (of length n) of complex data are to be transformed, the sequences are held in a single complex array; in the simple interfaces the array will be of length $m * n$ containing m end-to-end sequences and the results of the m FFTs are returned in the original array. Expert interfaces are provided which give: greater flexibility in the storage of the original data and results, user provided scaling, and whether results should be written to a separate array or not.

The definition of a complex DFT used here is given by:

$$\tilde{x}_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} x_k \exp\left(\pm i \frac{2\pi jk}{n}\right) \text{ for } j = 0, 1, \ldots, n-1$$

where x_k are the complex data to be transformed, $\tilde{x}_j$ are the transformed data, and the sign of $\pm$ determines the direction of the transform: $(-)$ for forward and $(+)$ for backward.

Note that, in this definition, both directional transforms have the same scaling and performing both consecutively recovers the original data; this is the prescribed scaling provided in the simple FFT interfaces, whereas, in the expert interfaces, the scaling factor must be supplied by the user.

For the simple interfaces, a two dimensional array of complex data, with m rows and n columns is stored in the same order as a set of n sequences of length m (as described above). That is, column elements are stored contiguously and the first element of the next column follows the last element of the current column. In the expert interfaces, column elements may be separated by a fixed step length (increment) while row elements may be separated by a second increment; if the first increment is 1 and the second increment is m then we have the same storage as in the simple interface.

The definition of a complex 2D DFT used here is given by:

$$\tilde{x}_{jp} = \frac{1}{\sqrt{m*n}} \sum_{l=0}^{m-1} \sum_{k=0}^{n-1} x_{kl} \exp\left(\pm i \frac{2\pi jk}{n}\right) \exp\left(\pm i \frac{2\pi pl}{m}\right)$$

for $j = 0, 1, \ldots, n-1$ and $p = 0, 1, \ldots, m-1$, where x_{kl} are the complex data to be transformed, $\tilde{x}_{jp}$ are the transformed data, and the sign of $\pm$ determines the direction of the transform.

5.2.2 Transform definitions and Storage for Real Data

The DFT of a sequence of real data results in a special form of complex sequence known as a Hermitian sequence.

If the original sequence is purely real valued, i.e. $z_j = x_j$, then the definition of the real DFT used here is given by:

$$\tilde{z}_j = a_j + ib_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} x_k \exp\left(-i \frac{2\pi jk}{n}\right) \text{ for } j = 0, 1, \ldots, n-1$$

where x_k are the real data to be transformed, $\tilde{z}_j$ are the transformed complex data.

In full complex representation, the Hermitian sequence would be a sequence of n complex values $\tilde{z}_j$ for $j = 0, 1, ..., n-1$, where $\tilde{z}_{n-j}$ is the complex conjugate of $\tilde{z}_j$ for $j = 1, 2, ..., (n-1)/2$; $\tilde{z}_0$ is real valued; and, if n is even, $\tilde{z}_{n/2}$ is also real valued. The symmetries defining Hermitian sequence mean that it can be stored using reduced amount of memory.

In ACML, there are two storage formats for the representation of Hermitian sequences. The first format, called Hermitian-packed format, is used on output from `DZFFT` routines and on input to `ZDFFT` routines and is defined as follows: let X be an array of length n and with first index 0,

- $X(j)$ contains the real part of $\tilde{z}_j$ (i.e. a_j) for $j = 0, ..., n/2$
- $X(n-j)$ contains the imaginary part of $\tilde{z}_j$ (i.e. b_j) for $j = 1, ..., (n-1)/2$

As seen, a Hermitian sequence can be fully represented by a set of n real values, where n is the length of the original real sequence. It is therefore conventional for the array containing the real sequence to be overwritten by such a representation of the transformed Hermitian sequence.

An alternative way to store the Hermitian sequence, called complex-Hermitian format, is to keep the complex representation, store only the first $n/2+1$ complex numbers and drop the remaining elements that contain redundant information. This approach can be conveniently extended to multiple dimensions. For example, a 3D real data set of size $l*m*n$ can be transformed into Fourier space as a complex data set of size $(l/2+1)*m*n$. It is still possible to use $l*m*n$ real numbers to store all Fourier coefficients but intepreting such information would be extremely difficult. The 16 new subroutines (such as `DZFFT1D/1M/2D/3D`) introduced in version 5.0 of ACML all use the complex-Hermitian format.

The following table shows the storage of a one-dimensional Hermitian sequence with 8 elements.

Real Input x_j	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
Complex Output $\tilde{z}_j$ = $a_j + ib_j$	$\tilde{z}_0$	$\tilde{z}_1$	$\tilde{z}_2$	$\tilde{z}_3$	$\tilde{z}_4$	$\tilde{z}_5 = \tilde{z}_3^*$	$\tilde{z}_6 = \tilde{z}_2^*$	$\tilde{z}_7 = \tilde{z}_1^*$
Hermitian-Packed Storage	a_0	a_1	a_2	a_3	a_4	b_3	b_2	b_1
Complex-Hermitian Storage	$\tilde{z}_0$	$\tilde{z}_1$	$\tilde{z}_2$	$\tilde{z}_3$	$\tilde{z}_4$	–	–	–

Given a Hermitian sequence, the inverse discrete transform can be defined as:

$$x_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \tilde{z}_k \exp\left(i\frac{2\pi jk}{n}\right) \text{ for } j = 0, 1, \ldots, n-1$$

or if written using the real and complex components:

$$x_j = \frac{1}{\sqrt{n}} \left(a_0 + 2 \sum_{k=1}^{n/2-1} \left(a_k \cos\left(\frac{2\pi jk}{n}\right) - b_k \sin\left(\frac{2\pi jk}{n}\right)\right) + a_{n/2} \right)$$

where $a_{n/2} = 0$ if n is odd, and $\tilde{z}_k = a_k + ib_k$ is the Hermitian sequence to be transformed.

Note that, in the second definition above, the inverse transform has a negative sign in the exponent. So if a Hermitian sequence is stored in a Hermitian-packed format (for DZFFT, DZFFTM, SCFFT and SCFFTM), performing a forward and a backward transform consecutively does not recover the original data. To recover original real data, or otherwise to perform an inverse transform on a set of Hermitian data, the Hermitian data must be conjugated prior to performing the transform (i.e. changing the sign of the stored imaginary parts). This however does not apply to the remaining subroutines using the complex-Hermitian data storage.

5.2.3 Efficiency

The efficiency of the FFT is maximized by choosing the sequence length to be a power of 2. Good efficiency can also be achieved when the sequence length has small prime factors, up to a factor 13; however, the time taken for an FFT increases as the size of the prime factor increases.

5.2.4 Default and Generated Plans

For those FFT routines that can be initialized prior to computing the FFTs, the initialization can be performed in one of two ways. In either case, initialization involves the storing of the factorization of N, and the twiddle factors associated with this factorization, in the communication array *COMM*.

The simpler way to initialize is by setting the argument *MODE* to zero. This means that a default plan, for the given input dimensions, is used to calculate the FFT. This has the advantage that the initialization phase is very quick and is generally a small fraction of the time required to perform the FFT computation. However, for some problem dimensions the default plan may not be optimal, especially where there is a mixture of prime factors.

Under some circumstances, optimality of performance of an FFT computation may be crucial. For example, where a very large number of FFTs are to be performed on problems of a fixed size (e.g. N remains the same), then it is best to initialize by setting the argument *MODE* to 100. This will time a number of plans (this number can be quite large when N

has a significant number of prime factors) and initialize using the plan with the best time. Using this form of initialization can, potentially, lead to significant improvements in the performance of the FFT computation for the given dimensions.

Where problem dimensions will not change over a number of runs of a program, the communication array could, for example, be written out to a file during an initialization run, and then read in from the same file on subsequent computation runs. This would be effective for problem dimensions that have a large number of possible plans (factor orderings and groupings) and therefore take a significant amount of time to find the optimal plan.

Please consult the individual FFT routine documents to determine whether plan generation is enabled.

5.3 FFTs on Complex Sequences

5.3.1 FFT of a single sequence

The routines documented here compute the discrete Fourier transform (DFT) of a sequence of complex numbers in either single or double precision arithmetic. The DFT is computed using a highly-efficient FFT algorithm. There are two sets of interfaces available: simple drivers and expert drivers. The simple drivers perform in-place transforms on data held contiguously in memory using a fixed scaling factor; these are simpler to use and are sufficient for many problems. The expert drivers offer greater flexibility by including a number of additional arguments. These allow you to control: the scaling factor applied; whether the result should be output to a separate vector; and, the increments used in storing successive elements of both the input sequence and the result.

ZFFT1D Routine Documentation

ZFFT1D (*MODE*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZFFT1D. On input:

- *MODE*=0 : only default initializations (specific to N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1D.
- *MODE*=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1D.
- *MODE*=−2 : initializations and a forward transform are performed.
- *MODE*=2 : initializations and a backward transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER N [Input]

On input: N is the length of the complex sequence X

COMPLEX*16 X(N) [Input/Output]

On input: X contains the complex sequence of length N to be transformed.
On output: X contains the transformed sequence.

COMPLEX*16 COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL ZFFT1D(0,N,X,COMM,INFO)
      CALL ZFFT1D(-1,N,X,COMM,INFO)
      CALL ZFFT1D(-1,N,Y,COMM,INFO)
      DO 10 I = 1, N
         X(I) = X(I)*DCONJG(Y(I))
10    CONTINUE
      CALL ZFFT1D(1,N,X,COMM,INFO)
```

CFFT1D Routine Documentation

CFFT1D (*MODE*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CFFT1D. On input:

- *MODE*=0 : only default initializations (specific to N) are performed; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1D.
- *MODE*=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1D.
- *MODE*=−2 : (default) initializations and a forward transform are performed.
- *MODE*=2 : (default) initializations and a backward transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER N [Input]

On input: N is the length of the complex sequence X

COMPLEX X(N) [Input/Output]

On input: X contains the complex sequence of length N to be transformed.
On output: X contains the transformed sequence.

COMPLEX COMM(5*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT1D(0,N,X,COMM,INFO)
      CALL CFFT1D(-1,N,X,COMM,INFO)
      CALL CFFT1D(-1,N,Y,COMM,INFO)
      DO 10 I = 1, N
         X(I) = X(I)*CONJG(Y(I))
10    CONTINUE
      CALL CFFT1D(1,N,X,COMM,INFO)
```

ZFFT1DX Routine Documentation

ZFFT1DX (*MODE*,*SCALE*,*INPL*,*N*,*X*,*INCX*,*Y*,*INCY*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZFFT1DX.
On input:

- *MODE*=0 : only initializations (specific to the value of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1DX.
- *MODE*=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1DX.
- *MODE*=−2 : (default) initializations and a forward transform are performed.
- *MODE*=2 : (default) initializations and a backward transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of N) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of N.

DOUBLE PRECISION SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequence

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequence; otherwise the output sequence is returned in Y.

INTEGER N [Input]

On input: N is the number of elements to be transformed

COMPLEX*16 X(1+(N-1)*INCX) [Input/Output]

On input: X contains the complex sequence of length N to be transformed, with the ith element stored in X(1+(i-1)*INCX).
On output: if *INPL* is .TRUE. then X contains the transformed sequence in the same locations as on input; otherwise X remains unchanged.

INTEGER INCX [Input]

On input: *INCX* is the increment used to store successive elements of a sequence in X.
Constraint: *INCX* > 0.

COMPLEX*16 Y(1+(N-1)*INCY) [Output]

On output: if *INPL* is .FALSE. then Y contains the transformed sequence, with the ith element stored in Y(1+(i-1)*INCY); otherwise Y is not referenced.

`INTEGER INCY` [Input]

On input: *INCY* is the increment used to store successive elements of a sequence in *Y*. If *INPL* is .TRUE. then *INCY* is not referenced.
Constraint: *INCY* > 0.

`COMPLEX*16 COMM(3*N+100)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward FFTs are performed unscaled and in-place on contiguous
C     vectors X and Y following initialization. Manipulations on
C     resultant Fourier coefficients are stored in X which is then
C     transformed back.
C
        SCALE = 1.0D0
        INPL = .TRUE.
        CALL ZFFT1DX(0,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL ZFFT1DX(-1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL ZFFT1DX(-1,SCALE,INPL,N,Y,1,DUM,1,COMM,INFO)
        DO 10 I = 1, N
           X(I) = X(I)*DCONJG(Y(I))/DBLE(N)
   10   CONTINUE
        CALL ZFFT1DX(1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
```

CFFT1DX Routine Documentation

CFFT1DX (*MODE*,*SCALE*,*INPL*,*N*,*X*,*INCX*,*Y*,*INCY*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CFFT1DX.
On input:

- *MODE*=0 : only initializations (specific to the value of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=-1 or 1.
- *MODE*=-1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1DX.
- *MODE*=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to CFFT1DX.
- *MODE*=-2 : (default) initializations and a forward transform are performed.
- *MODE*=2 : (default) initializations and a backward transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of N) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of N.

REAL SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequence

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequence; otherwise the output sequence is returned in Y.

INTEGER N [Input]

On input: N is the number of elements to be transformed

COMPLEX X(1+(N-1)*INCX) [Input/Output]

On input: X contains the complex sequence of length N to be transformed, with the ith element stored in X(1+(i-1)*INCX).
On output: if *INPL* is .TRUE. then X contains the transformed sequence in the same locations as on input; otherwise X remains unchanged.

INTEGER INCX [Input]

On input: *INCX* is the increment used to store successive elements of a sequence in X.
Constraint: *INCX* > 0.

COMPLEX Y(1+(N-1)*INCY) [Output]

On output: if *INPL* is .FALSE. then Y contains the transformed sequence, with the ith element stored in Y(1+(i-1)*INCY); otherwise Y is not referenced.

`INTEGER INCY` [Input]

On input: *INCY* is the increment used to store successive elements of a sequence in *Y*. If *INPL* is .TRUE. then *INCY* is not referenced.
Constraint: *INCY* > 0.

`COMPLEX COMM(5*N+100)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward FFTs are performed unscaled and in-place on contiguous
C     vectors X and Y following initialization. Manipulations on
C     resultant Fourier coefficients are stored in X which is then
C     transformed back.
C
        SCALE = 1.0
        INPL = .TRUE.
        CALL CFFT1DX(0,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL CFFT1DX(-1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
        CALL CFFT1DX(-1,SCALE,INPL,N,Y,1,DUM,1,COMM,INFO)
        DO 10 I = 1, N
           X(I) = X(I)*CONJG(Y(I))/REAL(N)
   10   CONTINUE
        CALL CFFT1DX(1,SCALE,INPL,N,X,1,DUM,1,COMM,INFO)
```

5.3.2 FFT of multiple complex sequences

The routines documented here compute the discrete Fourier transforms (DFTs) of a number of sequences of complex numbers in either single or double precision arithmetic. The sequences must all have the same length. The DFTs are computed using a highly-efficient FFT algorithm. There are two sets of interfaces available: simple drivers and expert drivers. The simple drivers perform in-place transforms on data held contiguously in memory using a fixed scaling factor; these are simpler to use and are sufficient for many problems. The expert drivers offer greater flexibility by including a number of additional arguments. These allow you to control: the scaling factor applied; whether the result should be output to a separate vector; the increments used in storing successive elements of a given sequence (for both input and output sequences); and the increments used in storing corresponding elements in successive sequences (for both input and output).

ZFFT1M Routine Documentation

ZFFT1M (*MODE*,*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZFFT1M. On input:

- *MODE*=0 : only initializations (specific to the value of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=-1 or 1.
- *MODE*=-1 : forward transforms are performed. Initializations are assumed to have been performed by a prior call to ZFFT1M.
- *MODE*=1 : backward (reverse) transforms are performed. Initializations are assumed to have been performed by a prior call to ZFFT1M.
- *MODE*=-2 : (default) initializations and forward transforms are performed.
- *MODE*=2 : (default) initializations and backward transforms are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of N) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of N.

INTEGER M [Input]

On input: M is the number of sequences to be transformed.

INTEGER N [Input]

On input: N is the length of the complex sequences in X

COMPLEX*16 X(N*M) [Input/Output]

On input: X contains the M complex sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed sequences.

COMPLEX*16 COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = $-$i on exit, the i-th argument had an illegal value.

Example:

```
      CALL ZFFT1M(0,1,N,X,COMM,INFO)
      CALL ZFFT1M(-1,2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*DCONJG(X(I,2))
         X(I,2) = DCMPLX(0.0D0,1.0D0)*X(I,2)
10    CONTINUE
      CALL ZFFT1M(1,2,N,X(1,2),COMM,INFO)
```

CFFT1M Routine Documentation

CFFT1M (*MODE*,*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CFFT1M. On input:

- *MODE*=0 : only initializations (specific to the value of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=-1 or 1.
- *MODE*=-1 : forward transforms are performed. Initializations are assumed to have been performed by a prior call to CFFT1M.
- *MODE*=1 : backward (reverse) transforms are performed. Initializations are assumed to have been performed by a prior call to CFFT1M.
- *MODE*=-2 : (default) initializations and forward transforms are performed.
- *MODE*=2 : (default) initializations and backward transforms are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of N) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of N.

INTEGER M [Input]

On input: M is the number of sequences to be transformed.

INTEGER N [Input]

On input: N is the length of the complex sequences in X

COMPLEX X(N*M) [Input/Output]

On input: X contains the M complex sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed sequences.

COMPLEX COMM(5*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = $-$i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT1M(0,1,N,X,COMM,INFO)
      CALL CFFT1M(-1,2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*CONJG(X(I,2))
         X(I,2) = CMPLX(0.0D0,1.0D0)*X(I,2)
10    CONTINUE
      CALL CFFT1M(1,2,N,X(1,2),COMM,INFO)
```

ZFFT1MX Routine Documentation

ZFFT1MX (*MODE*,*SCALE*,*INPL*,*NSEQ*,*N*,*X*,*INCX1*,*INCX2*, [SUBROUTINE]
Y,*INCY1*,*INCY2*,*COMM*,*INFO*)

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZFFT1MX.
On input:

- *MODE*=0 : only initializations (specific to the value of *N*) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1MX.
- *MODE*=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT1MX.
- *MODE*=−2 : (default) initializations and a forward transform are performed.
- *MODE*=2 : (default) initializations and a backward transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of *N*) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of *N*.

DOUBLE PRECISION SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequences; otherwise the output sequences are returned in *Y*.

INTEGER NSEQ [Input]

On input: *NSEQ* is the number of sequences to be transformed

INTEGER N [Input]

On input: *N* is the number of elements in each sequence to be transformed

COMPLEX*16 X(1+(N-1)*INCX1+(NSEQ-1)*INCX2) [Input/Output]

On input: *X* contains the *NSEQ* complex sequences of length *N* to be transformed; the ith element of sequence j is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).
On output: if *INPL* is .TRUE. then *X* contains the transformed sequences in the same locations as on input; otherwise *X* remains unchanged.

INTEGER INCX1 [Input]

On input: *INCX1* is the increment used to store successive elements of a given sequence in *X* (*INCX1*=1 for contiguous data).
Constraint: *INCX1* > 0.

`INTEGER INCX2` [Input]

On input: *INCX2* is the increment used to store corresponding elements of successive sequences in *X* (*INCX2*=*N* for contiguous data).
Constraint: *INCX2* > 0.

`COMPLEX*16 Y(1+(N-1)*INCY1+(NSEQ-1)*INCY2)` [Output]

On output: if *INPL* is .FALSE. then *Y* contains the transformed sequences with the ith element of sequence j stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); otherwise *Y* is not referenced.

`INTEGER INCY1` [Input]

On input: *INCY1* is the increment used to store successive elements of a given sequence in *Y*. If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

`INTEGER INCY2` [Input]

On input: *INCY2* is the increment used to store corresponding elements of successive sequences in *Y* (*INCY2*=*N* for contiguous data). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraint: *INCY2* > 0.

`COMPLEX*16 COMM(3*N+100)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward FFTs are performed unscaled and in-place on two
C     contiguous vectors stored in the first two columns of X.
C     Manipulations are stored in 2nd and 3rd columns of X which are
C     then transformed back.
C
        COMPLEX *16 X(N,3)
        SCALE = 1.0D0
        INPL = .TRUE.
        CALL ZFFT1MX(0,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        CALL ZFFT1MX(-1,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        DO 10 I = 1, N
           X(I,3) = X(I,1)*DCONJG(X(I,2))/DBLE(N)
           X(I,2) = DCMPLX(0.0D0,1.0D0)*X(I,2)/DBLE(N)
   10   CONTINUE
        CALL ZFFT1MX(1,SCALE,INPL,2,N,X(1,2),1,N,DUM,1,N,COMM,INFO)
```

`CFFT1MX` Routine Documentation

`CFFT1MX` (*MODE*,*SCALE*,*INPL*,*NSEQ*,*N*,*X*,*INCX1*,*INCX2*, *Y*,*INCY1*,*INCY2*,*COMM*,*INFO*) [SUBROUTINE]

`INTEGER MODE` [Input]

The value of *MODE* on input determines the operation performed by `CFFT1MX`.
On input:

- *MODE*=0 : only initializations (specific to the value of *N*) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to `CFFT1MX`.
- *MODE*=1 : a backward (reverse) transform is performed. Initializations are assumed to have been performed by a prior call to `CFFT1MX`.
- *MODE*=−2 : (default) initializations and a forward transform are performed.
- *MODE*=2 : (default) initializations and a backward transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of *N*) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of *N*.

`REAL SCALE` [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

`LOGICAL INPL` [Input]

On input: if *INPL* is .TRUE. then *X* is overwritten by the output sequences; otherwise the output sequences are returned in *Y*.

`INTEGER NSEQ` [Input]

On input: *NSEQ* is the number of sequences to be transformed

`INTEGER N` [Input]

On input: *N* is the number of elements in each sequence to be transformed

`COMPLEX X(1+(N-1)*INCX1+(NSEQ-1)*INCX2)` [Input/Output]

On input: *X* contains the *NSEQ* complex sequences of length *N* to be transformed; the ith element of sequence j is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).
On output: if *INPL* is .TRUE. then *X* contains the transformed sequences in the same locations as on input; otherwise *X* remains unchanged.

`INTEGER INCX1` [Input]

On input: *INCX1* is the increment used to store successive elements of a given sequence in *X* (*INCX1*=1 for contiguous data).
Constraint: *INCX1* > 0.

`INTEGER INCX2` [Input]

On input: *INCX2* is the increment used to store corresponding elements of successive sequences in *X* (*INCX2*=*N* for contiguous data).
Constraint: *INCX2* > 0.

`COMPLEX Y(1+(N-1)*INCY1+(NSEQ-1)*INCY2)` [Output]

On output: if *INPL* is .FALSE. then *Y* contains the transformed sequences with the ith element of sequence j stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); otherwise *Y* is not referenced.

`INTEGER INCY1` [Input]

On input: *INCY1* is the increment used to store successive elements of a given sequence in *Y*. If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

`INTEGER INCY2` [Input]

On input: *INCY2* is the increment used to store corresponding elements of successive sequences in *Y* (*INCY2*=*N* for contiguous data). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraint: *INCY2* > 0.

`COMPLEX COMM(5*N+100)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward FFTs are performed unscaled and in-place on two
C     contiguous vectors stored in the first two columns of X.
C     Manipulations are stored in 2nd and 3rd columns of X which are
C     then transformed back.
C
        COMPLEX X(N,3)
        SCALE = 1.0
        INPL = .TRUE.
        CALL CFFT1MX(0,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        CALL CFFT1MX(-1,SCALE,INPL,2,N,X,1,N,DUM,1,N,COMM,INFO)
        DO 10 I = 1, N
           X(I,3) = X(I,1)*CONJG(X(I,2))/REAL(N)
           X(I,2) = CMPLX(0.0D0,1.0D0)*X(I,2)/REAL(N)
   10   CONTINUE
        CALL CFFT1MX(1,SCALE,INPL,2,N,X(1,2),1,N,DUM,1,N,COMM,INFO)
```

5.3.3 2D FFT of two-dimensional arrays of data

The routines documented here compute the two-dimensional discrete Fourier transforms (DFT) of a two-dimensional array of complex numbers in either single or double precision arithmetic. The 2D DFT is computed using a highly-efficient FFT algorithm.
There are two sets of interfaces available: simple drivers and expert drivers. The simple drivers perform in-place transforms on data held contiguously in memory using a fixed scaling factor; these are simpler to use and are sufficient for many problems. The expert drivers offer greater flexibility by including a number of additional arguments. These allow you to control: the scaling factor applied; whether the result should be output to a separate array; the increments used in storing successive elements in each dimension (for both input and output); and the facility to not perform a final transposition. This final facility is useful for those cases where a forward and backward transform are to be applied with some data manipulations in between; here two whole transpositions can be saved.

ZFFT2D Routine Documentation

ZFFT2D (*MODE*,*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the direction of transform to be performed by ZFFT2D.

On input:

- *MODE*=−1 : forward 2D transform is performed.
- *MODE*=1 : backward (reverse) 2D transform is performed.

INTEGER M [Input]

On input: M is the number of rows in the 2D array of data to be transformed. If X is declared as a 2D array then M is the first dimension of X.

INTEGER N [Input]

On input: N is the number of columns in the 2D array of data to be transformed. If X is declared as a 2D array then M is the second dimension of X.

COMPLEX*16 X(M*N) [Input/Output]

On input: X contains the M by N complex 2D array to be transformed. Element ij is stored in location $i + (j-1) * M$ of X.

On output: X contains the transformed sequence.

COMPLEX*16 COMM(M*N+3*(M+N)+100) [Input/Output]

COMM is a communication array used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
          CALL ZFFT2D(-1,M,N,X,COMM,INFO)
          DO 20 J = 1, N
             DO 10 I = 1, MIN(J-1,M)
                X(I,J) = 0.5D0*(X(I,J) + X(J,I))
                X(J,I) = DCONJG(X(I,J))
   10        CONTINUE
   20     CONTINUE
          CALL ZFFT2D(1,M,N,X,COMM,INFO)
```

CFFT2D Routine Documentation

CFFT2D (*MODE*,*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the direction of transform to be performed by CFFT2D.

On input:

- *MODE*=−1 : a forward 2D transform is performed.
- *MODE*=1 : a backward (reverse) 2D transform is performed.

INTEGER M [Input]

On input: M is the number of rows in the 2D array of data to be transformed. If X is declared as a 2D array then M is the first dimension of X.

INTEGER N [Input]

On input: N is the number of columns in the 2D array of data to be transformed. If X is declared as a 2D array then M is the second dimension of X.

COMPLEX X(M*N) [Input/Output]

On input: X contains the M by N complex 2D array to be transformed. Element ij is stored in location $i + (j-1) * M$ of X.

On output: X contains the transformed sequence.

COMPLEX COMM(M*N+5*(M+N)) [Input/Output]

COMM is a communication array used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT2D(-1,M,N,X,COMM,INFO)
      DO 20 J = 1, N
         DO 10 I = 1, MIN(J-1,M)
            X(I,J) = 0.5D0*(X(I,J) + X(J,I))
            X(J,I) = CONJG(X(I,J))
10       CONTINUE
20    CONTINUE
      CALL CFFT2D(1,M,N,X,COMM,INFO)
```

ZFFT2DX Routine Documentation

ZFFT2DX (*MODE*,*SCALE*,*LTRANS*,*INPL*,*M*,*N*,*X*,*INCX1*,*INCX2*, *Y*,*INCY1*,*INCY2*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZFFT2DX. On input:

- *MODE*=0 : only initializations (specific to the value of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward 2D transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT2DX.
- *MODE*=1 : a backward (reverse) 2D transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT2DX.
- *MODE*=−2 : (default) initializations and a forward 2D transform are performed.
- *MODE*=2 : (default) initializations and a backward 2D transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the values of N and M) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the values of N and M.

DOUBLE PRECISION SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL LTRANS [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data consistent with the values for arguments *INPL*, *INCX1*, *INCX2*, *INCY1* and *INCY2*. If *LTRANS* is .FALSE. then the final transposition is not performed explicitly; the storage format on output is determined by whether the output data is stored contiguously or not – please see the output specifications for X and Y for details.

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER M [Input]

On input: M is the first dimension of the 2D transform.

INTEGER N [Input]

On input: N is the second dimension of the 2D transform.

`COMPLEX*16 X(1+(M-1)*INCX1+(N-1)*INCX2)` [Input/Output]

On input: X contains the M by N complex 2D data array to be transformed; the (ij)th element is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).
On output: if *INPL* is .TRUE. then X contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; in locations X((i-1)*N+j) when *LTRANS*=.FALSE., *INCX1*=1 and *INCX2*=M; and otherwise in the same locations as on input. If *INPL* is .FALSE. X remains unchanged.

`INTEGER INCX1` [Input]

On input: *INCX1* is the increment used to store, in X, successive elements in the first dimension (*INCX1*=1 for contiguous data).
Constraint: *INCX1* > 0.

`INTEGER INCX2` [Input]

On input: *INCX2* is the increment used to store, in X, successive elements in the second dimension (*INCX2*=M for contiguous data).
Constraint: *INCX2* > 0;
INCX2 > (M-1)*INCX1 if N > 1.

`COMPLEX*16 Y(1+(M-1)*INCY1+(N-1)*INCY2)` [Output]

On output: if *INPL* is .FALSE. then Y contains the transformed data. If *LTRANS*=.TRUE. then the (ij)th data element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); if *LTRANS*=.FALSE., *INCY1*=1 and *INCY2*=N then the (ij)th data element is stored in Y((i-1)*N+j); and otherwise the (ij)th element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2). If *INPL* is .TRUE. then Y is not referenced.

`INTEGER INCY1` [Input]

On input: *INCY1* is the increment used to store successive elements in the first dimension in Y (*INCY1*=1 for contiguous data). If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

`INTEGER INCY2` [Input]

On input: *INCY2* is the increment used to store successive elements in the second dimension in Y (for contiguous data, *INCY2*=M when *LTRANS* is .TRUE. or *INCY2*=N when *LTRANS* is .FALSE.). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraints: *INCY2* > 0;
INCY2 > (M-1)*INCY1 if N > 1 and *LTRANS* is .TRUE.;
INCY2 = N if M > 1 and *LTRANS* is .FALSE..

`COMPLEX*16 COMM(M*N+3*M+3*N+200)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same dimensions M and N. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 2D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        COMPLEX *16 X(M,N), Y(N,M)
        SCALE = 1.0D0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL ZFFT2DX(0,SCALE,LTRANS,INPL,M,N,X,1,M,Y,1,N,COMM,INFO)
        CALL ZFFT2DX(-1,SCALE,LTRANS,INPL,M,N,X,1,M,Y,1,N,COMM,INFO)
        DO 20 I = M
           DO 10 J = 1, N
              Y(J,I) = 0.5D0*Y(J,I)*EXP(0.001D0*(I+J-2))
   10      CONTINUE
   20   CONTINUE
        SCALE = 1.0D0/DBLE(M*N)
        CALL ZFFT2DX(1,SCALE,LTRANS,INPL,N,M,Y,1,N,X,1,M,COMM,INFO)
```

CFFT2DX Routine Documentation

CFFT2DX (*MODE*,*SCALE*,*LTRANS*,*INPL*,*M*,*N*,*X*,*INCX1*,*INCX2*, *Y*,*INCY1*,*INCY2*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CFFT2DX. On input:

- *MODE*=0 : only initializations (specific to the value of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward 2D transform is performed. Initializations are assumed to have been performed by a prior call to CFFT2DX.
- *MODE*=1 : a backward (reverse) 2D transform is performed. Initializations are assumed to have been performed by a prior call to CFFT2DX.
- *MODE*=−2 : (default) initializations and a forward 2D transform are performed.
- *MODE*=2 : (default) initializations and a backward 2D transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the values of N and M) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the values of N and M.

REAL SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL LTRANS [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data consistent with the values for arguments *INPL*, *INCX1*, *INCX2*, *INCY1* and *INCY2*. If *LTRANS* is .FALSE. then the final transposition is not performed explicitly; the storage format on output is determined by whether the output data is stored contiguously or not – please see the output specifications for X and Y for details.

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER M [Input]

On input: M is the first dimension of the 2D transform.

INTEGER N [Input]

On input: N is the second dimension of the 2D transform.

`COMPLEX X(1+(M-1)*INCX1+(N-1)*INCX2)` [Input/Output]

On input: X contains the M by N complex 2D data array to be transformed; the (ij)th element is stored in X(1+(i-1)*INCX1+(j-1)*INCX2).
On output: if *INPL* is .TRUE. then X contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; in locations X((i-1)*N+j) when *LTRANS*=.FALSE., *INCX1*=1 and *INCX2*=M; and otherwise in the same locations as on input. If *INPL* is .FALSE. X remains unchanged.

`INTEGER INCX1` [Input]

On input: *INCX1* is the increment used to store, in X, successive elements in the first dimension (*INCX1*=1 for contiguous data).
Constraint: *INCX1* > 0.

`INTEGER INCX2` [Input]

On input: *INCX2* is the increment used to store, in X, successive elements in the second dimension (*INCX2*=M for contiguous data).
Constraint: *INCX2* > 0;
INCX2 > (M-1)*INCX1 if N > 1.

`COMPLEX Y(1+(M-1)*INCY1+(N-1)*INCY2)` [Output]

On output: if *INPL* is .FALSE. then Y contains the transformed data. If *LTRANS*=.TRUE. then the (ij)th data element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2); if *LTRANS*=.FALSE., *INCY1*=1 and *INCY2*=N then the (ij)th data element is stored in Y((i-1)*N+j); and otherwise the (ij)th element is stored in Y(1+(i-1)*INCY1+(j-1)*INCY2). If *INPL* is .TRUE. then Y is not referenced.

`INTEGER INCY1` [Input]

On input: *INCY1* is the increment used to store successive elements in the first dimension in Y (*INCY1*=1 for contiguous data). If *INPL* is .TRUE. then *INCY1* is not referenced.
Constraint: *INCY1* > 0.

`INTEGER INCY2` [Input]

On input: *INCY2* is the increment used to store successive elements in the second dimension in Y (for contiguous data, *INCY2*=M when *LTRANS* is .TRUE. or *INCY2*=N when *LTRANS* is .FALSE.). If *INPL* is .TRUE. then *INCY2* is not referenced.
Constraints: *INCY2* > 0;
INCY2 > (M-1)*INCY1 if N > 1 and *LTRANS* is .TRUE.;
INCY2 = N if M > 1 and *LTRANS* is .FALSE..

`COMPLEX COMM(M*N+5*M+5*N+200)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same dimensions M and N. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 2D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        COMPLEX X(M,N), Y(N,M)
        SCALE = 1.0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL CFFT2DX(0,SCALE,LTRANS,INPL,M,N,X,1,M,Y,1,N,COMM,INFO)
        CALL CFFT2DX(-1,SCALE,LTRANS,INPL,M,N,X,1,M,Y,1,N,COMM,INFO)
        DO 20 I = M
           DO 10 J = 1, N
              Y(J,I) = 0.5*Y(J,I)*EXP(-0.001*REAL(I+J-2))
              IY = IY + 1
   10      CONTINUE
   20   CONTINUE
        SCALE = 1.0/REAL(M*N)
        CALL CFFT2DX(1,SCALE,LTRANS,INPL,N,M,Y,1,N,X,1,M,COMM,INFO)
```

5.3.4 3D FFT of three-dimensional arrays of data

The routines documented here compute the three-dimensional discrete Fourier transforms (DFT) of a three-dimensional array of complex numbers in either single or double precision arithmetic. The 3D DFT is computed using a highly-efficient FFT algorithm.

Please note that at Release 2.7 of ACML it was necessary to modify slightly the interfaces of two of the expert FFT drivers introduced at Release 2.2 of ACML. The two routines are `CFFT3DX` and `ZFFT3DX`. The changes were required to permit the optimization of these routines by adding an initialization stage which can then use the plan generator (MODE=100) to select the optimal plan. User codes that called `CFFT3DX` or `ZFFT3DX` using a release of ACML prior to 2.7 will need to be modified in one of two ways. Calls to `CFFT3DX`/`ZFFT3DX` with MODE = -1 or 1 can be fixed for ACML Release 2.7 and later by either:

- preceding the call with a call setting MODE = 0 (default initialization), or MODE = 100 (initialization using plan generator); or,
- doubling the MODE argument value to MODE = -2 or 2 respectively (thus incorporating default initialization).

Additionally, the minimum length of the communication (work)space arrays in `CFFT3DX` and `ZFFT3DX` has been increased by 100 to allow for plan storage. Please consult the individual routine documents for full details on their use.

ZFFT3D Routine Documentation

ZFFT3D (*MODE*,*L*,*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the direction of transform to be performed by ZFFT3D.

On input:

- *MODE*=−1 : forward 3D transform is performed.
- *MODE*=1 : backward (reverse) 3D transform is performed.

INTEGER L [Input]

On input: the length of the first dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the first dimension of X.

INTEGER M [Input]

On input: the length of the second dimension of the 3D array of data to be transformed. If X is declared as a 3D array then M is the second dimension of X.

INTEGER N [Input]

On input: the length of the third dimension of the 3D array of data to be transformed. If X is declared as a 3D array then N is the third dimension of X.

COMPLEX*16 X(L*M*N) [Input/Output]

On input: X contains the L by M by N complex 3D array to be transformed. Element ijk is stored in location $i + (j-1) * L + (k-1) * L * M$ of X.

On output: X contains the transformed sequence.

COMPLEX*16 COMM(L*M*N+3*(L+M+N)) [Input/Output]

COMM is a communication array used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL ZFFT3D(-1,L,M,N,X,COMM,INFO)
      DO 30 K = 1, N
        DO 20 J = 1, M
          DO 10 I = 1, L
            X(I,J) = X(I,J)*EXP(-0.001D0*DBLE(I+J+K))
10        CONTINUE
20      CONTINUE
30    CONTINUE
      CALL ZFFT3D(1,L,M,N,X,COMM,INFO)
```

CFFT3D Routine Documentation

CFFT3D (*MODE*,*L*,*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the direction of transform to be performed by CFFT3D.

On input:

- *MODE*=−1 : forward 3D transform is performed.
- *MODE*=1 : backward (reverse) 3D transform is performed.

INTEGER L [Input]

On input: the length of the first dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the first dimension of X.

INTEGER M [Input]

On input: the length of the second dimension of the 3D array of data to be transformed. If X is declared as a 3D array then M is the second dimension of X.

INTEGER N [Input]

On input: the length of the third dimension of the 3D array of data to be transformed. If X is declared as a 3D array then N is the third dimension of X.

COMPLEX X(L*M*N) [Input/Output]

On input: X contains the L by M by N complex 3D array to be transformed. Element ijk is stored in location $i + (j-1) * L + (k-1) * L * M$ of X.

On output: X contains the transformed sequence.

COMPLEX COMM(5*(L+M+N)+4) [Input/Output]

COMM is a communication array used as temporary store. Note that the amount of store explicitly required here is less than in some versions prior to this release (version 4.1 and older). Some further workspace will be allocated internally; the amount of allocated memory requested will be MAX(L*N+N, L*M + L + M, N).

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL CFFT3D(-1,L,M,N,X,COMM,INFO)
      DO 30 K = 1, N
        DO 20 J = 1, M
          DO 10 I = 1, L
            X(I,J) = X(I,J)*EXP(-0.001D0*REAL(I+J+K))
10        CONTINUE
20      CONTINUE
30    CONTINUE
      CALL CFFT3D(1,L,M,N,X,COMM,INFO)
```

ZFFT3DX Routine Documentation

ZFFT3DX (*MODE*,*SCALE*,*LTRANS*,*INPL*,*L*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZFFT3DX.
On input:

- *MODE*=0 : only initializations (specific to the values of L, M and N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=-1 or 1.
- *MODE*=-1 : a forward 3D transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT3DX.
- *MODE*=1 : a backward (reverse) 3D transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT3DX.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the values of L, M and M) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the values of L, M and N.

DOUBLE PRECISION SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL LTRANS [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data using the same storage format as the input data. If *LTRANS* is .FALSE. then the final transposition is not performed and transformed data is stored, in X or Y, in transposed form.

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER L [Input]

On input: L is the first dimension of the 3D transform.

INTEGER M [Input]

On input: M is the second dimension of the 3D transform.

INTEGER N [Input]

On input: N is the third dimension of the 3D transform.

COMPLEX*16 X(L*M*N) [Input/Output]

On input: X contains the L by M by N complex 3D data array to be transformed; the (ijk)th element is stored in X(i+(j-1)*L+(k-1)*L*M).
On output: if *INPL* is .TRUE. then X contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; or in locations X(k+(j-1)*N+(i-1)*N*M) when *LTRANS*=.FALSE. If *INPL* is .FALSE. X remains unchanged.

`COMPLEX*16 Y(L*M*N)` [Output]

On output: if *INPL* is .FALSE. then Y contains the three-dimensional transformed data. If *LTRANS*=.TRUE. then the (ijk)th data element is stored in Y(i+(j-1)*L+(k-1)*L*M); otherwise, the (ijk)th data element is stored in Y(k+(j-1)*N+(i-1)*N*M). If *INPL* is .TRUE. then Y is not referenced.

`COMPLEX*16 COMM(L*M*N+3*(L+M+N)+300)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence dimensions. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If $INFO = -$i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 3D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        COMPLEX *16 X(L*M*N), Y(L*M*N)
        SCALE = 1.0D0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL ZFFT3DX(0,SCALE,LTRANS,INPL,L,M,N,X,Y,COMM,INFO)
        CALL ZFFT3DX(-1,SCALE,LTRANS,INPL,L,M,N,X,Y,COMM,INFO)
        IY = 1
        DO 20 I = 1, L
           DO 40 J = 1, M
              DO 10 K = 1, N
                 Y(IY) = Y(IY)*EXP(-0.001D0*DBLE(I+J+K-3))
                 IY = IY + 1
   10         CONTINUE
   20      CONTINUE
           SCALE = 1.0D0/DBLE(L*M*N)
           CALL ZFFT3DX(1,SCALE,LTRANS,INPL,N,M,L,Y,X,COMM,INFO)
```

CFFT3DX Routine Documentation

CFFT3DX (*MODE*,*SCALE*,*LTRANS*,*INPL*,*L*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CFFT3DX.
On input:

- *MODE*=0 : only initializations (specific to the values of L, M and N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward 3D transform is performed. Initializations are assumed to have been performed by a prior call to CFFT3DX.
- *MODE*=1 : a backward (reverse) 3D transform is performed. Initializations are assumed to have been performed by a prior call to CFFT3DX.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the values of L, M and M) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the values of L, M and N.

REAL SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL LTRANS [Input]

On input: if *LTRANS* is .TRUE. then a normal final transposition is performed internally to return transformed data using the same storage format as the input data. If *LTRANS* is .FALSE. then the final transposition is not performed and transformed data is stored, in X or Y, in transposed form.

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER L [Input]

On input: L is the first dimension of the 3D transform.

INTEGER M [Input]

On input: M is the second dimension of the 3D transform.

INTEGER N [Input]

On input: N is the third dimension of the 3D transform.

COMPLEX X(L*M*N) [Input/Output]

On input: X contains the L by M by N complex 3D data array to be transformed; the (ijk)th element is stored in X(i+(j-1)*L+(k-1)*L*M).
On output: if *INPL* is .TRUE. then X contains the transformed data, either in the same locations as on input when *LTRANS*=.TRUE.; or in locations X(k+(j-1)*N+(i-1)*N*M) when *LTRANS*=.FALSE. If *INPL* is .FALSE. X remains unchanged.

`COMPLEX Y(L*M*N)` [Output]

On output: if *INPL* is .FALSE. then *Y* contains the three-dimensional transformed data. If *LTRANS*=.TRUE. then the (ijk)th data element is stored in Y(i+(j-1)*L+(k-1)*L*M); otherwise, the (ijk)th data element is stored in Y(k+(j-1)*N+(k-1)*N*M). If *INPL* is .TRUE. then *Y* is not referenced.

`COMPLEX COMM(*)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence dimensions. The remainder is used as temporary store. The amount of store required depends on the values of the arguments *L*, *M*, *N* and *LTRANS*. If *LTRANS*=.TRUE. then for a genuine 3D transform (all of *L*, *M*, *N* greater than 1) the dimension of *COMM* need only be 5*(L+M+N) + 150; in this case some further workspace will be allocated internally; the amount of allocated memory requested will be MAX(L*N+N, L*M + L + M, N). If *LTRANS*=.FALSE. then for a genuine 3D transform the workspace requirement is considerably more to allow for the storage of an intermediate 3D transposed array; in this case the dimension of *COMM* must be at least L*M*N+5*(L+M+N)+150. It is recommended that the appropriate 1D or 2D FFT routine be called when at least one of *L*, *M* or *N* is 1.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 3D FFT is performed unscaled, without final transpose
C     and out-of-place on data stored in array X and output to Y.
C     Manipulations are stored in vector Y which is then transformed
C     back, with scaling, into the first M rows of X.
C
        SCALE = 1.0
        INPL = .FALSE.
        LTRANS = .FALSE.
        CALL CFFT3DX(0,SCALE,LTRANS,INPL,L,M,N,X,Y,COMM,INFO)
        CALL CFFT3DX(-1,SCALE,LTRANS,INPL,L,M,N,X,Y,COMM,INFO)
        IY = 1
        DO 20 I = 1, L
           DO 40 J = 1, M
              DO 10 K = 1, N
                 Y(IY) = Y(IY)*EXP(-0.001*REAL(I+J+K-3))
                 IY = IY + 1
   10         CONTINUE
   20      CONTINUE
           SCALE = 1.0/REAL(L*M*N)
           CALL CFFT3DX(1,SCALE,LTRANS,INPL,N,M,L,Y,X,COMM,INFO)
```

ZFFT3DY Routine Documentation

ZFFT3DY (*MODE*,*SCALE*,*INPL*,*L*,*M*,*N*,*X*,*INCX1*,*INCX2*,*INCX3*,*Y*, *INCY1*,*INCY2*,*INCY3*,*COMM*,*LCOMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZFFT3DY. On input:

- *MODE*=0 : only initializations (specific to the values of L, M and N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward 3D transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT3DY.
- *MODE*=1 : a backward (reverse) 3D transform is performed. Initializations are assumed to have been performed by a prior call to ZFFT3DY.
- *MODE*=−2 : (default) initializations and a forward 3D transform are performed.
- *MODE*=2 : (default) initializations and a backward 3D transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the values of L, M and M) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the values of L, M and N.

REAL SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER L [Input]

On input: L is the first dimension of the 3D transform.

INTEGER M [Input]

On input: M is the second dimension of the 3D transform.

INTEGER N [Input]

On input: N is the third dimension of the 3D transform.

COMPLEX*16 X(*) [Input/Output]

On input: X contains the L by M by N complex 3D data array to be transformed; the (ijk)th element is stored in X(1+(i-1)*INCX1+(j-1)*INCX2+(k-1)*INCX3).

On output: if *INPL* is .TRUE. then X contains the transformed data in the same locations as on input. If *INPL* is .FALSE. X remains unchanged.

`INTEGER INCX1` [Input]

On input: *INCX1* is the step in index of X between successive data elements in the first dimension of the 3D data. Usually *INCX1*=1 so that succesive elements in the first dimension are stored contiguously.
Constraint: *INCX1* > 0.

`INTEGER INCX2` [Input]

On input: *INCX2* is the step in index of X between successive data elements in the second dimension of the 3D data. For completely contiguous data (no gaps in X) *INCX2* should be set to L.
Constraint: *INCX2* > 0;
INCX2 > (L-1)*INCX1 if max(M,N) > 1.

`INTEGER INCX3` [Input]

On input: *INCX3* is the step in index of X between successive data elements in the third dimension of the 3D data. For completely contiguous data (no gaps in X) *INCX3* should be set to $L*M$.
Constraint: *INCX3* > 0;
INCX3 > (L-1)**INCX1*+(M-1)**INCX2* if N > 1.

`COMPLEX*16 Y(*)` [Output]

On output: if *INPL* is .FALSE. then Y contains the three-dimensional transformed data. If *LTRANS*=.TRUE. then the the (ijk)th element is stored in Y(1+(i-1)**INCY1*+(j-1)**INCY2*+(k-1)**INCY3*).
If *INPL* is .TRUE. then Y is not referenced.

`INTEGER INCY1` [Input]

On input: if *INPL* is .FALSE. then *INCY1* is the step in index of Y between successive data elements in the first dimension of the 3D transformed data. Usually *INCY1*=1 so that succesive elements in the first dimension are stored contiguously.
If *INPL* is .TRUE. then *INCY1* is not referenced. Constraint: If *INPL* is .FALSE. then *INCY1* > 0.

`INTEGER INCY2` [Input]

On input: if *INPL* is .FALSE. then *INCY2* is the step in index of Y between successive data elements in the second dimension of the 3D transformed data. For completely contiguous data (no gaps in Y) *INCY2* should be set to L.
Constraint: *INCY2* > 0 if *INPL* is .FALSE.;
INCY2 > (L-1)**INCY1*, if *INPL* is .FALSE. and max(M,N) > 1.

`INTEGER INCY3` [Input]

On input: if *INPL* is .FALSE. then *INCY3* is the step in index of Y between successive data elements in the third dimension of the 3D transformed data. For completely contiguous data (no gaps in Y) *INCY3* should be set to $L*M$.
Constraint: *INCY3* > 0 if *INPL* is .FALSE.;
INCY3 > (L-1)**INCY1*+(M-1)**INCY2*, if *INPL* is .FALSE. and N > 1.

`COMPLEX*16 COMM(`*LCOMM*`)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence dimensions. The remainder is used as temporary store; if this is not sufficient for the requirements of the routine then temporary storage space will be dynamically allocated internally.

`INTEGER LCOMM` [Input]

On input: *LCOMM* is the length of the communication array *COMM*. The amount of internal dynamic allocation of temporary storage can be reduced significantly by declaring *COMM* to be of length at least L*M*N + 4*(L+M+N) + 300.

Constraint: *LCOMM* > 3*(L+M+N) + 150.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 3D FFT is performed unscaled and in-place, on the leading
C     10x10x10 submatrix of a larger 100x100x100 array of data.
C     The result is transformed back with scaling.
C
        SCALE = 1.0D0
        INPL = .TRUE.
        L = 10
        M = 10
        N = 10
        LCOMM = 2000000
        CALL ZFFT3DY(0,SCALE,INPL,L,M,N,X,1,100,10000,Y,1,1,1,
     *               COMM,LCOMM,INFO)
        CALL ZFFT3DY(-1,SCALE,INPL,L,M,N,X,1,100,10000,Y,1,1,1,
     *               COMM,LCOMM,INFO)
        IY = 1
        DO 20 I = 1, L
           DO 40 J = 1, M
              DO 10 K = 1, N
                 X(I,J,K) = X(I,J,K)*EXP(-1.0D-3*DBLE(I+J+K-3))
   10      CONTINUE
   20   CONTINUE
        SCALE = 1.0/DBLE(L*M*N)
        CALL ZFFT3DY(1,SCALE,INPL,L,M,N,X,1,100,10000,Y,1,1,1,
     *               COMM,LCOMM,INFO)
```

CFFT3DY Routine Documentation

CFFT3DY (*MODE*,*SCALE*,*INPL*,*L*,*M*,*N*,*X*,*INCX1*,*INCX2*,*INCX3*,*Y*, *INCY1*,*INCY2*,*INCY3*,*COMM*,*LCOMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CFFT3DY. On input:

- *MODE*=0 : only initializations (specific to the values of L, M and N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=−1 or 1.
- *MODE*=−1 : a forward 3D transform is performed. Initializations are assumed to have been performed by a prior call to CFFT3DY.
- *MODE*=1 : a backward (reverse) 3D transform is performed. Initializations are assumed to have been performed by a prior call to CFFT3DY.
- *MODE*=−2 : (default) initializations and a forward 3D transform are performed.
- *MODE*=2 : (default) initializations and a backward 3D transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the values of L, M and M) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the values of L, M and N.

REAL SCALE [Input]

On input: *SCALE* is the scaling factor to apply to the output sequences

LOGICAL INPL [Input]

On input: if *INPL* is .TRUE. then X is overwritten by the output sequences; otherwise the output sequences are returned in Y.

INTEGER L [Input]

On input: L is the first dimension of the 3D transform.

INTEGER M [Input]

On input: M is the second dimension of the 3D transform.

INTEGER N [Input]

On input: N is the third dimension of the 3D transform.

COMPLEX X(*) [Input/Output]

On input: X contains the L by M by N complex 3D data array to be transformed; the (ijk)th element is stored in X(1+(i-1)*INCX1+(j-1)*INCX2+(k-1)*INCX3).

On output: if *INPL* is .TRUE. then X contains the transformed data in the same locations as on input. If *INPL* is .FALSE. X remains unchanged.

`INTEGER INCX1` [Input]

On input: *INCX1* is the step in index of X between successive data elements in the first dimension of the 3D data. Usually *INCX1*=1 so that succesive elements in the first dimension are stored contiguously.
Constraint: *INCX1* > 0.

`INTEGER INCX2` [Input]

On input: *INCX2* is the step in index of X between successive data elements in the second dimension of the 3D data. For completely contiguous data (no gaps in X) *INCX2* should be set to L.
Constraint: *INCX2* > 0;
INCX2 > (L-1)*INCX1 if max(M,N) > 1.

`INTEGER INCX3` [Input]

On input: *INCX3* is the step in index of X between successive data elements in the third dimension of the 3D data. For completely contiguous data (no gaps in X) *INCX3* should be set to $L*M$.
Constraint: *INCX3* > 0;
INCX3 > (L-1)**INCX1*+(M-1)**INCX2* if N > 1.

`COMPLEX Y(*)` [Output]

On output: if *INPL* is .FALSE. then Y contains the three-dimensional transformed data. If *LTRANS*=.TRUE. then the the (ijk)th element is stored in Y(1+(i-1)**INCY1*+(j-1)**INCY2*+(k-1)**INCY3*).
If *INPL* is .TRUE. then Y is not referenced.

`INTEGER INCY1` [Input]

On input: if *INPL* is .FALSE. then *INCY1* is the step in index of Y between successive data elements in the first dimension of the 3D transformed data. Usually *INCY1*=1 so that succesive elements in the first dimension are stored contiguously.
If *INPL* is .TRUE. then *INCY1* is not referenced. Constraint: If *INPL* is .FALSE. then *INCY1* > 0.

`INTEGER INCY2` [Input]

On input: if *INPL* is .FALSE. then *INCY2* is the step in index of Y between successive data elements in the second dimension of the 3D transformed data. For completely contiguous data (no gaps in Y) *INCY2* should be set to L.
Constraint: *INCY2* > 0 if *INPL* is .FALSE.;
INCY2 > (L-1)**INCY1*, if *INPL* is .FALSE. and max(M,N) > 1.

`INTEGER INCY3` [Input]

On input: if *INPL* is .FALSE. then *INCY3* is the step in index of Y between successive data elements in the third dimension of the 3D transformed data. For completely contiguous data (no gaps in Y) *INCY3* should be set to $L*M$.
Constraint: *INCY3* > 0 if *INPL* is .FALSE.;
INCY3 > (L-1)**INCY1*+(M-1)**INCY2*, if *INPL* is .FALSE. and N > 1.

`COMPLEX COMM(`*LCOMM*`)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence dimensions. The remainder is used as temporary store; if this is not sufficient for the requirements of the routine then temporary storage space will be dynamically allocated internally.

`INTEGER LCOMM` [Input]

On input: *LCOMM* is the length of the communication array *COMM*. The amount of internal dynamic allocation of temporary storage is dependent on the values of the increment arguments for arrays X and Y. The amount is minimized when the increments for the output array are 1, L and L^*M respectively, since this represents a contiguous array in which calculations can be performedd in-place.

Constraint: If min(L,M,N) > 1, *LCOMM* >= 5*(L+M+N) + 150 (otherwise see see [CFFT2DX], page 50).

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Forward 3D FFT is performed unscaled and in-place, on the leading
C     10x10x10 submatrix of a larger 100x100x100 array of data.
C     The result is transformed back with scaling.
C
        SCALE = 1.0
        INPL = .TRUE.
        L = 10
        M = 10
        N = 10
        LCOMM = 2000000
        CALL CFFT3DY(0,SCALE,INPL,L,M,N,X,1,100,10000,Y,1,1,1,
     *               COMM,LCOMM,INFO)
        CALL CFFT3DY(-1,SCALE,INPL,L,M,N,X,1,100,10000,Y,1,1,1,
     *               COMM,LCOMM,INFO)
        IY = 1
        DO 20 I = 1, L
           DO 40 J = 1, M
              DO 10 K = 1, N
                 X(I,J,K) = X(I,J,K)*EXP(-0.001*REAL(I+J+K-3))
   10      CONTINUE
   20   CONTINUE
        SCALE = 1.0/REAL(L*M*N)
        CALL CFFT3DY(1,SCALE,INPL,L,M,N,X,1,100,10000,Y,1,1,1,
     *               COMM,LCOMM,INFO)
```

5.4 FFTs on Real and Hermitian Data Sequences

The routines documented here compute discrete Fourier transforms (DFTs) of sequences of real numbers or of Hermitian sequences in either single or double precision arithmetic. The DFTs are computed using a highly-efficient FFT algorithm. Hermitian sequences are represented in one of the two formats that is described in Section 5.2 [Introduction to FFTs], page 24. The DFT of a real sequence results in a Hermitian sequence; the DFT of a Hermitian sequence is a real sequence.

Please note that prior to Release 2.0 of ACML the routine ZDFFT/CSFFT and ZDFFTM/CSFFTM returned results that were scaled by a factor 0.5 compared with the currently returned results.

5.4.1 1D Real-To-Complex FFT (Hermitian-Packed Storage)

DZFFT Routine Documentation

DZFFT (*MODE*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by DZFFT.
On input:

- *MODE*=0 : only default initializations (specific to N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to DZFFT.
- *MODE*=2 : (default) initializations and a real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER N [Input]

On input: N is the length of the real sequence X

DOUBLE PRECISION X(N) [Input/Output]

On input: X contains the real sequence of length N to be transformed.
On output: X contains the transformed Hermitian sequence.

DOUBLE PRECISION COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFT(0,N,X,COMM,INFO)
      CALL DZFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL ZDFFT(2,N,X,COMM,INFO)
```

SCFFT Routine Documentation

SCFFT (*MODE*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by SCFFT.
On input:

- *MODE*=0 : only default initializations (specific to *N*) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to SCFFT.
- *MODE*=2 : (default) initializations and a real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER N [Input]

On input: *N* is the length of the real sequence *X*

REAL X(N) [Input/Output]

On input: *X* contains the real sequence of length *N* to be transformed.
On output: *X* contains the transformed Hermitian sequence.

REAL COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFT(0,N,X,COMM,INFO)
      CALL SCFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL CSFFT(2,N,X,COMM,INFO)
```

5.4.2 Multiple 1D Real-To-Complex FFT (Hermitian-Packed Storage)

DZFFTM Routine Documentation

DZFFTM (*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: M is the number of sequences to be transformed.

INTEGER N [Input]
On input: N is the length of the real sequences in X

DOUBLE PRECISION X(N*M) [Input/Output]
On input: X contains the M real sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j - 1) * N$ of X.
On output: X contains the transformed Hermitian sequences.

DOUBLE PRECISION COMM(3*N+100) [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFTM(1,N,X,COMM,INFO)
      CALL DZFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL ZDFFTM(2,N,X(1,3),COMM,INFO)
```

SCFFTM Routine Documentation

SCFFTM (*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: M is the number of sequences to be transformed.

INTEGER N [Input]
On input: N is the length of the real sequences in X

REAL X(N*M) [Input/Output]
On input: X contains the M real sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j - 1) * N$ of X.
On output: X contains the transformed Hermitian sequences.

REAL COMM(3*N+100) [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFTM(1,N,X,COMM,INFO)
      CALL SCFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL CSFFTM(1,N,X(1,3),COMM,INFO)
```

5.4.3 1D Complex-To-Real FFT (Hermitian-Packed Storage)

ZDFFT Routine Documentation

ZDFFT (*MODE*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZDFFT.
On input:

- *MODE*=0 : only initializations (specific to the values of *N*) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to ZDFFT.
- *MODE*=2 : (default) initializations and a real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of *N*) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of *N*.

INTEGER N [Input]

On input: *N* is length of the sequence in *X*

DOUBLE PRECISION X(N) [Input/Output]

On input: *X* contains the Hermitian sequence of length *N* to be transformed.
On output: *X* contains the transformed real sequence.

DOUBLE PRECISION COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFT(0,N,X,COMM,INFO)
      CALL DZFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL ZDFFT(2,N,X,COMM,INFO)
```

CSFFT Routine Documentation

CSFFT (*MODE*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CSFFT.
On input:

- *MODE*=0 : only initializations (specific to the values of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to CSFFT.
- *MODE*=2 : (default) initializations and a real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of N) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of N.

INTEGER N [Input]

On input: N is the length of the sequence in X

REAL X(N) [Input/Output]

On input: X contains the Hermitian sequence of length N to be transformed.
On output: X contains the transformed real sequence.

REAL COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFT(0,N,X,COMM,INFO)
      CALL SCFFT(1,N,X,COMM,INFO)
      DO 10 I = N/2+2, N
         X(I) = -X(I)
10    CONTINUE
      CALL CSFFT(2,N,X,COMM,INFO)
```

5.4.4 Multiple 1D Complex-To-Real FFT (Hermitian-Packed Storage)

ZDFFTM Routine Documentation

ZDFFTM (*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: M is the number of sequences to be transformed.

INTEGER N [Input]
On input: N is the length of the sequences in X

DOUBLE PRECISION X(N*M) [Input/Output]
On input: X contains the M Hermitian sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j-1) * N$ of X.
On output: X contains the transformed real sequences.

DOUBLE PRECISION COMM(3*N+100) [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFTM(1,N,X,COMM,INFO)
      CALL DZFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL ZDFFTM(1,N,X(1,3),COMM,INFO)
```

CSFFTM Routine Documentation

CSFFTM (*M*,*N*,*X*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: M is the number of sequences to be transformed.

INTEGER N [Input]
On input: N is the length of the sequences in X

REAL X(N*M) [Input/Output]
On input: X contains the M Hermitian sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j - 1) * N$ of X.

On output: X contains the transformed real sequences.

REAL COMM(3*N+100) [Input/Output]
COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* $= -$i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFTM(1,N,X,COMM,INFO)
      CALL SCFFTM(2,N,X,COMM,INFO)
      DO 10 I = 1, N
         X(I,3) = X(I,1)*X(N-I+1,2)
10    CONTINUE
      CALL CSFFTM(1,N,X(1,3),COMM,INFO)
```

5.4.5 1D Real-To-Complex FFT (Complex-Hermitian Storage)

DZFFT1D Routine Documentation

DZFFT1D (*MODE*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by DZFFT1D. On input:

- *MODE*=0 : only default initializations (specific to N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to DZFFT1D.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER N [Input]

On input: N is length of the sequence in X

DOUBLE PRECISION X(N) [Input]

On input: X contains the real sequence of length N to be transformed.

DOUBLE COMPLEX Y(N/2+1) [Output]

On output: Y contains the tansformed complex sequence with roughly half redundant information due to complex conjugate removed.

DOUBLE PRECISION COMM(4*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL DZFFT1D(0,N,X,Y,COMM,INFO)
      CALL DZFFT1D(1,N,X,Y,COMM,INFO)
      DO 10 I = 1, N/2+1
         Y(I) = -Y(I)*EXP(-DBLE(I-1)/DBLE(N))
      10 CONTINUE
      CALL ZDFFT1D(2,N,Y,X,COMM,INFO)
```

SCFFT1D Routine Documentation

SCFFT1D (*MODE*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by SCFFT1D. On input:

- *MODE*=0 : only default initializations (specific to N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to SCFFT1D.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER N [Input]

On input: N is length of the sequence in X

REAL X(N) [Input]

On input: X contains the real sequence of length N to be transformed.

COMPLEX Y(N/2+1) [Output]

On output: Y contains the tansformed complex sequence with roughly half redundant information due to complex conjugate removed.

REAL COMM(4*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
      CALL SCFFT1D(0,N,X,Y,COMM,INFO)
      CALL SCFFT1D(1,N,X,Y,COMM,INFO)
      DO 10 I = 1, N/2+1
         Y(I) = -Y(I)*EXP(-REAL(I-1)/REAL(N))
      10 CONTINUE
      CALL CSFFT1D(2,N,Y,X,COMM,INFO)
```

5.4.6 1D Complex-To-Real FFT (Complex-Hermitian Storage)

ZDFFT1D Routine Documentation

ZDFFT1D (*MODE*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZDFFT1D. On input:

- *MODE*=0 : only initializations (specific to the values of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to ZDFFT1D.
- *MODE*=2 : (default) initializations and a complex-to-real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of N) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of N.

INTEGER N [Input]

On input: N is length of the sequence in Y

DOUBLE COMPLEX X(N/2+1) [Input]

On input: X contains the complex sequence to be transformed.

DOUBLE PRECISION Y(N) [Output]

On output: Y contains the transformed real sequence of length N.

DOUBLE PRECISION COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = $-$i on exit, the i-th argument had an illegal value.

Example:

```
CALL DZFFT1D(0,N,X,Y,COMM,INFO)
CALL DZFFT1D(1,N,X,Y,COMM,INFO)
DO 10 I = 1, N/2+1
   Y(I) = -Y(I)*EXP(-DBLE(I-1)/DBLE(N))
10 CONTINUE
CALL ZDFFT1D(2,N,Y,X,COMM,INFO)
```

CSFFT1D Routine Documentation

CSFFT1D (*MODE*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CSFFT1D. On input:

- *MODE*=0 : only initializations (specific to the values of N) are performed using a default plan; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to CSFFT1D.
- *MODE*=2 : (default) initializations and a complex-to-real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations (specific to the value of N) are performed, but these are based on a plan that is first generated by timing a subset of all possible plans and choosing the quickest (i.e. the FFT computation was timed as fastest based on the chosen plan). The plan generation phase may take a significant amount of time depending on the value of N.

INTEGER N [Input]

On input: N is length of the sequence in Y

COMPLEX X(N/2+1) [Input]

On input: X contains the complex sequence to be transformed.

REAL Y(N) [Output]

On output: Y contains the transformed real sequence of length N.

REAL COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL SCFFT1D(0,N,X,Y,COMM,INFO)
CALL SCFFT1D(1,N,X,Y,COMM,INFO)
DO 10 I = 1, N/2+1
   Y(I) = -Y(I)*EXP(-REAL(I-1)/REAL(N))
10 CONTINUE
CALL CSFFT1D(2,N,Y,X,COMM,INFO)
```

5.4.7 Multiple 1D Real-To-Complex FFT (Complex-Hermitian Storage)

DZFFT1M Routine Documentation

DZFFT1M (*MODE*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by DZFFT1M. On input:

- *MODE*=0 : only default initializations (specific to N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to DZFFT1M.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER M [Input]

On input: M is the number of sequences to be transformed.

INTEGER N [Input]

On input: N is length of the sequence in X

DOUBLE PRECISION X(N*M) [Input]

On input: X contains the M real sequences of length N to be transformed. Element i of sequence j is stored in location i + *(j - 1)* N of X.

DOUBLE COMPLEX Y((N/2+1)*M) [Output]

On output: Y contains the transformed Hermitian sequences in complex-Hermitian storage.

DOUBLE PRECISION COMM(4*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL DZFFT1M(0,M,N,X,Y,COMM,INFO)
CALL DZFFT1M(1,M,N,X,Y,COMM,INFO)
DO J = 1, M
   DO I = 1, N/2+1
      Y(I,J) = -Y(I,J)*EXP(-DBLE(I-1)/DBLE(N))
   END DO
EMD DO
CALL ZDFFT1M(2,M,N,Y,X,COMM,INFO)
```

SCFFT1M Routine Documentation

SCFFT1M (*MODE*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by SCFFT1M. On input:

- *MODE*=0 : only default initializations (specific to N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to SCFFT1M.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER M [Input]

On input: M is the number of sequences to be transformed.

INTEGER N [Input]

On input: N is length of the sequence in X

REAL X(N*M) [Input]

On input: X contains the M real sequences of length N to be transformed. Element i of sequence j is stored in location $i + (j - 1) * N$ of X.

COMPLEX Y((N/2+1)*M) [Output]

On output: Y contains the transformed Hermitian sequences in complex-Hermitian storage.

REAL COMM(4*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length N. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL SCFFT1M(0,M,N,X,Y,COMM,INFO)
CALL SCFFT1M(1,M,N,X,Y,COMM,INFO)
DO J = 1, M
   DO I = 1, N/2+1
      Y(I,J) = -Y(I,J)*EXP(-REAL(I-1)/REAL(N))
   END DO
EMD DO
CALL CSFFT1M(2,M,N,Y,X,COMM,INFO)
```

5.4.8 Multiple 1D Complex-To-Real FFT (Complex-Hermitian Storage)

ZDFFT1M Routine Documentation

ZDFFT1M (*MODE*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZDFFT1M. On input:

- *MODE*=0 : only default initializations (specific to *N*) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to ZDFFT1M.
- *MODE*=2 : (default) initializations and a complex-to-real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER M [Input]

On input: *M* is the number of sequences to be transformed.

INTEGER N [Input]

On input: *N* is length of the sequence in *Y*

DOUBLE COMPLEX X((N/2+1)*M) [Input]

On input: *X* contains the Hermitian sequences to be transformed in complex-Hermitian storage.

DOUBLE PRECISION Y(N*M) [Output]

On output: *Y* contains the *M* real sequences of length *N* transformed. Element *i* of sequence *j* is stored in location *i + (j - 1) * N* of *Y*.

DOUBLE PRECISION COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL DZFFT1M(0,M,N,X,Y,COMM,INFO)
CALL DZFFT1M(1,M,N,X,Y,COMM,INFO)
DO J = 1, M
   DO I = 1, N/2+1
      Y(I,J) = -Y(I,J)*EXP(-DBLE(I-1)/DBLE(N))
   END DO
EMD DO
CALL ZDFFT1M(2,M,N,Y,X,COMM,INFO)
```

CSFFT1M Routine Documentation

CSFFT1M (*MODE*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CSFFT1M. On input:

- *MODE*=0 : only default initializations (specific to *N*) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to CSFFT1M.
- *MODE*=2 : (default) initializations and a complex-to-real transform a re performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER M [Input]

On input: *M* is the number of sequences to be transformed.

INTEGER N [Input]

On input: *N* is length of the sequence in *Y*

COMPLEX X((N/2+1)*M) [Input]

On input: *X* contains the Hermitian sequences to be transformed in complex-Hermitian storage.

REAL Y(N*M) [Output]

On output: *Y* contains the *M* real sequences of length *N* transformed. Element *i* of sequence *j* is stored in location *i* + *(j - 1)* * *N* of *Y*.

REAL COMM(3*N+100) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL SCFFT1M(0,M,N,X,Y,COMM,INFO)
CALL SCFFT1M(1,M,N,X,Y,COMM,INFO)
DO J = 1, M
   DO I = 1, N/2+1
      Y(I,J) = -Y(I,J)*EXP(-REAL(I-1)/REAL(N))
   END DO
EMD DO
CALL CSFFT1M(2,M,N,Y,X,COMM,INFO)
```

5.4.9 2D Real-To-Complex FFT

`DZFFT2D` Routine Documentation

`DZFFT2D` (*MODE,M,N,X,Y,COMM,INFO*) [SUBROUTINE]

`INTEGER MODE` [Input]

The value of *MODE* on input determines the operation performed by `DZFFT2D`. On input:

- *MODE*=0 : only default initializations (specific to M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to `DZFFT2D`.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

`INTEGER M` [Input]

On input: M is the number of rows in the 2D array of data to be transformed. If X is declared as a 2D array then M is the first dimension of X.

`INTEGER N` [Input]

On input: N is the number of columns in the 2D array of data to be transformed. If X is declared as a 2D array then N is the second dimension of X.

`DOUBLE PRECISION X(M*N)` [Input]

On input: X contains the M by N real 2D array to be transformed. Element ij is stored in location *i + (j - 1) * M* of X.

`DOUBLE COMPLEX Y((M/2+1)*N)` [Output]

On output: Y contains the transformed Hermitian sequences in complex-Hermitian storage.

`DOUBLE PRECISION COMM(4*M+6*N+300)` [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *M*N*. The remainder is used as temporary store.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL DZFFT2D(0,M,N,X,Y,COMM,INFO)
CALL DZFFT2D(1,M,N,X,Y,COMM,INFO)
DO J = 1, N
   DO I = 1, M/2+1
      Y(I,J) = -Y(I,J)/SQRT(DBLE(M*N))
   END DO
EMD DO
CALL ZDFFT2D(2,M,N,Y,X,COMM,INFO)
```

SCFFT2D Routine Documentation

SCFFT2D (*MODE*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by SCFFT2D. On input:

- *MODE*=0 : only default initializations (specific to M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to SCFFT2D.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER M [Input]

On input: M is the number of rows in the 2D array of data to be transformed. If X is declared as a 2D array then M is the first dimension of X.

INTEGER N [Input]

On input: N is the number of columns in the 2D array of data to be transformed. If X is declared as a 2D array then N is the second dimension of X.

REAL X(M*N) [Input]

On input: X contains the M by N real 2D array to be transformed. Element ij is stored in location *i + (j - 1) * M* of X.

COMPLEX Y((M/2+1)*N) [Output]

On output: Y contains the transformed Hermitian sequences in complex-Hermitian storage.

REAL COMM(4*M+10*N+300) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *M*N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL SCFFT2D(0,M,N,X,Y,COMM,INFO)
CALL SCFFT2D(1,M,N,X,Y,COMM,INFO)
DO J = 1, N
   DO I = 1, M/2+1
      Y(I,J) = -Y(I,J)/SQRT(REAL(M*N))
   END DO
EMD DO
CALL CSFFT2D(2,M,N,Y,X,COMM,INFO)
```

5.4.10 2D Complex-To-Real FFT

ZDFFT2D Routine Documentation

ZDFFT2D (*MODE*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZDFFT2D. On input:

- *MODE*=0 : only default initializations (specific to M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to ZDFFT2D.
- *MODE*=2 : (default) initializations and a complex-to-real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER M [Input]

On input: M is the number of rows in the 2D array of the real data obtained from the transform. If Y is declared as a 2D array then M is the first dimension of Y.

INTEGER N [Input]

On input: N is the number of columns in the 2D array of the real data obtained from the transform. If Y is declared as a 2D array then N is the second dimension of Y.

DOUBLE COMPLEX X((M/2+1)*N) [Input]

On input: X contains the Hermitian sequences in complex-Hermitian storage to be transformed.

DOUBLE PRECISION Y(M*N) [Output]

On output: Y contains the M by N real 2D array obtained from the transform. Element *ij* is stored in location *i + (j - 1) * M* of Y.

DOUBLE PRECISION COMM(4*M+6*N+300) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *M*N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL DZFFT2D(0,M,N,X,Y,COMM,INFO)
CALL DZFFT2D(1,M,N,X,Y,COMM,INFO)
DO J = 1, N
   DO I = 1, M/2+1
      Y(I,J) = -Y(I,J)/SQRT(DBLE(M*N))
   END DO
EMD DO
CALL ZDFFT2D(2,M,N,Y,X,COMM,INFO)
```

CSFFT2D Routine Documentation

CSFFT2D (*MODE*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CSFFT2D. On input:

- *MODE*=0 : only default initializations (specific to M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to CSFFT2D.
- *MODE*=2 : (default) initializations and a complex-to-real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER M [Input]

On input: M is the number of rows in the 2D array of the real data obtained from the transform. If Y is declared as a 2D array then M is the first dimension of Y.

INTEGER N [Input]

On input: N is the number of columns in the 2D array of the real data obtained from the transform. If Y is declared as a 2D array then N is the second dimension of Y.

COMPLEX X((M/2+1)*N) [Input]

On input: X contains the Hermitian sequences in complex-Hermitian storage to be transformed.

REAL Y(M*N) [Output]

On output: Y contains the M by N real 2D array obtained from the transform. Element *ij* is stored in location *i + (j - 1) * M* of Y.

REAL COMM(4*M+10*N+300) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *M*N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL SCFFT2D(0,M,N,X,Y,COMM,INFO)
CALL SCFFT2D(1,M,N,X,Y,COMM,INFO)
DO J = 1, N
   DO I = 1, M/2+1
      Y(I,J) = -Y(I,J)/SQRT(REAL(M*N))
   END DO
EMD DO
CALL CSFFT2D(2,M,N,Y,X,COMM,INFO)
```

5.4.11 3D Real-To-Complex FFT

DZFFT3D Routine Documentation

DZFFT3D (*MODE,L,M,N,X,Y,COMM,INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by DZFFT3D. On input:

- *MODE*=0 : only default initializations (specific to L, M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to DZFFT3D.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER L [Input]

On input: L is the length of the first dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the first dimension of X.

INTEGER M [Input]

On input: M is the length of the second dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the second dimension of X.

INTEGER N [Input]

On input: N is the length of the third dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the third dimension of X.

DOUBLE PRECISION X(L*M*N) [Input]

On input: X contains the L by M by N real 3D array to be transformed. Element *ijk* is stored in location *i + (j - 1)* L + (k - 1) * L * M* of X.

DOUBLE COMPLEX Y((L/2+1)*M*N) [Output]

On output: Y contains the transformed Hermitian sequences in complex-Hermitian storage.

DOUBLE PRECISION COMM(4*L+6*M+6*N+500) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *L*M*N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL DZFFT3D(0,L,M,N,X,Y,COMM,INFO)
CALL DZFFT3D(1,L,M,N,X,Y,COMM,INFO)
DO K = 1, N
   DO J = 1, M
      DO I = 1, L/2+1
         Y(I,J,K) = -Y(I,J,K)/SQRT(DBLE(L*M*N))
      END DO
   END DO
EMD DO
CALL ZDFFT3D(2,L,M,N,Y,X,COMM,INFO)
```

SCFFT3D Routine Documentation

SCFFT3D (*MODE*,*L*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by SCFFT3D. On input:

- *MODE*=0 : only default initializations (specific to L, M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a real-to-complex transform is performed. Initializations are assumed to have been performed by a prior call to SCFFT3D.
- *MODE*=2 : (default) initializations and a real-to-complex transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER L [Input]

On input: L is the length of the first dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the first dimension of X.

INTEGER M [Input]

On input: M is the length of the second dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the second dimension of X.

INTEGER N [Input]

On input: N is the length of the third dimension of the 3D array of data to be transformed. If X is declared as a 3D array then L is the third dimension of X.

REAL X(L*M*N) [Input]

On input: X contains the L by M by N real 3D array to be transformed. Element *ijk* is stored in location *i + (j - 1)* L + (k - 1) * L * M* of X.

COMPLEX Y((L/2+1)*M*N) [Output]

On output: Y contains the transformed Hermitian sequences in complex-Hermitian storage.

REAL COMM(4*L+10*M+10*N+500) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *L*M*N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL SCFFT3D(0,L,M,N,X,Y,COMM,INFO)
CALL SCFFT3D(1,L,M,N,X,Y,COMM,INFO)
DO K = 1, N
   DO J = 1, M
      DO I = 1, L/2+1
         Y(I,J,K) = -Y(I,J,K)/SQRT(REAL(L*M*N))
      END DO
   END DO
EMD DO
CALL CSFFT3D(2,L,M,N,Y,X,COMM,INFO)
```

5.4.12 3D Complex-To-Real FFT

ZDFFT3D Routine Documentation

ZDFFT3D (*MODE*,*L*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by ZDFFT3D. On input:

- *MODE*=0 : only default initializations (specific to L, M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to ZDFFT3D.
- *MODE*=2 : (default) initializations and a complex-to-real transform are performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER L [Input]

On input: L is the length of the first dimension of the 3D array of data obtained from the transform. If Y is declared as a 3D array then L is the first dimension of Y.

INTEGER M [Input]

On input: M is the length of the second dimension of the 3D array of data obtained from the transform. If Y is declared as a 3D array then L is the second dimension of Y.

INTEGER N [Input]

On input: N is the length of the third dimension of the 3D array of data obtained from the transform. If Y is declared as a 3D array then L is the third dimension of Y.

DOUBLE COMPLEX X((L/2+1)*M*N) [Input]

On input: X contains the Hermitian sequences in complex-Hermitian storage to be transformed.

DOUBLE PRECISION Y(L*M*N) [Output]

On output: Y contains the L by M by N real 3D array obtained from the transform. Element *ijk* is stored in location *i + (j - 1) * L + (k - 1) * L * M* of Y.

DOUBLE PRECISION COMM(4*L+6*M+6*N+500) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *L*M*N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL DZFFT3D(0,L,M,N,X,Y,COMM,INFO)
CALL DZFFT3D(1,L,M,N,X,Y,COMM,INFO)
DO K = 1, N
   DO J = 1, M
      DO I = 1, L/2+1
         Y(I,J,K) = -Y(I,J,K)/SQRT(DBLE(L*M*N))
      END DO
   END DO
EMD DO
CALL ZDFFT3D(2,L,M,N,Y,X,COMM,INFO)
```

CSFFT3D Routine Documentation

CSFFT3D (*MODE*,*L*,*M*,*N*,*X*,*Y*,*COMM*,*INFO*) [SUBROUTINE]

INTEGER MODE [Input]

The value of *MODE* on input determines the operation performed by CSFFT3D. On input:

- *MODE*=0 : only default initializations (specific to L, M and N) are performed; this is usually followed by calls to the same routine with *MODE*=1.
- *MODE*=1 : a complex-to-real transform is performed. Initializations are assumed to have been performed by a prior call to CSFFT3D.
- *MODE*=2 : (default) initializations and a complex-to-real transform a re performed.
- *MODE*=100 : similar to *MODE*=0; only initializations are performed, but first a plan is generated. This plan is chosen based on the fastest FFT computation for a subset of all possible plans.

INTEGER L [Input]

On input: L is the length of the first dimension of the 3D array of data obtained from the transform. If Y is declared as a 3D array then L is the first dimension of Y.

INTEGER M [Input]

On input: M is the length of the second dimension of the 3D array of data obtained from the transform. If Y is declared as a 3D array then L is the second dimension of Y.

INTEGER N [Input]

On input: N is the length of the third dimension of the 3D array of data obtained from the transform. If Y is declared as a 3D array then L is the third dimension of Y.

COMPLEX X((L/2+1)*M*N) [Input]

On input: X contains the Hermitian sequences in complex-Hermitian storage to be transformed.

REAL Y(L*M*N) [Output]

On output: Y contains the L by M by N real 3D array obtained from the transform. Element *ijk* is stored in location *i + (j - 1) * L + (k - 1) * L * M* of Y.

REAL COMM(4*L+10*M+10*N+500) [Input/Output]

COMM is a communication array. Some portions of the array are used to store initializations for subsequent calls with the same sequence length *L*M*N*. The remainder is used as temporary store.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
CALL SCFFT3D(0,L,M,N,X,Y,COMM,INFO)
CALL SCFFT3D(1,L,M,N,X,Y,COMM,INFO)
DO K = 1, N
   DO J = 1, M
      DO I = 1, L/2+1
         Y(I,J,K) = -Y(I,J,K)/SQRT(REAL(L*M*N))
      END DO
   END DO
EMD DO
CALL CSFFT3D(2,L,M,N,Y,X,COMM,INFO)
```

6 Random Number Generators

Within the context of this document, a base random number generator (BRNG) is a mathematical algorithm that, given an initial state, produces a sequence (or stream) of variates (or values) uniformly distributed over the semi-open interval (0,1]. The period of the BRNG is defined as the maximum number of values that can be generated before the sequence starts to repeat. The initial state of a BRNG is often called the seed.

Note that this definition means that the value 1.0 may be returned, but the value 0.0 will not.

A pseudo-random number generator (PRNG) is a BRNG that produces a stream of variates that are independent and statistically indistinguishable from a random sequence. A PRNG has several advantages over a true random number generator in that the generated sequence is repeatable, has known mathematical properties and is usually much quicker to generate. A quasi-random number generator (QRNG) is similar to a PRNG, however the variates generated are not statistically independent, rather they are designed to give a more even distribution in multidimensional space. Many books on statistics and computer science have good introductions to PRNGs and QRNGs, see for example Knuth [6] or Banks [7]. All of the BRNGs supplied in the ACML are PRNGs.

In addition to standard PRNGs some applications require cryptologically secure generators. A PRNG is said to be cryptologically secure if there is no polynomial-time algorithm which, on input of the first l bits of the output sequence can predict the $(l + 1)$st bit of the sequence with probability significantly greater than 0.5. This is equivalent to saying there exists no polynomial-time algorithm that can correctly distinguish between an output sequence from the PRNG and a truly random sequence of the same length with probability significantly greater than 0.5 [8].

A distribution generator is a routine that takes variates generated from a BRNG and transforms them into variates from a specified distribution, for example the Gaussian (Normal) distribution.

The ACML contains five base generators, (Section 6.1 [Base Generators], page 100), and twenty-three distribution generators (Section 6.3 [Distribution Generators], page 122). In addition users can supply a custom built generator as the base generator for all of the distribution generators (Section 6.1.8 [User Supplied Generators], page 111).

The base generators were tested using the Big Crush, Small Crush and Pseudo Diehard test suites from the TestU01 software library [15].

6.1 Base Generators

The five base generators (BRNGs) supplied with the ACML are; the NAG basic generator [9], a series of Wichmann-Hill generators [10], the Mersenne Twister [11], L'Ecuyer's combined recursive generator MRG32k3a [12] and the Blum-Blum-Shub generator [8].

Some of the generators have been slightly modified from their usual form to make them consistent between themselves. For instance, the Wichmann-Hill generators in standard form may return exactly 0.0 but not exactly 1.0. In ACML we return $1.0 - x$ to convert the value x into the semi-open interval (0, 1] without affecting any other randomness properties. The original Mersenne Twister algorithm returns an exact zero about one time in a few billion; the ACML implementation returns a tiny non-zero number as surrogate for zero.

If a single stream of variates is required it is recommended that the Mersenne Twister (Section 6.1.5 [Mersenne Twister], page 109) base generator is used. This generator combines speed with good statistical properties and an extremely long period. The NAG basic generator (Section 6.1.3 [Basic NAG Generator], page 108) is another quick generator suitable for generating a single stream. However it has a shorter period than the Mersenne Twister and being a linear congruential generator, its statistical properties are not as good.

If 273 or fewer multiple streams, with a period of up to 2^{80} are required then it is recommended that the Wichmann-Hill generators are used (Section 6.1.4 [Wichmann-Hill Generator], page 109). For more streams or multiple streams with a longer period it is recommended that the L'Ecuyer combined recursive generator (Section 6.1.6 [L'Ecuyer's Combined Recursive Generator], page 110) is used in combination with the skip ahead routine (Section 6.2.3 [Skip Ahead], page 116). Generating multiple streams of variates by skipping ahead is generally quicker than generating the streams using the leap frog method. More details on multiple streams can be found in Section 6.2 [Multiple Streams], page 115.

The Blum-Blum-Shub generator (Section 6.1.7 [Blum-Blum-Shub Generator], page 110) should only be used if a cryptologically secure generator is required. This generator is extremely slow and has poor statistical properties when used as a base generator for any of the distributional generators.

6.1.1 Initialization of the Base Generators

A random number generator must be initialized before use. Three routines are supplied within the ACML for this purpose: `DRANDINITIALIZE`, `DRANDINITIALIZEBBS` and `DRANDINITIALIZEUSER` (see [DRANDINITIALIZE], page 103, [DRANDINITIALIZEBBS], page 106 and [DRANDINITIALIZEUSER], page 112, respectively). Of these, `DRANDINITIALIZE` is used to initialize all of the supplied base generators, `DRANDINITIALIZEBBS` supplies an alternative interface to `DRANDINITIALIZE` for the Blum-Blum-Shub generator, and `DRANDINITIALIZEUSER` allows the user to register and initialize their own base generator.

Both double and single precision versions of all RNG routines are supplied. Double precision names are prefixed by `DRAND`, and single precision by `SRAND`. Note that if a generator has been initialized using the relevant double precision routine, then the double precision versions of the distribution generators must also be used, and vice versa. This even applies to generators with no double or single precision parameters; for example, a call of `DRANDDISCRETEUNIFORM` must be preceded by a call to one of the double precision initializers (typically `DRANDINITIALIZE`).

No utilities for saving, retrieving or copying the current state of a generator have been provided. All of the information on the current state of a generator (or stream, if multiple streams are being used) is stored in the integer array *STATE* and as such this array can be treated as any other integer array, allowing for easy copying, restoring etc.

The statistical properties of a sequence of random numbers are only guaranteed within the sequence, and not between sequences provided by the same generator. Therefore it is likely that repeated initialization will render the numbers obtained less, rather than more, independent. In most cases there should only be a single call to one of the initialization routines, per application, and this call must be made before any variates are generated. One example of where multiple initialization may be required is briefly touched upon in Section 6.2 [Multiple Streams], page 115.

In order to initialize the Blum-Blum-Shub generator a number of additional parameters, as well as an initial state (seed), are required. Although this generator can be initialized through the `DRANDINITIALIZE` routine it is recommended that the `DRANDINITIALIZEBBS` routine is used instead.

DRANDINITIALIZE / SRANDINITIALIZE

Initialize one of the five supplied base generators; NAG basic generator, Wichmann-Hill generator, Mersenne Twister, L'Ecuyer's combined recursive generator (MRG32k3a) or the Blum-Blum-Shub generator.

(Note that SRANDINITIALIZE is the single precision version of DRANDINITIALIZE. The argument lists of both routines are identical except that any double precision arguments of DRANDINITIALIZE are replaced in SRANDINITIALIZE by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDINITIALIZE (`*`GENID`*`,`*`SUBID`*`,`*`SEED`*`,`*`LSEED`*`,`*`STATE`*`,` [SUBROUTINE]
`LSTATE`*`,`*`INFO`*`)`

`INTEGER GENID` [Input]

On input: a numerical code indicating which of the five base generators to initialize.

- 1 = NAG basic generator (Section 6.1.3 [Basic NAG Generator], page 108).
- 2 = Wichmann-Hill generator (Section 6.1.4 [Wichmann-Hill Generator], page 109).
- 3 = Mersenne Twister (Section 6.1.5 [Mersenne Twister], page 109).
- 4 = L'Ecuyer's Combined Recursive generator (Section 6.1.6 [L'Ecuyer's Combined Recursive Generator], page 110).
- 5 = Blum-Blum-Shub generator (Section 6.1.7 [Blum-Blum-Shub Generator], page 110).

Constraint: $1 \leq GENID \leq 5$.

`INTEGER SUBID` [Input]

On input: if $GENID = 2$, then *SUBID* indicates which of the 273 Wichmann-Hill generators to use. If $GENID = 5$ then *SUBID* indicates the number of bits to use (v) from each of iteration of the Blum-Blum-Shub generator. In all other cases *SUBID* is not referenced.
Constraint: If $GENID = 2$ then $1 \leq SUBID \leq 273$.

`INTEGER SEED(`*`LSEED`*`)` [Input]

On input: if $GENID \neq 5$, then *SEED* is a vector of initial values for the base generator. These values must be positive integers. The number of values required depends on the base generator being used. The NAG basic generator requires one initial value, the Wichmann-Hill generator requires four initial values, the L'Ecuyer combined recursive generator requires six initial values and the Mersenne Twister requires 624 initial values. If the number of seeds required by the chosen generator is $> LSEED$ then $SEED(1)$ is used to initialize the NAG basic generator. This is then used to generate all of the remaining seed values required. In general it is best not to set all the elements of *SEED* to anything too obvious, such as a single repeated value or a simple sequence. Using such a seed array may lead to several similar values being created in a row when the generator is subsequently called. This is particularly true for the Mersenne Twister generator.

In order to initialize the Blum-Blum-Shub generator two large prime values, p and q are required as well as an initial value s. As p, q and s can be of an arbitrary size, these values are expressed as a polynomial in B, where $B = 2^{24}$. For example, p can be factored into a polynomial of order l_p, with $p = p_1 + p_2B + p_3B^2 + \cdots + p_{l_p}B^{l_p-1}$. The elements of *SEED* should then be set to the following:

- $SEED(1) = l_p$
- $SEED(2)$ to $SEED(l_p + 1) = p_1$ to p_{l_p}
- $SEED(l_p + 2) = l_q$
- $SEED(l_p + 3)$ to $SEED(l_p + l_q + 2) = q_1$ to q_{l_q}
- $SEED(l_p + l_q + 3) = l_s$
- $SEED(l_p + l_q + 4)$ to $SEED(l_p + l_q + l_s + 3) = s_1$ to s_{l_s}

Constraint: If *GENID*$\neq 5$ then $SEED(i) > 0, i = 1, 2, \cdots$. If *GENID* $= 5$ then *SEED* must take the values described above.

`INTEGER LSEED` [Input/Output]

On input: either the length of the seed vector, *SEED*, or a value ≤ 0 .
On output: if *LSEED*≤ 0 on input, then *LSEED* is set to the number of initial values required by the selected generator, and the routine returns. Otherwise *LSEED* is left unchanged.

`INTEGER STATE(`*LSTATE*`)` [Output]

On output: the state vector required by all of the supplied distributional and base generators.

`INTEGER LSTATE` [Input/Output]

On input: either the length of the state vector, *STATE*, or a value ≤ 0 .
On output: if *LSTATE*≤ 0 on input, then *LSTATE* is set to the minimum length of the state vector *STATE* for the base generator chosen, and the routine returns. Otherwise *LSTATE* is left unchanged.
Constraint: *LSTATE*≤ 0 or the minimum length for the chosen base generator, given by:

- *GENID* $= 1$: *LSTATE*≥ 16,
- *GENID* $= 2$: *LSTATE*≥ 20,
- *GENID* $= 3$: *LSTATE*≥ 633,
- *GENID* $= 4$: *LSTATE*≥ 61,
- *GENID* $= 5$: *LSTATE*$\geq l_p + l_q + l_s + 6$, where l_p, l_q and l_s are the order of the polynomials used to express the parameters p, q and s respectively.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. If *INFO* $= -$i on exit, the i-th argument had an illegal value. If *INFO* $= 1$ on exit, then either, or both of *LSEED* and / or *LSTATE* have been set to the required length for vectors *SEED* and *STATE* respectively. Of the two variables *LSEED* and *LSTATE*, only those which had an input value ≤ 0 will have been set. The *STATE* vector will not have been initialized. If *INFO* $= 0$ then the state vector, *STATE*, has been successfully initialized.

Example:

```
C      Generate 100 values from the Beta distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A,B
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Beta distribution
       CALL DRANDBETA(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDINITIALIZEBBS / SRANDINITIALIZEBBS

Alternative initialization routine for the Blum-Blum-Shub generator. Unlike the other base generators supplied with the ACML, the Blum-Blum-Shub generator requires two additional parameters, p and q as well as an initial state, s. The parameters p, q and s can be of an arbitrary size. In order to avoid overflow these values are expressed as a polynomial in B, where $B = 2^{24}$. For example, p can be factored into a polynomial of order l_p, with $p = p_1 + p_2B + p_3B^2 + \cdots + p_{l_p}B^{l_p-1}$, similarly $q = q_1 + q_2B + q_3B^2 + \cdots + q_{l_q}B^{l_q-1}$ and $s = s_1 + s_2B + s_3B^2 + \cdots + s_{l_s}B^{l_s-1}$.

(Note that SRANDINITIALIZEBBS is the single precision version of DRANDINITIALIZEBBS. The argument lists of both routines are identical except that any double precision arguments of DRANDINITIALIZEBBS are replaced in SRANDINITIALIZEBBS by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDINITIALIZEBBS (`*`NBITS,LP,P,LQ,Q,LS,S,STATE,LSTATE,`* [SUBROUTINE]
`INFO``)`

`INTEGER NBITS` [Input]
On input: the number of bits, v, to use from each iteration of the Blum-Blum-Shub generator. If $NBITS < 1$ then $NBITS = 1$. If $NBITS > 15$ then $NBITS = 15$.

`INTEGER LP` [Input]
On input: the order of the polynomial used to express p (l_p).
Constraint: $1 \leq LP \leq 25$.

`INTEGER P(`*`LP`*`)` [Input]
On input: the coefficients of the polynomial used to express p. $P(i) = p_i, i = 1$ to l_p.
Constraint: $0 \leq P\ (i) < 2^{24}$

`INTEGER LQ` [Input]
On input: the order of the polynomial used to express q (l_q).
Constraint: $1 \leq LQ \leq 25$.

`INTEGER Q(`*`LQ`*`)` [Input]
On input: the coefficients of the polynomial used to express q. $Q(i) = q_i, i = 1$ to l_q.
Constraint: $0 \leq Q\ (i) < 2^{24}$

`INTEGER LS` [Input]
On input: the order of the polynomial used to express s (l_s).
Constraint: $1 \leq LS \leq 25$.

`INTEGER S(`*`LS`*`)` [Input]
On input: the coefficients of the polynomial used to express s. $S(i) = s_i, i = 1$ to l_s.
Constraint: $0 \leq S\ (i) < 2^{24}$

`INTEGER STATE(*)` [Output]
On output: the initial state for the Blum-Blum-Shub generator with parameters P,Q,S and $NBITS$.

`INTEGER LSTATE` [Input/Output]

On input: either the length of the state vector, *STATE*, or a value ≤ 0 .
On output: if *LSTATE*≤ 0 on input, then *LSTATE* is set to the minimum length of the state vector *STATE* for the parameters chosen, and the routine returns. Otherwise *LSTATE* is left unchanged.
Constraint: *LSTATE*≤ 0 or *LSTATE* $\geq l_p + l_q + l_s + 6$

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. If *INFO* $= -$i on exit, the i-th argument had an illegal value. If *INFO* $= 1$ on exit, then *LSTATE* has been set to the required length for the *STATE* vector. If *INFO* $= 0$ then the state vector, *STATE*, has been successfully initialized.

6.1.2 Calling the Base Generators

With the exception of the Blum-Blum-Shub generator, there are no interfaces for direct access to the base generators. All of the base generators return variates uniformly distributed over the semi-open interval $(0, 1]$. This functionality can be accessed using the uniform distributional generator `DRANDUNIFORM`, with parameter $A = 0.0$ and parameter $B = 1.0$ (see [DRANDUNIFORM], page 144). The base generator used is, as usual, selected during the initialization process (see Section 6.1.1 [Initialization of the Base Generators], page 101).

To directly access the Blum-Blum-Shub generator, use the routine `DRANDBLUMBLUMSHUB`.

`DRANDBLUMBLUMSHUB / SRANDBLUMBLUMSHUB`

Allows direct access to the bit stream generated by the Blum-Blum-Shub generator.

(Note that SRANDBLUMBLUMSHUB is the single precision version of DRANDBLUMBLUMSHUB. The argument lists of both routines are identical except that any double precision arguments of DRANDBLUMBLUMSHUB are replaced in SRANDBLUMBLUMSHUB by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDBLUMBLUMSHUB (`*N*,*STATE*,*X*,*INFO*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required. The total number of bits generated is $24N$.
Constraint: $N \geq 0$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDBLUMBLUMSHUB` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`INTEGER X(`*N*`)` [Output]
On output: vector holding the bit stream. The least significant 24 bits of each of the $X(i)$ contain the bit stream as generated by the Blum-Blum-Shub generator. The least significant bit of $X(1)$ is the first bit generated, the second least significant bit of $X(1)$ is the second bit generated etc.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* $= -$i on exit, the i-th argument had an illegal value.

6.1.3 Basic NAG Generator

The NAG basic generator is a linear congruential generator (LCG) and, like all LCGs, has the form:

$$x_i = a_1 x_{i-1} \bmod m_1,$$
$$u_i = \frac{x_i}{m_1},$$

where the $u_i, i = 1, 2, \cdots$ form the required sequence.

The NAG basic generator takes $a_1 = 13^{13}$ and $m_1 = 2^{59}$, which gives a period of approximately 2^{57}. This generator has been part of the NAG numerical library [9] since Mark 6 and as such has been widely used. It suffers from no known problems, other than those due to the lattice structure inherent in all LCGs, and, even though the period is relatively short compared to many of the newer generators, it is sufficiently large for many practical problems.

6.1.4 Wichmann-Hill Generator

The Wichmann-Hill [10] base generator uses a combination of four linear congruential generators (LCGs) and has the form:

$$\begin{aligned} w_i &= a_1 w_{i-1} \bmod m_1 \\ x_i &= a_2 x_{i-1} \bmod m_2 \\ y_i &= a_3 y_{i-1} \bmod m_3 \\ z_i &= a_4 z_{i-1} \bmod m_4 \\ u_i &= (\frac{w_i}{m_1} + \frac{x_i}{m_2} + \frac{y_i}{m_3} + \frac{z_i}{m_4}) \bmod 1, \end{aligned}$$

where the $u_i, i = 1, 2, \cdots$ form the required sequence. There are 273 sets of parameters, $\{a_i, m_i : i = 1, 2, 3, 4\}$, to choose from. These values have been selected so that the resulting generators are independent and have a period of approximately 2^{80} [10].

6.1.5 Mersenne Twister

The Mersenne Twister [11] is a twisted generalized feedback shift register generator. The algorithm is as follows:

- Set some arbitrary initial values $x_1, x_2, \cdots, x_r$, each consisting of w bits.
- Letting

  $$A = \begin{pmatrix} 0 & I_{w-1} \\ a_w & a_{w-1} \cdots a_1 \end{pmatrix},$$

  where I_{w-1} is the $(w-1) \times (w-1)$ identity matrix and each of the $a_i, i = 1$ to w take a value of either 0 or 1 (i.e. they can be represented as bits). Define

  $$x_{i+r} = (x_{i+s} \oplus (x_i^{(w:(l+1))} | x_{i+1}^{(l:1)}) A),$$

  where $x_i^{(w:(l+1))} | x_{i+1}^{(l:1)}$ indicates the concatenation of the most significant (upper) $w - l$ bits of x_i and the least significant (lower) l bits of x_{i+1}.
- Perform the following operations sequentially:

  $$\begin{aligned} z &= x_{i+r} \oplus (x_{i+r} \gg t_1) \\ z &= z \oplus ((z \ll t_2) \text{ AND } m_1) \\ z &= z \oplus ((z \ll t_3) \text{ AND } m_2) \\ z &= z \oplus (z \gg t_4) \\ u_{i+r} &= z/(2^w - 1), \end{aligned}$$

  where t_1, t_2, t_3 and t_4 are integers and m_1 and m_2 are bit-masks and "$\gg t$" and "$\ll t$" represent a t bit shift right and left respectively, $\oplus$ is bit-wise exclusively or (xor) operation and "AND" is a bit-wise and operation.

The $u_{i+r} : i = 1, 2, \cdots$ then form a pseudo-random sequence, with $u_i \in (0, 1)$, for all i. This implementation of the Mersenne Twister uses the following values for the algorithmic constants:

$$\begin{aligned} w &= 32 \\ a &= \text{0x9908b0df} \\ l &= 31 \\ r &= 624 \\ s &= 397 \\ t_1 &= 11 \\ t_2 &= 7 \\ t_3 &= 15 \\ t_4 &= 18 \\ m_1 &= \text{0x9d2c5680} \\ m_2 &= \text{0xefc60000} \end{aligned}$$

where the notation $0xDD\cdots$ indicates the bit pattern of the integer whose hexadecimal representation is $DD\cdots$.

This algorithm has a period length of approximately $2^{19,937} - 1$ and has been shown to be uniformly distributed in 623 dimensions.

6.1.6 L'Ecuyer's Combined Recursive Generator

The base generator referred to as L'Ecuyer's combined recursive generator is referred to as MRG32k3a in [12] and combines two multiple recursive generators:

$$\begin{aligned} x_i &= a_{11}x_{i-1} + a_{12}x_{i-2} + a_{13}x_{i-3} \bmod m_1 \\ y_i &= a_{21}y_{i-1} + a_{22}y_{i-2} + a_{23}y_{i-3} \bmod m_2 \\ z_i &= x_i - y_i \bmod m_1 \\ u_i &= \frac{z_i}{m_1}, \end{aligned}$$

where the $u_i, i = 1, 2, \cdots$ form the required sequence and $a_{11} = 0, a_{12} = 1403580, a_{13} = -810728, m_1 = 2^{32} - 209, a_{21} = 527612, a_{22} = 0, a_{23} = -1370589$ and $m_2 = 2^{32} - 22853$.

Combining the two multiple recursive generators (MRG) results in sequences with better statistical properties in high dimensions and longer periods compared with those generated from a single MRG. The combined generator described above has a period length of approximately 2^{191}

6.1.7 Blum-Blum-Shub Generator

The Blum-Blum-Shub pseudo random number generator is cryptologically secure under the assumption that the quadratic residuosity problem is intractable [8]. The algorithm consists of the following:

- Generate two large and distinct primes, p and q, each congruent to 3 mod 4. Define $m = pq$.
- Select a seed s taking a value between 1 and $m - 1$, such that the greatest common divisor between s and m is 1.

- Let $x_0 = s^2 \bmod m$. For $i = 1, 2, \cdots$ generate:

$$x_i = x_{i-1}^2 \bmod m$$
$$z_i = v \text{ least significant bits of } x_i$$

  where $v \geq 1$.
- The bit-sequence $z_1, z_2, z_3, \cdots$ is then the output sequence used.

6.1.8 User Supplied Generators

All of the distributional generators described in Section 6.3 [Distribution Generators], page 122 require a base generator which returns a uniformly distributed value in the semi-open interval $(0, 1]$ and ACML includes several such generators (as detailed in Section 6.1 [Base Generators], page 100). However, for greater flexibility, the ACML routines allow the user to register their own base generator function. This user-supplied generator then becomes the base generator for all of the distribution generators.

A user supplied generator comes in the form of two routines, one to initialize the generator and one to generate a set of uniformly distributed values in the semi-open interval $(0, 1]$. These two routines can be named anything, but are referred to as `UINI` for the initialization routine and `UGEN` for the generation routine in the following documentation.

In order to register a user supplied generator a call to `DRANDINITIALIZEUSER` must be made. Once registered the generator can be accessed and used in the same manner as the ACML supplied base generators. The specifications for `DRANDINTIALIZEUSER`, `UINI` and `UGEN` are given below. See the ACML example programs `drandinitializeuser_example.f` and `drandinitializeuser_c_example.c` (Section 2.8 [Examples], page 12) to understand how to use these routines in ACML.

DRANDINITIALIZEUSER / SRANDINITIALIZEUSER

Registers a user supplied base generator so that it can be used with the ACML distributional generators.

(Note that SRANDINITIALIZEUSER is the single precision version of DRANDINITIALIZEUSER. The argument lists of both routines are identical except that any double precision arguments of DRANDINITIALIZEUSER are replaced in SRANDINITIALIZEUSER by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDINITIALIZEUSER (*UINI*,*UGEN*,*GENID*,*SUBID*,*SEED*,*LSEED*, *STATE*,*LSTATE*,*INFO*) [SUBROUTINE]

SUBROUTINE UINI [Input]
: On input: routine that will be used to initialize the user supplied generator, *UGEN*.

SUBROUTINE UGEN [Input]
: On input: user supplied base generator.

INTEGER GENID [Input]
: On input: parameter is passed directly to *UINI*. Its function therefore depends on that routine.

INTEGER SUBID [Input]
: On input: parameter is passed directly to *UINI*. Its function therefore depends on that routine.

INTEGER SEED(*LSEED*) [Input]
: On input: parameter is passed directly to *UINI*. Its function therefore depends on that routine.

INTEGER LSEED [Input/Output]
: On input: length of the vector *SEED*. This parameter is passed directly to *UINI* and therefore its required value depends on that routine.
On output: whether *LSEED* changes will depend on *UINI*.

INTEGER STATE(*LSTATE*) [Output]
: On output: the state vector required by all of the supplied distributional generators. The value of *STATE* returned by *UINI* has some housekeeping elements appended to the end before being returned by DRANDINITIALIZEUSER. See Section 6.1.8 [User Supplied Generators], page 111 for details about the form of *STATE*.

INTEGER LSTATE [Input/Output]
: On input: length of the vector *STATE*. This parameter is passed directly to *UINI* and therefore its required value depends on that routine.
On output: whether *LSTATE* changes will depend on *UINI*. If $LSTATE \leq 0$ then it is assumed that a request for the required length of *STATE* has been made. The value of *LSTATE* returned from *UINI* is therefore adjusted to allow for housekeeping elements to be added to the end of the *STATE* vector. This results in the value of *LSTATE* returned by DRANDINITIALIZEUSER being 3 larger than that returned by *UINI*.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. `DRANDINITIALIZEUSER` will return a value of -6 if the value of *LSTATE* is between 1 and 3. Otherwise *INFO* is passed directly back from *UINI*. It is recommended that the value of *INFO* returned by *UINI* is kept consistent with the rest of the ACML, that is if *INFO* $= -$i on exit, the i-th argument had an illegal value. If *INFO* $= 1$ on exit, then either, or both of *LSEED* and / or *LSTATE* have been set to the required length for vectors *SEED* and *STATE* respectively and the *STATE* vector has not have been initialized. If *INFO* $= 0$ then the state vector, *STATE*, has been successfully initialized.

Example:

```
C     Generate 100 values from the Uniform distribution using
C     a user supplied base generator
      INTEGER LSTATE,N
      PARAMETER (LSTATE=16,N=100)
      INTEGER I,INFO,NSKIP,SEED(1),STATE(LSTATE)
      INTEGER X(N)
      DOUBLE PRECISION A,B

C     Set the seed
      SEED(1) = 1234

C     Set the distributional parameters
      A = 0.0D0
      B = 1.0D0

C     Initialize the base generator. Here ACMLRNGNBOGND is a user
C     supplied generator and ACMLRNGNBOINI its initializer
      CALL DRANDINITIALIZEUSER(ACMLRNGNBOINI,ACMLRNGNBOGND,1,0,SEED,
     *    LSEED,STATE,LSTATE,INFO)

C     Generate N variates from the Univariate distribution
      CALL DRANDUNIFORM(N,A,B,STATE,X,LDX,INFO)

C     Print the results
      WRITE(6,*) (X(I),I=1,N)
```

UINI

Specification for a user supplied initialization routine.

UINI (*GENID*,*SUBID*,*SEED*,*LSEED*,*STATE*,*LSTATE*,*INFO*) [SUBROUTINE]

INTEGER GENID [Input]

On input: the ID associated with the generator. It may be used for anything you like.

INTEGER SUBID [Input]

On input: the sub-ID associated with the generator. It may be used for anything you like.

INTEGER SEED(*LSEED*) [Input]

On input: an array containing the initial seed for your generator.

INTEGER LSEED [Input/Output]

On input: either the size of the *SEED* array, or a value < 1.
On output: if $LSEED < 1$ on entry, *LSEED* must be set to the required size of the *SEED* array. This allows a caller of UINI to query the required size.

INTEGER STATE(*LSTATE*) [Output]

On output: if $LSTATE < 1$ on entry, *STATE* should be unchanged.
Otherwise, *STATE* is a state vector holding internal details required by your generator. On exit from UINI, the array *STATE* must hold the following information:

STATE(1) = ESTATE, where ESTATE is your minimum allowed size of array *STATE*.

STATE(2) = MAGIC, where MAGIC is a magic number of your own choice. This can be used by your routine UGEN as a check that UINI has previously been called.

STATE(3) = GENID

STATE(4) = SUBID

STATE(5) ... STATE(ESTATE-1) = internal state values required by your generator routine UGEN; for example, the current value of your seed.

STATE(ESTATE) = MAGIC, i.e. the same value as STATE(2).

INTEGER LSTATE [Input/Output]

On input: either the size of the *STATE* array, or a value < 1.
On output: if $LSTATE < 1$ on entry, *LSTATE* should be set to the required size of the *STATE* array, i.e. the value ESTATE as described above. This allows the caller of UINI to query the required size.
Constraint: either $LSTATE < 1$ or $LSTATE \geq ESTATE$.

INTEGER INFO [Output]

On output: an error code, to be used in whatever way you wish; for example to flag an incorrect argument to UINI. If no error is encountered, UINI must set *INFO* to 0.

`UGEN`

Specification for a user supplied base generator.

`UGEN (N,STATE,X,INFO)` [SUBROUTINE]

`INTEGER N` [Input]
On input: the number of random numbers to be generated.

`INTEGER STATE(*)` [Input/Output]
On input: the internal state of your generator.

`DOUBLE PRECISION X(N)` [Output]
On output: the array of N uniform distributed random numbers, each in the semi-open interval $(0.0, 1.0]$ - i.e. 1.0 is a legitimate return value, but 0.0 is not.

`INTEGER INFO` [Output]
On output: a flag which you can use to signal an error in the call of `UGEN` - for example, if `UGEN` is called without being initialized by `UINI`.

6.2 Multiple Streams

It is often advantageous to be able to generate variates from multiple, independent, streams. For example when running a simulation in parallel on several processors. There are four ways of generating multiple streams using the routines available in the ACML:

- (a) Using different seeds
- (b) Using different sequences
- (c) Block-splitting or skipping ahead
- (d) Leap frogging

The four methods are detailed in the following sections. Of the four, (a) should be avoided in most cases, (b) is only really of any practical use when using the Wichmann-Hill generator, and is then still limited to 273 streams. Both block-splitting and leap-frogging work using the sequence from a single generator, both guarantee that the different sequences will not overlap and both can be scaled to an arbitrary number of streams. Leap-frogging requires no *a-priori* knowledge about the number of variates being generated, whereas block-splitting requires the user to know (approximately) the maximum number of variates required from each stream. Block-splitting requires no *a-priori* information on the number of streams required. In contrast leap-frogging requires the user to know the maximum number of streams required, prior to generating the first value.

It is known that, dependent on the number of streams required, leap-frogging can lead to sequences with poor statistical properties, especially when applied to linear congruential generators (see Section 6.2.4 [Leap Frogging], page 119 for a brief explanation). In addition, for more complicated generators like a L'Ecuyer's multiple recursive generator leap-frogging can increase the time required to generate each variate compared to block-splitting. The additional time required by block-splitting occurs at the initialization stage, and not at the variate generation stage. Therefore in most instances block-splitting would be the preferred method for generating multiple sequences.

6.2.1 Using Different Seeds

A different sequence of variates can be generated from the same base generator by initializing the generator using a different set of seeds. Of the four methods for creating multiple streams described here, this is the least satisfactory. As mentioned in Section 6.1.1 [Initialization of the Base Generators], page 101, the statistical properties of the base generators are only guaranteed within sequences, not between sequences. For example, sequences generated from different starting points may overlap if the initial values are not far enough apart. The potential for overlapping sequences is reduced if the period of the generator being used is large. Although there is no guarantee of the independence of the sequences, due to its extremely large period, using the Mersenne Twister with random starting values is unlikely to lead to problems, especially if the number of sequences required is small. This is the only way in which multiple sequences can be generated with the ACML using the Mersenne Twister as the base generator.

If the statistical properties of different sequences must be provable then one of the other methods should be adopted.

6.2.2 Using Different Generators

Independent sequences of variates can be generated using different base generators for each sequence. For example, sequence 1 can be generated using the NAG basic generator, sequence 2 using the L'Ecuyer's Combined Recursive generator, sequence 3 using the Mersenne Twister. The Wichmann-Hill generator implemented in the ACML is in fact a series of 273 independent generators. The particular sub-generator being used can be selected using the *SUBID* variable (see [DRANDINITIALIZE], page 103 for details). Therefore, in total, 277 independent streams can be generated with each using an independent generator (273 Wichmann-Hill generators, and 4 additional base generators).

6.2.3 Skip Ahead

Independent sequences of variates can be generated from a single base generator through the use of block-splitting, or skipping-ahead. This method consists of splitting the sequence into k non-overlapping blocks, each of length n, where n is larger than the maximum number of variates required from any of the sequences. For example:

$$\underbrace{x_1, x_2, \cdots, x_n,}_{\text{block 1}} \ \underbrace{x_{n+1}, x_{n+2}, \cdots, x_{2n},}_{\text{block 2}} \ \underbrace{x_{2n+1}, x_{2n+2}, \cdots, x_{3n},}_{\text{block 3}} \ \text{etc}$$

where $x_1, x_2, \cdots$ is the sequence produced by the generator of interest. Each of the k blocks provide an independent sequence.

The block splitting algorithm therefore requires the sequence to be advanced a large number of places. Due to their form this can be done efficiently for linear congruential generators and multiple congruential generators. The ACML provides block-splitting for the NAG Basic generator, the Wichmann-Hill generators and L'Ecuyer's Combined Recursive generator.

DRANDSKIPAHEAD / SRANDSKIPAHEAD

Advance a generator N places.

(Note that SRANDSKIPAHEAD is the single precision version of DRANDSKIPAHEAD. The argument lists of both routines are identical except that any double precision arguments of DRANDSKIPAHEAD are replaced in SRANDSKIPAHEAD by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDSKIPAHEAD (`*`N,STATE,INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]

On input: number of places to skip ahead.
Constraint: $N \geq 0$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDSKIPAHEAD` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: The *STATE* vector for a generator that has been advanced N places.
Constraint: The *STATE* vector must be for either the NAG basic, Wichmann-Hill or L'Ecuyer Combined Recursive base generators.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If $INFO = -$i on exit, the i-th argument had an illegal value.

Example:

```
C     Generate 3 * 100 values from the Uniform distribution
C     Multiple streams generated using the Skip Ahead method
      INTEGER LSTATE,N
      PARAMETER (LSTATE=16,N=100)
      INTEGER I,INFO,NSKIP
      INTEGER SEED(1),STATE1(LSTATE),STATE2(LSTATE),STATE3(LSTATE)
      DOUBLE PRECISION X1(N),X2(N),X3(N)
      DOUBLE PRECISION A,B

C     Set the seed
      SEED(1) = 1234

C     Set the distributional parameters
      A = 0.0D0
      B = 1.0D0

C     Initialize the STATE1 vector
      CALL DRANDINITIALIZE(1,1,SEED,1,STATE1,LSTATE,INFO)

C     Copy the STATE1 vector into other state vectors
      DO 20 I = 1,LSTATE
        STATE2(I) = STATE1(I)
        STATE3(I) = STATE1(I)
20    CONTINUE

C     Calculate how many places we want to skip, this
C     should be >> than the number of variates we
C     wish to generate from each stream
      NSKIP = N * N

C     Advance each stream, first does not need changing
      CALL DRANDSKIPAHEAD(NSKIP,STATE2,INFO)
      CALL DRANDSKIPAHEAD(2*NSKIP,STATE3,INFO)

C     Generate 3 sets of N variates from the Univariate distribution
      CALL DRANDUNIFORM(N,A,B,STATE1,X1,LDX,INFO)
      CALL DRANDUNIFORM(N,A,B,STATE2,X2,LDX,INFO)
      CALL DRANDUNIFORM(N,A,B,STATE3,X3,LDX,INFO)

C     Print the results
      DO 40 I = 1,N
        WRITE(6,*) X1(I),X2(I),X3(I)
40    CONTINUE
```

6.2.4 Leap Frogging

Independent sequences of variates can be generated from a single base generator through the use of leap-frogging. This method involves splitting the sequence from a single generator into k disjoint subsequences. For example:

$$\begin{aligned}
&\text{Subsequence 1} : x_1, x_{k+1}, x_{2k+1}, \cdots \\
&\text{Subsequence 2} : x_2, x_{k+2}, x_{2k+2}, \cdots \\
&\qquad\qquad \vdots \\
&\text{Subsequence k} : x_k, x_{2k}, x_{3k}, \cdots
\end{aligned}$$

each subsequence is then provides an independent stream.

The leap-frog algorithm therefore requires the generation of every kth variate of a sequence. Due to their form this can be done efficiently for linear congruential generators and multiple congruential generators. The ACML provides leap-frogging for the NAG Basic generator, the Wichmann-Hill generators and L'Ecuyer's Combined Recursive generator.

As an illustrative example, a brief description of the algebra behind the implementation of the leap-frog algorithm (and block-splitting algorithm) for a linear congruential generator (LCG) will be given. A linear congruential generator has the form $x_{i+1} = a_1 x_i \bmod m_1$. The recursive nature of a LCG means that

$$\begin{aligned}
x_{i+v} &= a_1 x_{i+v-1} \bmod m_1 \\
&= a_1 (a_1 x_{i+v-2} \bmod m_1) \bmod m_1 \\
&= a_1^2 x_{i+v-2} \bmod m_1 \\
&= a_1^v x_i \bmod m_1
\end{aligned}$$

The sequence can be quickly advanced v places by multiplying the current state (x_i) by $a_1^v \bmod m_1$, hence allowing block-splitting. Leap-frogging is implemented by using a_1^k, where k is the number of streams required, in place of a_1 in the standard LCG recursive formula. In a linear congruential generator the multiplier a_1 is constructed so that the generator has good statistical properties in, for example, the spectral test. When using leap-frogging to construct multiple streams this multiplier is replaced with a_1^k, and there is no guarantee that this new multiplier will have suitable properties especially as the value of k depends on the number of streams required and so is likely to change depending on the application. This problem can be emphasised by the lattice structure of LCGs.

Note that, due to rounding, a sequence generated using leap-frogging and a sequence constructed by taking every kth value from a set of variates generated without leap-frogging may differ slightly. These differences should only affect the least significant digit.

DRANDLEAPFROG / SRANDLEAPFROG

Amend a generator so that it will generate every Kth value.

(Note that SRANDLEAPFROG is the single precision version of DRANDLEAPFROG. The argument lists of both routines are identical except that any double precision arguments of DRANDLEAPFROG are replaced in SRANDLEAPFROG by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDLEAPFROG` (*N*,*K*,*STATE*,*INFO*) [SUBROUTINE]

`INTEGER N` [Input]

On input: total number of streams being used.
Constraint: $N > 0$.

`INTEGER K` [Input]

On input: number of the current stream
Constraint: $0 < K \leq N$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDLEAPFROG` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: The *STATE* vector for a generator that has been advanced $K - 1$ places and will return every Nth value.
Constraint: The *STATE* array must be for either the NAG basic, Wichmann-Hill or L'Ecuyer Combined Recursive base generators.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C     Generate 3 * 100 values from the Uniform distribution
C     Multiple streams generated using the Leap Frog method
      INTEGER LSTATE,N
      PARAMETER (LSTATE=16,N=100)
      INTEGER I,INFO
      INTEGER SEED(1),STATE1(LSTATE),STATE2(LSTATE),STATE3(LSTATE)
      DOUBLE PRECISION X1(N),X2(N),X3(N)
      DOUBLE PRECISION A,B

C     Set the seed
      SEED(1) = 1234

C     Set the distributional parameters
      A = 0.0D0
      B = 1.0D0

C     Initialize the STATE1 vector
      CALL DRANDINITIALIZE(1,1,SEED,1,STATE1,LSTATE,INFO)

C     Copy the STATE1 vector into other state vectors
      DO 20 I = 1,LSTATE
        STATE2(I) = STATE1(I)
        STATE3(I) = STATE1(I)
20    CONTINUE

C     Update each stream so they generate every 3rd value
      CALL DRANDLEAPFROG(3,1,STATE1,INFO)
      CALL DRANDLEAPFROG(3,2,STATE2,INFO)
      CALL DRANDLEAPFROG(3,3,STATE3,INFO)

C     Generate 3 sets of N variates from the Univariate distribution
      CALL DRANDUNIFORM(N,A,B,STATE1,X1,LDX,INFO)
      CALL DRANDUNIFORM(N,A,B,STATE2,X2,LDX,INFO)
      CALL DRANDUNIFORM(N,A,B,STATE3,X3,LDX,INFO)

C     Print the results
      DO 40 I = 1,N
        WRITE(6,*) X1(I),X2(I),X3(I)
40    CONTINUE
```

6.3 Distribution Generators

6.3.1 Continuous Univariate Distributions

`DRANDBETA / SRANDBETA`

Generates a vector of random variates from a beta distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{\Gamma(A+B)}{\Gamma(A)\Gamma(B)} X^{A-1}(1-X)^{B-1}$$

if $0 \leq X \leq 1$ and $A, B > 0.0$, otherwise $f(X) = 0$.

(Note that SRANDBETA is the single precision version of DRANDBETA. The argument lists of both routines are identical except that any double precision arguments of DRANDBETA are replaced in SRANDBETA by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDBETA (`*N*,*A*,*B*,*STATE*,*X*,*INFO*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION A` [Input]
On input: first parameter for the distribution.
Constraint: $A > 0$.

`DOUBLE PRECISION B` [Input]
On input: second parameter for the distribution.
Constraint: $B > 0$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDBETA` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*N*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Beta distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A,B
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Beta distribution
       CALL DRANDBETA(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDCAUCHY / SRANDCAUCHY

Generates a vector of random variates from a Cauchy distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{1}{\pi B(1 + (\frac{X-A}{B})^2)}$$

(Note that SRANDCAUCHY is the single precision version of DRANDCAUCHY. The argument lists of both routines are identical except that any double precision arguments of DRANDCAUCHY are replaced in SRANDCAUCHY by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDCAUCHY (`*`N`*`,`*`A`*`,`*`B`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION A` [Input]
On input: median of the distribution.

`DOUBLE PRECISION B` [Input]
On input: semi-quartile range of the distribution.
Constraint: $B \geq 0$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDCAUCHY` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Cauchy distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A,B
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Cauchy distribution
       CALL DRANDCAUCHY(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDCHISQUARED / SRANDCHISQUARED

Generates a vector of random variates from a χ^2 distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{X^{\frac{\nu}{2}-1}e^{-\frac{X}{2}}}{2^{\frac{\nu}{2}}(\frac{\nu}{2}-1)!},$$

if $X > 0$, otherwise $f(X) = 0$. Here ν is the degrees of freedom, *DF*.

(Note that SRANDCHISQUARED is the single precision version of DRANDCHISQUARED. The argument lists of both routines are identical except that any double precision arguments of DRANDCHISQUARED are replaced in SRANDCHISQUARED by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDCHISQUARED (`*`N`*`,`*`DF`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]

On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER DF` [Input]

On input: degrees of freedom of the distribution.
Constraint: $DF > 0$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDCHISQUARED` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]

On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Chi-squared distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER DF
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) DF

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Chi-squared distribution
       CALL DRANDCHISQUARED(N,DF,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDEXPONENTIAL / SRANDEXPONENTIAL

Generates a vector of random variates from an exponential distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{e^{-\frac{X}{A}}}{A}$$

if $X > 0$, otherwise $f(X) = 0$.

(Note that SRANDEXPONENTIAL is the single precision version of DRANDEXPONENTIAL. The argument lists of both routines are identical except that any double precision arguments of DRANDEXPONENTIAL are replaced in SRANDEXPONENTIAL by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDEXPONENTIAL (`*`N`*`,`*`A`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]

On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION A` [Input]

On input: exponential parameter.
Constraint: $A \geq 0$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDEXPONENTIAL` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]

On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Exponential distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Exponential distribution
       CALL DRANDEXPONENTIAL(N,A,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDF / SRANDF

Generates a vector of random variates from an F distribution, also called the Fisher's variance ratio distribution, with probability density function, $f(X)$, where:

$$f(X) = \frac{(\frac{\mu+\nu-2}{2})! X^{\frac{\mu}{2}-1} \mu^{\frac{\mu}{2}}}{(\frac{\mu}{2}-1)!(\frac{\nu}{2}-1)!(1+\frac{\mu X}{\nu})^{\frac{\mu+\nu}{2}} \nu^{\frac{\mu}{2}}},$$

if $X > 0$, otherwise $f(X) = 0$. Here μ is the first degrees of freedom, (*DF1*) and ν is the second degrees of freedom, (*DF2*).

(Note that SRANDF is the single precision version of DRANDF. The argument lists of both routines are identical except that any double precision arguments of DRANDF are replaced in SRANDF by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDF (`*`N`*`,`*`DF1`*`,`*`DF2`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]

On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER DF1` [Input]

On input: first degrees of freedom.
Constraint: $DF1 \geq 0$.

`INTEGER DF2` [Input]

On input: second degrees of freedom.
Constraint: $DF2 \geq 0$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDF` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]

On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the F distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER DF1,DF2
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) DF1,DF2

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the F distribution
       CALL DRANDF(N,DF1,DF2,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDGAMMA / SRANDGAMMA

Generates a vector of random variates from a Gamma distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{X^{A-1}e^{-\frac{X}{B}}}{B^A\Gamma(A)},$$

if $X \geq 0$ and $A, B > 0.0$, otherwise $f(X) = 0$.

(Note that SRANDGAMMA is the single precision version of DRANDGAMMA. The argument lists of both routines are identical except that any double precision arguments of DRANDGAMMA are replaced in SRANDGAMMA by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDGAMMA (`*`N`*`,`*`A`*`,`*`B`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION A` [Input]
On input: first parameter of the distribution.
Constraint: $A > 0$.

`DOUBLE PRECISION B` [Input]
On input: second parameter of the distribution.
Constraint: $B > 0$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDGAMMA` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Gamma distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A,B
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Gamma distribution
       CALL DRANDGAMMA(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDGAUSSIAN / DRANDGAUSSIAN

Generates a vector of random variates from a Gaussian distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{e^{-\frac{(X-\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}.$$

Here μ is the mean, (XMU) and σ^2 the variance, (VAR) of the distribution.

(Note that SRANDGAUSSIAN is the single precision version of DRANDGAUSSIAN. The argument lists of both routines are identical except that any double precision arguments of DRANDGAUSSIAN are replaced in SRANDGAUSSIAN by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDGAUSSIAN (`*`N`*`,`*`XMU`*`,`*`VAR`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION XMU` [Input]
On input: mean of the distribution.

`DOUBLE PRECISION VAR` [Input]
On input: variance of the distribution.
Constraint: $VAR \geq 0$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDGAUSSIAN` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Gaussian distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION XMU,VAR
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) XMU,VAR

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Gaussian distribution
       CALL DRANDGAUSSIAN(N,XMU,VAR,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDLOGISTIC / SRANDLOGISTIC

Generates a vector of random variates from a logistic distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{e^{\frac{(X-A)}{B}}}{B(1+e^{\frac{(X-A)}{B}})^2}.$$

(Note that SRANDLOGISTIC is the single precision version of DRANDLOGISTIC. The argument lists of both routines are identical except that any double precision arguments of DRANDLOGISTIC are replaced in SRANDLOGISTIC by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDLOGISTIC (`*`N`*`,`*`A`*`,`*`B`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION A` [Input]
On input: mean of the distribution.

`DOUBLE PRECISION B` [Input]
On input: spread of the distribution. $B = \sqrt{3}\sigma/\pi$ where σ is the standard deviation of the distribution.
Constraint: $B > 0$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDLOGISTIC` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Logistic distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A,B
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Logistic distribution
       CALL DRANDLOGISTIC(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDLOGNORMAL / SRANDLOGNORMAL

Generates a vector of random variates from a lognormal distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{e^{-\frac{(\log X - \mu)^2}{2\sigma^2}}}{X\sigma\sqrt{2\pi}},$$

if $X > 0$, otherwise $f(X) = 0$. Here μ is the mean, (XMU) and σ^2 the variance, (VAR) of the underlying Gaussian distribution.

(Note that SRANDLOGNORMAL is the single precision version of DRANDLOGNORMAL. The argument lists of both routines are identical except that any double precision arguments of DRANDLOGNORMAL are replaced in SRANDLOGNORMAL by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDLOGNORMAL (*N*,*XMU*,*VAR*,*STATE*,*X*,*INFO*) [SUBROUTINE]

INTEGER N [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

DOUBLE PRECISION XMU [Input]
On input: mean of the underlying Gaussian distribution.

DOUBLE PRECISION VAR [Input]
On input: variance of the underlying Gaussian distribution.
Constraint: $VAR \geq 0$.

INTEGER STATE(*) [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling DRANDLOGNORMAL *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

DOUBLE PRECISION X(*N*) [Output]
On output: vector of variates from the specified distribution.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Lognormal distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION XMU,VAR
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) XMU,VAR

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Lognormal distribution
       CALL DRANDLOGNORMAL(N,XMU,VAR,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDSTUDENTST / SRANDSTUDENTST

Generates a vector of random variates from a Students T distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{\frac{(\nu-1)}{2}!}{(\frac{\nu}{2})!\sqrt{\pi\nu}(1+\frac{X^2}{\nu})^{\frac{(\nu+1)}{2}}}.$$

Here ν is the degrees of freedom, *DF*.

(Note that SRANDSTUDENTST is the single precision version of DRANDSTUDENTST. The argument lists of both routines are identical except that any double precision arguments of DRANDSTUDENTST are replaced in SRANDSTUDENTST by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDSTUDENTST (`*`N`*`,`*`DF`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]

On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER DF` [Input]

On input: degrees of freedom.
Constraint: $DF > 0$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDSTUDENTST` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]

On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If $INFO = -$i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Students T distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER DF
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) DF

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Students T distribution
       CALL DRANDSTUDENTST(N,DF,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDTRIANGULAR / SRANDTRIANGULAR

Generates a vector of random variates from a Triangular distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{2(X - X_{\mathrm{MIN}})}{(X_{\mathrm{MAX}} - X_{\mathrm{MIN}})(X_{\mathrm{MED}} - X_{\mathrm{MIN}})},$$

if $X_{\mathrm{MIN}} < X \leq X_{\mathrm{MED}}$, else

$$f(X) = \frac{2(X_{\mathrm{MAX}} - X)}{(X_{\mathrm{MAX}} - X_{\mathrm{MIN}})(X_{\mathrm{MAX}} - X_{\mathrm{MED}})},$$

if $X_{\mathrm{MED}} < X \leq X_{\mathrm{MAX}}$, otherwise $f(X) = 0$.

(Note that SRANDTRIANGULAR is the single precision version of DRANDTRIANGULAR. The argument lists of both routines are identical except that any double precision arguments of DRANDTRIANGULAR are replaced in SRANDTRIANGULAR by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDTRIANGULAR (`*`N`*`,`*`XMIN`*`,`*`XMED`*`,`*`XMAX`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION XMIN` [Input]
On input: minimum value for the distribution.

`DOUBLE PRECISION XMED` [Input]
On input: median value for the distribution.
Constraint: $XMIN \leq XMED \leq XMAX$.

`DOUBLE PRECISION XMAX` [Input]
On input: maximum value for the distribution.
Constraint: $XMAX \geq XMIN$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDTRIANGULAR` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Triangular distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION XMIN,XMED,XMAX
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) XMIN,XMED,XMAX

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Triangular distribution
       CALL DRANDTRIANGULAR(N,XMIN,XMED,XMAX,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDUNIFORM / SRANDUNIFORM

Generates a vector of random variates from a Uniform distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{1}{B - A}.$$

(Note that SRANDUNIFORM is the single precision version of DRANDUNIFORM. The argument lists of both routines are identical except that any double precision arguments of DRANDUNIFORM are replaced in SRANDUNIFORM by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDUNIFORM (`*`N`*`,`*`A`*`,`*`B`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]

On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION A` [Input]

On input: minimum value for the distribution.

`DOUBLE PRECISION B` [Input]

On input: maximum value for the distribution.
Constraint: $B \geq A$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDUNIFORM` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]

On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Uniform distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A,B
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Uniform distribution
       CALL DRANDUNIFORM(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDVONMISES / SRANDVONMISES

Generates a vector of random variates from a Von Mises distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{e^{\kappa \cos X}}{2\pi I_0(\kappa)}$$

where X is reduced modulo 2π so that it lies between $\pm\pi$, and κ is the concentration parameter *VK*.

(Note that SRANDVONMISES is the single precision version of DRANDVONMISES. The argument lists of both routines are identical except that any double precision arguments of DRANDVONMISES are replaced in SRANDVONMISES by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDVONMISES (`*`N,VK,,STATE,X,INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]

On input: number of variates required.
Constraint: $N \geq 0$.

`DOUBLE PRECISION VK` [Input]

On input: concentration parameter.
Constraint: $VK > 0$.

`INTEGER STATE(*)` [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDVONMISES` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(`*`N`*`)` [Output]

On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If $INFO = -$i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Von Mises distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION VK
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) VK

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Von Mises distribution
       CALL DRANDVONMISES(N,VK,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDWEIBULL / SRANDWEIBULL

Generates a vector of random variates from a Weibull distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{AX^{A-1}e^{-\frac{X^A}{B}}}{B},$$

if $X > 0$, otherwise $f(X) = 0$.

(Note that SRANDWEIBULL is the single precision version of DRANDWEIBULL. The argument lists of both routines are identical except that any double precision arguments of DRANDWEIBULL are replaced in SRANDWEIBULL by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDWEIBULL (*N*,*A*,*B*,*STATE*,*X*,*INFO*) [SUBROUTINE]

INTEGER N [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

DOUBLE PRECISION A [Input]
On input: shape parameter for the distribution.
Constraint: $A > 0$.

DOUBLE PRECISION B [Input]
On input: scale parameter for the distribution.
Constraint: $B > 0$.

INTEGER STATE(*) [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling DRANDWEIBULL *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

DOUBLE PRECISION X(*N*) [Output]
On output: vector of variates from the specified distribution.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Weibull distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION A,B
       DOUBLE PRECISION X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Weibull distribution
       CALL DRANDWEIBULL(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

6.3.2 Discrete Univariate Distributions

`DRANDBINOMIAL / SRANDBINOMIAL`

Generates a vector of random variates from a Binomial distribution with probability, $f(X)$, defined by:

$$f(X) = \frac{M!P^X(1-P)^{(M-X)}}{X!(M-1)!}, X = 0, 1, \cdots, M$$

(Note that SRANDBINOMIAL is the single precision version of DRANDBINOMIAL. The argument lists of both routines are identical except that any double precision arguments of DRANDBINOMIAL are replaced in SRANDBINOMIAL by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDBINOMIAL (`*`N`*`,`*`M`*`,`*`P`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER M` [Input]
On input: number of trials.
Constraint: $M \geq 0$.

`DOUBLE PRECISION P` [Input]
On input: probability of success.
Constraint: $0 \leq P < 1$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDBINOMIAL` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`INTEGER X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Binomial distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER M
       DOUBLE PRECISION P
       INTEGER X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) M,P

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Binomial distribution
       CALL DRANDBINOMIAL(N,M,P,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDGEOMETRIC / SRANDGEOMETRIC

Generates a vector of random variates from a Geometric distribution with probability, $f(X)$, defined by:

$$f(X) = P(1-P)^X, X = 0, 1, \cdots$$

(Note that SRANDGEOMETRIC is the single precision version of DRANDGEOMETRIC. The argument lists of both routines are identical except that any double precision arguments of DRANDGEOMETRIC are replaced in SRANDGEOMETRIC by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDGEOMETRIC (*N*,*P*,*STATE*,*X*,*INFO*) [SUBROUTINE]

INTEGER N [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

DOUBLE PRECISION P [Input]
On input: distribution parameter.
Constraint: $0 \leq P < 1$.

INTEGER STATE(*) [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling DRANDGEOMETRIC *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

INTEGER X(*N*) [Output]
On output: vector of variates from the specified distribution.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Geometric distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION P
       INTEGER X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) P

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Geometric distribution
       CALL DRANDGEOMETRIC(N,P,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDHYPERGEOMETRIC / SRANDHYPERGEOMETRIC

Generates a vector of random variates from a Hypergeometric distribution with probability, $f(X)$, defined by:

$$f(X) = \frac{s!m!(p-s)!(p-m)!}{X!(s-X)!(m-X)!(p-m-s+X)!p!},$$

if $X = \max(0, m+s-p), \cdots, \min(l, m)$, otherwise $f(X) = 0$. Here p is the size of the population, (NP), s is the size of the sample taken from the population, (NS) and m is the number of labeled, or specified, items in the population, (M).

(Note that SRANDHYPERGEOMETRIC is the single precision version of DRANDHYPERGEOMETRIC. The argument lists of both routines are identical except that any double precision arguments of DRANDHYPERGEOMETRIC are replaced in SRANDHYPERGEOMETRIC by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDHYPERGEOMETRIC (`*`N,NP,NS,M,STATE,X,INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER NP` [Input]
On input: size of population.
Constraint: $NP \geq 0$.

`INTEGER NS` [Input]
On input: size of sample being taken from population.
Constraint: $0 \leq NS \leq NP$.

`INTEGER M` [Input]
On input: number of specified items in the population.
Constraint: $0 \leq M \leq NP$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDHYPERGEOMETRIC` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`INTEGER X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Hypergeometric distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER NP,NS,M
       INTEGER X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) NP,NS,M

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Hypergeometric distribution
       CALL DRANDHYPERGEOMETRIC(N,NP,NS,M,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDNEGATIVEBINOMIAL / SRANDNEGATIVEBINOMIAL

Generates a vector of random variates from a Negative Binomial distribution with probability $f(X)$ defined by:

$$f(X) = \frac{(M+X-1)!P^X(1-P)^M}{X!(M-1)!}, X = 0, 1, \cdots$$

(Note that SRANDNEGATIVEBINOMIAL is the single precision version of DRANDNEGATIVEBINOMIAL. The argument lists of both routines are identical except that any double precision arguments of DRANDNEGATIVEBINOMIAL are replaced in SRANDNEGATIVEBINOMIAL by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDNEGATIVEBINOMIAL (`*`N`*`,`*`M`*`,`*`P`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER M` [Input]
On input: number of failures.
Constraint: $M \geq 0$.

`DOUBLE PRECISION P` [Input]
On input: probability of success.
Constraint: $0 \leq P < 1$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDNEGATIVEBINOMIAL` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`INTEGER X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Negative Binomial distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER M
       DOUBLE PRECISION P
       INTEGER X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) M,P

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Negative Binomial distribution
       CALL DRANDNEGATIVEBINOMIAL(N,M,P,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDPOISSON / SRANDPOISSON

Generates a vector of random variates from a Poisson distribution with probability $f(X)$ defined by:

$$f(X) = \frac{\lambda^X e^{-\lambda}}{X!}, X = 0, 1, \cdots,$$

where λ is the mean of the distribution, *LAMBDA*.

(Note that SRANDPOISSON is the single precision version of DRANDPOISSON. The argument lists of both routines are identical except that any double precision arguments of DRANDPOISSON are replaced in SRANDPOISSON by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDPOISSON (`*`N`*`,`*`LAMBDA`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER M` [Input]
On input: number of failures.
Constraint: $M \geq 0$.

`DOUBLE PRECISION LAMBDA` [Input]
On input: mean of the distribution.
Constraint: $LAMBDA \geq 0$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDPOISSON` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`INTEGER X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If $INFO = -$i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Poisson distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION LAMBDA
       INTEGER X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) LAMBDA

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Poisson distribution
       CALL DRANDPOISSON(N,LAMBDA,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDDISCRETEUNIFORM / SRANDDISCRETEUNIFORM

Generates a vector of random variates from a Uniform distribution with probability $f(X)$ defined by:

$$f(X) = \frac{1}{(B-A)}, X = A, A+1, \cdots, B$$

(Note that SRANDDISCRETEUNIFORM is the single precision version of DRANDDISCRETEUNIFORM. The argument lists of both routines are identical except that any double precision arguments of DRANDDISCRETEUNIFORM are replaced in SRANDDISCRETEUNIFORM by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDDISCRETEUNIFORM (`*`N`*`,`*`A`*`,`*`B`*`,`*`STATE`*`,`*`X`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER A` [Input]
On input: minimum for the distribution.

`INTEGER B` [Input]
On input: maximum for the distribution.
Constraint: $B \geq A$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDDISCRETEUNIFORM` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`INTEGER X(`*`N`*`)` [Output]
On output: vector of variates from the specified distribution.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Uniform distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER A,B
       INTEGER X(N)
C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) A,B

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C      Generate N variates from the Uniform distribution
       CALL DRANDDISCRETEUNIFORM(N,A,B,STATE,X,INFO)

C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDGENERALDISCRETE / SRANDGENERALDISCRETE

Takes a reference vector initialized via one of DRANDBINOMIALREFERENCE, DRANDGEOMETRICREFERENCE, DRANDHYPERGEOMETRICREFERENCE, DRANDNEGATIVEBINOMIALREFERENCE, DRANDPOISSONREFERENCE and generates a vector of random variates from it.

(Note that SRANDGENERALDISCRETE is the single precision version of DRANDGENERALDISCRETE. The argument lists of both routines are identical except that any double precision arguments of DRANDGENERALDISCRETE are replaced in SRANDGENERALDISCRETE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDGENERALDISCRETE (*N*,*REF*,*STATE*,*X*,*INFO*) [SUBROUTINE]

INTEGER N [Input]

On input: number of variates required.
Constraint: $N \geq 0$.

DOUBLE PRECISION REF(*) [Input]

On input: reference vector generated by one of the following: DRANDBINOMIALREFERENCE, DRANDGEOMETRICREFERENCE, DRANDHYPERGEOMETRICREFERENCE, DRANDNEGATIVEBINOMIALREFERENCE, DRANDPOISSONREFERENCE.

INTEGER STATE(*) [Input/Output]

The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling DRANDGENERALDISCRETE *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

INTEGER X(*N*) [Output]

On output: vector of variates from the specified distribution.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the Binomial distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER M
       DOUBLE PRECISION P
       INTEGER X(N)
       INTEGER LREF
       DOUBLE PRECISION REF(1000)

C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) M,P

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C      Initialize the reference vector
       LREF = 1000
       CALL DRANDBINOMIALREFERENCE(M,P,REF,LREF,INFO)

C      Generate N variates from the Binomial distribution
       CALL DRANDGENERALDISCRETE(N,REF,STATE,X,INFO)
C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDBINOMIALREFERENCE / SRANDBINOMIALREFERENCE

Initializes a reference vector for use with DRANDGENERALDISCRETE. Reference vector is for a Binomial distribution with probability, $f(X)$, defined by:

$$f(X) = \frac{M!P^X(1-P)^{(M-X)}}{X!(M-1)!}, X = 0, 1, \cdots, M$$

(Note that SRANDBINOMIALREFERENCE is the single precision version of DRANDBINOMIALREFERENCE. The argument lists of both routines are identical except that any double precision arguments of DRANDBINOMIALREFERENCE are replaced in SRANDBINOMIALREFERENCE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDBINOMIALREFERENCE (*M*,*P*,*REF*,*LREF*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: number of trials.
Constraint: $M \geq 0$.

DOUBLE PRECISION P [Input]
On input: probability of success.
Constraint: $0 \leq P < 1$.

DOUBLE PRECISION REF(LREF) [Output]
On output: if *INFO* returns with a value of 0 then *REF* contains reference information required to generate values from a Binomial distribution using DRANDGENERALDISCRETE.

INTEGER LREF [Input/Output]
On input: either the length of the reference vector *REF*, or -1.
On output: if $LREF = -1$ on input, then *LREF* is set to the recommended length of the reference vector and the routine returns. Otherwise *LREF* is left unchanged.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. If $INFO = -$i on exit, the i-th argument had an illegal value. If $INFO = 1$ on exit, then *LREF* has been set to the recommended length for the reference vector *REF*. If $INFO = 0$ then the reference vector, *REF*, has been successfully initialized.

Example:

```
C       Generate 100 values from the Binomial distribution
        INTEGER LSTATE,N
        PARAMETER (LSTATE=16,N=100)
        INTEGER I,INFO,SEED(1),STATE(LSTATE)
        INTEGER M
        DOUBLE PRECISION P
        INTEGER X(N)
        INTEGER LREF
        DOUBLE PRECISION REF(1000)

C       Set the seed
        SEED(1) = 1234

C       Read in the distributional parameters
        READ(5,*) M,P

C       Initialize the STATE vector
        CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C       Initialize the reference vector
        LREF = 1000
        CALL DRANDBINOMIALREFERENCE(M,P,REF,LREF,INFO)

C       Generate N variates from the Binomial distribution
        CALL DRANDGENERALDISCRETE(N,REF,STATE,X,INFO)
C       Print the results
        WRITE(6,*) (X(I),I=1,N)
```

DRANDGEOMETRICREFERENCE / SRANDGEOMETRICREFERENCE

Initializes a reference vector for use with DRANDGENERALDISCRETE. Reference vector is for a Geometric distribution with probability, $f(X)$, defined by:

$$f(X) = P(1-P)^X, X = 0, 1, \cdots$$

(Note that SRANDGEOMETRICREFERENCE is the single precision version of DRANDGEOMETRICREFERENCE. The argument lists of both routines are identical except that any double precision arguments of DRANDGEOMETRICREFERENCE are replaced in SRANDGEOMETRICREFERENCE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDGEOMETRICREFERENCE (*P*,*REF*,*LREF*,*INFO*) [SUBROUTINE]

DOUBLE PRECISION P [Input]

On input: distribution parameter.
Constraint: $0 \leq P < 1$.

DOUBLE PRECISION REF(LREF) [Output]

On output: if *INFO* returns with a value of 0 then *REF* contains reference information required to generate values from a Geometric distribution using DRANDGENERALDISCRETE.

INTEGER LREF [Input/Output]

On input: either the length of the reference vector *REF*, or -1.
On output: if $LREF = -1$ on input, then *LREF* is set to the recommended length of the reference vector and the routine returns. Otherwise *LREF* is left unchanged.

INTEGER INFO [Output]

On output: *INFO* is an error indicator. If $INFO = -$i on exit, the i-th argument had an illegal value. If $INFO = 1$ on exit, then *LREF* has been set to the recommended length for the reference vector *REF*. If $INFO = 0$ then the reference vector, *REF*, has been successfully initialized.

Example:

```
C      Generate 100 values from the Geometric distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       DOUBLE PRECISION P
       INTEGER X(N)
       INTEGER LREF
       DOUBLE PRECISION REF(1000)

C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) P

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C      Initialize the reference vector
       LREF = 1000
       CALL DRANDGEOMETRICREFERENCE(P,REF,LREF,INFO)

C      Generate N variates from the Geometric distribution
       CALL DRANDGENERALDISCRETE(N,REF,STATE,X,INFO)
C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDHYPERGEOMETRICREFERENCE / SRANDHYPERGEOMETRICREFERENCE

Initializes a reference vector for use with DRANDGENERALDISCRETE. Reference vector is for a Hypergeometric distribution with probability, $f(X)$, defined by:

$$f(X) = \frac{s!m!(p-s)!(p-m)!}{X!(s-X)!(m-X)!(p-m-s+X)!p!},$$

if $X = \max(0, m+s-p), \cdots, \min(l, m)$, otherwise $f(X) = 0$. Here p is the size of the population, (NP), s is the size of the sample taken from the population, (NS) and m is the number of labeled, or specified, items in the population, (M).

(Note that SRANDHYPERGEOMETRICREFERENCE is the single precision version of DRANDHYPERGEOMETRICREFERENCE. The argument lists of both routines are identical except that any double precision arguments of DRANDHYPERGEOMETRICREFERENCE are replaced in SRANDHYPERGEOMETRICREFERENCE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDHYPERGEOMETRICREFERENCE (*NP*,*NS*,*M*,*REF*,*LREF*,*INFO*) [SUBROUTINE]

INTEGER NP [Input]
On input: size of population.
Constraint: $NP \geq 0$.

INTEGER NS [Input]
On input: size of sample being taken from population.
Constraint: $0 \leq NS \leq NP$.

INTEGER M [Input]
On input: number of specified items in the population.
Constraint: $0 \leq M \leq NP$.

DOUBLE PRECISION REF(LREF) [Output]
On output: if *INFO* returns with a value of 0 then *REF* contains reference information required to generate values from a Hypergeometric distribution using DRANDGENERALDISCRETE.

INTEGER LREF [Input/Output]
On input: either the length of the reference vector *REF*, or -1.
On output: if $LREF = -1$ on input, then *LREF* is set to the recommended length of the reference vector and the routine returns. Otherwise *LREF* is left unchanged.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. If $INFO = -$i on exit, the i-th argument had an illegal value. If $INFO = 1$ on exit, then *LREF* has been set to the recommended length for the reference vector *REF*. If $INFO = 0$ then the reference vector, *REF*, has been successfully initialized.

Example:

```
C      Generate 100 values from the Hypergeometric distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER NP, NS,M
       INTEGER X(N)
       INTEGER LREF
       DOUBLE PRECISION REF(1000)

C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) NP, NS,M

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C      Initialize the reference vector
       LREF = 1000
       CALL DRANDHYPERGEOMETRICREFERENCE(NP, NS,M,REF,LREF,INFO)

C      Generate N variates from the Hypergeometric distribution
       CALL DRANDGENERALDISCRETE(N,REF,STATE,X,INFO)
C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDNEGATIVEBINOMIALREFERENCE / SRANDNEGATIVEBINOMIALREFERENCE

Initializes a reference vector for use with DRANDGENERALDISCRETE. Reference vector is for a Negative Binomial distribution with probability $f(X)$ defined by:

$$f(X) = \frac{(M+X-1)!P^X(1-P)^M}{X!(M-1)!}, X = 0, 1, \cdots$$

(Note that SRANDNEGATIVEBINOMIALREFERENCE is the single precision version of DRANDNEGATIVEBINOMIALREFERENCE. The argument lists of both routines are identical except that any double precision arguments of DRANDNEGATIVEBINOMIALREFERENCE are replaced in SRANDNEGATIVEBINOMIALREFERENCE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDNEGATIVEBINOMIALREFERENCE (*M*,*P*,*REF*,*LREF*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: number of failures.
Constraint: $M \geq 0$.

DOUBLE PRECISION P [Input]
On input: probability of success.
Constraint: $0 \leq P < 1$.

DOUBLE PRECISION REF(LREF) [Output]
On output: if *INFO* returns with a value of 0 then *REF* contains reference information required to generate values from a Negative Binomial distribution using DRANDGENERALDISCRETE.

INTEGER LREF [Input/Output]
On input: either the length of the reference vector *REF*, or -1.
On output: if $LREF = -1$ on input, then *LREF* is set to the recommended length of the reference vector and the routine returns. Otherwise *LREF* is left unchanged.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. If $INFO = -$i on exit, the i-th argument had an illegal value. If $INFO = 1$ on exit, then *LREF* has been set to the recommended length for the reference vector *REF*. If $INFO = 0$ then the reference vector, *REF*, has been successfully initialized.

Example:

```
C      Generate 100 values from the Negative Binomial distribution
       INTEGER LSTATE,N
       PARAMETER (LSTATE=16,N=100)
       INTEGER I,INFO,SEED(1),STATE(LSTATE)
       INTEGER M
       DOUBLE PRECISION P
       INTEGER X(N)
       INTEGER LREF
       DOUBLE PRECISION REF(1000)

C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) M,P

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C      Initialize the reference vector
       LREF = 1000
       CALL DRANDNEGATIVEBINOMIALREFERENCE(M,P,REF,LREF,INFO)

C      Generate N variates from the Negative Binomial distribution
       CALL DRANDGENERALDISCRETE(N,REF,STATE,X,INFO)
C      Print the results
       WRITE(6,*) (X(I),I=1,N)
```

DRANDPOISSONREFERENCE / SRANDPOISSONREFERENCE

Initializes a reference vector for use with DRANDGENERALDISCRETE. Reference vector is for a Poisson distribution with probability $f(X)$ defined by:

$$f(X) = \frac{\lambda^X e^{-\lambda}}{X!}, X = 0, 1, \cdots,$$

where λ is the mean of the distribution, *LAMBDA*.

(Note that SRANDPOISSONREFERENCE is the single precision version of DRANDPOISSONREFERENCE. The argument lists of both routines are identical except that any double precision arguments of DRANDPOISSONREFERENCE are replaced in SRANDPOISSONREFERENCE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDPOISSONREFERENCE (*LAMBDA*,*REF*,*LREF*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: number of failures.
Constraint: $M \geq 0$.

DOUBLE PRECISION LAMBDA [Input]
On input: mean of the distribution.
Constraint: $LAMBDA \geq 0$.

DOUBLE PRECISION REF(LREF) [Output]
On output: if *INFO* returns with a value of 0 then *REF* contains reference information required to generate values from a Poisson distribution using DRANDGENERALDISCRETE.

INTEGER LREF [Input/Output]
On input: either the length of the reference vector *REF*, or -1.
On output: if $LREF = -1$ on input, then *LREF* is set to the recommended length of the reference vector and the routine returns. Otherwise *LREF* is left unchanged.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. If $INFO = -$i on exit, the i-th argument had an illegal value. If $INFO = 1$ on exit, then *LREF* has been set to the recommended length for the reference vector *REF*. If $INFO = 0$ then the reference vector, *REF*, has been successfully initialized.

Example:

```
C       Generate 100 values from the Poisson distribution
        INTEGER LSTATE,N
        PARAMETER (LSTATE=16,N=100)
        INTEGER I,INFO,SEED(1),STATE(LSTATE)
        DOUBLE PRECISION LAMBDA
        INTEGER X(N)
        INTEGER LREF
        DOUBLE PRECISION REF(1000)

C       Set the seed
        SEED(1) = 1234

C       Read in the distributional parameters
        READ(5,*) LAMBDA

C       Initialize the STATE vector
        CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C       Initialize the reference vector
        LREF = 1000
        CALL DRANDPOISSONREFERENCE(LAMBDA,REF,LREF,INFO)

C       Generate N variates from the Poisson distribution
        CALL DRANDGENERALDISCRETE(N,REF,STATE,X,INFO)
C       Print the results
        WRITE(6,*) (X(I),I=1,N)
```

6.3.3 Continuous Multivariate Distributions

`DRANDMULTINORMAL / SRANDMULTINORMAL`

Generates an array of random variates from a Multivariate Normal distribution with probability density function, $f(X)$, where:

$$f(X) = \sqrt{\frac{|C^{-1}|}{(2\pi)^M}} e^{-(X-\mu)^T C^{-1}(X-\mu)},$$

where μ is the vector of means, *XMU*.

(Note that SRANDMULTINORMAL is the single precision version of DRANDMULTINORMAL. The argument lists of both routines are identical except that any double precision arguments of DRANDMULTINORMAL are replaced in SRANDMULTINORMAL by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDMULTINORMAL (`*`N`*`,`*`M`*`,`*`XMU`*`,`*`C`*`,`*`LDC`*`,`*`STATE`*`,`*`X`*`,`*`LDX`*`,`*`INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER M` [Input]
On input: number of dimensions for the distribution.
Constraint: $M \geq 1$.

`DOUBLE PRECISION XMU(M)` [Input]
On input: vector of means for the distribution.

`DOUBLE PRECISION C(LDC,M)` [Input]
On input: variance / covariance matrix for the distribution.

`INTEGER LDC` [Input]
On input: leading dimension of C in the calling routine.
Constraint: $LDC \geq M$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDMULTINORMAL` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`DOUBLE PRECISION X(LDX,M)` [Output]
On output: matrix of variates from the specified distribution.

`INTEGER LDX` [Input]
On input: leading dimension of X in the calling routine.
Constraint: $LDX \geq N$.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C       Generate 100 values from the
C       Multivariate Normal distribution
        INTEGER LSTATE,N, MM
        PARAMETER (LSTATE=16,N=100,MM=10)
        INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
        INTEGER LDC,LDX,M
        DOUBLE PRECISION X(N,MM),XMU(MM),C(MM,MM)
C       Set array sizes
        LDC = MM
        LDX = N

C       Set the seed
        SEED(1) = 1234

C       Read in the distributional parameters
        READ(5,*) M
        READ(5,*) (XMU(I),I=1,M)
        DO 20 I = 1,M
          READ(5,*) (C(I,J),J=1,M)
20      CONTINUE

C       Initialize the STATE vector
        CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
C       Generate N variates from the
C       Multivariate Normal distribution
        CALL DRANDMULTINORMAL(N,M,XMU,C,LDC,STATE,X,LDX,INFO)

C       Print the results
        DO 40 I = 1,N
          WRITE(6,*) (X(I,J),J=1,M)
40      CONTINUE
```

DRANDMULTISTUDENTST / SRANDMULTISTUDENTST

Generates an array of random variates from a Multivariate Students T distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{\Gamma\left(\frac{(\nu+M)}{2}\right)}{(\pi\nu)^{\frac{m}{2}}\Gamma(\frac{\nu}{2})\,|C|^{\frac{1}{2}}}\left(1+\frac{(X-\mu)^T C^{-1}(X-\mu)}{\nu}\right)^{-\frac{(\nu+M)}{2}},$$

where μ is the vector of means, *XMU* and ν is the degrees of freedom, *DF*.

(Note that SRANDMULTISTUDENTST is the single precision version of DRANDMULTISTUDENTST. The argument lists of both routines are identical except that any double precision arguments of DRANDMULTISTUDENTST are replaced in SRANDMULTISTUDENTST by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDMULTISTUDENTST (*N*,*M*,*DF*,*XMU*,*C*,*LDC*,*STATE*,*X*,*LDX*,*INFO*) [SUBROUTINE]

INTEGER N [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

INTEGER M [Input]
On input: number of dimensions for the distribution.
Constraint: $M \geq 1$.

INTEGER DF [Input]
On input: degrees of freedom.
Constraint: $DF > 2$.

DOUBLE PRECISION XMU(M) [Input]
On input: vector of means for the distribution.

DOUBLE PRECISION C(LDC,M) [Input]
On input: matrix defining the variance / covariance for the distribution. The variance / covariance matrix is given by $\frac{\nu}{\nu-2}C$, where ν are the degrees of freedom, *DF*.

INTEGER LDC [Input]
On input: leading dimension of C in the calling routine.
Constraint: $LDC \geq M$.

INTEGER STATE(*) [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling DRANDMULTISTUDENTST *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

DOUBLE PRECISION X(LDX,M) [Output]
On output: matrix of variates from the specified distribution.

`INTEGER LDX` [Input]

On input: leading dimension of X in the calling routine.
Constraint: $LDX \geq N$.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If $INFO = -$i on exit, the i-th argument had an illegal value.

Example:

```
      C      Generate 100 values from the
      C      Multivariate Students T distribution
             INTEGER LSTATE,N, MM
             PARAMETER (LSTATE=16,N=100,MM=10)
             INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
             INTEGER LDC,LDX,M,DF
             DOUBLE PRECISION X(N,MM),XMU(MM),C(MM,MM)
      C      Set array sizes
             LDC = MM
             LDX = N

      C      Set the seed
             SEED(1) = 1234

      C      Read in the distributional parameters
             READ(5,*) M,DF
             READ(5,*) (XMU(I),I=1,M)
             DO 20 I = 1,M
               READ(5,*) (C(I,J),J=1,M)
      20     CONTINUE

      C      Initialize the STATE vector
             CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)
      C      Generate N variates from the
      C      Multivariate Students T distribution
             CALL DRANDMULTISTUDENTST(N,M,DF,XMU,C,LDC,STATE,X,LDX,INFO)

      C      Print the results
             DO 40 I = 1,N
               WRITE(6,*) (X(I,J),J=1,M)
      40     CONTINUE
```

DRANDMULTINORMALR / SRANDMULTINORMALR

Generates an array of random variates from a Multivariate Normal distribution using a reference vector initialized by DRANDMULTINORMALREFERENCE.

(Note that SRANDMULTINORMALR is the single precision version of DRANDMULTINORMALR. The argument lists of both routines are identical except that any double precision arguments of DRANDMULTINORMALR are replaced in SRANDMULTINORMALR by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDMULTINORMALR (*N*,*REF*,*STATE*,*X*,*LDX*,*INFO*) [SUBROUTINE]

INTEGER N [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

DOUBLE PRECISION REF(*) [Input]
On input: a reference vector generated by DRANDMULTINORMALREFERENCE.

INTEGER STATE(*) [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling DRANDMULTINORMALR *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

DOUBLE PRECISION X(LDX,M) [Output]
On output: matrix of variates from the specified distribution.

INTEGER LDX [Input]
On input: leading dimension of X in the calling routine.
Constraint: $LDX \geq N$.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If $INFO = -$i on exit, the i-th argument had an illegal value.

Example:

```
C      Generate 100 values from the
C      Multivariate Normal distribution
       INTEGER LSTATE,N, MM
       PARAMETER (LSTATE=16,N=100,MM=10)
       INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
       INTEGER LDC,LDX,M
       DOUBLE PRECISION X(N,MM),XMU(MM),C(MM,MM)
       INTEGER LREF
       DOUBLE PRECISION REF(1000)

C      Set array sizes
       LDC = MM
       LDX = N

C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) M
       READ(5,*) (XMU(I),I=1,M)
       DO 20 I = 1,M
         READ(5,*) (C(I,J),J=1,M)
20     CONTINUE

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C      Initialize the reference vector
       LREF = 1000
       CALL DRANDMULTINORMALREFERENCE(M,XMU,C,LDC,REF,LREF,INFO)

C      Generate N variates from the
C      Multivariate Normal distribution
       CALL DRANDMULTINORMALR(N,REF,STATE,X,LDX,INFO)
C      Print the results
       DO 40 I = 1,N
         WRITE(6,*) (X(I,J),J=1,M)
40     CONTINUE
```

DRANDMULTISTUDENTSTR / SRANDMULTISTUDENTSTR

Generates an array of random variates from a Multivariate Students T distribution using a reference vector initialized by DRANDMULTISTUDENTSTREFERENCE.

(Note that SRANDMULTISTUDENTSTR is the single precision version of DRANDMULTISTUDENTSTR. The argument lists of both routines are identical except that any double precision arguments of DRANDMULTISTUDENTSTR are replaced in SRANDMULTISTUDENTSTR by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDMULTISTUDENTSTR (*N*,*REF*,*STATE*,*X*,*LDX*,*INFO*) [SUBROUTINE]

INTEGER N [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

DOUBLE PRECISION REF(*) [Input]
On input: a reference vector generated by DRANDMULTISTUDENTSTREFERENCE.

INTEGER STATE(*) [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling DRANDMULTISTUDENTSTR *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

DOUBLE PRECISION X(LDX,M) [Output]
On output: matrix of variates from the specified distribution.

INTEGER LDX [Input]
On input: leading dimension of X in the calling routine.
Constraint: $LDX \geq N$.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C       Generate 100 values from the
C       Multivariate Students T distribution
        INTEGER LSTATE,N, MM
        PARAMETER (LSTATE=16,N=100,MM=10)
        INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
        INTEGER LDC,LDX,M,DF
        DOUBLE PRECISION X(N,MM),XMU(MM),C(MM,MM)
        INTEGER LREF
        DOUBLE PRECISION REF(1000)

C       Set array sizes
        LDC = MM
        LDX = N

C       Set the seed
        SEED(1) = 1234

C       Read in the distributional parameters
        READ(5,*) M,DF
        READ(5,*) (XMU(I),I=1,M)
        DO 20 I = 1,M
          READ(5,*) (C(I,J),J=1,M)
20      CONTINUE

C       Initialize the STATE vector
        CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C       Initialize the reference vector
        LREF = 1000
        CALL DRANDMULTISTUDENTSTREFERENCE(M,DF,XMU,C,LDC,REF,LREF,INFO)

C       Generate N variates from the
C       Multivariate Students T distribution
        CALL DRANDMULTISTUDENTSTR(N,REF,STATE,X,LDX,INFO)
C       Print the results
        DO 40 I = 1,N
          WRITE(6,*) (X(I,J),J=1,M)
40      CONTINUE
```

DRANDMULTINORMALREFERENCE / SRANDMULTINORMALREFERENCE

Initializes a reference vector for use with DRANDMULTINORMALR. Reference vector is for a Multivariate Normal distribution with probability density function, $f(X)$, where:

$$f(X) = \sqrt{\frac{|C^{-1}|}{(2\pi)^M}} e^{-(X-\mu)^T C^{-1}(X-\mu)},$$

where μ is the vector of means, *XMU*.

(Note that SRANDMULTINORMALREFERENCE is the single precision version of DRANDMULTINORMALREFERENCE. The argument lists of both routines are identical except that any double precision arguments of DRANDMULTINORMALREFERENCE are replaced in SRANDMULTINORMALREFERENCE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDMULTINORMALREFERENCE (*M*,*XMU*,*C*,*LDC*,*REF*,*LREF*,*INFO*) [SUBROUTINE]

INTEGER M [Input]
On input: number of dimensions for the distribution.
Constraint: $M \geq 1$.

DOUBLE PRECISION XMU(M) [Input]
On input: vector of means for the distribution.

DOUBLE PRECISION C(LDC,M) [Input]
On input: variance / covariance matrix for the distribution.

INTEGER LDC [Input]
On input: leading dimension of C in the calling routine.
Constraint: $LDC \geq M$.

DOUBLE PRECISION REF(LREF) [Output]
On output: if *INFO* returns with a value of 0 then *REF* contains reference information required to generate values from a Multivariate Normal distribution using DRANDMULTINORMALR.

INTEGER LREF [Input/Output]
On input: either the length of the reference vector *REF*, or -1.
On output: if $LREF = -1$ on input, then *LREF* is set to the recommended length of the reference vector and the routine returns. Otherwise *LREF* is left unchanged.

INTEGER INFO [Output]
On output: *INFO* is an error indicator. If $INFO = -$i on exit, the i-th argument had an illegal value. If $INFO = 1$ on exit, then *LREF* has been set to the recommended length for the reference vector *REF*. If $INFO = 0$ then the reference vector, *REF*, has been successfully initialized.

Example:

```
C      Generate 100 values from the
C      Multivariate Normal distribution
       INTEGER LSTATE,N, MM
       PARAMETER (LSTATE=16,N=100,MM=10)
       INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
       INTEGER LDC,LDX,M
       DOUBLE PRECISION X(N,MM),XMU(MM),C(MM,MM)
       INTEGER LREF
       DOUBLE PRECISION REF(1000)

C      Set array sizes
       LDC = MM
       LDX = N

C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) M
       READ(5,*) (XMU(I),I=1,M)
       DO 20 I = 1,M
         READ(5,*) (C(I,J),J=1,M)
20     CONTINUE

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C      Initialize the reference vector
       LREF = 1000
       CALL DRANDMULTINORMALREFERENCE(M,XMU,C,LDC,REF,LREF,INFO)

C      Generate N variates from the
C      Multivariate Normal distribution
       CALL DRANDMULTINORMALR(N,REF,STATE,X,LDX,INFO)
C      Print the results
       DO 40 I = 1,N
         WRITE(6,*) (X(I,J),J=1,M)
40     CONTINUE
```

DRANDMULTISTUDENTSTREFERENCE / SRANDMULTISTUDENTSTREFERENCE

Initializes a reference vector for use with DRANDMULTISTUDENTSTR. Reference vector is for a Multivariate Students T distribution with probability density function, $f(X)$, where:

$$f(X) = \frac{\Gamma\left(\frac{(\nu+M)}{2}\right)}{(\pi\nu)^{\frac{m}{2}}\Gamma(\frac{\nu}{2})\,|C|^{\frac{1}{2}}}\left(1+\frac{(X-\mu)^T C^{-1}(X-\mu)}{\nu}\right)^{-\frac{(\nu+M)}{2}},$$

where μ is the vector of means, *XMU* and ν is the degrees of freedom, *DF*.

(Note that SRANDMULTISTUDENTSTREFERENCE is the single precision version of DRANDMULTISTUDENTSTREFERENCE. The argument lists of both routines are identical except that any double precision arguments of DRANDMULTISTUDENTSTREFERENCE are replaced in SRANDMULTISTUDENTSTREFERENCE by single precision arguments - type REAL in FORTRAN or type float in C).

DRANDMULTISTUDENTSREFERENCE [SUBROUTINE]
(*M*,*DF*,*XMU*,*C*,*LDC*,*REF*,*LREF*,*INFO*)

INTEGER M [Input]
On input: number of dimensions for the distribution.
Constraint: $M \geq 1$.

INTEGER DF [Input]
On input: degrees of freedom.
Constraint: $DF > 2$.

DOUBLE PRECISION XMU(M) [Input]
On input: vector of means for the distribution.

DOUBLE PRECISION C(LDC,M) [Input]
On input: matrix defining the variance / covariance for the distribution. The variance / covariance matrix is given by $\frac{\nu}{\nu-2}C$, where ν are the degrees of freedom, *DF*.

INTEGER LDC [Input]
On input: leading dimension of C in the calling routine.
Constraint: $LDC \geq M$.

DOUBLE PRECISION REF(LREF) [Output]
On output: if *INFO* returns with a value of 0 then *REF* contains reference information required to generate values from a Multivariate Students T distribution using DRANDMULTISTUDENTSTR.

INTEGER LREF [Input/Output]
On input: either the length of the reference vector *REF*, or -1.
On output: if $LREF = -1$ on input, then *LREF* is set to the recommended length of the reference vector and the routine returns. Otherwise *LREF* is left unchanged.

`INTEGER INFO` [Output]

On output: *INFO* is an error indicator. If *INFO* = −i on exit, the i-th argument had an illegal value. If *INFO* = 1 on exit, then *LREF* has been set to the recommended length for the reference vector *REF*. If *INFO* = 0 then the reference vector, *REF*, has been successfully initialized.

Example:

```
C      Generate 100 values from the
C      Multivariate Students T distribution
       INTEGER LSTATE,N, MM
       PARAMETER (LSTATE=16,N=100,MM=10)
       INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
       INTEGER LDC,LDX,M,DF
       DOUBLE PRECISION X(N,MM),XMU(MM),C(MM,MM)
       INTEGER LREF
       DOUBLE PRECISION REF(1000)

C      Set array sizes
       LDC = MM
       LDX = N

C      Set the seed
       SEED(1) = 1234

C      Read in the distributional parameters
       READ(5,*) M,DF
       READ(5,*) (XMU(I),I=1,M)
       DO 20 I = 1,M
         READ(5,*) (C(I,J),J=1,M)
20     CONTINUE

C      Initialize the STATE vector
       CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C      Initialize the reference vector
       LREF = 1000
       CALL DRANDMULTISTUDENTSTREFERENCE(M,DF,XMU,C,LDC,REF,LREF,INFO)

C      Generate N variates from the
C      Multivariate Students T distribution
       CALL DRANDMULTISTUDENTSTR(N,REF,STATE,X,LDX,INFO)
C      Print the results
       DO 40 I = 1,N
         WRITE(6,*) (X(I,J),J=1,M)
40     CONTINUE
```

6.3.4 Discrete Multivariate Distributions

`DRANDMULTINOMIAL / SRANDMULTINOMIAL`

Generates a matrix of random variates from a Multinomial distribution with probability, $f(X)$, defined by:

$$f(X) = \frac{M!}{\prod_{i=1}^{K} X_i!} \prod_{i=1}^{K} p_i^{X_i},$$

where $X = \{X_1, X_2, \cdots, X_K\}, P = \{P_1, P_2, \cdots, P_K\}, \sum_{i=1}^{K} X_i = 1$ and $\sum_{i=1}^{K} P_i = 1$.

(Note that SRANDMULTINOMIAL is the single precision version of DRANDMULTINOMIAL. The argument lists of both routines are identical except that any double precision arguments of DRANDMULTINOMIAL are replaced in SRANDMULTINOMIAL by single precision arguments - type REAL in FORTRAN or type float in C).

`DRANDMULTINOMIAL (`*`N,M,P,K,STATE,X,LDX,INFO`*`)` [SUBROUTINE]

`INTEGER N` [Input]
On input: number of variates required.
Constraint: $N \geq 0$.

`INTEGER M` [Input]
On input: number of trials.
Constraint: $M \geq 0$.

`DOUBLE PRECISION P(K)` [Input]
On input: vector of probabilities for each of the K possible outcomes.
Constraint: $0 \leq P_i \leq 1, i = 1, 2, \cdots, K, \sum_{i=1}^{K} P_i = 1$.

`INTEGER K` [Input]
On input: number of possible outcomes.
Constraint: $K \geq 2$.

`INTEGER STATE(*)` [Input/Output]
The *STATE* vector holds information on the state of the base generator being used and as such its minimum length varies. Prior to calling `DRANDBINOMIAL` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 101 for information on initialization of the *STATE* variable.
On input: the current state of the base generator.
On output: the updated state of the base generator.

`INTEGER X(LDX,K)` [Output]
On output: matrix of variates from the specified distribution.

`INTEGER LDX` [Input]
On input: leading dimension of X in the calling routine.
Constraint: $LDX \geq N$.

`INTEGER INFO` [Output]
On output: *INFO* is an error indicator. On successful exit, *INFO* contains 0. If *INFO* = −i on exit, the i-th argument had an illegal value.

Example:

```
C Generate 100 values from the Multinomial distribution
      INTEGER LSTATE,N,M
      PARAMETER (LSTATE=16,N=100,M=10)
      INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
      INTEGER LDX,K
      INTEGER  X(N,M)
      DOUBLE PRECISION P(M)

C Set array sizes
      LDX = N

C Set the seed
      SEED(1) = 1234

C Read in the distributional parameters
      READ(5,*) K
      READ(5,*) (P(I),I=1,K)

C Initialize the STATE vector
      CALL DRANDINITIALIZE(1,1,SEED,1,STATE,LSTATE,INFO)

C Generate N variates from the Multinomial distribution
      CALL DRANDMULTINOMIAL(N,M,P,K,STATE,X,LDX,INFO)

C Print the results
      DO 20 I = 1,N
        WRITE(6,*) (X(I,J),J=1,K)
      20 CONTINUE
```

7 ACMLScript: ACML scripting language

ACMLScript is a scripting language embedded within the ACML library, introduced with the version 6 release. It allows ACML to embed programming logic within text files, which avoids hard-coding logic within the library itself. This provides both flexibility and transparency for end users.

The challenge of ACML version 6 is to create a library that takes advantage of heterogeneous compute resources in a platform (such as OpenCL compute devices or shared memory), but to keep the user visible API backwards compatible with ACML version 5. ACML should not expose the implementation details of OpenCL to the user, and the goal is to provide a drop-in replacement library that leverages these resources transparently.

At this time, only the BLAS and FFTW components of ACML are heterogeneous (i.e. able to be offloaded to OpenCL devices), and the existing ACMLScript files only relate to those components. Within BLAS, only the Level 3 routines are enabled with script files and for FFTW, only the basic and advanced API's are enabled. More routines will come over time.

The primary purpose for ACMLScript at this time is to encode the load balancing logic in scriptable form. This allows the logic on how the library decides to offload computation onto OpenCL devices to be presented in an easy to read and open manner. This also allows the logic to be changed to fit the needs of the user.

ACMLScript files are implemented in the Lua scripting language (`http://www.lua.org/`), using the Lua syntax and structure. ACMLScript files can call back into the library to request a service to be performed; typically this a request for more information from the library for something that the script does not have direct access to. The entry points that the library provides for the ACML scripts are documented below.

7.1 Directory layout

ACML now loads resources from disk (aside from the binary code itself, such as the scripts) during its runtime. These resources should be located in known locations to the library, or the library assumes that the resources are not available, and reverts to default behavior which closely resembles ACML version 5.

The first place the library looks for its resources is in a subdirectory to the location of the library itself called '`resources`'. For example, if the libacml library is located in `/opt/acml5.3.1/gfortran64/lib`, then the library attempts to access `/opt/acml5.3.1/gfortran64/lib/resources`. This is the location that the installer of ACML will install the resources by default, as a subdirectory of the library location.

If an end user wishes for any reason to move the location of the on disk resources, ACML will check for the existence of an environment variable at startup, `ACML_SCRIPT_PATH`. This should point to the root of the resources tree, such as `/opt/acml5.3.1/gfortran64/lib/resources`. Underneath this root resources tree, subdirectories exist for each particular OpenCL device that ACML ships tuned script files for. For instance, directories like `ACML_SCRIPT_PATH/Tahiti` or `ACML_SCRIPT_PATH/Spectre` might show up here, and the scripts therein are loaded for when those devices are detected. The name for these subdirectories are queried from OpenCL itself, corresponding to the `CL_DEVICE_NAME` property. If ACML does not find a subdirectory for a particular device, it attempts to load fall-back scripts from the `ACML_SCRIPT_PATH/Defaults` subdirectory.

This subdirectory contains script files that direct ACML to revert to ACML 5 behavior, which is to compute only on the CPU cores.

7.2 ACMLScript files

7.2.1 context.lua

This script lives in the root `ACML_SCRIPT_PATH` directory. It is called when ACML first initializes its internal data structures, while the library is still loading. At this time, ACML is searching the platform for all of the OpenCL devices that are available, and then passes control to this script to allow the user to customize the device selection process. This file contains a subroutine called `createContexts( ... )` which is called by ACML and whose single parameter is a table of the detected OpenCL platforms on the system. In most cases, only a single platform will be seen, but there can be instances of multiple platforms when an end user has devices from multiple vendors installed. The table passed as a parameter is an array of key-value pairs; the key an unsigned integer starting from 1, and the value a nested table of the various OpenCL platform properties. The nested table itself contains keys of the following names, with strings as their values:

`platforms` *profile, version, name, vendor, extensions* [Table]
profile corresponds to CL_PLATFORM_PROFILE
version corresponds to CL_PLATFORM_VERSION
name corresponds to CL_PLATFORM_NAME
vendor corresponds to CL_PLATFORM_VENDOR
extentions corresponds to CL_PLATFORM_EXTENSIONS

Since the platforms parameter is passed into the script as a nested table, it is most convenient to browse the platform information with nested Lua pair iterators Example:

```
-- An example of printing platform properties to stdout
for index, platform in ipairs( platforms ) do
    print( index )
    for name, value in pairs( platform ) do
        print( "\t", name, "\t", value )
    end
end
```

⇒ The return value from `createContexts( ... )` must be a table, which specifies the index value of the chosen platform as the key, and the index value of the device within that platform as the value. It is always valid to hardcode a return value of (1, 1), which says to pick the first platform, and pick the 1st device within that platform. This is possible because the ACML runtime will not enumerate a platform with 0 possible devices within it. The returned pair of the platform and device index helps to generate the OpenCL context within ACML library.
Constraint: This routine is currently limited to return a single context only, and that context may only have exactly one device in it

7.2.2 heuristic scripts

Inside the device specific subdirectories (or the `Default` subdirectory), there exists ACMLScript files that are specific to various API's supported by ACML. For ACML version 6.0, those API's are limited to the Level 3 BLAS routines or the basic and advanced interfaces for FFTW. All of these files contain a function called `heuristic( ... )`. The parameters passed into the heuristic function correspond to the API of the referring function.

For instance, the heuristic function for the GEMM routine looks like:

```
-- The heuristic function analyses input parameters, and determines
-- where a give problem should be computer, on host or device.
-- The signature is similar in nature to the corresponding BLAS API.
-- Documentation for the individual parameters can be found online
-- in the netlib website.
-- type( transa ) == string; either 'n' or 't' or 'c'
-- type( transb ) == string; either 'n' or 't' or 'c'
-- type( m ) == number
-- type( n ) == number
-- type( k ) == number
-- type( alpha_real ) == number; real portion of a complex number
-- type( alpha_imag ) == number; imaginary portion of a complex number
-- type( lda ) == number
-- type( ldb ) == number
-- type( beta_real ) == number; real portion of a complex number
-- type( beta_imag ) == number; imaginary portion of a comlex number
-- type( ldc ) == number
-- type( precision ) == string; either 's or 'd' or 'c' or 'z'
-- return boolean expression as integer; true means to offload
-- problem to device, false to offload on host
function heuristic( transa, transb, m, n, k, alpha_real, alpha_imag,
                    lda, ldb, beta_real, beta_imag, ldc, precision )

    local myThreshold = tableOfThresholds[ precision ]

    -- If any dimension is too small, compute on host
    if( ( m > myThreshold.m ) and ( n > myThreshold.n ) and
         ( k > myThreshold.k ) ) then

        local psize = m * n * k

        -- If the total size is greater than threshold,
        -- compute on device
        if( psize > myThreshold.psize ) then
            return true
        end
    end

    return false
end
```

⇒ The heuristic functions is to return a boolean value to indicate where the computation for this particular problem should take place; a true value indicates the OpenCL context created in the previously run `createContexts( ... )` function, or false to process on host. Host processing does imply multi-threading if the user linked with the OpenMP version of ACML.

7.3 ACMLScript types

ACMLScripts have a few tables embedded at the top of the script, which are used more or less like an enumeration within the script file.

7.3.1 DEVICE_TYPE

`DEVICE_TYPE` *DEFAULT, CPU, GPU, ACCELERATOR, CUSTOM, ALL* [local Table]

This table acts like an enumeration, which should be passed as the second parameter to the acml_getDevices callback. It is typically passed with the Lua '.' shorthand notation, such as `DEVICE_TYPE.ALL`

`number DEFAULT` [enum]
: This instructs the library to pick a reasonable default to enumerate; this is guaranteed to only return 1 device

`number CPU` [enum]
: This instructs the library to enumerate all OpenCL devices that identify themselves as being of the CPU class. This can return 0 to N devices.

`number GPU` [enum]
: This instructs the library to enumerate all OpenCL devices that identify themselves as being of the GPU class. This can return 0 to N devices.

`number ACCELERATOR` [enum]
: This instructs the library to enumerate all OpenCL devices that identify themselves as being of the ACCELERATOR class. This can return 0 to N devices.

`number CUSTOM` [enum]
: This instructs the library to enumerate all OpenCL devices that identify themselves as being of the CUSTOM class. This can return 0 to N devices.

`number ALL` [enum]
: This instructs the library to enumerate all OpenCL devices on a given platform. This can return 1 to N devices.

7.3.2 VERSION

`VERSION` *MAJOR, MINOR, PATCH, TWEAK* [global Table]

This encodes the particular version of the ACMLScript. The library will read this table and validate that the version of the ACMLScript is compatible with the library. The version of a script consists of 4 independent numbers forming a version quad.

`number MAJOR` [input]
: The 1st number in the version quad. It is only incremented for major breaking changes, in which an older library would no longer be able to call the script

`number MINOR` [input]
: The 2nd number in the version quad. It is incremented to represented logic changes or functionality differences, but is still interface compatible with the previous version.

`number PATCH` [input]

The 3rd number in the version quad. This is typically used to represent bug fixes or little functionality changes.

`number TWEAK` [input]

The 4th number in the version quad. This is typically used to represent a build number or an incrementing version change.

7.4 ACMLScript callbacks

The ACML library contains API entry points that a script may use to retrieve additional information from the library.

7.4.1 acml_getDevices

`Table acml_getDevices ( platIndex, devType )` [Script Callback]

Returns a list of OpenCL devices detected by the OpenCL runtime, given the platform index and a filter for device type

`int platIndex` [Input]

The index of the platform to retrieve devices from
Constraint: platIndex must be between 1 and the number of platforms, usually passed into the script.

`DEVICE_TYPE devType` [Input]

One of the following enumerated types, to filter the types of devices returned from the API.

⇒ A table is returned from the API that is set up in key-value pairs. The key is an integer index number, starting from 1 as is typical in Lua array numbering. The value is an embedded table that contains the relevant information about the device, stored as key-string/value pairs. The value string describes the data, and the value is the returned result.

This table is frequently iterated through with the Lua `ipairs()` or `pairs()` iterators.

```
-- An example of choosing a platform based upon the
-- platform properties
local myPlatIndex = 1
for index, platform in ipairs( platforms ) do
    if( platform.vendor == "Advanced Micro Devices, Inc." ) then
        devices = acml_getDevices( index, DEVICE_TYPE.ALL )
        myPlatIndex = index
    end
end

-- An example of printing device properties to stdout
for index, device in ipairs( devices ) do
    print( index )
    for name, value in pairs( device ) do
        print( "\t", name, "\t", value )
    end
end
print( )
```

The fields of the table returned by `acml_getDevices` are given by:

`deviceProperties` *name, maxfrequency, maxcomputeunits, globalmemsize, maxalloc, unifiedmemory, extensions, hasdouble* [Table]

name corresponds to CL_DEVICE_NAME
maxfrequency corresponds to CL_DEVICE_MAX_CLOCK_FREQUENCY
maxcomputeunits corresponds to CL_DEVICE_MAX_COMPUTE_UNITS
globalmemsize corresponds to CL_DEVICE_GLOBAL_MEM_SIZE
maxalloc corresponds to CL_DEVICE_MAX_MEM_ALLOC_SIZE
unifiedmemory corresponds to CL_DEVICE_HOST_UNIFIED_MEMORY
extensions corresponds to CL_DEVICE_EXTENSIONS
hasdouble looks for `cl_khr_fp64`,
then `cl_amd_fp64` in CL_DEVICE_EXTENSIONS

7.4.2 acml_getDeviceName

`Table acml_getDeviceName ( contextIndex )` [Script Callback]

Returns a string that identifies the name of the device in the chosen context

`int contextIndex` [Input]

The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ A string representing the name of the device

7.4.3 acml_getHostMaxFrequency

`Table acml_getHostMaxFrequency ( )` [Script Callback]

Return a number that represents the maximum frequency at which the host could operate at. Note: depending on power management, the host could be operating at a lower clock rate at any given point in time.

⇒ An unsigned number given in MHz

7.4.4 acml_getDeviceMaxFrequency

`Table acml_getDeviceMaxFrequency ( contextIndex )` [Script Callback]

Return a number that represents the maximum frequency at which the device in the given context operates at. Note: depending on power management, the device could be operating at a lower clock rate at any given point in time.

`int contextIndex` [Input]

The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ An unsigned number given in MHz

7.4.5 acml_getDeviceMaxComputeUnits

`Table acml_getDeviceMaxComputeUnits ( contextIndex )` [Script Callback]
Returns the maximum number of individual compute units the device in the given context contains.

`int contextIndex` [Input]
The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ An unsigned number

7.4.6 acml_getDeviceMaxMemoryAlloc

`Table acml_getDeviceMaxMemoryAlloc ( contextIndex )` [Script Callback]
Returns the maximum amount of memory that can be allocated within a single buffer for the device in the given context

`int contextIndex` [Input]
The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ An unsigned number representing MB

7.4.7 acml_getUnifiedHostDeviceMemory

`Table acml_getUnifiedHostDeviceMemory ( contextIndex )` [Script Callback]
Return a boolean indicating the the device in the given context shares its main memory with the host; an APU is an example of this

`int contextIndex` [Input]
The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ A boolean

7.4.8 acml_getDeviceGlobalMemorySize

`Table acml_getDeviceGlobalMemorySize ( contextIndex )` [Script Callback]
Returns the amount of memory on the device in the given context

`int contextIndex` [Input]
The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ An unsigned number representing MB

7.4.9 acml_getDeviceDoubleSupport

`Table acml_getDeviceDoubleSupport ( contextIndex )` [Script Callback]

Returns whether the device in the given context supports double precision operations

`int contextIndex` [Input]

The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ A boolean

7.4.10 acml_getDeviceExtensions

`Table acml_getDeviceExtensions ( contextIndex )` [Script Callback]

Returns a string that contains all the OpenCL extensions that the device in the given context supports.

`int contextIndex` [Input]

The index of the context to query
Constraint: contextIndex must be 1 at this time; the library does not support multi-GPU configurations at this time

⇒ A whitespace separated string that contains all OpenCL extensions

8 References

- [1] C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Maths. Soft., 5 (1979), pp. 308–323.
- [2] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson, *An extended set of FORTRAN basic linear algebra subroutines*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.
- [3] J.J. Dongarra, J. Du Croz, I.S. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.
- [4] David S. Dodson, Roger G. Grimes, John G. Lewis, *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 17 (1991), pp. 253–263.
- [5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide*, SIAM, Philidelphia, (1999).
- [6] D. E. Knuth, *The Art of Computer Programming* Addison-Wesley, 1997.
- [7] J. Banks, *Handbook on Simulation*, Wiley, 1998.
- [8] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, Chapter 5, CRC Press, 1996.
- [9] Chapter Introduction G05 - Random Number Generators *The NAG Fortran Library Manual, Mark 21* Numerical Algorithms Group, 2005.
- [10] N. M. Maclaren, The generation of multiple independent sequences of pseudorandom numbers, *Appl. Statist.*, 1989, 38, 351-359.
- [11] M. Matsumoto and T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Transactions on Modelling and Computer Simulations*, 1998.
- [12] P. L'Ecuyer, Good parameter sets for combined multiple recursive random number generators, *Operations Research*, 1999, 47, 159-164.
- [13] Programming languages - C - ISO/IEC 9899:1999
- [14] IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)
- [15] P. L'Ecuyer and R. Simard, *TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators*, Departement d'Informatique et de Recherche Operationnelle, Universite de Montreal, 2002. Software and user's guide available at http://www.iro.umontreal.ca/~lecuyer

Subject Index

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W

Routine Index

A

C

D

F

I

S

U

Z