AMD Core Math Library (ACML)

Version 3.6.0

Short Contents

1 Introduction 1
2 General Information 2
3 BLAS: Basic Linear Algebra Subprograms 19
4 LAPACK: Package of Linear Algebra Subroutines 20
5 Fast Fourier Transforms (FFTs) 24
6 Random Number Generators 74
7 ACML_MV: Fast Math and Fast Vector Math Library 161
8 References 227
Subject Index 228
Routine Index 231

Table of Contents

1 Introduction 1

2 General Information 2

2.1 Determining the best ACML version for your system 2
2.2 Accessing the Library (Linux) 4
2.2.1 Accessing the Library under Linux using GNU g77/gcc 4
2.2.2 Accessing the Library under Linux using GNU gfortran/gcc 5
2.2.3 Accessing the Library under Linux using PGI compilers pgf77/pgf90/pgcc 6
2.2.4 Accessing the Library under Linux using PathScale compilers pathf90/pathcc 6
2.2.5 Accessing the Library under Linux using the NAGWare f95 compiler 7
2.2.6 Accessing the Library under Linux using the Intel ifort compiler 7
2.2.7 Accessing the Library under Linux using compilers other than GNU, PGI, PathScale, NAGWare or Intel 8
2.3 Accessing the Library (Microsoft Windows) 8
2.3.1 Accessing the Library under 32-bit Windows using GNU g77/gcc 8
2.3.2 Accessing the Library under 32-bit Windows using PGI compilers pgf77/pgf90/Microsoft C 9
2.3.3 Accessing the Library under 32-bit Windows using Microsoft C or Intel Fortran 10
2.3.4 Accessing the Library under 32-bit Windows using the Compaq Visual Fortran compiler 11
2.3.5 Accessing the Library under 32-bit Windows using the Salford FTN95 compiler 11
2.3.6 Accessing the Library under 64-bit Windows using PGI compilers pgf77/pgf90/pgcc 11
2.3.7 Accessing the Library under 64-bit Windows using Microsoft C or Intel Fortran 12
2.4 Accessing the Library (Solaris) 13
2.4.1 Accessing the Library under Solaris 13
2.5 ACML FORTRAN and C interfaces 14
2.6 ACML variants using 64-bit integer (INTEGER*8) arguments . . 15
2.7 Library Version and Build Information 16
2.8 Library Documentation 16
2.9 Example programs calling ACML 17
2.10 Example ACML programs demonstrating performance 17

3 BLAS: Basic Linear Algebra Subprograms . . 19

4 LAPACK: Package of Linear Algebra Subroutines 20

4.1 Introduction to LAPACK 20
4.2 Reference sources for LAPACK 20
4.3 LAPACK block sizes, ILAENV and ILAENVSET 21
4.4 IEEE exceptions and LAPACK 23

5 Fast Fourier Transforms (FFTs) 24

5.1 Introduction to FFTs 24
5.1.1 Transform definitions and Storage for Complex Data 24
5.1.2 Transform definitions and Storage for Real Data 25
5.1.3 Efficiency 25
5.1.4 Default and Generated Plans 26
5.2 FFTs on Complex Sequences 27
5.2.1 FFT of a single sequence 27
`ZFFT1D` Routine Documentation 28
`CFFT1D` Routine Documentation 29
`ZFFT1DX` Routine Documentation 30
`CFFT1DX` Routine Documentation 32
5.2.2 FFT of multiple complex sequences 34
`ZFFT1M` Routine Documentation 35
`CFFT1M` Routine Documentation 37
`ZFFT1MX` Routine Documentation 39
`CFFT1MX` Routine Documentation 41
5.2.3 2D FFT of two-dimensional arrays of data 43
`ZFFT2D` Routine Documentation 44
`CFFT2D` Routine Documentation 45
`ZFFT2DX` Routine Documentation 46
`CFFT2DX` Routine Documentation 49
5.2.4 3D FFT of three-dimensional arrays of data 52
`ZFFT3D` Routine Documentation 53
`CFFT3D` Routine Documentation 54
`ZFFT3DX` Routine Documentation 55
`CFFT3DX` Routine Documentation 57
`ZFFT3DY` Routine Documentation 59
`CFFT3DY` Routine Documentation 62
5.3 FFTs on real and Hermitian data sequences 65
5.3.1 FFT of single sequences of real data 66
`DZFFT` Routine Documentation 66
`SCFFT` Routine Documentation 67
5.3.2 FFT of multiple sequences of real data 68
`DZFFTM` Routine Documentation 68
`SCFFTM` Routine Documentation 69
5.3.3 FFT of single Hermitian sequences 70
`ZDFFT` Routine Documentation 70
`CSFFT` Routine Documentation 71
5.3.4 FFT of multiple Hermitian sequences 72
`ZDFFTM` Routine Documentation 72

CSFFTM Routine Documentation 73

6 Random Number Generators 74

6.1 Base Generators 74
6.1.1 Initialization of the Base Generators 75
DRANDINITIALIZE / SRANDINITIALIZE 76
DRANDINITIALIZEBBS / SRANDINITIALIZEBBS 79
6.1.2 Calling the Base Generators 80
DRANDBLUMBLUMSHUB / SRANDBLUMBLUMSHUB 81
6.1.3 Basic NAG Generator 81
6.1.4 Wichmann-Hill Generator 82
6.1.5 Mersenne Twister 82
6.1.6 L'Ecuyer's Combined Recursive Generator 83
6.1.7 Blum-Blum-Shub Generator 83
6.1.8 User Supplied Generators 84
DRANDINITIALIZEUSER / SRANDINITIALIZEUSER 85
UINI 87
UGEN 88
6.2 Multiple Streams 88
6.2.1 Using Different Seeds 89
6.2.2 Using Different Generators 89
6.2.3 Skip Ahead 89
DRANDSKIPAHEAD / SRANDSKIPAHEAD 90
6.2.4 Leap Frogging 92
DRANDLEAPFROG / SRANDLEAPFROG 93
6.3 Distribution Generators 95
6.3.1 Continuous Univariate Distributions 95
DRANDBETA / SRANDBETA 95
DRANDCAUCHY / SRANDCAUCHY 97
DRANDCHISQUARED / SRANDCHISQUARED 99
DRANDEXPONENTIAL / SRANDEXPONENTIAL 101
DRANDF / SRANDF 103
DRANDGAMMA / SRANDGAMMA 105
DRANDGAUSSIAN / DRANDGAUSSIAN 107
DRANDLOGISTIC / SRANDLOGISTIC 109
DRANDLOGNORMAL / SRANDLOGNORMAL 111
DRANDSTUDENTST / SRANDSTUDENTST 113
DRANDTRIANGULAR / SRANDTRIANGULAR 115
DRANDUNIFORM / SRANDUNIFORM 117
DRANDVONMISES / SRANDVONMISES 119
DRANDWEIBULL / SRANDWEIBULL 121
6.3.2 Discrete Univariate Distributions 123
DRANDBINOMIAL / SRANDBINOMIAL 123
DRANDGEOMETRIC / SRANDGEOMETRIC 125
DRANDHYPERGEOMETRIC / SRANDHYPERGEOMETRIC 127
DRANDNEGATIVEBINOMIAL / SRANDNEGATIVEBINOMIAL 129
DRANDPOISSON / SRANDPOISSON 131
DRANDDISCRETEUNIFORM / SRANDDISCRETEUNIFORM 133

DRANDGENERALDISCRETE / SRANDGENERALDISCRETE 135
DRANDBINOMIALREFERENCE / SRANDBINOMIALREFERENCE 137
DRANDGEOMETRICREFERENCE / SRANDGEOMETRICREFERENCE ... 139
DRANDHYPERGEOMETRICREFERENCE / SRANDHYPERGEOMETRICREFERENCE 141
DRANDNEGATIVEBINOMIALREFERENCE / SRANDNEGATIVEBINOMIALREFERENCE 143
DRANDPOISSONREFERENCE / SRANDPOISSONREFERENCE 145
6.3.3 Continuous Multivariate Distributions 147
DRANDMULTINORMAL / SRANDMULTINORMAL 147
DRANDMULTISTUDENTST / SRANDMULTISTUDENTST 149
DRANDMULTINORMALR / SRANDMULTINORMALR 151
DRANDMULTISTUDENTSTR / SRANDMULTISTUDENTSTR 153
DRANDMULTINORMALREFERENCE / SRANDMULTINORMALREFERENCE 155
DRANDMULTISTUDENTSTREFERENCE / SRANDMULTISTUDENTSTREFERENCE 157
6.3.4 Discrete Multivariate Distributions 159
DRANDMULTINOMIAL / SRANDMULTINOMIAL 159

7 ACML_MV: Fast Math and Fast Vector Math Library 161

7.1 Introduction to ACML_MV 161
7.1.1 Terminology 161
7.1.2 Weak Aliases 162
7.1.3 Defined Types 162
7.2 Fast Basic Math Functions 163
fastcos: fast double precision Cosine 163
fastcosf: fast single precision Cosine 164
fastexp: fast double precision exponential function 165
fastexpf: fast single precision exponential function 166
fastlog: fast double precision natural logarithm function 167
fastlogf: fast single precision natural logarithm function 168
fastlog10: fast double precision base-10 logarithm function 169
fastlog10f: fast single precision base-10 logarithm function 170
fastlog2: fast double precision base-2 logarithm function 171
fastlog2f: fast single precision base-2 logarithm function 172
fastpow: fast double precision power function 173
fastpowf: fast single precision power function 175
fastsin: fast double precision Sine 177
fastsinf: fast single precision Sine 178
fastsincos: fast double precision Sine and Cosine 179
fastsincosf: fast single precision Sine and Cosine 180
7.3 Fast Vector Math Functions 181
vrd2_cos: Two-valued double precision Cosine 181
vrd4_cos: Four-valued double precision Cosine 182
vrda_cos: Array double precision Cosine 183
vrs4_cosf: Four-valued single precision Cosine 184

vrsa_cosf: Array single precision Cosine 185
vrd2_exp: Two-valued double precision exponential function 186
vrd4_exp: Four-valued double precision exponential function . . . 187
vrda_exp: Array double precision exponential function 188
vrs4_expf: Four-valued single precision exponential function 189
vrs8_expf: Eight-valued single precision exponential function . . . 190
vrsa_expf: Array single precision exponential function 191
vrd2_log: Two-valued double precision natural logarithm 192
vrd4_log: Four-valued double precision natural logarithm 193
vrda_log: Array double precision natural logarithm 194
vrs4_logf: Four-valued single precision natural logarithm 195
vrs8_logf: Eight-valued single precision natural logarithm 196
vrsa_logf: Array single precision natural logarithm 197
vrd2_log10: Two-valued double precision base-10 logarithm 198
vrd4_log10: Four-valued double precision base-10 logarithm 199
vrda_log10: Array double precision base-10 logarithm 200
vrs4_log10f: Four-valued single precision base-10 logarithm 201
vrs8_log10f: Eight-valued single precision base-10 logarithm 202
vrsa_log10f: Array single precision base-10 logarithm 203
vrd2_log2: Two-valued double precision base-2 logarithm 204
vrd4_log2: Four-valued double precision base-2 logarithm 205
vrda_log2: Array double precision base-2 logarithm 206
vrs4_log2f: Four-valued single precision base-2 logarithm 207
vrs8_log2f: Eight-valued single precision base-2 logarithm 208
vrsa_log2f: Array single precision base-2 logarithm 209
vrs4_powf: Four-valued single precision power function 210
vrsa_powf: Array single precision power function 211
vrs4_powxf: Four-valued single precision power function with constant y 213
vrsa_powxf: Array single precision power function, constant y . . 215
vrd2_sin: Two-valued double precision Sine 217
vrd4_sin: Four-valued double precision Sine 218
vrda_sin: Array double precision Sine 219
vrs4_sinf: Four-valued single precision Sine 220
vrsa_sinf: Array single precision Sine 221
vrd2_sincos: Two-valued double precision Sine and Cosine 222
vrda_sincos: Array double precision Sine and Cosine 223
vrs4_sincosf: Four-valued single precision Sine and Cosine 225
vrsa_sincosf: Array single precision Sine and Cosine 226

8 References 227

Subject Index 228

Routine Index 231

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 19 and see Chapter 4 [LAPACK], page 20).

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. The functionality, interfaces and use of the ACML FFT routines are described below (see Chapter 5 [Fast Fourier Transforms], page 24).

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 74).

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.

A supplementary library of fast math and fast vector math functions (ACML_MV) is also provided with some 64-bit versions of ACML. Some of the functions included in ACML_MV are not callable from high-level languages, but must be called via assembly language; the documentation of ACML_MV (see Chapter 7 [Fast Vector Math Library], page 161) gives details for each individual routine.

2 General Information

2.1 Determining the best ACML version for your system

ACML comes in versions for 64-bit and 32-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)? Or are you running a 32-bit operating system?
- Is the operating system Linux or Microsoft Windows?
- Do you have the GNU compilers (g77/gcc or gfortran/gcc) or compatible compilers (compilers that are interoperable with the GNU compilers) installed?
- Do you have the PGI compilers (pgf77/pgf90/pgcc) installed?
- Do you have the PathScale compilers (pathf90/pathcc) installed?
- Do you have the NAGWare compiler (f95) installed?
- On a 32-bit Windows machine, do you have Microsoft C, or PGI Visual Fortran, or Intel Fortran, or compatible compilers 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.
- If you're on a 32-bit machine, does it support Streaming SIMD Extension instructions (SSE and SSE2)?

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 whether SSE and SSE2 instructions 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 "sse" then your machine supports SSE instructions. If it also includes "sse2" then your machine supports SSE2 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. Older (32-bit) AMD chips may support SSE but not SSE2, or neither SSE nor SSE2 instructions. Other manufacturers' hardware may or may not support SSE or SSE2.

If you link to a version of ACML that was built to use SSE or SSE2 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.

For 32-bit machines, older versions of ACML (ACML 3.1.0 and earlier) came in variants suitable for hardware without SSE/SSE2 instructions (Streaming SIMD Extensions). This is no longer the case, and if you have older 32-bit hardware that does not support SSE/SSE2, and wish to use ACML, you must continue to use an older version.

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

Linux 64-bit

Number of processors	*Compilers*	*ACML install directory*
Single processor	GNU g77/gcc or compatible	`acml3.6.0/gnu64`
"	GNU gfortran/gcc	`acml3.6.0/gfortran64`
"	PGI pgf77/pgf90/pgcc	`acml3.6.0/pgi64`
"	PathScale pathf90/pathcc	`acml3.6.0/pathscale64`
"	NAGWare f95	`acml3.6.0/nag64`
"	Intel Fortran	`acml3.6.0/ifort64`
Multi processor	PGI pgf77/pgf90/pgcc	`acml3.6.0/pgi64_mp`
"	PathScale pathf90/pathcc	`acml3.6.0/pathscale64_mp`
"	GNU gfortran/gcc	`acml3.6.0/gfortran64_mp`
"	Intel Fortran	`acml3.6.0/ifort64_mp`

Linux 32-bit

Number of processors	*Compilers*	*ACML install directory*
Single	GNU g77 / gcc or compat.	`acml3.6.0/gnu32`
"	GNU gfortran / gcc	`acml3.6.0/gfortran32`
"	PGI pgf77 / pgf90 / pgcc	`acml3.6.0/pgi32`
"	PathScale pathf90 / pathcc	`acml3.6.0/pathscale32`
"	NAGWare f95	`acml3.6.0/nag32`
"	Intel Fortran	`acml3.6.0/ifort32`
Multiple	PGI pgf77 / pgf90 / pgcc	`acml3.6.0/pgi32_mp`
"	PathScale pathf90 / pathcc	`acml3.6.0/pathscale32_mp`
"	GNU gfortran / gcc	`acml3.6.0/gfortran32_mp`
"	Intel Fortran	`acml3.6.0/ifort32_mp`

Microsoft Windows 64-bit

Number of processors	*Compilers*	*ACML install directory*
Single processor	PGI pgf77/pgf90/pgcc/MSC	`acml3.6.0/win64`
"	Intel Fortran/Microsoft C	`acml3.6.0/ifort64`
Multi processor	PGI pgf77/pgf90/pgcc/MSC	`acml3.6.0/win64_mp`
"	Intel Fortran/Microsoft C	`acml3.6.0/ifort64_mp`

Microsoft Windows 32-bit

Number of processors	*Compilers*	*ACML install directory*
Single	GNU g77/gcc	`acml3.6.0/gnu32`
”	PGI pgf77/pgf90/Microsoft C	`acml3.6.0/pgi32`
”	Intel Fortran/Microsoft C	`acml3.6.0/ifort32`
Multi	PGI pgf77/pgf90/Microsoft C	`acml3.6.0/pgi32_mp`
”	Intel Fortran/Microsoft C	`acml3.6.0/ifort32_mp`

2.2 Accessing the Library (Linux)

2.2.1 Accessing the Library under Linux using GNU g77/gcc

If the Linux 64-bit g77 version of ACML was installed in the default directory, /opt/acml3.6.0/gnu64, then the command:

```
g77 -m64 driver.f -L/opt/acml3.6.0/gnu64/lib -lacml
```

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

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.

```
g77 -m64 driver.f -L/opt/acml3.6.0/gnu64/lib -static -lacml
```

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

```
g77 -m64 driver.f /opt/acml3.6.0/gnu64/lib/libacml.a
```

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, for example, by the C-shell command:

```
setenv LD_LIBRARY_PATH /opt/acml3.6.0/gnu64/lib
```

where it is assumed that libacml.so was installed in the directory /opt/acml3.6.0/gnu64/lib (see the man page for ld(1) for more information about `LD_LIBRARY_PATH`.).

The command

```
g77 -m32 driver.f -L/opt/acml3.6.0/gnu32/lib -lacml
```

will compile and link a 32-bit program with a 32-bit ACML.

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

```
gcc -m64 -I/opt/acml3.6.0/gnu64/include driver.c
    -L/opt/acml3.6.0/gnu64/lib -lacml -lg2c
```

The switch "-I/opt/acml3.6.0/gnu64/include" tells the compiler to search the directory /opt/acml3.6.0/gnu64/include for the ACML C header file acml.h, which should be included by driver.c. Note that it is necessary to add the compiler run-time library -lg2c when linking the program.

2.2.2 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/acml3.6.0/gfortran64, then the command:

```
gfortran -m64 driver.f -L/opt/acml3.6.0/gfortran64/lib -lacml
```

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

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/acml3.6.0/gfortran64/lib -static -lacml
```

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

```
gfortran -m64 driver.f /opt/acml3.6.0/gfortran64/lib/libacml.a
```

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/acml3.6.0/gfortran64/lib, then `LD_LIBRARY_PATH` may be set by, for example, the C-shell command

```
setenv LD_LIBRARY_PATH /opt/acml3.6.0/gfortran64/lib
```

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

The command

```
gfortran -m32 driver.f -L/opt/acml3.6.0/gfortran32/lib -lacml
```

will compile and link a 32-bit program with a 32-bit ACML.

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/acml3.6.0/gfortran64_mp/lib -lacml_mp
gfortran -fopenmp -m32 driver.f
         -L/opt/acml3.6.0/gfortran32_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/acml3.6.0/gfortran64/include driver.c
    -L/opt/acml3.6.0/gfortran64/lib -lacml -lgfortran
```

The switch "-I/opt/acml3.6.0/gfortran64/include" tells the compiler to search the directory /opt/acml3.6.0/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.

2.2.3 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/acml3.6.0/pgi64/lib -lacml
pgf77 -tp=k8-32 -Mcache_align driver.f -L/opt/acml3.6.0/pgi32/lib -lacml
```

will compile driver.f and link it to the ACML using 64-bit and 32-bit versions respectively. In the example above we are linking with the single-processor PGI version of ACML.

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/acml3.6.0/pgi64_mp/lib -lacml_mp
pgf77 -tp=k8-32 -mp -Mcache_align driver.f
      -L/opt/acml3.6.0/pgi32_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/acml3.6.0/pgi64_mp/include driver.c
pgcc -tp=k8-64 -mp -Mcache_align driver.o
      -L/opt/acml3.6.0/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/acml3.6.0/pgi64_mp/include for the ACML C header file acml.h, which should be included by driver.c, by using the switch "-I/opt/acml3.6.0/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.

2.2.4 Accessing the Library under Linux using PathScale compilers pathf90/pathcc

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

```
pathf90 driver.f -L/opt/acml3.6.0/pathscale64/lib -lacml
```

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

The commands

```
pathcc -c -I/opt/acml3.6.0/pathscale64/include driver.c
pathcc driver.o -L/opt/acml3.6.0/pathscale64/lib -lacml -lpathfortran
```

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

```
-I/opt/acml3.6.0/pathscale64/include
```

tells the C compiler to search directory /opt/acml3.6.0/pathscale64/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 -lpathfortran to the link command, so that the required PathScale compiler run-time library is found.

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

```
pathf90 -mp driver.f -L/opt/acml3.6.0/pathscale64_mp/lib -lacml_mp
pathf90 -mp driver.f -L/opt/acml3.6.0/pathscale32_mp/lib -lacml_mp
```

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

The -mp flag is important - it tells pathf90 to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time.

The commands

```
pathcc -c -mp -I/opt/acml3.6.0/pathscale64_mp/include driver.c
pathcc -mp driver.o -L/opt/acml3.6.0/pathscale64_mp/lib -lacml_mp
       -lpathfortran
```

will compile driver.c and link it to the 64-bit ACML. Again, the -mp flag is important if you are linking to the PathScale OpenMP version of ACML. The C compiler is instructed to search the directory /opt/acml3.6.0/pathscale64_mp/include for the ACML C header file acml.h, which should be included by driver.c, by using the switch "-I/opt/acml3.6.0/pathscale64_mp/include". Note that in the example we add the library -lpathfortran to the link command, so that a required PathScale compiler run-time library is found.

2.2.5 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/acml3.6.0/nag64/lib -lacml
f95 -32 driver.f -L/opt/acml3.6.0/nag32/lib -lacml
```

will compile driver.f and link it to the ACML using the 64-bit version and 32-bit version respectively.

2.2.6 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/acml3.6.0/ifort64/lib -lacml
```

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

The commands

```
gcc -c -I/opt/acml3.6.0/ifort64/include driver.c
ifort -nofor-main driver.o -L/opt/acml3.6.0/ifort64/lib -lacml
```

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

```
-I/opt/acml3.6.0/ifort64/include
```

tells the C compiler to search directory /opt/acml3.6.0/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/acml3.6.0/ifort64_mp/lib -lacml_mp
ifort -openmp driver.f -L/opt/acml3.6.0/ifort32_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.

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

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 32-bit Windows using GNU g77/gcc

Under Microsoft Windows®, for the g77/gcc version of ACML it is assumed that you have the Cygwin UNIX-like tools installed (see http://www.cygwin.com/), including the g77/gcc compiler and associated tools. Assuming you have installed the ACML in the default place, then in a DOS command prompt window, the command

```
g77 driver.f "c:\Program Files\AMD\acml3.6.0\gnu32\lib\libacml.a"
```

can be used to link the application program driver.f to the static library version of the ACML.

The g77 version of the ACML Library is supplied in both static and shareable versions, libacml.a and libacml.dll, respectively. The command given above links to the static version of the library, libacml.a. To link to the DLL version, the command

```
g77 driver.f "c:\Program Files\AMD\acml3.6.0\gnu32\lib\libacml.dll"
```

can be used. Notice that if the application program has been linked to the DLL version of the ACML Library, then before running the program, the environment variable `PATH` must have been set to include the location of the DLL, for example by the DOS command:

```
PATH="c:\Program Files\AMD\acml3.6.0\gnu32\lib";%PATH%
```

where it was assumed that libacml.dll was installed in the directory "c:\Program Files\AMD\acml3.6.0\gnu32\lib". Alternatively, the PATH environment variable may be set in the system category of the Windows control panel.

The command

```
gcc "-Ic:\Program Files\AMD\acml3.6.0\gnu32\include" driver.c
    "c:\Program Files\AMD\acml3.6.0\gnu32\lib\libacml.a" -lg2c
```

will compile driver.c and link it to the 32-bit g77/gcc version of ACML. The switch "-Ic:\Program Files\AMD\acml3.6.0\gnu32\include" tells the gcc compiler to search directory "c:\Program Files\AMD\acml3.6.0\gnu32\include" for the ACML C header file acml.h, which should be included by driver.c. Note that it is necessary to add the compiler run-time library -lg2c when linking the program.

2.3.2 Accessing the Library under 32-bit Windows using PGI compilers pgf77/pgf90/Microsoft C

To use the 32-bit Windows PGI version of ACML, use a command like

```
pgf77 -Munix driver.f
      "c:\Program Files\AMD\acml3.6.0\pgi32\lib\libacml_dll.lib"
```

where libacml_dll.lib is the import library for the ACML DLL. Note that it is important to use the compiler switch -Munix in order to tell the compiler to use the same calling convention as was used to build ACML.

In the example above we are linking with the single-processor PGI version of ACML.

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 -Munix -mp driver.f
      "c:\Program Files\AMD\acml3.6.0\pgi32\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.

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

```
cl "-Ic:\Program Files\AMD\acml3.6.0\pgi32\include"
     /MD driver.c
     "c:\Program Files\AMD\acml3.6.0\pgi32\lib\libacml_dll.lib"
cl "-Ic:\Program Files\AMD\acml3.6.0\pgi32_mp\include"
     /MD driver.c
     "c:\Program Files\AMD\acml3.6.0\pgi32_mp\lib\libacml_mp_dll.lib"
```

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

2.3.3 Accessing the Library under 32-bit Windows using Microsoft C or Intel Fortran

To use the 32-bit Windows MSC/Intel Fortran version of ACML, use a command like

```
ifort /threads /libs:dll driver.f
      "c:\Program Files\AMD\acml3.6.0\ifort32\lib\libacml_dll.lib"
```

where libacml_dll.lib is the import library for the ACML 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:\acml3.6.0\ifort32_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.

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

```
cl "-Ic:\Program Files\AMD\acml3.6.0\ifort32\include"
     /MD driver.c
     "c:\Program Files\AMD\acml3.6.0\ifort32\lib\libacml_dll.lib"
cl "-Ic:\Program Files\AMD\acml3.6.0\ifort32_mp\include"
     /MD driver.c
     "c:\Program Files\AMD\acml3.6.0\ifort32_mp\lib\libacml_mp_dll.lib"
```

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

ACML can also be linked from inside a development environment such as Microsoft Visual Studio or Visual Studio.NET. Again, it is important to get compilation options correct. The directory acml3.6.0\ifort32\examples\Projects contains a few sample Visual Studio project directories showing how this can be done.

Note that in both examples above we linked to a DLL version of ACML, and so before running the resulting programs the environment variable `PATH` must be set to include the location of the DLL. For example, assuming that libacml_dll.dll was installed in "c:\Program Files\AMD\acml3.6.0\ifort32\lib", `PATH` may be set by, for example, the DOS command

```
PATH="c:\Program Files\AMD\acml3.6.0\ifort32\lib";%PATH%
```

Alternatively, the PATH environment variable may be set in the system category of the Windows control panel.

ACML also comes as a static (non-DLL) library, named libacml.lib, in the same directory as the DLL. If you link to the static library instead of the DLL import library then there is no need to set the PATH.

2.3.4 Accessing the Library under 32-bit Windows using the Compaq Visual Fortran compiler

The win32 Intel Fortran variant of ACML can be used with the Compaq Visual Fortran compiler as follows:

```
f90 /iface:cref,nomixed_str_len_arg /threads /libs:dll driver.f
      "c:\Program Files\AMD\acml3.6.0\ifort32\lib\libacml_dll.lib"
```

where f90 is the Compaq Visual Fortran command line compiler and libacml_dll.lib is the import library for the ACML DLL. The switch /iface:cref,nomixed_str_len_arg used on the f90 compiler command line is important - it tells the compiler to use a calling convention equivalent to the default Intel Fortran calling convention, rather than the default cvf _stdcall calling convention. If you forget to use this switch your program is likely to crash on execution.

2.3.5 Accessing the Library under 32-bit Windows using the Salford FTN95 compiler

The win32 Intel Fortran variant of ACML can be used with the Salford ftn95 compiler as follows:

```
ftn95 driver.f
```

The resulting object file can be linked using the Salford linker, slink, for example like this:

```
slink driver.obj install_dir\libacml_dll.dll
```

where install_dir is the location of the DLL. The full pathname of install_dir should be specified to the DLL and should be enclosed within quotes if it contains spaces. It is worth emphasising that the linker should link directly against the DLL itself, not the libacml_dll.lib import library.

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

Under 64-bit versions of Windows, ACML 3.6.0 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:/acml3.6.0/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:/acml3.6.0/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:/acml3.6.0/win64/include
     c:/acml3.6.0/win64/lib/libacml_dll.lib
pgcc -Mdll -mp driver.c -Ic:/acml3.6.0/win64_mp/include
     c:/acml3.6.0/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:/acml3.6.0/win64/include
     c:/acml3.6.0/win64/lib/libacml_dll.lib
cl driver.c -Ic:/acml3.6.0/win64_mp/include
     c:/acml3.6.0/win64_mp/lib/libacml_mp_dll.lib
```

for single- and multi-threaded ACML variants respectively.

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

Under 64-bit versions of Windows, ACML 3.6.0 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:\acml3.6.0\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:\acml3.6.0\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:/acml3.6.0/ifort64/include
     c:/acml3.6.0/ifort64/lib/libacml_dll.lib
cl driver.c -Ic:/acml3.6.0/ifort64_mp/include
     c:/acml3.6.0/ifort64_mp/lib/libacml_mp_dll.lib
```

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

2.4 Accessing the Library (Solaris)

2.4.1 Accessing the Library under Solaris

If the Solaris 64-bit f95 version of ACML was installed in the default directory, /opt/acml3.6.0/sun64, then the command:

```
f95 -xarch=amd64 driver.f -L/opt/acml3.6.0/sun64/lib -lacml
```

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

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 `-Bstatic`, e.g.

```
f95 -xarch=amd64 driver.f -L/opt/acml3.6.0/sun64/lib -Bstatic -lacml
```

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

```
f95 -xarch=amd64 driver.f /opt/acml3.6.0/sun64/lib/libacml.a
```

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, for example, by the C-shell command:

```
setenv LD_LIBRARY_PATH /opt/acml3.6.0/sun64/lib
```

where it is assumed that libacml.so was installed in the directory /opt/acml3.6.0/sun64/lib (see the man page for ld(1) for more information about `LD_LIBRARY_PATH`.).

The command

```
f95 -xarch=sse2 driver.f -L/opt/acml3.6.0/sun32/lib -lacml
```

will compile and link a 32-bit program with a 32-bit ACML.

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

```
cc -xarch=amd64 -I/opt/acml3.6.0/sun64/include driver.c
    -L/opt/acml3.6.0/sun64/lib -lacml -lfsu -lsunmath -lm
```

The switch "-I/opt/acml3.6.0/sun64/include" tells the compiler to search the directory /opt/acml3.6.0/sun64/include for the ACML C header file acml.h, which should be included by driver.c. Note that it is necessary to add the Sun compiler run-time libraries -lfsu -lsunmath -lm when linking the program.

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

```
f95 -openmp -xarch=amd64 driver.f -L/opt/acml3.6.0/sun64_mp/lib -lacml_mp
f95 -openmp -xarch=sse2 driver.f -L/opt/acml3.6.0/sun32_mp/lib -lacml_mp
```

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

The -openmp flag is important - it tells f95 to link with the appropriate compiler OpenMP run-time library. Without it you might get an "unresolved symbol" message at link time.

The command

```
cc -openmp -xarch=amd64 -I/opt/acml3.6.0/sun64/include driver.c
    -L/opt/acml3.6.0/sun64/lib -lacml_mp -lfsu -lsunmath -lm -lmtsk
```

will compile driver.c and link it to the 64-bit ACML. Again, the -openmp flag is important if you are linking to the OpenMP version of ACML. The C compiler is instructed to search the directory /opt/acml3.6.0/sun64_mp/include for the ACML C header file acml.h, which should be included by driver.c, by using the switch "-I/opt/acml3.6.0/sun64_mp/include". Note that in the example we add the libraries -lfsu -lsunmath -lm -lmtsk to the link command, so that required compiler run-time libraries are found.

2.5 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 32-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 2.9 [Examples], page 17) 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);
```

2.6 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, PathScale, and gfortran 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.

2.7 Library Version and Build Information

This document is applicable to version 3.6.0 of ACML. The utility routine `acmlversion` can be called to obtain the major, minor and patch version numbers of the installed ACML. This routine returns three integers; the major, minor and patch version numbers, respectively.

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`*) [SUBROUTINE]

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

`ACMLINFO ()` [SUBROUTINE]

C specifications:

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

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

2.8 Library Documentation

The /Doc subdirectory of the top ACML installation directory, (e.g. /opt/acml3.6.0/Doc under Linux, or c:\Program Files\AMD\acml3.6.0\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/acml3.6.0/Doc

% info acml
```

or simply by using the full name of the file:

```
% info /opt/acml3.6.0/Doc/acml.info
```

2.9 Example programs calling ACML

The /examples subdirectory of the top ACML installation directory (for example, possible default locations are /opt/acml3.6.0/gnu64/examples under Linux, or, under windows, c:\Program Files\AMD\acml3.6.0\gnu32\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.

The 32-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.

For the 64-bit Windows version of ACML, 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.

2.10 Example ACML programs demonstrating performance

The /examples/performance subdirectory of the top ACML installation directory (for example, possible default locations are /opt/acml3.6.0/gnu64/examples/performance under Linux, or c:\Program Files\AMD\acml3.6.0\gnu32\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 32- and 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.f, time_dgemm.f and time_dgetrf.f, 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.

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.

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 21) 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.

5 Fast Fourier Transforms (FFTs)

5.1 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.

5.1.1 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 $l = 0, 1, \ldots, m-1$, 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.

5.1.2 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. The symmetries defining such a sequence mean that it 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.

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 $Z(i)$ for $i = 0, 1, ..., n-1$, where $Z(n-j)$ is the complex conjugate of $Z(j)$ for $j = 1, 2, ..., (n-1)/2$; $Z(0)$ is real valued; and, if n is even, $Z(n/2)$ is real valued. In ACML, the representation of Hermitian sequences used on output from `DZFFT` routines and on input to `ZDFFT` routines is as follows:
let X be an array of length N and with first index 0,

- $X(i)$ contains the real part of $Z(i)$ for $i = 0, ..., N/2$
- $X(N-i)$ contains the imaginary part of $Z(i)$ for $i = 1, ..., (N-1)/2$

Also, given a Hermitian sequence, the discrete transform can be written as:

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

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 above definitions, both transforms have the same (negative) sign in the exponent; performing both 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).

5.1.3 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.

5.1.4 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.

5.2 FFTs on Complex Sequences

5.2.1 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)
```

5.2.2 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*, *Y*,*INCY1*,*INCY2*,*COMM*,*INFO*) [SUBROUTINE]

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)
```

5.2.3 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)) [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)
```

5.2.4 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 has been 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 are 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(L*M*N+5*(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 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*=−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.

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*=−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 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(L*M*N+5*(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
        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*, [SUBROUTINE]
INCX1,*INCX2*,*INCX3*,*Y*,*INCY1*,*INCY2*,*INCY3*,*COMM*,*LCOMM*,*INFO*)

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*, [SUBROUTINE]
INCX1,*INCX2*,*INCX3*,*Y*,*INCY1*,*INCY2*,*INCY3*,*COMM*,*LCOMM*,*INFO*)

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 can be reduced significantly by declaring *COMM* to be of length at least L*M*N + 4*(L+M+N) + 300..

Constraint: *LCOMM* > L*M*N + 2*(L+M+N) + 300.

`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)
```

5.3 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 a condensed form that is described in Section 5.1 [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.

5.3.1 FFT of single sequences of real data

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 or 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 or 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)
```

5.3.2 FFT of multiple sequences of real data

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)
```

5.3.3 FFT of single Hermitian sequences

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)
```

5.3.4 FFT of multiple Hermitian sequences

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)
```

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 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.

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 74), and twenty-three distribution generators (Section 6.3 [Distribution Generators], page 95). 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 84).

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].

If a single stream of variates is required it is recommended that the Mersenne Twister (Section 6.1.5 [Mersenne Twister], page 82) 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 81) 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 82). 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 83) is used in combination with the skip ahead routine (Section 6.2.3 [Skip Ahead], page 89). 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 88.

The Blum-Blum-Shub generator (Section 6.1.7 [Blum-Blum-Shub Generator], page 83) 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 76, [DRANDINITIALIZEBBS], page 79 and [DRANDINITIALIZEUSER], page 85, 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 88.

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`*`,` *`LSTATE`*`,`*`INFO`*`)` [SUBROUTINE]

`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 81).
- 2 = Wichmann-Hill generator (Section 6.1.4 [Wichmann-Hill Generator], page 82).
- 3 = Mersenne Twister (Section 6.1.5 [Mersenne Twister], page 82).
- 4 = L'Ecuyer's Combined Recursive generator (Section 6.1.6 [L'Ecuyer's Combined Recursive Generator], page 83).
- 5 = Blum-Blum-Shub generator (Section 6.1.7 [Blum-Blum-Shub Generator], page 83).

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*, *INFO*) [SUBROUTINE]

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 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 117). The base generator used is, as usual, selected during the initialization process (see Section 6.1.1 [Initialization of the Base Generators], page 75).

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 75 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 $0\text{x}DD\cdots$ 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 $a_{11} = 0, a_{12} = 1403580, a_{13} = 810728, m_1 = 2^{32} - 209, a_{21} = 527612, a_{22} = 0, a_{33} = 1370589$ 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 95 require a base generator which returns a uniformly distributed value in the open interval $(0, 1)$ and ACML includes several such generators (as detailed in Section 6.1 [Base Generators], page 74). 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 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 2.9 [Examples], page 17) 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`*`,` [SUBROUTINE]
`  `*`STATE`*`,`*`LSTATE`*`,`*`INFO`*`)`

`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 84 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*≤ 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 half-open interval $[0.0, 1.0)$ - i.e. 0.0 is a legitimate return value, but 1.0 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 92 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 75, 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 76 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:

$$\underbrace{x_1, x_2, \cdots, x_n,}_{\text{block 1}} \quad \underbrace{x_{n+1}, x_{n+2}, \cdots, x_{2n},}_{\text{block 2}} \quad \underbrace{x_{2n+1}, x_{2n+2}, \cdots, x_{3n},}_{\text{block 3}} \text{ etc}$$

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 75 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)
      INTEGER 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 75 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)
      INTEGER  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 75 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 75 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 75 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 75 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 75 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 75 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 75 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 75 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 75 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 75 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 75 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,XMAX,XMED
        DOUBLE PRECISION X(N)

C       Set the seed
        SEED(1) = 1234

C       Read in the distributional parameters
        READ(5,*) XMIN,XMAX,XMED

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,XMAX,XMED,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 75 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 75 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 75 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 75 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:

$$f(X) = P(1-P)^{(X-1)}, X = 1, 2, \cdots$$

(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 75 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 75 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 75 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 75 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 75 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 75 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:

$$f(X) = P(1-P)^{(X-1)}, X = 1, 2, \cdots$$

(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: $LDC \geq N$.

`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 `DRANDMULTINORMAL` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 75 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: $LDX \geq M$.

`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: $LDC \geq N$.

`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 `DRANDMULTISTUDENTST` *STATE* must have been initialized. See Section 6.1.1 [Initialization of the Base Generators], page 75 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: $LDX \geq M$.

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 75 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: $LDX \geq M$.

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 75 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: $LDX \geq M$.

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: $LDC \geq N$.

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: $LDC \geq N$.

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 75 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: $LDX \geq M$.

`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, MK
      PARAMETER (LSTATE=16,N=100,MK=10)
      INTEGER I,J,INFO,SEED(1),STATE(LSTATE)
      INTEGER LDC,LDX,K,M
      INTEGER  X(N,MK)
      DOUBLE PRECISION P(MK)

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 ACML_MV: Fast Math and Fast Vector Math Library

7.1 Introduction to ACML_MV

ACML_MV is a library which contains fast and/or vectorized versions of some familiar math library routines such as sin, cos and exp. The routines take advantage of the AMD64 architecture for performance, and so are currently only available with 64-bit versions of ACML. The routines in the library are very accurate over the range of acceptable input arguments.

Some of the performance is gained by sacrificing error handling or the acceptance of certain arguments. It is therefore the responsibility of the caller of these routines to ensure that their arguments are suitable. Furthermore, some of the routines are not callable from high-level languages at all, but must be called via assembly language; see the documentation of individual routines for details. Hence, these routines are intended to be utilized by knowledgeable users only.

7.1.1 Terminology

The individual documentation for a routine states what outputs will be returned for special arguments, and also gives an indication of performance of the routine. In general, special case arguments for any routine will cause a return value in accordance with the C99 language standard [13].

Special case arguments include NaNs and infinities, as defined by the IEEE arithmetic standard [14]. In these documents, *NaN* means *Not a Number*, *QNaN* means *Quiet NaN*, and *SNaN* means *Signalling NaN*.

A *denormal* number is a number which is very tiny (close to the machine arithmetic underflow threshold) and is stored to less precision than a normal number. Due to their special nature, operations on such numbers are often very slow. While such numbers might not necessarily be regarded as special case arguments, for the sake of performance some of the ACML_MV routines have been designed not to handle them. This has been noted in the documentation for each ACML_MV routine.

Performance of a routine is given in machine cycles, and is thus independent of processor speed.

Accuracy of a routine is quoted in *ulps*, where *ulp* stands for *Unit in the Last Place*. Since floating-point numbers on a computer are limited precision approximations of mathematical numbers, not all real numbers can be represented by machine numbers, and the machine number must in general be rounded to available precision. An *ulp* is the distance between the two machine numbers that bracket a real number.

In this document, the *ulp* is used as a measure of the error in a returned result when compared with the mathematically exact expected result. Because of the finite nature of machine arithmetic, a routine can never in general achieve accuracy of better than 0.5 *ulps*, and an accuracy of less than 1 *ulp* is good.

7.1.2 Weak Aliases

Some of the functions in ACML_MV include a weak alias to an equivalent function in libm. For example, the fastcos function includes a weak alias to cos. If ACML_MV is included in the link order before libm, then all calls to the aliased libm function name (e.g. cos) will use the equivalent ACML_MV routine (e.g. fastcos). If ACML_MV is included in the link order after libm, then all calls to libm functions will use the libm versions.

ACML_MV routines can always be accessed using their ACML_MV names (e.g. fastcos), regardless of link order.

7.1.3 Defined Types

The following types are used to describe the functions contained in this chapter:

__m128d a pair of double precision values;
__m128 four single precision values.

7.2 Fast Basic Math Functions

This section documents the interfaces to a set of basic mathematical functions.

fastcos: fast double precision Cosine

`double fastcos` (*double x*)

Weak alias: cos

C Prototype:

double fastcos (double x);

Inputs:

double x - the double precision input value.

Outputs:

Cosine of x.

Fortran Function Interface:

DOUBLE PRECISION FASTCOS(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

Cosine of X.

Notes:

fastcos computes the Cosine function of its argument x.

This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 2 ulp over most of the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

88 cycles for most valid inputs < 5e5.

fastcosf: fast single precision Cosine

`float fastcosf` (*float x*)

Weak alias: cosf

C Prototype:

float fastcosf (float x);

Inputs:

float x - the single precision input value.

Outputs:

Single precision Cosine of x.

Fortran Function Interface:

REAL FASTCOSF(X)

Inputs:

REAL X - the single precision input value.

Return Value:

Cosine of X.

Notes:

fastcosf computes the single precision Cosine function of its argument x.

This is a *relaxed* version of cosf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 ulp over most of the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

91 cycles for most valid inputs < 5e5.

fastexp: fast double precision exponential function

`double fastexp` (*double x*)

Weak alias: exp

C Prototype:

double fastexp (double x);

Inputs:

double x - the double precision input value.

Outputs:

e raised to the power x (exponential of x).

Fortran Function Interface:

DOUBLE PRECISION FASTEXP(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

e raised to the power X (exponential of X).

Notes:

fastexp computes the double precision exponential function of the input argument x.

This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

75 cycles for most valid inputs.

fastexpf: fast single precision exponential function

`float fastexpf` (*float x*)

Weak alias: expf

C Prototype:

float fastexpf (float x);

Inputs:

float x - the single precision input value.

Outputs:

e raised to the power x (exponential of x).

Fortran Function Interface:

REAL FASTEXPF(X)

Inputs:

REAL X - the single precision input value.

Return Value:

e raised to the power X (exponential of X).

Notes:

fastexpf computes the single precision exponential function of the input argument x.

This is a *relaxed* version of expf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

75 cycles for most valid inputs.

fastlog: fast double precision natural logarithm function

`double fastlog` (*double x*)

Weak alias: log

C Prototype:

double fastlog (double x);

Inputs:

double x - the double precision input value.

Outputs:

The natural logarithm (base e) of x.

Fortran Function Interface:

DOUBLE PRECISION FASTLOG(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

The natural logarithm (base e) of X.

Notes:

fastlog computes the double precision natural logarithm of its argument x.

This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

97 cycles for most valid inputs.
86 cycles for .97 < x < 1.03

fastlogf: fast single precision natural logarithm function

`float fastlogf` (*float x*)

Weak alias: logf

C Prototype:

float fastlogf (float x);

Inputs:

float x - the single precision input value.

Outputs:

The natural logarithm (base e) of x.

Fortran Function Interface:

REAL FASTLOGF(X)

Inputs:

REAL X - the single precision input value.

Return Value:

The natural logarithm (base e) of X.

Notes:

fastlogf computes the single precision natural logarithm of its argument x.

This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

94 cycles for most valid inputs.
85 cycles for .97 < x < 1.03

fastlog10: fast double precision base-10 logarithm function

`double fastlog10` (*double x*)

Weak alias: log10

C Prototype:

double fastlog10 (double x);

Inputs:

double x - the double precision input value.

Outputs:

The base-10 logarithm of x.

Fortran Function Interface:

DOUBLE PRECISION FASTLOG10(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

The base-10 logarithm of X.

Notes:

fastlog10 computes the double precision base-10 logarithm of its argument x.

This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

112 cycles for most valid inputs.

fastlog10f: fast single precision base-10 logarithm function

`float fastlog10f` (*float x*)

Weak alias: log10f

C Prototype:

float fastlog10f (float x);

Inputs:

float x - the single precision input value.

Outputs:

The base-10 logarithm of x.

Fortran Function Interface:

REAL FASTLOG10F(X)

Inputs:

REAL X - the single precision input value.

Return Value:

The base-10 logarithm of X.

Notes:

fastlog10f computes the single precision base-10 logarithm of its argument x.

This is a *relaxed* version of log10f, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

104 cycles for most valid inputs.

fastlog2: fast double precision base-2 logarithm function

`double fastlog2` (*double x*)

Weak alias: log2

C Prototype:

double fastlog2 (double x);

Inputs:

double x - the double precision input value.

Outputs:

The base-2 logarithm of x.

Fortran Function Interface:

DOUBLE PRECISION FASTLOG2(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

The base-2 logarithm of X.

Notes:

fastlog2 computes the double precision base-2 logarithm of its argument x.

This is a *relaxed* version of log2, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

112 cycles for most valid inputs.

fastlog2f: fast single precision base-2 logarithm function

`float fastlog2f` (*float x*)

Weak alias: log2f

C Prototype:

float fastlog2f (float x);

Inputs:

float x - the single precision input value.

Outputs:

The base-2 logarithm of x.

Fortran Function Interface:

REAL FASTLOG2F(X)

Inputs:

REAL X - the single precision input value.

Return Value:

The base-2 logarithm of X.

Notes:

fastlog2f computes the single precision base-2 logarithm of its argument x.

This is a *relaxed* version of log2f, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

107 cycles for most valid inputs.

fastpow: fast double precision power function

`double fastpow` (*double x, double y*)

Weak alias: pow

C Prototype:

double fastpow (double x, double y);

Inputs:

double x - the double precision base input value.

double y - the double precision exponent input value.

Outputs:

x raised to the power y.

Fortran Function Interface:

DOUBLE PRECISION FASTPOW(X,Y)

Inputs:

DOUBLE PRECISION X - the base value.

DOUBLE PRECISION Y - the exponent value.

Return Value:

X raised to the power Y.

Notes:

fastpow computes the x raised to the power y in double precision.

This is a *relaxed* version of pow, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs will produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0
$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$

$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	NaN
x<>1	NaN,	NaN

Performance:

200 cycles for most valid inputs.

fastpowf: fast single precision power function

`float fastpowf` (*float x, float y*)

Weak alias: powf

C Prototype:

float fastpowf (float x, float y);

Inputs:

float x - the single precision base input value.

float y - the single precision exponent input value.

Outputs:

x raised to the power y.

Fortran Function Interface:

REAL FASTPOWF(X,Y)

Inputs:

REAL X - the single precision base value.

REAL Y - the single precision exponent value.

Return Value:

X raised to the power Y.

Notes:

fastpowf computes the x raised to the power y in single precision.

This is a *relaxed* version of powf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs will produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 0.5 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0
$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$

$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	NaN
x<>1	NaN,	NaN

Performance:

175 cycles for most valid inputs.

fastsin: fast double precision Sine

`double fastsin` (*double x*)

Weak alias: sin

C Prototype:

double fastsin (double x);

Inputs:

double x - the double precision input value.

Outputs:

Sine of x.

Fortran Function Interface:

DOUBLE PRECISION FASTSIN(X)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Return Value:

Sine of X.

Notes:

fastsin computes the Sine function of its argument x.

This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

88 cycles for most valid inputs < 5e5.

fastsinf: fast single precision Sine

`float fastsinf` (*float x*)

Weak alias: sinf

C Prototype:

float fastsinf (float x);

Inputs:

float x - the single precision input value.

Outputs:

Single precision Sine of x.

Fortran Function Interface:

REAL PRECISION FASTSINF(X)

Inputs:

REAL PRECISION X - the single precision input value.

Return Value:

Sine of X.

Notes:

fastsinf computes the Sine function of its argument x.

This is a *relaxed* version of sinf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

88 cycles for most valid inputs < 5e5.

fastsincos: fast double precision Sine and Cosine

`void fastsincos` (*double x, double s, double c*)

Weak alias: sincos

C Prototype:

void fastsincos (double x, double s, double c);

Inputs:

double x - the double precision input value.

Outputs:

double s - Sine of x.

double c - Cosine of x.

Fortran Subroutine Interface:

SUBROUTINE FASTSINCOS(X,S,C)

Inputs:

DOUBLE PRECISION X - the double precision input value.

Outputs:

DOUBLE PRECISION S - Sine of X.

DOUBLE PRECISION C - Cosine of X.

Notes:

fastsincos computes the Sine and Cosine functions of its argument x.

This function can provide a significant performance advantage for applications that require both the sine and cosine of an angle, such as axis and matrix rotation. This is a *relaxed* version of sincos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

99 cycles for most valid inputs < 5e5.

fastsincosf: fast single precision Sine and Cosine

`void fastsincosf` (*float x, float s, float c*)

Weak alias: sincosf

C Prototype:

void fastsincosf (float x, float s, float c);

Inputs:

float x - the single precision input value.

Outputs:

float s - Sine of x.

float c - Cosine of x.

Fortran Subroutine Interface:

SUBROUTINE FASTSINCOSF(X,S,C)

Inputs:

REAL X - the single precision input value.

Outputs:

REAL S - Sine of X.

REAL C - Cosine of X.

Notes:

fastsincosf computes the Sine and Cosine functions of its argument x.

This function can provide a significant performance advantage for applications that require both the sine and cosine of an angle, such as axis and matrix rotation. This is a *relaxed* version of sincosf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Error inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

91-102 cycles for most valid inputs < 5e5.

7.3 Fast Vector Math Functions

This section documents the interfaces to a set of vector mathematical functions.

vrd2_cos: Two-valued double precision Cosine

`__m128d __vrd2_cos` (*__m128d x*)

C Prototype:

__m128d __vrd2_cos(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

__m128d y - the double precision Cosine result pair, returned in xmm0.

Notes:

__vrd2_cos computes the Cosine function of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision Cosine of both values, returned as a __m128d value. This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

120 cycles for most valid inputs < 5e5 (60 cycles per value).

vrd4_cos: Four-valued double precision Cosine

`__m128d,__m128d __vrd4_cos` (*__m128d x1,__m128d x2*)

C Prototype:

__m128d __vrd2_cos(__m128d x);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

__m128d y1 - the first double precision Cosine result pair, returned in xmm0.

__m128d y2 - second double precision Cosine result pair, returned in xmm1.

Notes:

__vrd4_cos computes the Cosine function of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision Cosine of the four values, returned as two __m128d values. This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value).

vrda_cos: Array double precision Cosine

`void vrda_cos` (*int n, double *x, double *y*)

C Prototype:

void vrda_cos (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

Cosine for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_COS(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of Cosines of input values.

Notes:

vrda_cos computes the Cosine function for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes cos(x) for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of cos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value), n = 24.

vrs4_cosf: Four-valued single precision Cosine

`__m128 __vrs4_cosf (__m128 x)`

C Prototype:

__m128 __vrs4_cosf(__m128 x);

Inputs:

__m128 x - the four single precision input values.

Outputs:

__m128 y - the four single precision Cosine results , returned in xmm0.

Notes:

__vrs4_cosf computes the Cosine function of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision Cosine of all four values, returned as a __m128 value. This is a *relaxed* version of cosf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

176 cycles for most valid inputs < 5e5 (44 cycles per value).

vrsa_cosf: Array single precision Cosine

`void vrsa_cosf` (*int n, float *x, float *y*)

C Prototype:

void vrsa_cosf (int n, float *x, float *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

Cosine for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_COSF(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of Cosines of input values.

Notes:

vrsa_cosf computes the Cosine function for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes cosf(x) for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of cosf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

43 cycles per value for most valid inputs < 5e5, n = 24.

vrd2_exp: Two-valued double precision exponential function

`__m128d __vrd2_exp` (*__m128d x*)

C Prototype:

__m128d __vrd2_exp(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

e raised to the power x (exponential of x).

__m128d y - the double precision exponent result pair, returned in xmm0.

Notes:

__vrd2_exp computes the exponential function of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision exponent of both values, returned as a __m128d value. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

80 cycles for most valid inputs (40 cycles per value).

vrd4_exp: Four-valued double precision exponential function

`__m128d,__m128d __vrd4_exp` (*__m128d x1, __m128d x2*)

Prototype:

__m128d,__m128d __vrd4_exp(__m128d x1, __m128d x2);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

__m128d y1 - the first double precision exponent result pair, returned in xmm0.

__m128d y2 - the second double precision exponent result pair, returned in xmm1.

Notes:

__vrd4_exp computes the double precision exponential function of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision exponent of the four values, returned as two __m128d values. This is a *relaxed* version of exp, suitable for use with fast-math compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

132 cycles for most valid inputs (33 cycles per value).

vrda_exp: Array double precision exponential function

`void vrda_exp` (*int n, double *x, double *y*)

C Prototype:

void vrda_exp (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

e raised to the power x (exponential of x) for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_EXP(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of exponentials (e raised to the power x) of input values.

Notes:

vrda_exp computes the double precision exponential function for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes the e^x for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -708.5	0
> 709.8	$+\infty$

Performance:

33 cycles per value for valid inputs, n = 24.

vrs4_expf: Four-valued single precision exponential function

`__m128 __vrs4_expf (__m128 x)`

C Prototype:

__m128 __vrs4_expf(__m128 x);

Inputs:

__m128 x - the four single precision input values.

Outputs:

e raised to the power x (exponential of x) for each input value x.
__m128 y - the four single precision exponent results, returned in xmm0.

Notes:

__vrs4_expf computes the double precision exponential function of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision exponent of the four values, returned as a __m128 value. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

91 cycles for most valid inputs (23 cycles per value).

vrs8_expf: Eight-valued single precision exponential function

`__m128,__m128 __vrs8_expf` (*__m128 x1, __m128 x2*)

Prototype:

__m128,__m128 __vrs8_expf(__m128 x1, __m128 x2);

Note that this function uses a non-standard programming interface. The two __m128 inputs, which contain eight single precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128 is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128 x1 - the first single precision vector of four input values.
__m128 x2 - the second single precision vector of four input values.

Outputs:

__m128 y1 - the first four single precision exponent results, returned in xmm0.
__m128 y2 - the second four single precision exponent results, returned in xmm1.

Notes:

__vrs8_expf computes the single precision exponential function of eight input arguments.

This routine accepts eight single precision input values passed as two __m128 values. The result is the single precision exponent of the eight values, returned as two __m128 values. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

155 cycles for most valid inputs (19 cycles per value).

vrsa_expf: Array single precision exponential function

`void vrsa_expf` (*int n, float *x, float *y*)

C Prototype:

void vrsa_expf (int n, float *x, float *y)

Inputs:

int	n	- the number of single precision values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

e raised to the power x (exponential of x) for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_EXPF(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of exponentials (e raised to the power x) of input values.

Notes:

vrsa_expf computes the single precision exponential function for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes the e^x for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of exp, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
< -87.5	0
> 88	$+\infty$

Performance:

15 cycles per value for valid inputs, n = 24.

vrd2_log: Two-valued double precision natural logarithm

`__m128d __vrd2_log` (*__m128d x*)

C Prototype:

__m128d __vrd2_log(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

The natural (base e) logarithm of x.

__m128d y - the double precision natural logarithm result pair, returned in xmm0.

Notes:

__vrd2_log computes the natural logarithm for each of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision natural logarithm of both values, returned as a __m128d value. This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

130 cycles for most valid inputs (65 cycles per value).

vrd4_log: Four-valued double precision natural logarithm

`__m128d,__m128d __vrd4_log` (*__m128d x1, __m128d x2*)

Prototype:

__m128d,__m128d __vrd4_log(__m128d x1, __m128d x2);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

The natural (base e) logarithm of x.

__m128d y1 - the first double precision natural logarithm result pair, returned in xmm0.

__m128d y2 - the second double precision natural logarithm result pair, returned in xmm1.

Notes:

__vrd4_log computes the natural logarithm for each of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision natural logarithm of the four values, returned as two __m128d values. This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

196 cycles for most valid inputs (49 cycles per value).

vrda_log: Array double precision natural logarithm

`void vrda_log` (*int n, double *x, double *y*)

C Prototype:

void vrda_log (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

The natural (base e) logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_LOG(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of natural (base e) logarithms of input values.

Notes:

vrda_log computes the double precision natural logarithm for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes the natural log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of log, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

51 cycles per value for valid inputs, n = 24.

vrs4_logf: Four-valued single precision natural logarithm

`__m128 __vrs4_logf (__m128 x)`

C Prototype:

__m128 __vrs4_logf(__m128 x);

Inputs:

__m128 x - the single precision input values.

Outputs:

The natural (base e) logarithm of x.

__m128 y - the single precision natural logarithm results, returned in xmm0.

Notes:

__vrs4_logf computes the natural logarithm for each of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision natural logarithm of all four values, returned as a __m128 value. This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

124 cycles for most valid inputs (31 cycles per value).

vrs8_logf: Eight-valued single precision natural logarithm

`__m128,__m128 __vrs8_logf (__m128 x1, __m128 x2)`

Prototype:

__m128,__m128 __vrs8_logf(__m128 x1, __m128 x2);

Note that this function uses a non-standard programming interface. The two __m128 inputs, which contain eight single precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128 is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128 x1 - the first single precision input value pair.

__m128 x2 - the second single precision input value pair.

Outputs:

The natural (base e) logarithm of x.

__m128 y1 - the first single precision natural logarithm result pair, returned in xmm0.

__m128 y2 - the second single precision natural logarithm result pair, returned in xmm1.

Notes:

__vrs8_logf computes the natural logarithm for each of eight input arguments.

This routine accepts eight single precision input values passed as two __m128 values. The result is the single precision natural logarithm of the eight values, returned as two __m128 values. This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

200 cycles for most valid inputs (25 cycles per value).

vrsa_logf: Array single precision natural logarithm

`void vrsa_logf` (*int n, float *x, float *y*)

C Prototype:

void vrsa_logf (int n, float *x, float *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

The natural (base e) logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_LOGF(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of natural (base e) logarithms of input values.

Notes:

vrsa_logf computes the single precision natural logarithm for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes the natural log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of logf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

26 cycles per value for valid inputs, n = 24.

vrd2_log10: Two-valued double precision base-10 logarithm

`__m128d __vrd2_log10 (__m128d x)`

C Prototype:

__m128d __vrd2_log10(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

The base-10 logarithm of x.

__m128d y - the double precision base-10 logarithm result pair, returned in xmm0.

Notes:

__vrd2_log10 computes the base-10 logarithm for each of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision base-10 logarithm of both values, returned as a __m128d value. This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

142 cycles for most valid inputs (71 cycles per value), longer for input values very close to 1.0.

vrd4_log10: Four-valued double precision base-10 logarithm

`__m128d,__m128d __vrd4_log10` (*__m128d x1, __m128d x2*)

Prototype:

__m128d,__m128d __vrd4_log10(__m128d x1, __m128d x2);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

The base-10 logarithm of x.

__m128d y1 - the first double precision base-10 logarithm result pair, returned in xmm0.

__m128d y2 - the second double precision base-10 logarithm result pair, returned in xmm1.

Notes:

__vrd4_log10 computes the base-10 logarithm for each of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision base-10 logarithm of the four values, returned as two __m128d values. This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

235 cycles for most valid inputs (59 cycles per value), longer for input values very close to 1.0.

vrda_log10: Array double precision base-10 logarithm

`void vrda_log10` (*int n, double *x, double *y*)

C Prototype:

void vrda_log10 (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

The base-10 logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_LOG10(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of base-10 logarithms of input values.

Notes:

vrda_log10 computes the double precision base-10 logarithm for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes the base-10 log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

54 cycles per value for valid inputs, n = 24, longer for input values very close to 1.0.

vrs4_log10f: Four-valued single precision base-10 logarithm

`__m128 __vrs4_log10f (__m128 x)`

Prototype:

__m128 __vrs4_log10f(__m128 x);

Inputs:

__m128 x - the four single precision inputs.

Outputs:

The base-10 logarithm of x.

__m128 y - the four single precision base-10 logarithm results, returned in xmm0.

Notes:

__vrs4_log10f computes the base-10 logarithm for each of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision base-10 logarithm of the four values, returned as a __m128 value. This is a *relaxed* version of log10, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

141 cycles for most valid inputs (35 cycles per value), longer for input values very close to 1.0.

vrs8_log10f: Eight-valued single precision base-10 logarithm

`__m128,__m128 __vrs8_log10f` (*__m128 x1, __m128 x2*)

Prototype:

__m128,__m128 __vrs8_log10f(__m128 x1, __m128 x2);

Note that this function uses a non-standard programming interface. The two __m128 inputs, which contain eight single precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128 is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128 x1 - the first set of four single precision input values.

__m128 x2 - the second set of four single precision input values.

Outputs:

The base-10 logarithm of x.

__m128 y1 - the first set of four single precision base-10 logarithm results, returned in xmm0.

__m128 y2 - the second set of four single precision base-10 logarithm results, returned in xmm1.

Notes:

__vrs8_log10f computes the base-10 logarithm for each of eight input arguments.

This routine accepts eight single precision input values passed as two __m128 values. The result is the single precision base-10 logarithm of the eight values, returned as two __m128 values. This is a *relaxed* version of log10f, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

231 cycles for most valid inputs (29 cycles per value), longer for input values very close to 1.0.

vrsa_log10f: Array single precision base-10 logarithm

`void vrsa_log10f` (*int n, float *x, float *y*)

C Prototype:

void vrsa_log10f (int n, float *x, float *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

The base-10 logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_LOG10F(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of base-10 logarithms of input values.

Notes:

vrsa_log10f computes the single precision base-10 logarithm for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes the base-10 log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of log10f, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

28 cycles per value for valid inputs, n = 24, longer for input values very close to 1.0.

vrd2_log2: Two-valued double precision base-2 logarithm

`__m128d __vrd2_log2 (__m128d x)`

C Prototype:

__m128d __vrd2_log2(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

The base-2 logarithm of x.

__m128d y - the double precision base-2 logarithm result pair, returned in xmm0.

Notes:

__vrd2_log2 computes the base-2 logarithm for each of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision base-2 logarithm of both values, returned as a __m128d value. This is a *relaxed* version of log2, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

142 cycles for most valid inputs (71 cycles per value), longer for input values very close to 1.0.

vrd4_log2: Four-valued double precision base-2 logarithm

`__m128d,__m128d __vrd4_log2` (*__m128d x1, __m128d x2*)

Prototype:

__m128d,__m128d __vrd4_log2(__m128d x1, __m128d x2);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

The base-2 logarithm of x.

__m128d y1 - the first double precision base-2 logarithm result pair, returned in xmm0.

__m128d y2 - the second double precision base-2 logarithm result pair, returned in xmm1.

Notes:

__vrd4_log2 computes the base-2 logarithm for each of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision base-2 logarithm of the four values, returned as two __m128d values. This is a *relaxed* version of log2, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

235 cycles for most valid inputs (59 cycles per value), longer for input values very close to 1.0.

vrda_log2: Array double precision base-2 logarithm

`void vrda_log2` (*int n, double *x, double *y*)

C Prototype:

void vrda_log2 (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

The base-2 logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_LOG2(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of base-2 logarithms of input values.

Notes:

vrda_log2 computes the double precision base-2 logarithm for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes the base-2 log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of log2, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

54 cycles per value for valid inputs, n = 24, longer for input values very close to 1.0.

vrs4_log2f: Four-valued single precision base-2 logarithm

`__m128 __vrs4_log2f (__m128 x)`

Prototype:

__m128 __vrs4_log2f(__m128 x);

Inputs:

__m128 x - the four single precision inputs.

Outputs:

The base-2 logarithm of x.

__m128 y - the four single precision base-2 logarithm results, returned in xmm0.

Notes:

__vrs4_log2f computes the base-2 logarithm for each of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision base-2 logarithm of the four values, returned as a __m128 value. This is a *relaxed* version of log2, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

141 cycles for most valid inputs (35 cycles per value), longer for input values very close to 1.0.

vrs8_log2f: Eight-valued single precision base-2 logarithm

`__m128,__m128 __vrs8_log2f` (*__m128 x1, __m128 x2*)

Prototype:

__m128,__m128 __vrs8_log2f(__m128 x1, __m128 x2);

Note that this function uses a non-standard programming interface. The two __m128 inputs, which contain eight single precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128 is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128 x1 - the first set of four single precision input values.

__m128 x2 - the second set of four single precision input values.

Outputs:

The base-2 logarithm of x.

__m128 y1 - the first set of four single precision base-2 logarithm results, returned in xmm0.

__m128 y2 - the second set of four single precision base-2 logarithm results, returned in xmm1.

Notes:

__vrs8_log2f computes the base-2 logarithm for each of eight input arguments.

This routine accepts eight single precision input values passed as two __m128 values. The result is the single precision base-2 logarithm of the eight values, returned as two __m128 values. This is a *relaxed* version of log2f, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

203 cycles for most valid inputs (25 cycles per value), longer for input values very close to 1.0.

vrsa_log2f: Array single precision base-2 logarithm

`void vrsa_log2f` (*int n, float *x, float *y*)

C Prototype:

void vrsa_log2f (int n, float *x, float *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

The base-2 logarithm of each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_LOG2F(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of base-2 logarithms of input values.

Notes:

vrsa_log2f computes the single precision base-2 logarithm for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes the base-2 log for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of log2f, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
± 0	$-\infty$
negative	*QNaN*
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	$+\infty$
$-\infty$	*QNaN*

Performance:

29 cycles per value for valid inputs, n = 24, longer for input values very close to 1.0.

vrs4_powf: Four-valued single precision power function

`__m128 __vrs4_powf (__m128 x,__m128 y)`

C Prototype:

__m128 __vrs4_powf(__m128 x,__m128 y);

Inputs:

__m128 x - the single precision input base values.
__m128 y - the single precision input exponent values.

Outputs:

__m128 z - the single precision results of each x raised to the y power, returned in xmm0.

Notes:

__vrs4_powf() computes the single precision x raised to the y power for four pairs of input arguments. This routine accepts four single precision input value pairs passed as __m128 values. The result is the x raised to the y power for all four input pairs, returned as a __m128 value.

This is a *relaxed* version of powf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 0.5 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0
$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$
$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	*NaN*
x<>1	*NaN*,	*NaN*

Performance:

400 cycles for most valid inputs (100 cycles per value).

vrsa_powf: Array single precision power function

`void vrsa_powf` (*int n, float *x, float *y, float *z*)

C Prototype:

void vrsa_powf(int n, float *x, float *y, float *z);

Inputs:

float *x - pointer to the array of single precision input x values.
float *y - pointer to the array of single precision input y values. float *z - pointer to the array of single precision output values. int n - the number of single precision values in both the input and output arrays.

Outputs:

x raised to the y value for each array pair, filled into the z array.

Fortran Subroutine Interface:

VRSA_POWF(INTEGER*4 N, REAL*4 X(), REAL*4 Y(), REAL*4 Z())

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of real x input values.

REAL Y(N) - array of real y input values.

Outputs:

REAL Z(N) - array of real result values.

Notes:

vrsa_powf() computes x to the y power in single precision for each pair of elements in the x and y input arrays.

This routine accepts an array of single precision input x values and an arrayi of single precision input y values, computes x^y for each input value pair, and stores the result in the array pointed to by the z input.

This is a *relaxed* version of powf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 0.5 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$

|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0
$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$
$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	*NaN*
x<>1	*NaN*,	*NaN*

Performance:

107 cycles per value for valid inputs, n = 24.

vrs4_powxf: Four-valued single precision power function with constant y

`__m128 __vrs4_powxf` (*__m128 x,float y*)

C Prototype:

__m128 __vrs4_powxf(__m128 x,float y);

Inputs:

__m128 x - the single precision input base values.
float y - the common single precision input exponent value.

Outputs:

__m128 z - the single precision results of each x raised to the y power, returned in xmm0.

Notes:

__vrs4_powxf() computes the single precision x raised to the y power for four input x arguments and a constant y input value. This routine accepts four single precision input values passed as an __m128 value. The y value is passed as one single precision value. The result is the x raised to the y power for all four input values, returned as a __m128 value.

This is a *relaxed* version of powxf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 0.5 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1
$+1$	y (incl. *NaN*)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	*QNaN*
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0
$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$
$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	*NaN*
x<>1	*NaN*,	*NaN*

Performance:

372 cycles for most valid inputs (93 cycles per value).

vrsa_powxf: Array single precision power function, constant y

`void vrsa_powf` (*int n, float *x, float y, float *z*)

C Prototype:

void vrsa_powxf(int n, float *x, float y, float *z);

Inputs:

int n - the number of single precision values in both the x input and output arrays.

float *x - pointer to the array of single precision input x values.

float *z - pointer to the array of single precision output values.

float y - the constant single precision input y value.

Outputs:

x raised to the y value for each x array value, filled into the z array

Fortran Subroutine Interface:

VRSA_POWF(INTEGER*4 N, REAL*4 X(), REAL*4 Y, REAL*4 Z())

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of real x input values.

REAL Y - the constant single precision input y value.

Outputs:

REAL Z(N) - array of real result values.

Notes:

vrsa_powxf() computes x to the y power in single precision for each element in the x input arrays, using a constant y.

This routine accepts an array of single precision input x values and one single precision input y value, computes x^y for each x input value, and stores the result in the array pointed to by the z pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array.

This is a *relaxed* version of powf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 0.5 *ulp* over the valid input range.

Special case return values:

Input x	**Input** y	**Output**
± 0	$y < 0$, odd integer	$\pm\infty$
± 0	$y < 0$, not odd integer	$+\infty$
± 0	$y > 0$, odd integer	± 0
± 0	$y > 0$, not odd integer	$+0$
-1	$+\infty$	1

$+1$	y (incl. NaN)	1
x (incl. Nan)	± 0	1
$x < 0$	y, not integer	$QNaN$
|x|<1	$-\infty$	$+\infty$
|x|>1	$-\infty$	$+0$
|x|<1	$+\infty$	$+0$
|x|>1	$+\infty$	$+\infty$
$-\infty$	$y < 0$, odd integer	-0
$-\infty$	$y < 0$, not odd integer	$+0$
$-\infty$	$y > 0$, odd integer	$-\infty$
$-\infty$	$y > 0$, not odd integer	$+\infty$
$+\infty$	$y < 0$,	$+0$
$+\infty$	$y > 0$,	$+\infty$
NaN	y nonzero,	NaN
x<>1	NaN,	NaN

Performance:

115 cycles per value for valid inputs, n = 24.

vrd2_sin: Two-valued double precision Sine

`__m128d __vrd2_sin` (*__m128d x*)

C Prototype:

__m128d __vrd2_sin(__m128d x);

Inputs:

__m128d x - the double precision input value pair.

Outputs:

__m128d y - the double precision Sine result pair, returned in xmm0.

Notes:

__vrd2_sin computes the Sine function of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision Sine of both values, returned as a __m128d value. This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

120 cycles for most valid inputs < 5e5 (60 cycles per value).

vrd4_sin: Four-valued double precision Sine

`__m128d,__m128d __vrd4_sin` (*__m128d x1,__m128d x2*)

C Prototype:

__m128d __vrd4_sin(__m128d x);

Note that this function uses a non-standard programming interface. The two __m128d inputs, which contain four double precision values, are passed by the AMD64 C ABI in registers xmm0, and xmm1. The corresponding results are returned in xmm0 and xmm1. The use of xmm1 to return a __m128d is non-standard, and this function can not be called directly from C. It can be called directly from assembly language. It is intended for internal use by vectorizing compilers, that may be able to take advantage of the non-standard calling interface.

Inputs:

__m128d x1 - the first double precision input value pair.

__m128d x2 - the second double precision input value pair.

Outputs:

__m128d y1 - the first double precision Sine result pair, returned in xmm0.

__m128d y2 - second double precision Sine result pair, returned in xmm1.

Notes:

__vrd4_sin computes the Sine function of four input arguments.

This routine accepts four double precision input values passed as two __m128d values. The result is the double precision Sine of the four values, returned as two __m128d values. This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range. This routine may return slightly worse than 1 *ulp* for very large values between 4e5 and 5e5.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value).

vrda_sin: Array double precision Sine

`void vrda_sin` (*int n, double *x, double *y*)

C Prototype:

void vrda_sin (int n, double *x, double *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*y	- pointer to the array of output values.

Outputs:

Sine for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_SIN(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION Y(N) - array of Sines of input values.

Notes:

vrda_sin computes the Sine function for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes sin(x) for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of sin, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range. This routine may return slightly worse than 1 *ulp* for very large values between 4e5 and 5e5.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

172 cycles for most valid inputs < 5e5 (43 cycles per value), n = 24.

vrs4_sinf: Four-valued single precision Sine

`__m128 __vrs4_sinf (__m128 x)`

C Prototype:

__m128 __vrs4_sinf(__m128 x);

Inputs:

__m128 x - the four single precision inputs.

Outputs:

__m128 y - the four single precision Sine results, returned in xmm0.

Notes:

__vrs4_sinf computes the Sine function of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision Sine of the four values, returned as a __m128 value. This is a *relaxed* version of sinf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range. This routine may return slightly worse than 1 *ulp* for very large values between 4e5 and 5e5.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

176 cycles for most valid inputs < 5e5 (44 cycles per value).

vrsa_sinf: Array single precision Sine

`void vrsa_sinf` (*int n, float *x, float *y*)

C Prototype:

void vrsa_sinf (int n, float *x, float *y)

Inputs:

int	n	- the number of values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*y	- pointer to the array of output values.

Outputs:

Sine for each x value, filled into the y array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_SINF(N,X,Y)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL Y(N) - array of Sines of input values.

Notes:

vrsa_sinf computes the Sine function for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes sin(x) for each input value, and stores the result in the array pointed to by the y pointer input. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of sinf, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

43 cycles per value for most valid inputs < 5e5, n = 24.

vrd2_sincos: Two-valued double precision Sine and Cosine

`void __vrd2_sincos (__m128d x, __m128d* S, __m128d* C)`

C Prototype:

void __vrd2_sincos(__m128d x, __m128d* S, __m128d* C));

Inputs:

__m128d x - the double precision input value pair.

Outputs:

(Sine of x and Cosine of x.)

__m128d *S - Pointer to the double precision Sine result pair.

__m128d *C - Pointer to the double precision Cosine result pair.

Notes:

__vrd2_sincos computes the Sine and Cosine functions of two input arguments.

This routine accepts a pair of double precision input values passed as a __m128d value. The result is the double precision Sin and Cosine of both values, returned as a __m128d value. This is a *relaxed* version of sincos, suitable for use with fastmath compiler flags or application

not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

154 cycles for most valid inputs < 5e5 (77 cycles per Sine and Cosine of a value).

vrda_sincos: Array double precision Sine and Cosine

`void vrda_sincos` (*int n, double *x, double *ys, double *yc*)

C Prototype:

void vrda_sincos (int n, double *x, double *ys, double *yc)

Inputs:

int	n	- the number of values in both the input and output arrays.
double	*x	- pointer to the array of input values.
double	*ys	- pointer to the array of sin output values.
double	*yc	- pointer to the array of cos output values.

Outputs:

Sine for each x value, filled into the ys array.

Cosine for each x value, filled into the yc array.

Fortran Subroutine Interface:

SUBROUTINE VRDA_SINCOS(N,X,YS,YC)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

DOUBLE PRECISION X(N) - array of double precision input values.

Outputs:

DOUBLE PRECISION YS(N) - array of Sines of input values.

DOUBLE PRECISION YC(N) - array of Cosines of input values.

Notes:

vrda_sincos computes the Sine and Cosine functions for each element of an array of input arguments.

This routine accepts an array of double precision input values, computes sincos(x) for each input value, and stores the results in the arrays pointed to by the ys and yc pointer inputs. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of sincos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 2 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

180 cycles for most valid inputs < 5e5 (43 cycles per Sin and Cos of a value), n = 24.

vrs4_sincosf: Four-valued single precision Sine and Cosine

`void __vrs4_sincosf (__m128 x, __m128* S, __m128* C)`

C Prototype:

void __vrs4_sincosf(__m128 x, __m128* S, __m128* C));

Inputs:

__m128 x - the single precision input value pair.

Outputs:

(Sine of x and Cosine of x.)

__m128 *S - Pointer to the single precision Sine result pair.

__m128 *C - Pointer to the single precision Cosine result pair.

Notes:

__vrs4_sincosf computes the Sine and Cosine functions of four input arguments.

This routine accepts four single precision input values passed as a __m128 value. The result is the single precision Sin and Cosine of all four values, returned as a __m128 value. This is a *relaxed* version of sincosf, suitable for use with fastmath compiler flags or application

not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

220 cycles for most valid inputs < 5e5 (55 cycles per Sine and Cosine of a value).

vrsa_sincosf: Array single precision Sine and Cosine

`void vrsa_sincosf` (*int n, float *x, float *ys, float *yc*)

C Prototype:

void vrsa_sincosf (int n, float *x, float *ys, float *yc)

Inputs:

int	n	- the number of values in both the input and output arrays.
float	*x	- pointer to the array of input values.
float	*ys	- pointer to the array of sin output values.
float	*yc	- pointer to the array of cos output values.

Outputs:

Sine for each x value, filled into the ys array.

Cosine for each x value, filled into the yc array.

Fortran Subroutine Interface:

SUBROUTINE VRSA_SINCOSF(N,X,YS,YC)

Inputs:

INTEGER N - the number of values in both the input and output arrays.

REAL X(N) - array of single precision input values.

Outputs:

REAL YS(N) - array of Sines of input values.

REAL YC(N) - array of Cosines of input values.

Notes:

vrsa_sincosf computes the Sine and Cosine functions for each element of an array of input arguments.

This routine accepts an array of single precision input values, computes sincos(x) for each input value, and stores the results in the arrays pointed to by the ys and yc pointer inputs. It is the responsibility of the calling program to allocate/deallocate enough storage for the output array. This is a *relaxed* version of sincos, suitable for use with fastmath compiler flags or applications not requiring full error handling. Denormal inputs may produce unpredictable results. Special case inputs produce C99 return values. The routine is accurate to better than 1 *ulp* over the valid input range.

Special case return values:

Input	**Output**
QNaN	same *QNaN*
SNaN	same *NaN* converted to *QNaN*
$+\infty$	*QNaN*
$-\infty$	*QNaN*

Performance:

53 cycles per value for most valid inputs < 5e5, n = 24.

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

2

2D FFT 43

3

3D FFT 52

A

accessing ACML (Compaq Visual Fortran under 32-bit Windows) 11
accessing ACML (GNU g77/gcc under 32-bit Windows) 8
accessing ACML (GNU g77/gcc under Linux) . . . 4
accessing ACML (GNU gfortran/gcc under Linux) 5
accessing ACML (Intel Fortran/Microsoft C under 32-bit Windows) 10
accessing ACML (Intel Fortran/Microsoft C under 64-bit Windows) 12
accessing ACML (Intel ifort under Linux) 7
accessing ACML (Linux) 4
accessing ACML (NAGware f95 compiler under Linux) 7
accessing ACML (other compilers under Linux) . . 8
accessing ACML (PathScale pathf90/pathcc under Linux) 6
accessing ACML (PGI pgf77/pgf90/Microsoft C under 32-bit Windows) 9
accessing ACML (PGI pgf77/pgf90/pgcc or Microsoft C under 64-bit Windows) 11
accessing ACML (PGI pgf77/pgf90/pgcc under Linux) 6
accessing ACML (Salford ftn95 under 32-bit Windows) 11
accessing ACML (Solaris) 13
accessing ACML (Sun f95/cc under Solaris) 13
accessing ACML under Windows 8
accessing the base generators 80
ACML C Interfaces 14
ACML FORTRAN interfaces 14
ACML installation test 17
ACML performance examples 17
ACML version information 16
ACML_MV (ACML vector math functions) . . . 161
ACML_MV types 162

B

base generator 74
base generator, basic NAG generator 81
base generator, blum-blum-shub 83
base generator, calling 80
base generator, definition 74
base generator, initialization 75
base generator, L'Ecuyer's combined recursive generator 83
base generator, Mersenne twister 82
base generator, recommendation 74
base generator, user supplied 84
base generator, Wichmann-Hill 82
basic NAG base generator 81
beta distribution 95
binomial distribution 123
binomial distribution, using reference vector . . 137
BLAS 19
blum-blum-shub generator 83
BRNG, definition 74

C

C interfaces in ACML 14
calling the base generators 80
cauchy distribution 97
chi-squared distribution 99
complex FFT 24
continuous multivariate distribution, gaussian 147
continuous multivariate distribution, gaussian, using reference vector 151
continuous multivariate distribution, normal . . 147
continuous multivariate distribution, normal, using reference vector 151
continuous multivariate distribution, students t 149
continuous multivariate distribution, students t, using reference vector 153
continuous univariate distribution, beta 95
continuous univariate distribution, cauchy 97
continuous univariate distribution, chi-squared 99
continuous univariate distribution, exponential 101
continuous univariate distribution, f 103
continuous univariate distribution, fisher's variance ratio distribution 103
continuous univariate distribution, gamma 105
continuous univariate distribution, gaussian . . . 107
continuous univariate distribution, logistic 109
continuous univariate distribution, lognormal . . 111
continuous univariate distribution, normal 107
continuous univariate distribution, students t . . 113
continuous univariate distribution, t 113
continuous univariate distribution, triangular . . 115
continuous univariate distribution, uniform . . . 117
continuous univariate distribution, von mises . . 119
continuous univariate distribution, weibull 121
copying a generator 75
cryptologically secure, definition 74
cryptologically secure, generator 83

D

determining the best ACML version for your system . . . 2
discrete multivariate distribution, multinomial . . . 159
discrete univariate distribution, binomial . . . 123
discrete univariate distribution, binomial, using reference vector . . . 137
discrete univariate distribution, geometric . . . 125
discrete univariate distribution, geometric, using reference vector . . . 139
discrete univariate distribution, hypergeometric . . . 127
discrete univariate distribution, hypergeometric, using reference vector . . . 141
discrete univariate distribution, negative binomial . . . 129
discrete univariate distribution, negative binomial, using reference vector . . . 143
discrete univariate distribution, poisson . . . 131
discrete univariate distribution, poisson, using reference vector . . . 145
discrete univariate distribution, uniform . . . 133
distribution generator, definition . . . 74

E

example programs . . . 17
exponential distribution . . . 101

F

f distribution . . . 103
fast basic math functions . . . 163
Fast Fourier Transforms . . . 24
feedback shift generator . . . 82
FFT . . . 24
FFT efficiency . . . 25
FFT of multiple complex sequences . . . 34
FFT of multiple Hermitian sequences . . . 72
FFT of multiple real sequences . . . 68
FFT of single complex sequence . . . 27
FFT of single Hermitian sequence . . . 70
FFT of single real sequence . . . 66
FFT plan . . . 26
fisher's variance ratio distribution . . . 103
FORTRAN interfaces in ACML . . . 14

G

gamma distribution . . . 105
gaussian distribution (multivariate) . . . 147
gaussian distribution (univariate) . . . 107
gaussian distribution, multivariate, using reference vector . . . 151
general information . . . 2
generalized feedback shift generator . . . 82
generating discrete variates from a reference vector . . . 135
geometric distribution . . . 125
geometric distribution, using reference vector . . . 139

H

Hermitian data sequences (FFT) . . . 65
hypergeometric distribution . . . 127
hypergeometric distribution, using reference vector . . . 141

I

IEEE exceptions and LAPACK . . . 23
initialization of a generator . . . 75
installation test . . . 17
INTEGER*8 arguments . . . 15
introduction . . . 1

L

L'Ecuyer's combined recursive generator . . . 83
language interfaces . . . 14
LAPACK . . . 20
LAPACK blocking factors . . . 21
LAPACK reference sources . . . 20
libm names . . . 162
library manual . . . 16
library version information . . . 16
linear congruential generator, basic NAG generator . . . 81
linear congruential generator, Wichmann-Hill . . . 82
linking with ACML . . . 2
linking with Linux ACML . . . 4
linking with Solaris ACML . . . 13
linking with Windows ACML . . . 8
logistic distribution . . . 109
lognormal distribution . . . 111

M

Mersenne twister . . . 82
Mersenne twister, multiple streams . . . 89
multinomial distribution . . . 159
multiple recursive generator, L'Ecuyer's combined recursive generator . . . 83
multiple streams . . . 88
multiple streams, block splitting . . . 89
multiple streams, L'Ecuyer's combined recursive generator . . . 89, 92
multiple streams, leap frogging . . . 92
multiple streams, Mersenne twister . . . 89
multiple streams, NAG basic generator . . . 89, 92
multiple streams, skip ahead . . . 89
multiple streams, using different generators . . . 89
multiple streams, using different seeds . . . 89
multiple streams, Wichmann-Hill generator . . . 89, 92
multivariate distribution, gaussian . . . 147
multivariate distribution, gaussian, using reference vector . . . 151
multivariate distribution, multinomial . . . 159

multivariate distribution, normal 147
multivariate distribution, normal, using reference vector 151
multivariate distribution, students t 149
multivariate distribution, students t, using reference vector 153

N

negative binomial distribution 129
negative binomial distribution, using reference vector 143
normal distribution (multivariate) 147
normal distribution (univariate) 107
normal distribution, multivariate, using reference vector 151

P

performance example programs 17
period of a random number generator, definition 74
plan, default, FFTs 26
plan, generated, FFTs 26
poisson distribution 131
poisson distribution, using reference vector . . . 145
PRNG, definition 74
pseudo-random number, definition 74

Q

QRNG, definition 74
quasi-random number, definition 74

R

random bit stream 80
real data sequences (FFT) 65
real FFT 65
reference vector, binomial distribution 137
reference vector, gaussian (multivariate) 155
reference vector, generating discrete variates from 135
reference vector, geometric distribution 139
reference vector, hypergeometric distribution . . 141
reference vector, negative binomial distribution 143
reference vector, normal (multivariate) 155
reference vector, poisson distribution 145
reference vector, students t (multivariate) 157
retrieving the state of a generator 75

S

saving the state of a generator 75
seed, definition 74
size of integer arguments 15
sparse BLAS 19
students t distribution 113
students t distribution (multivariate) 149
students t distribution, multivariate, using reference vector 153

T

t distribution 113
triangular distribution 115

U

uniform distribution (continuous) 117
uniform distribution (discrete) 133
univariate distribution, beta 95
univariate distribution, binomial 123
univariate distribution, binomial, using reference vector 137
univariate distribution, cauchy 97
univariate distribution, chi-squared 99
univariate distribution, exponential 101
univariate distribution, f 103
univariate distribution, fisher's variance ratio . . 103
univariate distribution, gamma 105, 107
univariate distribution, geometric 125
univariate distribution, geometric, using reference vector 139
univariate distribution, hypergeometric 127
univariate distribution, hypergeometric, using reference vector 141
univariate distribution, logistic 109
univariate distribution, lognormal 111
univariate distribution, negative binomial 129
univariate distribution, negative binomial, using reference vector 143
univariate distribution, normal 107
univariate distribution, poisson 131
univariate distribution, poisson, using reference vector 145
univariate distribution, students t 113
univariate distribution, t 113
univariate distribution, triangular 115
univariate distribution, uniform (continuous) . . 117
univariate distribution, uniform (discrete) 133
univariate distribution, von mises 119
univariate distribution, weibull 121
user supplied generators 84

V

vector math functions 181
von mises distribution 119

W

weak aliases 162
weibull distribution 121
Wichmann-Hill base generator 82
Wichmann-Hill, multiple streams 89

Routine Index

_

__vrd2_cos ... 181
__vrd2_exp ... 186
__vrd2_log ... 192
__vrd2_log10 ... 198
__vrd2_log2 ... 204
__vrd2_sin ... 217
__vrd2_sincos ... 222
__vrd4_cos ... 182
__vrd4_exp ... 187
__vrd4_log ... 193
__vrd4_log10 ... 199
__vrd4_log2 ... 205
__vrd4_sin ... 218
__vrs4_cosf ... 184
__vrs4_expf ... 189
__vrs4_log10f ... 201
__vrs4_log2f ... 207
__vrs4_logf ... 195
__vrs4_powf ... 210
__vrs4_powxf ... 213
__vrs4_sincosf ... 225
__vrs4_sinf ... 220
__vrs8_expf ... 190
__vrs8_log10f ... 202
__vrs8_log2f ... 208
__vrs8_logf ... 196

A

acmlinfo ... 16
ACMLINFO ... 16
acmlversion ... 16
ACMLVERSION ... 16

C

CFFT1D ... 29
CFFT1DX ... 32
CFFT1M ... 37
CFFT1MX ... 41
CFFT2D ... 45
CFFT2DX ... 49
CFFT3D ... 54
CFFT3DX ... 57
CFFT3DY ... 62
CSFFT ... 71
CSFFTM ... 73

D

DRANDBETA ... 95
DRANDBINOMIAL ... 123
DRANDBINOMIALREFERENCE ... 137
DRANDBLUMBLUMSHUB ... 81
DRANDCAUCHY ... 97
DRANDCHISQUARED ... 99
DRANDDISCRETEUNIFORM ... 133
DRANDEXPONENTIAL ... 101
DRANDF ... 103
DRANDGAMMA ... 105
DRANDGAUSSIAN ... 107
DRANDGENERALDISCRETE ... 135
DRANDGEOMETRIC ... 125
DRANDGEOMETRICREFERENCE ... 139
DRANDHYPERGEOMETRIC ... 127
DRANDHYPERGEOMETRICREFERENCE ... 141
DRANDINITIALIZE ... 76
DRANDINITIALIZEBBS ... 79
DRANDINITIALIZEUSER ... 85
DRANDLEAPFROG ... 93
DRANDLOGISTIC ... 109
DRANDLOGNORMAL ... 111
DRANDMULTINOMIAL ... 159
DRANDMULTINORMAL ... 147
DRANDMULTINORMALR ... 151
DRANDMULTINORMALREFERENCE ... 155
DRANDMULTISTUDENTSREFERENCE ... 157
DRANDMULTISTUDENTST ... 149
DRANDMULTISTUDENTSTR ... 153
DRANDNEGATIVEBINOMIAL ... 129
DRANDNEGATIVEBINOMIALREFERENCE ... 143
DRANDPOISSON ... 131
DRANDPOISSONREFERENCE ... 145
DRANDSKIPAHEAD ... 90
DRANDSTUDENTST ... 113
DRANDTRIANGULAR ... 115
DRANDUNIFORM ... 117
DRANDVONMISES ... 119
DRANDWEIBULL ... 121
DZFFT ... 66
DZFFTM ... 68

F

fastcos ... 163
fastcosf ... 164
fastexp ... 165
fastexpf ... 166
fastlog ... 167
fastlog10 ... 169
fastlog10f ... 170
fastlog2 ... 171
fastlog2f ... 172
fastlogf ... 168
fastpow ... 173
fastpowf ... 175
fastsin ... 177
fastsincos ... 179
fastsincosf ... 180
fastsinf ... 178

I

ILAENVSET 21

S

SCFFT 67
SCFFTM 69
SRANDBINOMIALREFERENCE 137
SRANDCHISQUARED 99
SRANDGEOMETRICREFERENCE 139
SRANDBETA 95
SRANDBINOMIAL 123
SRANDBLUMBLUMSHUB 81
SRANDCAUCHY 97
SRANDDISCRETEUNIFORM 133
SRANDEXPONENTIAL 101
SRANDF 103
SRANDGAMMA 105
SRANDGAUSSIAN 107
SRANDGENERALDISCRETE 135
SRANDGEOMETRIC 125
SRANDHYPERGEOMETRIC 127
SRANDHYPERGEOMETRICREFERENCE 141
SRANDINITIALIZE 76
SRANDINITIALIZEBBS 79
SRANDINITIALIZEUSER 85
SRANDLEAPFROG 93
SRANDLOGISTIC 109
SRANDLOGNORMAL 111
SRANDMULTINOMIAL 159
SRANDMULTINORMAL 147
SRANDMULTINORMALR 151
SRANDMULTINORMALREFERENCE 155
SRANDMULTISTUDENTST 149
SRANDMULTISTUDENTSTR 153
SRANDMULTISTUDENTSTREFERENCE 157
SRANDNEGATIVEBINOMIAL 129
SRANDNEGATIVEBINOMIALREFERENCE 143
SRANDPOISSON 131
SRANDPOISSONREFERENCE 145
SRANDSKIPAHEAD 90
SRANDSTUDENTST 113
SRANDTRIANGULAR 115
SRANDUNIFORM 117
SRANDVONMISES 119
SRANDWEIBULL 121

U

UGEN 88
UINI 87

V

vrda_cos 183
vrda_exp 188
vrda_log 194
vrda_log10 200
vrda_log2 206
vrda_sin 219
vrda_sincos 223
vrsa_cosf 185
vrsa_expf 191
vrsa_log10f 203
vrsa_log2f 209
vrsa_logf 197
vrsa_powf 211, 215
vrsa_sincosf 226
vrsa_sinf 221

Z

ZDFFT 70
ZDFFTM 72
ZFFT1D 28
ZFFT1DX 30
ZFFT1M 35
ZFFT1MX 39
ZFFT2D 44
ZFFT2DX 46
ZFFT3D 53
ZFFT3DX 55
ZFFT3DY 59