AMD Core Math Library (ACML)

Version 2.1.0

Short Contents

Table of Contents

1 Introduction

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

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

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 11 and see Chapter 4 [LAPACK], page 12).

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 13).

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

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

2 General Information

2.1 Determining the best ACML version for your system

ACML comes in versions for 64-bit 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 compatible compilers (compilers that are interperable with the GNU compilers) installed?
- Do you have the PGI compilers (pgf77/pgf90/pgcc) 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 or SSE2)?

Under a Linux operating system, one 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 cannot determine whether or not your machine handles SSE or SSE2 instructions, you may prefer to assume that it does not. 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.

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	`acml2.1/gnu64`
”	PGI pgf77/pgf90/pgcc	`acml2.1/pgi64`
Multi processor	PGI pgf77/pgf90/pgcc	`acml2.1/pgi64_mp`

Linux 32-bit

Number of processors	*Compilers*	*SSE supported*	*ACML install directory*
Single	GNU g77/gcc or compatible	SSE and SSE2	`acml2.1/gnu32`
”	”	SSE but no SSE2	`acml2.1/gnu32_nosse2`
”	”	No SSE or SSE2	`acml2.1/gnu32_nosse`
”	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.1/pgi32`
”	”	SSE but no SSE2	`acml2.1/pgi32_nosse2`
”	”	No SSE or SSE2	`acml2.1/pgi32_nosse`
Multiple	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.1/pgi32_mp`

Microsoft Windows 64-bit

Number of processors	*Compilers*	*ACML install directory*
Single processor	PGI pgf77/pgf90/pgcc	`acml2.1/win64`

Microsoft Windows 32-bit

Number of processors	*Compilers*	*SSE supported*	*ACML install directory*
Single	GNU g77/gcc	SSE and SSE2	`acml2.1/gnu32`
”	”	SSE but no SSE2	`acml2.1/gnu32_nosse2`
”	”	No SSE or SSE2	`acml2.1/gnu32_nosse`
”	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.1/pgi32`
”	”	SSE but no SSE2	`acml2.1/pgi32_nosse2`
”	”	No SSE or SSE2	`acml2.1/pgi32_nosse`
Multiple	PGI pgf77/pgf90/pgcc	SSE and SSE2	`acml2.1/pgi32_mp`

2.2 Accessing the Library (Linux)

2.2.1 Accessing the Library under Linux using GNU g77/gcc

If the 64-bit g77 version of ACML was installed in the default directory, /opt/acml2.1/gnu64, then the command:

```
g77 -m64 driver.f -L/opt/acml2.1/gnu64 -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/acml2.1/gnu64 -static -lacml
```

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

```
g77 -m64 driver.f /opt/acml2.1/gnu64/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/acml2.1/gnu64
```

where it is assumed that libacml.so was installed in the directory /opt/acml2.1/gnu64 (see the man page for ld(1) for more information about `LD_LIBRARY_PATH`.).

The command

```
g77 -m32 driver.f -L/opt/acml2.1/gnu32 -lacml
```

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

The command

```
gcc -m64 -I/opt/acml2.1/include driver.c
    -L/opt/acml2.1/gnu64 -lacml -lg2c
```

will compile and link a 64-bit C program with a 64-bit ACML, using the switch "-I/opt/acml2.1/include" to tell the compiler to search directory /opt/acml2.1/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 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/acml2.1/pgi64 -lacml
pgf77 -tp=k8-32 -Mcache_align driver.f -L/opt/acml2.1/pgi32 -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/acml2.1/pgi64_mp -lacml
pgf77 -tp=k8-32 -mp -Mcache_align driver.f
      -L/opt/acml2.1/pgi32_mp -lacml
```

Note that the location of the ACML is now specified as pgi64_mp or pgi32_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/acml2.1/include driver.c
pgcc -tp=k8-64 -mp -Mcache_align driver.o
      -L/opt/acml2.1/pgi64_mp -lacml -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 switch "-I/opt/acml2.1/include" tells the C compiler to search directory /opt/acml2.1/include for the ACML C header file acml.h, which should be included by driver.c. 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.

The 32-bit ACML libraries come in several versions, applicable to hardware with or without SSE instructions (Streaming SIMD Extensions), and it is important to link to a library that is appropriate to your hardware. The variant library versions are distinguished by being in directories with different names.

2.2.3 Accessing the Library under Linux using compilers other than GNU or PGI

It may be possible to link to the g77/gcc versions of ACML using other compilers, if they are compatible with g77/gcc. An important thing to note is that you will need to link in required compiler run time libraries. An example using the 32-bit Intel FORTRAN compiler *ifc* might look like this:

```
ifc driver.f -L/opt/acml2.1/gnu32 -lacml /usr/lib/libg2c.so
```

where /usr/lib/libg2c.so is required to resolve g77 compiler run-time library symbols.

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\acml2.1\gnu32\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\acml2.1\gnu32\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\acml2.1\gnu32";%PATH%
```

where it was assumed that libacml.dll was installed in the directory "c:\Program Files\AMD\acml2.1\gnu32". Alternatively, the PATH environment variable may be set in the system category of the Windows control panel.

The command

```
gcc "-Ic:\Program Files\AMD\acml2.1\include" driver.c
    "c:\Program Files\AMD\acml2.1\gnu32\libacml.a" -lg2c
```

will compile and link a C program with ACML. The switch "-Ic:\Program Files\AMD\acml2.1\include" tells the compiler to search directory "c:\Program Files\AMD\acml2.1\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/pgcc

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

```
pgf77 -Mcache_align driver.f
      "c:\Program Files\AMD\acml2.1\pgi32\libacml.a"
```

or

```
pgcc -c "-Ic:\Program Files\AMD\acml2.1\include"
     -Mcache_align driver.c
pgcc -Mcache_align driver.o
      "c:\Program Files\AMD\acml2.1\pgi32\libacml.a"
      -lpgftnrtl -lpgsse1 -lpgsse2 -lm
```

Note that in the example we link the program with -lpgftnrtl -lpgsse1 -lpgsse2 -lm so that required PGI run-time libraries are located.

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 -mp -Mcache_align driver.f
      "c:\Program Files\AMD\acml2.1\pgi32\libacml.a"
```

Note that the location of the ACML is now specified as pgi32_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 -mp flag is also required if you compile and link a C program to the pgi32_mp libraries.

2.3.3 Accessing the Library under 64-bit Windows

Under 64-bit versions of Windows, ACML 2.1 comes only as a static (.LIB) library for single processor machines (the library can be used on an SMP machine but will not take advantage of more than one processor). The compiler can be used with the PGI compilers pgf77/pgf90/pgcc or with the Microsoft C compiler, though with the latter compiler it is necessary also to link with PGI run-time libraries.

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

```
pgf77 driver.f c:/Program Files/AMD/acml2.1/win64/libacml.lib
```

or, for a C program,

```
pgcc driver.c -Ic:/Program Files/AMD/acml2.1/include
     c:/Program Files/AMD/acml2.1/win64/libacml.lib
     -lpgftnrtl -lm
```

Note that in the C example we link the program with -lpgftnrtl -lm so that required PGI run-time libraries are located.

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

```
cl driver.c -Ic:/Program Files/AMD/acml2.1/include
     c:/Program Files/AMD/acml2.1/win64/libacml.lib
     c:/usr/pgi/win64/1.0/lib/libpgftnrtl.lib
     c:/usr/pgi/win64/1.0/lib/libpgc.lib
```

The references to libpgftnrtl.lib and libpgc.lib must point at the location of an installed copy of the PGI compilers.

2.4 ACML FORTRAN and C interfaces

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

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

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

- The FORTRAN interface names are appended by an underscore.
- 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 2.7 [Examples], page 9) make use of these types.

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

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

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

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

FORTRAN:

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

C:

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

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

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

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

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

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

2.5 Library Version and Build Information

This document is applicable to version 2.1 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]

2.6 Library Documentation

The /Doc subdirectory of the top ACML installation directory, (e.g. /opt/acml2.1/Doc under Linux, or c:\Program Files\AMD\acml2.1\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/acml2.1/Doc

% info acml
```

or simply by using the full name of the file:

```
% info /opt/acml2.1/Doc/acml.info
```

2.7 Example programs calling ACML

The /examples subdirectory of the top ACML installation directory (e.g. /opt/acml2.1/gnu64/examples under Linux, or c:\Program Files\AMD\acml2.1\gnu32\examples under Windows), 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. 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.

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

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 avilable 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 PGI compiler versions of ACML.

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

5 Fast Fourier Transforms (FFTs)

5.1 Introduction to FFTs

5.1.1 Data Types and Storage

There are two main types of DFTs:

- 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`.

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 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 of length $m * n$ as m end-to-end sequences; again, the results of the m FFTs are returned in the original array.

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.

A two dimensional array of complex data, with n rows and m columns is stored in the same order as a set of m sequences of length n (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.

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 tranformed data, and the sign of $\pm$ determines the direction of the transform.

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 tranformed 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).

5.1.2 Efficiency

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

5.2 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.

`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 initializations (specific to the value of *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.

`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 initializations (specific to the value of 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 : initializations and a forward transform are performed.

MODE=2 : initializations and a backward transform are performed.

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)
```

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.

`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; 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 : initializations and forward transforms are performed.

MODE=2 : initializations and backward transforms are performed.

`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; 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 : initializations and forward transforms are performed.

MODE=2 : initializations and backward transforms are performed.

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)
```

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.

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 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 **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)
```

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.

`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)
```

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 13. 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 initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*= 1.

MODE=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to `DZFFT`.

MODE=2 : initializations and a real transform are performed.

`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(2*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
      CALLD ZDFFT(1,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 initializations (specific to the value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*= 1.

MODE=1 : a real transform is performed. Initializations are assumed to have been performed by a prior call to SCFFT.

MODE=2 : initializations and a real transform are performed.

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(2*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
      CALLD CSFFT(1,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(2*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)
```

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(2*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 value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*=1.

MODE=1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to ZDFFT.

MODE=2 : initializations and transform are performed.

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(2*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(1,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 value of *N*) are performed; this is usually followed by calls to the same routine with *MODE*=1.

MODE=1 : a forward transform is performed. Initializations are assumed to have been performed by a prior call to CSFFT.

MODE=2 : initializations and transform are performed.

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(2*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(1,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(2*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(2*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 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).

Routine Index