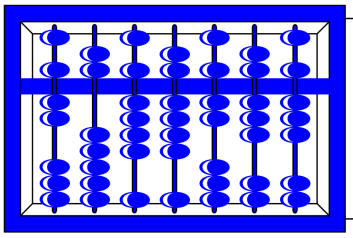

NASoftware Limited

Incorporating InfoSAR

FFTW
Reference Manual

FFTW/Ref [3.20.14]

Release 3.20.14
February 2021

This document is derived from the FFTW API 3.3.3 specification document written by Steven G. Johnson and available from `http://www.fftw.org/`.

Contents

Chapter 1. Introduction

This document describes the functions available in the NAS FFTW low level optimised Fast Fourier Transform (FFT) library. The library is optimized to perform the FFT algorithm on vectors and matrices in single precision. The functionality of the library can be summarized as follows:

- Data shape:
  - 1D Vectors
  - 2D Matrices
- FFT types:
  - Interleaved complex to complex
  - Split complex to complex
  - Interleaved complex to real
  - Split complex to real
  - Real to interleaved complex
  - Real to split complex
- FFT shape:
  - 1D FFT
  - Multiple 1D FFTs
  - 2D FFT
  - Multiple 2D FFTs
- In-Place or Out-of-Place operations.
- Forward or Backward FFTs.
- The data type can be strided or compact (stride=1).

The original versions of the FFTW library was implemeted by the Fastest Fourier Transform in the West (FFTW) organization and Massachusetts Institute of Technology who designed the API of the library. The website to the original FFTW library can be found at:

http://www.fftw.org/

The documentation to the original FFTW API can be found at:

http://www.fftw.org/fftw3.pdf

NAS have produced a similar API to the above specification which is described by this document. The main difference is that the NAS library does not support 3D FFTs and above. This document describes the NAS FFTW library and is organized into the following Chapters:

Chapter 2. Overview of the FFTW Library

This section gives an overview of the definitions used by the FFTW library, the functions contained in it and a brief description of the function calling sequence.

2.1 Structures and types

The following structures and types are used by the FFTW library and are declared within the include file **fftw.h**:

2.1.1 Float data.

The following definition (**fftw_scalar_d**) is used to define a double-precision floating point type :

```
typedef double fftw_scalar_d;
```

2.1.2 Interleaved complex data.

The user may use one of two definitions (**fftw_cscalar_d** and **fftw_complex**) to define a interleaved complex type:

Interleaved structure definition.

```
typedef struct
{
   fftw_scalar_d r; /* Real complex part.      */
   fftw_scalar_d i; /* Imaginary complex part. */
} fftw_cscalar_d;
```

Interleaved array definition.

```
typedef double fftw_complex[2];
```

2.1.3 FFTW data plans.

Before any FFT processing function can be called a FFTW plan must be setup containing algorithm information and data such as FFT weights. The plan data is stored in a opaque structure called **fftw_fftw_d**. The plan creation functions return an object of type **fftw_plan** and the execution function take a variable of type **fftw_plan** as the first function argument. The object **fftw_plan** is defined as a pointer to **fftw_fftw_d**:

```
typedef fftw_fftw_d *fftw_plan;
```

2.1.4 FFTW Guru data stucture.

Most FFTW plan creation functions set up a plan to process data of a compact type. This means that the elements in a complex buffer or a real buffer are in adjacent cells and the data stride is 1. However, with the FFTW Guru plans the data is allowed to be strided. The stride information and the data length is given to the plan in a structure called **fftw_iodim** as defined as follows:

```
typedef struct
{
  int n;    /* Data length.   */
  int is;   /* Input stride.  */
  int os;a  /* Output stride. */
} fftw_iodim;
```

2.2 Symbols and Flags

The following symbolic constants are defined within the **fftw.h** include file:

2.2.1 Boolean Flags.

FFTW_TRUE	FFTW true flag.
FFTW_FALSE	FFTW false flag.

2.2.2 FFT Direction Flags.

FFTW_FORWARD	Forward FFT flag.
FFTW_BACKWARD	Backward FFT flag.

2.2.3 FFTW Plan Flags.

FFTW_MEASURE	Maximum accurancy flag.
FFTW_DESTROY_INPUT	Input buffers are not required again after call.
FFTW_UNALIGNED	FFTW should expect unaligned data.
FFTW_CONSERVE_MEMORY	Free up memory as a priority.
FFTW_EXHAUSTIVE	Consider all FFT algorithms.
FFTW_PRESERVE_INPUT	Opposite of FFTW _DESTROY_INPUT.
FFTW_PATIENT	Processing speed is a low priority.
FFTW_ESTIMATE	Low accuracy flag.

2.3 Function Names

The functions in the FFTW library adhere to the following naming convensions:

1. All FFTW function names are prefixed with text "**fftw_**";
2. All FFTW structures and type are prefixed with the text "**fftw_**";
3. All FFTW definitions are prefixed with the text "**FFTW**";
4. FFTW Plan function names are prefixed with the text "**fftw_plan**";
5. FFTW Execute Functions are prefixed with the text "**fftw_execute**";
6. All FFTW plan functions return a object of type **fftw_plan**.
7. The first argument of every FFTW Execute function takes an argument of type **fftw_plan**.

2.4 Program Structure.

Any program calling the FFTW library must contain the following parts:

- **Declare required FFTW objects.** eg:

```
fftw_complex *in_buffer, *out_buffer;
int length = 1024;
fftw_plan plan;
```

- **Malloc aligned memory.** The support function **fftw_malloc** guarantees that the allocated memory is fully alaigned for the target hardware:

```
in_buffer = (fftw_complex*)
   fftw_malloc(length*sizeof(fftw_complex));
out_buffer = (fftw_complex*)
   fftw_malloc(length*sizeof(fftw_complex));
if(in_buffer==NULL || out_buffer==NULL) return 0;
```

- **Create an FFTW plan.** eg:

```
plan = fftw_plan_dft_1d(length, in_buffer, out_buffer, FFTW_FORWARD,
                        FFTW_ESTIMATE);
```

- **Call FFTW processing function.**

```
fftw_execute(plan);
```

- **Free up plan and buffers**, before the end of the application. eg

```
fftw_destroy_plan(plan);
fftw_free(out_buffer);
fftw_free(in_buffer);
```

To FFT a data buffer with the FFTW library requires a plan to be created with a call to a FFTW plan creation routine and a processing function to be called by a call to a FFTW Execute routine. Which Plan routine and which Execute routine to call in the library depends on the type of FFT required, the type of data and the locality of the data. The following sections show which combinations of routines can be called for different FFT types.

Multiple FFTs can be specified when using a Guru plan. Non-Guru plans setup the execute routines to do a single FFT at a time. The Guru plans also allow the user to specify a data stride. This means that the data can either be in a compact form (stride=1) or the elements can be strided by more than one. Non-Guru plans only process compact data. The following tables show which routines support multiple and strided options for each FFT type.

The function **fftw_execute** processes data buffers attached to the FFT plan. These data buffers are given to the plan creation function as function arguments. All other Execute functions process data buffers passed to the Execute routine in the form of function arguments. If these arguments are set to NULL then the function will process the data buffers attached to the FFT plan instead.

2.5 Complex-to-complex FFTs

2.5.1 1D Interleaved Complex-to-complex FFTs

Table 1: 1D Interleaved Complex-to-complex Plan Functions.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_dft_1d	No	No	Create 1D plan for compact data.
fftw_plan_dft	No	No	Create plan of any rank for compact data.
fftw_plan_guru_dft	Yes	Yes	Create plan of any rank and stride.

Table 2: 1D Interleaved Complex-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	No args	Yes	Processes data buffers attached to plan.
fftw_execute_dft	Yes	Optional	Processes data buffers passed by arguments.

2.5.2 1D Split Complex-to-complex FFTs

Table 3: 1D Split Complex-to-complex Plan.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_guru_split_dft	Yes	Yes	Create plan of any rank and stride.

Table 4: 1D Split Complex-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_split_dft	Yes	Optional	Processes data buffers passed by arguments.

2.5.3 2D Interleaved Complex-to-complex FFTs

Table 5: 2D Interleaved Complex-to-complex Plans.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_dft_2d	No	No	Create 2D plan for compact data.
fftw_plan_dft	No	No	Create plan of any rank for compact data.
fftw_plan_guru_dft	Yes	Yes	Create plan of any rank and stride.

Table 6: 2D Interleaved Complex-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_dft	Yes	Optional	Processes data buffers passed by arguments.

2.5.4 2D Split Complex-to-complex FFTs

Table 7: 2D Split Complex-to-complex Plan.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_guru_split_dft	Yes	Yes	Create plan of any rank and stride.

Table 8: 2D Split Complex-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_split_dft	Yes	Optional	Processes data buffers passed by arguments.

2.6 Complex-to-real FFTs

2.6.1 1D Interleaved Complex-to-real FFTs

Table 9: Interleaved Complex-to-real Plans.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_dft_c2r_1d	No	No	Create 1D plan for compact data.
fftw_plan_dft_c2r	No	No	Create plan of any rank for compact data.
fftw_plan_guru_dft_c2r	Yes	Yes	Create plan of any rank and stride.

Table 10: Interleaved Complex-to-Real Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_dft_c2r	Yes	Optional	Processes data buffers passed by arguments.

2.6.2 1D Split Complex-to-real FFTs

Table 11: Split Complex-to-real Plan.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_guru_split_dft_c2r	Yes	Yes	Create plan of any rank and stride.

Table 12: Split Complex-to-Real Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_split_dft_c2r	Yes	Optional	Processes data buffers passed by arguments.

2.6.3 2D Interleaved Complex-to-real FFTs

Table 13: Interleaved Complex-to-real Plans.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_dft_c2r_2d	No	No	Create 2D plan for compact data.
fftw_plan_dft_c2r	No	No	Create plan of any rank for compact data.
fftw_plan_guru_dft_c2r	Yes	Yes	Create plan of any rank and stride.

Table 14: Interleaved Complex-to-Real Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_dft_c2r	Yes	Optional	Processes data buffers passed by arguments.

2.6.4 2D Split Complex-to-real FFTs

Table 15: Split Complex-to-real Plan.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_guru_split_dft_c2r	Yes	Yes	Create plan of any rank and stride.

Table 16: Interleaved Complex-to-Real Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_split_dft_c2r	Yes	Optional	Processes data buffers passed by arguments.

2.7 Real-to-complex FFTs

2.7.1 1D Interleaved Real-to-complex FFTs

Table 17: Interleaved Real-to-complex Plans.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_dft_r2c_1d	No	No	Create 1D plan for compact data.
fftw_plan_dft_r2c	No	No	Create plan of any rank for compact data.
fftw_plan_guru_dft_r2c	Yes	Yes	Create plan of any rank and stride.

Table 18: Interleaved Real-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_dft_r2c	Yes	Optional	Processes data buffers passed by arguments.

2.7.2 1D Split Real-to-complex FFTs

Table 19: Split Real-to-complex Plan.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_guru_split_dft_r2c	Yes	Yes	Create plan of any rank and stride.

Table 20: Interleaved Real-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	No args	Yes	Processes data buffers attached to plan.
fftw_execute_split_dft_r2c	Yes	Optional	Processes data buffers passed by arguments.

2.7.3 2D Interleaved Real-to-complex FFTs

Table 21: Interleaved Real-to-complex Plans.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_dft_r2c_2d	No	No	Create 2D plan for compact data.
fftw_plan_dft_r2c	No	No	Create plan of any rank for compact data.
fftw_plan_guru_dft_r2c	Yes	Yes	Create plan of any rank and stride.

Table 22: Interleaved Real-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_dft_r2c	Yes	Optional	Processes data buffers passed by arguments.

2.7.4 2D Split Real-to-complex FFTs

Table 23: Split Real-to-complex Plan.

Plan Function	Multiple FFT Support	Strided Data Support	Description
fftw_plan_guru_split_dft_r2c	Yes	Yes	Create plan of any rank and stride.

Table 24: Interleaved Real-to-complex Processing Functions.

Execute Function	Process Execute Buffers	Process Plan Buffers	Description
fftw_execute	None	Yes	Processes data buffers attached to plan.
fftw_execute_dft_r2c	Yes	Optional	Processes data buffers passed by arguments.

2.8 Complex Split Functions

Complex split data represents complex data using two separate data buffers, one containing real data, the other containing imaginary data. This is in contrast to complex interleaved data, where each complex sample is represented by a pair of double precision floating point values (real, imaginary). Many operations on complex split data are faster than operations on interleaved data. This is a consequence of machine instructions.

Chapter 3. Getting the Best Performance

This section is a short guide for programmers using the FFTW Library. It contains explanations of library behaviour, and tips on selecting the right storage options for your data to increase performance.

3.1 Memory Alignment

The efficiency of many operations is improved if data within memory is correctly aligned on certain word boundaries. Vectors and matrices can be loaded and stored faster if they are vector aligned. The following table gives the vector alignment and minimum vector length for float data:

Technology	Vector Aligment
SSE	16
AVX	32
AltiVec	16

Alignment can be controlled using a function such as `fftwf_malloc`. Some operating systems (*e.g.* Apple's OSX) automatically align all memory to a 16-byte boundary so `fftwf_malloc` is not needed.

3.2 Vector/Matrix Format

When available, vector and matrix calculations are done using single instruction, multiple data (SIMD) instructions to process several elements simultaneously. This imposes a minimum vector length given in the table below:

Technology Vector Length (floats)	minimum
SSE	4
AVX	8
AltiVec	4

For best performance all input and output vectors should:

- have a stride of 1

  Vectors and matrices can be loaded and stored much quicker when they are contiguous in memory. The library includes special optimisations for a stride length of 2 (which was added for interleaved complex numbers), but all other non-unit strides will be significantly slower than a stride of 1 and, in many cases, almost as slow as unvectorised scalar code. Note that, a stride of -1 will also be significantly slower than a stride of $+1$.

- be vector aligned

- have length greater than or equal to the vector length

  The vector unit works on arrays of the vector length so no speed up is gained by using the library on vectors of length less than this.

- have row (row major matrices) or column (column major matrices) length divisible by the vector length

  For a row major matrix: if the row length of a matrix is not divisible by the vector length then the alignment of the first element of each row will change for each row/column. For optimal performance the first element of each row should be vector aligned.

  The same rule applies to columns in column major matrices.

- have a length divisible by the vector length

  Any elements at the end of the vector which cannot be dealt with by the vector unit must be dealt with in normal scalar code, which will decrease the performance. The decrease in performance becomes less important for longer vectors.

3.3 Error Checking and Debugging

Two versions of the FFTW library are provided: a performance version and a development version. The development version of the library (signified by a 'D' in the library's name) contains full error checking and should always be used when developing and debugging applications.

A few library functions return status information: always check the return code of those that do.

The performance version of the library contains no error checking, and consequently runs faster than the development library. The performance library should only be used with applications that have been run successfully with the development version of the

library.

When timing code, the performance version of the library should be used.

Note: the performance version of the library reads in data before it knows how much will be used and as a result often reads more data than is needed. This is not a problem, except when using memory checkers such as Electric Fence which object to this behaviour. The development library only reads in the data it intends to use and so is safe to use with memory checkers.

3.4 FFT Functions

To get the best performance from an FFT, a vector must have length a multiple of the numbers 2, 4, 8, and 3 only. If a vector length is not a multiple of these numbers, a DFT may be done, which is considerably slower than an FFT.

Factors of 3 should be avoided if possible. An FFT will only be done for factors of 3 if the length also has a factor of 16, otherwise a DFT is done.

When doing large FFT's, optimal routines have been developed for the lengths: 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, and 65536. These lengths should be much quicker than lengths of similar magnitude.

In-place FFT's are normally faster than out-of-place FFT's.

FFT's are fastest with a scale factor of 1. However, if you need to use a different scale factor, it is better to let the FFT routine do the scaling rather than to do it yourself.

The internal FFT routines only work on vector aligned data with a stride of 1. If vectors are used which do not match these restrictions an internal copy of the vector will be made. This is an important consideration when using large vectors. Also, if complex vectors are not stored split an internal copy will be made.

3.5 Controlling the Number of Threads

The Threaded FFTW library is multithreaded and will take advantage of multiple cores on the processor invoking it. Utilizing multiple threads is automatic:

- The maximum number of threads used is set when any FFTW function is called for the first time.
- The maximum number running at any one time is also set at that point. If a threaded routine is called with (say) 4 threads and we have hit this maximum

number running then four of them are shut down before the new function is executed.

- The number of threads invoked when a routine is called, is decided by that routine by reference to the data (vector or matrix) size specified in the call, to provide the best performance for that call.

It is possible to change the maximum number of threads used.

1. A threaded, and a non-threaded ("serial") version of the library are provided. If you wish to only ever use one thread in a library call, use the serial version of the library.

2. The maximum number of threads used for a specific function call, and the maximum number kept running at any one time, can be changed by a call to the routine `Thread_SetParams` with arguments `num_threads` and `max_num_running`. This call, if used, *must* be made before any FFTW function is called.

3. When calling the routine `Thread_SetParams` the value of `max_num_running` must be greater or equal to 3*`num_threads`. If the user enters a smaller value than this in their `Thread_SetParams` function call then the function will set the value of `max_num_running` to 3*`num_threads`.

   If no call to `Thread_SetParams` is made, the library default values will be utilized.

Chapter 4. FFTW Initialization Functions

The NAS FFTW library MUST be initialised and finalised by a call to the following two functions. The initialisation function must be called before any other FFTW function. The finalization function must be called after every other FFTW function.

4.1 fftw_init

FFTW initialization function.

Function Definitions

```
int fftw_init(
                    void *ptr);
```

Parameters

Argument	I/O	Description
ptr	input	Pointer to a void object type.

Return Value

This integer function returns 0 on success.

Description

This function initializes the FFTW library including the vector constants and other global data. The routine must be called once before any other function in the library.

Restrictions and Notes

1. The function also initializes the threads pool within the FFTW threaded library. This means a call to function fftw_init_threads is no longer required. However, the function fftw_init_threads has been provided with the library for backward compatability.
2. The function should only be called once and before any other FFTW function.

Example code

Example code of **fftw_init** can be found in all the code examples Chapter 10.

4.2 fftw_finalize

FFTW finalization function.

Function Definitions

```
int fftw_finalize(
                    void *ptr);
```

Parameters

Argument	I/O	Description
ptr	input	Pointer to a void object type.

Return Value

This integer function returns 0 on success.

Description

This function gives back memory to the computer system that the FFTW library has captured during execution. The routine must be called once at the end of a program after all FFTW library calls have been made.

Restrictions and Notes

1. The function also finalizes the threads pool within the FFTW threaded library. This means a call to function fftw_cleanup_threads is no longer required. However, the function fftw_cleanup_threads has been provided with the library for backward compatability.
2. The function should only be called once after all FFTW routine calls have been made.

Example code

Example code of **fftw_finalize** can be found in all the code examples contained in Chapter 10.

Chapter 5. FFTW Memory Support Functions

All memory allocation to buffers processed by the FFTW library should be handled with the following two support functions. These functions allocate fully aligned memory for a target computer system with a minimum data length size.

5.1 fftw_malloc

FFTW memory allocation function.

Function Definitions

```
void * fftw_malloc(
                    size_t n);
```

Parameters

Argument	I/O	Description
n	input	Number of bytes of memory required.

Return Value

Void pointer to the created block of memory.

Description

This function returns block of memory of the requested size. The returned block of memory is fully aligned for the target computer hardware.

Restrictions and Notes

1. All memory allocated through fftw_malloc should be returned by a call to fftw_free.
2. The FFT operations are significantly faster when the buffers are fully aligned for the target hardware.

Example code

Example code of **fftw_malloc** can be found in all the code examples contained in Chapter 10.

5.2 fftw_free

FFTW free memory allocation function.

Function Definitions

```
void fftw_free(
                    void *ptr);
```

Parameters

Argument	I/O	Description
ptr	input	Pointer to block of memory to be returned to the system.

Return Value

None: void function.

Description

This function returns the memory to the computer system that has been allocated through a call to fftw_malloc.

Restrictions and Notes

1. All memory allocated through fftw_malloc should be returned by a call to fftw_free.
2. The FFT operations are significantly faster when the buffers are fully ali gned for the target hardware.

Example code

Example code of **fftw_free** can be found in all the code examples contained in Chapter 10.

Chapter 6. FFTW Plan Functions

6.1 fftw_destroy_plan

Destroy FFTW plan function.

Function Definitions

```
void fftw_destroy_plan(
                    const fftw_plan plan);
```

Parameters

Argument	I/O	Description
plan	input/output	Generic FFTW plan to be destroyed.

Return Value

None: void function.

Description

This function destroys the given fft plan and returns the plan memory to the computer system. The function will destroy any plan created by any of the plan creation functions in this section. This includes 1D, multiple and 2D FFTs. It also includes all combinations of complex-to- complex, complex-to-real and real-to-complex operations.

Restrictions and Notes

1. A plan must have been initialized by a plan creation function before this function can be called.
2. All FFTW plans should be destroyed before the program terminates.

Example code

Example code of **fftw_destroy_plan** can be found in all the code examples contained in Chapter 10.

6.2 fftw_plan_dft

Multi-dim Interleaved Complex to Complex FFTW plan function.

Function Definitions

```
void fftw_plan_dft(
                     int            rank,
                     signed int    *length,
                     fftw_complex  *input_A,
                     fftw_complex  *output_R,
                     int            dir,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
length	input	Integer array of data lengths of length rank.
input_A	input	Interleaved complex input buffer.
output_R	output	Interleaved complex output buffer.
dir	input	Set to either FFTW_FORWARD; or FFTW_BACKWARD.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - 2D FFT;
- In-place or out-of-place operations.
- Forward or backward FFTs.

However, the transformation type must be:

- Complex-to-complex;
- Interleaved complex input and output buffers;
- Compact data elements;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. All data buffers in the plan must be of type compact, ie not strided.
3. The FFT operation may be an out-of-place or in-place operation depending on the buffer arguments.
4. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
5. The FFT operation is performed faster when the FFT lengths are a power of two.
6. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag will therefore be ignored.

Example code

Example code of **fftw_plan_dft** can be found in all the code examples contained in Section 10.3.1.

6.3 fftw_plan_dft_1d

1D Interleaved Complex to Complex FFTW plan function.

Function Definitions

```
void fftw_plan_dft_1d(
                    int            length,
                    fftw_complex  *input_A,
                    fftw_complex  *output_R,
                    int            dir,
                    unsigned int   flags);
```

Parameters

Argument	I/O	Description
length	input	Complex 1D FFT length
input_A	input	Interleaved complex input buffer.
output_R	output	Interleaved complex output buffer.
dir	input	Set to either FFTW_FORWARD; or FFTW_BACKWARD.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size.
- In-place or out-of-place operations.
- Forward or backward FFTs.

However, the transformation type must be:

- Complex-to-complex;
- Interleaved complex input and output buffers;
- Compact data elements;
- The data buffer must be 1D;

Restrictions and Notes

1. All data buffers in the plan must be of type compact, ie not strided.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_1d** can be found in all the code examples contained in Section 10.3.

6.4 fftw_plan_dft_2d

2D Interleaved Complex to Complex FFTW plan function.

Function Definitions

```
void fftw_plan_dft_2d(
                      int            rows,
                      int            cols,
                      fftw_complex  *input_A,
                      fftw_complex  *output_R,
                      int            dir,
                      unsigned int   flags);
```

Parameters

Argument	I/O	Description
rows	input	Matrix column length
cols	input	Matrix row length
input_A	input	Interleaved complex input buffer.
output_R	output	Interleaved complex output buffer.
dir	input	Set to either FFTW_FORWARD; or FFTW_BACKWARD.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size.
- In-place or out-of-place operations.
- Forward or backward FFTs.

However, the transformation type must be:

- Complex-to-complex;
- Interleaved complex input and output buffers;
- Compact data elements;
- The data buffer must be 2D;

Restrictions and Notes

1. All data buffers in the plan must be of type compact, ie not strided.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_2d** can be found in all the code examples contained in Section 10.3.3.

6.5 fftw_plan_guru_dft

Multi-dim Interleaved Complex to Complex Guru FFTW plan function.

Function Definitions

```
void fftw_plan_guru_dft(
                     int            rank,
                     fftw_iodim    *dims,
                     int            howmany,
                     fftw_iodim    *howmany_dims,
                     fftw_complex  *input_A,
                     fftw_complex  *output_R,
                     int            dir,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
dims	input	Structure defining data lengths and input/output strides.
howmany	input	howmany_dims array length.
howmany_dims	input	Structure defining how many FFTs and data block stride.
input_A	input	Interleaved complex input buffer.
output_R	output	Interleaved complex output buffer.
dir	input	Set to either FFTW_FORWARD; or FFTW_BACKWARD.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - Multiple 1D FFTs;
  - 2D FFT;
  - Multiple 2D FFTs;
- In-place or out-of-place operations.
- Forward or backward FFTs.
- The data type can be strided or compact. The stride is specified in the dims argument.

However, the transformation type must be:

- Complex-to-complex;
- Interleaved complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The FFT operation is performed faster when the data is compact. However, the user may specify strided data elements if it is a algorithm requirement.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library.

Example code

Example code of **fftw_plan_guru_dft** can be found in all the code examples contained in Section 10.3.2.

6.6 fftw_plan_guru_split_dft

Multi-dim Split Complex to Complex Guru FFTW plan function.

Function Definitions

```
void fftw_plan_guru_split_dft(
                     int            rank,
                     fftw_iodim    *dims,
                     int            howmany,
                     fftw_iodim    *howmany_dims,
                     double        *input_A_re,
                     double        *input_A_im,
                     double        *output_R_re,
                     double        *output_R_im,
                     int            dir,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
dims	input	Structure defining data lengths and input/output strides.
howmany	input	howmany_dims array length.
howmany_dims	input	Structure defining how many FFTs and data block stride.
input_A_re	input	Split complex real input buffer.
input_A_im	input	Split complex imaginary input buffer.
output_R_re	output	Split complex real output buffer.
output_R_im	output	Split complex imaginary output buffer.
dir	input	Set to either FFTW_FORWARD; or FFTW_BACKWARD.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - Multiple 1D FFTs;
  - 2D FFT;
  - Multiple 2D FFTs;

- In-place or out-of-place operations.
- Forward or backward FFTs.
- The data type can be strided or compact. The stride is specified in the di ms argument.

However, the transformation type must be:

- Complex-to-complex;
- Split complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The FFT operation is performed faster when the data is compact. However, t he user may specify strided data elements if it is a algorithm requirement.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library.

Example code

Example code of **fftw_plan_guru_split_dft** can be found in all the code examples contained in Section 10.4.

6.7 fftw_plan_dft_c2r

Multi-dim Interleaved Complex to Real FFTW plan function.

Function Definitions

```
void fftw_plan_dft_c2r(
                    int            rank,
                    signed int    *length,
                    fftw_complex  *input_A,
                    double        *output_R,
                    unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
length	input	Integer array of data lengths of length rank.
input_A	input	Interleaved complex input buffer.
output_R	output	Real output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - 2D FFT;
- In-place or out-of-place operations.

However, the transformation type must be:

- Complex-to-real;
- Interleaved complex input buffer;
- Compact data elements;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. All data buffers in the plan must be of type compact, ie not strided.
3. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
4. The FFT operation is performed faster when the FFT lengths are a power of two.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_c2r** can be found in all the code examples contained in Section 10.5.1.

6.8 fftw_plan_dft_c2r_1d

1D Interleaved Complex to Real FFTW plan function.

Function Definitions

```
void fftw_plan_dft_c2r_1d(
                     int            length,
                     fftw_complex  *input_A,
                     double        *output_R,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
length	input	Complex 1D FFT length
input_A	input	Interleaved complex input buffer.
output_R	output	Real output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size.
- In-place or out-of-place operations.

However, the transformation type must be:

- Complex-to-real;
- Interleaved complex input buffer;
- Compact data elements;
- The data buffer must be 1D;

Restrictions and Notes

1. All data buffers in the plan must be of type compact, ie not strided.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_c2r_1d** can be found in all the code examples contained in Section 10.5.

6.9 fftw_plan_dft_c2r_2d

2D Interleaved Complex to Real FFTW plan function.

Function Definitions

```
void fftw_plan_dft_c2r_2d(
                      int            rows,
                      int            cols,
                      fftw_complex  *input_A,
                      double        *output_R,
                      unsigned int   flags);
```

Parameters

Argument	I/O	Description
rows	input	Matrix column length
cols	input	Matrix row length
input_A	input	Interleaved complex input buffer.
output_R	output	Real output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size.

- In-place or out-of-place operations.

However, the transformation type must be:

- Complex-to-real;
- Interleaved complex input buffer;
- Compact data elements;
- The data buffer must be 2D;

Restrictions and Notes

1. All data buffers in the plan must be of type compact, ie not strided.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_c2r_2d** can be found in all the code examples contained in Section 10.5.3.

6.10 fftw_plan_guru_dft_c2r

Multi-dim Interleaved Complex to Real Guru FFTW plan function.

Function Definitions

```
void fftw_plan_guru_dft_c2r(
                     int            rank,
                     fftw_iodim    *dims,
                     int            howmany,
                     fftw_iodim    *howmany_dims,
                     fftw_complex  *input_A,
                     double        *output_R,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
dims	input	Structure defining data lengths and input/output strides.
howmany	input	howmany_dims array length.
howmany_dims	input	Structure defining how many FFTs and data block stride.
input_A	input	Interleaved complex input buffer.
output_R	output	Real output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - Multiple 1D FFTs;
  - 2D FFT;
  - Multiple 2D FFTs;
- In-place or out-of-place operations.
- The data type can be strided or compact. The stride is specified in the dims argument.

However, the transformation type must be:

- Complex-to-real;
- Interleaved complex input buffer;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The FFT operation is performed faster when the data is compact. However, t he user may specify strided data elements if it is a algorithm requirement.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library.

Example code

Example code of **fftw_plan_guru_dft_c2r** can be found in all the code examples contained in Section 10.7.2.

6.11 fftw_plan_guru_split_dft_c2r

Multi-dim Split Complex to Real Guru FFTW plan function.

Function Definitions

```
void fftw_plan_guru_split_dft_c2r(
                     int            rank,
                     fftw_iodim    *dims,
                     int            howmany,
                     fftw_iodim    *howmany_dims,
                     double        *input_A_re,
                     double        *input_A_im,
                     double        *output_R,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
dims	input	Structure defining data lengths and input/output strides.
howmany	input	howmany_dims array length.
howmany_dims	input	Structure defining how many FFTs and data block stride.
input_A_re	input	Split complex real input buffer.
input_A_im	input	Split complex imaginary input buffer.
output_R	output	Real output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - Multiple 1D FFTs;
  - 2D FFT;
  - Multiple 2D FFTs;
- In-place or out-of-place operations.
- The data type can be strided or compact. The stride is specified in the di ms argument.

However, the transformation type must be:

- Complex-to-real;
- Split complex input buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The FFT operation is performed faster when the data is compact. However, t he user may specify strided data elements if it is a algorithm requirement.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library.

Example code

Example code of **fftw_plan_guru_split_dft_c2r** can be found in all the code examples contained in Section 10.6.

6.12 fftw_plan_dft_r2c

Multi-dim Real to Interleaved Complex FFTW plan function.

Function Definitions

```
void fftw_plan_dft_r2c(
                    int            rank,
                    signed int    *length,
                    double        *input_A,
                    fftw_complex  *output_R,
                    unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
length	input	Integer array of data lengths of length rank.
input_A	input	Real input buffer.
output_R	output	Interleaved complex output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - 2D FFT;
- In-place or out-of-place operations.

However, the transformation type must be:

- Real-to-complex;
- Interleaved complex output buffer;
- Compact data elements;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. All data buffers in the plan must be of type compact, ie not strided.
3. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
4. The FFT operation is performed faster when the FFT lengths are a power of two.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_r2c** can be found in all the code examples contained in Section 10.7.1.

6.13 fftw_plan_dft_r2c_1d

1D Real to Interleaved Complex FFTW plan function.

Function Definitions

```
void fftw_plan_dft_r2c_1d(
                     int            length,
                     double        *input_A,
                     fftw_complex  *output_R,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
length	input	Complex 1D FFT length
input_A	input	Real input buffer.
output_R	output	Interleaved complex output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size.
- In-place or out-of-place operations.

However, the transformation type must be:

- Real-to-complex;
- Interleaved complex output buffer;
- Compact data elements;
- The data buffer must be 1D;

Restrictions and Notes

1. All data buffers in the plan must be of type compact, ie not strided.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_r2c_1d** can be found in all the code examples contained in Section 10.7.

6.14 fftw_plan_dft_r2c_2d

2D Real to Interleaved Complex FFTW plan function.

Function Definitions

```
void fftw_plan_dft_r2c_2d(
                      int            rows,
                      int            cols,
                      double        *input_A,
                      fftw_complex  *output_R,
                      unsigned int   flags);
```

Parameters

Argument	I/O	Description
rows	input	Matrix column length
cols	input	Matrix row length
input_A	input	Real input buffer.
output_R	output	Interleaved complex output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size.

- In-place or out-of-place operations.

However, the transformation type must be:

- Real-to-complex;
- Interleaved complex output buffer;
- Compact data elements;
- The data buffer must be 2D;

Restrictions and Notes

1. All data buffers in the plan must be of type compact, ie not strided.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library. Eg the algorithm works out if the input data is aligned or unaligned and will process it appropriately. The plan flag FFTW_UNALIGNED will therefore be ignored.

Example code

Example code of **fftw_plan_dft_r2c_2d** can be found in all the code examples contained in Section 10.7.3.

6.15 fftw_plan_guru_dft_r2c

Multi-dim Real to Interleaved Complex Guru FFTW plan function.

Function Definitions

```
void fftw_plan_guru_dft_r2c(
                     int            rank,
                     fftw_iodim    *dims,
                     int            howmany,
                     fftw_iodim    *howmany_dims,
                     double        *input_A,
                     fftw_complex  *output_R,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
dims	input	Structure defining data lengths and input/output strides.
howmany	input	howmany_dims array length.
howmany_dims	input	Structure defining how many FFTs and data block stride.
input_A	input	Real input buffer.
output_R	output	Interleaved complex output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - Multiple 1D FFTs;
  - 2D FFT;
  - Multiple 2D FFTs;
- In-place or out-of-place operations.
- The data type can be strided or compact. The stride is specified in the dims argument.

However, the transformation type must be:

- Real-to-complex;
- Interleaved complex output buffer;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The FFT operation is performed faster when the data is compact. However, t he user may specify strided data elements if it is a algorithm requirement.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library.

Example code

Example code of **fftw_plan_guru_dft_c2r** can be found in all the code examples contained in Section 10.8.2.

6.16 fftw_plan_guru_split_dft_r2c

Multi-dim Real to Split Complex Guru FFTW plan function.

Function Definitions

```
void fftw_plan_guru_split_dft_r2c(
                     int            rank,
                     fftw_iodim    *dims,
                     int            howmany,
                     fftw_iodim    *howmany_dims,
                     double        *input_A,
                     double        *output_R_re,
                     double        *output_R_im,
                     unsigned int   flags);
```

Parameters

Argument	I/O	Description
rank	input	Sets data dimension, ie 1=1D, 2=2D, etc
dims	input	Structure defining data lengths and input/output strides.
howmany	input	howmany_dims array length.
howmany_dims	input	Structure defining how many FFTs and data block stride.
input_A	input	Real input buffer.
output_R_re	output	Split complex real output buffer.
output_R_im	output	Split complex real output buffer.
flags	input	Flags allowed include: FFTW_MEASURE; FFTW_DESTROY_INPUT; FFTW_UNALIGNED; FFTW_CONSERVE_MEMORY; FFTW_EXHAUSTIVE; FFTW_PRESERVE_INPUT; FFTW_PATIENT; FFTW_ESTIMATE;

Return Value

A **fftw_plan** object. This is a pointer to a structure that defines the type of FFT to be carried out, the algorithm to be used and pointers to the input and output buffers.

Description

This function creates a FFTW plan structure with the users arguments. The arguments define the following user options:

- Data size plus type:
  - 1D FFT;
  - Multiple 1D FFTs;
  - 2D FFT;
  - Multiple 2D FFTs;
- In-place or out-of-place operations.
- The data type can be strided or compact. The stride is specified in the di ms argument.

However, the transformation type must be:

- Real-to-complex;
- Split complex output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. The FFT operation is performed faster if the input and output buffers are fully aligned for the target hardware.
3. The FFT operation is performed faster when the FFT lengths are a power of two.
4. The FFT operation is performed faster when the data is compact. However, t he user may specify strided data elements if it is a algorithm requirement.
5. The flag definitions are included for backward compatability purposes. Most flags get ignored by the algorithm in the library.

Example code

Example code of **fftw_plan_guru_split_dft_r2c** can be found in all the code examples contained in Section 10.8.

6.17 fftw_fprint_plan

Print FFTW plan information function.

Function Definitions

```
void fftw_fprint_plan(
                    const fftw_plan  plan,
                    FILE            *output_file);
```

Parameters

Argument	I/O	Description
plan	input	fftw_plan to print info to file from.
output_file	Output	Output file pointer.

Return Value

None: void function.

Description

This function prints information from the given FFTW plan to a file. The file must have already been opened by the program for text with write permission. The associated file pointer must then be given to this function along with the FFTW plan.

Restrictions and Notes

1. The plan must have been initialized by a FFTW plan creation function before this function is called.
2. The given file pointer must have been successfully opened to a text file with permission to write.
3. This function may be called as many times as required by the user.

Example code

Example code of **fftw_fprint_plan** can be found in all the code examples contained in Section 10.8.2.

6.18 fftw_print_plan

Print FFTW plan information function.

Function Definitions

```
void fftw_print_plan(
                    const fftw_plan plan,
```

Parameters

Argument	I/O	Description
plan	input	fftw_plan to print info to file from.

Return Value

None: void function.

Description

This function prints information from the given FFTW plan to standard output.

Restrictions and Notes

1. The plan must have been initialized by a FFTW plan creation function before this function is called.
2. This function may be called as many time as required by the user.

Example code

Example code of **fftw_print_plan** can be found in all the code examples contained in Section 10.8.3.

Chapter 7. FFTW Execute Functions

7.1 fftw_execute

General FFTW execute function.

Function Definitions

```
void fftw_execute(
                     const fftw_plan plan);
```

Parameters

Argument	I/O	Description
plan	input/output	FFTW plan including input and output buffers.

Return Value

None: void function.

Description

This function executes an FFT algorithm from the given plan. The plan must have been created by a FFTW plan creation routine and the input and output buffers must have been tagged onto the plan. The FFT operation can perform:

- Any transform type:
  - Complex-to-complex;
  - Complex-to-real;
  - Real-to-complex;
- Any data type:
  - 1D single FFT;
  - 1D multiple FFTs in a matrix;
  - 2D FFT;

  - Multiple 2D FFTs;

- Any complex type:
  - split complex;
  - interleaved complex;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. This execute function process a plan created by any FFTW plan creation function.
3. The data buffers attached to the plan are processed.
4. This may be an out-of-place or in-place operation depending on the plan creation.
5. The operation is performed faster if the input and output buffers are fully aligned for the target hardware.
6. The operation is performed faster when the FFT lengths are a power of two.

Example code

Example code of **fftw_execute** can be found in all the code examples contained in Section 10.3.1.

7.2 fftw_execute_dft

Interleaved Complex to Complex FFTW execute function.

Function Definitions

```
void fftw_execute_dft(
                      const fftw_plan  plan,
                      fftw_complex    *in,
                      fftw_complex    *out);
```

Parameters

Argument	I/O	Description
plan	input	FFTW plan setup to perform a complex-to-complex FFT.
in	input	Interleaved complex input buffer.
out	output	Interleaved complex output buffer.

Return Value

None: void function.

Description

This function executes a complex-to-complex FFT algorithm from the given plan. The plan must have been created by a FFTW plan creation routine. The FFT operation can be performed with any transformation type as defined by the plan:

- 1D single FFT;
- 1D multiple FFTs in a matrix;
- 2D FFTs;
- In-place or out-of-place operations.
- Forward or backward FFTs.

However, the transformation type must be:

- Complex-to-complex;
- Interleaved complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. All data buffers in the plan must be of type compact, ie not strided.
3. This may be an out-of-place or in-place operation depending on given buffer arguments.
4. The operation is performed faster if the input and output buffers are fully aligned for the target hardware.
5. The operation is performed faster when the FFT lengths are a power of two.
6. If the complex arguments are set to NULL then the buffers tagged onto the plan when it was created are used as the input and output complex buffers.

Example code

Example code of **fftw_execute** can be found in all the code examples contained in Section 10.3.2.

7.3 fftw_execute_split_dft

Split Complex to Complex FFTW execute function.

Function Definitions

```
void fftw_execute_split_dft(
                    const fftw_plan  plan,
                    double          *in_re,
                    double          *in_im,
                    double          *out_re,
                    double          *out_im);
```

Parameters

Argument	I/O	Description
plan	input	FFTW plan setup to perform a split complex-to-complex FFT.
in_re	input	Split complex real buffer.
in_im	input	Split complex imaginary buffer.
out_re	output	Split complex real buffer.
out_im	output	Split complex imaginary buffer.

Return Value

None: void function.

Description

This function executes a complex-to-complex FFT algorithm from the given plan. The plan must have been created by a FFTW guru split plan creation routine. The FFT operation can be performed with any transformation type as defined by the plan:

- 1D single FFT;
- 1D multiple FFTs in a matrix;
- 2D FFTs;
- In-place or out-of-place operations.
- Forward or backward FFTs.

However, the transformation type must be:

- Complex-to-complex;
- Split complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. This may be an out-of-place or in-place operation depending on given buffer arguments.
3. The operation is performed faster if the input and output buffers are fully aligned for the target hardware.
4. The operation is performed faster if the input and output buffers are are of type compact. ie the data stride is 1.
5. The operation is performed faster when the FFT lengths are a power of two.
6. If the complex arguments are set to NULL then the buffers tagged onto the plan when it was created are used as the input and output complex buffers.

Example code

Example code of **fftw_execute_split_dft** can be found in all the code examples contained in Section 10.4.

7.4 fftw_execute_dft_c2r

Interleaved Complex to Real FFTW execute function.

Function Definitions

```
void fftw_execute_dft_c2r(
                     const fftw_plan  plan,
                     fftw_complex    *in,
                     double          *out);
```

Parameters

Argument	I/O	Description
plan	input	FFTW plan setup to perform a Interleaved complex-to-real FFT.
in	input	Interleaved input complex buffer.
out	output	Float output buffer.

Return Value

None: void function.

Description

This function executes a interleaved complex-to-real FFT algorithm from the given plan. The plan must have been created by a FFTW c2r plan creation routine. The FFT operation can be performed with any transformation type as defined by the plan:

- 1D single FFT;
- 1D multiple FFTs in a matrix;
- 2D FFTs;
- In-place or out-of-place operations.

However, the transformation type must be:

- Complex-to-real;
- With interleaved complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. This may be an out-of-place or in-place operation depending on given buffer arguments.
3. The operation is performed faster if the input and output buffers are fully aligned for the target hardware.
4. The operation is performed faster if the input and output buffers are are of type compact. ie the data stride is 1.
5. The operation is performed faster when the FFT lengths are a power of two.
6. If the complex arguments are set to NULL then the buffers tagged onto the plan when it was created are used as the input and output complex buffers.

Example code

Example code of **fftw_execute_dft_c2r** can be found in all the code examples contained in Section 10.5.2.

7.5 fftw_execute_dft_r2c

Real to Interleaved Complex FFTW execute function.

Function Definitions

```
void fftw_execute_dft_r2c(
                     const fftw_plan  plan,
                     double          *in,
                     fftw_complex    *out);
```

Parameters

Argument	I/O	Description
plan	input	FFTW plan setup to perform a Real to Interleaved complex FFT.
in	input	Float input buffer.
out	output	Interleaved complex output buffer.

Return Value

None: void function.

Description

This function executes a real to interleaved complex FFT algorithm from the given plan. The plan must have been created by a FFTW r2c plan creation routine. The FFT operation can be performed with any transformation type as defined by the plan:

- 1D single FFT;
- 1D multiple FFTs in a matrix;
- 2D FFTs;
- In-place or out-of-place operations.

However, the transformation type must be:

- Real-to-complex;
- With interleaved complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. This may be an out-of-place or in-place operation depending on given buffer arguments.
3. The operation is performed faster if the input and output buffers are fully aligned for the target hardware.
4. The operation is performed faster if the input and output buffers are are of type compact. ie the data stride is 1.
5. The operation is performed faster when the FFT lengths are a power of two.
6. If the complex arguments are set to NULL then the buffers tagged onto the plan when it was created are used as the input and output complex buffers.

Example code

Example code of **fftw_execute_dft_c2r** can be found in all the code examples contained in Section 10.7.2.

7.6 fftw_execute_split_dft_c2r

Split Complex to Real FFTW execute function.

Function Definitions

```
void fftw_execute_split_dft_c2r(
                      const fftw_plan  plan,
                      double          *in_re,
                      double          *in_im,
                      double          *out);
```

Parameters

Argument	I/O	Description
plan	input	FFTW plan setup to perform a split complex-to-real FFT.
in_re	input	Split complex real input buffer.
in_im	input	Split complex imaginary input buffer.
out	output	Float output buffer.

Return Value

None: void function.

Description

This function executes a split complex-to-real FFT algorithm from the given plan. The plan must have been created by a FFTW c2r split guru plan creation routine. The FFT operation can be performed with any transformation type as defined by the plan:

- 1D single FFT;
- 1D multiple FFTs in a matrix;
- 2D FFTs;
- In-place or out-of-place operations.

However, the transformation type must be:

- Complex-to-real;
- With split complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. This may be an out-of-place or in-place operation depending on given buffer arguments.
3. The operation is performed faster if the input and output buffers are fully aligned for the target hardware.
4. The operation is performed faster if the input and output buffers are are of type compact. ie the data stride is 1.
5. The operation is performed faster when the FFT lengths are a power of two.
6. If the complex arguments are set to NULL then the buffers tagged onto the plan when it was created are used as the input and output complex buffers.

Example code

Example code of **fftw_execute_split_dft_c2r** can be found in all the code examples contained in Section 10.6.

7.7 fftw_execute_split_dft_r2c

Real to Split Complex FFTW execute function.

Function Definitions

```
void fftw_execute_split_dft_r2c(
                     const fftw_plan  plan,
                     double          *in,
                     double          *out_re,
                     double          *out_im);
```

Parameters

Argument	I/O	Description
plan	input	FTWF plan setup to perform a real to split complex FFT.
in	input	Float input buffer.
out_re	output	Split complex real output buffer.
out_im	output	Split complex imaginary output buffer.

Return Value

None: void function.

Description

This function executes a real to split complex FFT algorithm from the given plan. The plan must have been created by a FFTW r2c split guru plan creation routine. The FFT operation can be performed with any transformation type as defined by the plan:

- 1D single FFT;
- 1D multiple FFTs in a matrix;
- 2D FFTs;
- In-place or out-of-place operations.

However, the transformation type must be:

- Real-to-complex;
- With split complex input and output buffers;

Restrictions and Notes

1. 3D FFTs and above are not supported.
2. This may be an out-of-place or in-place operation depending on given buffer arguments.
3. The operation is performed faster if the input and output buffers are fully aligned for the target hardware.
4. The operation is performed faster if the input and output buffers are are of type compact. ie the data stride is 1.
5. The operation is performed faster when the FFT lengths are a power of two.
6. If the complex arguments are set to NULL then the buffers tagged onto the plan when it was created are used as the input and output complex buffers.

Example code

Example code of **fftw_execute_split_dft_r2c** can be found in all the code examples contained in Section 10.8.

Chapter 8. FFTW Thread Support Functions

WARNING: The threaded FFTW library must be initialised with a call to **fftw_init** before any FFTW function can be called. (See Chapter 4).

Once initialised each function in the library will use upto a number of maximum threads. This maximum number is hardwared into the code. and has been setup for your target platform at compile time. How many threads each function actually uses is dependent on the data size which the function is processing and the computational load of the function's algorithm. Small data lengths will only use one thread if the computational load is low. As the data length increases then more threads will be used to split the function's workload over multiple cores. Once the thread count has reached the maximum value then the thread count is not increased.

The threaded FFTW library is designed to be automatic and the user should not have to do anything after they have initialised the **fftw_init** interface function. The maximum number of threads is set at compile time and cannot be increased. However, the following function (**Thread_SetParams**) can be used to lower the maximum number of threads if this is required by the user. This is a non-standard FFTW function and is supplied to help the user control the mnumber of threads.

8.1 Thread_SetParams

Set Number of Threads.

Function Definitions

```
void Thread_SetParams(
                      int num_threads,
                      int max_num_running);
```

Parameters

Argument	I/O	Description
num_threads	input	Maximum number of threads.
max_num_running	input	Maximum number of threads left running.

Return Value

None: void function.

Description

The Threaded FFTW library is multithreaded and will take advantage of multiple cores on the processor invoking it. Utilising multiple threads is automatic:

- The maximum number of threads used is set when any FFTW function is called for the first time.
- The maximum number running at any one time is also set at that point. If a threaded routine is called with (say) 4 threads and we have hit this maximum number running then four of them are shut down before the new function is executed.
- The number of threads invoked when a routine is called, is decided by that routine by reference to the data (vector or matrix) size specified in the call, to provide the best performance for that call.

Restrictions and Notes

1. A threaded, and a non-threaded ("serial") version of the library are provided. If you wish to only ever use one thread in a library call, use the serial version of the

library.

2. The maximum number of threads used for a specific function call, and the maximum number kept running at any one time, can be changed by a call to the routine `Thread_SetParams` with arguments `num_threads` and `max_num_running`. This call, if used, *must* be made before any FFTW function is called.

3. If no call to `Thread_SetParams` is made, the library default values will be utilised.

The following thread support functions are provided largely for backward compatability purposes. If the function fftw_init has been called then the threaded libraries will already be initialised. The function fftw_finalize should also be called at the end of the application.

8.2 fftw_init_threads

FFTW thread initialization function.

Function Definitions

```
int fftw_init(void);
```

Parameters

None: No function arguments.

Return Value

Returns 1 on success.

Description

This function is equivalent to a call to fftw_init(). It is provided in the library for backward compatability to the real FFTW library.

Restrictions and Notes

None.

8.3 fftw_plan_with_nthreads

FFTW thread initialization function specifying the number of threads.

Function Definitions

```
void fftw_plan_with_nthreads(
                     int nthreads);
```

Parameters

Argument	I/O	Description
nthreads	input	Number of threads.

Return Value

None: void function.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

8.4 fftw_setthreads

FFTW thread initialization function specifying the number of threads.

Function Definitions

```
void fftw_setthreads(
                     int nthreads);
```

Parameters

Argument	I/O	Description
nthreads	input	Number of threads.

Return Value

None: void function.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

8.5 fftw_cleanup_threads

FFTW thread destroy function.

Function Definitions

```
int fftw_cleanup_threads(void);
```

Parameters

None: No function arguments.

Return Value

Returns 1 on success.

Description

This function is equivalent to a call to fftw_finalize(). It is provided in the library for backward compatability to the real FFTW library.

Restrictions and Notes

None.

Chapter 9. FFTW Wisdom Functions

The following Wisdom support functions are provided for backward compatability purposes. These routines in the NAS FFTW library are dummy functions that do not do anything. However, they enable old FFTW code to be compiled with the NAS FFTW library without requiring code modifications.

9.1 fftw_export_wisdom_to_file

FFTW wisdom file export function.

Function Definitions

```
void fftw_export_wisdom_to_file(
                    FILE *output_file);
```

Parameters

Argument	I/O	Description
output_file	output	File pointer.

Return Value

None: void function.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

9.2 fftw_import_wisdom_from_file

FFTW wisdom file import function.

Function Definitions

```
int fftw_import_wisdom_from_file(
                    FILE *input_file);
```

Parameters

Argument	I/O	Description
input_file	input	File pointer.

Return Value

Always returns 1.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

9.3 fftw_export_wisdom_to_string

FFTW wisdom string export function.

Function Definitions

```
char * fftw_export_wisdom_to_string(void);
```

Parameters

None: No function arguments.

Return Value

Pointer to a string. This function is a dummy function so always returns a NULL.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

9.4 fftw_import_wisdom_from_string

FFTW wisdom string import function.

Function Definitions

```
int fftw_import_wisdom_from_string(
                    const char *input_string);
```

Parameters

Argument	I/O	Description
input_string	input	Pointer to a string.

Return Value

Always returns a 1.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

9.5 fftw_forget_wisdom

FFTW wisdom destroy function.

Function Definitions

```
void fftw_forget_wisdom(void);
```

Parameters

None: no function arguments.

Return Value

None: void function.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

9.6 fftw_import_system_wisdom

FFTW wisdom import function.

Function Definitions

```
int fftw_import_system_wisdom(void);
```

Parameters

None: No function arguments.

Return Value

Always returns a 1.

Description

This function is a dummy function. It has been provided for the purposes of backward compatability to the original FFTW library.

Restrictions and Notes

None.

Chapter 10. Worked Examples of the Library

10.1 Header File

The routines in the NAS FFTW DSP library are defined within a single header file called:

[DISTRIBUTION]/include/fftw.h

This header file comes with every FFTW distribution and must be included in any C source code in order to use the library. If the user has C++ source code then they must befine the library as being of type C with:

```
extern "C"
{
#include "fftw.h"
}
```

10.2 C Source Code Examples

This chapter includes C source code examples that show how FFTW routines are called. The examples are made up of the following:

Section	Complex type	FFT type	Operation type
10.3.1	interleaved	complex-to-complex	1D
10.3.2	interleaved	complex-to-complex	Multiple 1D
10.3.3	interleaved	complex-to-complex	2D
10.4.1	split	complex-to-complex	1D
10.4.2	split	complex-to-complex	Multiple 1D
10.4.3	split	complex-to-complex	2D
10.5.1	interleaved	complex-to-real	1D
10.5.2	interleaved	complex-to-real	Multiple 1D
10.5.3	interleaved	complex-to-real	2D
10.6.1	split	complex-to-real	1D
10.6.2	split	complex-to-real	Multiple 1D
10.6.3	split	complex-to-real	2D
10.7.1	interleaved	real-to-complex	1D
10.7.2	interleaved	real-to-complex	Multiple 1D
10.7.3	interleaved	real-to-complex	2D
10.8.1	split	real-to-complex	1D
10.8.2	split	real-to-complex	Multiple 1D
10.8.3	split	real-to-complex	2D

10.3 Inteleaved Complex to Complex Code Examples.

10.3.1 Interleaved Complex to Complex 1D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/*************************************************************************
 *                                                                       *
 * test_plan_dft_cc1dopf()                                               *
 *                                                                       *
 *    TEST FUNCTION: fftw_plan_dft.                                      *
 *    TEST TYPE    : cc1dopf - complex to complex interleaved.           *
 *                             1 dimensional                             *
 *                             out-of-place                              *
 *                             forward fft.                              *
 *                                                                       *
 *    Arguments: time_output - file pointer to time file. Each vector    *
 *                             is timed and the estimated time is output *
 *                             to this file.                             *
 *               test_output - Result of each vector is output to this   *
 *                             file.                                     *
 *                                                                       *
 *************************************************************************/
int test_plan_dft_cc1dopf(NASFILE time_output,
                          NASFILE test_output)
{
   fftw_complex *in_buffer, *out_buffer;
   time_val start_time, end_time;
   int length, i1, runs, run;
   float sum_time, time1;
   fftw_plan plan=NULL;

   /* Set the number of runs for test function.                          */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                  */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_plan_dft : cc fft 1d op forward.\n");
```

```
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output, "fftw_plan_dft : cc fft 1d op forward.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(length=para.vec_start; length<=para.vec_end; length*=para.vec_inc)
{
   /* Malloc enough memory.                                              */
   in_buffer = (fftw_complex*)
      fftw_malloc(length*sizeof(fftw_complex));
   out_buffer = (fftw_complex*)
      fftw_malloc(length*sizeof(fftw_complex));
   if(in_buffer==NULL || out_buffer==NULL) return 0;

   sum_time = 0.0;
   set_cdata(in_buffer, length);
   for(run=0; run<runs; run++)
   {
      if(run != 0) fftw_destroy_plan(plan);

      /* Call timer.                                                     */
      start_time = test_time();

      /* Create 1d cc out of place plan.                                 */
      plan = fftw_plan_dft(1, &length, in_buffer, out_buffer,
         FFTW_FORWARD, FFTW_ESTIMATE);

      end_time = test_time();

      sum_time += time_diff(start_time, end_time);
   }

   /* Call fftw_execute.                                                 */
   fftw_execute(plan);

   /* Ouput time information.                                            */
   if(time_output != NASNULL)
```

```
      {
         time1 = sum_time/runs;
         nasfprintf(time_output, "vector %d time %f secs\n", length, time1);
         nasfflush(time_output);
      }

      /* Ouput result of fftw_execute as an extra test.                    */
      if(test_output != NASNULL)
      {
         nasfprintf(test_output, "vector %d:\n", length);
         for(i1=0; i1<length; i1++)
         {
            nasfprintf(test_output, "cell %d = %lf %lf\n", i1,
               out_buffer[i1][0], out_buffer[i1][1]);
         }
         nasfprintf(test_output, "\n");
         nasfflush(test_output);
      }


      /* Free the memory buffers and plan.                                 */
      fftw_destroy_plan(plan);
      fftw_free(out_buffer);
      fftw_free(in_buffer);
   }

   return 1;

} /* End of test_plan_dft_cc1dopf.                                        */
```

10.3.2 Interleaved Complex to Complex Multiple FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_dft_1dipfm()                                            *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_dft                                   *
 *    DATA           1d in-place complex interleaved.                   *
 *    DIR            multiple forward (inverse) 1d ffts.                *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_dft_1dipfm(NASFILE time_output,
                            NASFILE test_output)
{
   fftw_iodim dims[1], howmany_dims[1];
   int cols, rows, i1, i2, runs, run;
   time_val start_time, end_time;
   fftw_complex *in_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;

   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_execute_dft : cc fft 1d ip forward.\n");
      nasfflush(time_output);
   }
   if(test_output != NASNULL)
```

```
   {
      nasfprintf(test_output, "fftw_execute_dft : cc fft 1d ip foforrward.\n");
      nasfflush(test_output);
   }


   /* Loop though required vector lengths.                                  */
   for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
       cols*=para.mat_col_inc)
   {
      for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
          rows*=para.mat_row_inc)
      {
         /* Malloc enough memory.                                           */
         in_buffer = (fftw_complex*)
            fftw_malloc(cols*rows*sizeof(fftw_complex));
         if(in_buffer==NULL) return 0;

         /* Create a multiple 1d cc in place plan.                         */
         dims[0].n = cols;
         dims[0].is = 1;
         dims[0].os = 1;
         howmany_dims[0].n = rows;
         howmany_dims[0].is = cols;
         howmany_dims[0].os = cols;
         plan = fftw_plan_guru_dft(1, dims,
            1, howmany_dims,
            in_buffer, in_buffer,
            FFTW_FORWARD, FFTW_ESTIMATE);


         sum_time = 0.0;
         set_cdata(in_buffer, rows*cols);
         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                          */
#ifndef NO_RESET
            set_cdata(in_buffer, rows*cols);
#endif

            /* Call timer.                                                  */
            start_time = test_time();
```

```
            /* Call fftw_execut.                                          */
            fftw_execute_dft(plan, in_buffer, in_buffer);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }


         /* Ouput time information.                                       */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output, "matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Output result of fftw_execute_dft.                            */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output, "matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %f %f\n", i1, i2,
                     in_buffer[i1*cols+i2][0], in_buffer[i1*cols+i2][1]);
               }
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                             */
         fftw_destroy_plan(plan);
         fftw_free(in_buffer);
      }
   }
   return 1;
```

```
} /* End of test_execute_dft_1dipfm.                                    */
```

10.3.3 Interleaved Complex to Complex 2D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_plan_dft_2d_ccopf()                                             *
 *                                                                      *
 *    TEST FUNCTION: fftw_plan_dft_2d                                   *
 *    DATA           out-of-place complex interleaved.                  *
 *    DIR            forward fft.                                       *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_plan_dft_2d_ccopf(NASFILE time_output,
                           NASFILE test_output)
{
   int cols, rows, i1, i2, runs, run;
   time_val start_time, end_time;
   fftw_complex *out_buffer;
   fftw_complex *in_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;

   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_plan_dft_2d : cc fft 2d op forward.\n");
      nasfflush(time_output);
   }
   if(test_output != NASNULL)
```

```
{
   nasfprintf(test_output, "fftw_plan_dft_2d : cc fft 2d op forward.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                    */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                             */
      in_buffer = (fftw_complex*)
         fftw_malloc(cols*rows*sizeof(fftw_complex));
      out_buffer = (fftw_complex*)
         fftw_malloc(cols*rows*sizeof(fftw_complex));
      if(in_buffer==NULL || out_buffer==NULL) return 0;


      sum_time = 0.0;
      set_cdata(in_buffer, rows*cols);
      for(run=0; run<runs; run++)
      {
         if(run > 0) fftw_destroy_plan(plan);

         /* Call timer.                                                    */
         start_time = test_time();

         /* Create 2d cc out of place plan.                                */
         plan = fftw_plan_dft_2d(rows, cols, in_buffer, out_buffer,
            FFTW_FORWARD, FFTW_ESTIMATE);

         end_time = test_time();

         sum_time += time_diff(start_time, end_time);
      }


      /* Call fftw_execute.                                                */
      fftw_execute(plan);
```

```
         /* Ouput time information.                                    */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output, "matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Output result of fftw_execute as an extra check.          */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output, "matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf %lf\n", i1, i2,
                     out_buffer[i1*cols+i2][0], out_buffer[i1*cols+i2][1]);
               }
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                         */
         fftw_destroy_plan(plan);
         fftw_free(out_buffer);
         fftw_free(in_buffer);
      }
   }
   return 1;

} /* End of test_plan_dft_2d_ccopf                                    */
```

10.4 Split Complex to Complex Code Examples.

10.4.1 Split Complex to Complex 1D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_1dipb1()                                      *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft                             *
 *    TEST 1dipb1:   complex to complex split data.                     *
 *                   1 dimensional.                                     *
 *                   in-place data.                                     *
 *                   backward fft (inverse).                            *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_1dipb1(NASFILE time_output,
                                  NASFILE test_output)
{
   fftw_iodim dims[1], howmany_dims[1];
   time_val start_time, end_time;
   double *in_buffer_re = NULL;
   double *in_buffer_im = NULL;
   int length, i1, runs, run;
   float sum_time, time1;
   fftw_plan plan=NULL;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
```

```
if(time_output != NASNULL)
{
   nasfprintf(time_output,
      "fftw_execute_split_dft : cc fft 1d ip backward.\n");
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_split_dft : cc fft 1d ip backward.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                    */
for(length=para.vec_start; length<=para.vec_end; length*=para.vec_inc)
{
   /* Malloc enough memory.                                                */
   in_buffer_re = (double*)
      fftw_malloc(length*sizeof(double));
   in_buffer_im = (double*)
      fftw_malloc(length*sizeof(double));
   if(in_buffer_re==NULL || in_buffer_im==NULL) return 0;
   sum_time = 0.0;
   set_ddata(in_buffer_re, in_buffer_im, length);

   /* Create 1d cc out of place plan.                                      */
   dims[0].n = length;
   dims[0].is = 1;
   dims[0].os = 1;
   howmany_dims[0].n = 1;
   howmany_dims[0].is = length;
   howmany_dims[0].os = length;
   plan = fftw_plan_guru_split_dft(1, dims,
      0, howmany_dims,
      in_buffer_im, in_buffer_re,
      in_buffer_im, in_buffer_re,
      FFTW_ESTIMATE);


   for(run=0; run<runs; run++)
   {
```

```
         /* Set the input data.                                          */
#ifndef NO_RESET
         set_ddata(in_buffer_re, in_buffer_im, length);
#endif

         /* Call timer.                                                  */
         start_time = test_time();

         /* Call fftw_execute_split_dft.                                 */
         fftw_execute_split_dft(plan, in_buffer_im, in_buffer_re,
            in_buffer_im, in_buffer_re);

         end_time = test_time();
         sum_time += time_diff(start_time, end_time);
      }



      /* Ouput time information.                                         */
      if(time_output != NASNULL)
      {
         time1 = sum_time/runs;
         nasfprintf(time_output, "vector %d time %f secs\n", length, time1);
         nasfflush(time_output);
      }

      /* Output result off fft to data file.                            */
      if(test_output != NASNULL)
      {
         nasfprintf(test_output, "vector %d:\n", length);
         for(i1=0; i1<length; i1++)
         {
            nasfprintf(test_output, "cell %d = %lf %lf\n", i1,
               in_buffer_re[i1], in_buffer_im[i1]);
         }
         nasfprintf(test_output, "\n");
         nasfflush(test_output);
      }


      /* Free the memory buffers and plan.                              */
      fftw_destroy_plan(plan);
```

```
      fftw_free(in_buffer_re);
      fftw_free(in_buffer_im);
   }

   return 1;

} /* End of test_execute_split_dft_1dipb1.                                  */
```

10.4.2 Split Complex to Complex Multiple FFTs.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_1dopfm()                                      *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft                             *
 *    DATA           out-of-place complex split data.                   *
 *    DIR            multiple forward 1d fft.                           *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output*
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_1dopfm(NASFILE time_output,
                                  NASFILE test_output)
{
   fftw_iodim dims[1], howmany_dims[1];
   int cols, rows, i1, i2, runs, run;
   time_val start_time, end_time;
   double *out_buffer_re=NULL;
   double *in_buffer_re=NULL;
   double *out_buffer_im=NULL;
   double *in_buffer_im=NULL;
   float sum_time, time1;
   fftw_plan plan=NULL;

   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output,
```

```
      "fftw_execute_split_dft : multiple cc fft 1d op forward.\n");
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_split_dft : multiple cc fft 1d op forward.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                              */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                       */
      in_buffer_re = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      out_buffer_re = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      in_buffer_im = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      out_buffer_im = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer_re==NULL || out_buffer_re==NULL ||
         in_buffer_im==NULL || out_buffer_im==NULL) return 0;


      sum_time = 0.0;
      set_ddata(in_buffer_re, in_buffer_im, rows*cols);

      /* Create multiple 1d cc out of place plan.                    */
      dims[0].n = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      howmany_dims[0].n = rows;
      howmany_dims[0].is = cols;
      howmany_dims[0].os = cols;
      plan = fftw_plan_guru_split_dft(1, dims,
```

```
            1, howmany_dims,
            in_buffer_re, in_buffer_im,
            out_buffer_re, out_buffer_im,
            FFTW_ESTIMATE);

         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                    */
#ifndef NO_RESET
            set_ddata(in_buffer_re, in_buffer_im, rows*cols);
#endif
            /* Call timer.                                            */
            start_time = test_time();

            /* Call fftw_execute_split_dft.                           */
            fftw_execute_split_dft(plan, in_buffer_re, in_buffer_im,
               out_buffer_re, out_buffer_im);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }


         /* Output time information.                                  */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output, "matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Output result of fftw_execute_split_dft.                 */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output, "matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf %lf\n", i1, i2,
                     out_buffer_re[i1*cols+i2], out_buffer_im[i1*cols+i2]);
```

```
               }
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                              */
         fftw_destroy_plan(plan);
         fftw_free(out_buffer_re);
         fftw_free(in_buffer_re);
         fftw_free(out_buffer_im);
         fftw_free(in_buffer_im);
      }
   }
   return 1;

} /* End of test_execute_split_dft_1dopfm.                                */
```

10.4.3 Split Complex to Complex 2D FFTs.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_2dipf1()                                      *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft                             *
 *    DATA           2d in-place complex split data.                    *
 *    DIR            forward 2d fft.                                    *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_2dipf1(NASFILE time_output,
                                  NASFILE test_output)
{
   fftw_iodim dims[2], howmany_dims[2];
   int cols, rows, i1, i2, runs, run;
   time_val start_time, end_time;
   double *in_buffer_re=NULL;
   double *in_buffer_im=NULL;
   float sum_time, time1;
   fftw_plan plan=NULL;

   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output,
         "fftw_execute_split_dft : cc fft 2d ip forward.\n");
      nasfflush(time_output);
```

```
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_split_dft : cc fft 2d ip forward.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                           */
      in_buffer_re = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      in_buffer_im = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer_re==NULL || in_buffer_im==NULL) return 0;

      sum_time = 0.0;
      set_ddata(in_buffer_re, in_buffer_im, rows*cols);


      /* Create 2d cc in place plan.                                     */
      dims[0].n = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      dims[1].n = rows;
      dims[1].is = cols;
      dims[1].os = cols;
      howmany_dims[0].n = 1;
      howmany_dims[0].is = cols*rows;
      howmany_dims[0].os = cols*rows;
      plan = fftw_plan_guru_split_dft(2, dims,
         0, howmany_dims,
         in_buffer_re, in_buffer_im,
         in_buffer_re, in_buffer_im,
         FFTW_ESTIMATE);
```

```
         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                          */
#ifndef NO_RESET
            set_ddata(in_buffer_re, in_buffer_im, rows*cols);
#endif

            /* Call timer.                                                  */
            start_time = test_time();
            /* Call fftw_execute_split_dft.                                 */
            fftw_execute_split_dft(plan, in_buffer_re, in_buffer_im,
               in_buffer_re, in_buffer_im);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }


         /* Output time information.                                        */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output, "matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Output result of fftw_execute_split_dft.                     */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output, "matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf %lf\n", i1, i2,
                     in_buffer_re[i1*cols+i2], in_buffer_im[i1*cols+i2]);
               }
            }
            nasfprintf(test_output, "\n");
```

```
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                          */
         fftw_destroy_plan(plan);
         fftw_free(in_buffer_re);
         fftw_free(in_buffer_im);

      }
   }
   return 1;

} /* End of test_execute_split_dft_2dipf1.                            */
```

10.5 Inteleaved Complex to Real Code Examples.

10.5.1 Interleaved Complex to Real 1D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_plan_dft_c2r_1dop()                                             *
 *                                                                      *
 *    TEST FUNCTION: test_plan_dft_c2r                                  *
 *    DATA           1d out-of-place complex interleaved to real double.  *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector     *
 *                             is timed and the estimated time is output  *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this    *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_plan_dft_c2r_1dop(NASFILE time_output,
                           NASFILE test_output)
{
   int length, i1, runs, run, cmp_length;
   time_val start_time, end_time;
   fftw_complex *in_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *out_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_plan_dft_c2r : 1d op.\n");
      nasfflush(time_output);
```

```
}
if(test_output != NASNULL)
{
   nasfprintf(test_output, "fftw_plan_dft_c2r : 1d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                    */
for(length=para.vec_start; length<=para.vec_end; length*=para.vec_inc)
{
   /* Malloc enough memory.                                                */
   cmp_length = length/2+1;
   in_buffer = (fftw_complex*)
      fftw_malloc(cmp_length*sizeof(fftw_complex));
   out_buffer = (double*)
      fftw_malloc(length*sizeof(double));
   if(in_buffer==NULL || out_buffer==NULL) return 0;

   sum_time = 0.0;
   set_c2r_data(in_buffer, cmp_length);

   for(run=0; run<runs; run++)
   {
      if(run != 0) fftw_destroy_plan(plan);

      /* Call timer.                                                       */
      start_time = test_time();

      /* Create 1d cr out of place plan.                                   */
      plan = fftw_plan_dft_c2r(1, &length, in_buffer, out_buffer,
         FFTW_ESTIMATE);
      end_time = test_time();

      sum_time += time_diff(start_time, end_time);
   }


   /* Call fftw_execute.                                                   */
   fftw_execute(plan);
```

```
      /* Ouput time information.                                          */
      if(time_output != NASNULL)
      {
         time1 = sum_time/runs;
         nasfprintf(time_output, "vector real %d cmp %d: time %f secs\n",
            length, cmp_length, time1);
         nasfflush(time_output);
      }

      /* Ouput result of fftw_execute as an extra test.                  */
      if(test_output != NASNULL)
      {
         nasfprintf(test_output, "vector real %d cmp %d:\n", length,
            cmp_length);
         for(i1=0; i1<length; i1++)
         {
            nasfprintf(test_output, "cell %d = %lf \n", i1, out_buffer[i1]);
         }
         nasfprintf(test_output, "\n");
         nasfflush(test_output);
      }

      /* Free the memory buffers and plan.                               */
      fftw_destroy_plan(plan);
      fftw_free(out_buffer);
      fftw_free(in_buffer);
   }

   return 1;

} /* End of test_plan_dft_c2r_1dop.                                      */
```

10.5.2 Interleaved Complex to Real Multiple FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_dft_c2r_1dopm()                                         *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_dft_c2r                               *
 *    DATA           out-of-place complex interleaved to real.          *
 *    DIR            multiple 1d fft.                                  *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_dft_c2r_1dopm(NASFILE time_output,
                               NASFILE test_output)
{
   int cols, rows, i1, i2, runs, run, cmp_cols;
   fftw_iodim dims[1], howmany_dims[1];
   time_val start_time, end_time;
   fftw_complex *in_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *out_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output,
         "fftw_execute_dft_c2r : multiple cr fft 1d op.\n");
```

```
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_dft_c2r : multiple cc fft 1d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   cmp_cols = (cols/2)+1;
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                           */
      in_buffer = (fftw_complex*)
         fftw_malloc(cmp_cols*rows*sizeof(fftw_complex));
      out_buffer = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer==NULL || out_buffer==NULL) return 0;


      sum_time = 0.0;
      set_c2r_mdata(in_buffer, rows, cmp_cols);


      /* Create a multiple 1d cr in place plan.                          */
      dims[0].n = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      howmany_dims[0].n = rows;
      howmany_dims[0].is = cmp_cols;
      howmany_dims[0].os = cols;
      plan = fftw_plan_guru_dft_c2r(1, dims,
         1, howmany_dims,
         in_buffer, out_buffer,
         FFTW_ESTIMATE);
```

```
         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                          */
#ifndef NO_RESET
            set_c2r_mdata(in_buffer, rows, cmp_cols);
#endif

            /* Call timer.                                                  */
            start_time = test_time();

            /* Call fftw_execute_dft_c2r.                                   */
            fftw_execute_dft_c2r(plan, in_buffer, out_buffer);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }



         /* Ouput time information.                                         */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
               rows, cols, time1);
            nasfflush(time_output);
         }

         /* Output result of fftw_execute as an extra test.              */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
               "real matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf\n", i1, i2,
                     out_buffer[i1*cols+i2]);
               }
```

```
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                          */
         fftw_destroy_plan(plan);
         fftw_free(out_buffer);
         fftw_free(in_buffer);
      }
   }
   return 1;

} /* End of test_execute_dft_c2r_1dopm.                               */
```

10.5.3 Interleaved Complex to Real 2D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_plan_dft_c2r_2d_op()                                            *
 *                                                                      *
 *    TEST FUNCTION: fftw_plan_dft_c2r_2d                               *
 *    DATA           2d out-of-place complex interleaved to real.       *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_plan_dft_c2r_2d_op(NASFILE time_output,
                            NASFILE test_output)
{
   int cols, cmp_cols, rows, i1, i2, runs, run;
   time_val start_time, end_time;
   fftw_complex *in_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *out_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_plan_dft_c2r_2d  : cr fft 2d op.\n");
      nasfflush(time_output);
   }
   if(test_output != NASNULL)
```

```
{
   nasfprintf(test_output, "fftw_plan_dft_c2r_2d  : cr fft 2d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      cmp_cols = cols/2+1;

      /* Malloc enough memory.                                           */
      in_buffer = (fftw_complex*)
         fftw_malloc(cmp_cols*rows*sizeof(fftw_complex));
      out_buffer = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer==NULL || out_buffer==NULL) return 0;


      sum_time = 0.0;
      set_c2r3_mdata(in_buffer, rows, cmp_cols);

      for(run=0; run<runs; run++)
      {
         if(run > 0) fftw_destroy_plan(plan);

         /* Call timer.                                                  */
         start_time = test_time();

         /* Create 2d cr out of place plan.                              */
         plan = fftw_plan_dft_c2r_2d(rows, cols, in_buffer, out_buffer,
            FFTW_ESTIMATE);

         end_time = test_time();

         sum_time += time_diff(start_time, end_time);
      }
```

```
         /* Call fftw_execute.                                          */
         fftw_execute(plan);


         /* Ouput time information.                                     */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Ouput result to data file.                                  */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
               "real matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf \n", i1, i2,
                     out_buffer[i1*cols+i2]);
               }
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                           */
         fftw_destroy_plan(plan);
         fftw_free(out_buffer);
         fftw_free(in_buffer);
      }
   }
   return 1;

} /* End of test_plan_dft_c2r_2d_op.                                    */
```

10.6 Split Complex to Real Code Examples.

10.6.1 Split Complex to Real 1D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_c2r_1dop1()                                   *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft_c2r.                        *
 *    TEST TYTE    : 1dop1  - complex interleaved to real.              *
 *                            1 dimensional.                            *
 *                            out-of-place.                             *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output*
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_c2r_1dop1(NASFILE time_output,
                                     NASFILE test_output)
{
   int length, i1, runs, run, cmp_length;
   fftw_iodim dims[1], howmany_dims[1];
   time_val start_time, end_time;
   double *in_buffer_re=NULL;
   double *in_buffer_im=NULL;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *out_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
```

```
if(time_output != NASNULL)
{
   nasfprintf(time_output, "fftw_execute_split_dft_c2r : cr fft 1d op.\n");
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output, "fftw_execute_split_dft_c2r : cr fft 1d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(length=para.vec_start; length<=para.vec_end; length*=para.vec_inc)
{
   cmp_length = (length/2) + 1;

   /* Malloc enough memory.                                              */
   in_buffer_re = (double*)
      fftw_malloc(cmp_length*sizeof(double));
   in_buffer_im = (double*)
      fftw_malloc(cmp_length*sizeof(double));
   out_buffer = (double*)
      fftw_malloc(length*sizeof(double));
   if(in_buffer_re==NULL || in_buffer_im==NULL || out_buffer==NULL)
      return 0;

   sum_time = 0.0;
   set_c2r6_sdata(in_buffer_re, in_buffer_im, out_buffer, cmp_length);


   /* Define an out-place 1d fft.                                        */
   dims[0].n = length;
   dims[0].is = 1;
   dims[0].os = 1;
   howmany_dims[0].n = 1;
   howmany_dims[0].is = length;
   howmany_dims[0].os = length;
   plan = fftw_plan_guru_split_dft_c2r(1, dims,
      0, howmany_dims,
      in_buffer_re, in_buffer_im, out_buffer,
      FFTW_ESTIMATE);
```

```
      for(run=0; run<runs; run++)
      {
         /* Set the input data.                                          */
#ifndef NO_RESET
         set_c2r6_sdata(in_buffer_re, in_buffer_im, out_buffer, cmp_length);
#endif

         /* Call timer.                                                  */
         start_time = test_time();

         /* Call fftw_execute_split_dft_c2r.                             */
         fftw_execute_split_dft_c2r(plan, in_buffer_re, in_buffer_im,
            out_buffer);

         end_time = test_time();
         sum_time += time_diff(start_time, end_time);
      }


      /* Ouput time information.                                         */
      if(time_output != NASNULL)
      {
         time1 = sum_time/runs;
         nasfprintf(time_output,
            "vector real length %d cmp length %d - time %f secs\n",
            length, cmp_length, time1);
         nasfflush(time_output);
      }

      /* Ouput result of fftw_execute as an extra test.                  */
      if(test_output != NASNULL)
      {
         nasfprintf(test_output, "vector real length %d cmp length %d\n",
            length, cmp_length);
         for(i1=0; i1<length; i1++)
         {
            nasfprintf(test_output, "cell %d = %lf \n", i1,
               out_buffer[i1]);
         }
         nasfprintf(test_output, "\n");
         nasfflush(test_output);
```

```
      }


      /* Free the memory buffers and plan.                              */
      fftw_destroy_plan(plan);
      fftw_free(in_buffer_re);
      fftw_free(in_buffer_im);
      fftw_free(out_buffer);
   }

   return 1;

} /* End of test_execute_split_dft_c2r_1dop1.                          */
```

10.6.2 Split Complex to Real Multiple FFTs.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_c2r_1dipm()                                   *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft_c2r                         *
 *    DATA           1d in-place complex split to real ffts.           *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_c2r_1dipm(NASFILE time_output,
                                     NASFILE test_output)
{
   int cmp_cols, cols, rows, i1, i2, runs, run;
   fftw_iodim dims[1], howmany_dims[1];
   time_val start_time, end_time;
   double *in_buffer_re=NULL;
   double *in_buffer_im=NULL;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *out_buffer;

   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output,
         "fftw_execute_split_dft_c2r : multiple cr fft 1d ip.\n");
      nasfflush(time_output);
```

```
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_split_dft_c2r : multiple cr fft 1d ip.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   cmp_cols = (cols/2)+1;

   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                           */
      in_buffer_re = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      in_buffer_im = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer_re==NULL || in_buffer_im==NULL) return 0;
      out_buffer = (double*)in_buffer_re;

      sum_time = 0.0;
      set_c2r_smdata(in_buffer_re, in_buffer_im, rows, cmp_cols);


      /* Create a multiple 1d cr in place plan.                          */
      dims[0].n = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      howmany_dims[0].n = rows;
      howmany_dims[0].is = cols;
      howmany_dims[0].os = cols;
      plan = fftw_plan_guru_split_dft_c2r(1, dims,
         1, howmany_dims,
         in_buffer_re, in_buffer_im, out_buffer,
         FFTW_ESTIMATE);
```

```
         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                          */
#ifndef NO_RESET
            set_c2r_smdata(in_buffer_re, in_buffer_im, rows, cmp_cols);
#endif

            /* Call timer.                                                  */
            start_time = test_time();

            /* Call fftw_execute_split_dft_c2r.                             */
            fftw_execute_split_dft_c2r(plan, in_buffer_re, in_buffer_im,
                                       out_buffer);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }

         /* Ouput time information.                                         */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Ouput result of fftw_execute as an extra test.               */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
               "real matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf \n", i1, i2,
                     out_buffer[i1*cols+i2]);
               }
            }
            nasfprintf(test_output, "\n");
```

```
            nasfflush(test_output);
         }

         /* Free the memory buffers and plan.                          */
         fftw_destroy_plan(plan);
         fftw_free(in_buffer_re);
         fftw_free(in_buffer_im);

      }
   }
   return 1;

} /* End of test_execute_split_dft_c2r_1dipm.                         */
```

10.6.3 Split Complex to Real 2D FFTs.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_c2r_2dop1()                                   *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft_c2r                         *
 *    DATA           out-of-place complex split to real fft.           *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_c2r_2dop1(NASFILE time_output,
                                     NASFILE test_output)
{
   int cmp_cols, cols, rows, i1, i2, runs, run;
   fftw_iodim dims[2], howmany_dims[1];
   time_val start_time, end_time;
   double *in_buffer_re=NULL;
   double *in_buffer_im=NULL;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *out_buffer;

   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output,
         "fftw_execute_split_dft_c2r : cr fft 2d op.\n");
      nasfflush(time_output);
```

```
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_split_dft_c2r : cr fft 2d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   cmp_cols  = (cols/2)+1;

   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {

      /* Malloc enough memory.                                           */
      in_buffer_re = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      in_buffer_im = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      out_buffer = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer_re==NULL || in_buffer_im==NULL || out_buffer==NULL)
         return 0;

      sum_time = 0.0;
      set_c2r3_smdata(in_buffer_re, in_buffer_im, rows, cmp_cols);


      /* Create a multiple 2d cr out of place..                         */
      dims[0].n  = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      dims[1].n  = rows;
      dims[1].is = cols;
      dims[1].os = cols;
      howmany_dims[0].n = 1;
      howmany_dims[0].is = cols*rows;
```

```
         howmany_dims[0].os = cols*rows;
         plan = fftw_plan_guru_split_dft_c2r(2, dims,
            1, howmany_dims,
            in_buffer_re, in_buffer_im, out_buffer,
            FFTW_ESTIMATE);

         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                    */
#ifndef NO_RESET
            set_c2r3_smdata(in_buffer_re, in_buffer_im, rows, cmp_cols);
#endif

            /* Call timer.                                            */
            start_time = test_time();

            /* Call fftw_execute_split_dft_c2r.                       */
            fftw_execute_split_dft_c2r(plan, in_buffer_re, in_buffer_im,
                                       out_buffer);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }


         /* Ouput time information.                                   */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Ouput result to data file.                                */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
               "real matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
```

```
                for(i2=0; i2<cols; i2++)
                {
                   nasfprintf(test_output, "col %d row %d = %lf \n", i1, i2,
                      out_buffer[i1*cols+i2]);
                }
             }
             nasfprintf(test_output, "\n");
             nasfflush(test_output);
          }


          /* Free the memory buffers and plan.                              */
          fftw_destroy_plan(plan);
          fftw_free(in_buffer_re);
          fftw_free(in_buffer_im);
          fftw_free(out_buffer);

       }
    }
    return 1;

} /* End of test_execute_split_dft_c2r_2dop1                               */
```

10.7 Real to Inteleaved Complex Code Examples.

10.7.1 Real to Interleaved Complex 1D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/*************************************************************************
 *                                                                       *
 * test_plan_dft_r2c_2dop()                                              *
 *                                                                       *
 *    TEST FUNCTION: fftw_plan_dft_r2c                                   *
 *    DATA           2d out-of-place real to complex interleaved.        *
 *                                                                       *
 *    Arguments: time_output - file pointer to time file. Each vector    *
 *                             is timed and the estimated time is output *
 *                             to this file.                             *
 *               test_output - Result of each vector is output to this   *
 *                             file.                                     *
 *                                                                       *
 *************************************************************************/
int test_plan_dft_r2c_2dop(NASFILE time_output,
                           NASFILE test_output)
{
   int cols, cmp_cols, rows, i1, i2, runs, run, lengths[2];
   time_val start_time, end_time;
   fftw_complex *out_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *in_buffer;


   /* Set the number of runs for test function.                          */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                  */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_plan_dft_r2c  : rc fft 2d op.\n");
      nasfflush(time_output);
```

```
}
if(test_output != NASNULL)
{
   nasfprintf(test_output, "fftw_plan_dft_r2c  : rc fft 2d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      cmp_cols = cols/2+1;

      /* Malloc enough memory.                                           */
      out_buffer = (fftw_complex*)
         fftw_malloc(cmp_cols*rows*sizeof(fftw_complex));
      in_buffer = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer==NULL || out_buffer==NULL) return 0;


      sum_time = 0.0;
      set_fdata(in_buffer, rows*cols);

      for(run=0; run<runs; run++)
      {
         if(run > 0) fftw_destroy_plan(plan);

         /* Call timer.                                                  */
         start_time = test_time();

         /* Create 2d rc out of place plan.                              */
         lengths[0] = rows;
         lengths[1] =cols;
         plan = fftw_plan_dft_r2c(2, lengths, in_buffer, out_buffer,
            FFTW_ESTIMATE);

         end_time = test_time();
```

```
      sum_time += time_diff(start_time, end_time);
   }

   /* Call fftw_execute.                                          */
   fftw_execute(plan);


   /* Ouput time information.                                     */
   if(time_output != NASNULL)
   {
      time1 = sum_time/runs;
      nasfprintf(time_output,
         "real matrix rows %d cols %d = time %f secs\n",
      rows, cols, time1);
      nasfflush(time_output);
   }

   /* Ouput result to data file.                                  */
   if(test_output != NASNULL)
   {
      nasfprintf(test_output,
         "real matrix - cols %d rows %d:\n", cols, rows);
      for(i1=0; i1<rows; i1++)
      {
         for(i2=0; i2<cmp_cols; i2++)
         {
            nasfprintf(test_output, "col %d row %d = %lf %lf \n", i1, i2,
               out_buffer[i1*cmp_cols+i2][0],
               out_buffer[i1*cmp_cols+i2][1]);
         }
      }
      nasfprintf(test_output, "\n");
      nasfflush(test_output);
   }


   /* Free the memory buffers and plan.                           */
   fftw_destroy_plan(plan);
   fftw_free(out_buffer);
   fftw_free(in_buffer);
}
```

```
    }
    return 1;

} /* End of test_plan_dft_r2c_2dop.                                    */
```

10.7.2 Real to Interleaved Complex Multiple FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_dft_r2c_1dopm()                                         *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_dft_r2c                               *
 *    DATA           out-of-place real to complex interleaved.          *
 *    DIR            multiple 1d fft.                                  *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output*
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_dft_r2c_1dopm(NASFILE time_output,
                               NASFILE test_output)
{
   int cols, rows, i1, i2, runs, run, cmp_cols;
   fftw_iodim dims[1], howmany_dims[1];
   time_val start_time, end_time;
   fftw_complex *out_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *in_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output,
         "fftw_execute_dft_r2c : multiple rc fft 1d op.\n");
```

```
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_dft_r2c : multiple rc fft 1d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   cmp_cols = (cols/2)+1;
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                           */
      out_buffer = (fftw_complex*)
         fftw_malloc(cmp_cols*rows*sizeof(fftw_complex));
      in_buffer = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer==NULL || out_buffer==NULL) return 0;


      sum_time = 0.0;
      set_fdata(in_buffer, rows*cols);


      /* Create a multiple 1d rc in place plan.                          */
      dims[0].n = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      howmany_dims[0].n = rows;
      howmany_dims[0].is = cols;
      howmany_dims[0].os = cmp_cols;
      plan = fftw_plan_guru_dft_r2c(1, dims,
         1, howmany_dims,
         in_buffer, out_buffer,
         FFTW_ESTIMATE);
```

```
         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                    */
#ifndef NO_RESET
            set_fdata(in_buffer, rows*cols);
#endif

            /* Call timer.                                            */
            start_time = test_time();

            /* Call fftw_execute.                                     */
            fftw_execute_dft_r2c(plan, in_buffer, out_buffer);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }


         /* Ouput time information.                                   */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
               rows, cols, time1);
            nasfflush(time_output);
         }

         /* Output result of fftw_execute as an extra test.          */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
               "real matrix - cols %d rows %d\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cmp_cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf %lf\n", i1, i2,
                     out_buffer[i1*cmp_cols+i2][0],
                     out_buffer[i1*cmp_cols+i2][1]);
               }
```

```
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                          */
         fftw_destroy_plan(plan);
         fftw_free(out_buffer);
         fftw_free(in_buffer);
      }
   }
   return 1;

} /* End of test_execute_dft_r2c_1dopm.                               */
```

10.7.3 Real to Interleaved Complex 2D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_plan_dft_r2c_2d_op()                                            *
 *                                                                      *
 *    TEST FUNCTION: fftw_plan_dft_r2c_2d                               *
 *    DATA           2d out-of-place real to complex interleaved.       *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_plan_dft_r2c_2d_op(NASFILE time_output,
                            NASFILE test_output)
{
   int cols, cmp_cols, rows, i1, i2, runs, run;
   time_val start_time, end_time;
   fftw_complex *out_buffer;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *in_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_plan_dft_r2c_2d  : rc fft 2d op.\n");
      nasfflush(time_output);
   }
   if(test_output != NASNULL)
```

```
{
   nasfprintf(test_output, "fftw_plan_dft_r2c_2d  : rc fft 2d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      cmp_cols = cols/2+1;

      /* Malloc enough memory.                                           */
      out_buffer = (fftw_complex*)
         fftw_malloc(cmp_cols*rows*sizeof(fftw_complex));
      in_buffer = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer==NULL || out_buffer==NULL) return 0;


      sum_time = 0.0;
      set_fdata(in_buffer, rows*cols);

      for(run=0; run<runs; run++)
      {
         if(run > 0) fftw_destroy_plan(plan);

         /* Call timer.                                                  */
         start_time = test_time();

         /* Create 2d rc out of place plan.                              */
         plan = fftw_plan_dft_r2c_2d(rows, cols, in_buffer, out_buffer,
            FFTW_ESTIMATE);

         end_time = test_time();

         sum_time += time_diff(start_time, end_time);
      }
```

```
         /* Call fftw_execute.                                          */
         fftw_execute(plan);


         /* Ouput time information.                                     */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Ouput result to data file.                                  */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
               "real matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cmp_cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf %lf \n", i1, i2,
                     out_buffer[i1*cmp_cols+i2][0],
                     out_buffer[i1*cmp_cols+i2][1]);
               }
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                           */
         fftw_destroy_plan(plan);
         fftw_free(out_buffer);
         fftw_free(in_buffer);
      }
   }
   return 1;

} /* End of test_plan_dft_r2c_2d_op.                                    */
```

10.8 Real to Split Complex Code Examples.

10.8.1 Real to Split Complex 1D FFT.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_r2c_1dip1()                                   *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft_r2c                         *
 *    TEST 1dipf1:   real to split  complex.                            *
 *                   1 dimensional.                                     *
 *                   in-place data.                                     *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output*
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_r2c_1dip1(NASFILE time_output,
                                     NASFILE test_output)
{
   int length, cmp_length, i1, runs, run;
   fftw_iodim dims[1], howmany_dims[1];
   time_val start_time, end_time;
   double *out_buffer_re=NULL;
   double *out_buffer_im=NULL;
   double *real_input=NULL
   float sum_time, time1;
   fftw_plan plan=NULL;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
```

```
if(time_output != NASNULL)
{
   nasfprintf(time_output,
      "fftw_execute_split_dft_r2c : rc fft 1d ip.\n");
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_split_dft_r2c : rc fft 1d ip.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(length=para.vec_start; length<=para.vec_end; length*=para.vec_inc)
{
   cmp_length = length / 2 + 1;

   /* Malloc enough memory.                                              */
   real_input = (double*)
      fftw_malloc(length*sizeof(double));
   out_buffer_im = (double*)
      fftw_malloc(cmp_length*sizeof(double));
   if(real_input==NULL || out_buffer_im==NULL) return 0;
   out_buffer_re = real_input;

   sum_time = 0.0;
   set_fdata(real_input, length);


   /* Define an in-place 1d fft.                                         */
   dims[0].n = length;
   dims[0].is = 1;
   dims[0].os = 1;
   howmany_dims[0].n = 1;
   howmany_dims[0].is = length;
   howmany_dims[0].os = length;
   plan = fftw_plan_guru_split_dft_r2c(1, dims,
      0, howmany_dims,
      real_input, out_buffer_re, out_buffer_im,
      FFTW_ESTIMATE);
```

```
      for(run=0; run<runs; run++)
      {
         /* Set the input data.                                          */
#ifndef NO_RESET
         set_fdata(real_input, length);
#endif
         /* Call timer.                                                  */
         start_time = test_time();

         /* Call fftw_execute.                                           */
         fftw_execute_split_dft_r2c(plan, real_input,
            out_buffer_re, out_buffer_im);

         end_time = test_time();
         sum_time += time_diff(start_time, end_time);
      }


      /* Ouput time information.                                         */
      if(time_output != NASNULL)
      {
         time1 = sum_time/runs;
         nasfprintf(time_output,
            "vector real length %d cmp length %d - time %f secs\n",
            length, cmp_length, time1);
         nasfflush(time_output);
      }

      /* Ouput result off fft to data file just for an extra check.     */
      if(test_output != NASNULL)
      {
         nasfprintf(test_output, "vector real length %d cmp length %d:\n",
            length, cmp_length);
         for(i1=0; i1<cmp_length; i1++)
         {
            nasfprintf(test_output, "cell %d = %lf %lf\n", i1,
               out_buffer_re[i1], out_buffer_im[i1]);
         }
         nasfprintf(test_output, "\n");
         nasfflush(test_output);
```

```
      }


      /* Free the memory buffers and plan.                              */
      fftw_free(out_buffer_im);
      fftw_destroy_plan(plan);
      fftw_free(real_input);


   }

   return 1;

} /* End of test_execute_split_dft_r2c_1dip1.                           */
```

10.8.2 Split Complex to Complex Multiple FFTs.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_r2c_1dopm()                                   *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft_r2c                         *
 *    DATA           out-of-place real to complex split.                *
 *    DIR            multiple 1d fft.                                   *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output*
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *               fptr        - File pointer to opened file with write   *
 *                             permission.                              *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_r2c_1dopm(NASFILE time_output,
                                     NASFILE test_output,
                                     FILE *fptr)
{
   int cols, rows, i1, i2, runs, run, cmp_cols;
   fftw_iodim dims[1], howmany_dims[1];
   time_val start_time, end_time;
   double *out_buffer_re=NULL;
   double *out_buffer_im=NULL;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *in_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
```

```
if(time_output != NASNULL)
{
   nasfprintf(time_output,
      "fftw_execute_split_dft_r2c : multiple rc fft 1d op.\n");
   nasfflush(time_output);
}
if(test_output != NASNULL)
{
   nasfprintf(test_output,
      "fftw_execute_split_dft_r2c : multiple rc fft 1d op.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   cmp_cols = (cols/2)+1;
   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                           */
      out_buffer_re = (double*)
         fftw_malloc(cmp_cols*rows*sizeof(double));
      out_buffer_im = (double*)
         fftw_malloc(cmp_cols*rows*sizeof(double));
      in_buffer = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(in_buffer==NULL || out_buffer_re==NULL ||
         out_buffer_im==NULL) return 0;


      sum_time = 0.0;
      set_fdata(in_buffer, rows*cols);


      /* Create a multiple 1d rc in place plan.                          */
      dims[0].n = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      howmany_dims[0].n = rows;
```

```
         howmany_dims[0].is = cols;
         howmany_dims[0].os = cmp_cols;
         plan = fftw_plan_guru_split_dft_r2c(1, dims,
            1, howmany_dims,
            in_buffer, out_buffer_re, out_buffer_im,
            FFTW_ESTIMATE);

         /* Print out plan info to a file.                              */
         fftw_fprint_plan(plan, fptr);

         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                      */
#ifndef NO_RESET
            set_fdata(in_buffer, rows*cols);
#endif
            /* Call timer.                                              */
            start_time = test_time();

            /* Call fftw_execute_split_dft_r2c.                         */
            fftw_execute_split_dft_r2c(plan, in_buffer,
               out_buffer_re, out_buffer_im);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }


         /* Ouput time information.                                     */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
               rows, cols, time1);
            nasfflush(time_output);
         }

         /* Output result of fftw_execute as an extra test.           */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
```

```
                "real matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cmp_cols; i2++)
               {
                  nasfprintf(test_output, "col %d row %d = %lf %lf\n", i1, i2,
                     out_buffer_re[i1*cmp_cols+i2],
                     out_buffer_im[i1*cmp_cols+i2]);
               }
            }
            nasfprintf(test_output, "\n");
            nasfflush(test_output);
         }


         /* Free the memory buffers and plan.                              */
         fftw_free(out_buffer_re);
         fftw_free(out_buffer_im);
         fftw_destroy_plan(plan);
         fftw_free(in_buffer);
      }
   }
   return 1;

} /* End of test_execute_split_dft_r2c_1dopm.                              */
```

10.8.3 Real to Split Complex 2D FFTs.

```
#include <stdio.h>
#include "fftw.h"
#include "test_fftw.h"

/************************************************************************
 *                                                                      *
 * test_execute_split_dft_r2c_2dip1()                                   *
 *                                                                      *
 *    TEST FUNCTION: fftw_execute_split_dft_r2c                         *
 *    DATA           in-place real to complex split fft.                *
 *                                                                      *
 *    Arguments: time_output - file pointer to time file. Each vector   *
 *                             is timed and the estimated time is output *
 *                             to this file.                            *
 *               test_output - Result of each vector is output to this  *
 *                             file.                                    *
 *                                                                      *
 ************************************************************************/
int test_execute_split_dft_r2c_2dip1(NASFILE time_output,
                                     NASFILE test_output)
{
   int cmp_cols, cols, rows, i1, i2, runs, run;
   fftw_iodim dims[2], howmany_dims[1];
   time_val start_time, end_time;
   double *out_buffer_re=NULL;
   double *out_buffer_im=NULL;
   float sum_time, time1;
   fftw_plan plan=NULL;
   double *in_buffer;


   /* Set the number of runs for test function.                         */
   if(time_output == NASNULL) runs = 1;
   else                       runs = para.runs;

   /* Output test name.                                                 */
   if(time_output != NASNULL)
   {
      nasfprintf(time_output, "fftw_execute_split_dft_r2c : rc fft 2d ip.\n");
      nasfflush(time_output);
```

```
}
if(test_output != NASNULL)
{
   nasfprintf(test_output, "fftw_execute_split_dft_r2c : rc fft 2d ip.\n");
   nasfflush(test_output);
}


/* Loop though required vector lengths.                                  */
for(cols=para.mat_cols_start; cols<=para.mat_cols_end;
    cols*=para.mat_col_inc)
{
   cmp_cols  = (cols/2)+1;

   for(rows=para.mat_rows_start; rows<=para.mat_rows_end;
       rows*=para.mat_row_inc)
   {
      /* Malloc enough memory.                                           */
      out_buffer_re = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      out_buffer_im = (double*)
         fftw_malloc(cols*rows*sizeof(double));
      if(out_buffer_re==NULL || out_buffer_im==NULL) return 0;
      in_buffer = (double*)out_buffer_re;

      sum_time = 0.0;
      set_ddata(in_buffer, rows*cols);
      set_ddata(out_buffer_im, rows*cols);


      /* Create a multiple 2d rc in place plan.                          */
      dims[0].n  = cols;
      dims[0].is = 1;
      dims[0].os = 1;
      dims[1].n  = rows;
      dims[1].is = cols;
      dims[1].os = cols;
      howmany_dims[0].n = 1;
      howmany_dims[0].is = cols*rows;
      howmany_dims[0].os = cols*rows;
      plan = fftw_plan_guru_split_dft_r2c(2, dims,
         1, howmany_dims,
```

```
            in_buffer, out_buffer_re, out_buffer_im,
            FFTW_ESTIMATE);

         /* Print out plan info to the screen.                          */
         fftw_print_plan(plan);

         for(run=0; run<runs; run++)
         {
            /* Set the input data.                                      */
#ifndef NO_RESET
            set_ddata(in_buffer, rows*cols);
            set_ddata(out_buffer_im, rows*cols);
#endif

            /* Call timer.                                              */
            start_time = test_time();

            /* Call fftw_execute_split_dft_r2c.                         */
            fftw_execute_split_dft_r2c(plan, in_buffer,
               out_buffer_re, out_buffer_im);

            end_time = test_time();
            sum_time += time_diff(start_time, end_time);
         }
         /* Ouput time information.                                     */
         if(time_output != NASNULL)
         {
            time1 = sum_time/runs;
            nasfprintf(time_output,
               "real matrix rows %d cols %d = time %f secs\n",
            rows, cols, time1);
            nasfflush(time_output);
         }

         /* Ouput result to data file.                                  */
         if(test_output != NASNULL)
         {
            nasfprintf(test_output,
               "real matrix - cols %d rows %d:\n", cols, rows);
            for(i1=0; i1<rows; i1++)
            {
               for(i2=0; i2<cmp_cols; i2++)
```

```
                {
                   nasfprintf(test_output, "col %d row %d = %lf %lf \n", i1, i2,
                      out_buffer_re[i1*cols+i2],
                      out_buffer_im[i1*cols+i2]);
                }
             }
             nasfprintf(test_output, "\n");
             nasfflush(test_output);
          }


          /* Free the memory buffers and plan.                              */
          fftw_free(out_buffer_re);
          fftw_free(out_buffer_im);
          fftw_destroy_plan(plan);
       }
    }
    return 1;

} /* End of test_execute_split_dft_r2c_2dipf1                              */
```