# FFTW User's Manual

For version 2.1.5, 16 March 2003

Matteo Frigo
Steven G. Johnson

# Table of Contents

# 1 Introduction

This manual documents version 2.1.5 of FFTW, the *Fastest Fourier Transform in the West*. FFTW is a comprehensive collection of fast C routines for computing the discrete Fourier transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. FFTW also includes parallel transforms for both shared- and distributed-memory systems. We assume herein that the reader is already familiar with the properties and uses of the DFT that are relevant to her application. Otherwise, see e.g. *The Fast Fourier Transform* by E. O. Brigham (Prentice-Hall, Englewood Cliffs, NJ, 1974). Our web page also has links to FFT-related information online.

FFTW is usually faster (and sometimes much faster) than all other freely-available Fourier transform programs found on the Net. For transforms whose size is a power of two, it compares favorably with the FFT codes in Sun's Performance Library and IBM's ESSL library, which are targeted at specific machines. Moreover, FFTW's performance is *portable*. Indeed, FFTW is unique in that it automatically adapts itself to your machine, your cache, the size of your memory, the number of registers, and all the other factors that normally make it impossible to optimize a program for more than one machine. An extensive comparison of FFTW's performance with that of other Fourier transform codes has been made. The results are available on the Web at the benchFFT home page.

In order to use FFTW effectively, you need to understand one basic concept of FFTW's internal structure. FFTW does not used a fixed algorithm for computing the transform, but it can adapt the DFT algorithm to details of the underlying hardware in order to achieve best performance. Hence, the computation of the transform is split into two phases. First, FFTW's *planner* is called, which "learns" the fastest way to compute the transform on your machine. The planner produces a data structure called a *plan* that contains this information. Subsequently, the plan is passed to FFTW's *executor*, along with an array of input data. The executor computes the actual transform, as dictated by the plan. The plan can be reused as many times as needed. In typical high-performance applications, many transforms of the same size are computed, and consequently a relatively-expensive initialization of this sort is acceptable. On the other hand, if you need a single transform of a given size, the one-time cost of the planner becomes significant. For this case, FFTW provides fast planners based on heuristics or on previously computed plans.

The pattern of planning/execution applies to all four operation modes of FFTW, that is, I) one-dimensional complex transforms (FFTW), II) multi-dimensional complex transforms (FFTWND), III) one-dimensional transforms of real data (RFFTW), IV) multi-dimensional transforms of real data (RFFTWND). Each mode comes with its own planner and executor.

Besides the automatic performance adaptation performed by the planner, it is also possible for advanced users to customize FFTW for their special needs. As distributed, FFTW works most efficiently for arrays whose size can be factored into small primes (2, 3, 5, and 7), and uses a slower general-purpose routine for other factors. FFTW, however, comes with a code generator that can produce fast C programs for any particular array size you may care about. For example, if you need transforms of size $513 = 19 \cdot 3^3$, you can customize FFTW to support the factor 19 efficiently.

FFTW can exploit multiple processors if you have them. FFTW comes with a shared-memory implementation on top of POSIX (and similar) threads, as well as a distributed-memory implementation based on MPI. We also provide an experimental parallel implementation written in Cilk, *the superior programming tool of choice for discriminating hackers* (Olin Shivers). (See the Cilk home page.)

For more information regarding FFTW, see the paper, “The Fastest Fourier Transform in the West,” by M. Frigo and S. G. Johnson, which is the technical report MIT-LCS-TR-728 (Sep. ’97). See also, “FFTW: An Adaptive Software Architecture for the FFT,” by M. Frigo and S. G. Johnson, which appeared in the 23rd International Conference on Acoustics, Speech, and Signal Processing (*Proc. ICASSP 1998* **3**, p. 1381). The code generator is described in the paper “A Fast Fourier Transform Compiler”, by M. Frigo, to appear in the *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, May 1999*. These papers, along with the latest version of FFTW, the FAQ, benchmarks, and other links, are available at the FFTW home page. The current version of FFTW incorporates many good ideas from the past thirty years of FFT literature. In one way or another, FFTW uses the Cooley-Tukey algorithm, the Prime Factor algorithm, Rader’s algorithm for prime sizes, and the split-radix algorithm (with a variation due to Dan Bernstein). Our code generator also produces new algorithms that we do not yet completely understand. The reader is referred to the cited papers for the appropriate references.

The rest of this manual is organized as follows. We first discuss the sequential (one-processor) implementation. We start by describing the basic features of FFTW in Chapter 2 [Tutorial], page 3. This discussion includes the storage scheme of multi-dimensional arrays (Section 2.5 [Multi-dimensional Array Format], page 11) and FFTW’s mechanisms for storing plans on disk (Section 2.6 [Words of Wisdom], page 13). Next, Chapter 3 [FFTW Reference], page 17 provides comprehensive documentation of all FFTW’s features. Parallel transforms are discussed in their own chapter Chapter 4 [Parallel FFTW], page 37. Fortran programmers can also use FFTW, as described in Chapter 5 [Calling FFTW from Fortran], page 51. Chapter 6 [Installation and Customization], page 55 explains how to install FFTW in your computer system and how to adapt FFTW to your needs. License and copyright information is given in Chapter 8 [License and Copyright], page 63. Finally, we thank all the people who helped us in Chapter 7 [Acknowledgments], page 61.

# 2 Tutorial

This chapter describes the basic usage of FFTW, i.e., how to compute the Fourier transform of a single array. This chapter tells the truth, but not the *whole* truth. Specifically, FFTW implements additional routines and flags, providing extra functionality, that are not documented here. See Chapter 3 [FFTW Reference], page 17, for more complete information. (Note that you need to compile and install FFTW before you can use it in a program. See Chapter 6 [Installation and Customization], page 55, for the details of the installation.)

Here, we assume a default installation of FFTW. In some installations (particulary from binary packages), the FFTW header files and libraries are prefixed with '`d`' or '`s`' to indicate versions in double or single precision, respectively. The usage of FFTW in that case is the same, except that `#include` directives and link commands must use the appropriate prefix. See Section 6.3 [Installing FFTW in both single and double precision], page 57, for more information.

This tutorial chapter is structured as follows. Section 2.1 [Complex One-dimensional Transforms Tutorial], page 3 describes the basic usage of the one-dimensional transform of complex data. Section 2.2 [Complex Multi-dimensional Transforms Tutorial], page 4 describes the basic usage of the multi-dimensional transform of complex data. Section 2.3 [Real One-dimensional Transforms Tutorial], page 6 describes the one-dimensional transform of real data and its inverse. Finally, Section 2.4 [Real Multi-dimensional Transforms Tutorial], page 7 describes the multi-dimensional transform of real data and its inverse. We recommend that you read these sections in the order that they are presented. We then discuss two topics in detail. In Section 2.5 [Multi-dimensional Array Format], page 11, we discuss the various alternatives for storing multi-dimensional arrays in memory. Section 2.6 [Words of Wisdom], page 13 shows how you can save FFTW's plans for future use.

## 2.1 Complex One-dimensional Transforms Tutorial

The basic usage of FFTW is simple. A typical call to FFTW looks like:

```
#include <fftw.h>
...
{
     fftw_complex in[N], out[N];
     fftw_plan p;
     ...
     p = fftw_create_plan(N, FFTW_FORWARD, FFTW_ESTIMATE);
     ...
     fftw_one(p, in, out);
     ...
     fftw_destroy_plan(p);
}
```

The first thing we do is to create a *plan*, which is an object that contains all the data that FFTW needs to compute the FFT, using the following function:

```
fftw_plan fftw_create_plan(int n, fftw_direction dir, int flags);
```

The first argument, `n`, is the size of the transform you are trying to compute. The size `n` can be any positive integer, but sizes that are products of small factors are transformed most efficiently. The second argument, dir, can be either FFTW_FORWARD or FFTW_BACKWARD, and indicates the direction of the transform you are interested in. Alternatively, you can use the sign of the exponent in the transform, $-1$ or $+1$, which corresponds to FFTW_FORWARD or FFTW_BACKWARD respectively. The flags argument is either FFTW_MEASURE or FFTW_ESTIMATE. FFTW_MEASURE means that FFTW actually runs and measures the execution time of several FFTs in order to find the best way to compute the transform of size n. This may take some time, depending on your installation and on the precision of the timer in your machine. FFTW_ESTIMATE, on the contrary, does not run any computation, and just builds a reasonable plan, which may be sub-optimal. In other words, if your program performs many transforms of the same size and initialization time is not important, use FFTW_MEASURE; otherwise use the estimate. (A compromise between these two extremes exists. See Section 2.6 [Words of Wisdom], page 13.)

Once the plan has been created, you can use it as many times as you like for transforms on arrays of the same size. When you are done with the plan, you deallocate it by calling fftw_destroy_plan(plan).

The transform itself is computed by passing the plan along with the input and output arrays to fftw_one:

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

Note that the transform is out of place: in and out must point to distinct arrays. It operates on data of type fftw_complex, a data structure with real (in[i].re) and imaginary (in[i].im) floating-point components. The in and out arrays should have the length specified when the plan was created. An alternative function, fftw, allows you to efficiently perform multiple and/or strided transforms (see Chapter 3 [FFTW Reference], page 17).

The DFT results are stored in-order in the array out, with the zero-frequency (DC) component in out[0]. The array in is not modified. Users should note that FFTW computes an unnormalized DFT, the sign of whose exponent is given by the dir parameter of fftw_create_plan. Thus, computing a forward followed by a backward transform (or vice versa) results in the original array scaled by n. See Section 3.2.5 [What FFTW Really Computes], page 21, for the definition of DFT.

A program using FFTW should be linked with -lfftw -lm on Unix systems, or with the FFTW and standard math libraries in general.

## 2.2 Complex Multi-dimensional Transforms Tutorial

FFTW can also compute transforms of any number of dimensions (*rank*). The syntax is similar to that for the one-dimensional transforms, with 'fftw_' replaced by 'fftwnd_' (which stands for "fftw in N dimensions").

As before, we #include <fftw.h> and create a plan for the transforms, this time of type fftwnd_plan:

```
fftwnd_plan fftwnd_create_plan(int rank, const int *n,
                               fftw_direction dir, int flags);
```

rank is the dimensionality of the array, and can be any non-negative integer. The next argument, n, is a pointer to an integer array of length rank containing the (positive) sizes of each dimension of the array. (Note that the array will be stored in row-major order. See

Section 2.5 [Multi-dimensional Array Format], page 11, for information on row-major order.) The last two parameters are the same as in `fftw_create_plan`. We now, however, have an additional possible flag, `FFTW_IN_PLACE`, since `fftwnd` supports true in-place transforms. Multiple flags are combined using a bitwise *or* ('`|`'). (An *in-place* transform is one in which the output data overwrite the input data. It thus requires half as much memory as—and is often faster than—its opposite, an *out-of-place* transform.)

For two- and three-dimensional transforms, FFTWND provides alternative routines that accept the sizes of each dimension directly, rather than indirectly through a rank and an array of sizes. These are otherwise identical to `fftwnd_create_plan`, and are sometimes more convenient:

```
fftwnd_plan fftw2d_create_plan(int nx, int ny,
                               fftw_direction dir, int flags);
fftwnd_plan fftw3d_create_plan(int nx, int ny, int nz,
                               fftw_direction dir, int flags);
```

Once the plan has been created, you can use it any number of times for transforms of the same size. When you do not need a plan anymore, you can deallocate the plan by calling `fftwnd_destroy_plan(plan)`.

Given a plan, you can compute the transform of an array of data by calling:

```
void fftwnd_one(fftwnd_plan plan, fftw_complex *in, fftw_complex *out);
```

Here, `in` and `out` point to multi-dimensional arrays in row-major order, of the size specified when the plan was created. In the case of an in-place transform, the `out` parameter is ignored and the output data are stored in the `in` array. The results are stored in-order, unnormalized, with the zero-frequency component in `out[0]`. A forward followed by a backward transform (or vice-versa) yields the original data multiplied by the size of the array (i.e. the product of the dimensions). See Section 3.3.4 [What FFTWND Really Computes], page 25, for a discussion of what FFTWND computes.

For example, code to perform an in-place FFT of a three-dimensional array might look like:

```
#include <fftw.h>
...
{
     fftw_complex in[L][M][N];
     fftwnd_plan p;
     ...
     p = fftw3d_create_plan(L, M, N, FFTW_FORWARD,
                            FFTW_MEASURE | FFTW_IN_PLACE);
     ...
     fftwnd_one(p, &in[0][0][0], NULL);
     ...
     fftwnd_destroy_plan(p);
}
```

Note that `in` is a statically-declared array, which is automatically in row-major order, but we must take the address of the first element in order to fit the type expected by `fftwnd_one`. (See Section 2.5 [Multi-dimensional Array Format], page 11.)

## 2.3 Real One-dimensional Transforms Tutorial

If the input data are purely real, you can save roughly a factor of two in both time and storage by using the *rfftw* transforms, which are FFTs specialized for real data. The output of a such a transform is a *halfcomplex* array, which consists of only half of the complex DFT amplitudes (since the negative-frequency amplitudes for real data are the complex conjugate of the positive-frequency amplitudes).

In exchange for these speed and space advantages, the user sacrifices some of the simplicity of FFTW's complex transforms. First of all, to allow maximum performance, the output format of the one-dimensional real transforms is different from that used by the multi-dimensional transforms. Second, the inverse transform (halfcomplex to real) has the side-effect of destroying its input array. Neither of these inconveniences should pose a serious problem for users, but it is important to be aware of them. (Both the inconvenient output format and the side-effect of the inverse transform can be ameliorated for one-dimensional transforms, at the expense of some performance, by using instead the multi-dimensional transform routines with a rank of one.)

The computation of the plan is similar to that for the complex transforms. First, you `#include <rfftw.h>`. Then, you create a plan (of type `rfftw_plan`) by calling:

```
rfftw_plan rfftw_create_plan(int n, fftw_direction dir, int flags);
```

`n` is the length of the *real* array in the transform (even for halfcomplex-to-real transforms), and can be any positive integer (although sizes with small factors are transformed more efficiently). `dir` is either `FFTW_REAL_TO_COMPLEX` or `FFTW_COMPLEX_TO_REAL`. The `flags` parameter is the same as in `fftw_create_plan`.

Once created, a plan can be used for any number of transforms, and is deallocated when you are done with it by calling `rfftw_destroy_plan(plan)`.

Given a plan, a real-to-complex or complex-to-real transform is computed by calling:

```
void rfftw_one(rfftw_plan plan, fftw_real *in, fftw_real *out);
```

(Note that `fftw_real` is an alias for the floating-point type for which FFTW was compiled.) Depending upon the direction of the plan, either the input or the output array is halfcomplex, and is stored in the following format:

$$r_0, r_1, r_2, \ldots, r_{n/2}, i_{(n+1)/2-1}, \ldots, i_2, i_1$$

Here, $r_k$ is the real part of the $k$th output, and $i_k$ is the imaginary part. (We follow here the C convention that integer division is rounded down, e.g. $7/2 = 3$.) For a halfcomplex array `hc[]`, the $k$th component has its real part in `hc[k]` and its imaginary part in `hc[n-k]`, with the exception of `k == 0` or `n/2` (the latter only if n is even)—in these two cases, the imaginary part is zero due to symmetries of the real-complex transform, and is not stored. Thus, the transform of `n` real values is a halfcomplex array of length `n`, and vice versa.[1] This is actually only half of the DFT spectrum of the data. Although the other half can be obtained by complex conjugation, it is not required by many applications such as convolution and filtering.

[1] The output for the multi-dimensional rfftw is a more-conventional array of `fftw_complex` values, but the format here permitted us greater efficiency in one dimension.

Like the complex transforms, the RFFTW transforms are unnormalized, so a forward followed by a backward transform (or vice-versa) yields the original data scaled by the length of the array, `n`.

Let us reiterate here our warning that an `FFTW_COMPLEX_TO_REAL` transform has the side-effect of destroying its (halfcomplex) input. The `FFTW_REAL_TO_COMPLEX` transform, however, leaves its (real) input untouched, just as you would hope.

As an example, here is an outline of how you might use RFFTW to compute the power spectrum of a real array (i.e. the squares of the absolute values of the DFT amplitudes):

```
#include <rfftw.h>
...
{
     fftw_real in[N], out[N], power_spectrum[N/2+1];
     rfftw_plan p;
     int k;
     ...
     p = rfftw_create_plan(N, FFTW_REAL_TO_COMPLEX, FFTW_ESTIMATE);
     ...
     rfftw_one(p, in, out);
     power_spectrum[0] = out[0]*out[0];  /* DC component */
     for (k = 1; k < (N+1)/2; ++k)  /* (k < N/2 rounded up) */
          power_spectrum[k] = out[k]*out[k] + out[N-k]*out[N-k];
     if (N % 2 == 0) /* N is even */
          power_spectrum[N/2] = out[N/2]*out[N/2];  /* Nyquist freq. */
     ...
     rfftw_destroy_plan(p);
}
```

Programs using RFFTW should link with `-lrfftw -lfftw -lm` on Unix, or with the FFTW, RFFTW, and math libraries in general.

## 2.4 Real Multi-dimensional Transforms Tutorial

FFTW includes multi-dimensional transforms for real data of any rank. As with the one-dimensional real transforms, they save roughly a factor of two in time and storage over complex transforms of the same size. Also as in one dimension, these gains come at the expense of some increase in complexity—the output format is different from the one-dimensional RFFTW (and is more similar to that of the complex FFTW) and the inverse (complex to real) transforms have the side-effect of overwriting their input data.

To use the real multi-dimensional transforms, you first `#include <rfftw.h>` and then create a plan for the size and direction of transform that you are interested in:

```
rfftwnd_plan rfftwnd_create_plan(int rank, const int *n,
                                 fftw_direction dir, int flags);
```

The first two parameters describe the size of the real data (not the halfcomplex data, which will have different dimensions). The last two parameters are the same as those for `rfftw_create_plan`. Just as for fftwnd, there are two alternate versions of this routine, `rfftw2d_create_plan` and `rfftw3d_create_plan`, that are sometimes more convenient for two- and three-dimensional transforms. Also as in fftwnd, rfftwnd supports true in-place transforms, specified by including `FFTW_IN_PLACE` in the flags.

Once created, a plan can be used for any number of transforms, and is deallocated by calling `rfftwnd_destroy_plan(plan)`.

Given a plan, the transform is computed by calling one of the following two routines:

```
void rfftwnd_one_real_to_complex(rfftwnd_plan plan,
                                 fftw_real *in, fftw_complex *out);
void rfftwnd_one_complex_to_real(rfftwnd_plan plan,
                                 fftw_complex *in, fftw_real *out);
```

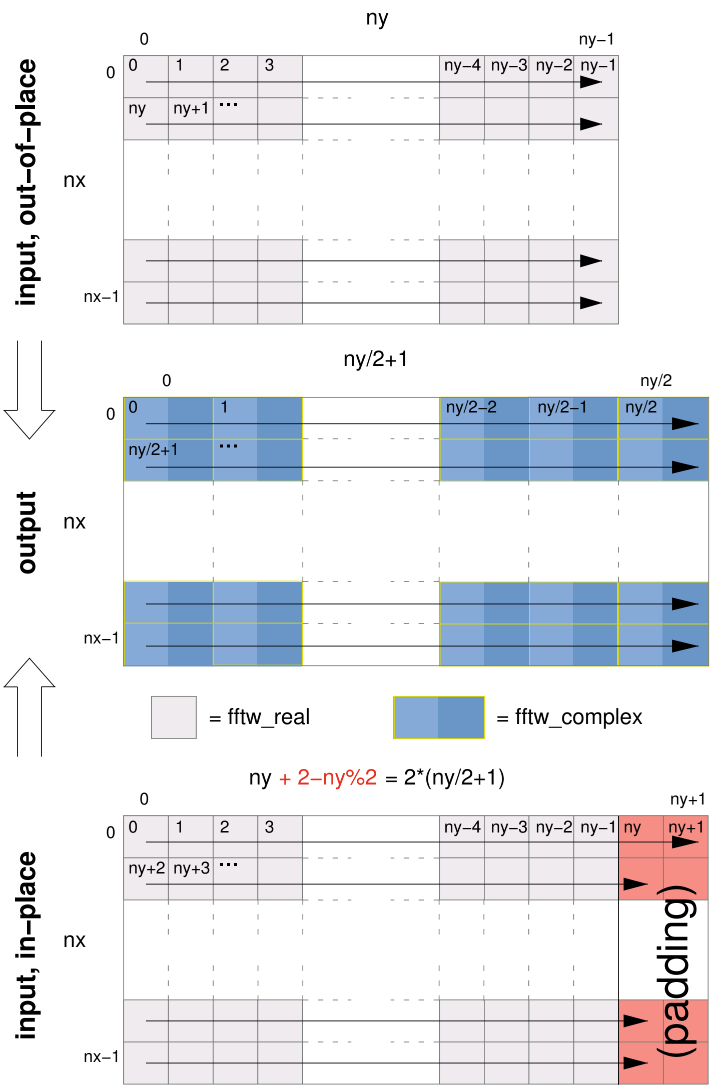
As is clear from their names and parameter types, the former function is for `FFTW_REAL_TO_COMPLEX` transforms and the latter is for `FFTW_COMPLEX_TO_REAL` transforms. (We could have used only a single routine, since the direction of the transform is encoded in the plan, but we wanted to correctly express the datatypes of the parameters.) The latter routine, as we discuss elsewhere, has the side-effect of overwriting its input (except when the rank of the array is one). In both cases, the `out` parameter is ignored for in-place transforms.

The format of the complex arrays deserves careful attention. Suppose that the real data has dimensions $n_1 \times n_2 \times \cdots \times n_d$ (in row-major order). Then, after a real-to-complex transform, the output is an $n_1 \times n_2 \times \cdots \times (n_d/2+1)$ array of `fftw_complex` values in row-major order, corresponding to slightly over half of the output of the corresponding complex transform. (Note that the division is rounded down.) The ordering of the data is otherwise exactly the same as in the complex case. (In principle, the output could be exactly half the size of the complex transform output, but in more than one dimension this requires too complicated a format to be practical.) Note that, unlike the one-dimensional RFFTW, the real and imaginary parts of the DFT amplitudes are here stored together in the natural way.

Since the complex data is slightly larger than the real data, some complications arise for in-place transforms. In this case, the final dimension of the real data must be padded with extra values to accommodate the size of the complex data—two extra if the last dimension is even and one if it is odd. That is, the last dimension of the real data must physically contain $2(n_d/2+1)$ `fftw_real` values (exactly enough to hold the complex data). This physical array size does not, however, change the *logical* array size—only $n_d$ values are actually stored in the last dimension, and $n_d$ is the last dimension passed to `rfftwnd_create_plan`.

For example, consider the transform of a two-dimensional real array of size `nx` by `ny`. The output of the `rfftwnd` transform is a two-dimensional complex array of size `nx` by `ny/2+1`, where the `y` dimension has been cut nearly in half because of redundancies in the output. Because `fftw_complex` is twice the size of `fftw_real`, the output array is slightly bigger than the input array. Thus, if we want to compute the transform in place, we must *pad* the input array so that it is of size `nx` by `2*(ny/2+1)`. If `ny` is even, then there are two padding elements at the end of each row (which need not be initialized, as they are only used for output).

The following illustration depicts the input and output arrays just described, for both the out-of-place and in-place transforms (with the arrows indicating consecutive memory locations):



The RFFTWND transforms are unnormalized, so a forward followed by a backward transform will result in the original data scaled by the number of real data elements—that is, the product of the (logical) dimensions of the real data.

Below, we illustrate the use of RFFTWND by showing how you might use it to compute the (cyclic) convolution of two-dimensional real arrays `a` and `b` (using the identity that a convolution corresponds to a pointwise product of the Fourier transforms). For variety, in-place transforms are used for the forward FFTs and an out-of-place transform is used for the inverse transform.

```
#include <rfftw.h>
...
{
     fftw_real a[M][2*(N/2+1)], b[M][2*(N/2+1)], c[M][N];
     fftw_complex *A, *B, C[M][N/2+1];
     rfftwnd_plan p, pinv;
     fftw_real scale = 1.0 / (M * N);
     int i, j;
     ...
     p    = rfftw2d_create_plan(M, N, FFTW_REAL_TO_COMPLEX,
                                FFTW_ESTIMATE | FFTW_IN_PLACE);
     pinv = rfftw2d_create_plan(M, N, FFTW_COMPLEX_TO_REAL,
                                FFTW_ESTIMATE);

     /* aliases for accessing complex transform outputs: */
     A = (fftw_complex*) &a[0][0];
     B = (fftw_complex*) &b[0][0];
     ...
     for (i = 0; i < M; ++i)
          for (j = 0; j < N; ++j) {
               a[i][j] = ... ;
               b[i][j] = ... ;
          }
     ...
     rfftwnd_one_real_to_complex(p, &a[0][0], NULL);
     rfftwnd_one_real_to_complex(p, &b[0][0], NULL);

     for (i = 0; i < M; ++i)
          for (j = 0; j < N/2+1; ++j) {
               int ij = i*(N/2+1) + j;
               C[i][j].re = (A[ij].re * B[ij].re
                             - A[ij].im * B[ij].im) * scale;
               C[i][j].im = (A[ij].re * B[ij].im
                             + A[ij].im * B[ij].re) * scale;
          }

     /* inverse transform to get c, the convolution of a and b;
        this has the side effect of overwriting C */
     rfftwnd_one_complex_to_real(pinv, &C[0][0], &c[0][0]);
     ...
     rfftwnd_destroy_plan(p);
     rfftwnd_destroy_plan(pinv);
}
```

We access the complex outputs of the in-place transforms by casting each real array to a `fftw_complex` pointer. Because this is a “flat” pointer, we have to compute the row-major index `ij` explicitly in the convolution product loop. In order to normalize the convolution, we must multiply by a scale factor—we can do so either before or after the inverse transform, and choose the former because it obviates the necessity of an additional loop. Notice the limits of the loops and the dimensions of the various arrays.

As with the one-dimensional RFFTW, an out-of-place `FFTW_COMPLEX_TO_REAL` transform has the side-effect of overwriting its input array. (The real-to-complex transform, on the other hand, leaves its input array untouched.) If you use RFFTWND for a rank-one transform, however, this side-effect does not occur. Because of this fact (and the simpler output format), users may find the RFFTWND interface more convenient than RFFTW for one-dimensional transforms. However, RFFTWND in one dimension is slightly slower than RFFTW because RFFTWND uses an extra buffer array internally.

## 2.5 Multi-dimensional Array Format

This section describes the format in which multi-dimensional arrays are stored. We felt that a detailed discussion of this topic was necessary, since it is often a source of confusion among users and several different formats are common. Although the comments below refer to `fftwnd`, they are also applicable to the `rfftwnd` routines.

### 2.5.1 Row-major Format

The multi-dimensional arrays passed to `fftwnd` are expected to be stored as a single contiguous block in *row-major* order (sometimes called "C order"). Basically, this means that as you step through adjacent memory locations, the first dimension's index varies most slowly and the last dimension's index varies most quickly.

To be more explicit, let us consider an array of rank $d$ whose dimensions are $n_1 \times n_2 \times n_3 \times \cdots \times n_d$. Now, we specify a location in the array by a sequence of (zero-based) indices, one for each dimension: $(i_1, i_2, i_3, \ldots, i_d)$. If the array is stored in row-major order, then this element is located at the position $i_d + n_d(i_{d-1} + n_{d-1}(\ldots + n_2 i_1))$.

Note that each element of the array must be of type `fftw_complex`; i.e. a (real, imaginary) pair of (double-precision) numbers. Note also that, in `fftwnd`, the expression above is multiplied by the stride to get the actual array index—this is useful in situations where each element of the multi-dimensional array is actually a data structure or another array, and you just want to transform a single field. In most cases, however, you use a stride of 1.

### 2.5.2 Column-major Format

Readers from the Fortran world are used to arrays stored in *column-major* order (sometimes called "Fortran order"). This is essentially the exact opposite of row-major order in that, here, the *first* dimension's index varies most quickly.

If you have an array stored in column-major order and wish to transform it using `fftwnd`, it is quite easy to do. When creating the plan, simply pass the dimensions of the array to `fftwnd_create_plan` in *reverse order*. For example, if your array is a rank three `N x M x L` matrix in column-major order, you should pass the dimensions of the array as if it were an `L x M x N` matrix (which it is, from the perspective of `fftwnd`). This is done for you automatically by the FFTW Fortran wrapper routines (see Chapter 5 [Calling FFTW from Fortran], page 51).

### 2.5.3 Static Arrays in C

Multi-dimensional arrays declared statically (that is, at compile time, not necessarily with the `static` keyword) in C are *already* in row-major order. You don't have to do anything special to transform them. (See Section 2.2 [Complex Multi-dimensional Transforms Tutorial], page 4, for an example of this sort of code.)

### 2.5.4 Dynamic Arrays in C

Often, especially for large arrays, it is desirable to allocate the arrays dynamically, at runtime. This isn't too hard to do, although it is not as straightforward for multi-dimensional arrays as it is for one-dimensional arrays.

Creating the array is simple: using a dynamic-allocation routine like `malloc`, allocate an array big enough to store N `fftw_complex` values, where N is the product of the sizes of the array dimensions (i.e. the total number of complex values in the array). For example, here is code to allocate a 5x12x27 rank 3 array:

```
fftw_complex *an_array;

an_array = (fftw_complex *) malloc(5 * 12 * 27 * sizeof(fftw_complex));
```

Accessing the array elements, however, is more tricky—you can't simply use multiple applications of the '`[]`' operator like you could for static arrays. Instead, you have to explicitly compute the offset into the array using the formula given earlier for row-major arrays. For example, to reference the $(i, j, k)$-th element of the array allocated above, you would use the expression `an_array[k + 27 * (j + 12 * i)]`.

This pain can be alleviated somewhat by defining appropriate macros, or, in C++, creating a class and overloading the '`()`' operator.

### 2.5.5 Dynamic Arrays in C—The Wrong Way

A different method for allocating multi-dimensional arrays in C is often suggested that is incompatible with `fftwnd`: *using it will cause FFTW to die a painful death.* We discuss the technique here, however, because it is so commonly known and used. This method is to create arrays of pointers of arrays of pointers of ...etcetera. For example, the analogue in this method to the example above is:

```
int i,j;
fftw_complex ***a_bad_array;  /* another way to make a 5x12x27 array */

a_bad_array = (fftw_complex ***) malloc(5 * sizeof(fftw_complex **));
for (i = 0; i < 5; ++i) {
     a_bad_array[i] =
        (fftw_complex **) malloc(12 * sizeof(fftw_complex *));
     for (j = 0; j < 12; ++j)
          a_bad_array[i][j] =
                (fftw_complex *) malloc(27 * sizeof(fftw_complex));
}
```

As you can see, this sort of array is inconvenient to allocate (and deallocate). On the other hand, it has the advantage that the $(i, j, k)$-th element can be referenced simply by `a_bad_array[i][j][k]`.

If you like this technique and want to maximize convenience in accessing the array, but still want to pass the array to FFTW, you can use a hybrid method. Allocate the array as one contiguous block, but also declare an array of arrays of pointers that point to appropriate places in the block. That sort of trick is beyond the scope of this documentation; for more information on multi-dimensional arrays in C, see the `comp.lang.c` FAQ.

## 2.6 Words of Wisdom

FFTW implements a method for saving plans to disk and restoring them. In fact, what FFTW does is more general than just saving and loading plans. The mechanism is called `wisdom`. Here, we describe this feature at a high level. See Chapter 3 [FFTW Reference], page 17, for a less casual (but more complete) discussion of how to use `wisdom` in FFTW.

Plans created with the `FFTW_MEASURE` option produce near-optimal FFT performance, but it can take a long time to compute a plan because FFTW must actually measure the runtime of many possible plans and select the best one. This is designed for the situations where so many transforms of the same size must be computed that the start-up time is irrelevant. For short initialization times but slightly slower transforms, we have provided `FFTW_ESTIMATE`. The `wisdom` mechanism is a way to get the best of both worlds. There are, however, certain caveats that the user must be aware of in using `wisdom`. For this reason, `wisdom` is an optional feature which is not enabled by default.

At its simplest, `wisdom` provides a way of saving plans to disk so that they can be reused in other program runs. You create a plan with the flags `FFTW_MEASURE` and `FFTW_USE_WISDOM`, and then save the `wisdom` using `fftw_export_wisdom`:

```
plan = fftw_create_plan(..., ... | FFTW_MEASURE | FFTW_USE_WISDOM);
fftw_export_wisdom(...);
```

The next time you run the program, you can restore the `wisdom` with `fftw_import_wisdom`, and then recreate the plan using the same flags as before. This time, however, the same optimal plan will be created very quickly without measurements. (FFTW still needs some time to compute trigonometric tables, however.) The basic outline is:

```
fftw_import_wisdom(...);
plan = fftw_create_plan(..., ... | FFTW_USE_WISDOM);
```

Wisdom is more than mere rote memorization, however. FFTW's `wisdom` encompasses all of the knowledge and measurements that were used to create the plan for a given size. Therefore, existing `wisdom` is also applied to the creation of other plans of different sizes.

Whenever a plan is created with the `FFTW_MEASURE` and `FFTW_USE_WISDOM` flags, `wisdom` is generated. Thereafter, plans for any transform with a similar factorization will be computed more quickly, so long as they use the `FFTW_USE_WISDOM` flag. In fact, for transforms with the same factors and of equal or lesser size, no measurements at all need to be made and an optimal plan can be created with negligible delay!

For example, suppose that you create a plan for $N = 2^{16}$. Then, for any equal or smaller power of two, FFTW can create a plan (with the same direction and flags) quickly, using the precomputed `wisdom`. Even for larger powers of two, or sizes that are a power of two times some other prime factors, plans will be computed more quickly than they would otherwise (although some measurements still have to be made).

The `wisdom` is cumulative, and is stored in a global, private data structure managed internally by FFTW. The storage space required is minimal, proportional to the logarithm of the sizes the `wisdom` was generated from. The `wisdom` can be forgotten (and its associated memory freed) by a call to `fftw_forget_wisdom()`; otherwise, it is remembered until the program terminates. It can also be exported to a file, a string, or any other medium using `fftw_export_wisdom` and restored during a subsequent execution of the program (or a different program) using `fftw_import_wisdom` (these functions are described below).

Because `wisdom` is incorporated into FFTW at a very low level, the same `wisdom` can be used for one-dimensional transforms, multi-dimensional transforms, and even the parallel extensions to FFTW. Just include `FFTW_USE_WISDOM` in the flags for whatever plans you create (i.e., always plan wisely).

Plans created with the `FFTW_ESTIMATE` plan can use `wisdom`, but cannot generate it; only `FFTW_MEASURE` plans actually produce `wisdom`. Also, plans can only use `wisdom` generated from plans created with the same direction and flags. For example, a size `42` `FFTW_BACKWARD` transform will not use `wisdom` produced by a size `42` `FFTW_FORWARD` transform. The only exception to this rule is that `FFTW_ESTIMATE` plans can use `wisdom` from `FFTW_MEASURE` plans.

### 2.6.1 Caveats in Using Wisdom

> For in much wisdom is much grief, and he that increaseth knowledge increaseth sorrow. [Ecclesiastes 1:18]

There are pitfalls to using `wisdom`, in that it can negate FFTW's ability to adapt to changing hardware and other conditions. For example, it would be perfectly possible to export `wisdom` from a program running on one processor and import it into a program running on another processor. Doing so, however, would mean that the second program would use plans optimized for the first processor, instead of the one it is running on.

It should be safe to reuse `wisdom` as long as the hardware and program binaries remain unchanged. (Actually, the optimal plan may change even between runs of the same binary on identical hardware, due to differences in the virtual memory environment, etcetera. Users seriously interested in performance should worry about this problem, too.) It is likely that, if the same `wisdom` is used for two different program binaries, even running on the same machine, the plans may be sub-optimal because of differing code alignments. It is therefore wise to recreate `wisdom` every time an application is recompiled. The more the underlying hardware and software changes between the creation of `wisdom` and its use, the greater grows the risk of sub-optimal plans.

### 2.6.2 Importing and Exporting Wisdom

```
void fftw_export_wisdom_to_file(FILE *output_file);
fftw_status fftw_import_wisdom_from_file(FILE *input_file);
```

`fftw_export_wisdom_to_file` writes the `wisdom` to `output_file`, which must be a file open for writing. `fftw_import_wisdom_from_file` reads the `wisdom` from `input_file`, which must be a file open for reading, and returns `FFTW_SUCCESS` if successful and `FFTW_FAILURE` otherwise. In both cases, the file is left open and must be closed by the caller. It is perfectly fine if other data lie before or after the `wisdom` in the file, as long as the file is positioned at the beginning of the `wisdom` data before import.

```
char *fftw_export_wisdom_to_string(void);
fftw_status fftw_import_wisdom_from_string(const char *input_string)
```

`fftw_export_wisdom_to_string` allocates a string, exports the `wisdom` to it in `NULL`-terminated format, and returns a pointer to the string. If there is an error in allocating or writing the data, it returns `NULL`. The caller is responsible for deallocating the string (with `fftw_free`) when she is done with it. `fftw_import_wisdom_from_string` imports the wisdom from input_string, returning FFTW_SUCCESS if successful and FFTW_FAILURE otherwise.

Exporting wisdom does not affect the store of wisdom. Imported wisdom supplements the current store rather than replacing it (except when there is conflicting wisdom, in which case the older wisdom is discarded). The format of the exported wisdom is “nerd-readable” LISP-like ASCII text; we will not document it here except to note that it is insensitive to white space (interested users can contact us for more details).

See Chapter 3 [FFTW Reference], page 17, for more information, and for a description of how you can implement wisdom import/export for other media besides files and strings.

The following is a brief example in which the wisdom is read from a file, a plan is created (possibly generating more wisdom), and then the wisdom is exported to a string and printed to stdout.

```
{
     fftw_plan plan;
     char *wisdom_string;
     FILE *input_file;

     /* open file to read wisdom from */
     input_file = fopen("sample.wisdom", "r");
     if (FFTW_FAILURE == fftw_import_wisdom_from_file(input_file))
          printf("Error reading wisdom!\n");
     fclose(input_file); /* be sure to close the file! */

     /* create a plan for N=64, possibly creating and/or using wisdom */
     plan = fftw_create_plan(64,FFTW_FORWARD,
                             FFTW_MEASURE | FFTW_USE_WISDOM);

     /* ... do some computations with the plan ... */

     /* always destroy plans when you are done */
     fftw_destroy_plan(plan);

     /* write the wisdom to a string */
     wisdom_string = fftw_export_wisdom_to_string();
     if (wisdom_string != NULL) {
          printf("Accumulated wisdom: %s\n",wisdom_string);

          /* Just for fun, destroy and restore the wisdom */
          fftw_forget_wisdom(); /* all gone! */
          fftw_import_wisdom_from_string(wisdom_string);
          /* wisdom is back! */

          fftw_free(wisdom_string); /* deallocate it since we're done */
     }
}
```

# 3 FFTW Reference

This chapter provides a complete reference for all sequential (i.e., one-processor) FFTW functions. We first define the data types upon which FFTW operates, that is, real, complex, and "halfcomplex" numbers (see Section 3.1 [Data Types], page 17). Then, in four sections, we explain the FFTW program interface for complex one-dimensional transforms (see Section 3.2 [One-dimensional Transforms Reference], page 18), complex multi-dimensional transforms (see Section 3.3 [Multi-dimensional Transforms Reference], page 22), and real one-dimensional transforms (see Section 3.4 [Real One-dimensional Transforms Reference], page 26), real multi-dimensional transforms (see Section 3.5 [Real Multi-dimensional Transforms Reference], page 29). Section 3.6 [Wisdom Reference], page 34 describes the `wisdom` mechanism for exporting and importing plans. Finally, Section 3.7 [Memory Allocator Reference], page 35 describes how to change FFTW's default memory allocator. For parallel transforms, See Chapter 4 [Parallel FFTW], page 37.

## 3.1 Data Types

The routines in the FFTW package use three main kinds of data types. *Real* and *complex* numbers should be already known to the reader. We also use the term *halfcomplex* to describe complex arrays in a special packed format used by the one-dimensional real transforms (taking advantage of the *hermitian* symmetry that arises in those cases).

By including `<fftw.h>` or `<rfftw.h>`, you will have access to the following definitions:

```
typedef double fftw_real;

typedef struct {
     fftw_real re, im;
} fftw_complex;

#define c_re(c)  ((c).re)
#define c_im(c)  ((c).im)
```

All FFTW operations are performed on the `fftw_real` and `fftw_complex` data types. For `fftw_complex` numbers, the two macros `c_re` and `c_im` retrieve, respectively, the real and imaginary parts of the number.

A *real array* is an array of real numbers. A *complex array* is an array of complex numbers. A one-dimensional array $X$ of $n$ complex numbers is *hermitian* if the following property holds: for all $0 \leq i < n$, we have $X_i = X^*_{n-i}$, where $x^*$ denotes the complex conjugate of $x$. Hermitian arrays are relevant to FFTW because the Fourier transform of a real array is hermitian.

Because of its symmetry, a hermitian array can be stored in half the space of a complex array of the same size. FFTW's one-dimensional real transforms store hermitian arrays as *halfcomplex* arrays. A halfcomplex array of size $n$ is a one-dimensional array of $n$ `fftw_real` numbers. A hermitian array $X$ in stored into a halfcomplex array $Y$ as follows. For all integers $i$ such that $0 \leq i \leq n/2$, we have $Y_i := \mathrm{Re}(X_i)$. For all integers $i$ such that $0 < i < n/2$, we have $Y_{n-i} := \mathrm{Im}(X_i)$.

We now illustrate halfcomplex storage for $n = 4$ and $n = 5$, since the scheme depends on the parity of $n$. Let $n = 4$. In this case, we have $Y_0 := \mathrm{Re}(X_0)$, $Y_1 := \mathrm{Re}(X_1)$, $Y_2 := \mathrm{Re}(X_2)$, and $Y_3 := \mathrm{Im}(X_1)$. Let now $n = 5$. In this case, we have $Y_0 := \mathrm{Re}(X_0)$, $Y_1 := \mathrm{Re}(X_1)$, $Y_2 := \mathrm{Re}(X_2)$, $Y_3 := \mathrm{Im}(X_2)$, and $Y_4 := \mathrm{Im}(X_1)$.

By default, the type `fftw_real` equals the C type `double`. To work in single precision rather than double precision, `#define` the symbol `FFTW_ENABLE_FLOAT` in `fftw.h` and then recompile the library. On Unix systems, you can instead use `configure --enable-float` at installation time (see Chapter 6 [Installation and Customization], page 55).

In version 1 of FFTW, the data types were called `FFTW_REAL` and `FFTW_COMPLEX`. We changed the capitalization for consistency with the rest of FFTW's conventions. The old names are still supported, but their use is deprecated.

## 3.2 One-dimensional Transforms Reference

The one-dimensional complex routines are generally prefixed with `fftw_`. Programs using FFTW should be linked with `-lfftw -lm` on Unix systems, or with the FFTW and standard math libraries in general.

### 3.2.1 Plan Creation for One-dimensional Transforms

```
#include <fftw.h>

fftw_plan fftw_create_plan(int n, fftw_direction dir,
                           int flags);

fftw_plan fftw_create_plan_specific(int n, fftw_direction dir,
                                    int flags,
                                    fftw_complex *in, int istride,
                                    fftw_complex *out, int ostride);
```

The function `fftw_create_plan` creates a plan, which is a data structure containing all the information that `fftw` needs in order to compute the 1D Fourier transform. You can create as many plans as you need, but only one plan for a given array size is required (a plan can be reused many times).

`fftw_create_plan` returns a valid plan, or `NULL` if, for some reason, the plan can't be created. In the default installation, this cannot happen, but it is possible to configure FFTW in such a way that some input sizes are forbidden, and FFTW cannot create a plan.

The `fftw_create_plan_specific` variant takes as additional arguments specific input/output arrays and their strides. For the last four arguments, you should pass the arrays and strides that you will eventually be passing to `fftw`. The resulting plans will be optimized for those arrays and strides, although they may be used on other arrays as well. Note: the contents of the in and out arrays are *destroyed* by the specific planner (the initial contents are ignored, so the arrays need not have been initialized).

#### Arguments

- `n` is the size of the transform. It can be any positive integer.
  - FFTW is best at handling sizes of the form $2^a 3^b 5^c 7^d 11^e 13^f$, where $e+f$ is either 0 or 1, and the other exponents are arbitrary. Other sizes are computed by means of a slow, general-purpose routine (which nevertheless retains $O(n \log n)$ performance,

even for prime sizes). (It is possible to customize FFTW for different array sizes. See Chapter 6 [Installation and Customization], page 55, for more information.) Transforms whose sizes are powers of 2 are especially fast.

- `dir` is the sign of the exponent in the formula that defines the Fourier transform. It can be $-1$ or $+1$. The aliases `FFTW_FORWARD` and `FFTW_BACKWARD` are provided, where `FFTW_FORWARD` stands for $-1$.
- `flags` is a boolean OR (‘`|`’) of zero or more of the following:
  - `FFTW_MEASURE`: this flag tells FFTW to find the optimal plan by actually *computing* several FFTs and measuring their execution time. Depending on the installation, this can take some time.[1]
  - `FFTW_ESTIMATE`: do not run any FFT and provide a “reasonable” plan (for a RISC processor with many registers). If neither `FFTW_ESTIMATE` nor `FFTW_MEASURE` is provided, the default is `FFTW_ESTIMATE`.
  - `FFTW_OUT_OF_PLACE`: produce a plan assuming that the input and output arrays will be distinct (this is the default).
  - `FFTW_IN_PLACE`: produce a plan assuming that you want the output in the input array. The algorithm used is not necessarily in place: FFTW is able to compute true in-place transforms only for small values of `n`. If FFTW is not able to compute the transform in-place, it will allocate a temporary array (unless you provide one yourself), compute the transform out of place, and copy the result back. *Warning: This option changes the meaning of some parameters of* `fftw` (see Section 3.2.3 [Computing the One-dimensional Transform], page 20).

    The in-place option is mainly provided for people who want to write their own in-place multi-dimensional Fourier transform, using FFTW as a base. For example, consider a three-dimensional `n * n * n` transform. An out-of-place algorithm will need another array (which may be huge). However, FFTW can compute the in-place transform along each dimension using only a temporary array of size `n`. Moreover, if FFTW happens to be able to compute the transform truly in-place, no temporary array and no copying are needed. As distributed, FFTW ‘knows’ how to compute in-place transforms of size 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 32 and 64.

    The default mode of operation is `FFTW_OUT_OF_PLACE`.
  - `FFTW_USE_WISDOM`: use any `wisdom` that is available to help in the creation of the plan. (See Section 2.6 [Words of Wisdom], page 13.) This can greatly speed the creation of plans, especially with the `FFTW_MEASURE` option. `FFTW_ESTIMATE` plans can also take advantage of `wisdom` to produce a more optimal plan (based on past measurements) than the estimation heuristic would normally generate. When the `FFTW_MEASURE` option is used, new `wisdom` will also be generated if the current transform size is not completely understood by existing `wisdom`.
- `in`, `out`, `istride`, `ostride` (only for `fftw_create_plan_specific`): see corresponding arguments in the description of `fftw`. (See Section 3.2.3 [Computing the One-dimensional Transform], page 20.) In particular, the `out` and `ostride` parameters have the same special meaning for `FFTW_IN_PLACE` transforms as they have for `fftw`.

[1] The basic problem is the resolution of the clock: FFTW needs to run for a certain time for the clock to be reliable.

### 3.2.2 Discussion on Specific Plans

We recommend the use of the specific planners, even in cases where you will be transforming arrays different from those passed to the specific planners, as they confer the following advantages:

- The resulting plans will be optimized for your specific arrays and strides. This may or may not make a significant difference, but it certainly doesn't hurt. (The ordinary planner does its planning based upon a stride-one temporary array that it allocates.)
- Less intermediate storage is required during the planning process. (The ordinary planner uses O(`N`) temporary storage, where `N` is the maximum dimension, while it is creating the plan.)
- For multi-dimensional transforms, new parameters become accessible for optimization by the planner. (Since multi-dimensional arrays can be very large, we don't dare to allocate one in the ordinary planner for experimentation. This prevents us from doing certain optimizations that can yield dramatic improvements in some cases.)

On the other hand, note that *the specific planner destroys the contents of the* `in` *and* `out` *arrays.*

### 3.2.3 Computing the One-dimensional Transform

```
#include <fftw.h>

void fftw(fftw_plan plan, int howmany,
          fftw_complex *in, int istride, int idist,
          fftw_complex *out, int ostride, int odist);

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

The function `fftw` computes the one-dimensional Fourier transform, using a plan created by `fftw_create_plan` (See Section 3.2.1 [Plan Creation for One-dimensional Transforms], page 18.) The function `fftw_one` provides a simplified interface for the common case of single input array of stride 1.

## Arguments

- `plan` is the plan created by `fftw_create_plan` (see Section 3.2.1 [Plan Creation for One-dimensional Transforms], page 18).
- `howmany` is the number of transforms `fftw` will compute. It is faster to tell FFTW to compute many transforms, instead of simply calling `fftw` many times.
- `in`, `istride` and `idist` describe the input array(s). There are `howmany` input arrays; the first one is pointed to by `in`, the second one is pointed to by `in + idist`, and so on, up to `in + (howmany - 1) * idist`. Each input array consists of complex numbers (see Section 3.1 [Data Types], page 17), which are not necessarily contiguous in memory. Specifically, `in[0]` is the first element of the first array, `in[istride]` is the second element of the first array, and so on. In general, the `i`-th element of the `j`-th input array will be in position `in[i * istride + j * idist]`.

- `out`, `ostride` and `odist` describe the output array(s). The format is the same as for the input array.
  - *In-place transforms*: If the `plan` specifies an in-place transform, `ostride` and `odist` are always ignored. If `out` is `NULL`, `out` is ignored, too. Otherwise, `out` is interpreted as a pointer to an array of `n` complex numbers, that FFTW will use as temporary space to perform the in-place computation. `out` is used as scratch space and its contents destroyed. In this case, `out` must be an ordinary array whose elements are contiguous in memory (no striding).

The function `fftw_one` transforms a single, contiguous input array to a contiguous output array. By definition, the call

```
fftw_one(plan, in, out)
```

is equivalent to

```
fftw(plan, 1, in, 1, 0, out, 1, 0)
```

### 3.2.4 Destroying a One-dimensional Plan

```
#include <fftw.h>

void fftw_destroy_plan(fftw_plan plan);
```

The function `fftw_destroy_plan` frees the plan `plan` and releases all the memory associated with it. After destruction, a plan is no longer valid.

### 3.2.5 What FFTW Really Computes

In this section, we define precisely what FFTW computes. Please be warned that different authors and software packages might employ different conventions than FFTW does.

The forward transform of a complex array $X$ of size $n$ computes an array $Y$, where

$$Y_i = \sum_{j=0}^{n-1} X_j e^{-2\pi ij\sqrt{-1}/n} .$$

The backward transform computes

$$Y_i = \sum_{j=0}^{n-1} X_j e^{2\pi ij\sqrt{-1}/n} .$$

FFTW computes an unnormalized transform, that is, the equation $IFFT(FFT(X)) = nX$ holds. In other words, applying the forward and then the backward transform will multiply the input by $n$.

An `FFTW_FORWARD` transform corresponds to a sign of $-1$ in the exponent of the DFT. Note also that we use the standard "in-order" output ordering—the $k$-th output corresponds to the frequency $k/n$ (or $k/T$, where $T$ is your total sampling period). For those who like to think in terms of positive and negative frequencies, this means that the positive frequencies are stored in the first half of the output and the negative frequencies are stored in backwards order in the second half of the output. (The frequency $-k/n$ is the same as the frequency $(n-k)/n$.)

## 3.3 Multi-dimensional Transforms Reference

The multi-dimensional complex routines are generally prefixed with `fftwnd_`. Programs using FFTWND should be linked with `-lfftw -lm` on Unix systems, or with the FFTW and standard math libraries in general.

### 3.3.1 Plan Creation for Multi-dimensional Transforms

```
#include <fftw.h>

fftwnd_plan fftwnd_create_plan(int rank, const int *n,
                               fftw_direction dir, int flags);

fftwnd_plan fftw2d_create_plan(int nx, int ny,
                               fftw_direction dir, int flags);

fftwnd_plan fftw3d_create_plan(int nx, int ny, int nz,
                               fftw_direction dir, int flags);

fftwnd_plan fftwnd_create_plan_specific(int rank, const int *n,
                                        fftw_direction dir,
                                        int flags,
                                        fftw_complex *in, int istride,
                                        fftw_complex *out, int ostride);

fftwnd_plan fftw2d_create_plan_specific(int nx, int ny,
                                        fftw_direction dir,
                                        int flags,
                                        fftw_complex *in, int istride,
                                        fftw_complex *out, int ostride);

fftwnd_plan fftw3d_create_plan_specific(int nx, int ny, int nz,
                                        fftw_direction dir, int flags,
                                        fftw_complex *in, int istride,
                                        fftw_complex *out, int ostride);
```

The function `fftwnd_create_plan` creates a plan, which is a data structure containing all the information that `fftwnd` needs in order to compute a multi-dimensional Fourier transform. You can create as many plans as you need, but only one plan for a given array size is required (a plan can be reused many times). The functions `fftw2d_create_plan` and `fftw3d_create_plan` are optional, alternative interfaces to `fftwnd_create_plan` for two and three dimensions, respectively.

`fftwnd_create_plan` returns a valid plan, or `NULL` if, for some reason, the plan can't be created. This can happen if memory runs out or if the arguments are invalid in some way (e.g. if `rank` < 0).

The `create_plan_specific` variants take as additional arguments specific input/output arrays and their strides. For the last four arguments, you should pass the arrays and strides that you will eventually be passing to `fftwnd`. The resulting plans will be optimized for those arrays and strides, although they may be used on other arrays as well. Note: the contents of the in and out arrays are *destroyed* by the specific planner (the initial contents are ignored, so the arrays need not have been initialized). See Section 3.2.2 [Discussion on Specific Plans], page 20, for a discussion on specific plans.

## Arguments

- `rank` is the dimensionality of the arrays to be transformed. It can be any non-negative integer.
- `n` is a pointer to an array of `rank` integers, giving the size of each dimension of the arrays to be transformed. These sizes, which must be positive integers, correspond to the dimensions of row-major arrays—i.e. `n[0]` is the size of the dimension whose indices vary most slowly, and so on. (See Section 2.5 [Multi-dimensional Array Format], page 11, for more information on row-major storage.) See Section 3.2.1 [Plan Creation for One-dimensional Transforms], page 18, for more information regarding optimal array sizes.
- `nx` and `ny` in `fftw2d_create_plan` are positive integers specifying the dimensions of the rank 2 array to be transformed. i.e. they specify that the transform will operate on `nx x ny` arrays in row-major order, where `nx` is the number of rows and `ny` is the number of columns.
- `nx`, `ny` and `nz` in `fftw3d_create_plan` are positive integers specifying the dimensions of the rank 3 array to be transformed. i.e. they specify that the transform will operate on `nx x ny x nz` arrays in row-major order.
- `dir` is the sign of the exponent in the formula that defines the Fourier transform. It can be $-1$ or $+1$. The aliases `FFTW_FORWARD` and `FFTW_BACKWARD` are provided, where `FFTW_FORWARD` stands for $-1$.
- `flags` is a boolean OR (‘`|`’) of zero or more of the following:
  - `FFTW_MEASURE`: this flag tells FFTW to find the optimal plan by actually *computing* several FFTs and measuring their execution time.
  - `FFTW_ESTIMATE`: do not run any FFT and provide a “reasonable” plan (for a RISC processor with many registers). If neither `FFTW_ESTIMATE` nor `FFTW_MEASURE` is provided, the default is `FFTW_ESTIMATE`.
  - `FFTW_OUT_OF_PLACE`: produce a plan assuming that the input and output arrays will be distinct (this is the default).
  - `FFTW_IN_PLACE`: produce a plan assuming that you want to perform the transform in-place. (Unlike the one-dimensional transform, this “really”[2] performs the transform in-place.) Note that, if you want to perform in-place transforms, you *must* use a plan created with this option.

    The default mode of operation is `FFTW_OUT_OF_PLACE`.
  - `FFTW_USE_WISDOM`: use any `wisdom` that is available to help in the creation of the plan. (See Section 2.6 [Words of Wisdom], page 13.) This can greatly speed the creation of plans, especially with the `FFTW_MEASURE` option. `FFTW_ESTIMATE`

[2] `fftwnd` actually may use some temporary storage (hidden in the plan), but this storage space is only the size of the largest dimension of the array, rather than being as big as the entire array. (Unless you use `fftwnd` to perform one-dimensional transforms, in which case the temporary storage required for in-place transforms *is* as big as the entire array.)

plans can also take advantage of `wisdom` to produce a more optimal plan (based on past measurements) than the estimation heuristic would normally generate. When the `FFTW_MEASURE` option is used, new `wisdom` will also be generated if the current transform size is not completely understood by existing `wisdom`. Note that the same `wisdom` is shared between one-dimensional and multi-dimensional transforms.

- `in`, `out`, `istride`, `ostride` (only for the `_create_plan_specific` variants): see corresponding arguments in the description of `fftwnd`. (See Section 3.3.2 [Computing the Multi-dimensional Transform], page 24.)

### 3.3.2 Computing the Multi-dimensional Transform

```
#include <fftw.h>

void fftwnd(fftwnd_plan plan, int howmany,
            fftw_complex *in, int istride, int idist,
            fftw_complex *out, int ostride, int odist);

void fftwnd_one(fftwnd_plan p, fftw_complex *in,
                fftw_complex *out);
```

The function `fftwnd` computes one or more multi-dimensional Fourier Transforms, using a plan created by `fftwnd_create_plan` (see Section 3.3.1 [Plan Creation for Multi-dimensional Transforms], page 22). (Note that the plan determines the rank and dimensions of the array to be transformed.) The function `fftwnd_one` provides a simplified interface for the common case of single input array of stride 1.

## Arguments

- `plan` is the plan created by `fftwnd_create_plan`. (see Section 3.3.1 [Plan Creation for Multi-dimensional Transforms], page 22). In the case of two and three-dimensional transforms, it could also have been created by `fftw2d_create_plan` or `fftw3d_create_plan`, respectively.
- `howmany` is the number of multi-dimensional transforms `fftwnd` will compute.
- `in`, `istride` and `idist` describe the input array(s). There are `howmany` multi-dimensional input arrays; the first one is pointed to by `in`, the second one is pointed to by `in + idist`, and so on, up to `in + (howmany - 1) * idist`. Each multi-dimensional input array consists of complex numbers (see Section 3.1 [Data Types], page 17), stored in row-major format (see Section 2.5 [Multi-dimensional Array Format], page 11), which are not necessarily contiguous in memory. Specifically, `in[0]` is the first element of the first array, `in[istride]` is the second element of the first array, and so on. In general, the `i`-th element of the `j`-th input array will be in position `in[i * istride + j * idist]`. Note that, here, `i` refers to an index into the row-major format for the multi-dimensional array, rather than an index in any particular dimension.
  - *In-place transforms*: For plans created with the `FFTW_IN_PLACE` option, the transform is computed in-place—the output is returned in the `in` array, using the same strides, etcetera, as were used in the input.

- `out`, `ostride` and `odist` describe the output array(s). The format is the same as for the input array.
  - *In-place transforms*: These parameters are ignored for plans created with the `FFTW_IN_PLACE` option.

The function `fftwnd_one` transforms a single, contiguous input array to a contiguous output array. By definition, the call

```
fftwnd_one(plan, in, out)
```

is equivalent to

```
fftwnd(plan, 1, in, 1, 0, out, 1, 0)
```

### 3.3.3 Destroying a Multi-dimensional Plan

```
#include <fftw.h>

void fftwnd_destroy_plan(fftwnd_plan plan);
```

The function `fftwnd_destroy_plan` frees the plan `plan` and releases all the memory associated with it. After destruction, a plan is no longer valid.

### 3.3.4 What FFTWND Really Computes

The conventions that we follow for the multi-dimensional transform are analogous to those for the one-dimensional transform. In particular, the forward transform has a negative sign in the exponent and neither the forward nor the backward transforms will perform any normalization. Computing the backward transform of the forward transform will multiply the array by the product of its dimensions. The output is in-order, and the zeroth element of the output is the amplitude of the zero frequency component.

The exact mathematical definition of our multi-dimensional transform follows. Let $X$ be a $d$-dimensional complex array whose elements are $X[j_1, j_2, \ldots, j_d]$, where $0 \le j_s < n_s$ for all $s \in \{1, 2, \ldots, d\}$. Let also $\omega_s = e^{2\pi\sqrt{-1}/n_s}$, for all $s \in \{1, 2, \ldots, d\}$.

The forward transform computes a complex array $Y$, whose structure is the same as that of $X$, defined by

$$Y[i_1, i_2, \ldots, i_d] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, j_2, \ldots, j_d] \omega_1^{-i_1 j_1} \omega_2^{-i_2 j_2} \cdots \omega_d^{-i_d j_d} .$$

The backward transform computes

$$Y[i_1, i_2, \ldots, i_d] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, j_2, \ldots, j_d] \omega_1^{i_1 j_1} \omega_2^{i_2 j_2} \cdots \omega_d^{i_d j_d} .$$

Computing the forward transform followed by the backward transform will multiply the array by $\prod_{s=1}^{d} n_d$.

## 3.4 Real One-dimensional Transforms Reference

The one-dimensional real routines are generally prefixed with `rfftw_`.[3] Programs using RFFTW should be linked with `-lrfftw -lfftw -lm` on Unix systems, or with the RFFTW, the FFTW, and the standard math libraries in general.

### 3.4.1 Plan Creation for Real One-dimensional Transforms

```
#include <rfftw.h>

rfftw_plan rfftw_create_plan(int n, fftw_direction dir, int flags);

rfftw_plan rfftw_create_plan_specific(int n, fftw_direction dir,
    int flags, fftw_real *in, int istride,
    fftw_real *out, int ostride);
```

The function `rfftw_create_plan` creates a plan, which is a data structure containing all the information that `rfftw` needs in order to compute the 1D real Fourier transform. You can create as many plans as you need, but only one plan for a given array size is required (a plan can be reused many times).

`rfftw_create_plan` returns a valid plan, or `NULL` if, for some reason, the plan can't be created. In the default installation, this cannot happen, but it is possible to configure RFFTW in such a way that some input sizes are forbidden, and RFFTW cannot create a plan.

The `rfftw_create_plan_specific` variant takes as additional arguments specific input/output arrays and their strides. For the last four arguments, you should pass the arrays and strides that you will eventually be passing to `rfftw`. The resulting plans will be optimized for those arrays and strides, although they may be used on other arrays as well. Note: the contents of the in and out arrays are *destroyed* by the specific planner (the initial contents are ignored, so the arrays need not have been initialized). See Section 3.2.2 [Discussion on Specific Plans], page 20, for a discussion on specific plans.

#### Arguments

- `n` is the size of the transform. It can be any positive integer.
  - RFFTW is best at handling sizes of the form $2^a 3^b 5^c 7^d 11^e 13^f$, where $e+f$ is either 0 or 1, and the other exponents are arbitrary. Other sizes are computed by means of a slow, general-purpose routine (reducing to $O(n^2)$ performance for prime sizes). (It is possible to customize RFFTW for different array sizes. See Chapter 6 [Installation and Customization], page 55, for more information.) Transforms whose sizes are powers of 2 are especially fast. If you have large prime factors, it may be faster to switch over to the complex FFTW routines, which have $O(n \log n)$ performance even for prime sizes (we don't know of a similar algorithm specialized for real data, unfortunately).
- `dir` is the direction of the desired transform, either `FFTW_REAL_TO_COMPLEX` or `FFTW_COMPLEX_TO_REAL`, corresponding to `FFTW_FORWARD` or `FFTW_BACKWARD`, respectively.

[3] The etymologically-correct spelling would be `frftw_`, but it is hard to remember.

- `flags` is a boolean OR ('`|`') of zero or more of the following:
  - `FFTW_MEASURE`: this flag tells RFFTW to find the optimal plan by actually *computing* several FFTs and measuring their execution time. Depending on the installation, this can take some time.
  - `FFTW_ESTIMATE`: do not run any FFT and provide a "reasonable" plan (for a RISC processor with many registers). If neither `FFTW_ESTIMATE` nor `FFTW_MEASURE` is provided, the default is `FFTW_ESTIMATE`.
  - `FFTW_OUT_OF_PLACE`: produce a plan assuming that the input and output arrays will be distinct (this is the default).
  - `FFTW_IN_PLACE`: produce a plan assuming that you want the output in the input array. The algorithm used is not necessarily in place: RFFTW is able to compute true in-place transforms only for small values of `n`. If RFFTW is not able to compute the transform in-place, it will allocate a temporary array (unless you provide one yourself), compute the transform out of place, and copy the result back. *Warning: This option changes the meaning of some parameters of* `rfftw` (see Section 3.4.2 [Computing the Real One-dimensional Transform], page 27).

    The default mode of operation is `FFTW_OUT_OF_PLACE`.
  - `FFTW_USE_WISDOM`: use any `wisdom` that is available to help in the creation of the plan. (See Section 2.6 [Words of Wisdom], page 13.) This can greatly speed the creation of plans, especially with the `FFTW_MEASURE` option. `FFTW_ESTIMATE` plans can also take advantage of `wisdom` to produce a more optimal plan (based on past measurements) than the estimation heuristic would normally generate. When the `FFTW_MEASURE` option is used, new `wisdom` will also be generated if the current transform size is not completely understood by existing `wisdom`.
- `in`, `out`, `istride`, `ostride` (only for `rfftw_create_plan_specific`): see corresponding arguments in the description of `rfftw`. (See Section 3.4.2 [Computing the Real One-dimensional Transform], page 27.) In particular, the `out` and `ostride` parameters have the same special meaning for `FFTW_IN_PLACE` transforms as they have for `rfftw`.

### 3.4.2 Computing the Real One-dimensional Transform

```
#include <rfftw.h>

void rfftw(rfftw_plan plan, int howmany,
           fftw_real *in, int istride, int idist,
           fftw_real *out, int ostride, int odist);

void rfftw_one(rfftw_plan plan, fftw_real *in, fftw_real *out);
```

The function `rfftw` computes the Real One-dimensional Fourier Transform, using a plan created by `rfftw_create_plan` (see Section 3.4.1 [Plan Creation for Real One-dimensional Transforms], page 26). The function `rfftw_one` provides a simplified interface for the common case of single input array of stride 1.

*Important:* When invoked for an out-of-place, `FFTW_COMPLEX_TO_REAL` transform, the input array is overwritten with scratch values by these routines. The input array is not modified for `FFTW_REAL_TO_COMPLEX` transforms.

## Arguments

- `plan` is the plan created by `rfftw_create_plan` (see Section 3.4.1 [Plan Creation for Real One-dimensional Transforms], page 26).
- `howmany` is the number of transforms `rfftw` will compute. It is faster to tell RFFTW to compute many transforms, instead of simply calling `rfftw` many times.
- `in`, `istride` and `idist` describe the input array(s). There are two cases. If the `plan` defines a `FFTW_REAL_TO_COMPLEX` transform, `in` is a real array. Otherwise, for `FFTW_COMPLEX_TO_REAL` transforms, `in` is a halfcomplex array *whose contents will be destroyed*.
- `out`, `ostride` and `odist` describe the output array(s), and have the same meaning as the corresponding parameters for the input array.
  - *In-place transforms*: If the `plan` specifies an in-place transform, `ostride` and `odist` are always ignored. If `out` is `NULL`, `out` is ignored, too. Otherwise, `out` is interpreted as a pointer to an array of `n` complex numbers, that FFTW will use as temporary space to perform the in-place computation. `out` is used as scratch space and its contents destroyed. In this case, `out` must be an ordinary array whose elements are contiguous in memory (no striding).

The function `rfftw_one` transforms a single, contiguous input array to a contiguous output array. By definition, the call

```
rfftw_one(plan, in, out)
```

is equivalent to

```
rfftw(plan, 1, in, 1, 0, out, 1, 0)
```

### 3.4.3 Destroying a Real One-dimensional Plan

```
#include <rfftw.h>

void rfftw_destroy_plan(rfftw_plan plan);
```

The function `rfftw_destroy_plan` frees the plan `plan` and releases all the memory associated with it. After destruction, a plan is no longer valid.

### 3.4.4 What RFFTW Really Computes

In this section, we define precisely what RFFTW computes.

The real to complex (`FFTW_REAL_TO_COMPLEX`) transform of a real array $X$ of size $n$ computes an hermitian array $Y$, where

$$Y_i = \sum_{j=0}^{n-1} X_j e^{-2\pi ij\sqrt{-1}/n}$$

(That $Y$ is a hermitian array is not intended to be obvious, although the proof is easy.) The hermitian array $Y$ is stored in halfcomplex order (see Section 3.1 [Data Types], page 17). Currently, RFFTW provides no way to compute a real to complex transform with a positive sign in the exponent.

The complex to real (`FFTW_COMPLEX_TO_REAL`) transform of a hermitian array $X$ of size $n$ computes a real array $Y$, where

$$Y_i = \sum_{j=0}^{n-1} X_j e^{2\pi ij\sqrt{-1}/n}$$

(That $Y$ is a real array is not intended to be obvious, although the proof is easy.) The hermitian input array $X$ is stored in halfcomplex order (see Section 3.1 [Data Types], page 17). Currently, RFFTW provides no way to compute a complex to real transform with a negative sign in the exponent.

Like FFTW, RFFTW computes an unnormalized transform. In other words, applying the real to complex (forward) and then the complex to real (backward) transform will multiply the input by $n$.

## 3.5 Real Multi-dimensional Transforms Reference

The multi-dimensional real routines are generally prefixed with `rfftwnd_`. Programs using RFFTWND should be linked with `-lrfftw -lfftw -lm` on Unix systems, or with the FFTW, RFFTW, and standard math libraries in general.

### 3.5.1 Plan Creation for Real Multi-dimensional Transforms

```
#include <rfftw.h>

rfftwnd_plan rfftwnd_create_plan(int rank, const int *n,
                                 fftw_direction dir, int flags);

rfftwnd_plan rfftw2d_create_plan(int nx, int ny,
                                 fftw_direction dir, int flags);

rfftwnd_plan rfftw3d_create_plan(int nx, int ny, int nz,
                                 fftw_direction dir, int flags);
```

The function `rfftwnd_create_plan` creates a plan, which is a data structure containing all the information that `rfftwnd` needs in order to compute a multi-dimensional real Fourier transform. You can create as many plans as you need, but only one plan for a given array size is required (a plan can be reused many times). The functions `rfftw2d_create_plan` and `rfftw3d_create_plan` are optional, alternative interfaces to `rfftwnd_create_plan` for two and three dimensions, respectively.

`rfftwnd_create_plan` returns a valid plan, or `NULL` if, for some reason, the plan can't be created. This can happen if the arguments are invalid in some way (e.g. if `rank` < 0).

### Arguments

- `rank` is the dimensionality of the arrays to be transformed. It can be any non-negative integer.
- `n` is a pointer to an array of `rank` integers, giving the size of each dimension of the arrays to be transformed. Note that these are always the dimensions of the *real* arrays; the

complex arrays have different dimensions (see Section 3.5.3 [Array Dimensions for Real Multi-dimensional Transforms], page 32). These sizes, which must be positive integers, correspond to the dimensions of row-major arrays—i.e. `n[0]` is the size of the dimension whose indices vary most slowly, and so on. (See Section 2.5 [Multi-dimensional Array Format], page 11, for more information.)

  - See Section 3.4.1 [Plan Creation for Real One-dimensional Transforms], page 26, for more information regarding optimal array sizes.

- `nx` and `ny` in `rfftw2d_create_plan` are positive integers specifying the dimensions of the rank 2 array to be transformed. i.e. they specify that the transform will operate on `nx x ny` arrays in row-major order, where `nx` is the number of rows and `ny` is the number of columns.
- `nx`, `ny` and `nz` in `rfftw3d_create_plan` are positive integers specifying the dimensions of the rank 3 array to be transformed. i.e. they specify that the transform will operate on `nx x ny x nz` arrays in row-major order.
- `dir` is the direction of the desired transform, either `FFTW_REAL_TO_COMPLEX` or `FFTW_COMPLEX_TO_REAL`, corresponding to `FFTW_FORWARD` or `FFTW_BACKWARD`, respectively.
- `flags` is a boolean OR (‘`|`’) of zero or more of the following:
  - `FFTW_MEASURE`: this flag tells FFTW to find the optimal plan by actually *computing* several FFTs and measuring their execution time.
  - `FFTW_ESTIMATE`: do not run any FFT and provide a “reasonable” plan (for a RISC processor with many registers). If neither `FFTW_ESTIMATE` nor `FFTW_MEASURE` is provided, the default is `FFTW_ESTIMATE`.
  - `FFTW_OUT_OF_PLACE`: produce a plan assuming that the input and output arrays will be distinct (this is the default).
  - `FFTW_IN_PLACE`: produce a plan assuming that you want to perform the transform in-place. (Unlike the one-dimensional transform, this “really” performs the transform in-place.) Note that, if you want to perform in-place transforms, you *must* use a plan created with this option. The use of this option has important implications for the size of the input/output array (see Section 3.5.2 [Computing the Real Multi-dimensional Transform], page 30).

    The default mode of operation is `FFTW_OUT_OF_PLACE`.
  - `FFTW_USE_WISDOM`: use any `wisdom` that is available to help in the creation of the plan. (See Section 2.6 [Words of Wisdom], page 13.) This can greatly speed the creation of plans, especially with the `FFTW_MEASURE` option. `FFTW_ESTIMATE` plans can also take advantage of `wisdom` to produce a more optimal plan (based on past measurements) than the estimation heuristic would normally generate. When the `FFTW_MEASURE` option is used, new `wisdom` will also be generated if the current transform size is not completely understood by existing `wisdom`. Note that the same `wisdom` is shared between one-dimensional and multi-dimensional transforms.

### 3.5.2 Computing the Real Multi-dimensional Transform

```
#include <rfftw.h>
```

```
void rfftwnd_real_to_complex(rfftwnd_plan plan, int howmany,
                             fftw_real *in, int istride, int idist,
                             fftw_complex *out, int ostride, int odist);
void rfftwnd_complex_to_real(rfftwnd_plan plan, int howmany,
                             fftw_complex *in, int istride, int idist,
                             fftw_real *out, int ostride, int odist);

void rfftwnd_one_real_to_complex(rfftwnd_plan p, fftw_real *in,
                                 fftw_complex *out);
void rfftwnd_one_complex_to_real(rfftwnd_plan p, fftw_complex *in,
                                 fftw_real *out);
```

These functions compute the real multi-dimensional Fourier Transform, using a plan created by `rfftwnd_create_plan` (see Section 3.5.1 [Plan Creation for Real Multi-dimensional Transforms], page 29). (Note that the plan determines the rank and dimensions of the array to be transformed.) The '`rfftwnd_one_`' functions provide a simplified interface for the common case of single input array of stride 1. Unlike other transform routines in FFTW, we here use separate functions for the two directions of the transform in order to correctly express the datatypes of the parameters.

*Important:* When invoked for an out-of-place, `FFTW_COMPLEX_TO_REAL` transform with `rank > 1`, the input array is overwritten with scratch values by these routines. The input array is not modified for `FFTW_REAL_TO_COMPLEX` transforms or for `FFTW_COMPLEX_TO_REAL` with `rank == 1`.

## Arguments

- `plan` is the plan created by `rfftwnd_create_plan`. (see Section 3.5.1 [Plan Creation for Real Multi-dimensional Transforms], page 29). In the case of two and three-dimensional transforms, it could also have been created by `rfftw2d_create_plan` or `rfftw3d_create_plan`, respectively.

  `FFTW_REAL_TO_COMPLEX` plans must be used with the '`real_to_complex`' functions, and `FFTW_COMPLEX_TO_REAL` plans must be used with the '`complex_to_real`' functions. It is an error to mismatch the plan direction and the transform function.
- `howmany` is the number of transforms to be computed.
- `in`, `istride` and `idist` describe the input array(s). There are `howmany` input arrays; the first one is pointed to by `in`, the second one is pointed to by `in + idist`, and so on, up to `in + (howmany - 1) * idist`. Each input array is stored in row-major format (see Section 2.5 [Multi-dimensional Array Format], page 11), and is not necessarily contiguous in memory. Specifically, `in[0]` is the first element of the first array, `in[istride]` is the second element of the first array, and so on. In general, the `i`-th element of the `j`-th input array will be in position `in[i * istride + j * idist]`. Note that, here, `i` refers to an index into the row-major format for the multi-dimensional array, rather than an index in any particular dimension.

  The dimensions of the arrays are different for real and complex data, and are discussed in more detail below (see Section 3.5.3 [Array Dimensions for Real Multi-dimensional Transforms], page 32).
  - *In-place transforms*: For plans created with the `FFTW_IN_PLACE` option, the transform is computed in-place—the output is returned in the `in` array. The meaning

of the `stride` and `dist` parameters in this case is subtle and is discussed below (see Section 3.5.4 [Strides in In-place RFFTWND], page 32).

- `out`, `ostride` and `odist` describe the output array(s). The format is the same as that for the input array. See below for a discussion of the dimensions of the output array for real and complex data.
  - *In-place transforms*: These parameters are ignored for plans created with the `FFTW_IN_PLACE` option.

The function `rfftwnd_one` transforms a single, contiguous input array to a contiguous output array. By definition, the call

```
rfftwnd_one_...(plan, in, out)
```

is equivalent to

```
rfftwnd_...(plan, 1, in, 1, 0, out, 1, 0)
```

### 3.5.3 Array Dimensions for Real Multi-dimensional Transforms

The output of a multi-dimensional transform of real data contains symmetries that, in principle, make half of the outputs redundant (see Section 3.5.6 [What RFFTWND Really Computes], page 33). In practice, it is not possible to entirely realize these savings in an efficient and understandable format. Instead, the output of the rfftwnd transforms is *slightly* over half of the output of the corresponding complex transform. We do not "pack" the data in any way, but store it as an ordinary array of `fftw_complex` values. In fact, this data is simply a subsection of what would be the array in the corresponding complex transform.

Specifically, for a real transform of dimensions $n_1 \times n_2 \times \cdots \times n_d$, the complex data is an $n_1 \times n_2 \times \cdots \times (n_d/2+1)$ array of `fftw_complex` values in row-major order (with the division rounded down). That is, we only store the lower half (plus one element) of the last dimension of the data from the ordinary complex transform. (We could have instead taken half of any other dimension, but implementation turns out to be simpler if the last, contiguous, dimension is used.)

Since the complex data is slightly larger than the real data, some complications arise for in-place transforms. In this case, the final dimension of the real data must be padded with extra values to accommodate the size of the complex data—two extra if the last dimension is even and one if it is odd. That is, the last dimension of the real data must physically contain $2(n_d/2+1)$ `fftw_real` values (exactly enough to hold the complex data). This physical array size does not, however, change the *logical* array size—only $n_d$ values are actually stored in the last dimension, and $n_d$ is the last dimension passed to `rfftwnd_create_plan`.

### 3.5.4 Strides in In-place RFFTWND

The fact that the input and output datatypes are different for rfftwnd complicates the meaning of the `stride` and `dist` parameters of in-place transforms—are they in units of `fftw_real` or `fftw_complex` elements? When reading the input, they are interpreted in units of the datatype of the input data. When writing the output, the `istride` and `idist` are translated to the output datatype's "units" in one of two ways, corresponding to the two most common situations in which `stride` and `dist` parameters are useful. Below, we refer to these "translated" parameters as `ostride_t` and `odist_t`. (Note that these are computed internally by rfftwnd; the actual `ostride` and `odist` parameters are ignored for in-place transforms.)

First, there is the case where you are transforming a number of contiguous arrays located one after another in memory. In this situation, `istride` is `1` and `idist` is the product of the physical dimensions of the array. `ostride_t` and `odist_t` are then chosen so that the output arrays are contiguous and lie on top of the input arrays. `ostride_t` is therefore `1`. For a real-to-complex transform, `odist_t` is `idist/2`; for a complex-to-real transform, `odist_t` is `idist*2`.

The second case is when you have an array in which each element has `nc` components (e.g. a structure with `nc` numeric fields), and you want to transform all of the components at once. Here, `istride` is `nc` and `idist` is `1`. For this case, it is natural to want the output to also have `nc` consecutive components, now of the output data type; this is exactly what rfftwnd does. Specifically, it uses an `ostride_t` equal to `istride`, and an `odist_t` of `1`. (Astute readers will realize that some extra buffer space is required in order to perform such a transform; this is handled automatically by rfftwnd.)

The general rule is as follows. `ostride_t` equals `istride`. If `idist` is `1` and `idist` is less than `istride`, then `odist_t` is `1`. Otherwise, for a real-to-complex transform `odist_t` is `idist/2` and for a complex-to-real transform `odist_t` is `idist*2`.

## 3.5.5 Destroying a Multi-dimensional Plan

```
#include <rfftw.h>

void rfftwnd_destroy_plan(rfftwnd_plan plan);
```

The function `rfftwnd_destroy_plan` frees the plan `plan` and releases all the memory associated with it. After destruction, a plan is no longer valid.

## 3.5.6 What RFFTWND Really Computes

The conventions that we follow for the real multi-dimensional transform are analogous to those for the complex multi-dimensional transform. In particular, the forward transform has a negative sign in the exponent and neither the forward nor the backward transforms will perform any normalization. Computing the backward transform of the forward transform will multiply the array by the product of its dimensions (that is, the logical dimensions of the real data). The forward transform is real-to-complex and the backward transform is complex-to-real.

The exact mathematical definition of our real multi-dimensional transform follows.

*Real to complex (forward) transform.* Let $X$ be a $d$-dimensional real array whose elements are $X[j_1, j_2, \ldots, j_d]$, where $0 \le j_s < n_s$ for all $s \in \{1, 2, \ldots, d\}$. Let also $\omega_s = e^{2\pi\sqrt{-1}/n_s}$, for all $s \in \{1, 2, \ldots, d\}$.

The real to complex transform computes a complex array $Y$, whose structure is the same as that of $X$, defined by

$$Y[i_1, i_2, \ldots, i_d] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, j_2, \ldots, j_d] \omega_1^{-i_1 j_1} \omega_2^{-i_2 j_2} \cdots \omega_d^{-i_d j_d} \, .$$

The output array $Y$ enjoys a multidimensional hermitian symmetry, that is, the identity $Y[i_1, i_2, \ldots, i_d] = Y[n_1 - i_1, n_2 - i_2, \ldots, n_d - i_d]^*$ holds for all $0 \le i_s < n_s$. Because of this symmetry, $Y$ is stored in the peculiar way described in Section 3.5.3 [Array Dimensions for Real Multi-dimensional Transforms], page 32.

*Complex to real (backward) transform.* Let $X$ be a $d$-dimensional complex array whose elements are $X[j_1, j_2, \ldots, j_d]$, where $0 \le j_s < n_s$ for all $s \in \{1, 2, \ldots, d\}$. The array $X$ must be hermitian, that is, the identity $X[j_1, j_2, \ldots, j_d] = X[n_1 - j_1, n_2 - j_2, \ldots, n_d - j_d]^*$ must hold for all $0 \le j_s < n_s$. Moreover, $X$ must be stored in memory in the peculiar way described in Section 3.5.3 [Array Dimensions for Real Multi-dimensional Transforms], page 32.

Let $\omega_s = e^{2\pi\sqrt{-1}/n_s}$, for all $s \in \{1, 2, \ldots, d\}$. The complex to real transform computes a real array $Y$, whose structure is the same as that of $X$, defined by

$$Y[i_1, i_2, \ldots, i_d] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, j_2, \ldots, j_d] \omega_1^{i_1 j_1} \omega_2^{i_2 j_2} \cdots \omega_d^{i_d j_d} .$$

(That $Y$ is real is not meant to be obvious, although the proof is easy.)

Computing the forward transform followed by the backward transform will multiply the array by $\prod_{s=1}^{d} n_d$.

## 3.6 Wisdom Reference

### 3.6.1 Exporting Wisdom

```
#include <fftw.h>

void fftw_export_wisdom(void (*emitter)(char c, void *), void *data);
void fftw_export_wisdom_to_file(FILE *output_file);
char *fftw_export_wisdom_to_string(void);
```

These functions allow you to export all currently accumulated `wisdom` in a form from which it can be later imported and restored, even during a separate run of the program. (See Section 2.6 [Words of Wisdom], page 13.) The current store of `wisdom` is not affected by calling any of these routines.

`fftw_export_wisdom` exports the `wisdom` to any output medium, as specified by the callback function `emitter`. `emitter` is a `putc`-like function that writes the character `c` to some output; its second parameter is the `data` pointer passed to `fftw_export_wisdom`. For convenience, the following two "wrapper" routines are provided:

`fftw_export_wisdom_to_file` writes the `wisdom` to the current position in `output_file`, which should be open with write permission. Upon exit, the file remains open and is positioned at the end of the `wisdom` data.

`fftw_export_wisdom_to_string` returns a pointer to a `NULL`-terminated string holding the `wisdom` data. This string is dynamically allocated, and it is the responsibility of the caller to deallocate it with `fftw_free` when it is no longer needed.

All of these routines export the wisdom in the same format, which we will not document here except to say that it is LISP-like ASCII text that is insensitive to white space.

### 3.6.2 Importing Wisdom

```
#include <fftw.h>

fftw_status fftw_import_wisdom(int (*get_input)(void *), void *data);
fftw_status fftw_import_wisdom_from_file(FILE *input_file);
fftw_status fftw_import_wisdom_from_string(const char *input_string);
```

These functions import `wisdom` into a program from data stored by the `fftw_export_wisdom` functions above. (See Section 2.6 [Words of Wisdom], page 13.) The imported `wisdom` supplements rather than replaces any `wisdom` already accumulated by the running program (except when there is conflicting `wisdom`, in which case the existing wisdom is replaced).

`fftw_import_wisdom` imports `wisdom` from any input medium, as specified by the callback function `get_input`. `get_input` is a `getc`-like function that returns the next character in the input; its parameter is the `data` pointer passed to `fftw_import_wisdom`. If the end of the input data is reached (which should never happen for valid data), it may return either `NULL` (ASCII 0) or `EOF` (as defined in `<stdio.h>`). For convenience, the following two "wrapper" routines are provided:

`fftw_import_wisdom_from_file` reads `wisdom` from the current position in `input_file`, which should be open with read permission. Upon exit, the file remains open and is positioned at the end of the `wisdom` data.

`fftw_import_wisdom_from_string` reads `wisdom` from the `NULL`-terminated string `input_string`.

The return value of these routines is `FFTW_SUCCESS` if the wisdom was read successfully, and `FFTW_FAILURE` otherwise. Note that, in all of these functions, any data in the input stream past the end of the `wisdom` data is simply ignored (it is not even read if the `wisdom` data is well-formed).

### 3.6.3 Forgetting Wisdom

```
#include <fftw.h>

void fftw_forget_wisdom(void);
```

Calling `fftw_forget_wisdom` causes all accumulated `wisdom` to be discarded and its associated memory to be freed. (New `wisdom` can still be gathered subsequently, however.)

## 3.7 Memory Allocator Reference

```
#include <fftw.h>

void *(*fftw_malloc_hook) (size_t n);
void (*fftw_free_hook) (void *p);
```

Whenever it has to allocate and release memory, FFTW ordinarily calls `malloc` and `free`. If `malloc` fails, FFTW prints an error message and exits. This behavior may be undesirable in some applications. Also, special memory-handling functions may be necessary in certain environments. Consequently, FFTW provides means by which you can install your own memory allocator and take whatever error-correcting action you find appropriate. The variables fftw_malloc_hook and fftw_free_hook are pointers to functions, and they are normally NULL. If you set those variables to point to other functions, then FFTW will use your routines instead of malloc and free. fftw_malloc_hook must point to a malloc-like function, and fftw_free_hook must point to a free-like function.

## 3.8 Thread safety

Users writing multi-threaded programs must concern themselves with the *thread safety* of the libraries they use—that is, whether it is safe to call routines in parallel from multiple threads. FFTW can be used in such an environment, but some care must be taken because certain parts of FFTW use private global variables to share data between calls. In particular, the plan-creation functions share trigonometric tables and accumulated wisdom. (Users should note that these comments only apply to programs using shared-memory threads. Parallelism using MPI or forked processes involves a separate address-space and global variables for each process, and is not susceptible to problems of this sort.)

The central restriction of FFTW is that it is not safe to create multiple plans in parallel. You must either create all of your plans from a single thread, or instead use a semaphore, mutex, or other mechanism to ensure that different threads don't attempt to create plans at the same time. The same restriction also holds for destruction of plans and importing/forgetting wisdom. Once created, a plan may safely be used in any thread.

The actual transform routines in FFTW (fftw_one, etcetera) are re-entrant and thread-safe, so it is fine to call them simultaneously from multiple threads. Another question arises, however—is it safe to use the *same plan* for multiple transforms in parallel? (It would be unsafe if, for example, the plan were modified in some way by the transform.) We address this question by defining an additional planner flag, FFTW_THREADSAFE. When included in the flags for any of the plan-creation routines, FFTW_THREADSAFE guarantees that the resulting plan will be read-only and safe to use in parallel by multiple threads.

# 4 Parallel FFTW

In this chapter we discuss the use of FFTW in a parallel environment, documenting the different parallel libraries that we have provided. (Users calling FFTW from a multi-threaded program should also consult Section 3.8 [Thread safety], page 36.) The FFTW package currently contains three parallel transform implementations that leverage the uniprocessor FFTW code:

- The first set of routines utilizes shared-memory threads for parallel one- and multi-dimensional transforms of both real and complex data. Any program using FFTW can be trivially modified to use the multi-threaded routines. This code can use any common threads implementation, including POSIX threads. (POSIX threads are available on most Unix variants, including Linux.) These routines are located in the `threads` directory, and are documented in Section 4.1 [Multi-threaded FFTW], page 37.
- The `mpi` directory contains multi-dimensional transforms of real and complex data for parallel machines supporting MPI. It also includes parallel one-dimensional transforms for complex data. The main feature of this code is that it supports distributed-memory transforms, so it runs on everything from workstation clusters to massively-parallel supercomputers. More information on MPI can be found at the MPI home page. The FFTW MPI routines are documented in Section 4.2 [MPI FFTW], page 40.
- We also have an experimental parallel implementation written in Cilk, a C-like parallel language developed at MIT and currently available for several SMP platforms. For more information on Cilk see the Cilk home page. The FFTW Cilk code can be found in the `cilk` directory, with parallelized one- and multi-dimensional transforms of complex data. The Cilk FFTW routines are documented in `cilk/README`.

## 4.1 Multi-threaded FFTW

In this section we document the parallel FFTW routines for shared-memory threads on SMP hardware. These routines, which support parallel one- and multi-dimensional transforms of both real and complex data, are the easiest way to take advantage of multiple processors with FFTW. They work just like the corresponding uniprocessor transform routines, except that they take the number of parallel threads to use as an extra parameter. Any program that uses the uniprocessor FFTW can be trivially modified to use the multi-threaded FFTW.

### 4.1.1 Installation and Supported Hardware/Software

All of the FFTW threads code is located in the `threads` subdirectory of the FFTW package. On Unix systems, the FFTW threads libraries and header files can be automatically configured, compiled, and installed along with the uniprocessor FFTW libraries simply by including `--enable-threads` in the flags to the `configure` script (see Section 6.1 [Installation on Unix], page 55). (Note also that the threads routines, when enabled, are automatically tested by the '`make check`' self-tests.)

The threads routines require your operating system to have some sort of shared-memory threads support. Specifically, the FFTW threads package works with POSIX threads (available on most Unix variants, including Linux), Solaris threads, BeOS threads (tested on

BeOS DR8.2), Mach C threads (reported to work by users), and Win32 threads (reported to work by users). (There is also untested code to use MacOS MP threads.) We also support using OpenMP or SGI MP compiler directives to launch threads, enabled by using `--with-openmp` or `--with-sgimp` in addition to `--enable-threads`. This is especially useful if you are employing that sort of directive in your own code, in order to minimize conflicts. If you have a shared-memory machine that uses a different threads API, it should be a simple matter of programming to include support for it; see the file `fftw_threads-int.h` for more detail.

SMP hardware is not required, although of course you need multiple processors to get any benefit from the multithreaded transforms.

### 4.1.2 Usage of Multi-threaded FFTW

Here, it is assumed that the reader is already familiar with the usage of the uniprocessor FFTW routines, described elsewhere in this manual. We only describe what one has to change in order to use the multi-threaded routines.

First, instead of including `<fftw.h>` or `<rfftw.h>`, you should include the files `<fftw_threads.h>` or `<rfftw_threads.h>`, respectively.

Second, before calling any FFTW routines, you should call the function:

```
int fftw_threads_init(void);
```

This function, which should only be called once (probably in your `main()` function), performs any one-time initialization required to use threads on your system. It returns zero if successful, and a non-zero value if there was an error (in which case, something is seriously wrong and you should probably exit the program).

Third, when you want to actually compute the transform, you should use one of the following transform routines instead of the ordinary FFTW functions:

```
fftw_threads(nthreads, plan, howmany, in, istride,
             idist, out, ostride, odist);

fftw_threads_one(nthreads, plan, in, out);

fftwnd_threads(nthreads, plan, howmany, in, istride,
               idist, out, ostride, odist);

fftwnd_threads_one(nthreads, plan, in, out);

rfftw_threads(nthreads, plan, howmany, in, istride,
              idist, out, ostride, odist);

rfftw_threads_one(nthreads, plan, in, out);

rfftwnd_threads_real_to_complex(nthreads, plan, howmany, in,
                                istride, idist, out, ostride, odist);

rfftwnd_threads_one_real_to_complex(nthreads, plan, in, out);

rfftwnd_threads_complex_to_real(nthreads, plan, howmany, in,
                                istride, idist, out, ostride, odist);

rfftwnd_threads_one_real_to_complex(nthreads, plan, in, out);

rfftwnd_threads_one_complex_to_real(nthreads, plan, in, out);
```

All of these routines take exactly the same arguments and have exactly the same effects as their uniprocessor counterparts (i.e. without the '`_threads`') *except* that they take one extra parameter, `nthreads` (of type `int`), before the normal parameters.[1] The `nthreads` parameter specifies the number of threads of execution to use when performing the transform (actually, the maximum number of threads).

For example, to parallelize a single one-dimensional transform of complex data, instead of calling the uniprocessor `fftw_one(plan, in, out)`, you would call `fftw_threads_one(nthreads, plan, in, out)`. Passing an `nthreads` of `1` means to use only one thread (the main thread), and is equivalent to calling the uniprocessor routine. Passing an `nthreads` of `2` means that the transform is potentially parallelized over two threads (and two processors, if you have them), and so on.

These are the only changes you need to make to your source code. Calls to all other FFTW routines (plan creation, destruction, wisdom, etcetera) are not parallelized and remain the same. (The same plans and wisdom are used by both uniprocessor and multi-threaded transforms.) Your arrays are allocated and formatted in the same way, and so on.

Programs using the parallel complex transforms should be linked with `-lfftw_threads -lfftw -lm` on Unix. Programs using the parallel real transforms should be linked with `-lrfftw_threads -lfftw_threads -lrfftw -lfftw -lm`. You will also need to link with whatever library is responsible for threads on your system (e.g. `-lpthread` on Linux).

## 4.1.3 How Many Threads to Use?

There is a fair amount of overhead involved in spawning and synchronizing threads, so the optimal number of threads to use depends upon the size of the transform as well as on the number of processors you have.

As a general rule, you don't want to use more threads than you have processors. (Using more threads will work, but there will be extra overhead with no benefit.) In fact, if the problem size is too small, you may want to use fewer threads than you have processors.

You will have to experiment with your system to see what level of parallelization is best for your problem size. Useful tools to help you do this are the test programs that are automatically compiled along with the threads libraries, `fftw_threads_test` and `rfftw_threads_test` (in the `threads` subdirectory). These take the same arguments as the other FFTW test programs (see `tests/README`), except that they also take the number of threads to use as a first argument, and report the parallel speedup in speed tests. For example,

```
     fftw_threads_test 2 -s 128x128
```

[1] There is one exception: when performing one-dimensional in-place transforms, the `out` parameter is always ignored by the multi-threaded routines, instead of being used as a workspace if it is non-`NULL` as in the uniprocessor routines. The multi-threaded routines always allocate their own workspace (the size of which depends upon the number of threads).

will benchmark complex 128x128 transforms using two threads and report the speedup relative to the uniprocessor transform.

For instance, on a 4-processor 200MHz Pentium Pro system running Linux 2.2.0, we found that the "crossover" point at which 2 threads became beneficial for complex transforms was about 4k points, while 4 threads became beneficial at 8k points.

### 4.1.4 Using Multi-threaded FFTW in a Multi-threaded Program

It is perfectly possible to use the multi-threaded FFTW routines from a multi-threaded program (e.g. have multiple threads computing multi-threaded transforms simultaneously). If you have the processors, more power to you! However, the same restrictions apply as for the uniprocessor FFTW routines (see Section 3.8 [Thread safety], page 36). In particular, you should recall that you may not create or destroy plans in parallel.

### 4.1.5 Tips for Optimal Threading

Not all transforms are equally well-parallelized by the multi-threaded FFTW routines. (This is merely a consequence of laziness on the part of the implementors, and is not inherent to the algorithms employed.) Mainly, the limitations are in the parallel one-dimensional transforms. The things to avoid if you want optimal parallelization are as follows:

### 4.1.6 Parallelization deficiencies in one-dimensional transforms

- Large prime factors can sometimes parallelize poorly. Of course, you should avoid these anyway if you want high performance.
- Single in-place transforms don't parallelize completely. (Multiple in-place transforms, i.e. `howmany > 1`, are fine.) Again, you should avoid these in any case if you want high performance, as they require transforming to a scratch array and copying back.
- Single real-complex (`rfftw`) transforms don't parallelize completely. This is unfortunate, but parallelizing this correctly would have involved a lot of extra code (and a much larger library). You still get some benefit from additional processors, but if you have a very large number of processors you will probably be better off using the parallel complex (`fftw`) transforms. Note that multi-dimensional real transforms or multiple one-dimensional real transforms are fine.

## 4.2 MPI FFTW

This section describes the MPI FFTW routines for distributed-memory (and shared-memory) machines supporting MPI (Message Passing Interface). The MPI routines are significantly different from the ordinary FFTW because the transform data here are *distributed* over multiple processes, so that each process gets only a portion of the array. Currently, multi-dimensional transforms of both real and complex data, as well as one-dimensional transforms of complex data, are supported.

### 4.2.1 MPI FFTW Installation

The FFTW MPI library code is all located in the `mpi` subdirectoy of the FFTW package (along with source code for test programs). On Unix systems, the FFTW MPI libraries and header files can be automatically configured, compiled, and installed along with the uniprocessor FFTW libraries simply by including `--enable-mpi` in the flags to the `configure` script (see Section 6.1 [Installation on Unix], page 55).

The only requirement of the FFTW MPI code is that you have the standard MPI 1.1 (or later) libraries and header files installed on your system. A free implementation of MPI is available from the MPICH home page.

Previous versions of the FFTW MPI routines have had an unfortunate tendency to expose bugs in MPI implementations. The current version has been largely rewritten, and hopefully avoids some of the problems. If you run into difficulties, try passing the optional workspace to `(r)fftwnd_mpi` (see below), as this allows us to use the standard (and hopefully well-tested) `MPI_Alltoall` primitive for communications. Please let us know (`fftw@fftw.org`) how things work out.

Several test programs are included in the `mpi` directory. The ones most useful to you are probably the `fftw_mpi_test` and `rfftw_mpi_test` programs, which are run just like an ordinary MPI program and accept the same parameters as the other FFTW test programs (c.f. `tests/README`). For example, `mpirun` *...params...* `fftw_mpi_test -r 0` will run non-terminating complex-transform correctness tests of random dimensions. They can also do performance benchmarks.

### 4.2.2 Usage of MPI FFTW for Complex Multi-dimensional Transforms

Usage of the MPI FFTW routines is similar to that of the uniprocessor FFTW. We assume that the reader already understands the usage of the uniprocessor FFTW routines, described elsewhere in this manual. Some familiarity with MPI is also helpful.

A typical program performing a complex two-dimensional MPI transform might look something like:

```
#include <fftw_mpi.h>

int main(int argc, char **argv)
{
      const int NX = ..., NY = ...;
      fftwnd_mpi_plan plan;
      fftw_complex *data;

      MPI_Init(&argc,&argv);

      plan = fftw2d_mpi_create_plan(MPI_COMM_WORLD,
                                    NX, NY,
                                    FFTW_FORWARD, FFTW_ESTIMATE);

      ...allocate and initialize data...

      fftwnd_mpi(p, 1, data, NULL, FFTW_NORMAL_ORDER);

      ...
```

```
             fftwnd_mpi_destroy_plan(plan);
             MPI_Finalize();
     }
```

The calls to `MPI_Init` and `MPI_Finalize` are required in all MPI programs; see the MPI home page for more information. Note that all of your processes run the program in parallel, as a group; there is no explicit launching of threads/processes in an MPI program.

As in the ordinary FFTW, the first thing we do is to create a plan (of type `fftwnd_mpi_plan`), using:

```
fftwnd_mpi_plan fftw2d_mpi_create_plan(MPI_Comm comm,
                                       int nx, int ny,
                                       fftw_direction dir, int flags);
```

Except for the first argument, the parameters are identical to those of `fftw2d_create_plan`. (There are also analogous `fftwnd_mpi_create_plan` and `fftw3d_mpi_create_plan` functions. Transforms of any rank greater than one are supported.) The first argument is an MPI *communicator*, which specifies the group of processes that are to be involved in the transform; the standard constant `MPI_COMM_WORLD` indicates all available processes.

Next, one has to allocate and initialize the data. This is somewhat tricky, because the transform data is distributed across the processes involved in the transform. It is discussed in detail by the next section (see Section 4.2.3 [MPI Data Layout], page 43).

The actual computation of the transform is performed by the function `fftwnd_mpi`, which differs somewhat from its uniprocessor equivalent and is described by:

```
void fftwnd_mpi(fftwnd_mpi_plan p,
                int n_fields,
                fftw_complex *local_data, fftw_complex *work,
                fftwnd_mpi_output_order output_order);
```

There are several things to notice here:

- First of all, all `fftw_mpi` transforms are in-place: the output is in the `local_data` parameter, and there is no need to specify `FFTW_IN_PLACE` in the plan flags.
- The MPI transforms also only support a limited subset of the `howmany`/`stride`/`dist` functionality of the uniprocessor routines: the `n_fields` parameter is equivalent to `howmany=n_fields`, `stride=n_fields`, and `dist=1`. (Conceptually, the `n_fields` parameter allows you to transform an array of contiguous vectors, each with length `n_fields`.) `n_fields` is `1` if you are only transforming a single, ordinary array.
- The `work` parameter is an optional workspace. If it is not `NULL`, it should be exactly the same size as the `local_data` array. If it is provided, FFTW is able to use the built-in `MPI_Alltoall` primitive for (often) greater efficiency at the expense of extra storage space.
- Finally, the last parameter specifies whether the output data has the same ordering as the input data (`FFTW_NORMAL_ORDER`), or if it is transposed (`FFTW_TRANSPOSED_ORDER`). Leaving the data transposed results in significant performance improvements due to a saved communication step (needed to un-transpose the data). Specifically, the first two dimensions of the array are transposed, as is described in more detail by the next section.

The output of fftwnd_mpi is identical to that of the corresponding uniprocessor transform. In particular, you should recall our conventions for normalization and the sign of the transform exponent.

The same plan can be used to compute many transforms of the same size. After you are done with it, you should deallocate it by calling fftwnd_mpi_destroy_plan.

**Important:** The FFTW MPI routines must be called in the same order by all processes involved in the transform. You should assume that they all are blocking, as if each contained a call to MPI_Barrier.

Programs using the FFTW MPI routines should be linked with -lfftw_mpi -lfftw -lm on Unix, in addition to whatever libraries are required for MPI.

### 4.2.3 MPI Data Layout

The transform data used by the MPI FFTW routines is *distributed*: a distinct portion of it resides with each process involved in the transform. This allows the transform to be parallelized, for example, over a cluster of workstations, each with its own separate memory, so that you can take advantage of the total memory of all the processors you are parallelizing over.

In particular, the array is divided according to the rows (first dimension) of the data: each process gets a subset of the rows of the data. (This is sometimes called a "slab decomposition.") One consequence of this is that you can't take advantage of more processors than you have rows (e.g. 64x64x64 matrix can at most use 64 processors). This isn't usually much of a limitation, however, as each processor needs a fair amount of data in order for the parallel-computation benefits to outweight the communications costs.

Below, the first dimension of the data will be referred to as 'x' and the second dimension as 'y'.

FFTW supplies a routine to tell you exactly how much data resides on the current process:

```
void fftwnd_mpi_local_sizes(fftwnd_mpi_plan p,
                            int *local_nx,
                            int *local_x_start,
                            int *local_ny_after_transpose,
                            int *local_y_start_after_transpose,
                            int *total_local_size);
```

Given a plan p, the other parameters of this routine are set to values describing the required data layout, described below.

total_local_size is the number of fftw_complex elements that you must allocate for your local data (and workspace, if you choose). (This value should, of course, be multiplied by n_fields if that parameter to fftwnd_mpi is not 1.)

The data on the current process has local_nx rows, starting at row local_x_start. If fftwnd_mpi is called with FFTW_TRANSPOSED_ORDER output, then y will be the first dimension of the output, and the local y extent will be given by local_ny_after_transpose and local_y_start_after_transpose. Otherwise, the output has the same dimensions and layout as the input.

For instance, suppose you want to transform three-dimensional data of size `nx x ny x nz`. Then, the current process will store a subset of this data, of size `local_nx x ny x nz`, where the `x` indices correspond to the range `local_x_start` to `local_x_start+local_nx-1` in the "real" (i.e. logical) array. If `fftwnd_mpi` is called with `FFTW_TRANSPOSED_ORDER` output, then the result will be a `ny x nx x nz` array, of which a `local_ny_after_transpose x nx x nz` subset is stored on the current process (corresponding to `y` values starting at `local_y_start_after_transpose`).

The following is an example of allocating such a three-dimensional array array (`local_data`) before the transform and initializing it to some function `f(x,y,z)`:

```
fftwnd_mpi_local_sizes(plan, &local_nx, &local_x_start,
                       &local_ny_after_transpose,
                       &local_y_start_after_transpose,
                       &total_local_size);

local_data = (fftw_complex*) malloc(sizeof(fftw_complex) *
                                    total_local_size);

for (x = 0; x < local_nx; ++x)
        for (y = 0; y < ny; ++y)
                for (z = 0; z < nz; ++z)
                        local_data[(x*ny + y)*nz + z]
                                = f(x + local_x_start, y, z);
```

Some important things to remember:

- Although the local data is of dimensions `local_nx x ny x nz` in the above example, do *not* allocate the array to be of size `local_nx*ny*nz`. Use `total_local_size` instead.
- The amount of data on each process will not necessarily be the same; in fact, `local_nx` may even be zero for some processes. (For example, suppose you are doing a `6x6` transform on four processors. There is no way to effectively use the fourth processor in a slab decomposition, so we leave it empty. Proof left as an exercise for the reader.)
- All arrays are, of course, in row-major order (see Section 2.5 [Multi-dimensional Array Format], page 11).
- If you want to compute the inverse transform of the output of `fftwnd_mpi`, the dimensions of the inverse transform are given by the dimensions of the output of the forward transform. For example, if you are using `FFTW_TRANSPOSED_ORDER` output in the above example, then the inverse plan should be created with dimensions `ny x nx x nz`.
- The data layout only depends upon the dimensions of the array, not on the plan, so you are guaranteed that different plans for the same size (or inverse plans) will use the same (consistent) data layouts.

### 4.2.4 Usage of MPI FFTW for Real Multi-dimensional Transforms

MPI transforms specialized for real data are also available, similiar to the uniprocessor `rfftwnd` transforms. Just as in the uniprocessor case, the real-data MPI functions gain roughly a factor of two in speed (and save a factor of two in space) at the expense of more complicated data formats in the calling program. Before reading this section, you should definitely understand how to call the uniprocessor `rfftwnd` functions and also the complex MPI FFTW functions.

The following is an example of a program using `rfftwnd_mpi`. It computes the size `nx x ny x nz` transform of a real function `f(x,y,z)`, multiplies the imaginary part by `2` for fun, then computes the inverse transform. (We'll also use `FFTW_TRANSPOSED_ORDER` output for the transform, and additionally supply the optional workspace parameter to `rfftwnd_mpi`, just to add a little spice.)

```
#include <rfftw_mpi.h>

int main(int argc, char **argv)
{
     const int nx = ..., ny = ..., nz = ...;
     int local_nx, local_x_start, local_ny_after_transpose,
         local_y_start_after_transpose, total_local_size;
     int x, y, z;
     rfftwnd_mpi_plan plan, iplan;
     fftw_real *data, *work;
     fftw_complex *cdata;

     MPI_Init(&argc,&argv);

     /* create the forward and backward plans: */
     plan = rfftw3d_mpi_create_plan(MPI_COMM_WORLD,
                                    nx, ny, nz,
                                    FFTW_REAL_TO_COMPLEX,
                                    FFTW_ESTIMATE);
     iplan = rfftw3d_mpi_create_plan(MPI_COMM_WORLD,
      /* dim.'s of REAL data --> */  nx, ny, nz,
                                     FFTW_COMPLEX_TO_REAL,
                                     FFTW_ESTIMATE);

     rfftwnd_mpi_local_sizes(plan, &local_nx, &local_x_start,
                            &local_ny_after_transpose,
                            &local_y_start_after_transpose,
                            &total_local_size);

     data = (fftw_real*) malloc(sizeof(fftw_real) * total_local_size);

     /* workspace is the same size as the data: */
     work = (fftw_real*) malloc(sizeof(fftw_real) * total_local_size);

     /* initialize data to f(x,y,z): */
     for (x = 0; x < local_nx; ++x)
             for (y = 0; y < ny; ++y)
                     for (z = 0; z < nz; ++z)
                             data[(x*ny + y) * (2*(nz/2+1)) + z]
                                     = f(x + local_x_start, y, z);
```

```
          /* Now, compute the forward transform: */
          rfftwnd_mpi(plan, 1, data, work, FFTW_TRANSPOSED_ORDER);

          /* the data is now complex, so typecast a pointer: */
          cdata = (fftw_complex*) data;

          /* multiply imaginary part by 2, for fun:
             (note that the data is transposed) */
          for (y = 0; y < local_ny_after_transpose; ++y)
                  for (x = 0; x < nx; ++x)
                          for (z = 0; z < (nz/2+1); ++z)
                                  cdata[(y*nx + x) * (nz/2+1) + z].im
                                          *= 2.0;

          /* Finally, compute the inverse transform; the result
             is transposed back to the original data layout: */
          rfftwnd_mpi(iplan, 1, data, work, FFTW_TRANSPOSED_ORDER);

          free(data);
          free(work);
          rfftwnd_mpi_destroy_plan(plan);
          rfftwnd_mpi_destroy_plan(iplan);
          MPI_Finalize();
     }
```

There's a lot of stuff in this example, but it's all just what you would have guessed, right? We replaced all the `fftwnd_mpi*` functions by `rfftwnd_mpi*`, but otherwise the parameters were pretty much the same. The data layout distributed among the processes just like for the complex transforms (see Section 4.2.3 [MPI Data Layout], page 43), but in addition the final dimension is padded just like it is for the uniprocessor in-place real transforms (see Section 3.5.3 [Array Dimensions for Real Multi-dimensional Transforms], page 32). In particular, the `z` dimension of the real input data is padded to a size `2*(nz/2+1)`, and after the transform it contains `nz/2+1` complex values.

Some other important things to know about the real MPI transforms:

- As for the uniprocessor `rfftwnd_create_plan`, the dimensions passed for the `FFTW_COMPLEX_TO_REAL` plan are those of the *real* data. In particular, even when `FFTW_TRANSPOSED_ORDER` is used as in this case, the dimensions are those of the (untransposed) real output, not the (transposed) complex input. (For the complex MPI transforms, on the other hand, the dimensions are always those of the input array.)
- The output ordering of the transform (`FFTW_TRANSPOSED_ORDER` or `FFTW_TRANSPOSED_ORDER`) *must* be the same for both forward and backward transforms. (This is not required in the complex case.)
- `total_local_size` is the required size in `fftw_real` values, not `fftw_complex` values as it is for the complex transforms.
- `local_ny_after_transpose` and `local_y_start_after_transpose` describe the portion of the array after the transform; that is, they are indices in the complex array for an `FFTW_REAL_TO_COMPLEX` transform and in the real array for an `FFTW_COMPLEX_TO_REAL` transform.

- `rfftwnd_mpi` always expects `fftw_real*` array arguments, but of course these pointers can refer to either real or complex arrays, depending upon which side of the transform you are on. Just as for in-place uniprocessor real transforms (and also in the example above), this is most easily handled by typecasting to a complex pointer when handling the complex data.
- As with the complex transforms, there are also `rfftwnd_create_plan` and `rfftw2d_create_plan` functions, and any rank greater than one is supported.

Programs using the MPI FFTW real transforms should link with `-lrfftw_mpi -lfftw_mpi -lrfftw -lfftw -lm` on Unix.

### 4.2.5 Usage of MPI FFTW for Complex One-dimensional Transforms

The MPI FFTW also includes routines for parallel one-dimensional transforms of complex data (only). Although the speedup is generally worse than it is for the multi-dimensional routines,[2] these distributed-memory one-dimensional transforms are especially useful for performing one-dimensional transforms that don't fit into the memory of a single machine.

The usage of these routines is straightforward, and is similar to that of the multi-dimensional MPI transform functions. You first include the header `<fftw_mpi.h>` and then create a plan by calling:

```
fftw_mpi_plan fftw_mpi_create_plan(MPI_Comm comm, int n,
                                   fftw_direction dir, int flags);
```

The last three arguments are the same as for `fftw_create_plan` (except that all MPI transforms are automatically `FFTW_IN_PLACE`). The first argument specifies the group of processes you are using, and is usually `MPI_COMM_WORLD` (all processes). A plan can be used for many transforms of the same size, and is destroyed when you are done with it by calling `fftw_mpi_destroy_plan(plan)`.

If you don't care about the ordering of the input or output data of the transform, you can include `FFTW_SCRAMBLED_INPUT` and/or `FFTW_SCRAMBLED_OUTPUT` in the `flags`. These save some communications at the expense of having the input and/or output reordered in an undocumented way. For example, if you are performing an FFT-based convolution, you might use `FFTW_SCRAMBLED_OUTPUT` for the forward transform and `FFTW_SCRAMBLED_INPUT` for the inverse transform.

The transform itself is computed by:

```
void fftw_mpi(fftw_mpi_plan p, int n_fields,
              fftw_complex *local_data, fftw_complex *work);
```

`n_fields`, as in `fftwnd_mpi`, is equivalent to `howmany=n_fields`, `stride=n_fields`, and `dist=1`, and should be `1` when you are computing the transform of a single array. `local_data` contains the portion of the array local to the current process, described below. `work` is either `NULL` or an array exactly the same size as `local_data`; in the latter case,

[2] The 1D transforms require much more communication. All the communication in our FFT routines takes the form of an all-to-all communication: the multi-dimensional transforms require two all-to-all communications (or one, if you use `FFTW_TRANSPOSED_ORDER`), while the one-dimensional transforms require *three* (or two, if you use scrambled input or output).

FFTW can use the `MPI_Alltoall` communications primitive which is (usually) faster at the expense of extra storage. Upon return, `local_data` contains the portion of the output local to the current process (see below).

To find out what portion of the array is stored local to the current process, you call the following routine:

```
void fftw_mpi_local_sizes(fftw_mpi_plan p,
                          int *local_n, int *local_start,
                          int *local_n_after_transform,
                          int *local_start_after_transform,
                          int *total_local_size);
```

`total_local_size` is the number of `fftw_complex` elements you should actually allocate for `local_data` (and `work`). `local_n` and `local_start` indicate that the current process stores `local_n` elements corresponding to the indices `local_start` to `local_start+local_n-1` in the "real" array. *After the transform, the process may store a different portion of the array.* The portion of the data stored on the process after the transform is given by `local_n_after_transform` and `local_start_after_transform`. This data is exactly the same as a contiguous segment of the corresponding uniprocessor transform output (i.e. an in-order sequence of sequential frequency bins).

Note that, if you compute both a forward and a backward transform of the same size, the local sizes are guaranteed to be consistent. That is, the local size after the forward transform will be the same as the local size before the backward transform, and vice versa.

Programs using the FFTW MPI routines should be linked with `-lfftw_mpi -lfftw -lm` on Unix, in addition to whatever libraries are required for MPI.

## 4.2.6 MPI Tips

There are several things you should consider in order to get the best performance out of the MPI FFTW routines.

First, if possible, the first and second dimensions of your data should be divisible by the number of processes you are using. (If only one can be divisible, then you should choose the first dimension.) This allows the computational load to be spread evenly among the processes, and also reduces the communications complexity and overhead. In the one-dimensional transform case, the size of the transform should ideally be divisible by the *square* of the number of processors.

Second, you should consider using the `FFTW_TRANSPOSED_ORDER` output format if it is not too burdensome. The speed gains from communications savings are usually substantial.

Third, you should consider allocating a workspace for `(r)fftw(nd)_mpi`, as this can often (but not always) improve performance (at the cost of extra storage).

Fourth, you should experiment with the best number of processors to use for your problem. (There comes a point of diminishing returns, when the communications costs outweigh the computational benefits.[3]) The `fftw_mpi_test` program can output helpful performance benchmarks. It accepts the same parameters as the uniprocessor test programs (c.f. `tests/README`) and is run like an ordinary MPI program. For example, `mpirun -np`

[3] An FFT is particularly hard on communications systems, as it requires an *all-to-all* communication, which is more or less the worst possible case.

4 `fftw_mpi_test -s 128x128x128` will benchmark a `128x128x128` transform on four processors, reporting timings and parallel speedups for all variants of `fftwnd_mpi` (transposed, with workspace, etcetera). (Note also that there is the `rfftw_mpi_test` program for the real transforms.)

# 5 Calling FFTW from Fortran

The standard FFTW libraries include special wrapper functions that allow Fortran programs to call FFTW subroutines. This chapter describes how those functions may be employed to use FFTW from Fortran. We assume here that the reader is already familiar with the usage of FFTW in C, as described elsewhere in this manual.

In general, it is not possible to call C functions directly from Fortran, due to Fortran's inability to pass arguments by value and also because Fortran compilers typically expect identifiers to be mangled somehow for linking. However, if C functions are written in a special way, they *are* callable from Fortran, and we have employed this technique to create Fortran-callable "wrapper" functions around the main FFTW routines. These wrapper functions are included in the FFTW libraries by default, unless a Fortran compiler isn't found on your system or `--disable-fortran` is included in the `configure` flags.

As a result, calling FFTW from Fortran requires little more than appending '`_f77`' to the function names and then linking normally with the FFTW libraries. There are a few wrinkles, however, as we shall discuss below.

## 5.1 Wrapper Routines

All of the uniprocessor and multi-threaded transform routines have Fortran-callable wrappers, except for the wisdom import/export functions (since it is not possible to exchange string and file arguments portably with Fortran) and the specific planner routines (see Section 3.2.2 [Discussion on Specific Plans], page 20). The name of the wrapper routine is the same as that of the corresponding C routine, but with `fftw/fftwnd/rfftw/rfftwnd` replaced by `fftw_f77/fftwnd_f77/rfftw_f77/rfftwnd_f77`. For example, in Fortran, instead of calling `fftw_one` you would call `fftw_f77_one`.[1] For the most part, all of the arguments to the functions are the same, with the following exceptions:

- `plan` variables (what would be of type `fftw_plan`, `rfftwnd_plan`, etcetera, in C), must be declared as a type that is the same size as a pointer (address) on your machine. (Fortran has no generic pointer type.) The Fortran `integer` type is usually the same size as a pointer, but you need to be wary (especially on 64-bit machines). (You could also use `integer*4` on a 32-bit machine and `integer*8` on a 64-bit machine.) Ugh. (`g77` has a special type, `integer(kind=7)`, that is defined to be the same size as a pointer.)
- Any function that returns a value (e.g. `fftw_create_plan`) is converted into a subroutine. The return value is converted into an additional (first) parameter of the wrapper subroutine. (The reason for this is that some Fortran implementations seem to have trouble with C function return values.)
- When performing one-dimensional `FFTW_IN_PLACE` transforms, you don't have the option of passing `NULL` for the `out` argument (since there is no way to pass `NULL` from Fortran). Therefore, when performing such transforms, you *must* allocate and pass a contiguous scratch array of the same size as the transform. Note that for in-place multi-dimensional (`(r)fftwnd`) transforms, the `out` argument is ignored, so you can pass anything for that parameter.

[1] Technically, Fortran 77 identifiers are not allowed to have more than 6 characters, nor may they contain underscores. Any compiler that enforces this limitation doesn't deserve to link to FFTW.

- The wrapper routines expect multi-dimensional arrays to be in column-major order, which is the ordinary format of Fortran arrays. They do this transparently and costlessly simply by reversing the order of the dimensions passed to FFTW, but this has one important consequence for multi-dimensional real-complex transforms, discussed below.

In general, you should take care to use Fortran data types that correspond to (i.e. are the same size as) the C types used by FFTW. If your C and Fortran compilers are made by the same vendor, the correspondence is usually straightforward (i.e. `integer` corresponds to `int`, `real` corresponds to `float`, etcetera). Such simple correspondences are assumed in the examples below. The examples also assume that FFTW was compiled in double precision (the default).

## 5.2 FFTW Constants in Fortran

When creating plans in FFTW, a number of constants are used to specify options, such as `FFTW_FORWARD` or `FFTW_USE_WISDOM`. The same constants must be used with the wrapper routines, but of course the C header files where the constants are defined can't be incorporated directly into Fortran code.

Instead, we have placed Fortran equivalents of the FFTW constant definitions in the file `fortran/fftw_f77.i` of the FFTW package. If your Fortran compiler supports a preprocessor, you can use that to incorporate this file into your code whenever you need to call FFTW. Otherwise, you will have to paste the constant definitions in directly. They are:

```
integer FFTW_FORWARD,FFTW_BACKWARD
parameter (FFTW_FORWARD=-1,FFTW_BACKWARD=1)

integer FFTW_REAL_TO_COMPLEX,FFTW_COMPLEX_TO_REAL
parameter (FFTW_REAL_TO_COMPLEX=-1,FFTW_COMPLEX_TO_REAL=1)

integer FFTW_ESTIMATE,FFTW_MEASURE
parameter (FFTW_ESTIMATE=0,FFTW_MEASURE=1)

integer FFTW_OUT_OF_PLACE,FFTW_IN_PLACE,FFTW_USE_WISDOM
parameter (FFTW_OUT_OF_PLACE=0)
parameter (FFTW_IN_PLACE=8,FFTW_USE_WISDOM=16)

integer FFTW_THREADSAFE
parameter (FFTW_THREADSAFE=128)
```

In C, you combine different flags (like `FFTW_USE_WISDOM` and `FFTW_MEASURE`) using the '`|`' operator; in Fortran you should just use '`+`'.

## 5.3 Fortran Examples

In C you might have something like the following to transform a one-dimensional complex array:

```
fftw_complex in[N], *out[N];
fftw_plan plan;
```

```
plan = fftw_create_plan(N,FFTW_FORWARD,FFTW_ESTIMATE);
fftw_one(plan,in,out);
fftw_destroy_plan(plan);
```

In Fortran, you use the following to accomplish the same thing:

```
double complex in, out
dimension in(N), out(N)
integer plan

call fftw_f77_create_plan(plan,N,FFTW_FORWARD,FFTW_ESTIMATE)
call fftw_f77_one(plan,in,out)
call fftw_f77_destroy_plan(plan)
```

Notice how all routines are called as Fortran subroutines, and the plan is returned via the first argument to `fftw_f77_create_plan`. *Important:* these examples assume that `integer` is the same size as a pointer, and may need modification on a 64-bit machine. See Section 5.1 [Wrapper Routines], page 51, above. To do the same thing, but using 8 threads in parallel (see Section 4.1 [Multi-threaded FFTW], page 37), you would simply replace the call to `fftw_f77_one` with:

```
call fftw_f77_threads_one(8,plan,in,out)
```

To transform a three-dimensional array in-place with C, you might do:

```
fftw_complex arr[L][M][N];
fftwnd_plan plan;
int n[3] = {L,M,N};

plan = fftwnd_create_plan(3,n,FFTW_FORWARD,
                          FFTW_ESTIMATE | FFTW_IN_PLACE);
fftwnd_one(plan, arr, 0);
fftwnd_destroy_plan(plan);
```

In Fortran, you would use this instead:

```
 double complex arr
 dimension arr(L,M,N)
 integer n
 dimension n(3)
 integer plan

 n(1) = L
 n(2) = M
 n(3) = N
 call fftwnd_f77_create_plan(plan,3,n,FFTW_FORWARD,
+                            FFTW_ESTIMATE + FFTW_IN_PLACE)
 call fftwnd_f77_one(plan, arr, 0)
 call fftwnd_f77_destroy_plan(plan)
```

Instead of calling `fftwnd_f77_create_plan(plan,3,n,...)`, we could also have called `fftw3d_f77_create_plan(plan,L,M,N,...)`.

Note that we pass the array dimensions in the "natural" order; also note that the last argument to `fftwnd_f77` is ignored since the transform is `FFTW_IN_PLACE`.

To transform a one-dimensional real array in Fortran, you might do:

```
          double precision in, out
          dimension in(N), out(N)
          integer plan

          call rfftw_f77_create_plan(plan,N,FFTW_REAL_TO_COMPLEX,
         +                           FFTW_ESTIMATE)
          call rfftw_f77_one(plan,in,out)
          call rfftw_f77_destroy_plan(plan)
```

To transform a two-dimensional real array, out of place, you might use the following:

```
          double precision in
          double complex out
          dimension in(M,N), out(M/2 + 1, N)
          integer plan

          call rfftw2d_f77_create_plan(plan,M,N,FFTW_REAL_TO_COMPLEX,
         +                             FFTW_ESTIMATE)
          call rfftwnd_f77_one_real_to_complex(plan, in, out)
          call rfftwnd_f77_destroy_plan(plan)
```

**Important:** Notice that it is the *first* dimension of the complex output array that is cut in half in Fortran, rather than the last dimension as in C. This is a consequence of the wrapper routines reversing the order of the array dimensions passed to FFTW so that the Fortran program can use its ordinary column-major order.

# 6 Installation and Customization

This chapter describes the installation and customization of FFTW, the latest version of which may be downloaded from the FFTW home page.

As distributed, FFTW makes very few assumptions about your system. All you need is an ANSI C compiler (`gcc` is fine, although vendor-provided compilers often produce faster code). However, installation of FFTW is somewhat simpler if you have a Unix or a GNU system, such as Linux. In this chapter, we first describe the installation of FFTW on Unix and non-Unix systems. We then describe how you can customize FFTW to achieve better performance. Specifically, you can I) enable `gcc`/x86-specific hacks that improve performance on Pentia and PentiumPro's; II) adapt FFTW to use the high-resolution clock of your machine, if any; III) produce code (*codelets*) to support fast transforms of sizes that are not supported efficiently by the standard FFTW distribution.

## 6.1 Installation on Unix

FFTW comes with a `configure` program in the GNU style. Installation can be as simple as:

```
./configure
make
make install
```

This will build the uniprocessor complex and real transform libraries along with the test programs. We strongly recommend that you use GNU `make` if it is available; on some systems it is called `gmake`. The "`make install`" command installs the fftw and rfftw libraries in standard places, and typically requires root privileges (unless you specify a different install directory with the `--prefix` flag to `configure`). You can also type "`make check`" to put the FFTW test programs through their paces. If you have problems during configuration or compilation, you may want to run "`make distclean`" before trying again; this ensures that you don't have any stale files left over from previous compilation attempts.

The `configure` script knows good `CFLAGS` (C compiler flags) for a few systems. If your system is not known, the `configure` script will print out a warning.[1] In this case, you can compile FFTW with the command

```
make CFLAGS="<write your CFLAGS here>"
```

If you do find an optimal set of `CFLAGS` for your system, please let us know what they are (along with the output of `config.guess`) so that we can include them in future releases.

The `configure` program supports all the standard flags defined by the GNU Coding Standards; see the `INSTALL` file in FFTW or the GNU web page. Note especially `--help` to list all flags and `--enable-shared` to create shared, rather than static, libraries. `configure` also accepts a few FFTW-specific flags, particularly:

- `--enable-float` Produces a single-precision version of FFTW (`float`) instead of the default double-precision (`double`). See Section 6.3 [Installing FFTW in both single and double precision], page 57.

[1] Each version of `cc` seems to have its own magic incantation to get the fastest code most of the time—you'd think that people would have agreed upon some convention, e.g. `"-Omax"`, by now.

- `--enable-type-prefix` Adds a '`d`' or '`s`' prefix to all installed libraries and header files to indicate the floating-point precision. See Section 6.3 [Installing FFTW in both single and double precision], page 57. (`--enable-type-prefix=<prefix>` lets you add an arbitrary prefix.) By default, no prefix is used.
- `--enable-threads` Enables compilation and installation of the FFTW threads library (see Section 4.1 [Multi-threaded FFTW], page 37), which provides a simple interface to parallel transforms for SMP systems. (By default, the threads routines are not compiled.)
- `--with-openmp`, `--with-sgimp` In conjunction with `--enable-threads`, causes the multi-threaded FFTW library to use OpenMP or SGI MP compiler directives in order to induce parallelism, rather than spawning its own threads directly. (Useful especially for programs already employing such directives, in order to minimize conflicts between different parallelization mechanisms.)
- `--enable-mpi` Enables compilation and installation of the FFTW MPI library (see Section 4.2 [MPI FFTW], page 40), which provides parallel transforms for distributed-memory systems with MPI. (By default, the MPI routines are not compiled.)
- `--disable-fortran` Disables inclusion of Fortran-callable wrapper routines (see Chapter 5 [Calling FFTW from Fortran], page 51) in the standard FFTW libraries. These wrapper routines increase the library size by only a negligible amount, so they are included by default as long as the `configure` script finds a Fortran compiler on your system.
- `--with-gcc` Enables the use of `gcc`. By default, FFTW uses the vendor-supplied `cc` compiler if present. Unfortunately, `gcc` produces slower code than `cc` on many systems.
- `--enable-i386-hacks` See Section 6.4 [gcc and Pentium hacks], page 57, below.
- `--enable-pentium-timer` See Section 6.4 [gcc and Pentium hacks], page 57, below.

To force `configure` to use a particular C compiler (instead of the default, usually `cc`), set the environment variable `CC` to the name of the desired compiler before running `configure`; you may also need to set the flags via the variable `CFLAGS`.

## 6.2 Installation on non-Unix Systems

It is quite straightforward to install FFTW even on non-Unix systems lacking the niceties of the `configure` script. The FFTW Home Page may include some FFTW packages preconfigured for particular systems/compilers, and also contains installation notes sent in by users. All you really need to do, though, is to compile all of the `.c` files in the appropriate directories of the FFTW package. (You needn't worry about the many extraneous files lying around.)

For the complex transforms, compile all of the `.c` files in the `fftw` directory and link them into a library. Similarly, for the real transforms, compile all of the `.c` files in the `rfftw` directory into a library. Note that these sources `#include` various files in the `fftw` and `rfftw` directories, so you may need to set up the `#include` paths for your compiler appropriately. Be sure to enable the highest-possible level of optimization in your compiler.

By default, FFTW is compiled for double-precision transforms. To work in single precision rather than double precision, `#define` the symbol `FFTW_ENABLE_FLOAT` in `fftw.h` (in the `fftw` directory) and (re)compile FFTW.

These libraries should be linked with any program that uses the corresponding transforms. The required header files, `fftw.h` and `rfftw.h`, are located in the `fftw` and `rfftw` directories respectively; you may want to put them with the libraries, or wherever header files normally go on your system.

FFTW includes test programs, `fftw_test` and `rfftw_test`, in the `tests` directory. These are compiled and linked like any program using FFTW, except that they use additional header files located in the `fftw` and `rfftw` directories, so you will need to set your compiler `#include` paths appropriately. `fftw_test` is compiled from `fftw_test.c` and `test_main.c`, while `rfftw_test` is compiled from `rfftw_test.c` and `test_main.c`. When you run these programs, you will be prompted interactively for various possible tests to perform; see also `tests/README` for more information.

## 6.3 Installing FFTW in both single and double precision

It is often useful to install both single- and double-precision versions of the FFTW libraries on the same machine, and we provide a convenient mechanism for achieving this on Unix systems.

When the `--enable-type-prefix` option of configure is used, the FFTW libraries and header files are installed with a prefix of '`d`' or '`s`', depending upon whether you compiled in double or single precision. Then, instead of linking your program with `-lrfftw -lfftw`, for example, you would link with `-ldrfftw -ldfftw` to use the double-precision version or with `-lsrfftw -lsfftw` to use the single-precision version. Also, you would `#include <drfftw.h>` or `<srfftw.h>` instead of `<rfftw.h>`, and so on.

*The names of FFTW functions, data types, and constants remain unchanged!* You still call, for instance, `fftw_one` and not `dfftw_one`. Only the names of header files and libraries are modified. One consequence of this is that *you* **cannot** *use both the single- and double-precision FFTW libraries in the same program, simultaneously,* as the function names would conflict.

So, to install both the single- and double-precision libraries on the same machine, you would do:

```
./configure --enable-type-prefix [ other options ]
make
make install
make clean
./configure --enable-float --enable-type-prefix [ other options ]
make
make install
```

## 6.4 `gcc` and Pentium hacks

The `configure` option `--enable-i386-hacks` enables specific optimizations for the Pentium and later x86 CPUs under gcc, which can significantly improve performance of double-precision transforms. Specifically, we have tested these hacks on Linux with `gcc` 2.[789] and versions of `egcs` since 1.0.3. These optimizations affect only the performance and not the correctness of FFTW (i.e. it is always safe to try them out).

These hacks provide a workaround to the incorrect alignment of local `double` variables in `gcc`. The compiler aligns these variables to multiples of 4 bytes, but execution is much faster (on Pentium and PentiumPro) if `doubles` are aligned to a multiple of 8 bytes. By carefully counting the number of variables allocated by the compiler in performance-critical regions of the code, we have been able to introduce dummy allocations (using `alloca`) that align the stack properly. The hack depends crucially on the compiler flags that are used. For example, it won't work without `-fomit-frame-pointer`.

In principle, these hacks are no longer required under `gcc` versions 2.95 and later, which automatically align the stack correctly (see `-mpreferred-stack-boundary` in the `gcc` manual). However, we have encountered a bug in the stack alignment of versions 2.95.[012] that causes FFTW's stack to be misaligned under some circumstances. The `configure` script automatically detects this bug and disables `gcc`'s stack alignment in favor of our own hacks when `--enable-i386-hacks` is used.

The `fftw_test` program outputs speed measurements that you can use to see if these hacks are beneficial.

The `configure` option `--enable-pentium-timer` enables the use of the Pentium and PentiumPro cycle counter for timing purposes. In order to get correct results, you must define `FFTW_CYCLES_PER_SEC` in `fftw/config.h` to be the clock speed of your processor; the resulting FFTW library will be nonportable. The use of this option is deprecated. On serious operating systems (such as Linux), FFTW uses `gettimeofday()`, which has enough resolution and is portable. (Note that Win32 has its own high-resolution timing routines as well. FFTW contains unsupported code to use these routines.)

## 6.5 Customizing the timer

FFTW needs a reasonably-precise clock in order to find the optimal way to compute a transform. On Unix systems, `configure` looks for `gettimeofday` and other system-specific timers. If it does not find any high resolution clock, it defaults to using the `clock()` function, which is very portable, but forces FFTW to run for a long time in order to get reliable measurements.

If your machine supports a high-resolution clock not recognized by FFTW, it is therefore advisable to use it. You must edit `fftw/fftw-int.h`. There are a few macros you must redefine. The code is documented and should be self-explanatory. (By the way, `fftw-int` stands for `fftw-internal`, but for some inexplicable reason people are still using primitive systems with 8.3 filenames.)

Even if you don't install high-resolution timing code, we still recommend that you look at the `FFTW_TIME_MIN` constant in `fftw/fftw-int.h`. This constant holds the minimum time interval (in seconds) required to get accurate timing measurements, and should be (at least) several hundred times the resolution of your clock. The default constants are on the conservative side, and may cause FFTW to take longer than necessary when you create a plan. Set `FFTW_TIME_MIN` to whatever is appropriate on your system (be sure to set the *right* `FFTW_TIME_MIN`...there are several definitions in `fftw-int.h`, corresponding to different platforms and timers).

As an aid in checking the resolution of your clock, you can use the `tests/fftw_test` program with the `-t` option (c.f. `tests/README`). Remember, the mere fact that your clock reports times in, say, picoseconds, does not mean that it is actually *accurate* to that resolution.

## 6.6 Generating your own code

If you know that you will only use transforms of a certain size (say, powers of 2) and want to reduce the size of the library, you can reconfigure FFTW to support only those sizes you are interested in. You may even generate code to enable efficient transforms of a size not supported by the default distribution. The default distribution supports transforms of any size, but not all sizes are equally fast. The default installation of FFTW is best at handling sizes of the form $2^a 3^b 5^c 7^d 11^e 13^f$, where $e+f$ is either 0 or 1, and the other exponents are arbitrary. Other sizes are computed by means of a slow, general-purpose routine. However, if you have an application that requires fast transforms of size, say, `17`, there is a way to generate specialized code to handle that.

The directory `gensrc` contains all the programs and scripts that were used to generate FFTW. In particular, the program `gensrc/genfft.ml` was used to generate the code that FFTW uses to compute the transforms. We do not expect casual users to use it. `genfft` is a rather sophisticated program that generates directed acyclic graphs of FFT algorithms and performs algebraic simplifications on them. `genfft` is written in Objective Caml, a dialect of ML. Objective Caml is described at `http://pauillac.inria.fr/ocaml/` and can be downloaded from from `ftp://ftp.inria.fr/lang/caml-light`.

If you have Objective Caml installed, you can type `sh bootstrap.sh` in the top-level directory to re-generate the files. If you change the `gensrc/config` file, you can optimize FFTW for sizes that are not currently supported efficiently (say, 17 or 19).

We do not provide more details about the code-generation process, since we do not expect that users will need to generate their own code. However, feel free to contact us at `fftw@fftw.org` if you are interested in the subject.

You might find it interesting to learn Caml and/or some modern programming techniques that we used in the generator (including monadic programming), especially if you heard the rumor that Java and object-oriented programming are the latest advancement in the field. The internal operation of the codelet generator is described in the paper, “A Fast Fourier Transform Compiler,” by M. Frigo, which is available from the FFTW home page and will appear in the *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

# 7 Acknowledgments

Matteo Frigo was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grants N00014-94-1-0985 and F30602-97-1-0270, and by a Digital Equipment Corporation Fellowship. Steven G. Johnson was supported in part by a DoD NDSEG Fellowship, an MIT Karl Taylor Compton Fellowship, and by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334.

Both authors were also supported in part by their respective girlfriends, by the letters "Q" and "R", and by the number 12.

We are grateful to SUN Microsystems Inc. for its donation of a cluster of 9 8-processor Ultra HPC 5000 SMPs (24 Gflops peak). These machines served as the primary platform for the development of earlier versions of FFTW.

We thank Intel Corporation for donating a four-processor Pentium Pro machine. We thank the Linux community for giving us a decent OS to run on that machine.

The `genfft` program was written using Objective Caml, a dialect of ML. Objective Caml is a small and elegant language developed by Xavier Leroy. The implementation is available from `ftp.inria.fr` in the directory `lang/caml-light`. We used versions 1.07 and 2.00 of the software. In previous releases of FFTW, `genfft` was written in Caml Light, by the same authors. An even earlier implementation of `genfft` was written in Scheme, but Caml is definitely better for this kind of application.

FFTW uses many tools from the GNU project, including `automake`, `texinfo`, and `libtool`.

Prof. Charles E. Leiserson of MIT provided continuous support and encouragement. This program would not exist without him. Charles also proposed the name "codelets" for the basic FFT blocks.

Prof. John D. Joannopoulos of MIT demonstrated continuing tolerance of Steven's "extra-curricular" computer-science activities. Steven's chances at a physics degree would not exist without him.

Andrew Sterian contributed the Windows timing code.

Didier Miras reported a bug in the test procedure used in FFTW 1.2. We now use a completely different test algorithm by Funda Ergun that does not require a separate FFT program to compare against.

Wolfgang Reimer contributed the Pentium cycle counter and a few fixes that help portability.

Ming-Chang Liu uncovered a well-hidden bug in the complex transforms of FFTW 2.0 and supplied a patch to correct it.

The FFTW FAQ was written in `bfnn` (Bizarre Format With No Name) and formatted using the tools developed by Ian Jackson for the Linux FAQ.

*We are especially thankful to all of our users for their continuing support, feedback, and interest during our development of FFTW.*

# 8 License and Copyright

In addition, we kindly ask you to acknowledge FFTW and its authors in any program or publication in which you use FFTW. (You are not *required* to do so; it is up to your common sense to decide whether you want to comply with this request or not.)

Non-free versions of FFTW are available under terms different than the General Public License. (e.g. they do not require you to accompany any object code using FFTW with the corresponding source code.) For these alternate terms you must purchase a license from MIT's Technology Licensing Office. Users interested in such a license should contact us (`fftw@fftw.org`) for more information.

# 9 Concept Index

# 10 Library Index