FFTW

for version 3.3.4, 5 June 2016

Matteo Frigo
Steven G. Johnson

This manual is for FFTW (version 3.3.4, 5 June 2016).

Table of Contents

1 Introduction

This manual documents version 3.3.4 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) and various special cases thereof.

- FFTW computes the DFT of complex data, real data, even- or odd-symmetric real data (these symmetric transforms are usually known as the discrete cosine or sine transform, respectively), and the discrete Hartley transform (DHT) of real data.
- The input data can have arbitrary length. FFTW employs $O(n \log n)$ algorithms for all lengths, including prime numbers.
- FFTW supports arbitrary multi-dimensional data.
- FFTW supports the SSE, SSE2, AVX, Altivec, and MIPS PS instruction sets.
- FFTW includes parallel (multi-threaded) transforms for shared-memory systems.
- Starting with version 3.3, FFTW includes distributed-memory parallel transforms using MPI.

We assume herein that you are familiar with the properties and uses of the DFT that are relevant to your application. Otherwise, see e.g. *The Fast Fourier Transform and Its Applications* by E. O. Brigham (Prentice-Hall, Englewood Cliffs, NJ, 1988). Our web page also has links to FFT-related information online.

In order to use FFTW effectively, you need to learn one basic concept of FFTW's internal structure: FFTW does not use a fixed algorithm for computing the transform, but instead it adapts the DFT algorithm to details of the underlying hardware in order to maximize performance. Hence, the computation of the transform is split into two phases. First, FFTW's *planner* "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 *executed* to transform the array of input data 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.

FFTW supports transforms of data with arbitrary length, rank, multiplicity, and a general memory layout. In simple cases, however, this generality may be unnecessary and confusing. Consequently, we organized the interface to FFTW into three levels of increasing generality.

- The *basic interface* computes a single transform of contiguous data.
- The *advanced interface* computes transforms of multiple or strided arrays.
- The *guru interface* supports the most general data layouts, multiplicities, and strides.

We expect that most users will be best served by the basic interface, whereas the guru interface requires careful attention to the documentation to avoid problems.

Besides the automatic performance adaptation performed by the planner, it is also possible for advanced users to customize FFTW manually. For example, if code space is a concern, we

provide a tool that links only the subset of FFTW needed by your application. Conversely, you may need to extend FFTW because the standard distribution is not sufficient for your needs. For example, the standard FFTW distribution works most efficiently for arrays whose size can be factored into small primes (2, 3, 5, and 7), and otherwise it uses a slower general-purpose routine. If you need efficient transforms of other sizes, you can use FFTW's code generator, which produces fast C programs ("codelets") 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.

For more information regarding FFTW, see the paper, "The Design and Implementation of FFTW3," by M. Frigo and S. G. Johnson, which was an invited paper in *Proc. IEEE* **93** (2), p. 216 (2005). The code generator is described in the paper "A fast Fourier transform compiler", by M. Frigo, 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 a split-radix algorithm (with a "conjugate-pair" variation pointed out to us by Dan Bernstein). FFTW's code generator also produces new algorithms that we do not 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 (single-processor) implementation. We start by describing the basic interface/features of FFTW in Chapter 2 [Tutorial], page 3. Next, Chapter 3 [Other Important Topics], page 15 discusses data alignment (see Section 3.1 [SIMD alignment and fftw_malloc], page 15), the storage scheme of multi-dimensional arrays (see Section 3.2 [Multi-dimensional Array Format], page 15), and FFTW's mechanism for storing plans on disk (see Section 3.3 [Words of Wisdom-Saving Plans], page 18). Next, Chapter 4 [FFTW Reference], page 21 provides comprehensive documentation of all FFTW's features. Parallel transforms are discussed in their own chapters: Chapter 5 [Multi-threaded FFTW], page 49 and Chapter 6 [Distributed-memory FFTW with MPI], page 53. Fortran programmers can also use FFTW, as described in Chapter 8 [Calling FFTW from Legacy Fortran], page 87 and Chapter 7 [Calling FFTW from Modern Fortran], page 77. Chapter 10 [Installation and Customization], page 97 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 12 [License and Copyright], page 105. Finally, we thank all the people who helped us in Chapter 11 [Acknowledgments], page 103.

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 that are not documented here, although in many cases we try to indicate where added capabilities exist. For more complete information, see Chapter 4 [FFTW Reference], page 21. (Note that you need to compile and install FFTW before you can use it in a program. For the details of the installation, see Chapter 10 [Installation and Customization], page 97.)

We recommend that you read this tutorial in order.[1] At the least, read the first section (see Section 2.1 [Complex One-Dimensional DFTs], page 3) before reading any of the others, even if your main interest lies in one of the other transform types.

Users of FFTW version 2 and earlier may also want to read Chapter 9 [Upgrading from FFTW version 2], page 93.

2.1 Complex One-Dimensional DFTs

> Plan: To bother about the best method of accomplishing an accidental result. [Ambrose Bierce, *The Enlarged Devil's Dictionary*.]

The basic usage of FFTW to compute a one-dimensional DFT of size `N` is simple, and it typically looks something like this code:

```
#include <fftw3.h>
...
{
    fftw_complex *in, *out;
    fftw_plan p;
    ...
    in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
    ...
    fftw_execute(p); /* repeat as needed */
    ...
    fftw_destroy_plan(p);
    fftw_free(in); fftw_free(out);
}
```

You must link this code with the `fftw3` library. On Unix systems, link with `-lfftw3 -lm`.

The example code first allocates the input and output arrays. You can allocate them in any way that you like, but we recommend using `fftw_malloc`, which behaves like `malloc` except that it properly aligns the array when SIMD instructions (such as SSE and Altivec) are available (see Section 3.1 [SIMD alignment and fftw_malloc], page 15). [Alternatively, we provide a convenient wrapper function `fftw_alloc_complex(N)` which has the same effect.]

[1] You can read the tutorial in bit-reversed order after computing your first transform.

The data is an array of type `fftw_complex`, which is by default a `double[2]` composed of the real (`in[i][0]`) and imaginary (`in[i][1]`) parts of a complex number.

The next step is to create a *plan*, which is an object that contains all the data that FFTW needs to compute the FFT. This function creates the plan:

```
fftw_plan fftw_plan_dft_1d(int n, fftw_complex *in, fftw_complex *out,
                           int sign, unsigned 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 (although prime sizes still use an $O(n \log n)$ algorithm).

The next two arguments are pointers to the input and output arrays of the transform. These pointers can be equal, indicating an *in-place* transform.

The fourth argument, `sign`, can be either `FFTW_FORWARD` (`-1`) or `FFTW_BACKWARD` (`+1`), and indicates the direction of the transform you are interested in; technically, it is the sign of the exponent in the transform.

The `flags` argument is usually either `FFTW_MEASURE` or `FFTW_ESTIMATE`. `FFTW_MEASURE` instructs FFTW to run and measure the execution time of several FFTs in order to find the best way to compute the transform of size `n`. This process takes some time (usually a few seconds), depending on your machine and on the size of the transform. `FFTW_ESTIMATE`, on the contrary, does not run any computation and just builds a reasonable plan that is probably sub-optimal. In short, if your program performs many transforms of the same size and initialization time is not important, use `FFTW_MEASURE`; otherwise use the estimate.

You must create the plan before initializing the input, because `FFTW_MEASURE` overwrites the `in`/`out` arrays. (Technically, `FFTW_ESTIMATE` does not touch your arrays, but you should always create plans first just to be sure.)

Once the plan has been created, you can use it as many times as you like for transforms on the specified `in`/`out` arrays, computing the actual transforms via `fftw_execute(plan)`:

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

The DFT results are stored in-order in the array `out`, with the zero-frequency (DC) component in `out[0]`. If `in != out`, the transform is *out-of-place* and the input array `in` is not modified. Otherwise, the input array is overwritten with the transform.

If you want to transform a *different* array of the same size, you can create a new plan with `fftw_plan_dft_1d` and FFTW automatically reuses the information from the previous plan, if possible. Alternatively, with the "guru" interface you can apply a given plan to a different array, if you are careful. See Chapter 4 [FFTW Reference], page 21.

When you are done with the plan, you deallocate it by calling `fftw_destroy_plan(plan)`:

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

If you allocate an array with `fftw_malloc()` you must deallocate it with `fftw_free()`. Do not use `free()` or, heaven forbid, `delete`.

FFTW computes an *unnormalized* DFT. Thus, computing a forward followed by a backward transform (or vice versa) results in the original array scaled by `n`. For the definition of the DFT, see Section 4.8 [What FFTW Really Computes], page 42.

If you have a C compiler, such as `gcc`, that supports the C99 standard, and you `#include <complex.h>` *before* `<fftw3.h>`, then `fftw_complex` is the native double-precision complex type and you can manipulate it with ordinary arithmetic. Otherwise, FFTW defines its own complex type, which is bit-compatible with the C99 complex type. See Section 4.1.1 [Complex numbers], page 21. (The C++ `<complex>` template class may also be usable via a typecast.)

To use single or long-double precision versions of FFTW, replace the `fftw_` prefix by `fftwf_` or `fftwl_` and link with `-lfftw3f` or `-lfftw3l`, but use the *same* `<fftw3.h>` header file.

Many more flags exist besides `FFTW_MEASURE` and `FFTW_ESTIMATE`. For example, use `FFTW_PATIENT` if you're willing to wait even longer for a possibly even faster plan (see Chapter 4 [FFTW Reference], page 21). You can also save plans for future use, as described by Section 3.3 [Words of Wisdom-Saving Plans], page 18.

2.2 Complex Multi-Dimensional DFTs

Multi-dimensional transforms work much the same way as one-dimensional transforms: you allocate arrays of `fftw_complex` (preferably using `fftw_malloc`), create an `fftw_plan`, execute it as many times as you want with `fftw_execute(plan)`, and clean up with `fftw_destroy_plan(plan)` (and `fftw_free`).

FFTW provides two routines for creating plans for 2d and 3d transforms, and one routine for creating plans of arbitrary dimensionality. The 2d and 3d routines have the following signature:

```
fftw_plan fftw_plan_dft_2d(int n0, int n1,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_3d(int n0, int n1, int n2,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
```

These routines create plans for `n0` by `n1` two-dimensional (2d) transforms and `n0` by `n1` by `n2` 3d transforms, respectively. All of these transforms operate on contiguous arrays in the C-standard *row-major* order, so that the last dimension has the fastest-varying index in the array. This layout is described further in Section 3.2 [Multi-dimensional Array Format], page 15.

FFTW can also compute transforms of higher dimensionality. In order to avoid confusion between the various meanings of the the word "dimension", we use the term *rank* to denote the number of independent indices in an array.[2] For example, we say that a 2d transform has rank 2, a 3d transform has rank 3, and so on. You can plan transforms of arbitrary rank by means of the following function:

```
fftw_plan fftw_plan_dft(int rank, const int *n,
                        fftw_complex *in, fftw_complex *out,
                        int sign, unsigned flags);
```

[2] The term "rank" is commonly used in the APL, FORTRAN, and Common Lisp traditions, although it is not so common in the C world.

Here, `n` is a pointer to an array `n[rank]` denoting an `n[0]` by `n[1]` by ... by `n[rank-1]` transform. Thus, for example, the call

```
fftw_plan_dft_2d(n0, n1, in, out, sign, flags);
```

is equivalent to the following code fragment:

```
int n[2];
n[0] = n0;
n[1] = n1;
fftw_plan_dft(2, n, in, out, sign, flags);
```

`fftw_plan_dft` is not restricted to 2d and 3d transforms, however, but it can plan transforms of arbitrary rank.

You may have noticed that all the planner routines described so far have overlapping functionality. For example, you can plan a 1d or 2d transform by using `fftw_plan_dft` with a `rank` of `1` or `2`, or even by calling `fftw_plan_dft_3d` with `n0` and/or `n1` equal to `1` (with no loss in efficiency). This pattern continues, and FFTW's planning routines in general form a "partial order," sequences of interfaces with strictly increasing generality but correspondingly greater complexity.

`fftw_plan_dft` is the most general complex-DFT routine that we describe in this tutorial, but there are also the advanced and guru interfaces, which allow one to efficiently combine multiple/strided transforms into a single FFTW plan, transform a subset of a larger multi-dimensional array, and/or to handle more general complex-number formats. For more information, see Chapter 4 [FFTW Reference], page 21.

2.3 One-Dimensional DFTs of Real Data

In many practical applications, the input data `in[i]` are purely real numbers, in which case the DFT output satisfies the "Hermitian" redundancy: `out[i]` is the conjugate of `out[n-i]`. It is possible to take advantage of these circumstances in order to achieve roughly a factor of two improvement in both speed and memory usage.

In exchange for these speed and space advantages, the user sacrifices some of the simplicity of FFTW's complex transforms. First of all, the input and output arrays are of *different sizes and types*: the input is `n` real numbers, while the output is `n/2+1` complex numbers (the non-redundant outputs); this also requires slight "padding" of the input array for in-place transforms. Second, the inverse transform (complex to real) has the side-effect of *overwriting its input array*, by default. Neither of these inconveniences should pose a serious problem for users, but it is important to be aware of them.

The routines to perform real-data transforms are almost the same as those for complex transforms: you allocate arrays of `double` and/or `fftw_complex` (preferably using `fftw_malloc` or `fftw_alloc_complex`), create an `fftw_plan`, execute it as many times as you want with `fftw_execute(plan)`, and clean up with `fftw_destroy_plan(plan)` (and `fftw_free`). The only differences are that the input (or output) is of type `double` and there are new routines to create the plan. In one dimension:

```
fftw_plan fftw_plan_dft_r2c_1d(int n, double *in, fftw_complex *out,
                               unsigned flags);
```

```
fftw_plan fftw_plan_dft_c2r_1d(int n, fftw_complex *in, double *out,
                               unsigned flags);
```

for the real input to complex-Hermitian output (*r2c*) and complex-Hermitian input to real output (*c2r*) transforms. Unlike the complex DFT planner, there is no `sign` argument. Instead, r2c DFTs are always `FFTW_FORWARD` and c2r DFTs are always `FFTW_BACKWARD`. (For single/long-double precision `fftwf` and `fftwl`, `double` should be replaced by `float` and `long double`, respectively.)

Here, `n` is the "logical" size of the DFT, not necessarily the physical size of the array. In particular, the real (`double`) array has `n` elements, while the complex (`fftw_complex`) array has `n/2+1` elements (where the division is rounded down). For an in-place transform, `in` and `out` are aliased to the same array, which must be big enough to hold both; so, the real array would actually have `2*(n/2+1)` elements, where the elements beyond the first `n` are unused padding. (Note that this is very different from the concept of "zero-padding" a transform to a larger length, which changes the logical size of the DFT by actually adding new input data.) The kth element of the complex array is exactly the same as the kth element of the corresponding complex DFT. All positive `n` are supported; products of small factors are most efficient, but an $O(n \log n)$ algorithm is used even for prime sizes.

As noted above, the c2r transform destroys its input array even for out-of-place transforms. This can be prevented, if necessary, by including `FFTW_PRESERVE_INPUT` in the `flags`, with unfortunately some sacrifice in performance. This flag is also not currently supported for multi-dimensional real DFTs (next section).

Readers familiar with DFTs of real data will recall that the 0th (the "DC") and `n/2`-th (the "Nyquist" frequency, when `n` is even) elements of the complex output are purely real. Some implementations therefore store the Nyquist element where the DC imaginary part would go, in order to make the input and output arrays the same size. Such packing, however, does not generalize well to multi-dimensional transforms, and the space savings are miniscule in any case; FFTW does not support it.

An alternative interface for one-dimensional r2c and c2r DFTs can be found in the '`r2r`' interface (see Section 2.5.1 [The Halfcomplex-format DFT], page 11), with "halfcomplex"-format output that *is* the same size (and type) as the input array. That interface, although it is not very useful for multi-dimensional transforms, may sometimes yield better performance.

2.4 Multi-Dimensional DFTs of Real Data

Multi-dimensional DFTs of real data use the following planner routines:

```
fftw_plan fftw_plan_dft_r2c_2d(int n0, int n1,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_3d(int n0, int n1, int n2,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c(int rank, const int *n,
                            double *in, fftw_complex *out,
                            unsigned flags);
```

as well as the corresponding `c2r` routines with the input/output types swapped. These routines work similarly to their complex analogues, except for the fact that here the complex output array is cut roughly in half and the real array requires padding for in-place transforms (as in 1d, above).

As before, `n` is the logical size of the array, and the consequences of this on the the format of the complex arrays deserve careful attention. Suppose that the real data has dimensions $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ (in row-major order). Then, after an r2c transform, the output is an $n_0 \times n_1 \times n_2 \times \cdots \times (n_{d-1}/2 + 1)$ array of `fftw_complex` values in row-major order, corresponding to slightly over half of the output of the corresponding complex DFT. (The division is rounded down.) The ordering of the data is otherwise exactly the same as in the complex-DFT case.

For out-of-place transforms, this is the end of the story: the real data is stored as a row-major array of size $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ and the complex data is stored as a row-major array of size $n_0 \times n_1 \times n_2 \times \cdots \times (n_{d-1}/2 + 1)$.

For in-place transforms, however, extra padding of the real-data array is necessary because the complex array is larger than the real array, and the two arrays share the same memory locations. Thus, for in-place transforms, the final dimension of the real-data array must be padded with extra values to accommodate the size of the complex data—two values if the last dimension is even and one if it is odd. That is, the last dimension of the real data must physically contain $2(n_{d-1}/2 + 1)$ `double` values (exactly enough to hold the complex data). This physical array size does not, however, change the *logical* array size—only n_{d-1} values are actually stored in the last dimension, and n_{d-1} is the last dimension passed to the plan-creation routine.

For example, consider the transform of a two-dimensional real array of size `n0` by `n1`. The output of the r2c transform is a two-dimensional complex array of size `n0` by `n1/2+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 `double`, 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 `n0` by `2*(n1/2+1)`. If `n1` 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).

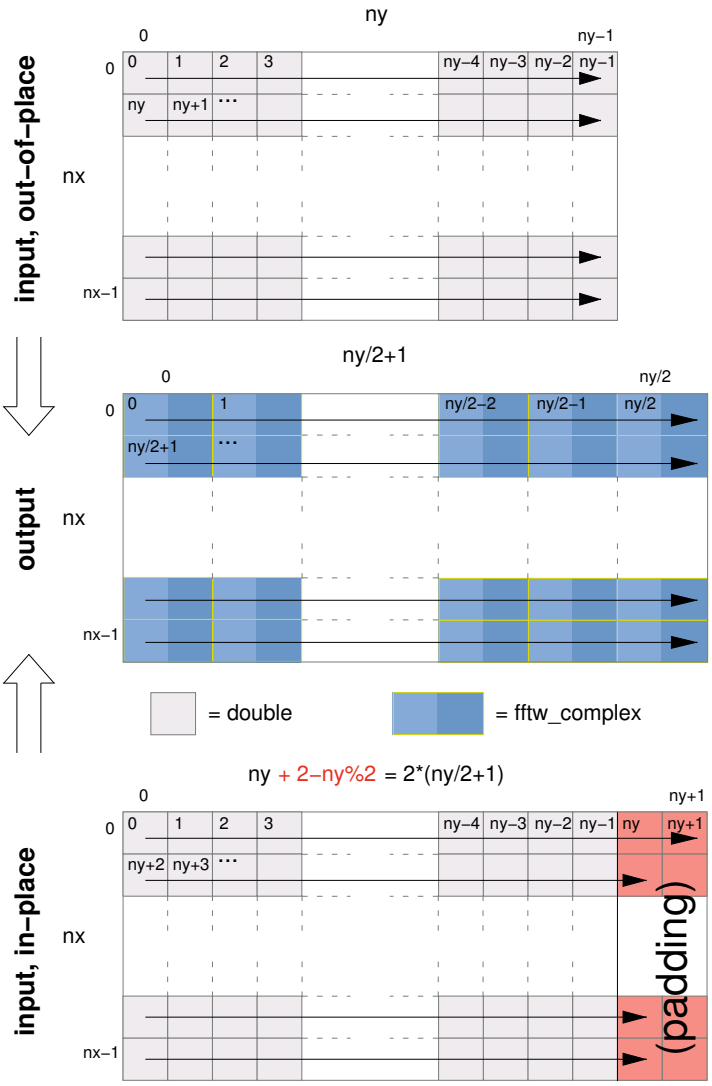

Figure 2.1: Illustration of the data layout for a 2d `nx` by `ny` real-to-complex transform.

Figure 2.1 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):

These transforms are unnormalized, so an r2c followed by a c2r transform (or vice versa) 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.

(Because the last dimension is treated specially, if it is equal to `1` the transform is *not* equivalent to a lower-dimensional r2c/c2r transform. In that case, the last complex dimension also has size `1` (`=1/2+1`), and no advantage is gained over the complex transforms.)

2.5 More DFTs of Real Data

FFTW supports several other transform types via a unified *r2r* (real-to-real) interface, so called because it takes a real (`double`) array and outputs a real array of the same size. These r2r transforms currently fall into three categories: DFTs of real input and complex-Hermitian output in halfcomplex format, DFTs of real input with even/odd symmetry (a.k.a. discrete cosine/sine transforms, DCTs/DSTs), and discrete Hartley transforms (DHTs), all described in more detail by the following sections.

The r2r transforms follow the by now familiar interface of creating an `fftw_plan`, executing it with `fftw_execute(plan)`, and destroying it with `fftw_destroy_plan(plan)`. Furthermore, all r2r transforms share the same planner interface:

```
fftw_plan fftw_plan_r2r_1d(int n, double *in, double *out,
                           fftw_r2r_kind kind, unsigned flags);
fftw_plan fftw_plan_r2r_2d(int n0, int n1, double *in, double *out,
                           fftw_r2r_kind kind0, fftw_r2r_kind kind1,
                           unsigned flags);
fftw_plan fftw_plan_r2r_3d(int n0, int n1, int n2,
                           double *in, double *out,
                           fftw_r2r_kind kind0,
                           fftw_r2r_kind kind1,
                           fftw_r2r_kind kind2,
                           unsigned flags);
fftw_plan fftw_plan_r2r(int rank, const int *n, double *in, double *out,
                        const fftw_r2r_kind *kind, unsigned flags);
```

Just as for the complex DFT, these plan 1d/2d/3d/multi-dimensional transforms for contiguous arrays in row-major order, transforming (real) input to output of the same size, where `n` specifies the *physical* dimensions of the arrays. All positive `n` are supported (with the exception of `n=1` for the `FFTW_REDFT00` kind, noted in the real-even subsection below); products of small factors are most efficient (factorizing `n-1` and `n+1` for `FFTW_REDFT00` and `FFTW_RODFT00` kinds, described below), but an $O(n \log n)$ algorithm is used even for prime sizes.

Each dimension has a *kind* parameter, of type `fftw_r2r_kind`, specifying the kind of r2r transform to be used for that dimension. (In the case of `fftw_plan_r2r`, this is an array `kind[rank]` where `kind[i]` is the transform kind for the dimension `n[i]`.) The kind can be one of a set of predefined constants, defined in the following subsections.

In other words, FFTW computes the separable product of the specified r2r transforms over each dimension, which can be used e.g. for partial differential equations with mixed boundary conditions. (For some r2r kinds, notably the halfcomplex DFT and the DHT, such a separable product is somewhat problematic in more than one dimension, however, as is described below.)

In the current version of FFTW, all r2r transforms except for the halfcomplex type are computed via pre- or post-processing of halfcomplex transforms, and they are therefore not as fast as they could be. Since most other general DCT/DST codes employ a similar algorithm, however, FFTW's implementation should provide at least competitive performance.

2.5.1 The Halfcomplex-format DFT

An r2r kind of `FFTW_R2HC` (*r2hc*) corresponds to an r2c DFT (see Section 2.3 [One-Dimensional DFTs of Real Data], page 6) but with "halfcomplex" format output, and may sometimes be faster and/or more convenient than the latter. The inverse *hc2r* transform is of kind `FFTW_HC2R`. This consists of the non-redundant half of the complex output for a 1d real-input DFT of size `n`, stored as a sequence of `n` real numbers (`double`) in the 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 kth output, and i_k is the imaginary part. (Division by 2 is rounded down.) For a halfcomplex array `hc[n]`, the kth component thus 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-input DFT, and is not stored. Thus, the r2hc transform of `n` real values is a halfcomplex array of length `n`, and vice versa for hc2r.

Aside from the differing format, the output of `FFTW_R2HC`/`FFTW_HC2R` is otherwise exactly the same as for the corresponding 1d r2c/c2r transform (i.e. `FFTW_FORWARD`/`FFTW_BACKWARD` transforms, respectively). Recall that these transforms are unnormalized, so r2hc followed by hc2r will result in the original data multiplied by `n`. Furthermore, like the c2r transform, an out-of-place hc2r transform will *destroy its input* array.

Although these halfcomplex transforms can be used with the multi-dimensional r2r interface, the interpretation of such a separable product of transforms along each dimension is problematic. For example, consider a two-dimensional `n0` by `n1`, r2hc by r2hc transform planned by `fftw_plan_r2r_2d(n0, n1, in, out, FFTW_R2HC, FFTW_R2HC, FFTW_MEASURE)`. Conceptually, FFTW first transforms the rows (of size `n1`) to produce halfcomplex rows, and then transforms the columns (of size `n0`). Half of these column transforms, however, are of imaginary parts, and should therefore be multiplied by i and combined with the r2hc transforms of the real columns to produce the 2d DFT amplitudes; FFTW's r2r transform does *not* perform this combination for you. Thus, if a multi-dimensional real-input/output DFT is required, we recommend using the ordinary r2c/c2r interface (see Section 2.4 [Multi-Dimensional DFTs of Real Data], page 7).

2.5.2 Real even/odd DFTs (cosine/sine transforms)

The Fourier transform of a real-even function $f(-x) = f(x)$ is real-even, and i times the Fourier transform of a real-odd function $f(-x) = -f(x)$ is real-odd. Similar results hold for a discrete Fourier transform, and thus for these symmetries the need for complex inputs/outputs is entirely eliminated. Moreover, one gains a factor of two in speed/space from the fact that the data are real, and an additional factor of two from the even/odd symmetry: only the non-redundant (first) half of the array need be stored. The result is the real-even DFT (*REDFT*) and the real-odd DFT (*RODFT*), also known as the discrete cosine and sine transforms (*DCT* and *DST*), respectively.

(In this section, we describe the 1d transforms; multi-dimensional transforms are just a separable product of these transforms operating along each dimension.)

Because of the discrete sampling, one has an additional choice: is the data even/odd around a sampling point, or around the point halfway between two samples? The latter corresponds to *shifting* the samples by *half* an interval, and gives rise to several transform variants denoted by REDFTab and RODFTab: a and b are 0 or 1, and indicate whether the input (a) and/or output (b) are shifted by half a sample (1 means it is shifted). These are also known as types I-IV of the DCT and DST, and all four types are supported by FFTW's r2r interface.[3]

The r2r kinds for the various REDFT and RODFT types supported by FFTW, along with the boundary conditions at both ends of the *input* array (`n` real numbers `in[j=0..n-1]`), are:

- `FFTW_REDFT00` (DCT-I): even around $j = 0$ and even around $j = n - 1$.
- `FFTW_REDFT10` (DCT-II, "the" DCT): even around $j = -0.5$ and even around $j = n - 0.5$.
- `FFTW_REDFT01` (DCT-III, "the" IDCT): even around $j = 0$ and odd around $j = n$.
- `FFTW_REDFT11` (DCT-IV): even around $j = -0.5$ and odd around $j = n - 0.5$.
- `FFTW_RODFT00` (DST-I): odd around $j = -1$ and odd around $j = n$.
- `FFTW_RODFT10` (DST-II): odd around $j = -0.5$ and odd around $j = n - 0.5$.
- `FFTW_RODFT01` (DST-III): odd around $j = -1$ and even around $j = n - 1$.
- `FFTW_RODFT11` (DST-IV): odd around $j = -0.5$ and even around $j = n - 0.5$.

Note that these symmetries apply to the "logical" array being transformed; **there are no constraints on your physical input data**. So, for example, if you specify a size-5 REDFT00 (DCT-I) of the data *abcde*, it corresponds to the DFT of the logical even array *abcdedcb* of size 8. A size-4 REDFT10 (DCT-II) of the data *abcd* corresponds to the size-8 logical DFT of the even array *abcddcba*, shifted by half a sample.

All of these transforms are invertible. The inverse of R*DFT00 is R*DFT00; of R*DFT10 is R*DFT01 and vice versa (these are often called simply "the" DCT and IDCT, respectively); and of R*DFT11 is R*DFT11. However, the transforms computed by FFTW are unnormalized, exactly like the corresponding real and complex DFTs, so computing a transform followed by its inverse yields the original array scaled by N, where N is the *logical* DFT size. For REDFT00, $N = 2(n - 1)$; for RODFT00, $N = 2(n + 1)$; otherwise, $N = 2n$.

Note that the boundary conditions of the transform output array are given by the input boundary conditions of the inverse transform. Thus, the above transforms are all inequivalent in terms of input/output boundary conditions, even neglecting the 0.5 shift difference.

FFTW is most efficient when N is a product of small factors; note that this *differs* from the factorization of the physical size `n` for REDFT00 and RODFT00! There is another oddity: `n=1` REDFT00 transforms correspond to $N = 0$, and so are *not defined* (the planner will return `NULL`). Otherwise, any positive `n` is supported.

For the precise mathematical definitions of these transforms as used by FFTW, see Section 4.8 [What FFTW Really Computes], page 42. (For people accustomed to the

[3] There are also type V-VIII transforms, which correspond to a logical DFT of *odd* size N, independent of whether the physical size `n` is odd, but we do not support these variants.

DCT/DST, FFTW's definitions have a coefficient of 2 in front of the cos/sin functions so that they correspond precisely to an even/odd DFT of size N. Some authors also include additional multiplicative factors of $\sqrt{2}$ for selected inputs and outputs; this makes the transform orthogonal, but sacrifices the direct equivalence to a symmetric DFT.)

Which type do you need?

Since the required flavor of even/odd DFT depends upon your problem, you are the best judge of this choice, but we can make a few comments on relative efficiency to help you in your selection. In particular, R*DFT01 and R*DFT10 tend to be slightly faster than R*DFT11 (especially for odd sizes), while the R*DFT00 transforms are sometimes significantly slower (especially for even sizes).[4]

Thus, if only the boundary conditions on the transform inputs are specified, we generally recommend R*DFT10 over R*DFT00 and R*DFT01 over R*DFT11 (unless the half-sample shift or the self-inverse property is significant for your problem).

If performance is important to you and you are using only small sizes (say $n < 200$), e.g. for multi-dimensional transforms, then you might consider generating hard-coded transforms of those sizes and types that you are interested in (see Section 10.4 [Generating your own code], page 101).

We are interested in hearing what types of symmetric transforms you find most useful.

2.5.3 The Discrete Hartley Transform

If you are planning to use the DHT because you've heard that it is "faster" than the DFT (FFT), **stop here**. The DHT is not faster than the DFT. That story is an old but enduring misconception that was debunked in 1987.

The discrete Hartley transform (DHT) is an invertible linear transform closely related to the DFT. In the DFT, one multiplies each input by $cos - i * sin$ (a complex exponential), whereas in the DHT each input is multiplied by simply $cos + sin$. Thus, the DHT transforms `n` real numbers to `n` real numbers, and has the convenient property of being its own inverse. In FFTW, a DHT (of any positive `n`) can be specified by an r2r kind of `FFTW_DHT`.

Like the DFT, in FFTW the DHT is unnormalized, so computing a DHT of size `n` followed by another DHT of the same size will result in the original array multiplied by `n`.

The DHT was originally proposed as a more efficient alternative to the DFT for real data, but it was subsequently shown that a specialized DFT (such as FFTW's r2hc or r2c transforms) could be just as fast. In FFTW, the DHT is actually computed by post-processing an r2hc transform, so there is ordinarily no reason to prefer it from a performance per-

[4] R*DFT00 is sometimes slower in FFTW because we discovered that the standard algorithm for computing this by a pre/post-processed real DFT—the algorithm used in FFTPACK, Numerical Recipes, and other sources for decades now—has serious numerical problems: it already loses several decimal places of accuracy for 16k sizes. There seem to be only two alternatives in the literature that do not suffer similarly: a recursive decomposition into smaller DCTs, which would require a large set of codelets for efficiency and generality, or sacrificing a factor of ~ 2 in speed to use a real DFT of twice the size. We currently employ the latter technique for general n, as well as a limited form of the former method: a split-radix decomposition when n is odd (N a multiple of 4). For N containing many factors of 2, the split-radix method seems to recover most of the speed of the standard algorithm without the accuracy tradeoff.

spective.[5] However, we have heard rumors that the DHT might be the most appropriate transform in its own right for certain applications, and we would be very interested to hear from anyone who finds it useful.

If `FFTW_DHT` is specified for multiple dimensions of a multi-dimensional transform, FFTW computes the separable product of 1d DHTs along each dimension. Unfortunately, this is not quite the same thing as a true multi-dimensional DHT; you can compute the latter, if necessary, with at most `rank-1` post-processing passes [see e.g. H. Hao and R. N. Bracewell, *Proc. IEEE* **75**, 264–266 (1987)].

For the precise mathematical definition of the DHT as used by FFTW, see Section 4.8 [What FFTW Really Computes], page 42.

[5] We provide the DHT mainly as a byproduct of some internal algorithms. FFTW computes a real input/output DFT of *prime* size by re-expressing it as a DHT plus post/pre-processing and then using Rader's prime-DFT algorithm adapted to the DHT.

3 Other Important Topics

3.1 SIMD alignment and fftw_malloc

SIMD, which stands for "Single Instruction Multiple Data," is a set of special operations supported by some processors to perform a single operation on several numbers (usually 2 or 4) simultaneously. SIMD floating-point instructions are available on several popular CPUs: SSE/SSE2/AVX on recent x86/x86-64 processors, AltiVec (single precision) on some PowerPCs (Apple G4 and higher), NEON on some ARM models, and MIPS Paired Single (currently only in FFTW 3.2.x). FFTW can be compiled to support the SIMD instructions on any of these systems.

A program linking to an FFTW library compiled with SIMD support can obtain a nonnegligible speedup for most complex and r2c/c2r transforms. In order to obtain this speedup, however, the arrays of complex (or real) data passed to FFTW must be specially aligned in memory (typically 16-byte aligned), and often this alignment is more stringent than that provided by the usual `malloc` (etc.) allocation routines.

In order to guarantee proper alignment for SIMD, therefore, in case your program is ever linked against a SIMD-using FFTW, we recommend allocating your transform data with `fftw_malloc` and de-allocating it with `fftw_free`. These have exactly the same interface and behavior as `malloc`/`free`, except that for a SIMD FFTW they ensure that the returned pointer has the necessary alignment (by calling `memalign` or its equivalent on your OS).

You are not *required* to use `fftw_malloc`. You can allocate your data in any way that you like, from `malloc` to `new` (in C++) to a fixed-size array declaration. If the array happens not to be properly aligned, FFTW will not use the SIMD extensions.

Since `fftw_malloc` only ever needs to be used for real and complex arrays, we provide two convenient wrapper routines `fftw_alloc_real(N)` and `fftw_alloc_complex(N)` that are equivalent to `(double*)fftw_malloc(sizeof(double) * N)` and `(fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N)`, respectively (or their equivalents in other precisions).

3.2 Multi-dimensional Array Format

This section describes the format in which multi-dimensional arrays are stored in FFTW. We felt that a detailed discussion of this topic was necessary. Since several different formats are common, this topic is often a source of confusion.

3.2.1 Row-major Format

The multi-dimensional arrays passed to `fftw_plan_dft` etcetera 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 $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$. Now, we specify a location in the array by a sequence of d (zero-based) indices, one for each dimension: $(i_0, i_1, i_2, \ldots, i_{d-1})$. If the array is stored in row-major order, then this element is located at the position $i_{d-1} + n_{d-1}(i_{d-2} + n_{d-2}(\ldots + n_1 i_0))$.

Note that, for the ordinary complex DFT, each element of the array must be of type `fftw_complex`; i.e. a (real, imaginary) pair of (double-precision) numbers.

In the advanced FFTW interface, the physical dimensions n from which the indices are computed can be different from (larger than) the logical dimensions of the transform to be computed, in order to transform a subset of a larger array. Note also that, in the advanced interface, the expression above is multiplied by a *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 the basic interface, however, the stride is 1.

3.2.2 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 FFTW, it is quite easy to do. When creating the plan, simply pass the dimensions of the array to the planner 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 FFTW). This is done for you *automatically* by the FFTW legacy-Fortran interface (see Chapter 8 [Calling FFTW from Legacy Fortran], page 87), but you must do it manually with the modern Fortran interface (see Section 7.2 [Reversing array dimensions], page 78).

3.2.3 Fixed-size Arrays in C

A multi-dimensional array whose size is declared at compile time in C is *already* in row-major order. You don't have to do anything special to transform it. For example:

```
{
     fftw_complex data[N0][N1][N2];
     fftw_plan plan;
     ...
     plan = fftw_plan_dft_3d(N0, N1, N2, &data[0][0][0], &data[0][0][0],
                             FFTW_FORWARD, FFTW_ESTIMATE);
     ...
}
```

This will plan a 3d in-place transform of size `N0 x N1 x N2`. Notice how we took the address of the zero-th element to pass to the planner (we could also have used a typecast).

However, we tend to *discourage* users from declaring their arrays in this way, for two reasons. First, this allocates the array on the stack ("automatic" storage), which has a very limited size on most operating systems (declaring an array with more than a few thousand elements will often cause a crash). (You can get around this limitation on many systems by declaring the array as `static` and/or global, but that has its own drawbacks.) Second, it may not optimally align the array for use with a SIMD FFTW (see Section 3.1 [SIMD alignment and fftw_malloc], page 15). Instead, we recommend using `fftw_malloc`, as described below.

3.2.4 Dynamic Arrays in C

We recommend allocating most arrays dynamically, with `fftw_malloc`. 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 `fftw_malloc`, allocate an array big enough to store N `fftw_complex` values (for a complex DFT), 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 $5 \times 12 \times 27$ rank-3 array:

```
fftw_complex *an_array;
an_array = (fftw_complex*) fftw_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 fixed-size 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. The recent C99 standard provides a way to reinterpret the dynamic array as a "variable-length" multi-dimensional array amenable to '`[]`', but this feature is not yet widely supported by compilers.

3.2.5 Dynamic Arrays in C—The Wrong Way

A different method for allocating multi-dimensional arrays in C is often suggested that is incompatible with FFTW: *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.

3.3 Words of Wisdom—Saving Plans

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 4 [FFTW Reference], page 21, for a less casual but more complete discussion of how to use wisdom in FFTW.

Plans created with the `FFTW_MEASURE`, `FFTW_PATIENT`, or `FFTW_EXHAUSTIVE` options produce near-optimal FFT performance, but may require a long time to compute because FFTW must measure the runtime of many possible plans and select the best one. This setup 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 slower transforms, we have provided `FFTW_ESTIMATE`. The `wisdom` mechanism is a way to get the best of both worlds: you compute a good plan once, save it to disk, and later reload it as many times as necessary. The wisdom mechanism can actually save and reload many plans at once, not just one.

Whenever you create a plan, the FFTW planner accumulates wisdom, which is information sufficient to reconstruct the plan. After planning, you can save this information to disk by means of the function:

```
int fftw_export_wisdom_to_filename(const char *filename);
```

(This function returns non-zero on success.)

The next time you run the program, you can restore the wisdom with `fftw_import_wisdom_from_filename` (which also returns non-zero on success), and then recreate the plan using the same flags as before.

```
int fftw_import_wisdom_from_filename(const char *filename);
```

Wisdom is automatically used for any size to which it is applicable, as long as the planner flags are not more "patient" than those with which the wisdom was created. For example, wisdom created with `FFTW_MEASURE` can be used if you later plan with `FFTW_ESTIMATE` or `FFTW_MEASURE`, but not with `FFTW_PATIENT`.

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. If memory usage is a concern, however, the wisdom can be forgotten and its associated memory freed by calling:

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

Wisdom can be exported to a file, a string, or any other medium. For details, see Section 4.7 [Wisdom], page 40.

3.4 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.

Nevertheless, if the choice is between using `FFTW_ESTIMATE` or using possibly-suboptimal wisdom (created on the same machine, but for a different binary), the wisdom is likely to be better. For this reason, we provide a function to import wisdom from a standard system-wide location (`/etc/fftw/wisdom` on Unix):

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

FFTW also provides a standalone program, `fftw-wisdom` (described by its own `man` page on Unix) with which users can create wisdom, e.g. for a canonical set of sizes to store in the system wisdom file. See Section 4.7.4 [Wisdom Utilities], page 41.

4 FFTW Reference

This chapter provides a complete reference for all sequential (i.e., one-processor) FFTW functions. Parallel transforms are described in later chapters.

4.1 Data Types and Files

All programs using FFTW should include its header file:

```
#include <fftw3.h>
```

You must also link to the FFTW library. On Unix, this means adding `-lfftw3 -lm` at the *end* of the link command.

4.1.1 Complex numbers

The default FFTW interface uses `double` precision for all floating-point numbers, and defines a `fftw_complex` type to hold complex numbers as:

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

Here, the `[0]` element holds the real part and the `[1]` element holds the imaginary part.

Alternatively, if you have a C compiler (such as `gcc`) that supports the C99 revision of the ANSI C standard, you can use C's new native complex type (which is binary-compatible with the typedef above). In particular, if you `#include <complex.h>` *before* `<fftw3.h>`, then `fftw_complex` is defined to be the native complex type and you can manipulate it with ordinary arithmetic (e.g. `x = y * (3+4*I)`, where `x` and `y` are `fftw_complex` and `I` is the standard symbol for the imaginary unit);

C++ has its own `complex<T>` template class, defined in the standard `<complex>` header file. Reportedly, the C++ standards committee has recently agreed to mandate that the storage format used for this type be binary-compatible with the C99 type, i.e. an array `T[2]` with consecutive real `[0]` and imaginary `[1]` parts. (See report `http://www.open-std.org/jtc1/sc22/WG21/docs/papers/2002/n1388.pdf WG21/N1388`.) Although not part of the official standard as of this writing, the proposal stated that: "This solution has been tested with all current major implementations of the standard library and shown to be working." To the extent that this is true, if you have a variable `complex<double> *x`, you can pass it directly to FFTW via `reinterpret_cast<fftw_complex*>(x)`.

4.1.2 Precision

You can install single and long-double precision versions of FFTW, which replace `double` with `float` and `long double`, respectively (see Chapter 10 [Installation and Customization], page 97). To use these interfaces, you:

- Link to the single/long-double libraries; on Unix, `-lfftw3f` or `-lfftw3l` instead of (or in addition to) `-lfftw3`. (You can link to the different-precision libraries simultaneously.)
- Include the *same* `<fftw3.h>` header file.
- Replace all lowercase instances of '`fftw_`' with '`fftwf_`' or '`fftwl_`' for single or long-double precision, respectively. (`fftw_complex` becomes `fftwf_complex`, `fftw_execute` becomes `fftwf_execute`, etcetera.)

- Uppercase names, i.e. names beginning with '`FFTW_`', remain the same.
- Replace `double` with `float` or `long double` for subroutine parameters.

Depending upon your compiler and/or hardware, `long double` may not be any more precise than `double` (or may not be supported at all, although it is standard in C99).

We also support using the nonstandard `__float128` quadruple-precision type provided by recent versions of `gcc` on 32- and 64-bit x86 hardware (see Chapter 10 [Installation and Customization], page 97). To use this type, link with `-lfftw3q -lquadmath -lm` (the `libquadmath` library provided by `gcc` is needed for quadruple-precision trigonometric functions) and use '`fftwq_`' identifiers.

4.1.3 Memory Allocation

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

These are functions that behave identically to `malloc` and `free`, except that they guarantee that the returned pointer obeys any special alignment restrictions imposed by any algorithm in FFTW (e.g. for SIMD acceleration). See Section 3.1 [SIMD alignment and fftw_malloc], page 15.

Data allocated by `fftw_malloc` *must* be deallocated by `fftw_free` and not by the ordinary `free`.

These routines simply call through to your operating system's `malloc` or, if necessary, its aligned equivalent (e.g. `memalign`), so you normally need not worry about any significant time or space overhead. You are *not required* to use them to allocate your data, but we strongly recommend it.

Note: in C++, just as with ordinary `malloc`, you must typecast the output of `fftw_malloc` to whatever pointer type you are allocating.

We also provide the following two convenience functions to allocate real and complex arrays with `n` elements, which are equivalent to `(double *) fftw_malloc(sizeof(double) * n)` and `(fftw_complex *) fftw_malloc(sizeof(fftw_complex) * n)`, respectively:

```
double *fftw_alloc_real(size_t n);
fftw_complex *fftw_alloc_complex(size_t n);
```

The equivalent functions in other precisions allocate arrays of `n` elements in that precision. e.g. `fftwf_alloc_real(n)` is equivalent to `(float *) fftwf_malloc(sizeof(float) * n)`.

4.2 Using Plans

Plans for all transform types in FFTW are stored as type `fftw_plan` (an opaque pointer type), and are created by one of the various planning routines described in the following sections. An `fftw_plan` contains all information necessary to compute the transform, including the pointers to the input and output arrays.

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

This executes the `plan`, to compute the corresponding transform on the arrays for which it was planned (which must still exist). The plan is not modified, and `fftw_execute` can be called as many times as desired.

To apply a given plan to a different array, you can use the new-array execute interface. See Section 4.6 [New-array Execute Functions], page 38.

`fftw_execute` (and equivalents) is the only function in FFTW guaranteed to be thread-safe; see Section 5.4 [Thread safety], page 51.

This function:

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

deallocates the `plan` and all its associated data.

FFTW's planner saves some other persistent data, such as the accumulated wisdom and a list of algorithms available in the current configuration. If you want to deallocate all of that and reset FFTW to the pristine state it was in when you started your program, you can call:

```
void fftw_cleanup(void);
```

After calling `fftw_cleanup`, all existing plans become undefined, and you should not attempt to execute them nor to destroy them. You can however create and execute/destroy new plans, in which case FFTW starts accumulating wisdom information again.

`fftw_cleanup` does not deallocate your plans, however. To prevent memory leaks, you must still call `fftw_destroy_plan` before executing `fftw_cleanup`.

Occasionally, it may useful to know FFTW's internal "cost" metric that it uses to compare plans to one another; this cost is proportional to an execution time of the plan, in undocumented units, if the plan was created with the `FFTW_MEASURE` or other timing-based options, or alternatively is a heuristic cost function for `FFTW_ESTIMATE` plans. (The cost values of measured and estimated plans are not comparable, being in different units. Also, costs from different FFTW versions or the same version compiled differently may not be in the same units. Plans created from wisdom have a cost of 0 since no timing measurement is performed for them. Finally, certain problems for which only one top-level algorithm was possible may have required no measurements of the cost of the whole plan, in which case `fftw_cost` will also return 0.) The cost metric for a given plan is returned by:

```
double fftw_cost(const fftw_plan plan);
```

The following two routines are provided purely for academic purposes (that is, for entertainment).

```
void fftw_flops(const fftw_plan plan,
                double *add, double *mul, double *fma);
```

Given a `plan`, set `add`, `mul`, and `fma` to an exact count of the number of floating-point additions, multiplications, and fused multiply-add operations involved in the plan's execution. The total number of floating-point operations (flops) is `add + mul + 2*fma`, or `add + mul + fma` if the hardware supports fused multiply-add instructions (although the number of FMA operations is only approximate because of compiler voodoo). (The number of operations should be an integer, but we use `double` to avoid overflowing `int` for large transforms; the arguments are of type `double` even for single and long-double precision versions of FFTW.)

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

This outputs a "nerd-readable" representation of the `plan` to the given file, to `stdout`, or two a newly allocated NUL-terminated string (which the caller is responsible for deallocating with `free`), respectively.

4.3 Basic Interface

Recall that the FFTW API is divided into three parts[1]: the *basic interface* computes a single transform of contiguous data, the *advanced interface* computes transforms of multiple or strided arrays, and the *guru interface* supports the most general data layouts, multiplicities, and strides. This section describes the the basic interface, which we expect to satisfy the needs of most users.

4.3.1 Complex DFTs

```
fftw_plan fftw_plan_dft_1d(int n0,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_2d(int n0, int n1,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_3d(int n0, int n1, int n2,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft(int rank, const int *n,
                        fftw_complex *in, fftw_complex *out,
                        int sign, unsigned flags);
```

Plan a complex input/output discrete Fourier transform (DFT) in zero or more dimensions, returning an `fftw_plan` (see Section 4.2 [Using Plans], page 22).

Once you have created a plan for a certain transform type and parameters, then creating another plan of the same type and parameters, but for different arrays, is fast and shares constant data with the first plan (if it still exists).

The planner returns `NULL` if the plan cannot be created. In the standard FFTW distribution, the basic interface is guaranteed to return a non-`NULL` plan. A plan may be `NULL`, however, if you are using a customized FFTW configuration supporting a restricted set of transforms.

Arguments

- `rank` is the rank of the transform (it should be the size of the array `*n`), and can be any non-negative integer. (See Section 2.2 [Complex Multi-Dimensional DFTs], page 5, for the definition of "rank".) The '`_1d`', '`_2d`', and '`_3d`' planners correspond to a `rank` of `1`, `2`, and `3`, respectively. The rank may be zero, which is equivalent to a rank-1 transform of size 1, i.e. a copy of one number from input to output.

[1] *Gallia est omnis divisa in partes tres* (Julius Caesar).

- `n0`, `n1`, `n2`, or `n[0..rank-1]` (as appropriate for each routine) specify the size of the transform dimensions. They can be any positive integer.
  - Multi-dimensional arrays are stored in row-major order with dimensions: `n0` x `n1`; or `n0` x `n1` x `n2`; or `n[0]` x `n[1]` x ... x `n[rank-1]`. See Section 3.2 [Multi-dimensional Array Format], page 15.
  - 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 algorithm (which nevertheless retains $O(n \log n)$ performance even for prime sizes). It is possible to customize FFTW for different array sizes; see Chapter 10 [Installation and Customization], page 97. Transforms whose sizes are powers of 2 are especially fast.
- `in` and `out` point to the input and output arrays of the transform, which may be the same (yielding an in-place transform). These arrays are overwritten during planning, unless `FFTW_ESTIMATE` is used in the flags. (The arrays need not be initialized, but they must be allocated.)

  If `in == out`, the transform is *in-place* and the input array is overwritten. If `in != out`, the two arrays must not overlap (but FFTW does not check for this condition).
- `sign` is the sign of the exponent in the formula that defines the Fourier transform. It can be -1 (= `FFTW_FORWARD`) or $+1$ (= `FFTW_BACKWARD`).
- `flags` is a bitwise OR ('`|`') of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 25.

FFTW computes an unnormalized transform: computing a forward followed by a backward transform (or vice versa) will result in the original data multiplied by the size of the transform (the product of the dimensions). For more information, see Section 4.8 [What FFTW Really Computes], page 42.

4.3.2 Planner Flags

All of the planner routines in FFTW accept an integer `flags` argument, which is a bitwise OR ('`|`') of zero or more of the flag constants defined below. These flags control the rigor (and time) of the planning process, and can also impose (or lift) restrictions on the type of transform algorithm that is employed.

Important: the planner overwrites the input array during planning unless a saved plan (see Section 4.7 [Wisdom], page 40) is available for that problem, so you should initialize your input data after creating the plan. The only exceptions to this are the `FFTW_ESTIMATE` and `FFTW_WISDOM_ONLY` flags, as mentioned below.

In all cases, if wisdom is available for the given problem that was created with equal-or-greater planning rigor, then the more rigorous wisdom is used. For example, in `FFTW_ESTIMATE` mode any available wisdom is used, whereas in `FFTW_PATIENT` mode only wisdom created in patient or exhaustive mode can be used. See Section 3.3 [Words of Wisdom-Saving Plans], page 18.

Planning-rigor flags

- `FFTW_ESTIMATE` specifies that, instead of actual measurements of different algorithms, a simple heuristic is used to pick a (probably sub-optimal) plan quickly. With this flag, the input/output arrays are not overwritten during planning.

- `FFTW_MEASURE` tells FFTW to find an optimized plan by actually *computing* several FFTs and measuring their execution time. Depending on your machine, this can take some time (often a few seconds). `FFTW_MEASURE` is the default planning option.
- `FFTW_PATIENT` is like `FFTW_MEASURE`, but considers a wider range of algorithms and often produces a "more optimal" plan (especially for large transforms), but at the expense of several times longer planning time (especially for large transforms).
- `FFTW_EXHAUSTIVE` is like `FFTW_PATIENT`, but considers an even wider range of algorithms, including many that we think are unlikely to be fast, to produce the most optimal plan but with a substantially increased planning time.
- `FFTW_WISDOM_ONLY` is a special planning mode in which the plan is only created if wisdom is available for the given problem, and otherwise a `NULL` plan is returned. This can be combined with other flags, e.g. '`FFTW_WISDOM_ONLY | FFTW_PATIENT`' creates a plan only if wisdom is available that was created in `FFTW_PATIENT` or `FFTW_EXHAUSTIVE` mode. The `FFTW_WISDOM_ONLY` flag is intended for users who need to detect whether wisdom is available; for example, if wisdom is not available one may wish to allocate new arrays for planning so that user data is not overwritten.

Algorithm-restriction flags

- `FFTW_DESTROY_INPUT` specifies that an out-of-place transform is allowed to *overwrite its input* array with arbitrary data; this can sometimes allow more efficient algorithms to be employed.
- `FFTW_PRESERVE_INPUT` specifies that an out-of-place transform must *not change its input* array. This is ordinarily the *default*, except for c2r and hc2r (i.e. complex-to-real) transforms for which `FFTW_DESTROY_INPUT` is the default. In the latter cases, passing `FFTW_PRESERVE_INPUT` will attempt to use algorithms that do not destroy the input, at the expense of worse performance; for multi-dimensional c2r transforms, however, no input-preserving algorithms are implemented and the planner will return `NULL` if one is requested.
- `FFTW_UNALIGNED` specifies that the algorithm may not impose any unusual alignment requirements on the input/output arrays (i.e. no SIMD may be used). This flag is normally *not necessary*, since the planner automatically detects misaligned arrays. The only use for this flag is if you want to use the new-array execute interface to execute a given plan on a different array that may not be aligned like the original. (Using `fftw_malloc` makes this flag unnecessary even then. You can also use `fftw_alignment_of` to detect whether two arrays are equivalently aligned.)

Limiting planning time

```
extern void fftw_set_timelimit(double seconds);
```

This function instructs FFTW to spend at most `seconds` seconds (approximately) in the planner. If `seconds == FFTW_NO_TIMELIMIT` (the default value, which is negative), then planning time is unbounded. Otherwise, FFTW plans with a progressively wider range of algorithms until the the given time limit is reached or the given range of algorithms is explored, returning the best available plan.

For example, specifying FFTW_PATIENT first plans in FFTW_ESTIMATE mode, then in FFTW_MEASURE mode, then finally (time permitting) in FFTW_PATIENT. If FFTW_EXHAUSTIVE is specified instead, the planner will further progress to FFTW_EXHAUSTIVE mode.

Note that the seconds argument specifies only a rough limit; in practice, the planner may use somewhat more time if the time limit is reached when the planner is in the middle of an operation that cannot be interrupted. At the very least, the planner will complete planning in FFTW_ESTIMATE mode (which is thus equivalent to a time limit of 0).

4.3.3 Real-data DFTs

```
fftw_plan fftw_plan_dft_r2c_1d(int n0,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_2d(int n0, int n1,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_3d(int n0, int n1, int n2,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c(int rank, const int *n,
                            double *in, fftw_complex *out,
                            unsigned flags);
```

Plan a real-input/complex-output discrete Fourier transform (DFT) in zero or more dimensions, returning an fftw_plan (see Section 4.2 [Using Plans], page 22).

Once you have created a plan for a certain transform type and parameters, then creating another plan of the same type and parameters, but for different arrays, is fast and shares constant data with the first plan (if it still exists).

The planner returns NULL if the plan cannot be created. A non-NULL plan is always returned by the basic interface unless you are using a customized FFTW configuration supporting a restricted set of transforms, or if you use the FFTW_PRESERVE_INPUT flag with a multi-dimensional out-of-place c2r transform (see below).

Arguments

- rank is the rank of the transform (it should be the size of the array *n), and can be any non-negative integer. (See Section 2.2 [Complex Multi-Dimensional DFTs], page 5, for the definition of "rank".) The '_1d', '_2d', and '_3d' planners correspond to a rank of 1, 2, and 3, respectively. The rank may be zero, which is equivalent to a rank-1 transform of size 1, i.e. a copy of one real number (with zero imaginary part) from input to output.
- n0, n1, n2, or n[0..rank-1], (as appropriate for each routine) specify the size of the transform dimensions. They can be any positive integer. This is different in general from the *physical* array dimensions, which are described in Section 4.3.4 [Real-data DFT Array Format], page 28.
  - 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 algorithm (which nevertheless retains $O(n \log n)$ performance even for prime sizes). (It is possible to customize FFTW for different array sizes; see Chapter 10 [Installation and Customization], page 97.) Transforms whose sizes are powers of 2 are especially fast, and it is generally beneficial for the *last* dimension of an r2c/c2r transform to be *even*.

- `in` and `out` point to the input and output arrays of the transform, which may be the same (yielding an in-place transform). These arrays are overwritten during planning, unless `FFTW_ESTIMATE` is used in the flags. (The arrays need not be initialized, but they must be allocated.) For an in-place transform, it is important to remember that the real array will require padding, described in Section 4.3.4 [Real-data DFT Array Format], page 28.
- `flags` is a bitwise OR ('`|`') of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 25.

The inverse transforms, taking complex input (storing the non-redundant half of a logically Hermitian array) to real output, are given by:

```
fftw_plan fftw_plan_dft_c2r_1d(int n0,
                               fftw_complex *in, double *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_c2r_2d(int n0, int n1,
                               fftw_complex *in, double *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_c2r_3d(int n0, int n1, int n2,
                               fftw_complex *in, double *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_c2r(int rank, const int *n,
                            fftw_complex *in, double *out,
                            unsigned flags);
```

The arguments are the same as for the r2c transforms, except that the input and output data formats are reversed.

FFTW computes an unnormalized transform: computing an r2c followed by a c2r transform (or vice versa) will result in the original data multiplied by the size of the transform (the product of the logical dimensions). An r2c transform produces the same output as a `FFTW_FORWARD` complex DFT of the same input, and a c2r transform is correspondingly equivalent to `FFTW_BACKWARD`. For more information, see Section 4.8 [What FFTW Really Computes], page 42.

4.3.4 Real-data DFT Array Format

The output of a DFT of real data (r2c) contains symmetries that, in principle, make half of the outputs redundant (see Section 4.8 [What FFTW Really Computes], page 42). (Similarly for the input of an inverse c2r transform.) In practice, it is not possible to entirely realize these savings in an efficient and understandable format that generalizes to multi-dimensional transforms. Instead, the output of the r2c 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 d (= `rank`) dimensions $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$, the complex data is an $n_0 \times n_1 \times n_2 \times \cdots \times (n_{d-1}/2 + 1)$ array of `fftw_complex` values in row-major order (with the division rounded down). That is, we only store the *lower* half (non-negative frequencies), 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.)

For an out-of-place transform, the real data is simply an array with physical dimensions $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ in row-major order.

For an in-place transform, some complications arise since the complex data is slightly larger than the real data. 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 $2(n_{d-1}/2 + 1)$ `double` values (exactly enough to hold the complex data). This physical array size does not, however, change the *logical* array size—only n_{d-1} values are actually stored in the last dimension, and n_{d-1} is the last dimension passed to the planner.

4.3.5 Real-to-Real Transforms

```
fftw_plan fftw_plan_r2r_1d(int n, double *in, double *out,
                           fftw_r2r_kind kind, unsigned flags);
fftw_plan fftw_plan_r2r_2d(int n0, int n1, double *in, double *out,
                           fftw_r2r_kind kind0, fftw_r2r_kind kind1,
                           unsigned flags);
fftw_plan fftw_plan_r2r_3d(int n0, int n1, int n2,
                           double *in, double *out,
                           fftw_r2r_kind kind0,
                           fftw_r2r_kind kind1,
                           fftw_r2r_kind kind2,
                           unsigned flags);
fftw_plan fftw_plan_r2r(int rank, const int *n, double *in, double *out,
                        const fftw_r2r_kind *kind, unsigned flags);
```

Plan a real input/output (r2r) transform of various kinds in zero or more dimensions, returning an `fftw_plan` (see Section 4.2 [Using Plans], page 22).

Once you have created a plan for a certain transform type and parameters, then creating another plan of the same type and parameters, but for different arrays, is fast and shares constant data with the first plan (if it still exists).

The planner returns `NULL` if the plan cannot be created. A non-`NULL` plan is always returned by the basic interface unless you are using a customized FFTW configuration supporting a restricted set of transforms, or for size-1 `FFTW_REDFT00` kinds (which are not defined).

Arguments

- `rank` is the dimensionality of the transform (it should be the size of the arrays `*n` and `*kind`), and can be any non-negative integer. The '`_1d`', '`_2d`', and '`_3d`' planners correspond to a `rank` of `1`, `2`, and `3`, respectively. A `rank` of zero is equivalent to a copy of one number from input to output.

- `n`, or `n0`/`n1`/`n2`, or `n[rank]`, respectively, gives the (physical) size of the transform dimensions. They can be any positive integer.
  - Multi-dimensional arrays are stored in row-major order with dimensions: `n0` x `n1`; or `n0` x `n1` x `n2`; or `n[0]` x `n[1]` x ... x `n[rank-1]`. See Section 3.2 [Multi-dimensional Array Format], page 15.
  - FFTW is generally 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 algorithm (which nevertheless retains $O(n \log n)$ performance even for prime sizes). (It is possible to customize FFTW for different array sizes; see Chapter 10 [Installation and Customization], page 97.) Transforms whose sizes are powers of 2 are especially fast.
  - For a `REDFT00` or `RODFT00` transform kind in a dimension of size n, it is $n-1$ or $n+1$, respectively, that should be factorizable in the above form.
- `in` and `out` point to the input and output arrays of the transform, which may be the same (yielding an in-place transform). These arrays are overwritten during planning, unless `FFTW_ESTIMATE` is used in the flags. (The arrays need not be initialized, but they must be allocated.)
- `kind`, or `kind0`/`kind1`/`kind2`, or `kind[rank]`, is the kind of r2r transform used for the corresponding dimension. The valid kind constants are described in Section 4.3.6 [Real-to-Real Transform Kinds], page 30. In a multi-dimensional transform, what is computed is the separable product formed by taking each transform kind along the corresponding dimension, one dimension after another.
- `flags` is a bitwise OR (`'|'`) of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 25.

4.3.6 Real-to-Real Transform Kinds

FFTW currently supports 11 different r2r transform kinds, specified by one of the constants below. For the precise definitions of these transforms, see Section 4.8 [What FFTW Really Computes], page 42. For a more colloquial introduction to these transform kinds, see Section 2.5 [More DFTs of Real Data], page 10.

For dimension of size `n`, there is a corresponding "logical" dimension `N` that determines the normalization (and the optimal factorization); the formula for `N` is given for each kind below. Also, with each transform kind is listed its corrsponding inverse transform. FFTW computes unnormalized transforms: a transform followed by its inverse will result in the original data multiplied by `N` (or the product of the `N`'s for each dimension, in multi-dimensions).

- `FFTW_R2HC` computes a real-input DFT with output in "halfcomplex" format, i.e. real and imaginary parts for a transform of size `n` stored as:

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

  (Logical `N=n`, inverse is `FFTW_HC2R`.)
- `FFTW_HC2R` computes the reverse of `FFTW_R2HC`, above. (Logical `N=n`, inverse is `FFTW_R2HC`.)
- `FFTW_DHT` computes a discrete Hartley transform. (Logical `N=n`, inverse is `FFTW_DHT`.)

- `FFTW_REDFT00` computes an REDFT00 transform, i.e. a DCT-I. (Logical `N=2*(n-1)`, inverse is `FFTW_REDFT00`.)
- `FFTW_REDFT10` computes an REDFT10 transform, i.e. a DCT-II (sometimes called "the" DCT). (Logical `N=2*n`, inverse is `FFTW_REDFT01`.)
- `FFTW_REDFT01` computes an REDFT01 transform, i.e. a DCT-III (sometimes called "the" IDCT, being the inverse of DCT-II). (Logical `N=2*n`, inverse is `FFTW_REDFT=10`.)
- `FFTW_REDFT11` computes an REDFT11 transform, i.e. a DCT-IV. (Logical `N=2*n`, inverse is `FFTW_REDFT11`.)
- `FFTW_RODFT00` computes an RODFT00 transform, i.e. a DST-I. (Logical `N=2*(n+1)`, inverse is `FFTW_RODFT00`.)
- `FFTW_RODFT10` computes an RODFT10 transform, i.e. a DST-II. (Logical `N=2*n`, inverse is `FFTW_RODFT01`.)
- `FFTW_RODFT01` computes an RODFT01 transform, i.e. a DST-III. (Logical `N=2*n`, inverse is `FFTW_RODFT=10`.)
- `FFTW_RODFT11` computes an RODFT11 transform, i.e. a DST-IV. (Logical `N=2*n`, inverse is `FFTW_RODFT11`.)

4.4 Advanced Interface

FFTW's "advanced" interface supplements the basic interface with four new planner routines, providing a new level of flexibility: you can plan a transform of multiple arrays simultaneously, operate on non-contiguous (strided) data, and transform a subset of a larger multi-dimensional array. Other than these additional features, the planner operates in the same fashion as in the basic interface, and the resulting `fftw_plan` is used in the same way (see Section 4.2 [Using Plans], page 22).

4.4.1 Advanced Complex DFTs

```
fftw_plan fftw_plan_many_dft(int rank, const int *n, int howmany,
                             fftw_complex *in, const int *inembed,
                             int istride, int idist,
                             fftw_complex *out, const int *onembed,
                             int ostride, int odist,
                             int sign, unsigned flags);
```

This routine plans multiple multidimensional complex DFTs, and it extends the `fftw_plan_dft` routine (see Section 4.3.1 [Complex DFTs], page 24) to compute `howmany` transforms, each having rank `rank` and size `n`. In addition, the transform data need not be contiguous, but it may be laid out in memory with an arbitrary stride. To account for these possibilities, `fftw_plan_many_dft` adds the new parameters `howmany`, {`i`,`o`}`nembed`, {`i`,`o`}`stride`, and {`i`,`o`}`dist`. The FFTW basic interface (see Section 4.3.1 [Complex DFTs], page 24) provides routines specialized for ranks 1, 2, and 3, but the advanced interface handles only the general-rank case.

`howmany` is the number of transforms to compute. The resulting plan computes `howmany` transforms, where the input of the `k`-th transform is at location `in+k*idist` (in C pointer arithmetic), and its output is at location `out+k*odist`. Plans obtained in this way can

often be faster than calling FFTW multiple times for the individual transforms. The basic `fftw_plan_dft` interface corresponds to `howmany=1` (in which case the `dist` parameters are ignored).

Each of the `howmany` transforms has rank `rank` and size `n`, as in the basic interface. In addition, the advanced interface allows the input and output arrays of each transform to be row-major subarrays of larger rank-`rank` arrays, described by `inembed` and `onembed` parameters, respectively. `{i,o}nembed` must be arrays of length `rank`, and `n` should be elementwise less than or equal to `{i,o}nembed`. Passing `NULL` for an `nembed` parameter is equivalent to passing `n` (i.e. same physical and logical dimensions, as in the basic interface.)

The `stride` parameters indicate that the `j`-th element of the input or output arrays is located at `j*istride` or `j*ostride`, respectively. (For a multi-dimensional array, `j` is the ordinary row-major index.) When combined with the `k`-th transform in a `howmany` loop, from above, this means that the (`j`,`k`)-th element is at `j*stride+k*dist`. (The basic `fftw_plan_dft` interface corresponds to a stride of 1.)

For in-place transforms, the input and output `stride` and `dist` parameters should be the same; otherwise, the planner may return `NULL`.

Arrays `n`, `inembed`, and `onembed` are not used after this function returns. You can safely free or reuse them.

Examples: One transform of one 5 by 6 array contiguous in memory:

```
int rank = 2;
int n[] = {5, 6};
int howmany = 1;
int idist = odist = 0; /* unused because howmany = 1 */
int istride = ostride = 1; /* array is contiguous in memory */
int *inembed = n, *onembed = n;
```

Transform of three 5 by 6 arrays, each contiguous in memory, stored in memory one after another:

```
int rank = 2;
int n[] = {5, 6};
int howmany = 3;
int idist = odist = n[0]*n[1]; /* = 30, the distance in memory
                                  between the first element
                                  of the first array and the
                                  first element of the second array */
int istride = ostride = 1; /* array is contiguous in memory */
int *inembed = n, *onembed = n;
```

Transform each column of a 2d array with 10 rows and 3 columns:

```
int rank = 1; /* not 2: we are computing 1d transforms */
int n[] = {10}; /* 1d transforms of length 10 */
int howmany = 3;
int idist = odist = 1;
int istride = ostride = 3; /* distance between two elements in
```

```
                                    the same column */
     int *inembed = n, *onembed = n;
```

4.4.2 Advanced Real-data DFTs

```
fftw_plan fftw_plan_many_dft_r2c(int rank, const int *n, int howmany,
                                 double *in, const int *inembed,
                                 int istride, int idist,
                                 fftw_complex *out, const int *onembed,
                                 int ostride, int odist,
                                 unsigned flags);
fftw_plan fftw_plan_many_dft_c2r(int rank, const int *n, int howmany,
                                 fftw_complex *in, const int *inembed,
                                 int istride, int idist,
                                 double *out, const int *onembed,
                                 int ostride, int odist,
                                 unsigned flags);
```

Like `fftw_plan_many_dft`, these two functions add `howmany`, `nembed`, `stride`, and `dist` parameters to the `fftw_plan_dft_r2c` and `fftw_plan_dft_c2r` functions, but otherwise behave the same as the basic interface.

The interpretation of `howmany`, `stride`, and `dist` are the same as for `fftw_plan_many_dft`, above. Note that the `stride` and `dist` for the real array are in units of `double`, and for the complex array are in units of `fftw_complex`.

If an `nembed` parameter is `NULL`, it is interpreted as what it would be in the basic interface, as described in Section 4.3.4 [Real-data DFT Array Format], page 28. That is, for the complex array the size is assumed to be the same as `n`, but with the last dimension cut roughly in half. For the real array, the size is assumed to be `n` if the transform is out-of-place, or `n` with the last dimension "padded" if the transform is in-place.

If an `nembed` parameter is non-`NULL`, it is interpreted as the physical size of the corresponding array, in row-major order, just as for `fftw_plan_many_dft`. In this case, each dimension of `nembed` should be `>=` what it would be in the basic interface (e.g. the halved or padded `n`).

Arrays `n`, `inembed`, and `onembed` are not used after this function returns. You can safely free or reuse them.

4.4.3 Advanced Real-to-real Transforms

```
fftw_plan fftw_plan_many_r2r(int rank, const int *n, int howmany,
                             double *in, const int *inembed,
                             int istride, int idist,
                             double *out, const int *onembed,
                             int ostride, int odist,
                             const fftw_r2r_kind *kind, unsigned flags);
```

Like `fftw_plan_many_dft`, this functions adds `howmany`, `nembed`, `stride`, and `dist` parameters to the `fftw_plan_r2r` function, but otherwise behave the same as the basic interface. The interpretation of those additional parameters are the same as for `fftw_plan_many_dft`. (Of course, the `stride` and `dist` parameters are now in units of `double`, not `fftw_complex`.)

Arrays `n`, `inembed`, `onembed`, and `kind` are not used after this function returns. You can safely free or reuse them.

4.5 Guru Interface

The "guru" interface to FFTW is intended to expose as much as possible of the flexibility in the underlying FFTW architecture. It allows one to compute multi-dimensional "vectors" (loops) of multi-dimensional transforms, where each vector/transform dimension has an independent size and stride. One can also use more general complex-number formats, e.g. separate real and imaginary arrays.

For those users who require the flexibility of the guru interface, it is important that they pay special attention to the documentation lest they shoot themselves in the foot.

4.5.1 Interleaved and split arrays

The guru interface supports two representations of complex numbers, which we call the interleaved and the split format.

The *interleaved* format is the same one used by the basic and advanced interfaces, and it is documented in Section 4.1.1 [Complex numbers], page 21. In the interleaved format, you provide pointers to the real part of a complex number, and the imaginary part understood to be stored in the next memory location.

The *split* format allows separate pointers to the real and imaginary parts of a complex array.

Technically, the interleaved format is redundant, because you can always express an interleaved array in terms of a split array with appropriate pointers and strides. On the other hand, the interleaved format is simpler to use, and it is common in practice. Hence, FFTW supports it as a special case.

4.5.2 Guru vector and transform sizes

The guru interface introduces one basic new data structure, `fftw_iodim`, that is used to specify sizes and strides for multi-dimensional transforms and vectors:

```
typedef struct {
     int n;
     int is;
     int os;
} fftw_iodim;
```

Here, `n` is the size of the dimension, and `is` and `os` are the strides of that dimension for the input and output arrays. (The stride is the separation of consecutive elements along this dimension.)

The meaning of the stride parameter depends on the type of the array that the stride refers to. *If the array is interleaved complex, strides are expressed in units of complex numbers (*`fftw_complex`*). If the array is split complex or real, strides are expressed in units of real numbers (*`double`*).* This convention is consistent with the usual pointer arithmetic in the C language. An interleaved array is denoted by a pointer `p` to `fftw_complex`, so that `p+1`

points to the next complex number. Split arrays are denoted by pointers to `double`, in which case pointer arithmetic operates in units of `sizeof(double)`.

The guru planner interfaces all take a (`rank`, `dims[rank]`) pair describing the transform size, and a (`howmany_rank`, `howmany_dims[howmany_rank]`) pair describing the "vector" size (a multi-dimensional loop of transforms to perform), where `dims` and `howmany_dims` are arrays of `fftw_iodim`.

For example, the `howmany` parameter in the advanced complex-DFT interface corresponds to `howmany_rank` = 1, `howmany_dims[0].n` = `howmany`, `howmany_dims[0].is` = `idist`, and `howmany_dims[0].os` = `odist`. (To compute a single transform, you can just use `howmany_rank` = 0.)

A row-major multidimensional array with dimensions `n[rank]` (see Section 3.2.1 [Row-major Format], page 15) corresponds to `dims[i].n` = `n[i]` and the recurrence `dims[i].is` = `n[i+1] * dims[i+1].is` (similarly for `os`). The stride of the last (`i=rank-1`) dimension is the overall stride of the array. e.g. to be equivalent to the advanced complex-DFT interface, you would have `dims[rank-1].is` = `istride` and `dims[rank-1].os` = `ostride`.

In general, we only guarantee FFTW to return a non-`NULL` plan if the vector and transform dimensions correspond to a set of distinct indices, and for in-place transforms the input/output strides should be the same.

4.5.3 Guru Complex DFTs

```
fftw_plan fftw_plan_guru_dft(
     int rank, const fftw_iodim *dims,
     int howmany_rank, const fftw_iodim *howmany_dims,
     fftw_complex *in, fftw_complex *out,
     int sign, unsigned flags);

fftw_plan fftw_plan_guru_split_dft(
     int rank, const fftw_iodim *dims,
     int howmany_rank, const fftw_iodim *howmany_dims,
     double *ri, double *ii, double *ro, double *io,
     unsigned flags);
```

These two functions plan a complex-data, multi-dimensional DFT for the interleaved and split format, respectively. Transform dimensions are given by (`rank`, `dims`) over a multi-dimensional vector (loop) of dimensions (`howmany_rank`, `howmany_dims`). `dims` and `howmany_dims` should point to `fftw_iodim` arrays of length `rank` and `howmany_rank`, respectively.

`flags` is a bitwise OR ('`|`') of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 25.

In the `fftw_plan_guru_dft` function, the pointers `in` and `out` point to the interleaved input and output arrays, respectively. The sign can be either -1 (= `FFTW_FORWARD`) or $+1$ (= `FFTW_BACKWARD`). If the pointers are equal, the transform is in-place.

In the `fftw_plan_guru_split_dft` function, `ri` and `ii` point to the real and imaginary input arrays, and `ro` and `io` point to the real and imaginary output arrays. The input

and output pointers may be the same, indicating an in-place transform. For example, for `fftw_complex` pointers `in` and `out`, the corresponding parameters are:

```
ri = (double *) in;
ii = (double *) in + 1;
ro = (double *) out;
io = (double *) out + 1;
```

Because `fftw_plan_guru_split_dft` accepts split arrays, strides are expressed in units of `double`. For a contiguous `fftw_complex` array, the overall stride of the transform should be 2, the distance between consecutive real parts or between consecutive imaginary parts; see Section 4.5.2 [Guru vector and transform sizes], page 34. Note that the dimension strides are applied equally to the real and imaginary parts; real and imaginary arrays with different strides are not supported.

There is no `sign` parameter in `fftw_plan_guru_split_dft`. This function always plans for an `FFTW_FORWARD` transform. To plan for an `FFTW_BACKWARD` transform, you can exploit the identity that the backwards DFT is equal to the forwards DFT with the real and imaginary parts swapped. For example, in the case of the `fftw_complex` arrays above, the `FFTW_BACKWARD` transform is computed by the parameters:

```
ri = (double *) in + 1;
ii = (double *) in;
ro = (double *) out + 1;
io = (double *) out;
```

4.5.4 Guru Real-data DFTs

```
fftw_plan fftw_plan_guru_dft_r2c(
     int rank, const fftw_iodim *dims,
     int howmany_rank, const fftw_iodim *howmany_dims,
     double *in, fftw_complex *out,
     unsigned flags);

fftw_plan fftw_plan_guru_split_dft_r2c(
     int rank, const fftw_iodim *dims,
     int howmany_rank, const fftw_iodim *howmany_dims,
     double *in, double *ro, double *io,
     unsigned flags);

fftw_plan fftw_plan_guru_dft_c2r(
     int rank, const fftw_iodim *dims,
     int howmany_rank, const fftw_iodim *howmany_dims,
     fftw_complex *in, double *out,
     unsigned flags);

fftw_plan fftw_plan_guru_split_dft_c2r(
     int rank, const fftw_iodim *dims,
     int howmany_rank, const fftw_iodim *howmany_dims,
     double *ri, double *ii, double *out,
```

```
unsigned flags);
```

Plan a real-input (r2c) or real-output (c2r), multi-dimensional DFT with transform dimensions given by (`rank`, `dims`) over a multi-dimensional vector (loop) of dimensions (`howmany_rank`, `howmany_dims`). `dims` and `howmany_dims` should point to `fftw_iodim` arrays of length `rank` and `howmany_rank`, respectively. As for the basic and advanced interfaces, an r2c transform is `FFTW_FORWARD` and a c2r transform is `FFTW_BACKWARD`.

The *last* dimension of `dims` is interpreted specially: that dimension of the real array has size `dims[rank-1].n`, but that dimension of the complex array has size `dims[rank-1].n/2+1` (division rounded down). The strides, on the other hand, are taken to be exactly as specified. It is up to the user to specify the strides appropriately for the peculiar dimensions of the data, and we do not guarantee that the planner will succeed (return non-`NULL`) for any dimensions other than those described in Section 4.3.4 [Real-data DFT Array Format], page 28 and generalized in Section 4.4.2 [Advanced Real-data DFTs], page 33. (That is, for an in-place transform, each individual dimension should be able to operate in place.)

`in` and `out` point to the input and output arrays for r2c and c2r transforms, respectively. For split arrays, `ri` and `ii` point to the real and imaginary input arrays for a c2r transform, and `ro` and `io` point to the real and imaginary output arrays for an r2c transform. `in` and `ro` or `ri` and `out` may be the same, indicating an in-place transform. (In-place transforms where `in` and `io` or `ii` and `out` are the same are not currently supported.)

`flags` is a bitwise OR (‘`|`’) of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 25.

In-place transforms of rank greater than 1 are currently only supported for interleaved arrays. For split arrays, the planner will return `NULL`.

4.5.5 Guru Real-to-real Transforms

```
fftw_plan fftw_plan_guru_r2r(int rank, const fftw_iodim *dims,
                             int howmany_rank,
                             const fftw_iodim *howmany_dims,
                             double *in, double *out,
                             const fftw_r2r_kind *kind,
                             unsigned flags);
```

Plan a real-to-real (r2r) multi-dimensional `FFTW_FORWARD` transform with transform dimensions given by (`rank`, `dims`) over a multi-dimensional vector (loop) of dimensions (`howmany_rank`, `howmany_dims`). `dims` and `howmany_dims` should point to `fftw_iodim` arrays of length `rank` and `howmany_rank`, respectively.

The transform kind of each dimension is given by the `kind` parameter, which should point to an array of length `rank`. Valid `fftw_r2r_kind` constants are given in Section 4.3.6 [Real-to-Real Transform Kinds], page 30.

`in` and `out` point to the real input and output arrays; they may be the same, indicating an in-place transform.

`flags` is a bitwise OR (‘`|`’) of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 25.

4.5.6 64-bit Guru Interface

When compiled in 64-bit mode on a 64-bit architecture (where addresses are 64 bits wide), FFTW uses 64-bit quantities internally for all transform sizes, strides, and so on—you don't have to do anything special to exploit this. However, in the ordinary FFTW interfaces, you specify the transform size by an `int` quantity, which is normally only 32 bits wide. This means that, even though FFTW is using 64-bit sizes internally, you cannot specify a single transform dimension larger than $2^31 - 1$ numbers.

We expect that few users will require transforms larger than this, but, for those who do, we provide a 64-bit version of the guru interface in which all sizes are specified as integers of type `ptrdiff_t` instead of `int`. (`ptrdiff_t` is a signed integer type defined by the C standard to be wide enough to represent address differences, and thus must be at least 64 bits wide on a 64-bit machine.) We stress that there is *no performance advantage* to using this interface—the same internal FFTW code is employed regardless—and it is only necessary if you want to specify very large transform sizes.

In particular, the 64-bit guru interface is a set of planner routines that are exactly the same as the guru planner routines, except that they are named with '`guru64`' instead of '`guru`' and they take arguments of type `fftw_iodim64` instead of `fftw_iodim`. For example, instead of `fftw_plan_guru_dft`, we have `fftw_plan_guru64_dft`.

```
fftw_plan fftw_plan_guru64_dft(
     int rank, const fftw_iodim64 *dims,
     int howmany_rank, const fftw_iodim64 *howmany_dims,
     fftw_complex *in, fftw_complex *out,
     int sign, unsigned flags);
```

The `fftw_iodim64` type is similar to `fftw_iodim`, with the same interpretation, except that it uses type `ptrdiff_t` instead of type `int`.

```
typedef struct {
     ptrdiff_t n;
     ptrdiff_t is;
     ptrdiff_t os;
} fftw_iodim64;
```

Every other '`fftw_plan_guru`' function also has a '`fftw_plan_guru64`' equivalent, but we do not repeat their documentation here since they are identical to the 32-bit versions except as noted above.

4.6 New-array Execute Functions

Normally, one executes a plan for the arrays with which the plan was created, by calling `fftw_execute(plan)` as described in Section 4.2 [Using Plans], page 22. However, it is possible for sophisticated users to apply a given plan to a *different* array using the "new-array execute" functions detailed below, provided that the following conditions are met:

- The array size, strides, etcetera are the same (since those are set by the plan).
- The input and output arrays are the same (in-place) or different (out-of-place) if the plan was originally created to be in-place or out-of-place, respectively.

- For split arrays, the separations between the real and imaginary parts, `ii-ri` and `io-ro`, are the same as they were for the input and output arrays when the plan was created. (This condition is automatically satisfied for interleaved arrays.)
- The *alignment* of the new input/output arrays is the same as that of the input/output arrays when the plan was created, unless the plan was created with the `FFTW_UNALIGNED` flag. Here, the alignment is a platform-dependent quantity (for example, it is the address modulo 16 if SSE SIMD instructions are used, but the address modulo 4 for non-SIMD single-precision FFTW on the same machine). In general, only arrays allocated with `fftw_malloc` are guaranteed to be equally aligned (see Section 3.1 [SIMD alignment and fftw_malloc], page 15).

The alignment issue is especially critical, because if you don't use `fftw_malloc` then you may have little control over the alignment of arrays in memory. For example, neither the C++ `new` function nor the Fortran `allocate` statement provide strong enough guarantees about data alignment. If you don't use `fftw_malloc`, therefore, you probably have to use `FFTW_UNALIGNED` (which disables most SIMD support). If possible, it is probably better for you to simply create multiple plans (creating a new plan is quick once one exists for a given size), or better yet re-use the same array for your transforms.

For rare circumstances in which you cannot control the alignment of allocated memory, but wish to determine where a given array is aligned like the original array for which a plan was created, you can use the `fftw_alignment_of` function:

```
int fftw_alignment_of(double *p);
```

Two arrays have equivalent alignment (for the purposes of applying a plan) if and only if `fftw_alignment_of` returns the same value for the corresponding pointers to their data (typecast to `double*` if necessary).

If you are tempted to use the new-array execute interface because you want to transform a known bunch of arrays of the same size, you should probably go use the advanced interface instead (see Section 4.4 [Advanced Interface], page 31).

The new-array execute functions are:

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

void fftw_execute_split_dft(
     const fftw_plan p,
     double *ri, double *ii, double *ro, double *io);

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

void fftw_execute_split_dft_r2c(
     const fftw_plan p,
     double *in, double *ro, double *io);
```

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

void fftw_execute_split_dft_c2r(
     const fftw_plan p,
     double *ri, double *ii, double *out);

void fftw_execute_r2r(
     const fftw_plan p,
     double *in, double *out);
```

These execute the `plan` to compute the corresponding transform on the input/output arrays specified by the subsequent arguments. The input/output array arguments have the same meanings as the ones passed to the guru planner routines in the preceding sections. The `plan` is not modified, and these routines can be called as many times as desired, or intermixed with calls to the ordinary `fftw_execute`.

The `plan` *must* have been created for the transform type corresponding to the execute function, e.g. it must be a complex-DFT plan for `fftw_execute_dft`. Any of the planner routines for that transform type, from the basic to the guru interface, could have been used to create the plan, however.

4.7 Wisdom

This section documents the FFTW mechanism for saving and restoring plans from disk. This mechanism is called *wisdom*.

4.7.1 Wisdom Export

```
int fftw_export_wisdom_to_filename(const char *filename);
void fftw_export_wisdom_to_file(FILE *output_file);
char *fftw_export_wisdom_to_string(void);
void fftw_export_wisdom(void (*write_char)(char c, void *), void *data);
```

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 3.3 [Words of Wisdom-Saving Plans], page 18.) 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 `write_char`. `write_char` 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 three "wrapper" routines are provided:

`fftw_export_wisdom_to_filename` writes wisdom to a file named `filename` (which is created or overwritten), returning `1` on success and `0` on failure. A lower-level function, which requires you to open and close the file yourself (e.g. if you want to write wisdom to a portion of a larger file) is `fftw_export_wisdom_to_file`. This 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 `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.

4.7.2 Wisdom Import

```
int fftw_import_system_wisdom(void);
int fftw_import_wisdom_from_filename(const char *filename);
int fftw_import_wisdom_from_string(const char *input_string);
int fftw_import_wisdom(int (*read_char)(void *), void *data);
```

These functions import wisdom into a program from data stored by the `fftw_export_wisdom` functions above. (See Section 3.3 [Words of Wisdom-Saving Plans], page 18.) The imported wisdom replaces any wisdom already accumulated by the running program.

`fftw_import_wisdom` imports wisdom from any input medium, as specified by the callback function `read_char`. `read_char` 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), `read_char` should return `EOF` (as defined in `<stdio.h>`). For convenience, the following three "wrapper" routines are provided:

`fftw_import_wisdom_from_filename` reads wisdom from a file named `filename`. A lower-level function, which requires you to open and close the file yourself (e.g. if you want to read wisdom from a portion of a larger file) is `fftw_import_wisdom_from_file`. This reads wisdom from the current position in `input_file` (which should be open with read permission); upon exit, the file remains open, but the position of the read pointer is unspecified.

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

`fftw_import_system_wisdom` reads wisdom from an implementation-defined standard file (`/etc/fftw/wisdom` on Unix and GNU systems).

The return value of these import routines is `1` if the wisdom was read successfully and `0` otherwise. Note that, in all of these functions, any data in the input stream past the end of the wisdom data is simply ignored.

4.7.3 Forgetting Wisdom

```
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.)

4.7.4 Wisdom Utilities

FFTW includes two standalone utility programs that deal with wisdom. We merely summarize them here, since they come with their own `man` pages for Unix and GNU systems (with HTML versions on our web site).

The first program is `fftw-wisdom` (or `fftwf-wisdom` in single precision, etcetera), which can be used to create a wisdom file containing plans for any of the transform sizes and types supported by FFTW. It is preferable to create wisdom directly from your executable (see Section 3.4 [Caveats in Using Wisdom], page 18), but this program is useful for creating global wisdom files for `fftw_import_system_wisdom`.

The second program is `fftw-wisdom-to-conf`, which takes a wisdom file as input and produces a *configuration routine* as output. The latter is a C subroutine that you can compile and link into your program, replacing a routine of the same name in the FFTW library, that determines which parts of FFTW are callable by your program. `fftw-wisdom-to-conf` produces a configuration routine that links to only those parts of FFTW needed by the saved plans in the wisdom, greatly reducing the size of statically linked executables (which should only attempt to create plans corresponding to those in the wisdom, however).

4.8 What FFTW Really Computes

In this section, we provide precise mathematical definitions for the transforms that FFTW computes. These transform definitions are fairly standard, but some authors follow slightly different conventions for the normalization of the transform (the constant factor in front) and the sign of the complex exponent. We begin by presenting the one-dimensional (1d) transform definitions, and then give the straightforward extension to multi-dimensional transforms.

4.8.1 The 1d Discrete Fourier Transform (DFT)

The forward (`FFTW_FORWARD`) discrete Fourier transform (DFT) of a 1d complex array X of size n computes an array Y, where:

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

The backward (`FFTW_BACKWARD`) DFT computes:

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

FFTW computes an unnormalized transform, in that there is no coefficient in front of the summation in the DFT. In other words, applying the forward and then the backward transform will multiply the input by n.

From above, 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 $(n-k)/n$.)

4.8.2 The 1d Real-data DFT

The real-input (r2c) DFT in FFTW computes the *forward* transform Y of the size `n` real array X, exactly as defined above, i.e.

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

This output array Y can easily be shown to possess the "Hermitian" symmetry $Y_k = Y^*_{n-k}$, where we take Y to be periodic so that $Y_n = Y_0$.

As a result of this symmetry, half of the output Y is redundant (being the complex conjugate of the other half), and so the 1d r2c transforms only output elements $0 \ldots n/2$ of Y ($n/2+1$ complex numbers), where the division by 2 is rounded down.

Moreover, the Hermitian symmetry implies that Y_0 and, if n is even, the $Y_{n/2}$ element, are purely real. So, for the `R2HC` r2r transform, these elements are not stored in the halfcomplex output format.

The c2r and `H2RC` r2r transforms compute the backward DFT of the *complex* array X with Hermitian symmetry, stored in the r2c/`R2HC` output formats, respectively, where the backward transform is defined exactly as for the complex case:

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

The outputs `Y` of this transform can easily be seen to be purely real, and are stored as an array of real numbers.

Like FFTW's complex DFT, these transforms are unnormalized. In other words, applying the real-to-complex (forward) and then the complex-to-real (backward) transform will multiply the input by n.

4.8.3 1d Real-even DFTs (DCTs)

The Real-even symmetry DFTs in FFTW are exactly equivalent to the unnormalized forward (and backward) DFTs as defined above, where the input array X of length N is purely real and is also *even* symmetry. In this case, the output array is likewise real and even symmetry.

For the case of `REDFT00`, this even symmetry means that $X_j = X_{N-j}$, where we take X to be periodic so that $X_N = X_0$. Because of this redundancy, only the first n real numbers are actually stored, where $N = 2(n-1)$.

The proper definition of even symmetry for `REDFT10`, `REDFT01`, and `REDFT11` transforms is somewhat more intricate because of the shifts by 1/2 of the input and/or output, although the corresponding boundary conditions are given in Section 2.5.2 [Real even/odd DFTs (cosine/sine transforms)], page 11. Because of the even symmetry, however, the sine terms in the DFT all cancel and the remaining cosine terms are written explicitly below. This formulation often leads people to call such a transform a *discrete cosine transform* (DCT), although it is really just a special case of the DFT.

In each of the definitions below, we transform a real array X of length n to a real array Y of length n:

REDFT00 (DCT-I)

An `REDFT00` transform (type-I DCT) in FFTW is defined by:

$$Y_k = X_0 + (-1)^k X_{n-1} + 2\sum_{j=1}^{n-2} X_j \cos[\pi jk/(n-1)].$$

Note that this transform is not defined for $n = 1$. For $n = 2$, the summation term above is dropped as you might expect.

REDFT10 (DCT-II)

An `REDFT10` transform (type-II DCT, sometimes called "the" DCT) in FFTW is defined by:

$$Y_k = 2\sum_{j=0}^{n-1} X_j \cos[\pi(j+1/2)k/n].$$

REDFT01 (DCT-III)

An `REDFT01` transform (type-III DCT) in FFTW is defined by:

$$Y_k = X_0 + 2\sum_{j=1}^{n-1} X_j \cos[\pi j(k+1/2)/n].$$

In the case of $n = 1$, this reduces to $Y_0 = X_0$. Up to a scale factor (see below), this is the inverse of `REDFT10` ("the" DCT), and so the `REDFT01` (DCT-III) is sometimes called the "IDCT".

REDFT11 (DCT-IV)

An `REDFT11` transform (type-IV DCT) in FFTW is defined by:

$$Y_k = 2\sum_{j=0}^{n-1} X_j \cos[\pi(j+1/2)(k+1/2)/n].$$

Inverses and Normalization

These definitions correspond directly to the unnormalized DFTs used elsewhere in FFTW (hence the factors of 2 in front of the summations). The unnormalized inverse of `REDFT00` is `REDFT00`, of `REDFT10` is `REDFT01` and vice versa, and of `REDFT11` is `REDFT11`. Each unnormalized inverse results in the original array multiplied by N, where N is the *logical* DFT size. For `REDFT00`, $N = 2(n-1)$ (note that $n = 1$ is not defined); otherwise, $N = 2n$.

In defining the discrete cosine transform, some authors also include additional factors of $\sqrt{2}$ (or its inverse) multiplying selected inputs and/or outputs. This is a mostly cosmetic change that makes the transform orthogonal, but sacrifices the direct equivalence to a symmetric DFT.

4.8.4 1d Real-odd DFTs (DSTs)

The Real-odd symmetry DFTs in FFTW are exactly equivalent to the unnormalized forward (and backward) DFTs as defined above, where the input array X of length N is purely real and is also *odd* symmetry. In this case, the output is odd symmetry and purely imaginary.

For the case of `RODFT00`, this odd symmetry means that $X_j = -X_{N-j}$, where we take X to be periodic so that $X_N = X_0$. Because of this redundancy, only the first n real numbers starting at $j = 1$ are actually stored (the $j = 0$ element is zero), where $N = 2(n + 1)$.

The proper definition of odd symmetry for `RODFT10`, `RODFT01`, and `RODFT11` transforms is somewhat more intricate because of the shifts by 1/2 of the input and/or output, although the corresponding boundary conditions are given in Section 2.5.2 [Real even/odd DFTs (cosine/sine transforms)], page 11. Because of the odd symmetry, however, the cosine terms in the DFT all cancel and the remaining sine terms are written explicitly below. This formulation often leads people to call such a transform a *discrete sine transform* (DST), although it is really just a special case of the DFT.

In each of the definitions below, we transform a real array X of length n to a real array Y of length n:

RODFT00 (DST-I)

An `RODFT00` transform (type-I DST) in FFTW is defined by:

$$Y_k = 2\sum_{j=0}^{n-1} X_j \sin[\pi(j+1)(k+1)/(n+1)].$$

RODFT10 (DST-II)

An `RODFT10` transform (type-II DST) in FFTW is defined by:

$$Y_k = 2\sum_{j=0}^{n-1} X_j \sin[\pi(j+1/2)(k+1)/n].$$

RODFT01 (DST-III)

An `RODFT01` transform (type-III DST) in FFTW is defined by:

$$Y_k = (-1)^k X_{n-1} + 2\sum_{j=0}^{n-2} X_j \sin[\pi(j+1)(k+1/2)/n].$$

In the case of $n = 1$, this reduces to $Y_0 = X_0$.

RODFT11 (DST-IV)

An `RODFT11` transform (type-IV DST) in FFTW is defined by:

$$Y_k = 2\sum_{j=0}^{n-1} X_j \sin[\pi(j+1/2)(k+1/2)/n].$$

Inverses and Normalization

These definitions correspond directly to the unnormalized DFTs used elsewhere in FFTW (hence the factors of 2 in front of the summations). The unnormalized inverse of `RODFT00` is `RODFT00`, of `RODFT10` is `RODFT01` and vice versa, and of `RODFT11` is `RODFT11`. Each unnormalized inverse results in the original array multiplied by N, where N is the *logical* DFT size. For `RODFT00`, $N = 2(n+1)$; otherwise, $N = 2n$.

In defining the discrete sine transform, some authors also include additional factors of $\sqrt{2}$ (or its inverse) multiplying selected inputs and/or outputs. This is a mostly cosmetic change that makes the transform orthogonal, but sacrifices the direct equivalence to an antisymmetric DFT.

4.8.5 1d Discrete Hartley Transforms (DHTs)

The discrete Hartley transform (DHT) of a 1d real array X of size n computes a real array Y of the same size, where:

$$Y_k = \sum_{j=0}^{n-1} X_j[\cos(2\pi jk/n) + \sin(2\pi jk/n)].$$

FFTW computes an unnormalized transform, in that there is no coefficient in front of the summation in the DHT. In other words, applying the transform twice (the DHT is its own inverse) will multiply the input by n.

4.8.6 Multi-dimensional Transforms

The multi-dimensional transforms of FFTW, in general, compute simply the separable product of the given 1d transform along each dimension of the array. Since each of these transforms is unnormalized, computing the forward followed by the backward/inverse multi-dimensional transform will result in the original array scaled by the product of the normalization factors for each dimension (e.g. the product of the dimension sizes, for a multi-dimensional DFT).

As an explicit example, consider the following exact mathematical definition of our multi-dimensional DFT. 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

$$Y[k_1, k_2, \ldots, k_d] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, j_2, \ldots, j_d] \omega_1^{-j_1 k_1} \omega_2^{-j_2 k_2} \cdots \omega_d^{-j_d k_d} \ .$$

The backward transform computes

$$Y[k_1, k_2, \ldots, k_d] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, j_2, \ldots, j_d] \omega_1^{j_1 k_1} \omega_2^{j_2 k_2} \cdots \omega_d^{j_d k_d} \ .$$

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

The definition of FFTW's multi-dimensional DFT of real data (r2c) deserves special attention. In this case, we logically compute the full multi-dimensional DFT of the input data; since the input data are purely real, the output data have the Hermitian symmetry and therefore only one non-redundant half need be stored. More specifically, for an $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ multi-dimensional real-input DFT, the full (logical) complex output array $Y[k_0, k_1, \ldots, k_{d-1}]$ has the symmetry:

$$Y[k_0, k_1, \ldots, k_{d-1}] = Y[n_0 - k_0, n_1 - k_1, \ldots, n_{d-1} - k_{d-1}]^*$$

(where each dimension is periodic). Because of this symmetry, we only store the $k_{d-1} = 0 \cdots n_{d-1}/2$ elements of the *last* dimension (division by 2 is rounded down). (We could instead have cut any other dimension in half, but the last dimension proved computationally convenient.) This results in the peculiar array format described in more detail by Section 4.3.4 [Real-data DFT Array Format], page 28.

The multi-dimensional c2r transform is simply the unnormalized inverse of the r2c transform. i.e. it is the same as FFTW's complex backward multi-dimensional DFT, operating on a Hermitian input array in the peculiar format mentioned above and outputting a real array (since the DFT output is purely real).

We should remind the user that the separable product of 1d transforms along each dimension, as computed by FFTW, is not always the same thing as the usual multi-dimensional transform. A multi-dimensional `R2HC` (or `HC2R`) transform is not identical to the multi-dimensional DFT, requiring some post-processing to combine the requisite real and imaginary parts, as was described in Section 2.5.1 [The Halfcomplex-format DFT], page 11. Likewise, FFTW's multidimensional `FFTW_DHT` r2r transform is not the same thing as the logical multi-dimensional discrete Hartley transform defined in the literature, as discussed in Section 2.5.3 [The Discrete Hartley Transform], page 13.

5 Multi-threaded FFTW

In this chapter we document the parallel FFTW routines for shared-memory parallel 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 you have an extra initialization routine to call, and there is a routine to set the number of threads to employ. Any program that uses the uniprocessor FFTW can therefore be trivially modified to use the multi-threaded FFTW.

A shared-memory machine is one in which all CPUs can directly access the same main memory, and such machines are now common due to the ubiquity of multi-core CPUs. FFTW's multi-threading support allows you to utilize these additional CPUs transparently from a single program. However, this does not necessarily translate into performance gains—when multiple threads/CPUs are employed, there is an overhead required for synchronization that may outweigh the computatational parallelism. Therefore, you can only benefit from threads if your problem is sufficiently large.

5.1 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 10.1 [Installation on Unix], page 97), or `--enable-openmp` to use OpenMP threads.

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, from GNU/Linux to MacOS X) and Win32 threads. OpenMP threads, which are supported in many common compilers (e.g. gcc) are also supported, and may give better performance on some systems. (OpenMP threads are also useful if you are employing OpenMP in your own code, in order to minimize conflicts between threading models.) 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 `threads/threads.c` for more detail.

You can compile FFTW with *both* `--enable-threads` and `--enable-openmp` at the same time, since they install libraries with different names ('`fftw3_threads`' and '`fftw3_omp`', as described below). However, your programs may only link to *one* of these two libraries at a time.

Ideally, of course, you should also have multiple processors in order to get any benefit from the threaded transforms.

5.2 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, programs using the parallel complex transforms should be linked with `-lfftw3_threads -lfftw3 -lm` on Unix, or `-lfftw3_omp -lfftw3 -lm` if you compiled with OpenMP. You will also need to link with whatever library is responsible for threads on your system (e.g. `-lpthread` on GNU/Linux) or include whatever compiler flag enables OpenMP (e.g. `-fopenmp` with gcc).

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

```
int fftw_init_threads(void);
```

This function, which need only be called once, performs any one-time initialization required to use threads on your system. It returns zero if there was some error (which should not happen under normal circumstances) and a non-zero value otherwise.

Third, before creating a plan that you want to parallelize, you should call:

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

The `nthreads` argument indicates the number of threads you want FFTW to use (or actually, the maximum number). All plans subsequently created with any planner routine will use that many threads. You can call `fftw_plan_with_nthreads`, create some plans, call `fftw_plan_with_nthreads` again with a different argument, and create some more plans for a new number of threads. Plans already created before a call to `fftw_plan_with_nthreads` are unaffected. If you pass an `nthreads` argument of `1` (the default), threads are disabled for subsequent plans.

With OpenMP, to configure FFTW to use all of the currently running OpenMP threads (set by `omp_set_num_threads(nthreads)` or by the `OMP_NUM_THREADS` environment variable), you can do: `fftw_plan_with_nthreads(omp_get_max_threads())`. (The '`omp_`' OpenMP functions are declared via `#include <omp.h>`.)

Given a plan, you then execute it as usual with `fftw_execute(plan)`, and the execution will use the number of threads specified when the plan was created. When done, you destroy it as usual with `fftw_destroy_plan`. As described in Section 5.4 [Thread safety], page 51, plan *execution* is thread-safe, but plan creation and destruction are *not*: you should create/destroy plans only from a single thread, but can safely execute multiple plans in parallel.

There is one additional routine: if you want to get rid of all memory and other resources allocated internally by FFTW, you can call:

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

which is much like the `fftw_cleanup()` function except that it also gets rid of threads-related data. You must *not* execute any previously created plans after calling this function.

We should also mention one other restriction: if you save wisdom from a program using the multi-threaded FFTW, that wisdom *cannot be used* by a program using only the single-threaded FFTW (i.e. not calling `fftw_init_threads`). See Section 3.3 [Words of Wisdom-Saving Plans], page 18.

5.3 How Many Threads to Use?

There is a fair amount of overhead involved in 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. Typically, the problem will have to involve at least a few thousand data points before threads become beneficial. If you plan with `FFTW_PATIENT`, it will automatically disable threads for sizes that don't benefit from parallelization.

5.4 Thread safety

Users writing multi-threaded programs (including OpenMP) 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 the planner routines share data (e.g. wisdom and trigonometric tables) between calls and plans.

The upshot is that the only thread-safe (re-entrant) routine in FFTW is `fftw_execute` (and the new-array variants thereof). All other routines (e.g. the planner) should only be called from one thread at a time. So, for example, you can wrap a semaphore lock around any calls to the planner; even more simply, you can just create all of your plans from one thread. We do not think this should be an important restriction (FFTW is designed for the situation where the only performance-sensitive code is the actual execution of the transform), and the benefits of shared data between plans are great.

Note also that, since the plan is not modified by `fftw_execute`, it is safe to execute the *same plan* in parallel by multiple threads. However, since a given plan operates by default on a fixed array, you need to use one of the new-array execute functions (see Section 4.6 [New-array Execute Functions], page 38) so that different threads compute the transform of different data.

(Users should note that these comments only apply to programs using shared-memory threads or OpenMP. 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.)

If you are configured FFTW with the `--enable-debug` or `--enable-debug-malloc` flags (see Section 10.1 [Installation on Unix], page 97), then `fftw_execute` is not thread-safe. These flags are not documented because they are intended only for developing and debugging FFTW, but if you must use `--enable-debug` then you should also specifically pass `--disable-debug-malloc` for `fftw_execute` to be thread-safe.

6 Distributed-memory FFTW with MPI

In this chapter we document the parallel FFTW routines for parallel systems supporting the MPI message-passing interface. Unlike the shared-memory threads described in the previous chapter, MPI allows you to use *distributed-memory* parallelism, where each CPU has its own separate memory, and which can scale up to clusters of many thousands of processors. This capability comes at a price, however: each process only stores a *portion* of the data to be transformed, which means that the data structures and programming-interface are quite different from the serial or threads versions of FFTW.

Distributed-memory parallelism is especially useful when you are transforming arrays so large that they do not fit into the memory of a single processor. The storage per-process required by FFTW's MPI routines is proportional to the total array size divided by the number of processes. Conversely, distributed-memory parallelism can easily pose an unacceptably high communications overhead for small problems; the threshold problem size for which parallelism becomes advantageous will depend on the precise problem you are interested in, your hardware, and your MPI implementation.

A note on terminology: in MPI, you divide the data among a set of "processes" which each run in their own memory address space. Generally, each process runs on a different physical processor, but this is not required. A set of processes in MPI is described by an opaque data structure called a "communicator," the most common of which is the predefined communicator `MPI_COMM_WORLD` which refers to *all* processes. For more information on these and other concepts common to all MPI programs, we refer the reader to the documentation at the MPI home page.

We assume in this chapter that the reader is familiar with the usage of the serial (uniprocessor) FFTW, and focus only on the concepts new to the MPI interface.

6.1 FFTW MPI Installation

All of the FFTW MPI code is located in the `mpi` subdirectory of the FFTW package. On Unix systems, the FFTW MPI libraries and header files are 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 10.1 [Installation on Unix], page 97).

Any implementation of the MPI standard, version 1 or later, should work with FFTW. The `configure` script will attempt to automatically detect how to compile and link code using your MPI implementation. In some cases, especially if you have multiple different MPI implementations installed or have an unusual MPI software package, you may need to provide this information explicitly.

Most commonly, one compiles MPI code by invoking a special compiler command, typically `mpicc` for C code. The `configure` script knows the most common names for this command, but you can specify the MPI compilation command explicitly by setting the `MPICC` variable, as in '`./configure MPICC=mpicc ...`'.

If, instead of a special compiler command, you need to link a certain library, you can specify the link command via the `MPILIBS` variable, as in '`./configure MPILIBS=-lmpi`

`...`'. Note that if your MPI library is installed in a non-standard location (one the compiler does not know about by default), you may also have to specify the location of the library and header files via `LDFLAGS` and `CPPFLAGS` variables, respectively, as in '`./configure LDFLAGS=-L/path/to/mpi/libs CPPFLAGS=-I/path/to/mpi/include ...`'.

6.2 Linking and Initializing MPI FFTW

Programs using the MPI FFTW routines should be linked with `-lfftw3_mpi -lfftw3 -lm` on Unix in double precision, `-lfftw3f_mpi -lfftw3f -lm` in single precision, and so on (see Section 4.1.2 [Precision], page 21). You will also need to link with whatever library is responsible for MPI on your system; in most MPI implementations, there is a special compiler alias named `mpicc` to compile and link MPI code.

Before calling any FFTW routines except possibly `fftw_init_threads` (see Section 6.11 [Combining MPI and Threads], page 66), but after calling `MPI_Init`, you should call the function:

```
void fftw_mpi_init(void);
```

If, at the end of your program, you want to get rid of all memory and other resources allocated internally by FFTW, for both the serial and MPI routines, you can call:

```
void fftw_mpi_cleanup(void);
```

which is much like the `fftw_cleanup()` function except that it also gets rid of FFTW's MPI-related data. You must *not* execute any previously created plans after calling this function.

6.3 2d MPI example

Before we document the FFTW MPI interface in detail, we begin with a simple example outlining how one would perform a two-dimensional `N0` by `N1` complex DFT.

```
#include <fftw3-mpi.h>

int main(int argc, char **argv)
{
    const ptrdiff_t N0 = ..., N1 = ...;
    fftw_plan plan;
    fftw_complex *data;
    ptrdiff_t alloc_local, local_n0, local_0_start, i, j;

    MPI_Init(&argc, &argv);
    fftw_mpi_init();

    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_2d(N0, N1, MPI_COMM_WORLD,
                                         &local_n0, &local_0_start);
    data = fftw_alloc_complex(alloc_local);

    /* create plan for in-place forward DFT */
```

```
    plan = fftw_mpi_plan_dft_2d(N0, N1, data, data, MPI_COMM_WORLD,
                                FFTW_FORWARD, FFTW_ESTIMATE);

    /* initialize data to some function my_function(x,y) */
    for (i = 0; i < local_n0; ++i) for (j = 0; j < N1; ++j)
       data[i*N1 + j] = my_function(local_0_start + i, j);

    /* compute transforms, in-place, as many times as desired */
    fftw_execute(plan);

    fftw_destroy_plan(plan);

    MPI_Finalize();
}
```

As can be seen above, the MPI interface follows the same basic style of allocate/plan/execute/destroy as the serial FFTW routines. All of the MPI-specific routines are prefixed with '`fftw_mpi_`' instead of '`fftw_`'. There are a few important differences, however:

First, we must call `fftw_mpi_init()` after calling `MPI_Init` (required in all MPI programs) and before calling any other '`fftw_mpi_`' routine.

Second, when we create the plan with `fftw_mpi_plan_dft_2d`, analogous to `fftw_plan_dft_2d`, we pass an additional argument: the communicator, indicating which processes will participate in the transform (here `MPI_COMM_WORLD`, indicating all processes). Whenever you create, execute, or destroy a plan for an MPI transform, you must call the corresponding FFTW routine on *all* processes in the communicator for that transform. (That is, these are *collective* calls.) Note that the plan for the MPI transform uses the standard `fftw_execute` and `fftw_destroy` routines (on the other hand, there are MPI-specific new-array execute functions documented below).

Third, all of the FFTW MPI routines take `ptrdiff_t` arguments instead of `int` as for the serial FFTW. `ptrdiff_t` is a standard C integer type which is (at least) 32 bits wide on a 32-bit machine and 64 bits wide on a 64-bit machine. This is to make it easy to specify very large parallel transforms on a 64-bit machine. (You can specify 64-bit transform sizes in the serial FFTW, too, but only by using the '`guru64`' planner interface. See Section 4.5.6 [64-bit Guru Interface], page 38.)

Fourth, and most importantly, you don't allocate the entire two-dimensional array on each process. Instead, you call `fftw_mpi_local_size_2d` to find out what *portion* of the array resides on each processor, and how much space to allocate. Here, the portion of the array on each process is a `local_n0` by `N1` slice of the total array, starting at index `local_0_start`. The total number of `fftw_complex` numbers to allocate is given by the `alloc_local` return value, which *may* be greater than `local_n0 * N1` (in case some intermediate calculations require additional storage). The data distribution in FFTW's MPI interface is described in more detail by the next section.

Given the portion of the array that resides on the local process, it is straightforward to initialize the data (here to a function `myfunction`) and otherwise manipulate it. Of course,

at the end of the program you may want to output the data somehow, but synchronizing this output is up to you and is beyond the scope of this manual. (One good way to output a large multi-dimensional distributed array in MPI to a portable binary file is to use the free HDF5 library; see the HDF home page.)

6.4 MPI Data Distribution

The most important concept to understand in using FFTW's MPI interface is the data distribution. With a serial or multithreaded FFT, all of the inputs and outputs are stored as a single contiguous chunk of memory. With a distributed-memory FFT, the inputs and outputs are broken into disjoint blocks, one per process.

In particular, FFTW uses a *1d block distribution* of the data, distributed along the *first dimension*. For example, if you want to perform a 100×200 complex DFT, distributed over 4 processes, each process will get a 25×200 slice of the data. That is, process 0 will get rows 0 through 24, process 1 will get rows 25 through 49, process 2 will get rows 50 through 74, and process 3 will get rows 75 through 99. If you take the same array but distribute it over 3 processes, then it is not evenly divisible so the different processes will have unequal chunks. FFTW's default choice in this case is to assign 34 rows to processes 0 and 1, and 32 rows to process 2.

FFTW provides several '`fftw_mpi_local_size`' routines that you can call to find out what portion of an array is stored on the current process. In most cases, you should use the default block sizes picked by FFTW, but it is also possible to specify your own block size. For example, with a 100×200 array on three processes, you can tell FFTW to use a block size of 40, which would assign 40 rows to processes 0 and 1, and 20 rows to process 2. FFTW's default is to divide the data equally among the processes if possible, and as best it can otherwise. The rows are always assigned in "rank order," i.e. process 0 gets the first block of rows, then process 1, and so on. (You can change this by using `MPI_Comm_split` to create a new communicator with re-ordered processes.) However, you should always call the '`fftw_mpi_local_size`' routines, if possible, rather than trying to predict FFTW's distribution choices.

In particular, it is critical that you allocate the storage size that is returned by '`fftw_mpi_local_size`', which is *not* necessarily the size of the local slice of the array. The reason is that intermediate steps of FFTW's algorithms involve transposing the array and redistributing the data, so at these intermediate steps FFTW may require more local storage space (albeit always proportional to the total size divided by the number of processes). The '`fftw_mpi_local_size`' functions know how much storage is required for these intermediate steps and tell you the correct amount to allocate.

6.4.1 Basic and advanced distribution interfaces

As with the planner interface, the '`fftw_mpi_local_size`' distribution interface is broken into basic and advanced ('`_many`') interfaces, where the latter allows you to specify the block size manually and also to request block sizes when computing multiple transforms simultaneously. These functions are documented more exhaustively by the FFTW MPI Reference, but we summarize the basic ideas here using a couple of two-dimensional examples.

For the 100×200 complex-DFT example, above, we would find the distribution by calling the following function in the basic interface:

```
ptrdiff_t fftw_mpi_local_size_2d(ptrdiff_t n0, ptrdiff_t n1, MPI_Comm comm,
                                 ptrdiff_t *local_n0, ptrdiff_t *local_0_start);
```

Given the total size of the data to be transformed (here, `n0 = 100` and `n1 = 200`) and an MPI communicator (`comm`), this function provides three numbers.

First, it describes the shape of the local data: the current process should store a `local_n0` by `n1` slice of the overall dataset, in row-major order (`n1` dimension contiguous), starting at index `local_0_start`. That is, if the total dataset is viewed as a `n0` by `n1` matrix, the current process should store the rows `local_0_start` to `local_0_start+local_n0-1`. Obviously, if you are running with only a single MPI process, that process will store the entire array: `local_0_start` will be zero and `local_n0` will be `n0`. See Section 3.2.1 [Row-major Format], page 15.

Second, the return value is the total number of data elements (e.g., complex numbers for a complex DFT) that should be allocated for the input and output arrays on the current process (ideally with `fftw_malloc` or an '`fftw_alloc`' function, to ensure optimal alignment). It might seem that this should always be equal to `local_n0 * n1`, but this is *not* the case. FFTW's distributed FFT algorithms require data redistributions at intermediate stages of the transform, and in some circumstances this may require slightly larger local storage. This is discussed in more detail below, under Section 6.4.2 [Load balancing], page 58.

The advanced-interface '`local_size`' function for multidimensional transforms returns the same three things (`local_n0`, `local_0_start`, and the total number of elements to allocate), but takes more inputs:

```
ptrdiff_t fftw_mpi_local_size_many(int rnk, const ptrdiff_t *n,
                                   ptrdiff_t howmany,
                                   ptrdiff_t block0,
                                   MPI_Comm comm,
                                   ptrdiff_t *local_n0,
                                   ptrdiff_t *local_0_start);
```

The two-dimensional case above corresponds to `rnk = 2` and an array `n` of length 2 with `n[0] = n0` and `n[1] = n1`. This routine is for any `rnk > 1`; one-dimensional transforms have their own interface because they work slightly differently, as discussed below.

First, the advanced interface allows you to perform multiple transforms at once, of interleaved data, as specified by the `howmany` parameter. (`howmany` is 1 for a single transform.)

Second, here you can specify your desired block size in the `n0` dimension, `block0`. To use FFTW's default block size, pass `FFTW_MPI_DEFAULT_BLOCK` (0) for `block0`. Otherwise, on `P` processes, FFTW will return `local_n0` equal to `block0` on the first `P / block0` processes (rounded down), return `local_n0` equal to `n0 - block0 * (P / block0)` on the next process, and `local_n0` equal to zero on any remaining processes. In general, we recommend using the default block size (which corresponds to `n0 / P`, rounded up).

For example, suppose you have `P = 4` processes and `n0 = 21`. The default will be a block size of 6, which will give `local_n0 = 6` on the first three processes and `local_n0 = 3` on the last process. Instead, however, you could specify `block0 = 5` if you wanted, which would give `local_n0 = 5` on processes 0 to 2, `local_n0 = 6` on process 3. (This choice, while it may look superficially more "balanced," has the same critical path as FFTW's default but requires more communications.)

6.4.2 Load balancing

Ideally, when you parallelize a transform over some P processes, each process should end up with work that takes equal time. Otherwise, all of the processes end up waiting on whichever process is slowest. This goal is known as "load balancing." In this section, we describe the circumstances under which FFTW is able to load-balance well, and in particular how you should choose your transform size in order to load balance.

Load balancing is especially difficult when you are parallelizing over heterogeneous machines; for example, if one of your processors is a old 486 and another is a Pentium IV, obviously you should give the Pentium more work to do than the 486 since the latter is much slower. FFTW does not deal with this problem, however—it assumes that your processes run on hardware of comparable speed, and that the goal is therefore to divide the problem as equally as possible.

For a multi-dimensional complex DFT, FFTW can divide the problem equally among the processes if: (i) the *first* dimension `n0` is divisible by P; and (ii), the *product* of the subsequent dimensions is divisible by P. (For the advanced interface, where you can specify multiple simultaneous transforms via some "vector" length `howmany`, a factor of `howmany` is included in the product of the subsequent dimensions.)

For a one-dimensional complex DFT, the length `N` of the data should be divisible by P *squared* to be able to divide the problem equally among the processes.

6.4.3 Transposed distributions

Internally, FFTW's MPI transform algorithms work by first computing transforms of the data local to each process, then by globally *transposing* the data in some fashion to redistribute the data among the processes, transforming the new data local to each process, and transposing back. For example, a two-dimensional `n0` by `n1` array, distributed across the `n0` dimension, is transformd by: (i) transforming the `n1` dimension, which are local to each process; (ii) transposing to an `n1` by `n0` array, distributed across the `n1` dimension; (iii) transforming the `n0` dimension, which is now local to each process; (iv) transposing back.

However, in many applications it is acceptable to compute a multidimensional DFT whose results are produced in transposed order (e.g., `n1` by `n0` in two dimensions). This provides a significant performance advantage, because it means that the final transposition step can be omitted. FFTW supports this optimization, which you specify by passing the flag `FFTW_MPI_TRANSPOSED_OUT` to the planner routines. To compute the inverse transform of transposed output, you specify `FFTW_MPI_TRANSPOSED_IN` to tell it that the input is transposed. In this section, we explain how to interpret the output format of such a transform.

Suppose you have are transforming multi-dimensional data with (at least two) dimensions $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$. As always, it is distributed along the first dimension n_0 . Now, if we compute its DFT with the `FFTW_MPI_TRANSPOSED_OUT` flag, the resulting output data are stored with the first *two* dimensions transposed: $n_1 \times n_0 \times n_2 \times \cdots \times n_{d-1}$, distributed along the n_1 dimension. Conversely, if we take the $n_1 \times n_0 \times n_2 \times \cdots \times n_{d-1}$ data and transform it with the `FFTW_MPI_TRANSPOSED_IN` flag, then the format goes back to the original $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ array.

There are two ways to find the portion of the transposed array that resides on the current process. First, you can simply call the appropriate '`local_size`' function, passing $n_1 \times n_0 \times n_2 \times \cdots \times n_{d-1}$ (the transposed dimensions). This would mean calling the '`local_size`' function twice, once for the transposed and once for the non-transposed dimensions. Alternatively, you can call one of the '`local_size_transposed`' functions, which returns both the non-transposed and transposed data distribution from a single call. For example, for a 3d transform with transposed output (or input), you might call:

```
ptrdiff_t fftw_mpi_local_size_3d_transposed(
                ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t n2, MPI_Comm comm,
                ptrdiff_t *local_n0, ptrdiff_t *local_0_start,
                ptrdiff_t *local_n1, ptrdiff_t *local_1_start);
```

Here, `local_n0` and `local_0_start` give the size and starting index of the `n0` dimension for the *non*-transposed data, as in the previous sections. For *transposed* data (e.g. the output for `FFTW_MPI_TRANSPOSED_OUT`), `local_n1` and `local_1_start` give the size and starting index of the `n1` dimension, which is the first dimension of the transposed data (`n1` by `n0` by `n2`).

(Note that `FFTW_MPI_TRANSPOSED_IN` is completely equivalent to performing `FFTW_MPI_TRANSPOSED_OUT` and passing the first two dimensions to the planner in reverse order, or vice versa. If you pass *both* the `FFTW_MPI_TRANSPOSED_IN` and `FFTW_MPI_TRANSPOSED_OUT` flags, it is equivalent to swapping the first two dimensions passed to the planner and passing *neither* flag.)

6.4.4 One-dimensional distributions

For one-dimensional distributed DFTs using FFTW, matters are slightly more complicated because the data distribution is more closely tied to how the algorithm works. In particular, you can no longer pass an arbitrary block size and must accept FFTW's default; also, the block sizes may be different for input and output. Also, the data distribution depends on the flags and transform direction, in order for forward and backward transforms to work correctly.

```
ptrdiff_t fftw_mpi_local_size_1d(ptrdiff_t n0, MPI_Comm comm,
                int sign, unsigned flags,
                ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
                ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

This function computes the data distribution for a 1d transform of size `n0` with the given transform `sign` and `flags`. Both input and output data use block distributions. The input on the current process will consist of `local_ni` numbers starting at index `local_i_start`; e.g. if only a single process is used, then `local_ni` will be `n0` and `local_i_start` will be `0`. Similarly for the output, with `local_no` numbers starting at index `local_o_start`. The return value of `fftw_mpi_local_size_1d` will be the total number of elements to allocate on the current process (which might be slightly larger than the local size due to intermediate steps in the algorithm).

As mentioned above (see Section 6.4.2 [Load balancing], page 58), the data will be divided equally among the processes if `n0` is divisible by the *square* of the number of processes. In this case, `local_ni` will equal `local_no`. Otherwise, they may be different.

For some applications, such as convolutions, the order of the output data is irrelevant. In this case, performance can be improved by specifying that the output data be stored in an FFTW-defined "scrambled" format. (In particular, this is the analogue of transposed output in the multidimensional case: scrambled output saves a communications step.) If you pass `FFTW_MPI_SCRAMBLED_OUT` in the flags, then the output is stored in this (undocumented) scrambled order. Conversely, to perform the inverse transform of data in scrambled order, pass the `FFTW_MPI_SCRAMBLED_IN` flag.

In MPI FFTW, only composite sizes `n0` can be parallelized; we have not yet implemented a parallel algorithm for large prime sizes.

6.5 Multi-dimensional MPI DFTs of Real Data

FFTW's MPI interface also supports multi-dimensional DFTs of real data, similar to the serial r2c and c2r interfaces. (Parallel one-dimensional real-data DFTs are not currently supported; you must use a complex transform and set the imaginary parts of the inputs to zero.)

The key points to understand for r2c and c2r MPI transforms (compared to the MPI complex DFTs or the serial r2c/c2r transforms), are:

- Just as for serial transforms, r2c/c2r DFTs transform $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ real data to/from $n_0 \times n_1 \times n_2 \times \cdots \times (n_{d-1}/2+1)$ complex data: the last dimension of the complex data is cut in half (rounded down), plus one. As for the serial transforms, the sizes you pass to the '`plan_dft_r2c`' and '`plan_dft_c2r`' are the $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ dimensions of the real data.
- Although the real data is *conceptually* $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$, it is *physically* stored as an $n_0 \times n_1 \times n_2 \times \cdots \times [2(n_{d-1}/2+1)]$ array, where the last dimension has been *padded* to make it the same size as the complex output. This is much like the in-place serial r2c/c2r interface (see Section 2.4 [Multi-Dimensional DFTs of Real Data], page 7), except that in MPI the padding is required even for out-of-place data. The extra padding numbers are ignored by FFTW (they are *not* like zero-padding the transform to a larger size); they are only used to determine the data layout.
- The data distribution in MPI for *both* the real and complex data is determined by the shape of the *complex* data. That is, you call the appropriate '`local size`' function for the $n_0 \times n_1 \times n_2 \times \cdots \times (n_{d-1}/2+1)$ complex data, and then use the *same* distribution for the real data except that the last complex dimension is replaced by a (padded) real dimension of twice the length.

For example suppose we are performing an out-of-place r2c transform of $L \times M \times N$ real data [padded to $L \times M \times 2(N/2+1)$], resulting in $L \times M \times N/2+1$ complex data. Similar to the example in Section 6.3 [2d MPI example], page 54, we might do something like:

```
#include <fftw3-mpi.h>

int main(int argc, char **argv)
{
    const ptrdiff_t L = ..., M = ..., N = ...;
    fftw_plan plan;
```

```
    double *rin;
    fftw_complex *cout;
    ptrdiff_t alloc_local, local_n0, local_0_start, i, j, k;

    MPI_Init(&argc, &argv);
    fftw_mpi_init();

    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_3d(L, M, N/2+1, MPI_COMM_WORLD,
                                         &local_n0, &local_0_start);
    rin = fftw_alloc_real(2 * alloc_local);
    cout = fftw_alloc_complex(alloc_local);

    /* create plan for out-of-place r2c DFT */
    plan = fftw_mpi_plan_dft_r2c_3d(L, M, N, rin, cout, MPI_COMM_WORLD,
                                    FFTW_MEASURE);

    /* initialize rin to some function my_func(x,y,z) */
    for (i = 0; i < local_n0; ++i)
       for (j = 0; j < M; ++j)
         for (k = 0; k < N; ++k)
       rin[(i*M + j) * (2*(N/2+1)) + k] = my_func(local_0_start+i, j, k);

    /* compute transforms as many times as desired */
    fftw_execute(plan);

    fftw_destroy_plan(plan);

    MPI_Finalize();
}
```

Note that we allocated `rin` using `fftw_alloc_real` with an argument of `2 * alloc_local`: since `alloc_local` is the number of *complex* values to allocate, the number of *real* values is twice as many. The `rin` array is then $local_n 0 \times M \times 2(N/2+1)$ in row-major order, so its `(i,j,k)` element is at the index `(i*M + j) * (2*(N/2+1)) + k` (see ⟨undefined⟩ [Multi-dimensional Array Format], page ⟨undefined⟩).

As for the complex transforms, improved performance can be obtained by specifying that the output is the transpose of the input or vice versa (see Section 6.4.3 [Transposed distributions], page 58). In our $L \times M \times N$ r2c example, including `FFTW_TRANSPOSED_OUT` in the flags means that the input would be a padded $L \times M \times 2(N/2+1)$ real array distributed over the `L` dimension, while the output would be a $M \times L \times N/2+1$ complex array distributed over the `M` dimension. To perform the inverse c2r transform with the same data distributions, you would use the `FFTW_TRANSPOSED_IN` flag.

6.6 Other multi-dimensional Real-Data MPI Transforms

FFTW's MPI interface also supports multi-dimensional '`r2r`' transforms of all kinds supported by the serial interface (e.g. discrete cosine and sine transforms, discrete Hartley transforms, etc.). Only multi-dimensional '`r2r`' transforms, not one-dimensional transforms, are currently parallelized.

These are used much like the multidimensional complex DFTs discussed above, except that the data is real rather than complex, and one needs to pass an r2r transform kind (`fftw_r2r_kind`) for each dimension as in the serial FFTW (see Section 2.5 [More DFTs of Real Data], page 10).

For example, one might perform a two-dimensional $L \times M$ that is an REDFT10 (DCT-II) in the first dimension and an RODFT10 (DST-II) in the second dimension with code like:

```
const ptrdiff_t L = ..., M = ...;
fftw_plan plan;
double *data;
ptrdiff_t alloc_local, local_n0, local_0_start, i, j;

/* get local data size and allocate */
alloc_local = fftw_mpi_local_size_2d(L, M, MPI_COMM_WORLD,
                                     &local_n0, &local_0_start);
data = fftw_alloc_real(alloc_local);

/* create plan for in-place REDFT10 x RODFT10 */
plan = fftw_mpi_plan_r2r_2d(L, M, data, data, MPI_COMM_WORLD,
                            FFTW_REDFT10, FFTW_RODFT10, FFTW_MEASURE);

/* initialize data to some function my_function(x,y) */
for (i = 0; i < local_n0; ++i) for (j = 0; j < M; ++j)
   data[i*M + j] = my_function(local_0_start + i, j);

/* compute transforms, in-place, as many times as desired */
fftw_execute(plan);

fftw_destroy_plan(plan);
```

Notice that we use the same '`local_size`' functions as we did for complex data, only now we interpret the sizes in terms of real rather than complex values, and correspondingly use `fftw_alloc_real`.

6.7 FFTW MPI Transposes

The FFTW's MPI Fourier transforms rely on one or more *global transposition* step for their communications. For example, the multidimensional transforms work by transforming along some dimensions, then transposing to make the first dimension local and transforming that, then transposing back. Because global transposition of a block-distributed matrix has many other potential uses besides FFTs, FFTW's transpose routines can be called directly, as documented in this section.

6.7.1 Basic distributed-transpose interface

In particular, suppose that we have an `n0` by `n1` array in row-major order, block-distributed across the `n0` dimension. To transpose this into an `n1` by `n0` array block-distributed across the `n1` dimension, we would create a plan by calling the following function:

```
fftw_plan fftw_mpi_plan_transpose(ptrdiff_t n0, ptrdiff_t n1,
                                  double *in, double *out,
                                  MPI_Comm comm, unsigned flags);
```

The input and output arrays (`in` and `out`) can be the same. The transpose is actually executed by calling `fftw_execute` on the plan, as usual.

The `flags` are the usual FFTW planner flags, but support two additional flags: `FFTW_MPI_TRANSPOSED_OUT` and/or `FFTW_MPI_TRANSPOSED_IN`. What these flags indicate, for transpose plans, is that the output and/or input, respectively, are *locally* transposed. That is, on each process input data is normally stored as a `local_n0` by `n1` array in row-major order, but for an `FFTW_MPI_TRANSPOSED_IN` plan the input data is stored as `n1` by `local_n0` in row-major order. Similarly, `FFTW_MPI_TRANSPOSED_OUT` means that the output is `n0` by `local_n1` instead of `local_n1` by `n0`.

To determine the local size of the array on each process before and after the transpose, as well as the amount of storage that must be allocated, one should call `fftw_mpi_local_size_2d_transposed`, just as for a 2d DFT as described in the previous section:

```
ptrdiff_t fftw_mpi_local_size_2d_transposed
                (ptrdiff_t n0, ptrdiff_t n1, MPI_Comm comm,
                 ptrdiff_t *local_n0, ptrdiff_t *local_0_start,
                 ptrdiff_t *local_n1, ptrdiff_t *local_1_start);
```

Again, the return value is the local storage to allocate, which in this case is the number of *real* (`double`) values rather than complex numbers as in the previous examples.

6.7.2 Advanced distributed-transpose interface

The above routines are for a transpose of a matrix of numbers (of type `double`), using FFTW's default block sizes. More generally, one can perform transposes of *tuples* of numbers, with user-specified block sizes for the input and output:

```
fftw_plan fftw_mpi_plan_many_transpose
                (ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t howmany,
                 ptrdiff_t block0, ptrdiff_t block1,
                 double *in, double *out, MPI_Comm comm, unsigned flags);
```

In this case, one is transposing an `n0` by `n1` matrix of `howmany`-tuples (e.g. `howmany = 2` for complex numbers). The input is distributed along the `n0` dimension with block size `block0`, and the `n1` by `n0` output is distributed along the `n1` dimension with block size `block1`. If `FFTW_MPI_DEFAULT_BLOCK` (0) is passed for a block size then FFTW uses its default block size. To get the local size of the data on each process, you should then call `fftw_mpi_local_size_many_transposed`.

6.7.3 An improved replacement for MPI_Alltoall

We close this section by noting that FFTW's MPI transpose routines can be thought of as a generalization for the `MPI_Alltoall` function (albeit only for floating-point types), and in some circumstances can function as an improved replacement.

`MPI_Alltoall` is defined by the MPI standard as:

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                 MPI_Comm comm);
```

In particular, for `double*` arrays `in` and `out`, consider the call:

```
MPI_Alltoall(in, howmany, MPI_DOUBLE, out, howmany MPI_DOUBLE, comm);
```

This is completely equivalent to:

```
MPI_Comm_size(comm, &P);
plan = fftw_mpi_plan_many_transpose(P, P, howmany, 1, 1, in, out, comm, FFTW_ESTIMATE)
fftw_execute(plan);
fftw_destroy_plan(plan);
```

That is, computing a $P \times P$ transpose on `P` processes, with a block size of 1, is just a standard all-to-all communication.

However, using the FFTW routine instead of `MPI_Alltoall` may have certain advantages. First of all, FFTW's routine can operate in-place (`in == out`) whereas `MPI_Alltoall` can only operate out-of-place.

Second, even for out-of-place plans, FFTW's routine may be faster, especially if you need to perform the all-to-all communication many times and can afford to use `FFTW_MEASURE` or `FFTW_PATIENT`. It should certainly be no slower, not including the time to create the plan, since one of the possible algorithms that FFTW uses for an out-of-place transpose *is* simply to call `MPI_Alltoall`. However, FFTW also considers several other possible algorithms that, depending on your MPI implementation and your hardware, may be faster.

6.8 FFTW MPI Wisdom

FFTW's "wisdom" facility (see Section 3.3 [Words of Wisdom-Saving Plans], page 18) can be used to save MPI plans as well as to save uniprocessor plans. However, for MPI there are several unavoidable complications.

First, the MPI standard does not guarantee that every process can perform file I/O (at least, not using C stdio routines)—in general, we may only assume that process 0 is capable of I/O.[1] So, if we want to export the wisdom from a single process to a file, we must first export the wisdom to a string, then send it to process 0, then write it to a file.

[1] In fact, even this assumption is not technically guaranteed by the standard, although it seems to be universal in actual MPI implementations and is widely assumed by MPI-using software. Technically, you need to query the `MPI_IO` attribute of `MPI_COMM_WORLD` with `MPI_Attr_get`. If this attribute is `MPI_PROC_NULL`, no I/O is possible. If it is `MPI_ANY_SOURCE`, any process can perform I/O. Otherwise, it is the rank of a process that can perform I/O ... but since it is not guaranteed to yield the *same* rank on all processes, you have to do an `MPI_Allreduce` of some kind if you want all processes to agree about which is going to do I/O. And even then, the standard only guarantees that this process can perform output, but not input. See e.g. *Parallel Programming with MPI* by P. S. Pacheco, section 8.1.3. Needless to say, in our experience virtually no MPI programmers worry about this.

Second, in principle we may want to have separate wisdom for every process, since in general the processes may run on different hardware even for a single MPI program. However, in practice FFTW's MPI code is designed for the case of homogeneous hardware (see Section 6.4.2 [Load balancing], page 58), and in this case it is convenient to use the same wisdom for every process. Thus, we need a mechanism to synchronize the wisdom.

To address both of these problems, FFTW provides the following two functions:

```
void fftw_mpi_broadcast_wisdom(MPI_Comm comm);
void fftw_mpi_gather_wisdom(MPI_Comm comm);
```

Given a communicator `comm`, `fftw_mpi_broadcast_wisdom` will broadcast the wisdom from process 0 to all other processes. Conversely, `fftw_mpi_gather_wisdom` will collect wisdom from all processes onto process 0. (If the plans created for the same problem by different processes are not the same, `fftw_mpi_gather_wisdom` will arbitrarily choose one of the plans.) Both of these functions may result in suboptimal plans for different processes if the processes are running on non-identical hardware. Both of these functions are *collective* calls, which means that they must be executed by all processes in the communicator.

So, for example, a typical code snippet to import wisdom from a file and use it on all processes would be:

```
{
    int rank;

    fftw_mpi_init();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) fftw_import_wisdom_from_filename("mywisdom");
    fftw_mpi_broadcast_wisdom(MPI_COMM_WORLD);
}
```

(Note that we must call `fftw_mpi_init` before importing any wisdom that might contain MPI plans.) Similarly, a typical code snippet to export wisdom from all processes to a file is:

```
{
    int rank;

    fftw_mpi_gather_wisdom(MPI_COMM_WORLD);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) fftw_export_wisdom_to_filename("mywisdom");
}
```

6.9 Avoiding MPI Deadlocks

An MPI program can *deadlock* if one process is waiting for a message from another process that never gets sent. To avoid deadlocks when using FFTW's MPI routines, it is important to know which functions are *collective*: that is, which functions must *always* be called in the *same order* from *every* process in a given communicator. (For example, `MPI_Barrier` is the canonical example of a collective function in the MPI standard.)

The functions in FFTW that are *always* collective are: every function beginning with '`fftw_mpi_plan`', as well as `fftw_mpi_broadcast_wisdom` and `fftw_mpi_gather_wisdom`.

Also, the following functions from the ordinary FFTW interface are collective when they are applied to a plan created by an '`fftw_mpi_plan`' function: `fftw_execute`, `fftw_destroy_plan`, and `fftw_flops`.

6.10 FFTW MPI Performance Tips

In this section, we collect a few tips on getting the best performance out of FFTW's MPI transforms.

First, because of the 1d block distribution, FFTW's parallelization is currently limited by the size of the first dimension. (Multidimensional block distributions may be supported by a future version.) More generally, you should ideally arrange the dimensions so that FFTW can divide them equally among the processes. See Section 6.4.2 [Load balancing], page 58.

Second, if it is not too inconvenient, you should consider working with transposed output for multidimensional plans, as this saves a considerable amount of communications. See Section 6.4.3 [Transposed distributions], page 58.

Third, the fastest choices are generally either an in-place transform or an out-of-place transform with the `FFTW_DESTROY_INPUT` flag (which allows the input array to be used as scratch space). In-place is especially beneficial if the amount of data per process is large.

Fourth, if you have multiple arrays to transform at once, rather than calling FFTW's MPI transforms several times it usually seems to be faster to interleave the data and use the advanced interface. (This groups the communications together instead of requiring separate messages for each transform.)

6.11 Combining MPI and Threads

In certain cases, it may be advantageous to combine MPI (distributed-memory) and threads (shared-memory) parallelization. FFTW supports this, with certain caveats. For example, if you have a cluster of 4-processor shared-memory nodes, you may want to use threads within the nodes and MPI between the nodes, instead of MPI for all parallelization.

In particular, it is possible to seamlessly combine the MPI FFTW routines with the multi-threaded FFTW routines (see Chapter 5 [Multi-threaded FFTW], page 49). However, some care must be taken in the initialization code, which should look something like this:

```
int threads_ok;

int main(int argc, char **argv)
{
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    threads_ok = provided >= MPI_THREAD_FUNNELED;

    if (threads_ok) threads_ok = fftw_init_threads();
    fftw_mpi_init();

    ...
    if (threads_ok) fftw_plan_with_nthreads(...);
```

```
    ...

    MPI_Finalize();
}
```

First, note that instead of calling `MPI_Init`, you should call `MPI_Init_threads`, which is the initialization routine defined by the MPI-2 standard to indicate to MPI that your program will be multithreaded. We pass `MPI_THREAD_FUNNELED`, which indicates that we will only call MPI routines from the main thread. (FFTW will launch additional threads internally, but the extra threads will not call MPI code.) (You may also pass `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`, which requests additional multithreading support from the MPI implementation, but this is not required by FFTW.) The `provided` parameter returns what level of threads support is actually supported by your MPI implementation; this *must* be at least `MPI_THREAD_FUNNELED` if you want to call the FFTW threads routines, so we define a global variable `threads_ok` to record this. You should only call `fftw_init_threads` or `fftw_plan_with_nthreads` if `threads_ok` is true. For more information on thread safety in MPI, see the MPI and Threads section of the MPI-2 standard.

Second, we must call `fftw_init_threads` *before* `fftw_mpi_init`. This is critical for technical reasons having to do with how FFTW initializes its list of algorithms.

Then, if you call `fftw_plan_with_nthreads(N)`, *every* MPI process will launch (up to) `N` threads to parallelize its transforms.

For example, in the hypothetical cluster of 4-processor nodes, you might wish to launch only a single MPI process per node, and then call `fftw_plan_with_nthreads(4)` on each process to use all processors in the nodes.

This may or may not be faster than simply using as many MPI processes as you have processors, however. On the one hand, using threads within a node eliminates the need for explicit message passing within the node. On the other hand, FFTW's transpose routines are not multi-threaded, and this means that the communications that do take place will not benefit from parallelization within the node. Moreover, many MPI implementations already have optimizations to exploit shared memory when it is available, so adding the multithreaded FFTW on top of this may be superfluous.

6.12 FFTW MPI Reference

This chapter provides a complete reference to all FFTW MPI functions, datatypes, and constants. See also Chapter 4 [FFTW Reference], page 21 for information on functions and types in common with the serial interface.

6.12.1 MPI Files and Data Types

All programs using FFTW's MPI support should include its header file:

```
#include <fftw3-mpi.h>
```

Note that this header file includes the serial-FFTW `fftw3.h` header file, and also the `mpi.h` header file for MPI, so you need not include those files separately.

You must also link to *both* the FFTW MPI library and to the serial FFTW library. On Unix, this means adding `-lfftw3_mpi -lfftw3 -lm` at the end of the link command.

Different precisions are handled as in the serial interface: See Section 4.1.2 [Precision], page 21. That is, '`fftw_`' functions become `fftwf_` (in single precision) etcetera, and the libraries become `-lfftw3f_mpi -lfftw3f -lm` etcetera on Unix. Long-double precision is supported in MPI, but quad precision ('`fftwq_`') is not due to the lack of MPI support for this type.

6.12.2 MPI Initialization

Before calling any other FFTW MPI ('`fftw_mpi_`') function, and before importing any wisdom for MPI problems, you must call:

```
void fftw_mpi_init(void);
```

If FFTW threads support is used, however, `fftw_mpi_init` should be called *after* `fftw_init_threads` (see Section 6.11 [Combining MPI and Threads], page 66). Calling `fftw_mpi_init` additional times (before `fftw_mpi_cleanup`) has no effect.

If you want to deallocate all persistent data and reset FFTW to the pristine state it was in when you started your program, you can call:

```
void fftw_mpi_cleanup(void);
```

(This calls `fftw_cleanup`, so you need not call the serial cleanup routine too, although it is safe to do so.) After calling `fftw_mpi_cleanup`, all existing plans become undefined, and you should not attempt to execute or destroy them. You must call `fftw_mpi_init` again after `fftw_mpi_cleanup` if you want to resume using the MPI FFTW routines.

6.12.3 Using MPI Plans

Once an MPI plan is created, you can execute and destroy it using `fftw_execute`, `fftw_destroy_plan`, and the other functions in the serial interface that operate on generic plans (see Section 4.2 [Using Plans], page 22).

The `fftw_execute` and `fftw_destroy_plan` functions, applied to MPI plans, are *collective* calls: they must be called for all processes in the communicator that was used to create the plan.

You must *not* use the serial new-array plan-execution functions `fftw_execute_dft` and so on (see Section 4.6 [New-array Execute Functions], page 38) with MPI plans. Such functions are specialized to the problem type, and there are specific new-array execute functions for MPI plans:

```
void fftw_mpi_execute_dft(fftw_plan p, fftw_complex *in, fftw_complex *out);
void fftw_mpi_execute_dft_r2c(fftw_plan p, double *in, fftw_complex *out);
void fftw_mpi_execute_dft_c2r(fftw_plan p, fftw_complex *in, double *out);
void fftw_mpi_execute_r2r(fftw_plan p, double *in, double *out);
```

These functions have the same restrictions as those of the serial new-array execute functions. They are *always* safe to apply to the *same* `in` and `out` arrays that were used to create the plan. They can only be applied to new arrarys if those arrays have the same types, dimensions, in-placeness, and alignment as the original arrays, where the best way to ensure the same alignment is to use FFTW's `fftw_malloc` and related allocation functions for all arrays (see Section 4.1.3 [Memory Allocation], page 22). Note that distributed transposes (see Section 6.7 [FFTW MPI Transposes], page 62) use `fftw_mpi_execute_r2r`, since they count as rank-zero r2r plans from FFTW's perspective.

6.12.4 MPI Data Distribution Functions

As described above (see Section 6.4 [MPI Data Distribution], page 56), in order to allocate your arrays, *before* creating a plan, you must first call one of the following routines to determine the required allocation size and the portion of the array locally stored on a given process. The `MPI_Comm` communicator passed here must be equivalent to the communicator used below for plan creation.

The basic interface for multidimensional transforms consists of the functions:

```
ptrdiff_t fftw_mpi_local_size_2d(ptrdiff_t n0, ptrdiff_t n1, MPI_Comm comm,
                                 ptrdiff_t *local_n0, ptrdiff_t *local_0_start);
ptrdiff_t fftw_mpi_local_size_3d(ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t n2,
                                 MPI_Comm comm,
                                 ptrdiff_t *local_n0, ptrdiff_t *local_0_start);
ptrdiff_t fftw_mpi_local_size(int rnk, const ptrdiff_t *n, MPI_Comm comm,
                              ptrdiff_t *local_n0, ptrdiff_t *local_0_start);

ptrdiff_t fftw_mpi_local_size_2d_transposed(ptrdiff_t n0, ptrdiff_t n1, MPI_Comm comm,
                                            ptrdiff_t *local_n0, ptrdiff_t *local_0_st
                                            ptrdiff_t *local_n1, ptrdiff_t *local_1_st
ptrdiff_t fftw_mpi_local_size_3d_transposed(ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t n2,
                                            MPI_Comm comm,
                                            ptrdiff_t *local_n0, ptrdiff_t *local_0_st
                                            ptrdiff_t *local_n1, ptrdiff_t *local_1_st
ptrdiff_t fftw_mpi_local_size_transposed(int rnk, const ptrdiff_t *n, MPI_Comm comm,
                                         ptrdiff_t *local_n0, ptrdiff_t *local_0_start
                                         ptrdiff_t *local_n1, ptrdiff_t *local_1_start
```

These functions return the number of elements to allocate (complex numbers for DFT/r2c/c2r plans, real numbers for r2r plans), whereas the `local_n0` and `local_0_start` return the portion (`local_0_start` to `local_0_start + local_n0 - 1`) of the first dimension of an $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ array that is stored on the local process. See Section 6.4.1 [Basic and advanced distribution interfaces], page 56. For `FFTW_MPI_TRANSPOSED_OUT` plans, the '`_transposed`' variants are useful in order to also return the local portion of the first dimension in the $n_1 \times n_0 \times n_2 \times \cdots \times n_{d-1}$ transposed output. See Section 6.4.3 [Transposed distributions], page 58. The advanced interface for multidimensional transforms is:

```
ptrdiff_t fftw_mpi_local_size_many(int rnk, const ptrdiff_t *n, ptrdiff_t howmany,
                                   ptrdiff_t block0, MPI_Comm comm,
                                   ptrdiff_t *local_n0, ptrdiff_t *local_0_start);
ptrdiff_t fftw_mpi_local_size_many_transposed(int rnk, const ptrdiff_t *n, ptrdiff_t h
                                              ptrdiff_t block0, ptrdiff_t block1, MPI_
                                              ptrdiff_t *local_n0, ptrdiff_t *local_0_
                                              ptrdiff_t *local_n1, ptrdiff_t *local_1_
```

These differ from the basic interface in only two ways. First, they allow you to specify block sizes `block0` and `block1` (the latter for the transposed output); you can pass `FFTW_MPI_DEFAULT_BLOCK` to use FFTW's default block size as in the basic interface. Second, you

can pass a `howmany` parameter, corresponding to the advanced planning interface below: this is for transforms of contiguous `howmany`-tuples of numbers (`howmany = 1` in the basic interface).

The corresponding basic and advanced routines for one-dimensional transforms (currently only complex DFTs) are:

```
ptrdiff_t fftw_mpi_local_size_1d(
             ptrdiff_t n0, MPI_Comm comm, int sign, unsigned flags,
             ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
             ptrdiff_t *local_no, ptrdiff_t *local_o_start);
ptrdiff_t fftw_mpi_local_size_many_1d(
             ptrdiff_t n0, ptrdiff_t howmany,
             MPI_Comm comm, int sign, unsigned flags,
             ptrdiff_t *local_ni, ptrdiff_t *local_i_start,
             ptrdiff_t *local_no, ptrdiff_t *local_o_start);
```

As above, the return value is the number of elements to allocate (complex numbers, for complex DFTs). The `local_ni` and `local_i_start` arguments return the portion (`local_i_start` to `local_i_start + local_ni - 1`) of the 1d array that is stored on this process for the transform *input*, and `local_no` and `local_o_start` are the corresponding quantities for the input. The `sign` (`FFTW_FORWARD` or `FFTW_BACKWARD`) and `flags` must match the arguments passed when creating a plan. Although the inputs and outputs have different data distributions in general, it is guaranteed that the *output* data distribution of an `FFTW_FORWARD` plan will match the *input* data distribution of an `FFTW_BACKWARD` plan and vice versa; similarly for the `FFTW_MPI_SCRAMBLED_OUT` and `FFTW_MPI_SCRAMBLED_IN` flags. See Section 6.4.4 [One-dimensional distributions], page 59.

6.12.5 MPI Plan Creation

Complex-data MPI DFTs

Plans for complex-data DFTs (see Section 6.3 [2d MPI example], page 54) are created by:

```
fftw_plan fftw_mpi_plan_dft_1d(ptrdiff_t n0, fftw_complex *in, fftw_complex *out,
                               MPI_Comm comm, int sign, unsigned flags);
fftw_plan fftw_mpi_plan_dft_2d(ptrdiff_t n0, ptrdiff_t n1,
                               fftw_complex *in, fftw_complex *out,
                               MPI_Comm comm, int sign, unsigned flags);
fftw_plan fftw_mpi_plan_dft_3d(ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t n2,
                               fftw_complex *in, fftw_complex *out,
                               MPI_Comm comm, int sign, unsigned flags);
fftw_plan fftw_mpi_plan_dft(int rnk, const ptrdiff_t *n,
                            fftw_complex *in, fftw_complex *out,
                            MPI_Comm comm, int sign, unsigned flags);
fftw_plan fftw_mpi_plan_many_dft(int rnk, const ptrdiff_t *n,
                                 ptrdiff_t howmany, ptrdiff_t block, ptrdiff_t tblock,
                                 fftw_complex *in, fftw_complex *out,
                                 MPI_Comm comm, int sign, unsigned flags);
```

These are similar to their serial counterparts (see Section 4.3.1 [Complex DFTs], page 24) in specifying the dimensions, sign, and flags of the transform. The `comm` argument gives an MPI communicator that specifies the set of processes to participate in the transform; plan creation is a collective function that must be called for all processes in the communicator. The `in` and `out` pointers refer only to a portion of the overall transform data (see Section 6.4 [MPI Data Distribution], page 56) as specified by the '`local_size`' functions in the previous section. Unless `flags` contains `FFTW_ESTIMATE`, these arrays are overwritten during plan creation as for the serial interface. For multi-dimensional transforms, any dimensions `> 1` are supported; for one-dimensional transforms, only composite (non-prime) `n0` are currently supported (unlike the serial FFTW). Requesting an unsupported transform size will yield a `NULL` plan. (As in the serial interface, highly composite sizes generally yield the best performance.)

The advanced-interface `fftw_mpi_plan_many_dft` additionally allows you to specify the block sizes for the first dimension (`block`) of the $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ input data and the first dimension (`tblock`) of the $n_1 \times n_0 \times n_2 \times \cdots \times n_{d-1}$ transposed data (at intermediate steps of the transform, and for the output if `FFTW_TRANSPOSED_OUT` is specified in `flags`). These must be the same block sizes as were passed to the corresponding '`local_size`' function; you can pass `FFTW_MPI_DEFAULT_BLOCK` to use FFTW's default block size as in the basic interface. Also, the `howmany` parameter specifies that the transform is of contiguous `howmany`-tuples rather than individual complex numbers; this corresponds to the same parameter in the serial advanced interface (see Section 4.4.1 [Advanced Complex DFTs], page 31) with `stride = howmany` and `dist = 1`.

MPI flags

The `flags` can be any of those for the serial FFTW (see Section 4.3.2 [Planner Flags], page 25), and in addition may include one or more of the following MPI-specific flags, which improve performance at the cost of changing the output or input data formats.

- `FFTW_MPI_SCRAMBLED_OUT`, `FFTW_MPI_SCRAMBLED_IN`: valid for 1d transforms only, these flags indicate that the output/input of the transform are in an undocumented "scrambled" order. A forward `FFTW_MPI_SCRAMBLED_OUT` transform can be inverted by a backward `FFTW_MPI_SCRAMBLED_IN` (times the usual $1/N$ normalization). See Section 6.4.4 [One-dimensional distributions], page 59.
- `FFTW_MPI_TRANSPOSED_OUT`, `FFTW_MPI_TRANSPOSED_IN`: valid for multidimensional (`rnk > 1`) transforms only, these flags specify that the output or input of an $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ transform is transposed to $n_1 \times n_0 \times n_2 \times \cdots \times n_{d-1}$. See Section 6.4.3 [Transposed distributions], page 58.

Real-data MPI DFTs

Plans for real-input/output (r2c/c2r) DFTs (see Section 6.5 [Multi-dimensional MPI DFTs of Real Data], page 60) are created by:

```
fftw_plan fftw_mpi_plan_dft_r2c_2d(ptrdiff_t n0, ptrdiff_t n1,
                                   double *in, fftw_complex *out,
                                   MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_dft_r2c_2d(ptrdiff_t n0, ptrdiff_t n1,
                                   double *in, fftw_complex *out,
```

```
                                     MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_dft_r2c_3d(ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t n2,
                                     double *in, fftw_complex *out,
                                     MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_dft_r2c(int rnk, const ptrdiff_t *n,
                                double *in, fftw_complex *out,
                                MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_dft_c2r_2d(ptrdiff_t n0, ptrdiff_t n1,
                                     fftw_complex *in, double *out,
                                     MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_dft_c2r_2d(ptrdiff_t n0, ptrdiff_t n1,
                                     fftw_complex *in, double *out,
                                     MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_dft_c2r_3d(ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t n2,
                                     fftw_complex *in, double *out,
                                     MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_dft_c2r(int rnk, const ptrdiff_t *n,
                                fftw_complex *in, double *out,
                                MPI_Comm comm, unsigned flags);
```

Similar to the serial interface (see Section 4.3.3 [Real-data DFTs], page 27), these transform logically $n_0 \times n_1 \times n_2 \times \cdots \times n_{d-1}$ real data to/from $n_0 \times n_1 \times n_2 \times \cdots \times (n_{d-1}/2+1)$ complex data, representing the non-redundant half of the conjugate-symmetry output of a real-input DFT (see Section 4.8.6 [Multi-dimensional Transforms], page 46). However, the real array must be stored within a padded $n_0 \times n_1 \times n_2 \times \cdots \times [2(n_{d-1}/2+1)]$ array (much like the in-place serial r2c transforms, but here for out-of-place transforms as well). Currently, only multi-dimensional (`rnk > 1`) r2c/c2r transforms are supported (requesting a plan for `rnk = 1` will yield `NULL`). As explained above (see Section 6.5 [Multi-dimensional MPI DFTs of Real Data], page 60), the data distribution of both the real and complex arrays is given by the '`local_size`' function called for the dimensions of the *complex* array. Similar to the other planning functions, the input and output arrays are overwritten when the plan is created except in `FFTW_ESTIMATE` mode.

As for the complex DFTs above, there is an advance interface that allows you to manually specify block sizes and to transform contiguous `howmany`-tuples of real/complex numbers:

```
fftw_plan fftw_mpi_plan_many_dft_r2c
              (int rnk, const ptrdiff_t *n, ptrdiff_t howmany,
               ptrdiff_t iblock, ptrdiff_t oblock,
               double *in, fftw_complex *out,
               MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_many_dft_c2r
              (int rnk, const ptrdiff_t *n, ptrdiff_t howmany,
               ptrdiff_t iblock, ptrdiff_t oblock,
               fftw_complex *in, double *out,
               MPI_Comm comm, unsigned flags);
```

MPI r2r transforms

There are corresponding plan-creation routines for r2r transforms (see Section 2.5 [More DFTs of Real Data], page 10), currently supporting multidimensional (`rnk > 1`) transforms only (`rnk = 1` will yield a `NULL` plan):

```
fftw_plan fftw_mpi_plan_r2r_2d(ptrdiff_t n0, ptrdiff_t n1,
                               double *in, double *out,
                               MPI_Comm comm,
                               fftw_r2r_kind kind0, fftw_r2r_kind kind1,
                               unsigned flags);
fftw_plan fftw_mpi_plan_r2r_3d(ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t n2,
                               double *in, double *out,
                               MPI_Comm comm,
                               fftw_r2r_kind kind0, fftw_r2r_kind kind1, fftw_r2r_kind
                               unsigned flags);
fftw_plan fftw_mpi_plan_r2r(int rnk, const ptrdiff_t *n,
                            double *in, double *out,
                            MPI_Comm comm, const fftw_r2r_kind *kind,
                            unsigned flags);
fftw_plan fftw_mpi_plan_many_r2r(int rnk, const ptrdiff_t *n,
                                 ptrdiff_t iblock, ptrdiff_t oblock,
                                 double *in, double *out,
                                 MPI_Comm comm, const fftw_r2r_kind *kind,
                                 unsigned flags);
```

The parameters are much the same as for the complex DFTs above, except that the arrays are of real numbers (and hence the outputs of the '`local_size`' data-distribution functions should be interpreted as counts of real rather than complex numbers). Also, the `kind` parameters specify the r2r kinds along each dimension as for the serial interface (see Section 4.3.6 [Real-to-Real Transform Kinds], page 30). See Section 6.6 [Other Multi-dimensional Real-data MPI Transforms], page 62.

MPI transposition

FFTW also provides routines to plan a transpose of a distributed `n0` by `n1` array of real numbers, or an array of `howmany`-tuples of real numbers with specified block sizes (see Section 6.7 [FFTW MPI Transposes], page 62):

```
fftw_plan fftw_mpi_plan_transpose(ptrdiff_t n0, ptrdiff_t n1,
                                  double *in, double *out,
                                  MPI_Comm comm, unsigned flags);
fftw_plan fftw_mpi_plan_many_transpose
                (ptrdiff_t n0, ptrdiff_t n1, ptrdiff_t howmany,
                 ptrdiff_t block0, ptrdiff_t block1,
                 double *in, double *out, MPI_Comm comm, unsigned flags);
```

These plans are used with the `fftw_mpi_execute_r2r` new-array execute function (see ⟨undefined⟩ [Using MPI Plans], page ⟨undefined⟩), since they count as (rank zero) r2r plans from FFTW's perspective.

6.12.6 MPI Wisdom Communication

To facilitate synchronizing wisdom among the different MPI processes, we provide two functions:

```
void fftw_mpi_gather_wisdom(MPI_Comm comm);
void fftw_mpi_broadcast_wisdom(MPI_Comm comm);
```

The `fftw_mpi_gather_wisdom` function gathers all wisdom in the given communicator `comm` to the process of rank 0 in the communicator: that process obtains the union of all wisdom on all the processes. As a side effect, some other processes will gain additional wisdom from other processes, but only process 0 will gain the complete union.

The `fftw_mpi_broadcast_wisdom` does the reverse: it exports wisdom from process 0 in `comm` to all other processes in the communicator, replacing any wisdom they currently have.

See Section 6.8 [FFTW MPI Wisdom], page 64.

6.13 FFTW MPI Fortran Interface

The FFTW MPI interface is callable from modern Fortran compilers supporting the Fortran 2003 `iso_c_binding` standard for calling C functions. As described in Chapter 7 [Calling FFTW from Modern Fortran], page 77, this means that you can directly call FFTW's C interface from Fortran with only minor changes in syntax. There are, however, a few things specific to the MPI interface to keep in mind:

- Instead of including `fftw3.f03` as in ⟨undefined⟩ [Overview of Fortran interface], page ⟨undefined⟩, you should `include 'fftw3-mpi.f03'` (after `use, intrinsic :: iso_c_binding` as before). The `fftw3-mpi.f03` file includes `fftw3.f03`, so you should *not* `include` them both yourself. (You will also want to include the MPI header file, usually via `include 'mpif.h'` or similar, although though this is not needed by `fftw3-mpi.f03` *per se*.) (To use the '`fftwl_`' `long double` extended-precision routines in supporting compilers, you should include `fftw3f-mpi.f03` in *addition* to `fftw3-mpi.f03`. See Section 7.1.1 [Extended and quadruple precision in Fortran], page 78.)
- Because of the different storage conventions between C and Fortran, you reverse the order of your array dimensions when passing them to FFTW (see Section 7.2 [Reversing array dimensions], page 78). This is merely a difference in notation and incurs no performance overhead. However, it means that, whereas in C the *first* dimension is distributed, in Fortran the *last* dimension of your array is distributed.
- In Fortran, communicators are stored as `integer` types; there is no `MPI_Comm` type, nor is there any way to access a C `MPI_Comm`. Fortunately, this is taken care of for you by the FFTW Fortran interface: whenever the C interface expects an `MPI_Comm` type, you should pass the Fortran communicator as an `integer`.[2]
- Because you need to call the '`local_size`' function to find out how much space to allocate, and this may be *larger* than the local portion of the array (see Section 6.4 [MPI Data Distribution], page 56), you should *always* allocate your arrays dynamically

[2] Technically, this is because you aren't actually calling the C functions directly. You are calling wrapper functions that translate the communicator with `MPI_Comm_f2c` before calling the ordinary C interface. This is all done transparently, however, since the `fftw3-mpi.f03` interface file renames the wrappers so that they are called in Fortran with the same names as the C interface functions.

using FFTW's allocation routines as described in Section 7.5 [Allocating aligned memory in Fortran], page 82. (Coincidentally, this also provides the best performance by guaranteeing proper data alignment.)

- Because all sizes in the MPI FFTW interface are declared as `ptrdiff_t` in C, you should use `integer(C_INTPTR_T)` in Fortran (see Section 7.3 [FFTW Fortran type reference], page 80).
- In Fortran, because of the language semantics, we generally recommend using the new-array execute functions for all plans, even in the common case where you are executing the plan on the same arrays for which the plan was created (see Section 7.4 [Plan execution in Fortran], page 81). However, note that in the MPI interface these functions are changed: `fftw_execute_dft` becomes `fftw_mpi_execute_dft`, etcetera. See Section 6.12.3 [Using MPI Plans], page 68.

For example, here is a Fortran code snippet to perform a distributed $L \times M$ complex DFT in-place. (This assumes you have already initialized MPI with `MPI_init` and have also performed `call fftw_mpi_init`.)

```
use, intrinsic :: iso_c_binding
include 'fftw3-mpi.f03'
integer(C_INTPTR_T), parameter :: L = ...
integer(C_INTPTR_T), parameter :: M = ...
type(C_PTR) :: plan, cdata
complex(C_DOUBLE_COMPLEX), pointer :: data(:,:)
integer(C_INTPTR_T) :: i, j, alloc_local, local_M, local_j_offset

!   get local data size and allocate (note dimension reversal)
alloc_local = fftw_mpi_local_size_2d(M, L, MPI_COMM_WORLD, &
                                     local_M, local_j_offset)
cdata = fftw_alloc_complex(alloc_local)
call c_f_pointer(cdata, data, [L,local_M])

!   create MPI plan for in-place forward DFT (note dimension reversal)
plan = fftw_mpi_plan_dft_2d(M, L, data, data, MPI_COMM_WORLD, &
                            FFTW_FORWARD, FFTW_MEASURE)

! initialize data to some function my_function(i,j)
do j = 1, local_M
  do i = 1, L
    data(i, j) = my_function(i, j + local_j_offset)
  end do
end do

! compute transform (as many times as desired)
call fftw_mpi_execute_dft(plan, data, data)

call fftw_destroy_plan(plan)
call fftw_free(cdata)
```

Note that when we called `fftw_mpi_local_size_2d` and `fftw_mpi_plan_dft_2d` with the dimensions in reversed order, since a $L \times M$ Fortran array is viewed by FFTW in C as a $M \times L$ array. This means that the array was distributed over the `M` dimension, the local portion of which is a $L \times local_M$ array in Fortran. (You must *not* use an `allocate` statement to allocate an $L \times local_M$ array, however; you must allocate `alloc_local` complex numbers, which may be greater than `L * local_M`, in order to reserve space for intermediate steps of the transform.) Finally, we mention that because C's array indices are zero-based, the `local_j_offset` argument can conveniently be interpreted as an offset in the 1-based `j` index (rather than as a starting index as in C).

If instead you had used the `ior(FFTW_MEASURE, FFTW_MPI_TRANSPOSED_OUT)` flag, the output of the transform would be a transposed $M \times local_L$ array, associated with the *same* `cdata` allocation (since the transform is in-place), and which you could declare with:

```
complex(C_DOUBLE_COMPLEX), pointer :: tdata(:,:)
...
call c_f_pointer(cdata, tdata, [M,local_L])
```

where `local_L` would have been obtained by changing the `fftw_mpi_local_size_2d` call to:

```
alloc_local = fftw_mpi_local_size_2d_transposed(M, L, MPI_COMM_WORLD, &
                          local_M, local_j_offset, local_L, local_i_offset)
```

7 Calling FFTW from Modern Fortran

Fortran 2003 standardized ways for Fortran code to call C libraries, and this allows us to support a direct translation of the FFTW C API into Fortran. Compared to the legacy Fortran 77 interface (see Chapter 8 [Calling FFTW from Legacy Fortran], page 87), this direct interface offers many advantages, especially compile-time type-checking and aligned memory allocation. As of this writing, support for these C interoperability features seems widespread, having been implemented in nearly all major Fortran compilers (e.g. GNU, Intel, IBM, Oracle/Solaris, Portland Group, NAG).

This chapter documents that interface. For the most part, since this interface allows Fortran to call the C interface directly, the usage is identical to C translated to Fortran syntax. However, there are a few subtle points such as memory allocation, wisdom, and data types that deserve closer attention.

7.1 Overview of Fortran interface

FFTW provides a file `fftw3.f03` that defines Fortran 2003 interfaces for all of its C routines, except for the MPI routines described elsewhere, which can be found in the same directory as `fftw3.h` (the C header file). In any Fortran subroutine where you want to use FFTW functions, you should begin with:

```
use, intrinsic :: iso_c_binding
include 'fftw3.f03'
```

This includes the interface definitions and the standard `iso_c_binding` module (which defines the equivalents of C types). You can also put the FFTW functions into a module if you prefer (see Section 7.7 [Defining an FFTW module], page 85).

At this point, you can now call anything in the FFTW C interface directly, almost exactly as in C other than minor changes in syntax. For example:

```
type(C_PTR) :: plan
complex(C_DOUBLE_COMPLEX), dimension(1024,1000) :: in, out
plan = fftw_plan_dft_2d(1000,1024, in,out, FFTW_FORWARD,FFTW_ESTIMATE)
...
call fftw_execute_dft(plan, in, out)
...
call fftw_destroy_plan(plan)
```

A few important things to keep in mind are:

- FFTW plans are `type(C_PTR)`. Other C types are mapped in the obvious way via the `iso_c_binding` standard: `int` turns into `integer(C_INT)`, `fftw_complex` turns into `complex(C_DOUBLE_COMPLEX)`, `double` turns into `real(C_DOUBLE)`, and so on. See Section 7.3 [FFTW Fortran type reference], page 80.
- Functions in C become functions in Fortran if they have a return value, and subroutines in Fortran otherwise.
- The ordering of the Fortran array dimensions must be *reversed* when they are passed to the FFTW plan creation, thanks to differences in array indexing conventions (see

Section 3.2 [Multi-dimensional Array Format], page 15). This is *unlike* the legacy Fortran interface (see Section 8.1 [Fortran-interface routines], page 87), which reversed the dimensions for you. See Section 7.2 [Reversing array dimensions], page 78.

- Using ordinary Fortran array declarations like this works, but may yield suboptimal performance because the data may not be not aligned to exploit SIMD instructions on modern proessors (see Section 3.1 [SIMD alignment and fftw_malloc], page 15). Better performance will often be obtained by allocating with '`fftw_alloc`'. See Section 7.5 [Allocating aligned memory in Fortran], page 82.
- Similar to the legacy Fortran interface (see Section 8.3 [FFTW Execution in Fortran], page 88), we currently recommend *not* using `fftw_execute` but rather using the more specialized functions like `fftw_execute_dft` (see Section 4.6 [New-array Execute Functions], page 38). However, you should execute the plan on the `same arrays` as the ones for which you created the plan, unless you are especially careful. See Section 7.4 [Plan execution in Fortran], page 81. To prevent you from using `fftw_execute` by mistake, the `fftw3.f03` file does not provide an `fftw_execute` interface declaration.
- Multiple planner flags are combined with `ior` (equivalent to '`|`' in C). e.g. `FFTW_MEASURE | FFTW_DESTROY_INPUT` becomes `ior(FFTW_MEASURE, FFTW_DESTROY_INPUT)`. (You can also use '`+`' as long as you don't try to include a given flag more than once.)

7.1.1 Extended and quadruple precision in Fortran

If FFTW is compiled in `long double` (extended) precision (see Chapter 10 [Installation and Customization], page 97), you may be able to call the resulting `fftwl_` routines (see Section 4.1.2 [Precision], page 21) from Fortran if your compiler supports the `C_LONG_DOUBLE_COMPLEX` type code.

Because some Fortran compilers do not support `C_LONG_DOUBLE_COMPLEX`, the `fftwl_` declarations are segregated into a separate interface file `fftw3l.f03`, which you should include *in addition* to `fftw3.f03` (which declares precision-independent '`FFTW_`' constants):

```
use, intrinsic :: iso_c_binding
include 'fftw3.f03'
include 'fftw3l.f03'
```

We also support using the nonstandard `__float128` quadruple-precision type provided by recent versions of `gcc` on 32- and 64-bit x86 hardware (see Chapter 10 [Installation and Customization], page 97), using the corresponding `real(16)` and `complex(16)` types supported by `gfortran`. The quadruple-precision '`fftwq_`' functions (see Section 4.1.2 [Precision], page 21) are declared in a `fftw3q.f03` interface file, which should be included in addition to `fftw3l.f03`, as above. You should also link with `-lfftw3q -lquadmath -lm` as in C.

7.2 Reversing array dimensions

A minor annoyance in calling FFTW from Fortran is that FFTW's array dimensions are defined in the C convention (row-major order), while Fortran's array dimensions are the opposite convention (column-major order). See Section 3.2 [Multi-dimensional Array Format], page 15. This is just a bookkeeping difference, with no effect on performance. The only

consequence of this is that, whenever you create an FFTW plan for a multi-dimensional transform, you must always *reverse the ordering of the dimensions*.

For example, consider the three-dimensional ($L \times M \times N$) arrays:

```
complex(C_DOUBLE_COMPLEX), dimension(L,M,N) :: in, out
```

To plan a DFT for these arrays using `fftw_plan_dft_3d`, you could do:

```
plan = fftw_plan_dft_3d(N,M,L, in,out, FFTW_FORWARD,FFTW_ESTIMATE)
```

That is, from FFTW's perspective this is a $N \times M \times L$ array. *No data transposition need occur*, as this is *only notation*. Similarly, to use the more generic routine `fftw_plan_dft` with the same arrays, you could do:

```
integer(C_INT), dimension(3) :: n = [N,M,L]
plan = fftw_plan_dft_3d(3, n, in,out, FFTW_FORWARD,FFTW_ESTIMATE)
```

Note, by the way, that this is different from the legacy Fortran interface (see Section 8.1 [Fortran-interface routines], page 87), which automatically reverses the order of the array dimension for you. Here, you are calling the C interface directly, so there is no "translation" layer.

An important thing to keep in mind is the implication of this for multidimensional real-to-complex transforms (see Section 2.4 [Multi-Dimensional DFTs of Real Data], page 7). In C, a multidimensional real-to-complex DFT chops the last dimension roughly in half ($N \times M \times L$ real input goes to $N \times M \times L/2+1$ complex output). In Fortran, because the array dimension notation is reversed, the *first* dimension of the complex data is chopped roughly in half. For example consider the '`r2c`' transform of $L \times M \times N$ real input in Fortran:

```
type(C_PTR) :: plan
real(C_DOUBLE), dimension(L,M,N) :: in
complex(C_DOUBLE_COMPLEX), dimension(L/2+1,M,N) :: out
plan = fftw_plan_dft_r2c_3d(N,M,L, in,out, FFTW_ESTIMATE)
...
call fftw_execute_dft_r2c(plan, in, out)
```

Alternatively, for an in-place r2c transform, as described in the C documentation we must *pad* the *first* dimension of the real input with an extra two entries (which are ignored by FFTW) so as to leave enough space for the complex output. The input is *allocated* as a $2[L/2+1] \times M \times N$ array, even though only $L \times M \times N$ of it is actually used. In this example, we will allocate the array as a pointer type, using '`fftw_alloc`' to ensure aligned memory for maximum performance (see Section 7.5 [Allocating aligned memory in Fortran], page 82); this also makes it easy to reference the same memory as both a real array and a complex array.

```
real(C_DOUBLE), pointer :: in(:,:,:)
complex(C_DOUBLE_COMPLEX), pointer :: out(:,:,:)
type(C_PTR) :: plan, data
data = fftw_alloc_complex(int((L/2+1) * M * N, C_SIZE_T))
call c_f_pointer(data, in, [2*(L/2+1),M,N])
call c_f_pointer(data, out, [L/2+1,M,N])
```

```
plan = fftw_plan_dft_r2c_3d(N,M,L, in,out, FFTW_ESTIMATE)
...
call fftw_execute_dft_r2c(plan, in, out)
...
call fftw_destroy_plan(plan)
call fftw_free(data)
```

7.3 FFTW Fortran type reference

The following are the most important type correspondences between the C interface and Fortran:

- Plans (`fftw_plan` and variants) are `type(C_PTR)` (i.e. an opaque pointer).
- The C floating-point types `double`, `float`, and `long double` correspond to `real(C_DOUBLE)`, `real(C_FLOAT)`, and `real(C_LONG_DOUBLE)`, respectively. The C complex types `fftw_complex`, `fftwf_complex`, and `fftwl_complex` correspond in Fortran to `complex(C_DOUBLE_COMPLEX)`, `complex(C_FLOAT_COMPLEX)`, and `complex(C_LONG_DOUBLE_COMPLEX)`, respectively. Just as in C (see Section 4.1.2 [Precision], page 21), the FFTW subroutines and types are prefixed with '`fftw_`', `fftwf_`, and `fftwl_` for the different precisions, and link to different libraries (`-lfftw3`, `-lfftw3f`, and `-lfftw3l` on Unix), but use the *same* include file `fftw3.f03` and the *same* constants (all of which begin with '`FFTW_`'). The exception is `long double` precision, for which you should *also* include `fftw3l.f03` (see Section 7.1.1 [Extended and quadruple precision in Fortran], page 78).
- The C integer types `int` and `unsigned` (used for planner flags) become `integer(C_INT)`. The C integer type `ptrdiff_t` (e.g. in the Section 4.5.6 [64-bit Guru Interface], page 38) becomes `integer(C_INTPTR_T)`, and `size_t` (in `fftw_malloc` etc.) becomes `integer(C_SIZE_T)`.
- The `fftw_r2r_kind` type (see Section 4.3.6 [Real-to-Real Transform Kinds], page 30) becomes `integer(C_FFTW_R2R_KIND)`. The various constant values of the C enumerated type (`FFTW_R2HC` etc.) become simply integer constants of the same names in Fortran.
- Numeric array pointer arguments (e.g. `double *`) become `dimension(*), intent(out)` arrays of the same type, or `dimension(*), intent(in)` if they are pointers to constant data (e.g. `const int *`). There are a few exceptions where numeric pointers refer to scalar outputs (e.g. for `fftw_flops`), in which case they are `intent(out)` scalar arguments in Fortran too. For the new-array execute functions (see Section 4.6 [New-array Execute Functions], page 38), the input arrays are declared `dimension(*), intent(inout)`, since they can be modified in the case of in-place or `FFTW_DESTROY_INPUT` transforms.
- Pointer *return* values (e.g `double *`) become `type(C_PTR)`. (If they are pointers to arrays, as for `fftw_alloc_real`, you can convert them back to Fortran array pointers with the standard intrinsic function `c_f_pointer`.)
- The `fftw_iodim` type in the guru interface (see Section 4.5.2 [Guru vector and transform sizes], page 34) becomes `type(fftw_iodim)` in Fortran, a derived data type (the Fortran analogue of C's `struct`) with three `integer(C_INT)` components: `n`, `is`, and

`os`, with the same meanings as in C. The `fftw_iodim64` type in the 64-bit guru interface (see Section 4.5.6 [64-bit Guru Interface], page 38) is the same, except that its components are of type `integer(C_INTPTR_T)`.

- Using the wisdom import/export functions from Fortran is a bit tricky, and is discussed in Section 7.6 [Accessing the wisdom API from Fortran], page 83. In brief, the `FILE *` arguments map to `type(C_PTR)`, `const char *` to `character(C_CHAR), dimension(*), intent(in)` (null-terminated!), and the generic read-char/write-char functions map to `type(C_FUNPTR)`.

You may be wondering if you need to search-and-replace `real(kind(0.0d0))` (or whatever your favorite Fortran spelling of "double precision" is) with `real(C_DOUBLE)` everywhere in your program, and similarly for `complex` and `integer` types. The answer is no; you can still use your existing types. As long as these types match their C counterparts, things should work without a hitch. The worst that can happen, e.g. in the (unlikely) event of a system where `real(kind(0.0d0))` is different from `real(C_DOUBLE)`, is that the compiler will give you a type-mismatch error. That is, if you don't use the `iso_c_binding` kinds you need to accept at least the theoretical possibility of having to change your code in response to compiler errors on some future machine, but you don't need to worry about silently compiling incorrect code that yields runtime errors.

7.4 Plan execution in Fortran

In C, in order to use a plan, one normally calls `fftw_execute`, which executes the plan to perform the transform on the input/output arrays passed when the plan was created (see Section 4.2 [Using Plans], page 22). The corresponding subroutine call in modern Fortran is:

```
call fftw_execute(plan)
```

However, we have had reports that this causes problems with some recent optimizing Fortran compilers. The problem is, because the input/output arrays are not passed as explicit arguments to `fftw_execute`, the semantics of Fortran (unlike C) allow the compiler to assume that the input/output arrays are not changed by `fftw_execute`. As a consequence, certain compilers end up repositioning the call to `fftw_execute`, assuming incorrectly that it does nothing to the arrays.

There are various workarounds to this, but the safest and simplest thing is to not use `fftw_execute` in Fortran. Instead, use the functions described in Section 4.6 [New-array Execute Functions], page 38, which take the input/output arrays as explicit arguments. For example, if the plan is for a complex-data DFT and was created for the arrays `in` and `out`, you would do:

```
call fftw_execute_dft(plan, in, out)
```

There are a few things to be careful of, however:

- You must use the correct type of execute function, matching the way the plan was created. Complex DFT plans should use `fftw_execute_dft`, Real-input (r2c) DFT plans should use use `fftw_execute_dft_r2c`, and real-output (c2r) DFT plans should use `fftw_execute_dft_c2r`. The various r2r plans should use `fftw_execute_r2r`.

Fortunately, if you use the wrong one you will get a compile-time type-mismatch error (unlike legacy Fortran).

- You should normally pass the same input/output arrays that were used when creating the plan. This is always safe.
- *If* you pass *different* input/output arrays compared to those used when creating the plan, you must abide by all the restrictions of the new-array execute functions (see Section 4.6 [New-array Execute Functions], page 38). The most tricky of these is the requirement that the new arrays have the same alignment as the original arrays; the best (and possibly only) way to guarantee this is to use the '`fftw_alloc`' functions to allocate your arrays (see Section 7.5 [Allocating aligned memory in Fortran], page 82). Alternatively, you can use the `FFTW_UNALIGNED` flag when creating the plan, in which case the plan does not depend on the alignment, but this may sacrifice substantial performance on architectures (like x86) with SIMD instructions (see Section 3.1 [SIMD alignment and fftw_malloc], page 15).

7.5 Allocating aligned memory in Fortran

In order to obtain maximum performance in FFTW, you should store your data in arrays that have been specially aligned in memory (see Section 3.1 [SIMD alignment and fftw_malloc], page 15). Enforcing alignment also permits you to safely use the new-array execute functions (see Section 4.6 [New-array Execute Functions], page 38) to apply a given plan to more than one pair of in/out arrays. Unfortunately, standard Fortran arrays do *not* provide any alignment guarantees. The *only* way to allocate aligned memory in standard Fortran is to allocate it with an external C function, like the `fftw_alloc_real` and `fftw_alloc_complex` functions. Fortunately, Fortran 2003 provides a simple way to associate such allocated memory with a standard Fortran array pointer that you can then use normally.

We therefore recommend allocating all your input/output arrays using the following technique:

1. Declare a `pointer`, `arr`, to your array of the desired type and dimensions. For example, `real(C_DOUBLE), pointer :: a(:,:)` for a 2d real array, or `complex(C_DOUBLE_COMPLEX), pointer :: a(:,:,:)` for a 3d complex array.
2. The number of elements to allocate must be an `integer(C_SIZE_T)`. You can either declare a variable of this type, e.g. `integer(C_SIZE_T) :: sz`, to store the number of elements to allocate, or you can use the `int(..., C_SIZE_T)` intrinsic function. e.g. set `sz = L * M * N` or use `int(L * M * N, C_SIZE_T)` for an $L \times M \times N$ array.
3. Declare a `type(C_PTR) :: p` to hold the return value from FFTW's allocation routine. Set `p = fftw_alloc_real(sz)` for a real array, or `p = fftw_alloc_complex(sz)` for a complex array.
4. Associate your pointer `arr` with the allocated memory `p` using the standard `c_f_pointer` subroutine: `call c_f_pointer(p, arr, [...dimensions...])`, where `[...dimensions...])` are an array of the dimensions of the array (in the usual Fortran order). e.g. `call c_f_pointer(p, arr, [L,M,N])` for an $L \times M \times N$ array. (Alternatively, you can omit the dimensions argument if you specified the shape explicitly when declaring `arr`.) You can now use `arr` as a usual multidimensional array.

5. When you are done using the array, deallocate the memory by `call fftw_free(p)` on `p`.

For example, here is how we would allocate an $L \times M$ 2d real array:

```
real(C_DOUBLE), pointer :: arr(:,:)
type(C_PTR) :: p
p = fftw_alloc_real(int(L * M, C_SIZE_T))
call c_f_pointer(p, arr, [L,M])
...use arr and arr(i,j) as usual...
call fftw_free(p)
```

and here is an $L \times M \times N$ 3d complex array:

```
complex(C_DOUBLE_COMPLEX), pointer :: arr(:,:,:)
type(C_PTR) :: p
p = fftw_alloc_complex(int(L * M * N, C_SIZE_T))
call c_f_pointer(p, arr, [L,M,N])
...use arr and arr(i,j,k) as usual...
call fftw_free(p)
```

See Section 7.2 [Reversing array dimensions], page 78 for an example allocating a single array and associating both real and complex array pointers with it, for in-place real-to-complex transforms.

7.6 Accessing the wisdom API from Fortran

As explained in Section 3.3 [Words of Wisdom-Saving Plans], page 18, FFTW provides a "wisdom" API for saving plans to disk so that they can be recreated quickly. The C API for exporting (see Section 4.7.1 [Wisdom Export], page 40) and importing (see Section 4.7.2 [Wisdom Import], page 41) wisdom is somewhat tricky to use from Fortran, however, because of differences in file I/O and string types between C and Fortran.

7.6.1 Wisdom File Export/Import from Fortran

The easiest way to export and import wisdom is to do so using `fftw_export_wisdom_to_filename` and `fftw_wisdom_from_filename`. The only trick is that these require you to pass a C string, which is an array of type `CHARACTER(C_CHAR)` that is terminated by `C_NULL_CHAR`. You can call them like this:

```
integer(C_INT) :: ret
ret = fftw_export_wisdom_to_filename(C_CHAR_'my_wisdom.dat' // C_NULL_CHAR)
if (ret .eq. 0) stop 'error exporting wisdom to file'
ret = fftw_import_wisdom_from_filename(C_CHAR_'my_wisdom.dat' // C_NULL_CHAR)
if (ret .eq. 0) stop 'error importing wisdom from file'
```

Note that prepending '`C_CHAR_`' is needed to specify that the literal string is of kind `C_CHAR`, and we null-terminate the string by appending '`// C_NULL_CHAR`'. These functions return an `integer(C_INT)` (`ret`) which is `0` if an error occurred during export/import and nonzero otherwise.

It is also possible to use the lower-level routines `fftw_export_wisdom_to_file` and `fftw_import_wisdom_from_file`, which accept parameters of the C type `FILE*`, expressed in

Fortran as `type(C_PTR)`. However, you are then responsible for creating the `FILE*` yourself. You can do this by using `iso_c_binding` to define Fortran intefaces for the C library functions `fopen` and `fclose`, which is a bit strange in Fortran but workable.

7.6.2 Wisdom String Export/Import from Fortran

Dealing with FFTW's C string export/import is a bit more painful. In particular, the `fftw_export_wisdom_to_string` function requires you to deal with a dynamically allocated C string. To get its length, you must define an interface to the C `strlen` function, and to deallocate it you must define an interface to C `free`:

```
use, intrinsic :: iso_c_binding
interface
  integer(C_INT) function strlen(s) bind(C, name='strlen')
    import
    type(C_PTR), value :: s
  end function strlen
  subroutine free(p) bind(C, name='free')
    import
    type(C_PTR), value :: p
  end subroutine free
end interface
```

Given these definitions, you can then export wisdom to a Fortran character array:

```
character(C_CHAR), pointer :: s(:)
integer(C_SIZE_T) :: slen
type(C_PTR) :: p
p = fftw_export_wisdom_to_string()
if (.not. c_associated(p)) stop 'error exporting wisdom'
slen = strlen(p)
call c_f_pointer(p, s, [slen+1])
...
call free(p)
```

Note that `slen` is the length of the C string, but the length of the array is `slen+1` because it includes the terminating null character. (You can omit the '`+1`' if you don't want Fortran to know about the null character.) The standard `c_associated` function checks whether `p` is a null pointer, which is returned by `fftw_export_wisdom_to_string` if there was an error.

To import wisdom from a string, use `fftw_import_wisdom_from_string` as usual; note that the argument of this function must be a `character(C_CHAR)` that is terminated by the `C_NULL_CHAR` character, like the `s` array above.

7.6.3 Wisdom Generic Export/Import from Fortran

The most generic wisdom export/import functions allow you to provide an arbitrary callback function to read/write one character at a time in any way you want. However, your callback function must be written in a special way, using the `bind(C)` attribute to be passed to a C interface.

In particular, to call the generic wisdom export function `fftw_export_wisdom`, you would write a callback subroutine of the form:

```
subroutine my_write_char(c, p) bind(C)
  use, intrinsic :: iso_c_binding
  character(C_CHAR), value :: c
  type(C_PTR), value :: p
  ...write c...
end subroutine my_write_char
```

Given such a subroutine (along with the corresponding interface definition), you could then export wisdom using:

```
call fftw_export_wisdom(c_funloc(my_write_char), p)
```

The standard `c_funloc` intrinsic converts a Fortran `bind(C)` subroutine into a C function pointer. The parameter `p` is a `type(C_PTR)` to any arbitrary data that you want to pass to `my_write_char` (or `C_NULL_PTR` if none). (Note that you can get a C pointer to Fortran data using the intrinsic `c_loc`, and convert it back to a Fortran pointer in `my_write_char` using `c_f_pointer`.)

Similarly, to use the generic `fftw_import_wisdom`, you would define a callback function of the form:

```
integer(C_INT) function my_read_char(p) bind(C)
  use, intrinsic :: iso_c_binding
  type(C_PTR), value :: p
  character :: c
  ...read a character c...
  my_read_char = ichar(c, C_INT)
end function my_read_char

....

integer(C_INT) :: ret
ret = fftw_import_wisdom(c_funloc(my_read_char), p)
if (ret .eq. 0) stop 'error importing wisdom'
```

Your function can return `-1` if the end of the input is reached. Again, `p` is an arbitrary `type(C_PTR` that is passed through to your function. `fftw_import_wisdom` returns `0` if an error occurred and nonzero otherwise.

7.7 Defining an FFTW module

Rather than using the `include` statement to include the `fftw3.f03` interface file in any subroutine where you want to use FFTW, you might prefer to define an FFTW Fortran module. FFTW does not install itself as a module, primarily because `fftw3.f03` can be shared between different Fortran compilers while modules (in general) cannot. However, it is trivial to define your own FFTW module if you want. Just create a file containing:

```
module FFTW3
  use, intrinsic :: iso_c_binding
```

```
  include 'fftw3.f03'
end module
```

Compile this file into a module as usual for your compiler (e.g. with `gfortran -c` you will get a file `fftw3.mod`). Now, instead of `include 'fftw3.f03'`, whenever you want to use FFTW routines you can just do:

```
use FFTW3
```

as usual for Fortran modules. (You still need to link to the FFTW library, of course.)

8 Calling FFTW from Legacy Fortran

This chapter describes the interface to FFTW callable by Fortran code in older compilers not supporting the Fortran 2003 C interoperability features (see Chapter 7 [Calling FFTW from Modern Fortran], page 77). This interface has the major disadvantage that it is not type-checked, so if you mistake the argument types or ordering then your program will not have any compiler errors, and will likely crash at runtime. So, greater care is needed. Also, technically interfacing older Fortran versions to C is nonstandard, but in practice we have found that the techniques used in this chapter have worked with all known Fortran compilers for many years.

The legacy Fortran interface differs from the C interface only in the prefix ('`dfftw_`' instead of '`fftw_`' in double precision) and a few other minor details. This Fortran interface is 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. We assume here that the reader is already familiar with the usage of FFTW in C, as described elsewhere in this manual.

The MPI parallel interface to FFTW is *not* currently available to legacy Fortran.

8.1 Fortran-interface routines

Nearly all of the FFTW functions have Fortran-callable equivalents. The name of the legacy Fortran routine is the same as that of the corresponding C routine, but with the '`fftw_`' prefix replaced by '`dfftw_`'.[1] The single and long-double precision versions use '`sfftw_`' and '`lfftw_`', respectively, instead of '`fftwf_`' and '`fftwl_`'; quadruple precision (`real*16`) is available on some systems as '`fftwq_`' (see Section 4.1.2 [Precision], page 21). (Note that `long double` on x86 hardware is usually at most 80-bit extended precision, *not* quadruple precision.)

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` in C), must be declared as a type that is at least as big as a pointer (address) on your machine. We recommend using `integer*8` everywhere, since this should always be big enough.
- Any function that returns a value (e.g. `fftw_plan_dft`) is converted into a *subroutine*. The return value is converted into an additional *first* parameter of this subroutine.[2]
- The Fortran routines expect multi-dimensional arrays to be in *column-major* order, which is the ordinary format of Fortran arrays (see Section 3.2 [Multi-dimensional Array Format], page 15). 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.
- Wisdom import and export is somewhat more tricky because one cannot easily pass files or strings between C and Fortran; see Section 8.5 [Wisdom of Fortran?], page 91.

[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.

[2] The reason for this is that some Fortran implementations seem to have trouble with C function return values, and vice versa.

- Legacy Fortran cannot use the `fftw_malloc` dynamic-allocation routine. If you want to exploit the SIMD FFTW (see Section 3.1 [SIMD alignment and fftw_malloc], page 15), you'll need to figure out some other way to ensure that your arrays are at least 16-byte aligned.
- Since Fortran 77 does not have data structures, the `fftw_iodim` structure from the guru interface (see Section 4.5.2 [Guru vector and transform sizes], page 34) must be split into separate arguments. In particular, any `fftw_iodim` array arguments in the C guru interface become three integer array arguments (`n`, `is`, and `os`) in the Fortran guru interface, all of whose lengths should be equal to the corresponding `rank` argument.
- The guru planner interface in Fortran does *not* do any automatic translation between column-major and row-major; you are responsible for setting the strides etcetera to correspond to your Fortran arrays. However, as a slight bug that we are preserving for backwards compatibility, the '`plan_guru_r2r`' in Fortran *does* reverse the order of its `kind` array parameter, so the `kind` array of that routine should be in the reverse of the order of the iodim arrays (see above).

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. In practice, this correspondence is usually straightforward (i.e. `integer` corresponds to `int`, `real` corresponds to `float`, etcetera). The native Fortran double/single-precision complex type should be compatible with `fftw_complex`/`fftwf_complex`. Such simple correspondences are assumed in the examples below.

8.2 FFTW Constants in Fortran

When creating plans in FFTW, a number of constants are used to specify options, such as `FFTW_MEASURE` or `FFTW_ESTIMATE`. 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 `fftw3.f`, which can be found in the same directory as `fftw3.h`. If your Fortran compiler supports a preprocessor of some sort, you should be able to `include` or `#include` this file; otherwise, you can paste it directly into your code.

In C, you combine different flags (like `FFTW_PRESERVE_INPUT` and `FFTW_MEASURE`) using the '`|`' operator; in Fortran you should just use '`+`'. (Take care not to add in the same flag more than once, though. Alternatively, you can use the `ior` intrinsic function standardized in Fortran 95.)

8.3 FFTW Execution in Fortran

In C, in order to use a plan, one normally calls `fftw_execute`, which executes the plan to perform the transform on the input/output arrays passed when the plan was created (see Section 4.2 [Using Plans], page 22). The corresponding subroutine call in legacy Fortran is:

```
call dfftw_execute(plan)
```

However, we have had reports that this causes problems with some recent optimizing Fortran compilers. The problem is, because the input/output arrays are not passed as explicit

arguments to `dfftw_execute`, the semantics of Fortran (unlike C) allow the compiler to assume that the input/output arrays are not changed by `dfftw_execute`. As a consequence, certain compilers end up optimizing out or repositioning the call to `dfftw_execute`, assuming incorrectly that it does nothing.

There are various workarounds to this, but the safest and simplest thing is to not use `dfftw_execute` in Fortran. Instead, use the functions described in Section 4.6 [New-array Execute Functions], page 38, which take the input/output arrays as explicit arguments. For example, if the plan is for a complex-data DFT and was created for the arrays `in` and `out`, you would do:

```
call dfftw_execute_dft(plan, in, out)
```

There are a few things to be careful of, however:

- You must use the correct type of execute function, matching the way the plan was created. Complex DFT plans should use `dfftw_execute_dft`, Real-input (r2c) DFT plans should use use `dfftw_execute_dft_r2c`, and real-output (c2r) DFT plans should use `dfftw_execute_dft_c2r`. The various r2r plans should use `dfftw_execute_r2r`.
- You should normally pass the same input/output arrays that were used when creating the plan. This is always safe.
- *If* you pass *different* input/output arrays compared to those used when creating the plan, you must abide by all the restrictions of the new-array execute functions (see Section 4.6 [New-array Execute Functions], page 38). The most difficult of these, in Fortran, is the requirement that the new arrays have the same alignment as the original arrays, because there seems to be no way in legacy Fortran to obtain guaranteed-aligned arrays (analogous to `fftw_malloc` in C). You can, of course, use the `FFTW_UNALIGNED` flag when creating the plan, in which case the plan does not depend on the alignment, but this may sacrifice substantial performance on architectures (like x86) with SIMD instructions (see Section 3.1 [SIMD alignment and fftw_malloc], page 15).

8.4 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_plan_dft_1d(N,in,out,FFTW_FORWARD,FFTW_ESTIMATE);
fftw_execute(plan);
fftw_destroy_plan(plan);
```

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

```
double complex in, out
dimension in(N), out(N)
integer*8 plan

call dfftw_plan_dft_1d(plan,N,in,out,FFTW_FORWARD,FFTW_ESTIMATE)
call dfftw_execute_dft(plan, in, out)
```

```
call dfftw_destroy_plan(plan)
```

Notice how all routines are called as Fortran subroutines, and the plan is returned via the first argument to `dfftw_plan_dft_1d`. Notice also that we changed `fftw_execute` to `dfftw_execute_dft` (see Section 8.3 [FFTW Execution in Fortran], page 88). To do the same thing, but using 8 threads in parallel (see Chapter 5 [Multi-threaded FFTW], page 49), you would simply prefix these calls with:

```
integer iret
call dfftw_init_threads(iret)
call dfftw_plan_with_nthreads(8)
```

(You might want to check the value of `iret`: if it is zero, it indicates an unlikely error during thread initialization.)

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

```
fftw_complex arr[L][M][N];
fftw_plan plan;

plan = fftw_plan_dft_3d(L,M,N, arr,arr,
                        FFTW_FORWARD, FFTW_ESTIMATE);
fftw_execute(plan);
fftw_destroy_plan(plan);
```

In Fortran, you would use this instead:

```
double complex arr
dimension arr(L,M,N)
integer*8 plan

call dfftw_plan_dft_3d(plan, L,M,N, arr,arr,
&                      FFTW_FORWARD, FFTW_ESTIMATE)
call dfftw_execute_dft(plan, arr, arr)
call dfftw_destroy_plan(plan)
```

Note that we pass the array dimensions in the "natural" order in both C and Fortran.

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

```
double precision in
dimension in(N)
double complex out
dimension out(N/2 + 1)
integer*8 plan

call dfftw_plan_dft_r2c_1d(plan,N,in,out,FFTW_ESTIMATE)
call dfftw_execute_dft_r2c(plan, in, out)
call dfftw_destroy_plan(plan)
```

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

```
double precision in
dimension in(M,N)
```

```
double complex out
dimension out(M/2 + 1, N)
integer*8 plan

call dfftw_plan_dft_r2c_2d(plan,M,N,in,out,FFTW_ESTIMATE)
call dfftw_execute_dft_r2c(plan, in, out)
call dfftw_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 interface routines reversing the order of the array dimensions passed to FFTW so that the Fortran program can use its ordinary column-major order.

8.5 Wisdom of Fortran?

In this section, we discuss how one can import/export FFTW wisdom (saved plans) to/from a Fortran program; we assume that the reader is already familiar with wisdom, as described in Section 3.3 [Words of Wisdom-Saving Plans], page 18.

The basic problem is that is difficult to (portably) pass files and strings between Fortran and C, so we cannot provide a direct Fortran equivalent to the `fftw_export_wisdom_to_file`, etcetera, functions. Fortran interfaces *are* provided for the functions that do not take file/string arguments, however: `dfftw_import_system_wisdom`, `dfftw_import_wisdom`, `dfftw_export_wisdom`, and `dfftw_forget_wisdom`.

So, for example, to import the system-wide wisdom, you would do:

```
integer isuccess
call dfftw_import_system_wisdom(isuccess)
```

As usual, the C return value is turned into a first parameter; `isuccess` is non-zero on success and zero on failure (e.g. if there is no system wisdom installed).

If you want to import/export wisdom from/to an arbitrary file or elsewhere, you can employ the generic `dfftw_import_wisdom` and `dfftw_export_wisdom` functions, for which you must supply a subroutine to read/write one character at a time. The FFTW package contains an example file `doc/f77_wisdom.f` demonstrating how to implement `import_wisdom_from_file` and `export_wisdom_to_file` subroutines in this way. (These routines cannot be compiled into the FFTW library itself, lest all FFTW-using programs be required to link with the Fortran I/O library.)

9 Upgrading from FFTW version 2

In this chapter, we outline the process for updating codes designed for the older FFTW 2 interface to work with FFTW 3. The interface for FFTW 3 is not backwards-compatible with the interface for FFTW 2 and earlier versions; codes written to use those versions will fail to link with FFTW 3. Nor is it possible to write "compatibility wrappers" to bridge the gap (at least not efficiently), because FFTW 3 has different semantics from previous versions. However, upgrading should be a straightforward process because the data formats are identical and the overall style of planning/execution is essentially the same.

Unlike FFTW 2, there are no separate header files for real and complex transforms (or even for different precisions) in FFTW 3; all interfaces are defined in the `<fftw3.h>` header file.

Numeric Types

The main difference in data types is that `fftw_complex` in FFTW 2 was defined as a `struct` with macros `c_re` and `c_im` for accessing the real/imaginary parts. (This is binary-compatible with FFTW 3 on any machine except perhaps for some older Crays in single precision.) The equivalent macros for FFTW 3 are:

```
#define c_re(c) ((c)[0])
#define c_im(c) ((c)[1])
```

This does not work if you are using the C99 complex type, however, unless you insert a `double*` typecast into the above macros (see Section 4.1.1 [Complex numbers], page 21).

Also, FFTW 2 had an `fftw_real` typedef that was an alias for `double` (in double precision). In FFTW 3 you should just use `double` (or whatever precision you are employing).

Plans

The major difference between FFTW 2 and FFTW 3 is in the planning/execution division of labor. In FFTW 2, plans were found for a given transform size and type, and then could be applied to *any* arrays and for *any* multiplicity/stride parameters. In FFTW 3, you specify the particular arrays, stride parameters, etcetera when creating the plan, and the plan is then executed for *those* arrays (unless the guru interface is used) and *those* parameters *only*. (FFTW 2 had "specific planner" routines that planned for a particular array and stride, but the plan could still be used for other arrays and strides.) That is, much of the information that was formerly specified at execution time is now specified at planning time.

Like FFTW 2's specific planner routines, the FFTW 3 planner overwrites the input/output arrays unless you use `FFTW_ESTIMATE`.

FFTW 2 had separate data types `fftw_plan`, `fftwnd_plan`, `rfftw_plan`, and `rfftwnd_plan` for complex and real one- and multi-dimensional transforms, and each type had its own '`destroy`' function. In FFTW 3, all plans are of type `fftw_plan` and all are destroyed by `fftw_destroy_plan(plan)`.

Where you formerly used `fftw_create_plan` and `fftw_one` to plan and compute a single 1d transform, you would now use `fftw_plan_dft_1d` to plan the transform. If you used

the generic `fftw` function to execute the transform with multiplicity (`howmany`) and stride parameters, you would now use the advanced interface `fftw_plan_many_dft` to specify those parameters. The plans are now executed with `fftw_execute(plan)`, which takes all of its parameters (including the input/output arrays) from the plan.

In-place transforms no longer interpret their output argument as scratch space, nor is there an `FFTW_IN_PLACE` flag. You simply pass the same pointer for both the input and output arguments. (Previously, the output `ostride` and `odist` parameters were ignored for in-place transforms; now, if they are specified via the advanced interface, they are significant even in the in-place case, although they should normally equal the corresponding input parameters.)

The `FFTW_ESTIMATE` and `FFTW_MEASURE` flags have the same meaning as before, although the planning time will differ. You may also consider using `FFTW_PATIENT`, which is like `FFTW_MEASURE` except that it takes more time in order to consider a wider variety of algorithms.

For multi-dimensional complex DFTs, instead of `fftwnd_create_plan` (or `fftw2d_create_plan` or `fftw3d_create_plan`), followed by `fftwnd_one`, you would use `fftw_plan_dft` (or `fftw_plan_dft_2d` or `fftw_plan_dft_3d`). followed by `fftw_execute`. If you used `fftwnd` to to specify strides etcetera, you would instead specify these via `fftw_plan_many_dft`.

The analogues to `rfftw_create_plan` and `rfftw_one` with `FFTW_REAL_TO_COMPLEX` or `FFTW_COMPLEX_TO_REAL` directions are `fftw_plan_r2r_1d` with kind `FFTW_R2HC` or `FFTW_HC2R`, followed by `fftw_execute`. The stride etcetera arguments of `rfftw` are now in `fftw_plan_many_r2r`.

Instead of `rfftwnd_create_plan` (or `rfftw2d_create_plan` or `rfftw3d_create_plan`) followed by `rfftwnd_one_real_to_complex` or `rfftwnd_one_complex_to_real`, you now use `fftw_plan_dft_r2c` (or `fftw_plan_dft_r2c_2d` or `fftw_plan_dft_r2c_3d`) or `fftw_plan_dft_c2r` (or `fftw_plan_dft_c2r_2d` or `fftw_plan_dft_c2r_3d`), respectively, followed by `fftw_execute`. As usual, the strides etcetera of `rfftwnd_real_to_complex` or `rfftwnd_complex_to_real` are no specified in the advanced planner routines, `fftw_plan_many_dft_r2c` or `fftw_plan_many_dft_c2r`.

Wisdom

In FFTW 2, you had to supply the `FFTW_USE_WISDOM` flag in order to use wisdom; in FFTW 3, wisdom is always used. (You could simulate the FFTW 2 wisdom-less behavior by calling `fftw_forget_wisdom` after every planner call.)

The FFTW 3 wisdom import/export routines are almost the same as before (although the storage format is entirely different). There is one significant difference, however. In FFTW 2, the import routines would never read past the end of the wisdom, so you could store extra data beyond the wisdom in the same file, for example. In FFTW 3, the file-import routine may read up to a few hundred bytes past the end of the wisdom, so you cannot store other data just beyond it.[1]

Wisdom has been enhanced by additional humility in FFTW 3: whereas FFTW 2 would re-use wisdom for a given transform size regardless of the stride etc., in FFTW 3 wisdom is

[1] We do our own buffering because GNU libc I/O routines are horribly slow for single-character I/O, apparently for thread-safety reasons (whether you are using threads or not).

only used with the strides etc. for which it was created. Unfortunately, this means FFTW 3 has to create new plans from scratch more often than FFTW 2 (in FFTW 2, planning e.g. one transform of size 1024 also created wisdom for all smaller powers of 2, but this no longer occurs).

FFTW 3 also has the new routine `fftw_import_system_wisdom` to import wisdom from a standard system-wide location.

Memory allocation

In FFTW 3, we recommend allocating your arrays with `fftw_malloc` and deallocating them with `fftw_free`; this is not required, but allows optimal performance when SIMD acceleration is used. (Those two functions actually existed in FFTW 2, and worked the same way, but were not documented.)

In FFTW 2, there were `fftw_malloc_hook` and `fftw_free_hook` functions that allowed the user to replace FFTW's memory-allocation routines (e.g. to implement different error-handling, since by default FFTW prints an error message and calls `exit` to abort the program if `malloc` returns `NULL`). These hooks are not supported in FFTW 3; those few users who require this functionality can just directly modify the memory-allocation routines in FFTW (they are defined in `kernel/alloc.c`).

Fortran interface

In FFTW 2, the subroutine names were obtained by replacing '`fftw_`' with '`fftw_f77`'; in FFTW 3, you replace '`fftw_`' with '`dfftw_`' (or '`sfftw_`' or '`lfftw_`', depending upon the precision).

In FFTW 3, we have begun recommending that you always declare the type used to store plans as `integer*8`. (Too many people didn't notice our instruction to switch from `integer` to `integer*8` for 64-bit machines.)

In FFTW 3, we provide a `fftw3.f` "header file" to include in your code (and which is officially installed on Unix systems). (In FFTW 2, we supplied a `fftw_f77.i` file, but it was not installed.)

Otherwise, the C-Fortran interface relationship is much the same as it was before (e.g. return values become initial parameters, and multi-dimensional arrays are in column-major order). Unlike FFTW 2, we do provide some support for wisdom import/export in Fortran (see Section 8.5 [Wisdom of Fortran?], page 91).

Threads

Like FFTW 2, only the execution routines are thread-safe. All planner routines, etcetera, should be called by only a single thread at a time (see Section 5.4 [Thread safety], page 51). *Unlike* FFTW 2, there is no special `FFTW_THREADSAFE` flag for the planner to allow a given plan to be usable by multiple threads in parallel; this is now the case by default.

The multi-threaded version of FFTW 2 required you to pass the number of threads each time you execute the transform. The number of threads is now stored in the plan, and is specified before the planner is called by `fftw_plan_with_nthreads`. The threads initialization routine used to be called `fftw_threads_init` and would return zero on success;

the new routine is called `fftw_init_threads` and returns zero on failure. See Chapter 5 [Multi-threaded FFTW], page 49.

There is no separate threads header file in FFTW 3; all the function prototypes are in `<fftw3.h>`. However, you still have to link to a separate library (`-lfftw3_threads -lfftw3 -lm` on Unix), as well as to the threading library (e.g. POSIX threads on Unix).

10 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.

In principle, FFTW should work on any system with an ANSI C compiler (`gcc` is fine). However, planner time is drastically reduced if FFTW can exploit a hardware cycle counter; FFTW comes with cycle-counter support for all modern general-purpose CPUs, but you may need to add a couple of lines of code if your compiler is not yet supported (see Section 10.3 [Cycle Counters], page 100). (On Unix, there will be a warning at the end of the `configure` output if no cycle counter is found.)

Installation of FFTW is simplest if you have a Unix or a GNU system, such as GNU/Linux, and we describe this case in the first section below, including the use of special configuration options to e.g. install different precisions or exploit optimizations for particular architectures (e.g. SIMD). Compilation on non-Unix systems is a more manual process, but we outline the procedure in the second section. It is also likely that pre-compiled binaries will be available for popular systems.

Finally, we describe how you can customize FFTW for particular needs by generating *codelets* for fast transforms of sizes not supported efficiently by the standard FFTW distribution.

10.1 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 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 chooses the `gcc` compiler by default, if it is available; you can select some other compiler with:

```
./configure CC="<the name of your C compiler>"
```

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. In this case, you should re-configure FFTW with the command

```
./configure CFLAGS="<write your CFLAGS here>"
```

and then compile as usual. 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.

`configure` 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 4.1.2 [Precision], page 21.
- `--enable-long-double`: Produces a long-double precision version of FFTW (`long double`) instead of the default double-precision (`double`). The `configure` script will halt with an error message if `long double` is the same size as `double` on your machine/compiler. See Section 4.1.2 [Precision], page 21.
- `--enable-quad-precision`: Produces a quadruple-precision version of FFTW using the nonstandard `__float128` type provided by `gcc` 4.6 or later on x86, x86-64, and Itanium architectures, instead of the default double-precision (`double`). The `configure` script will halt with an error message if the compiler is not `gcc` version 4.6 or later or if `gcc`'s `libquadmath` library is not installed. See Section 4.1.2 [Precision], page 21.
- `--enable-threads`: Enables compilation and installation of the FFTW threads library (see Chapter 5 [Multi-threaded FFTW], page 49), which provides a simple interface to parallel transforms for SMP systems. By default, the threads routines are not compiled.
- `--enable-openmp`: Like `--enable-threads`, but using OpenMP compiler directives in order to induce parallelism rather than spawning its own threads directly, and installing an '`fftw3_omp`' library rather than an '`fftw3_threads`' library (see Chapter 5 [Multi-threaded FFTW], page 49). You can use both `--enable-openmp` and `--enable-threads` since they compile/install libraries with different names. By default, the OpenMP routines are not compiled.
- `--with-combined-threads`: By default, if `--enable-threads` is used, the threads support is compiled into a separate library that must be linked in addition to the main FFTW library. This is so that users of the serial library do not need to link the system threads libraries. If `--with-combined-threads` is specified, however, then no separate threads library is created, and threads are included in the main FFTW library. This is mainly useful under Windows, where no system threads library is required and inter-library dependencies are problematic.
- `--enable-mpi`: Enables compilation and installation of the FFTW MPI library (see Chapter 6 [Distributed-memory FFTW with MPI], page 53), which provides parallel transforms for distributed-memory systems with MPI. (By default, the MPI routines are not compiled.) See Section 6.1 [FFTW MPI Installation], page 53.
- `--disable-fortran`: Disables inclusion of legacy-Fortran wrapper routines (see Chapter 8 [Calling FFTW from Legacy Fortran], page 87) 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. (To specify a particular Fortran compiler *foo*, pass `F77=`*foo* to `configure`.)

- `--with-g77-wrappers`: By default, when Fortran wrappers are included, the wrappers employ the linking conventions of the Fortran compiler detected by the `configure` script. If this compiler is GNU `g77`, however, then *two* versions of the wrappers are included: one with `g77`'s idiosyncratic convention of appending two underscores to identifiers, and one with the more common convention of appending only a single underscore. This way, the same FFTW library will work with both `g77` and other Fortran compilers, such as GNU `gfortran`. However, the converse is not true: if you configure with a different compiler, then the `g77`-compatible wrappers are not included. By specifying `--with-g77-wrappers`, the `g77`-compatible wrappers are included in addition to wrappers for whatever Fortran compiler `configure` finds.
- `--with-slow-timer`: Disables the use of hardware cycle counters, and falls back on `gettimeofday` or `clock`. This greatly worsens performance, and should generally not be used (unless you don't have a cycle counter but still really want an optimized plan regardless of the time). See Section 10.3 [Cycle Counters], page 100.
- `--enable-sse`, `--enable-sse2`, `--enable-avx`, `--enable-altivec`, `--enable-neon`: Enable the compilation of SIMD code for SSE (Pentium III+), SSE2 (Pentium IV+), AVX (Sandy Bridge, Interlagos), AltiVec (PowerPC G4+), NEON (some ARM processors). SSE, AltiVec, and NEON only work with `--enable-float` (above). SSE2 works in both single and double precision (and is simply SSE in single precision). The resulting code will *still work* on earlier CPUs lacking the SIMD extensions (SIMD is automatically disabled, although the FFTW library is still larger).
  - These options require a compiler supporting SIMD extensions, and compiler support is always a bit flaky: see the FFTW FAQ for a list of compiler versions that have problems compiling FFTW.
  - With AltiVec and `gcc`, you may have to use the `-mabi=altivec` option when compiling any code that links to FFTW, in order to properly align the stack; otherwise, FFTW could crash when it tries to use an AltiVec feature. (This is not necessary on MacOS X.)
  - With SSE/SSE2 and `gcc`, you should use a version of gcc that properly aligns the stack when compiling any code that links to FFTW. By default, `gcc` 2.95 and later versions align the stack as needed, but you should not compile FFTW with the `-Os` option or the `-mpreferred-stack-boundary` option with an argument less than 4.
  - Because of the large variety of ARM processors and ABIs, FFTW does not attempt to guess the correct `gcc` flags for generating NEON code. In general, you will have to provide them on the command line. This command line is known to have worked at least once:

```
./configure --with-slow-timer --host=arm-linux-gnueabi \
  --enable-single --enable-neon \
  "CC=arm-linux-gnueabi-gcc -march=armv7-a -mfloat-abi=softfp"
```

To force `configure` to use a particular C compiler *foo* (instead of the default, usually `gcc`), pass `CC=`*foo* to the `configure` script; you may also need to set the flags via the variable `CFLAGS` as described above.

10.2 Installation on non-Unix systems

It should be relatively straightforward to compile FFTW even on non-Unix systems lacking the niceties of a `configure` script. Basically, you need to edit the `config.h` header (copy it from `config.h.in`) to `#define` the various options and compiler characteristics, and then compile all the '`.c`' files in the relevant directories.

The `config.h` header contains about 100 options to set, each one initially an `#undef`, each documented with a comment, and most of them fairly obvious. For most of the options, you should simply `#define` them to `1` if they are applicable, although a few options require a particular value (e.g. `SIZEOF_LONG_LONG` should be defined to the size of the `long long` type, in bytes, or zero if it is not supported). We will likely post some sample `config.h` files for various operating systems and compilers for you to use (at least as a starting point). Please let us know if you have to hand-create a configuration file (and/or a pre-compiled binary) that you want to share.

To create the FFTW library, you will then need to compile all of the '`.c`' files in the `kernel`, `dft`, `dft/scalar`, `dft/scalar/codelets`, `rdft`, `rdft/scalar`, `rdft/scalar/r2cf`, `rdft/scalar/r2cb`, `rdft/scalar/r2r`, `reodft`, and `api` directories. If you are compiling with SIMD support (e.g. you defined `HAVE_SSE2` in `config.h`), then you also need to compile the `.c` files in the `simd-support`, `{dft,rdft}/simd`, `{dft,rdft}/simd/*` directories.

Once these files are all compiled, link them into a library, or a shared library, or directly into your program.

To compile the FFTW test program, additionally compile the code in the `libbench2/` directory, and link it into a library. Then compile the code in the `tests/` directory and link it to the `libbench2` and FFTW libraries. To compile the `fftw-wisdom` (command-line) tool (see Section 4.7.4 [Wisdom Utilities], page 41), compile `tools/fftw-wisdom.c` and link it to the `libbench2` and FFTW libraries

10.3 Cycle Counters

FFTW's planner actually executes and times different possible FFT algorithms in order to pick the fastest plan for a given n. In order to do this in as short a time as possible, however, the timer must have a very high resolution, and to accomplish this we employ the hardware *cycle counters* that are available on most CPUs. Currently, FFTW supports the cycle counters on x86, PowerPC/POWER, Alpha, UltraSPARC (SPARC v9), IA64, PA-RISC, and MIPS processors.

Access to the cycle counters, unfortunately, is a compiler and/or operating-system dependent task, often requiring inline assembly language, and it may be that your compiler is not supported. If you are *not* supported, FFTW will by default fall back on its estimator (effectively using `FFTW_ESTIMATE` for all plans).

You can add support by editing the file `kernel/cycle.h`; normally, this will involve adapting one of the examples already present in order to use the inline-assembler syntax for your C compiler, and will only require a couple of lines of code. Anyone adding support for a new system to `cycle.h` is encouraged to email us at `fftw@fftw.org`.

If a cycle counter is not available on your system (e.g. some embedded processor), and you don't want to use estimated plans, as a last resort you can use the `--with-slow-timer`

option to `configure` (on Unix) or `#define WITH_SLOW_TIMER` in `config.h` (elsewhere). This will use the much lower-resolution `gettimeofday` function, or even `clock` if the former is unavailable, and planning will be extremely slow.

10.4 Generating your own code

The directory `genfft` contains the programs that were used to generate FFTW's "codelets," which are hard-coded transforms of small sizes. We do not expect casual users to employ the generator, which is a rather sophisticated program that generates directed acyclic graphs of FFT algorithms and performs algebraic simplifications on them. It was written in Objective Caml, a dialect of ML, which is available at http://caml.inria.fr/ocaml/index.en.html.

If you have Objective Caml installed (along with recent versions of GNU `autoconf`, `automake`, and `libtool`), then you can change the set of codelets that are generated or play with the generation options. The set of generated codelets is specified by the `{dft,rdft}/{codelets,simd}/*/Makefile.am` files. For example, you can add efficient REDFT codelets of small sizes by modifying `rdft/codelets/r2r/Makefile.am`. After you modify any `Makefile.am` files, you can type `sh bootstrap.sh` in the top-level directory followed by `make` to re-generate the files.

We do not provide more details about the code-generation process, since we do not expect that most 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 also appeared in the *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

11 Acknowledgments

Matteo Frigo was supported in part by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF and by MIT Lincoln Laboratory. For previous versions of FFTW, he 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 Dept. of Defense 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.

Code for the Cell Broadband Engine was graciously donated to the FFTW project by the IBM Austin Research Lab and included in fftw-3.2. (This code was removed in fftw-3.3.)

Code for the MIPS paired-single SIMD support was graciously donated to the FFTW project by CodeSourcery, Inc.

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 early versions of FFTW.

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

We are thankful to the AMD corporation for donating an AMD Athlon XP 1700+ computer to the FFTW project.

We thank the Compaq/HP testdrive program and VA Software Corporation (SourceForge.net) for providing remote access to machines that were used to test FFTW.

The `genfft` suite of code generators 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 `http://caml.inria.fr/`. 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 "extracurricular" computer-science activities, as well as remarkable creativity in working them into his grant proposals. Steven's physics degree would not exist without him.

Franz Franchetti wrote SIMD extensions to FFTW 2, which eventually led to the SIMD support in FFTW 3.

Stefan Kral wrote most of the K7 code generator distributed with FFTW 3.0.x and 3.1.x.

Andrew Sterian contributed the Windows timing code in FFTW 2.

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.

12 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.) For general publications, we suggest referencing: Matteo Frigo and Steven G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE* **93** (2), 216–231 (2005).

Non-free versions of FFTW are available under terms different from those of 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 alternative 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.

13 Concept Index

O

P

R

S

T

V

W

14 Library Index

C

D

F

M

P

R