FFTW

for version 3.1.1, 4 March 2006

Matteo Frigo
Steven G. Johnson

This manual is for FFTW (version 3.1.1, 4 March 2006).

Copyright © 2003 Matteo Frigo.

Copyright © 2003 Massachusetts Institute of Technology.

> Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.
>
> Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.
>
> Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Table of Contents

1 Introduction 1

2 Tutorial 3
- 2.1 Complex One-Dimensional DFTs 3
- 2.2 Complex Multi-Dimensional DFTs 5
- 2.3 One-Dimensional DFTs of Real Data 6
- 2.4 Multi-Dimensional DFTs of Real Data 7
- 2.5 More DFTs of Real Data 9
  - 2.5.1 The Halfcomplex-format DFT 10
  - 2.5.2 Real even/odd DFTs (cosine/sine transforms) 10
  - 2.5.3 The Discrete Hartley Transform 12

3 Other Important Topics 15
- 3.1 Data Alignment 15
  - 3.1.1 SIMD alignment and fftw_malloc 15
  - 3.1.2 Stack alignment on x86 15
- 3.2 Multi-dimensional Array Format 16
  - 3.2.1 Row-major Format 16
  - 3.2.2 Column-major Format 16
  - 3.2.3 Fixed-size Arrays in C 16
  - 3.2.4 Dynamic Arrays in C 17
  - 3.2.5 Dynamic Arrays in C—The Wrong Way 17
- 3.3 Words of Wisdom—Saving Plans 18
- 3.4 Caveats in Using Wisdom 19

4 FFTW Reference 21
- 4.1 Data Types and Files 21
  - 4.1.1 Complex numbers 21
  - 4.1.2 Precision 21
  - 4.1.3 Memory Allocation 22
- 4.2 Using Plans 22
- 4.3 Basic Interface 23
  - 4.3.1 Complex DFTs 23
  - 4.3.2 Planner Flags 24
  - 4.3.3 Real-data DFTs 26
  - 4.3.4 Real-data DFT Array Format 27
  - 4.3.5 Real-to-Real Transforms 28
  - 4.3.6 Real-to-Real Transform Kinds 29
- 4.4 Advanced Interface 30
  - 4.4.1 Advanced Complex DFTs 30
  - 4.4.2 Advanced Real-data DFTs 31
  - 4.4.3 Advanced Real-to-real Transforms 31

- 4.5 Guru Interface ... 32
  - 4.5.1 Interleaved and split arrays ... 32
  - 4.5.2 Guru vector and transform sizes ... 32
  - 4.5.3 Guru Complex DFTs ... 33
  - 4.5.4 Guru Real-data DFTs ... 34
  - 4.5.5 Guru Real-to-real Transforms ... 35
  - 4.5.6 Guru Execution of Plans ... 36
- 4.6 Wisdom ... 37
  - 4.6.1 Wisdom Export ... 37
  - 4.6.2 Wisdom Import ... 38
  - 4.6.3 Forgetting Wisdom ... 38
  - 4.6.4 Wisdom Utilities ... 38
- 4.7 What FFTW Really Computes ... 39
  - 4.7.1 The 1d Discrete Fourier Transform (DFT) ... 39
  - 4.7.2 The 1d Real-data DFT ... 40
  - 4.7.3 1d Real-even DFTs (DCTs) ... 40
  - 4.7.4 1d Real-odd DFTs (DSTs) ... 42
  - 4.7.5 1d Discrete Hartley Transforms (DHTs) ... 43
  - 4.7.6 Multi-dimensional Transforms ... 43

5 Parallel FFTW ... 45

- 5.1 Multi-threaded FFTW ... 45
  - 5.1.1 Installation and Supported Hardware/Software ... 45
  - 5.1.2 Usage of Multi-threaded FFTW ... 45
  - 5.1.3 How Many Threads to Use? ... 46
- 5.2 Thread safety ... 47

6 Calling FFTW from Fortran ... 49

- 6.1 Fortran-interface routines ... 49
- 6.2 FFTW Constants in Fortran ... 50
- 6.3 Fortran Examples ... 50
- 6.4 Wisdom of Fortran? ... 51

7 Upgrading from FFTW version 2 ... 53

8 Installation and Customization ... 57

- 8.1 Installation on Unix ... 57
- 8.2 Installation on non-Unix systems ... 59
- 8.3 Cycle Counters ... 60
- 8.4 Generating your own code ... 60

9 Acknowledgments ... 63

10 License and Copyright ... 65

11 Concept Index 67

12 Library Index 69

1 Introduction

This manual documents version 3.1.1 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, 3DNow!, and Altivec instruction sets.
- FFTW 3.1.1 includes parallel (multi-threaded) transforms for shared-memory systems. FFTW 3.1.1 does not include distributed-memory parallel transforms, but we plan to implement an MPI version soon. (Meanwhile, you can use the MPI implementation from FFTW 2.1.3.)

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, "FFTW: An adaptive software architecture for the FFT," by M. Frigo and S. G. Johnson, which appeared in the 23rd International Conference on Acoustics, Speech, and Signal Processing (*Proc. ICASSP 1998* **3**, p. 1381). See also, "The Fastest Fourier Transform in the West," by M. Frigo and S. G. Johnson, which is the technical report MIT-LCS-TR-728 (Sep. '97). 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 variation due to 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. The following chapter discusses Chapter 3 [Other Important Topics], page 15, including Section 3.1 [Data Alignment], page 15, the storage scheme of multi-dimensional arrays (see Section 3.2 [Multi-dimensional Array Format], page 16), 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 chapter Chapter 5 [Parallel FFTW], page 45. Fortran programmers can also use FFTW, as described in Chapter 6 [Calling FFTW from Fortran], page 49. Chapter 8 [Installation and Customization], page 57 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 10 [License and Copyright], page 65. Finally, we thank all the people who helped us in Chapter 9 [Acknowledgments], page 63.

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 8 [Installation and Customization], page 57.)

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 7 [Upgrading from FFTW version 2], page 53.

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);
}
```

(When you compile, you must also link with the `fftw3` library, e.g. `-lfftw3 -lm` on Unix systems.)

First you allocate 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 3.1.1 [SIMD alignment and fftw_malloc], page 15).

---

[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. The data in the `in`/`out` arrays is *overwritten* during `FFTW_MEASURE` planning, so such planning should be done *before* the input is initialized by the user.

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);
```

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);
```

Arrays allocated with `fftw_malloc` should be deallocated by `fftw_free` rather than the ordinary `free` (or, heaven forbid, `delete`).

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.

Users should note that 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 4.7 [What FFTW Really Computes], page 39.

If you have a C compiler, such as `gcc`, that supports the recent 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.)

Single and long-double precision versions of FFTW may be installed; to use them, 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`). The only difference is the routine you use to create the plan:

```
fftw_plan fftw_plan_dft_2d(int nx, int ny,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_3d(int nx, int ny, int nz,
                           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);
```

These routines create plans for `nx` by `ny` two-dimensional (2d) transforms, `nx` by `ny` by `nz` 3d transforms, and arbitrary `rank`-dimensional transforms, respectively. In the third case, `n` is a pointer to an array `n[rank]` denoting an `n[0]` by `n[1]` by . . . by `n[rank-1]` transform. 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 16.

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 `nx` and/or `ny` 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 *destroying 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`), 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. 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 10), 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 nx, int ny,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_3d(int nx, int ny, int nz,
                               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);
```

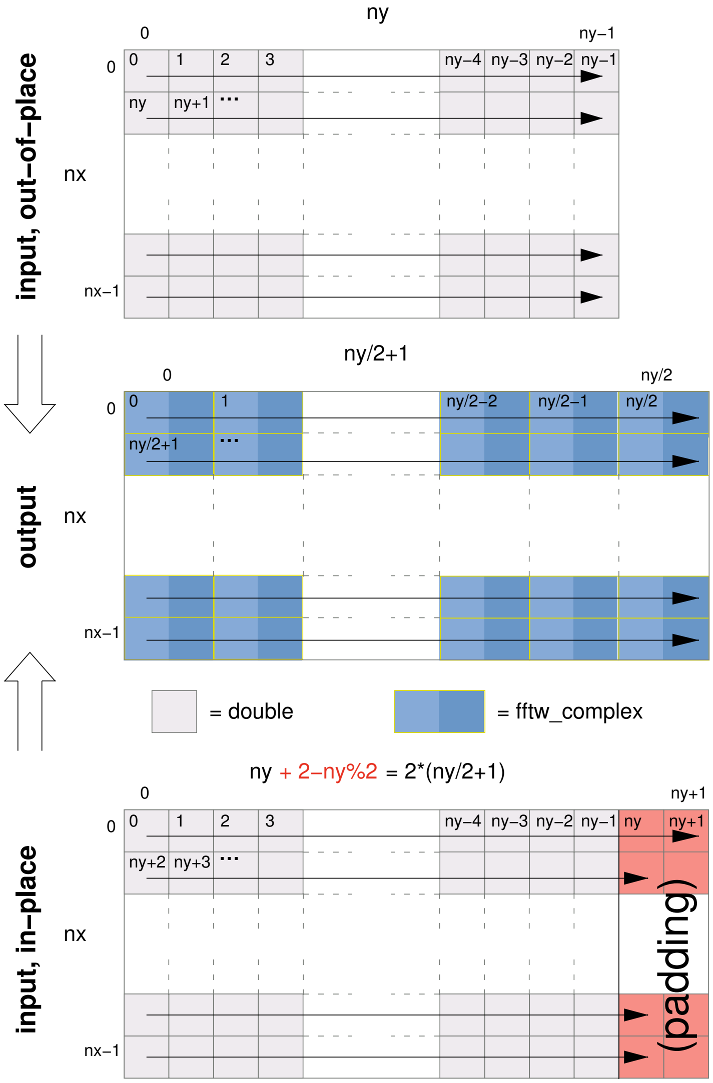
as well as the corresponding `c2r` routines with the input/output types swapped. These routines work similarly to their complex analogues, except for the fact that here the complex output array is cut roughly in half and the real array requires padding for in-place transforms (as in 1d, above).

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 $n_1 \times n_2 \times n_3 \times \cdots \times n_d$ (in row-major order). Then, after an r2c transform, the output is an $n_1 \times n_2 \times n_3 \times \cdots \times (n_d/2+1)$ array of `fftw_complex` values in row-major order, corresponding to slightly over half of the output of the corresponding complex DFT. (The division is rounded down.) The ordering of the data is otherwise exactly the same as in the complex-DFT case.

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

For example, consider the transform of a two-dimensional real array of size `nx` by `ny`. The output of the r2c transform is a two-dimensional complex array of size `nx` by `ny/2+1`, where the `y` dimension has been cut nearly in half because of redundancies in the output. Because `fftw_complex` is twice the size of `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 `nx` by `2*(ny/2+1)`. If `ny` is even, then there are two padding elements at the end of each row (which need not be initialized, as they are only used for output).

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



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 nx, int ny, double *in, double *out,
                           fftw_r2r_kind kindx, fftw_r2r_kind kindy,
                           unsigned flags);
fftw_plan fftw_plan_r2r_3d(int nx, int ny, int nz,
                           double *in, double *out,
                           fftw_r2r_kind kindx,
                           fftw_r2r_kind kindy,
                           fftw_r2r_kind kindz,
                           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 `nx` by `ny`, r2hc by r2hc transform planned by `fftw_plan_r2r_2d(nx, ny, in, out, FFTW_R2HC, FFTW_R2HC, FFTW_MEASURE)`. Conceptually, FFTW first transforms the rows (of size `ny`) to produce halfcomplex rows, and then transforms the columns (of size `nx`). 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.[2]

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 4.7 [What FFTW Really Computes], page 39. (For people accustomed to the

---

[2] 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).[3]

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 8.4 [Generating your own code], page 60).

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

2.5.3 The Discrete Hartley Transform

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`.

If you are planning to use the DHT because you've heard that it is "faster" than the DFT (FFT), **stop here**. That story is an old but enduring misconception that was debunked in 1987: a properly designed real-input FFT (such as FFTW's) has no more operations in general than an FHT. Moreover, in FFTW, the DHT is ordinarily *slower* than the DFT for composite sizes (see below).

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

---

[3] 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.

an r2hc transform, so there is ordinarily no reason to prefer it from a performance perspective.[4] 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 4.7 [What FFTW Really Computes], page 39.

---

[4] 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 Data Alignment

In order to get the best performance from FFTW, one needs to be somewhat aware of two problems related to data alignment on x86 (Pentia) architectures: alignment of allocated arrays (for use with SIMD acceleration), and alignment of the stack.

3.1.1 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 (single/double precision) on Pentium III/IV and higher, 3DNow! (single precision) on the AMD K7 and higher, and AltiVec (single precision) on the PowerPC G4 and higher. 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.

3.1.2 Stack alignment on x86

On the Pentium and subsequent x86 processors, there is a substantial performance penalty if double-precision variables are not stored 8-byte aligned; a factor of two or more is not unusual. Unfortunately, the stack (the place that local variables and subroutine arguments live) is not guaranteed by the Intel ABI to be 8-byte aligned.

Recent versions of `gcc` (as well as most other compilers, we are told, such as Intel's, Metrowerks', and Microsoft's) are able to keep the stack 8-byte aligned; `gcc` does this by default (see `-mpreferred-stack-boundary` in the `gcc` documentation). If you are not certain whether your compiler maintains stack alignment by default, it is a good idea to make sure.

Unfortunately, `gcc` only *preserves* the stack alignment—as a result, if the stack starts off misaligned, it will always be misaligned, with a disastrous effect on performance (in double precision). To prevent this, FFTW includes hacks to align its own stack if necessary, so it should perform well even if you call it from a program with a misaligned stack. Currently, our hacks support `gcc` and the Intel C compiler; if you use another compiler you are on your own. Fortunately, recent versions of glibc (on GNU/Linux) provide a properly-aligned starting stack, but this was not the case with a number of older versions, and we are not certain of the situation on other operating systems. Hopefully, as time goes by this will become less of a concern.

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 among users.

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

Note that, 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 Fortran interface (see Chapter 6 [Calling FFTW from Fortran], page 49).

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[NX][NY][NZ];
        fftw_plan plan;
        ...
        plan = fftw_plan_dft_3d(NX, NY, NZ, &data[0][0][0], &data[0][0][0],
                                FFTW_FORWARD, FFTW_ESTIMATE);
        ...
    }
```

This will plan a 3d in-place transform of size `NX x NY x NZ`. 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 3.1.1 [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 5x12x27 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:

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

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

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

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 4.6 [Wisdom], page 37.

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 4.6.4 [Wisdom Utilities], page 38.

4 FFTW Reference

This chapter provides a complete reference for all sequential (i.e., one-processor) FFTW functions. For parallel transforms, See Chapter 5 [Parallel FFTW], page 45.

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 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 8 [Installation and Customization], page 57). 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).

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 [Data Alignment], 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.

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 guru interface. See Section 4.5 [Guru Interface], page 32.

`fftw_execute` (and equivalents) is the only function in FFTW guaranteed to be thread-safe; see Section 5.2 [Thread safety], page 47.

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; you should still call `fftw_destroy_plan` for this purpose.

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);
```

This outputs a "nerd-readable" representation of the `plan` to the given file or to `stdout`, respectively.

4.3 Basic Interface

The basic interface, which we expect to satisfy the needs of most users, provides planner routines for transforms of a single contiguous array with any of FFTW's supported transform types.

4.3.1 Complex DFTs

```
fftw_plan fftw_plan_dft_1d(int n,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_2d(int nx, int ny,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_3d(int nx, int ny, int nz,
                           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. 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.

Arguments

- `rank` is the dimensionality of the transform (it should be the size of the array `*n`), 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 transform of size 1, i.e. a copy of one number from input to output.
- `n`, or `nx`/`ny`/`nz`, or `n[rank]`, respectively, gives the size of the transform dimensions. They can be any positive integer.
  - Multi-dimensional arrays are stored in row-major order with dimensions: `nx` x `ny`; or `nx` x `ny` x `nz`; or `n[0]` x `n[1]` x ... x `n[rank-1]`. See Section 3.2 [Multi-dimensional Array Format], page 16.
  - 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 8 [Installation and Customization], page 57. 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 24.

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 4.7 [What FFTW Really Computes], page 39.

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.

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.

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 guru 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.)

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 n,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_2d(int nx, int ny,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_3d(int nx, int ny, int nz,
                               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 dimensionality of the transform (it should be the size of the array `*n`), 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 transform of size 1, i.e. a copy of one number (with zero imaginary part) from input to output.
- `n`, or `nx`/`ny`/`nz`, or `n[rank]`, respectively, gives the size of the *logical* 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 27.
    - 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 8 [Installation and Customization], page 57.) 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 27.

- `flags` is a bitwise OR ('`|`') of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 24.

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 n,
                               fftw_complex *in, double *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_c2r_2d(int nx, int ny,
                               fftw_complex *in, double *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_c2r_3d(int nx, int ny, int nz,
                               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 4.7 [What FFTW Really Computes], page 39.

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 4.7 [What FFTW Really Computes], page 39). (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 $n_1 \times n_2 \times n_3 \times \cdots \times n_d$, the complex data is an $n_1 \times n_2 \times n_3 \times \cdots \times (n_d/2+1)$ array of `fftw_complex` values in row-major order (with the division rounded down). That is, we only store the *lower* half (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 $n_1 \times n_2 \times n_3 \times \cdots \times n_d$ 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 $2(n_d/2+1)$ `double` values (exactly enough to hold the complex data). This physical array size does not, however, change the *logical* array size—only n_d values are actually stored in the last dimension, and n_d 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 nx, int ny, double *in, double *out,
                           fftw_r2r_kind kindx, fftw_r2r_kind kindy,
                           unsigned flags);
fftw_plan fftw_plan_r2r_3d(int nx, int ny, int nz,
                           double *in, double *out,
                           fftw_r2r_kind kindx,
                           fftw_r2r_kind kindy,
                           fftw_r2r_kind kindz,
                           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 `nx`/`ny`/`nz`, 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: `nx` x `ny`; or `nx` x `ny` x `nz`; or `n[0]` x `n[1]` x ... x `n[rank-1]`. See Section 3.2 [Multi-dimensional Array Format], page 16.
  - 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 8 [Installation and Customization], page 57.) 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 `kindx`/`kindy`/`kindz`, 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 29. 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 24.

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 4.7 [What FFTW Really Computes], page 39. For a more colloquial introduction to these transform kinds, see Section 2.5 [More DFTs of Real Data], page 9.

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 plans multidimensional complex DFTs, and is exactly the same as `fftw_plan_dft` except for the new parameters `howmany`, `{i,o}nembed`, `{i,o}stride`, and `{i,o}dist`.

`howmany` is the number of transforms to compute, where the `k`-th transform is of the arrays starting at `in+k*idist` and `out+k*odist`. The resulting plans 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).

The two `nembed` parameters (which should be arrays of length `rank`) indicate the sizes of the input and output array dimensions, respectively, where the transform is of a subarray of size `n`. (Each dimension of `n` should be `<=` the corresponding dimension of the `nembed` arrays.) That is, the input and output arrays are stored in row-major order with size given by `nembed` (not counting the strides and howmany multiplicities). 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.

So, for example, to transform a sequence of contiguous arrays, stored one after another, one would use a `stride` of 1 and a `dist` of N, where N is the product of the dimensions. In another example, to transform an array of contiguous "vectors" of length M, one would use a `howmany` of M, a `stride` of M, and a `dist` of 1.

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 27. 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.

In addition to the more flexible planner interface, there are also guru generalizations of `fftw_execute` that allow a given plan to be executed on a different array (sufficiently similar to the original). Note that in order to use the guru execute functions, you do not need to use the guru planner interface: you can create plans with the basic or advanced interface and execute them with the guru interface.

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 16) 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 24.

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 32. 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 27 and generalized in Section 4.4.2 [Advanced Real-data DFTs], page 31. (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.

`flags` is a bitwise OR (‘`|`’) of zero or more planner flags, as defined in Section 4.3.2 [Planner Flags], page 24.

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 29.

`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 24.

4.5.6 Guru Execution of Plans

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 to apply a given plan to a *different* array using the guru 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.

If you are tempted to use this guru interface because you want to transform a known bunch of arrays of the same size, **stop here** and go use the advanced interface instead (see Section 4.4 [Advanced Interface], page 30)).

The guru 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.

4.6 Wisdom

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

4.6.1 Wisdom Export

```
     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 two "wrapper" routines are provided:

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

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

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

4.6.2 Wisdom Import

```
int fftw_import_system_wisdom(void);
int fftw_import_wisdom_from_file(FILE *input_file);
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 two "wrapper" routines are provided:

`fftw_import_wisdom_from_file` reads wisdom from the current position in `input_file`, which should be open with read permission. Upon exit, the file remains open, 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.

4.6.3 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.)

4.6.4 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 19), 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).

4.7 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.

4.7.1 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 $(n-k)/n$.)

4.7.2 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.

4.7.3 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 10. 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.

4.7.4 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 10. 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.

4.7.5 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.

4.7.6 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 $0 \leq j_s < n_s$ 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 $n_1 \times n_2 \times n_3 \times \cdots \times n_d$ multi-dimensional real-input DFT, the full (logical) complex output array $Y[k_1, k_2, \ldots, k_d]$ has the symmetry:

$$Y[k_1, k_2, \ldots, k_d] = Y[n_1 - k_1, n_2 - k_2, \ldots, n_d - k_d]^*$$

(where each dimension is periodic). Because of this symmetry, we only store the $k_d = 0 \cdots n_d/2$ 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 27.

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 10. 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 12.

5 Parallel FFTW

In this chapter we discuss the use of FFTW in a parallel environment.

Currently, FFTW 3 includes parallel transforms for shared-memory machines with some flavor of threads (e.g. POSIX threads); any program using FFTW can be trivially modified to use these transforms, which are documented in Section 5.1 [Multi-threaded FFTW], page 45.

Users calling FFTW from a multi-threaded program should also consult Section 5.2 [Thread safety], page 47. This section tells you which routines of FFTW it is safe to call in parallel from different shared-memory threads.

FFTW 2 also contains distributed-memory parallel transforms using the MPI message-passing standard. MPI transforms are not yet available in FFTW 3, so users requiring that capability must use FFTW 2 for now.

5.1 Multi-threaded FFTW

In this section we document the parallel FFTW routines for shared-memory threads on SMP hardware. These routines, which support parallel one- and multi-dimensional transforms of both real and complex data, are the easiest way to take advantage of multiple processors with FFTW. They work just like the corresponding uniprocessor transform routines, except that 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.

5.1.1 Installation and Supported Hardware/Software

All of the FFTW threads code is located in the `threads` subdirectory of the FFTW package. On Unix systems, the FFTW threads libraries and header files can be automatically configured, compiled, and installed along with the uniprocessor FFTW libraries simply by including `--enable-threads` in the flags to the `configure` script (see Section 8.1 [Installation on Unix], page 57).

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

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

5.1.2 Usage of Multi-threaded FFTW

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

First, programs using the parallel complex transforms should be linked with `-lfftw3_threads -lfftw3 -lm` on Unix. You will also need to link with whatever library is responsible for threads on your system (e.g. `-lpthread` on GNU/Linux).

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.

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`.

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.

5.1.3 How Many Threads to Use?

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

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

You will have to experiment with your system to see what level of parallelization is best for your problem size. 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.

5.2 Thread safety

Users writing multi-threaded programs must concern themselves with the *thread safety* of the libraries they use—that is, whether it is safe to call routines in parallel from multiple threads. FFTW can be used in such an environment, but some care must be taken because 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 guru 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 guru variants (see Section 4.5.6 [Guru Execution of Plans], page 36) so that different threads compute the transform of different data.

(Users should note that these comments only apply to programs using shared-memory threads. Parallelism using MPI or forked processes involves a separate address-space and global variables for each process, and is not susceptible to problems of this sort.)

6 Calling FFTW from Fortran

This chapter describes the Fortran-callable interface to FFTW, which differs from the C interface only in the prefix (‘`dfftw_`’ instead of ‘`fftw_`’), and a few other minor details. The 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.

6.1 Fortran-interface routines

Nearly all of the FFTW functions have Fortran-callable equivalents. The name of the Fortran routine is the same as that of the corresponding C routine, but with the ‘`fftw_`’ prefix replaced by ‘`dfftw_`’. (The single and long-double precision versions use ‘`sfftw_`’ and ‘`lfftw_`’, respectively, instead of ‘`fftwf_`’ and ‘`fftwl_`’.)[1]

For the most part, all of the arguments to the functions are the same, with the following exceptions:

- `plan` variables (what would be of type `fftw_plan` 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`.
- 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 16). 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 6.4 [Wisdom of Fortran?], page 51.
- Fortran cannot use the `fftw_malloc` dynamic-allocation routine. If you want to exploit the SIMD FFTW (see Section 3.1 [Data Alignment], 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 32) 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.

In general, you should take care to use Fortran data types that correspond to (i.e. are the same size as) the C types used by FFTW. If your C and Fortran compilers are made by the same vendor, the correspondence is usually straightforward (i.e. `integer` corresponds

---

[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.

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.

6.2 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.)

6.3 Fortran Examples

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

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

plan = fftw_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(plan)
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`. To do the same thing, but using 8 threads in parallel (see Section 5.1 [Multi-threaded FFTW], page 45), you would simply prefix these calls with:

```
call dfftw_init_threads
call dfftw_plan_with_nthreads(8)
```

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(plan)
             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(plan)
             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(plan)
             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.

6.4 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.)

7 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 6.4 [Wisdom of Fortran?], page 51).

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 5.2 [Thread safety], page 47). *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 Section 5.1 [Multi-threaded FFTW], page 45.

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).

8 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 8.3 [Cycle Counters], page 60). (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.

8.1 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-portable-binary`: Disable compiler optimizations that would produce unportable binaries. **Important:** Use this if you are distributing compiled binaries to people who many not use exactly the same processor as you.
- `--with-gcc-arch=`*arch*: When compiling with `gcc`, FFTW tries to deduce the current CPU in order to tell `gcc` what architecture to tune for; this option overrides that guess (i.e. *arch* should be a valid argument for `gcc`'s `-march` or `-mtune` flags). You might do this because the deduced architecture was wrong or because you want to tune for a different CPU than the one you are compiling with. You can use `--without-gcc-arch` to disable architecture-specific tuning entirely. Note that if `--enable-portable-binary` is enabled (above), then we use `-mtune` but not `-march`, so the resulting binary will run on any architecture even though it is optimized for a particular one.
- `--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 is `long double` is the same size as `double` on your machine/compiler. See Section 4.1.2 [Precision], page 21.
- `--enable-threads`: Enables compilation and installation of the FFTW threads library (see Section 5.1 [Multi-threaded FFTW], page 45), which provides a simple interface to parallel transforms for SMP systems. (By default, the threads routines are not compiled.)
- `--with-openmp`, `--with-sgimp`: In conjunction with `--enable-threads`, causes the multi-threaded FFTW library to use OpenMP or SGI MP compiler directives in order to induce parallelism, rather than spawning its own threads directly. (Useful especially for programs already employing such directives, in order to minimize conflicts between different parallelization mechanisms.)
- `--disable-fortran`: Disables inclusion of Fortran-callable wrapper routines (see Chapter 6 [Calling FFTW from Fortran], page 49) 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-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 8.3 [Cycle Counters], page 60.
- `--enable-sse`, `--enable-sse2`, `--enable-k7`, `--enable-altivec`: Enable the compilation of SIMD code for SSE (Pentium III+), SSE2 (Pentium IV+), 3dNow! (AMD

K7 and others), or AltiVec (PowerPC G4+). SSE, 3dNow!, and AltiVec only work with `--enable-float` (above), while SSE2 only works in double precision (the default). 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, with the exception of `--enable-k7` (which uses assembly), require a compiler supporting SIMD extensions, and compiler support is still a bit flaky: see the FFTW FAQ for a list of compiler versions that have problems compiling FFTW.
  - With the Linux kernel, you may have to recompile the kernel with the option to support SSE/SSE2/AltiVec (see the "Processor type and features" settings).
  - 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.

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.

8.2 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/codelets`, `dft/codelets/standard`, `dft/codelets/inplace`, `rdft`, `rdft/codelets`, `rdft/codelets/r2hc`, `rdft/codelets/hc2r`, `rdft/codelets/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`, `dft/simd`, and `dft/simd/codelets` directories. If you are compiling with AMD K7 optimizations (i.e. you defined `HAVE_K7`), then you also need to include the `dft/k7` and `dft/k7/codelets` directories. (See the previous section for more information on configuration options like

SIMD and K7 optimization; each Unix configuration option has a corresponding `#define` in `config.h`.)

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 4.6.4 [Wisdom Utilities], page 38), compile `tools/fftw-wisdom.c` and link it to the `libbench2` and FFTW libraries

8.3 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.

8.4 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://pauillac.inria.fr/ocaml/`.

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/codelets/*/Makefile.am`, `dft/simd/codelets/Makefile.am`, `dft/k7/codelets/Makefile.am`, and `rdft/codelets/*/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)*.

9 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.

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 "extra-curricular" 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.

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.

10 License and Copyright

FFTW is Copyright © 2003 Matteo Frigo, Copyright © 2003 Massachusetts Institute of Technology.

FFTW is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. You can also find the GPL on the GNU web site.

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.

11 Concept Index

3

3DNow! 15

A

advanced interface 1, 5, 16, 30
algorithm 2
alignment 15, 22, 25
AltiVec 15

B

basic interface 1, 3, 23

C

C multi-dimensional arrays 16
C++ 5, 15, 17, 21, 22
c2r 6, 25, 27
C99 . . . . . 17, 21, 22
Caml 60, 63
code generator 2, 60
codelet 2, 57, 60, 63
column-major 16, 49, 51
compiler 2, 57, 59, 60
compiler flags 57, 59
configuration routines 39
`configure` 45, 57
cycle counter 57, 60

D

DCT 10, 29, 40
Devil 3
DFT 1, 4, 39
DHT 12, 43
discrete cosine transform 10, 29, 40
discrete Fourier transform 1, 39
discrete Hartley transform 12, 29, 43
discrete sine transform 10, 30, 42
dist 30, 33
DST 10, 30, 42

E

Ecclesiastes 19
execute 1, 4, 36

F

FFTW 1
fftw-wisdom utility 19, 39
fftw-wisdom-to-conf utility 39
flags 4, 6, 24, 27, 29, 34, 35, 36, 50
Fortran interface 16, 49
Fortran-callable wrappers 58
frequency 4, 39

G

guru interface 1, 5, 32, 49

H

halfcomplex format 7, 10, 40
hc2r 10, 25
Hermitian 6, 40
howmany loop 33
howmany parameter 30

I

IDCT 11, 29, 41
in-place 4, 6, 24, 26, 28, 29, 35
installation 57
interleaved format 32

K

kind (r2r) 9, 29

L

linking on Unix 46
LISP 63

M

monadic programming 61

N

normalization . . 4, 8, 10, 11, 12, 24, 27, 29, 39, 40, 41, 43
number of threads 46

O

out-of-place 25, 27

P

padding 6, 7, 27, 28
parallel transform 45
partial order 5
plan 1, 4

planner 1
portability 15, 19, 21, 45, 49, 50, 51, 57, 58
precision 5, 6, 15, 21, 58

R

r2c 6, 10, 26, 44
r2c/c2r multi-dimensional array format . . 7, 27, 51
r2hc 10
r2r 9, 28, 40
rank 5
real-even DFT 10, 40
real-odd DFT 10, 42
REDFT 10, 40, 61
RODFT 10, 42
row-major 16, 24, 28

S

saving plans to disk 18, 37
SIMD 3, 15
split format 32
SSE 15
SSE2 15
stride 16, 30, 33

T

thread safety 47
threads 45, 47, 58

V

vector 31, 32

W

wisdom 18, 37
wisdom, problems with 19
wisdom, system-wide 19, 38

12 Library Index

D

dfftw_destroy_plan . . . 50
dfftw_execute . . . 50
dfftw_export_wisdom . . . 52
dfftw_forget_wisdom . . . 52
dfftw_import_system_wisdom . . . 52
dfftw_import_wisdom . . . 52
dfftw_init_threads . . . 50
dfftw_plan_dft_1d . . . 50
dfftw_plan_dft_3d . . . 51
dfftw_plan_dft_r2c_1d . . . 51
dfftw_plan_dft_r2c_2d . . . 51
dfftw_plan_with_nthreads . . . 50

F

FFTW_BACKWARD . . . 4, 6
fftw_cleanup . . . 23
fftw_cleanup_threads . . . 46
fftw_complex . . . 4, 21
FFTW_DESTROY_INPUT . . . 25
fftw_destroy_plan . . . 4, 22
FFTW_DHT . . . 12, 29
FFTW_ESTIMATE . . . 4, 18, 25, 60
fftw_execute . . . 4, 22, 36
fftw_execute_dft . . . 37
fftw_execute_dft_c2r . . . 37
fftw_execute_dft_r2c . . . 37
fftw_execute_dft_r2r . . . 37
fftw_execute_split_dft . . . 37
fftw_execute_split_dft_c2r . . . 37
fftw_execute_split_dft_r2c . . . 37
FFTW_EXHAUSTIVE . . . 18, 25
fftw_export_wisdom . . . 37
fftw_export_wisdom_to_file . . . 18, 37
fftw_export_wisdom_to_string . . . 37
fftw_flops . . . 23
fftw_forget_wisdom . . . 19, 38
FFTW_FORWARD . . . 4, 6
fftw_fprint_plan . . . 23
fftw_free . . . 4, 15, 22
FFTW_HC2R . . . 10, 29
fftw_import_system_wisdom . . . 19, 38
fftw_import_wisdom . . . 38
fftw_import_wisdom_from_file . . . 18, 38
fftw_import_wisdom_from_string . . . 38
fftw_init_threads . . . 46
fftw_iodim . . . 33, 49
fftw_malloc . . . 3, 15, 17, 22
FFTW_MEASURE . . . 4, 18, 25
FFTW_NO_TIMELIMIT . . . 25
FFTW_PATIENT . . . 5, 18, 25, 46
fftw_plan . . . 4, 22
fftw_plan_dft . . . 5, 23
fftw_plan_dft_1d . . . 4, 23
fftw_plan_dft_2d . . . 5, 23
fftw_plan_dft_3d . . . 5, 23
fftw_plan_dft_c2r . . . 27
fftw_plan_dft_c2r_1d . . . 6, 27
fftw_plan_dft_c2r_2d . . . 27
fftw_plan_dft_c2r_3d . . . 27
fftw_plan_dft_r2c . . . 7, 26
fftw_plan_dft_r2c_1d . . . 6, 26
fftw_plan_dft_r2c_2d . . . 7, 26
fftw_plan_dft_r2c_3d . . . 7, 26
fftw_plan_guru_dft . . . 33
fftw_plan_guru_dft_c2r . . . 35
fftw_plan_guru_dft_r2c . . . 35
fftw_plan_guru_r2r . . . 35
fftw_plan_guru_split_dft . . . 33
fftw_plan_guru_split_dft_c2r . . . 35
fftw_plan_guru_split_dft_r2c . . . 35
fftw_plan_many_dft . . . 30
fftw_plan_many_dft_c2r . . . 31
fftw_plan_many_dft_r2c . . . 31
fftw_plan_many_r2r . . . 32
fftw_plan_r2r . . . 9, 28
fftw_plan_r2r_1d . . . 9, 28
fftw_plan_r2r_2d . . . 9, 28
fftw_plan_r2r_3d . . . 9, 28
fftw_plan_with_nthreads . . . 46
FFTW_PRESERVE_INPUT . . . 6, 25
fftw_print_plan . . . 23
FFTW_R2HC . . . 10, 29
fftw_r2r_kind . . . 9
FFTW_REDFT00 . . . 11, 28, 29
FFTW_REDFT01 . . . 11, 29
FFTW_REDFT10 . . . 11, 29
FFTW_REDFT11 . . . 11, 30
FFTW_RODFT00 . . . 11, 30
FFTW_RODFT01 . . . 11, 30
FFTW_RODFT10 . . . 11, 30
FFTW_RODFT11 . . . 11, 30
fftw_set_timelimit . . . 25
FFTW_UNALIGNED . . . 25, 36

R

R2HC . . . 40
REDFT00 . . . 40, 41
REDFT01 . . . 41
REDFT10 . . . 41
REDFT11 . . . 41
RODFT00 . . . 42
RODFT01 . . . 42
RODFT10 . . . 42
RODFT11 . . . 42