pyFFTW Documentation

Release 0.11.1+3.g898bce5

Henry Gomersall

Nov 27, 2018

Contents

1 Introduction **1**

2 Contents **3**

2.1 Overview and A Short Tutorial 3

2.2 License 9

2.3 API Reference 10

3 Indices and tables **35**

Python Module Index **37**

CHAPTER 1

Introduction

pyFFTW is a pythonic wrapper around FFTW, the speedy FFT library. The ultimate aim is to present a unified interface for all the possible transforms that FFTW can perform.

Both the complex DFT and the real DFT are supported, as well as on arbitrary axes of abitrary shaped and strided arrays, which makes it almost feature equivalent to standard and real FFT functions of `numpy.fft` (indeed, it supports the `clongdouble` dtype which `numpy.fft` does not).

Operating FFTW in multithreaded mode is supported.

The core interface is provided by a unified class, *`pyfftw.FFTW`*. This core interface can be accessed directly, or through a series of helper functions, provided by the *`pyfftw.builders`* module. These helper functions provide an interface similar to `numpy.fft` for ease of use.

In addition to using *`pyfftw.FFTW`*, a convenient series of functions are included through *`pyfftw.interfaces`* that make using *`pyfftw`* almost equivalent to `numpy.fft` or `scipy.fftpack`.

The source can be found in github and its page in the python package index is here.

A comprehensive unittest suite is included with the source on the repository. If any aspect of this library is not covered by the test suite, that is a bug (please report it!).

CHAPTER 2

Contents

2.1 Overview and A Short Tutorial

Before we begin, we assume that you are already familiar with the discrete Fourier transform, and why you want a faster library to perform your FFTs for you.

FFTW is a very fast FFT C library. The way it is designed to work is by planning *in advance* the fastest way to perform a particular transform. It does this by trying lots of different techniques and measuring the fastest way, so called *planning*.

One consequence of this is that the user needs to specify in advance exactly what transform is needed, including things like the data type, the array shapes and strides and the precision. This is quite different to how one uses, for example, the `numpy.fft` module.

The purpose of this library is to provide a simple and pythonic way to interact with FFTW, benefiting from the substantial speed-ups it offers. In addition to the method of using FFTW as described above, a convenient series of functions are included through *`pyfftw.interfaces`* that make using *`pyfftw`* almost equivalent to `numpy.fft`.

This tutorial is split into three parts. A quick introduction to the *`pyfftw.interfaces`* module is *given*, the most simple and direct way to use *`pyfftw`*. Secondly an *overview* is given of *`pyfftw.FFTW`*, the core of the library. Finally, the *`pyfftw.builders`* helper functions are *introduced*, which ease the creation of *`pyfftw.FFTW`* objects.

2.1.1 Quick and easy: the `pyfftw.interfaces` module

The easiest way to begin using *`pyfftw`* is through the *`pyfftw.interfaces`* module. This module implements three APIs: *`pyfftw.interfaces.numpy_fft`*, *`pyfftw.interfaces.scipy_fftpack`*, and *`pyfftw.interfaces.dask_fft`*, which are (apart from a small caveat[1]) drop in replacements for `numpy.fft`, `scipy.fftpack`, and `dask.fft` respectively.

[1] *`pyfftw.interfaces`* deals with repeated values in the `axes` argument differently to `numpy.fft` (and probably to `scipy.fftpack` to, but that's not documented clearly). Specifically, `numpy.fft` takes the transform along a given axis as many times as it appears in the `axes` argument. *`pyfftw.interfaces`* takes the transform only once along each axis that appears, regardless of how many times it appears. This is deemed to be such a fringe corner case that it is ignored.

```
>>> import pyfftw
>>> import numpy
>>> a = pyfftw.empty_aligned(128, dtype='complex128', n=16)
>>> a[:] = numpy.random.randn(128) + 1j*numpy.random.randn(128)
>>> b = pyfftw.interfaces.numpy_fft.fft(a)
>>> c = numpy.fft.fft(a)
>>> numpy.allclose(b, c)
True
```

We initially create and fill a complex array, `a`, of length 128. *`pyfftw.empty_aligned()`* is a helper function that works like `numpy.empty()` but returns the array aligned to a particular number of bytes in memory, in this case 16. If the alignment is not specified then the library inspects the CPU for an appropriate alignment value. Having byte aligned arrays allows FFTW to performed vector operations, potentially speeding up the FFT (a similar *`pyfftw.byte_align()`* exists to align a pre-existing array as necessary).

Calling *`pyfftw.interfaces.numpy_fft.fft()`* on `a` gives the same output (to numerical precision) as calling `numpy.fft.fft()` on `a`.

If you wanted to modify existing code that uses `numpy.fft` to use *`pyfftw.interfaces`*, this is done simply by replacing all instances of `numpy.fft` with *`pyfftw.interfaces.numpy_fft`* (similarly for `scipy.fftpack` and *`pyfftw.interfaces.scipy_fftpack`*), and then, optionally, enabling the cache (see below).

The first call for a given transform size and shape and dtype and so on may be slow, this is down to FFTW needing to plan the transform for the first time. Once this has been done, subsequent equivalent transforms during the same session are much faster. It's possible to export and save the internal knowledge (the *wisdom*) about how the transform is done. This is described *below*.

Even after the first transform of a given specification has been performed, subsequent transforms are never as fast as using *`pyfftw.FFTW`* objects directly, and in many cases are substantially slower. This is because of the internal overhead of creating a new *`pyfftw.FFTW`* object on every call. For this reason, a cache is provided, which is recommended to be used whenever *`pyfftw.interfaces`* is used. Turn the cache on using *`pyfftw.interfaces.cache.enable()`*. This function turns the cache on globally. Note that using the cache invokes the threading module.

The cache temporarily stores a copy of any interim *`pyfftw.FFTW`* objects that are created. If they are not used for some period of time, which can be set with *`pyfftw.interfaces.cache.set_keepalive_time()`*, then they are removed from the cache (liberating any associated memory). The default keepalive time is 0.1 seconds.

Monkey patching 3rd party libraries

Since *`pyfftw.interfaces.numpy_fft`* and *`pyfftw.interfaces.scipy_fftpack`* are drop-in replacements for their `numpy.fft` and `scipy.fftpack` libraries respectively, it is possible to use them as replacements at run-time through monkey patching. Note that the interfaces (and builders) all currently default to a single thread. The number of threads to use can be configured by assigning a positive integer to *pyfftw.config.NUM_THREADS* (see more details under :ref:configuration <interfaces_tutorial>).

The following code demonstrates `scipy.signal.fftconvolve()` being monkey patched in order to speed it up.

```
import pyfftw
import multiprocessing
import scipy.signal
import numpy
from timeit import Timer

a = pyfftw.empty_aligned((128, 64), dtype='complex128')
```

(continues on next page)

(continued from previous page)

```
b = pyfftw.empty_aligned((128, 64), dtype='complex128')

a[:] = numpy.random.randn(128, 64) + 1j*numpy.random.randn(128, 64)
b[:] = numpy.random.randn(128, 64) + 1j*numpy.random.randn(128, 64)

t = Timer(lambda: scipy.signal.fftconvolve(a, b))

print('Time with scipy.fftpack: %1.3f seconds' % t.timeit(number=100))

# Configure PyFFTW to use all cores (the default is single-threaded)
pyfftw.config.NUM_THREADS = multiprocessing.cpu_count()

# Monkey patch fftpack with pyfftw.interfaces.scipy_fftpack
scipy.fftpack = pyfftw.interfaces.scipy_fftpack
scipy.signal.fftconvolve(a, b) # We cheat a bit by doing the planning first

# Turn on the cache for optimum performance
pyfftw.interfaces.cache.enable()

print('Time with monkey patched scipy_fftpack: %1.3f seconds' %
       t.timeit(number=100))
```

which outputs something like:

```
Time with scipy.fftpack: 0.598 seconds
Time with monkey patched scipy_fftpack: 0.251 seconds
```

Note that prior to Scipy 0.16, it was necessary to patch the individual functions in `scipy.signal.signaltools`. For example:

```
scipy.signal.signaltools.ifftn = pyfftw.interfaces.scipy_fftpack.ifftn
```

2.1.2 The workhorse `pyfftw.FFTW` class

The core of this library is provided through the *`pyfftw.FFTW`* class. FFTW is fully encapsulated within this class.

The following gives an overview of the *`pyfftw.FFTW`* class, but the easiest way to of dealing with it is through the *`pyfftw.builders`* helper functions, also *discussed in this tutorial*.

For users that already have some experience of FFTW, there is no interface distinction between any of the supported data types, shapes or transforms, and operating on arbitrarily strided arrays (which are common when using `numpy`) is fully supported with no copies necessary.

In its simplest form, a *`pyfftw.FFTW`* object is created with a pair of complementary `numpy` arrays: an input array and an output array. They are complementary insomuch as the data types and the array sizes together define exactly what transform should be performed. We refer to a valid transform as a *scheme*.

Internally, three precisions of FFT are supported. These correspond to single precision floating point, double precision floating point and long double precision floating point, which correspond to `numpy`'s `float32`, `float64` and `longdouble` dtypes respectively (and the corresponding complex types). The precision is decided by the relevant scheme, which is specified by the dtype of the input array.

Various schemes are supported by *`pyfftw.FFTW`*. The scheme that is used depends on the data types of the input array and output arrays, the shape of the arrays and the direction flag. For a full discussion of the schemes available, see the API documentation for *`pyfftw.FFTW`*.

One-Dimensional Transforms

We will first consider creating a simple one-dimensional transform of a one-dimensional complex array:

```
import pyfftw

a = pyfftw.empty_aligned(128, dtype='complex128')
b = pyfftw.empty_aligned(128, dtype='complex128')

fft_object = pyfftw.FFTW(a, b)
```

In this case, we create 2 complex arrays, `a` and `b` each of length 128. As before, we use *pyfftw.empty_aligned()* to make sure the array is aligned.

Given these 2 arrays, the only transform that makes sense is a 1D complex DFT. The direction in this case is the default, which is forward, and so that is the transform that is *planned*. The returned `fft_object` represents such a transform.

In general, the creation of the *pyfftw.FFTW* object clears the contents of the arrays, so the arrays should be filled or updated after creation.

Similarly, to plan the inverse:

```
c = pyfftw.empty_aligned(128, dtype='complex128')
ifft_object = pyfftw.FFTW(b, c, direction='FFTW_BACKWARD')
```

In this case, the direction argument is given as `'FFTW_BACKWARD'` (to override the default of `'FFTW_FORWARD'`).

The actual FFT is performed by calling the returned objects:

```
import numpy

# Generate some data
ar, ai = numpy.random.randn(2, 128)
a[:] = ar + 1j*ai

fft_a = fft_object()
```

Note that calling the object like this performs the FFT and returns the result in an array. This is the *same* array as `b`:

```
>>> fft_a is b
True
```

This is particularly useful when using *pyfftw.builders* to generate the *pyfftw.FFTW* objects.

Calling the FFT object followed by the inverse FFT object yields an output that is numerically the same as the original `a` (within numerical accuracy).

```
>>> fft_a = fft_object()
>>> ifft_b = ifft_object()
>>> ifft_b is c
True
>>> numpy.allclose(a, c)
True
>>> a is c
False
```

In this case, the normalisation of the DFT is performed automatically by the inverse FFTW object (`ifft_object`). This can be disabled by setting the `normalise_idft=False` argument.

It is possible to change the data on which a *pyfftw.FFTW* operates. The *pyfftw.FFTW.__call__()* accepts both an `input_array` and an `output_array` argument to update the arrays. The arrays should be compatible with the arrays with which the *pyfftw.FFTW* object was originally created. Please read the API docs on *pyfftw.FFTW.__call__()* to fully understand the requirements for updating the array.

```
>>> d = pyfftw.empty_aligned(4, dtype='complex128')
>>> e = pyfftw.empty_aligned(4, dtype='complex128')
>>> f = pyfftw.empty_aligned(4, dtype='complex128')
>>> fft_object = pyfftw.FFTW(d, e)
>>> fft_object.input_array is d # get the input array from the object
True
>>> f[:] = [1, 2, 3, 4] # Add some data to f
>>> fft_object(f)
array([ 10.+0.j,  -2.+2.j,  -2.+0.j,  -2.-2.j])
>>> fft_object.input_array is d # No longer true!
False
>>> fft_object.input_array is f # It has been updated with f :)
True
```

If the new input array is of the wrong dtype or wrongly strided, *pyfftw.FFTW.__call__()* method will copy the new array into the internal array, if necessary changing it's dtype in the process.

It should be made clear that the *pyfftw.FFTW.__call__()* method is simply a helper routine around the other methods of the object. Though it is expected that most of the time *pyfftw.FFTW.__call__()* will be sufficient, all the FFTW functionality can be accessed through other methods at a slightly lower level.

Multi-Dimensional Transforms

Arrays of more than one dimension are easily supported as well. In this case, the `axes` argument specifies over which axes the transform is to be taken.

```
import pyfftw

a = pyfftw.empty_aligned((128, 64), dtype='complex128')
b = pyfftw.empty_aligned((128, 64), dtype='complex128')

# Plan an fft over the last axis
fft_object_a = pyfftw.FFTW(a, b)

# Over the first axis
fft_object_b = pyfftw.FFTW(a, b, axes=(0,))

# Over the both axes
fft_object_c = pyfftw.FFTW(a, b, axes=(0,1))
```

For further information on all the supported transforms, including real transforms, as well as full documentaion on all the instantiation arguments, see the *pyfftw.FFTW* documentation.

Wisdom

When creating a *pyfftw.FFTW* object, it is possible to instruct FFTW how much effort it should put into finding the fastest possible method for computing the DFT. This is done by specifying a suitable planner flag in `flags` argument to *pyfftw.FFTW*. Some of the planner flags can take a very long time to complete which can be problematic.

When the a particular transform has been created, distinguished by things like the data type, the shape, the stridings and the flags, FFTW keeps a record of the fastest way to compute such a transform in future. This is referred to as

wisdom. When the program is completed, the wisdom that has been accumulated is forgotten.

It is possible to output the accumulated wisdom using the *wisdom output routines*. *pyfftw.export_wisdom()* exports and returns the wisdom as a tuple of strings that can be easily written to file. To load the wisdom back in, use the *pyfftw.import_wisdom()* function which takes as its argument that same tuple of strings that was returned from *pyfftw.export_wisdom()*.

If for some reason you wish to forget the accumulated wisdom, call *pyfftw.forget_wisdom()*.

2.1.3 The pyfftw.builders functions

If you absolutely need the flexibility of dealing with *pyfftw.FFTW* directly, an easier option than constructing valid arrays and so on is to use the convenient *pyfftw.builders* package. These functions take care of much of the difficulty in specifying the exact size and dtype requirements to produce a valid scheme.

The *pyfftw.builders* functions are a series of helper functions that provide an interface very much like that provided by numpy.fft, only instead of returning the result of the transform, a *pyfftw.FFTW* object (or in some cases a wrapper around *pyfftw.FFTW*) is returned.

```
import pyfftw

a = pyfftw.empty_aligned((128, 64), dtype='complex128')

# Generate some data
ar, ai = numpy.random.randn(2, 128, 64)
a[:] = ar + 1j*ai

fft_object = pyfftw.builders.fft(a)

b = fft_object()
```

fft_object is an instance of *pyfftw.FFTW*, b is the result of the DFT.

Note that in this example, unlike creating a *pyfftw.FFTW* object using the direct interface, we can fill the array in advance. This is because by default all the functions in *pyfftw.builders* keep a copy of the input array during creation (though this can be disabled).

The *pyfftw.builders* functions construct an output array of the correct size and type. In the case of the regular DFTs, this always creates an output array of the same size as the input array. In the case of the real transform, the output array is the right shape to satisfy the scheme requirements.

The precision of the transform is determined by the dtype of the input array. If the input array is a floating point array, then the precision of the floating point is used. If the input array is not a floating point array then a double precision transform is used. Any calls made to the resultant object with an array of the same size will then be copied into the internal array of the object, changing the dtype in the process.

Like numpy.fft, it is possible to specify a length (in the one-dimensional case) or a shape (in the multi-dimensional case) that may be different to the array that is passed in. In such a case, a wrapper object of type *pyfftw.builders._utils._FFTWWrapper* is returned. From an interface perspective, this is identical to *pyfftw.FFTW*. The difference is in the way calls to the object are handled. With *pyfftw.builders._utils._FFTWWrapper* objects, an array that is passed as an argument when calling the object is *copied* into the internal array. This is done by a suitable slicing of the new passed-in array and the internal array and is done precisely because the shape of the transform is different to the shape of the input array.

```
a = pyfftw.empty_aligned((128, 64), dtype='complex128')

fft_wrapper_object = pyfftw.builders.fftn(a, s=(32, 256))
```

(continues on next page)

(continued from previous page)

```
b = fft_wrapper_object()
```

Inspecting these objects gives us their shapes:

```
>>> b.shape
(32, 256)
>>> fft_wrapper_object.input_array.shape
(32, 256)
>>> a.shape
(128, 64)
```

It is only possible to call `fft_wrapper_object` with an array that is the same shape as `a`. In this case, the first axis of `a` is sliced to include only the first 32 elements, and the second axis of the internal array is sliced to include only the last 64 elements. This way, shapes are made consistent for copying.

Understanding `numpy.fft`, these functions are largely self-explanatory. We point the reader to the *API docs* for more information.

2.1.4 Configuring FFTW planning effort and number of threads

The user may set the default number of threads used by the interfaces and builders at run time by assigning to `pyfftw.config.NUM_THREADS`. Similarly the default planning effort may be set by assigning a string such as `'FFTW_ESTIMATE'` or `'FFTW_MEASURE'` to `pyfftw.config.PLANNER_EFFORT`.

For example, to change the effort to `'FFTW_MEASURE'` and specify 4 threads:

```
import pyfftw

pyfftw.config.NUM_THREADS = 4

pyfftw.config.PLANNER_EFFORT = 'FFTW_MEASURE'
```

All functions in *pyfftw.interfaces* and *pyfftw.builders* use the values from `pyfftw.config` when determining the default number of threads and planning effort.

The initial values in pyfftw.config at import time can be controlled via the environment variables as detailed in the configuration documentation.

2.2 License

Note: While all the code in pyfftw (except for fftw3.h) is released under the 3-clause BSD license (set out below), pyfftw requires FFTW3 to function. FFTW3 is available under two licenses, the free GPL and a non-free license that allows it to be used in proprietary programs.

If you intend to use the GPLed FFTW3 library, your code must also be GPL licensed. If you do not wish to comply with the terms of the GPL, you have to buy a FFTW3 license from the copyright holder MIT, see here for more information.

---

fftw3.h is released under the 2-clause BSD license.

See each file for the copyright holder.

2.3 API Reference

2.3.1 `pyfftw` - The core

The core of `pyfftw` consists of the *`FFTW`* class, *wisdom functions* and a couple of *utility functions* for dealing with aligned arrays.

This module represents the full interface to the underlying FFTW library. However, users may find it easier to use the helper routines provided in *`pyfftw.builders`*. Default values used by the helper routines can be controlled as via *configuration variables*.

FFTW Class

class pyfftw.**FFTW**(*input_array*, *output_array*, *axes=(-1,)*, *direction='FFTW_FORWARD'*, *flags=('FFTW_MEASURE',)*, *threads=1*, *planning_timelimit=None*)

FFTW is a class for computing the complex N-Dimensional DFT or inverse DFT of an array using the FFTW library. The interface is designed to be somewhat pythonic, with the correct transform being inferred from the dtypes of the passed arrays.

On instantiation, the dtypes and relative shapes of the input array and output arrays are compared to the set of valid (and implemented) *FFTW schemes*. If a match is found, the plan that corresponds to that scheme is created, operating on the arrays that are passed in. If no scheme can be created, then `ValueError` is raised.

The actual FFT or iFFT is performed by calling the *`execute()`* method.

The arrays can be updated by calling the *`update_arrays()`* method.

The created instance of the class is itself callable, and can perform the execution of the FFT, both with or without array updates, returning the result of the FFT. Unlike calling the *`execute()`* method, calling the class instance will also optionally normalise the output as necessary. Additionally, calling with an input array update will also coerce that array to be the correct dtype.

See the documentation on the *`__call__()`* method for more information.

Arguments:

- `input_array` and `output_array` should be numpy arrays. The contents of these arrays will be destroyed by the planning process during initialisation. Information on supported dtypes for the arrays is *given below*.
- `axes` describes along which axes the DFT should be taken. This should be a valid list of axes. Repeated axes are only transformed once. Invalid axes will raise an `IndexError` exception. This argument is equivalent to the same argument in `numpy.fft.fftn()`, except for the fact that the behaviour of repeated axes is different (`numpy.fft` will happily take the fft of the same axis if it is repeated in the `axes` argument). Rudimentary testing has suggested this is down to the underlying FFTW library and so unlikely to be fixed in these wrappers.
- `direction` should be a string and one of `'FFTW_FORWARD'` or `'FFTW_BACKWARD'`, which dictate whether to take the DFT (forwards) or the inverse DFT (backwards) respectively (specifically, it dictates the sign of the exponent in the DFT formulation).

  Note that only the Complex schemes allow a free choice for `direction`. The direction *must* agree with the the *table below* if a Real scheme is used, otherwise a `ValueError` is raised.

- `flags` is a list of strings and is a subset of the flags that FFTW allows for the planners:
  - `'FFTW_ESTIMATE'`, `'FFTW_MEASURE'`, `'FFTW_PATIENT'` and `'FFTW_EXHAUSTIVE'` are supported. These describe the increasing amount of effort spent during the planning stage to create the fastest possible transform. Usually `'FFTW_MEASURE'` is a good compromise. If no flag is passed, the default `'FFTW_MEASURE'` is used.
  - `'FFTW_UNALIGNED'` is supported. This tells FFTW not to assume anything about the alignment of the data and disabling any SIMD capability (see below).
  - `'FFTW_DESTROY_INPUT'` is supported. This tells FFTW that the input array can be destroyed during the transform, sometimes allowing a faster algorithm to be used. The default behaviour is, if possible, to preserve the input. In the case of the 1D Backwards Real transform, this may result in a performance hit. In the case of a backwards real transform for greater than one dimension, it is not possible to preserve the input, making this flag implicit in that case. A little more on this is given *below*.
  - `'FFTW_WISDOM_ONLY'` is supported. This tells FFTW to raise an error if no plan for this transform and data type is already in the wisdom. It thus provides a method to determine whether planning would require additional effort or the cached wisdom can be used. This flag should be combined with the various planning-effort flags (`'FFTW_ESTIMATE'`, `'FFTW_MEASURE'`, etc.); if so, then an error will be raised if wisdom derived from that level of planning effort (or higher) is not present. If no planning-effort flag is used, the default of `'FFTW_ESTIMATE'` is assumed. Note that wisdom is specific to all the parameters, including the data alignment. That is, if wisdom was generated with input/output arrays with one specific alignment, using `'FFTW_WISDOM_ONLY'` to create a plan for arrays with any different alignment will cause the `'FFTW_WISDOM_ONLY'` planning to fail. Thus it is important to specifically control the data alignment to make the best use of `'FFTW_WISDOM_ONLY'`.

  The FFTW planner flags documentation has more information about the various flags and their impact. Note that only the flags documented here are supported.

- `threads` tells the wrapper how many threads to use when invoking FFTW, with a default of 1. If the number of threads is greater than 1, then the GIL is released by necessity.
- `planning_timelimit` is a floating point number that indicates to the underlying FFTW planner the maximum number of seconds it should spend planning the FFT. This is a rough estimate and corresponds to calling of `fftw_set_timelimit()` (or an equivalent dependent on type) in the underlying FFTW library. If `None` is set, the planner will run indefinitely until all the planning modes allowed by the flags have been tried. See the FFTW planner flags page for more information on this.

Schemes

The currently supported schemes are as follows:

Type	`input_array.dtype`	`output_array.dtype`	Direction
Complex	`complex64`	`complex64`	Both
Complex	`complex128`	`complex128`	Both
Complex	`clongdouble`	`clongdouble`	Both
Real	`float32`	`complex64`	Forwards
Real	`float64`	`complex128`	Forwards
Real	`longdouble`	`clongdouble`	Forwards
Real[1]	`complex64`	`float32`	Backwards
Real[1]	`complex128`	`float64`	Backwards
Real[1]	`clongdouble`	`longdouble`	Backwards

[1] Note that the Backwards Real transform for the case in which the dimensionality of the transform is greater than 1 will destroy the input array. This is inherent to FFTW and the only general work-around for this is to copy the array prior to performing the transform. In the case where the dimensionality of the transform is 1, the default is to preserve the input array. This is different from the default in the underlying library, and some speed gain may be achieved by allowing the input array to be destroyed by passing the `'FFTW_DESTROY_INPUT'` *flag*.

`clongdouble` typically maps directly to `complex256` or `complex192`, and `longdouble` to `float128` or `float96`, dependent on platform.

The relative shapes of the arrays should be as follows:

- For a Complex transform, `output_array.shape == input_array.shape`
- For a Real transform in the Forwards direction, both the following should be true:
  - `output_array.shape[axes][-1] == input_array.shape[axes][-1]//2 + 1`
  - All the other axes should be equal in length.
- For a Real transform in the Backwards direction, both the following should be true:
  - `input_array.shape[axes][-1] == output_array.shape[axes][-1]//2 + 1`
  - All the other axes should be equal in length.

In the above expressions for the Real transform, the `axes` arguments denotes the unique set of axes on which we are taking the FFT, in the order passed. It is the last of these axes that is subject to the special case shown.

The shapes for the real transforms corresponds to those stipulated by the FFTW library. Further information can be found in the FFTW documentation on the real DFT.

The actual arrangement in memory is arbitrary and the scheme can be planned for any set of strides on either the input or the output. The user should not have to worry about this and any valid numpy array should work just fine.

What is calculated is exactly what FFTW calculates. Notably, this is an unnormalized transform so should be scaled as necessary (fft followed by ifft will scale the input by N, the product of the dimensions along which the DFT is taken). For further information, see the FFTW documentation.

The FFTW library benefits greatly from the beginning of each DFT axes being aligned on the correct byte boundary, enabling SIMD instructions. By default, if the data begins on such a boundary, then FFTW will be allowed to try and enable SIMD instructions. This means that all future changes to the data arrays will be checked for similar alignment. SIMD instructions can be explicitly disabled by setting the FFTW_UNALIGNED flags, to allow for updates with unaligned data.

byte_align() and *empty_aligned()* are two methods included with this module for producing aligned arrays.

The optimum alignment for the running platform is provided by *pyfftw.simd_alignment*, though a different alignment may still result in some performance improvement. For example, if the processor supports AVX (requiring 32-byte alignment) as well as SSE (requiring 16-byte alignment), then if the array is 16-byte aligned, SSE will still be used.

It's worth noting that just being aligned may not be sufficient to create the fastest possible transform. For example, if the array is not contiguous (i.e. certain axes are displaced in memory), it may be faster to plan a transform for a contiguous array, and then rely on the array being copied in before the transform (which *pyfftw.FFTW* will handle for you when accessed through *__call__()*).

N
The product of the lengths of the DFT over all DFT axes. 1/N is the normalisation constant. For any input array A, and for any set of axes, 1/N * ifft(fft(A)) = A

simd_aligned
Return whether or not this FFTW object requires simd aligned input and output data.

input_alignment
Returns the byte alignment of the input arrays for which the *FFTW* object was created.

Input array updates with arrays that are not aligned on this byte boundary will result in a ValueError being raised, or a copy being made if the *__call__()* interface is used.

output_alignment
Returns the byte alignment of the output arrays for which the *FFTW* object was created.

Output array updates with arrays that are not aligned on this byte boundary will result in a ValueError being raised.

flags
Return which flags were used to construct the FFTW object.

This includes flags that were added during initialisation.

input_array
Return the input array that is associated with the FFTW instance.

output_array
Return the output array that is associated with the FFTW instance.

input_shape
Return the shape of the input array for which the FFT is planned.

output_shape
Return the shape of the output array for which the FFT is planned.

input_strides
Return the strides of the input array for which the FFT is planned.

output_strides
Return the strides of the output array for which the FFT is planned.

input_dtype
Return the dtype of the input array for which the FFT is planned.

output_dtype
Return the shape of the output array for which the FFT is planned.

direction
Return the planned FFT direction. Either *'FFTW_FORWARD'* or *'FFTW_BACKWARD'*.

axes
Return the axes for the planned FFT in canonical form. That is, as a tuple of positive integers. The order in which they were passed is maintained.

ortho
If `ortho=True` both the forward and inverse transforms are scaled by 1/sqrt(N).

normalise_idft
If `normalise_idft=True`, the inverse transform is scaled by 1/N.

__call__()

__call__(input_array=None, output_array=None, normalise_idft=True, ortho=False)

Calling the class instance (optionally) updates the arrays, then calls *execute()*, before optionally normalising the output and returning the output array.

It has some built-in helpers to make life simpler for the calling functions (as distinct from manually updating the arrays and calling *execute()*).

If `normalise_idft` is `True` (the default), then the output from an inverse DFT (i.e. when the direction flag is `'FFTW_BACKWARD'`) is scaled by 1/N, where N is the product of the lengths of input array on which the FFT is taken. If the direction is `'FFTW_FORWARD'`, this flag makes no difference to the output array.

If `ortho` is `True`, then the output of both forward and inverse DFT operations is scaled by 1/sqrt(N), where N is the product of the lengths of input array on which the FFT is taken. This ensures that the DFT is a unitary operation, meaning that it satisfies Parseval's theorem (the sum of the squared values of the transform output is equal to the sum of the squared values of the input). In other words, the energy of the signal is preserved.

If either `normalise_idft` or `ortho` are `True`, then ifft(fft(A)) = A.

When `input_array` is something other than None, then the passed in array is coerced to be the same dtype as the input array used when the class was instantiated, the byte-alignment of the passed in array is made consistent with the expected byte-alignment and the striding is made consistent with the expected striding. All this may, but not necessarily, require a copy to be made.

As noted in the *scheme table*, if the FFTW instance describes a backwards real transform of more than one dimension, the contents of the input array will be destroyed. It is up to the calling function to make a copy if it is necessary to maintain the input array.

`output_array` is always used as-is if possible. If the dtype, the alignment or the striding is incorrect for the FFTW object, then a `ValueError` is raised.

The coerced input array and the output array (as appropriate) are then passed as arguments to *update_arrays()*, after which *execute()* is called, and then normalisation is applied to the output array if that is desired.

Note that it is possible to pass some data structure that can be converted to an array, such as a list, so long as it fits the data requirements of the class instance, such as array shape.

Other than the dtype and the alignment of the passed in arrays, the rest of the requirements on the arrays mandated by *update_arrays()* are enforced.

A `None` argument to either keyword means that that array is not updated.

The result of the FFT is returned. This is the same array that is used internally and will be overwritten again on subsequent calls. If you need the data to persist longer than a subsequent call, you should copy the returned array.

update_arrays(*new_input_array*, *new_output_array*)
Update the arrays upon which the DFT is taken.

The new arrays should be of the same dtypes as the originals, the same shapes as the originals and should have the same strides between axes. If the original data was aligned so as to allow SIMD instructions (e.g. by being aligned on a 16-byte boundary), then the new array must also be aligned so as to allow SIMD instructions (assuming, of course, that the `FFTW_UNALIGNED` flag was not enabled).

The byte alignment requirement extends to requiring natural alignment in the non-SIMD cases as well, but this is much less stringent as it simply means avoiding arrays shifted by, say, a single byte (which invariably takes some effort to achieve!).

If all these conditions are not met, a `ValueError` will be raised and the data will *not* be updated (though the object will still be in a sane state).

execute()
Execute the planned operation, taking the correct kind of FFT of the input array (i.e. *FFTW.input_array*), and putting the result in the output array (i.e. *FFTW.output_array*).

get_input_array()
Return the input array that is associated with the FFTW instance.

Deprecated since 0.10. Consider using the FFTW.input_array property instead.

get_output_array()
Return the output array that is associated with the FFTW instance.

Deprecated since 0.10. Consider using the FFTW.output_array property instead.

Wisdom Functions

Functions for dealing with FFTW's ability to export and restore plans, referred to as *wisdom*. For further information, refer to the FFTW wisdom documentation.

pyfftw.**export_wisdom**()
Return the FFTW wisdom as a tuple of strings.

The first string in the tuple is the string for the double precision wisdom, the second is for single precision, and the third for long double precision. If any of the precisions is not supported in the build, the string is empty.

The tuple that is returned from this function can be used as the argument to *import_wisdom()*.

pyfftw.**import_wisdom**(*wisdom*)
Function that imports wisdom from the passed tuple of strings.

The first string in the tuple is the string for the double precision wisdom. The second string in the tuple is the string for the single precision wisdom. The third string in the tuple is the string for the long double precision wisdom.

The tuple that is returned from *export_wisdom()* can be used as the argument to this function.

This function returns a tuple of boolean values indicating the success of loading each of the wisdom types (double, float and long double, in that order).

pyfftw.**forget_wisdom**()
Forget all the accumulated wisdom.

Utility Functions

pyfftw.**simd_alignment**
An integer giving the optimum SIMD alignment in bytes, found by inspecting the CPU (e.g. if AVX is supported, its value will be 32).

This can be used as `n` in the arguments for *byte_align()*, *empty_aligned()*, *zeros_aligned()*, and *ones_aligned()* to create optimally aligned arrays for the running platform.

pyfftw.**byte_align**(*array*, *n=None*, *dtype=None*)
Function that takes a numpy array and checks it is aligned on an n-byte boundary, where `n` is an optional parameter. If `n` is not provided then this function will inspect the CPU to determine alignment. If the array is aligned then it is returned without further ado. If it is not aligned then a new array is created and the data copied in, but aligned on the n-byte boundary.

`dtype` is an optional argument that forces the resultant array to be of that dtype.

pyfftw.**empty_aligned**(*shape*, *dtype='float64'*, *order='C'*, *n=None*)
Function that returns an empty numpy array that is n-byte aligned, where `n` is determined by inspecting the CPU if it is not provided.

The alignment is given by the final optional argument, `n`. If `n` is not provided then this function will inspect the CPU to determine alignment. The rest of the arguments are as per `numpy.empty()`.

pyfftw.**zeros_aligned**(*shape*, *dtype='float64'*, *order='C'*, *n=None*)
Function that returns a numpy array of zeros that is n-byte aligned, where `n` is determined by inspecting the CPU if it is not provided.

The alignment is given by the final optional argument, `n`. If `n` is not provided then this function will inspect the CPU to determine alignment. The rest of the arguments are as per `numpy.zeros()`.

pyfftw.**ones_aligned**(*shape*, *dtype='float64'*, *order='C'*, *n=None*)
Function that returns a numpy array of ones that is n-byte aligned, where `n` is determined by inspecting the CPU if it is not provided.

The alignment is given by the final optional argument, `n`. If `n` is not provided then this function will inspect the CPU to determine alignment. The rest of the arguments are as per `numpy.ones()`.

pyfftw.**is_byte_aligned**()
is_n_byte_aligned(array, n=None)

Function that takes a numpy array and checks it is aligned on an n-byte boundary, where `n` is an optional parameter, returning `True` if it is, and `False` if it is not. If `n` is not provided then this function will inspect the CPU to determine alignment.

pyfftw.**n_byte_align**(*array*, *n*, *dtype=None*)
This function is deprecated: `byte_align` **should be used instead.**

Function that takes a numpy array and checks it is aligned on an n-byte boundary, where `n` is an optional parameter. If `n` is not provided then this function will inspect the CPU to determine alignment. If the array is aligned then it is returned without further ado. If it is not aligned then a new array is created and the data copied in, but aligned on the n-byte boundary.

`dtype` is an optional argument that forces the resultant array to be of that dtype.

pyfftw.**n_byte_align_empty**(*shape*, *n*, *dtype='float64'*, *order='C'*)
This function is deprecated: `empty_aligned` **should be used instead.**

Function that returns an empty numpy array that is n-byte aligned.

The alignment is given by the first optional argument, `n`. If `n` is not provided then this function will inspect the CPU to determine alignment. The rest of the arguments are as per `numpy.empty()`.

pyfftw.**is_n_byte_aligned**(*array*, *n*)
This function is deprecated: `is_byte_aligned` **should be used instead.**

Function that takes a numpy array and checks it is aligned on an n-byte boundary, where `n` is a passed parameter, returning `True` if it is, and `False` if it is not.

pyfftw.**next_fast_len**(*target*)

Find the next fast transform length for FFTW.

FFTW has efficient functions for transforms of length 2**a * 3**b * 5**c * 7**d * 11**e * 13**f, where e + f is either 0 or 1.

Parameters **target** (*int*) – Length to start searching from. Must be a positive integer.

Returns **out** – The first fast length greater than or equal to *target*.

Return type int

Examples

On a particular machine, an FFT of prime length takes 2.1 ms:

```
>>> from pyfftw.interfaces import scipy_fftpack
>>> min_len = 10007  # prime length is worst case for speed
>>> a = numpy.random.randn(min_len)
>>> b = scipy_fftpack.fft(a)
```

Zero-padding to the next fast length reduces computation time to 406 us, a speedup of ~5 times:

```
>>> next_fast_len(min_len)
10080
>>> b = scipy_fftpack.fft(a, 10080)
```

Rounding up to the next power of 2 is not optimal, taking 598 us to compute, 1.5 times as long as the size selected by next_fast_len.

```
>>> b = fftpack.fft(a, 16384)
```

Similar speedups will occur for pre-planned FFTs as generated via pyfftw.builders.

FFTW Configuration

pyfftw.config.**NUM_THREADS**

This variable controls the default number of threads used by the functions in *pyfftw.builders* and *pyfftw.interfaces*.

The default value is read from the environment variable `PYFFTW_NUM_THREADS`. If this variable is undefined and the user's underlying FFTW library was built using OpenMP threading, the number of threads will be read from the environment variable `OMP_NUM_THREADS` instead. If neither environment variable is defined, the default value is 1.

If the specified value is `<= 0`, the library will use `multiprocessing.cpu_count()` to determine the number of threads.

The user can modify the value at run time by assigning to this variable.

pyfftw.config.**PLANNER_EFFORT**

This variable controls the default planning effort used by the functions in *pyfftw.builders* and *pyfftw.interfaces*.

The default value of is determined by reading from the environment variable `PYFFTW_PLANNER_EFFORT`. If this environment variable is undefined, it defaults to `'FFTW_ESTIMATE'`.

The user can modify the value at run time by assigning to this variable.

2.3.2 pyfftw.builders - Get FFTW objects using a numpy.fft like interface

Overview

This module contains a set of functions that return *pyfftw.FFTW* objects.

The interface to create these objects is mostly the same as numpy.fft, only instead of the call returning the result of the FFT, a *pyfftw.FFTW* object is returned that performs that FFT operation when it is called. Users should be familiar with numpy.fft before reading on.

In the case where the shape argument, s (or n in the 1-dimensional case), dictates that the passed-in input array be copied into a different processing array, the returned object is an instance of a child class of *pyfftw.FFTW*, *_FFTWWrapper*, which wraps the call method in order to correctly perform that copying. That is, subsequent calls to the object (i.e. through *__call__()*) should occur with an input array that can be sliced to the same size as the expected internal array. Note that a side effect of this is that subsequent calls to the object can be made with an array that is *bigger* than the original (but not smaller).

Only the call method is wrapped; *update_arrays()* still expects an array with the correct size, alignment, dtype etc for the *pyfftw.FFTW* object.

When the internal input array is bigger along any axis than the input array that is passed in (due to s dictating a larger size), then the extra entries are padded with zeros. This is a one time action. If the internal input array is then extracted using *pyfftw.FFTW.input_array*, it is possible to persistently fill the padding space with whatever the user desires, so subsequent calls with a new input only overwrite the values that aren't padding (even if the array that is used for the call is bigger than the original - see the point above about bigger arrays being sliced to fit).

The precision of the FFT operation is acquired from the input array. If an array is passed in that is not of float type, or is of an unknown float type, an attempt is made to convert the array to a double precision array. This results in a copy being made.

If an array of the incorrect complexity is passed in (e.g. a complex array is passed to a real transform routine, or vice-versa), then an attempt is made to convert the array to an array of the correct complexity. This results in a copy being made.

Although the array that is internal to the *pyfftw.FFTW* object will be correctly loaded with the values within the input array, it is not necessarily the case that the internal array *is* the input array. The actual internal input array can always be retrieved with *pyfftw.FFTW.input_array*.

The behaviour of the norm option in all builder routines matches that of the corresponding numpy functions. In short, if norm is None, then the output from the forward DFT is unscaled and the inverse DFT is scaled by 1/N, where N is the product of the lengths of input array on which the FFT is taken. If norm == 'ortho', then the output of both forward and inverse DFT operations are scaled by 1/sqrt(N).

Example:

```
>>> import pyfftw
>>> a = pyfftw.empty_aligned(4, dtype='complex128')
>>> fft = pyfftw.builders.fft(a)
>>> a[:] = [1, 2, 3, 4]
>>> fft() # returns the output
array([ 10.+0.j,  -2.+2.j,  -2.+0.j,  -2.-2.j])
```

More examples can be found in the *tutorial*.

Supported Functions and Caveats

The following functions are supported. They can be used with the same calling signature as their respective functions in numpy.fft.

Standard FFTs

- *fft()*
- *ifft()*
- *fft2()*
- *ifft2()*
- *fftn()*
- *ifftn()*

Real FFTs

- *rfft()*
- *irfft()*
- *rfft2()*
- *irfft2()*
- *rfftn()*
- *irfftn()*

The first caveat is that the dtype of the input array must match the transform. For example, for `fft` and `ifft`, the dtype must be complex, for `rfft` it must be real, and so on. The other point to note from this is that the precision of the transform matches the precision of the input array. So, if a single precision input array is passed in, then a single precision transform will be used.

The second caveat is that repeated axes are handled differently; with the returned *pyfftw.FFTW* object, axes that are repeated in the axes argument are considered only once, as compared to `numpy.fft` in which repeated axes results in the DFT being taken along that axes as many times as the axis occurs (this is down to the underlying library).

Note that unless the `auto_align_input` argument to the function is set to `True`, the `'FFTW_UNALIGNED'` *flag* is set in the returned *pyfftw.FFTW* object. This disables some of the FFTW optimisations that rely on aligned arrays. Also worth noting is that the `auto_align_input` flag only results in a copy when calling the resultant *pyfftw.FFTW* object if the input array is not already aligned correctly.

Additional Arguments

In addition to the arguments that are present with their complementary functions in `numpy.fft`, each of these functions also offers the following additional keyword arguments:

- `overwrite_input`: Whether or not the input array can be overwritten during the transform. This sometimes results in a faster algorithm being made available. It causes the `'FFTW_DESTROY_INPUT'` flag to be passed to the *pyfftw.FFTW* object. This flag is not offered for the multi-dimensional inverse real transforms, as FFTW is unable to not overwrite the input in that case.
- `planner_effort`: A string dictating how much effort is spent in planning the FFTW routines. This is passed to the creation of the *pyfftw.FFTW* object as an entry in the flags list. They correspond to flags passed to the *pyfftw.FFTW* object.

  The valid strings, in order of their increasing impact on the time to compute are: `'FFTW_ESTIMATE'`, `config.PLANNER_EFFORT` (default), `'FFTW_PATIENT'` and `'FFTW_EXHAUSTIVE'`.

  The Wisdom that FFTW has accumulated or has loaded (through *pyfftw.import_wisdom()*) is used during the creation of *pyfftw.FFTW* objects.
- `threads`: The number of threads used to perform the FFT.

- `auto_align_input`: Correctly byte align the input array for optimal usage of vector instructions. This can lead to a substantial speedup.

  Setting this argument to `True` makes sure that the input array is correctly aligned. It is possible to correctly byte align the array prior to calling this function (using, for example, *pyfftw.byte_align()*). If and only if a realignment is necessary is a new array created. If a new array *is* created, it is up to the calling code to acquire that new input array using *pyfftw.FFTW.input_array*.

  The resultant *pyfftw.FFTW* object that is created will be designed to operate on arrays that are aligned. If the object is called with an unaligned array, this would result in a copy. Despite this, it may still be faster to set the `auto_align_input` flag and incur a copy with unaligned arrays than to set up an object that uses aligned arrays.

  It's worth noting that just being aligned may not be sufficient to create the fastest possible transform. For example, if the array is not contiguous (i.e. certain axes have gaps in memory between slices), it may be faster to plan a transform for a contiguous array, and then rely on the array being copied in before the transform (which *pyfftw.FFTW* will handle for you). The `auto_contiguous` argument controls whether this function also takes care of making sure the array is contiguous or not.

- `auto_contiguous`: Make sure the input array is contiguous in memory before performing the transform on it. If the array is not contiguous, it is copied into an interim array. This is because it is often faster to copy the data before the transform and then transform a contiguous array than it is to try to take the transform of a non-contiguous array. This is particularly true in conjunction with the `auto_align_input` argument which is used to make sure that the transform is taken of an aligned array.

  Like `auto_align_input`, If a new array is created, it is up to the calling code to acquire that new input array using *pyfftw.FFTW.input_array*.

- `avoid_copy`: By default, these functions will always create a copy (and sometimes more than one) of the passed in input array. This is because the creation of the *pyfftw.FFTW* object generally destroys the contents of the input array. Setting this argument to `True` will try not to create a copy of the input array, likely resulting in the input array being destroyed. If it is not possible to create the object without a copy being made, a `ValueError` is raised.

  Example situations that require a copy, and so cause the exception to be raised when this flag is set:

  - The shape of the FFT input as dictated by `s` is necessarily different from the shape of the passed-in array.
  - The dtypes are incompatible with the FFT routine.
  - The `auto_contiguous` or `auto_align` flags are True and the input array is not already contiguous or aligned.

  This argument is distinct from `overwrite_input` in that it only influences a copy during the creation of the object. It changes no flags in the *pyfftw.FFTW* object.

The exceptions raised by each of these functions are as per their equivalents in `numpy.fft`, or as documented above.

The Functions

pyfftw.builders.**fft**(*a*, *n=None*, *axis=-1*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 1D FFT.

The first three arguments are as per `numpy.fft.fft()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**ifft**(*a*, *n=None*, *axis=-1*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 1D inverse FFT.

The first three arguments are as per `numpy.fft.ifft()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**fft2**(*a*, *s=None*, *axes=(-2, -1)*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 2D FFT.

The first three arguments are as per `numpy.fft.fft2()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**ifft2**(*a*, *s=None*, *axes=(-2, -1)*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 2D inverse FFT.

The first three arguments are as per `numpy.fft.ifft2()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**fftn**(*a*, *s=None*, *axes=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a n-D FFT.

The first three arguments are as per `numpy.fft.fftn()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**ifftn**(*a*, *s=None*, *axes=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing an n-D inverse FFT.

The first three arguments are as per `numpy.fft.ifftn()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**rfft**(*a*, *n=None*, *axis=-1*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 1D real FFT.

The first three arguments are as per `numpy.fft.rfft()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**irfft**(*a*, *n=None*, *axis=-1*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 1D real inverse FFT.

The first three arguments are as per `numpy.fft.irfft()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**rfft2**(*a*, *s=None*, *axes=(-2, -1)*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 2D real FFT.

The first three arguments are as per `numpy.fft.rfft2()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**irfft2**(*a*, *s=None*, *axes=(-2, -1)*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing a 2D real inverse FFT.

The first three arguments are as per `numpy.fft.irfft2()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**rfftn**(*a*, *s=None*, *axes=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing an n-D real FFT.

The first three arguments are as per `numpy.fft.rfftn()`; the rest of the arguments are documented *in the module docs*.

pyfftw.builders.**irfftn**(*a*, *s=None*, *axes=None*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*, *avoid_copy=False*, *norm=None*)

Return a *pyfftw.FFTW* object representing an n-D real inverse FFT.

The first three arguments are as per `numpy.fft.rfftn()`; the rest of the arguments are documented *in the module docs*.

2.3.3 `pyfftw.builders._utils` - Helper functions for `pyfftw.builders`

A set of utility functions for use with the builders. Users should not need to use the functions directly, but they are included here for completeness and to aid with understanding of what is happening behind the scenes.

Certainly, users may encounter instances of *_FFTWWrapper*.

Everything documented in this module is *not* part of the public API and may change in future versions.

class pyfftw.builders._utils.**_FFTWWrapper**(*input_array*, *output_array*, *axes=[-1]*, *direction='FFTW_FORWARD'*, *flags=['FFTW_MEASURE']*, *threads=1*, *input_array_slicer=None*, *FFTW_array_slicer=None*, *normalise_idft=True*, *ortho=False*)

A class that wraps *pyfftw.FFTW*, providing a slicer on the input stage during calls to *__call__()*.

The arguments are as per *pyfftw.FFTW*, but with the addition of 2 keyword arguments: `input_array_slicer` and `FFTW_array_slicer`.

These arguments represent 2 slicers: `input_array_slicer` slices the input array that is passed in during a call to instances of this class, and `FFTW_array_slicer` slices the internal array.

The arrays that are returned from both of these slicing operations should be the same size. The data is then copied from the sliced input array into the sliced internal array.

__call__(*input_array=None*, *output_array=None*, *normalise_idft=None*, *ortho=None*)

Wrap *pyfftw.FFTW.__call__()* by firstly slicing the passed-in input array and then copying it into a sliced version of the internal array. These slicers are set at instantiation.

When input array is not `None`, this method always results in a copy. Consequently, the alignment and dtype are maintained in the internal array.

`output_array` and `normalise_idft` are passed through to *pyfftw.FFTW.__call__()* untouched.

pyfftw.builders._utils.**_Xfftn**(*a*, *s*, *axes*, *overwrite_input*, *planner_effort*, *threads*, *auto_align_input*, *auto_contiguous*, *avoid_copy*, *inverse*, *real*, *normalise_idft=True*, *ortho=False*)

Generic transform interface for all the transforms. No defaults exist. The transform must be specified exactly.

pyfftw.builders._utils.**_setup_input_slicers**(*a_shape*, *input_shape*)

This function returns two slicers that are to be used to copy the data from the input array to the FFTW object internal array, which can then be passed to _FFTWWrapper:

```
(update_input_array_slicer, FFTW_array_slicer)
```

On calls to *_FFTWWrapper* objects, the input array is copied in as:

```
FFTW_array[FFTW_array_slicer] = input_array[update_input_array_slicer]
```

pyfftw.builders._utils.**_compute_array_shapes**(*a*, *s*, *axes*, *inverse*, *real*)

Given a passed in array `a`, and the rest of the arguments (that have been fleshed out with *_cook_nd_args()*), compute the shape the input and output arrays need to be in order to satisfy all the requirements for the transform. The input shape *may* be different to the shape of a.

returns: `(input_shape, output_shape)`

pyfftw.builders._utils.**_precook_1d_args**(*a*, *n*, *axis*)

Turn `*(n, axis)` into `(s, axes)`

pyfftw.builders._utils.**_cook_nd_args**(*a*, *s=None*, *axes=None*, *invreal=False*)

Similar to `numpy.fft.fftpack._cook_nd_args()`.

2.3.4 pyfftw.interfaces - Drop in replacements for other FFT implementations

numpy.fft interface

This module implements those functions that replace aspects of the `numpy.fft` module. This module *provides* the entire documented namespace of `numpy.fft`, but those functions that are not included here are imported directly from `numpy.fft`.

It is notable that unlike `numpy.fftpack`, these functions will generally return an output array with the same precision as the input array, and the transform that is chosen is chosen based on the precision of the input array. That is, if the input array is 32-bit floating point, then the transform will be 32-bit floating point and so will the returned array. Half precision input will be converted to single precision. Otherwise, if any type conversion is required, the default will be double precision. If pyFFTW was not built with support for double precision, the default is long double precision. If that is not available, it defaults to single precision.

One known caveat is that repeated axes are handled differently to `numpy.fft`; axes that are repeated in the axes argument are considered only once, as compared to `numpy.fft` in which repeated axes results in the DFT being taken along that axes as many times as the axis occurs.

The exceptions raised by each of these functions are mostly as per their equivalents in `numpy.fft`, though there are some corner cases in which this may not be true.

pyfftw.interfaces.numpy_fft.**fft**(*a*, *n=None*, *axis=-1*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D FFT.

The first four arguments are as per `numpy.fft.fft()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**ifft**(*a*, *n=None*, *axis=-1*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D inverse FFT.

The first four arguments are as per `numpy.fft.ifft()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**fft2**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 2D FFT.

The first four arguments are as per `numpy.fft.fft2()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**ifft2**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 2D inverse FFT.

The first four arguments are as per `numpy.fft.ifft2()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**fftn**(*a*, *s=None*, *axes=None*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform an n-D FFT.

The first four arguments are as per `numpy.fft.fftn()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**ifftn**(*a*, *s=None*, *axes=None*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform an n-D inverse FFT.

The first four arguments are as per `numpy.fft.ifftn()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**rfft**(*a*, *n=None*, *axis=-1*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D real FFT.

The first four arguments are as per `numpy.fft.rfft()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**irfft**(*a*, *n=None*, *axis=-1*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D real inverse FFT.

The first four arguments are as per `numpy.fft.irfft()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**rfft2**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 2D real FFT.

The first four arguments are as per `numpy.fft.rfft2()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**irfft2**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 2D real inverse FFT.

The first four arguments are as per `numpy.fft.irfft2()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**rfftn**(*a*, *s=None*, *axes=None*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform an n-D real FFT.

The first four arguments are as per `numpy.fft.rfftn()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**irfftn**(*a*, *s=None*, *axes=None*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform an n-D real inverse FFT.

The first four arguments are as per `numpy.fft.rfftn()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**hfft**(*a*, *n=None*, *axis=-1*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D FFT of a signal with hermitian symmetry. This yields a real output spectrum. See `numpy.fft.hfft()` for more information.

The first four arguments are as per `numpy.fft.hfft()`; the rest of the arguments are documented in the *additional arguments docs*.

pyfftw.interfaces.numpy_fft.**ihfft**(*a*, *n=None*, *axis=-1*, *norm=None*, *overwrite_input=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D inverse FFT of a real-spectrum, yielding a signal with hermitian symmetry. See `numpy.fft.ihfft()` for more information.

The first four arguments are as per `numpy.fft.ihfft()`; the rest of the arguments are documented in the *additional arguments docs*.

`scipy.fftpack` interface

This module implements those functions that replace aspects of the `scipy.fftpack` module. This module *provides* the entire documented namespace of `scipy.fftpack`, but those functions that are not included here are imported directly from `scipy.fftpack`.

The exceptions raised by each of these functions are mostly as per their equivalents in `scipy.fftpack`, though there are some corner cases in which this may not be true.

It is notable that unlike `scipy.fftpack`, these functions will generally return an output array with the same precision as the input array, and the transform that is chosen is chosen based on the precision of the input array. That is, if the input array is 32-bit floating point, then the transform will be 32-bit floating point and so will the returned array. Half precision input will be converted to single precision. Otherwise, if any type conversion is required, the default will be double precision.

Some corner (mis)usages of `scipy.fftpack` may not transfer neatly. For example, using `scipy.fftpack.fft2()` with a non 1D array and a 2D *shape* argument will return without exception whereas *`pyfftw.interfaces.scipy_fftpack.fft2()`* will raise a *ValueError*.

pyfftw.interfaces.scipy_fftpack.**fft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D FFT.

The first three arguments are as per `scipy.fftpack.fft()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**ifft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D inverse FFT.

The first three arguments are as per `scipy.fftpack.ifft()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**fftn**(*x*, *shape=None*, *axes=None*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform an n-D FFT.

The first three arguments are as per `scipy.fftpack.fftn()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**ifftn**(*x*, *shape=None*, *axes=None*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform an n-D inverse FFT.

The first three arguments are as per `scipy.fftpack.ifftn()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**rfft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D real FFT.

The first three arguments are as per `scipy.fftpack.rfft()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**irfft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 1D real inverse FFT.

The first three arguments are as per `scipy.fftpack.irfft()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**fft2**(*x*, *shape=None*, *axes=(-2, -1)*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 2D FFT.

The first three arguments are as per `scipy.fftpack.fft2()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**ifft2**(*x*, *shape=None*, *axes=(-2, -1)*, *overwrite_x=False*, *planner_effort=None*, *threads=None*, *auto_align_input=True*, *auto_contiguous=True*)

Perform a 2D inverse FFT.

The first three arguments are as per `scipy.fftpack.ifft2()`; the rest of the arguments are documented in the *additional argument docs*.

pyfftw.interfaces.scipy_fftpack.**next_fast_len**(*target*)

Find the next fast transform length for FFTW.

FFTW has efficient functions for transforms of length 2**a * 3**b * 5**c * 7**d * 11**e * 13**f, where e + f is either 0 or 1.

Parameters **target** (*int*) – Length to start searching from. Must be a positive integer.

Returns **out** – The first fast length greater than or equal to *target*.

Return type int

Examples

On a particular machine, an FFT of prime length takes 2.1 ms:

```
>>> from pyfftw.interfaces import scipy_fftpack
>>> min_len = 10007  # prime length is worst case for speed
>>> a = numpy.random.randn(min_len)
>>> b = scipy_fftpack.fft(a)
```

Zero-padding to the next fast length reduces computation time to 406 us, a speedup of ~5 times:

```
>>> next_fast_len(min_len)
10080
>>> b = scipy_fftpack.fft(a, 10080)
```

Rounding up to the next power of 2 is not optimal, taking 598 us to compute, 1.5 times as long as the size selected by next_fast_len.

```
>>> b = fftpack.fft(a, 16384)
```

Similar speedups will occur for pre-planned FFTs as generated via pyfftw.builders.

`dask.fft` interface

This module implements those functions that replace aspects of the `dask.fft` module. This module *provides* the entire documented namespace of `dask.fft`, but those functions that are not included here are imported directly from `dask.fft`.

It is notable that unlike `numpy.fftpack`, which `dask.fft` wraps, these functions will generally return an output array with the same precision as the input array, and the transform that is chosen is chosen based on the precision of the input array. That is, if the input array is 32-bit floating point, then the transform will be 32-bit floating point and so will the returned array. Half precision input will be converted to single precision. Otherwise, if any type conversion is required, the default will be double precision.

The exceptions raised by each of these functions are mostly as per their equivalents in `dask.fft`, though there are some corner cases in which this may not be true.

pyfftw.interfaces.dask_fft.**fft**(*a*, *n=None*, *axis=None*)

Wrapping of pyfftw.interfaces.numpy_fft.fft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.fft docstring follows below:

Perform a 1D FFT.

The first four arguments are as per `numpy.fft.fft()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`ifft`**(*a*, *n=None*, *axis=None*)
Wrapping of pyfftw.interfaces.numpy_fft.ifft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.ifft docstring follows below:

Perform a 1D inverse FFT.

The first four arguments are as per `numpy.fft.ifft()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`fft2`**(*a*, *s=None*, *axes=None*)
Wrapping of pyfftw.interfaces.numpy_fft.fft2

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.fft2 docstring follows below:

Perform a 2D FFT.

The first four arguments are as per `numpy.fft.fft2()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`ifft2`**(*a*, *s=None*, *axes=None*)
Wrapping of pyfftw.interfaces.numpy_fft.ifft2

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.ifft2 docstring follows below:

Perform a 2D inverse FFT.

The first four arguments are as per `numpy.fft.ifft2()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`fftn`**(*a*, *s=None*, *axes=None*)
Wrapping of pyfftw.interfaces.numpy_fft.fftn

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.fftn docstring follows below:

Perform an n-D FFT.

The first four arguments are as per `numpy.fft.fftn()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`ifftn`**(*a*, *s=None*, *axes=None*)
Wrapping of pyfftw.interfaces.numpy_fft.ifftn

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.ifftn docstring follows below:

Perform an n-D inverse FFT.

The first four arguments are as per `numpy.fft.ifftn()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`rfft`**(*a*, *n=None*, *axis=None*)
Wrapping of pyfftw.interfaces.numpy_fft.rfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.rfft docstring follows below:

Perform a 1D real FFT.

The first four arguments are as per `numpy.fft.rfft()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`irfft`**(*a*, *n=None*, *axis=None*)
Wrapping of pyfftw.interfaces.numpy_fft.irfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.irfft docstring follows below:

Perform a 1D real inverse FFT.

The first four arguments are as per `numpy.fft.irfft()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`rfft2`**(*a*, *s=None*, *axes=None*)
Wrapping of pyfftw.interfaces.numpy_fft.rfft2

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.rfft2 docstring follows below:

Perform a 2D real FFT.

The first four arguments are as per `numpy.fft.rfft2()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`irfft2`**(*a*, *s=None*, *axes=None*)
Wrapping of pyfftw.interfaces.numpy_fft.irfft2

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.irfft2 docstring follows below:

Perform a 2D real inverse FFT.

The first four arguments are as per `numpy.fft.irfft2()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`rfftn`**(*a*, *s=None*, *axes=None*)
Wrapping of pyfftw.interfaces.numpy_fft.rfftn

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.rfftn docstring follows below:

Perform an n-D real FFT.

The first four arguments are as per `numpy.fft.rfftn()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`irfftn`**(*a*, *s=None*, *axes=None*)

Wrapping of pyfftw.interfaces.numpy_fft.irfftn

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.irfftn docstring follows below:

Perform an n-D real inverse FFT.

The first four arguments are as per `numpy.fft.rfftn()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`hfft`**(*a*, *n=None*, *axis=None*)

Wrapping of pyfftw.interfaces.numpy_fft.hfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.hfft docstring follows below:

Perform a 1D FFT of a signal with hermitian symmetry. This yields a real output spectrum. See `numpy.fft.hfft()` for more information.

The first four arguments are as per `numpy.fft.hfft()`; the rest of the arguments are documented in the *additional arguments docs*.

`pyfftw.interfaces.dask_fft.`**`ihfft`**(*a*, *n=None*, *axis=None*)

Wrapping of pyfftw.interfaces.numpy_fft.ihfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The pyfftw.interfaces.numpy_fft.ihfft docstring follows below:

Perform a 1D inverse FFT of a real-spectrum, yielding a signal with hermitian symmetry. See `numpy.fft.ihfft()` for more information.

The first four arguments are as per `numpy.fft.ihfft()`; the rest of the arguments are documented in the *additional arguments docs*.

The *pyfftw.interfaces* package provides interfaces to *pyfftw* that implement the API of other, more commonly used FFT libraries; specifically `numpy.fft` and `scipy.fftpack`. The intention is to satisfy two clear use cases:

1. Simple, clean and well established interfaces to using *pyfftw*, removing the requirement for users to know or understand about creating and using *pyfftw.FFTW* objects, whilst still benefiting from most of the speed benefits of FFTW.
2. A library that can be dropped into code that is already written to use a supported FFT library, with no significant change to the existing code. The power of python allows this to be done at runtime to a third party library, without changing any of that library's code.

The *pyfftw.interfaces* implementation is designed to sacrifice a small amount of the flexibility compared to accessing the *pyfftw.FFTW* object directly, but implements a reasonable set of defaults and optional tweaks that should satisfy most situations.

The precision of the transform that is used is selected from the array that is passed in, defaulting to double precision if any type conversion is required.

This module works by generating a *pyfftw.FFTW* object behind the scenes using the *pyfftw.builders* interface, which is then executed. There is therefore a potentially substantial overhead when a new plan needs to be created. This is down to FFTW's internal planner process. After a specific transform has been planned once, subsequent calls in which the input array is equivalent will be much faster, though still not without potentially significant overhead. *This* overhead can be largely alleviated by enabling the *pyfftw.interfaces.cache* functionality. However, even when the cache is used, very small transforms may suffer a significant relative slow-down not present when accessing *pyfftw.FFTW* directly (because the transform time can be negligibly small compared to the fixed *pyfftw.interfaces* overhead).

In addition, potentially extra copies of the input array might be made.

If speed or memory conservation is of absolutely paramount importance, the suggestion is to use *pyfftw.FFTW* (which provides better control over copies and so on), either directly or through *pyfftw.builders*. As always, experimentation is the best guide to optimisation.

In practice, this means something like the following (taking *numpy_fft* as an example):

```
>>> import pyfftw, numpy
>>> a = pyfftw.empty_aligned((128, 64), dtype='complex64', n=16)
>>> a[:] = numpy.random.randn(*a.shape) + 1j*numpy.random.randn(*a.shape)
>>> fft_a = pyfftw.interfaces.numpy_fft.fft2(a) # Will need to plan
```

```
>>> b = pyfftw.empty_aligned((128, 64), dtype='complex64', n=16)
>>> b[:] = a
>>> fft_b = pyfftw.interfaces.numpy_fft.fft2(b) # Already planned, so faster
```

```
>>> c = pyfftw.empty_aligned(132, dtype='complex128', n=16)
>>> fft_c = pyfftw.interfaces.numpy_fft.fft(c) # Needs a new plan
>>> c[:] = numpy.random.randn(*c.shape) + 1j*numpy.random.randn(*c.shape)
```

```
>>> pyfftw.interfaces.cache.enable()
>>> fft_a = pyfftw.interfaces.numpy_fft.fft2(a) # still planned
>>> fft_b = pyfftw.interfaces.numpy_fft.fft2(b) # much faster, from the cache
```

The usual wisdom import and export functions work well for the case where the initial plan might be prohibitively expensive. Just use *pyfftw.export_wisdom()* and *pyfftw.import_wisdom()* as needed after having performed the transform once.

Implemented Functions

The implemented functions are listed below. `numpy.fft` is implemented by *pyfftw.interfaces.numpy_fft* and `scipy.fftpack` by *pyfftw.interfaces.scipy_fftpack*. All the implemented functions are extended by the use of additional arguments, which are *documented below*.

Not all the functions provided by `numpy.fft` and `scipy.fftpack` are implemented by *pyfftw.interfaces*. In the case where a function is not implemented, the function is imported into the namespace from the corresponding library. This means that all the documented functionality of the library *is* provided through *pyfftw.interfaces*.

One known caveat is that repeated axes are potentially handled differently. This is certainly the case for `numpy.fft` and probably also true for `scipy.fftpack` (though it is not defined in the docs); axes that are repeated in the axes argument are considered only once, as compared to `numpy.fft` in which repeated axes results in the DFT being taken along that axes as many times as the axis occurs.

numpy_fft

- *pyfftw.interfaces.numpy_fft.fft()*
- *pyfftw.interfaces.numpy_fft.ifft()*
- *pyfftw.interfaces.numpy_fft.fft2()*
- *pyfftw.interfaces.numpy_fft.ifft2()*
- *pyfftw.interfaces.numpy_fft.fftn()*
- *pyfftw.interfaces.numpy_fft.ifftn()*
- *pyfftw.interfaces.numpy_fft.rfft()*
- *pyfftw.interfaces.numpy_fft.irfft()*
- *pyfftw.interfaces.numpy_fft.rfft2()*
- *pyfftw.interfaces.numpy_fft.irfft2()*
- *pyfftw.interfaces.numpy_fft.rfftn()*
- *pyfftw.interfaces.numpy_fft.irfftn()*
- *pyfftw.interfaces.numpy_fft.hfft()*
- *pyfftw.interfaces.numpy_fft.ihfft()*

scipy_fftpack

- *pyfftw.interfaces.scipy_fftpack.fft()*
- *pyfftw.interfaces.scipy_fftpack.ifft()*
- *pyfftw.interfaces.scipy_fftpack.fft2()*
- *pyfftw.interfaces.scipy_fftpack.ifft2()*
- *pyfftw.interfaces.scipy_fftpack.fftn()*
- *pyfftw.interfaces.scipy_fftpack.ifftn()*
- *pyfftw.interfaces.scipy_fftpack.rfft()*
- *pyfftw.interfaces.scipy_fftpack.irfft()*
- *pyfftw.interfaces.scipy_fftpack.next_fast_len()*

dask_fft

- *pyfftw.interfaces.dask_fft.fft()*
- *pyfftw.interfaces.dask_fft.ifft()*
- *pyfftw.interfaces.dask_fft.rfft()*
- *pyfftw.interfaces.dask_fft.irfft()*
- *pyfftw.interfaces.dask_fft.hfft()*
- *pyfftw.interfaces.dask_fft.ihfft()*

Additional Arguments

In addition to the equivalent arguments in `numpy.fft` and `scipy.fftpack`, all these functions also add several additional arguments for finer control over the FFT. These additional arguments are largely a subset of the keyword arguments in *pyfftw.builders* with a few exceptions and with different defaults.

- `overwrite_input`: Whether or not the input array can be overwritten during the transform. This sometimes results in a faster algorithm being made available. It causes the `'FFTW_DESTROY_INPUT'` flag to be passed to the intermediate *pyfftw.FFTW* object. Unlike with *pyfftw.builders*, this argument is included with *every* function in this package.

  In *scipy_fftpack*, this argument is replaced by `overwrite_x`, to which it is equivalent (albeit at the same position).

  The default is `False` to be consistent with `numpy.fft`.

- `planner_effort`: A string dictating how much effort is spent in planning the FFTW routines. This is passed to the creation of the intermediate *pyfftw.FFTW* object as an entry in the flags list. They correspond to flags passed to the *pyfftw.FFTW* object.

  The valid strings, in order of their increasing impact on the time to compute are: `'FFTW_ESTIMATE'`, `'FFTW_MEASURE'` (default), `'FFTW_PATIENT'` and `'FFTW_EXHAUSTIVE'`.

  The Wisdom that FFTW has accumulated or has loaded (through *pyfftw.import_wisdom()*) is used during the creation of *pyfftw.FFTW* objects.

  Note that the first time planning stage can take a substantial amount of time. For this reason, the default is to use `'FFTW_ESTIMATE'`, which potentially results in a slightly suboptimal plan being used, but with a substantially quicker first-time planner step.

- `threads`: The number of threads used to perform the FFT.

  The default is `1`.

- `auto_align_input`: Correctly byte align the input array for optimal usage of vector instructions. This can lead to a substantial speedup.

  This argument being `True` makes sure that the input array is correctly aligned. It is possible to correctly byte align the array prior to calling this function (using, for example, *pyfftw.byte_align()*). If and only if a realignment is necessary is a new array created.

  It's worth noting that just being aligned may not be sufficient to create the fastest possible transform. For example, if the array is not contiguous (i.e. certain axes have gaps in memory between slices), it may be faster to plan a transform for a contiguous array, and then rely on the array being copied in before the transform (which *pyfftw.FFTW* will handle for you). The `auto_contiguous` argument controls whether this function also takes care of making sure the array is contiguous or not.

  The default is `True`.

- `auto_contiguous`: Make sure the input array is contiguous in memory before performing the transform on it. If the array is not contiguous, it is copied into an interim array. This is because it is often faster to copy the data before the transform and then transform a contiguous array than it is to try to take the transform of a non-contiguous array. This is particularly true in conjunction with the `auto_align_input` argument which is used to make sure that the transform is taken of an aligned array.

  The default is `True`.

Caching

During calls to functions implemented in *pyfftw.interfaces*, a *pyfftw.FFTW* object is necessarily created. Although the time to create a new *pyfftw.FFTW* is short (assuming that the planner possesses the necessary wisdom

to create the plan immediately), it may still take longer than a short transform.

This module implements a method by which objects that are created through *pyfftw.interfaces* are temporarily cached. If an equivalent transform is then performed within a short period, the object is acquired from the cache rather than a new one created. The equivalency is quite conservative and in practice means that if any of the arguments change, or if the properties of the array (shape, strides, dtype) change in any way, then the cache lookup will fail.

The cache temporarily stores a copy of any interim *pyfftw.FFTW* objects that are created. If they are not used for some period of time, which can be set with *pyfftw.interfaces.cache.set_keepalive_time()*, then they are removed from the cache (liberating any associated memory). The default keepalive time is 0.1 seconds.

Enable the cache by calling *pyfftw.interfaces.cache.enable()*. Disable it by calling *pyfftw.interfaces.cache.disable()*. By default, the cache is disabled.

Note that even with the cache enabled, there is a fixed overhead associated with lookups. This means that for small transforms, the overhead may exceed the transform. At this point, it's worth looking at using *pyfftw.FFTW* directly.

When the cache is enabled, the module spawns a new thread to keep track of the objects. If `threading` is not available, then the cache is not available and trying to use it will raise an ImportError exception.

The actual implementation of the cache is liable to change, but the documented API is stable.

`pyfftw.interfaces.cache.`**`enable`**`()`
Enable the cache.

`pyfftw.interfaces.cache.`**`disable`**`()`
Disable the cache.

`pyfftw.interfaces.cache.`**`set_keepalive_time`**`(`*keepalive_time*`)`
Set the minimum time in seconds for which any *pyfftw.FFTW* object in the cache is kept alive.

When the cache is enabled, the interim objects that are used through a *pyfftw.interfaces* function are cached for the time set through this function. If the object is not used for the that time, it is removed from the cache. Using the object zeros the timer.

The time is not precise, and sets a minimum time to be alive. In practice, it may be quite a bit longer before the object is deleted from the cache (due to implementational details - e.g. contention from other threads).

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pyfftw, 10
pyfftw.builders, 18
pyfftw.builders._utils, 22
pyfftw.interfaces, 30
pyfftw.interfaces.cache, 33
pyfftw.interfaces.dask_fft, 27
pyfftw.interfaces.numpy_fft, 23
pyfftw.interfaces.scipy_fftpack, 25

Index

Symbols

_FFTWWrapper (*class in pyfftw.builders._utils*), 22
_Xfftn() (*in module pyfftw.builders._utils*), 22
__call__() (*pyfftw.FFTW method*), 14
__call__() (*pyfftw.builders._utils._FFTWWrapper method*), 22
_compute_array_shapes() (*in module pyfftw.builders._utils*), 23
_cook_nd_args() (*in module pyfftw.builders._utils*), 23
_precook_1d_args() (*in module pyfftw.builders._utils*), 23
_setup_input_slicers() (*in module pyfftw.builders._utils*), 23

A

axes (*pyfftw.FFTW attribute*), 13

B

byte_align() (*in module pyfftw*), 16

D

direction (*pyfftw.FFTW attribute*), 13
disable() (*in module pyfftw.interfaces.cache*), 34

E

empty_aligned() (*in module pyfftw*), 16
enable() (*in module pyfftw.interfaces.cache*), 34
execute() (*pyfftw.FFTW method*), 15
export_wisdom() (*in module pyfftw*), 15

F

fft() (*in module pyfftw.builders*), 20
fft() (*in module pyfftw.interfaces.dask_fft*), 27
fft() (*in module pyfftw.interfaces.numpy_fft*), 23
fft() (*in module pyfftw.interfaces.scipy_fftpack*), 25
fft2() (*in module pyfftw.builders*), 21
fft2() (*in module pyfftw.interfaces.dask_fft*), 28
fft2() (*in module pyfftw.interfaces.numpy_fft*), 24
fft2() (*in module pyfftw.interfaces.scipy_fftpack*), 26
fftn() (*in module pyfftw.builders*), 21
fftn() (*in module pyfftw.interfaces.dask_fft*), 28
fftn() (*in module pyfftw.interfaces.numpy_fft*), 24
fftn() (*in module pyfftw.interfaces.scipy_fftpack*), 26
FFTW (*class in pyfftw*), 10
flags (*pyfftw.FFTW attribute*), 13
forget_wisdom() (*in module pyfftw*), 15

G

get_input_array() (*pyfftw.FFTW method*), 15
get_output_array() (*pyfftw.FFTW method*), 15

H

hfft() (*in module pyfftw.interfaces.dask_fft*), 30
hfft() (*in module pyfftw.interfaces.numpy_fft*), 25

I

ifft() (*in module pyfftw.builders*), 20
ifft() (*in module pyfftw.interfaces.dask_fft*), 28
ifft() (*in module pyfftw.interfaces.numpy_fft*), 23
ifft() (*in module pyfftw.interfaces.scipy_fftpack*), 26
ifft2() (*in module pyfftw.builders*), 21
ifft2() (*in module pyfftw.interfaces.dask_fft*), 28
ifft2() (*in module pyfftw.interfaces.numpy_fft*), 24
ifft2() (*in module pyfftw.interfaces.scipy_fftpack*), 26
ifftn() (*in module pyfftw.builders*), 21
ifftn() (*in module pyfftw.interfaces.dask_fft*), 28
ifftn() (*in module pyfftw.interfaces.numpy_fft*), 24
ifftn() (*in module pyfftw.interfaces.scipy_fftpack*), 26
ihfft() (*in module pyfftw.interfaces.dask_fft*), 30
ihfft() (*in module pyfftw.interfaces.numpy_fft*), 25
import_wisdom() (*in module pyfftw*), 15
input_alignment (*pyfftw.FFTW attribute*), 13
input_array (*pyfftw.FFTW attribute*), 13
input_dtype (*pyfftw.FFTW attribute*), 13
input_shape (*pyfftw.FFTW attribute*), 13
input_strides (*pyfftw.FFTW attribute*), 13
irfft() (*in module pyfftw.builders*), 21

irfft() (*in module pyfftw.interfaces.dask_fft*), 29
irfft() (*in module pyfftw.interfaces.numpy_fft*), 24
irfft() (*in module pyfftw.interfaces.scipy_fftpack*), 26
irfft2() (*in module pyfftw.builders*), 21
irfft2() (*in module pyfftw.interfaces.dask_fft*), 29
irfft2() (*in module pyfftw.interfaces.numpy_fft*), 24
irfftn() (*in module pyfftw.builders*), 22
irfftn() (*in module pyfftw.interfaces.dask_fft*), 30
irfftn() (*in module pyfftw.interfaces.numpy_fft*), 25
is_byte_aligned() (*in module pyfftw*), 16
is_n_byte_aligned() (*in module pyfftw*), 16

N

N (*pyfftw.FFTW attribute*), 13
n_byte_align() (*in module pyfftw*), 16
n_byte_align_empty() (*in module pyfftw*), 16
next_fast_len() (*in module pyfftw*), 16
next_fast_len() (*in module pyfftw.interfaces.scipy_fftpack*), 26
normalise_idft (*pyfftw.FFTW attribute*), 14

O

ones_aligned() (*in module pyfftw*), 16
ortho (*pyfftw.FFTW attribute*), 14
output_alignment (*pyfftw.FFTW attribute*), 13
output_array (*pyfftw.FFTW attribute*), 13
output_dtype (*pyfftw.FFTW attribute*), 13
output_shape (*pyfftw.FFTW attribute*), 13
output_strides (*pyfftw.FFTW attribute*), 13

P

pyfftw (*module*), 10
pyfftw.builders (*module*), 18
pyfftw.builders._utils (*module*), 22
pyfftw.config.NUM_THREADS (*in module pyfftw*), 17
pyfftw.config.PLANNER_EFFORT (*in module pyfftw*), 17
pyfftw.interfaces (*module*), 30
pyfftw.interfaces.cache (*module*), 33
pyfftw.interfaces.dask_fft (*module*), 27
pyfftw.interfaces.numpy_fft (*module*), 23
pyfftw.interfaces.scipy_fftpack (*module*), 25
pyfftw.simd_alignment (*in module pyfftw*), 15

R

rfft() (*in module pyfftw.builders*), 21
rfft() (*in module pyfftw.interfaces.dask_fft*), 29
rfft() (*in module pyfftw.interfaces.numpy_fft*), 24
rfft() (*in module pyfftw.interfaces.scipy_fftpack*), 26
rfft2() (*in module pyfftw.builders*), 21
rfft2() (*in module pyfftw.interfaces.dask_fft*), 29
rfft2() (*in module pyfftw.interfaces.numpy_fft*), 24
rfftn() (*in module pyfftw.builders*), 22
rfftn() (*in module pyfftw.interfaces.dask_fft*), 29
rfftn() (*in module pyfftw.interfaces.numpy_fft*), 25

S

set_keepalive_time() (*in module pyfftw.interfaces.cache*), 34
simd_aligned (*pyfftw.FFTW attribute*), 13

U

update_arrays() (*pyfftw.FFTW method*), 14

Z

zeros_aligned() (*in module pyfftw*), 16