pyFFTW Documentation

Release 0.10.4

Henry Gomersall

Mar 31, 2017

Contents

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

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.

Quick and easy: the `pyfftw.interfaces` module

The easiest way to begin using `pyfftw` is through the `pyfftw.interfaces` module. This module implements two APIs: `pyfftw.interfaces.numpy_fft` and `pyfftw.interfaces.scipy_fftpack` which are (apart from a small caveat[1]) drop in replacements for `numpy.fft` and `scipy.fftpack` 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 use them as replacements at run-time through monkey patching.

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

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

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

# 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
```

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().

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))

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.

API Reference

`pyfftw` - The core

FFTW Class

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.

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

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

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

`numpy.fft` interface

`scipy.fftpack` interface

Caching

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

P