SciPy: Revamp `scipy.fftpack`

Project Abstract

`scipy.fftpack` provides several variants of fast fourier transforms for use in numerical and scientific computing applications. Currently this a python wrapper around it's namesake, the fortran fftpack library. However, a number of concerns exist about the precision and performance of the current fft implementations. Thus, it is desirable to allow 3rd party libraries to be used instead of fftpack, allowing for improved performance and accuracy.

This project would first design and implement a backend interface which will allow different libraries to be called underneath the `scipy.fftpack` interface. Then a selection of 3rd party fft libraries can be adapted to implement this interface and provide users with a range of backends to choose from. These backends may be selected at runtime to accelerate existing users of the `scipy.fftpack` interface without any changes to their code.

Detailed description

Fast fourier transforms (FFTs) are a class of algorithms which compute the discrete fourier transform of an array in O(n logn) time. FFTs are used heavily in scientific computing and signal processing both because they allow analysis of a signal in the 'frequency domain' and because it can reduce the algorithmic complexity of convolution-like operations from $O(n^2)$ to O(n logn).

There are many highly optimised implementations of the FFT, including portable libraries like FFTW and platform specific libraries such as intel's mkl fft[1] and nvidia cuFFT (available in python through CuPy). Both FFTW and mkl fft support multithreading to accelerate their transforms and cuFFT even employs GPGPU acceleration. It would be good if SciPy users could leverage these implementations without having to change their code. However, none of their licenses are compatible with SciPy and so they cannot be included directly. Moreover, there is no "one size fits all" solution - the best library may vary by use case and available hardware. This can be solved if SciPy allows these libraries to expose a `scipy.fftpack` compatible backend and provide a method for the user to select it at runtime. This will allow the user to decide the best backend for their use case, as well as avoiding licensing restrictions.

Design goals

These could be thought of as requirements but since there is much discussion left to be had, I would like to just give these as a starting point. This should help give context to the

[1] Technically portable but consider: https://software.intel.com/en-us/articles/optimization-notice

decisions made in the discussion of the design here but are free to be adapted in response to further discussion.

1. Any code that works with the current fftpack should work with all backends
2. Conversely, any code that *doesn't* work with the current fftpack should *not* work with any other backends.
3. Performance should be as close as possible to using the libraries own API directly
4. It should be possible to change the backend at runtime, after fftpack is imported
5. Backends should be as configurable as possible
6. It must be possible to register a backend without SciPy knowing about the package beforehand.

Rationale

The reasoning behind the first goal is extremely fundamental. It means that a user can take existing code and use a different fft backend without requiring any further changes to their code. This would absolutely be a hard requirement and I don't expect that to change, even after further discussions.

I anticipate the second goal may be rather more contentious though, especially since long double support in pyFFTW was listed as a reason in favour of backends. However, I think that the backends should be as interchangeable as possible. Whereas, any code relying on long double support will *need* to be run with FFTW. I agree with what Ralf Gommers has said on the mailing list: "if your code has become specific to one backend, then you may as well use that backend directly right?"

As for goal three, I expect that someone motivated to search around for different fft implementations is most likely concerned about performance. We should want `scipy.fftpack` to be the go-to option and not just a compromise for when you don't have time to rewrite your code. That means committing to good performance.

Goal four is perhaps best motivated by a use case: consider a user who has profiled their code and found that different points within their program benefit from different fft backends. This would allow them to change these out whilst the program is already running. For example, cuFFT may only be profitable for really large FFTs due to memory transfer overheads. Additionally, I expect this flexibility will make it easier to test and benchmark multiple backends at the same time.

The fifth goal may be difficult to attain but I still think is important. Both pyFFTW and mkl-fft allow the number of threads they use to be configured and pyFFTW has its planner effort as well. For some use cases these may be very important to performance. However, it would be far too easy for these backend specific configurations to violate the second goal, so caution will be required.

Components of the design

Front end

Generally speaking, this project doesn't aim to change the front end of scipy.fftpack at all. The meaning of calls to the current `scipy.fftpack` functions should not change. Moreover, if any changes are made to the interfaces, they should be completely backwards compatible and never break existing code. Anything else would violate the first goal of this project.

Back end

In its simplest form, each backend would be required to exactly mirror the current fftpack interface. This would simplify the front end implementation as all it has to do is to forward the function calls down to the backend

```
def fft(x, n=None, axis=-1, overwrite_x=False):
    return _backend_module.fft(x, n, axis, overwrite_x)
```

However, this fails to meet the design goals in a few ways. First, several fftpack replacements are missing the real-to-real transforms and so this could break the user's code. Instead, we would have to implement a fallback strategy. This could either mean enforcing a limited subset of functionality and implementing the other functions in terms of those (in principle only one fft and ifft are required but we could be more strict). OR, we could simply use fftpack as the backend in those cases. Some experimentation will likely be required to see what works best. Either way, this fallback mechanism might look something like

```
def _set_backend(backend_module):
    try:
        _dct_func = backend_module.dct
    except NameError:
        # either fftpack_backend.dct or a translation to fft
        _dct_func = _fallback_dct
```

Where the dct function now has to call _dct_func instead of calling the backend directly.

```
def dct(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False):
    return _dct_func(x, type, n, axis, norm, overwrite_x)
```

Secondly, what if the backend supports a different set of numpy dtypes? For example pyFFTW supports long doubles but In keeping with current fftpack behaviour, this should result in an error. Otherwise, calling code would break when used with the fftpack backend. Similarly, float16 values should always be promoted to float32 before the transform. So depending on the community's reaction, we may also want to add some pre-processing of the inputs to enforce these restrictions.

```
def dct(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False):
    # From current implementation, also converts float16 to float32
    x = _asfarray(x)
```

```
    if x.dtype not in (np.float32, np.float64,
                       np.complex64, np.complex128):
        raise ValueError("type {} is not supported".format(x.dtype))

    return _dct_func(x, type, n, axis, norm, overwrite_x)
```

Installing back ends

As for how to install backends, there are at least 3 ways that a user might want to install a new backed. First, since it should be possible to change back ends at runtime, the user must be able to set the backend from within python. We should provide a function `set_backend` that the user calls, after which any calls to `scipy.fftpack` would use their specified backend. This could be done similar to [matplotlib backends] where a fixed set of backends are treated as special and referenced by name,

```
fftpack.set_backend('pyfftw')
```

and any other custom backends would be accessed by a string of the form `'module://my_backend'`. However, this does mean adding implicit package dependencies where the user might otherwise expect the named backends to come as default. This would either need to be made clear in the documentation; or another option is to simply require the argument to be a module object. In this model, the user must import the library themself

```
import pyfftw.interfaces.scipy_fftpack as fftw_pack
fftpack.set_backend(fftw_pack)
```

Both options are reasonable in my opinion, though the second option is is slightly more verbose from the user's perspective.

We should also add an environment variable such as `SCIPY_FFTPACK_BACKEND` that the user can set before calling their program. This would allow users to switch backends without any source code modifications whatsoever; they just need to call their program like so

```
$ SCIPY_FFTPACK_BACKEND="pyfftw" python process.py
```

and with no further modification, they can leverage the new backend. This would require adding to fftpack's initialisation code

```
backend = os.environ.get('SCIPY_FFTPACK_BACKEND')
if backend is not None:
    fftpack.set_backend(backend)
```

We could also add a scipy dotfile such as '.scipyrc' that allows the user to specify their backend globally. This would mean keeping an entry in the config file:

```
fftpack_backend pyfftw
```

And then when scipy is initialised we should read the value from the dotfile. If combined with the environment variable then one would have to be assigned priority

```
backend = (os.environ.get('SCIPY_FFTPACK_BACKEND') or
           _dotfile.get('fftpack_backend'))
if backend is not None:
    fftpack.set_backend(backend)
```

However, supporting this would mean that whenever fftpack is initialised we perform extra file system calls which might be surprising for a user that isn't using this feature. Since there is no precedent within scipy for such config files, it would absolutely need to be discussed before heading down this direction.

Configuring back ends

In order to leverage the best performance from the back ends, it is desirable for users to configure various aspects of the backend. This might include the default number of threads to run on, or even to configure more backend specific options like FFTW's planning stage which is wasteful for one-off ffts but potentially crucial to the performance of many repeated ffts. In both of these cases, the user is able to configure this already through the use of package specific environment variables and configuration interfaces. However, it may be valuable to provide a generic interface so the user's code doesn't need to assume which backend is being used.

This does somewhat conflict with our goal that users code should be independent of the backend. Not all of the backends will fully support the same configuration options. However, this could be made to work if the options are phrased as requests or limits and not as requirements. For example, instead of a `num_threads` option we should have `max_threads` since backends without multithreading will always use 1 thread. Additionally, it might be possible to include FFTW's `planner_effort` if rephrased as a hint for whether you call the same ffts regularly. The fftw backend will then take this hint and translate it into an appropriate planner effort enum for internal use. This is tricky, however, as it might mislead users into thinking every backend will respond to these options.

For the user to set their preferred config, we could either add global state which is modified via an `update_config` function or family of functions. Alternatively, these could be passed as a set of additional arguments to each of the front end functions. Or indeed, both of these methods could be useful together with the global option providing the defaults and any function parameters overriding the global config. All of these will be very straightforward to implement in code but the main issue is just deciding what config options should be allowed. I anticipate that if everything is given a sensible default behaviour then it would be safe to start conservatively and add more config options slowly over time without breaking backwards compatibility.