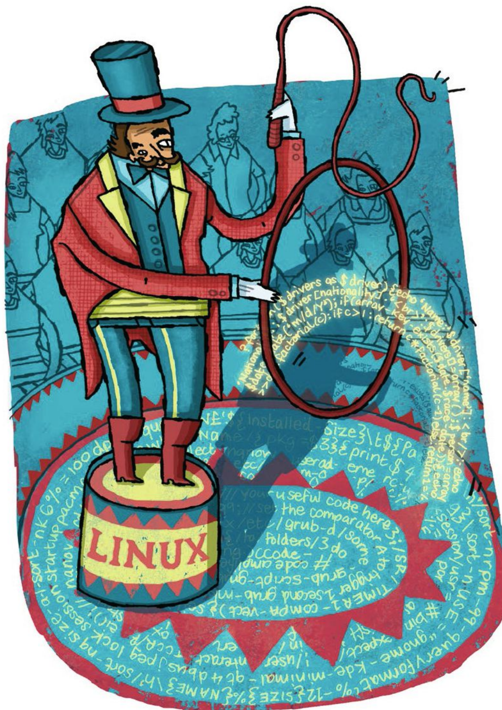
Cython: Speed up Python

Jonni Bidwell explains how to feed some Cython-flavoured accelerant into your system, using image compression as a working example.

Our expert

Jonni Bidwell is all about getting things done in a timely manner.



Python is a great language. It has a clean and easy-to-learn syntax and you can do an awful lot in a handful of lines. It's just not very fast, which, depending on your purposes, could be a deal-breaker. The main reason for this is that Python is interpreted: it is read line by line and converted on the fly to intermediate bytecode which gets shuffled around and eventually executed on the CPU. This takes time, but it makes life easier: there's no need to compile your code every time you change something, and there's no need to type your variables.

The interpreter will figure out which data type everything should be, and even if you change, say, a list into an integer, it will accommodate your changes without complaint. If you really want your Python code to go fast, then rewrite it in C and fast it will be. This is easier said than done, though: C is hard, and more often than not you'll only be interested in accelerating a handful of bottlenecks in your code.

Enter Cython, commonly misconceived as a Python-to-C translator. On some level this is true: Cython will take your Python code (slightly modified), and spit out a C file which you can compile and then import as an extension module, availing you of turbo-charged versions of all the functions in your original code. However, you still need your original code: the emitted module is there to convert the relevant parts of it to native machine code, rather than Python bytecode.

The Cython language is pretty much a superset of Python, so (excepting a few specialised modules and functions) any valid Python is also valid Cython, and as such can be saved as a PYX file and fed to the Cython binary. However, for optimal 'cythonising' one needs to use some of the extra Cython keywords, which can type variables (including function parameters and return types) and provide faster array access.

Many programs won't really gain anything from this Cython treatment, and if you're not careful you can end up actually slowing things down. For example, if your program spends most of its time drawing graphics, or is heavily I/O dependent, these are not things Cython can help you with. However, if your program is spending most of its life looping over arrays, shifting bits back and forth and doing arithmetic, then you are in luck.

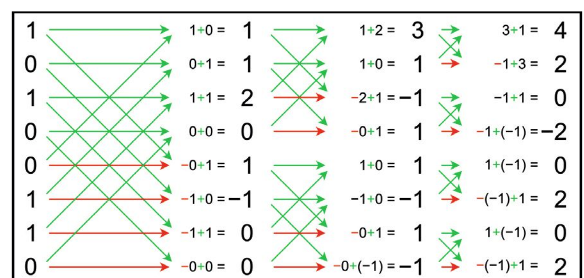
Compressing data

We're going to use Cython to speed up a crude implementation of the fast Walsh-Hadamard transform. We are going to use the transform to lossily compress greyscale image data, although the principle applies to any data. In the early days of satellite imagery, NASA used techniques like this since the transform relies only on computationally cheap addition and subtraction operations, and thanks to some mathematical trickery, the number of these operations can be reduced (down to $O(n \log n)$ from $O(n\hat{}2)$ if you care about such things).

An 8-bit greyscale image can be represented as a list of unsigned integers from 0 to 255, ie bytes. Each byte corresponds to the intensity of each pixel, and so a 256x256

Quick tip

If you've **cdef**'d everything and still want more speed, you can pass directives (such as the infamous **-O3**) to the compiler. Check the official docs – **http://bit.ly/CythonDocs**.



In-place addition and subtraction calculates the Walsh spectrum without having to multiply by a large matrix.

image will take up 65,536 bytes, or 64kB. The Walsh functions are a well-known family of functions which take on the values 1 or -1. By summing various component Walsh functions, it's possible to compose any discrete-valued function. For example, a row of pixels in our image, or even the whole image, could be exactly reproduced by, say, summing one Walsh function 300 times, subtracting 84 of another, adding 6 of yet another, then subtracting 2 of yet another other. The Walsh-Hadamard transform will tell you exactly which coefficients go with which functions quickly and efficiently.

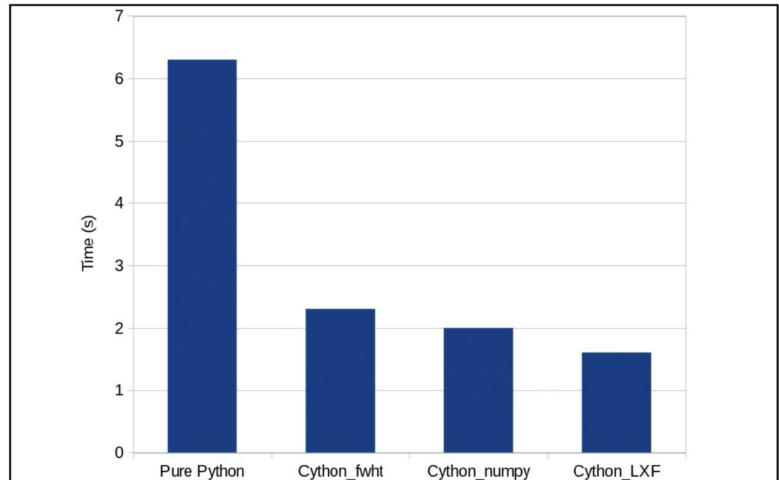
In practice, unless you're working with contrived data, there's no benefit to storing the parent function in this way (you usually need to sum as many functions as you have pixels, or data points). However, if you're not too worried about losing some data, then you can often get a very reasonable approximation of your data by discarding those functions with smaller Walsh coefficients. We won't worry too much about storing or even how to store the approximated image. Instead, we'll make some educated assumptions about its file size – in particular that each coefficient will take 10 bits to store (so that it can take values between -511 and 511) in addition to some bits for each index. We can show what the compressed image looks like, but it will still be represented as an uncompressed array in Python.

The Walsh-Hadamard transform

The Walsh-Hadamard transform is commonly represented as a matrix transform, where a power-of-two-sized square matrix multiplies a power-of-two-sized column vector (our data). The matrix is orthogonal and (when an appropriate scaling factor is used) unitary, so that the transform can be reversed by applying it again. The matrix is an example of a Hadamard matrix, the entries of the matrix (when the scaling factor is excluded) are all +/- 1, and the rows form the Walsh functions. (Fun fact: these were originally discovered 20 years before Walsh was born, in the context of eliminating crosstalk along parallel telegraph wires.)

The fast Walsh-Hadamard transform exploits the recursive structure of the Walsh matrix (it can be defined as a tensor product of 2x2 matrices) to perform the computation much quicker using some neat in-place calculations summed up in the diagram shown on the opposite page.

In the following code, we cheat a little here and use the **log2** function from NumPy. Don't worry too much about the logic arcana surrounding **j** and **k** below. It's just a neat way to recreate the butterfly structure shown in the diagram.



› The first optimisations bring the most benefit. After that, it's easy to spend hours trying to save a few milliseconds.

The algorithm works directly on the input, summing and subtracting pairs of entries, and so doesn't need to return anything as a result:

```
import numpy as np

def fwht(arr):
    n = len(arr)
    b = int(np.log2(n))

    for bit in range(b):
        for k in range(n):
            if k & (1 << bit) == 0:
                j = (1 << bit) | k
                tmp = arr[k]
                arr[k] += arr[j]
                arr[j] = tmp - arr[j]
```

The bitshift operators **<<** and **>>** aren't particularly quick in Python, but in C they correspond to a machine level operation and are much quicker than the equivalent literal multiplication or integer division by powers of two.

Our compression algorithm will read, using the Python imaging library, a greyscale image as a 1D array. We will divide this array into chunks and perform the transform on these chunks. We require a function to select and store the largest coefficients resulting from each of them. It makes sense to do some shifting and rounding here too; you can see the result in »

Cythonic decorations

As well as typing variables, we can also specify input or return types for functions. To do this, we define the function with **cdef** and then specify its return type before its name. For example, our core function **fwht** doesn't return anything, and hence should be typed **void**. After we have optimised the stuffing out of **fwht**, it then takes a memory view of C **int**s as input, so it's defined:

```
cdef void fwht(int[:] arr)
```

Using **cdef** means that your function won't be available to other Python modules, but you can use **cpdef** (which will incur a slight overhead) if you need your function to work from outside too. By **cimport**-ing the **cython** module, we can access a few decorators which change behaviours at the function level. For example, to turn off profiling for an individual function, use:

```
@cython.profile(False)
def too_cool_for_timing:
```

You'll find that this is particularly useful when used in conjunction with the **inline** keyword, which is used to 'unroll' small but frequently used functions, and for reducing the overhead associated with the function call. You will need to put the **inline** keyword right after **cdef**.

Finally, there are a couple of 'dangerous' things that are quite popular, namely:

```
@cython.boundscheck(False)
```

and:

```
@cython.cdivision(True)
```

which respectively deactivate out-of-bounds checking for arrays and checks for division by zero. You really should make sure that your code is correct before doing this, since they have the potential to corrupt memory.

» **If you missed last issue** Call 0844 848 2852 or +44 1604 251045

the function **squishChunk()** in the files on the **LXFDVD**. Decompression, via the **expandImage()** function, involves taking each chunk, collating the coefficient indices and magnitudes into a vector, and then performing the transform again, and shifting everything back to the 0-255 range. The chunks are then rejoined and we use the **show()** method to display the resulting lossily compressed image. This method requires you to be running an X server, since it uses the *xv* program (which you also require) to display.

You can test everything works by copying the directory on the **LXFDVD** to a local folder and from there running:

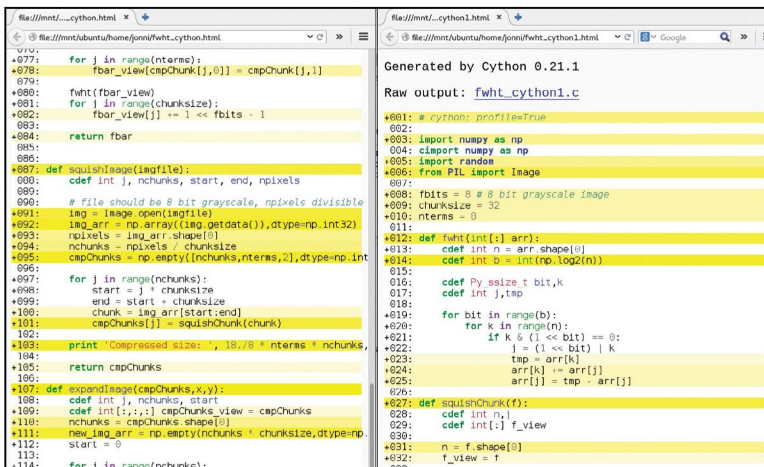
```
$ python proftest.py
```

This program will compress then expand a photo from the Philae module's new home: you can see it on the page opposite. You can experiment with the **chunksize** and **nterms** parameters at the beginning of the **fwht_python** file.

The initial values (32 and 8) give a nominal compression ratio of just over 2:1, although this is meaningless as we aren't storing the compressed data. It could also be vastly improved by varying the number of terms for each chunk – areas of the same colour need only a single term. We can profile this code using the cProfile module by running:

```
$ python -m cProfile proftest.py
```

This lists every single function involved in the program, including all the weird functions involved in decoding a PNG image, so we can filter this to show our own efforts by adding **| grep fwht** to the above. On a dusty **LXF** office machine, the whole execution took about five seconds, just over three seconds of which was spent in the **fwht** function: pretty reasonable, given that **fwht** is the heart of our program.



Using the -a option generates HTML files which show you the clean, white C-like code and the dirty, yellow Python code.

We ought to be able to speed this up quite a bit by using NumPy arrays instead of lists. NumPy arrays can be initialised with zeros, but it's slightly quicker to avoid this step (so the array initially contains whatever random data was in the memory allocated to it) if you know the values will be filled in later. You also need to specify a data type for the array, and it's better to not use methods for Python lists such as **len()**. You'll find the code in the file **fwht_numpy.py**. You'll also find that it takes about twice as long to run – benchmarking is full of surprises. Despite this slight disappointment, we'll stick with our arrays – Cython might do a better job with them.

Enter Cython

It makes sense to initially concentrate our efforts on speeding up the **fwht()** function, which is at present quite readable. A simple first step is to specify data types for all the local variables in this function. Although they are all integers, and you can get away with declaring them as such, the **for** loop indices have a special type **Py_ssize_t** so we may as well use it. Add the following lines at the beginning of the **fwht** function in **fwht_numpy.py** and save the file as, say, **fwht_cython1.pyx**:

```
def fwht(arr):
    cdef int n = arr.shape[0]
    cdef int b = int(np.log2(n))
    cdef Py_ssize_t bit,k
    cdef int j,tmp
```

Now run:

```
$ cython -a fwht_cython1.pyx[/b]
```

This will generate some very messy C in a file called **fwht_cython1.c**. The **-a** switch tells Cython to additionally generate a similarly named HTML file which you should look at. The lines with your newly typed variables are white, whereas most of the rest of the code will be various shades of yellow. You can even click on each line to see how it looks in C, and in so doing you will discover that the yellow lines correspond to lengthier or more involved code. We will still want to do some benchmarking, so add the following decorator at the top of the file:

```
# cython: profile=True
```

Getting your Cython code compiled is a little bit of effort. You can do it manually, but it's easier to use the **cythonize** function and the distutils module. Create a file **setup.py**

Benchmarking

It's easy to take benchmarks too seriously – graphics card enthusiasts have been doing so for years. In our tutorial we use the cProfile module which enables you to count and time each function call. This can provide valuable data about where the bottlenecks in your code are, which might not be immediately obvious.

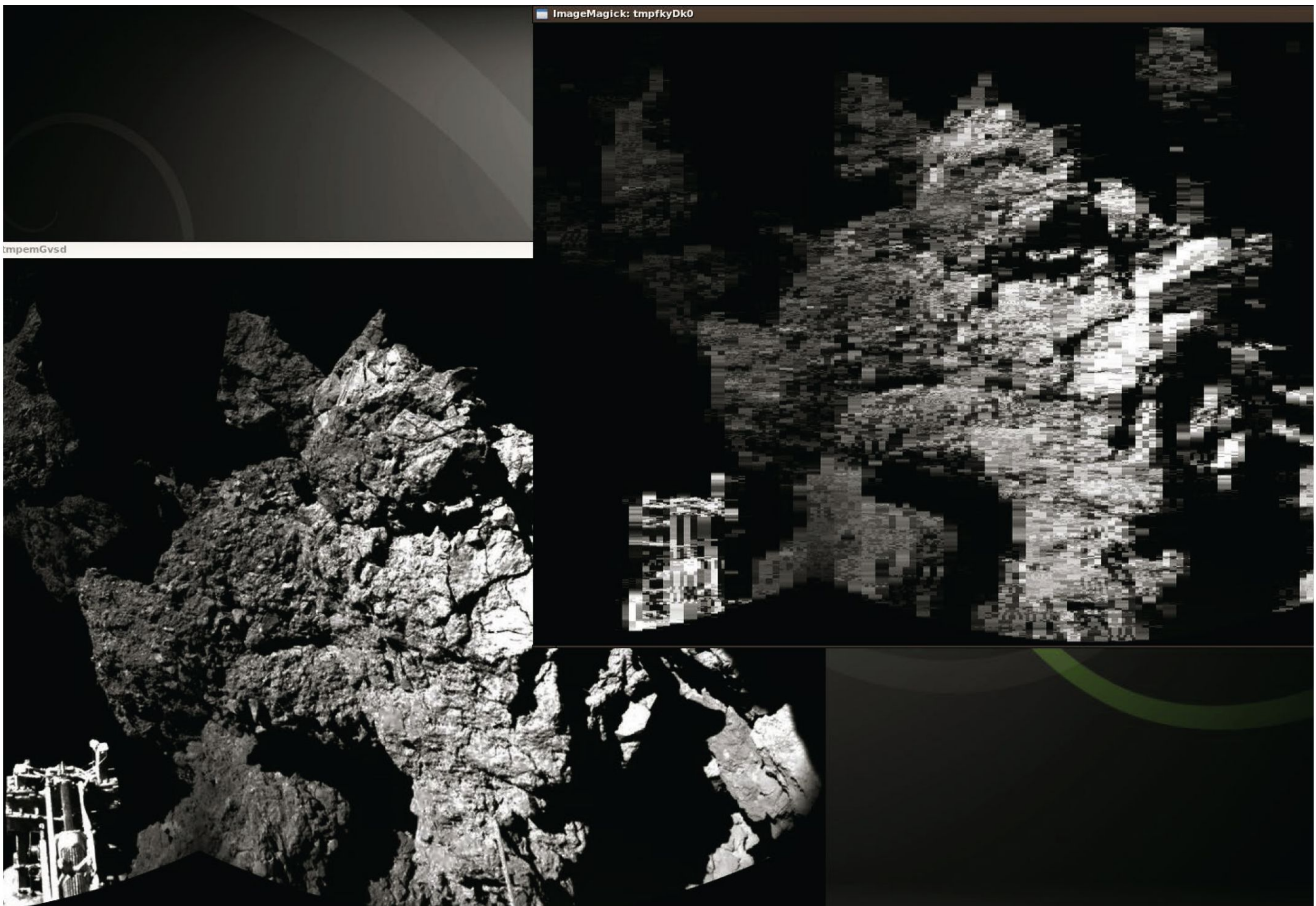
CProfile is designed to be as lightweight and unobtrusive as possible, but if you have a tiny function that's called millions of times, then that's millions of tallying calls and they all add up. If the function is really small, this means that more time is spent benchmarking than is spent doing whatever it is the function does, and so the result is largely meaningless. If you're confident a small function can't be sped up any more, it's best to disable the profiler. If you're not so confident, then by all means continue to benchmark such small functions, but rest assured that they will work a lot faster without the profiler interfering.

If you just want to measure 'wall time', which is the total time it takes for your code fragment to run, then the **timeit** module may be more appropriate. For example to test (from the interpreter) a function called **testfunction()** from a module **testmodule** three times:

```
import timeit
timeit.timeit(stmt='testmodule.testfunction()',
setup='import testmodule', number=3)
```

The default for **number** is a million, which is why it's a good idea to specify your own value here. You will need to specify your module in the **setup** parameter even if you have previously imported it, since **timeit** will not inherit this namespace.

Feed your inner mathematician Learn how to program in R on page 88.



› A postcard from comet 67P in both the original (left) and heavily compressed version (right). Landing stuff on extraterrestrial bodies is pretty much the only way to make space engineers hug each other.

(you can find an example on the **LXFDVD**) with the following contents:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
   ext_modules = cythonize("fwht_cython1.pyx")
)
```

Now if you run:

```
$ python setup.py build_ext --inplace
```

everything will be built and you can modify **proftest.py** to use your new **fwht_cython1** module. Benchmarking this (still using NumPy arrays) on our machine actually slowed things down – the **fwht()** function alone took about 20 seconds. But don't lose heart: the reason for the slowdown is that C has to access **arr** through Python and NumPy methods.

Memory views

Recently Cython introduced a new way to access array data through something called memory views. These enable C to directly access the array data in memory, and hence are damned fast. But they don't work with Python lists, which is why we are stuck with our slower NumPy arrays. We'll make **fwht()** work on a memory view by changing the definition line:

```
def fwht(int[:] arr):
```

The functions **squishChunk()** and **expandChunk()** will need to be modified too. So lets go ahead and define **f_view** in **squishChunk()** like so:

```
cdef int[:] f_view
f_view = f
```

Replace all further references in the function to the array **f** with **f_view**, with the exception of the **enumerate** call, which is a Python function. Likewise define **fbar_view** in **expandChunk()**, and replace all **fbar** references except the **return fbar** statement. While we're at it, we may as well type all the **j**s and **n**s and whatnot as **int**s too. Now re-run **setup.py** and benchmark. Now we're cooking with gas – the total execution time was less than three seconds, most of which is now spent in the **squishChunk()** function. The bottleneck here is ranking our coefficients and powers, so let's separate that into a separate **rankArray()** function, which is less reliant on Python constructs:

```
def rankArray(int[:] F):
   cdef int n,j
   n = F.shape[0]
   Franked = np.empty([n,3],dtype=np.int32)
   cdef int[:,:] Franked_view = Franked

   for j in range(n):
      Franked_view[j,0] = j
      Franked_view[j,1] = F[j]
      Franked_view[j,2] = - abs(F[j])
   Franked = Franked[Franked[:,2].argsort()]
   return Franked[:nterms,:2]
```

Now change the **return** line in **squishChunk()** to use this, and check the benchmarks. On our machine this shaved off nearly a second, and we were quite happy about that. From here on in it's diminishing returns, but we've provided as much optimisation as we can in the file **fwht_cython.pyx** on the **LXFDVD**. This took total execution time down to 1.6 seconds – see if you can do better! LXF