Writing Fast Code with Cython

Robert Bradshaw

SciPy 2008

What is Cython?

Cython is...

- an Open Source project
- a Python compiler (almost)
- an extended Python language for
  - writing fast Python extension modules
  - interfacing Python with C (C++, etc.) libraries

A bit of history

- April 2002: release of Pyrex 0.1 by Greg Ewing
  - Greg considers Pyrex a language in design phase
    - over the years, many people patched their Pyrex
    - not many patches were accepted by Greg
- minor forks and enhanced branches followed
  - March 2006: Stefan Behnel's fork of Pyrex for lxml XML toolkit
  - November 2006: SageX fork of Pyrex
    - by Robert Bradshaw, William Stein (Univ. Washington)
    - context: Sage, a free mathematics software package
- 28th July 2007: official Cython launch
  - integration of lxml's Pyrex fork into SageX
  - the rest is in http://hg.cython.org/cython-devel/

Speed

Cython generates very efficient C code

- according to PyBench
  - conditions and loops run 2-8x faster than in Py2.5
  - most benchmarks run 30%-80% faster
  - overall more than 30% faster for plain Python code
- optional type declarations
  - let Cython generate plain C instead of C-API calls
  - make code several times faster than the above (up to 100x)
- incremental optimization
  - optimize, don't re-write
  - only the pieces you need

How it Works

- the Cython compiler reads in a `.pxy` file and produces a `.c` file
- a C compiler (such as gcc) produces a `.so` file
- the resulting module is linked against the CPython library, and loaded into the Python runtime environment

The details can be handled by `distutils` and/or `pyximport`.

Lets get started

foo.pyx

```
print "Welcome to SciPy 2008!"
```

setup.py

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
  cmdclass = {'build_ext': build_ext},
  ext_modules = [ Extension("foo", ["foo.pyx"]) ],
)
```

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

Declaring types

The primary speedup in Cython comes from statically declaring types. This is done with the **cdef** keyword.

cdef keyword

```
cdef type var

def function(type arg, ...):
        ...

cdef return_type cfunction(type arg, ...):
        ...
```

In the future one will be able to provide typing information via decorators.

more on decorators: https://www.python-course.eu/python3_decorators.php

Declaring types

integrate.py

```
def f(x):
    return x*x*x - 3*x

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Goal: 100x speedup.

Declaring types

integrate.pyx

```
cdef double f(double x):
    return x**x*x - 3*x

def integrate_f(double a, double b, int N):
    cdef double s = 0
    cdef double dx = (b-a)/N
    cdef int i
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Using external functions

External functions are declared in a `cdef extern` block

integrate.pyx

```
cdef extern from "math.h":
    double sin(double)
    double cos(double)
    ...

...
```

Modify your integration code to approximate $\int_a^b \frac{1}{\sqrt{1+x^2}}$.

Using external functions

You may have to modify your `setup.py` file as well.

setup.py

```
ext_modules=[ Extension("integrate", ["integrate.pyx"],
                        libraries=['mymath'],
                        library_dirs=['extras'],
                        include_dirs=['extras']),
              ...
```

integrate.pyx

```
cdef extern from "mymath.h":
    double sinc(double)
```

Modify your integration code to approximate $\int_a^b \frac{\sin x}{x}$.

Using external functions

One can put declarations in a `.pxd` file for reuse.

cmath.pxd

```
cdef extern from "math.h":
    double sin(double)
    double cos(double)
    ...
```

integrate.pyx

```
from cmath cimport *
```

Several `.pxd` files are provided in `/Cython/Includes`. Additions are welcome.

Pointers and memory allocation

- Python is fully memory managed, C is not.
- Cython is *both*.

stdlib.pxd

```
cdef extern from "stdlib.h":
    ctypedef unsigned long size_t
    void *malloc(size_t size)
    void *realloc(void *ptr, size_t size)
    void free(void *ptr)
```

Pointers and memory allocation

A list of `ints` is an `int*`.

fib.pyx

```
from stdlib cimport malloc, free

def fib(n):
    cdef int* mylist = <int*>malloc(sizeof(int) * n)
    mylist[0], mylist[1] = 1, 1
    for i in range(n):
        mylist[i] = mylist[i-1]+mylist[i-2]
    print mylist[n-1]
    free(mylist) # or you will never get it back
```

Note: The python `list` type turns out to be quite fast for objects.

NumPy and Buffers

- NumPy arrays are fast, efficient, and flexible.
- This summer Dag Seljebotn was funded by Google and Enthought to provide easy C access to NumPy arrays, and did an excellent job.
- Up to 1000x speedup over pure Python code

NumPy and Buffers

fastnumpy.pyx

```
cimport numpy

def sum(x):
    cdef numpy.ndarray[int, ndim=1] arr = x
    cdef int i, s = 0
    for i in range(arr.shape[0]):
        s += arr[i]
    return s
```

This loop gets translated into pure C.

NumPy and Buffers

heat.pyx

```
def solve_heat(initial_conditions, dx, dt, iter):
    cur = initial_conditions.copy()
    next = numpy.zeros_like(initial_conditions)
    M, N = cur.shape
    for count in range(iter):
        print count
        for i in range(1, M-1):
            for j in range(1, N-1):
                step = cur[i-1,j] + cur[i+1,j] + cur[i,j-1] +
                next[i,j] = cur[i,j] + dt*step/dx^2
        cur, next = next, cur
    return cur
```

Goal: 1000x speedup.

NumPy and Buffers

To speed things up even more, you can declare

- `@cython.boundscheck(False)`
- unsigned int indices

Declaring Extension Classes

`.pyx` files may contain both normal and extension classes

Declaring Extension Classes

`.pyx` files may contain both normal and extension classes

Extension classes can have

- c attributes (public/readonly/private)
- c methods (cdef and cpdef)

Declaring Extension Classes

`.pyx` files may contain both normal and extension classes

Extension classes can have

- c attributes (public/readonly/private)
- c methods (cdef and cpdef)

However, extension classes do not have

- dictionaries
- multiple inheritance

Declaring Extension Classes

`.pyx` files may contain both normal and extension classes

Extension classes can have

- c attributes (public/readonly/private)
- c methods (cdef and cpdef)

However, extension classes do not have

- dictionaries
- multiple inheritance

Take care with special methods

- some methods (e.g. arithmetic) work differently
- No __new__, but __cinit__ and __dealloc__

Declaring Extension Classes

cclass.pyx

```
cdef class Foo(Bar):

    cdef int var

    def __init__(self, ...):
        ...
    def foo_py(self, ...):
        ...

    cdef foo_c(self, ...):
        ...
    cpdef foo_c_py(self, ...):
        ...
```

Classes must be `cimported` to use C-level properties.

More types

One can declare structs, unions, enums, and typedefs just as in C.

foo.pyx

```
ctypedef long foo

cdef struct point:
    double x
    double y
    double z

cdef enum color:
    red
    orange
    yellow
    ...
```

C++

Cython is not fully C++ aware (yet) but has the ability to wrap C++.

rect.h

```
class Rectangle {
public:
   int x0, y0, x1, y1;
   Rectangle(int x0, int y0, int x1, int y1);
   ~Rectangle();
   int getLength();
   int getHeight();
   int getArea();
   void move(int dx, int dy);
};
```

C++

setup.py

```
lang="c++"
```

rectangle.pyx

```
cdef extern from "rect.h":
    ctypedef struct c_Rectangle "Rectangle":
        int x0, y0, x1, y1
        int getLength()
        int getHeight()
        int getArea()
        void move(int dx, int dy)
    c_Rectangle *new_Rectangle "new Rectangle" (int x0, int y0
    void del_Rectangle "delete" (c_Rectangle *rect)
```

Common Pitfalls

Watch out for

- untyped extension classes
  - especially with public attributes and cpdef methods

Common Pitfalls

Watch out for

- untyped extension classes
  - especially with public attributes and cpdef methods
- excessive conversions

Common Pitfalls

Watch out for

- untyped extension classes
  - especially with public attributes and cpdef methods
- excessive conversions
- too much Python

Common Pitfalls

Watch out for

- untyped extension classes
  - especially with public attributes and cpdef methods
- excessive conversions
- too much Python

The easy answer: `cython -a` is your friend.

Questions?