ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Cython

What is Cython?

Cython is a Python-like language for writing extension modules. It allows one to mix Python with C or C++ and significantly lowers the barrier to speeding up Python code.

The **`cython`** command generates a C or C++ source file from a Cython source file; the C/C++ source is then compiled into a heavily optimized extension module.

Cython has built-in support for working with NumPy arrays and Python buffers, making numerical programming nearly as fast as C and (nearly) as easy as Python.

`http://www.cython.org/`

A really simple example

PI.PYX

```
# Define a function. Include type information for the argument.
def multiply_by_pi(int num):
    return num * 3.14159265359
```

SETUP_PI.PY

```
# Cython has its own "extension builder" module that knows how
# to build cython files into python modules.
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext = Extension("pi", sources=["pi.pyx"])

setup(ext_modules=[ext],
      cmdclass={'build_ext': build_ext})
```

See demo/cython for this example.
Build it using build.bat.

A simple Cython example

CALLING MULTIPLY_BY_PI FROM PYTHON

```
$ python setup_pi.py build_ext --inplace -c mingw32

>>> import pi
>>> pi.multiply_by_pi()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)
>>> pi.multiply_by_pi("dsa")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required
>>> pi.multiply_by_pi(3)
9.4247779607700011
```

(some of) the generated code

C CODE GENERATED BY CYTHON

```
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds); /*proto*/
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds) {
  int __pyx_v_num;
  PyObject *__pyx_r;
  PyObject *__pyx_1 = 0;
  static char *__pyx_argnames[] = {"num",0};
  if (!PyArg_ParseTupleAndKeywords(__pyx_args, __pyx_kwds, "i",
__pyx_argnames,
&__pyx_v_num)) return 0;

  /* "C:\pi.pyx":2 */
  __pyx_1 = PyFloat_FromDouble((__pyx_v_num * 3.14159265359));
if (!__pyx_1) {__
pyx_filename = __pyx_f[0]; __pyx_lineno = 2; goto __pyx_L1;}
  __pyx_r = __pyx_1;
  __pyx_1 = 0;
```

Def vs. CDef

DEF — PYTHON FUNCTIONS

```
# Python callable function.
def inc(int num, int offset):
    return num + offset

# Call inc for values in sequence.
def inc_seq(seq, offset):
    result = []
    for val in seq:
        res = inc(val, offset)
        result.append(res)
    return result
```

INC FROM PYTHON

```
# inc is callable from Python.
>>> inc.inc(1,3)
4
>>> a = range(4)
>>> inc.inc_seq(a, 3)
[3,4,5,6]
```

CDEF — C FUNCTIONS

```
# cdef becomes a C function call.
cdef int fast_inc(int num,
                  int offset):
    return num + offset
# fast_inc for a sequence
def fast_inc_seq(seq, offset):
    result = []
    for val in seq:
        res = fast_inc(val, offset)
        result.append(res)
    return result
```

FAST_INC FROM PYTHON

```
# fast_inc not callable in Python
>>> inc.fast_inc(1,3)
Traceback: ... no 'fast_inc'
# But fast_inc_seq is 2x faster
# for large arrays.
>>> inc.fast_inc_seq(a, 3)
[3,4,5,6]
```

CPdef: combines def + cdef

CPDEF — C AND PYTHON FUNCTIONS

```
# cdef becomes a C function call.
cpdef fast_inc(int num, int offset):
    return num + offset

# Calls compiled version inside Cython file
def inc_seq(seq, offset):
   result = []
   for val in seq:
       res = fast_inc(val, offset)
       result.append(res)
   return result
```

FAST_INC FROM PYTHON

```
# fast_inc is now callable in Python via Python wrapper
>>> inc.fast_inc(1,3)
4
# No speed degradation here
>>> inc.inc_seq(a, 3)
[3,4,5,6]
```

Functions from C Libraries

EXTERNAL C FUNCTIONS

```
# len_extern.pyx
# First, "include" the header file you need.
cdef extern from "string.h":
    # Describe the interface for the functions used.
    int strlen(char *c)

def get_len(char *message):
    # strlen can now be used from Cython code (but not Python)…
    return strlen(message)
```

CALL FROM PYTHON

```
>>> import len_extern
>>> len_extern.strlen
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'strlen'
>>> len_extern.get_len("woohoo!")
```

Structures from C Libraries

TIME_EXTERN.PYX

```
cdef extern from "time.h":
    # Declare only what is used from `tm` structure.
    struct tm:
        int tm_mday # Day of the month: 1-31
        int tm_mon  # Months *since* january: 0-11
        int tm_year # Years since 1900

    ctypedef long time_t
    tm* localtime(time_t *timer)
    time_t time(time_t *tloc)

def get_date():
    """ Return a tuple with the current day, month, and year."""
    cdef time_t t
    cdef tm* ts
    t = time(NULL)
    ts = localtime(&t)
    return ts.tm_mday, ts.tm_mon + 1, ts.tm_year
```

CALLING FROM PYTHON

```
>>> extern_time.get_date()
(8, 4, 2011)
```

Classes

SHRUBBERY.PYX

```
cdef class Shrubbery:
  # Class level variables
  cdef int width, height

  def __init__(self, w, h):
    self.width = w
    self.height = h
  def describe(self):
    print "This shrubbery is",self.width,"by ",self.height," cubits."
```

CALLING FROM PYTHON

```
>>> import shrubbery
>>> x = shrubbery.Shrubbery(1, 2)
>>> x.describe()
This shrubbery is 1 by 2 cubits.
>>> print x.width
AttributeError: 'shrubbery.Shrubbery' object has no attribute 'width'
```

Classes from C++ libraries

rectangle_extern.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);
};
```

The implementation of the class and methods is done inside rectangle_extern.cpp. See demo/cython for this example.

Classes from C++ libraries

rectangle.pyx

```
cdef extern from "rectangle_extern.h":
    cdef cppclass Rectangle:
        Rectangle(int, int, int, int)
        int x0, y0, x1, y1
        int getLength()
        int getHeight()
        int getArea()
        void move(int, int)

cdef class PyRectangle:
    cdef Rectangle *thisptr   # hold a C++ instance which we're wrapping
    def __cinit__(self, int x0, int y0, int x1, int y1):
        self.thisptr = new Rectangle(x0, y0, x1, y1)
    def __dealloc__(self):
        del self.thisptr
    def getLength(self):
        return self.thisptr.getLength()
```

Classes from C++ libraries

SETUP.PY

```
from distutils.core import setup
from Cython.Distutils import build_ext
from distutils.extension import Extension
setup(
  ext_modules=[Extension("rectangle", sources = ["rectangle.pyx",
                                                 "rectangle_extern.cpp"],
                language = "c++")],
  cmdclass = {'build_ext': build_ext})
```

CALLING FROM PYTHON

```
In [1]: import rectangle
In [2]: r = rectangle.PyRectangle(1,1,2,2)  # calls __cinit__
In [3]: r.getLength()
Out[3]: 1
In [4]: r.getHeight()
AttributeError:rectangle.PyRectangle object has no attribute getHeight
In [5]: del r  # calls __dealloc__
```

Using NumPy with Cython

```
#cython: boundscheck=False
# Import the numpy cython module shipped with Cython.
cimport numpy as np
ctypedef np.float64_t DOUBLE

def sum(np.ndarray[DOUBLE] ary):
    # How long is the array in the first dimension?
    cdef int n = ary.shape[0]
    # Define local variables used in calculations.
    cdef unsigned int i
    cdef double sum
    # Sum algorithm implementation.
    sum = 0.0
    for i in range(0, n):
        sum = sum + ary[i]
    return sum
```

```
C:\demo\cython>test_sum.py
elements: 1,000,000
python sum(approx sec, result): 7.030
numpy sum(sec, result): 0.047
cython sum(sec, result): 0.047
```

Problem: Make This Fast!

```
def mandelbrot_escape(x, y, n):
    z_x = x
    z_y = y
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i

def generate_mandelbrot(xs, ys, n):
    d = empty(shape=(len(ys), len(xs)))
    for j in range(len(ys)):
        for i in range(len(xs)):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

Step 1: Add Type Information

Type information can be added to function signatures:

```
def mandelbrot_escape(double x, double y, int n):
    ...
def generate_mandelbrot(xs, ys, int n):
    ...
```

Variables can be declared to have a type using 'cdef':

```
def generate_mandelbrot(xs, ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    ...
```

Step 2: Use Cython C Functions

In Cython you can declare functions to be C functions using 'cdef' instead of 'def':

```
cdef int mandelbrot_escape(float x, float y, int n):
    ...
```

This makes the functions:

- Generate actual C functions, so they are much faster.
- Not visible to Python, but freely usable in your Cython module.

Arbitrary Python objects can still be passed in and out of C functions using the 'object' type.

Solution 2: This is Fast!

```
cdef int mandelbrot_escape(double x, double y, int n):
    cdef double z_x = x
    cdef double z_y = y
    cdef int i
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i
def generate_mandelbrot(xs, ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    d = empty(dtype=int, shape=(N, M))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

Step 3: Use NumPy in Cython

In our example, we are still using Python-level numpy calls to do our array indexing:

```
d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
```

If we use the Cython interface to NumPy, we can declare our arrays to be C-level numpy extension types, and gain even more speed.

NumPy arrays are declared using a special buffer notation:

```
cimport numpy as np
...
cdef np.ndarray[int, ndim=2] my_array
```

You must declare both the type of the array, and the number of dimensions. All the standard numpy types are declared in the numpy cython declarations.

Solution 3: This is *Really* Fast!

mandel.pyx

```
cimport numpy as np
...
def generate_mandelbrot(np.ndarray[double, ndim=1] xs,
        np.ndarray[double, ndim=1] ys, int n):
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)
    cdef np.ndarray[int, ndim=2] d = empty(dtype=int, shape=(N, M))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

setup.py

```
...
import numpy
...
ext = Extension("mandel", ["mandel.pyx"],
    include_dirs = [numpy.get_include()])
```

Step 4: Parallelization using OpenMP

Cython supports native parallelism. To use this kind of parallelism, the Global Interpreter Lock (GIL) must be released. It currently supports OpenMP (more backends might be supported in the future).

1) Release the GIL before a block of code:

```
with nogil:
    # This block of code is executed after releasing the GIL
```

2) Declare that a cdef function can be called safely without the GIL:

```
cdef int mandelbrot_escape(double x, double y, int n) nogil:
    ...
```

3) Parallelize for-loops with prange:

```
from cython.parallel import prange
...
for j in prange(M):
    for i in prange(N):
        d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
```

Solution 4: Even faster!

mandel.pyx

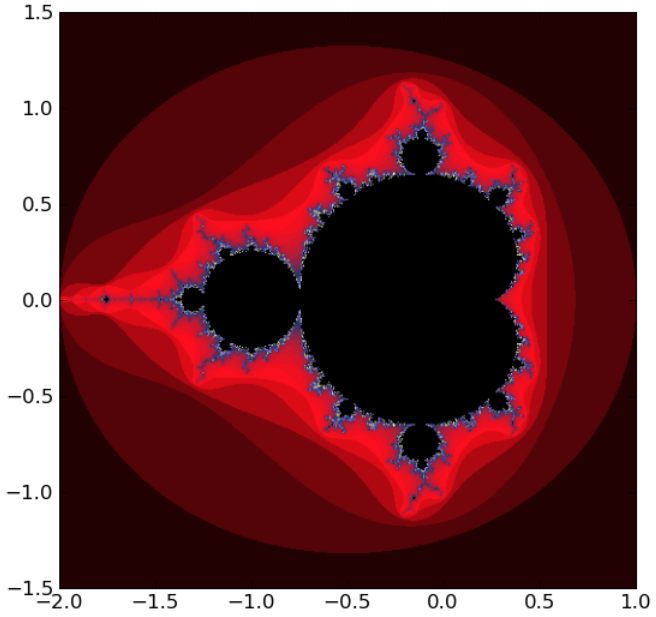
```
from cython.parallel import prange
...
cdef int mandelbrot_escape(double x, double y, int n) nogil:
...
def generate_mandelbrot(np.ndarray[double, ndim=1] xs,
                        np.ndarray[double, ndim=1] ys, int n):
    """ Generate a mandelbrot set """
    cdef ...
    with nogil:
        for j in prange(M):
            for i in prange(N):
                d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

setup.py

```
ext = Extension("mandel", ["mandelbrot.pyx"],
                include_dirs = [numpy.get_include()],
                extra_compile_args=['-fopenmp'],
                extra_link_args=['-fopenmp'])
```

Conclusion

Solution	Time	Speed-up
Pure Python	630.72 s	x 1
Cython (Step 1)	2.7776 s	x 227
Cython (Step 2)	1.9608 s	x 322
Cython+Numpy (Step 3)	0.4012 s	x 1572
Cython+Numpy+prange (Step 4)	0.2449 s	x 2575



Timing performed on a 2.3 GHz Intel Core i7 MacBook Pro with 8GB RAM using a 2000x2000 array and an escape time of n=100.

[July 20, 2012]