

Scientific computing using Cython: Best of both worlds!

July 14, 2017
EuroPython 2017

About me

Simmi Mourya

Deep learning Engineer, Predible
Health

<http://prediblehealth.com/>

Github: @simmimourya1

What this talk is about?

What is Cython?

Unleash Cython superpowers.

Build your first cython module.

Cython and NumPy

Unleash more superpowers.

Project demo: Cyvlfeat

What is Cython?

Optimising static compiler for both the Python and Cython

It makes writing C extensions for Python as easy as Python itself.

UNLEASH SUPERPOWERS.



SUPERPOWERS

I HAS THEM

-
- Write Python code that calls back and forth from and to C or C++ code natively at any point.

-
- Easily tune readable Python code into plain C performance by adding static type declarations.

Pure Python code:

```
def f(x):  
    return x**2-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
```


With additional
Static Type
declarations: 4
times speedup

```
def f(double x):  
    return x**2-x
```

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

Def

def - Basically, it's Python

def is used for code that will be:

- Called directly from Python code with Python objects as arguments.
- Returns a Python object

Cdef

cdef - Basically, it's C

cdef is used:

- Where Cython functions are intended to be pure 'C' functions.
- All types *must* be declared.
- The generated code is about as fast as you can get though.

Cpdef

cpdef - It's both

cpdef combines both **def** and **cdef** by creating two functions; a **cdef** for C types and a **def** for Python types.

- Uses early binding when using C fundamental types
- Uses dynamic binding when Python objects are passed

→ It can call C/C++ libraries directly.

‘cimport’

→ Integrate natively with existing code.

A meme featuring a tabby cat with its front paws raised, emitting colorful laser beams (yellow, green, blue, and purple) that point towards a city skyline at night. The cat has a determined expression. The text "THE FORCE IS STRONG WITH THIS ONE." is overlaid in the center.

THE FORCE IS STRONG WITH THIS ONE.

Where's the magic?



Motivation

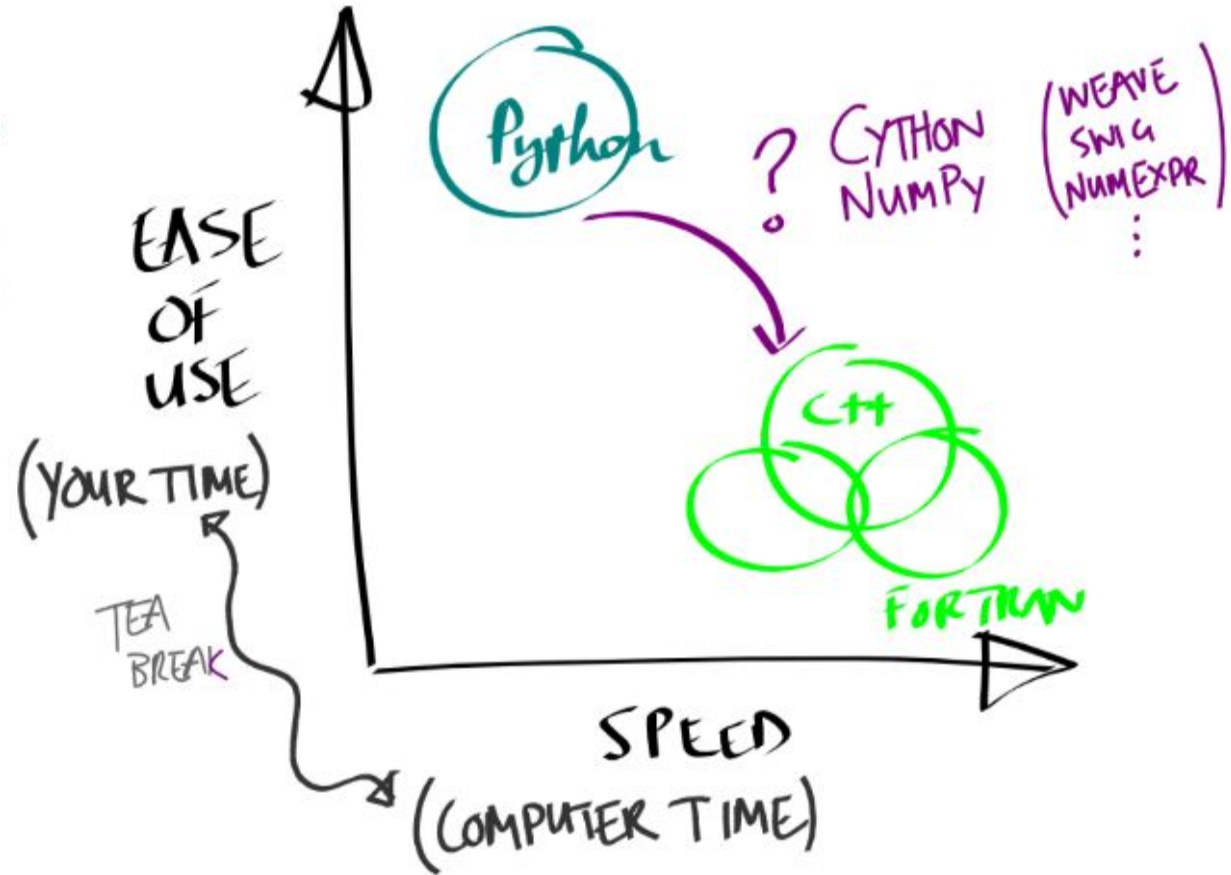


Illustration borrowed from: Stéfan van der Walt's presentation at Advanced Python Summer School, Kiel 2012.

Cython by example

Python

1x

```
def fib(n):  
    a,b = 1,1  
    for i in range(n):  
        a, b = a+b, a  
    return a
```

C/C++

100x

```
int fib(int n)  
{  
    int tmp, i, a, b;  
    a = b = 1;  
    for(i=0; i<n; i++){  
        tmp = a;  
        a += b;  
        b = tmp;  
    }  
}
```

Cython

80x

```
def fib(int n):  
    cdef int i, a, b  
    a, b = 1,1  
    for i in range(n):  
        a, b = a+b, a  
    return a
```

Cython in wild

Project	Cython files	Cython SLOC
sage	761	477,000
numpy	14	5,000
scipy	28	24,000
pandas	21	27,000
lxml	12	22,000
scikits-learn	35	15,000
scikits-image	48	11,000
mpi4py	48	12,000
yt	45	18,000

Projects master branches as of November 2014

A close-up photograph of a kitten's face. The kitten has light blue eyes and is looking directly at the camera. Its mouth is wide open, showing its pink tongue and teeth. The kitten's fur is a mix of white and brown. The background is dark and out of focus.

CODING IN CYTHON IS LIKE CODING IN
PYTHON AND C AT THE SAME TIME!

Use Case 1: Library Wrapping

Cython is a popular choice for writing Python interface modules for C libraries

Use Case 2: Performance-critical code

- Python
- High-level
- Slow
- No variables typed
- C/C++/Fortran
- Lower-level
- Fast
- All variables typed

Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python.

Use Case 1: Breaking out of the GIL

Cython As a few of you might know, C Python has an infamous Global Interpreter Lock (GIL)

- It limits thread performance

The Unwritten Rules of Python

- You do not talk about the GIL.
- You do NOT talk about the GIL.

**Don't even
mention the
GIL.
No seriously...**



Consider this simple script, that runs twice, sequentially, a `busy_sleep`, i.e. a function that simulate a CPU intensive task:

```
from def busy_sleep(n):  
    while n > 0:  
        n -= 1  
N = 99999999  
busy_sleep(N)  
    busy_sleep(N)
```

This takes 6.7 seconds to run.

Now consider the threaded version:

```
from threading import Thread
def busy_sleep(n):
    while n > 0:
        n -= 1
N = 999999999
t1 = Thread(target=busy_sleep, args=(N, ))
t2 = Thread(target=busy_sleep, args=(N, ))
t1.start()
t2.start()
t1.join()
t2.join()
```

This takes 11.1 seconds to run. What?

Cython offers a wonderful context manager to run instructions without the GIL: *with nogil*.

Demo

With Nogil

```
def busy_sleep(int n):  
    _busy_sleep(n)  
  
def busy_sleep_nogil(int n):  
    with nogil:  
        _busy_sleep(n)  
  
cdef inline void _busy_sleep(int n) nogil:  
    cdef double tmp = 0.0  
    while n > 0:  
        tmp = (n ** 0.5) ** 0.5  
        n -= 1
```

Annotated version of `_busy_sleep` function

```
1: def busy_sleep(int n):  
2:     _busy_sleep(n)  
3:  
4:  
5: def busy_sleep_nogil(int n):  
6:     with nogil:  
7:         _busy_sleep(n)  
8:  
9:  
10: cdef inline void _busy_sleep(int n) nogil:  
11:     while n > 0:  
12:         n -= 1
```

Building Cython code

Building Cython code

Cython code must, unlike Python, be compiled. This happens in two stages:

- A .pyx file is compiled by Cython to a .c file, containing the code of a Python extension module
- The .c file is compiled by a C compiler into a .so file (or .pyd on Windows) which can be imported directly into a Python session.

Building Cython code

There are several ways to build Cython code:

- Write a distutils setup.py.
- Use pyximport, importing Cython .pyx files as if they were .py files (using distutils to compile and build in the background).
- Use the Jupyter notebook or the Sage notebook, both of which allow Cython code inline.

Building Cython modules using distutils

Imagine a simple “hello world” script in a file `hello.pyx`:

```
def say_hello_to(name):  
    print("Hello %s!" % name)
```

The following could be a corresponding `setup.py` script:

```
from distutils.core import setup  
from Cython.Build import cythonize  
  
setup(  
    name = 'Hello world app',  
    ext_modules = cythonize("hello.pyx"),  
)
```

Building Cython modules using distutils

To use this to build your Cython file use the command line options:

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

It leaves a file in your local directory called helloworld.so (unix) or helloworld.pyd (Windows).

```
>>> import helloworld  
Hello World
```

pyximport: Cython Compilation the Easy Way

To load .pyx files directly on import, without having to write a setup.py file.

```
>>> import pyximport; pyximport.install()  
>>> import hello  
Hello World
```

The conclusions:

1. Naive Cython does speed things up, but not by much (x1.8)
2. Optimised Cython is fairly effortless (in this case) and worthwhile (x2.5)
3. cpdef gives a good improvement over def
4. cdef is really valuable (x72)
5. Cython's cdef is insignificantly different from the more complicated C extension that is our best attempt

NumPy and Cython

Provides fast access to NumPy arrays

It has a C-level type -> *typed memoryview*

Similar to NumPy array buffer support.

Allows us to work with buffers without having to know the details.



Example: Typed Memoryviews

The `double[:] foo` syntax declares `foo` to be a typed memoryview.

```
def sum_arg(double[:] foo):  
    """Sums its argument's contents."""  
    cdef double i, total = 0.0  
    for i in foo:  
        total += i  
    return total
```


Providing C-Level access to Typed Memoryview data

```
def sum_arg(double[:] foo):  
    """Sums its argument's contents."""  
    cdef:  
        double total = 0.0  
        int j, length  
        length = foo.shape[0]  
    for j in range(length):  
        total += foo[j]  
    return total
```



Trading Safety for performance

Trading Safety for performance

```
from cython cimport boundscheck, wraparound
def sum_arg(double[:] foo):
    ...
    ...

    with boundscheck(False), wraparound(False)
        for j in range(length):
            ...
```

Trading Safety for performance

```
from cython cimport boundscheck, wraparound
@boundscheck(False)
@wraparound(False)

def sum_arg(double[:] foo):
    ...
    ...

    with boundscheck(False), wraparound(False)
        for j in range(length):
            ...
```

Trading Safety for performance

```
# cython: boundscheck = False
# cython: wraparound = False

def sum_arg(double[:] foo):
    ...
    ...

    with boundscheck(False), wraparound(False)
        for j in range(length):
            ...
```





So what have we learned?

PROJECT DEMO
CYVLFEAT
GOOGLE SUMMER OF CODE 2016
@ PORTLAND STATE UNIVERSITY

A CYTHON / PYTHON WRAPPER FOR VLFEAT



WHAT IS VLFEAT?



WHAT IS CYVLFEAT?

[GITHUB.COM/MENPO/CYVLFEAT](https://github.com/menpo/cyvlfeat)

A photograph of a man and two children playing soccer on a grassy field. The man, in the center, is wearing a white soccer jersey with red accents and red shorts. He is smiling and has his arms around two young boys. The boy on the right is wearing a black soccer jersey and black shorts, also smiling. The boy on the left is wearing a black hoodie and black shorts, and is walking away. The background shows a black fence and some trees. The text "CONTRIBUTIONS ARE WELCOME!" and "GITHUB.COM/SIMMIMOURYA1/CYVLFEAT" is overlaid in white, handwritten-style font.

CONTRIBUTIONS ARE WELCOME!
GITHUB.COM/SIMMIMOURYA1/CYVLFEAT

A dark, low-key photograph of a soccer match at night. The scene is dimly lit, with a bright light source from above creating a circular glow on the field. Several players in various colored jerseys are visible, running across the grass. The overall atmosphere is dark and moody.

FEATURES WHICH NEED MORE WORK

1. BINSUM
2. KDTREE MODULE

QUESTIONS ?



THANK YOU



github.com/simmimourya1/europython17