The Quest for Speed:
An Introduction to Cython

Presented at the G-Node Autumn School on

Advanced Scientific Programming in Python,

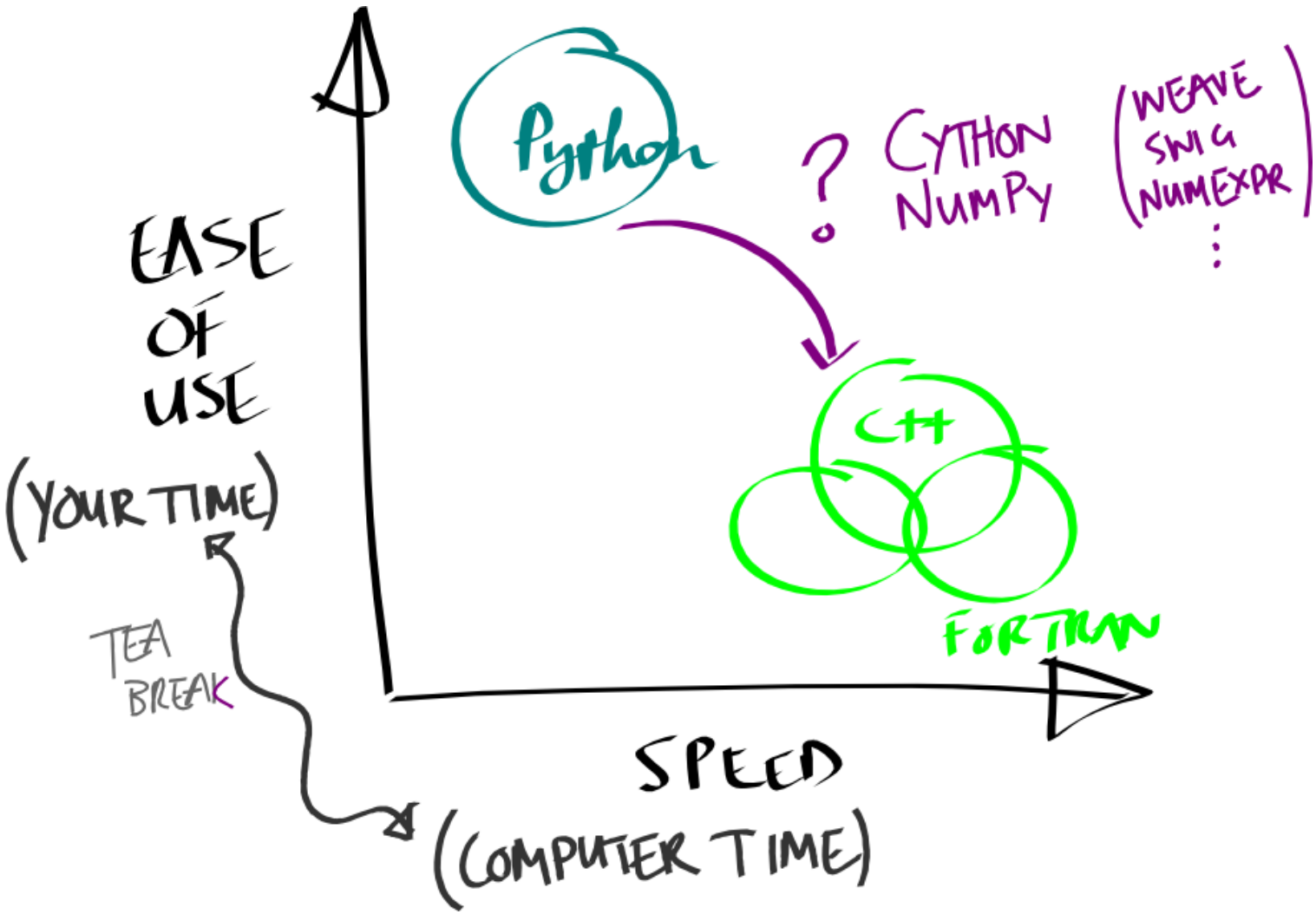
held in Trento, Italy

Stéfan van der Walt
Stellenbosch University, South Africa

October, 2010

Introduction

Motivation



Motivation (continued)

- Cython allows us to cross the gap!
- This is good news because
  - we get to keep coding in Python (or something close to Python)
  - we get the speed advantage of C
- You can't have your cake and eat it. (*Non si puo avere la botte piena è la moglie ubriaca*.) But this comes pretty close!
- Cython originates from Pyrex (been used in NumPy's `mtrand` module for a long time, e.g.); it is well maintained with an active user community, wide adoption.

Use Cases

- Optimising execution of Python code (profile, if possible!)
- Wrapping existing C, C++ (and soon Fortran) code
- Breaking out of the GIL!
- Mixing C and Python, but without the pain of the Python C API

Tutorial Overview

For this quick introduction, we'll take the following route:

1. Take a piece of pure Python code and benchmark (we'll find that it is too slow)
2. Run the code through Cython, compile and benchmark (we'll find that it is somewhat faster)
3. Annotate the types and benchmark (we'll find that it is much faster)

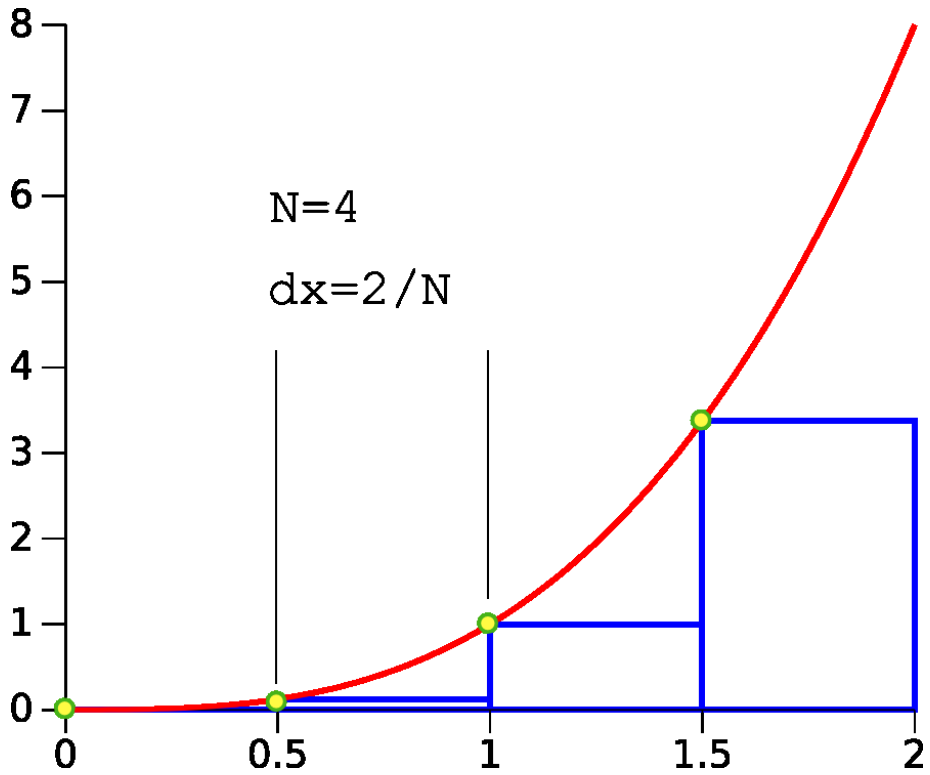
Then we'll look at how Cython allows us to

- Work with NumPy arrays
- Use multiple threads from Python
- Wrap native C libraries

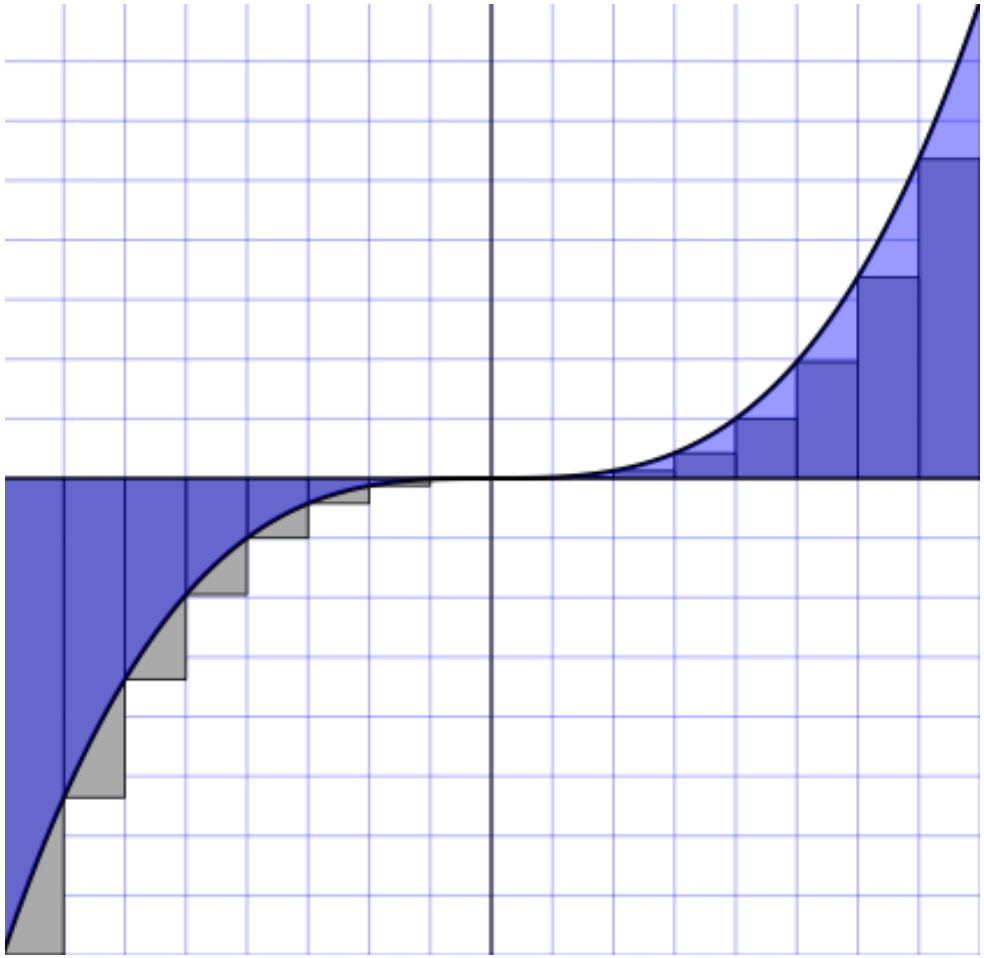
From Python to Cython

Benchmark Python code

Our code aims to compute (an approximation of) $\int_a^b f(x)dx$



More Segments



Benchmark Python Code


```
from __future__ import division

def f(x):
    return x**4 - 3 * x

def integrate_f(a, b, N):
    """Rectangle integration of a function.

    Parameters
    ----------
    a, b : ints
        Interval over which to integrate.
    N : int
        Number of intervals to use in the discretisation.

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

Apply Cython to the Python code

- `cython filename.[py|pyx]`
- What is happening behind the scenes? `cython -a filename.[py|pyx]`
- Cython translates Python to C, using the Python C API (let's have a look)
- Cython has a basic type inferencing engine, it is very conservative for safety reasons.
- This code has some serious *bottlenecks.*

Compile generated code

In `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("integrate_cy",
              ["integrate.py"],
             ),
  ])
```

Run using `python setup.py build_ext -i`. This means: build extensions «in-place».

If no extra C libraries or special build setup are needed, you may use pyximport to automagically compile `.pyx` files:

```
>>> import pyximport; pyximport.install()
```

Benchmark the new code

- `Use IPython's %timeit` (could do this manually using `from timeit import timeit; timeit(...)`)
- Slight speed increase ($\approx 1.4\times$) probably not worth it.
- Can we help Cython to do even better?
  - Yes—by giving it some clues.

Annotate Types using Decorators (Pure Python)


```
# This code still runs under Python!
from __future__ import division
import cython

@cython.locals(x=cython.double)
def f(x):
    return x**4 - 3 * x

@cython.locals(a=cython.double, b=cython.double,
               N=cython.int, s=cython.double,
               dx=cython.double, i=cython.int)
def integrate_f(a, b, N):
    """Rectangle integration of a function.
    ...
    """
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

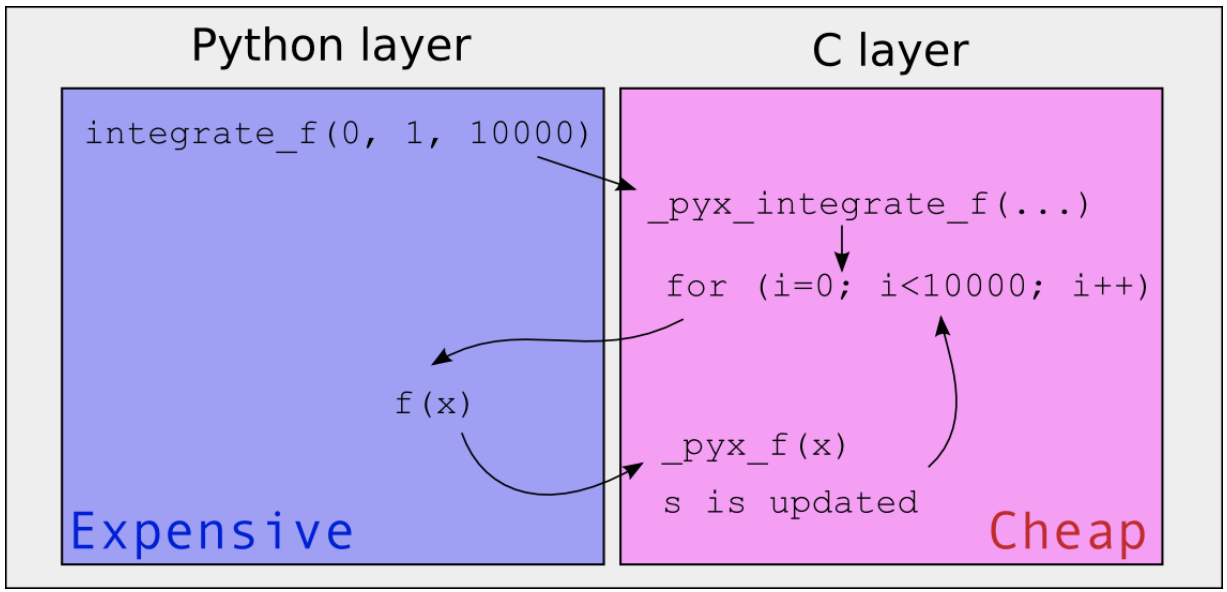
Benchmark...

Alternative syntax


```
# This code DOES NOT run under Python!
from __future__ import division

def f( double x ):
    return x**4 - 3 * x

def integrate_f( double a, double b, int N ):
    """Rectangle integration of a function.
    ...
    """
    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
```

Expense of Python Function Calls


```
def f(double x):
    return x**4 - 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
```



The Last Bottlenecks


```
# cython: cdivision=True

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

Benchmark!

Exploring Cython Further

Handling NumPy Arrays

Build Setup for Numpy


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

setup(
  cmdclass = {'build_ext': build_ext},
  ext_modules = [
    Extension("matmul",
              ["matmul.py"],
              include_dirs=[numpy.get_include()],
             ),
  ])
```

Declaring the Array Type


```
cimport numpy as np

def foo( np.ndarray[np.float64_t, ndim=2] arr ):
    cdef int i, j
    for i in range( arr.shape[0] ):
        for j in range(arr.shape[1]):
            arr[i, j] = i + j
```

Different types are defined in the file `/usr/share/pyshared/Cython/Includes/numpy.pxd` on your virtual machines.

Matrix Multiplication


```
out = np.zeros(A.shape[0], B.shape[1])

# Take each row of A
for i in range(0, A.shape[0]):

    # And multiply by every column of B
    for j in range(B.shape[1]):
        s = 0
        for k in \
            range(A.shape[1])
            s += A[i, k] *
                 B[k, j]

        out[i, j] = s
```

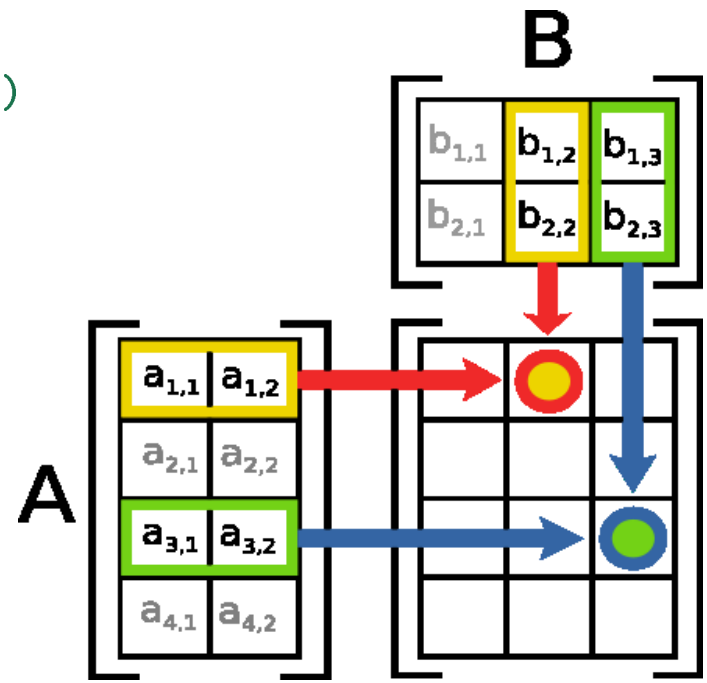


Image credits: http://en.wikipedia.org/wiki/File:Matrix_multiplication_diagram_2.svg

Our Own MatMul

We won't even try this in pure Python (way too slow).


```
cimport numpy as np

def matmul(np.ndarray[np.float64_t, ndim=2] A,
           np.ndarray[np.float64_t, ndim=2] B,
           np.ndarray[np.float64_t, ndim=2] out):

    cdef int i, j, k
    cdef np.float64_t s

    # Take each row of A
    for i in range(0, A.shape[0]):

        # And multiply by every column of B
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]

            out[i, j] = s
```

Parallel Threads with Cython

Averting the Global Interpreter Lock


```
@cython.boundscheck(False)
def matmul_partitioned(int start, int end,
                       np.ndarray[np.float64_t, ndim=2] A,
                       np.ndarray[np.float64_t, ndim=2] B,
                       np.ndarray[np.float64_t, ndim=2] out):
    cdef int i, j, k
    cdef np.float64_t s

    with nogil:

        # Take a selected few rows from A
        for i in range(start, end):

            # And multiply each column of B
            for j in range(B.shape[1]):
                s = 0
                for k in range(A.shape[1]):
                    s += A[i, k] * B[k, j]

                out[i, j] = s
```

Set Up Threads


```
A = np.random.random((800, 200))
B = np.random.random((200, 300))
C = np.zeros((800, 300))

N = len(A)

a = threading.Thread(target=matmul_partitioned,
                     args=(0, N//2, A, B, C))
b = threading.Thread(target=matmul_partitioned,
                     args=(N//2, N, A, B, C))

a.start()
b.start()
a.join()
b.join()
```

Benchmark!

Wrapping C Libraries

External Definitions

Create a file, `trig.pyx`, with the following content:

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

    double M_PI

def test_trig():
    print 'Some trig functions from C:', \
          cos(0), cos(M_PI)
```

Build: Link Math Library


```
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("trig" ,
                  ["trig.pyx"],
                  libraries=["m"] ,
        ),
])
```

Exercises