Cython Tutorial

Dag Sverre Seljebotn
dagss@student.matnat.uio.no

University of Oslo

EuroSciPy 2010

Introduction

About the tutorial

Form:

- Start exercise-driven
- As things get more time-consuming, move to tutorial-on-screen

To do exercises yourself should have Cython 0.11.2 or later set up (http://cython.org).

Tutorial web home: `http://github.com/dagss/euroscipy2010`

Where to learn more

Most convenient source of information is probably two papers from the proceedings of SciPy 2009:

- http://conference.scipy.org/proceedings/SciPy2009/paper_1/
- http://conference.scipy.org/proceedings/SciPy2009/paper_2/

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - supports a big subset of the Python language
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - supports a big subset of the Python language
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)
- In addition:
  - Add types for speedups (hundreds of times)
  - Easily use native libraries (C/C++/Fortran) directly

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - supports a big subset of the Python language
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)
- In addition:
  - Add types for speedups (hundreds of times)
  - Easily use native libraries (C/C++/Fortran) directly
- How it works: Cython code is turned into C code which uses the CPython API and runtime.

Coding in Cython is like coding in Python and C at the same time!

Usecase 1: Library wrapping

- Cython is a popular choice for writing Python interface modules for C libraries
- Works very well for writing a higher-level Pythonized wrapper
- For 1:1 wrapping other tools might be better suited, depending on the exact usecase

Usecase 2: Performance-critical code

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

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

Usecase 2: Performance-critical code

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

- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python
- Cython: Incremental optimization workflow
  - Optimize, don't re-write
  - Only the pieces you need

Not a usecase: Static type checking

- Cython is (partially) statically typed because it has too, not because it wants to
- You still need to run the program to catch a typo

(Demo)

Building Cython code

How Cython works

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

1. A `.pyx` file is compiled by Cython to a `.c` file, containing the code of a Python extension module

How Cython works

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

1. A `.pyx` file is compiled by Cython to a `.c` file, containing the code of a Python extension module
2. The `.c` file is compiled by a C compiler
   - Generated C code can be built without Cython installed; Cython is a developer dependency, not a build-time dependency
   - Generated C code works with Python 2.3 through Python 3.1 (!)

The result is a `.so` file (or `.pyd` on Windows) which can be `import`-ed directly into a Python session.

How to build Cython modules

Ways of building Cython code:

- Sage (`http://sagemath.org`) allows Cython code inline in the notebook (as shown in demo).

How to build Cython modules

Ways of building Cython code:

- Sage (`http://sagemath.org`) allows Cython code inline in the notebook (as shown in demo).
- `pyximport` allows importing Cython `.pyx` files as if they were `.py` files; building on the fly.
  - Things get complicated if you must link to native libraries
  - Larger projects tend to need a build phase anyway

How to build Cython modules

Ways of building Cython code:

- Sage (`http://sagemath.org`) allows Cython code inline in the notebook (as shown in demo).
- `pyximport` allows importing Cython `.pyx` files as if they were `.py` files; building on the fly.
  - Things get complicated if you must link to native libraries
  - Larger projects tend to need a build phase anyway
- Write a distutils `setup.py`. This is what we will use today.
  - Not a real build tool

How to build Cython modules

Ways of building Cython code:

- Sage (`http://sagemath.org`) allows Cython code inline in the notebook (as shown in demo).
- `pyximport` allows importing Cython `.pyx` files as if they were `.py` files; building on the fly.
  - Things get complicated if you must link to native libraries
  - Larger projects tend to need a build phase anyway
- Write a distutils `setup.py`. This is what we will use today.
  - Not a real build tool
- Run `cython` command-line utility and compile the resulting C file, possibly using favourite build tool (make, scons, waf, ...)
  - For cross-system operation you need to query Python for the C build options to use

Exercise 1: Building a Cython module

1. Write some Python code in `ex1.pyx`
2. A typical `setup.py` (get it from `http://github.com/dagss/euroscipy2010` if you can):

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_modules = [Extension("ex1", ["ex1.pyx"])]
setup(
  name = 'Demos',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

3. To build: `python setup.py build_ext --inplace`
   - `python setup.py install` installs
4. Then simply `import ex1`
5. Finally have a look at the generated C code

Types

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef float z
    z = x / 3
    cdef int i, y = two * x
    print z**2
    return z
```

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef float z
    z = x / 3
    cdef int i, y = two * x
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Don't apply prematurely!

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef float z
    z = x / 3
    cdef int i, y = two * x
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Don't apply prematurely!
- All C types are available (and some Python builtins)

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef float z
    z = x / 3
    cdef int i, y = two * x
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Don't apply prematurely!
- All C types are available (and some Python builtins)
- Conversion to and from Python objects happen automatically.

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef float z
    z = x / 3
    cdef int i, y = two * x
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Don't apply prematurely!
- All C types are available (and some Python builtins)
- Conversion to and from Python objects happen automatically.
- `float` and `int` here refers to the C types, not the Python `float` and `int`

Faster code: cdef functions

Python function calls can be expensive – in Cython doubly so because one might need to convert to and from Python objects to do the call.

Therefore Cython provides a syntax for declaring a Cython-only function:

```
cdef double f(double x): return x**3 + 2*x**2
```

The downside is that the function is not available from Python-space. Using `cpdef` makes a fast version available to Cython and a slower one to Python:

```
cpdef double f(double x): return x**3 + 2*x**2
```

Raising exceptions from `cdef` functions

For speed reasons, some manual exception declarations must be done on `cdef` functions capable of raising exceptions.

You can always add "`except *`" and not worry about this though.

```
cdef int is_monday() except -2:
    # Must never return -2 explicitly!
    return time.localtime().tm_wday == 0

cdef int divide(int a, int b) except? 45345:
    # If result is 45345, make an additional check
    return a // b # possible ZeroDivisionError

cdef int divide(int a, int b) except *:
    # I'd rather not bother, just ask Python
    return a // b
```

Exercise 2: Adding types

Add types etc. to speed up the following code (name the file `integrate.pyx` for future reference):

```
from __future__ import division
def f(x): return 1/(x**3 + 2*x**2)
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
```

- Use the `double` type for floating point and `Py_ssize_t` or `int` for integers.
- Expect about 100x speedup. It is easy to miss typing a variable; experiment and see what happens to speed then.
- Use the `-a` switch to the `cython` command-line tool; the generated view of the code explains the speed differences

Calling C functions

Calling C functions

What about this?

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

One *can* do "`from math import sin`" to get Python's `sin` function. Calling C's `sin` function is faster though:

```
cdef extern from "math.h":
    double sin(double)
cdef double f(double x): return sin(x*x)
```

Note that one must "redeclare" the function from `math.h` for the benefit of Cython. In the C compilation stage, only the declaration in `math.h` is seen.

Calling C functions

When calling C functions, one must take care to link in the appropriate libraries. New `setup.py`:

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

ext_modules=[
    Extension("demo",
              ["demo.pyx"],
              libraries=["m"]) # Unix-like specific
]

setup(
  name = "Demos",
  cmdclass = {"build_ext": build_ext},
  ext_modules = ext_modules
)
```

Exercise 3: Call a C function

Just make sure you now know how to calculate $\int_a^b \sin(x^2)\,\mathrm{d}x$.
Check the speed difference between calling Python's sine and C's sine.

Demo: GSL's gamma function

NumPy and Cython

NumPy and Cython

- Cython provides fast access to NumPy arrays
- As libraries start to support Python 3 protocols, any object with array-like data can be accessed in the same way
- 1000x speedup over pure Python loops in extreme cases

NumPy and Cython

(Contrived) example: Compute $\sqrt{x_i^2 + y_i^2 + z_i^2}$.

```
import numpy as np # run-time scope
cimport numpy as np # compile-time scope
cdef extern from "math.h":
    cdef double sqrt(double)
def f(np.ndarray[double] x,
      np.ndarray[double] y,
      np.ndarray[double] z):
    cdef np.ndarray[double] out = np.zeros_like(x)
    if not (x.shape[0] == y.shape[0] == z.shape[0]):
        raise ValueError("Invalid input")
    cdef Py_ssize_t i
    for i in range(x.shape[0]):
        out[i] = sqrt(x[i]**2 + y[i]**2 + z[i]**2)
    return out
```

4 times speed increase over NumPy (ignoring memory allocation of output)

The NumPy C headers need to be present in the include path. For maximum portability, use get_include:

setup.py

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

setup(
  name = 'Matrix␣multiplication',
  cmdclass = {'build_ext': build_ext},
  ext_modules = [
    Extension("matmul_cy",
              ["matmul_cy.pyx"],
              include_dirs=[np.get_include()]),
  ])
```

Limitations

- Only useful when the number of dimensions and the datatypes of the arrays are known up front.

Limitations

- Only useful when the number of dimensions and the datatypes of the arrays are known up front.
- Only single item indexing will be faster (slicing runs at Python speed).

Limitations

- Only useful when the number of dimensions and the datatypes of the arrays are known up front.
- Only single item indexing will be faster (slicing runs at Python speed).
- All indices must be typed

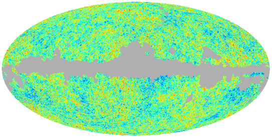
Limitations

- Only useful when the number of dimensions and the datatypes of the arrays are known up front.
- Only single item indexing will be faster (slicing runs at Python speed).
- All indices must be typed
- **NOTE:** If a typed array variable is indexed while it has the value `None`, very bad things will happen (demo)

Example: ODEs

ODE solving – Code first, then speed up

Consider Einstein-Boltzmann equations for energy content in the universe (simplified model, 1st order). In our case, 11 real variables. (Demo)



Perturbations to the Cosmic

Microwave Background

(http://lambda.gsfc.nasa.gov)

$$
\begin{aligned}
\Theta_0' &= -\frac{ck}{\mathcal{H}}\Theta_1 - \Phi' \\
\Theta_1' &= \frac{ck}{3\mathcal{H}}\Theta_0 - \frac{2ck}{3\mathcal{H}}\Theta_2 + \frac{ck}{3\mathcal{H}}\Psi + \tau'\left[\Theta_1 + \frac{1}{3}v_b\right], \\
\Theta_\ell' &= \frac{\ell ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell-1} - \frac{(\ell+1)ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell+1} + \tau'\left[\Theta_\ell - \frac{1}{10}\Theta_\ell\delta_{\ell,2}\right], \quad \ell \geq 2 \\
\delta' &= \frac{ck}{\mathcal{H}}v - 3\Phi', \qquad v' = -v - \frac{ck}{\mathcal{H}}\Psi \\
\delta_b' &= \frac{ck}{\mathcal{H}}v_b - 3\Phi', \qquad v_b' = -v_b - \frac{ck}{\mathcal{H}}\Psi + \tau' R(3\Theta_1 + v_b) \\
\Phi' &= \Psi - \frac{c^2k^2}{3\mathcal{H}^2}\Phi + \frac{H_0^2}{2\mathcal{H}^2}\left[\Omega_m a^{-1}\delta + \Omega_b a^{-1}\delta_b + 4\Omega_r a^{-2}\Theta_0\right] \\
\Psi &= -\Phi - \frac{12H_0^2}{c^2k^2a^2}\Omega_r\Theta_2 \\
R &= \frac{4\Omega_r}{3\Omega_b a}
\end{aligned}
$$

ODE – Code first, then speed up

Results in code such as:

```
def f(self, x, y):
    dy = self.dy; k = self.k; lmax = self.lmax
    (...)
    ddtau = get_ddtau(x) # looks up a spline
    q = ( (-((1-2*R)*dtau+(1+R)*ddtau)*(3*Theta1+v_b)
                - (c*k/H_p)*Psi
                + (1-(H_p/H))*(c*k/H_p)*(-Theta0+2*Theta2)
                - (c*k/H_p)*dTheta0)
              / ((1+R)*dtau + (H_p/H) - 1))
    (...dozens more blocks such as the above...)
(...)
from scipy.integrate import ode
rhs = EBEquations(k, lmax)
integrator = ode(rhs.f)
for i in range(0, x_grid.shape[0]):
    integrator.integrate(x_grid[i])
    if not integrator.successful():
        raise Exception()
```

Cythonizing

```
def f(self, x, y):
    cdef np.ndarray[double, mode='c'] dy
    cdef double k
    cdef int lmax, l
    cdef double a, H, H_p, dtau, ddtau, R, delta, delta_b, v,
                v_b, Phi, Theta0, Theta1, Theta2, Psi, dPhi,
                dTheta0, dv_b, eta, q
    (...)
    ddtau = get_ddtau_fast(x)
    (...)
```

- Add a lot of types
- Need to call fast, Cython-only versions of a couple of functions (spline evaluations).
- This viral aspect only went so far – computing the splined functions could still be done in pure Python

End result: 38 times speedup (YMMV!)

Optimization

NumPy and Cython

Need something CPU-bound, such as naive matrix multiplication:

matmul_py.py

```
import numpy as np
def matmul_py(A, B, out):
    for i in range(A.shape[0]):
        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
```

1.3 times speedup by compiling with Cython as-is

Cython version:

```
import numpy as np
cimport numpy as np
def matmul2(np.ndarray[double, ndim=2] A,
            np.ndarray[double, ndim=2] B,
            np.ndarray[double, ndim=2] out):
    cdef Py_ssize_t i, j, k
    cdef double s
    if A is None or B is None or out is None:
        raise ValueError(<...>)
    for i in range(A.shape[0]):
        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
```

150 times speedup (in-cache)

Wraparound and bounds checking

```
cimport cython
import numpy as np
cimport numpy as np
ctypedef double dtype_t

@cython.boundscheck(False)
@cython.wraparound(False)
def matmul(np.ndarray[dtype_t, ndim=2] A,
           np.ndarray[dtype_t, ndim=2] B,
           np.ndarray[dtype_t, ndim=2] out=None):
    <...snip...>
```

620 times speedup (in-cache)

Out-of-cache: Array layout matters!

- Accessing arrays in a good order $\Rightarrow$ less jumping around in memory $\Rightarrow$ faster execution in out-of-cache situations.

Out-of-cache: Array layout matters!

- Accessing arrays in a good order ⇒ less jumping around in memory ⇒ faster execution in out-of-cache situations.
- In matmul, we access the rows of `A` and columns of `B`, so (with our naive algorithm) the optimal layout is to have `A` stored with contiguous rows ("C order") and `B` stored with contiguous columns ("Fortran order").

Assuming `X`, `Y` and `out` are C-contiguous (the NumPy default):

	80x80 (50 KB)	600x600 (2.75 MB)
`matmul_cython(X, Y.T, out)`	1.4 ms	1.0 s
`matmul_cython(X, Y, out)`	1.4 ms	1.9 s
`matmul_cython(X.T, Y, out)`	1.4 ms	6.7 s
`matmul_cython(X.T, Y, out.T)`	1.5 ms	6.7 s

More on array memory layout

NumPy arrays are not necesarrily stored as one contiguous block of memory:

```
A = np.reshape(np.arange(18), (3, 6))
```

`A` =

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

	`A`
Start:	0
Shape:	(3, 6)
Strides:	(6, 1)
`A[1,2]` at:	$0 + 1 \cdot 6 + 2 \cdot 1 = 7$

More on array memory layout

NumPy arrays are not necesarrily stored as one contiguous block of memory:

```
A = np.arange(18); A.shape = (3, 6); B = A[0::2, 5:0:-2]
```

$$\mathtt{A} = \begin{array}{|c|c|c|c|c|c|}\hline 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 12 & 13 & 14 & 15 & 16 & 17 \\ \hline \end{array} \quad \Rightarrow \quad \mathtt{B} = \begin{array}{|c|c|c|}\hline 5 & 3 & 1 \\ \hline 17 & 15 & 13 \\ \hline \end{array}$$

	A	B
Start:	0	5
Shape:	(3, 6)	(2, 3)
Strides:	(6, 1)	(12, -2)
Element [1,2] at:	$0 + 1 \cdot 6 + 2 \cdot 1 = 8$	$5 + 1 \cdot 12 + 2 \cdot (-2) = 13$

More on array memory layout

NumPy arrays are not necesarrily stored as one contiguous block of memory:

```
A = np.arange(18); A.shape = (3, 6); B = A[0::2, 5:0:-2]
```

$$\mathtt{A} = \begin{array}{|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 12 & 13 & 14 & 15 & 16 & 17 \\ \hline \end{array} \quad \Rightarrow \quad \mathtt{B} = \begin{array}{|c|c|c|} \hline 5 & 3 & 1 \\ \hline 17 & 15 & 13 \\ \hline \end{array}$$

	A	B
Start:	0	5
Shape:	(3, 6)	(2, 3)
Strides:	(6, 1)	(12, -2)
Element [1,2] at:	$0 + 1 \cdot 6 + \mathbf{2} \cdot \mathbf{1} = 8$	$5 + 1 \cdot 12 + 2 \cdot (-2) = 13$

Array access method

If one knows compile-time that the array is contiguous, one can save one stride multiplication operation per array access.

```
def matmul4(np.ndarray[dtype_t, ndim=2, mode="c"] A,
            np.ndarray[dtype_t, ndim=2, mode="fortran"] B,
            np.ndarray[dtype_t, ndim=2, mode="c"] out=None)
    <...snip...>
```

This assumes that the arguments have the right layout:

```
>>> matmul4(A, B, out)
Traceback (most recent call last):
    ...
ValueError: ndarray is not Fortran contiguous
```

So: Copy the arrays (if needed) before assigning to typed variables.

Result: **780** times speedup (in-cache).

(Of course, NumPy/LAPACK is much faster still, about 1500x)

Remember

These things can have a dramatic impact when the computation is CPU-bound.

They may not matter of all when the computation is data-bound.

Using C libraries

Wrapping GSL's splines

We will simply walk through a simple wrapper around the spline functionality in the GNU Scientific Library. Covered:

- Cython classes ("cdef classes", extension types)
- Wrapping C library code
- More sophisticated use of NumPy arrays
- Passing strings (error messages) from C to Python
- Lots of subtle points – we'll see how far we get

Commented example code is up at http://github.com/dagss/euroscipy2010