Optimizing and interfacing with Cython

Konrad HINSEN

Centre de Biophysique Moléculaire (Orléans)

and

Synchrotron Soleil (St Aubin)

Extension modules

- Python permits modules to be written in C.
  Such modules are called **extension modules**.
- Extension modules can define **extension types**, which are very similar to classes, but more efficient.
- Extension modules are usually compiled to produce *shared libraries* (Unix) or *dynamic-link libraries* (DLL, Windows).
  This causes certain restrictions on the C code in the module.

- To client code, extension modules look just like Python modules.
- Many modules in the standard library are in fact extension modules.
- Many scientific packages (including NumPy) consist of a mix of Python modules and extension modules.

- Writing extension modules in plain C is both difficult and a lot of work. Nowadays most programmers use interface generators (SWIG, f2py, ...) or a special extension-module language: Cython.

Cython

- Compiler that compiles a Python module to a C extension module
  ➥ 5% acceleration at best!

- Language extensions for writing C in Python syntax
  ➥ hybrid Python/C programming

Applications:

- optimizing a Python function by translating it to C incrementally
- writing extension modules more conveniently
- writing interfaces to C libraries

Cython vs. Pyrex

Pyrex: the original compiler, developed by Greg Ewing as a research project.

Cython: a fork of the Pyrex source code, made by the Sage development team because they needed to add features at a faster pace than Greg was willing to handle.

Present state:

- Pyrex is a slowly-evolving small and stable compiler written in Python.
- Cython is a much larger and more rapidly evolving compiler that includes compiled modules itself.
- The two projects exchange ideas and source code.
- Cython has some features (optimization, array support) that make it the better choice for numerical code.

Example: Python

```
def exp(x, terms = 50):
    sum = 0.
    power = 1.
    fact = 1.
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

Note: This is not the best algorithm for calculating an exponential function!

Example: Cython

Automatic conversion Python->C

```
def exp(double x, int terms = 50):
    cdef double sum
    cdef double power
    cdef double fact
    cdef int i
```

Declaration of C variables

```
    sum = 0.
    power = 1.
    fact = 1.
```

Conversion to integer loop

```
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

Loop in C

Automatic conversion C->Python

Performance

50 000 exponential calculations on my laptop:

Python: 1.05 s

Cython: 0.042 s

math.exp: 0.013 s

math.exp uses a better algorithm than our Cython function!

Compiling Cython modules

Use distutils as for C extension modules, with some modifications:

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

setup (name = "Exponential",                    name of the package
       version = "0.1",                         package version number

       ext_modules = [Extension('exp_cython',   name of the module
                                ['exp_cython.pyx'])],source code files

       cmdclass = {'build_ext': build_ext}
       )
```

Compile using: `python setup.py build_ext --inplace`

Pure Python mode

```
import cython

@cython.locals(x=cython.double, terms=cython.int,
               sum=cython.double, power=cython.double,
               factorial=cython.double, i=cython.int)
def exp(x, terms = 50):
    sum = 0.
    power = 1.
    fact = 1.
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

Python functions vs C functions

Python function:

```
def exp(double x, int terms = 50):
    cdef double sum
    cdef double power
    cdef double fact
    cdef int i
    sum = 0.
    power = 1.
    fact = 1.
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

- Callable from Python code
- Python objects as arguments,
  automatic conversion to C values
- Return value converted to Python object

C function:

```
cdef double exp(double x, int terms = 50):
    cdef double sum
    cdef double power
    cdef double fact
    cdef int i
    sum = 0.
    power = 1.
    fact = 1.
    for i in range(terms):
        sum += power/fact
        power *= x
        fact *= i+1
    return sum
```

- Pure C function in Python syntax
- Callable only from a Cython module
- No data conversion whatsoever

Python -> Cython

1) Write a Python module and test it.
2) Use a profiler to find the time-intensive sections.
3) Change name from module.py to module.pyx. Write setup.py for compilation. Compile and test again.
4) In the critical sections, convert the loop indices to C integers (cdef int ...).
5) Convert all variables used in the critical sections by C variables.

IMPORTANT C DATA TYPES

Data type	C name	Typical size
Integer	int	32 or 64 bits
Long integer	long	64 bits
Byte	char	8 bits
SP Real	float	32-bit IEEE
DP Real	double	64-bit IEEE

Note: all data type sizes are compiler-dependent!

Relelasing the GIL

```
def exp(double x, int terms = 50):
    cdef double sum
    cdef double power
    cdef double fact
    cdef int i
    with nogil:
        sum = 0.
        power = 1.
        fact = 1.
        for i in range(terms):
            sum += power/fact
            power *= x
            fact *= i+1
    return sum
```

NumPy arrays in Cython

```
cimport numpy
import numpy

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

Verification of Python data type

Variable declarations in C

Loop in C

Automatic Conversion C->Python

Compiling with NumPy

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

include_dirs = numpy.distutils.misc_util.get_numpy_include_dirs()

setup (name = "ArraySum",
       version = "0.1",

ext_modules = [Extension('array_sum',
                         ['array_sum.pyx'],
                         include_dirs=include_dirs)],

cmdclass = {'build_ext': build_ext}
)
```

locate the NumPy header files

Interfacing to C code

GSL definitions:

```
cdef extern from "gsl/gsl_sf_bessel.h":

    ctypedef struct gsl_sf_result:
        double val
        double err

    int gsl_sf_bessel_I0_e(double x,
                           gsl_sf_result *result)

cdef extern from "gsl/gsl_errno.h":

    ctypedef void gsl_error_handler_t
    int GSL_SUCCESS
    int GSL_EUNDRFLW
    char *gsl_strerror(int gsl_errno)
    gsl_error_handler_t* gsl_set_error_handler_off()

gsl_set_error_handler_off()
```

Bessel function I0:

```
def I0(double x):
    cdef gsl_sf_result result
    cdef int status
    status = gsl_sf_bessel_I0_e(x, &result)
    if status == GSL_SUCCESS \
          or status == GSL_EUNDRFLW:
        return result.val
    raise ValueError(gsl_strerror(status))
```

Extension types

Python class:

```
class Counter:

    def __init__(self, value=0):
        self.value = value

    def increment(self):
        self.value += 1

    def getValue(self):
        return self.value
```

Extension type:

```
cdef class Counter:

    cdef int value

    def __init__(self, int value=0):
        self.value = value

    def increment(self):
        self.value += 1

    def getValue(self):
        return self.value
```

Main differences:
- the extension type stores its internal state in C variables
- the extension type can have C methods defined with cdef

C methods

```
cdef class Counter:

    cdef int value

    def __init__(self, int value=0):
        self.value = value

    cdef void inc(self):
        self.value += 1

    def increment(self):
        self.inc()

    def getValue(self):
        return self.value
```

```
cdef class CounterBy10(Counter):

    cdef void inc(self):
        self.value += 10
```

C methods have the same inheritance rules as Python methods. That's OO programming at the speed of C and the convenience of Python!

Restriction: object creation is always handled at the Python level, with the resulting memory and performance overhead.

Exercises

Optimisation 1: système solaire

Dans le simulateur du système solaire, la fonction responsable pour la majeure partie du temps CPU est calc_forces. Convertissez-la en Cython.

0) Prenez la version NumPy du simulateur comme point de départ.
1) Transférez la fonction calc_forces dans un nouveau module et importez-la dans le script.
2) Transposez l'algorithme de la version non-NumPy (avec les deux boucles explicites) dans cette fonction à optimiser.
3) Passez du Python au Cython en rajoutant les déclarations des types C.

Après chaque modification, vérifiez que le programme fonctionne encore correctement !

Optimisation 2: plaque chauffée

Dans le simulateur de la plaque chauffée, la méthode responsable pour la majeure partie du temps CPU est laplacien. Remplacez-la par une version qui appelle une fonction Cython.

0) Prenez la version NumPy du simulateur comme point de départ.
1) Créez un nouveau module qui contient une seule fonction “laplacien” et modifiez la méthode d’origine pour l’appeller.
2) Transposez l’algorithme de la version non-NumPy (avec les boucles explicites) dans cette fonction à optimiser.
3) Passez du Python au Cython en rajoutant les déclarations des types C.

Après chaque modification, vérifiez que le programme fonctionne encore correctement !

Interfacing

The files ndtr.c, polevl.c, and mconf.h from the Cephes library (http://www.netlib.org/cephes/) contain the C functions erf() and erfc() plus routines used in them.

Provide a Python interface to erf and erfc. Write a Python script that verifies that erf(x)+erfc(x)=1 for several x.

Note: if you want to use the symbols erf and erfc for your Python functions, you will have to rename the C functions. This is done as follows:

```
cdef extern from "mconf.h":
    double c_erf "erf" (double x)
```