# 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

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

Declaration of C variables

Conversion to integer loop

Loop in C

Automatic conversion C->Python

# 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`

# 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

# 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!

# 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))
```

# Exercice: 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)
```