Slides from INF3331 lectures – combining Python with Fortran/C/C++

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

October 2012

Mixed language programming

Contents

More info

- Ch. 5 in the course book
- F2PY manual
- SWIG manual
- Examples coming with the SWIG source code
- Ch. 9 and 10 in the course book

Optimizing slow Python code

- Identify bottlenecks (via profiling)
- Migrate slow functions to Fortran, C, or C++
- Tools make it easy to combine Python with Fortran, C, or C++

Getting started: Scientific Hello World

- (Python-F77 via F2PY)
- Python-C via SWIG
- Python-C++ via SWIG

The nature of Python vs. C

- A Python variable can hold different objects:

```
d = 3.2     # d holds a float
d = 'txt'   # d holds a string
d = Button(frame, text='push')  # instance of class Button
```

- In C, C++ and Fortran, a variable is declared of a specific type:

```
double d; d = 4.2;
d = "some string";  /* illegal, compiler error */
```

- This difference makes it quite complicated to call C, C++ or Fortran from Python

Calling C from Python

- Suppose we have a C function

```
extern double hw1(double r1, double r2);
```

- We want to call this from Python as

```
from hw import hw1
r1 = 1.2; r2 = -1.2
s = hw1(r1, r2)
```

- The Python variables `r1` and `r2` hold numbers (`float`), we need to extract these in the C code, convert to `double` variables, then call `hw1`, and finally convert the `double` result to a Python `float`
- All this conversion is done in *wrapper code*

Wrapper code

- Every object in Python is represented by C struct `PyObject`
- Wrapper code converts between `PyObject` variables and plain C variables (from `PyObject r1` and `r2` to `double`, and `double result` to `PyObject`):

```
static PyObject *_wrap_hw1(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    double arg1, arg2, result;

    PyArg_ParseTuple(args,(char *)"dd:hw1",&arg1,&arg2)

    result = hw1(arg1,arg2);

    resultobj = PyFloat_FromDouble(result);
    return resultobj;
}
```

Extension modules

- The wrapper function and `hw1` must be compiled and linked to a shared library file
- This file can be loaded in Python as module
- Such modules written in other languages are called *extension modules*

Integration issues

- Direct calls through wrapper code enables efficient data transfer; large arrays can be sent by pointers
- COM, CORBA, ILU, .NET are different technologies; more complex, less efficient, but safer (data are copied)
- Jython provides a seamless integration of Python and Java.
- Cython is a rapidly developing tool for integrating C and Python.
- The module `ctypes` provides C compatible data types in Python, and enables calling functions in shared libraries.

Writing wrapper code

- A wrapper function is needed for each C function we want to call from Python
- Wrapper codes are tedious to write
- There are tools for automating wrapper code development
- We shall use SWIG (for C/C++) and F2PY (for Fortran)

Scientific Hello World example

- Consider this Scientific Hello World module (`hw`):

  ```
  import math

  def hw1(r1, r2):
      s = math.sin(r1 + r2)
      return s

  def hw2(r1, r2):
      s = math.sin(r1 + r2)
      print 'Hello, World! sin(%g+%g)=%g' % (r1,r2,s)
  ```

  Usage:

  ```
  from hw import hw1, hw2
  print hw1(1.0, 0)
  hw2(1.0, 0)
  ```

- We want to implement the module in C and C++, and use it as if it were a pure Python module

Using SWIG to wrap C and C++

- Wrappers to C and C++ codes can be automatically generated by SWIG
- SWIG is more complicated to use than F2PY
- First make a SWIG interface file
- Then run SWIG to generate wrapper code
- Then compile and link the C code and the wrapper code

SWIG interface file

- The interface file contains C preprocessor directives and special SWIG directives:

```
/* file: hw.i */
%module hw
%{
/* include C header files necessary to compile the interface */
#include "hw.h"
%}

/* list functions to be interfaced: */
double hw1(double r1, double r2);
void   hw2(double r1, double r2);
void   hw3(double r1, double r2, double *s);
// or
// %include "hw.h"  /* make interface to all funcs in hw.h */
```

Making the module

- Run SWIG (preferably in a subdirectory):

```
swig -python -I.. hw.i
```

- SWIG generates wrapper code in

```
hw_wrap.c
```

- Compile and link a shared library module:

```
gcc -I.. -fPIC -I/some/path/include/python2.5 \
    -c ../hw.c hw_wrap.c
gcc -shared -fPIC -o _hw.so hw.o hw_wrap.o
```

  Note the underscore prefix in `_hw.so`

A build script

- Can automate the compile+link process
- Can use Python to extract where `Python.h` resides (needed by any wrapper code)

  ```
  swig -python -I.. hw.i

  root=`python -c 'import sys; print sys.prefix'`
  ver=`python -c 'import sys; print sys.version[:3]'`
  gcc -fPIC -I.. -I$root/include/python$ver -c ../hw.c hw_wrap.c
  gcc -shared -fPIC -o _hw.so hw.o hw_wrap.o

  python -c "import hw" # test
  ```

  this script `make_module_1.sh` is found here:

  `http://www.ifi.uio.no/~inf3331/scripting/src/py/mixed/hw/C/swig-hw/`
- The module consists of two files: `hw.py` (which loads) `_hw.so`

Building modules with Distutils (1)

- Python has a tool, Distutils, for compiling and linking extension modules
- First write a script `setup.py`:

```
import os
from distutils.core import setup, Extension

name = 'hw'                  # name of the module
version = 1.0                # the module's version number

swig_cmd = 'swig -python -I.. %s.i' % name
print 'running SWIG:', swig_cmd
os.system(swig_cmd)

sources = ['../hw.c', 'hw_wrap.c']

setup(name = name, version = version,
      ext_modules = [Extension('_' + name,  # SWIG requires _
                               sources,
                               include_dirs=[os.pardir])
                    ])
```

Building modules with Distutils (2)

- Now run

```
python setup.py build_ext
python setup.py install --install-platlib=.
python -c 'import hw'  # test
```

- Can install resulting module files in any directory
- Use Distutils for professional distribution!

Testing the hw3 function

- Recall `hw3`:

```
void hw3(double r1, double r2, double *s)
{
  *s = sin(r1 + r2);
}
```

- Test:

```
>>> from hw import hw3
>>> r1 = 1;  r2 = -1;  s = 10
>>> hw3(r1, r2, s)
>>> print s
10   # should be 0 (sin(1-1)=0)
```

Major problem - as in the Fortran case

Specifying input/output arguments

- We need to adjust the SWIG interface file:

```
/* typemaps.i allows input and output pointer arguments to be
   specified using the names INPUT, OUTPUT, or INOUT */
%include "typemaps.i"

void   hw3(double r1, double r2, double *OUTPUT);
```

- Now the usage from Python is

```
s = hw3(r1, r2)
```

- Unfortunately, SWIG does not document this in doc strings

Other tools

- SIP: tool for wrapping C++ libraries
- Boost.Python: tool for wrapping C++ libraries
- CXX: C++ interface to Python (Boost is a replacement)
- Instant, Weave: simple tools for inlining C and C++ code in Python scripts
- Note: SWIG can generate interfaces to most scripting languages (Perl, Ruby, Tcl, Java, Guile, Mzscheme, ...)

Integrating Python with C++

- SWIG supports C++
- The only difference is when we run SWIG (`-c++` option):

```
swig -python -c++ -I.. hw.i
# generates wrapper code in hw_wrap.cxx
```

- Use a C++ compiler to compile and link:

```
root=`python -c 'import sys; print sys.prefix'`
ver=`python -c 'import sys; print sys.version[:3]'`
g++ -fPIC -I.. -I$root/include/python$ver \
    -c ../hw.cpp hw_wrap.cxx
g++ -shared -fPIC -o _hw.so hw.o hw_wrap.o
```

Interfacing C++ functions (1)

- This is like interfacing C functions, except that pointers are usual replaced by references

```
void hw3(double r1, double r2, double *s)  // C style
{ *s = sin(r1 + r2); }
```

```
void hw4(double r1, double r2, double& s)  // C++ style
{ s = sin(r1 + r2); }
```

Interfacing C++ functions (2)

- Interface file (`hw.i`):

```
%module hw
%{
#include "hw.h"
%}
%include "typemaps.i"
%apply double *OUTPUT { double* s }
%apply double *OUTPUT { double& s }
%include "hw.h"
```

- That’s it!

Interfacing C++ classes

- C++ classes add more to the SWIG-C story
- Consider a class version of our Hello World module:

```
class HelloWorld
{
 protected:
  double r1, r2, s;
  void compute();    // compute s=sin(r1+r2)
 public:
  HelloWorld();
  ~HelloWorld();

  void set(double r1, double r2);
  double get() const { return s; }
  void message(std::ostream& out) const;
};
```

- Goal: use this class as a Python class

Function bodies and usage

- Function bodies:

```
void HelloWorld:: set(double r1_, double r2_)
{
  r1 = r1_;  r2 = r2_;
  compute();  // compute s
}
void HelloWorld:: compute()
{ s = sin(r1 + r2); }
```

  etc.

- Usage:

```
HelloWorld hw;
hw.set(r1, r2);
hw.message(std::cout);  // write "Hello, World!" message
```

- Files: `HelloWorld.h`, `HelloWorld.cpp`

Adding a subclass

- To illustrate how to handle class hierarchies, we add a subclass:

```
class HelloWorld2 : public HelloWorld
{
 public:
  void gets(double& s_) const;
};
```

```
void HelloWorld2:: gets(double& s_) const { s_ = s; }
```

  i.e., we have a function with an output argument
- Note: `gets` should return the value when called from Python
- Files: `HelloWorld2.h`, `HelloWorld2.cpp`

SWIG interface file

```
/* file: hw.i */
%module hw
%{
/* include C++ header files necessary to compile the interface */
#include "HelloWorld.h"
#include "HelloWorld2.h"
%}

%include "HelloWorld.h"

%include "typemaps.i"
%apply double* OUTPUT { double& s }
%include "HelloWorld2.h"
```

Adding a class method

- SWIG allows us to add class methods
- Calling `message` with standard output (`std::cout`) is tricky from Python so we add a `print` method for printing to std.output
- `print` coincides with Python's keyword `print` so we follow the convention of adding an underscore:

```
%extend HelloWorld {
    void print_() { self->message(std::cout); }
}
```

- This is basically C++ syntax, but `self` is used instead of `this` and `%extend HelloWorld` is a SWIG directive
- Make extension module:

```
swig -python -c++ -I.. hw.i
# compile HelloWorld.cpp HelloWorld2.cpp hw_wrap.cxx
# link HelloWorld.o HelloWorld2.o hw_wrap.o to _hw.so
```

Using the module

```
from hw import HelloWorld

hw = HelloWorld()   # make class instance
r1 = float(sys.argv[1]);   r2 = float(sys.argv[2])
hw.set(r1, r2)      # call instance method
s = hw.get()
print "Hello, World! sin(%g + %g)=%g" % (r1, r2, s)
hw.print_()

hw2 = HelloWorld2()   # make subclass instance
hw2.set(r1, r2)
s = hw.gets()         # original output arg. is now return value
print "Hello, World2! sin(%g + %g)=%g" % (r1, r2, s)
```

Remark

- It looks that the C++ class hierarchy is mirrored in Python
- Actually, SWIG wraps a *function* interface to any class:

```
import _hw   # use _hw.so directly
hw = _hw.new_HelloWorld()
_hw.HelloWorld_set(hw, r1, r2)
```

- SWIG also makes a proxy class in `hw.py`, mirroring the original C++ class:

```
import hw    # use hw.py interface to _hw.so
c = hw.HelloWorld()
c.set(r1, r2)   # calls _hw.HelloWorld_set(r1, r2)
```

- The proxy class introduces overhead

Computational steering

- Consider a simulator written in F77, C or C++
- Aim: write the administering code and run-time visualization in Python
- Use a Python interface to Gnuplot
- Use NumPy arrays in Python
- F77/C and NumPy arrays share the same data
- Result:
  - steer simulations through scripts
  - do low-level numerics efficiently in C/F77
  - send simulation data to plotting a program

  The best of all worlds?

Mixed language numerical Python

Contents

More info

- Ch. 5, 9 and 10 in the course book
- F2PY manual
- SWIG manual
- Examples coming with the SWIG source code
- Electronic Python documentation:
  Extending and Embedding..., Python/C API
- Python in a Nutshell
- Python Essential Reference (Beazley)

Is Python slow for numerical computing?

- Fill a NumPy array with function values:

```
n = 2000
a = zeros((n,n))
xcoor = arange(0,1,1/float(n))
ycoor = arange(0,1,1/float(n))

for i in range(n):
    for j in range(n):
        a[i,j] = f(xcoor[i], ycoor[j])  # f(x,y) = sin(x*y) + 8*x
```

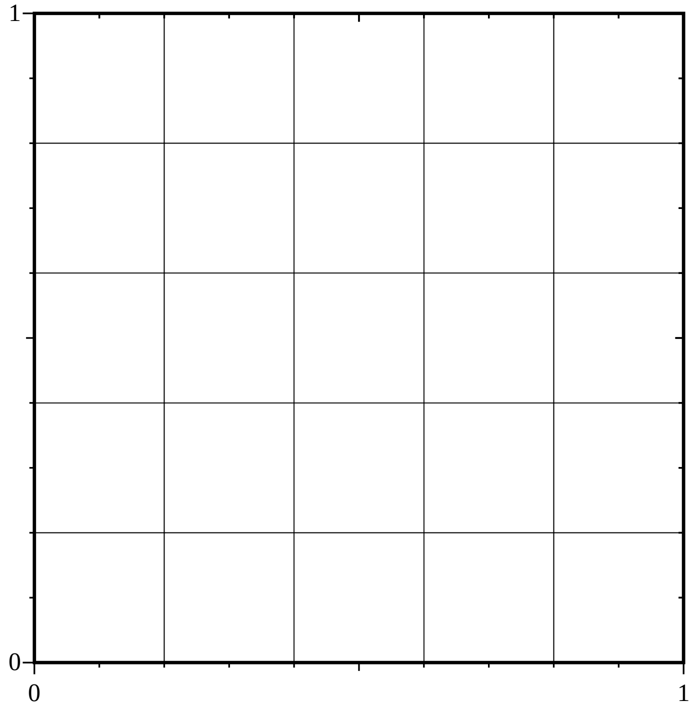
- Fortran/C/C++ version: (normalized) time 1.0
- NumPy vectorized evaluation of `f`: time 3.0
- Python loop version (version): time 140 (`math.sin`)
- Python loop version (version): time 350 (`numpy.sin`)

Comments

- Python loops over arrays are extremely slow
- NumPy vectorization may be sufficient
- However, NumPy vectorization may be inconvenient
  - plain loops in Fortran/C/C++ are much easier
- Write administering code in Python
- Identify bottlenecks (via profiling)
- Migrate slow Python code to Fortran, C, or C++
- Python-Fortran w/NumPy arrays via F2PY: easy
- (Python-C/C++ w/NumPy arrays via SWIG: not that easy)
- Inlining C/C++ code with Instant or Weave: easy

Case: filling a grid with point values

- Consider a rectangular 2D grid



- A NumPy array `a[i,j]` holds values at the grid points

Python object for grid data

- Python class:

```
class Grid2D:
    def __init__(self,
                 xmin=0, xmax=1, dx=0.5,
                 ymin=0, ymax=1, dy=0.5):
        self.xcoor = sequence(xmin, xmax, dx)
        self.ycoor = sequence(ymin, ymax, dy)

        # make two-dim. versions of these arrays:
        # (needed for vectorization in __call__)
        self.xcoorv = self.xcoor[:,newaxis]
        self.ycoorv = self.ycoor[newaxis,:]

    def __call__(self, f):
        # vectorized code:
        return f(self.xcoorv, self.ycoorv)
```

Slow loop

- Include a straight Python loop also:

```
class Grid2D:
    ....
    def gridloop(self, f):
        lx = size(self.xcoor); ly = size(self.ycoor)
        a = zeros((lx,ly))

        for i in xrange(lx):
            x = self.xcoor[i]
            for j in xrange(ly):
                y = self.ycoor[j]
                a[i,j] = f(x, y)
        return a
```

- Usage:

```
g = Grid2D(dx=0.01, dy=0.2)
def myfunc(x, y):
    return sin(x*y) + y
a = g(myfunc)
i=4; j=10;
print 'value at (%g,%g) is %g' % (g.xcoor[i],g.ycoor[j],a[i,j])
```

gridloop1 with C++ array object

- Programming with NumPy arrays in C is much less convenient than programming with C++ array objects

  ```
  SomeArrayClass a(10, 21);
  a(1,2) = 3;       // indexing
  ```

- Idea: wrap NumPy arrays in a C++ class
- Goal: use this class wrapper to simplify the `gridloop1` wrapper

src/py/mixed/Grid2D/C++/plain

The C++ class wrapper (1)

```
class NumPyArray_Float
{
 private:
  PyArrayObject* a;

 public:
  NumPyArray_Float () { a=NULL; }
  NumPyArray_Float (int n1, int n2)  { create(n1, n2); }
  NumPyArray_Float (double* data, int n1, int n2)
    { wrap(data, n1, n2); }
  NumPyArray_Float (PyArrayObject* array) { a = array; }
```

The C++ class wrapper (2)

```
// redimension (reallocate) an array:
int create (int n1, int n2) {
  int dim2[2]; dim2[0] = n1; dim2[1] = n2;
  a = (PyArrayObject*) PyArray_FromDims(2, dim2, PyArray_DOUBLE);
  if (a == NULL) { return 0; } else { return 1; } }

// wrap existing data in a NumPy array:
void wrap (double* data, int n1, int n2) {
  int dim2[2]; dim2[0] = n1; dim2[1] = n2;
  a = (PyArrayObject*) PyArray_FromDimsAndData(\
      2, dim2, PyArray_DOUBLE, (char*) data);
}

// for consistency checks:
int checktype () const;
int checkdim  (int expected_ndim) const;
int checksize (int expected_size1, int expected_size2=0,
               int expected_size3=0) const;
```

The C++ class wrapper (3)

```
  // indexing functions (inline!):
  double  operator() (int i, int j) const
  { return *((double*) (a->data +
                        i*a->strides[0] + j*a->strides[1])); }
  double& operator() (int i, int j)
  { return *((double*) (a->data +
                        i*a->strides[0] + j*a->strides[1])); }

  // extract dimensions:
  int dim() const { return a->nd; }  // no of dimensions
  int size1() const { return a->dimensions[0]; }
  int size2() const { return a->dimensions[1]; }
  int size3() const { return a->dimensions[2]; }
  PyArrayObject* getPtr () { return a; }
};
```

Using the wrapper class

```
static PyObject* gridloop2(PyObject* self, PyObject* args)
{
  PyArrayObject *xcoor_, *ycoor_;
  PyObject *func1, *arglist, *result;
  /* arguments: xcoor, ycoor, func1 */
  if (!PyArg_ParseTuple(args, "O!O!O:gridloop2",
                        &PyArray_Type, &xcoor_,
                        &PyArray_Type, &ycoor_,
                        &func1)) {
    return NULL; /* PyArg_ParseTuple has raised an exception */
  }
  NumPyArray_Float xcoor (xcoor_); int nx = xcoor.size1();
  if (!xcoor.checktype()) { return NULL; }
  if (!xcoor.checkdim(1)) { return NULL; }
  NumPyArray_Float ycoor (ycoor_); int ny = ycoor.size1();
  // check ycoor dimensions, check that func1 is callable...
  NumPyArray_Float a(nx, ny);  // return array
```

The loop is straightforward

```
int i,j;
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    arglist = Py_BuildValue("(dd)", xcoor(i), ycoor(j));
    result = PyEval_CallObject(func1, arglist);
    a(i,j) = PyFloat_AS_DOUBLE(result);
  }
}

return PyArray_Return(a.getPtr());
```

The Instant tool (1)

- Instant allows inlining of C and C++ functions in Python codes
- A quick demo shows its potential

```
class Grid2Deff:
    ...
    def ext_gridloop1_instant(self, fstr):
        if not isinstance(fstr,str):
            raise TypeError, \
            'fstr must be string expression, not %s', type(fstr)

        #generate C source (fstr string must be valid C code)
        source = """
void gridloop1(double *a, int nx, int ny,
                        double *xcoor, double *ycoor)
{
# define index(a,i,j) a{i*ny+j}
   int i, j; double x, y;
   for (i = 0; i <nx; i++) {
      for (j = 0; j <= ny; j++){
         x = xcoor[i]; y = ycoor[i];
         index(a,i,j) = %s
      }
   }
}""" %fstr
```

The Instant tool (2)

```
 try:
    from instant import inline_with_numpy
    a = zeros((self.nx,self.ny))
    arrays = [[’nx’,’ny’,’a’],
            [’nx’,’xcoor’],
            [’ny’,’ycoor’]]
    self.gridloop1_instant = \
           inline_with_numpy(source, arrays=arrays)
except:
   self.gridloop1_instant = None
```

The Instant tool (3)

- `g` is a `Grid2Deff` instance
- We call `g.ext_gridloop_instant(fstr)` to make a C function from `fstr`
- Then we call

  ```
  a = zeros((g.nx,g.ny))
  g.gridloop1_instant(a,g.nx,g.ny,g.xcoor,g.ycoor)
  ```
- Instant detects any changes to the C code (e.g. `fstr`), and automatically recompiles

The Weave tool (1)

- Weave is an easy-to-use tool for inlining C++ snippets in Python codes
- Similar to instant, but with the added flexibility that the C++ code does not need to be a function
- Quick demo example

```
class Grid2Deff:
    ...
    def ext_gridloop1_weave(self, fstr):
        """Migrate loop to C++ with aid of Weave."""

        from scipy import weave

        # the callback function is now coded in C++
        # (fstr must be valid C++ code):

        extra_code = r"""
double cppcb(double x, double y) {
  return %s;
}
""" % fstr
```

The Weave tool (2)

- The loops: inline C++ with Blitz++ array syntax:

```
        code = r"""
int i,j;
for (i=0; i<nx; i++) {
  for (j=0; j<ny; j++) {
    a(i,j) = cppcb(xcoor(i), ycoor(j));
  }
}
"""
```

The Weave tool (3)

- Compile and link the extra code `extra_code` and the main code (loop) `code`:

```
nx = size(self.xcoor);  ny = size(self.ycoor)
a = zeros((nx,ny))
xcoor = self.xcoor;  ycoor = self.ycoor
err = weave.inline(code, ['a', 'nx', 'ny', 'xcoor', 'ycoor'],
         type_converters=weave.converters.blitz,
         support_code=extra_code, compiler='gcc')
return a
```

- Note that we pass the names of the Python objects we want to access in the C++ code
- Weave only recompiles the code if it has changed since last compilation

Summary

We have implemented several versions of `gridloop1` and `gridloop2`:

- Fortran subroutines, working on Fortran arrays, automatically wrapped by F2PY
- Hand-written C++ wrapper, working on a C++ class wrapper for NumPy arrays
- Instant and Weave for inlining C and C++ code

Comparison

- What is the most convenient approach in this case?
  Instant or Weave for inlining. Fortran if we want to interface external code.
- C++ is far more attracting for wrapping NumPy arrays than C, with classes allowing higher-level programming