Mixed language programming
Contents

- Why Python and C are two different worlds
- Wrapper code
- Wrapper tools
- F2PY: wrapping Fortran (and C) code
- SWIG: wrapping C and C++ code
- Alternative tools; ctypes, Instant, Cython
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

(Maybe later: Python interface to oscillator code for interactive computational steering of simulations (using F2PY).)
The nature of Python vs. C

A Python variable can hold different objects:

```python
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:

```c
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

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

We want to call this from Python as

```python
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`):

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

```python
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:

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

We want to implement the module in Fortran 77, C and C++, and use it as if it were a pure Python module.
Fortran 77 implementation

We start with Fortran (F77)

F77 code in a file hw.f:

```fortran
real*8 function hw1(r1, r2)
real*8 r1, r2
hw1 = sin(r1 + r2)
return
end

subroutine hw2(r1, r2)
real*8 r1, r2, s
s = sin(r1 + r2)
write(*,1000) 'Hello, World! sin(',r1+r2,')=',s
1000 format(A,F6.3,A,F8.6)
return
end
```
One-slide F77 course

- Fortran is case insensitive (*reAL is as good as real*)
- One statement per line, must start in column 7 or later
- Comments on separate lines
- All function arguments are input and output
  (as pointers in C, or references in C++)
- A function returning one value is called *function*
- A function returning no value is called *subroutine*
- **Types:** *real, double precision, real*×4, *real*×8, *integer, character* (*array*)
- **Arrays:** just add dimension, as in
  *real*×8 *a*(*0:m, 0:n*)
- Format control of output requires *FORMAT* statements
Using F2PY

- F2PY automates integration of Python and Fortran
- Say the F77 code is in the file hw.f
- Run F2PY (-m module name, -c for compile+link):
  ```
  f2py -m hw -c hw.f
  ```
- Load module into Python and test:
  ```
  from hw import hw1, hw2
  print hw1(1.0, 0)
  hw2(1.0, 0)
  ```
- In Python, hw appears as a module with Python code...
- It cannot be simpler!
Call by reference issues

- In Fortran (and C/C++) functions often modify arguments; here the result \( s \) is an output argument:
  
  ```fortran
  subroutine hw3(r1, r2, s)
  real*8 r1, r2, s
  s = sin(r1 + r2)
  return
  end
  ```

- Running F2PY results in a module with wrong behavior:
  
  ```python
  >>> from hw import hw3
  >>> r1 = 1; r2 = -1; s = 10
  >>> hw3(r1, r2, s)
  >>> print s
  10  # should be 0
  ```

- Why? F2PY assumes that all arguments are input arguments.
- Output arguments must be explicitly specified!
General adjustment of interfaces to Fortran

Function with multiple input and output variables

subroutine somef(i1, i2, o1, o2, o3, o4, io1)

input: i1, i2
output: o1, ..., o4
input and output: io1

Pythonic interface, as generated by F2PY:

o1, o2, o3, o4, io1 = somef(i1, i2, io1)
Check F2PY-generated doc strings

What happened to our hw3 subroutine?

F2PY generates doc strings that document the interface:

```python
g>>> import hw
g>>> print hw.__doc__  # brief module doc string
Functions:
   hw1 = hw1(r1,r2)
   hw2(r1,r2)
   hw3(r1,r2,s)

g>>> print hw.hw3.__doc__  # more detailed function doc string
hw3 - Function signature:
   hw3(r1,r2,s)
Required arguments:
   r1 : input float
   r2 : input float
   s : input float
```

We see that hw3 assumes s is input argument!

Remedy: adjust the interface
Interface files

- We can tailor the interface by editing an F2PY-generated *interface file*
- Run F2PY in two steps: (i) generate interface file, (ii) generate wrapper code, compile and link
- Generate interface file `hw.pyf` (*-h option):
  ```
f2py -m hw -h hw.pyf hw.f
  ```
Outline of the interface file

- The interface applies a Fortran 90 module (class) syntax
- Each function/subroutine, its arguments and its return value is specified:

```python
python module hw ! in
    interface ! in :hw
        ...
        subroutine hw3(r1,r2,s) ! in :hw:hw.f
            real*8 :: r1
            real*8 :: r2
            real*8 :: s
        end subroutine hw3
    end interface
end python module hw
```

(Fortran 90 syntax)
Adjustment of the interface

We may edit `hw.pyf` and specify `s` in `hw3` as an output argument, using F90’s `intent(out)` keyword:

```python
python module hw ! in
  interface ! in :hw
    ...
    subroutine hw3(r1,r2,s) ! in :hw:hw.f
      real*8 :: r1
      real*8 :: r2
      real*8, intent(out) :: s
    end subroutine hw3
  end interface
end python module hw
```

Next step: run F2PY with the edited interface file:

```bash
f2py -c hw.pyf hw.f
```
Output arguments are always returned

- Load the module and print its doc string:

  ```python
  >>> import hw
  >>> print hw.__doc__
  Functions:
   hw1 = hw1(r1,r2)
   hw2(r1,r2)
   s = hw3(r1,r2)
  ```

  **Oops!** `hw3` takes only two arguments and *returns* `s`!

- This is the “Pythonic” function style; input data are arguments, output data are returned

- By default, F2PY treats all arguments as input

- F2PY generates Pythonic interfaces, different from the original Fortran interfaces, so check out the module’s doc string!
General adjustment of interfaces

- Function with multiple input and output variables
  
  ```
  subroutine somef(i1, i2, o1, o2, o3, o4, io1)
  
  input: i1, i2
  output: o1, ..., o4
  input and output: io1
  
  Pythonic interface (as generated by F2PY):
  
  o1, o2, o3, o4, io1 = somef(i1, i2, io1)
  ```
Specification of input/output arguments; .pyf file

In the interface file:

```python
python module somemodule
    interface
        ...
        subroutine somef(i1, i2, o1, o2, o3, o4, io1)
            real*8, intent(in) :: i1
            real*8, intent(in) :: i2
            real*8, intent(out) :: o1
            real*8, intent(out) :: o2
            real*8, intent(out) :: o3
            real*8, intent(out) :: o4
            real*8, intent(in,out) :: io1
        end subroutine somef
    end interface
end python module somemodule
```

Note: no intent implies intent (in)
Specification of input/output arguments; .f file

Instead of editing the interface file, we can add special F2PY comments in the Fortran source code:

```fortran
subroutine somef(i1, i2, o1, o2, o3, o4, io1)
  real*8  i1, i2, o1, o2, o3, o4, io1
  Cf2py intent(in)  i1
  Cf2py intent(in)  i2
  Cf2py intent(out) o1
  Cf2py intent(out) o2
  Cf2py intent(out) o3
  Cf2py intent(out) o4
  Cf2py intent(in,out) io1
```

Now a single F2PY command generates correct interface:

```bash
f2py -m hw -c hw.f
```
Specification of input/output arguments; .f90 file

With Fortran 90:

```fortran
subroutine somef(i1, i2, o1, o2, o3, o4, io1)
real*8 i1, i2, o1, o2, o3, o4, io1
!f2py intent(in) i1
!f2py intent(in) i2
!f2py intent(out) o1
!f2py intent(out) o2
!f2py intent(out) o3
!f2py intent(out) o4
!f2py intent(in,out) io1
```

Now a single F2PY command generates correct interface:

```
f2py -m hw -c hw.f
```
Integration of Python and C

Let us implement the \texttt{hw} module in C:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

double hw1(double r1, double r2)
{
    double s;  s = sin(r1 + r2);  return s;
}

void hw2(double r1, double r2)
{
    double s;  s = sin(r1 + r2);
    printf("Hello, World! \sin(%g+%g)=%g\n", r1, r2, s);
}

/* special version of \texttt{hw1} where the result is an argument: */
void hw3(double r1, double r2, double *s)
{
    *s = sin(r1 + r2);
}
```
Using F2PY

- F2PY can also wrap C code if we specify the function signatures as Fortran 90 modules
- My procedure:
  - write the C functions as empty Fortran 77 functions or subroutines
  - run F2PY on the Fortran specification to generate an interface file
  - run F2PY with the interface file and the C source code
Step 1: Write Fortran 77 signatures

C file signatures.f

    real*8 function hw1(r1, r2)
    Cf2py intent(c) hw1
    real*8 r1, r2
    Cf2py intent(c) r1, r2
    end

    subroutine hw2(r1, r2)
    Cf2py intent(c) hw2
    real*8 r1, r2
    Cf2py intent(c) r1, r2
    end

    subroutine hw3(r1, r2, s)
    Cf2py intent(c) hw3
    real*8 r1, r2, s
    Cf2py intent(c) r1, r2
    Cf2py intent(out) s
    end
Step 2: Generate interface file

Run

Unix/DOS> f2py -m hw -h hw.pyf signatures.f

Result: hw.pyf

python module hw ! in
interface ! in :hw
  function hw1(r1,r2) ! in :hw:signatures.f
    intent(c) hw1
    real*8 intent(c) :: r1
    real*8 intent(c) :: r2
    real*8 intent(c) :: hw1
  end function hw1

  subroutine hw3(r1,r2,s) ! in :hw:signatures.f
    intent(c) hw3
    real*8 intent(c) :: r1
    real*8 intent(c) :: r2
    real*8 intent(out) :: s
  end subroutine hw3

end interface
end python module hw
Step 3: compile C code into extension module

- Run
  Unix/DOS> f2py -c hw.pyf hw.c

- Test:
  import hw
  print hw.hw3(1.0,-1.0)
  print hw.__doc__

- One can either write the interface file by hand or write F77 code to generate, but for every C function the Fortran signature must be specified
Using SWIG

- 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
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 ../*.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`:

```python
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])
                     ]
```

Mixed language programming – p. 37/99
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:

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

Test:

```python
>>> 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, ...)
SwIG supports C++

The only difference is when we run SWIG (\texttt{-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

```c
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):

```c
%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:

```cpp
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);  
}

e tc.

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:

```cpp
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
  ```
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:

  ```python
  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:

  ```python
  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?
Example on computational steering

Consider the oscillator code. The following interactive features would be nice:

- set parameter values
- run the simulator for a number of steps and visualize
- change a parameter
- option: rewind a number of steps
- continue simulation and visualization
Example on what we can do

Here is an interactive session:

```python
>>> from simviz_f77 import *
>>> A=1; w=4*math.pi  # change parameters
>>> setprm()         # send parameters to oscillator code
>>> run(60)          # run 60 steps and plot solution
>>> w=math.pi        # change frequency
>>> setprm()         # update prms in oscillator code
>>> rewind(30)       # rewind 30 steps
>>> run(120)         # run 120 steps and plot
>>> A=10; setprm()
>>> rewind()         # rewind to t=0
>>> run(400)
```
Principles

- The F77 code performs the numerics.
- Python is used for the interface (setprm, run, rewind, plotting).
- F2PY was used to make an interface to the F77 code (fully automated process).
- Arrays (NumPy) are created in Python and transferred to/from the F77 code.
- Python communicates with both the simulator and the plotting program (“sends pointers around”).
About the F77 code

- Physical and numerical parameters are in a common block.

- `scan2` sets parameters in this common block:

  ```fortran
  subroutine scan2(m_, b_, c_, A_, w_, y0_, tstop_, dt_, func_)
  real * 8 m_, b_, c_, A_, w_, y0_, tstop_, dt_
  character func_*(*)
  can use `scan2` to send parameters from Python to F77
  ```

- `timeloop2` performs `nsteps` time steps:

  ```fortran
  subroutine timeloop2(y, n, maxsteps, step, time, nsteps)
  integer n, step, nsteps, maxsteps
  real * 8 time, y(n,0:maxsteps-1)
  solution available in y
  ```
Creating a Python interface w/F2PY

- **scan2**: trivial (only input arguments)
- **timestep2**: need to be careful with
  - output and input/output arguments
  - multi-dimensional arrays (\(y\))

Note: multi-dimensional arrays are stored differently in Python (i.e. C) and Fortran!
Using timeloop2 from Python

This is how we would like to write the Python code:

```python
maxsteps = 10000; n = 2
y = zeros((n,maxsteps), order='Fortran')
step = 0; time = 0.0

def run(nsteps):
    global step, time, y
    y, step, time = \
        oscillator.timeloop2(y, step, time, nsteps)
    y1 = y[0,0:step+1]
g.plot(Gnuplot.Data(t, y1, with='lines'))
```

Arguments to timeloop2

Subroutine signature:

```fortran
subroutine timeloop2(y, n, maxsteps, step, time, nsteps)
  integer n, step, nsteps, maxsteps
  real*8 time, y(n,0:maxsteps-1)
```

Arguments:

- `y` : solution (all time steps), input and output
- `n` : no of solution components (2 in our example), input
- `maxsteps` : max no of time steps, input
- `step` : no of current time step, input and output
- `time` : current value of time, input and output
- `nsteps` : no of time steps to advance the solution
Interfacing the timeloop2 routine

Use **Cf2py** comments to specify argument type:

```plaintext
Cf2py intent(in,out) step
Cf2py intent(in,out) time
Cf2py intent(in,out) y
Cf2py intent(in) nsteps
```

Run **F2PY:**

```plaintext
f2py -m oscillator -c --build-dir tmp1 --fcompiler='Gnu' \
../timeloop2.f \n$scripting/src/app/oscillator/F77/oscillator.f \nonly: scan2 timeloop2 :
```
Testing the extension module

- Import and print documentation:

```python
>>> import oscillator
>>> print oscillator.__doc__
This module 'oscillator' is auto-generated with f2py

Functions:
  y, step, time = timeloop2(y, step, time, nsteps,
                          n=shape(y, 0), maxsteps=shape(y, 1))
  scan2(m_, b_, c_, a_, w_, y0_, tstop_, dt_, func_)

COMMON blocks:
  /data/ m, b, c, a, w, y0, tstop, dt, func(20)
```

- Note: array dimensions \((n, \text{maxsteps})\) are moved to the end of the argument list and given default values!

- Rule: always print and study the doc string since F2PY perturbs the argument list
More info on the current example

Directory with Python interface to the oscillator code:
src/py/mixed/simviz/f2py/

Files:

- simviz_steering.py : complete script running oscillator from Python by calling F77 routines
- simvizGUI_steering.py : as simviz_steering.py, but with a GUI
- make_module.sh : build extension module
Comparison with Matlab

- The demonstrated functionality can be coded in Matlab
- Why Python + F77?
- We can define our own interface in a much more powerful language (Python) than Matlab
- We can much more easily transfer data to and from our own F77 or C or C++ libraries
- We can use any appropriate visualization tool
- We can call up Matlab if we want
- Python + F77 gives tailored interfaces and maximum flexibility
Mixed language numerical Python
Contents

- Migrating slow for loops over NumPy arrays to Fortran, C and C++
- F2PY handling of arrays
- C++ class for wrapping NumPy arrays
- Alternative tools; instant, Weave
- Efficiency considerations
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:

```python
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 ($\text{math.sin}$)

Python loop version (version): time 350 ($\text{numpy.sin}$)
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:

```python
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)
```

Mixed language numerical Python – p. 70/99
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])
Migrate gridloop to F77

class Grid2Deff(Grid2D):
    def __init__(self,
        xmin=0, xmax=1, dx=0.5,
        ymin=0, ymax=1, dy=0.5):
        Grid2D.__init__(self, xmin, xmax, dx, ymin, ymax, dy)
        def ext_gridloop1(self, f):
            """compute a[i,j] = f(xi,yj) in an external routine."""
            lx = size(self.xcoor); ly = size(self.ycoor)
            a = zeros((lx,ly))
            ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)
            return a

We can also migrate to C and C++ (done later)
F77 function

- First try (typical attempt by a Fortran/C programmer):

```fortran
subroutine gridloop1(a, xcoor, ycoor, nx, ny, func1)
integer nx, ny
real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
real*8 func1
external func1

integer i,j
real*8 x, y
do j = 0, ny-1
  y = ycoor(j)
  do i = 0, nx-1
    x = xcoor(i)
    a(i,j) = func1(x, y)
  end do
end do
return
end
```

- Note: float type in NumPy array must match real*8 or double precision in Fortran! (Otherwise F2PY will take a copy of the array a so the type matches that in the F77 code)
Making the extension module

Run F2PY:

```bash
f2py -m ext_gridloop -c gridloop.f
```

Try it from Python:

```python
import ext_gridloop
ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, myfunc,
                     size(self.xcoor), size(self.ycoor))
```

wrong results; `a` is not modified!

Reason: the `gridloop1` function works on a copy of `a` (because higher-dimensional arrays are stored differently in C/Python and Fortran)
Array storage in Fortran and C/C++

- C and C++ has row-major storage (two-dimensional arrays are stored row by row)
- Fortran has column-major storage (two-dimensional arrays are stored column by column)
- Multi-dimensional arrays: first index has fastest variation in Fortran, last index has fastest variation in C and C++
Example: storing a 2x3 array

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>C storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>Fortran storage</td>
</tr>
</tbody>
</table>

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{pmatrix}
\]
F2PY and multi-dimensional arrays

- F2PY-generated modules treat storage schemes transparently.
- If input array has C storage, a copy is taken, calculated with, and returned as output.
- F2PY needs to know whether arguments are input, output or both.
- To monitor (hidden) array copying, turn on the flag:
  ```
  f2py ... -DF2PY_REPORT_ON_ARRAY_COPY=1
  ```
- In-place operations on NumPy arrays are possible in Fortran, but the default is to work on a copy, that is why our `gridloop1` function does not work.
Always specify input/output data

Insert Cf2py comments to tell that \( a \) is an output variable:

```fortran
subroutine gridloop2(a, xcoor, ycoor, nx, ny, func1)
    integer nx, ny
    real*8 a(0:nx-1,ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
    external func1
    Cf2py intent(out) a
    Cf2py intent(in) xcoor
    Cf2py intent(in) ycoor
    Cf2py depend(nx,ny) a
```

Mixed language numerical Python – p. 78/99
F2PY generates this Python interface:

```python
>>> import ext_gridloop
>>> print ext_gridloop.gridloop2.__doc__

gridloop2 - Function signature:
   a = gridloop2(xcoor, ycoor, func1, [nx, ny, func1_extra_args])
Required arguments:
   xcoor : input rank-1 array('d') with bounds (nx)
   ycoor : input rank-1 array('d') with bounds (ny)
   func1 : call-back function
Optional arguments:
   nx := len(xcoor) input int
   ny := len(ycoor) input int
   func1_extra_args := () input tuple
Return objects:
   a : rank-2 array('d') with bounds (nx,ny)
```

- `nx` and `ny` are optional (!)
Handling of arrays with F2PY

- Output arrays are returned and are not part of the argument list, as seen from Python.
- Need `depend(nx, ny) a` to specify that `a` is to be created with size `nx, ny` in the wrapper.
- Array dimensions are optional arguments (!)
  ```python
class Grid2Def(Grid2D):
    ...
    def ext_gridloop2(self, f):
      a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, f)
      return a
  ```
- The modified interface is well documented in the doc strings generated by F2PY.
Input/output arrays (1)

- What if we really want to send `a` as argument and let F77 modify it?

```python
def ext_gridloop1(self, f):
    lx = size(self.xcoor); ly = size(self.ycoor)
a = zeros((lx,ly))
ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)
return a
```

- This is not Pythonic code, but it can be realized
- 1. the array must have Fortran storage
- 2. the array argument must be `intent(inout)`
  (in general not recommended)
Input/output arrays (2)

F2PY generated modules has a function for checking if an array has column major storage (i.e., Fortran storage):

```python
>>> a = zeros((n,n), order='Fortran')
>>> isfortran(a)
True
>>> a = asarray(a, order='C')  # back to C storage
>>> isfortran(a)
False
```
Input/output arrays (3)

- **Fortran function:**

```fortran
subroutine gridloop1(a, xcoor, ycoor, nx, ny, func1)
    integer nx, ny
    real*8 a(0:nx-1,ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
    C call this function with an array a that has
    C column major storage!
    Cf2py intent(inout) a
    Cf2py intent(in) xcoor
    Cf2py intent(in) ycoor
    Cf2py depend(nx, ny) a
```

- **Python call:**

```python
def ext_gridloop1(self, f):
    lx = size(self.xcoor); ly = size(self.ycoor)
a = asarray(a, order='Fortran')
ext_gridloop1.gridloop1(a, self.xcoor, self.ycoor, f)
return a
```
Storage compatibility requirements

- Only when \( a \) has Fortran (column major) storage, the Fortran function works on \( a \) itself.
- If we provide a plain NumPy array, it has C (row major) storage, and the wrapper sends a copy to the Fortran function and transparently transposes the result.
- Hence, F2PY is very user-friendly, at a cost of some extra memory.
- The array returned from F2PY has Fortran (column major) storage.
F2PY and storage issues

- `intent(out) a` is the right specification; `a` should not be an argument in the Python call.
- F2PY wrappers will work on copies, if needed, and hide problems with different storage scheme in Fortran and C/Python.
- Python call:
  
  ```python
  a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, f)
  ```
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
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;
// 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

```c
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

```c
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

```python
class Grid2D:
    ...
    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
```

© www.simula.no/~hpl
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
  
  ```python
  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

```python
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:

```python
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`:

```python
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