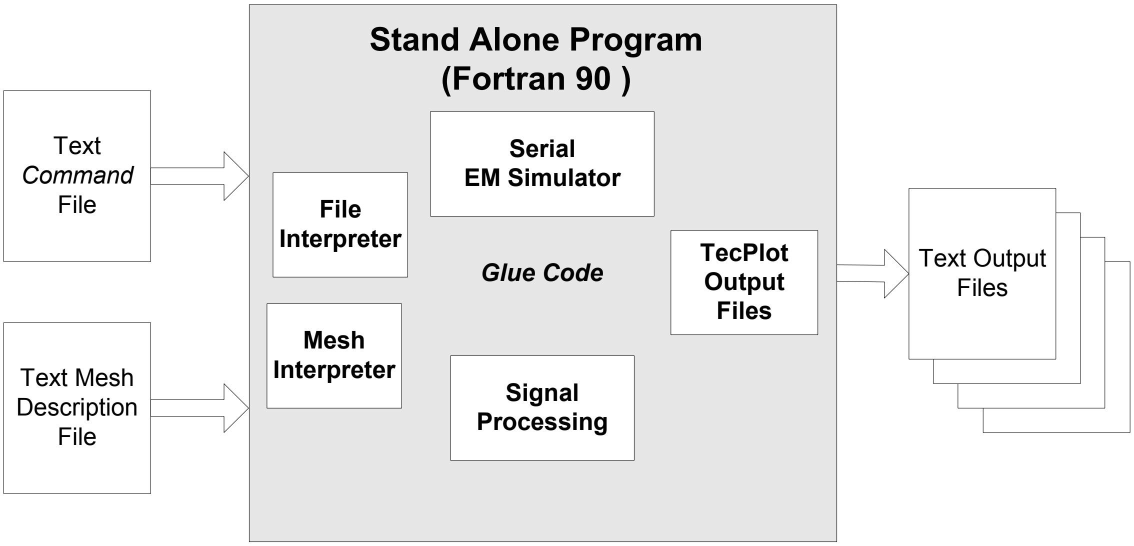
Python as “Glue”

Why?

- Interfacing with Legacy Code
- Speed up “Hot spots” in existing code by moving them to C or Fortran.

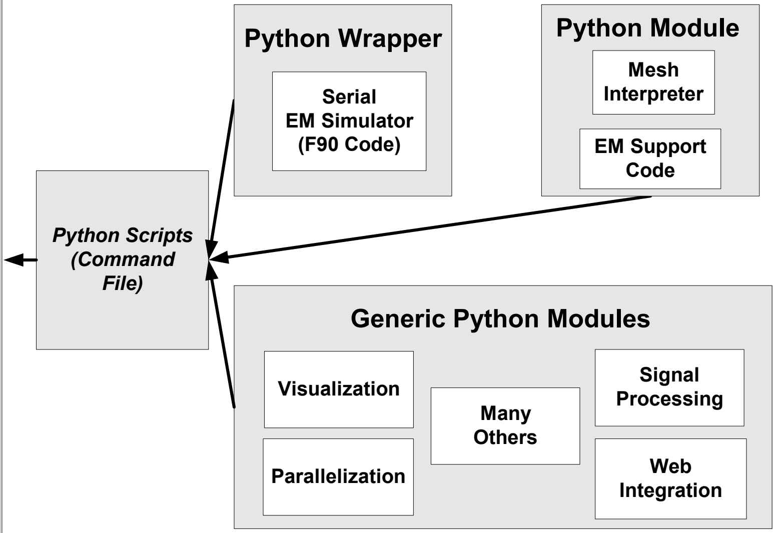
A Case Study for Transitioning from F90 to Python

Legacy System



The Python Approach

```
PythonWin 2.0b2 (#6, Sep 26 2000, 14:59:21) [MSC
Portions Copyright 1994-2000 Mark Hammond (Markl
>>> |
```



Python Scripts (Command File)

Python Wrapper

- Serial EM Simulator (F90 Code)

Python Module

- Mesh Interpreter
- EM Support Code

Generic Python Modules

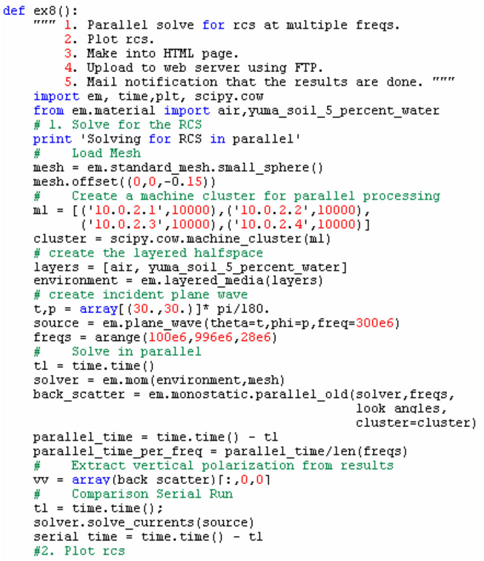
- Visualization
- Parallelization
- Many Others
- Signal Processing
- Web Integration

Simple Input Script

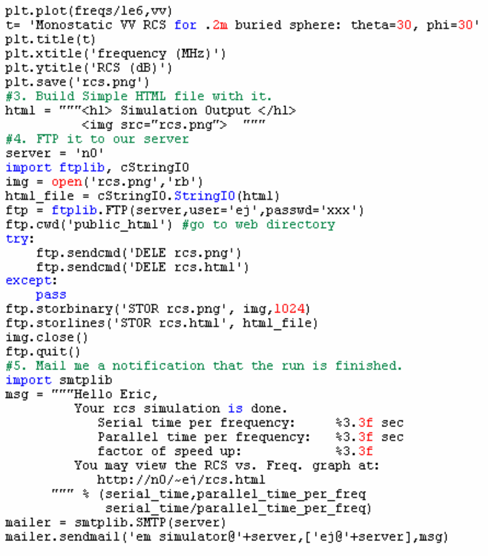
```
import em
from em.material import air,yuma_soil_5_percent_water

# 1. Load standard mesh file
mesh = em.standard_mesh.small_sphere()

# 2. Create layered halfspace from predefined materials.
layers = [air,yuma_soil_5_percent_water]
environment = em.layered_media(layers)

# 3. Build MoM solver based on the mesh and environement.
solver = em.mom(environment,mesh)

# 4. Solve and print results at multiple frequencies.
freqs = arange(100e6,201e6,10e6)
for f in freqs:
    source = em.plane_wave(theta=0,phi=0,freq=f)
    solver.solve_currents(source)
    this_rcs = solver.calc_rcs( [[0.,0.]] )
    print f, this_rcs[0]
```

Electromagnetics Example

(1) Parallel simulation
(2) Create plot
(3) Build HTML page
(4) FTP page to Web Server
(5) E-mail users that results are available.



```
def ex8():
    """ 1. Parallel solve for rcs at multiple freqs.
        2. Plot rcs.
        3. Make into HTML page.
        4. Upload to web server using FTP.
        5. Mail notification that the results are done. """
    import em, time,plt, scipy.cow
    from em.material import air,yuma_soil_5_percent_water
    # 1. Solve for the RCS
    print 'Solving for RCS in parallel'
    #    Load Mesh
    mesh = em.standard_mesh.small_sphere()
    mesh.offset((0,0,-0.15))
    #    Create a machine cluster for parallel processing
    ml = [('10.0.2.1',10000),('10.0.2.2',10000),
          ('10.0.2.3',10000),('10.0.2.4',10000)]
    cluster = scipy.cow.machine_cluster(ml)
    # create the layered halfspace
    layers = [air, yuma_soil_5_percent_water]
    environment = em.layered_media(layers)
    # create incident plane wave
    t,p = array[(30.,30.)]* pi/180.
    source = em.plane_wave(theta=t,phi=p,freq=300e6)
    freqs = arange(100e6,996e6,28e6)
    #    Solve in parallel
    tl = time.time()
    solver = em.mom(environment,mesh)
    back_scatter = em.monostatic.parallel_old(solver,freqs,
                                              look angles,
                                              cluster=cluster)
    parallel_time = time.time() - tl
    parallel_time_per_freq = parallel_time/len(freqs)
    #    Extract vertical polarization from results
    vv = array(back scatter)[:,0,0]
    #    Comparison Serial Run
    tl = time.time();
    solver.solve_currents(source)
    serial time = time.time() - tl
    #2. Plot rcs
```



```
    plt.plot(freqs/1e6,vv)
    t= 'Monostatic VV RCS for .2m buried sphere: theta=30, phi=30'
    plt.title(t)
    plt.xtitle('frequency (MHz)')
    plt.ytitle('RCS (dB)')
    plt.save('rcs.png')
    #3. Build Simple HTML file with it.
    html = """<h1> Simulation Output </h1>
              <img src="rcs.png">  """
    #4. FTP it to our server
    server = 'n0'
    import ftplib, cStringIO
    img = open('rcs.png','rb')
    html_file = cStringIO.StringIO(html)
    ftp = ftplib.FTP(server,user='ej',passwd='xxx')
    ftp.cwd('public_html') #go to web directory
    try:
        ftp.sendcmd('DELE rcs.png')
        ftp.sendcmd('DELE rcs.html')
    except:
        pass
    ftp.storbinary('STOR rcs.png', img,1024)
    ftp.storlines('STOR rcs.html', html_file)
    img.close()
    ftp.quit()
    #5. Mail me a notification that the run is finished.
    import smtplib
    msg = """Hello Eric,
             Your rcs simulation is done.
                Serial time per frequency:     %3.3f sec
                Parallel time per frequency:   %3.3f sec
                factor of speed up:            %3.3f
             You may view the RCS vs. Freq. graph at:
                http://n0/~ej/rcs.html
          """ % (serial_time,parallel_time_per_freq
                 serial_time/parallel_time_per_freq)
    mailer = smtplib.SMTP(server)
    mailer.sendmail('em simulator@'+server,['ej@'+server],msg)
```

General issues with Fortran

- Fortran 77 encouraged the use of a large number of globals.
  - causes *states* within wrapped modules.
  - makes threading difficult
- Legacy code is often not very modular.
  - occasionally one or two very long functions that do absolutely everything.
- Fortran 77 doesn’t handle memory allocation well
- Fortran 90 objects don’t map easily to C structures (the underpinnings of Python)

Global variables – here’s the rub

MODULE WITH GLOBALS CALLED SAFELY…

```
# Global variables create states in modules
>>> import f90_module
# f1() returns an array and also quietly sets a global variable zzz
>>> a = f90_module.f1(5,6) # → zzz = 5
# f2() uses a AS WELL AS zzz
>>> f90_module.f2(a)
xxx # returns some value
```

AND THE HAVOC AN INTERMEDIATE CALL CAN CAUSE

```
# Users of interactive interpreters and scripting languages can and
# do call functions in a variety of orders.  As a result, they can
# unknowingly corrupt the internal state of the module.
>>> a = f90_module.f1(5,6)    # → zzz = 5
# user makes additional call to f1 with different variables
>>> b = f90_module.f1(20,30)  # → zzz = 20
# Now user calls f2 expecting to get same answer as above
>>> f90_module.f2(a)
yyy #Yikes! result is different because globals were different
```

Solutions to global variables

1. Remove the ability of calling functions out of order by wrapping functions at the highest level.

```
# users call the following instead of f1 and f2 individually
def wrapper_for_f1_and_f2(a,b):
    c = f90_module.f1_QUITELY_SETS_LOCAL_VARIABLES(a,b)
    return f90_module.f2_that_uses_a_AND_GLOBAL_VARIABLES(c)
```

2. Get rid of global variables and include them in the argument list for functions. (The best option, but potentially a lot of work)

```
# Return the affected global variables b, c, and d.
>>> a,b,c,d = f90_module.f1(5,6)
# Now call f2 passing in the variables that used to be global
>>> f90_module.f2 (a,b,c,d)
```

Problems with non-modularity

Generally only want to wrap Fortran simulation engine and leave pre and post processing to Python.

If input/engine/output are all closely coupled or live in one huge function (worst case), it is extremely difficult to find convenient places in the code to *hook* into the simulator

---

Solutions to modularity

For codes not written with modularity in mind, this is a lot of work. Try and find logical boundaries in the code and rewrite long functions that do several things as multiple functions that do only one thing.

Dividing the code into smaller functions has other advantages also. It generally leads to more readable and robust code, and it facilitates unit testing. A suite of unit tests can be run on each of the functions after code changes to quickly locate bugs.
See PyUnit at http://pyunit.sourceforge.net/

The starting place for our code

Fortunately, Norbert and Larry had written a very good piece of F90 software that is fairly modular and had only a few global variables – the impedance matrix being the main one.

We chose to expose only three functions:

`tpm_hs_cfie_zcomplete()`

Given a problem description, this function creates the impedance matrix.

`tpm_hs_pwi_cfie_vdrive()`

Given a problem description and a source, calculate the right hand side.

`tpm_hs_ffs_pec_smatrix()`

Given the problem description and currents, calculate the far field in a given direction.

Memory allocation & generic math

- All memory allocation is handled in Python
- Mathematically general operations are left to Python.
  - Linear Algebra is handled by a wrapper to netlib's LAPACK.
  - FFT's are handled by a wrapper to netlib's FFTPACK. FFTW wrappers are also available.

Long argument lists

Fortran encourages long argument lists – unless you use globals which we did not allow. One of our typical functions has *21* arguments. This is unmanageable for an interactive interface.

```
subroutine tpm_hs_pwi_cfie_vdrive (alpha,i12,e0,cfreq,thi,phi,
                                   epstj,muej,halfspace,tnj,
                                   tnjmax,tnn,tnnmax,tnp,
                                   tnpmax, tx3j,tj3p,tn6p,vmth,
                                   vmph,errlog,errtxt)
```

Black arguments affect the CFIE equation.

Red arguments describe the source.

Green arguments describe the environment.

Orange arguments describe the target mesh.

Blue arguments are return values from the function

Solution to long argument lists

- Python wrappers alone remove the need for 9 arguments by removing array indices and return arguments.

```
vmth,vmph,errlog,errtxt = tpm_hs_pwi_cfie_vdrive(alpha,i12,
                              e0,cfreq,thi,phi,epstj,muej,
                              halfspace,tnn,tx3j,tj3p,tn6p)
```

- Using classes to group the other arguments can further simply the interface and facilitate user interaction.

Object oriented view of problem

- Problem Definition
  - Environment
    - free-space/ half-space, material types
  - Target description
    - mesh and perhaps material types
- Source Signal
    - near/far, time/frequency domain
- An algorithm for solving the problem
    - MoM, MLFMA, FDTD, etc

> The words highlighted in **blue** indicate objects that are used as objects in the Python interface to the Duke MoM code.

Using the OO interface

```
# 1. Specify the mesh you would like to use.
>>> import em
>>> mesh = em.standard_mesh.small_sphere()
# 2. Perhaps burry it .15 meters underground.
#    (sphere radius is 0.10 meters)
>>> mesh.offset((0,0,-0.15))
# 3. Specify the layers of materials in the environment.
>>> from em.material import air,yuma_soil_5_percent_water
>>> layers = [air, yuma_soil_5_percent_water]
>>> environment = em.layered_media(layers)
# 4. Build a MoM solver for this mesh and environment.
>>> solver = em.mom(environment,mesh)
# 5. Specify a source frequency and angle (in radians)
#    and solve for the currents on the mesh.
>>> source = em.plane_wave(theta=0,phi=0,freq=50e6)
>>> solver.solve_currents(source)
# 6. Post process.  Here we'll calculate the monostatic
# scattered field.
>>> angles = [[0.,0.]]
>>> rcs = solver.calc_rcs(angles)
```

Review

- Reduce usage of globals as much as possible when ever practical.
- Divide large functions in to smaller pieces to provide more control over simulator.
- Use Python to allocate memory arrays. f2py will handle converting them to Fortran arrays.
- Be careful about using F90 constructs. They are difficult to wrap.
- Object Oriented interfaces hide low level array representations and allow users to work with the actual components of the problem.

Tools

- C/C++ Integration
  - SWIG www.swig.org
  - ctypes python standard library
  - Pyrex nz.cosc.canterbury.ac.nz/~greg/python/Pyrex
  - boost www.boost.org/libs/python/doc/index.html
  - weave www.scipy.org/site_content/weave
- FORTRAN Integration
  - f2py cens.ioc.ee/projects/f2py2e/ (now part of numpy)

Hand Wrapping

The Wrapper Function

fact.h

```
#ifndef FACT_H
#define FACT_H

int fact(int n);

#endif
```

fact.c

```
#include “fact.h”
int fact(int n)
{
   if (n <=1) return 1;
   else return n*fact(n-1);
}
```

WRAPPER FUNCTION

```
static PyObject* wrap_fact(PyObject *self, PyObject *args)
{
    /* Python-C data conversion */
    int n, result;
    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;
    /* C Function Call */
    result = fact(n);
    /* C->Python data conversion */
    return Py_BuildValue("i", result);
}
```

Complete Example

```
/* Must include Python.h before any standard headers! */
#include "Python.h"
#include "fact.h"

/* Define the wrapper functions exposed to Python (must be static) */
static PyObject* wrap_fact(PyObject *self, PyObject *args) {
    int n, result;
    /* Python->C Conversion */
    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;
    /* Call our function */
    result = fact(n);
    /* C->Python Conversion */
    return Py_BuildValue("i", result);
}
/* Method table declaring the names of functions exposed to Python */
static PyMethodDef ExampleMethods[] = {
    {"fact",  wrap_fact, METH_VARARGS, "Calculate the factorial of n"},
    {NULL, NULL, 0, NULL}      /* Sentinel */
};
/* Module initialization function called at "import example" */
PyMODINIT_FUNC initexample(void) {
{ (void) Py_InitModule("example", ExampleMethods);  }
```

Compiling your extension

BUILD BY HAND (PYTHON, ENTHOUGHT EDITION ON WINDOWS)

```
C:\...\demo\hand_wrap> gcc -c -fpic fact.c fact_wrap.c -I. \
                      -Ic:\python25\include
C:\...\demo\hand_wrap> gcc -shared fact.o fact_wrap.o \
                       -Lc:\python25\libs -lpython25 -o example.pyd
```

BUILD BY HAND (UNIX)

```
~\...\demo\hand_wrap> gcc -c -fpic fact.c fact_wrap.c -I. \
                      -I\usr\include\python25\include
~\...\demo\hand_wrap> gcc -shared fact.o fact_wrap.o \
                      -L\usr\lib\python2.5\config -lpython25 \
                      -o example.pyd
```

Note that include and library directories are platform specific. Also, be sure to link against the appropriate version of python (2.5 in these examples).

Using setup.py

SETUP.PY

```
# setup.py files are the Python equivalent of Make
from distutils.core import setup, Extension

ext = Extension(name='example', sources=['fact_wrap.c', 'fact.c'])

setup(name="example", ext_modules=[ext])
```

CALLING SETUP SCRIPTS

```
# Build and install the module
[eric@monster hand_wrap] setup.py build
[eric@monster hand_wrap] setup.py install

# Or
# Build the module “inplace” to ease testing.
[eric@monster hand_wrap] setup.py build_ext --inplace \
                         --compiler=mingw32
* Blue text only needed on windows
```

Weave

weave

- **`weave.blitz()`**

  Translation of Numeric array expressions to C/C++ for fast execution
- **`weave.inline()`**

  Include C/C++ code directly in Python code for on-the-fly execution
- **`weave.ext_tools`**

  Classes for building C/C++ extension modules in Python

---

weave.inline

```
>>> from scipy import weave
>>> a=1
>>> weave.inline(‘return_val = a;',['a'])
<weave: compiling>
creating c:\docume…_12e837eb1ea3ab5199fbcc0e83015e3f
1
>>> weave.inline(‘return_val = a;',['a'])
1
>>> a='qwerty'
>>> weave.inline(‘return_val = a;',['a'])
<weave: compiling>
creating c:\docume…_12e837eb1ea3ab5199fbcc0e83015e3f
qwerty
>>> weave.inline(‘return_val = a;',['a'])
qwerty
```

Support code example

```
>>> from scipy import weave
>>> a = 1
>>> support_code = ‘int bob(int val) { return val;}’
>>> weave.inline(‘return_val = bob(a);’,['a'],support_code=support_code)
<weave: compiling>
creating: c:\...sc_19f0a1876e0022290e9104c0cce4f00c0
1
>>> a = 'string'
>>> weave.inline(‘return_val = bob(a);’,['a'],support_code = support_code)
<weave: compiling>
creating: c:\...sc_19f0a1876e0022290e9104c0cce4f00c1
C:\DOCUME~1\eric\LOCALS~1\Temp\python25_compiled\sc_19f0a1876e0022290e9104c0cce4
f00c1.cpp(417) : error C2664: 'bob' : cannot convert parameter 1 from 'class Py:
:String' to 'int' No user-defined-conversion operator available that can
perform this conversion, or the operator cannot be called
Traceback (most recent call last):
    <snip>
weave.build_tools.CompileError: error: command '"C:\Program Files\Microsoft Visu
al Studio\VC98\BIN\cl.exe"' failed with exit status 2
```

---

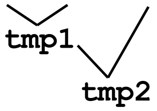
ext_tools example

```
import string
from weave import ext_tools
def build_ex1():
    ext = ext_tools.ext_module('_ex1')
    # Type declarations– define a sequence and a function
    seq = []
    func = string.upper
    code = """
           py::tuple args(1);
           py::list result(seq.length());
           for(int i = 0; i < seq.length();i++)
           {
              args[0] = seq[i];
              result[i] = func.call(args);
              seq[i].call(“foo”,args)
           }
           return_val = result;
           """
    func = ext_tools.ext_function('my_map',code,['func','seq'])
    ext.add_function(func)
    ext.compile()

try:
    from _ex1 import *
except ImportError:
    build_ex1()
    from _ex1 import *

if __name__ == '__main__':
    print my_map(string.lower,['asdf','ADFS','ADSD'])
```

Efficiency Issues

PSEUDO C FOR STANDARD NUMERIC EVALUATION

```
>>> c = a + b + c
```



```
// c code
// tmp1 = a + b
tmp1 = malloc(len_a * el_sz);
for(i=0; i < len_a; i++)
    tmp1[i] = a[i] + b[i];
// tmp2 = tmp1 + c
tmp2 = malloc(len_c * el_sz);
for(i=0; i < len_c; i++)
    tmp2[i] = tmp1[i] + c[i];
```

FAST, IDIOMATIC C CODE

```
>>> c = a + b + c
```

```
// c code
// 1. loops “fused”
// 2. no memory allocation
for(i=0; i < len_a; i++)
    c[i] = a[i] + b[i] + c[i];
```

Finite Difference Equation

MAXWELL’S EQUATIONS: FINITE DIFFERENCE TIME DOMAIN (FDTD), UPDATE OF X COMPONENT OF ELECTRIC FIELD

$$E_x = \frac{1 - \frac{\sigma_x \Delta t}{2\varepsilon_x}}{1 + \frac{\sigma_x \Delta t}{2\varepsilon_x}} E_x + \frac{\Delta t}{\varepsilon_x + \frac{\sigma_x \Delta t}{2}} \frac{dH_z}{dy} - \frac{\Delta t}{\varepsilon_x + \frac{\sigma_x \Delta t}{2}} \frac{dH_y}{dz}$$

PYTHON VERSION OF SAME EQUATION, PRE-CALCULATED CONSTANTS

```
ex[:,1:,1:] =    ca_x[:,1:,1:]   * ex[:,1:,1:]
               + cb_y_x[:,1:,1:] * (hz[:,1:,1:] - hz[:,:-1,1:])
               - cb_z_x[:,1:,1:] * (hy[:,1:,1:] - hy[:,1:,:-1])
```

weave.blitz

weave.blitz compiles array expressions
to C/C++ code using the Blitz++ library.

WEAVE.BLITZ VERSION OF SAME EQUATION

```
>>> from scipy import weave
>>> # <instantiate all array variables...>
>>> expr = “ex[:,1:,1:] =   ca_x[:,1:,1:]   * ex[:,1:,1:]”\
           “+ cb_y_x[:,1:,1:] * (hz[:,1:,1:] - hz[:,:-1,:])”\
           “- cb_z_x[:,1:,1:] * (hy[:,1:,1:] - hy[:,1:,:-1])”

>>> weave.blitz(expr)
 < 1. translate expression to blitz++ expression>
 < 2. compile with gcc using array variables in local scope>
 < 3. load compiled module and execute code>
```

---

weave.blitz benchmarks

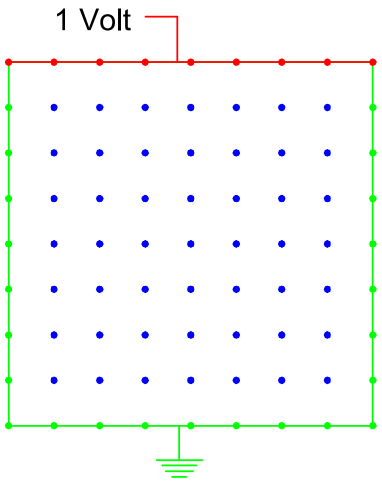
Equation	Numeric (sec)	Inplace (sec)	compiler (sec)	Speed Up
Float (4 bytes)				
a = b + c (512,512)	0.027	0.019	0.024	1.13
a = b + c + d (512x512)	0.060	0.037	0.029	2.06
5 pt. avg filter (512x512)	0.161	-	0.060	2.68
FDTD (100x100x100)	0.890	-	0.323	2.75
Double (8 bytes)				
a = b + c (512,512)	0.128	0.106	0.042	3.05
a = b + c + d (512x512)	0.248	0.210	0.054	4.59
5 pt. avg filter (512x512)	0.631	-	0.070	9.01
FDTD (100x100x100)	3.399	-	0.395	8.61

Pentium II, 300 MHz, Python 2.0, Numeric 17.2.0
Speed-up taken as ratio of scipy.compiler to standard Numeric runs.

weave and Laplace’s equation

Weave case study: An iterative solver for Laplace’s Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$



PURE PYTHON 2000 SECONDS

```
for i in range(1, nx-1):
  for j in range(1, ny-1):
    tmp = u[i,j]
    u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                (u[i, j-1] + u[i, j+1])*dx2)
              / (2.0*(dx2 + dy2))
    diff = u[i,j] – tmp
    err = err + diff**2
```

Thanks to Prabhu Ramachandran for designing and running this example. His complete write-up is available at: http://www.scipy.org/PerformancePython

---

weave and Laplace’s equation

USING NUMERIC 29.0 SECONDS

```
old_u = u.copy() # needed to compute the error.
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                 (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)
                * dnr_inv
err = sum(dot(old_u – u))
```

WEAVE.BLITZ 10.2 SECONDS

```
old_u = u.copy() # needed to compute the error.
expr = ””” \
       u[1:-1, 1:-1] =  ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                         (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)
                       * dnr_inv
       ”””
weave.blitz(expr,size_check=0)
err = sum((old_u – u)**2)
```

weave and Laplace’s equation

WEAVE.INLINE **4.3 SECONDS**

```
code = """
       #line 120 "laplace.py" (This is only useful for debugging)
       double tmp, err, diff;
       err = 0.0;
       for (int i=1; i<nx-1; ++i) {
         for (int j=1; j<ny-1; ++j) {
           tmp = u(i,j);
           u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +
                     (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv;
           diff = u(i,j) - tmp;
           err += diff*diff;
         }
       }
       return_val = sqrt(err);
       """
err = weave.inline(code, ['u','dx2','dy2','dnr_inv','nx','ny'],
                        type_converters = converters.blitz,
                        compiler = 'gcc',
                        extra_compile_args = ['-O3','-malign-double'])
```

Laplace Benchmarks

Method	Run Time (sec)	Speed Up
Pure Python	≈ 133.00	≈ 0.01
Numeric	1.36	1.00
weave.blitz	0.56	2.43
weave.inline	0.28	4.85
weave.inline (fast)	0.25	5.44
Python/Fortran (with f2py)	0.25	5.44
Pyrex	0.25	5.44
Pure C++ Program	0.24	5.66

500x500 grid for 100 iterations
Intel X5355 Xeon Quad Core, 2.66 GHz, 8 GB Memory
Linux, CentOS 5, Python 2.5.1, Numpy 1.0.4

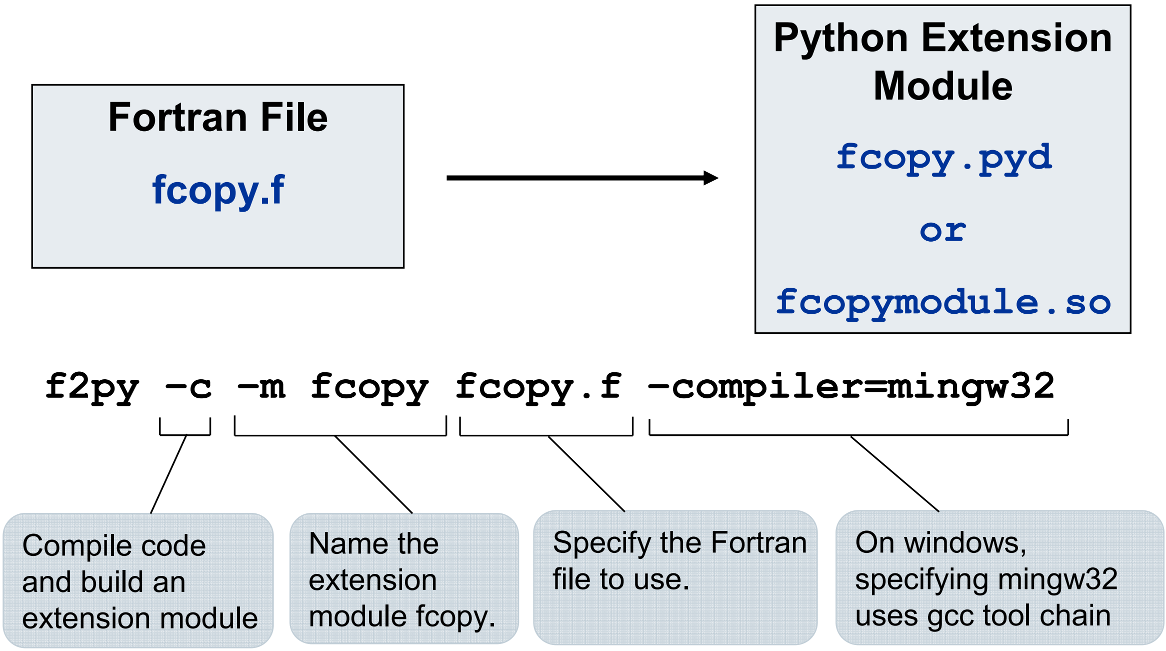
f2py

---

f2py

- Author: **Pearu Peterson** at Center for Nonlinear Studies Tallinn, Estonia
- Automagically “wraps” Fortran 77/90/95 libraries for use in Python. *Amazing.*
- f2py is specifically built to wrap Fortran functions using NumPy arrays.

Simplest f2py Usage



Fortran File

fcopy.f

Python Extension Module

`fcopy.pyd`

or

`fcopymodule.so`

```
f2py -c -m fcopy fcopy.f -compiler=mingw32
```

Compile code and build an extension module

Name the extension module fcopy.

Specify the Fortran file to use.

On windows, specifying mingw32 uses gcc tool chain

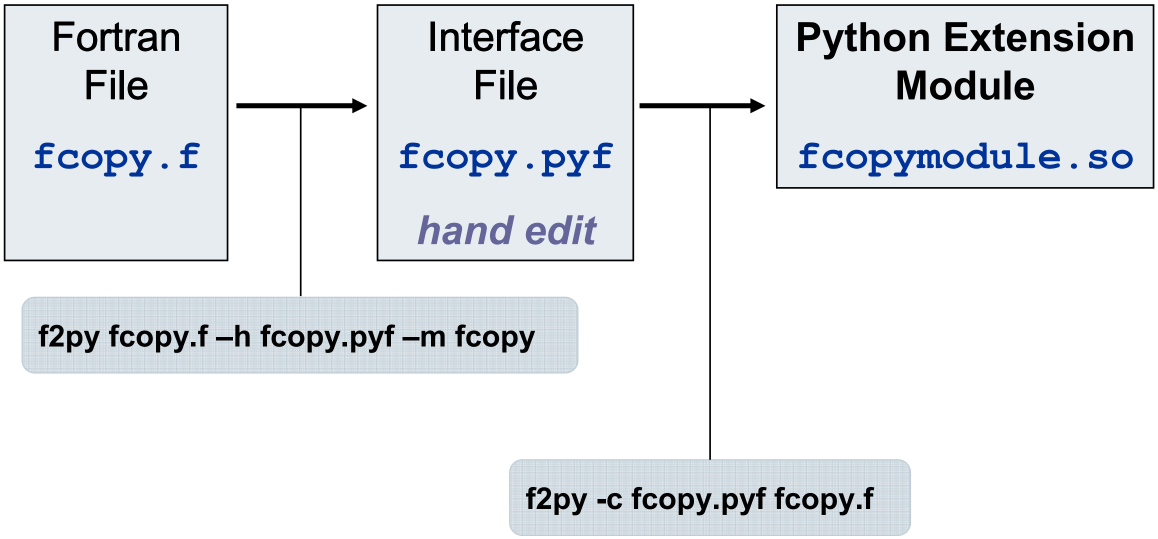
Simplest Usage Result

```
Fortran file fcopy.f
C
    SUBROUTINE FCOPY(AIN,N,AOUT)
C
    DOUBLE COMPLEX AIN(*)
    INTEGER N
    DOUBLE COMPLEX AOUT(*)
    DO 20 J = 1, N
       AOUT(J) = AIN(J)
20  CONTINUE
    END
```

```
>>> a = rand(1000) + 1j*rand(1000)

>>> b = zeros((1000,), dtype=complex128)

>>> fcopy.fcopy(a,1000,b)

>>> alltrue(a==b)
True
```

Looks exactly like the Fortran --- but now it is callable from Python.

More Sophisticated



Fortran File

`fcopy.f`

Interface File

`fcopy.pyf`

hand edit

Python Extension Module

`fcopymodule.so`

```
f2py fcopy.f –h fcopy.pyf –m fcopy
```

```
f2py -c fcopy.pyf fcopy.f
```

More Sophisticated

Interface file fcopy2.pyf

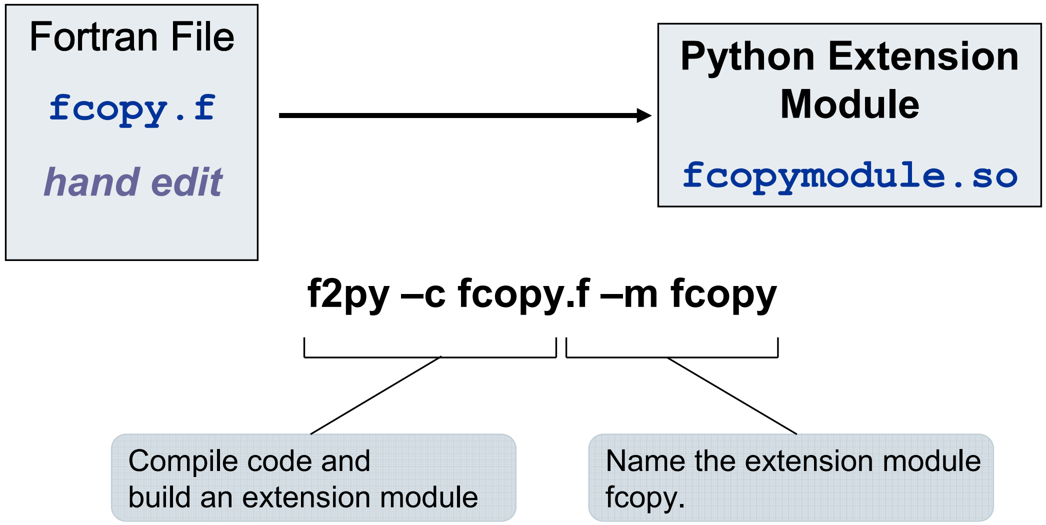
```
!    -*- f90 -*-
python module fcopy2 ! in
    interface  ! in :fcopy
        subroutine fcopy(ain,n,aout) ! in :fcopy:fcopy.f
            double complex dimension(n), intent(in) :: ain
            integer, intent(hide),depend(ain) :: n=len(ain)
            double complex dimension(n),intent(out) :: aout
        end subroutine fcopy
    end interface
end python module fcopy

! This file was auto-generated with f2py (version:2.37.233-1545).
! See http://cens.ioc.ee/projects/f2py2e/
```

Give f2py some hints as to what these variables are used for and how they may be related in Python.

```
fcopy - Function signature:
  aout = fcopy(ain)
Required arguments:
  ain : input rank-1 array('D')
        with bounds (n)
Return objects:
  aout : rank-1 array('D') with
         bounds (n)
```

```
# More pythonic behavior
>>> a = rand(5).astype(complex64)
>>> b = fcopy2.fcopy(a)
>>> alltrue(a==b)
True
# However, b is complex128, not
# 64 because of signature
>>> print b.dtype
dtype(‘complex128')
```

Simply Sophisticated



Fortran File

`fcopy.f`

hand edit

Python Extension Module

`fcopymodule.so`

f2py –c fcopy.f –m fcopy

Compile code and build an extension module

Name the extension module fcopy.

Simply Sophisticated

Directives help f2py interpret the source.

```
Fortran file fcopy2.f
C
      SUBROUTINE FCOPY(AIN,N,AOUT)
C
CF2PY INTENT(IN), AIN
CF2PY INTENT(OUT), AOUT
CF2PY INTENT(HIDE), DEPEND(A), N=LEN(A)
      DOUBLE COMPLEX AIN(*)
      INTEGER N
      DOUBLE COMPLEX AOUT(*)
      DO 20 J = 1, N
         AOUT(J) = AIN(J)
 20   CONTINUE
      END
```

The resulting interface is more *Pythonic*.

```
>>> import fcopy
>>> info(fcopy.fcopy)
fcopy - Function signature:
  aout = fcopy(ain)
Required arguments:
  ain : input rank-1 array('D') with
bounds (n)
Return objects:
  aout : rank-1 array('D') with bounds (n)
>>> a = rand(1000)
>>> import fcopy
>>> b = fcopy.fcopy(a)
```

Multidimensional array issues

Python and Numeric use C conventions for array storage (row major order).
Fortran uses column major ordering.

Numeric:

A[0,0], A[0,1], A[0,2],…, A[N-1,N-2], A[N-1,N-1]
(last dimension varies the fastest)

Fortran:

A(1,1), A(2,1), A(3,1), …, A(N-1,N), A(N,N)
(first dimension varies the fastest)

f2py handles the conversion back and forth between the representations if you mix them in your code. Your code will be faster, however, if you can avoid mixing the representations (impossible if you are calling out to both C and Fortran libraries that are interpreting matrices differently).

---

numpy.distutils

How do I distribute this great new extension module?

- Recipient must have f2py and scipy_distutils installed (both are simple installs)
- Create setup.py file
- Distribute *.f files with setup.py file.
- Optionally distribute *.pyf file if you've spruced up the interface in a separate interface file.

Supported Compilers

g77, Compaq Fortran, VAST/f90 Fortran, Absoft F77/F90, Forte (Sun), SGI, Intel, Itanium, NAG, Lahey, PG

Complete Example

In scipy.stats there is a function written entirely in Python

```
>>> help(stats.morestats._find_repeats)
 _find_repeats(arr)

Find repeats in the array and return a list of the
repeats and how many there were.
```

Goal: Write an equivalent fortran function and link it in to Python with f2py so it can be distributed with scipy_base (which uses scipy_distutils) and be available for stats.

Python algorithm uses sort and so we will need a fortran function for that, too.

The "Decorated" Fortran File

```
Fortran file futil.f
C     Sorts an array arr(1:N) into
      SUBROUTINE DQSORT(N,ARR)
CF2PY INTENT(IN,OUT,COPY), ARR
CF2PY INTENT(HIDE), DEPEND(ARR), N=len(ARR)
      INTEGER N,M,NSTACK
      REAL*8 ARR(N)
      PARAMETER (M=7, NSTACK=100)
      INTEGER I,IR,J,JSTACK, K,L, ISTACK(NSTACK)
      REAL*8 A,TEMP
        …
      END

C     Finds repeated elements of ARR
      SUBROUTINE DFREPS(ARR,N,REPLIST,REPNUM,NLIST)
CF2PY INTENT(IN), ARR
CF2PY INTENT(OUT), REPLIST
CF2PY INTENT(OUT), REPNUM
CF2PY INTENT(OUT), NLIST
CF2PY INTENT(HIDE), DEPEND(ARR), N=len(ARR)
      REAL*8 REPLIST(N), ARR(N)
      REAL*8 LASTVAL
      INTEGER REPNUM(N)
      INTEGER HOWMANY, REPEAT, IND, NLIST, NNUM
         …
      END
```

setup.py

```
from numpy.distutils.core import Extension

# For f2py extensions that have a .pyf file.
futil1 = Extension(name='futil',
                   sources=['futil.pyf','futil.f'])
# fortran files with f2py directives embedded in them
# are processed by the build_src step of setup.py
futil2 = Extension(name='futil2',
                   sources=['futil2.f'])

if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(name=‘repeats_util',
          description = "f2py example",
          ext_modules = [futil1, futil2],
          py_modules = ‘util.py’)
```

Building:

```
C:\demo\f2py_repeats> python setup.py build_src \
                      build_ext --inplace --compiler=mingw32
```

or

```
C:\demo\f2py_repeats> python setup.py build_src \
                      build_ext --compiler=mingw32 build
```

The Python “wrapper” function

```
# util.py

import futil2

def find_repeats(arr):
    """Find repeats in arr and return (repeats, repeat_count)
    """
    v1,v2, n = futil2.dfreps(arr)
    return v1[:n],v2[:n]

if __name__ == "__main__":
    from scipy import stats, float64
    a = stats.randint(1, 30).rvs(size=1000)
    print a.astype(float64)
    repeats, nums = find_repeats(a)
    print 'repeats:'
    print repeats
    print 'nums:’
    print nums
```

Complete Example

Try It Out!!

```
>>> from scipy import stats
>>> from util import find_repeats
>>> a = stats.randint(1,30).rvs(size=1000)
>>> reps, nums = find_repeats(a)
>>> print reps
[  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.
  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.
  23.  24.  25.  26.  27.  28.  29.]
>>> print nums
[29 37 29 30 34 39 46 20 30 32 35 42 40 39 35 26 38 33 40
 29 34 26 38 45 39 38 29 39 29]
```

New function is 25 times faster than the plain Python version

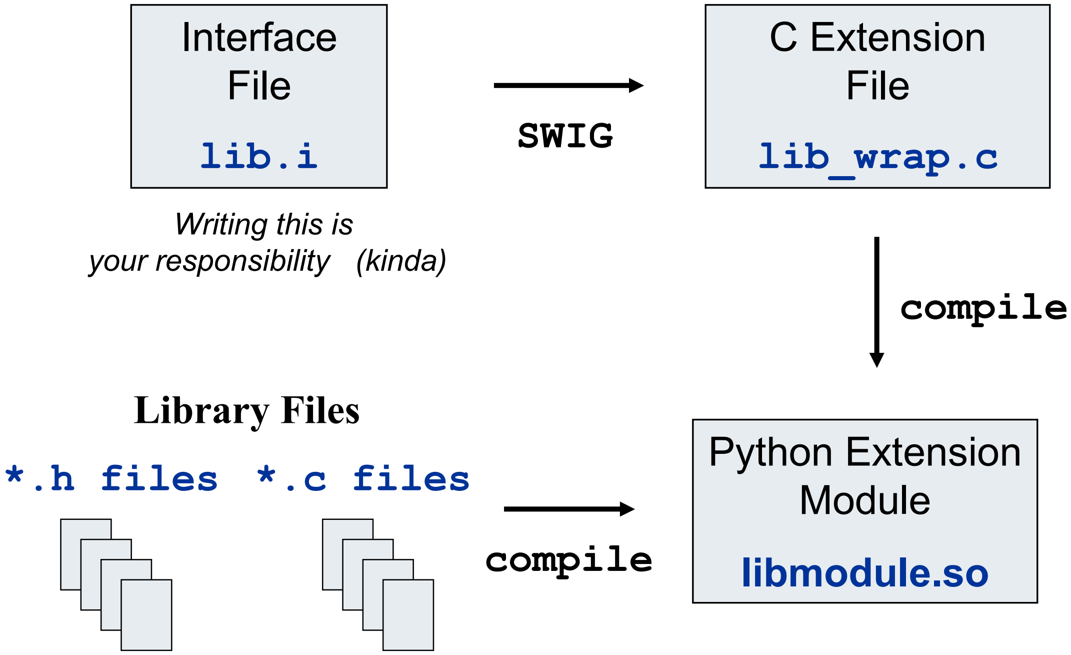
---

SWIG

SWIG

- Author: David Beazley
- Automatically “wraps” C/C++ libraries for use in Python. *Amazing.*
- SWIG uses interface files to describe library functions
  - No need to modify original library code
  - Flexible approach allowing both simple and complex library interfaces
- Well Documented

SWIG Process



Interface File

`lib.i`

Writing this is your responsibility (kinda)

`SWIG`

C Extension File

`lib_wrap.c`

`compile`

Library Files

`*.h files` `*.c files`

`compile`

Python Extension Module

libmodule.so

Simple Example

fact.h

```
#ifndef FACT_H
#define FACT_H

int fact(int n);

#endif
```

fact.c

```
#include “fact.h”
int fact(int n)
{
   if (n <=1) return 1;
   else return n*fact(n-1);
}
```

See demo/swig for this example.
Build it using build.bat

example.i

```
// Define the modules name
%module example

// Specify code that should
// be included at top of
// wrapper file.
%{
   #include “fact.h”
%}

// Define interface. Easy way
// out - Simply include the
// header file and let SWIG
// figure everything out.
%include “fact.h”
```

Command Line Build

LINUX

```
# Create example_wrap.c file
[ej@bull ej]$ swig -python example.i
# Compile library and example_wrap.c code using
# “position independent code” flag
[ej@bull ej]$ gcc -c -fpic example_wrap.c fact.c      \
              -I/usr/local/include/python2.1           \
              -I/usr/local/lib/python2.1/config
# link as a shared library.
[ej@bull ej]$ gcc -shared example_wrap.o fact.o        \
              -o examplemodule.so
# test it in Python
[ej@bull ej]$ python
 ...
 >>> import example
 >>> example.fact(4)
 24
```

For notes on how to use SWIG with VC++ on Windows, see http://www.swig.org/Doc1.1/HTML/Python.html#n2

The Wrapper File

example_wrap.c

```
static PyObject *_wrap_fact(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    int arg0 ;
    int result ;
    /* parse the Python input arguments and extract */
```

name of function to return in case of error

```
    if(!PyArg_ParseTuple(args,"i:fact",&arg0)) return NULL;
```

first arg in args read into arg0 as int

```
    /* call the actual C function with arg0 as the argument*/
    result = (int )fact(arg0);

    /* Convert returned C value to Python type and return it*/
    resultobj = PyInt_FromLong((long)result);
    return resultobj;
}
```

SWIG Example 2

vect.h

```
int* vect(int x,int y,int z);
int sum(int* vector);
```

vect.c

```
#include <malloc.h>
#include “vect.h”
int* vect(int x,int y, int z){
  int* res;
  res = malloc(3*sizeof(int));
  res[0]=x;res[1]=y;res[2]=z;
  return res;
}
int sum(int* v) {
  return v[0]+v[1]+v[2];
}
```

example2.i

Identical to example.i if you replace “fact” with “vect”.

TEST IN PYTHON

```
>>> from example2 import *
>>> a = vect(1,2,3)
>>> sum(a)
6   #works fine!

# Let’s take a look at the
# integer array a.
>>> a
'_813d880_p_int'
# WHAT THE HECK IS THIS???
```

Complex Objects in SWIG

- SWIG treats all complex objects as pointers.
- These C pointers are mangled into string representations for Python’s consumption.
- This is one of SWIG’s secrets to wrapping virtually any library automatically,
- But… the string representation is pretty primitive and makes it “un-pythonic” to observe/manipulate the contents of the object.
- **Enter typemaps**

Typemaps

example_wrap.c

```
static PyObject *_wrap_sum(PyObject *self, PyObject *args) {
  ...
  if(!PyArg_ParseTuple(args,"O:sum",&arg0))
    return NULL;
  ...
  result = (int )sum(arg0);
  ...
  return resultobj;
}
```

Typemaps allow you to insert “type conversion” code into various location within the function wrapper.

Not for the faint of heart. Quoting David:

“You can blow your whole leg off, including your foot!”

Slide 63

TNV2 formatting needs fixing
Travis N. Vaught, 11/11/2004

Typemaps

The result? Standard C pointers are mapped to NumPy arrays for easy manipulation in Python.

YET ANOTHER EXAMPLE – NOW WITH TYPEMAPS

```
>>> import example3
>>> a = example3.vect(1,2,3)
>>> a                    # a should be an array now.
array([1, 2, 3], 'i') # It is!
>>> example3.sum(a)
6
```

The typemaps used for example3 are included in the handouts.

Another example that wraps a more complicated C function used in the previous VQ benchmarks is also provided. It offers more generic handling 1D and 2D arrays.

Pyrex

What is Pyrex?

Pyrex is a Python-like language for writing extension modules. It lets you mix Python and C data types any way you want, and compiles to (reasonably) fast code.

http://nz.cosc.canterbury.ac.nz/~greg/python/Pyrex

A really simple example

PI.PYX

```
# Define a function.  Include type information for the argument.
def multiply_by_pi(int num):
   return num * 3.14159265359
```

SETUP_PI.PY

```
# Pyrex has its own “extension builder” module that knows how
# to build pyrex files into python modules.
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext

ext = Extension("pi", sources=["pi.pyx"])

setup(ext_modules=[ext],
      cmdclass={'build_ext': build_ext})
```

See demo/pyrex for this example.
Build it using build.bat

A simple Pyrex example

CALLING MULTIPLY_BY_PI FROM PYTHON

```
C:\> python setup_pi.py build_ext --inplace -c mingw32

C:\> python
Enthought Edition build 1059
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) ... on win32
Type "help", "copyright", "credits" or "license" for more information

>>> import pi
>>> pi.multiply_by_pi()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)
>>> pi.multiply_by_pi("dsa")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required
>>> pi.multiply_by_pi(3)
9.4247779607700011
```

(some of) the generated code

C CODE GENERATED BY PYREX

```
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds); /*proto*/
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds) {
  int __pyx_v_num;
  PyObject *__pyx_r;
  PyObject *__pyx_1 = 0;
  static char *__pyx_argnames[] = {"num",0};
  if (!PyArg_ParseTupleAndKeywords(__pyx_args, __pyx_kwds, "i",
__pyx_argnames,
&__pyx_v_num)) return 0;

  /* “C:\pi.pyx":2 */
  __pyx_1 = PyFloat_FromDouble((__pyx_v_num * 3.14159265359));
if (!__pyx_1) {__
pyx_filename = __pyx_f[0]; __pyx_lineno = 2; goto __pyx_L1;}
  __pyx_r = __pyx_1;
  __pyx_1 = 0;
```

Def vs. CDef

DEF -- PYTHON FUNCTIONS

```
# Python callable function.
def inc(int num, int offset):
   return num + offset

# Call inc for values in sequence
def inc_seq(seq, offset):
   result = []
   for val in seq:
       res = inc(val, offset)
       result.append(res)
   return result
```

INC FROM PYTHON

```
# inc is callable from Python
>>> inc.inc(1,3)
4
>>> a = range(4)
>>> inc.inc_seq(a, 3)
[3,4,5,6]
```

CDEF -- C FUNCTIONS

```
# cdef becomes a C function call.
cdef double fast_inc(int num,
                     int offset):
    return num + offest
# fast_inc for a sequence.
def fast_inc_seq(seq, offset):
   result = []
   for val in seq:
       res = fast_inc(val, offset)
       result.append(res)
   return result
```

FAST_INC FROM PYTHON

```
# fast_inc not callable in Python
>>> inc.fast_inc(1,3)
Traceback: ... no 'fast_inc`
# But fast_inc_seq is 2x faster
# for large arrays.
>>> inc.fast_inc_seq(a, 3)
[3,4,5,6]
```

Functions from C Libraries

EXTERNAL C FUNCTIONS

```
# extern.pyx
# First, "include" the header file you need.
cdef extern from "string.h":
    # Describe the interface for the functions used.
    cdef extern int strlen(char *c)

def get_len(char *message):
    # strlen can now be used from Pyrex code (but not Python)…
    return strlen(message)
```

CALL FROM PYTHON

```
>>> import extern
>>> extern.strlen
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'strlen'
>>> extern.get_len("woohoo!")
7
```

Structures from C Libraries

TIME_EXTERN.PYX

```
cdef extern from "time.h":
    # Describe the structure you are using
    struct tm:
        ...
        int tm_mday # Day of the month: 1-31
        int tm_mon  # Months *since* january: 0-11
        int tm_year # Years since 1900
        ...
    ctypedef long time_t
    tm* localtime(time_t *timer)
    time_t time(time_t *tloc)

def get_date():
    " Return a tuple with the current day, month, and year."
    cdef time_t t
    cdef tm* ts
    t = time(NULL)
    ts = localtime(&t)
    return ts.tm_mday, ts.tm_mon+1, ts.tm_year % 100
```

CALLING FROM PYTHON

```
>>> extern_time.get_date()
(13, 8, 7)
```

Classes

SHRUBBERY.PYX

```
cdef class Shrubbery:
  # Class level variables.
  cdef int width, height

  def __init__(self, w, h):
    self.width = w
    self.height = h
  def describe(self):
    print "This shrubbery is", self.width, "by", self.height, "cubits."
```

CALLING FROM PYTHON

```
>>> import shrubbery
>>> x = shrubbery.Shrubbery(1, 2)
>>> x.describe()
This shrubbery is 1 by 2 cubits.
>>> print x.width
AttributeError: ‘shrubbery.Shrubbery' object has no attribute 'width
```

Using NumPy with Pyrex

```
# Import the c_numpy pyrex “module” shipped with numpy.
cimport c_numpy
from c_numpy cimport import_array, ndarray, NPY_DOUBLE, PyArray_ContiguousFromAny
# Intialize array module.
import_array()

def sum(object seq):
    # Declare C data type for ary and the raw data with the ary structure.
    cdef ndarray ary
    cdef double *ary_data
    # object, data type, minimum dimension, maximum dimensions
    ary = PyArray_ContiguousFromAny(seq, NPY_DOUBLE, 1, 1)
    # Cast ary.data to a double* pointer.
    ary_data = <double *>ary.data
    # How long is the array in the first dimension?
    n = ary.dimensions[0]
    # Define local variables used in calculations.
    cdef int i
    cdef double sum
    # Sum algorithm implementation.
    sum = 0.0
    for i from 0 <= i < n:
        sum = sum + ary_data[i]
    return sum
```

```
C:\demo\pyrex>test_sum.py
elements: 1,000,000
python sum(approx sec, result): 7.030
numpy sum(sec, result): 0.047
pyrex sum(sec, result): 0.078
```