Python in High-Performance Computing

Martin Sandve Alnæs, Are Magnus Bruaset, Xing Cai, Hans Petter Langtangen, Kent-Andre Mardal, Halvard Moe, Ola Skavhaug, Åsmund Ødegård

[simula . research laboratory]

Topics

Crash Course in Python 4

Scientific Computing with Python 17

Visualization in with Python 35

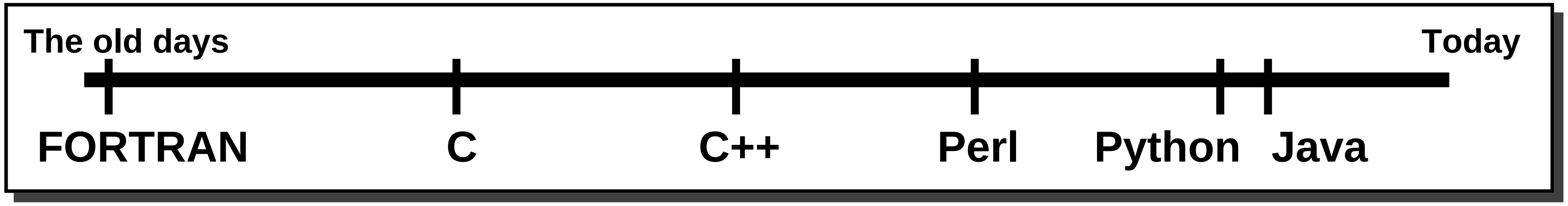
Performance Issues – Tips and Tricks 58

Parallel Computing via Python 64

Extend Your Favorite Library with Python 82

Various Packages for Scientific Computing 121

Why Scientific Computing with Python



- **Languages have moved from speed and efficiency to flexibility and convenience**

- **Python is a powerful yet easy to use language**
- **Python has a rich set of libraries and extensions**
- **Python is an ideal *glue* between your applications**
- **Using wrapping techniques, your legacy code may get another life**

- **Python is suitable for computational steering**
- **In this tutorial we also do number crunching with Python!**

Crash Course in Python

We begin with a crash course in Python.
In this section we cover:

- **Basic variables, containers and control structures**
- **Functions, lambda functions and callable objects**
- **Object oriented features like classes and operator overloading**
- **String editing and file handling**

- **For more information on the Python language, please check out:**
  - **The Slides "Scripting for Computational Science" [35]**
  - **The Book "Python Scripting for Computational Science" [36]**
  - **The Python tutorial on python.org [82]**
  - **The Introductory Material on Python [32]**

Interactive Sessions

- **Interactive sessions:**
  - **Each line start with `>>>`**
  - **Continuing lines start with `...`**
  - **Output appear on lines without prefix**
  - **Run the interactive sessions in either IPython[33] or IDLE.**

```
>>>
>>> def f(x):        # user input
...     return x     # cont. line
>>> f(2)             # user input
2                    # output, result
                     # of f(2)
```

- **Otherwise, code segments are supposed to appear in a program file**

```
# This is code
def f(x):
    return x*x
y = f(2.5)
```

A Python example

```
#!/usr/bin/env python

# load some modules
import sys, math

# read a float cmd. line argument
r = float(sys.argv[1])

# compute the sine
s = math.sin(r)

# printf-like string formatting
print "Hello SC, sin(%f) = %f" % (r,s)
```

A Scientific Hello World, demonstrates how to

- **Load library modules**
- **Read command-line arguments**
- **Load and call a math function**
- **Work with variables**
- **Print text and numbers**

Basic Types

- Python is a dynamically typed language, meaning that the type of every variable is determined runtime
- Everything is an object, even integers, modules and functions
- Python has the following basic variable types:
  - int: 1, 2, 3
  - long (arbitrary length!): 1L, 2L, 3L
  - complex: 1j, 4+5j
  - float (only double precision): 0.1, 0.2, 0.3
  - bool: True or False
  - str: "hello world"
- Casting is done like `intvar = int(stringvar)`

Basic Containers

Python has three built in container types, which can hold objects of any type.

- **Lists:**
  ```
  mylist = [1, 3, 4]; mylist.append(7.8); print mylist[0]
  ```
- **Tuples (immutable/constant lists):**
  ```
  mytuple = (1, 'text', 7.89); print mytuple[0:3]
  ```
- **Dictionaries (hash maps or associative arrays):**
  ```
  mymap = {'pi':3.14, 'mylist':mylist}; print mymap['pi']
  ```

In addition, we will use NumPy arrays (see later), which are wrappers around contiguous C arrays.

Basic Control Structures

```
import sys
# sys.argv is the list of commandline arguments
if len(sys.argv) > 1:
    # print all command line arguments
    for a in sys.argv:
        print a
else:
        # looping over integers are done with xrange
    for i in xrange(10):
        print i
```

- **Program blocks are defined by equal indentation**
- **for-loops work on anything that can be iterated over**
- **if-tests work on any type. None, 0, empty strings and empty lists evaluate to false**
- **while-loops work similar to if**

String and File Handling

Python has some powerful tools for working with strings. F.ex.:

- `str.split(delim)` **creates a list of the words in str**
- `str.join(somelist)` **joins the items of a list into one string, separated by str**
- **Perl-like regular expressions**

```
from Numeric import array, accumulate
ifile = open(filename, 'r')
ofile = open(filename2, 'w')
sums = []
for line in ifile:
    fitems = map(float, line.split())
    sums.append( sum(fitems) )
ofile.write(', '.join(sums))
```

Functions

```
def avg(a, b):
    return (a + b)/2.0
print avg(2.0, 1)

avg2 = lambda a, b: (a + b)/2.0
print avg2(2.0, 7+4j)

a, b = 2.0, 10.0
print eval("(a + b)/2.0")

class MyFunctor:
    """A functor for a*sin(b*x)."""
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __call__(self, x):
        return self.a * sin(self.b * x)
# evaluate 0.5 * sin(2.0 * 3.0)
w = MyFunctor(0.5, 2.0)
print w(3.0)
```

Functions come in several forms:

- **Regular functions defined with def**
- **Lambda functions, convenient for simple oneliners**
- **Strings can be evaluated with eval**
- **Callable objects, by defining the __call__ operator**

Object Oriented Numerics in Python

- **Python is a powerful object oriented language**
- **Everything in Python is an object**
- **There are no protected or private variables or methods, but the effect can be "simulated" with underscore prefix (`_protvar`, `__privvar`)**
- **Python supports multiple inheritance**
- **Dynamic typing implies support for generic programming (as with C++ templates)**

Special Methods (Operator Overloading)

Python supports overloading operators in your own classes by declaring some special methods, to let your own types behave like the builtin types. Some examples:

- `__add__(self,y)`: **Used for** `self+y`**, i.e.,** `x+y` **invokes** `x.__add__(y)`
- `__imul__(self,y)`: **Used for** `self *= y`
- `__cmp__(self,y)`: **Comparison, returning** `-1`**,** `0` **or** `+1` **to mean** `x<y`**,** `x==y` **or** `x>y`**, respectively**
- `__str__(self)`: **Used for** `str(self)`**, and in print statements**
- `__getitem__(self, i)`: **Used for** `y = self[i]`
- `__iter__(self)`: **Used for iterating like** `for v in self:  ...`

Functional Style Programming

**Lists are a central datatype in Python, like in functional languages.
A few built in functions let you do powerful yet simple manipulation of lists.**

- `map(<function>, <list>)` **creates a new list, which is a copy of the old list, and applies the passed function to each element in the new list**
- `filter(<function>, <list>)` **creates a new list containing only the elements from the old list where function evaluates to True**

```
# squares of integers in [0,10)
squares1 = map(lambda x: x**2, range(10))
# odd numbers in [0,10)
odd1 = filter(lambda x: x%2==1, range(10))
```

Introspection

```
if not type(a) is int:
    print 'Need an integer!'
```

Python lets you examine and edit objects and their properties runtime.

- `dir(instance)` **returns a list of the names of all the properties of the object, both variables and functions**
- `type(instance)` **returns the type of the object**
- `callable(instance)` **tells you if an object is something that can be called like a function**
- **The function** `setattr` **lets you add new variables or functions to a class**
- **All objects have a variable** `__doc__` **that can hold a documentation string**

Python has a comprehensive library

We mention a few:

- A portable interface to the operating system, e.g., file and process management (`os`), file compression (`gzip`), threads (`threads`)
- GUIs: Qt (`pyqt`), Gtk (`pygtk`), Tk (`Tkinter`), WxWindows (`wxpython`), ...
- String handling (`string`), regular expressions (`re`)
- Many applications with Python interface: Word/OpenOffice, Excel/Gnumeric, Oracle, Gimp ...
- Web modules: `cgi`, `httplib`, `xml`, `mimetools`

Scientifi c Computing with Python

- Computing with Numerical Python is introduced
  - Constructing arrays
  - Vectoring expressions
  - Slicing
  - Solving a linear system
- Some available modules for Scientific Computing are then presented
- The packages covered in this tutorial are chosen on basis of our research experiences with numerical solution of Partial Differential Equations.
- The Vaults of Parnassus[83] have an extensive list of Python modules

Numeric vs numarray vs. numpy

- There are actually three different implementations of Numerical Python (NumPy)
- `Numeric` is the original and hence widely used
- `numarray` was a reimplementation with some new features
- `numpy` is a blend of the other two, again with improvements
- The three packages have almost the same interface
- The performance differs greatly: numpy is fastest for vectorized operations, while Numeric is fastest for indexing and scalar operations
- Now it seems best to use a common interface to Numerical Python such that any of the three packages can be used as backend
- A common interface can be found at http://folk.uio.no/hpl/scripting/src/tools/py4cs/numpytools.py

Example: Solving a Differential Equation

- Python can be used as a scientific calculator, like Matlab, Octave, R, etc.

$$
\begin{aligned}
-u''(x) &= f(x) \\
x &\in [0,1] \\
u(0) = u(1) &= 0 \\
f(x) &\text{ user given}
\end{aligned}
$$

- Applications:
  - heat conductivity
  - string deflection
  - fluid flow
  - electrostatics
  - elasticity, ...

- Goal: Compute u using the Numerical Python module (aka NumPy)
- NumPy is an extension module for Python that enables efficient numerical computing

Numerical Solution Procedure

$$h = 1/(n+1)$$

$$x_i = h * i, \quad i = 0, 1, \ldots, n+1$$

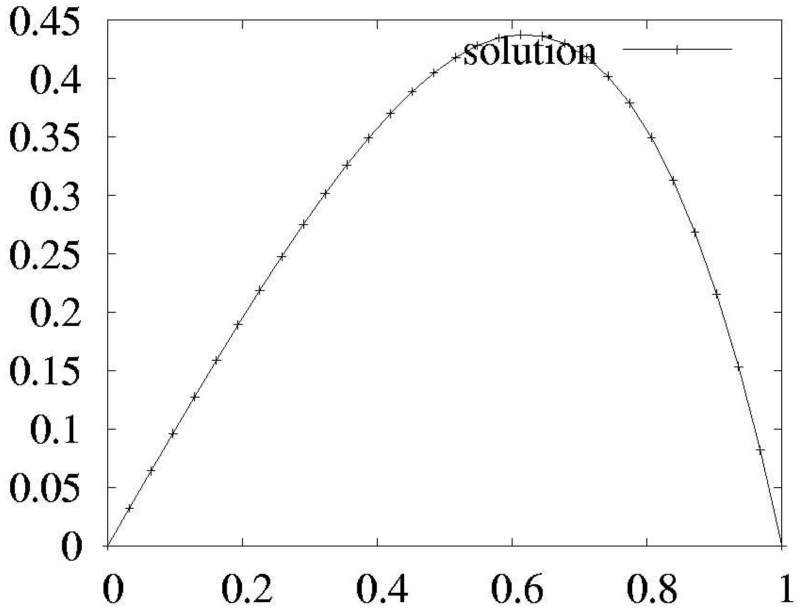
$$u_i = u(x_i)$$

$$b_i = h^2 f(x_i)$$

$$A = \begin{bmatrix} 1 & 0 & \cdots & & & & 0 \\ 0 & 2 & -1 & 0 & \cdots & & \vdots \\ \vdots & -1 & 2 & -1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -1 & 2 & -1 & \\ & & & 0 & -1 & 2 & 0 \\ 0 & \cdots & & & & 0 & 1 \end{bmatrix}$$

- **Divide $[0,1]$ into $n+1$ cells**
- **Discretization by the Finite Difference method**
- **Result:**
  linear system

$$Au = b$$



Implementation using Numerical Python

```
from Numeric import *
from LinearAlgebra import *
import sys, Gnuplot

# fetch n from commandline
n = int(sys.argv[1])

# define a source function:
def f(x):
    return (3*x+x**2)*exp(x)

h = 1.0/(n+1)                # stepsize

A = zeros((n+2,n+2),Float)   # matrix
u = zeros(n+2,Float)         # unknowns
b = zeros(n+2,Float)         # right hand side
x = zeros(n+2,Float)         # coordinates

# fill arrays
x[0] = 0.0
for i in xrange(1,n+2):     # loop interior
    x[i] = i*h
    A[i,i] = 2.0
    b[i] = h**2*f(x[i])
    if i < n:
      A[i+1,i] = A[i,i+1] = -1.0
```

```
A[0,0] = A[n+1,n+1] = 1.  # rest of matrix
b[0] = b[n+1] = 0         # rest of rhs.

# Solve the problem using a function from the
# LinearAlgebra module
u = solve_linear_equations(A,b)

# create a simple plot
g = Gnuplot.Gnuplot(persist=1)
g.title("Solution of -u''(x) = f(x)")
gdata = Gnuplot.Data(x,u,title='solution',
        with='linespoints')
g.plot(gdata)
```

Elements of the Code

- **The code on the previous slide uses NumPy arrays for storage**
- **The function `solve_linear_equations` from the LinearAlgebra module is used to solve the linear system**
- **Indices for multidimensional NumPy arrays are specified with tuples: `A[i,j]` instead of `A[i][j]`**
- **Besides this, the code looks very much like “normal” Python code**
- **On the following pages, we will speed up and shorten the code by introducing features of NumPy**
- **Note: The matrix in our example is tridiagonal, but we do not take advantage of this**

Making Arrays with NumPy

Array with zeros:

`x = zeros(shape,type)`

```
# Array with zeros:
# x[0] = x[1] = ... = x[n-1] = 0
x = zeros(n,Float)

# a matrix of zeros, type Int
A = zeros((n,n))
```

- **Default type is `Int` for all arrays**
- **Often you want to specify `Float`**
- **`shape` is an integer or a tuple of integers**

Array with ones:

`x = ones(shape,type)`

```
# Create array of ones
# x[0] = x[1] = ... = x[n-1] = 1
x = ones(n,Float)
A = ones((n,n),Float)
```

Making Arrays with NumPy, Continued

Equally spaced values:

`x=arange(start,stop,step,type)`

```
# from 0 to 1 in steps of h:
# h float => array of floats
x = arange(0,1,h)

# x0, x0+dx, ..., xend-dx<x1<xend
x = arange(x0,xend,dx)

# numbers from 0 to n-1,
x = arange(n,type=Float)
```

Array from Python sequence:

`x = array(sequence,type,shape`

```
# Create array from Python list
x = array(range(n),type=Float)

# Create matrix from Python list
z = array([0,1,2,3,4,5],shape=(2,3))

# Share data with another array
w = array(somearray,copy=False)
```

- **End of range is usually not included**

Warning: Dangerous Behavior!

- `arange` **sometimes includes the endpoint and sometimes not, depending on round-off error!**
- **Better solution: define a sequence function which behave consistently**
- **Right, we present a quick solution, and a version with most of the flexibility of** `arange, range, xrange`
- `sequence` **always include the endpoint**
- **We will use this function in our examples**

```
def sequence(min=0.0,max=None,
                 inc=1.0, type=Float):
    if max is None:
       max = min; min=0.0
    return arange(min,max + inc/2.0,
                     inc, type)

x = sequence(0,1,h,Float)
```

Array Computing

- **Consider the creation of `b`, where we used slow loops**
- **We can use arithmetic operations on NumPy arrays!**
- **Such operations apply to all elements in an array**
- **Our `f(x)` can work with both scalar `x` or NumPy `x`**

```
# Assume x created as NumPy array

# vectorized computing:
b = (h**2)*(3*x+x**2)*exp(x)

# We may also use f(x):
b = h**2*f(x)
```

Array Computing, In–place Arithmetics

- **Arithmetic array operations create temporary hidden arrays**
- **The first expression for** `b` **splits in 6 binary operations**
- **All of them implemented in C, resulting in temporary arrays**
- **We can do in-place arithmetics to save storage**
- **Remark:** `b = x` **will make** `b` **reference the same object as** `x`**!**

```
b = (h**2)*(3*x+x**2)*exp(x)
```

```
# Save temporary storage
b = x.copy() # create a copy of x
b *= 3       # multiply with scalar
b += x**2    # add directly to b
b *= h**2    # multiply in-place
b *= exp(x)
```

```
>>> b = x
>>> x[1]
1.0
>>> b *= 2
>>> x[1]
2.0
```

Indexing and Slicing of NumPy Arrays

- **We can extract parts of arrays (slices)**
- `[start:end]` **extracts** `start,start+1,...,end-1`
- `[start:end:stride]` **extracts** `start,start+stride,...,end-stride`
- **Negative indices count from the end:**
  - `[-1]` **last element**
  - `[-2]` **next last element**
- `[::]` **is valid! (whole array)**
- `[::-1]` **reverse an array!**

```
>>> # create the partition
>>> h=1./10
>>> x = sequence(0,1,h)
>>>
>>> # exclude first and last element
>>> # assign to variable:
>>> interior = x[1:-1]
>>> print x[1]
0.1
>>> interior[0] += h
>>> print x[1]   # original data changed?
0.2              # yes!
>>> # stride: pick each second element of x:
>>> xstride = x[1:-1:2]
```

- **Remark: unlike regular Python lists, a slice of NumPy arrays just references the original data!**

More on Slicing NumPy Arrays

- **We can slice multi–dimensional arrays in the same way**
- **Let us assign values to subsets of the matrix `A`**
- **The function `identity` returns a unit matrix of given size**

```
# All of A but the upper row and rightmost column:
A[1:,:-1] = -identity(n-1)

# All but the bottom row and leftmost column:
A[:-1,1:] = -identity(n-1)
```

- **In numarray, a slice can be specified as an array of indices**

```
>>> x = sequence(8)
>>> ind=[2,4]
>>> x[ind]
array([ 2., 4.])
```

```
>>> ind1=range(n-1)
>>> ind2=range(1,n)
>>> A[ind1,ind2] = A[ind2,ind1] = -1
```

Example Code, revisited

```
from numarray import *
from LinearAlgebra import *
import sys, Gnuplot

# read from commandline
n = int(sys.argv[1])

# define a source function
def f(x):
    return (3*x+x**2)*exp(x)

h = 1./(n+1) # set the stepsize

# Create and fill arrays
x = sequence(0,1,h,Float)   # The partition

# The matrix
A = identity(n+2, Float)
A[1:-1,1:-1] += identity(n)
ind1 = range(1,n)
ind2 = range(2,n+1)
A[ind1,ind2] = A[ind2,ind1] = -1.0
b = h**2*f(x[1:-1])
# force boundary condition
b[0] = b[n+1] = 0
```

```
# Solve the problem using a function from the
# LinearAlgebra module
u = solve_linear_equations(A,b)

# create a simple plot
g = Gnuplot.Gnuplot(persist=1)
g.title("Two point BV problem")
gdata = Gnuplot.Data(x,u,title='approx',
        with='linespoints')
g.plot(gdata)
```

- **Initialization of data is simplified using slices and array arithmetics**
- **All loops removed!**
- **This is an important technique for optimizing Python codes**

More Details on Computing with NumPy Arrays

- **NumPy offers the most common mathematical functions**
- **These operations are very efficient on arrays**
- **But they are slow on scalars, so use the functions from `math` in those cases**

```
>>> b = sequence(0,1,0.1)
>>> c = sin(b)
>>> c = arcsin(b)
>>> c = sinh(b)
>>> c = abs(b)
>>> c = c**2.5
>>> c = log(b)
>>> c = sqrt(b*b)
```

Vector-Matrix products

- **Matrix products are different from the mathematical tradition**
- **NewAxis can be used in slices to add a new dimension to an array**
- **Arithmetics with a column vector and a row vector in a matrix (like an outer product)**

```
# intro to all codes here:
from scipy import *
f = lambda x,y: return sin(x)*sin(y)
n = int(sys.argv[1])
x = linspace(0,1,n)
y = linspace(0,3,n)
```

```
# wrong! z will be a vector:
z = f(x,y)
```

```
# fill the array row by row:
z = zeros((n,n), Float)
for j in xrange(len(y)):
    z[:,j] = f(x,y[j])
```

```
# fully vectorized (fast!):
X, Y = x[NewAxis,:], y[:,NewAxis]
z = f(X,Y)
```

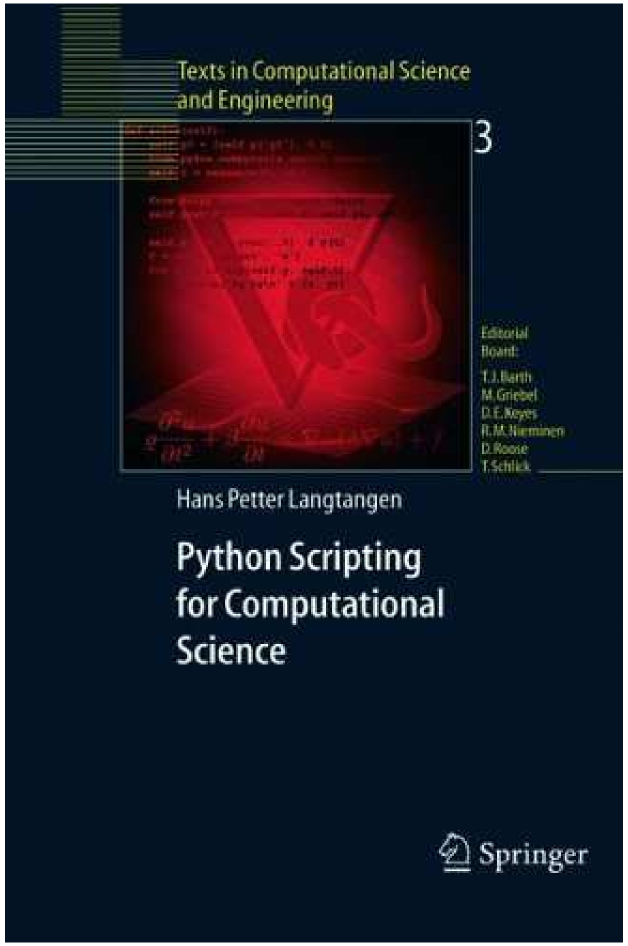
SciPy

SciPy[72] is a collection of modules for scientific computing. Most of the modules are wrappers around old, well tested and fast Fortran or C libraries.

- **Based on Numeric**
- **Linear algebra: Lapack, BLAS, iterative solvers (CG, BiCGstab etc.), eigenvalues, matrix decompositions**
- **Integration: Quadpack**
- **ODE solvers: ODEpack**
- **Interpolation: Fitpack, spline variants**
- **Optimalization: Least squares fit, various minimization algorithms**
- **Statistics: 81 continuous and 10 discrete distributions plus more**
- **Signal Processing: Convolution, filtering, fft etc.**
- **Special functions: Airy functions, Bessel functions, Fresnel integrals etc.**

Going Further

For further material on Python scripting for computer science, you might want to consult the book by Langtangen on Springer



Texts in Computational Science and Engineering

3

Editorial Board:
T.J. Barth
M. Griebel
D.E. Keyes
R.M. Nieminen
D. Roose
T. Schlick

Hans Petter Langtangen

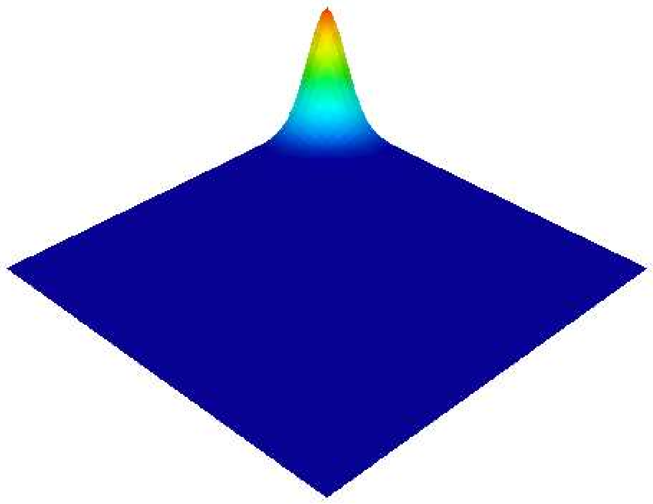
Python Scripting for Computational Science

Springer

Visualization in with Python

This section deals with plotting/visualization in a scientific setting

- **We briefly list of some of the available plotting software**
- **We use simple plotting from the Python shell without intermediate storage of data on files**
- **We will interface more advanced visualization programs - e.g. MayaVi**



Python Plotting and Visualization

- Plotting has traditionally been seen as a separate task after model simulations were accomplished
- Plots not corresponding to the data reported were (and still are) presented
- Python is well suited to control/assist the plotting process
- Automating tedious tasks helps to ensure consistency between input and output data

2D Plotting and Graphing

- We show examples for:
  - Gnuplot
  - Matplotlib
- The quality of the plots varies and so does the interfaces
- All packages presented here should run on both Windows and Linux unless otherwise stated. The pieces of code presented are however only tested on Linux

Comments on Code

All examples in this 2D section can use either Numeric or Numarray, i.e. either

```
from Numeric import *
```

or

```
from numarray import *
```

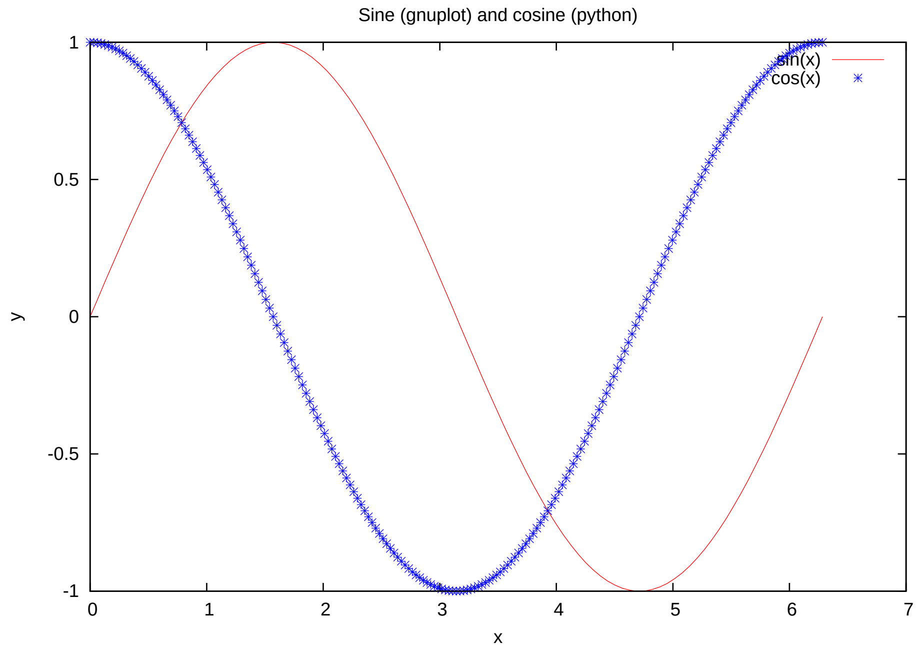
is implied.

Also: the import or inclusion of sequence as previously defined (but repeated here) is implied:

```
def sequence(min=0.0,max=None,
             inc=1.0, type=Float):
   if max is None:
      max = min; min=0.0
   return arange(min,max + inc/2.0,
                 inc, type)
```

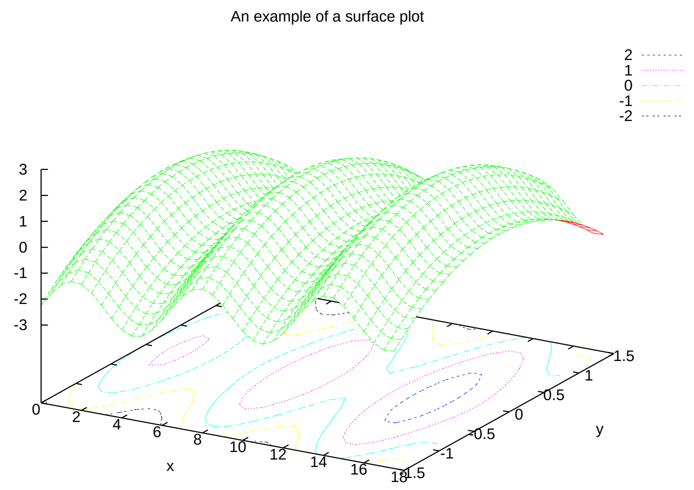
Gnuplot-py Example

```
import Gnuplot

g = Gnuplot.Gnuplot(debug=1)
x = sequence(0,2*pi,pi/100.)
y = cos(x)
d = Gnuplot.Data(x, y,\
         with='points 3 3',title='cos(x)')
g.title('Sine (gnuplot) and Cosine (Python)')
g.xlabel('x')
g.ylabel('y')
g('set output "gnuplot.eps"')
g('set terminal postscript eps color \
  "Times" 32')

g.plot(Gnuplot.Func('sin(x)'),d)
```



Gnuplot in 3D

```
from Numeric import *
import Gnuplot

g = Gnuplot.Gnuplot(debug=1)
x = sequence(0.,17.5,.5)
y = sequence(0.,3.,.1) -1.5

xm = x[:,NewAxis]
ym = y[NewAxis,:]
m = (sin(xm) + 0.1*xm) - ym**2

g('set parametric')
g('set data style lines')
g('set hidden')
g('set contour base')
g.title('An example of a surface plot')
g.xlabel('x')
g.ylabel('y')

g('set output "gnup3d.eps"')
g('set terminal postscript eps color \
   "Times" 24')

g.splot(Gnuplot.GridData(m,x,y))
```



Gnuplot-py

- http://gnuplot-py.sourceforge.net
- Python interface to Gnuplot[26], the "old" plotting program for Unix
- SciPy has also wrapped gnuplot in their plotting module scipy.gplt

+ Gnuplot itself has been refurbished over the last years

+ Has many users

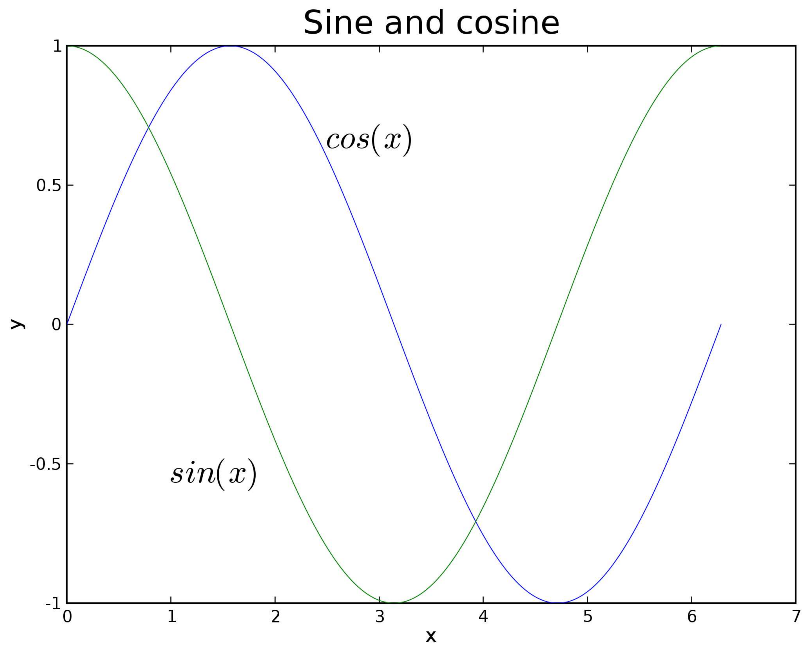
+ Has simple 3D capabilities (contour and surface plots)

– The default plots could be prettier

– The Python interface could be better

Matplotlib Example

```
from matplotlib.pylab import *

x = sequence(0,2*pi,pi/100.)
y = sin(x)
z = cos(x)
plot(x,y)
plot(x,z)

title(r'Sine and Cosine', fontsize=20)
text(1, -0.6, r'$sin(x)$', fontsize=20)
text(2.5, 0.6, r'$cos(x)$', fontsize=20)
xlabel('x')
ylabel('y')
savefig('matplotlib.png', dpi=300)
show()
```



Matplotlib

- http://matplotlib.sourceforge.net

+ High quality plots 2D plots
+ Supports several output formats
+ The plotting functions has a high degree of Matlab compatibility
+ Partially supports TeX fonts
+ Actively developed

– Lacks 3D capabilities
– Still in beta

Summary of 2D Plotting

- Because of portability, we would recommend Gnuplot if you find the plot quality sufficient
- Use Matplotlib if matlab compatible commands is important
- Other alternatives are PyX, Python-biggles, Pychart and RPy

Image Processing within Python

- Numarray contains a large set of image processing functions
- PIL (Python Imaging Library)
    - http://www.pythonware.com/products/pil/index.htm
    - Adds image processing capabilities to Python
    - `scipy.pilutil` has some extra utility functions (f.ex. mapping PIL images to Numeric arrays)

```
import Image
im = Image.open('heat.ppm')
print im.format, im.size, im.mode
im.rotate(90) # degrees counter-clockwise
im.save('heat.jpg','JPEG')
```

- PythonMagick
    - http://www.procoders.net/moinmoin/PythonMagick
    - Python bindings for GraphicsMagick
    - Supports $\sim$90 image formats

Computer graphics - OpenGL and Open Inventor

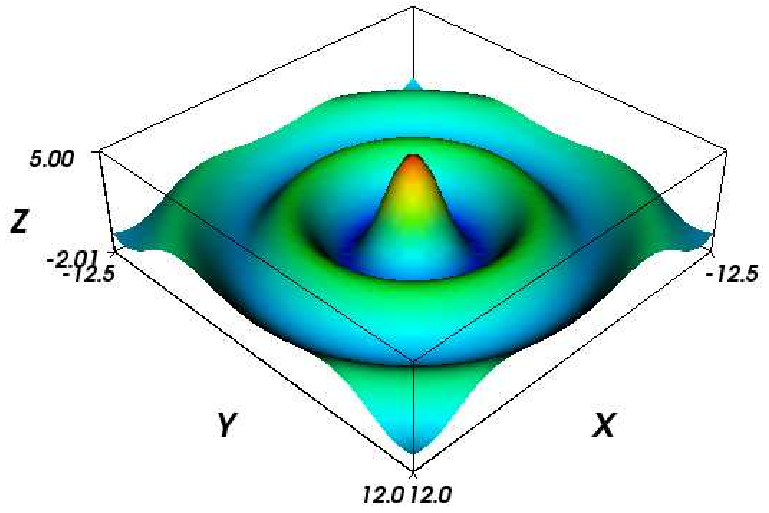
- When it comes to high quality rendering, OpenGL is the de facto standard
- Some like to program directly in OpenGL, others in libraries on top of OpenGL like Open Inventor
- PyOpenGL[62] is the cross platform Python binding to OpenGL
  - Complete low level control over the graphics
- Pivy is python bindings for Coin[11]
  - http://pivy.tammura.at
  - Open source implementation of Open Inventor
  - Lets you work with a more abstract scene graph

Visualization - MayaVi and VTK

- http://mayavi.sourceforge.net
- Python interface to the Visualization ToolKit:
  - http://www.vtk.org
  - VTK is a very powerful object-oriented library; it supports both structured and unstructured grids
  - VTK itself comes with a Python interface: VTK-Python
  - Using MayaVi is easier than using VTK-Python directly
- Focuses on visualization
- MayaVi comes with a GUI, but can also be used from scripts
- Recommended add on: Pyvtk[66], to manipulate VTK files
- The ivtk module makes it easy to experiment with VTK

MayaVi Visualization

```
import mayavi
v = mayavi.mayavi() # create a MayaVi window.
d = v.open_vtk('/tmp/test.vtk', config=0)
# open the data file.
# The config option turns on/off showing a GUI control for the data/filter/
# module.
# load the filters.
f = v.load_filter('WarpScalar', config=0)
n = v.load_filter('PolyDataNormals', 0)
n.fil.SetFeatureAngle (45)
# configure the normals.
# Load the necessary modules.
m = v.load_module('SurfaceMap', 0)
a = v.load_module('Axes', 0)
a.axes.SetCornerOffset(0.0)
# configure the axes module.
o = v.load_module('Outline', 0)
v.Render() # Re-render the scene.
v.renwin.save_png('/tmp/image.png')
```



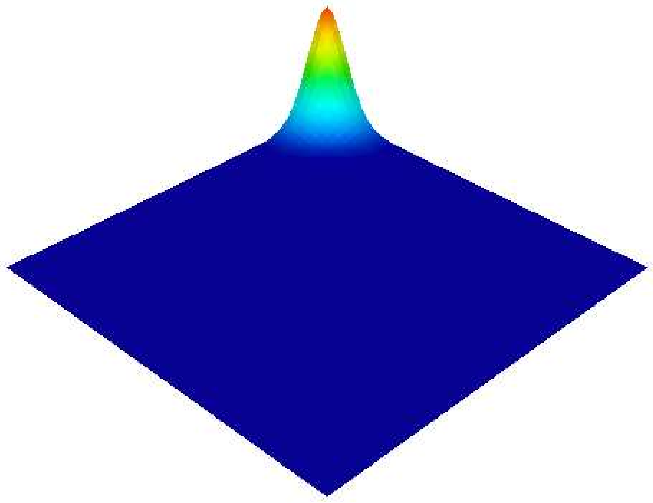
Play with MayaVi from Python

- The Interactive VTK module (ivtk)
- The ivtk module included in MayaVi makes it easier to experiment with VTK from Python
- Includes:
  - A VTK actor viewer
  - Access to VTK documentation
  - GUI for VTK configuration
  - Menus for saving images of the scene

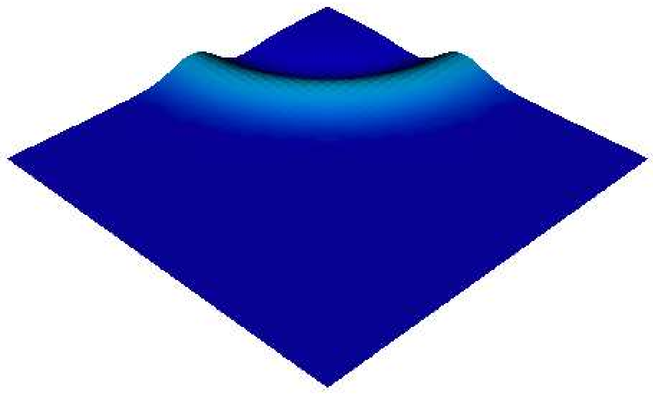
```
from mayavi import ivtk
from vtkpython import *
c = vtkConeSource()
m = vtkPolyDataMapper()
m.SetInput(c.GetOutput())
a = vtkActor()
a.SetMapper(m)
v = ivtk.create_viewer()

v.AddActors(a)
v.config(c)
v.doc(c)
v.help_browser()
v.RemoveActors(a)
```

Real MayaVi Example

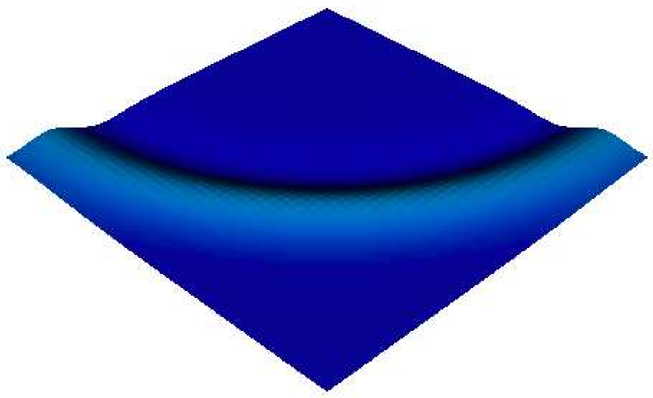
$t = 0$



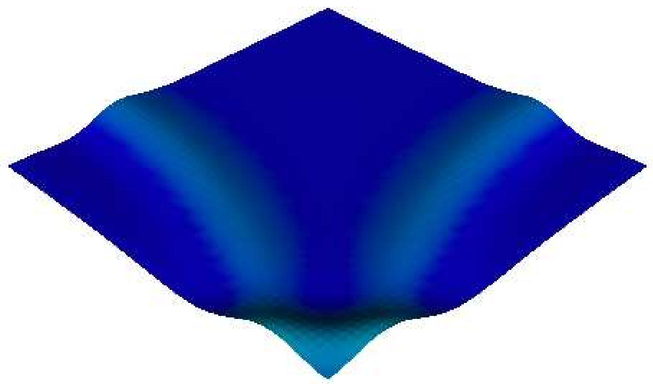
$t = 30$



$t = 60$



$t = 90$



Real MayaVi Example; Motivation

- We assume we have some efficient FORTRAN code (shown later) for the 2D Wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial}{\partial x}(\lambda \frac{\partial u}{\partial x}) + \frac{\partial}{\partial y}(\lambda \frac{\partial u}{\partial y}) \right)$$

- We want to make a nice animation of the solution, integrated with the program execution
- We want to accomplish this fast with much reuse of code:
  - Python class wrapping the FORTRAN simulator
  - Allocate data structures in Python, pass to F77,
    let MayaVi/VTK do the visualization

Real MayaVi Example; Python Code

```
from vtkpython import vtkDoubleArray,\
     vtkStructuredPoints
import Numeric as N
import wave1 #This is the FORTRAN module
from Numeric import sqrt, sin, exp, cos
from time import time
import mayavi

class Wave:
   def __init__(self,n=41,nsteps=40):
      self.setSize(n,nsteps)
      self.init()
      self.icTime=0
      self.solveTime=0
   def setSize(self, n, nsteps):
      self.n = n
      self.nsteps = nsteps
      self.L = 10.0
      self.delta = self.L/(n-1)
      self.dt = sqrt(1/(1/(self.delta**2)+\
      1/(self.delta**2)))
      self.tstop = self.dt*nsteps
   def init(self):
      t = "f"; n=self.n
      self.up  = N.zeros([n,n],t)
      self.u   = N.zeros([n,n],t)
      self.um  = N.zeros([n,n],t)
      self.lam = N.ones([n,n],t)
```

```
   def setIc(self, bottom=lambda i,j:\
      N.ones([i,j],'d'), surface=lambda x,y:\
      sin(x)*sin(y)):
      tmp = time()
      n=self.n
      x = self.delta*N.arange(n)
      y = x[:,N.NewAxis].copy()
      self.u = surface(x,y)
      self.lam = bottom(x,y)
      self.um = self.u.copy()
      self.u =  wave1.as_column_major_storage\
      (self.u )
      self.um = wave1.as_column_major_storage\
      (self.um)
      self.up = wave1.as_column_major_storage\
      (self.up)
      self.lam = wave1.as_column_major_storage\
      (self.lam)
      self.icTime= time()-tmp
   def initMovie(self):
      self.v = mayavi.mayavi()
      self.arr = vtkDoubleArray()
      data = vtkStructuredPoints()
      data.SetSpacing(self.delta,self.delta,1)
      data.SetDimensions(self.n,self.n,1)
      self.updateMovie()
```

Real MayaVi Example; Python Code, continued

```
      data.GetPointData().SetScalars(self.arr)
      self.v.open_vtk_data(data)
      self.v.load_module('SurfaceMap', 0)
      self.v.load_filter('WarpScalar', config=0)
   def updateMovie(self):
      self.arr.SetVoidArray(N.reshape\
      (N.transpose(self.u),(-1,)),self.n**2,1)
      self.arr.Modified()
#     Uncomment to rotate
#      self.v.renwin.camera.Azimuth(1)
      self.v.Render()
   def saveFile(self, number=0):
      self.v.renwin.save_png\
      ('/tmp/wave%04d.png'%number)
   def solveProblem(self, movie=False, file=False):
      if movie:
         self.initMovie()
         if file: self.saveFile()
      tmp = time()
      for i in range(self.nsteps):
         [self.up, self.u, self.um] = wave1.\
         solveatthistimestep(self.up, self.u,\
         self.um, self.lam, self.dt)
         if movie:
            self.updateMovie()
            if file: self.saveFile(i+1)
      self.solveTime=time()-tmp; self.printStats()
```

```
      return (self.u)
   def printStats(self):
      print "--------------"
      print "Timing results"
      print "--------------"
      print 'setIC        = %6.2f' \
      % (self.icTime,)
      print 'solveProblem = %6.2f' \
      % (self.solveTime)
      print 'Framerate    = %6.2f'
      % (self.nsteps/self.solveTime,)
def surface1(x,y): return 3*exp(-x*x-y*y)
def surface2(x,y): return sin(x)*cos(y)
def bottom1(x,y):
   return N.ones([len(x),len(y)],'d')
def fault(x,y,z): return x+y+z
if __name__ == '__main__':
   from sys import argv
   n_ = 51; nsteps_ = 1000
   try: n_ = int(argv[1]); nsteps_ =\
    int(argv[2])
   except: pass
   w = Wave(n = n_, nsteps = nsteps_)
   w.setIc(surface = surface1, \
    bottom=bottom1)
   w.solveProblem(movie=True,file=True)
```

Real MayaVi Example; FORTRAN Code

```
      SUBROUTINE setIC(u, um, lambda, n,
     &                 bottom, surface)
C     set initial conditions (rough
C     approximations) set lambda values
C     as well
      INTEGER n
      REAL*8 u(n,n), um(n,n), lambda(n,n),
     &       bottom, surface
Cf2py intent(in,out) u, um, lambda
      EXTERNAL bottom, surface
      INTEGER i, j
      REAL*8 x, y, delta
C     domain has size 10x10 in x and y
C     direction, delta is the cell size:
      delta = 10.0/(n-1)

      DO 20 j=1,n
         DO 10 i=1,n
            x = (i-1)*delta
            y = (j-1)*delta
            u(i,j) = surface(x,y)
C           this is a rough approximation
C           to du/dt=0:
            um(i,j) = u(i,j)
C           initialize the variable
C           coefficient as an array:
            lambda(i,j) = bottom(x,y)
 10      CONTINUE
```

```
 20   CONTINUE
      RETURN
      END

      SUBROUTINE solveAtThisTimeStep(up,u,um,
     &           lambda,n,dt)
      INTEGER n
      REAL*8 up(n,n), u(n,n), um(n,n),
     & lambda(n,n)
Cf2py intent(in, out) up,u,um
      REAL*8 dt
      REAL*8 delta, a, b, c
      delta = 10.0/(n-1)
      a = 1.0
      b = 1.0
      c = 1.0

      call F77WAVE(up, u, um, lambda, a, b, c,
     &             n, n, dt, delta, delta)
C     update for next step:
      DO 20 j=1,n
         DO 10 i=1,n
            um(i,j) = u(i,j)
            u(i,j) = up(i,j)
 10      CONTINUE
 20   CONTINUE
      END
```

Real MayaVi Example; FORTRAN Code, continued

```
      SUBROUTINE F77WAVE(up, u, um, lambda,
     & a, b, c, nx, ny, dt, dx, dy)
      IMPLICIT LOGICAL (A-Z)
      INTEGER nx, ny
      REAL*8 up(nx,ny), u(nx,ny), um(nx,ny),
     &lambda(nx,ny)
      REAL*8 a, b, c
      REAL*8 dt, dx, dy
      INTEGER i,j

      DO 20 j = 2, ny-1
         DO 10 i = 2, nx-1
            up(i,j) = a*2*u(i,j) - b*um(i,j) +
     & c*(dt*dt)/(dx*dx)* ( 0.5*(lambda(i+1,j )+
     & lambda(i ,j ))*(u(i+1,j)-u(i ,j )) -0.5*
     & (lambda(i ,j )+lambda(i-1,j ))*
     & (u(i ,j )-u(i-1,j))) +(dt*dt)/(dy*dy)*
     & ( 0.5*(lambda(i ,j+1)+lambda(i ,j ))*
     & (u(i ,j+1)-u(i ,j )) -0.5*(lambda(i ,j )+
     & lambda(i ,j-1))*(u(i ,j )-u(i ,j
     & -1)))
 10   CONTINUE
 20   CONTINUE

C Boundary points:
CCCCC Only presented for one of the sides!!!
```

```
      i=1
      DO 30 j = 2, ny-1
         up(i,j) = a*2*u(i,j) -
     & b*um(i,j)+ c*(dt*dt)/(dx*dx)*
     & ( 0.5* (lambda(i+1,j )+
     & lambda(i ,j ))*
     & (u(i+1,j )-u(i ,j ))- 0.5*
     & (lambda(i ,j )+ lambda(i+1,j ))*
     & (u(i ,j )-u(i+1,j)))
     & +(dt*dt)/(dy*dy)*
     & ( 0.5*(lambda(i ,j+1)+
     & lambda(i ,j ))*
     & (u(i ,j+1)-u(i ,j ))
     & -0.5*(lambda(i ,j )+
     & lambda(i ,j-1))*
     & (u(i ,j )-u(i ,j-1)))
 30   CONTINUE

CCCCCC[snip]

      RETURN
      END
```

Making a Movie

- In the MayaVi example above an image file is produced at each time step
- To make a movie, use mencoder from the mplayer project http://www.mplayerhq.hu
- Mencoder can be called from Python in this way:

```
import os
cmd = ('mencoder', 'mf://*.png', '-mf',\
       'type=png:w=800:h=600:fps=25','-ovc', 'lavc',\
       '-lavcopts', 'vcodec=mpeg4', '-oac', 'copy',\
       '-o', 'output.avi')
os.spawnvp(os.P_WAIT, 'mencoder', cmd)
```

Conclusions for High Quality Visualizations

- There are a several possibilities for making advanced plots from Python
- If absolute control is a must, PyOpenGL or Pivy might be the answer
- If the purpose is to make good and quick visualization, MayaVi will be our recommendation
- If you’re interested in Medical Image Processing, have a look at ITK/VTK

Performance Issues – Tips and Tricks

- Native Python is too slow for number crunching
- Difficult to port knowledge from C/C++ and FORTRAN
- Learn by testing
- We will briefly cover:
  - How to use the profiling and timing tools in Python
  - Some Python performance tricks

Manual Timing

- `time` **module:**

```
import time
e0 = time.time()  # elapsed time since the epoch (1970.01.01)
c0 = time.clock() # total CPU time spent in the script so far
<do tasks...>
elapsed_time = time.time()  - e0
cpu_time     = time.clock() - c0
```

- **If just a few statements are involved in the test, repeat them in a loop and compute the mean**
- **CPU time measurements less than a couple of seconds may be unreliable**
- **Run each test several times and choose the fastest result**
- **The** `os.times` **function returns user, system and elapsed time**

`timeit` module: for repeating code snippets

Which is fastest:

- `from math import sin; sin(1.2)`
- `import math; math.sin(1.2)`

```
>>> import timeit
>>> t = timeit.Timer('sin(1.2)', setup='from math import sin')
>>> t.timeit(10000000)   # run 'sin(1.2)' 10000000 times
11.830688953399658
>>> t = timeit.Timer('math.sin(1.2)', setup='import math')
>>> t.timeit(10000000)
16.234833955764771
```

Reason

- `sin` **needs one look-up (in** `globals`**)**
- `math.sin` **needs two look-ups**
  (in `globals` **and** `math`**)**

The `hotshot` Module

- **Basic usage:**

```
import hotshot
pr = hotshot.Profile("filename")
pr.run(cmd)
pr.close() # Close log-file and end profiler
```

- **Profile function calls:** `pr.runcall(func, *args, **kw)`
- **Execute and profile a string:** `pr.runctx(cmd, globals, locals)`
- **Read profile data:**

```
import hotshot.stats
data = hotshot.stats.load("filename")# profile.Stats instance
data.print_stats()
```

- **Sorting:** `data.sort_stats('sort order')` **e.g.** `call, time, name`
- **Multiple sort strings can be used to tune the order**

The `hotshot` profiling module, continued

```
import sys, os
script = sys.argv[1]

import hotshot, hotshot.stats
prof = hotshot.Profile("prof.out")
prof.run('execfile(' + script + ')')
p = hotshot.stats.load("prof.out")
p.strip_dirs().sort_stats('time').print_stats(20)
```

```
sample output:

      1082 function calls (728 primitive calls) in 17.890 CPU seconds

  Ordered by: internal time
  List reduced from 210 to 20 due to restriction <20>

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       5    5.850    1.170    5.850    1.170 m.py:43(loop1)
       1    2.590    2.590    2.590    2.590 m.py:26(empty)
       5    2.510    0.502    2.510    0.502 m.py:32(myfunc2)
       5    2.490    0.498    2.490    0.498 m.py:37(init)
       1    2.190    2.190    2.190    2.190 m.py:13(run1)
       6    0.050    0.008   17.720    2.953 funcs.py:126(timer)
  ...
```

Some Python Performance Tips

Use profiling and/or manual timing to find bottlenecks before bothering to optimize.

- **Exceptions are slow**
- **Function calls are slow**
- **The time of calling a function grows linearly with the number of arguments**
- **Symbols are found run-time in dictionaries:**
  - **Refering to global variables are slower than locals (the local namespace is searched first)**
  - **math.sin is slower than sin (two lookups)**
- **Be particularly careful in long loops, as usual**

Parallel Computing via Python

- **Message passing as the main principle (thread-based parallelization not yet mature for Python)**
- **Intensive serial computations by mixed language implementation**
- **High-level inter-processor communication via Python (user-friendly MPI modules)**
- **Satisfactory parallel performance relies on array slicing and reshaping**

Different Python MPI Modules

- **pyMPI (Pat Miller, LLNL)**
- **pypar (Ole Nielsen, Australian National University)**
- **MYMPI (Timothy Kaiser, San Diego Supercomputing Center)**
- **pyre (Michael Aivazis, CalTech)**
- **ScientificPython (Konrad Hinsen, Centre de Biophysique Moleculaie)**

pyMPI vs. pypar

- pyMPI is an MPI module plus a special Python interpreter (capable of interactively executing parallel programs)
  - pyMPI provides a rather complete interface to MPI
  - pyMPI has simple syntax
  - pyMPI is flexible
    (any serializable Python type can be communicated)
  - pyMPI is of relatively low performance

- pypar is a light-weight MPI module of high performance
  - pypar provides bindings to a small subset of MPI routines
  - pypar has simple syntax
    (optional functionality available via keyword arguments)
  - efficient mode and flexible mode of communication
    (array vs. arbitrary object)

An MPI Example Using pypar

```
import pypar                                  # The Python-MPI interface

numproc = pypar.size()
myid =    pypar.rank()
node =    pypar.get_processor_name()

print "I am proc %d of %d on node %s" %(myid, numproc, node)

if myid == 0:

  msg = "P0"
  pypar.send(msg, destination=1)
  msg = pypar.receive(source=numproc-1)

  print 'Processor 0 received message "%s" from processor %d' %(msg, numproc-1)

else:

  source = myid-1
  destination = (myid+1)%numproc

  msg = pypar.receive(source)
  msg = msg + 'P' + str(myid)                             # Update message
  pypar.send(msg, destination)

pypar.finalize()
```

Latency and Bandwidth

- **Ping-pong test: measurement of latency and bandwidth**
- **Platform: Linux cluster using fast ethernet (100 Mbit/s peak bandwidth)**

	Latency	Bandwidth
C-version MPI	133×10^{-6} s	**88.176 Mbit/s**
`pypar`**-layered MPI**	225×10^{-6} s	**88.064 Mbit/s**
`pyMPI`**-layered MPI**	542×10^{-6} s	**9.504 Mbit/s**

- **Correct use of `pypar` for efficiency:**

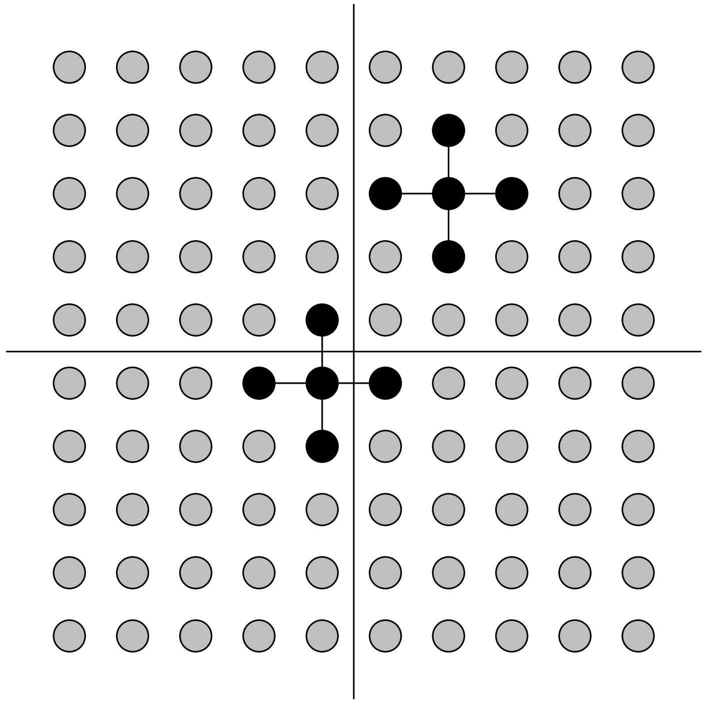
```
pypar.send (msg_out, destination=to, bypass=True)
msg_in = pypar.receive (from, buffer=msg_in_buffer, bypass=True)
```

Need for Communication; Example

- **Consider a five-point stencil associated with FDM**

```
u_loc = zeros((nx_loc+1,ny_loc+1), Float)
um_loc = ones((nx_loc+1,ny_loc+1), Float)
for i in xrange(1,nx_loc):
    for j in xrange(1,ny_loc):
        u_loc[i,j] = um_loc[i,j-1] + um_loc[i-1,j] \
                     -4*um_loc[i,j] + um_loc[i+1,j] + um_loc[i,j+1]
```

- **Communication is needed across internal boundaries between subdomains**



Communication in x-direction

```
if has_upper_x_neighbor:
    pypar.send (u_loc[nx_loc-1,:], destination=upper_x_neighbor_id, bypass=True)
    u_loc[nx_loc,:] = pypar.receive (upper_x_neighbor_id, \
                                     buffer=buffer_x, bypass=True)

if has_lower_x_neighbor:
    pypar.send (u_loc[1,:], destination=lower_x_neighbor_id, bypass=True)
    u_loc[0,:] = pypar.receive (lower_x_neighbor_id, \
                                buffer=buffer_x, bypass=True)
```

- **Preparation of an outgoing message** `u_loc[nx_loc-1,:]`
- **Use of** `bypass=True` **option for performance**
- **Allocation of** `buffer_x` **is done beforehand**

Communication in y-direction

```
if has_upper_y_neighbor:
    pypar.send (u_loc[:,ny_loc-1], destination=upper_y_neighbor_id, bypass=True)
    u_loc[:,ny_loc] = pypar.receive (upper_y_neighbor_id, \
                                     buffer=buffer_y, bypass=True)

if has_lower_y_neighbor:
    pypar.send (u_loc[:,1], destination=lower_y_neighbor_id, bypass=True)
    u_loc[:,0] = pypar.receive (lower_y_neighbor_id, \
                                buffer=buffer_y, bypass=True)
```

- **Allocation of `buffer_y` is done beforehand**
- **Use of array slicing is important!**

2D Wave Equation; FDM

- **Mathematical model**

$$\begin{aligned}
\frac{\partial^2 u(x,y,t)}{\partial t^2} &= c^2 \nabla^2 u(x,y,t) + f(x,y,t) \quad \text{in } \Omega, \\
u(x,y,t) &= g(x,y,t) \quad \text{on } \partial\Omega,
\end{aligned}$$

- **FDM discretization**

$$\begin{aligned}
u_{i,j}^{l+1} &= -u_{i,j}^{l-1} + 2u_{i,j}^{l} \\
&+ c^2 \frac{\Delta t^2}{\Delta x^2} \left(u_{i-1,j}^{l} - 2u_{i,j}^{l} + u_{i+1,j}^{l} \right) \\
&+ c^2 \frac{\Delta t^2}{\Delta y^2} \left(u_{i,j-1}^{l} - 2u_{i,j}^{l} + u_{i,j-1}^{l} \right) \\
&+ \Delta t^2 f(x_i, y_j, l\Delta t).
\end{aligned}$$

2D Wave Equation; Parallelization

- **Domain decomposition as work load partitioning**
- **Serial computation within each subdomain**
- **At the end of each time step:**
  - **preparation of outgoing messages (array slicing)**
  - **exchange of messages between each pair of neighboring subdomains**
  - **extraction of incoming messages for the update of ghost points**

2D Wave Equation; Measurements

- **Three approaches:**
  - **Fortran 77 (serial subdomain computation) +** `pypar`
  - **C (serial subdomain computation) +** `pypar`
  - **Pure C parallel implementation (no Python at all)**
- 2000×2000 **mesh;** 5656 **time steps**

	Python-Fortran		Python-C		Pure C	
P	Time	Speedup	Time	Speedup	Time	Speedup
1	223.34	N/A	253.98	N/A	225.89	N/A
2	114.75	1.95	129.72	1.96	115.83	1.95
4	60.02	3.72	68.69	3.70	61.34	3.68
8	33.28	6.71	36.79	6.90	32.59	6.93
16	18.48	12.09	20.89	12.16	18.34	12.32
32	13.85	16.13	14.75	17.22	12.15	18.59
64	9.41	23.73	10.12	25.10	7.66	29.49
128	6.72	33.24	7.42	34.23	3.83	58.98

Schwarz-Type Parallelization

- **Global PDE**

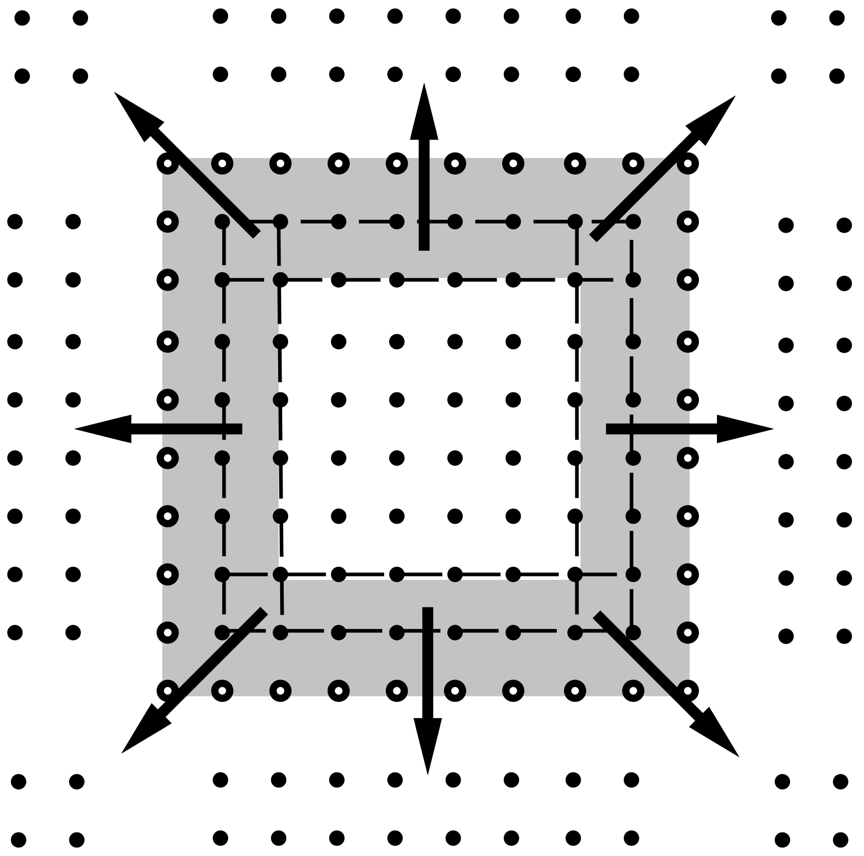
$$\begin{aligned} \mathcal{L}(u) &= f, \quad x \in \Omega \\ u &= g, \quad x \in \partial\Omega \end{aligned}$$

- **Overlapping domain decomposition:** $\Omega = \{\Omega_s\}_{s=1}^{P}$
- **Additive Schwarz iterations,** $k = 1, 2, \ldots$

$$\begin{aligned} u_{s,k} &= \tilde{\mathcal{L}}^{-1} f, \quad x \in \Omega_s \\ u &= g_s^{\text{artificial}}, \quad x \in \partial\Omega_s \backslash \partial\Omega \\ u &= g, \qquad x \in \partial\Omega \end{aligned}$$

- **Inherently suitable for parallelization**
- **Reuse of serial code on each subdomain**
- **Message passing for inter-subdomain communication**

Additive Schwarz Framework



- Generic tasks:
  - domain decomposition
  - communication between subdomains
  - control of subdomain computation and check of convergence
- Python can be used to implement a generic framework

A High-Level Parallelization Strategy

- **Reuse of existing serial code as subdomain solver (after small modification)**
- **Insertion of subdomain solvers into the additive Schwarz framework**
- **Python is well suited for this type of parallelization!**

Parallelizing a Legacy F77 Code

- **Boussinesq water wave equations**

$$\frac{\partial\eta}{\partial t}+\nabla\cdot\left((H+\alpha\eta)\nabla\phi+\epsilon H\left(\frac{1}{6}\frac{\partial\eta}{\partial t}-\frac{1}{3}\nabla H\cdot\nabla\phi\right)\nabla H\right) = 0 \qquad (1)$$

$$\frac{\partial\phi}{\partial t}+\frac{\alpha}{2}\nabla\phi\cdot\nabla\phi+\eta-\frac{\epsilon}{2}H\nabla\cdot\left(H\nabla\frac{\partial\phi}{\partial t}\right)+\frac{\epsilon}{6}H^2\nabla^2\frac{\partial\phi}{\partial t} = 0 \qquad (2)$$

- **A legacy F77 code consists of two main subroutines:**
  - `iteration_continuity` **solves (1) for one time step**
  - `iteration_bernoulli` **solves (2) for one time step**
- **A Python class** `BoxPartitioner` **hierarchy has implemented the additive Schwarz framework associated with FDM**

Parallelization Result

```
from BoxPartitioner import *

# read in ’gnum_cells’,’parts’,’overlaps’ (details skipped)

partitioner=PyParBoxPartitioner2D(my_id=my_id,num_procs=num_procs,
                                  global_num_cells=gnum_cells,
                                  num_parts=parts,
                                  num_overlaps=overlaps)

partitioner.prepare_communication ()

loc_nx,loc_ny = partitioner.get_num_loc_cells ()

# create subdomain data arrays (details skipped)
# enforce initial conditions (details skipped)

lower_x_neigh = partitioner.lower_neighbors[0]
upper_x_neigh = partitioner.upper_neighbors[0]
lower_y_neigh = partitioner.lower_neighbors[1]
upper_y_neigh = partitioner.upper_neighbors[1]

import BQ_solver_wrapper as f77 # interface to legacy code
```

Parallelization Result; Cont'd

```
t = 0.0
while t <= tstop:
    t += dt

    # solve the continuity equation:
    dd_iter = 0
    not_converged = True
    nbit = 0

    while not_converged and dd_iter < max_dd_iters:
        dd_iter++
        Y_prev = Y.copy()  # remember old eta values
        Y, nbit = f77.iteration_continuity (F, Y, YW, H, QY, WRK,
                                            dx, dy, dt, kit, ik,
                                            gg, alpha, eps, nbit,
                                      lower_x_neigh, upper_x_neigh,
                                      lower_y_neigh, upper_y_neigh)
        partitioner.update_overlap_regions (Y) # communication
        not_converged = check_convergence (Y, Y_prev)

    # solve the Bernoulli equation:
    # details skipped
```

Speedup Measurements

- 1000×1000 global mesh, number of time steps: 40

P	Partitioning	Wall-time	Speed-up
1	N/A	1810.69	N/A
2	1×2	847.53	2.14
4	2×2	483.11	3.75
6	2×3	332.91	5.44
8	2×4	269.85	6.71
12	3×4	187.61	9.65
16	2×8	118.53	15.28

- Better speedup results (than simple 2D wave equation) due to heavier computational work per subdomain

Extend Your Favorite Library with Python

We will briefly describe how to extend Python with FORTRAN/C/C++ code

This is done by:

- **Explaining the difference between Python and the compiled languages FORTRAN/C/C++**
- **Showing some simple manual wrapping code**
- **Describing the Numeric C-API**
- **Describing the tool F2PY for FORTRAN**
- **Describing the tool SWIG for C/C++**

Python Objects are Dynamically Typed

- **A variable may contain objects of different types**
- **All info is stored in a C struct** `PyObject`
- **A variable's type is declared statically in compiled languages**
- **We cannot necessarily pass a** `PyObject` **to a FORTRAN/C/C++ function**
- **It is necessary to convert to the underlying C/FORTRAN data types (int, float, . . .)**

```
d = 3
# d is an integer

import Tkinter
tk = Tkinter.Tk()
d = Tkinter.Button(tk)
# d is a Tk button
```

What Do We Know About a Python Object?

- **Every Python object is of the generic type `PyObject` (a C struct)**
- **A particular object is usually a sub type (sub class)**
- **For instance, the Python integer is of type `PyIntObject`**
- **A sub type such as a Python integer can be converted to a proper C integer**
- **The conversion needs to be done (before passing the data to FORTRAN/C/C++ code)**

It Is “Easy” to Extend Python With C/C++/FORTRAN

- C data types may be constructed from Python data types and vica versa
- A C-integer can be made from a Python-integer:

```
int PyInt_As_LONG(PyObject *io)
```

- A Python-integer can be made from a C-integer:

```
PyObject* PyInt_FromLong(int ival)
```

- A general function to extract C data is:

```
int PyArg_ParseTuple(...)
```

- The corresponding function for building Python objects:

```
PyObject *Py_BuildValue(...)
```

(The two last functions will be explained later)

A Simple Example

- Assume that we have implemented the factorial function in C
- The function takes one integer as input and returns an integer

To use the function from Python we must:

- Convert the Python integer to a C integer
- Call the C function
- Convert the returned integer to a Python integer

The code doing this is usually called the *wrapper* code

Use in Python

```
>>> from fact import fact
>>> d = 3
>>> e = fact(d)
>>> print e
6
```

C function

```
int fact(int i) {
  if (i <= 1) return 1;
  else return i*fact(i-1);
}
```

Wrapper Code Example

```c
#include "Python.h"

static PyObject *wrap_fact(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    int i, result;

    if(!PyArg_ParseTuple(args,"i",&i))  /* Extract the C integer */
      return  NULL;
    result = fact(i);                   /* Call the C function */

    return Py_BuildValue("i", result);  /* Build a Python integer and return it */
}

static PyMethodDef factMethods[] = {
         { "fact", wrap_fact, METH_VARARGS },  { NULL, NULL }
};

PyMODINIT_FUNC
initfact(void)
{
  (void) Py_InitModule("fact", factMethods);
}
```

Building the Python Module

We compile and link the wrapping code and the C code into a shared library (on Unix)

```
> gcc -c -fpic  fact_wrap.c fact.c -I..  \
  -DHAVE_CONFIG_H -I/home/kent-and/local/include/python2.3  \
  -I/home/kent-and/local/lib/python2.3/config

> gcc -shared fact.o  fact_wrap.o  -o fact.so
```

Alternatively, make a file `setup.py` **which uses** `distutils`

```
from distutils.core import setup, Extension
setup(name="fact", version="1.0",
          ext_modules=[Extension("fact", ["fact.c",
          "fact_wrap.c"], include_dirs=["."])])
```

To run this file type: `python setup.py build`

The C Factorial Function Used in Python

We can now use the C function:

```
>>> from fact import fact
>>> d = 12
>>> fact(d)
479001600
>>> d = "crash and burn?"
>>> fact(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required
>>>
```

- **The function works as expected when the input is an integer**
- **Inappropriate input data, such as a string, results in a (informative) Python exception**

Preliminary Conclusion

- **It is “relatively easy” to extend Python with C functions**
- **All wrapper functions are “similar”**
- **The writing of such functions are relatively easy, once you have done a couple of such**

Downside:

- **A wrapper function has to be written for each C function we want to access**
- **This wrapper function checks and converts the Python data to appropriate C data, if possible**
- **The writing of such functions is boring**
- **Lots of tools exist that aid the writing of wrapper functions**
- **In fact, the tools let you generate wrapper functions almost automatically, without knowing the Python C-API**

Tools That Aid Wrapping

- SWIG [76], David Beazley et. al.
- Boost.Python [6], Dave Abrahams et. al.
- F2PY [16], Pearu Peterson
- SCXX [73], McMillan Enterprises, Inc.
- Babel [2], LLNL
- SIP [75], Riverbank Computing Ltd.
- SILOON [74], LANL and LLNL

A more extensive list of projects can be found on the SWIG homepage

Topics Covered in this Session

- Brief explanation of the Numeric C-API
- Use of Numeric data structures in a more complicated setting
- Brief description of two tools: F2PY and SWIG
- This will highlight the some common problems and features with wrapper tools

NumPy, revisited

- **NumPy contains a set of efficient linear algebra tools for dense arrays**
- **It contains the usual BLAS and LAPACK routines**
- **NumPy is usually the basis for more special purpose Python packages**
- **The C-APIs of the two NumPy packages, Numeric and Numarray are different, we only describe Numeric**

```
>>> from Numeric import *
>>>
>>> c = array([1,0])
>>> b = sin(c)
>>> A = array(([2,4], [12,2]))
>>>
>>> from LinearAlgebra import *
>>>
>>> e = eigenvalues(A)
>>> print e
[ 8.92820323 -4.92820323]
>>>
>>>
>>> x = solve_linear_equations(A,b)
>>>
>>> print  x
[-0.03824868  0.22949209]
```

Numeric Arrays as Seen from C

An array is of type `PyArrayObject`**, which is a subtype of** `PyObject`

It has:

- `char *data`**, a pointer to the first element of the array**
- `int nd`**, the number of dimensions**
- `int *dimensions`**, the number of elements in each dimension**
- `int *strides`**, the address offset between two data elements along each dimension**

```
typedef struct {
  PyObject_HEAD
  char *data;
  int nd;
  int *dimensions, *strides;
  PyObject *base;
  PyArray_Descr *descr;
  int flags;
} PyArrayObject;
```

Numeric Array Example

Computing the l_2-norm of a 1-dimensional Numeric array:

```
PyArrayObject *array;
int n,i;
double norm_l2 = 0.0;
double tmp = 0.0;

n = array->dimensions[0];

for (i = 0; i < n; i++) {
  tmp = *(double*) (array->data + i*array->strides[0]);
  norm_l2 += tmp*tmp;
}
norm_l2 = sqrt(norm_l2);
```

We have left out a number of safety checks

Also, we want to pass the C data from the Numeric array to a C function (already implemented) that does not use the Numeric C-API

Numeric Array Example, continued

- **We want to use the following C function from Python**
- **It computes the l_2-norm of the plain C array `d`**
- **This function knows nothing about Numeric!**

```
double norm_C(int n, double *d) {
  int i;
  double norm_l2 = 0.0;
  double tmp;
  for (i = 0; i < n; i++) {
    tmp = *d;
    norm_l2 += tmp*tmp;
    d++;
  }
  norm_l2 = sqrt(norm_l2);
  return norm_l2;
}
```

Using the Numeric C-API

Safety checks:

- **Is it a Numeric array?**
- **Does it have the proper dimension and type?**

Casting and calling the C function:

- **Fetch the data, cast to the correct C type, and send the C array to the C function**

Returning to Python:

- **The return value is converted to a Python double and is returned to Python**

Complete Code

```
static PyObject* norm_ext (PyObject* self, PyObject *args) {
  PyArrayObject *array;
  int n;
  double norm_l2 = 0.0;
  double *tmp;

  /* some safety checks */
  if (!PyArg_ParseTuple(args, "O!", &PyArray_Type, &array))
    return NULL;

  if (array->nd != 1 || array->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError,
        "array must be a one-dimensional and of type double");
    return NULL;
  }
  n = array->dimensions[0];          /* find the dimension */
  tmp = (double*) array->data;       /* cast to double* */
  norm_l2 = norm_C(n, tmp);          /* call the C function */

  return PyFloat_FromDouble(norm_l2); /* return a Python double */
}
```

F2PY

F2PY is a "FORTRAN to Python interface generator" with the following features:

- Calling FORTRAN 77/90/95, and C functions from Python
- Accessing FORTRAN 77 COMMON blocks and FORTRAN 90/95 module data from Python
- Calling Python functions from FORTRAN or C (callbacks)
- It support NumPy, both Numeric and Numarray

Author: Pearu Peterson

http://cens.ioc.ee/projects/f2py2e/

FORTRAN CallBack Example

A function f defined in Python is used in Fortran,

$$y = f(x)$$

```
      subroutine evalf(y,x,n, func)
      integer n
Cf2py intent(out)  y
Cf2py intent(in)  x
      real*8 y(n), x(n)
      real*8 func
      external func
      integer i

      do i = 1, n
         y(i) = func(x(i))
      end do
      return
      end
```

```
      real*8 function f1(x)
      real*8 x
      f1 = sin(x) + 8*x
      return
      end
```

```
      call evalf(y, x, n, f1)
```

F2PY - Simple Example

Running F2PY:

```
f2py  -m callback -c callback.f
```

results in the file `callback.so`

This module is used as follows:

```
import callback
import Numeric,math

def f1(x):
  return math.sin(x) + 8*x

x = Numeric.zeros(4)
y = callback.evalf(f1,x)
```

It cannot be any simpler !!

Input and Output

Input and Output in Fortan ($y = f(x)$):

```
      subroutine evalf(y,x,n,func)
        integer n
        real*8 y(n), x(n)
        real*8 func
        external func
```

The array `y` (or `x`) may be input, output or both

In Python we normally want to specify whether it is input or output (good Python style)

```
y = callback.evalf(y,x,f1)
```

or

```
y = callback.evalf(x,f1)
```

Notice that the last example results in the allocation of a new array, which is not wanted if `a` is already made!

Signature File

F2PY employs signature files to adjust the Python interface (written in F90/F95)

Examples of options:

- `intent` **is used to indicate input, output or both(in this case):**

  ```
  real*8 dimension(n),intent(in,out) ::  y
  ```

- `optional` **is used to indicate optional arguments.**
  For instance, `n` **may be determined from** `y`**, if not given:**

  ```
  integer optional,check(shape(y,1)==n),
  depend(y)::n=shape(y,1)
  ```

Many more options!

Use the signature file generated by F2PY as a starting point

Example Signature File

Run F2PY to produce a signature file:

```
f2py -m callback -h callback.pyf callback.f
```

In the signature file we can specify whether it is input, output or both

```
python module callback ! in
    interface  ! in :callback
        subroutine evalf(y,x,n,func) ! in :callback:callback.f
            use evalf__user__routines
            real*8 dimension(n),intent(in,out) :: y
            real*8 dimension(n),intent(in),depend(n) :: x
            integer optional,check(len(y)>=n),depend(y) :: n=len(y)
            external func
        end subroutine evalf
    end interface
end python module callback
```

Specifying Input/Output in the FORTRAN Code

Earlier we saw the following code

```
          subroutine evalf(y,x,n,func)
          integer n,
Cf2py     intent(in,out)  y
Cf2py     intent(in)  x
          real*8 y(n)
          real*8 x(n)
          real*8 func
          external func
          integer i

          ...
```

Here the F2PY directive and FORTRAN comment `Cf2py` specifies that `y` is both input and output, `x` is only input

SWIG

SWIG generates wrapper code for C and C++

- It supports Python, Perl, Tcl, Java and many more languages
- It has a large user and developer base
- It is well documented
- It is more complicated than F2PY “because” C and C++ are more complicated than FORTRAN
- It relies on interface files (equivalent to F2PY signature files)

It has been developed for almost 10 years and supports C and most C++ features, i.e., operator/function overloading, templates (STL), classes and inheritance (even cross-language), . . .

Factorial Example

Remember the factorial function implemented in C:

```
int fact(int i) {
  if (i <= 1) return 1;
  else return i*fact(i-1);
}
```

A corresponding interface file is

```
%module fact // fact is the module name
%{
/* Put headers and other declarations here */
#include <fact.h>
%}
/* The interface definition (e.g. function signatures) */
int fact(int i);
```

Notice that:

- **SWIG directives start with %**
- **The rest is plain C/C++**

Making a Python Module with SWIG

Running SWIG:

```
swig -python  fact.i
```

produces a file `fact_wrap.c`

Both `fact_wrap.c` **and** `fact.c` **are compiled and linked to a shared library** `_fact.so` **(on Unix):**

```
> gcc -c -fpic  fact_wrap.c fact.c -I. -DHAVE_CONFIG_H  \
-I/local/include/python2.3 -I/local/lib/python2.3/config
> gcc -shared fact.o  fact_wrap.o  -lswigpy -L/local/lib/ -o _fact.so
```

Additionally, a Python module `fact.py` **is made, which is Python layer on top of** `_fact.so`

Use in Python

```
>>> from fact import fact
>>> f1 = fact(12)
>>> print "fact(12)=", f1
fact(12)= 479001600
>>>
>>> f2 = fact("crash and burn ?")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: a long is expected
```

This is almost as simple as with F2PY!!

A “Problem” with C/C++

Consider the function:

```
void foo(int n, double* c, int m, double *d);
```

What is `c` (or `d`)?

- **A pointer to a double?**
- **A pointer to a double array with length `n` (or `m`)?**

→ There is no “easy” correspondence between C arrays and Numeric arrays

Additionally, we do not know whether `c` and `d` are input, output or both?

A Vector Implemented in C++

Many numerical libraries have arrays as fundamental building blocks

- **We will now describe how a simple C++ implementation of a vector can be interfaced by SWIG**
- **We will show the details of a mapping between this Vector and a Numeric array**
- **We will subclass the C++ Vector class in Python**
- **We will notice that the cross language inheritance provides a nice way to construct certain types of callbacks**

(It is easy to extend this to n-dimensional arrays)

Comments About the Interface

- **Public data is wrapped, private or protected data is not**
- **There are some ambiguities, e.g,**
  - **C/C++ has both float and double, Python only has double**
  - **C/C++ distinguish between const and non-const, Python does not have const**
- **In case of ambiguity, SWIG only wrappes the first occurrence**

SWIG has many directives for adjusting the interface, some examples are

- `%rename` **for renaming e.g. functions**
- `%extend` **for extending the C++ with e.g. helper functions**
- `%ignore` **for ignoring problematic or unwanted things**

A Vector Implemented in C++ and Its Interface

```
// A Vector Class in C++
class Vec{
  int dim;        // protected data or
  double *data;   // functions are not
                  // wrapped,
public:
  Vec();
  Vec(int d);
  ~Vec(){}

  // some std functions
  void redim(int d);
  int size () const {return dim;}

  // some operators
  const double operator()(int i) const;
  double& operator()(int i);
  Vec& operator=(const Vec& vec);

  // some virtual functions
  virtual void fill(double value);
  virtual double norm();
};
```

```
//The interface file  Vec.i
%module Vec
%{
#include <Vec.h>
%}

// rename to the corresponding
// Python operator
%rename(__getitem__) Vec::operator();
%include Vec.h // wrap the whole class

// We can extend the class with some helper
// functions by using the extend directive.
// Here we add the __setitem__ operator
// which was ignored due to the ambiguity
// of const/non-const operator ()

%extend Vec {
  void __setitem__ (int i, double a)
  {
    (*self)(i) = a;
  }
};
```

A Vector to Numeric Filter

- There are many ways to construct mappings between two data structures
- SWIG has a mechanism: Typemaps (which we will not use)
- We are working with large arrays $\longrightarrow$ the programmer should explicitly invoke the mapping
- It is implemented as a class, where the constructor provides the necessary initialization (more on this later)
- We copy the data for safety, although it is possible to pass pointers

```
class Vec2NumPy{
  public:
  Vec2NumPy();

 void numpy2vec(PyObject* array, Vec& vec);
 PyObject* vec2numpy(const Vec& vec);
};
```

A Vector to Numeric Filter, continued

```
void Vec2NumPy:: numpy2vec(PyObject* py_obj, Vec& vec)
{
  if (!PyArray_Check(py_obj)) {
    PyErr_SetString(PyExc_TypeError, "Not a NumPy array");
    return;
  } else {
    PyArrayObject* array = (PyArrayObject*)
      PyArray_ContiguousFromObject(py_obj, PyArray_DOUBLE, 1,1);
    if (array == NULL) {
      PyErr_SetString(PyExc_TypeError,
                      "The NumPy array is not a vector (one dim.)");
      return;
    }
    int size = array->dimensions[0];
    vec.redim(size);
    for (int i=0; i< size; i++) {
      vec(i) = *(double*)(array->data + array->strides[0]*i);
    }
  }
}
```

A Vector to Numeric Filter, continued

```
Vec2NumPy:: Vec2NumPy () {
  /* The Numeric module must be initialized before use */
  import_array();
}

PyObject* Vec2NumPy:: vec2numpy(const Vec& vec)
{
  int dim = vec.size();
  PyArrayObject* array =  (PyArrayObject*) \
          PyArray_FromDims(1, &dim, PyArray_DOUBLE);
  for (int i=0; i< vec.size(); i++) {
    *(double*)(array->data + array->strides[0]*i) = vec(i);
  }
  return PyArray_Return(array);
}
```

Example of Use in Python

```
>>> from Vec import *
>>> from Numeric import *
>>>
>>> v = Vec(3)
>>> v[0] = 0; v[1] = 1; v[2] = -1
>>> print "v[0]=",  v[0]
v[0]= 0.0
>>> print "v[1]=",  v[1]
v[1]= 1.0
>>> print "v[2]=",  v[2]
v[2]= -1.0
>>>
>>> print "norm of v", v.norm()
norm of v 1.41421356237
>>>
>>> filter = Vec2NumPy()
>>> a = filter.vec2numpy(v)
>>>
>>> print "numpy array ", a
numpy array  [ 0.  1. -1.]
```

Some Computations on the Vector

Other classes employ `Vec`:

```
class DoSomeComputations{
  Vec* v;

  public:
    DoSomeComputations(){}
    void attach(Vec& v_);
    void compute();
};
```

Example of use:

```
void DoSomeComputations:: compute() {
  for (int i=1; i<= 2; i++) {
    v->fill(1.0/i);
  }
}
```

What happens in `compute` if we subclass `Vec` in Python?
(`compute` knows nothing about Python)

CallBack Through Inheritance

Director classes provide the mechanism:

```
%module(directors="1") Vec
%{
#include <Vec.h>
%}

%feature("director") Vec;
%include Vec.h

%include DoSomeComputations.i
```

Virtual functions in `Vec` can now be redefined in Python

(Notice that director classes in SWIG are new and are considered experimental)

A Sub Class of Vec in Python

```
>>> from Vec import *; import _Vec
>>> class VecPy(Vec):
...   def  __init__(self, n):
...     Vec.__init__(self,n) ;  self.counter = 0
...   def fill(self, a):
...     print "Inside VecPy::fill"
...     _Vec.Vec_fill(self,a)
...     self.counter += 1
...
>>> v = VecPy(3)
>>> v[0] = 0; v[1] = 1; v[2] = -1
>>>
>>> computer = DoSomeComputations()
>>> computer.attach(v)
>>> computer.compute()
Inside VecPy::fill
Inside VecPy::fill
>>>
>>> print v.counter
2
>>> print v.norm()
0.866025403784
```

Various Packages for Scientifi c Computing

Kent Andre Mardal

PyPkg

- Intention: Distribute software
- URL: http://home.simula.no/˜arvenk/pypkg
- Author: Arve Knudsen
- License: GPL-2

PyPkg Usage

```
python para06-installer.py
```

This will install:

- Dolfin, FFC, FIAT
- GiNaC, Swiginac
- Instant
- MayaVi, Vtk
- PETSc
- PyCC
- PySE
- Trilinos

GiNaC

- Intention: Computer Algebra System
- URL: www.ginac.de
- Authors: C. Bauer, A. Frink, R. Kreckel, and J. Vollinga
- License: GPL
- GiNaC is a C++ library
- GiNaC has strong support for polynomials
- Among other things, GiNaC supports differentiation, integration and code generation

Swiginac

- Intention: Python interface to GiNaC
- URL: http://swiginac.berlios.de/
- Authors: O. Skavhaug and O. Certik
- License: Open
- Swiginac provides a nice interface to GiNaC from Python
- Swiginac gives a seemless conversion between standard Python datatypes and GiNaC datatypes

Swiginac Usage: Differentiation and integration

```
from swiginac import *

x = symbol('x'); y = symbol('y')

f = sin(x*x*y)
print "f = ", f

# code generation
print "f in C ", f.printc()
print "f in LaTeX ", f.printlatex()

# differentiation
dfdx = diff(f,x)
print "df/dx = ", dfdx

# Taylor series
taylor_f = f.series(x==0, 10)
print "taylor expansion of f " , taylor_f

# integration
g = pow(x,9)*(1-x)
intg = eval_integ(integral(x,0,1,g))
print "integral of g from 0 to 1 ", intg
```

Inlining Tools

There are several tools that enable inlining of C/C++/Fortran code in Python

- **Weave (http://www.scipy.org/), C/C++ inlining**
- **PyInline (http://pyinline.sourceforge.net), C/C++ inlining**
- **Instant (http://pyinstant.sourceforge.net), C/C++ inlining**
- **F2PY (http://cens.ioc.ee/projects/f2py2e/), Fortran inlining**

Instant

- Intention: Inlining of plain C/C++ code
- URL: pyinstant.sourceforge.net
- Authors: M. Westlie and K.-A. Mardal
- License: Open
- Instant uses SWIG to generate wrapper code
- Instant uses Distutils to compile the wrapper code and create a shared library that can be imported in Python

Instant Usage: Inlining of a simple function

```
import Instant

c_code = """
double sum(double a, double b){
  return a+b;
}
"""

ext = Instant.Instant()
ext.create_extension(code=c_code,
                     module='test1_ext')

from test1_ext import sum
a = 3.7
b = 4.8
c = sum(a,b)
print "The sum of %g and %g is %g"% (a,b,c)
```

- **Instant also supports C/C++-functions with C-arrays, C++ classes etc.**

Linear Algebra Tools

Several Linear Algebra tools have Python interfaces

Dense Matrix Tools:

- Numeric

Sparse Matrix Tools:

- Trilinos (http://software.sandia.gov/trilinos)
- PETSc (http://www-unix.mcs.anl.gov/petsc/petsc-as)
- Hypre (http://acts.nersc.gov/hypre)

Trilinos

- Intention: Parallel framework for large scale linear/non-linear algebra problems
- URL: http://software.sandia.gov/trilinos
- Authors: M. Heroux and many more
- License: LGPL

Trilinos has

- Dense Matrices tools
- Standard Krylov solvers, preconditioners
- Algebraic Multigrid
- Eigenvalue/Eigenvector computations
- Nonlinear Solvers
- and much more

Trilinos Usage: Solving a Poisson equation

```
import ML, Triutils, AztecOO, Epetra

nx = 100; ny = 100
Comm = Epetra.PyComm()
Gallery = Triutils.CrsMatrixGallery("laplace_2d", Comm)
Gallery.Set("nx", nx); Gallery.Set("ny", ny)
Matrix = Gallery.GetMatrix()
LHS = Gallery.GetStartingSolution()
RHS = Gallery.GetRHS()

MLList = { "max levels" : 3, "output" : 10,
           "smoother: type" : "symmetric Gauss-Seidel",
           "aggregation: type" : "Uncoupled" }

Prec = ML.MultiLevelPreconditioner(Matrix, False)
Prec.SetParameterList(MLList)
Prec.ComputePreconditioner()

Solver = AztecOO.AztecOO(Matrix, LHS, RHS)
Solver.SetPrecOperator(Prec)
Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
Solver.SetAztecOption(AztecOO.AZ_output, 16)
Solver.Iterate(1550, 1e-5)
```

PySE

- Intention: Finite Difference Tools in Python
- URL: http://pyfdm.sourceforge.net/
- Author: Å. Ødegård
- License: Open
- PySE provides a parallel framework for finite difference methods
- PySE gives a high-level environment for working with stencils

PySE Usage: Solving a Heat Equation

```
from pyFDM import *

def neumannfunc(x,y): return sin(x)*cos(y)

def initialfunc(x,y): return sin(x)*cos(y)

g = Grid(domain=([0,1,[0,1]),division=(100,100))
u = Field(g)
t = 0; dt = T/n;
A = StencilSet(g)
innerstencil = Identity(g.nsd) + dt*Laplace(g)
innerind = A.addStencil(innerstencil, g.innerPoints())
A += createNeumannBoundary(innerstencil, g, neumannfunc)
u.fill(initialfunc)
for t < T:
    u = A(u)
    t += dt
plot(u)
```

Finite Element Tools

- FIAT, automates the generation of finite elements
- FFC, automates the evaluation of variational forms
- SyFi, a tool for defining finite elements and variational forms
- Dolfin, a finite element framework
- PyCC, a finite element framework

FIAT

- Intention: To automate the generation of finite elements
- URL: http://www.fenics.org/fiat/
- Author: R. C. Kirby
- License: LGPL
- FIAT currently supports Lagrange, Hermite, Crouzeix-Raviart, Raviart-Thomas and Nedelec elements

FIAT Usage: The Lagrange Element

```
from FIAT.Lagrange import *
from FIAT.shapes import *

element = Lagrange(TRIANGLE, 1)
P1 = element.function_space()

v0 = P1[0]
v1 = P1[1]
v2 = P1[2]

points = make_lattice(TRIANGLE, 10)

x = points[0]

for x in points:
    print v0(x)

for x in points:
    print v0(x) + v1(x) + v2(x)

print [v0(x) + v1(x) + v2(x) for x in points]

values = [v0(x) for x in points]
```

FFC

- Intention: Automatic and efficient evaluation of general multilinear forms
- URL: http://www.fenics.org/ffc/
- Author: A. Logg
- License: GPL
- FFC works as a compiler for multilinear forms by generating code (C or C++)
- The generated code is as efficient as hand-optimized code

FFC Usage: A Poisson Equation

```
from ffc import *


element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
U = TrialFunction(element)
f = Function(element)

a = dot(grad(v), grad(U))*dx
L = v*f*dx
a.compile()
```

Dolfin

- Intention: A Finite Element Framework
- URL: http://www.fenics.org/dolfin/
- Authors: J. Hoffman, J. Jansson, A. Logg and G. N. Wells
- License: GPL
- Large library with PDE and ODE solvers

Dolfin Usage: Solving a Poisson Equation

```
from dolfin import *

class Source(Function):
    def eval(self, point, i): return point.y + 1.0

class SimpleBC(BoundaryCondition):
    def eval(self, value, point, i):
        if point.x == 0.0 or point.x == 1.0: value = 0.0

f = Source(); bc = SimpleBC()
mesh = UnitSquare(10, 10)

forms = import_formfile("Poisson2D.form")

a = forms.Poisson2DBilinearForm()
L = forms.Poisson2DLinearForm(f)

A = Matrix()
b = Vector()
assemble(a, L, A, b, mesh, bc)

x = Vector()
solver = KrylovSolver()
solver.solve(A, x, b)
```

SyFi

- Intention: Ease the definition of finite elements and their usage by symbolic mathematics
- URL: syfi.sourceforge.net
- Author: K.-A. Mardal
- License: GPL
- SyFi relies on GiNaC and Swiginac
- SyFi supports the Lagrange, Hermite, Nedelec, Raviart-Thomas, Crouzeix-Raviart elements
- SyFi supports differentiation, integration etc of finite elements functions/polynomials over polygons
- SyFi/Swiginac/GiNaC have tools for C++ code generation

SyFi Usage: Element matrix for Poisson equation

```
from swiginac import *
from SyFi import *

triangle = ReferenceTriangle()

fe = LagrangeFE()
fe.set(3)
fe.set(triangle)
fe.compute_basis_functions()

for i in range(0,fe.nbf()):
  for j in range(0,fe.nbf()):
    integrand = inner(grad(fe.N(i)),grad(fe.N(j)))
    Aij = triangle.integrate(integrand)
    print "A(%d,%d)="%(i,j), Aij
```

SyFi Usage: The Jacobian of a nonlinear PDE

```
from swiginac import *
from SyFi import *

triangle = ReferenceTriangle()

fe = LagrangeFE()
fe.set(3)
fe.set(triangle)
fe.compute_basis_functions()

us = symbolic_matrix(1,fe.nbf(), "u")
u = 0
for j in range(0,fe.nbf()):
    u += us.op(j)*fe.N(j)

for i in range(0,fe.nbf()):

  fi = inner(grad(u*u), grad(fe.N(i)))
  Fi = triangle.integrate(fi)

  for j in range(0,fe.nbf()):
    uj = us.op(j)
    Jij = diff(Fi, uj)
    print "J(%d,%d)="%(i,j), Jij
```

PyCC

- Intention: Finite Element Framework
- URL: http://folk.uio.no/skavhaug/heart_simulations.html
- Author: G. Lines, K.-A. Mardal, O. Skavhaug, G. Staff, Å. Ødegård
- License: Soon to be open
- PyCC is a library with PDE and ODE solvers
- PyCC is particularly strong on computations concerning the electrical activity of the heart

PyCC Usage: Solving a Poisson equation

```
def f(x,y): return 2*pi*pi*cos(pi*x)*cos(pi*y)

mesh = Mesh(os.path.join(common.DataDir, "box", "box2D.xml.gz"))
matrix_factory = MatrixFactory(mesh)

A = matrix_factory.computeStiffnessMatrix()
b = zeros(A.n,  typecode='d')
for i in range(0, len(b)):
    b[i] = f(x[i], y[i])

M = matrix_factory.computeMassMatrix()
b = M*b

boundary_ind = boundary(matrix_factory.idof, mesh)
(A,BC,C) = dirichlet_boundarycondition(A, boundary_ind)
B = FastMatPrec(A)

u_bc = zeros(len(b), typecode='d')
for i in boundary_ind:
    u_bc[i] = exact(x[i], y[i])
b = BC*b - C*u_bc

u = u_bc.copy()
u = precondconjgrad(B, A, u, b, 10E-6, True)
```

References

[1] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. Numerical Python. *http://www.pfdubois.com/numpy/*.

[2] Babel software package. *http://www.llnl.gov/CASC/components/babel.html*.

[3] D. Beazley. *Python Essential Reference*. SAMS, 2nd edition, 2001.

[4] Biggles package. *http://biggles.sourceforge.net*.

[5] Blt software package. *http://blt.sourceforge.net*.

[6] Boost.Python software package. *http://www.boost.org*.

[7] M. C. Brown. *Python, The Complete Reference*. McGraw-Hill, 2001.

[8] X. Cai and H. P. Langtangen. *Parallelizing PDE solvers using the Python programming language*, pages 295–325. Lecture Notes in Computational Science and Engineering. Springer, 2006.

[9] X. Cai, H. P. Langtangen, and H. Moe. On the performance of the Python programming language for serial and parallel scientifi c computations. *Sci-entific Programming*, 13(1):31–56, 2005.

[10] ChomboVis package. *http://seesar.lbl.gov/anag/chombo/chombovis.html*.

[11] Coin - Inventor implementation. *http://www.coin3d.org*.

[12] Disipyl - Dislin Python interface. *http://kim.bio.upenn.edu/˜pmagwene/disipyl.html*.

[13] Dislin package. *http://www.linmpi.mpg.de/dislin*.

[14] Dolfin software package. *http://www.fenics.org/dolfin/*.

[15] D. Beazley et. al. Swig 1.3 Development Documentation. *http://www.swig.org/doc.html*.

[16] F2PY software package. *http://cens.ioc.ee/projects/f2py2e*.

[17] FFC software package. *http://www.fenics.org/ffc/*.

[18] FIAT software package. *http://www.fenics.org/fiat/*.

[19] Gd package. *http://www.boutell.com/gd*.

[20] Gd Python interface. *http://newcenturycomputers.net/projects/gdmodule.html*.

[21] Roman Geus. Pysparse - handling sparse matrices in python. *http://people.web.psi.ch/geus/pyfemax/pysparse.html*.

[22] GiNaC software package. *http://www.ginac.de*.

[23] Gmt package. *http://gmt.soest.hawaii.edu*.

[24] Gmt Python interface. *http://www.cdc.noaa.gov/˜jsw/python/gmt*.

[25] Gnu Plotutils package. *http://www.gnu.org/software/plotutils*.

[26] Gnuplot package. *http://www.gnuplot.info*.

[27] GraphicsMagick package. *http://www.graphicsmagick.org*.

[28] D. Harms and K. McDonald. *The Quick Python Book*. Manning, 1999.

[29] M. L. Hetland. *Practical Python*. APress, 2002.

[30] S. Holden. *Python Web Programming*. New Riders, 2002.

[31] Instant software package. *http://syfi.sf.net*.

[32] Introductory material on python. *http://www.python.org/doc/Intros.html*.

[33] IPython software package. *http://ipython.scipy.org*.

[34] K. Hinsen and H. P. Langtangen and O. Skavhaug and Å. Ødegård. Using BSP and Python to simplify parallel programming. *Future Generation Computer Systems*, 2004. In press.

[35] H. P. Langtangen. Slides collection: Scripting for Computational Science. *http://www.ifi.uio.no/˜inf3330/lecsplit/index.html*.

[36] H. P. Langtangen. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, 2004.

[37] H. P. Langtangen. Scripting Resources. http://www.ifi.uio.no/˜inf3330/scripting/doc.html, 2004.

[38] M. Lutz. *Programming Python*. O'Reilly, second edition, 2001.

[39] M. Lutz and D. Ascher. *Learning Python*. O'Reilly, 1999.

[40] A. Martelli. *Python in a Nutshell*. O'Reilly, 2003.

[41] A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, 2002.

[42] Matplotlib package. *http://matplotlib.sourceforge.net*.

[43] MayaVi package. *http://mayavi.sourceforge.net*.

[44] D. Mertz. *Text Processing in Python*. McGraw-Hill, 2003.

[45] Mplayer package. *http://www.mplayerhq.hu*.

[46] Ncarg package.
http://ngwww.ucar.edu/ng/download.html.

[47] Ncarg Python interface.
http://www.cdc.noaa.gov/people/jeffrey.s.whitaker/python/ncarg.

[48] Numerical Python software package.
http://sourceforge.net/projects/numpy.

[49] OpenDX package. *http://www.opendx.org*.

[50] Pgplot package. *http://www.astro.caltech.edu/ tjp/pgplot*.

[51] Piddle package. *http://piddle.sourceforge.net*.

[52] M. Pilgrim. *Dive Into Python*. *http://diveintopython.org/*, 2002.

[53] Pivy - Inventor Python interface. *http://pivy.tammura.at*.

[54] Plplot package. *http://plplot.sourceforge.net/index.html*.

[55] Pmw - python megawidgets. *http://pmw.sourceforge.net*.

[56] Ppgplot - Pgplot Python interface. *http://efault.net/npat/hacks/ppgplot*.

[57] Py4dat package. *http://pydvt.sourceforge.net*.

[58] PyCC software package. *http://folk.uio.no/skavhaug/heart_simulations.html*.

[59] Pychart package. *http://www.hpl.hp.com/personal/Yasushi_Saito/pychart*.

[60] Pymat Python-Matlab interface. *http://claymore.engineer.gvsu.edu/˜steriana/Python*.

[61] PyOpenDX - OpenDX Python interface. *http://people.freebsd.org/˜rhh/py-opendx*.

[62] PyOpenGL - OpenGL Python interface. *http://pyopengl.sourceforge.net*.

[63] PySE software package. *http://pyfdm.sf.net*.

[64] Python-gnuplot interface. *http://gnuplot-py.sourceforge.net*.

[65] Python Imaging Library. *http://www.pythonware.com/products/pil/index.htm*.

[66] Python Vtk manipulation. *http://cens.ioc.ee/projects/pyvtk*.

[67] Pythonmagick - GraphicsMagick Python interface. *http://www.procoders.net/moinmoin/PythonMagick*.

[68] PyX package. *http://pyx.sourceforge.net*.

[69] R package. *http://www.r-project.org*.

[70] Rpy - R Python interface. *http://rpy.sourceforge.net*.

[71] Scientifi cPython software package. *http://starship.python.net/crew/hinsen*.

[72] SciPy software package. *http://www.scipy.org*.

[73] SCXX software package. *http://davidf.sjsoft.com/mirrors/mcmillan-inc/scxx.html*.

[74] SILOON software package. *http://acts.nersc.gov/siloon/main.html*.

[75] SIP software package. *http://www.riverbankcomputing.co.uk/sip/*.

[76] SWIG software package. *http://www.swig.org*.

[77] Swiginac software package. *http://swiginac.berlios.de/*.

[78] SyFi software package. *http://syfi.sf.net*.

[79] Trilinos software package. *http://software.sandia.gov/trilinos*.

[80] G. van Rossum and F. L. Drake. Extending and Embedding the Python Interpreter. *http://docs.python.org/ext/ext.html*.

[81] G. van Rossum and F. L. Drake. Python Library Reference. *http://docs.python.org/lib/lib.html*.

[82] G. van Rossum and F. L. Drake. Python Tutorial. *http://docs.python.org/tut/tut.html*.

[83] The Vaults of Parnassus. *http://www.vex.net/parnassus*.

[84] Visual Python. *http://www.vpython.org*.

[85] Vtk package. *http://www.vtk.org*.