

Python for Scientific and High Performance Computing

SC09

Portland, Oregon, United States

Monday, November 16, 2009 1:30PM - 5:00PM

<http://www.mcs.anl.gov/~wscullin/python/tut/sc09>

Introductions

Your presenters:

- William R. Scullin
 - wscullin@alcf.anl.gov
- James B. Snyder
 - jbsnyder@northwestern.edu
- Nick Romero
 - naromero@alcf.anl.gov
- Massimo Di Pierro
 - mdipierro@cs.depaul.edu



Overview

We seek to cover:

- Python language and interpreter basics
- Popular modules and packages for scientific applications
- How to improve performance in Python programs
- How to visualize and share data using Python
- Where to find documentation and resources

Do:

- Feel free to interrupt
 - the slides are a guide - we're only successful if you learn what you came for; we can go anywhere you'd like
- Ask questions
- Find us after the tutorial

About the Tutorial Environment

Updated materials and code samples are available at:

<http://www.mcs.anl.gov/~wscullin/python/tut/sc09>

we suggest you retrieve them before proceeding. They should remain posted for at least a calendar year.

You should have login instructions, a username and password for the tutorial environment on the paper on your slip. Accounts will be terminated no later than 6:30PM USPT today. Do not leave any code or data on the system you would like to keep.

Your default environment on the remote system is set up for this tutorial, though the downloadable live dvd should provide a comparable environment.

Outline

1. Introduction

- Introductions
- Tutorial overview
- Why Python and why in scientific and high performance computing?
- Setting up for this tutorial

2. Python basics

- Interpreters
- data types, keywords, and functions
- Control Structures
- Exception Handling
- I/O
- Modules, Classes and OO

3. SciPy and NumPy: fundamentals and core components

4. Parallel and distributed programming

5. Performance

- Best practices for pure Python + NumPy
- Optimizing when necessary

6. Real world experiences and techniques

7. Python for plotting, visualization, and data sharing

- Overview of matplotlib
- Example of MC analysis tool

8. Where to find other resources

- There's a Python BOF!

9. Final exercise

10. Final questions

11. Acknowledgments



- Dynamic programming language
- Interpreted & interactive
- Object-oriented
- Strongly introspective
- Provides exception-based error handling
- Comes with "Batteries included" (extensive standard libraries)
- Easily extended with C, C++, Fortran, etc...
- Well documented (<http://docs.python.org/>)

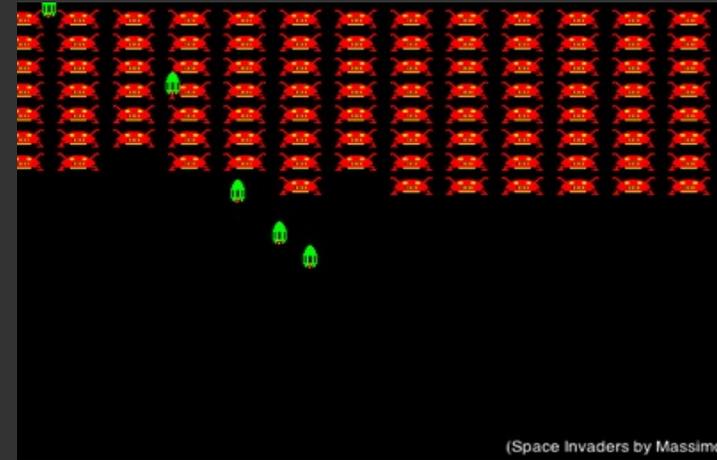
Easy to learn

```
#include "iostream"
#include "math"
int main(int argc, char** argv)
{
    int n = atoi(argv[1]);
    for(int i=2;
        i<(int) sqrt(n);
        i++)
    {
        p=0;
        while(n % i)
        {
            p+=1;
            n/=i;
        }
        if (p)
            cout << i << "^"
                << p << endl;
    }
    return 0;
}
```

```
import math, sys
n = int(sys.argv[1])
for i in range(2, math.sqrt(n)):
    p=0
    while n % i:
        (p,n) = (p+1, n/i)
    if p:
        print i, '^', p
```

Now try do this in C++

```
from Tkinter import Tk, Label, Canvas, PhotoImage
import math, time
root = Tk()
canvas, aliens, missiles = Canvas(root,width=800,height=400,bg='white'), {}, {}
canvas.pack()
i1, i2 = PhotoImage(format='gif',file="alien.gif"), PhotoImage(format='gif',file="missile.gif")
for x,y,p in [(100+40*j,160-20*i,100*i) for i in range(8) for j in range(15)]:
    aliens[canvas.create_image(x,y,image=i1)]=p
canvas.bind('<Button-1>', lambda e: missiles.update({canvas.create_image(e.x,390,image=i2):10}))
while aliens:
    try:
        for m in missiles:
            canvas.move(m,0,-5)
            if canvas.coords(m)[1]<0:
                score -= missiles[m];
                canvas.delete(m); del missiles[m]
        for a in aliens:
            canvas.move(a,2.0*math.sin(time.time()),0)
            p = canvas.coords(a)
            items = canvas.find_overlapping(p[0]-5,p[1]-5,p[0]+5,p[1]+5)
            for m in items[1:2]:
                canvas.delete(a); del aliens[a]; canvas.delete(m); del missiles[m]
        time.sleep(0.02); root.update()
    except: pass
```



- only 24 lines of python code
- uses standard Python libraries.

Why Use Python for Scientific Computing?

- "Batteries included" + rich scientific computing ecosystem
- Good balance between computational performance and time investment
 - Similar performance to expensive commercial solutions
 - Many ways to optimize critical components
 - Only spend time on speed if really needed
- Tools are mostly open source and free (many are MIT/BSD license)
- Strong community and commercial support options.
- No license management

Science Tools for Python

Large number of science-related modules:

General

NumPy
SciPy

GPGPU Computing

PyCUDA
PyOpenCL

Parallel Computing

PETSc
PyMPI
Pypar
mpi4py

Wrapping

C/C++/Fortran
SWIG
Cython
ctypes

Plotting & Visualization

matplotlib
VisIt
Chaco
MayaVi

AI & Machine Learning

pyem
ffnet
pymorph
Monte
hcluster

Biology (inc. neuro)

Brian
SloppyCell
NIPY
PySAT

Molecular & Atomic Modeling

PyMOL
Biskit
GPAW

Geosciences

GIS Python
PyClimate
ClimPy
CDAT

Bayesian Stats

PyMC

Optimization

OpenOpt

Symbolic Math

SymPy

Electromagnetics

PyFemax

Astronomy

AstroLib
PySolar

Dynamic Systems

Simpy
PyDSTool

Finite Elements

SfePy

For a more complete list: http://www.scipy.org/Topical_Software

Please login to the Tutorial Environment

Let the presenters know if you have any issues.

Start an iPython session:

```
santaka:~> wscullin$ ipython
Python 2.6.2 (r262:71600, Sep 30 2009, 00:28:07)
[GCC 3.3.3 (SuSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
```

```
IPython 0.9.1 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object'. ?object also works, ?? prints
more.
```

```
In [1]:
```

Python Basics

- Interpreter
- Built-in Types, keywords, functions
- Control Structures
- Exception Handling
- I/O
- Modules, Classes & OO

Interpreters

- CPython Standard python distribution
 - What most people think of as "python"
 - highly portable
 - <http://www.python.org/download/>
 - We're going to use 2.6.2 for this tutorial
 - The future is 3.x, the future isn't here yet
- iPython
 - A user friendly interface for testing and debugging
 - <http://ipython.scipy.org/moin/>

Other Interpreters You Might See...

- Unladen Swallow
 - Blazing fast, uses llvm and in turn may compile!
 - x86/x86_64 only really
 - Sponsored by Google
 - <http://code.google.com/p/unladen-swallow/>
- Jython
 - Python written in Java and running on the JVM
 - <http://www.jython.org/>
 - performance is about what you expect
- IronPython
 - Python running under .NET
 - <http://www.codeplex.com/IronPython>
- PyPy
 - Python in... Python
 - No where near ready for prime time
 - <http://codespeak.net/pypy/dist/pypy/doc/>

CPython Interpreter Notes

- Compilation affects interpreter performance
 - Precompiled distributions aim for compatibility and as few irritations as possible, not performance
 - compile your own or have your systems admin do it
 - same note goes for most modules
 - Regardless of compilation, you'll have the same bytecode and the same number of instructions
 - Bytecode is portable, binaries are not
 - Linking against shared libraries kills portability
- Not all modules are available on all platforms
 - Most are not OS specific (>90%)
 - x86/x86_64 is still better supported than most

A note about distutils and building modules

Unless your environment is very generic (ie: a major linux distribution under x86/x86_64), and even if it is, manual compilation and installation of modules is a very good idea.

Distutils and setuptools often make incorrect assumptions about your environment in HPC settings. Your presenters generally regard distutils as evil as they cross-compile a lot.

If you are running on PowerPC, IA-64, Sparc, or in an uncommon environment, let module authors know you're there and report problems!

Built-in Numeric Types

int, float, long, complex - different types of numeric data

```
>>> a = 1.2 # set a to floating point number
```

```
>>> type(a)
<type 'float'>
```

```
>>> a = 1 # redefine a as an integer
```

```
>>> type(a)
<type 'int'>
```

```
>>> a = 1e-10 # redefine a as a float with scientific notation
```

```
>>> type(a)
<type 'float'>
```

```
>>> a = 1L # redefine a as a long
```

```
>>> type(a)
<type 'long'>
```

```
>>> a = 1+5j # redefine a as complex
```

```
>>> type(a)
<type 'complex'>
```

Gotchas with Built-in Numeric Types

Python's int and float can become as large in size as your memory will permit, but ints will be automatically typed as long. The built-in long datatype is very slow and best avoided.

```
>>> a=2.0**999
```

```
>>> a
```

```
5.3575430359313366e+300
```

```
>>> import sys
```

```
>>> sys.maxint
```

```
2147483647
```

```
>>> a>sys.maxint
```

```
True
```

```
>>> a=2.0**9999
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
OverflowError: (34, 'Result too large')
```

```
>>> a=2**9999
```

```
>>> a-((2**9999)-1)
```

```
1L
```

Gotchas with Built-in Numeric Types

Python's int and float are not decimal types

- IEEE 754 compliant (<http://docs.python.org/tutorial/floatingpoint.html>)
- math with two integers always results in an integer

```
>>> a=1/3 # no type coercion, integer division
>>> a
0
```

```
>>> a=1/3.0 # binary op with int and float -> int coerced to float
>>> a
0.33333333333333331
```

```
>>> a=1.0/3.0 # float division
>>> a
0.33333333333333331
```

```
>>> 0.3
0.29999999999999999 # thanks binary fractions!
```

```
>>> a=1.0/10
>>> a
0.10000000000000001
```

NumPy Numeric Data Types

NumPy covers all the same numeric data types available in C/C++ and Fortran as variants of int, float, and complex

- all available signed and unsigned as applicable
- available in standard lengths
- floats are double precision by default
- generally available with names similar to C or Fortran
 - ie: long double is longdouble
- generally compatible with Python data types

Built-in Sequence Types

str, unicode - string types

```
>>> s = 'asd'
>>> u = u' fgh' # prepend u, gives unicode string
>>> s[1]
's'
```

list - mutable sequence

```
>>> l = [1,2,'three'] # make list
>>> type(l[2])
<type 'str'>

>>> l[2] = 3; # set 3rd element to 3
>>> l.append(4) # append 4 to the list
```

tuple - immutable sequence

```
>>> t = (1,2,'four')
```

Built-in Mapping Type

dict - match any immutable value to an object

```
>>> d = {'a' : 1, 'b' : 'two'}
```

```
>>> d['b'] # use key 'b' to get object 'two'
```

```
'two'
```

redefine b as a dict with two keys

```
>>> d['b'] = {'foo' : 128.2, 'bar' : 67.3}
```

```
>>> d
```

```
{'a': 1, 'b': {'bar': 67.299999999999997, 'foo':  
128.19999999999999}}
```

index nested dict within dict

```
>>> d['b']['foo']
```

```
128.19999999999999
```

any immutable type can be an index

```
>>> d['b'][(1, 2, 3)] = 'numbers'
```

Built-in Sequence & Mapping Type Gotchas

Python lacks C/C++ or Fortran style arrays.

- Best that can be done is nested lists or dictionaries
 - Tuples, being immutable are a bad idea
- You have to be very careful on how you create them
- Growing these types will cost performance (minimal pre-allocation)
- NumPy provides real n-dimensional arrays with low overhead

Python requires that you correctly indent your code.

- Only applies to indentation
- Will help keep your code readable
- Use 4 spaces for tabs, and you won't have any problems (if you indent correctly)

If you have further questions, see PEP 8:

<http://www.python.org/dev/peps/pep-0008/>

Control Structures

if - compound conditional statement

```
if (a and b) or (not c):  
    do_something()  
elif d:  
    do_something_else()  
else:  
    print "didn't do anything"
```

while - conditional loop statement

```
i = 0  
while i < 100:  
    i += 1
```

Control Structures

for - iterative loop statement

```
for item in list:  
    do_something_to_item(item)
```

start = 0, stop = 10

```
>>> for element in range(0,10):  
...     print element,  
0 1 2 3 4 5 6 7 8 9
```

start = 0, stop = 20, step size = 2

```
>>> for element in range(0,20,2):  
...     print element,  
0 2 4 6 8 10 12 14 16 18
```

Generators

Python makes it very easy to write functions you can iterate over- just use yield instead of return at the end of functions

```
def squares(lastterm):  
    for n in range(lastterm):  
        yield n**2
```

```
>>> for i in squares(4): print i
```

```
...
```

```
0
```

```
1
```

```
4
```

```
9
```

```
16
```

List Comprehensions

List Comprehensions are powerful tool, replacing Python's lambda function for functional programming

- **syntax:** `[f(x) for x in generator]`
- you can add a conditional `if` to a list comprehension

```
>>> [i for i in squares(10)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> [i for i in squares(10) if i%2==0]  
[0, 4, 16, 36, 64]
```

```
>>> [i for i in squares(10) if i%2==0 and i%3==1]  
[4, 16, 64]
```

Exception Handling

try - compound error handling statement

```
>>> 1/0
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: integer division or modulo by zero
```

```
>>> try:
```

```
...     1/0
```

```
... except ZeroDivisionError:
```

```
...     print "Oops! divide by zero!"
```

```
... except:
```

```
...     print "some other exception!"
```

```
Oops! divide by zero!
```

File I/O Basics

Most I/O in Python follows the model laid out for file I/O and should be familiar to C/C++ programmers.

- basic built-in file i/o calls include
 - `open()`, `close()`
 - `write()`, `writeline()`, `writelines()`
 - `read()`, `readline()`, `readlines()`
 - `flush()`
 - `seek()` and `tell()`
 - `fileno()`
- basic i/o supports both text and binary files
- POSIX like features are available via `fcntl` and `os` modules
- be a good citizen
 - if you open, close your descriptors
 - if you lock, unlock when done

Basic I/O examples

open a text file for reading with default buffering

```
>>> f=file.open('myfile.txt','r')
```

- for writing use 'w'
- for simultaneous reading and writing add '+' to either 'r' or 'w'
- for appending use 'a'
- to do binary files add 'b'

opens a text file for reading and writing with no buffering.

```
>>> f=file.open('myfile.txt','w+',0)
```

- a 1 means line buffering,
- other values are interpreted as buffer sizes in bytes

Let's write ten integers to disk without buffering, then read them back:

```
>>> f=open('frogs.dat','w+',0) # open for unbuffered reading and writing
>>> f.writelines([str(my_int) for my_int in range(10)])
>>> f.tell() # we're about to see we've made a mistake
10L # hmm... we seem short on stuff
>>> f.seek(0) # go back to the start of the file
>>> f.tell() # make sure we're there
0L
>>> f.readlines() # Let's see what's written on each line
['0123456789'] # we've written 10 chars, no line returns... oops
>>> f.seek(0) # jumping back to start, let's add line returns
>>> f.writelines([str(my_int)+'\n' for my_int in range(10)])
>>> f.tell() # jumping back to start, let's add line returns
20L
>>> f.seek(0) # return to start of the file
>>> f.readline() # grab one line
'0\n'
>>> f.next() # grab what ever comes next
'1\n'
>>> f.readlines() # read all remaining lines in the file
['2\n', '3\n', '4\n', '5\n', '6\n', '7\n', '8\n', '9\n']
>>> f.close() # always clean up after yourself - no need other than courtesy!
```

Pickling

a.k.a.: serializing Python objects

```
# make a list w/ numeric values, a string, and a dict
```

```
>>> a = [1, 3, 5, 'hello', {'key':'value', 'otherkey':'othervalue'}]
```

```
# use pickle to serialize and dump to a file
```

```
>>> import pickle
```

```
>>> pickle.dump(a,open('filename.pickle','wb'))
```

```
# unpickle serialized data
```

```
>>> b=pickle.load(open('filename.pickle','rb'))
```

```
>>> b
```

```
[1, 3, 5, 'hello', {'otherkey': 'othervalue', 'key': 'value'}]
```

I/O for scientific formats

i/o is relatively weak out of the box - luckily there are the following alternatives:

- h5py
 - Python bindings for HDF5
 - <http://code.google.com/p/h5py/>
- netCDF4
 - Python bindings for NetCDF
 - <http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF4-module.html>
- mpi4py allows for classic MPI-IO via MPI.File

Modules

import - load module, define in namespace

```
>>> import random # import module
```

```
>>> random.random() # execute module method  
0.82585453878964787
```

```
>>> import random as rd # import and rename
```

```
>>> rd.random()  
0.22715542164248681
```

bring randint into namespace from random

```
>>> from random import randint
```

```
>>> randint(0,10)
```

```
4
```

Classes & Object Orientation

```
>>> class SomeClass:
...     """A simple example class""" # docstring
...     pi = 3.14159 # attribute
...     def __init__(self, ival=89): # init w/ default
...         self.i = ival
...     def f(self): # class method
...         return 'Hello'
>>> c = SomeClass(42) # instantiate
>>> c.f() # call class method
'hello'

>>> c.pi = 3 # change attribute

>>> print c.i # print attribute
42
```



- N-dimensional homogeneous arrays (ndarray)
- Universal functions (ufunc)
 - built-in linear algebra, FFT, PRNGs
- Tools for integrating with C/C++/Fortran
- Heavy lifting done by optimized C/Fortran libraries
 - ATLAS or MKL, UMFPACK, FFTW, etc...

Creating NumPy Arrays

Initialize with lists: array with 2 rows, 4 cols

```
>>> import numpy as np
>>> np.array([[1,2,3,4],[8,7,6,5]])
array([[1, 2, 3, 4],
       [8, 7, 6, 5]])
```

Make array of evenly spaced numbers over an interval

```
>>> np.linspace(1,100,10)
array([ 1., 12., 23., 34., 45., 56., 67., 78., 89., 100.]
```

Create and prepopulate with zeros

```
>>> np.zeros((2,5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Slicing Arrays

```
>>> a = np.array([[1,2,3,4],[9,8,7,6],[1,6,5,4]])
>>> arow = a[0,:] # get slice referencing row zero
>>> arow
array([1, 2, 3, 4])
```

```
>>> cols = a[:,[0,2]] # get slice referencing columns 0 and 2
>>> cols
array([[1, 3],
       [9, 7],
       [1, 5]])
```

NOTE: arow & cols are NOT copies, they point to the original data

```
>>> arow[:] = 0
>>> arow
array([0, 0, 0, 0])
>>> a
array([[0, 0, 0, 0],
       [9, 8, 7, 6],
       [1, 6, 5, 4]])
```

Copy data

```
>>> copyrow = arow.copy()
```

Broadcasting with ufuncs

apply operations to many elements with a single call

```
>>> a = np.array([[1,2,3,4],[8,7,6,5]])
```

```
>>> a  
array([[1, 2, 3, 4],  
       [8, 7, 6, 5]])
```

Rule 1: Dimensions of one may be prepended to either array

```
>>> a + 1 # add 1 to each element in array
```

```
array([[2, 3, 4, 5],  
       [9, 8, 7, 6]])
```

Rule 2: Arrays may be repeated along dimensions of length 1

```
>>> a + np.array([1],[10]) # add 1 to 1st row, 10 to 2nd row
```

```
array([[ 2,  3,  4,  5],  
       [18, 17, 16, 15]])
```

```
>>> a**([2],[3]) # raise 1st row to power 2, 2nd to 3
```

```
array([[ 1,  4,  9, 16],  
       [512, 343, 216, 125]])
```



SciPy

- Extends NumPy with common scientific computing tools
 - optimization
 - additional linear algebra
 - integration
 - interpolation
 - FFT
 - signal and image processing
 - ODE solvers
- Heavy lifting done by C/Fortran code

Parallel & Distributed Programming

threading

- useful for certain concurrency issues, not usable for parallel computing due to Global Interpreter Lock (GIL)

subprocess

- relatively low level control for spawning and managing processes

multiprocessing - multiple Python instances (processes)

- basic, clean multiple process parallelism

MPI

- mpi4py exposes your full local MPI API within Python
- as scalable as your local MPI

Python Threading

Python threads

- real POSIX threads
- share memory and state with their parent processes
- do not use IPC or message passing
- light weight
- generally improve latency and throughput
- there's a heck of a catch, one that kills performance...

The Infamous GIL

To keep memory coherent, Python only allows a single thread to run in the interpreter's space at once. This is enforced by the Global Interpreter Lock, or GIL. It also kills performance for most serious workloads.

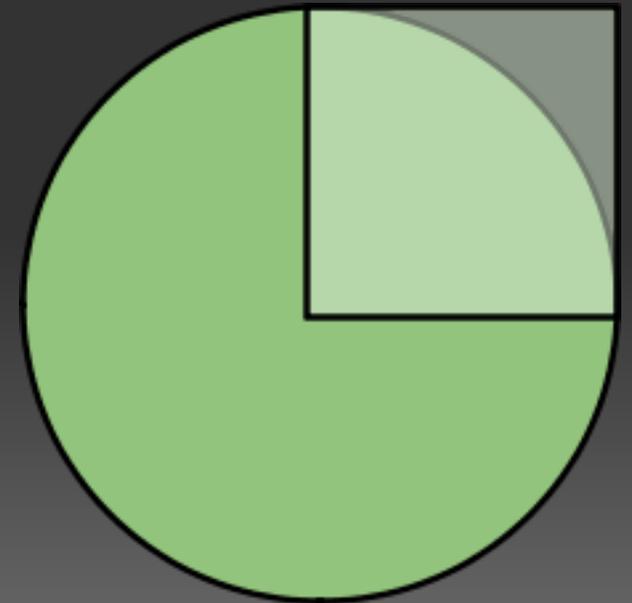
It's not all bad. The GIL:

- Is mostly sidestepped for I/O (files and sockets)
- Makes writing modules in C much easier
- Makes maintaining the interpreter much easier
- Makes for any easy target of abuse
- Gives people an excuse to write competing threading modules (please don't)

For the gory details See David Beazley's talk on the GIL: <http://blip.tv/file/2232410>

Implementation Example: Calculating Pi

- Generate random points inside a square
- Identify fraction (f) that fall inside a circle with radius equal to box width
 - $x^2 + y^2 < r$
- Area of quarter of circle (A) = $\pi \cdot r^2 / 4$
- Area of square (B) = r^2
- $A/B = f = \pi/4$
- $\pi = 4f$



Calculating pi with threads

```
from threading import Thread, Lock
import random
```

```
lock = Lock() # lock for making operations atomic
```

```
def calcInside(nsamples, rank):
    global inside # we need something everyone can share
    random.seed(rank)
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x*x)+(y*y)<1:
            lock.acquire() # GIL doesn't always save you
            inside += 1
            lock.release()
```

```
if __name__ == '__main__':
    nt=4 # thread count
    inside = 0 # you need to initialize this
    samples=int(12e6/nt)
    threads=[Thread(target=calcInside, args=(samples,i)) for i in range(nt)]

    for t in threads: t.start()
    for t in threads: t.join()

    print (4.0*inside)/(1.0*samples*nt)
```

Execution Time

nt=1: 15.45±0.22 sec

nt=2: 55.38±0.46 sec

Mac OS X, Python 2.6
Core 2 2.53 GHz

Subprocess

The `subprocess` module allows the Python interpreter to spawn and control processes. It is unaffected by the GIL. Using the `subprocess.Popen()` call, one may start any process you'd like.

```
>>> pi=subprocess.Popen('python -c "import math; print
math.pi"',shell=True,stdout=subprocess.PIPE)
>>> pi.stdout.read()
'3.14159265359\n'
>>> pi.pid
1797
>>> me.wait()
0
```

It goes without saying, there's better ways to do subprocesses...

Multiprocessing

- Added in Python 2.6
- Faster than threads as the GIL is sidestepped
- uses subprocesses
 - both local and remote subprocesses are supported
 - shared memory between subprocesses is risky
 - no coherent types
 - `Array` and `Value` are built in
 - others via `multiprocessing.sharedctypes`
 - IPC via pipes and queues
 - pipes are not entirely safe
- synchronization via locks
- `Manager` allows for safe distributed sharing, but it's slower than shared memory

Calculating pi with multiprocessing

```
import multiprocessing as mp
import numpy as np
import random

processes = mp.cpu_count()
nsamples = 120000/processes

def calcInside(rank):
    inside = 0
    random.seed(rank)
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x*x)+(y*y)<1:
            inside += 1
    return (4.0*inside)/nsamples

if __name__ == '__main__':
    pool = mp.Pool(processes)
    result = pool.map(calcInside, range(processes))
    print np.mean(result)
```

pi with multiprocessing, optimized

```
import multiprocessing as mp
import numpy as np

processes = mp.cpu_count()
nsamples = int(12e6/processes)
```

```
def calcInsideNumPy(rank):
    np.random.seed(rank)
```

```
    # "vectorized" sample gen, col 0 = x, col 1 = y
```

```
    xy = np.random.random((nsamples, 2))
    return 4.0*np.sum(np.sum(xy**2, 1)<1)/nsamples
```

```
if __name__ == '__main__':
    pool = mp.Pool(processes)
    result = pool.map(calcInsideNumPy, range(processes))
    print np.mean(result)
```

Execution Time

Unoptimized: 4.76±0.23 sec

Vectorized: 1.30±0.14 sec

mpi4py

- wraps your native mpi
 - prefers MPI2, but can work with MPI1
- works best with NumPy data types, but can pass around any serializable object
- provides all MPI2 features
- well maintained
- distributed with Enthought Python Distribution (EPD)
- requires NumPy
- portable and scalable
- <http://mpi4py.scipy.org/>

How mpi4py works...

- mpi4py jobs must be launched with mpirun
- each rank launches its own independent python interpreter
- each interpreter only has access to files and libraries available locally to it, unless distributed to the ranks
- communication is handled by MPI layer
- any function outside of an if block specifying a rank is assumed to be global
- any limitations of your local MPI are present in mpi4py

Calculating pi with mpi4py

```
from mpi4py import MPI
import numpy as np
import random

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()
nsamples = int(12e6/mpisize)

inside = 0
random.seed(rank)
for i in range(nsamples):
    x = random.random()
    y = random.random()
    if (x*x)+(y*y)<1:
        inside += 1

mypi = (4.0 * inside)/nsamples
pi = comm.reduce(mypi, op=MPI.SUM, root=0)

if rank==0:
    print (1.0 / mpisize)*pi
```

Performance

- Best practices with pure Python & NumPy
- Optimization where needed (we'll talk about this in GPAW)
 - profiling
 - inlining

Python Best Practices for Performance

If at all possible...

- Don't reinvent the wheel.
 - someone has probably already done a better job than your first (and probably third) attempt
- Build your own modules against optimized libraries
 - ESSL, ATLAS, FFTW, MKL
- Use NumPy data types & functions instead of built-in Python ones for homogeneous vectors/arrays
- "vectorize" operations on $\geq 1D$ data types.
 - avoid for loops, use single-shot operations
- Pre-allocate arrays instead of repeated concatenation
 - use `numpy.zeros`, `numpy.empty`, etc..

Real-World Examples and Techniques:

GPAW



a massively parallel Python-C code for KS-DFT

Many firsts:

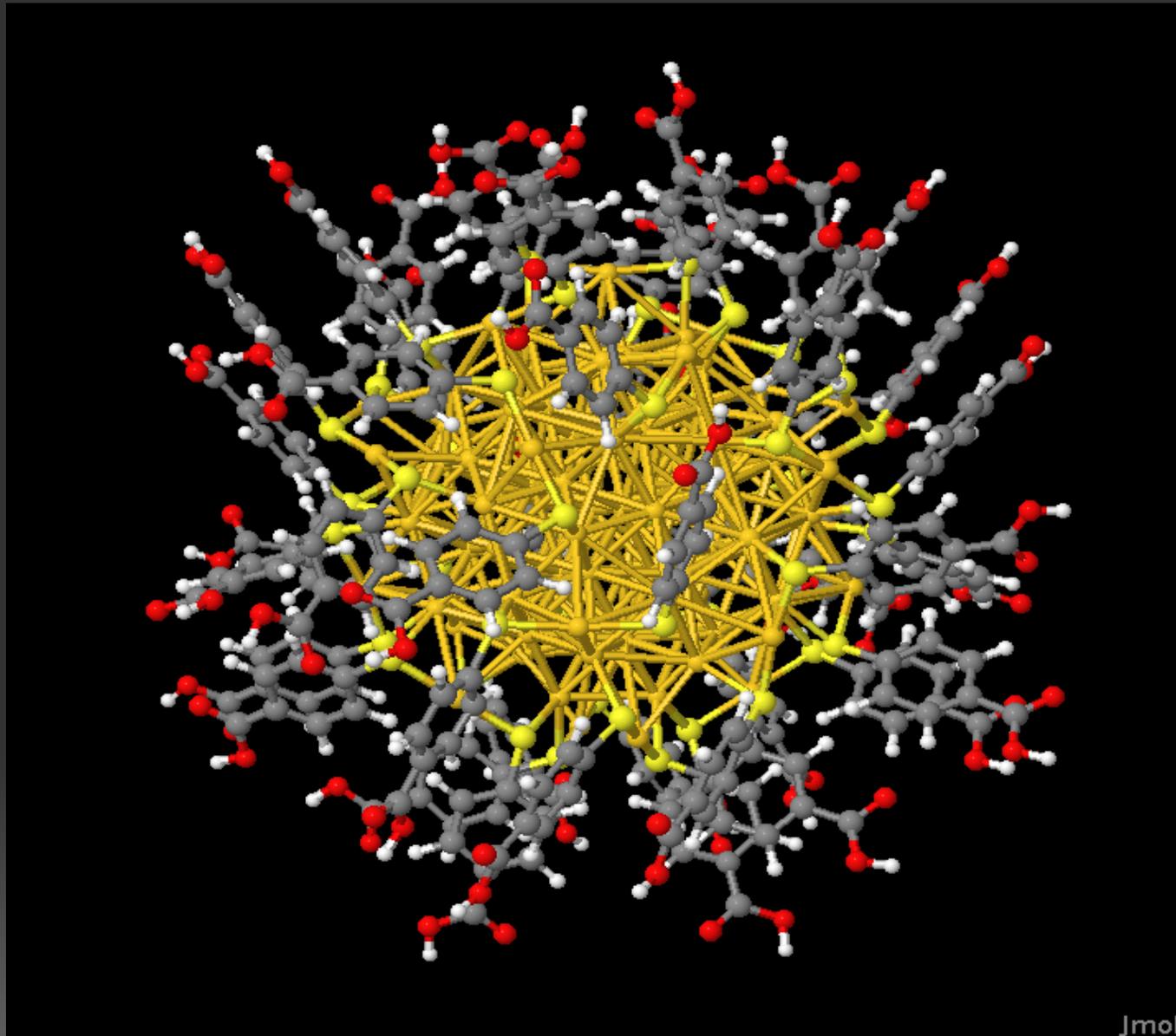
1. Programming in Python (me)
2. Compiling NumPy on BG/P
3. Running an MPI Python-C code on BG/P
4. Profiling Python, C, and MPI simultaneously
5. Running an application on BG/P codes that links in shared libraries instead of statics libraries.
6. Particular implementation of DFT algorithm: PAW + real-space
7. GoogleDocs

This is A LOT of firsts!

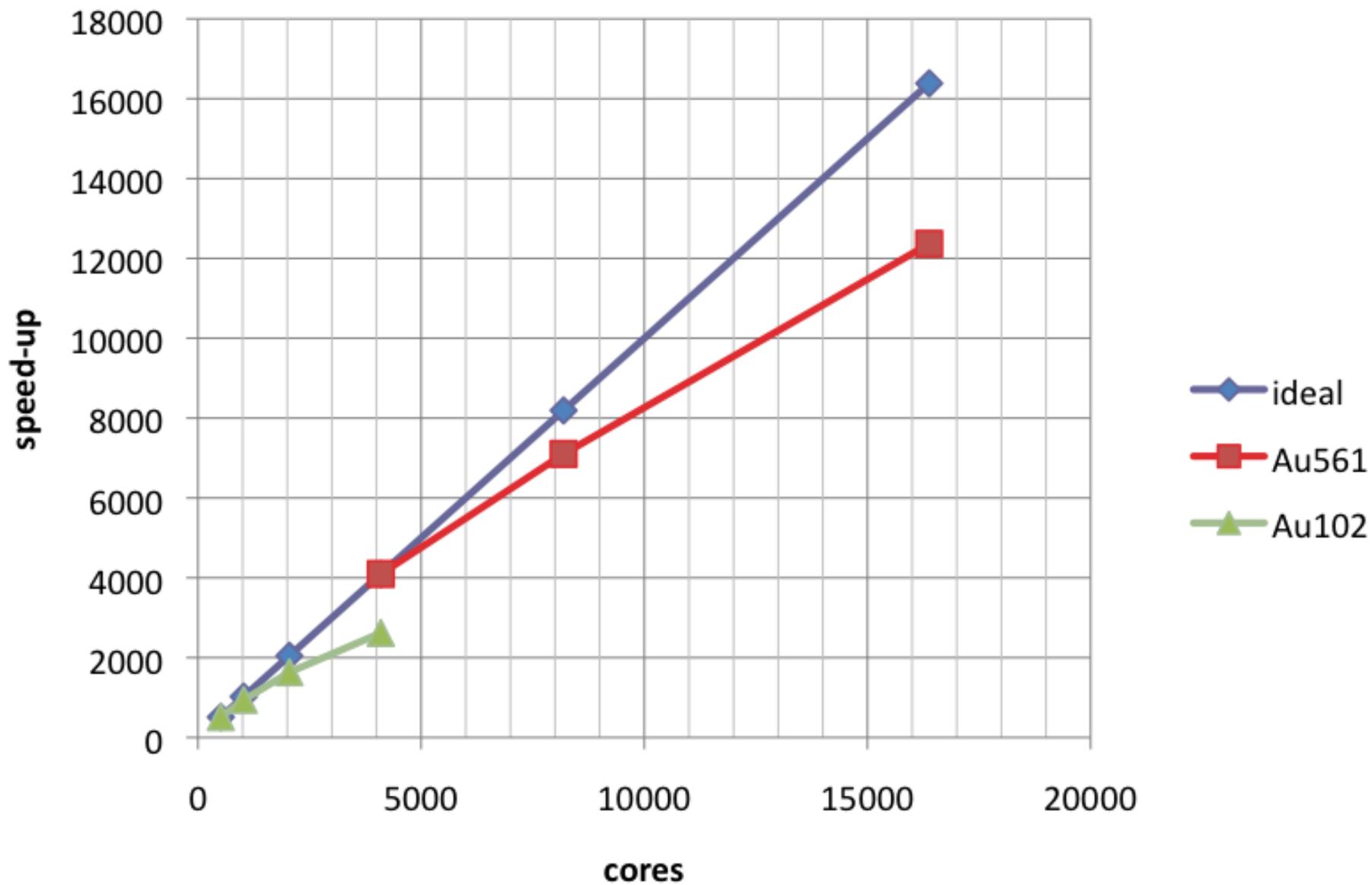
GPAW Overview

- GPAW is an implementation of the projector augmented wave method (PAW) method for Kohn-Sham (KS) - Density Functional Theory (DFT)
 - Mean-field approach to Schrodinger equation
 - Uniform real-space grid
 - Non-linear sparse eigenvalue problem
 - 10^6 grid points, 10^3 eigenvalues
 - Solved self-consistently using RMM-DIIS
 - Nobel prize in Chemistry to Walter Kohn (1998) for DFT
- *Ab initio* atomistic simulation for predicting material properties
- Massively parallel using MPI
- Written in Python-C using the NumPy library.

Electronic Structure of Nanoparticles



GPAW Strong-scaling Results



GPAW code structure

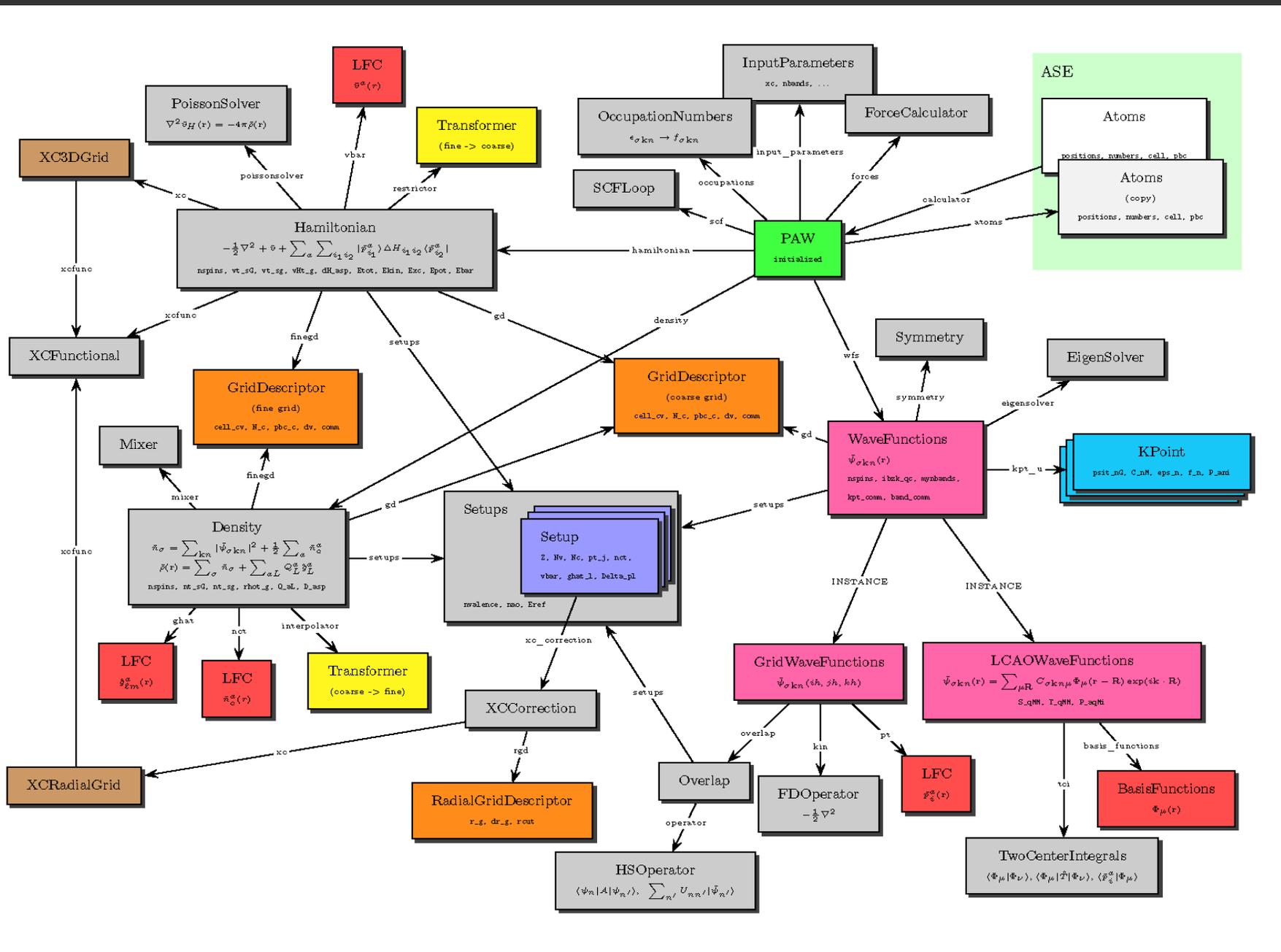
- Built on top of **NumPy** library.
- Not simply a Python wrapper on legacy Fortran/C code
- Python for coding the high-level algorithm
- **C** for coding numerical intense operations
- Use **BLAS** and **LAPACK** whenever possible

Here is some pseudo code for iterative eigensolver:

```
for i in xrange(max SCF):  
    for n in xrange(number of bands):  
        R_ng = apply(H_gg, Psi_ng) # Compute residuals  
        rk(1.0, R_ng, 0.0, H_mn) # construct Hamiltonian
```

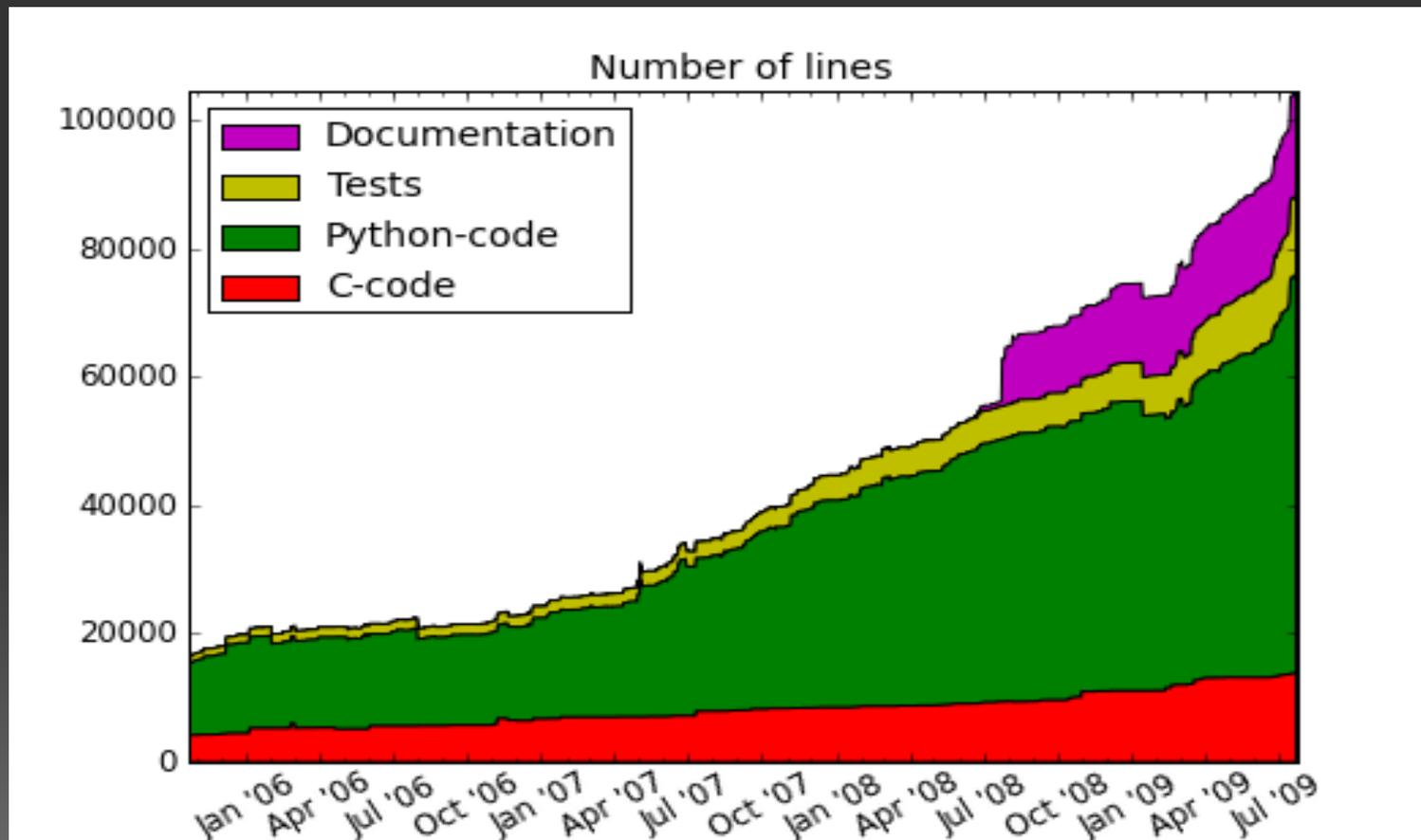
KS-DFT algorithms are well-known and computationally intensive parts are known *a priori*.

KS-DFT is a complex algorithm!



Source Code Timeline

- **Mostly Python-code, 10% C-code.**
- 90% of wall-clock time spend in C, BLAS, and LAPACK.



Performance Mantra

People are able to code complex algorithms in much less time by using a high-level language like Python. *There can be a performance penalty in the most pure sense of the term.*

"The best performance improvement is the transition from the nonworking to the working state."

--John Ousterhout

"Premature optimization is the root of all evil."

--Donald Knuth

"You can always optimize it later."

-- Unknown

NumPy - Weakly-typed data structures

Weakly-type data structures are handy. In KS-DFT, we basically need a real double-precision (G-point) and complex double-precision (K-point) of everything:

- Fortran77/Fortran90 - end up with lots of if-statements and modules
- C++ - handles this with templating and operator overloading
- Python - doesn't care, but your C extensions will but that is only 10% of your code.

NumPy - Memory

BlueGene/P has 512 MB per core.

- Compute node kernel ~ 34 MB.
- NumPy library ~ 38 MB.
- Python Interpreter ~ 12 MB.
- Can't always get the last 50 MB, NumPy to blame?

Try this simple test:

```
import numpy as np
A = np.zeros((N,N), dtype=float)
```

Only 350 MB of memory left on BG/P per core for calculation!

NumPy - FLOPS

Optimized BLAS available via NumPy `np.dot`. Handles general inner product of multi-dimensional arrays.

- Very difficult to cross-compile on BG/P. **Blame disutils!**
 - `core/_dotblas.so` is a sign of optimized `np.dot`
 - Python wrapper overhead is negligible
- For very large matrices (~50 MB), there is a big performance difference
 - unoptimized - 1% single core peak performance
 - optimized - 80% single core peak performance
- For matrix * vector products, `np.dot` can yield better performance than direct call to GEMV!

NumPy - FLOPS

Fused floating-point multiply-add instructions are not created for AXPY type operation in Python.

```
for i in xrange(N):  
    Y[i] += alpha*X[i]  
    C[i] += A[i]*B[i]
```

- 2X slower than separate multiple and add instructions, another 2X due to PPC double FPU
- May not be a problem in future version of Python, especially with LLVM

NumPy - FLOPS

WARNING: If you make heavy, use of BLAS & LAPACK type operations.

- Plan on investing a significant amount of time working to cross-compile optimized NumPy.
- Safest thing is to write your own C-wrappers.
- If all your NumPy arrays are < 2-dimensional, Python wrappers will be simple.
- Wrappers for multi-dimensional arrays can be challenging:
 - SCAL, AXPY is simple
 - GEMV more difficulty
 - GEMM non-trivial
- Remember C & NumPy arrays are row-ordered by default, Fortran arrays are column-ordered!

Python BLAS Interface

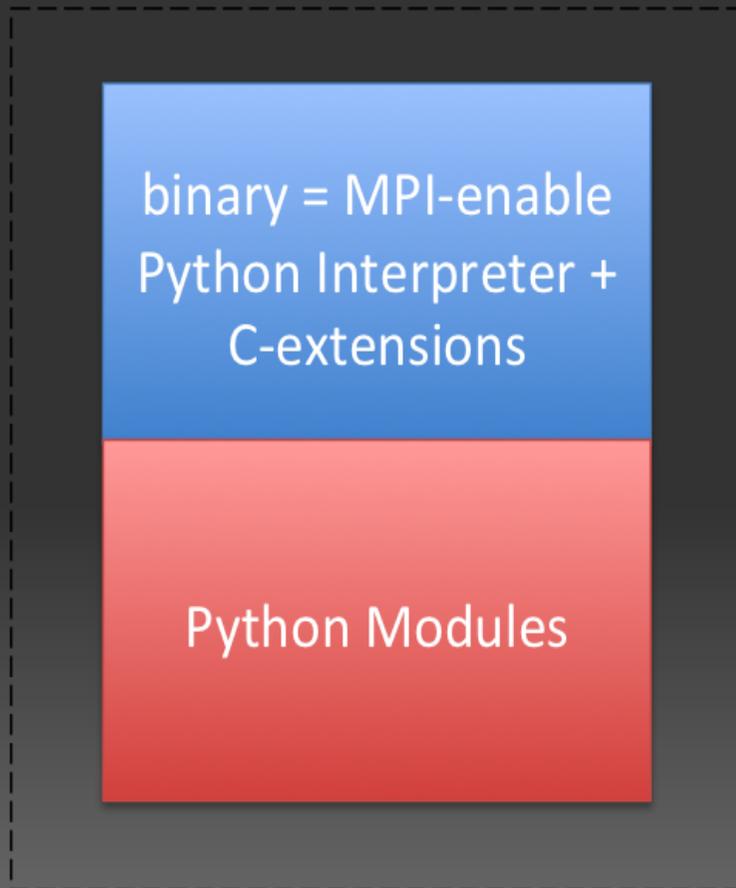
```
void dscal_(int*n, double* alpha, double* x, int* incx); // C prototype for Fortran
void zscal_(int*n, void* alpha, void* x, int* incx); // C prototype for Fortran
#define DOUBLEP(a) ((double*)((a)->data)) // Casting for NumPy data struc.
#define COMPLEXP(a) ((double_complex*)((a)->data)) // Casting for NumPy data struc.

PyObject* scal(PyObject *self, PyObject *args)
{
    Py_complex alpha;
    PyArrayObject* x;
    if (!PyArg_ParseTuple(args, "DO", &alpha, &x))
        return NULL;
    int n = x->dimensions[0];
    for (int d = 1; d < x->nd; d++) // NumPy arrays can be multi-dimensional!
        n *= x->dimensions[d];
    int incx = 1;

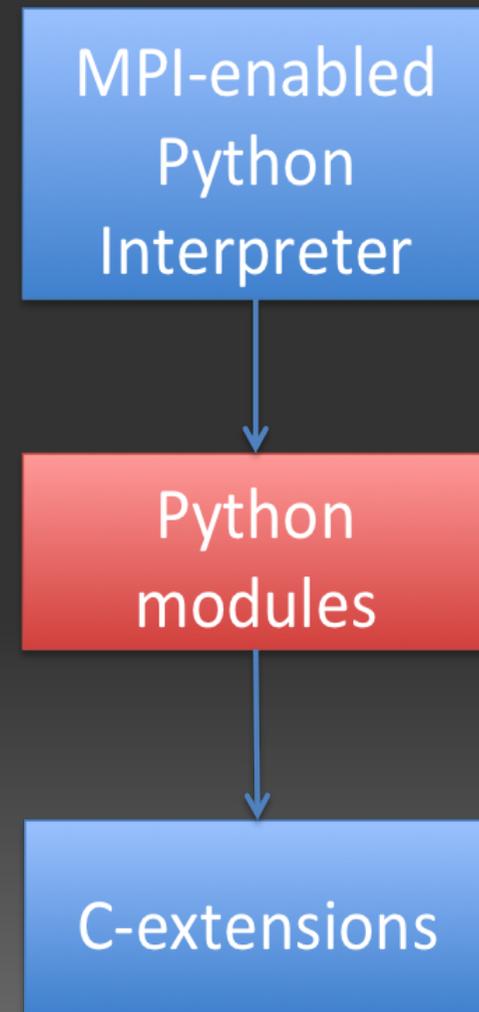
    if (x->descr->type_num == PyArray_DOUBLE)
        dscal_(&n, &(alpha.real), DOUBLEP(x), &incx);
    else
        zscal_(&n, &alpha, (void*)COMPLEXP(x), &incx);
    Py_RETURN_NONE;
}
```

Parallel Python Interpreter and Debugging

GPAW



call stack



Parallel Python Interpreter and Debugging

MPI-enabled "embedded" Python Interpreter:

```
int main(int argc, char **argv)
{
    int status;
    MPI_Init(&argc, &argv); // backwards compatible with MPI-1
    Py_Initialize(); // needed because of call in next line
    PyObject* m = Py_InitModule3("_gpaw", functions,
                                "C-extension for GPAW\n\n...
\n");
    import_array1(-1); // needed for NumPy C-API
    MPI_Barrier(MPI_COMM_WORLD); // sync up
    status = Py_Main(argc, argv); // call to Python Interpreter
    MPI_Finalize();
    return status;
}
```

Parallel Python Interpreter and Debugging

Errors in Python modules are OK, core dumps in C extensions are problematic:

- Python call stack is hidden; this is due to Python's interpreted nature.
- Totalview won't help, sorry.

The screenshot shows the Core Processor debugger interface. The main window displays a stack trace for a core dump. The stack trace is as follows:

```
pute Node (29)
AR=0xffffffff)  ffffffff (29)
AR=0x83171bd4)  83171bd4 (29)
AR=0x8317199c)  8317199c (29)
AR=0x0101701c)  main (29)
AR=0x8255125c)  8255125c (29)
AR=0x82546f70)  82546f70 (29)
AR=0x82546bf4)  82546bf4 (29)
AR=0x825468f0)  825468f0 (29)
AR=0x8251ff88)  8251ff88 (29)
AR=0x8251ff10)  8251ff10 (29)
AR=0x8251dd44)  8251dd44 (29)
AR=0x8251dc14)  8251dc14 (29)
AR=0x8251ff10)  8251ff10 (29)
AR=0x8251dc14)  8251dc14 (29)
AR=0x8251ff10)  8251ff10 (29)
AR=0x8251dc14)  8251dc14 (29)
AR=0x8251ff10)  8251ff10 (29)
AR=0x8251dd44)  8251dd44 (29)
AR=0x8251dbb8)  8251dbb8 (29)
AR=0x824c602c)  824c602c (29)
AR=0x01020c30)  calculate_potential_matrix (29)
```

The right-hand pane shows a list of common nodes:

```
Common nodes:
disasm 0x0101701c
core_TGID_108_Thread_0
core_TGID_109_Thread_0
core_TGID_110_Thread_0
core_TGID_111_Thread_0
core_TGID_112_Thread_0
core_TGID_113_Thread_0
core_TGID_114_Thread_0
core_TGID_115_Thread_0
core_TGID_124_Thread_0
core_TGID_125_Thread_0
core_TGID_126_Thread_0
core_TGID_128_Thread_0
core_TGID_129_Thread_0
core_TGID_130_Thread_0
core_TGID_131_Thread_0
core_TGID_140_Thread_0
core_TGID_141_Thread_0
core_TGID_142_Thread_0
core_TGID_144_Thread_0
core_TGID_145_Thread_0
core_TGID_146_Thread_0
core_TGID_147_Thread_0
core_TGID_156_Thread_0
core_TGID_157_Thread_0
core_TGID_158_Thread_0
core_TGID_160_Thread_0
core_TGID_161_Thread_0
```

The status bar at the bottom indicates the location as `c:/gpaw.c:256` and the corefile as `n/a`.

Profiling Mixed Python-C code

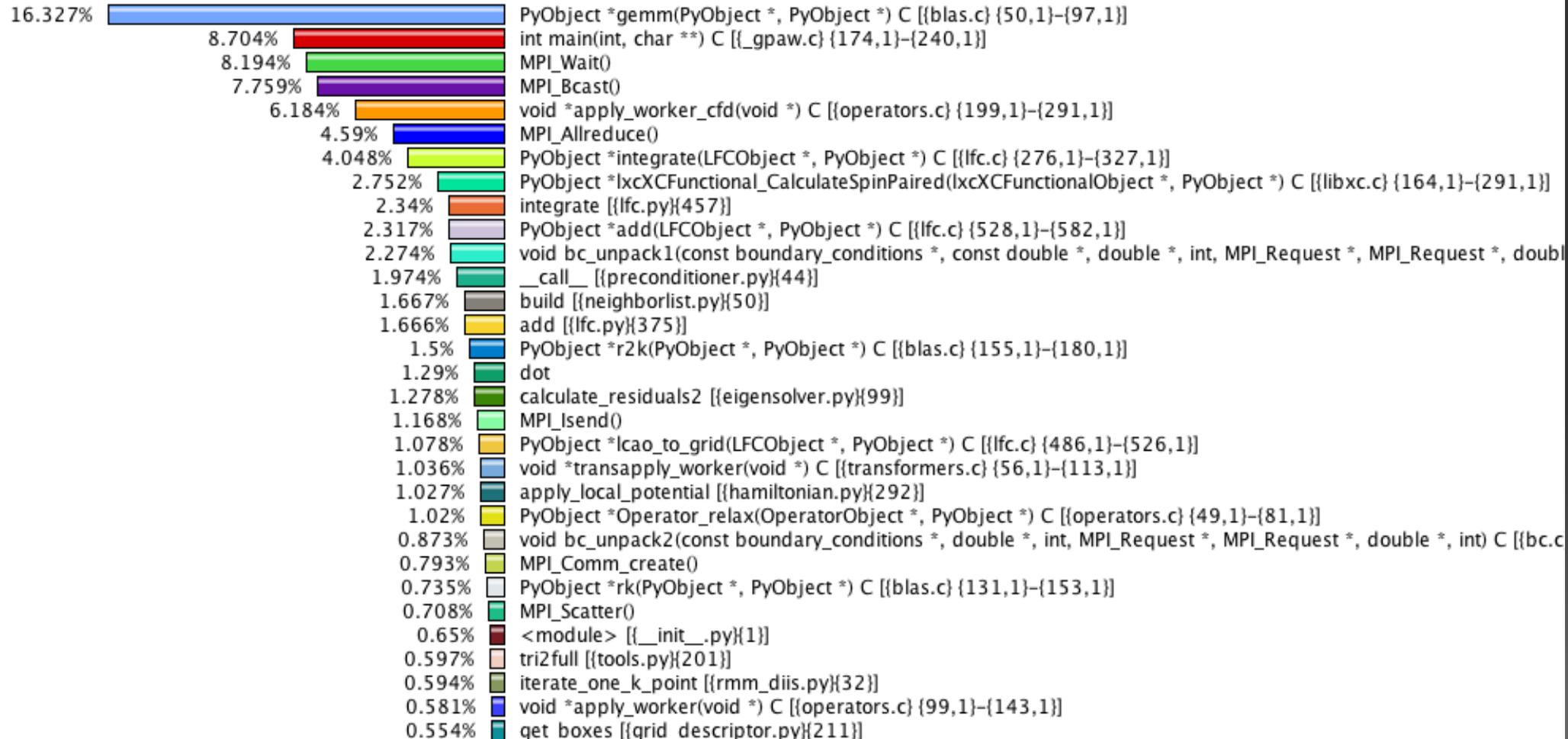
Number of profiling tools available:

- gprof, CrayPAT - C, Fortran
- import profile - Python
- TAU Performance System, <http://www.cs.uoregon.edu/research/tau/home.php> (next two slides)
 - Exclusive time for C, Python, MPI are reported simultaneously.
 - Heap memory profiling.
 - Interfaces with PAPI for performance counters.
 - Manual and automatic instrumentation available.
 - Does **not** cost any \$\$\$.
- Finding performance bottlenecks is critical to scalability on HPC platforms

Profiling Mixed Python-C code

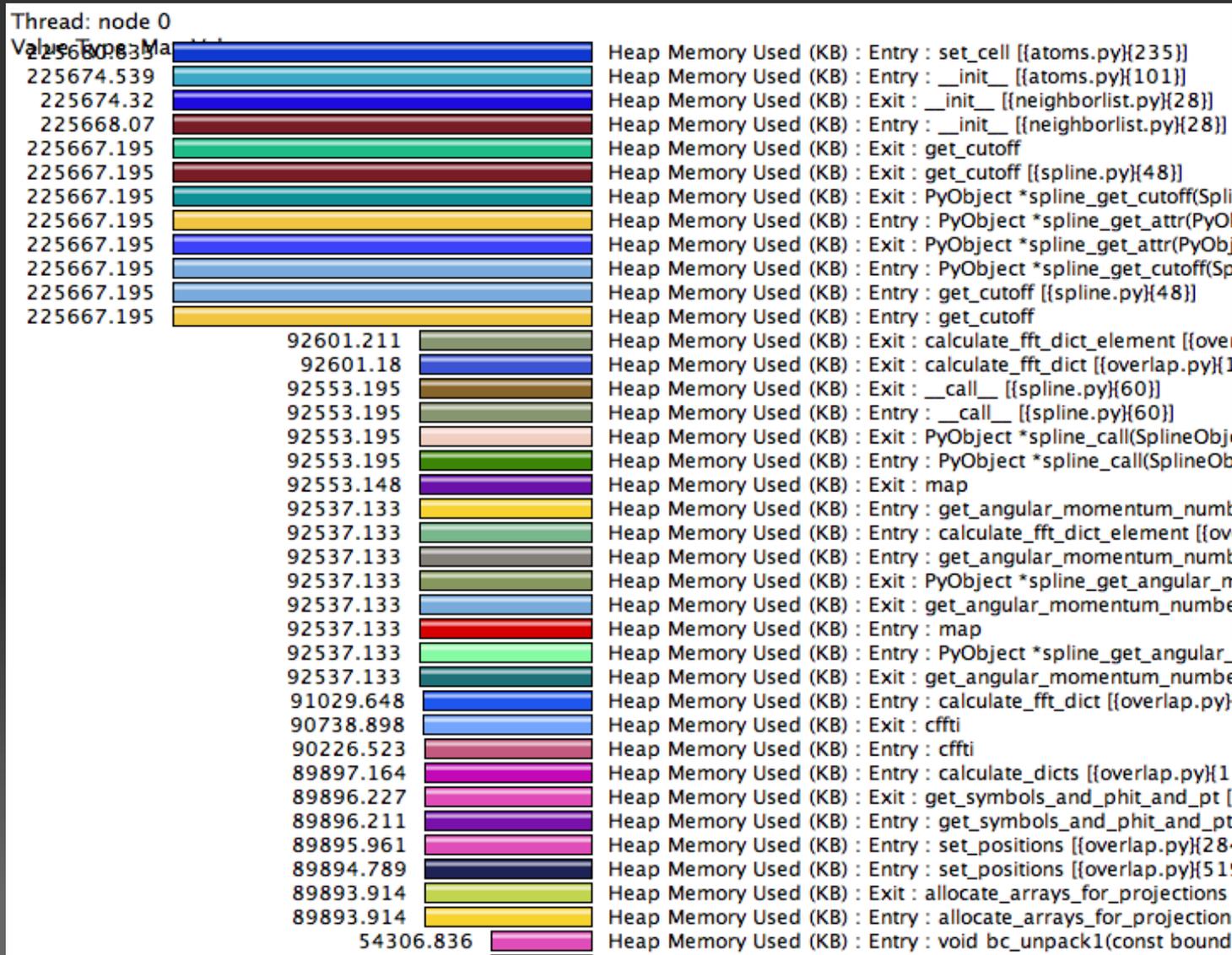
Flat profile shows time spent in Python, C, and MPI simultaneously:

Metric: BGP Timers
Value: Exclusive percent



Profiling Mixed Python-C code

Measure heap memory on subroutine entry/exit:



Motivation for Parallel Dense Linear Algebra

KS-DFT calculations depend roughly on two parameters N_g (number of grid points) and N_e (number of electrons), where $N_e \ll N_g$.

Computation scales:

$$a*N_g + b*N_g*N_e + c*(N_g)^2*N_e + d*(N_e)^3$$

Memory scales:

$$a*N_g*N_e + b*(N_e)^2$$

As the systems size grows, N_e computation requires **parallel dense linear algebra** on subspace matrices

- H_{mn} - Hamiltonian
- S_{mn} - Overlap

Python Interface to BLACS and ScaLAPACK

There is no parallel dense linear algebra in NumPy, there are some options:

- PyACTS, based on Numeric
- GAI_N, Global Arrays based on NumPy (very new)
- Write your own Python interface to ScaLAPACK.

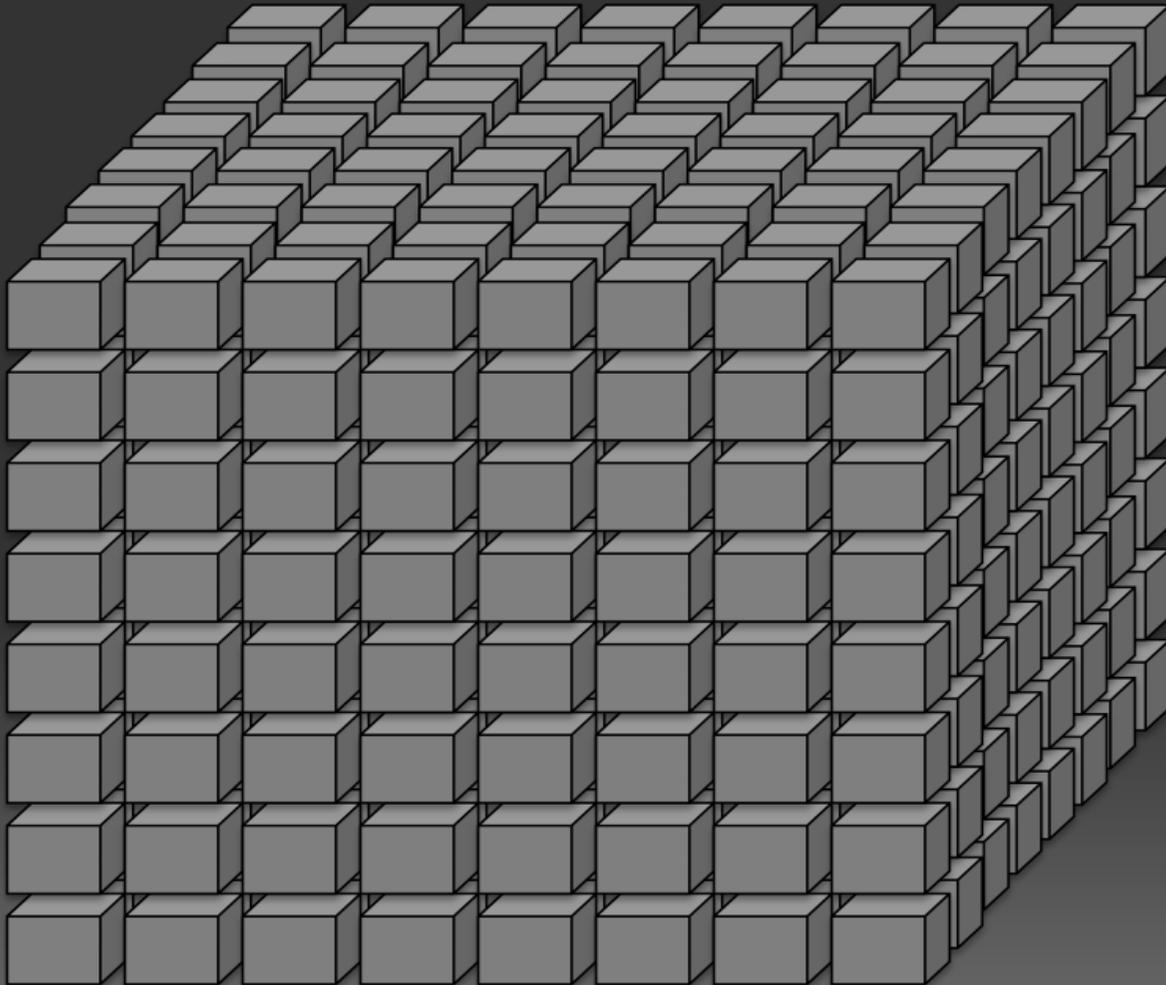
Python Interface to BLACS and ScaLAPACK

Mostly non-Python related challenges:

- Best way to understand ScaLAPACK is to read the source code.
- DFT leads to complicated scenarios for ScaLAPACK. `H_mn` and `O_mn` exist on a small subset of `MPI_COMM_WORLD`.
- ScaLAPACK does not distribute arrays object for you.
 - Local array must be created in a parallel by the application developer
 - ScaLAPACK allows you to manipulate them via descriptors
 - Array must be compatible with their native 2D-block cyclic layout
 - Distributed arrays assumed to be Fortran-ordered.

Python Interface to BLACS and ScaLAPACK

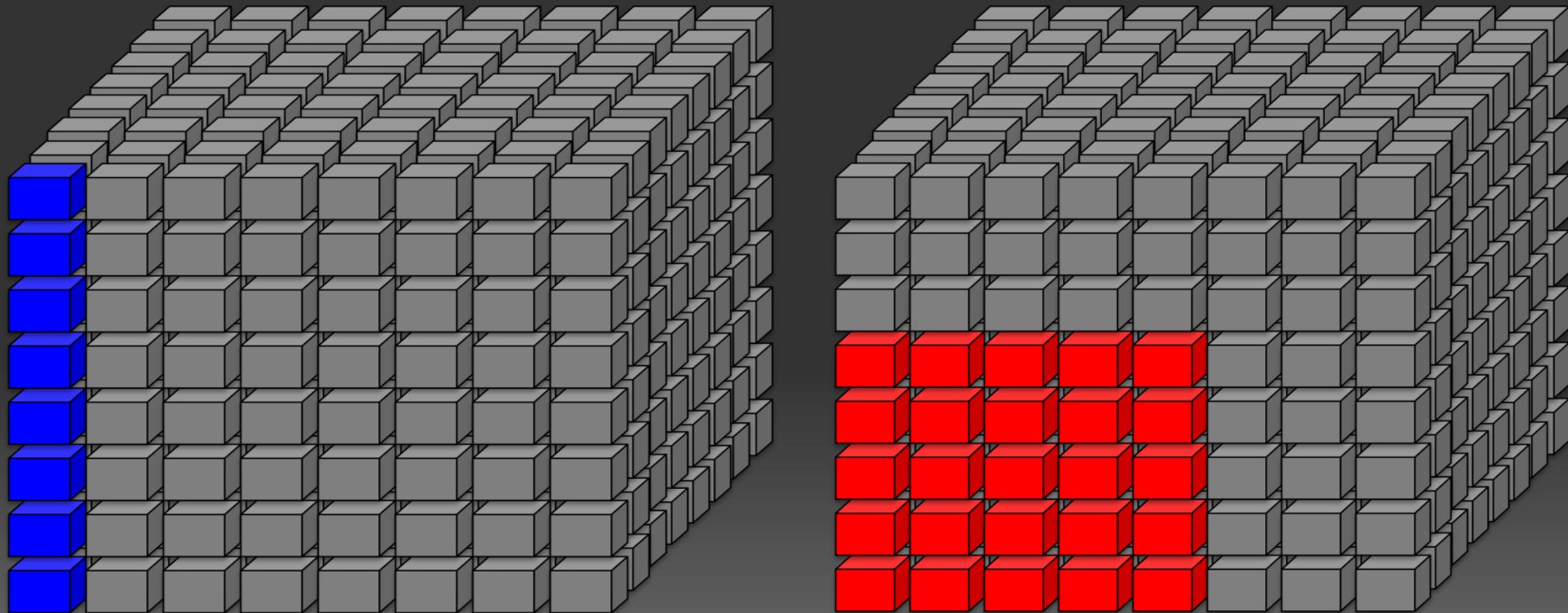
MPI_COMM_WORLD on a 512-node on 8x8x8 BG/P.



2048 cores!

Python Interface to BLACS and ScaLAPACK

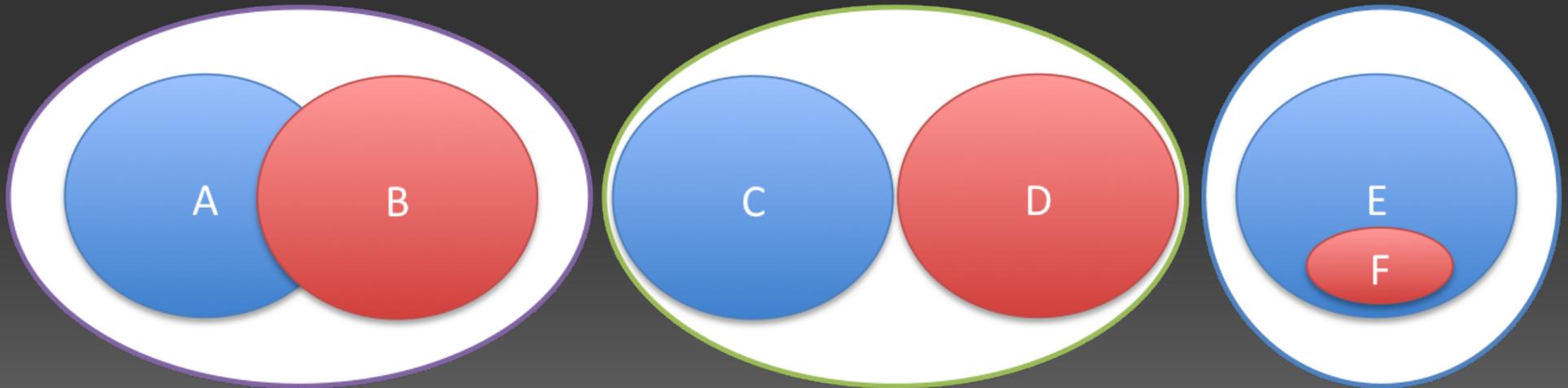
Physical 1D layout (left) of H_{mn} , S_{mn} requires redistribute to 2D block-cyclic layout (right) for use with ScaLAPACK.



Python Interface in BLACS and ScaLAPACK

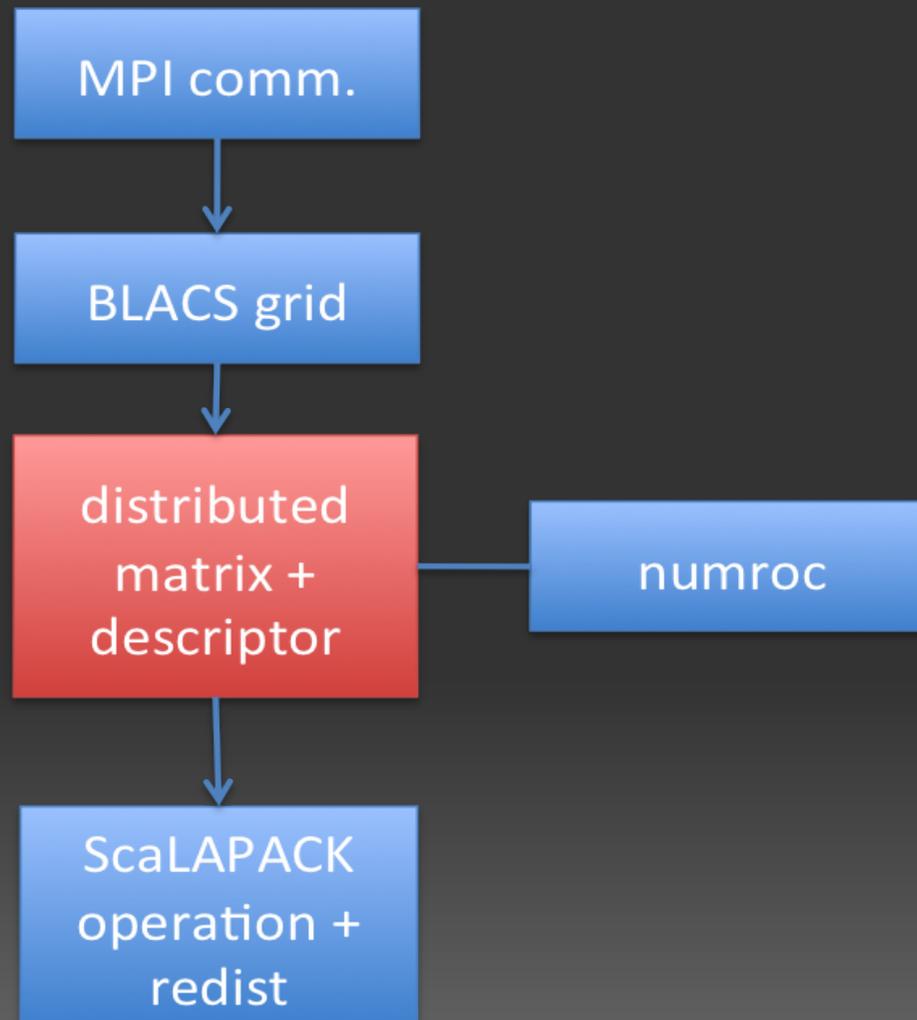
Source blacs grid (**blue**) and destination blacs grid (**red**).
Intermediate BLACS grid needed for ScaLAPACK redistribute:

- Must encompass both source and destination
- For multiple concurrent redist operations, intermediate cannot overlap.



Python Interface to BLACS and ScaLAPACK

Less than 1000 lines of **Python** and **C** code.



Python Interface in BLACS and ScaLAPACK

More information at:

<https://trac.fysik.dtu.dk/projects/gpaw/browser/trunk/c/blacs.c>

<https://trac.fysik.dtu.dk/projects/gpaw/browser/trunk/gpaw/blacs.py>

Summary

The Bad & Ugly:

- NumPy cross-compile.
- C Python extensions require learning NumPy & C API.
- Debugging C extensions can be difficult.
- Performance analysis will always be needed.
- OpenMP-like threading not available due to GIL.
- Python will need to support GPU acceleration in the future.

The Good:

GPAW has an extraordinary amount of functionality and scalability. A lot of features make coding complex algorithms easy:

- OOP
- weakly-typed data structures
- Interface with many things other languages: C, C++, Fortran, etc.

Acknowledgements

People:

- GPAW team
 - Technical University of Denmark - J. J. Mortensen, M. Dulak, A. H. Larsen, C. Glinsvaad, K. W. Jacobsen
 - CSC - IT Center for Science, Ltd., - J. Enkovaara
- ANL staff - V. A. Morozov, J. P. Greeley
- ParaTools, Inc. - S. Shende

Acknowledgements

Funding and Computational resources:

This research used resources at: Argonne Leadership Computing Facility and the Center for Nanoscale Materials at Argonne National Laboratory, which is supported by the office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357; High Performance Computing Center North (HPC2N). The Center for Atomic-scale Materials Design is sponsored by the Lundbeck Foundation. The authors acknowledge support from the Danish Center for Scientific Computing.

Python for plotting and visualization

- Overview of matplotlib
- Example of MC analysis tool written in Python
- Looking at data sharing on the web

From a Scientific Library To a Scientific Application

Massimo Di Pierro

From Lib to App (overview)

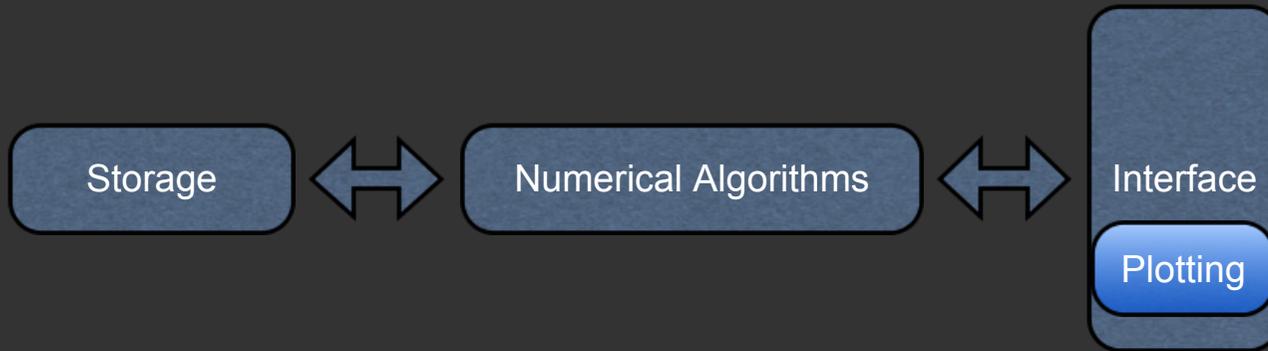
Numerical Algorithms

From Lib to App (overview)



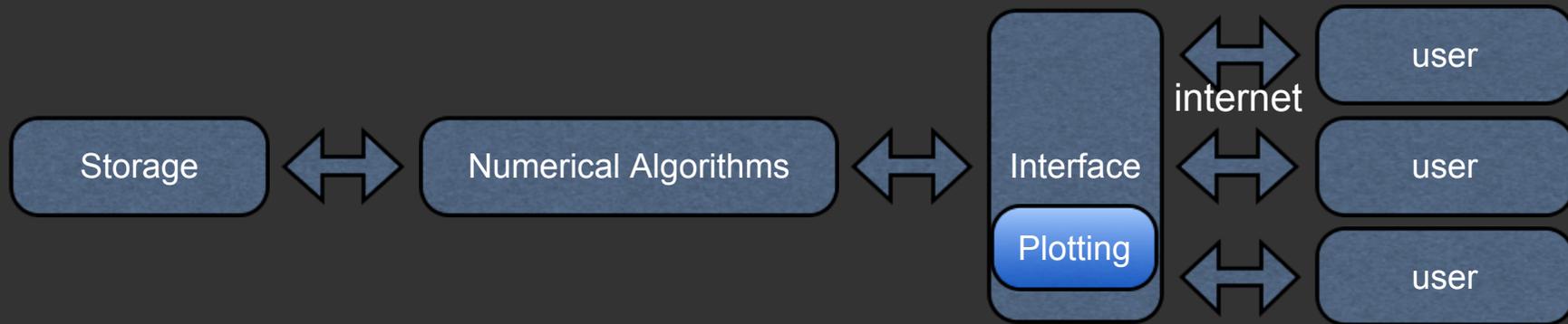
- Store and retrieve information in a relational database

From Lib to App (overview)



- Store and retrieve information in a relational database
- Provide a user interface
- input forms with input validation
- represent data (html, xml, csv, json, xls, pdf, rss)
- represent data graphically

From Lib to App (overview)



- Store and retrieve information in a relational database
- Provide a user interface
- input forms with input validation
- represent data (html, xml, csv, json, xls, pdf, rss)
- represent data graphically
- Communicate with users over the internet
- provide user authentication/authorization/access control
- provide persistence (cookies, sessions, cache)
- log activity and errors
- protect security of data

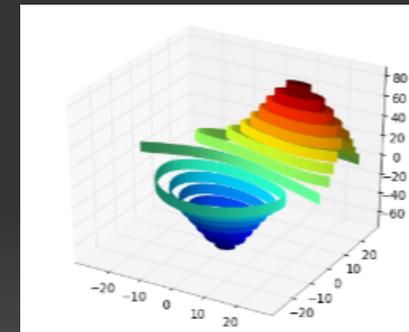
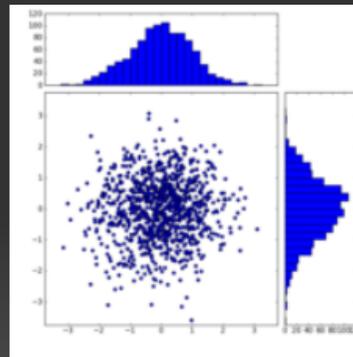
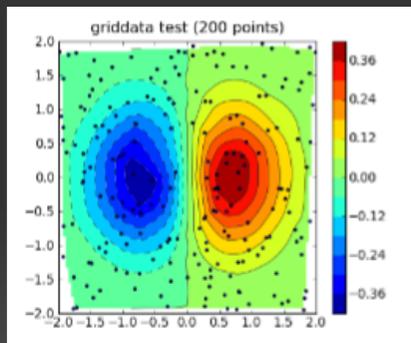
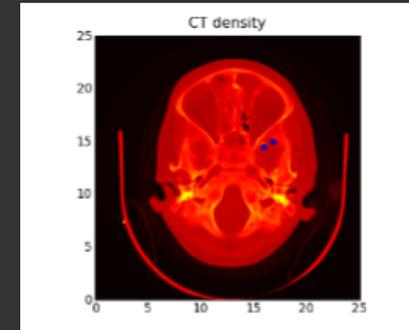
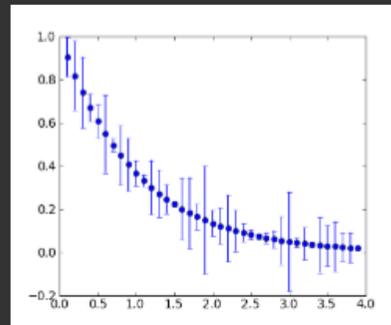
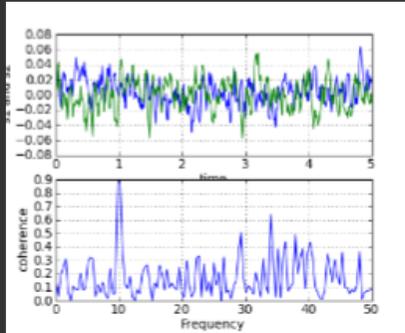
How? Use a framework!

- Ruby on Rails
- Django
- TurboGears
- Pylons
- ...
- web2py
- gnuplot.py
- r.py
- Chaco
- Dislin
- ...
- matplotlib

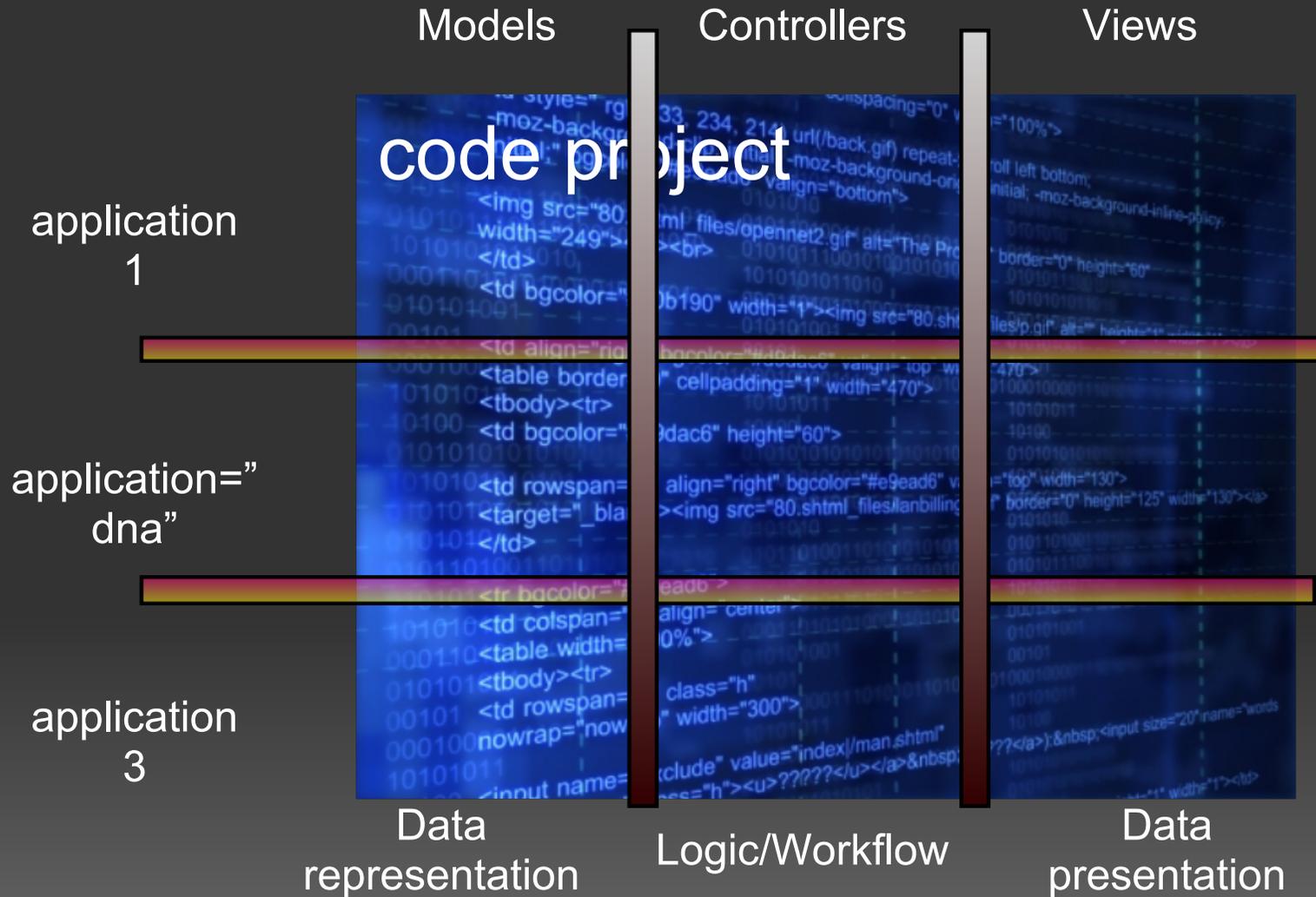
Why?

- web2py is really easy to use
- web2py is really powerful and does a lot for you
- web2py is really fast and scalable for production jobs
- I made web2py so I know it best
- matplotlib is the best library for plotting I have ever seen (not just in Python)

matplotlib gallery



web2py and MVC



web2py and MVC



web2py and Dispatching

```
<h1>
Upload DNA Seq.
</h1>
```

```
{{=form}}
```

The screenshot shows a web browser window with the URL `http://127.0.0.1:8000/dna/default/upload_dna`. The page features a teal header with the text "dna" and "customize me!". Below the header, the URL `/dna/default/upload_dna` is displayed. The main content area is divided into two sections: a left sidebar and a main form area. The sidebar contains three sections: "Authentication" with a "Login" button, "Main Menu" with an "Index" button, and "Edit This App" with an "Edit" button. The main form area is titled "Upload DNA Seq." and contains a "Sequence:" label followed by a text input field and a "Submit" button. The footer of the page reads "Copyright © 2009 - Powered by web2py". The browser's status bar at the bottom shows "Done" and a green checkmark icon.

web2py and Dispatching

hostnam
e dna

Getting Started Latest Headlines

dna
customize me!

/dna/default/upload_dna

Authentication
[Login](#)

Main Menu
[Index](#)

Edit This App
[Edit](#)

Upload DNA Seq.
Sequence:

Copyright © 2009 - Powered by **web2py**

Done

web2py and Dispatching

app name

http://127.0.0.1:8000/dna/default/upload_dna

Getting Started Latest Headlines

dna
customize me!

/dna/default/upload_dna

Authentication
[Login](#)

Main Menu
[Index](#)

Edit This App
[Edit](#)

Upload DNA Seq.
Sequence:

Copyright © 2009 - Powered by web2py

Done

web2py and Dispatching

controller
dna

Getting Started Latest Headlines

dna
customize me!

/dna/default/upload_dna

Authentication
[Login](#)

Main Menu
[Index](#)

Edit This App
[Edit](#)

Upload DNA Seq.
Sequence:

Copyright © 2009 - Powered by **web2py**

Done

web2py and Dispatching

action
name

dna
customize me!

/dna/default/upload_dna

Authentication
Login

Main Menu
Index

Edit This App
Edit

Upload DNA Seq.
Sequence:
Submit

Copyright © 2009 - Powered by web2py

Done

web2py and Views

```
<h1>
Upload DNA Seq.
</h1>
```

```
{{=form}}
```

The screenshot displays a web browser window with the URL `http://127.0.0.1:8000/dna/default/upload_dna`. The page layout includes a sidebar on the left with the following sections:

- Authentication**: [Login](#)
- Main Menu**: [Index](#)
- Edit This App**: [Edit](#)

The main content area contains a form titled "Upload DNA Seq." with the following elements:

- Label: "Sequence:"
- Input field: A text box for entering the DNA sequence.
- Submit button: A button labeled "Submit".

The footer of the page reads: "Copyright © 2009 - Powered by web2py".

web2py and Views

```
<h1>
Upload DNA Seq.
</h1>
```

```
{{=form}}
```

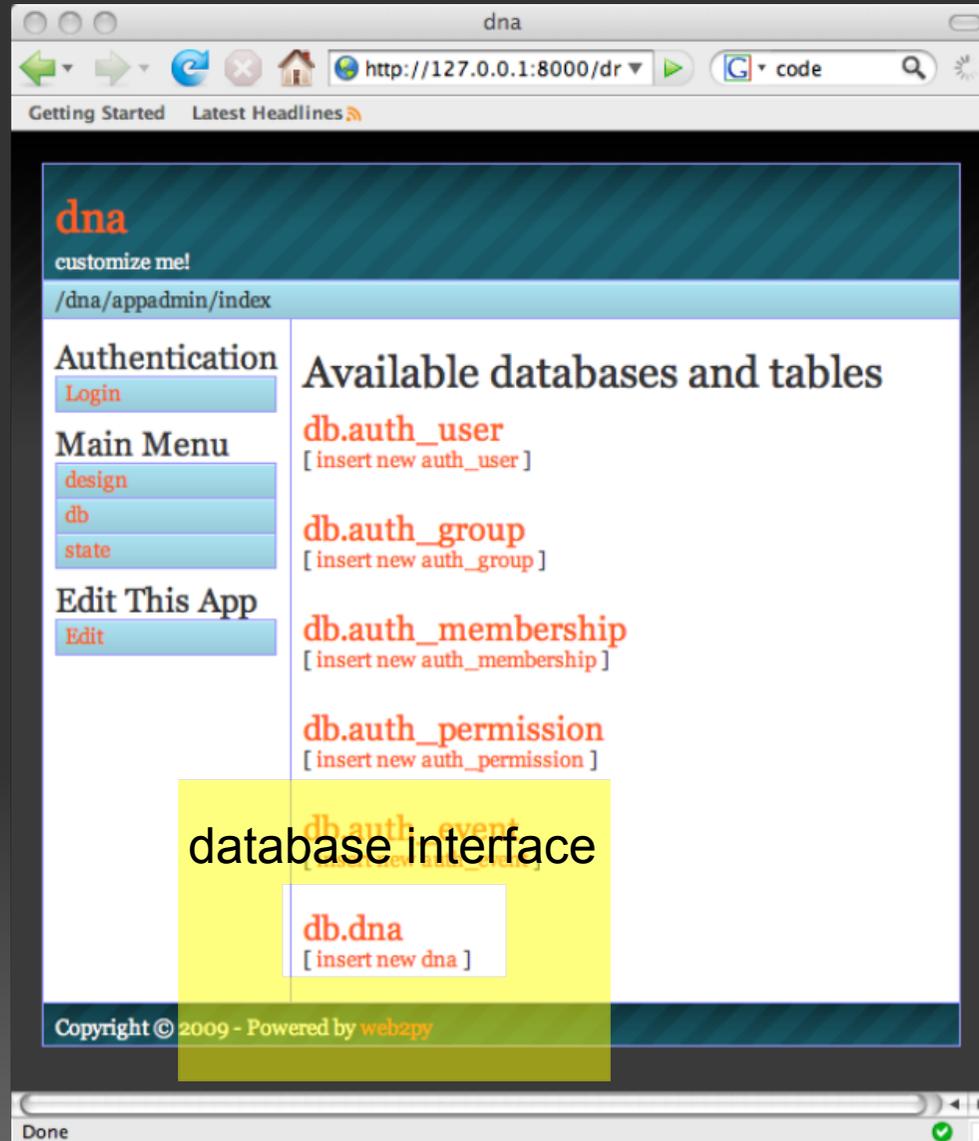
The screenshot displays a web browser window with the following elements:

- Browser Address Bar:** `http://127.0.0.1:8000/dna/default/upload_dna`
- Page Header:** **dna** customize me!
- URL:** `/dna/default/upload_dna`
- Left Sidebar:**
 - Authentication**
 - Login
 - Main Menu**
 - Index
 - Edit This App**
 - Edit
- Main Content Area (Yellow Background):**
 - Upload DNA Seq.**
 - Sequence:**
 - Submit** button
- Form Placeholder:** `{{=form}}`
- Footer:** Copyright © 2009 - Powered by **web2py**

web2py and Authentication

The screenshot shows a web browser window titled "dna" with the URL `http://127.0.0.1:8000/dna/default/upload_dna`. The browser's address bar also shows "code". The page content includes a sidebar on the left with the "dna" logo and navigation links: "Authentication" (with a "Login" button), "Main Menu" (with an "Index" button), and "Edit This App" (with an "Edit" button). The main content area features three forms: "Login" (with fields for "E-mail" containing "mdipierro@cs.depaul.edu" and "Password" containing "****", and a "Submit" button), "Upload DNA Seq." (with a "Sequence:" field and a "Submit" button), and "Register" (with fields for "First name:", "Last name:", "E-mail" (containing "mdipierro@cs.depaul.edu"), "Password" (containing "****"), and "Verify Password:", and a "Submit" button). A yellow highlight is present over the "Authentication" and "Main Menu" sections of the sidebar. A "Done" status bar is visible at the bottom left, and a green checkmark icon is at the bottom right.

web2py and AppAdmin



web2py web based IDE

design dna

Getting Started Latest Headlines

[web2py™] web based IDE

about errors logout help

Edit application "dna"

[models | controllers | views | languages | static | modules]

Models

the data representation, define database tables and sets

[database administration | sql.log]

- db.py [edit | delete]
- db_dna.py [edit | delete] defines tables dna
- menu.py [edit | delete]
- create file with filename: submit

Controllers

the application logic, each URL path is mapped in one exposed function in the controller

[shell | test | crontab]

- appadmin.py [edit | delete | test] exposes index, insert, download, csv, select, update, state
- default.py [edit | delete | test] exposes index, upload_dna, user, download, call
- create file with filename: submit

Views

Languages

Static files

Modules

additional code for your application

- __init__.py [edit]
- create file with filename: submit

Done

edit dna/models/db_dna.py

Getting Started Latest Headlines

[web2py™] admin site edit about errors logout help

Editing file "dna/models/db_dna.py"

[docs]

save Saved file hash: 126f1dfb2eccc20dd2ca59r Last saved on: Thu Sep 24 15:56:34 2009

```
1 db.define_table('dna',
2     Field('sequence'))
3
```

Position: Ln 1, Ch 1 Total: Ln 3, Ch 43

Done

Goal

- build a web based application
- store DNA sequences
- allow upload of DNA sequences
- allow analysis of DNA sequences
 - (reverse, count, align, etc.)
- allow plotting of results

Before we start

- download web2py from web2py.com
- unzip web2py and click on the executable
- when it asks for a password choose one
- visit <http://127.0.0.1:8000/admin> and login
- create a new “dna” application by:
 - type “dna” in the apposite box and press [submit]

Define model

in models/db_dna.py

```
import math, random, uuid, re
```

```
db.define_table('dna',  
    Field('name'),  
    Field('sequence', 'text'))
```

```
def random_gene(n):  
    return ''.join(['ATGC'[int(n+10*math.sin(n*k)) % 4] \  
        for k in range(10+n)])+'UAA'
```

```
def random_dna():  
    return ''.join([random_gene(random.randint(0,10)) \  
        for k in range(50)])
```

```
if not db(db.dna.id>0).count():  
    for k in range(100):  
        db.dna.insert(name=uuid.uuid4(), sequence=random_dna())
```

Define some algorithms

```
def find_gene_size(a):
    r=re.compile(' (UAA|UAG|UGA) (?P<gene>.*?) (UAA|UAG|UGA) ')
    return [(g.start(),len(g.group('gene')))\
            for g in r.finditer(a)]
```

```
def needleman_wunsch(a,b,p=0.97):
    """Needleman-Wunsch and Smith-Waterman"""
    z=[]
    for i,r in enumerate(a):
        z.append([])
    for j,c in enumerate(b):
        if r==c:
            z[-1].append(z[i-1][j-1]+1 if i*j>0 else 1)
        else:
            z[-1].append(p*max(z[i-1][j] if i>0 else 0,
                               z[i][j-1] if j>0 else 0))
    return z
```

in models/matplotlib_helpers.py

```
import random, cStringIO
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

def plot(title='title',xlab='x',ylab='y',data={}):
    fig=Figure()
    fig.set_facecolor('white')
    ax=fig.add_subplot(111)
    if title: ax.set_title(title)
    if xlab: ax.set_xlabel(xlab)
    if ylab: ax.set_ylabel(ylab)
    legend=[]
    keys=sorted(data)
    for key in keys:
        stream = data[key]
        (x,y)=([],[])
    for point in stream:
        x.append(point[0])
        y.append(point[1])
    ell=ax.hist(y,20)
    canvas=FigureCanvas(fig)
    response.headers['Content-Type']='image/png'
    stream=cStringIO.StringIO()
    canvas.print_png(stream)
    return stream.getvalue()
```

Define actions

in controllers/default.py

```
def index():
    rows=db(db.dna.id).select(db.dna.id,db.dna.name)
    return dict(rows=rows)

@auth.requires_login()
def gene_size():
    dna = db.dna[request.args(0)] or \
        redirect(URL(r=request,f='index'))
    lengths = find_gene_size(dna.sequence)
    return hist(data={'Lengths':lengths})
```

Define Views

in views/default/index.html

```
{{extend 'layout.html'}}
```

```
<a href="{%=URL(r=request, f='compare') %}">compare</a>
```

```
<ul>
```

```
  {{for row in rows:}}
```

```
  <li>{{=row.name}}
```

```
  [<a href="{%=URL(r=request, f='gene_size', args=row.id) %}">gene sizes</a>]
```

```
</li>
```

```
  {{pass}}
```

```
</ul>
```

Try it

Getting Started Latest Headlines

dna

customize me!

/dna/default/index

Authentication

User: Massimo

Main Menu

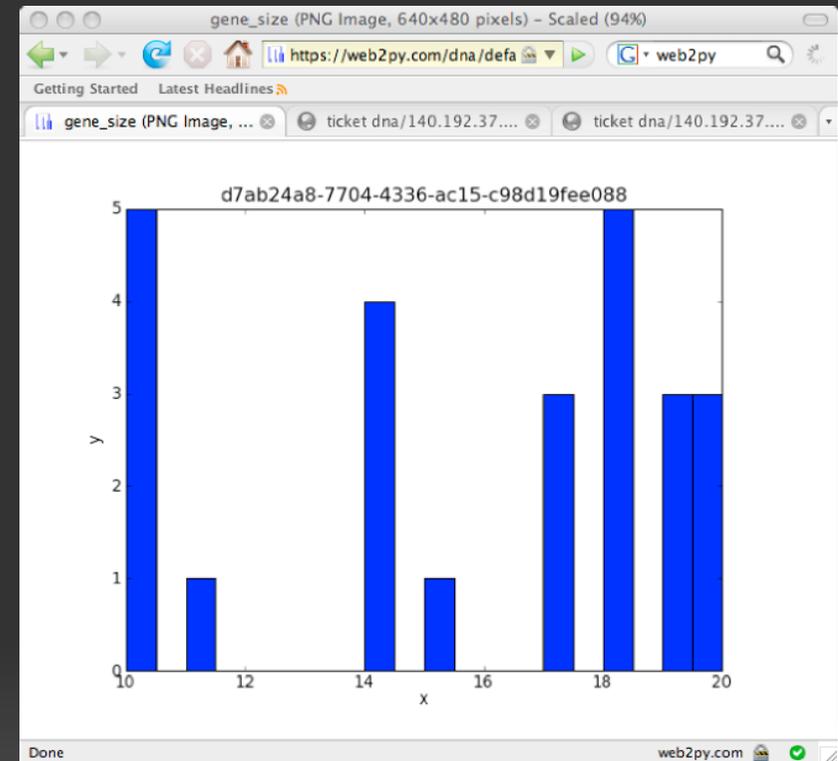
Index

Edit This App

Edit

compare

- [d63055b2-bd97-4451-b610-3d32e1497956\[gene sizes \]](#)
- [7874da90-889b-49f3-8c42-493180220ca8\[gene sizes \]](#)
- [bb86aaaf-03dc-4bbf-b94a-11fc9114849a\[gene sizes \]](#)
- [463ed2a5-11e6-4349-8dfe-1ed00112b011\[gene sizes \]](#)
- [bb307835-e079-46dd-8e19-0ea0b5202c4d\[gene sizes \]](#)
- [d7ab24a8-7704-4336-ac15-c98d19fee088\[gene sizes \]](#)
- [9ab42178-ca63-4bf8-9c78-76c65b70c5b7\[gene sizes \]](#)
- [1b4efc8f-e583-4384-8e0e-36edoed30d8b\[gene sizes \]](#)
- [68ada686-7077-4ab0-87c4-13d010444a8a\[gene sizes \]](#)
- [df949534-560f-4fac-964f-49aed1cc03f9\[gene sizes \]](#)
- [98fd5e5a-0324-4879-b5bc-119b3a409841\[gene sizes \]](#)
- [1727e107-0575-4dd9-a934-923f8df139ea\[gene sizes \]](#)
- [720a2239-8d9a-419b-b716-341cob7643b8\[gene sizes \]](#)
- [f3d260f6-878d-4741-b8ae-8e7fb3b6c455\[gene sizes \]](#)
- [13ff0462-e88a-4523-9472-7cb3ed91ad1f\[gene sizes \]](#)
- [aa64d8bc-dfoe-4022-9c9f-86d39323218e\[gene sizes \]](#)
- [6f2e2fb9-35f6-43ff-9e65-3dd37829df90\[gene sizes \]](#)
- [878bed89-ef4-4761-97b1-233983f17cdd\[gene sizes \]](#)
- [f68b637f-9219-40ba-af87-9e5c63ec4f1b\[gene sizes \]](#)
- [7be32644-18fd-4406-91c8-d017b78fca5f\[gene sizes \]](#)
- [1c2c8886-a7c0-4fa5-aboa-a67949094cff\[gene sizes \]](#)
- [969e442f-c102-4e7f-a95e-84257bbbe4ed\[gene sizes \]](#)
- [7678aee3-0780-4816-9a6b-255223ef4839\[gene sizes \]](#)
- [4c43de78-df38-4369-972a-4b9188bd2133\[gene sizes \]](#)



in models/matplotlib_helpers.py

```
def pcolor2d(title='title',xlab='x',ylab='y',
z=[[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7]]):
    fig=Figure()
    fig.set_facecolor('white')
    ax=fig.add_subplot(111)
    if title: ax.set_title(title)
    if xlab: ax.set_xlabel(xlab)
    if ylab: ax.set_ylabel(ylab)
    image=ax.imshow(z)
    image.set_interpolation('bilinear')
    canvas=FigureCanvas(fig)
    response.headers['Content-Type']='image/png'
    stream=cStringIO.StringIO()
    canvas.print_png(stream)
    return stream.getvalue()
```

Define Actions

in controllers/default.py

```
def needleman_wunsch_plot():
    dna1 = db.dna[request.vars.sequence1]
    dna2 = db.dna[request.vars.sequence2]
    z = needleman_wunsch(dna1.sequence, dna2.sequence)
    return pcolor2d(z=z)

def compare():
    form = SQLFORM.factory(
        Field('sequence1', db.dna,
            requires=IS_IN_DB(db, 'dna.id', '%(name)s')),
        Field('sequence2', db.dna,
            requires=IS_IN_DB(db, 'dna.id', '%(name)s')))
    if form.accepts(request.vars):
        image=URL(r=request, f='needleman_wunsch_plot',
            vars=form.vars)
    else:
        image=None
    return dict(form=form, image=image)
```

Define Views

in views/default/compare.html

```
{{extend 'layout.html'}}
```

```
{{=form}}
```

```
{{if image:}}
```

```
Sequence1 = {{=db.dna[request.vars.sequence1].name}}<br/>
```

```
Sequence2 = {{=db.dna[request.vars.sequence2].name}}<br/>
```

```

```

```
{{pass}}
```

Try it

Browser window showing a web application interface for comparing DNA sequences. The URL is `https://web2py.com/dna/default/compare`.

The interface includes a sidebar with navigation links: **Authentication** (User: Massimo), **Main Menu** (Index), and **Edit This App** (Edit).

The main content area displays two input fields for DNA sequences, both containing the same identifier: `0bda5686-564f-436b-a602-eb50a5da40bd`. A **Submit** button is located below the inputs.

Below the submit button, the selected sequences are displayed:

```
Sequence1 = 0bda5686-564f-436b-a602-eb50a5da40bd  
Sequence2 = 1127a5f9-08c6-4e2c-b3bb-051212909551
```

The visualization is a heatmap titled "title" showing the alignment of the two sequences. The x-axis is labeled "x" and the y-axis is labeled "y", both ranging from 0 to 800. The heatmap displays a complex pattern of yellow and orange highlights against a blue background, indicating regions of high similarity or alignment between the two DNA sequences. Two black arrows point to a specific region of high similarity in the lower right quadrant of the plot, around x=400 and y=750.

The browser status bar at the bottom shows "Done" and the address "web2py.com".

Resources

Python

- <http://www.python.org/>
 - all the current documentation, software, tutorials, news, and pointers to advice you'll ever need

GPAW

- <https://wiki.fysik.dtu.dk/gpaw/>
 - GPAW documentation and code

SciPy and NumPy

- <http://numpy.scipy.org/>
 - The official NumPy website
- <http://conference.scipy.org/>
 - The annual SciPy conference
- <http://www.enthought.com/>
 - Enthought, Inc. the commercial sponsors of SciPy, NumPy, Chaco, EPD and more

Matplotlib

- <http://matplotlib.sourceforge.net/>
 - best 2D package on the planet

mpi4py

- <http://mpi4py.scipy.org/>

Yet More Resources

Tau

- <http://www.cs.uoregon.edu/research/tau/home.php>
 - official open source site
- <http://www.paratools.com/index.php>
 - commercial tools and support for Tau

web2py

- <http://www.web2py.com/>
 - web framework used in this tutorial

Hey! There's a Python BOF

Python for High Performance and Scientific Computing

Primary Session Leader:

Andreas Schreiber (German Aerospace Center)

Secondary Session Leaders:

William R. Scullin (Argonne National Laboratory) Steven

Brandt (Louisiana State University) James B. Snyder (Northwestern

University) Nichols A. Romero (Argonne National Laboratory)

Birds-of-a-Feather Session

Wednesday, 05:30PM - 07:00PM Room A103-104

Abstract:

The Python for High Performance and Scientific Computing BOF is intended to provide current and potential Python users and tool providers in the high performance and scientific computing communities a forum to talk about their current projects; ask questions of experts; explore methodologies; delve into issues with the language, modules, tools, and libraries; build community; and discuss the path forward.

Let's review!

Questions?

Acknowledgments

This work is supported in part by the resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Extended thanks to

- Northwestern University
- De Paul University
- the families of the presenters
- Sameer Shende, ParaTools, Inc.
- Enthought, Inc. for their continued support and sponsorship of SciPy and NumPy
- Lisandro Dalcin for his work on mpi4py and tolerating a lot of questions

- the members of the Chicago Python User's Group (ChiPy) for allowing us to ramble on about science and HPC
- the Python community for their feedback and support
- CCT at LSU
- numerous others at HPC centers nationwide