HPC Python Programming

Ramses van Zon

SciNet HPC Consortium

IHPCSS, June 29, 2016

SciNet compute canada | calcul canada

In this session. . .

1. Performance and Python
2. Profiling tools for Python
3. Numpy: Fast arrays for python
4. Multicore computations:
   - **Numexpr**
   - **Threading**
   - **Multiprocessing**
   - **Mpi4py**
   - **IPython Parallel**
   - **Apache Spark**

SciNet compute | calcul canada

Getting started

SciNet compute canada | calcul canada

Packages and code

Requirements for this session

If following along on your own laptop, you need the following packages:

- numpy
- scipy
- numexpr
- matplotlib
- psutil
- line_profiler
- memory_profiler
- theano
- pyzmq
- mpi4py
- ipyparallel or IPython.parallel

Get the code and setup files on Bridges

Code and installation can be copies from my account on Bridges. It's in the directory `/home/rzon/hpcpy`. The code accompanying this session is in the code subdirectory.

Own laptop/Not on Bridges?

```
$ git clone https://gitrepos.scinet.utoronto.ca/public/hpcpy.git
```

compute canada | calcul canada

Setting up for today's class (Bridges)

To get set up for today's class, perform the following steps.

1. Login to Bridges

```
$ ssh -Y USERNAME@bridges.psc.edu
```

2. Install code and software to your own directory

```
$ cp -r /home/rzon/hpcpy .
$ cd hpcpy/code
$ source setup
```

The last command will install a few packages into your local account, so as to satisfy the requirements, and will load the correct modules.

3. Request an interactive session on a compute node

```
$ interact -n 28
$ source setup
$ make help
```

Introduction

SciNet compute canada | calcul canada

Performance and Python

- Python is a high-level, interpreted language.
- Those defining features are often at odds with "high performance".
- But the development in Python can be substantially easier (and thus faster) than compiled languages.
- In this session, we will explore when using Python still makes sense and xhow to get the most performance out of it, without loosing the flexibility and ease of development.

SciNet compute canada | calcul canada

What would make Python not "high performance"?

Interpreted language:

- Translation to machine language happens line-by-line as the script is read.
- Repeated lines are no faster.
- Cross-line optimizations are not possible.

SciNet compute | calcul canada

What would make Python not "high performance"?

Interpreted language:

- Translation to machine language happens line-by-line as the script is read.
- Repeated lines are no faster.
- Cross-line optimizations are not possible.

Dynamic language:

- Types are part of the data: extra overhead
- Memory management is automatic. Behind the scene that means reference counting and garbage collection.
- All this also interfers with optimal streaming of data to processor, which interfers with maximum performance.

canada | canada

Example: 2D diffusion equation

Suppose we are interested in the time evolution of the two-dimension diffusion equation:

$$\frac{\partial p(x,y,t)}{\partial t} = D\left(\frac{\partial^2 p(x,y,t)}{\partial x^2} + \frac{\partial^2 p(x,y,t)}{\partial y^2}\right),$$

on domain $[x_1, x_2] \otimes [x_1, x_2]$, with $P(x,y,t) = 0$ at all times for all points on the domain boundary, and for some given initial condition $p(x,y,t) = p_0(x,y)$.

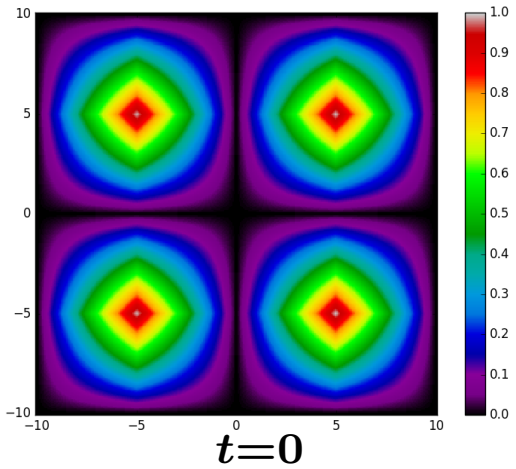
Here:

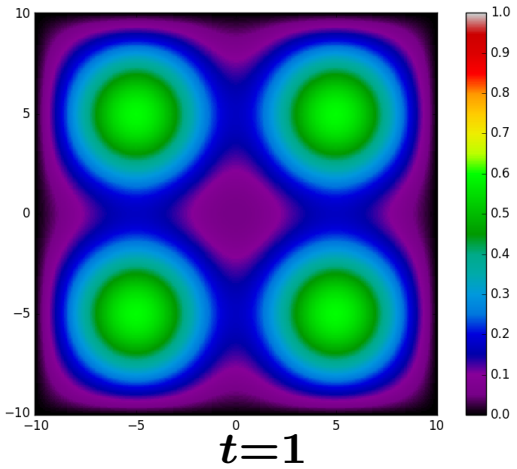
- P: density
- x, y: spatial coordinates
- t: time
- D: diffusion constant

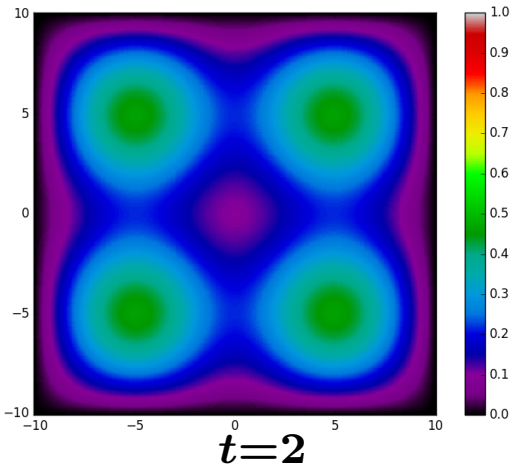
SciNet compute canada | calcul canada

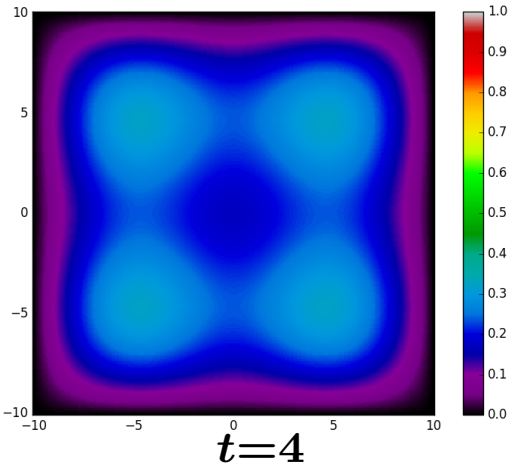
Example: 2D diffusion, result

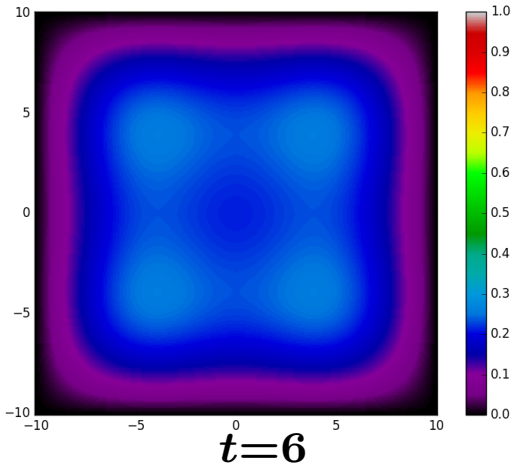
$x_1 = -10$, $x_2 = 10$, $D = 1$, four-peak initial condition.

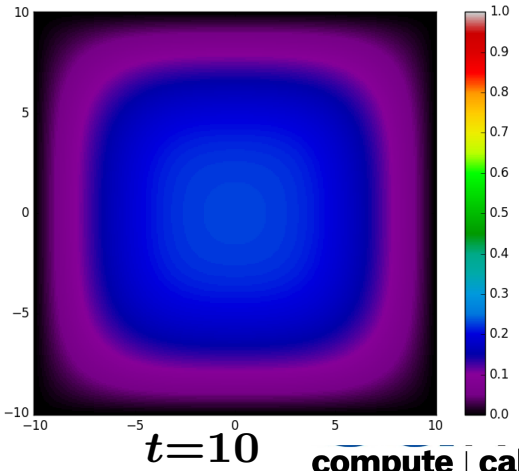








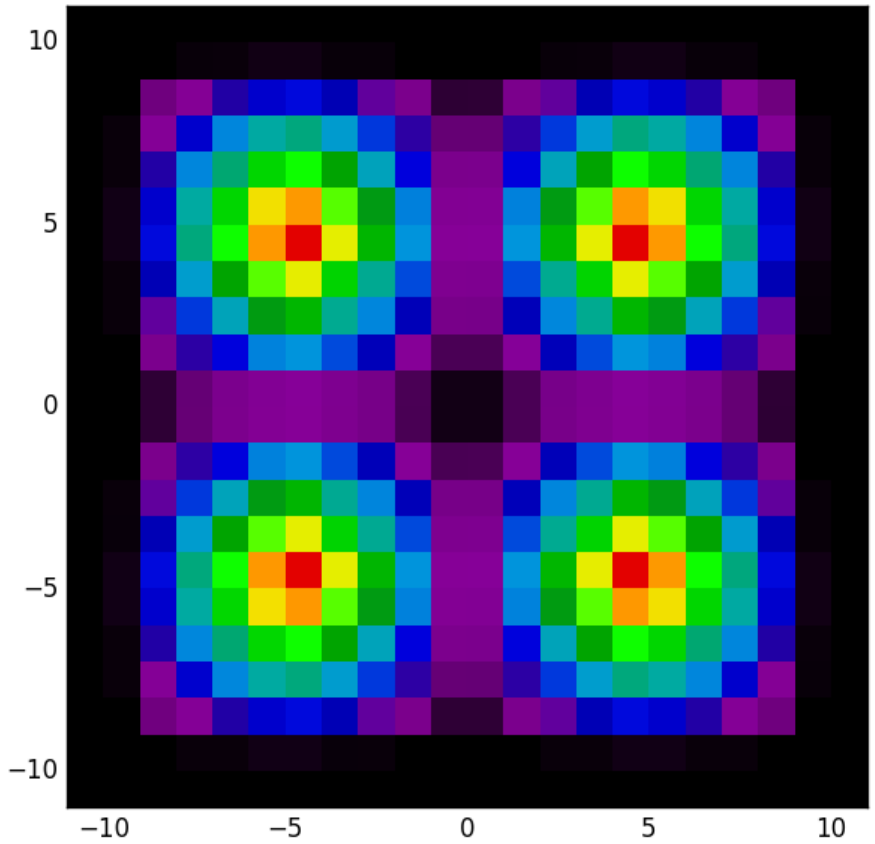




compute canada | calcul canada

Example: 2D diffusion, algorithm

- Discretize space in both directions (points dx apart)
- Replace derivatives with finite differences.
- Explicit finite time stepping scheme (time step set by dx)
- For graphics: Matplotlib for python, pgplot for c++/fortran, every `outtime` time units



Parameters in file diff2dparams.py

SciNet compute | calcul canada

Example: 2D diffusion, parameters

The fortran, C++ and python codes all read the same files (by some special tricks).

diff2dparams.py

```
D         =   1.0;
x1        = -10.0;
x2        =  10.0;
runtime   =  15.0;
dx        =  0.0667;
outtime   =  0.5;
graphics  =  False;
```

SciNet compute canada | calcul canada

Example: 2D diffusion, performance

The files `diff2d.cpp`, `diff2.f90` and `diff2d.py` contain the same algorithm, in C++, Fortran, and Python, respectively.

Example: 2D diffusion, performance

The files `diff2d.cpp`, `diff2.f90` and `diff2d.py` contain the same algorithm, in C++, Fortran, and Python, respectively.

```
$ etime() { /usr/bin/time -f "Elapsed: %e seconds" $@; }
$ etime make diff2d_cpp.ex diff2d_f90.ex
g++ -c -std=c++11 -O3 -o diff2d_cpp.o diff2d.cpp
gfortran -c -O3 -o pgplot90.o pgplot90.f90
...
Elapsed: 1.80 seconds
```

SciNet compute | calcul canada

Example: 2D diffusion, performance

The files `diff2d.cpp`, `diff2.f90` and `diff2d.py` contain the same algorithm, in C++, Fortran, and Python, respectively.

```
$ etime() { /usr/bin/time -f "Elapsed: %e seconds" $@; }
$ etime make diff2d_cpp.ex diff2d_f90.ex
g++ -c -std=c++11 -O3 -o diff2d_cpp.o diff2d.cpp
gfortran -c -O3 -o pgplot90.o pgplot90.f90
...
Elapsed: 1.80 seconds
```

```
$ etime ./diff2d_cpp.ex > output_c.txt
Elapsed: 2.44 seconds
$ etime ./diff2d_f90.ex > output_f.txt
Elapsed: 2.37 seconds
$ etime python diff2d.py > output_n.txt
Elapsed: 599.90 seconds
```

This doesn't look too promising for Python for HPC...

Then why do we bother with Python?

```
import numpy as np
from diff2dplot import plotdens
D         =   1.0;
x1        = -10.0;
x2        =  10.0;
runtime   =  15.0;
dx        =  0.0666;
outtime   =  0.5;
nrows     = int((x2-x1)/dx)
npnts     = nrows + 2
xm        = (x1+x2)/2
dx        = (x2-x1)/nrows
dt        = 0.25*dx**2/D
nsteps    = int(runtime/dt)
nper      = int(outtime/dt)
x=np.linspace(x1-dx,x2+dx,npnts)
dens1 = np.zeros((npnts,npnts))
dens2 = np.zeros((npnts,npnts))
lapl  = np.zeros((npnts,npnts))
simtime = 0
```

```
for i in xrange(1,npnts-1):
 a=1-abs(1-4*abs((x[i]-xm)/(x2-x1)
 for j in xrange(1,npnts-1):
  b=1-abs(1-4*abs((x[j]-xm)/(x2-x1
  dens1[i][j]=a*b
print(simtime)
plotdens(dens1,x[0],x[-1],True)
for s in xrange(nsteps):
 lapl[1:nrows+1,1:nrows+1]=(
    dens1[2:nrows+2,1:nrows+1]
   +dens1[0:nrows+0,1:nrows+1]
   +dens1[1:nrows+1,2:nrows+2]
   +dens1[1:nrows+1,0:nrows+0]
   -4*dens1[1:nrows+1,1:nrows+1])
 dens2[:,:]=dens1+D/dx**2*dt*lapl
 dens1,dens2 = dens2,dens1
 simtime += dt
 if (s+1)%nper == 0:
  print(simtime)
  plotdens(dens1,x[0],x[-1])
```

Then why do we bother with Python?

- Python lends itself easily to writing clear, concise code.
  (2d diffusion fits on one slide!)
- Python is very flexible: large set of very useful packages.
- Easy of use $\longrightarrow$ shorter development time
- Python's performance hit is most prominant on 'tightly coupled' calculation on fundamental data types that are known to the cpu (integers, doubles), which is exactly the case for the 2d diffusion.
- It does much less worse on file I/O, text comparisons, list manipularions etc.
- Hooks to compiled libraries to remove worst performance pitfalls.
- Once the performance isn't too bad, we can start thinking of parallelization, i.e., using more cpu cores working on the\ same problem.

SciNet compute | calcul canada

Performance tuning tools for Python

SciNet compute canada | calcul canada

CPU performance

- Performance is about maximizing the utility of a resource.
- This could be cpu processing power, memory, network, file I/O, etc.
- Let's focus on **cpu performance** first.

CPU Profiling by function

- To consider the cpu performance of functions, but not of individual lines in your code, there is the package called `cProfile`.

CPU Profiling by line

- To find cpu performance bottlenecks by line of code, there is package called `line_profiler`

SciNet compute canada | calcul canada

cProfile

- Use `cProfile` or `profile` to know in which functions your script spends its time.
- You usually do this on a smaller but representative case.
- The code should be reasonably modular, i.e., with separate functions for different tasks, for cProfile to be useful.

Example

```
$ python -m cProfile -s cumulative diff2d_numpy.py
...
         92333 function calls (92212 primitive calls) in 4.236 sec
   Ordered by: cumulative time
   ncalls  tottime  percall  cumtime  percall filename:lineno(func
        1    0.000    0.000    4.236    4.236 diff2d_numpy.py:9(<m
        1    4.171    4.171    4.176    4.176 diff2d_numpy.py:15(m
        3    0.010    0.003    0.078    0.026 __init__.py:1(<modul
        1    0.003    0.003    0.059    0.059 __init__.py:106(<mod
        1    0.000    0.000    0.045    0.045 add_newdocs.py:10(<m
        1    0.000    0.000    0.034    0.034 type_check.py:3(<mod
```

line_profiler

- Use `line_profiler` to know, line-by-line, where your script spends its time.
- You usually do this on a smaller but representative case.
- First thing to do is to have your code be in a function.
- You also need to include modify your script slightly:
  - Decorate your function with @profile
  - Run your script on the command line with

```
$ kernprof -l -v SCRIPTNAME
```

line_profiler script instrumentation

Script before:

```
x=[1.0]*(2048*2048)
a=str(x[0])
a+="\nis a one\n"
del x
print(a)
```

SciNet compute canada | calcul canada

line_profiler script instrumentation

Script before:

```
x=[1.0]*(2048*2048)
a=str(x[0])
a+="\nis a one\n"
del x
print(a)
```

Script after:

```
#file: profileme.py
@profile
def profilewrapper():
  x=[1.0]*(2048*2048)
  a=str(x[0])
  a+="\nis a one\n"
  del x
  print(a)
profilewrapper()
```

Run at the command line:

```
$ kernprof -l -v profileme.py
```

SciNet compute | calcul canada

Output of line_profiler

```
1.0
is a one

Wrote profile results to profileme.py.lprof
Timer unit: 1e-06 s

Total time: 0.03296 s
File: profileme.py
Function: profilewrapper at line 2

Line #    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def profilewrapper():
     4         1     23882  23882.0     72.5      x=[1.0]*(2048*2048)
     5         1        16     16.0      0.0      a=str(x[0])
     6         1         1      1.0      0.0      a+="\nis a one\n"
     7         1      9024   9024.0     27.4      del x
     8         1        37     37.0      0.1      print(a)
```

Memory performance

Why worry about this?

SciNet compute | calcul canada

Memory performance

Why worry about this?

Once your script runs out of memory, one of a number of things may happen:

- Computer may start using the harddrive as memory: **very slow**
- Your application crashes
- Your (compute) node crashes

SciNet compute canada | calcul canada

Memory performance

Why worry about this?

Once your script runs out of memory, one of a number of things may happen:

- Computer may start using the harddrive as memory: **very slow**
- Your application crashes
- Your (compute) node crashes

How could you run out of memory?

- You're not quite sure how much memory you program takes.
- Python objects may take more memory that expected.
- Some functions may temporarily use extra memory.
- Python relies on a garbage collector to clean up unused variables.

Garbage collector

- Python uses garbage collector to clean up un-needed variables
- You can force the garbage collection to run at any time by running:

```
>>> import gc
>>> collect = gc.collect()
```

- Running gc by hand should only be done in specific circumstances.
- You can also remove objects with del (if object larger than 32MB):

```
>>> x = [0,0,0,0]
>>> del x
>>> print (x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

But how would you know when the memory usage is problematic?

SciNet compute | calcul canada

memory_profiler

- This module/utility monitors the python memory usage and its changes throughout the run.
- Good for catching memory leaks and unexpectedly large memory usage.
- Needs same instrumentation as line profiler.
- Requires the psutil module (at least on windows, but helps on linux/mac too).

SciNet compute canada | calcul canada

memory_profiler, details

Your decorated script is usable by memory profiler.

You run your script through the profiler with the command

```
$ python -m memory_profiler profileme.py
```

memory_profiler, details

Your decorated script is usable by memory profiler.

You run your script through the profiler with the command

```
$ python -m memory_profiler profileme.py
```

```
1.0
is a one

Filename: profileme.py

Line #    Mem usage    Increment   Line Contents
================================================
     2   19.230 MiB    0.000 MiB   @profile
     3                             def profilewrapper():
     4   51.238 MiB   32.008 MiB     x=[1.0]*(2048*2048)
     5   51.242 MiB    0.004 MiB     a=str(x[0])
     6   51.242 MiB    0.000 MiB     a+="\nis a one\n"
     7   19.238 MiB  -32.004 MiB     del x
     8   19.242 MiB    0.004 MiB     print(a)
```

Hands-on

Profile the diff2d.py code

- Reduce the resolution in diff2dparams.py, i.e., increase `dx` to 0.1.
- In the same file, set `graphics=False`.
- Add @profile to the main function
- Run this through both the line and memory profilers.
  - What lines cause the most memory usage?
  - What lines cause the most cpu usage?

SciNet compute | calcul canada

Numpy: faster numerical arrays for python

SciNet compute | calcul canada

Lists aren't the ideal data type

Lists can do funny things that you don't expect, if you're not careful.

- Lists are just a collection of items, of any type.
- If you do mathematical operations on a list, you won't get what you expect.
- These are not the ideal data type for scientific computing.
- Arrays are a much better choice, but are not a native Python data type.

```
>>> a = [1,2,3,4]
>>> a
[1, 2, 3, 4]
>>> b = [3,5,5,6]
>>> b
[3, 5, 5, 6]
>>> 2*a
[1, 2, 3, 4, 1, 2, 3, 4]
>>> a+b
[1, 2, 3, 4, 3, 5, 5, 6]
```

SciNet compute canada | calcul canada

Useful arrays: NumPy

- Almost everything that you want to do starts with NumPy.
- Contains arrays of various types and forms: zeros, ones, linspace, etc.

```
>>> from numpy import zeros, ones
>>> zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
>>> ones(5, dtype=int)
array([1, 1, 1, 1, 1])
>>> zeros([2,2])
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> from numpy import arange
>>> from numpy import linspace
>>> arange(5)
array([0, 1, 2, 3, 4])
>>> linspace(1,5)
array([ 1.        ,  1.08163265,
        1.40816327,  1.48979592,
        1.81632653,  1.89795918,
        2.2244898 ,  2.30612245,
        2.63265306,  2.71428571,
        3.04081633,  3.12244898,
        3.44897959,  3.53061224,
        3.85714286,  3.93877551,
        4.26530612,  4.34693878,
        4.67346939,  4.75510204,
>>> linspace(1,5,6)
array([ 1. ,  1.8,  2.6,  3.4,  4.
```

Accessing array elements

Elements of arrays are accessed using square brackets.

- Python is *row-major* (like C++, Mathematica), NOT *column major* (like Fortran, MATLAB, R)
- This means the first index is the row, not the column.
- Indexing starts at zero.

Accessing array elements

Elements of arrays are accessed using square brackets.

- Python is *row-major* (like C++, Mathematica), NOT *column major* (like Fortran, MATLAB, R)
- This means the first index is the row, not the column.
- Indexing starts at zero.

```
>>> from numpy import *
>>> zeros([2,3])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> a = zeros([2,3])
>>> a[1,2] = 1
>>> a[0,1] = 2
>>> a
array([[ 0.,  2.,  0.],
       [ 0.,  0.,  1.]])
```

SciNet compute | calcul canada

Accessing array elements

Elements of arrays are accessed using square brackets.

- Python is *row-major* (like C++, Mathematica), NOT *column major* (like Fortran, MATLAB, R)
- This means the first index is the row, not the column.
- Indexing starts at zero.

```
>>> from numpy import *
>>> zeros([2,3])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> a = zeros([2,3])
>>> a[1,2] = 1
>>> a[0,1] = 2
>>> a
array([[ 0.,  2.,  0.],
       [ 0.,  0.,  1.]])
```

```
>>> a[2,1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <modu
IndexError: index 2 is out of bour
```

SciNet compute | calcul canada

Copying array variables

Use caution when copying array variables. There's a 'feature' here that is unexpected.

```
>>> a = 10; b = a; a = 20
>>> a, b
(20, 10)
```

SciNet compute canada | calcul canada

Copying array variables

Use caution when copying array variables. There's a 'feature' here that is unexpected.

```
>>> a = 10; b = a; a = 20
>>> a, b
(20, 10)
```

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...                [2,3,4]])
>>> b = a
>>> a[1,0] = 16
>>> a
array([[ 1,   2,   3],
       [16,   3,   4]])
>>> b
array([[ 1,   2,   3],
       [16,   3,   4]])
```

SciNet compute canada | calcul canada

Copying array variables

Use caution when copying array variables. There's a 'feature' here that is unexpected.

```
>>> a = 10; b = a; a = 20
>>> a, b
(20, 10)
```

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...                [2,3,4]])
>>> b = a
>>> a[1,0] = 16
>>> a
array([[ 1,   2,   3],
       [16,   3,   4]])
>>> b
array([[ 1,   2,   3],
       [16,   3,   4]])
```

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...                [2,3,4]])
>>> b = a.copy()
>>> a[1,0] = 16
>>> a
array([[ 1,  2,  3],
       [16,  3,  4]])
>>> b
array([[1, 2, 3],
       [2, 3, 4]])
```

Matrix arithmetic

vector-vector & vector-scalar multiplication

1-D arrays are often called 'vectors'.

- When vectors are multiplied you get element-by-element multiplication.
- When vectors are multiplied by a scalar (a 0-D array), you also get element-by-element multiplication.

```
>>> import numpy as np
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> b = np.arange(4.) + 3
>>> b
array([ 3.,  4.,  5.,  6.])
>>> c = 2
>>> c
2
>>> a * b
array([  0.,   4.,  10.,  18.])
>>> a * c
array([0, 2, 4, 6])
>>> b * c
array([  6.,   8.,  10.,  12.])
```

canada | canada

Matrix-vector multiplication

A 2-D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication gives element-by-element multiplication.
- With numpy, matrix-vector multiplication DOES NOT give the standard result!

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...               [2,3,4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])
>>> b = np.arange(3) + 1
>>> b
array([1, 2, 3])
>>> a * b
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

Numpy DOES NOT compute this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 \\ a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 \end{bmatrix}$$

SciNet compute | calcul canada

Matrix-vector multiplication

A 2-D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication gives element-by-element multiplication.
- With numpy, matrix-vector multiplication DOES NOT give the standard result!

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...               [2,3,4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])
>>> b = np.arange(3) + 1
>>> b
array([1, 2, 3])
>>> a * b
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

Numpy DOES compute this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} * b_1 & a_{12} * b_2 & a_{13} * b_3 \\ a_{21} * b_1 & a_{22} * b_2 & a_{23} * b_3 \end{bmatrix}$$

SciNet compute canada | calcul canada

Matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication doesn't work as expected either, instead doing the same thing as vector-vector multiplication.

```
>>> import numpy as np
>>> a = np.array([[1,2],
...               [4,3]])
>>> b = np.array([[1,2],
...               [4,3]])
>>> a
array([[1, 2],
       [4, 3]])
>>> a * b
array([[ 1,  4],
       [16,  9]])
```

Numpy DOES NOT do this:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}$$

SciNet compute canada | calcul canada

Matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication doesn't work as expected either, instead doing the same thing as vector-vector multiplication.

```
>>> import numpy as np
>>> a = np.array([[1,2],
...               [4,3]])
>>> b = np.array([[1,2],
...               [4,3]])
>>> a
array([[1, 2],
       [4, 3]])
>>> a * b
array([[ 1,  4],
       [16,  9]])
```

Numpy DOES NOT do this:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} * b_{11} & a_{12} * b_{12} \\ a_{21} * b_{21} & a_{22} * b_{22} \end{bmatrix}$$

SciNet compute canada | calcul canada

How to fix the matrix algebra?

There are two solutions to these matrix multiplication problems.

- The specially built-in array fixes (using 'array' types).
- The matrix module (using 'matrix' types).

The latter option is a bit clunkier, so we recommend the 'fixes'.

```
>>> import numpy as np
>>> a = np.array([[1,2],
...               [4,3]])
>>> b = np.array([[1,2],
...               [4,3]])
>>> a
array([[1, 2],
       [4, 3]])
```

```
>>> a.transpose()
array([[1, 4],
       [2, 3]])
>>> np.dot(a.transpose(), b)
array([[17, 14],
       [14, 13]])
>>> np.dot(b, a.transpose())
array([[ 5, 10],
       [10, 25]])
>>> c = np.arange(2) + 1
>>> np.dot(a,c)
array([5, 10])
```

SciNet compute canada | calcul canada

Does changing to numpy really help?

Let's return to our 2D diffusion example.

Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 588.68 seconds
```

Does changing to numpy really help?

Let's return to our 2D diffusion example.

Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 588.68 seconds
```

Numpy implementation:

```
$ etime python diff2d_numpy.py > output_n.txt
Elapsed: 20.97 seconds
```

Yeah! About 30× faster.

SciNet compute | calcul canada

Does changing to numpy really help?

Let's return to our 2D diffusion example.

Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 588.68 seconds
```

Numpy implementation:

```
$ etime python diff2d_numpy.py > output_n.txt
Elapsed: 20.97 seconds
```

Yeah! About 30× faster.

However, this is what the compiled versions do:

```
$ etime ./diff2d_cpp.ex > output_c.txt
Elapsed: 2.48 seconds
$ etime ./diff2d_f90.ex > output_f.txt
Elapsed: 2.16 seconds
```

What about cython?

- Cython is a compiler for python code
- Almost all python is valid cython
- Typically used for packages, to be used in regular python scripts.
- It is important to realize that the compilation preserves the pythonic nature of the language, i.e, garbage collection, range checking, reference counting, etc, are still done: no performance enhancement.

```
$ etime python diff2d_numpy.py        > output_n.txt
Elapsed: 21.61 seconds
$ etime python diff2d_numpy_cython.py > output_nc.txt
Elapsed: 22.20 seconds
```

- If you want to get around that, you need to use Cython specific extentions that essential use c types.
- From that point on, though, it isn't really python anymore, just a very convenient way to write compiled python extensions.

Parallel Python

Parallel Python

We will look at a number of approached to parallel programming with Python:

Package	Functionality
numexpr	threaded parallelization of certain numpy expressions
fork	create copies of existing process
threads	create threads sharing memory
multiprocessing	create processes that behave more like threads
mpi4py	message passing between processes
ipyparallel	framework of controling parallel workers
spark	framework of controling parallel workers

SciNet compute | calcul canada

Numexpr

The numexpr package

The numexpr package is useful if you're doing matrix algebra:

- It is essentially a just-in-time compiler for NumPy.
- It takes matrix expressions, breaks things up into threads, and does the calculation in parallel.
- Somewhat awkwardly, it takes it's input in as a string.
- In some situations using numexpr can significantly speed up your calculations.

SciNet compute canada | calcul canada

Numexpr in a nutshell

- Give it an array artihmetic expression, and it will compile and run it, and return or store the output.
- Supported operators:
  `+`, `-`, `*`, `/`, `**`, `%`, `<<`, `>>`, `<`, `<=`, `==`, `!=`, `>=`, `>`, `&`, `|`, `~`
- Supported functions:
  `where`, `sin`, `cos`, `tan`, `arcsin`, `arccos` `arctan`, `arctan2`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh` `arctanh`, `log`, `log10`, `log1p`, `exp`, `expm1`, `sqrt`, `abs`, `conj`, `real`, `imag`, `complex`, `contains`.
- Supported reductions:
  `sum`, `product`

SciNet compute | calcul canada

Using the numexpr package

Without numexpr:

```
>>> from etime import etime
>>> import numpy as np
>>> import numexpr as ne
>>> a = np.random.rand(1000000)
>>> b = np.random.rand(1000000)
>>> c = np.zeros(1000000)
>>> etime("c = a**2 + b**2 + 2*a*b", "a,b,c")
Elapsed: 0.0080178976059 seconds
```

Note: The python function `etime` *measures the elapsed time. It is defined in the file* `etime.py` *that is part of the code of this session. The second argument should list the variables used (though some will be picked up automatically).*

Ipython has its own version of this, invoked (without quotes) as

```
In [10]: %time c = a**2 + b**2 +2*a*b
```

Using the numexpr package

With numexpr:

```
>>> from etime import etime
>>> import numpy as np
>>> import numexpr as ne
>>> a = np.random.rand(1000000)
>>> b = np.random.rand(1000000)
>>> c = np.zeros(1000000)
>>> etime("c = a**2 + b**2 + 2*a*b")
Elapsed: 0.00992105007172 seconds
>>> old = ne.set_num_threads(1)
>>> etime("ne.evaluate('a**2 + b**2 + 2*a*b',out=c)", "a,b,c")
Elapsed: 0.00384114980698 seconds
>>> old = ne.set_num_threads(2)
>>> etime("ne.evaluate('a**2 + b**2 + 2*a*b',out=c)", "a,b,c")
Elapsed: 0.00219610929489 seconds
```

SciNet compute canada | calcul canada

Numexpr for the diffusion example

- Annoyingly, numexpr has no facilities for slicing or offsets, etc.
- This is troubling for our diffusion code, in which we have to do something like

```
laplacian[1:nrows+1,1:ncols+1] = (dens[2:nrows+2,1:ncols+1]
                                  + dens[0:nrows+0,1:ncols+1]
                                  + dens[1:nrows+1,2:ncols+2]
                                  + dens[1:nrows+1,0:ncols+0]
                                  - 4*dens[1:nrows+1,1:ncols+1])
```

- We would need to make a copy of `dens[2:nrows+2,1:ncols+1]` etc. into a new numpy array before we can use `numexpr`, but copies are expensive.
- We want `numexpr` to use the same data as in dens, but *viewed* differently.

Numexpr for the diffusion example (cont.)

- We want `numexpr` to use the same data as in dens, but *viewed* differently.
- That is tricky, and requires knowledge of the data's memory structure.
- `diff2d_numexpr` shows one possible solution.

```
$ etime python diff2d_numpy.py > diff2d_numpy.out
Elapsed: 21.03 seconds
$ etime python diff2d_numexpr.py  > diff2d_numexpr.out
Elapsed: 4.16 seconds
```

SciNet compute canada | calcul canada