An Overview of Python Programming

Lesson 7:
Using Compiled Code within Python

Nick Featherstone
feathern@colorado.edu

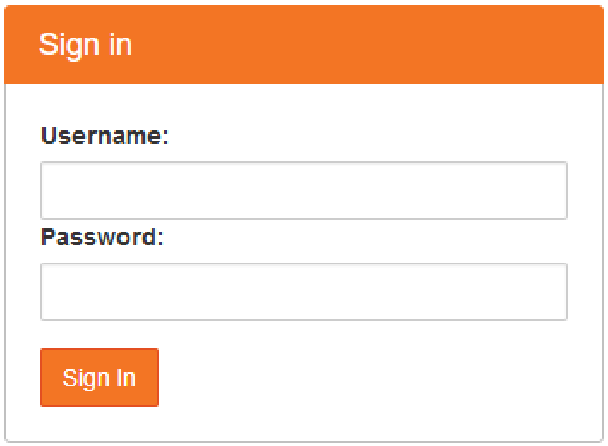
Outline

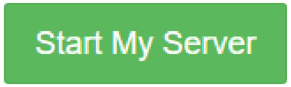
- Python & C with Cython
- Python and Fortran with F2PY

Today We Use RC Jupyterhub

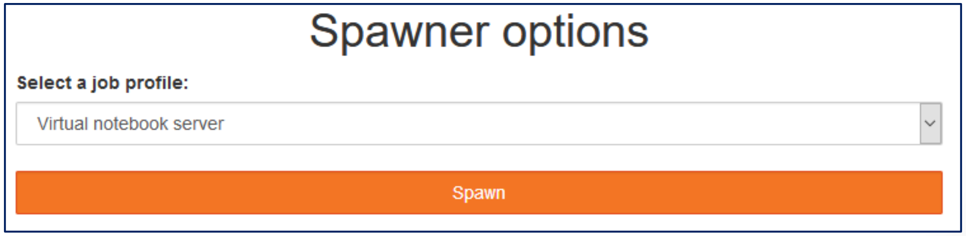
Visit https://jupyter.rc.colorado.edu

1) Log on with your tutorial credentials

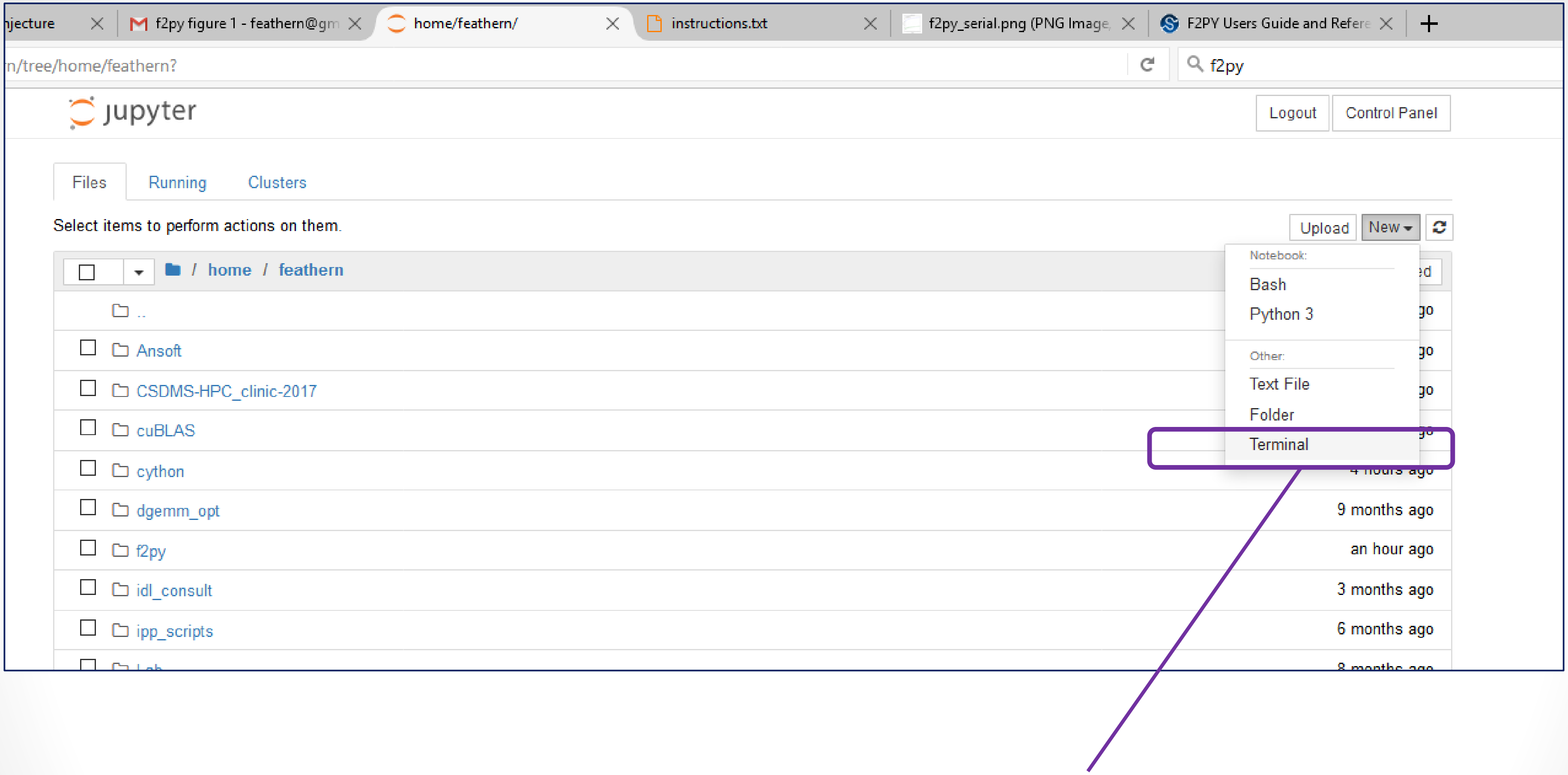


2) Click 

3) Select “virtual notebook server”



4) Click “Spawn”



Create a "New" "Terminal" session.

Once you have a terminal

- Clone the repository (all one line):

git clone
https://github.com/ResearchComputing/Python_Overview_Fall2017.git

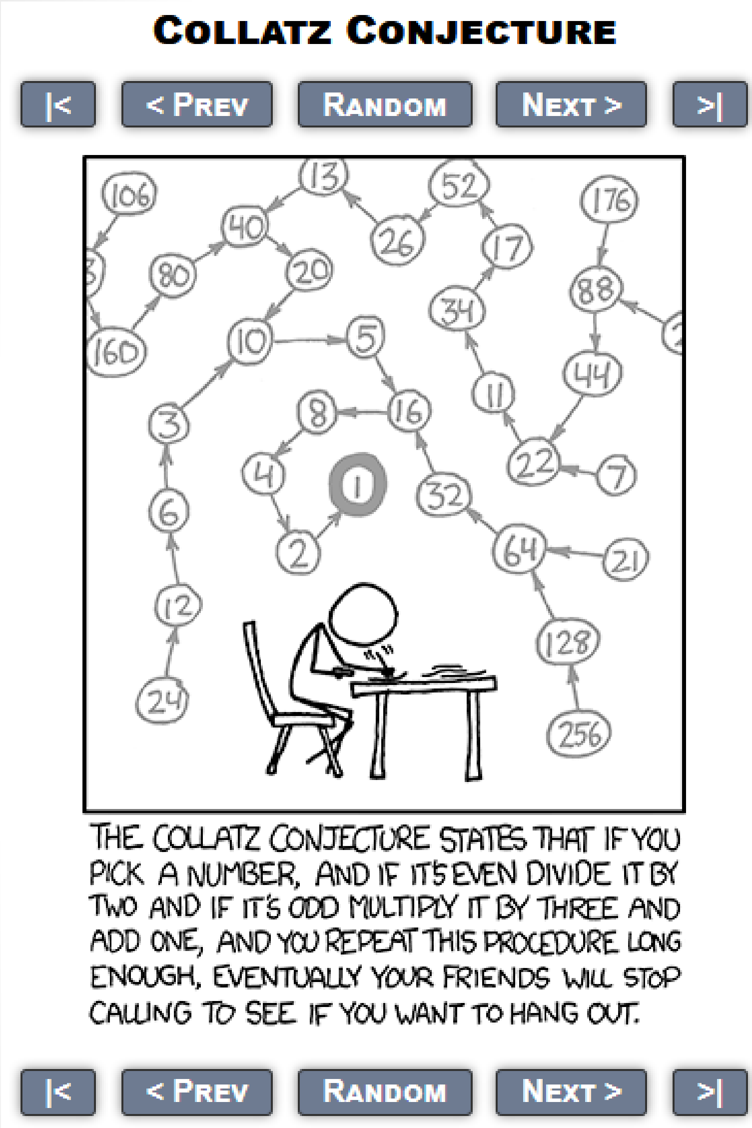
- Start an interactive session:

sinteractive –N1 –n24 –t60 - - reservation=python-10.31
 - -account=tutorial1
(two dashes before reservation and account, not one)

- Once your prompt changes, load the python module:
  - module purge
  - module load python/3.5.1

Editing in the Terminal

- If you don't know what to do, use nano
- Type "nano" to begin
- To exit: ctrl + x
- To save to file: ctrl + o (defaults to current file)
- To cut : ctrl +k
- To paste : ctrl +u



(www.xkcd.com)

Generating a Collatz Sequence:

1. Pick a positive integer (n)
2. Generate a new number (m)
    - If n is even, m = n//2
    - If n is odd, m=3n+1
3. Repeat this process until you arrive at 1

Collatz Conjecture:

- Every sequence will eventually terminate at the number 1
- Unproven (go try…)
- Verified for numbers through 87 x 2^{60}

Example Sequences:

1	Length 1
2, 1	Length 2
3, 10, 5, 16, 8, 4, 2, 1	Length 8
21, 64, 32, 16, 8, 4, 2, 1	Length 8

Preparation

- Write a function named pycollatz that
  - Accepts one parameter: an integer n
  - Returns the Collatz-sequence length of n
- Write a program that computes the time required to compute each sequence length for the first m integers (start with m = 10,000)
- Use Matplotlib to plot the results
- Name your program timeit.py

Sample Timing Code

```
import time
t0 = time.time()
…
clen = pycollatz(n)
t1 = time.time()
dt = t1-t0
times. append(dt)
…
```

Cython (cython.org)

- Python tool used for integrating C/C++ and Python
  - Use C libraries/functions from within Python
  - Translate Python code into optimized, compiled C code that can be called from within Python (Today)

- Our use case:

We have written some code in Python. We have no idea how to code in C or Fortran, but we want to gain some of the optimization benefits provided by a compiler.

Process Overview

1. Generate our python source code, save it with a .pyx extension
2. Create a setup.py script
3. Run setup.py to generate a python module built using compiled C-code
4. Use the module in a program

Before we begin:

Create a working directory: /projects/$USER/collatz

Step 1: Generate Python Source

- Save this to a file named collatz.pyx

```
def collatz(n):
 """Return the length of the Collatz series for n"""
    slen = 1
    while(n > 1):
        slen +=1
        if (n%2 == 1):
            n = 3*n+1
        else:
            n = n//2
    return slen
```

Step 2: Create the Setup Script

- Save this to a file named setup.py

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("collatz.pyx")
)
```

- Distutils (intrinsic module): Used for creating Python packages
  https://docs.python.org/3.1/distutils/

- Cythonize -> generate c-code from Python source

Step 3: Run the Setup Script

- Save this to a file named setup.py
- Build the module

```
module load python/3.5.1   (if on Summit)
python setup.py build
```

- Next we need to install the module, and tell Python where to put it.

```
export MODDIR=/home/$USER/my_modules
python setup.py install - -install-lib=$MODDIR
```

- Finally, we tell Python where to look for our modules:

```
export PYTHONPATH=$MODDIR:$PYTHONPATH
```

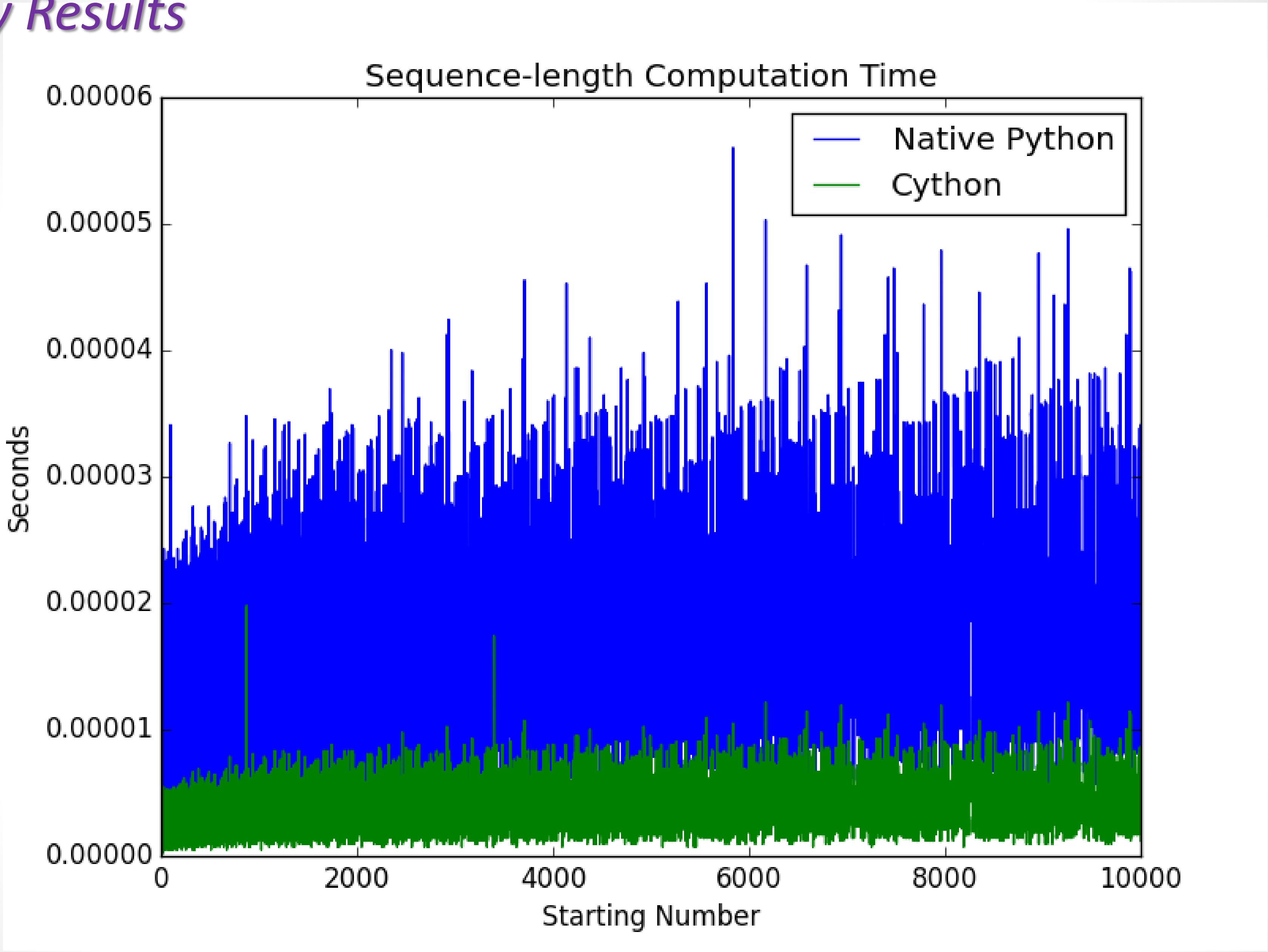
The PYTHONPATH Variable

- Paths in Unix/Linux are lists of directories, colon-separated:
- E.x.   PATH=/usr:/usr/bin:/usr/local/bin
  - Used by Linux when look for programs. First check /usr, then /usr/bin, then /usr/local/bin etc.
- PYTHONPATH
  - Colon-separated list of directories that tells Python where to look for modules
- Python checks several default locations, including subdirectories within its installation directory.
- Good practice: use a custom directory to store modules that you write + set PYTHONPATH to access them.

Step 4: Call Your Function

- Next, import your function into your Python code:
  - from collatz import collatz
  - You should be able to do this from any location
- Exercise: revise your original timing code to time both collatz and pycollatz.   Plot their results on the same graph.

My Results


Sequence-length Computation Time
Native Python
Cython
0.00006
0.00005
0.00004
0.00003
0.00002
0.00001
0.00000
Seconds
0
2000
4000
6000
8000
10000
Starting Number

F2PY

- Numpy tool used for integrating Fortran and Python
  - Can call Fortran subroutines within Python
  - Can access Fortran common blocks and module data from within Python
  - https://docs.scipy.org/doc/numpy-dev/f2py/

- Our use case:

  We have written some optimized subroutines in Fortran. We would like to use those routines in our Python code.

F2Py Process Overview

- Create your Fortran subroutine(s)
- Compile your Fortran code using F2Py
- From within Python:
  - Import the module created by F2Py
  - Call the your subroutine by passing Numpy datatypes that correspond to the Fortran datatypes
    - E.g., real*8 = float64,   integer*4 = int32, etc.

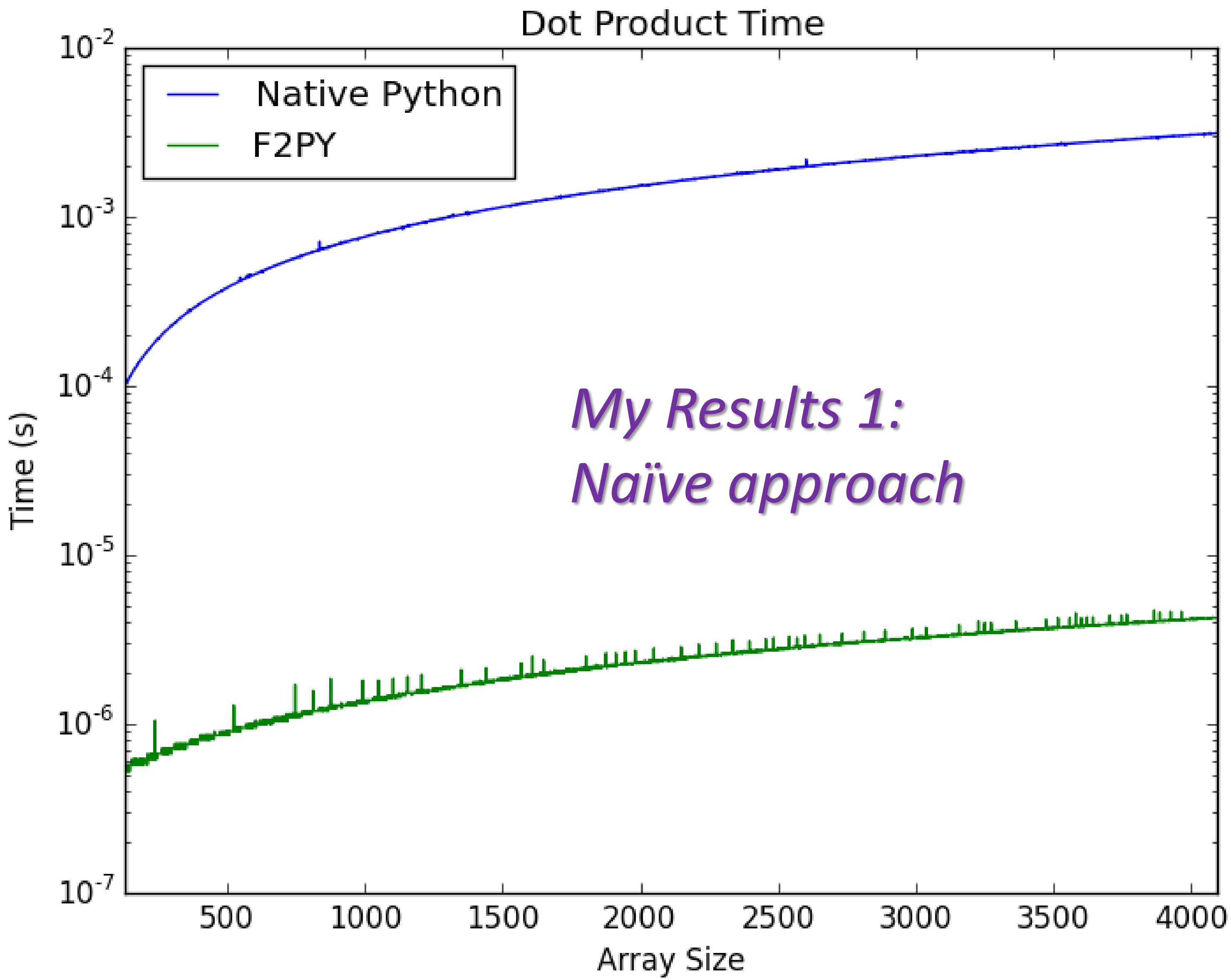
Building a Module with F2Py

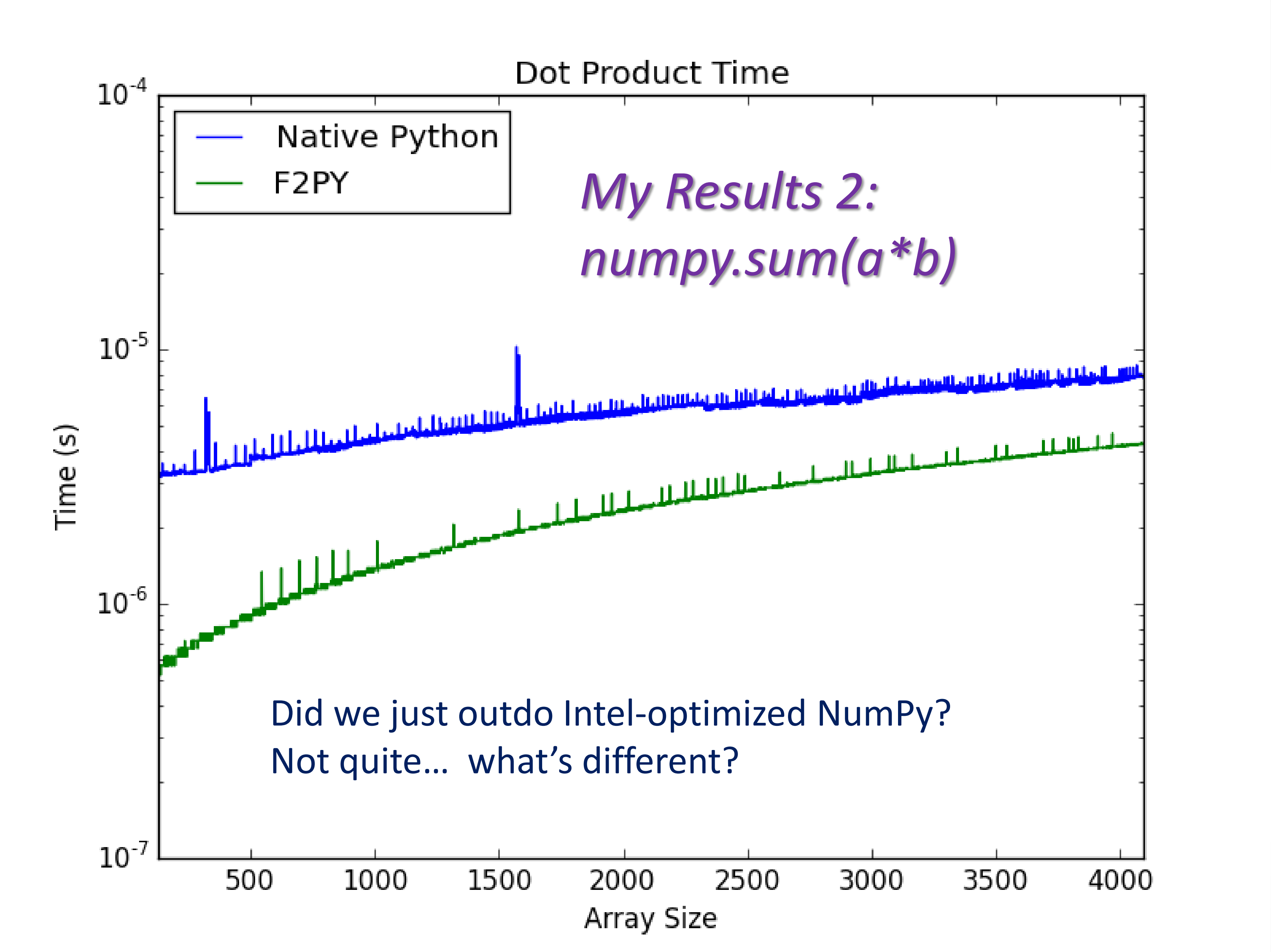
- Have a look at f2py/serial/example1.F90

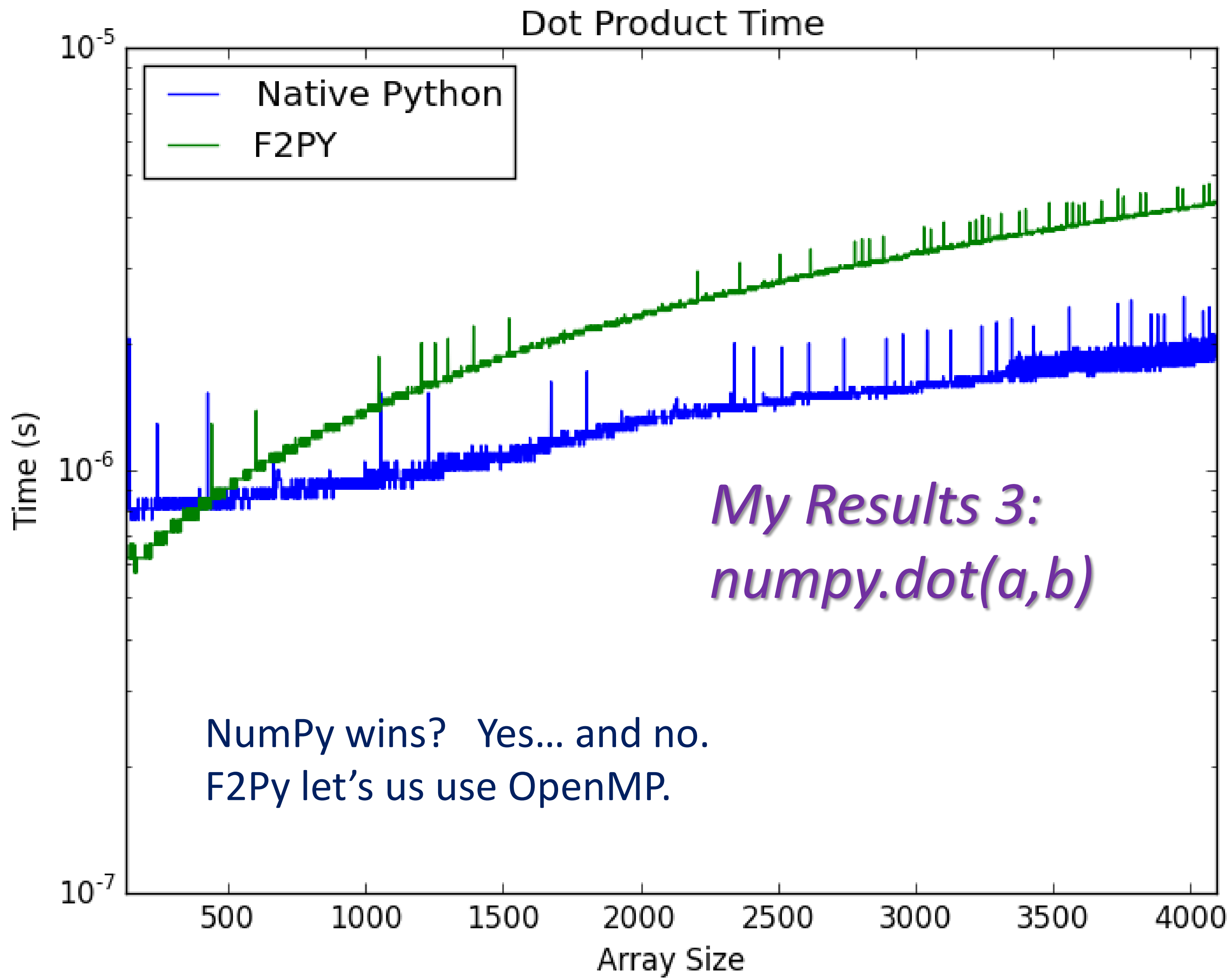
- Build the module via:
  - f2py -c example1.F90 -m ex1 (builds module named ex1)

- Examine the output (type “ls”)

- Run the code:
  - python timeit.py


Dot Product Time
Native Python
F2PY
Time (s)
Array Size
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
500
1000
1500
2000
2500
3000
3500
4000
My Results 1:
Naïve approach


Dot Product Time
Native Python
F2PY
My Results 2:
numpy.sum(a*b)
Did we just outdo Intel-optimized NumPy?
Not quite… what's different?
Time (s)
Array Size
10-4
10-5
10-6
10-7
500
1000
1500
2000
2500
3000
3500
4000



My Results 3: numpy.dot(a,b)

NumPy wins? Yes… and no.
F2Py let's us use OpenMP.

OpenMP with F2Py

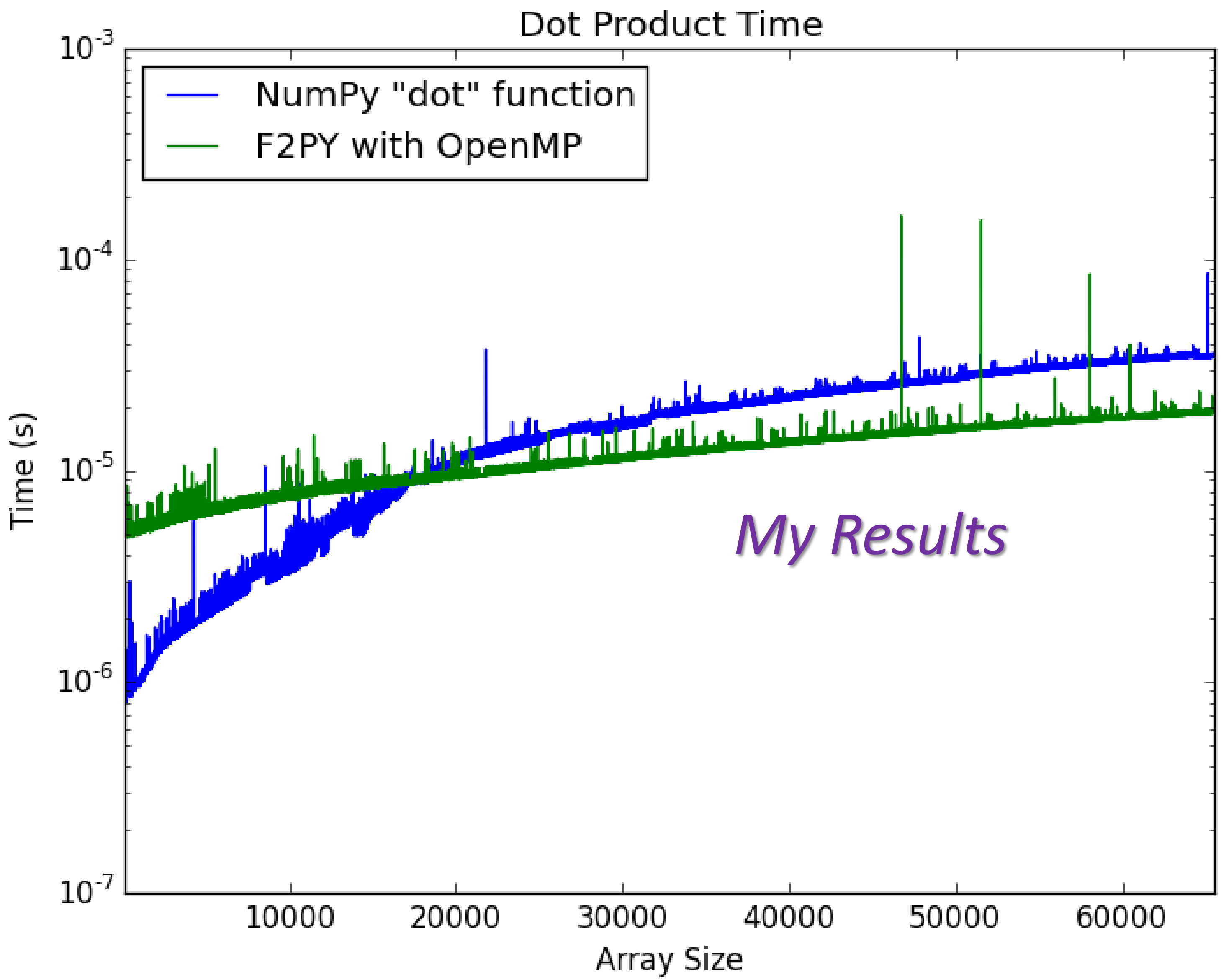
- We can make use of multiple cores by compiling our Fortran code using OpenMP directives

- Have a look at f2py/openmp/example1.F90

- Build the module via:
  - f2py -c example1.F90 -m ex1 --opt="-O3 -fopenmp" –lgomp
  - This compiles with the fopenmp flag and link to the GNU OpenMP library

- Set the OpenMP thread count:
  - export OMP_NUM_THREADS=8

- Run the code:
  - python timeit.py


Dot Product Time
NumPy "dot" function
F2PY with OpenMP
Time (s)
Array Size
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10000
20000
30000
40000
50000
60000
My Results