VIII-Benchmarks

December 7, 2014

1 VIII-Benchmarks

The first benchmark in this notebook first appeared as a post by Jake Vanderplas on the blog Pythonic Perambulations

1.0.1 Index

- Pairwise Distance
  - NumPy Broadcasting
  - Pure Python
  - Numba
  - Cython
  - Parakeet
  - Fortran/F2Py
  - SciPy Euclidean Distance
  - Scikit-learn Distance
- 1D Black-Scholes
  - Pure Python
  - Numba
  - Cython
  - Matlab

1.1 Pairwise distance

```
In [1]: import numpy as np
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        X = np.random.random((1000, 3))
```

```
In [2]: def parse_cap(s):
            return cap.stdout.split(' ')[-4:-2]
        Benchs = ['python\nloop', 'numpy\nbroadc.', 'sklearn', 'fortran/\nf2py', 'scipy', 'cython', 'nu
        Benchmarks = dict((bench,0) for bench in Benchs)
```

We'll start by defining the array which we'll use for the benchmarks: one thousand points in three dimensions.

```
In [3]: np.show_config()
```

```
lapack_opt_info:
    libraries = ['mkl_lapack95_lp64', 'mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread
    library_dirs = ['/home/jpsilva/anaconda/lib']
```

```
    define_macros = [('SCIPY_MKL_H', None)]
    include_dirs = ['/home/jpsilva/anaconda/include']
blas_opt_info:
    libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread']
    library_dirs = ['/home/jpsilva/anaconda/lib']
    define_macros = [('SCIPY_MKL_H', None)]
    include_dirs = ['/home/jpsilva/anaconda/include']
openblas_lapack_info:
  NOT AVAILABLE
lapack_mkl_info:
    libraries = ['mkl_lapack95_lp64', 'mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread
    library_dirs = ['/home/jpsilva/anaconda/lib']
    define_macros = [('SCIPY_MKL_H', None)]
    include_dirs = ['/home/jpsilva/anaconda/include']
blas_mkl_info:
    libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread']
    library_dirs = ['/home/jpsilva/anaconda/lib']
    define_macros = [('SCIPY_MKL_H', None)]
    include_dirs = ['/home/jpsilva/anaconda/include']
mkl_info:
    libraries = ['mkl_intel_lp64', 'mkl_intel_thread', 'mkl_core', 'iomp5', 'pthread']
    library_dirs = ['/home/jpsilva/anaconda/lib']
    define_macros = [('SCIPY_MKL_H', None)]
    include_dirs = ['/home/jpsilva/anaconda/include']
```

1.2 Numpy Function With Broadcasting

We'll start with a typical numpy broadcasting approach to this problem. Numpy broadcasting is an abstraction that allows loops over array indices to be executed in compiled C. For many applications, this is extremely fast and efficient. Unfortunately, there is a problem with broadcasting approaches that comes up here: it ends up allocating hidden temporary arrays which can eat up memory and cause computational overhead. Nevertheless, it's a good comparison to have. The function looks like this:

```
In [4]: %%capture cap

        def pairwise_numpy(X):
            return np.sqrt(((X[:, None, :] - X) ** 2).sum(-1))
        %timeit pairwise_numpy(X)

In [5]: Benchmarks['numpy\nbroadc.'] = parse_cap(cap)
        print cap.stdout

10 loops, best of 3: 40 ms per loop
```

1.3 Pure Python Function

A loop-based solution avoids the overhead associated with temporary arrays, and can be written like this:

```
In [6]: def pairwise_python(X):
            M = X.shape[0]
            N = X.shape[1]
            D = np.empty((M, M), dtype=np.float)
            for i in range(M):
                for j in range(M):
                    d = 0.0
```

```
                for k in range(N):
                    tmp = X[i, k] - X[j, k]
                    d += tmp * tmp
                D[i, j] = np.sqrt(d)
        return D
```

```
In [7]: %%capture cap
        %timeit pairwise_python(X)
```

```
In [8]: Benchmarks['python\nloop'] = parse_cap(cap)
        print cap.stdout
```

```
1 loops, best of 3: 2.7 s per loop
```

As we see, it is over 100 times slower than the numpy broadcasting approach! This is due to Python's dynamic type checking, which can drastically slow down nested loops. With these two solutions, we're left with a tradeoff between efficiency of computation and efficiency of memory usage. This is where tools like Numba and Cython become vital

1.4 Numba Wrapper

Numba is an LLVM compiler for python code, which allows code written in Python to be converted to highly efficient compiled code in real-time.
Numba is extremely simple to use. We just wrap our python function with `autojit` (JIT stands for "just in time" compilation) to automatically create an efficient, compiled version of the function:

Note Somehow, importing pylab breaks numba

```
In [12]: %%capture cap

         from numba import double, jit,autojit

         @jit
         def pairwise_numba(X):
             M = X.shape[0]
             N = X.shape[1]
             D = np.empty((M, M))#, dtype=np.float)
             for i in range(M):
                 for j in range(M):
                     d = 0.0
                     for k in range(N):
                         tmp = X[i, k] - X[j, k]
                         d += tmp * tmp
                     D[i, j] = np.sqrt(d)
             return D

         %timeit pairwise_numba(X)
```

```
In [13]: Benchmarks['numba'] = parse_cap(cap)
         print cap.stdout
```

```
1 loops, best of 3: 3.06 s per loop
```

1.5 Optimized Cython Function

Cython is another package which is built to convert Python-like statements into compiled code. The language is actually a superset of Python which acts as a sort of hybrid between Python and C. By adding type annotations to Python code and running it through the Cython interpreter, we obtain fast compiled code. Here is a highly-optimized Cython version of the pairwise distance function, which we compile using IPython's Cython magic:

```
In [14]: %load_ext cythonmagic
```

```
In [15]: %%cython
         import numpy as np
         cimport cython
         from libc.math cimport sqrt

         @cython.boundscheck(False)
         @cython.wraparound(False)
         def pairwise_cython(double[:, ::1] X):
             cdef int M = X.shape[0]
             cdef int N = X.shape[1]
             cdef double tmp, d
             cdef double[:, ::1] D = np.empty((M, M), dtype=np.float64)
             for i in range(M):
                 for j in range(M):
                     d = 0.0
                     for k in range(N):
                         tmp = X[i, k] - X[j, k]
                         d += tmp * tmp
                     D[i, j] = sqrt(d)
             return np.asarray(D)
```

```
In [16]: %%capture cap
         %timeit pairwise_cython(X)
```

```
In [17]: Benchmarks['cython'] = parse_cap(cap)
         print cap.stdout
```

```
100 loops, best of 3: 7.12 ms per loop
```

The Cython version, despite all the optimization, is more or less the same as the result of the simple Numba decorator!
By comparison, the Numba version is a simple, unadorned wrapper around plainly-written Python code.

2 Parakeet

Parakeet is a runtime accelerator for an array-oriented subset of Python. If you're doing a lot of number crunching in Python, Parakeet may be able to significantly speed up your code.

To accelerate a function, wrap it with Parakeet's @jit decorator:

```
In [18]: %%capture cap

         from parakeet import jit

         pairwise_parakeet = jit(pairwise_python)

         %timeit pairwise_parakeet(X)
```

```
In [19]: Benchmarks['parakeet'] = parse_cap(cap)
         print cap.stdout
```

```
100 loops, best of 3: 6.17 ms per loop
```

2.1 Fortran/F2Py

Another option for fast computation is to write a Fortran function directly, and use the `f2py` package to interface with the function. We can write the function as follows:

```
In [20]: %%file pairwise_fort.f

               subroutine pairwise_fort(X,D,m,n)
                   integer :: n,m
                   double precision, intent(in) :: X(m,n)
                   double precision, intent(out) :: D(m,m)
                   integer :: i,j,k
                   double precision :: r
                   do i = 1,m
                       do j = 1,m
                           r = 0
                           do k = 1,n
                               r = r + (X(i,k) - X(j,k)) * (X(i,k) - X(j,k))
                           end do
                           D(i,j) = sqrt(r)
                       end do
                   end do
               end subroutine pairwise_fort
```

```
Overwriting pairwise_fort.f
```

We can then use the shell interface to compile the Fortran function. In order to hide the output of this operation, we direct it into `/dev/null` (note: I tested this on Linux, and it may have to be modified for Mac or Windows).

```
In [21]: !f2py -c --help-fcompiler
```

```
Gnu95FCompiler instance properties:
  archiver        = ['/usr/bin/gfortran', '-cr']
  compile_switch  = '-c'
  compiler_f77    = ['/usr/bin/gfortran', '-Wall', '-g', '-ffixed-form', '-
                    fno-second-underscore', '-fPIC', '-O3', '-funroll-loops']
  compiler_f90    = ['/usr/bin/gfortran', '-Wall', '-g', '-fno-second-
                    underscore', '-fPIC', '-O3', '-funroll-loops']
  compiler_fix    = ['/usr/bin/gfortran', '-Wall', '-g', '-ffixed-form', '-
                    fno-second-underscore', '-Wall', '-g', '-fno-second-
                    underscore', '-fPIC', '-O3', '-funroll-loops']
  libraries       = ['gfortran']
  library_dirs    = []
  linker_exe      = ['/usr/bin/gfortran', '-Wall', '-Wall']
  linker_so       = ['/usr/bin/gfortran', '-Wall', '-g', '-Wall', '-g', '-
                    shared']
  object_switch   = '-o '
  ranlib          = ['/usr/bin/gfortran']
  version         = LooseVersion ('4.9.1-16')
  version_cmd     = ['/usr/bin/gfortran', '--version']
Fortran compilers found:
  --fcompiler=gnu95  GNU Fortran 95 compiler (4.9.1-16)
Compilers available for this platform, but not found:
  --fcompiler=absoft   Absoft Corp Fortran Compiler
  --fcompiler=compaq   Compaq Fortran Compiler
```

```
  --fcompiler=g95      G95 Fortran Compiler
  --fcompiler=gnu      GNU Fortran 77 compiler
  --fcompiler=intel    Intel Fortran Compiler for 32-bit apps
  --fcompiler=intele   Intel Fortran Compiler for Itanium apps
  --fcompiler=intelem  Intel Fortran Compiler for 64-bit apps
  --fcompiler=lahey    Lahey/Fujitsu Fortran 95 Compiler
  --fcompiler=nag      NAGWare Fortran 95 Compiler
  --fcompiler=pathf95  PathScale Fortran Compiler
  --fcompiler=pg       Portland Group Fortran Compiler
  --fcompiler=vast     Pacific-Sierra Research Fortran 90 Compiler
Compilers not available on this platform:
  --fcompiler=hpux      HP Fortran 90 Compiler
  --fcompiler=ibm       IBM XL Fortran Compiler
  --fcompiler=intelev   Intel Visual Fortran Compiler for Itanium apps
  --fcompiler=intelv    Intel Visual Fortran Compiler for 32-bit apps
  --fcompiler=intelvem  Intel Visual Fortran Compiler for 64-bit apps
  --fcompiler=mips      MIPSpro Fortran Compiler
  --fcompiler=none      Fake Fortran compiler
  --fcompiler=sun       Sun or Forte Fortran 95 Compiler
For compiler details, run 'config_fc --verbose' setup command.
Removing build directory /tmp/tmpiygPNS
```

```
In [22]: !f2py -c --help-compiler
```

```
List of available compilers:
  --compiler=bcpp     Borland C++ Compiler
  --compiler=cygwin   Cygwin port of GNU C Compiler for Win32
  --compiler=emx      EMX port of GNU C Compiler for OS/2
  --compiler=intel    Intel C Compiler for 32-bit applications
  --compiler=intele   Intel C Itanium Compiler for Itanium-based applications
  --compiler=intelem  Intel C Compiler for 64-bit applications
  --compiler=mingw32  Mingw32 port of GNU C Compiler for Win32
  --compiler=msvc     Microsoft Visual C++
  --compiler=pathcc   PathScale Compiler for SiCortex-based applications
  --compiler=unix     standard UNIX-style compiler
Removing build directory /tmp/tmp9lM8Xb
```

```
In [23]: # Compile the Fortran with f2py.
         # We'll direct the output into /dev/null so it doesn't fill the screen
         #!f2py -c --compiler=intel pairwise_fort.f -m pairwise_fort > /dev/null
         !f2py -c --fcompiler=gnu95 pairwise_fort.f -m pairwise_fort > /dev/null
```

We can import the resulting code into Python to time the execution of the function. To make sure we're being fair, we'll first convert the test array to Fortran-ordering so that no conversion needs to happen in the background:

```
In [24]: %%capture cap

         from pairwise_fort import pairwise_fort
         XF = np.asarray(X, order='F')
         %timeit pairwise_fort(XF)
```

```
In [25]: Benchmarks['fortran/\nf2py'] = parse_cap(cap)
         print cap.stdout
```

```
100 loops, best of 3: 6.19 ms per loop
```

The result is nearly a factor of two slower than the Cython and Numba versions.

2.2 Scipy Pairwise Distances

```
In [26]: %%capture cap

         from scipy.spatial.distance import cdist
         %timeit cdist(X, X)

In [27]: Benchmarks['scipy'] = parse_cap(cap)
         print cap.stdout

100 loops, best of 3: 5.79 ms per loop
```

2.3 Scikit-learn Pairwise Distances

Scikit-learn contains the `euclidean_distances` function, works on sparse matrices as well as numpy arrays, and is implemented in Cython:

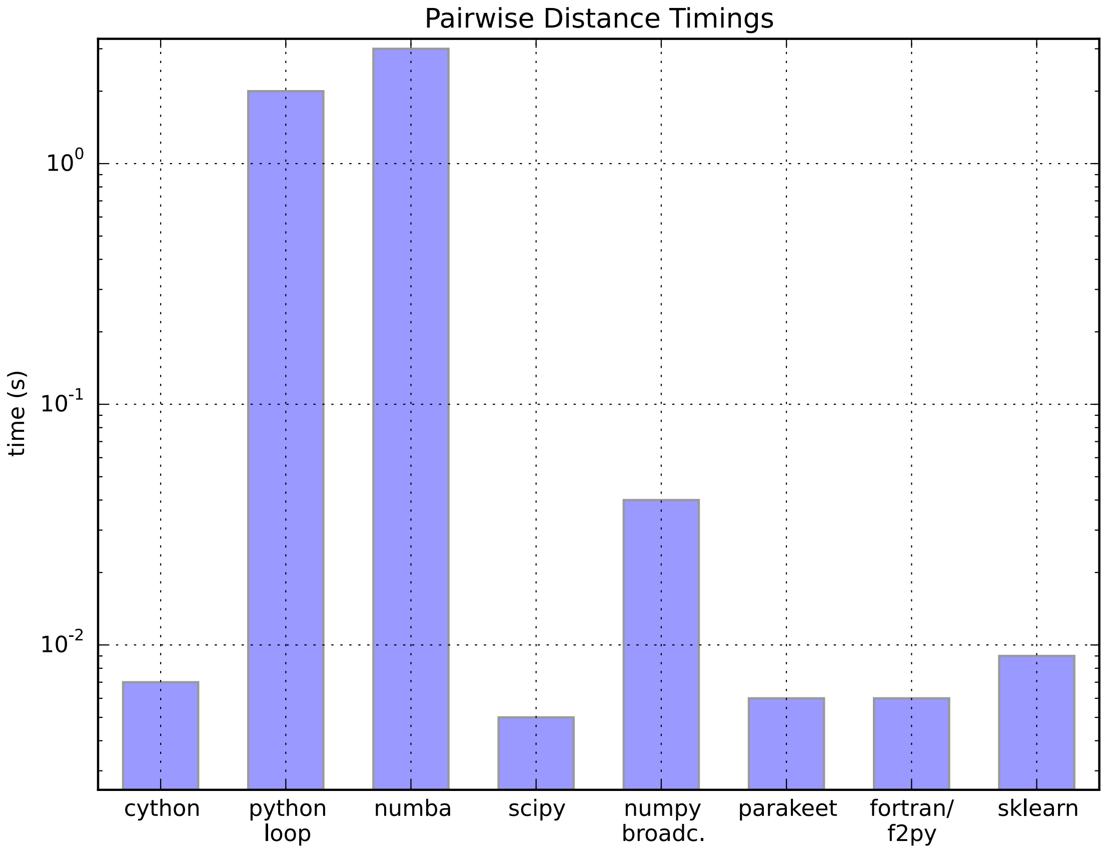
```
In [28]: %%capture cap

         from sklearn.metrics import euclidean_distances
         %timeit euclidean_distances(X, X)

In [29]: Benchmarks['sklearn'] = parse_cap(cap)
         print cap.stdout

100 loops, best of 3: 9.7 ms per loop
```

`euclidean_distances` is several times slower than the Numba pairwise function on dense arrays.

2.4 Comparing the Results

As a summary of the results, we'll create a bar-chart to visualize the timings:

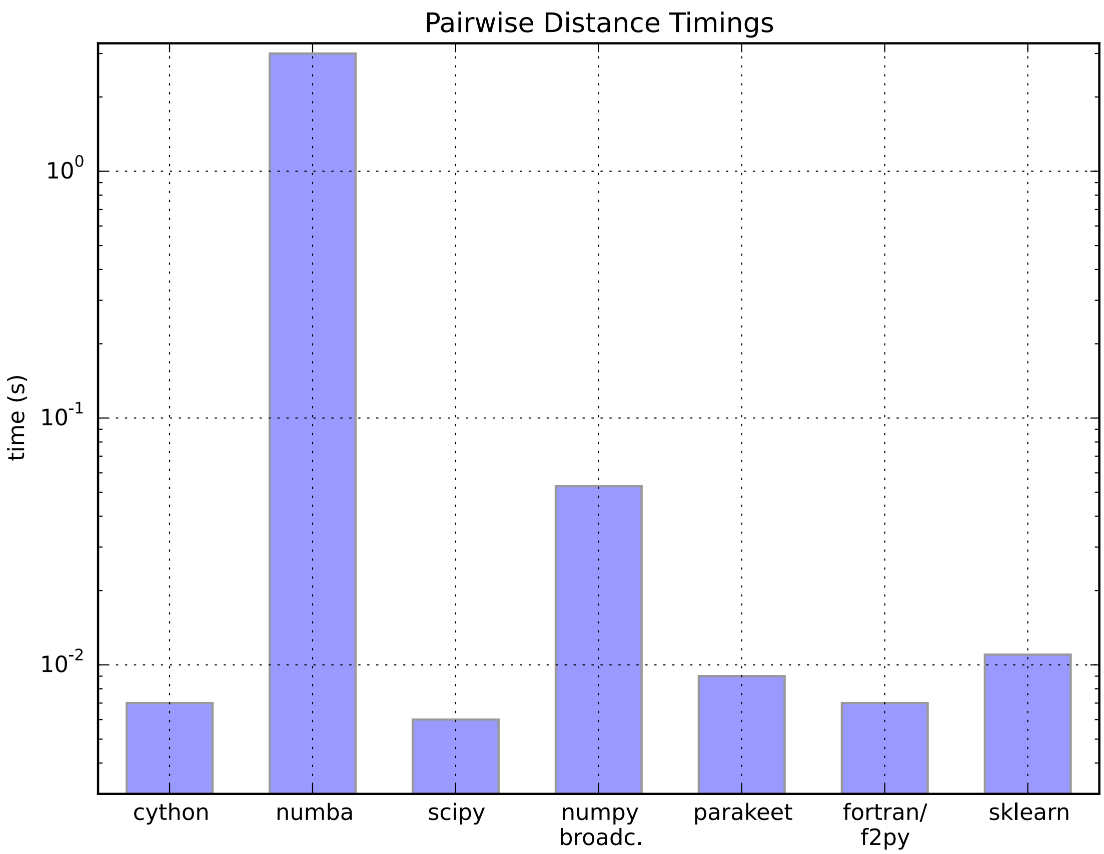
```
In [30]: %matplotlib inline

In [31]: labels = Benchmarks.keys()
         timings = [np.timedelta64(int(float(value[0])),str(value[1]))/np.timedelta64(1, 's') for value
         x = np.arange(len(labels))
         ax = plt.axes(xticks=x, yscale='log')
         ax.bar(x - 0.3, timings, width=0.6, alpha=0.4, bottom=1E-6)
         ax.grid()
         ax.set_xlim(-0.5, len(labels) - 0.5)
         ax.set_ylim(0.5*min(timings), 1.1*max(timings))
         ax.xaxis.set_major_formatter(plt.FuncFormatter(lambda i, loc: labels[int(i)]))
         ax.set_ylabel('time (s)')
         ax.set_title("Pairwise Distance Timings")

Out[31]: <matplotlib.text.Text at 0x7f7c57063f50>
```



Note that this is log-scaled, so the vertical space between two grid lines indicates a factor of 10 difference in computation time. As Pure Python takes more time by two orders of magnitude we'll now remove it from the benchmarks for a better comparison.

In [33]:
```
labels = [val for val in Benchmarks.iterkeys() if val != 'python\nloop']
timings = [np.timedelta64(int(float(value[0])),str(value[1]))/np.timedelta64(1, 's')\
           for k, value in Benchmarks.items() if k != 'python\nloop']
x = np.arange(len(labels))
ax = plt.axes(xticks=x, yscale='log')
ax.bar(x - 0.3, timings, width=0.6, alpha=0.4, bottom=1E-6)
ax.grid()
ax.set_xlim(-0.5, len(labels) - 0.5)
ax.set_ylim(0.5*min(timings), 1.1*max(timings))
ax.xaxis.set_major_formatter(plt.FuncFormatter(lambda i, loc: labels[int(i)]))
ax.set_ylabel('time (s)')
ax.set_title("Pairwise Distance Timings")
```

Out[33]:
```
<matplotlib.text.Text at 0x7fb6ba1d5b10>
```



2.5 1D European Option using Black-Scholes Formula

In [37]:
```
#Restart notebook!
```

We'll now benchmark a simple function which returns the price of a 1D European Option using the closed form solution with the following parameters:

- $S_0 = 100$
- $K = 100$
- $T = 1$ (years)
- $\sigma = 0.3$
- $r = 0.03$
- dividend = 0
- option type = -1 (+1 for call, -1 for put)

In [34]:
```
S0 = 100
K = 100
T = 1
vol = 0.3
r = 0.03
q = 0
optype = -1
```

In [35]:
```
from numpy import log, sqrt, zeros, exp, arange
from scipy.special import erf

def parse_cap(s):
    return cap.stdout.split(' ')[-4:-2]

benchs = ['python','numba','cython','matlab']
BS_bench = dict(zip(benchs,zeros(4)))
```

3 Pure Python

In [36]:
```
def std_norm_cdf(x):
    return 0.5*(1+erf(x/sqrt(2.0)))

def black_scholes(s, k, t, v, rf, div, cp):
    """Price an option using the Black-Scholes model.

    s : initial stock price
    k : strike price
    t : expiration time
    v : volatility
    rf : risk-free rate
    div : dividend
    cp : +1/-1 for call/put
    """
    d1 = (log(s/k)+(rf-div+0.5*pow(v,2))*t)/(v*sqrt(t))
    d2 = d1 - v*sqrt(t)
    optprice = cp*s*exp(-div*t)*std_norm_cdf(cp*d1) - \
            cp*k*exp(-rf*t)*std_norm_cdf(cp*d2)
    return optprice
```

In [37]:
```
%%capture cap
%timeit black_scholes(S0, K, T, vol, r, q, optype)
```

In [38]:
```
BS_bench['python'] = parse_cap(cap)
print cap.stdout
```

```
100000 loops, best of 3: 11.6 us per loop
```

4 Numba

In [39]:
```
from numba import double
from numba.decorators import jit, autojit
import math

@autojit
def std_norm_cdf_numba(x):
    return 0.5*(1+math.erf(x/math.sqrt(2.0)))

@autojit
def black_scholes_numba(s, k, t, v, rf, div, cp):
    """Price an option using the Black-Scholes model.

    s : initial stock price
```

```
    k : strike price
    t : expiration time
    v : volatility
    rf : risk-free rate
    div : dividend
    cp : +1/-1 for call/put
    """
    d1 = (math.log(s/k)+(rf-div+0.5*v*v)*t)/(v*math.sqrt(t))
    d2 = d1 - v*math.sqrt(t)
    optprice = cp*s*math.exp(-div*t)*std_norm_cdf_numba(cp*d1) - \
            cp*k*math.exp(-rf*t)*std_norm_cdf_numba(cp*d2)
    return optprice
```

```
In [40]: %%capture cap
         %timeit black_scholes_numba(S0, K, T, vol, r, q, optype)
```

```
In [41]: BS_bench['numba'] = parse_cap(cap)
         print cap.stdout
```

```
1 loops, best of 3: 5.01 us per loop
```

5 Cython

```
In [42]: %load_ext cythonmagic
```

```
The cythonmagic extension is already loaded. To reload it, use:
  %reload_ext cythonmagic
```

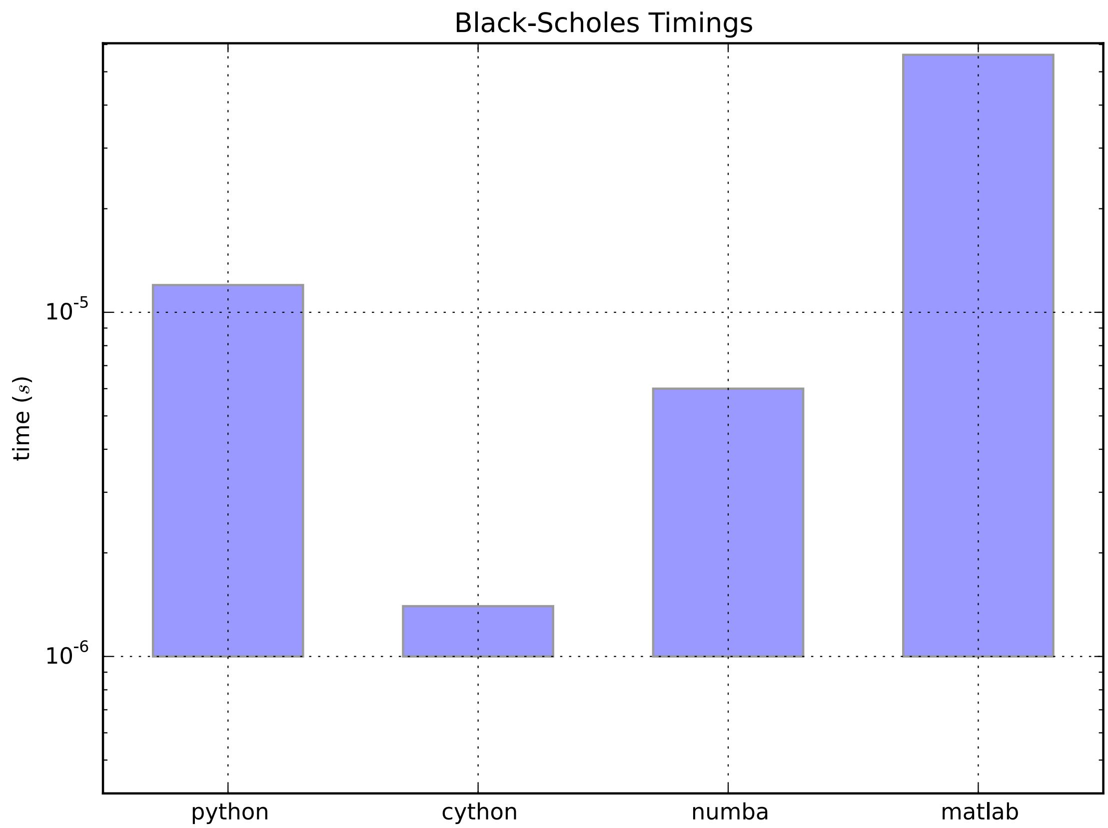
```
In [43]: %%cython
         cimport cython
         from libc.math cimport exp, sqrt, pow, log, erf

         @cython.cdivision(True)
         cdef double std_norm_cdf(double x) nogil:
             return 0.5*(1+erf(x/sqrt(2.0)))

         @cython.cdivision(True)
         def black_scholes(double s, double k, double t, double v,
                          double rf, double div, double cp):
             """Price an option using the Black-Scholes model.

             s : initial stock price
             k : strike price
             t : expiration time
             v : volatility
             rf : risk-free rate
             div : dividend
             cp : +1/-1 for call/put
             """
             cdef double d1, d2, optprice
             with nogil:
                 d1 = (log(s/k)+(rf-div+0.5*pow(v,2))*t)/(v*sqrt(t))
                 d2 = d1 - v*sqrt(t)
                 optprice = cp*s*exp(-div*t)*std_norm_cdf(cp*d1) - \
                     cp*k*exp(-rf*t)*std_norm_cdf(cp*d2)
             return optprice
```

```
In [44]: %%capture cap
         %timeit black_scholes(S0, K, T, vol, r, q, optype)

In [45]: BS_bench['cython'] = parse_cap(cap)
         print cap.stdout

1000000 loops, best of 3: 400 ns per loop
```

6 MATLAB

We now analyse the corresponding **MATLAB** code which we'll use for comparison

```
In [47]: !cat ../std_norm_cdf.m

function z = std_norm_cdf(x)
z = 0.5*(1+erf(x/sqrt(2.0)));

In [48]: !cat ../black_scholes.m

function z = black_scholes(s, k, t, v, rf, div, cp)

d1 = (log(s/k)+(rf-div+0.5*v*v)*t)/(v*sqrt(t));
d2 = d1 - v*sqrt(t);
optprice = cp*s*exp(-div*t)*std_norm_cdf(cp*d1)-cp*k*exp(-rf*t)*std_norm_cdf(cp*d2);
z = optprice;

In [49]: %matplotlib inline

In [50]: BS_bench['matlab'] = [u'55',u'us']
         labels= BS_bench.keys()
         timings = [np.timedelta64(int(float(value[0])),str(value[1]))/np.timedelta64(1, 's') for value
         x = arange(len(labels))
         ax = plt.axes(xticks=x, yscale='log')
         ax.bar(x - 0.3, timings, width=0.6, alpha=0.4, bottom=1E-6)
         ax.grid()
         ax.set_xlim(-0.5, len(labels) - 0.5)
         ax.set_ylim(min(timings), 1.1*max(timings))
         ax.xaxis.set_major_formatter(plt.FuncFormatter(lambda i, loc: labels[int(i)]))
         ax.set_ylabel('time ($s$)')
         ax.set_title("Black-Scholes Timings")

Out[50]: <matplotlib.text.Text at 0x7f7c55db4a50>
```



This post was written entirely as an IPython notebook. The full notebook can be downloaded here, or viewed statically on nbviewer

Back to top

In [51]:
```
%load_ext version_information
%version_information numpy,numba,parakeet,cython,matplotlib,scipy,sklearn
```

Out[51]:

Software	**Version**
Python	2.7.8 —Anaconda 2.1.0 (64-bit)— (default, Aug 21 2014, 18:22:21) [GCC 4.4.7 20120313 (Red Hat 4.4.7-1
IPython	2.3.1
OS	posix [linux2]
numpy	1.9.1
numba	0.15.1
parakeet	0.23.2
cython	0.21.1
matplotlib	1.4.2
scipy	0.14.0
sklearn	0.15.2
Fri Dec 05 11:18:11 2014 CET	

In [1]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

```
Out[1]: <IPython.core.display.HTML at 0x7fb6edb1b8d0>

In []:
```