www.joim.com

Journal Of Investment Management, Vol. 8, No. 4, (2010), pp. 1–12

© JOIM 2010

IMPLEMENTING OPTION PRICING MODELS USING PYTHON AND CYTHON

Sanjiv Das[a] *and Brian Granger*[b]

In this article we propose a new approach for implementing option pricing models in finance. Financial engineers typically prototype such models in an interactive language (such as Matlab) and then use a compiled language such as C/C++ for production systems. Code is therefore written twice. In this article we show that the Python programming language and the Cython compiler allows prototyping in a Matlab-like manner, followed by direct generation of optimized C code with very minor code modifications. The approach is able to call upon powerful scientific libraries, uses only open source tools, and is free of any licensing costs. We provide examples where Cython speeds up a prototype version by over 500 times. These performance gains in conjunction with vast savings in programmer time make the approach very promising.

1 Introduction

Computing in financial engineering needs to be fast in two critical ways. First, the software development process must be fast for human developers; programs must be easy and time efficient to write and maintain. Second, the programs themselves must be fast when they are run. Traditionally, to meet both of these requirements, at least two versions of a program have to be developed and maintained; a prototype in a high-level interactive language like Matlab, R, Excel, etc., and a high-performance production version in a compiled language like C, C++, or Fortran. This duplication of effort slows the deployment of new algorithms and creates ongoing software maintenance problems, not to mention added development costs.

In this paper we describe an approach to technical software development that enables a single version of a program to be written that is both easy to develop and maintain and achieves high levels of performance. This approach uses the Python programming language (Python, 2010) and the Cython compiler (Cython, 2010) to generate optimized C code whose performance is comparable to that of handwritten C code. The modifications needed to the original source code of the

[a]Santa Clara University, Santa Clara, CA 95053, USA.
[b]California Polytechnic State University, San Luis Obispo, CA 93407, USA.

program are minimal, basically amounting to a small number of static type declarations. We illustrate the approach in a finance context by showing how two option pricing models, the binomial tree and Black–Scholes models, can be implemented in Python and then optimized using the Cython compiler and language extensions.

1.1 *Python*

insert hyphen here

The Python programming language[1] has, in the last decade, become one of the premier languages for scientific and technical computing.[2] Python is an interpreted, dynamically typed programming language that supports a wide range of programming styles including object-oriented, functional, and procedural. It is an open source language that was started in the early 1990s by Guido van Rossum (now at Google) and continues to be developed by a large team of developers worldwide. Python is currently one of the most popular programming languages (TIOBE, 2010) and is used extensively by the DOE, DOD, NASA, academic researchers, and companies like Google, Industrial Light and Magic, and Rackspace (Python Success Stories, 2010).

Why is Python such an effective tool for scientific and technical computing? Our central argument in this paper is that Python combines the ease-of-use, interactive workflow, and integrated libraries of environments such as Matlab, Mathematica, R, and Excel with the performance of compiled languages such as C, C++, and Fortran. More specifically, Python has the following attributes that are relevant in a financial engineering context:

(1) Python has a friendly syntax that is easy to read and write. In this respect, Python stands in sharp contrast to other popular languages such as Java, C++, and C#, whose syntax is heavy, complex, subtle, and places a significant cognitive load on its users.
(2) Python can be used in an interactive and exploratory manner. Like Matlab, Excel, R, etc., Python offers an interactive shell/environment, where commands are typed at an interactive prompt and executed immediately. This type of workflow has become extremely popular among technical users at it is well matched to the exploratory nature of research, prototyping and data analysis. In this paper we will illustrate this aspect of Python through the usage of IPython,[3] which is an enhanced interactive Python shell (IPython, 2007).
(3) Python is easy to integrate with other languages. Statistical packages like R may call Python, and vice versa. The same holds for compiled languages like C, C++, and Fortran. There are a large number of tools in the Python ecosystem that ease this process, such as f2py,[4] SWIG,[5] Boost.Python,[6] ctypes,[7] and Cython.[8] The result is that Python is an excellent glue language for integrating codes in a wide variety of underlying languages. Furthermore, there are .NET (IronPython[9]) and Java (Jython[10]) implementations of the Python VM, which make it possible to use Python with .NET and Java libraries.
(4) Python can be made very fast. Because Python is interpreted and dynamically typed, it is not as fast natively as statically typed languages. However, because it is so easy to call compiled languages from Python, it is straightforward to develop performance critical code in C/C++/Fortran and then call it from Python. Furthermore, as we will describe in this paper, the Cython project provides a compiler and a set of language extensions that allow optimized C code to be autogenerated directly from Python code.
(5) There is a vast set of open source Python packages that provide all the tools needed in technical computing. The NumPy package[11] contains a powerful N-dimensional array object along with functions for linear algebra, Fourier analysis, random numbers, etc. The SciPy package[12] adds additional algorithms on top

of NumPy that include optimization, integration, statistics, sparse matrices, etc. For plotting and visualization, Matplotlib[13] and Chaco[14] provide publication quality 2D plotting and Mayavi[15] provides VTK-based 3D visualization capabilities.

(6) Python has language features that allow large-scale object-oriented applications to be built easily. These features include mature object-oriented capabilities and module/package namespaces. This offers a huge advantage over other interactive environments such as Matlab, Octave, or Excel, where it is quite difficult to build large-scale applications that are well encapsulated and have reusable components with well-defined interfaces. These features are critical in financial engineering, where applications can be large and complex.

(7) Python is easy to install and deploy. Python and most of the packages described in this paper are available through the standard Linux package managers and double clickable Windows/Mac OS X installers. In addition, there are a number of commercial and open source "distributions" that package all of the software described here, along with its documentation in an easy-to-use installer. These distributions include Enthought Python Distribution[16] (or EPD), PyIMSL Studio,[17] Sage,[18] and PythonXY.[19]

insert hyphen

1.2 Cython

Cython[20] is an open source project that provides a compiler and set of language extensions to Python. The name "Cython" is rooted in the idea that Cython is "C" + "Python". The fact that Python itself is an interpreted and dynamically typed language means that the Python virtual machine is unable to perform the compile time optimizations of a statically typed language like C. While this makes Python easy-to-use and flexible, it limits the raw performance Python can achieve. However, in technical computing, you often do know the types of variables beforehand, so it should be possible, at least in principle, to perform type-specific optimizations.

As an illustration, consider what happens when you add two Python objects. The Python compiler will generate bytecode instructions that will carry out that operation for any two Python objects, regardless of their type.[21] These bytecode instructions can be viewed using Python's `dis` module as follows:

```
In [1]: def add(a,b):
   ...:     return a+b

In [2]: dis.add(f)
  2         0 LOAD_FAST           0 (a)
            3 LOAD_FAST           1 (b)
            6 BINARY_ADD
            7 RETURN_VALUE
```

Each bytecode instruction, such as `BINARY_ADD`, can involve many low-level CPU instructions, which makes Python much slower than languages like C. On the other hand, if you want to add two double precision floats, a compiler should be able to reduce this operation to a small number of native CPU instructions. This is exactly what Cython brings to the Python language. As a language Cython is a superset of Python that introduces a few carefully chosen language extensions to enable static typing. Most important is the `cdef` keyword, which is used to declare the C datatypes of variables. In the above example the static type declarations would be:

```
cdef f(double a, double b):
    return a+b
```

From this simple modification, the Cython compiler is able to eliminate the costly Python bytecode instructions and generate optimized C code that a C compiler can express efficiently using few native CPU instructions. The magic of Cython is that these types declarations are entirely optional;

a program can freely mix statically typed `cdef` variables with dynamically typed Python objects.[22] The Cython compiler automatically generates the code to convert between Python and C datatypes when needed.

The result is that by simply adding static type declarations to an existing Python program using `cdef` you enable the Cython compiler to generate high-performance C code from your Python code. The resulting code remains callable from Python, but has performance near to that of handwritten C, C++, or Fortran. A side effect of this is that you can also call arbitrary external C/C++/Fortran libraries from your Cython source code, because of all of the type conversions can be handled by Cython.

The rest of this article proceeds as follows. Section 2 implements a binomial tree option pricing model using Python and Cython, starting from a plain Python version and then incrementally adding the Cython-specific optimizations. Section 3 repeats this process for the Black–Scholes model. We briefly mention parallelization options that exist for Python in Section 4 before our concluding remarks in Section 5.

2 Pricing options on binomial trees using Python

In order to demonstrate the power of Python and Cython we use a financial pricing example that is simple yet numerically intensive (a closed-form solution does not provide much insight). In this first example, we implement the binomial tree option pricing model for both American and European options, based on the original work of Cox *et al.* (1979). Our implementation is based on the framework of Jarrow and Rudd (1983). This model is both widely known by students in finance and numerically intensive, requiring backward recursion on a tree.

2.1 Plain Python

We begin by implementing the binomial tree model in plain Python (no C/Cython code) to set a performance baseline. While the performance of this version will be relatively poor, the code is easy to understand and can be used in an interactive and exploratory manner. For multidimensional arrays, we use the NumPy package, which is the standard Python array library for technical computing.

Listing 1 shows the straightforward Python code for this version, which is saved in a file called "binomial.py". This file (known as a "module" in Python) defines a single function, `jarrow_rudd`, that implements the algorithm and takes parameters of the option (initial stock price, strike price, the volatility, etc.) as arguments.

In order to use this module in an interactive manner we use IPython, which is an enhanced interactive Python shell with features such as tab completion, a built-in help system, easy access to the system shell and file system, interactive plotting support and a special "magic command" syntax that makes interactive work more pleasant (see the `%timeit` magic command used below). Here is an example of IPython session, in which we start up IPython, import the `jarrow_rudd` function and then use it to price a call option:

insert "an"

```
$ ipython
Enthought Python Distribution -- http://code.enthought.com

Python 2.5.2 |EPD Py25 4.1.30101| (r252:60911, Dec 19 2008, 15:28:32)
Type "copyright", "credits" or "license" for more information.
```

Listing 1 binomial.py

```
import numpy as np
import math

def jarrow_rudd(s, k, t, v, rf, cp, am=False, n=100):
    """Price an option using the Jarrow-Rudd binomial model.

    s : initial stock price
    k : strike price
    t : expiration time
    v : volatility
    rf : risk-free rate
    cp : +1/-1 for call/put
    am : True/False for American/European
    n : binomial steps
    """
    # Basic calculations
    h = t/n
    u = math.exp((rf-0.5*math.pow(v,2))*h+v*math.sqrt(h))
    d = math.exp((rf-0.5*math.pow(v,2))*h-v*math.sqrt(h))
    drift = math.exp(rf*h)
    q = (drift-d)/(u-d)

    # Process the terminal stock price
    stkval = np.zeros((n+1,n+1))
    optval = np.zeros((n+1,n+1))
    stkval[0,0] = s
    for i in range(1,n+1):
        stkval[i,0] = stkval[i-1,0]*u
        for j in range(1,i+1):
            stkval[i,j] = stkval[i-1,j-1]*d

    # Backward recursion for option price
    for j in range(n+1):
        optval[n,j] = max(0,cp*(stkval[n,j]-k))
    for i in range(n-1,-1,-1):
        for j in range(i+1):
            optval[i,j] = (q*optval[i+1,j]+(1-q)*optval[i+1,j+1])/drift
            if am:
                optval[i,j] = max(optval[i,j],cp*(stkval[i,j]-k))

    return optval[0,0]
```

```
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]: from binomial import jarrow_rudd

In [2]: jarrow_rudd(100.0, 100.0, 1.0, 0.3, 0.03, 1, False, 100)
13.2969231342
```

And if we price the put option, we obtain:

```
In [3]: jarrow_rudd(100.0, 100.0, 1.0,
0.3, 0.03, -1, False, 100)
10.341476489
```

Finally, pricing the American put option should result in a slightly higher value than for the European put:

```
In [4]: jarrow_rudd(100.0, 100.0, 1.0,
0.3, 0.03, -1, True, 100)
10.6276823809
```

As mentioned above, IPython has a set of "magic commands" that provide additional capabilities that are useful in interactive workflows. These magic commands are prefixed with the `%` symbol and have a syntax that mimics that of the system shell. Here we use the `%timeit` magic command to time the execution of the `jarrow_rudd` function for the American put option we priced above:

```
In [5]: %timeit jarrow_rudd(100.0,
100.0, 1.0, 0.3, 0.03, -1, True, 100)
10 loops, best of 3: 90 ms per loop
```

This time of 90 milliseconds per call is our performance baseline for this algorithm. We will use the `%timeit` magic command subsequently to measure the speed up of more efficient implementations of the algorithm relative to this baseline.

2.2 *Compiling the module using Cython*

Next, we show how the plain Python version of "binomial.py" can be compiled into a C extension module using Cython to improve its performance. Here are the steps involved in the process:

(1) Create a Cython source code file with a `.pyx` extension.[23] The Cython language is a superset of Python so these files can contain arbitrary Python code as well as Cython specific extensions to the language. These language extensions include C/C++ static type declarations and calls to other C/C++/Fortran libraries.
(2) Use the Cython compiler to compile the `.pyx` source code into a Python C extension module. In this phase, Cython uses the static type declarations to generate highly optimized C code. This C code is callable from Python once it is compiled by a C compiler and Cython takes care of converting datatypes between C and Python as needed.
(3) Import the C extension module and use it just like you would the plain Python version.

The benefit of this approach is that you can start with the slow, plain Python version of a code, and then incrementally improve its performance using the Cython language extensions. Throughout the process, the result remains usable interactively from

Python and IPython. At the end of the day, you have a single high-performance implementation that is both the prototype and production version. We now illustrate this incremental approach for our binomial tree pricing example.

2.2.1 Compiling with no C type declarations or C library calls

In this section we compile our binomial tree example using Cython, but without using any of the Cython language extensions, such as C type declarations or C library calls. To do this, we copy our original and unmodified Python program "binomial.py" into a new file called "binomial2.pyx" and compile it *as is* using Cython.

We fire up IPython and import the compiled C extension module. Note that before doing this we must import and install the `pyximport` module, which comes with Cython and enables `.pyx` files to be compiled on-the-fly during the import process:

```
In [1]: import pyximport

In [2]: pyximport.install()

In [3]: from binomial2 import
        jarrow_rudd
```

To see if there is any performance improvement after compilation, we again use IPython's `%timeit` magic command to time the pricing of the American put option:

```
In [4]: %timeit jarrow_rudd(100.0,
100.0, 1.0, 0.3, 0.03, -1, True, 100)
10 loops, best of 3: 82.6 ms per loop
```

Note that the speed up is relatively small. The function took 82.6 milliseconds in Cython versus 90 milliseconds for our plain Python version. This is only a 9% speed up from the original run time. The main purpose of this step is to show how Cython can take a plain Python program and compile it to C. Unless we exploit the Cython language extensions, however, the performance gains will be minimal. We turn to this next.

2.2.2 Compiling with C type declarations and C library calls

Fortunately, the changes required to utilize the Cython language extensions in our original program to exploit the benefits of C compilation are simple to make. The resulting optimized version of the program is saved in a file called "binomial3.pyx" and is shown in Listing 2. There are two optimizations that we have made that allow Cython to generate optimized C code.

First, we use the `cdef` Cython keyword to declare static C types for our variables. When those variables represent basic C datatypes (double, int), simple statements suffice, such as:

```
cdef double h, u, d, drift, q
cdef int i, j, m
```

For the multidimensional arrays representing the stock and option price trees, we use Cython's ~~has~~ built-in support for NumPy arrays. By using `cdef` and telling Cython the number of dimensions (`ndim=2`) and the underlying C datatype of the array (`np.double_t`),

```
cdef np.ndarray[np.double_t, ndim=2]
stkval = np.zeros((n+1,n+1))
```

Cython can generate fast C code that accesses the underlying memory buffer of the NumPy array using fast pointer arithmetic. The result is that Python style array element access such as `stkval[i,j]` becomes as fast as handwritten C code.

Listing 2 binomial3.pyx

```
import numpy as np
import math

cimport numpy as np
cimport cython

cdef extern from "math.h" nogil:
    double exp(double)
    double sqrt(double)
    double pow(double, double)
    double fmax(double, double)

def jarrow_rudd(double s, double k, double t, double v,
                double rf, double cp, int am=0, int n=100):
    """Price an option using the Jarrow Rudd binomial model.

    s : initial stock price
    k : strike price
    t : expiration time
    v : volatility
    rf : risk-free rate
    cp : +1/-1 for call/put
    am : 1/0 for American/European
    n : binomial steps
    """
    cdef double h, u, d, drift, q
    cdef int i, j, m
    cdef np.ndarray[np.double_t, ndim=2] stkval = np.zeros((n+1,n+1))
    cdef np.ndarray[np.double_t, ndim=2] optval = np.zeros((n+1,n+1))

    # Basic calculations
    h = t/n
    u = exp((rf-0.5*pow(v,2))*h+v*sqrt(h))
    d = exp((rf-0.5*pow(v,2))*h-v*sqrt(h))
    drift = exp(rf*h)
    q = (drift-d)/(u-d)

    # Process the terminal stock price
    stkval[0,0] = s
    for i in range(1,n+1):
        stkval[i,0] = stkval[i-1,0]*u
        for j in range(1,i+1):
            stkval[i,j] = stkval[i-1,j-1]*d

    # Backward recursion for option price
    for j in range(n+1):
        optval[n,j] = fmax(0,cp*(stkval[n,j]-k))
    for m in range(n):
        i = n-m-1
        for j in range(i+1):
            optval[i,j] = (q*optval[i+1,j]+(1-q)*optval[i+1,j+1])/drift
            if am==1:
                optval[i,j] = fmax(optval[i,j],cp*(stkval[i,j]-k))

    return optval[0,0]
```

Second, we replace calls to Python's built-in `math` module with optimized ones from the C standard library. This is done by simply telling Cython the name of the corresponding header file ("math.h") and listing the statically typed function declarations therein:

```
cdef extern from "math.h" nogil:
    double exp(double)
    double sqrt(double)
    double pow(double, double)
    double fmax(double, double)
```

From that point on, these C library functions can be used like any other Python function. With these simple Cython language extensions, Cython is able to generate C code that is nearly identical to what would be written by an experienced human developer in C. However, the big difference is that the resulting code is still callable from Python. Running the program inside IPython gives the following results:

```
In [2]: import pyximport, numpy

In [3]: pyximport.install
(setup_args=dict(include_dirs=
[numpy.get_include()]))

In [4]: from binomial3 import
jarrow_rudd

In [5]: jarrow_rudd(100.0, 100.0,
1.0, 0.3, 0.03, -1, True, 100)
Out[5]: 10.627682380924412

In [6]: %timeit jarrow_rudd(100.0,
100.0, 1.0, 0.3, 0.03, -1, True, 100)
1000 loops, best of 3: 579 us per loop
```

We now see that the run time has dropped to 579 microseconds, i.e., 0.579 milliseconds, compared to 90 milliseconds in original Python. This is a vast improvement (a 99% speed up), i.e., the program now runs 155 times as fast. Furthermore, this performance is comparable to that of handwritten pure C code.[24] Note that the actual run times will differ, depending on which machine the program is run on. These examples were run on an iMac with a 2-GHz Intel Core Duo processor and 1.5 GB SDRAM, and should run much faster on more recent hardware.

3 Black–Scholes pricing

A common question that arises is whether using the Cython language extensions to Python to generate optimized C code is always beneficial. It is possible that the benefit depends on the type of algorithm. To assess this question and for completeness, we provide here the Black–Scholes model (Black and Scholes, 1973), programmed in Python and in Cython. The example also shows the use of the SciPy statistics package (`scipy.stats`) to compute the cumulative normal distribution function. The initial plain Python version (in the file "blackscholes.py") is shown in Listing 3.

Running the program in python gives the following:

"Python" (capitalize)

```
In [1]: from blackscholes import
black_scholes

In [2]: black_scholes(100.0, 100.0,
1.0, 0.3, 0.03, 0.0, -1)
Out[2]: 10.327861752731728

In [3]: %timeit black_scholes(100.0,
100.0, 1.0, 0.3, 0.03, 0.0, -1)
1000 loops, best of 3: 409 us per loop
```

Note that the program is running in microseconds now. We now rework the same program using Cython using static C type declarations and C library calls. The program code ("blackscholes2.pyx") is shown in Listing 4.

replace "using" with "in"

Listing 3 blackscholes.py

```
from scipy import stats
import math

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 = (math.log(s/k)+(rf-div+0.5*math.pow(v,2))*t)/(v*math.sqrt(t))
    d2 = d1 - v*math.sqrt(t)
    optprice = cp*s*math.exp(-div*t)*stats.norm.cdf(cp*d1) - \
        cp*k*math.exp(-rf*t)*stats.norm.cdf(cp*d2)
    return optprice
```

Listing 4 blackscholes2.pyx

```
cdef extern from "math.h" nogil:
    double exp(double)
    double sqrt(double)
    double pow(double, double)
    double log(double)
    double erf(double)

cdef double std_norm_cdf(double x):
    return 0.5*(1+erf(x/sqrt(2.0)))

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
    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
```

The program run gives the same values of course, but its performance has improved greatly:

```
In [1]: import pyximport

In [2]: pyximport.install()

In [3]: from blackscholes2 import
black_scholes

In [4]: black_scholes(100.0, 100.0,
1.0, 0.3, 0.03, 0.0, -1)
Out[4]: 10.327861752731728

In [5]: %timeit black_scholes(100.0,
100.0, 1.0, 0.3, 0.03, 0.0, -1)
1000000 loops, best of 3: 716 ns
per loop
```

The run time is now down to 716 nanoseconds, representing a speedup of 571 times over the plain Python version! Again, like the binomial tree model above, there is a huge advantage to using the Cython-optimized version. One may wonder why the speedup of the Black–Scholes algorithm was so much greater than the binomial tree model. While we have not confirmed this, we expect that the binomial tree algorithm is closer to being memory bandwidth limited on the hardware used, so there is less room for improvement.

4 Parallelization

While Cython brings the performance of Python near to that of C/C++/Fortran, at times, even higher performance is needed. Typically, that means parallelizing a program to take advantage of various parallel hardware architectures, such as multicore CPUs, GPU, clusters, and supercomputers. While it is outside the scope of this paper to cover parallelization techniques and libraries for Python in depth, it is useful to summarize some of the more important options that exist in the Python ecosystem.

For low-level message passing, there are Python bindings to MPI[25] (mpi4py[26]), ZeroMQ[27] (PyZMQ[28]) and AMQP.[29] For parallel libraries with higher level interfaces, there is Python's built-in multiprocessing library,[30] IPython's parallel computing framework (IPython, 2007), Disco[31] and Parallel Python.[32] Apache's Hadoop project,[33] which implements Google's MapReduce model of parallelism, allows map and reduce functions to be written in Python. The PyCUDA[34] and PyOpenCL[35] projects make it quite easy to use GPUs for numerical computing through CUDA and OpenCL. Finally, for parallel computing in the cloud, there is PiCloud,[36] which is a commercial package that enables Python code to run in parallel on Amazon EC2. These tools and libraries allow Python programs to achieve even higher levels of performance and scalability and are making Python a compelling language for high-performance computing.

"ease-of-development"

5 Conclusion

Traditionally, easy-of-development and high-performance have been at odds with each other in technical computing. With the Python programming language, this dichotomy is easing. We have described how Cython allows programs written in Python to be quickly optimized to achieve high performance without losing the easy prototyping and interactive workflow that technical users find so productive.

There are many applications in finance that require high performance, such as pricing on trees, Monte Carlo simulations, high-frequency trading, risk management, and portfolio optimization. All these will be able to take advantage of the approach described here. In addition to the performance improvement, this approach eliminates the need to develop and maintain separate

Matlab/Mathematica/Excel (for prototyping) and C/C++/Fortran versions (for production) of algorithms.

Finally, because Python, Cython and the other software libraries described here are all completely open source,[37] they are free to use and have source codes that can be modified, extended, and bundled in any way needed.

Notes

1 http://www.python.org
2 See *Computing in Science and Engineering* 9 (2007) for an overview of Python in scientific computing.
3 http://ipython.scipy.org
4 http://www.scipy.org/F2py
5 http://www.swig.org/
6 http://www.boost.org/
7 http://docs.python.org/library/ctypes.html
8 http://www.cython.org/
9 http://ironpython.codeplex.com/
10 http://www.jython.org/
11 http://numpy.scipy.org/
12 http://www.scipy.org/
13 http://matplotlib.sourceforge.net/
14 http://code.enthought.com/chaco/
15 http://code.enthought.com/projects/mayavi/
16 http://www.enthought.com/
17 http://www.vni.com/products/imsl/pyimslstudio/
18 http://www.sagemath.org/
19 http://www.pythonxy.com/
20 http://www.cython.org
21 This includes the possibility of adding two objects that can't be added in which case a runtime exception will be raised.
22 This capability is similar to the new dynamic language runtime features of .NET.
23 The ".pyx" file extension name comes from Cython's predecessor, called Pyrex.
24 At this point, the only overhead is the conversion of Python datatypes to and from C datatypes entering and leaving the function.
25 http://www.mcs.anl.gov/research/projects/mpi/
26 http://code.google.com/p/mpi4py/
27 http://www.zeromq.org/
28 http://github.com/zeromq/pyzmq
29 http://www.amqp.org
30 http://docs.python.org/library/multiprocessing.html
31 http://discoproject.org/
32 http://www.parallelpython.com/
33 http://hadoop.apache.org/
34 http://mathema.tician.de/software/pycuda
35 http://mathema.tician.de/software/pyopencl
36 http://www.picloud.com/
37 Most of the projects described have very liberal open source licenses such as the BSD or MIT licenses.

References

Behnel, S., Bradshaw, R., Seljebotn, D. S. Ewing, G. *et al.* (2010). "Cython: C=Extensions for Python." http://www.cython.org.

Black, F. and Scholes, M. (1973). "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy* **81**, 637–654.

Cox, J., Ross, S. and Rubinstein, M. (1979). "Option Pricing: A Simplified Approach." *Journal of Financial Economics* 7, 229–263.

Jarrow, R. and Rudd, A. (1983). *Option Pricing*, Homewood, Illinois: Irwin.

Perez, F. and Granger, B. E. (2007). "IPython: A System for Interactive Scientific Computing." *Computing in Science and Engineering* **9**, 21–29.

Python Success Stories (2010). http://www.python.org/about/success/.

TIOBE Programming Community Index, http://www.tiobe.com/ (August 2010).

van Rossum, G. *et al.* (2010). "The Python Programming Language." http://www.python.org.

Keywords: Python, Cython, option pricing, performance computing.