Python for unified research in econometrics and statistics

Roseline Bilina, Steve Lawford

Python for Unified Research in Econometrics and Statistics*

Roseline Bilina† Steve Lawford‡

Cornell University ENAC

July 27, 2010

Abstract

Python is a powerful high-level open source programming language, that is available for multiple platforms. It supports object-oriented programming, and has recently become a serious alternative to low-level compiled languages such as C++. It is easy to learn and use, and is recognized for very fast development times, which makes it suitable for rapid software prototyping as well as teaching purposes. We motivate the use of Python and its free extension modules for high performance stand-alone applications in econometrics and statistics, and as a tool for gluing different applications together. (It is in this sense that Python forms a 'unified' environment for statistical research). We give details on the core language features, which will enable a user to immediately begin work, and then provide practical examples of advanced uses of Python. Finally, we compare the run-time performance of extended Python against a number of commonly-used statistical packages and programming environments.

*JEL classification: C6 (Mathematical methods and programming), C87 (Econometric software), C88 (Other computer software). Keywords: Object-Oriented Programming, Open Source Software, Programming Language, Python, Rapid Prototyping.

†Roseline Bilina, School of Operations Research and Information Engineering, Cornell University, Ithaca, NY, 14853, USA. Email: rb537 (at) cornell.edu.

‡Corresponding author. Steve Lawford, Department of Economics and Econometrics (LH/ECO), ENAC, 7 avenue Edouard Belin, BP 54005, 31055, Toulouse, Cedex 4, France. Email: steve_lawford (at) yahoo.co.uk.

1 Introduction

"And now for something completely different."

(catch phrase from Monty Python's Flying Circus.)

Python is a powerful high-level programming language, with object-oriented capability, that was designed in the early 1990s by Guido van Rossum, then a programmer at the Dutch National Research Institute for Mathematics and Computer Science (CWI) in Amsterdam. The core Python distribution is open source and is available for multiple platforms, including Windows, Linux/Unix and Mac OS X. The default CPython implementation, as well as the standard libraries and documentation, are available free of charge from `www.python.org`, and are managed by the Python Software Foundation, a non-profit body.[1] van Rossum still oversees the language development, which has ensured a strong continuity of features, design, and philosophy. Python is easy to learn and use, and is recognized for its very clear, concise, and logical syntax. This feature alone makes it particularly suitable for rapid software prototyping, and greatly eases subsequent program maintenance and debugging, and extension by the author or another user.

In software development, there is often a trade-off between computational efficiency and final performance, and programming efficiency, productivity, and readability. For both applied and theoretical econometricians and statisticians, this frequently leads to a choice between low-level languages such as C++, and high-level languages or software such as PcGive, GAUSS, or Matlab (e.g. [23] and [27]). A typical academic study might involve development of asymptotic theory for a new procedure with use of symbolic manipulation software such as Mathematica, assessment of the finite-sample properties through Monte Carlo simulation using C++ or Ox, treatment of a very large microeconometric database in MySQL, preliminary data analysis in EViews or Stata, production of high quality graphics in R, and finally creation of a written report using LaTeX. An industrial application will often add to this some degree of automation (of data treatment or

[1] For brevity, we will omit the prefix `http://` from internet URL references throughout the paper.

updating, or of report generation), and frequently a user-friendly front-end, perhaps in Excel.

We will motivate the use of Python as a particularly appropriate language for high performance stand-alone research applications in econometrics and statistics, as well as its more commonly known purpose as a scripting language for gluing different applications together. In industry and academia, Python has become an alternative to low-level compiled languages such as C++. Recent examples in large-scale computational applications include [4], [16], [17], [19] and [20, who explicitly refers to faster development times], and indicate comparable run times with C++ implementations in some situations (although we would generally expect some overhead from using an interpreted language). The Python Wiki lists Google, Industrial Light and Magic, and Yahoo! among major organizations with applications written in Python.[2] Furthermore, Python can be rapidly mastered, which also makes it suitable for training purposes ([3] discusses physics teaching).

The paper is organized as follows. Section 2 explains how Python and various important additional components can be installed on a Windows machine. Section 3 introduces the core features of Python, which are straightforward, even for users with little programming experience. While we do not attempt to replace the excellent book length introductions to Python such as [2, a comprehensive treatment of standard modules], [8, with good case studies and exercises], [11, with detailed examples], and [14, more oriented towards computational science], we provide enough detail to enable a user to immediately start serious work. Shorter introductions to the core language include the Python tutorial [32], which is regularly updated.[3] Python comes into its own as a general programming language when some of the many free external modules are imported. In Section 4, we detail some more advanced uses of Python, and show how it can be extended with scientific modules (in particular, NumPy and SciPy), and used to link different parts of a research project. We illustrate practical issues such as data input and output, access to the graphical capabilities of R (through the `rpy` module), automatic creation of a LaTeX code (program segment)

[2] See `wiki.python.org/moin/OrganizationsUsingPython`.

[3] For additional documentation, see [25], and references in, e.g. [14] and `www.python.org/doc/`.

from within Python (with numerical results computed using `scipy`), object-oriented programming in a rapid prototype of a copula estimation, incorporation of C++ code within a Python program, and numerical optimization and plotting (using `matplotlib`). In Section 5, we compare the speed of extended Python to a number of other scientific software packages and programming environments, in a series of tests. We find comparable performance to languages such as Ox, for a variety of mathematical operations. Section 6 concludes the paper, and mentions additional useful modules. Supplementary material is contained in Appendix A.

2 Installation of packages

Here, we describe the procedure for installation of Python and some important additional packages, on a Windows machine. We use Windows for illustration only, and discussion of installation for other operating systems is found in [11] and [33], and is generally straightforward. We use Python 2.6.5 (March 2010), which is the most recent production version for which compatible versions of the scientific packages NumPy and SciPy are available. Python 2.6.5 includes the core language and standard libraries, and is installed automatically from `www.python.org/download`, by following the download instructions.[4] After installation, the Python Integrated Development Environment (IDLE), or 'shell window' (interactive interpreter) becomes available (see Figure 1). The compatible Pywin32 should also be installed. This provides extension modules for access to many Windows API (Application Programming Interface) functions, and is available from `sourceforge.net`.

[4]The Windows Installer file for Python 2.6.5 is `python-2.6.5.msi`. The latest version of Python, 3.0, is less well-tested, and we do not use it in this paper. The Pywin32 file (build 210) is `pywin32-210.win32-py2.6.exe`. The NumPy 1.4.1 file is `numpy-1.4.1-win32-superpack-python2.6.exe`. The SciPy 0.8.0 file is `scipy-0.8.0rc2-win32-superpack-python2.6.exe`. The R-2.9.1 file is `R-2.9.1-win32.exe`. The RPy 1.0.3 file is `rpy-1.0.3-R-2.9.0-R-2.9.1-win32-py2.6.exe`. The MDP file is `MDP-2.6.win32.exe`. The Matplotlib 1.0.0 file is `matplotlib-1.0.0.win32-py2.6.exe`. The automatic MinGW 5.1.6 installer file is `MinGW-5.1.6.exe`. We have tested the installations on a 1.73GHz Celeron M machine with 2GB RAM running Windows Vista, and a 1.66MHz Centrino Duo machine with 1GB RAM running Windows XP. Further information on IDLE is available at `docs.python.org/library/idle.html`, while some alternative environments are listed in [11, Table 1-1]; also see IPython (`ipython.scipy.org/moin/Documentation`).

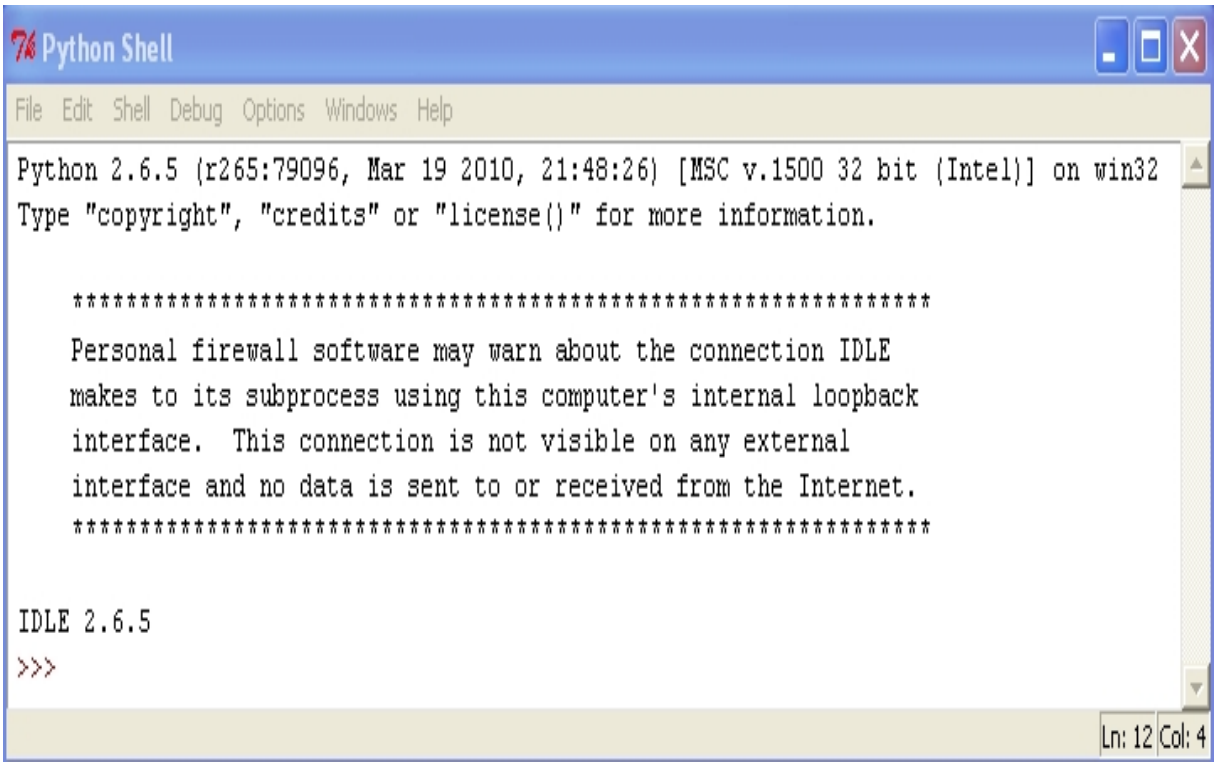

Figure 1: The Python 2.6.5 Integrated Development Environment (IDLE).

Two open source packages provide advanced functionality for scientific computing. The first of these, NumPy (`numpy.scipy.org`), enables Matlab-like multidimensional arrays and array methods, linear algebra, Fourier transforms, random number generation, and tools for integrating C++ and Fortran code into Python programs (see [21] for a comprehensive manual, and [14, Chapter 4] for applications). NumPy version 1.4.1 is stable with Python 2.6.5 and Pywin32 (build 210), and is available from `sourceforge.net` (follow the link from `numpy.scipy.org`). The second package, SciPy (`www.scipy.org`), which requires NumPy, provides further mathematical libraries, including statistics, numerical integration and optimization, genetic algorithms, and special functions. SciPy version 0.8.0 is stable with the above packages, and is available from `sourceforge.net` (follow the link from `www.scipy.org`); see [29] for a reference. We use NumPy and SciPy extensively below.

For statistical computing and an excellent graphical interface, Python can be linked to the R language (`www.r-project.org`) via the RPy interface (`rpy.sourceforge.net`). The strengths of R are discussed at length in [6], [12], [26] and [34]. An R journal exists (`journal.r-project.org`). Release R 2.9.1 (June 2009; follow the download links on `rpy.sourceforge.net`, and choose 'full installation') and RPy 1.0.3, available from `sourceforge.net`, are stable with Python 2.6.5.

An additional third-party module that is useful for data-processing is MDP 2.6: Modular Toolkit for Data Processing (see [35] for details). It contains a number of learning algorithms and, in particular, user-friendly routines for principal components analysis. It is available from `mdp-toolkit.sourceforge.net`. We also use Matplotlib 1.0.0, a comprehensive and Matlab-like advanced plotting tool, available from `sourceforge.net/projects/matplotlib`. Documentation ([7]) and examples can be found at `matplotlib.sourceforge.net`. A C++ compiler is also needed to run Python programs that contain C++ code segments, and a good free option is a full MinGW 5.1.6 (Minimalist GNU for Windows) installation. An automatic Windows installer is available from `www.mingw.org` (which links to `sourceforge.net`), and contains the GCC (GNU Compiler System), which supports C++. We refer to the above installation as 'extended Python', and use it throughout the paper, and especially in Sections 4 and 5. We have installed the individual packages for illustration, but bundled scientific distributions of Python and additional packages are available. These include pythonxy for Windows (`code.google.com/p/pythonxy/`) and the Enthought Python Distribution (`www.enthought.com/products/epd.php`), which is free for academic use.

Python is well supported by a dynamic community, with helpful online mailing lists, discussion forums, and archives. A number of Python-related conferences are held annually. A general discussion list is available at `mail.python.org/mailman/listinfo/python-list`, and the Python forum is at `www.python-forum.org/pythonforum/index.php`. It is always advisable to check the archives before posting new requests.[5]

[5] A full listing of mailing lists is available from `mail.python.org/mailman/listinfo`. The `www.python.org` helpdesk can be contacted at `help (at) python.org`. SciPy and NumPy mailing lists are available at

2.1 Modular code and package import

The core Python 2.6.5 implementation is made much more powerful by standard ([31]) and third-party *modules* (such as RPy and SciPy). A module is easily imported using the `import` command (this does not automatically run the module code.). For clarity, this is usually performed at the start of a program. For instance (see Example 0 below), `import scipy` (Python is case-sensitive) loads the main `scipy` module and methods, which are then called by, e.g. `scipy.pi` (this gives π). The available `scipy` packages (within the main module) can be viewed by `help(scipy)`. If a single `scipy` package is of interest, e.g. the `stats` package, then this can be imported by `import scipy.stats` (in which case methods are accessed as, e.g. `scipy.stats.kurtosis()`, which gives the excess kurtosis), or `from scipy import stats` (in which case methods are accessed by, e.g. `stats.kurtosis()`). It is often preferable to use the former, since it leads to more readable programs, while the latter will also overwrite any current packages called `stats`. Another way to overcome this problem is to rename the packages upon import, e.g. `from scipy import stats as NewStatsPackage`. If all `scipy` packages are of interest, then these can be imported by `from scipy import *`, although this will also overwrite any existing packages with the same names.

Longer Python programs can be split into multiple short modules, for convenience and re-usability. For instance, in Example 3 below, it is suggested that a user-defined LaTeX table function `tex_table` be saved in a file `tex_functions.py` (the module name is then `tex_functions`). As above, the function can be imported by `import tex_functions.tex_table` and then used directly by `tex_functions.tex_table()`. In the examples, we will use both forms of import.[6]

`mail.scipy.org/mailman/listinfo/scipy-user` and `mail.scipy.org/mailman/listinfo/numpy-discussion`. An RPy mailing list is at `lists.sourceforge.net/lists/listinfo/rpy-list`. Conference announcements are posted at `www.python.org/community/workshops`.

[6]When a script is run that imports `tex_functions`, a *compiled* Python file `tex_functions.pyc` will usually be created automatically in the same directory as `tex_functions.py`. This serves to speed up subsequent start-up (module load) times of the program, as long as the file `tex_functions.py` is not modified. A list of all functions defined within a module (and of all functions defined within all imported modules) is given by `dir()`, e.g. `import tex_functions` and `dir(tex_functions)` would give a list including '`tex_table`'.

3 Language basics

The IDLE can be used to test short commands in real-time (input is entered after the prompt `>>>`). Groups of commands can be written in a new IDLE window, saved with a `.py` suffix, and executed as a regular program in IDLE, or in a DOS window by double-clicking on the file. Single-line comments are preceded by a hash `#`, and multi-line comments are enclosed within multiple quotes `"""`. Multiple commands can be included on one line if separated by a semi-colon, and long commands can be enclosed within parentheses `()` and split over several lines (a back-slash can also be used at the end of each line to be split). A general Python object `a` (e.g. a variable, function, class instance) be used as a function *argument* `f(a)`, or can have *methods* (functions) applied to it, with dot syntax `a.f()`. There is no need to declare variables in Python since they are created at the moment of initialization. Objects can be printed to the screen by entering the object name at the prompt (i.e. `>>> a`) or from within a program with `print a`.

3.1 Basic types: numbers and strings

The operators `+`, `-`, `*` and `/` work for the three most commonly-used *numeric types*: integers, floats, and complex numbers (a real and an imaginary float). Float division is `x/y`, which returns the floor for integers. The modulus `x%y` returns the remainder of x divided by y, and powers x^y are given by `x**y`. Variable assignment is performed using `=`, and must take place before variable use. Python is a *dynamic language*, and variable types are checked at run-time. It is also *strongly-typed*, i.e. following variable initialization, a change in type must be explicitly requested.[7]

[7] Some care must be taken with variable assignment, which manipulates 'references', e.g. `a=3; b=a` does not make a copy of `a`, and so setting `a=2` will leave `b=3` (the old reference is deleted by re-assignment). Some standard mathematical functions are available in the `scipy.math` module. Note that variables may be assigned to functions, e.g. `b=math.cos` is allowed, and `b(0)` gives `1.0`. As for many other mathematical packages, Python also supports *long integer* arithmetic, e.g. `(2682440**4)+(15365639**4)+(18796760**4)` and `(20615673**4)` both give the 30-digit number `180630077292169281088848499041L` (this verifies the counterexample in [9] to Euler's (incorrect) conjectured generalization of Fermat's Last Theorem: here, that three fourth powers never sum to a fourth power.)

```
>>> x=y=2; z=(2+3j)*(2-3j); print x, y, z, z.real, type(z), 3/2, 3.0/2.0, 3%2, 3**2 \
   # simultaneous assignment, complex numbers, type, division, modulus, power
2 2 (13+0j) 13.0 <type 'complex'> 1 1.5 1 9
>>> y=float(x); print x, type(x), y, type(y) # variable type re-assignment
2 <type 'int'> 2.0 <type 'float'>
```

Python is particularly good at manipulating *strings*, which are immutable unless re-assigned. Strings can be written using single (or double) quotes, concatenated using `+`, repeated by `*`, and 'sliced' with the *slicing* operator `[r:s]`, where element `s` is not included in the slice (indexation starts at 0). Negative indices correspond to position relative to the right-hand-side. Numbers are converted to strings with `str()`, and strings to floats or integers by `float()` or `int()`.[8] When parsing data, it is often useful to remove all start and end whitespace, with `strip()`.

```
>>> a='data1'; b=a+'data2'; c=a*2; d=str(x); print b, c, b[5:]+b[:-5], d, len(a), 'data3' in b \
   # string operations: +, *, type conversion, slicing, len(), string search
data1data2 data1data1 data2data1 2 5 False
>>> e='   data1  data2       data3   '; print e.strip() # the strip() method
data1  data2       data3
```

3.2 Container types: lists and dictionaries

Python has a number of useful built-in data structures. A *list* is a mutable ordered set of *arbitrary* comma-separated objects, such as numbers, strings, and other lists. Lists (like strings) can be manipulated using `+`, `*`, and the slicing operator `[r:s]`. A list copy can be created using `[:]`. Nested list elements are indexed by `[r][s][···]`. Lists can be *sorted* in ascending order (numerically

[8] The `raw_input()` command can be used to prompt user input, e.g. `data=raw_input('enter:')` will prompt with '`enter:`' and creates a string `data` from the user input. Example 3 below shows that double and triple backslashes in a string will return a single and a double backslash when the string is printed (this is useful when automatically generating LaTeX code). An alternative is to use a raw string `r'string'`, which will almost always be printed as entered. It is also possible to search across strings, e.g. `a in 'string'`, while `len(a)` gives the number of characters in the string. Python also provides some support for Unicode strings (`www.unicode.org`).

and then alphabetically) with the method `sort()`, or reversed using `reverse()`. Lists can also be sorted according to complicated metrics, and this can be very useful in scientific computing. For instance, if `A=[[1,2],[4,6]]`, `B=[[3,1],[2,4]]` and `C=[[1,0,0],[0,2,0],[0,0,3]]` define three matrices, contained in a list `x=[A,B,C]`, then `x` can be ordered by the determinant (say) using `x.sort(key=det)`, which will give `x=[A,C,B]`, and where the `det` function has been imported by `from scipy.linalg import det`. List membership can be tested by `in`.

```
>>> output=[1.27,'converged',0,[1.25,'no']]; output+=[['ols','gmm']]; output.sort()
>>> print output, 'converged' in output, output[3][1], output[1:3] # list manipulation and methods
[0, 1.27, [1.25, 'no'], ['ols', 'gmm'], 'converged'] True gmm [1.27, [1.25, 'no']]
```

Strings can be *split* into a list of elements using the `split()` method, which is very useful when parsing databases. Note that methods can be combined, as in `strip().split()`, which are executed from left to right. New lists can be constructed easily by *list comprehension*, which loops over existing lists, may be combined with conditions, and can return nested lists.[9] The `enumerate()` function loops over the elements of a list, and returns their position (index) and value, and the `zip()` function can be used for pairwise-combination of lists.

[9]The slicing operator can also take a step-length argument, i.e. `[r:s:step]`. An empty list of length n is given by `[None]*n`, and `len()` gives the number of elements. The `append()` and `extend()` methods can also be used instead of `+=`, for lists, or elements of lists, respectively. List elements can be assigned to with `a[i]=b`, and a list slice can be replaced with a slice of a *different* length. Items can be added at a specific index with `insert()`, and removed with `remove()` or `pop()`. The index of a given element can be found using `index()`, and its number of occurrences by `count()`. Slices can be deleted (and the list dimension changed) by `del a[r:s]`. A Python *tuple* is essentially an immutable list that can be created from a list using the `tuple()` function. They behave like lists, although list methods that change the list cannot be applied to them. Tuples are often useful as dictionary keys (see below). The default of `split()` is to split on all runs of whitespace, and the inverse of e.g. `a.split(';')` is `';'.join(a.split(';'))`. Also, `range(n)` is equivalent to `[0,1,2,...,n-1]`. Another useful string method is `a.replace('string1','string2')`, which replaces all occurrences of '*string1*' in `a` with '*string2*'.

```
>>> print 'data1 data2 data3'.split(), 'data1;data2;data3'.split(';'), \
      '      data1:data2:data3      '.strip().split(':') # the split() method
['data1', 'data2', 'data3'] ['data1', 'data2', 'data3'] ['data1', 'data2', 'data3']
>>> squares_cubes=[[x**2,x**3] for x in range(6) if x**2<=16]; print squares_cubes \
    # list comprehension
[[0, 0], [1, 1], [4, 8], [9, 27], [16, 64]]
>>> for x,y in enumerate(squares_cubes): print x,y, # the enumerate() function
0 [0, 0] 1 [1, 1] 2 [4, 8] 3 [9, 27] 4 [16, 64]
>>> print zip(range(5),squares_cubes) # the zip(function)
[(0, [0, 0]), (1, [1, 1]), (2, [4, 8]), (3, [9, 27]), (4, [16, 64])]
```

Python *dictionaries* (also known as 'hash tables' or 'associative arrays') are flexible mappings, and contain values that are indexed by unique *keys*. The keys can be any immutable data type, such as strings, numbers and tuples. An empty dictionary is given by `a={}`, a list of keys is extracted using `a.keys()`, and elements are accessed by `a[`*`key`*`]` or `a.get(`*`key`*`)`. Elements can be added to (or re-assigned) and removed from dictionaries.[10] Dictionaries can also be constructed using the `dict()` function and list comprehension.

```
>>> results={'betahat':[-1.23,0.57],'loglik':-7.6245,'R2':0.18,'convergence':'yes'} \
    # dictionary construction
>>> print results.keys(), results['betahat'][1], results['R2']>0.50 # dictionary manipulation
['convergence', 'loglik', 'R2', 'betahat'] 0.57 False
>>> print dict([(x,[x**2,x**3]) for x in range(5)]) # dictionary build from list
{0: [0, 0], 1: [1, 1], 2: [4, 8], 3: [9, 27], 4: [16, 64]}
```

[10] As for lists, dictionary elements can be re-assigned and deleted, and membership is tested by `in`. For a full list of dictionary methods, see [11, Chapter 4].

3.3 Control structures and user-defined functions

Commands can be executed subject to conditions by using the `if`, `elif` (else if) and `else` statements, and can contain combinations of `==` (equal to), `<`, `>`, `<=`, `>=` and `!=` (not equal to), and the usual Boolean operators `and`, `or` and `not`. Python `while` statements repeat commands if some *condition* is satisfied, and `for` loops over the items of an *iterable* sequence or list, e.g. `for i in a:`.[11] The body of Python control structures and functions is defined by whitespace *indentation*.

```
>>> i=0 # variable initialization
>>> while i<=10: # while loop
        if 2<=i<=8: print i, # a compound conditional statement
        i+=1 # variable increment
2 3 4 5 6 7 8
```

A *function* is defined by `def` and `return` statements, and can be documented by including a `"""` comment within the body, which can be viewed using `help(`*`function_name`*`)`. In Example 0, the function `pdfnorm` returns the density function $f(x)$ of the normal $N(\texttt{mu}, \texttt{sigma**2})$, and specifies *default values* for the arguments `mu=0` and `sigma=1`.[12] The function can be called without these arguments, or they can be reset, or can be called using keyword arguments. Functions can also be called by 'unpacking' arguments from a list using `*[]`. Example 0 gives four ways in which the function can be called to give $f(2) \approx 0.0540$. The function can be evaluated at *multiple arguments* using list comprehension, e.g. `print [pdfnorm(x) for x in range(10)]`.[13] However, it is *much* faster to call the function with a vector *array* argument by `from scipy import arange` and `print pdfnorm(arange(10))`.[14] Python can also deal naturally with *composite functions*, e.g.

[11] The `break` and `continue` statements are respectively used to break out of the smallest enclosing `for` / `while` loop, and to continue with the next iteration. The `pass` statement performs no action, but is useful as a placeholder.

[12] This is for illustration only: in practice, the SciPy `stats` module can be used, e.g. `from scipy.stats import norm`, and then `norm.pdf(x)` gives the same result as `pdfnorm(x)`, for the standard normal $N(0, 1)$.

[13] This is also achieved by passing `pdfnorm` to the `map()` function, i.e. `print map(pdfnorm,range(10))`.

[14] *Array computations* are very efficient. In a simple speed test (using the machine described in Section 5), we compute `pdfnorm(x)` once only, where $x \in \{0, 1, \ldots, 1000000\}$, by (a) `[pdfnorm(x) for x in range(1000000)]`, (b) `map(pdfnorm,range(1000000))`, (c) `from scipy import arange` and `pdfnorm(arange(1000000))`, and also (d) `from`

`scipy.log(pdfnorm(0))` will return $\ln(f(x))$ at $x = 0$.[15]

(Example 0. A user-defined function pdfnorm)

```
from scipy import pi,sqrt,exp # import scipy module pi, sqrt, exp
def pdfnorm(x,mu=0,sigma=1): # function definition and default arguments
    """Normal N(mu,sigma**2) density function. # function documentation string
    Default values mu=0, sigma=1.""" # function name, arguments, string returned by help(pdfnorm)
    return (1/(sqrt(2*pi)*sigma))*exp(-0.5*((x-mu)**2)/(sigma**2)) # return function result
print pdfnorm(2), pdfnorm(2,0,1), pdfnorm(2,sigma=1), pdfnorm(*[2,0,1]) # call pdfnorm
```

Python uses *call by assignment*, which enables implementation of function calls *by value* and *by reference*. Essentially, the call by value effect can be obtained by appropriate use of immutable objects (such as numbers, strings or tuples), or by manipulating *but not re-assigning* mutable objects (such as lists, dictionaries or class instances). The call by reference effect can be obtained by re-assigning mutable objects; see e.g. [14, Sections 3.2.10, 3.3.4] for discussion.

4 Longer examples

4.1 (Example 1.) Reading and writing data

It is straightforward to import data of various forms into Python objects. We illustrate using 100 cross-sectional observations on income and expenditure data in a text file, from [10, Table F.8.1].[16]

`scipy.stats import norm` and `norm.pdf(arange(1000000))`. Tests (a) and (b) both run in roughly 58 seconds, while (c) and (d) take about 0.5 seconds, i.e. the array function is more than 100 times faster. It is encouraging that the user-defined `pdfnorm()` is comparable in speed terms to the SciPy `stats.norm.pdf()`.

[15] Given two functions $f : X \to Y$ and $g : Y \to Z$, where the range of f is the same set as the domain of g (otherwise the composition is undefined), then the composite function $g \circ f : X \to Z$ is defined as $(g \circ f)(x) := g(f(x))$.

[16] The dataset is freely available at `www.stern.nyu.edu/~wgreene/Text/Edition6/TableF8-1.txt`. The variable names ('MDR', 'Acc', 'Age', 'Income', 'Avgexp', 'Ownrent', and 'Selfempl') are given in a single header line. Data is reported in space-separated columns, which contain integers or floats. See Examples 2, 3, and 6 for analysis.

(Example 1a. Creating a data dictionary from a text file)

```
f=open('./TableF8-1.txt','r') # open raw data file for reading, specifying path
header=f.readline().strip().split(); data_dict=dict([x,[]] for x in header) \
        # read first line of data, initialize data dictionary, with header variables as keys
for j in f.readlines(): # read all data into dictionary, matching columns to variables
    for k,l in enumerate(j.strip().split()): data_dict[header[k]].append(eval(l))
f.close() # close data file object
```

In Example 1a, a *file object* `f` is opened with `open()`. A list of variable names, `header`, is created from the *first* line of the file: `readline()` leaves a new line character at the end of the string, which is removed by `strip()`, and the string is split into list elements by `split()`. A dictionary `data_dict` is initialized by list comprehension with keys taken from `header`, and corresponding values `[]` (an empty list). The dictionary is then filled with data (by iterating across the *remaining* lines of `f`), after which the file object is closed by `close()`.[17] The command `eval()` 'evaluates' the data into Python expressions, and the dictionary elements corresponding to the variables 'Acc', 'Age', 'MDR', 'Ownrent' and 'Selfempl' are automatically created as integers, while 'Avgexp' and 'Income' are floats. The formatted data dictionary is then ready for use in applied work.

The `cPickle` module provides a powerful means of saving *arbitrary* Python objects to file, and for retrieving them.[18] The 'pickling' (save) can be applied to, e.g. numbers and strings, lists and dictionaries, top-level module functions and classes, and creates a byte stream (string representation) without losing the original object structure. The original object can be reconstructed

[17] Valid arguments for `open()` are the filename (including the path if this is not the current location) and the mode of use of the file: useful are `'r'` read only (default) and `'w'` write only (`'r+'` is read and write). While `f.readlines()` reads `f` into a *list* of strings, `f.read()` would read `f` into a *single* string. The iteration `for j in f.readlines():` in this example could also be replaced by `for j in f:`.

[18] `cPickle` is an optimized C implementation of the standard `pickle` module, and is reported to be faster for data save and load ([2] and [31, Section 12.2]), although some of the `pickle` functionality is missing. The default `cPickle` save 'serializes' the Python object into a printable ASCII format (other protocols are available). See `docs.python.org/library/pickle.html` for further details. In Example 1b, the raw data file is 3.37k, and the Python `.bin`, which contains additional structural information, is 5.77k. The present authors have made frequent use of `cPickle` in parsing and manipulating the U.S. Department of Transportation Origin and Destination databases.

by 'unpickling' (load). The technique is very useful when *storing* the results of a large dataset parse to file, for later use, avoiding the need to parse the data more than once. It encourages short *modular code*, since *objects* can easily be passed from one code (or user) to another, or sent across a network. The speed of `cPickle` also makes Python a natural choice for *application checkpointing*, a technique which stores the current state of an application, and that is used to restart the execution should the system fail. Applications in econometrics include treatment of massive microeconometric datasets, and intensive Monte Carlo simulation (e.g. a bootstrap simulation, or one with a heavy numerical computation in each of a large number of replications).

In Example 1b, a file object `g` is created, and the dictionary `data_dict` is pickled to the file `python_data.bin`, before being immediately unpickled to a new dictionary `data_dict2`.

(Example 1b. cPickle Python object save and load)

```
import cPickle # import cPickle module and methods
g=open('./python_data.bin','w'); cPickle.dump(data_dict,g); g.close() # data save
h=open('./python_data.bin','r'); data_dict2=cPickle.load(h); h.close() # data load
```

4.2 (Example 2.) Graphics and R

We build on Example 1, and show how Python can be linked to the free statistical language R, and in particular to its graphical features, through the `rpy` module. Example 2 creates an empty *R pdf object* (by `r.pdf`), called `descriptive.pdf`, of dimension 1×2 (by `r.par`), to be filled with two generated plots: the first `r.plot` creates a scatter of the 'Income' and 'Avgexp' variables from `data_dict`, while the second creates a kernel density (`r.density`) plot of the 'Income' variable. The `.pdf` Figure 2 is *automatically* generated, and can be imported into a LaTeX file (as here!).

(Example 2. R graphics from a data dictionary)

```
from rpy import * # import rpy module and methods
r.pdf('./descriptive.pdf'); r.par(mfrow=[1,2]) # create empty R pdf object
(r.plot(data_dict['Income'],data_dict['Avgexp'],xlab='Income',
       ylab='Expenditure',main='Scatterplot',type='p',lwd=3)) # R scatterplot
income_density=r.density(data_dict['Income']) # R estimated kernel density
(r.plot(income_density['x'],income_density['y'], # R kernel plot
        xlab='Income',ylab='frequency',main='Kernel density',type='l',lwd=2))
r.dev_off() # close R pdf object
```

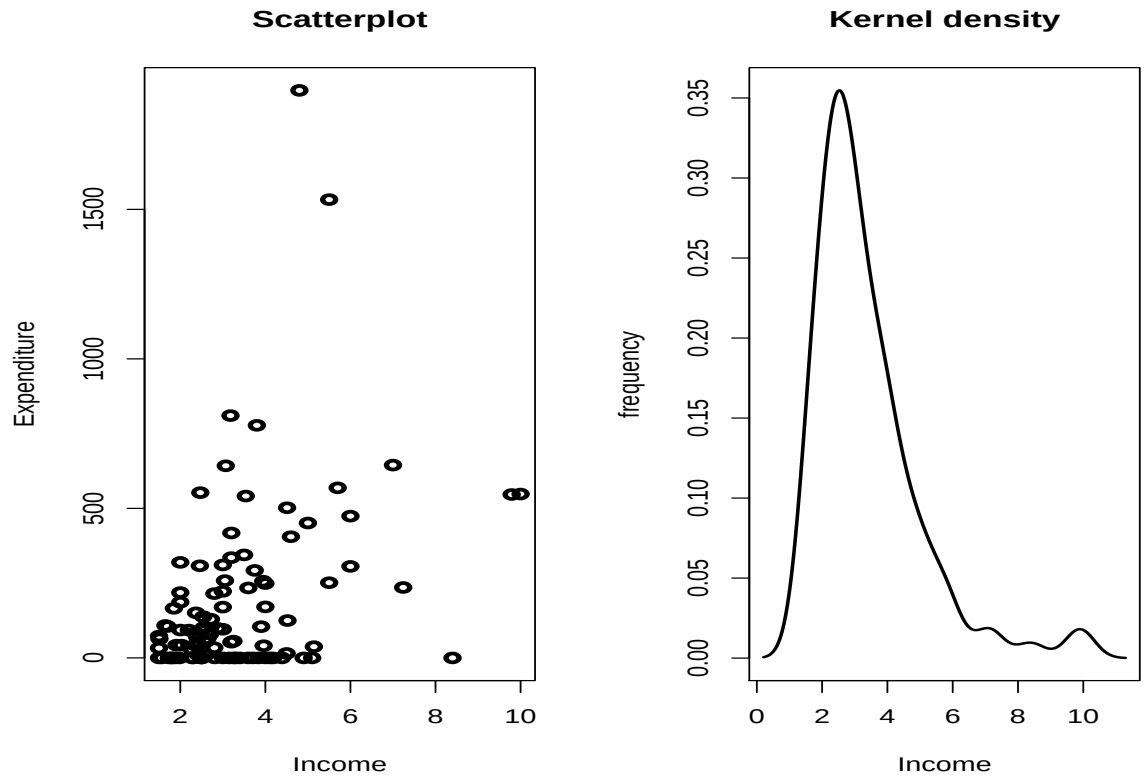

Figure 2: Python/R `.pdf` output plot, generated using `rpy` (see Example 2).

4.3 (Example 3.) Creation of LaTeX code

This example shows how to design a simple function that will create LaTeX code for a table of descriptive statistics (see [13] for related discussion of 'literate econometric practice', where models, data, and programs are dynamically linked to a written report of the research, and R's 'Sweave' for a complementary approach). The user-defined function `tex_table` takes arguments `filename` (the name of the `.tex` file), `data_dict` (a data dictionary, from Example 1), and `cap` (the table caption). The list `variables` contains the *sorted* keys from `data_dict`. The `output` string holds table header information (and where a double backslash corresponds to a single backslash in the string, and a triple backslash corresponds to a double backslash). For each variable name `i`, the `output` string is augmented with the mean, standard deviation, minimum and maximum of the corresponding variable from `data_dict`, computed using the `scipy.stats` functions. The *string formatting* symbol `%` separates a string from values to format. Here, each `%0.2f` (formats a value to a 2 decimal place float) acts as a placeholder within the string, and is replaced from left to right by the values given after the string.[19] The `output` string also contains table footer information, including the caption and a LaTeX label reference. Once `output` has been created, it is written (using the `write` method) to `filename`. The function `return` ends the procedure.

[19] For additional conversion specifiers, see [11, Table 3-1].

(Example 3. LaTeX table from a data dictionary; code saved in tex_functions.py)

```
from scipy.stats import * # import scipy.stats module and methods
def tex_table(filename,data_dict,cap): # user-defined function
    """Create .tex table.""" # function documentation string
    variables=data_dict.keys(); variables.sort() # define variable names (table rows)
    output=('\\begin{table}\n\\begin{center}\n\\begin{tabular}'+ # create output string
        '{'+'c'*5+'}\n'+'variable & mean & stdev & min & max\\\ \\hline\n')
    for i in variables: # loop across rows of table
        output+=i+' & %0.2f & %0.2f & %0.2f & %0.2f\\\ \n' % (mean(data_dict[i]),
            std(data_dict[i]), min(data_dict[i]), max(data_dict[i])) # add data to table
    output+='\\end{tabular}\n\\caption{'+cap+'}\n\\label{tab:tab1}\n\\end{center}\n\\end{table}'
    f=open(filename,'w'); f.write(output); f.close() # write output string to file, close file
    return
```

The `tex_table` function can be saved in a separate 'module' `tex_functions.py`, and then called simply by another Python program (user) with, e.g.:

```
from tex_functions import tex_table
tex_table('./table.tex',data_dict,'Descriptive statistics.') # call .tex table function
```

It is straightforward to import the resulting LaTeX output into a `.tex` file (as in Table 1 in this paper) using a LaTeX command, e.g. `\input{./table.tex}`.[20] The generic code can be used with any data dictionary of the same general form as `data_dict`, and is easily modified.

4.4 (Example 4.) Classes and object-oriented programming

The following example illustrates *rapid prototyping* and *object-oriented* Python, with a simple bivariate copula estimation. Appendix A.1 contains a short discussion of the relevant copula theory. We use 5042 time series observations on the daily closing prices of the Dow Jones Industrial Average

[20] It is also possible to automatically compile a valid *filename*`.tex` file (that includes all relevant preamble and document wrapper material) to a `.pdf` by `import os` followed by `os.system("pdflatex` *filename.tex*`")`.

variable	mean	stdev	min	max
Acc	0.73	0.45	0.00	1.00
Age	32.08	7.83	20.00	55.00
Avgexp	189.02	294.24	0.00	1898.03
Income	3.37	1.63	1.50	10.00
MDR	0.36	1.01	0.00	7.00
Ownrent	0.36	0.48	0.00	1.00
Selfempl	0.05	0.22	0.00	1.00

Table 1: Descriptive statistics.

and the S&P500 over 9 June 1989 to 9 June 2009.[21] The raw data is parsed in Python, and log returns are created, as `x` and `y` (not shown). There are two classes: a *base class* `Copula`, and a *derived class* `NormalCopula`. The methods available in each class, and the class *inheritance* structure, can be viewed by, e.g. `help(Copula)`. A `Copula` *instance* (conventionally referred to by `self`) is *initialized* by `a=Copula(x,y)`, where the initialization takes the data as arguments (and `__init__` is the 'constructor'). The instance variables `a.x` and `a.y` (SciPy data *arrays*) and `a.n` (sample size) become available. The method `a.empirical_cdf()`, with no argument, returns an empirical distribution function $\widehat{F}(x) = (n+1)^{-1} \sum_{i=1}^{n} 1_{X_i \leq x}$, for both `a.x` and `a.y`, evaluated at the observed datapoints (`sum` returns the sum). The `Copula` method `a.rhat()` will return an 'in development' message, and illustrates how code segments can be clearly reserved for future development (perhaps a general maximum-likelihood estimation procedure for multiple copula types). The `Copula` method `a.invert_cdf()` is again reserved for future development, and will return a *user-defined* error message, since this operation requires the estimated copula parameters, which have not yet been computed (and so `a` does not have a `simulate` attribute; this is tested with `hasattr`).[22]

[21] The dataset is available from the authors as `djia_sp500.csv`. The data was downloaded from `finance.yahoo.com` (under tickers ^DJI for Dow Jones and ^GSPC for S&P500). It is easy to iterate over the lines of a `.csv` file with `f=open('filename.csv','r')` and `for i in f:`, and it is not necessary to use the `csv` module for this. This example is not intended to represent a serious copula model (there is no dynamic aspect, for instance).

[22] Python has flexible built-in exception handling features, which we do not explore here (e.g. [2, Chapter 5]).

A `NormalCopula` instance can now be created by `b=NormalCopula(x,y)`. The derived class `NormalCopula(Copula)` inherits the methods of `Copula` (i.e. `__init__`, `empirical_cdf` as well as `invert_cdf`), replacing them by methods defined in the `NormalCopula` class if necessary (i.e. `rhat`), and adding any new methods (i.e. `simulate`). The first time that the method `b.rhat()` is called, it will compute the estimated copula parameters $\widehat{R} = n^{-1}\sum_{i=1}^{n} \upsilon_i \upsilon_i'$, where $\upsilon_i = (\Phi^{-1}(u_1^i), \Phi^{-1}(u_2^i))$ (`b.test`), Φ^{-1} is the inverse standard normal distribution (`norm.ppf` from `scipy.stats`), u_1 and u_2 are the empirical distributions of `b.x` and `b.y` respectively, and i indexes the observation (the NumPy matrix type `mat` is also used here, and has transpose method `.T`). The routine further corrects $\widehat{R}$ (`b.u`) by:

$$\frac{(\widehat{R})_{ij}}{\sqrt{(\widehat{R})_{ii}}\sqrt{(\widehat{R})_{jj}}} \mapsto (\widehat{R})_{ij},$$

and stores the result as `b.result`. All subsequent calls of `b.rhat()` will immediately return `b.result`, with no need for further computation.

For the stock index data, we find $\widehat{R}_{12} \approx 0.9473$ (the estimated copula correlation). Once $\widehat{R}$ has been estimated, it can be used for simulation with `b.simulate()` (which would have automatically called `b.rhat()` if this had not yet been done). This method computes the Cholesky decomposition $\widehat{R} = AA'$ (`cholesky`, from `scipy.linalg`, where the lower-triangular A is stored in `b.chol`), which is used to scale independent bivariate standard normal variates $x = (x_1, x_2)' = AN(0,1)^2 = N(0, \widehat{R})^2$, generated using the `rpy r.rnorm` function. Bivariate uniforms $u = (u_1, u_2)'$ are then computed by passing x through $\Phi(\cdot)$ (`norm.cdf`), where Φ is the univariate standard normal distribution. This gives $(u_1, u_2)' = (\widehat{F}_1(x_1), \widehat{F}_2(x_2))'$, where $\widehat{F}$ are the empirical marginals. We illustrate with 1000 simulations of $(u_1, u_2)'$ (see Figure 3).

In a full application, we may be interested in converting the uniform marginals back to the original scale. This requires numerical inversion of the empirical distribution functions, which could be tricky. In this example, the possibility is left open, and `b.invert_cdf()` will now return an 'in development' message, as required. We could imagine extending the code to include additional

classes, as the empirical study becomes deeper. For instance, `Copula`, `EllipticalCopula(Copula)` and `NormalCopula(EllipticalCopula)` and `StudentCopula(EllipticalCopula)`, where the base class `Copula` could contain general likelihood-based methods, in addition to computation of empirical or parametric marginals, and methods for graphical or descriptive data analysis; the derived class `EllipticalCopula` could contain computation of $\widehat{R}$, which is used by both the Normal 'closed-form' and the Student 'semi-closed form' estimation routines (but not by non-elliptical copulas, such as Archimedean copulas, which could have a separate class); and the derived classes `NormalCopula` and `StudentCopula` could use $\widehat{R}$ appropriately in estimation of the full set of copula parameters ($\widehat{R}$ directly for the normal copula; and $\widehat{R}$ and an estimated degrees-of-freedom parameter for the Student's copula), as well as containing specific simulation routines.

(Example 4. Rapid prototype of bivariate copula estimation using classes; comments in main text)

```
from scipy import *; from scipy.stats import *
from scipy.linalg import cholesky; from rpy import *
class Copula:
    def __init__(self,x,y):
        self.x=array(x); self.y=array(y); self.n=float(len(self.x))
    def empirical_cdf(self):
        return ([sum(self.x<=i)/(self.n+1) for i in self.x],
                [sum(self.y<=i)/(self.n+1) for i in self.y])
    def rhat(self):
        print 'Copula class rhat() not yet implemented.'; return
    def invert_cdf(self):
        if not hasattr(self,'simulate'): print 'Must estimate a copula first.'; return
        else: print 'Copula class invert_cdf() not yet implemented.'; return


class NormalCopula(Copula):
    def rhat(self):
        if not hasattr(self,'result'):
            self.test=(array(zip(norm.ppf(self.empirical_cdf())[0],
                      norm.ppf(self.empirical_cdf())[1])))
            self.u=array(mat(self.test).T*mat(self.test)/self.n)
            self.result=(array([[self.u[i][j]/(sqrt(self.u[i][i])*sqrt(self.u[j][j]))
                        for i in range(2)] for j in range(2)]))
        return self.result
    def simulate(self):
        if not hasattr(self,'result'): self.result=self.rhat()
        self.chol=cholesky(self.result,lower=1)
        return norm.cdf(array(mat(self.chol)*mat(r.rnorm(2,0,1)).T))
```

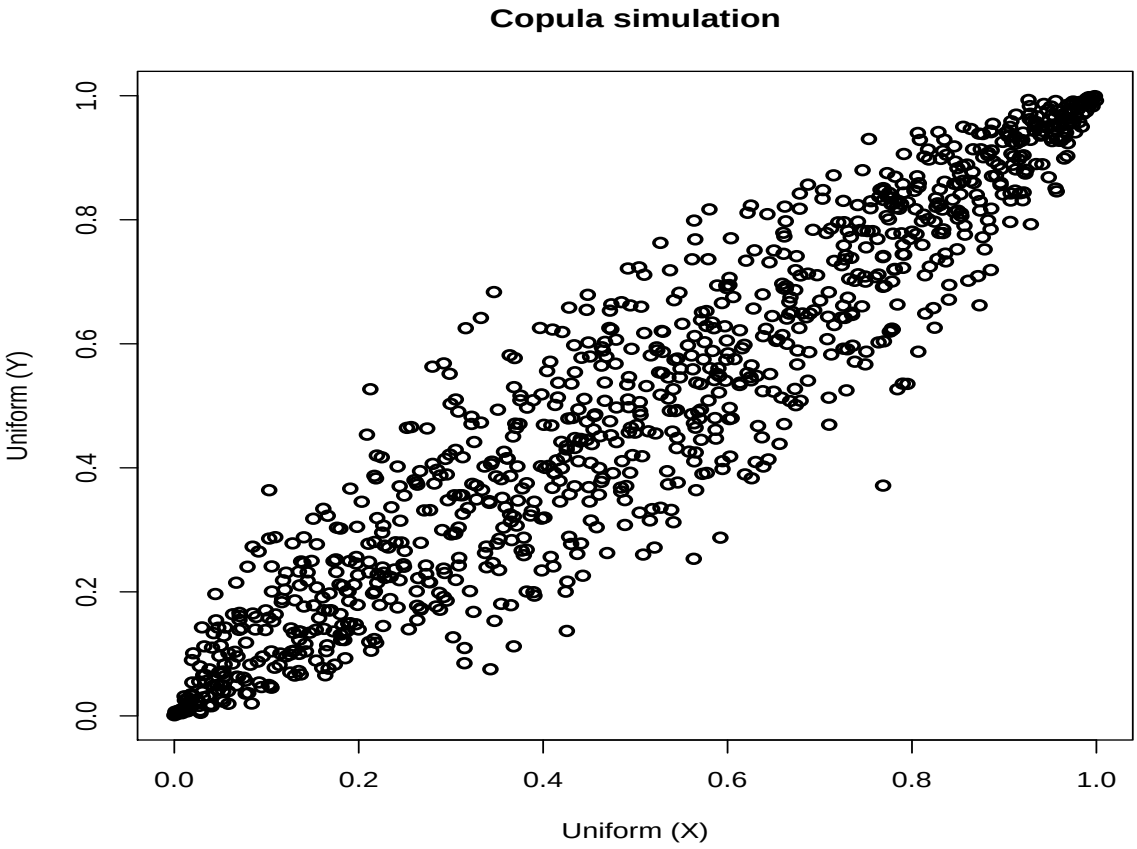

Figure 3: 1000 simulations from a simple bivariate copula model (see Example 4).

4.5 (Example 5.) Using C++ from Python for intensive computations

This example shows how to include C++ code directly within a Python program, using the `scipy.weave` module. We are motivated by the nested speed test result in Section 5, which shows that Python nested loops are quite inefficient compared to some software packages. Specifically, a 5000 $\times$ 5000 nested loop that only keeps track of the cumulative sum of the loop indexes runs in about 9 seconds in Python 2.5.4, compared to 5 seconds in Ox Professional 5.00, for instance (see Table 2). Generally, we would not advise use of Python nested loops for numerical

computations, and the problem worsens rapidly as the number of loops increases. However, it is easy to optimize Python programs by writing the heaviest computations in C++ (or Fortran). To illustrate, Example 5 solves the problem of the slow nested loop in the speed test. The C++ code that will perform the computation is simply included in the Python program as a raw string `r"""`*`string`*`"""`, exactly as it would be written in a C++ editor (but without the C++ preamble statements). The `scipy.weave.inline` module is called with the string that contains the C++ commands (`code`), the variables that are to be passed to and (although not in this example) from the C++ code (`dimension`), the method of variable type conversion (performed automatically using the `scipy.weave.converters.blitz` module), and optionally the compiler to be used (here, `gcc`, the GNU Compiler Collection). There will be a short compile time, when the Python program is first run, and we would expect some small overhead compared to the same code running directly in C++. However, we find over 1000 replications that the loop test now runs in a mean time of 0.02 seconds, or about 600 times faster than in Python! (roughly 300 times faster than Ox).

(Example 5. Including C++ code within a Python program)

```
from scipy.weave import inline,converters
dimension=5000
code=   r"""
        {
            long long kk=0;
            for (int i=0; i<dimension; i++) {
                for (int j=0; j<dimension; j++) {
                    kk+=i+j;
                }
            }
        }
        """
scipy.weave.inline(code,['dimension'],type_converters=scipy.weave.converters.blitz,compiler='gcc')
```

4.6 (Example 6.) Numerical optimization and Matlab-like plotting

This example illustrates the numerical optimization functionality of SciPy, and uses Matplotlib to create publication-quality graphics (see also [16] for an application). The code segment is included in Appendix A.3. We use the income and expenditure data that was formatted in Example 1, and analyzed in Examples 2 and 3.[23] The data dictionary is loaded using `cPickle`. Two SciPy arrays are created: y (100×1) contains the variable 'Acc', and X (100×6) contains a constant, 'Age', 'Income', 'MDR', 'Ownrent' and 'Selfempl' ('Avgexp' is dropped, since it perfectly predicts 'Acc').

We estimate a probit model $\text{Prob}(y_i = 1) = \Phi(x_i'\beta) + u_i,\ i = 1, 2, \ldots, 100$, where x_i' is the ith row of X, and $\beta = (\beta_0, \ldots, \beta_5)'$. Two user-defined functions specify the negative log-likelihood

$$-\ln \mathcal{L}(\beta) = -\sum_{i=1}^{100} \{y_i \ln \Phi(x_i'\beta) + (1 - y_i) \ln(1 - \Phi(x_i'\beta))\}$$

and the gradient function

$$\frac{\partial(-\ln \mathcal{L}(\beta))}{\partial \beta} = -\sum_{i=1}^{100} \left(\frac{\phi(x_i'\beta)(y_i - \Phi(x_i'\beta))}{\Phi(x_i'\beta)(1 - \Phi(x_i'\beta))} \right) x_i,$$

where $\phi()$ is the density function of the standard normal (`scipy.stats.norm`, `scipy.log`, and `numpy.dot` are used in the expressions). The unconstrained optimization $\widehat{\beta} = \arg\min(-\ln \mathcal{L}(\beta))$ is solved using the SciPy Newton-conjugate-gradient (`scipy.optimize.fmin_ncg`) method, with the least squares estimate of β used as starting value (`scipy.linalg.inv` is used in the calculation), and making use of the analytical gradient. The method converges rapidly, and the accuracy of the

[23] 'Acc' is a dummy variable taking value 1 if a credit card application is accepted, and 0 otherwise. 'Age' is age in years. 'Avgexp' is average monthly credit card expenditure. 'Income' is scaled (/10000) income. 'MDR' is the number of derogatory reports. 'Ownrent' is a dummy taking value 1 if the individual owns his home, and 0 if he rents. 'Selfempl' is a dummy taking value 1 if the individual is self-employed, and 0 otherwise.

maximum-likelihood estimate $\widehat{\beta}$ was checked using EViews 6.0 (which uses different start values).[24]

It could be useful in a teaching environment to explain the estimation procedure in more detail. Here, we use Matplotlib to design a figure for this purpose (Figure 4). We create a contour plot of $-\ln \mathcal{L}(\beta)$ in the (β_1, β_2)-space, using `numpy.meshgrid`, as well as `matplotlib.pyplot`. The plot settings can all be user-defined (e.g. line widths, colours, axis limits, labels, grid-lines, contour labels). We use LaTeX commands directly within Matplotlib to give mathematical axis labels, add a text box with information on the optimization, and annotate the figure with the position of the least squares starting values (in (β_1, β_2)-space), and the maximum-likelihood estimate. Matplotlib creates a `.png` graphic, which can be saved in various formats (here, as a `.pdf` file).

5 Speed comparisons

In this section, we compare the speed performance of extended Python 2.6.5 with Gauss 8.0, Mathematica 6.0, Ox Professional 5.00, R 2.11.1, and Scilab 5.1.1. We run 15 mathematical benchmark tests on a 1.66MHz Centrino Duo machine with 1GB RAM running Windows XP. The algorithms are adapted from [30, Section 8], and are described in Appendix A.2. They include a series of general mathematical, statistical and linear algebra operations, that occur frequently in applied work, as well as routines for nested loops and data import and analysis. The tests are generally performed on large dimension random vectors or matrices, which are implemented as SciPy arrays.[25] We summarize the tests, and report the extended Python functions that are used:

[24] Other numerical optimization routines are available. For instance, BFGS, with numerical or analytical gradient, is available from `scipy.optimize`, as `fmin_bfgs(f,x0,fprime=fp)`, where `f` is the function to be minimized, `x0` is the starting value, and `fp` is the derivative function (if `fprime=None`, a numerical derivative is used instead). Optional arguments control step-size, tolerance, display and execution parameters. Other optimization routines include a Nelder-Mead simplex algorithm (`fmin`), a modified level set method due to Powell (`fmin_powell`), a Polak-Ribière conjugate gradient algorithm (`fmin_cg`), constrained optimizers, and global optimizers including simulated annealing.

[25] All of the Python speed tests discussed in Section 5 and Appendix A.2 that require pseudo-random uniform numbers (13 of the 15 tests) use the well-known Mersenne Twister (MT19937). See Section 10.3 in [21] and Section 9.6 in [31] for details. Python supports random number generation from many discrete and continuous distributions. For instance, the continuous generators include the beta, Cauchy, χ^2, exponential, Fisher's F, gamma, Gumbel, Laplace, logistic, lognormal, noncentral χ^2, normal, Pareto, Student's t, and Weibull.

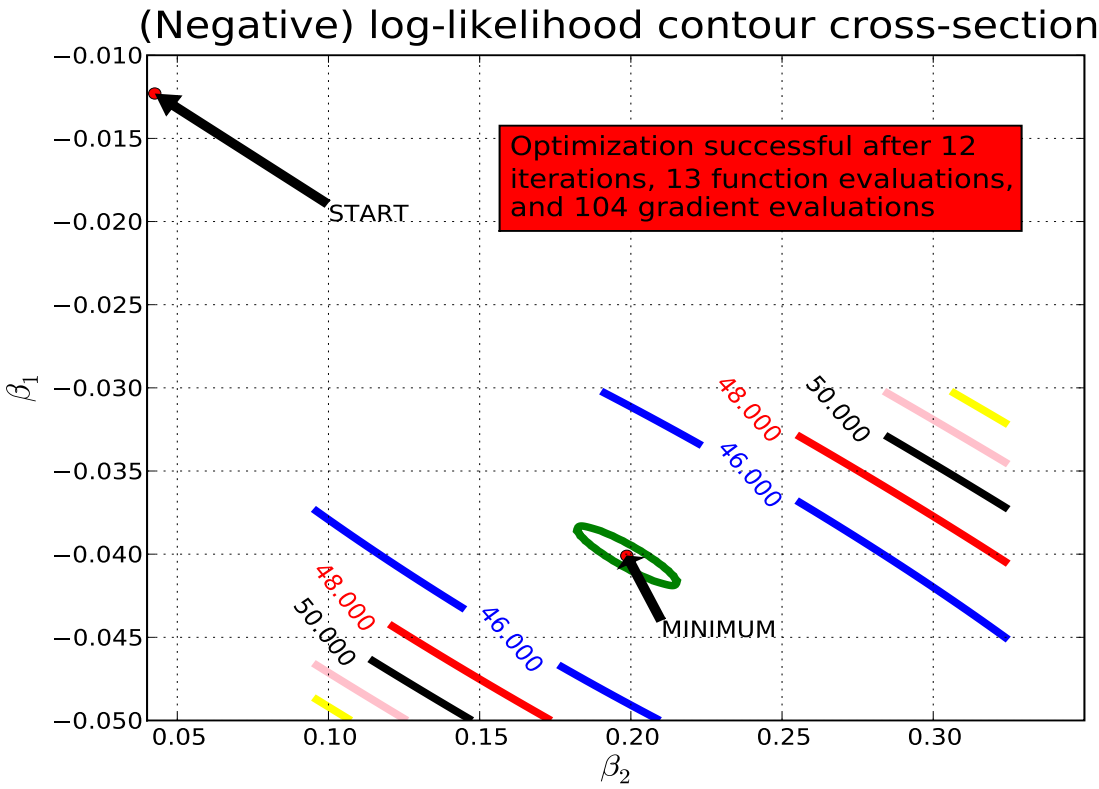

Figure 4: Annotated contour plot from probit estimation (see Example 6).

- Fast Fourier Transform over vector (`scipy.fftpack.fft()`).
- Linear solve of $Xw = y$ for w (`scipy.linalg.solve()`).
- Vector numerical sort (`scipy.sort()`).
- Gaussian error function over matrix (`scipy.special.erf()`).
- Random Fibonacci numbers using closed-form (uses SciPy array).
- Cholesky decomposition (`scipy.linalg.cholesky()`).

- Data import and descriptive statistics (uses `numpy.mat()` and `numpy.loadtxt()`).
- Gamma function over matrix (`scipy.special.gamma()`).
- Matrix element-wise exponentiation (`**`).
- Matrix determinant (`scipy.linalg.det()`).
- Matrix dot product (`numpy.dot()`).
- Matrix inverse (`scipy.linalg.inv()`).
- Two nested loops (core Python loops; fast Python/C++ routine implemented).
- Principal components analysis (`mdp.pca()`).
- Computation of eigenvalues (`scipy.linalg.eigvals()`).

For each software environment, the 15 tests were run over 10 (sometimes 5) replications. The code for the benchmark tests, implemented in Gauss, Mathematica, Ox, R, and Scilab, and the dataset that is required for the data import test, are available from `www.scientificweb.com/ncrunch`. We have made minor modifications to the timing (using the `time` module) and dimensions of some of the tests, but have not attempted to further optimize the code developed by [30], although we have looked for the fastest Python and R implementations in each case. Our results cannot be directly compared to those in [30], or [15], who run a previous version of these speed tests.

Full results are reported in Table 2, which gives the mean time (in seconds) across all replications for each of the tests. The tests have been ordered by increasing run-time for the extended Python implementation. The 'overall performance' of each software is calculated following [30], as:

$$\left(n^{-1} \sum_i \frac{\min_j(t_{ij})}{t_{ij}} \right) \times 100\%,$$

where $i = 1, 2, \ldots, n$ are the tests, j is the software, and t_{ij} is the speed (seconds) of test i with software j. Higher overall performance values correspond to higher overall speed (maximum 100%).

We remark briefly on the results. Extended Python is comparable in 'overall performance' to the econometric and statistical programming environments Ox and Scilab. For the first 12 tests, the Python implementation is either the fastest, or close to this, and displays some substantial speed gains over GAUSS, Mathematica, and R. While the data test imports directly into a NumPy array, Python is also able to parse complicated and heterogeneous data structures (see Example 1 for a simple illustration). The loop test takes almost twice as long as in GAUSS and Ox, but is considerably faster than Mathematica, Scilab, and R. It is well-known that Python loops are inefficient, and most such computations can usually be made much faster either by using *vectorized algorithms* ([14, Section 4.2]), or by optimizing one of the loops (often the inner loop). We would certainly not suggest that Python nested loops be used for heavy numerical work. In Section 4, Example 5, we show that it is straightforward to write the loop test as a Python/C++ routine, and that this implementation runs about 600 times faster than core Python. Code optimization is generally advisable, and not just for Python (see, e.g. `www.scipy.org/PerformancePython`). The principal components routine is the fastest implementation. The speed of the eigenvalue computation is roughly comparable to Mathematica.

Given the limited number of replications, and the difficulty of suppressing background processes, the values in Table 2 are only indicative (and especially for the heavier tests, which can sometimes be observed to have mean run-times that increase in the number of replications), although we do not expect the *qualitative* results to change dramatically with increased replications. In any given application, Python is likely to be comparably fast to some purpose-built mathematical software, and any slow time-critical code components can always be optimized by using C++ or Fortran.

Test	Python	GAUSS	Mathematica	Ox	R	Scilab
Fast Fourier Transform over vector	0.2	2.2	0.2	0.2	0.6	0.7
Linear solve of $Xw = y$ for w	0.2	2.4	0.2	0.7	0.8	0.2
Vector numerical sort	0.2	0.9	0.5	0.2	0.4	0.3
Gaussian error function over matrix	0.3	0.9	3.6	0.1	1.0	0.3
Random Fibonacci numbers	0.3	0.4	2.3	0.3	0.6	0.5
Cholesky decomposition	0.4	1.6	0.3	0.6	1.3	0.2
Data import and statistics	0.4	0.2	0.5	0.3	0.8	0.3
Gamma function over matrix	0.5	0.7	3.3	0.2	0.7	0.2
Matrix element-wise exponentiation	0.5	0.7	0.2	0.2	0.8	0.6
Matrix determinant	0.7	7.3	0.5	3.4	2.1	0.4
Matrix dot product	1.4	8.9	1.0	1.7	7.8	1.0
Matrix inverse	2.0	7.3	1.9	6.4	9.0	1.4
Two nested loops*	8.1	4.3	84.7	4.8	58.0	295.9
Principal components analysis	11.1	359.0	141.7	n/a	55.9	88.3
Computation of eigenvalues	32.3	90.2	24.2	21.7	13.6	17.3
Overall performance	**67%**	**30%**	**53%**	**70%**	**29%**	**65%**

Table 2: Speed test results. The mean time (seconds) across 10 replications is reported to 1 decimal place, for each of the 15 tests detailed in Appendix A.2. The GAUSS and R nested loops and the GAUSS, R, and Scilab principal components tests were run over 5 replications. The Scilab principal components test code ([30]) uses a third-party routine. The tests were performed in 'Python' (extended Python 2.6.5), 'GAUSS' (GAUSS 8.0), 'Mathematica' (Mathematica 6.0), 'Ox' (Ox Professional 5.00), 'R' (R 2.11.1) and 'Scilab' (Scilab 5.1.1), on a 1.66MHz Centrino Duo machine with 1GB RAM running Windows XP. The fastest implementation of each individual test is highlighted. 'Overall performance' is calculated as in [30]: $(n^{-1}\sum_i \min_j(t_{ij})/t_{ij}) \times 100\%$, where $i = 1, 2, \ldots, n$ are the tests, j is the software used, and t_{ij} is the speed (seconds) of test i with software j. The speed test codes are `python_speed_tests.py`, `benchga5.e`, `benchmath5.nb`, `benchox5.ox`, `r_speed_tests.r`, and `benchsc5.sce`, and are available from the authors (code for the Python and R tests was written by the authors). There is no principal components test in the [30] Ox code, and that result is not considered in the overall performance for Ox. *For a much faster (Python/C++) implementation of the Python nested loop test, see Section 4, Example 5, and the discussion in Section 5.

6 Concluding remarks

Knowledge of computer programming is indispensable for much applied and theoretical research. Although Python is now used in other scientific fields (e.g. physics), and as a teaching tool, it is much less well-known to econometricians and statisticians (exceptions are [5], which briefly introduces Python, and contains some nice examples; and [1]). We have tried to motivate Python as a powerful alternative for advanced econometric and statistical project work, but in particular as a means of linking different environments used in applied work.

Python is easy to learn, use, and extend, and has a large standard library and extensive third-party modules. The language has a supportive community, and excellent tutorials, resources, and references. The 'pythonic' language structure leads to readable (and manageable) programs, fast development times, and facilitates reproducible research. We agree with [13] that reproducibility is important (they also note the desirability of a single environment that can be used to manage multiple parts of a research project; consider also the survey paper [22]). Extended Python offers the possibility of direct programming of large-scale applications or, for users with high-performance software written in other languages, it can be useful as a strong 'glue' between different applications.

We have used the following packages here: (1) `cPickle` for data import and export, (2) `matplotlib` (`pyplot`) for graphics, (3) `mdp` for principal components analysis, (4) `numpy` for efficient array operations, (5) `rpy` for graphics and random number generation, (6) `scipy` for scientific computation (especially `arange` for array sequences, `constants` for built-in mathematical constants, `fftpack` for Fast Fourier transforms, `linalg` for matrix operations, `math` for standard mathematical functions, `optimize` for numerical optimization, `special` for special functions, `stats` for statistical functions, and `weave` for linking C++ and Python), and (7) `time` for timing the speed tests.

Many other Python modules can be useful in econometrics (see [2] and [31] for coverage of standard modules). These include `csv` (`.csv` file import and export), `os` (common operating-system tools), `random` (pseudo-random number generation), `sets` (set-theoretic operations), `sys`

(interpreter tools), `Tkinter` (see `wiki.python.org/moin/TkInter`; for building application front-ends), `urllib` and `urllib2` (for internet support, e.g. automated parsing of data from websites and creation of internet bots and web spiders), and `zipfile` (for manipulation of `.zip` compressed data files). For third-party modules, see [14] (and also `wiki.python.org/moin/NumericAndScientific`). Also useful are the 'Sage' mathematics system (`www.sagemath.org`), the `statsmodels` Python statistics package (`statsmodels.sourceforge.net`), and the 'SciPy Stats Project', a blog that developed out of the 'Google Summer of Code 2009' (`www.scipystats.blogspot.com`).

Of additional interest are the Psyco just-in-time compiler, which is reported to give substantial speed gains in some applications (see [14, Section 8], and [28] for a manual), and ScientificPython (not to be confused with SciPy), which provides further open source scientific tools for Python (see `dirac.cnrs-orleans.fr/plone/software/scientificpython`). The `f2py` module can be used to link Fortran and Python (`cens.ioc.ee/projects/f2py2e`), and the SWIG (Simplified Wrapper and Interface Generator) interface compiler (`www.swig.org`) provides advanced linking for C++ and Python. The 'Cython' language can be used to write C extensions for Python (`www.cython.org`). For completeness, we note that some commercial Python libraries are also available, e.g. the PyIMSL wrapper (to the IMSL C Numeric library). Python is also appropriate for network applications, animations, and application front-end management (e.g. it can be linked to Excel with the `xlwt` module, available from `pypi.python.org/pypi/xlwt`).

Parallel processing is possible in Python, with the `multiprocessing` package. Python is implemented so that only one simple thread can interact with the interpreter at a time (the Global Interpreter Lock: GIL). However, NumPy can release the GIL, leading to significant speed gains when arrays are used. Unlike threads, full processes each have their own GIL, and do not interfere with one another. In general, achieving optimal use of a multiprocessor machine or cluster is non-trivial. However, Python tools are also available for sophisticated parallelization.

We hope that this paper will encourage the reader to join the Python community!

Acknowledgements R.B. and S.L. thank the editor, Esfandiar Maasoumi, and two anonymous referees, for comments that improved the paper; and are grateful to Christophe Bontemps, Christine Choirat, David Joyner, Marko Loparic, Sébastien de Menten, Marius Ooms, Skipper Seabold and Michalis Stamatogiannis for helpful suggestions, Mehrdad Farzinpour for providing access to a Mac OS X machine, and John Muckstadt and Eric Johnson for providing access to a DualCore machine. R.B. thanks ENAC and Farid Zizi for providing partial research funding. R.B. was affiliated to ENAC when the first draft of this paper was completed. S.L. thanks Nathalie Lenoir for supporting this project, and Sébastien de Menten, who had the foresight to promote Python at Electrabel (which involved S.L. in the first place). This paper was typed by the authors in MiKTEX 2.8 and WinEdt 5, and numerical results were derived using extended Python 2.6.5 (described in Section 2), as well as C++, EViews 6.0, Gauss 8.0, Mathematica 6.0, Ox Professional 5.00, R 2.9.1 and R 2.11.1, and Scilab 5.1.1. The results in Table 2 depend upon our machine configuration, the number of replications, and our modification of the original [30] benchmark routines. They are not intended to be a definitive statement on the speed of the other software, most of which we have used productively at some time in our work. The authors are solely responsible for any views made in this paper, and for any errors that remain. All extended Python, C++, and R code for the examples and speed tests was written by the authors. Code and data are available on request.

References

[1] Almiron, M., Almeida, E., and Miranda, M. The reliability of statistical functions in four software packages freely used in numerical computation. *Brazilian Journal of Probability and Statistics 23* (2009), 107–119.

[2] Beazley, D. *Python Essential Reference*, 2nd ed. New Riders, 2001.

[3] Borcherds, P. Python: a language for computational physics. *Computer Physics Communications 177* (2007), 199–201.

[4] Bröker, O., Chinellato, O., and Geus, R. Using Python for large scale linear algebra applications. *Future Generation Computer Systems 21* (2005), 969–979.

[5] Choirat, C., and Seri, R. Econometrics with Python. *Journal of Applied Econometrics 24* (2009), 698–704.

[6] Cribari-Neto, F., and Zarkos, S. R: Yet another econometric programming environment. *Journal of Applied Econometrics 14* (1999), 319–329.

[7] Dale, D., Droettboom, M., Firing, E., and Hunter, J. Matplotlib Release 0.99.3. `matplotlib.sf.net/Matplotlib.pdf`, 2010.

[8] Downey, A. Think Python: How to think like a computer scientist - Version 1.1.22. `www.greenteapress.com/thinkpython/thinkpython.pdf`, 2008.

[9] Elkies, N. On $A^4 + B^4 + C^4 = D^4$. *Mathematics of Computation 51* (1988), 825–835.

[10] Greene, W. *Econometric Analysis*, 6th ed. Prentice-Hall, 2008.

[11] Hetland, M. *Beginning Python: From Novice to Professional.* Apress, 2005.

[12] Kleiber, C., and Zeileis, A. *Applied Econometrics with R.* Springer, 2008.

[13] Koenker, R., and Zeileis, A. On reproducible econometric research. *Journal of Applied Econometrics 24* (2009), 833–847.

[14] Langtangen, H. *Python Scripting for Computational Science*, 2nd ed. Springer, 2005.

[15] Laurent, S., and Urbain, J.-P. Bridging the gap between Gauss and Ox using OXGAUSS. *Journal of Applied Econometrics 20* (2005), 131–139.

[16] Lucks, J. Python - All a Scientist Needs. `arxiv.org/pdf/0803.1838`, 2008.

[17] Meinke, J., Mohanty, S., Eisenmenger, F., and Hansmann, U. SMMP v. 3.0 – Simulating proteins and protein interactions in Python and Fortran. *Computer Physics Communications 178* (2008), 459–470.

[18] Nelsen, R. *An Introduction to Copulas*, 2nd ed. Springer, 2006.

[19] Nilsen, J. MontePython: Implementing Quantum Monte Carlo using Python. *Computer Physics Communications 177* (2007), 799–814.

[20] Nilsen, J. Python in scientific computing: Applications to Bose-Einstein condensates. *Computer Physics Communications 177* (2007), 45.

[21] Oliphant, T. Guide to NumPy. `www.tramy.us/numpybook.pdf`, 2006.

[22] Ooms, M. Trends in applied econometrics software development 1985-2008. In *Palgrave Handbook of Econometrics*, T. Mills and K. Patterson, Eds., vol. 2. Palgrave MacMillan, 2009.

[23] Ooms, M., and Doornik, J. Econometric software development: past, present and future. *Statistica Neerlandica 60* (2006), 206–224.

[24] Patton, A. Copula-based models for financial time series. In *Handbook of Financial Time Series*, T. Andersen, R. Davis, J.-P. Kreiß, and T. Mikosch, Eds. Springer, 2009.

[25] Pilgrim, M. Dive into Python. www.diveintopython.org/download/diveintopython-pdf-5.4.zip, 2004.

[26] Racine, J., and Hyndman, R. Using R to teach econometrics. *Journal of Applied Econometrics 17* (2002), 175–189.

[27] Renfro, C. A compendium of existing econometric software packages. *Journal of Economic and Social Measurement 29* (2004), 359–409.

[28] Rigo, A. The Ultimate Psyco Guide - Release 1.6. psyco.sourceforge.net/psycoguide.ps.gz, 2007.

[29] SciPy community. SciPy Reference Guide - Release 0.9.0.dev6597. docs.scipy.org/doc/scipy/scipy-ref.pdf, 2010.

[30] Steinhaus, S. Comparison of mathematical programs for data analysis (Edition 5.04). www.scientificweb.com/ncrunch/ncrunch5.pdf, 2008.

[31] van Rossum, G. The Python Library Reference: Release 2.6.2. docs.python.org/archives/python-2.7-docs-pdf-a4.zip (ed. by F.J. Drake, Jr.), 2010.

[32] van Rossum, G. Python Tutorial: Release 2.6.2. docs.python.org/archives/python-2.7-docs-pdf-a4.zip (ed. by F.J. Drake, Jr.), 2010.

[33] van Rossum, G. Using Python: Release 2.6.2. docs.python.org/archives/python-2.7-docs-pdf-a4.zip (ed. by F.J. Drake, Jr.), 2010.

[34] Zeileis, A., and Koenker, R. Econometrics in R: Past, present, and future. *Journal of Statistical Software 27*, 1 (2008).

[35] Zito, T., Wilbert, N., Wiskott, L., and Berkes, P. Modular toolkit for Data Processing (MDP): a Python data processing framework. *Frontiers in Neuroinformatics 2* (2009). # 8.

A Appendix

A.1 A short introduction to bivariate copulas

Excellent references to copula theory and applications include [18] and [24]. Let X and Y be random variables such that

$$X \sim F, \quad Y \sim G, \quad (X, Y) \sim H,$$

where F and G are marginal distribution functions, and H is a joint distribution function. *Sklar's Theorem* states that

$$H(x, y) = C_\Theta(F(x), G(y)),$$

where $C_\Theta(u, v)$ is (in this paper) a parametric *copula function* that maps $C(u, v) : [0, 1]^2 \mapsto [0, 1]$, and that describes the dependence between $u := F(x)$ and $v := G(y)$, and 'binds' the marginals F and G together, to give a valid joint distribution H; and Θ is a set of parameters that characterize the copula. The probability integral transform $X \sim F \implies F(x) \sim U[0, 1]$, where U is a uniform distribution, implies that the copula arguments u and v are uniform.

Elliptical copulas (notably Normal and Student's t) are derived from elliptical distributions. They model symmetric dependence, and are relatively easy to estimate and simulate. Copulas are generally estimated using maximum likelihood, although for elliptical copulas some of the parameters in Θ can often be estimated using efficient (semi-)closed form formulae.

Here, we assume that the marginals are known, so that

$$H(x, y) = C_\Theta(\widehat{F}(x), \widehat{G}(y)),$$

and $\widehat{F}(x)$ and $\widehat{G}(x)$ are empirical marginal distributions, e.g. $\widehat{F}(x) := (n+1)^{-1} \sum_{i=1}^{n} 1_{X_i \leq x}$, where X_i is an observed datapoint, $1.$ is the indicator function, and n is the sample size.

The 2−dimensional Normal copula is given by:

$$C_R(u) = \Phi_R(\Phi^{-1}(u_1), \Phi^{-1}(u_2)) = \int_{-\infty}^{\Phi^{-1}(u_1)} \int_{-\infty}^{\Phi^{-1}(u_2)} (2\pi)^{-1}|R|^{-1/2} \exp(-x'R^{-1}x/2)dx_1dx_2,$$

where $\Theta := R$ is a 2×2 correlation matrix, $u_j = \widehat{F}_j(x_j)$, Φ_R is a 2-dimensional Normal distribution, and Φ^{-1} (Φ) is the univariate (inverse) standard Normal distribution. We further define a bivariate copula density $\partial^2 C_\Theta(u)/\partial u_1 \partial u_2$, which for the Normal gives $c_R(\upsilon) := |R|^{-1/2} \exp(-\upsilon'(R^{-1} - I_2)\upsilon/2)$, where $\upsilon = (\Phi^{-1}(u_1), \Phi^{-1}(u_2))'$. Maximum-likelihood estimation of R solves

$$\widehat{R} = \arg\max_R \; n^{-1} \sum_{i=1}^{n} \ln c_R(u).$$

Numerical optimization of the log-likelihood surface is usually slow and can lead to numerical errors. Fortunately, there is a closed-form for the Normal $\widehat{R}$:

$$\widehat{R} = n^{-1} \sum_{i=1}^{n} \upsilon_i \upsilon_i',$$

where $\upsilon_i = (\Phi^{-1}(u_1^i), \Phi^{-1}(u_2^i))'$. Due to numerical errors, we correct to a valid correlation matrix:

$$\frac{(\widehat{R})_{ij}}{\sqrt{(\widehat{R})_{ii}}\sqrt{(\widehat{R})_{jj}}} \mapsto (\widehat{R})_{ij}.$$

Once we have estimated the copula parameters we have, in the bivariate case, $H(x, y) = C_{\widehat{\Theta}}(\widehat{F}(x), \widehat{G}(y)) := C_{\widehat{\Theta}}(u, v)$. Simulation from a copula involves generation of (u_r, v_r), which can subsequently be transformed to the original data scale (not considered here). For the bivariate Normal, we simulate by: (1) Compute the Cholesky decomposition $\widehat{R} = AA'$, (2) Simulate a 2-vector of standard Normals $z \sim N(0, 1)^2$, (3) Scale the standard Normals: $x = Az \sim N(0, \widehat{R})^2$, (4) Simulate the uniforms u_j by $u_j = \Phi(x_j), j = 1, 2$: this gives $(u_1, u_2) = (\widehat{F}_1(x_1), \widehat{F}_2(x_2))$.

A.2 Speed test algorithms

We detail the algorithms that are used for the speed tests discussed in Section 5 (adapted from [30]), and reported in Table 2. We assume that all required Python modules have been imported. Matrices and vectors have representative elements $X = (X_{rs})$ and $x = (x_r)$ respectively. Further, we let $I = \{1, 2, \ldots, 10\}$. We give brief details on the extended Python implementations.

- **Fast Fourier Transform over vector**

  [Replication i] Generate a $2^{20} \times 1$ random uniform vector $x = (U(0, 1))$. [Start timer] Compute the Fast Fourier Transform of x. [End timer] [End replication i] Repeat for $i \in I$. Random numbers are generated with `x=scipy.random.random(2**20)`. The Fast Fourier transform is performed using `scipy.fftpack.fft(x)`.

- **Linear solve of $Xw = y$ for w**

  [Replication i] Generate a 1000×1000 random uniform matrix $X = (U(0, 1))$. Generate a 1000×1 vector $y = (j), j = 1, 2, \ldots, 1000$. [Start timer] Solve $Xw = y$ for w. [End timer] [End replication i] Repeat for $i \in I$. Random numbers are generated using the command `x=scipy.random.random((1000,1000))`. The array sequence is `y=scipy.arange(1,1001)`. The linear solve is performed by the command `scipy.linalg.solve(x,y)`.

- **Vector numerical sort**

  [Replication i] Generate a 1000000×1 random uniform vector $x = (U(0, 1))$. [Start timer] Sort the elements of x in ascending order. [End timer] [End replication i] Repeat for $i \in I$. Random numbers are generated using `x=scipy.random.random(1000000)`. The array sort method is `scipy.sort(x)`.

- **Gaussian error function over matrix**

  [Replication i] Generate a 1500×1500 random uniform matrix $X = (U(0, 1))$. [Start timer] Compute the Gaussian error function $\mathrm{erf}(x)$. [End timer] [End replication i] Repeat for $i \in I$.

The random array is generated with `x=scipy.random.random((1500,1500))`. The error function implementation is `scipy.special.erf(x)`.

- **Random Fibonacci numbers**

  [Replication i] Generate a 1000000×1 random uniform vector $x = (\lfloor 1000 \times U(0,1) \rfloor)$, where $\lfloor \cdot \rfloor$ returns the integer part. The *Fibonacci numbers* y_n are defined by the recurrence relation $y_n = y_{n-1} + y_{n-2}$, where $y_0 = 0$ and $y_1 = 1$ initialize the sequence. [Start timer] Compute the Fibonacci numbers y_n for $x = (n)$ (i.e. $n \in \{0, 1, \ldots, 999\}$ will give 1 million random drawings from the first 1000 Fibonacci numbers), using the closed-form *Binet formula*

  $$y_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}},$$

  where $\phi = (1 + \sqrt{5})/2$ is the *golden ratio*.[26] [End timer] [End replication i] Repeat for $i \in I$. The random vector is `x=scipy.floor(1000*scipy.random.random((1000000,1)))`. The Binet formula is implemented as `((golden**x)-(-golden)**(-x))/scipy.sqrt(5)`, where `golden` is taken from `scipy.constants`.

- **Cholesky decomposition**

  [Replication i] Generate a 1500×1500 random uniform matrix $X = (U(0,1))$. Compute the dot product $X'X$. [Start timer] Compute the upper-triangular Cholesky decomposition $X'X = U'U$, i.e. solve for square U. [End timer] [End replication i] Repeat for $i \in I$. The random array is `x=scipy.random.random((1500,1500))`. The dot product is computed by `y=numpy.dot(x.T,x)`. The Cholesky decomposition computation is performed by the command `scipy.linalg.cholesky(y,lower=False)`.

[26] A faster closed-form formula is $y_n = \lfloor (\phi^n/\sqrt{5}) + (1/2) \rfloor$, although we do not use it here. For extended Python, the faster formula takes roughly 0.2 seconds across 10 replications. We also correct for a typo in the [30] Mathematica code: `RandomInteger[{100,1000},...]` is replaced by `RandomInteger[{0,999},...]`.

- **Data import and statistics**

  [Replication i] [Start timer] Import the datafile `Currency2.txt`. This contains 3160 rows and 38 columns of data, on 34 daily exchange rates over the period 2 January 1990 to 8 February 2002. There is a single header line. For each currency-year, compute the mean, minimum and maximum of the data, and the percentage change over the year.[27] [End timer] [End replication i] Repeat for $i \in I$. We import the data with `numpy.mat(numpy.loadtxt("filename"))`.

- **Gamma function over matrix**

  [Replication i] Generate a 1500×1500 random uniform matrix $X = (U(0,1))$. [Start timer] Compute the gamma function $\Gamma(x)$. [End timer] [End replication i] Repeat for $i \in I$. The random array is `x=scipy.random.random((1500,1500))`. The gamma function command is `scipy.special.gamma(x)`.[28]

- **Matrix element-wise exponentiation**

  [Replication i] Generate a 2000×2000 random uniform matrix $X = (U(1,1.01)) := (X_{rs})$. [Start timer] Compute the matrix $Y = (X_{rs}^{1000})$. [End timer] [End replication i] Repeat for $i \in I$. The random array is `x=1+(scipy.random.random((2000,2000))/100)`. The exponentiation is `x**1000`.

- **Matrix determinant**

  [Replication i] Generate a 1500×1500 random uniform matrix $X = (U(0,1))$. [Start timer] Compute the determinant $\det(X)$. [End timer] [End replication i] Repeat for $i \in I$. The random array is generated by `x=scipy.random.random((1500,1500))`. The determinant is computed by `scipy.linalg.det(x)`.

[27] The file `Currency2.txt` is available from `www.scientificweb.com/ncrunch`. We modify the [30] GAUSS code to import data into a matrix using `load data[]=^"filename";` and `datmat=reshape(data,3159,38);`. For the GAUSS and Python implementations, we first removed the header line from the data file, before importing the data.

[28] We correct for a typo in the [30] Mathematica code: we use `RandomReal[{0,1},{1000,1000}]` instead of `RandomReal[NormalDistribution[],{1000,1000}]`.

- **Matrix dot product**

  [Replication i] Generate a 1500×1500 random uniform matrix $X = (U(0, 1))$. [Start timer] Compute the dot product $X'X$. [End timer] [End replication i] Repeat for $i \in I$. The random array is `x=scipy.random.random((1500,1500))`. The dot product is computed by `numpy.dot(x.T,x)`.

- **Matrix inverse**

  [Replication i] Generate a 1500×1500 random uniform matrix $X = (U(0, 1))$. [Start timer] Compute the inverse X^{-1}. [End timer] [End replication i] Repeat for $i \in I$. The random array is `x=scipy.random.random((1500,1500))`. The inverse is `scipy.linalg.inv(x)`.

- **Two nested loops**

  [Replication i] Set $a = 1$. [Start timer] (Loop across $l = 1, 2, \ldots, 5000$). (Loop across $m = 1, 2, \ldots, 5000$). Set $a = a + l + m$. (Close inner loop). (Close outer loop). [End timer] [End replication i] Repeat for $i \in I$. Routine written in core Python. See Section 4 (Example 5) and Section 5 for further discussion.

- **Principal components analysis**

  [Replication i] Generate a 10000×1000 random uniform matrix $X = (U(0, 1))$. [Start timer] Transform X into principal components using the covariance method. [End timer] [End replication i] Repeat for $i \in I$. The random array is `x=scipy.random.random((10000,1000))`. The principal components are computed using `mdp.pca(x)`.

- **Computation of eigenvalues**

  [Replication i] Generate a 1200×1200 random uniform matrix $X = (U(0, 1))$. [Start timer] Compute the eigenvalues of X. [End timer] [End replication i] Repeat for $i \in I$. The random array is `x=scipy.random.random((1200,1200))`. The eigenvalues are computed using the command `scipy.linalg.eigvals(x)`.

A.3 Python code segment for probit example

(Example 6(i). Numerical optimization and Matlab-like plotting)

```
import cPickle
from matplotlib import pyplot
from scipy import array,shape,arange,log,ones,random
from scipy.stats import norm
from scipy.optimize import fmin_ncg
from scipy.linalg import inv
from numpy import dot,meshgrid

f=open('./python_data.bin','r'); data_dict=cPickle.load(f); f.close()
y=array(data_dict['Acc']); vars=data_dict.keys(); vars.sort()
x=array([[1]*len(y)]+[data_dict[r] for r in vars if r!='Acc' and r!='Avgexp'])

def nll(beta,x,y):
    return (dot(-((y*log(norm.cdf(dot(x.T,beta))))+
                  ((1-y)*log(1-norm.cdf(dot(x.T,beta))))),ones(len(y))))

def nllprime(beta,x,y):
    return (-dot(x,(norm.pdf(dot(x.T,beta))*(y-norm.cdf(dot(x.T,beta))))/
                 (norm.cdf(dot(x.T,beta))*(1-norm.cdf(dot(x.T,beta))))))

beta_hat_ols=dot(inv(dot(x,x.T)),dot(x,y))

a=fmin_ncg(nll,beta_hat_ols,args=(x,y),fprime=nllprime,disp=1)

b1=arange(-0.05,-0.03,0.0003);b2=arange(0.095,0.325,0.0003)
X1,X2=meshgrid(b1,b2); Z=ones((len(b1),len(b2))); beta_test=array(list(a)[:])
```

(Example 6(ii). Numerical optimization and Matlab-like plotting)

```
for i in range(len(b1)):
    for j in range(len(b2)):
        beta_test[1]=b1[i]; beta_test[2]=b2[j]
        Z[i][j]=nll(beta_test,x,y)


pyplot.figure()
CS=pyplot.contour(b2,b1,Z,linewidths=(4,4,4,4,4,4),\
               colors=('green','blue','red','black','pink','yellow'))
pyplot.clabel(CS,inline=1,fontsize=12)
pyplot.xlabel(r'$\beta_{2}$',fontsize=16)
pyplot.ylabel(r'$\beta_{1}$',fontsize=16)
pyplot.title('(Negative) log-likelihood contour cross-section',fontsize=20)
pyplot.plot([0.19866354],[-0.04009722],'ro')
pyplot.plot([0.042544232],[-0.01230058],'ro')
pyplot.annotate('MINIMUM',xy=(0.19866354,-0.04009722),\
               xytext=(0.21,-0.045),arrowprops=dict(facecolor='black',\
               shrink=0.01))
pyplot.annotate('START',xy=(0.042544232,-0.01230058),\
               xytext=(0.10,-0.02),arrowprops=dict(facecolor='black',\
               shrink=0.01))
pyplot.text(0.16,-0.02,'Optimization successful after 12\niterations,\
           13 function evaluations,\nand 104 gradient evaluations',\
           bbox={'facecolor':'red','alpha':1,'pad':10},fontsize=14)
pyplot.ylim(-0.05,-0.01)
pyplot.xlim(0.04,0.35)
pyplot.grid(True)
pyplot.show()
```