# Python for research and teaching economics

David R. Pugh[‡*]

http://www.youtube.com/watch?v=xHkGW1l5X8k

**Abstract**—Together with theory and experimentation, computational modeling and simulation has become a "third pillar" of scientific inquiry. I am developing a curriculum for a three part, graduate level course on computational methods designed to increase the exposure of graduate students and researchers in the School of Economics at the University of Edinburgh to basic techniques used in computational modeling and simulation using the Python programming language. My course requires no prior knowledge or experience with computer programming or software development and all current and future course materials will be made freely available on-line via GitHub.

**Index Terms**—python, computational economics, dynamic economic models, numerical methods

## Introduction

Together with theory and experimentation, computational modeling and simulation has become a "third pillar" of scientific inquiry. In this paper, I discuss the goals, objectives, and pedagogical choices that I made in designing and teaching a Python-based course on computational modeling and simulation to first-year graduate students in the Scottish Graduate Programme in Economics (SGPE) at the University of Edinburgh. My course requires no prior knowledge or experience with computer programming or software development and all current and future course materials will be made freely available on-line.[1]

Like many first-year PhD students, I began my research career with great faith in the analytic methods that I learned as an undergraduate and graduate student. While I was aware that economic models without closed-form solutions did exist, at no time during my undergraduate or graduate studies was I presented with an example of an important economic result that could not be analytically derived. While these analytic results were often obtained by making seemingly restrictive assumptions, the manner in which these assumptions were often justified gives the impression that such assumptions did not substantially impact the economic content of the result. As such, I started work as a PhD student under the impression that most all "interesting" economic research questions could, and perhaps even should, be tackled analytically. Given that both of the leading graduate-level micro and macroeconomics textbooks, [mas-colell1995] and [romer2011], fail to mention that computational methods are needed to fully solve even basic economic models, I do not believe that I was alone in my ignorance of the import of these methods in economics.

Fortunately (or unfortunately?) I was rudely awakened to the reality of modern economics research during my first year as a PhD student. Most economic models, particularly dynamic economic models, exhibit essential non-linearities or binding constraints that render them analytically intractable. Faced with reality I was confronted with two options: give up my original PhD research agenda, which evidently required computational methods, in favor of a modified research program that I could pursue with analytic techniques; or teach myself the necessary numerical techniques to pursue my original research proposal. I ended up spending the better part of two (out of my alloted three!) years of my PhD teaching myself computational modeling and simulation methods. The fact that I spent two-thirds of my PhD learning the techniques necessary to pursue my original research agenda indicated, to me at least, that there was a substantial gap in the graduate economics training at the University of Edinburgh. In order to fill this gap, I decided to develop a three-part course on computational modeling and simulation.

The first part of my course is a suite of Python-based, interactive laboratory sessions designed to expose students to the basics of scientific programming in Python. The second part of the course is a week-long intensive computational methods "boot camp." The boot camp curriculum focuses on deepening students' computer programming skills using the Python programming language and teaching important software design principles that are crucial for generating high-quality, reproducible scientific research using computational methods. The final part of the course, which is very much under development, will be an advanced training course targeted at PhD students and will focus on applying more cutting edge computational science techniques to economic problems via a series of interactive lectures and tutorials.

## Why Python?

Python is a modern, object-oriented programming language widely used in academia and private industry, whose clean, yet expressive syntax, makes it an easy programming language to learn while still remaining powerful enough for serious scientific computing.[2] Python's syntax was designed from the start with the human reader in mind and generates code that is easy to understand and debug which shortens development time relative to

* *Corresponding author: pugh@maths.ox.ac.uk*
‡ *School of Economics, University of Edinburgh; Institute for New Economic Thinking at the Oxford Martin School and Oxford Mathematical Institute, University of Oxford*

*Copyright © 2014 David R. Pugh. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.*

1. This course would not have been possible without generous funding and support from the Scottish Graduate Programme in Economics (SGPE), the Scottish Institute for Research in Economics (SIRE), the School of Economics at the University of Edinburgh, and the Challenge Investment Fund (CIF).

low-level, compiled languages such as Fortran and C++. Among the high-level, general purpose languages, Python has the largest number of Matlab-style library modules (both installed in the standard library and through additional downloads) which meaning that one can quickly construct sophisticated scientific applications. While the Python programming language has found widespread use in private industry and many fields within academia, the capabilities of Python as a research tool remain relatively unknown within the economics research community. Notable exceptions are [stachurski2009] and [sargent2014].

Python is completely free and platform independent, making it a very attractive option as a teaching platform relative to other high-level scripting languages, particularly Matlab. Python is also open-source, making it equally attractive as a research tool for scientists interested in generating computational results that are more easily reproducible.[3] Finally, Python comes with a powerful interactive interpreter that allows real-time code development and live experimentation. The functionality of the basic Python interpreter can be greatly increased by using the Interactive Python (or IPython) interpreter. Working via the Python or IPython interpreter eliminates the time-consuming (and productivity-destroying) compilation step required when working with low-level languages at the expense of slower execution speed. In many cases, it may be possible to achieve the best of both worlds using "mixed language" programming as Python can be easily extended by wrapping compiled code written in Fortran, C, or C++ using libraries such as f2Py, Cython, or swig. See [oliphant2007], [peterson2009], [behnel2011], [van2011] and references therein for more details.

### Motivating the use of numerical methods in economics

The typical economics student enters graduate school with great faith in the analytical mathematical tools that he or she was taught as an undergraduate. In particular this student is under the impression that virtually all economic models have closed-form solutions. At worst the typical student believes that if he or she were to encounter an economic model without a close-form solution, then simplifying assumptions could be made that would render the model analytically tractable without sacrificing important economic content.

The typical economics student is, of course, wrong about general existence of closed-form solutions to economic models. In fact the opposite is true: most economic models, particular dynamic, non-linear models with meaningful constraints (i.e., most any *interesting* model) will fail to have an analytic solution. In order to demonstrate this fact and thereby motivate the use of numerical methods in economics, I begin my course with a laboratory session on the Solow model of economic growth [solow1956].

Economics graduate student are very familiar with the Solow growth model. For many students, the Solow model will have been one of the first macroeconomic models taught to them as undergraduates. Indeed, the dominant macroeconomics textbook for first and second year undergraduates, [mankiw2010], devotes two full chapters to motivating and deriving the Solow model. The first few chapters of [romer2011], one of the most widely used final year undergraduate and first-year graduate macroeconomics textbook, are also devoted to the Solow growth model and its descendants.

#### *The Solow growth model*

The Solow model boils down to a single non-linear differential equation and associated initial condition describing the time evolution of capital stock per effective worker, $k(t)$.

$$\dot{k}(t) = sf(k(t)) - (n+g+\delta)k(t),\ k(t) = k_0$$

The parameter $0 < s < 1$ is the fraction of output invested and the parameters $n, g, \delta$ are the rates of population growth, technological progress, and depreciation of physical capital. The intensive form of the production function $f$ is assumed to be to be strictly concave with

$$f(0) = 0,\ lim_{k\rightarrow 0}\ f' = \infty,\ lim_{k\rightarrow\infty}\ f' = 0.$$

A common choice for the function $f$ which satisfies the above conditions is known as the Cobb-Douglas production function.

$$f(k) = k^{\alpha}$$

Assuming a Cobb-Douglas functional form for $f$ also makes the model analytically tractable (and thus contributes to the typical economics student's belief that all such models "must" have an analytic solution). [sato1963] showed that the solution to the model under the assumption of Cobb-Douglas production is

$$k(t) = \Bigg[\left(\frac{s}{n+g+\delta}\right)\left(1-e^{-(n+g+\delta)(1-\alpha)t}\right)+ k_0^{1-\alpha}e^{-(n+g+\delta)(1-\alpha)t}\Bigg]^{\frac{1}{1-\alpha}}.$$

A notable property of the Solow model with Cobb-Douglas production is that the model predicts that the shares of real income going to capital and labor should be constant. Denoting capital's share of income as $\alpha_K(k)$, the model predicts that

$$\alpha_K(k) \equiv \frac{\partial \ln f(k)}{\partial \ln k} = \alpha$$

Unfortunately, from figure 1 it is clear that the prediction of constant factor shares is strongly at odds with the empirical data for most countries. Fortunately, there is a simple generalization of the Cobb-Douglas production function, known as the constant elasticity of substitution (CES) function, that is capable of generating the variable factor shares observed in the data.

$$f(k) = \Big[\alpha k^{\rho} + (1-\alpha)\Big]^{\frac{1}{\rho}}$$

where $-\infty < \rho < 1$ is the elasticity of substitution between capital and effective labor in production. Note that

$$\lim_{\rho\rightarrow 0} f(k) = k^{\alpha}$$

and thus the CES production function nests the Cobb-Douglas functional form as a special case. To see that the CES production function also generates variable factor shares note that

$$\alpha_K(k) \equiv \frac{\partial \ln f(k)}{\partial \ln k} = \frac{\alpha k^{\rho}}{\alpha k^{\rho} + (1-\alpha)}$$

2. A non-exhaustive list of organizations currently using Python for scientific research and teaching: MIT's legendary *Introduction to Computer Science and Programming*, CS 6.00, is taught using Python; Python is the in-house programming language at Google; NASA, CERN, Los Alamos National Labs (LANL), Lawrence Livermore National Labs (LLNL), and Industrial Light and Magic (ILM) all rely heavily on Python.

3. The Python Software Foundation License (PSFL) is a BSD-style license that allows a developer to sell, use, or distribute his Python-based application in anyway he sees fit. In addition, the source code for the entire Python scientific computing stack is available on GitHub making it possible to directly examine the code for any specific algorithm in order to better understand exactly how a result has been obtained.

which varies with $k$.

This seemingly simple generalization of the Cobb-Douglas production function, which is necessary in order for the Solow model generate variable factor share, an economically important feature of the post-war growth experience in most countries, renders the Solow model analytically intractable. To make progress solving a Solow growth model with CES production one needs to resort to computational methods.

### *Numerically solving the Solow model*

A computational solution to the Solow model allows me to demonstrate a number of numerical techniques that students will find generally useful in their own research.

First and foremost, solving the model requires efficiently and accurately approximating the solution to a non-linear ordinary differential equation (ODE) with a given initial condition (i.e., an non-linear initial value problem). Finite-difference methods are commonly employed to solve such problems. Typical input to such algorithms is the Jacobian matrix of partial derivatives of the system of ODEs. Solving the Solow growth model allows me to demonstrate the use of finite difference methods as well as how to compute Jacobian matrices of non-linear systems of ODEs.

Much of the empirical work based on the Solow model focuses on the model's predictions concerning the long-run or steady state equilibrium of the model. Solving for the steady state of the Solow growth model requires solving for the roots of a non-linear equation. Root finding problems, which are equivalent to solving systems of typically non-linear equations, are one of the most widely encountered computational problems in economic applications. Typical input to root-finding algorithms is the Jacobian matrix of partial derivatives of the system of non-linear equations. Solving for the steady state of the Solow growth model allows me to demonstrate the use of various root finding algorithms as well as how to compute Jacobian matrices of non-linear systems of equations.

Finally, given some data, estimation of the model's structural parameters (i.e., $g$, $n$, $s$, $\alpha$, $\delta$, $\rho$) using either as maximum likelihood or non-linear least squares requires solving a non-linear, constrained optimization problem. Typical inputs to algorithms for solving such non-linear programs are the Jacobian and Hessian of the objective function with respect to the parameters being estimated.[4] Thus structural estimation also allows me to demonstrate the symbolic and numerical differentiation techniques needed to compute the Jacobian and Hessian matrices.

## Course outline

Having motivated the need for computational methods in economics, in this section I outline the three major components of my computational methods course: laboratory sessions, an intensive week-long Python boot camp, and an advanced PhD training course. The first two components are already up and running (thanks to funding support from the SGPE, SIRE, and the CIF). I am still looking to secure funding to develop the advanced PhD training course component.

### *Laboratory sessions*

The first part of the course is a suite of Python-based laboratory sessions that run concurrently as part of the core macroeconomics sequence. There are 8 labs in total: two introductory sessions, three labs covering computational methods for solving models that students are taught in macroeconomics I (fall term), three labs covering computational methods for solving models taught in macroeconomics II (winter term). The overall objective of these laboratory sessions is to expose students to the basics of scientific computing using Python in a way that reinforces the economic models covered in the lectures. All of the laboratory sessions make use of the excellent IPython notebooks.

The material for the two introductory labs draws heavily from part I and part II of Quantitative Economics by Thomas Sargent and John Stachurski. In the first lab, I introduce and motivate the use of the Python programming language and cover the bare essentials of Python: data types, imports, file I/O, iteration, functions, comparisons and logical operators, conditional logic, and Python coding style. During the second lab, I attempt to provide a quick overview of the Python scientific computing stack (i.e., IPython, Matplotlib, NumPy, Pandas, and SymPy) with a particular focus on those pieces that students will encounter repeatedly in economic applications.

The material for the remaining 6 labs is designed to complement the core macroeconomic sequence of the SGPE and thus varies a bit from year to year. During the 2013-2014 academic year I covered the following material:

- Initial value problems: Using the [solow1956] model of economic growth as the motivating example, I demonstrate finite-difference methods for efficiently and accurately solving initial value problems of the type typically encountered in economics.
- Boundary value problems: Using the neo-classical optimal growth model of [ramsey1928], [cass1965], and [koopmans1965] as the motivating example, I demonstrate basic techniques for efficiently and accurately solving two-point boundary value problems of the type typically encountered in economics using finite-difference methods (specifically forward, reverse, and multiple shooting).
- Numerical dynamic programming: I demonstrate basic techniques for solving discrete-time, stochastic dynamic programming problems using a stochastic version of the neo-classical optimal growth model as the motivating example.
- Real business cycle models: I extend the stochastic optimal growth model to incorporate a household labor supply decision and demonstrate how to approximate the model solution using *dynare++*, a C++ library specializing in computing $k$-order Taylor approximations of dynamic stochastic general equilibrium (DSGE) models.

In future versions of the course I hope to include laboratory sessions on DSGE monetary policy models, DSGE models with financial frictions, and models of unemployment with search frictions. These additional labs are likely to be based around dissertations being written by current MSc students.

### *Python boot camp*

Whilst the laboratory sessions expose students to some of the basics of programming in Python as well as numerous applications of computational methods in economics, these lab sessions are inadequate preparation for those students wishing to apply such methods as part of their MSc dissertations or PhD theses.

4. The Hessian matrix is also used for computing standard errors of parameter estimates.

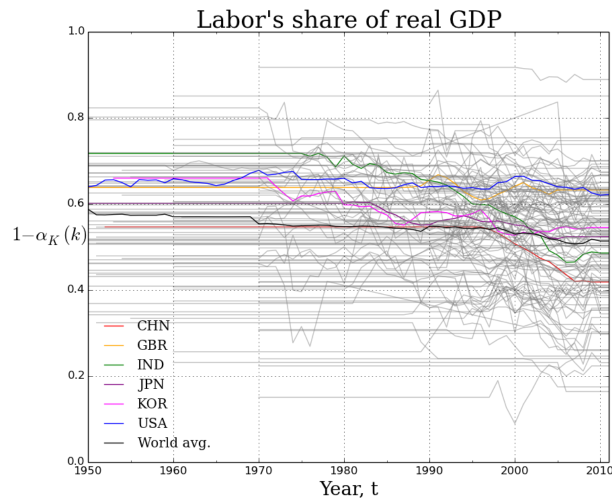


***Fig. 1:*** *Labor's share of real GDP has been declining, on average, for much of the post-war period. For many countries, such as India, China, and South Korea, the fall in labor's share has been dramatic.*

In order to provide interested students with the skills needed to apply computational methods in their own research I have developed a week-long intensive computational methods "boot camp." The boot camp requires no prior knowledge or experience with computer programming or software development and all current and future course materials are made freely available on-line.

Each day of the boot camp is split into morning and afternoon sessions. The morning sessions are designed to develop attendees Python programming skills while teaching important software design principles that are crucial for generating high-quality, reproducible scientific research using computational methods. The syllabus for the morning sessions closely follows Think Python by Allen Downey.

In teaching Python programming during the boot camp I subscribe to the principle of "learning by doing." As such my primary objective on day one of the Python boot camp is to get attendees up and coding as soon as possible. The goal for the first morning session is to cover the first four chapters of *Think Python*.

- Chapter 1: The way of the program;
- Chapter 2: Variables, expressions, and statements;
- Chapter 3: Functions;
- Chapter 4: Case study on interface design.

The material in these introductory chapters is clearly presented and historically students have generally had no trouble interactively working through the all four chapters before the lunch break. Most attendees break for lunch on the first day feeling quite good about themselves. Not only have they covered a lot of material, they have managed to write some basic computer programs. Maintaining student confidence is important: as long as students are confident and feel like they are progressing, they will remain focused on continuing to build their skills. If students get discouraged, perhaps because they are unable to solve a certain exercise or decipher a cryptic error traceback, they will lose their focus and fall behind.

The second morning session covers the next three chapters of *Think Python*:

- Chapter 5: Conditionals and recursion;
- Chapter 6: Fruitful functions;
- Chapter 7: Iteration.

At the start of the session I make a point to emphasize that the material being covered in chapters 5-7 is substantially more difficult than the introductory material covered in the previous morning session and that I do not expect many students to make it through the all of material before lunch. The idea is to manage student expectations by continually reminding them that the course is designed in order that they can learn at their own pace

The objective of for the third morning session is the morning session of day three the stated objective is for students to work through the material in chapters 8-10 of Think Python.

- Chapter 8: Strings;
- Chapter 9: A case study on word play;
- Chapter 10: Lists.

The material covered in chapter 8 and chapter 10 is particularly important as these chapters cover two commonly used Python data types: strings and lists. As a way of drawing attention to the importance of chapters 8 and 10, I encourage students to work through both of these chapters before returning to chapter 9.

The fourth morning session covers the next four chapters of *Think Python*:

- Chapter 11: Dictionaries;
- Chapter 12: Tuples;
- Chapter 13: Case study on data structure selection;
- Chapter 14: Files.

The morning session of day four is probably the most demanding. Indeed many students take two full session to work through this material. Chapters 11 and 12 cover two more commonly encountered and important Python data types: dictionaries and tuples. Chapter 13 is an important case study that demonstrates the importance of thinking about data structures when writing library code.

The final morning session is designed to cover the remaining five chapters of Think Python on object-oriented programming (OOP):

- Chapter 15: Classes and Objects;
- Chapter 16: Classes and Functions;
- Chapter 17: Classes and Methods;
- Chapter 18: Inheritance;
- Chapter 19: Case Study on Tkinter.

While this year a few students managed to get through at least some of the OOP chapters, the majority of students managed only to get through chapter 13 over the course of the five, three-hour morning sessions. Those students who did manage to reach the OOP chapters in general failed to grasp the point of OOP and did not see how they might apply OOP ideas in their own research. I see this as a major failing of my teaching as I have found OOP concepts to be incredibly useful in my own research. [stachurski2009], and [sargent2014] also make heavy use of OOP techniques.

While the morning sessions focus on building the foundations of the Python programming language, the afternoon sessions are devoted to demonstrating the use of Python in scientific computing by exploring in greater detail the Python scientific computing stack. During the afternoon session on day one I motivate the use of Python in scientific computing and spend considerable time getting students set up with a suitable Python environment and demonstrating the basic scientific work flow.

I provide a quick tutorial of the Enthought Canopy distribution. I then discuss the importance of working with a high quality text editor, such as Sublime, and make sure that students have installed both Sublime as well as the relevant Sublime plug-ins (i.e., SublimeGit and LatexTools for Git and LaTex integration, respectively; SublimeLinter for code linting, etc). I make sure that students can install Git and stress the importance of using distributed version control software in scientific computing and collaboration. Finally I cover the various flavors of the IPython interpreter: the basic IPython terminal, IPython QTconsole, and the IPython notebook.

The afternoon curriculum for days two through five is built around the Scientific Programming in Python lecture series and supplemented with specific use cases from my own research. My goal is to cover all of the material in lectures 1.3, 1.4, and 1.5 covering NumPy, Matplotlib and SciPy, respectively. In practice I am only able to cover a small subset of this material during the afternoon sessions.

### Advanced PhD training course

The final part of the course (for which I am still seeking funding to develop!) is a six week PhD advanced training course that focuses on applying cutting edge computational science techniques to economic problems via a series of interactive lectures and tutorials. The curriculum for this part of the course will derive primarily from [judd1998], [stachurski2009], and parts III and IV of [sargent2014]. In particular, I would like to cover the following material.

- Linear equations and iterative methods: Gaussian elimination, *LU* decomposition, sparse matrix methods, error analysis, iterative methods, matrix inverse, ergodic distributions over-identified systems.
- Optimization: 1D minimization, multi-dimensional minimization using comparative methods, Newton's method for multi-dimensional minimization, directed set methods for multi-dimensional minimization, non-linear least squares, linear programming, constrained non-linear optimization.
- Non-linear equations: 1D root-finding, simple methods for multi-dimensional root-finding, Newton's method for multi-dimensional root-finding, homotopy continuation methods.
- Approximation methods: local approximation methods, regression as approximation, orthogonal polynomials, least-squares orthogonal polynomial approximation, uniform approximation, interpolation, piece-wise polynomial interpolation, splines, shape-preserving approximation, multi-dimensional approximation, finite-element approximations.
- Economic applications: finite-state Markov chains, linear state space models, the Kalman filter, dynamic programming, linear-quadratic control problems, continuous-state Markov chains, robust control problems, linear stochastic models.

### Conclusion

In this paper I have outlined the three major components of my computational methods course: laboratory sessions, an intensive week-long Python boot camp, and an advanced PhD training course. The first two components are already up and running (thanks to funding support from the SGPE, SIRE, and the CIF). I am still looking to secure funding to develop the advanced PhD training course component.

I have been pleasantly surprised at the eagerness of economics graduate students both to learn computational modeling and simulation methods and to apply these techniques to the analytically intractable problems that they are encountering in their own research. Their eagerness to learn is, perhaps, a direct response to market forces. Both within academia, industry, and the public sector there is an increasing demand for both applied and theoretical economists interested in inter-disciplinary collaboration. The key to developing and building the capacity for inter-disciplinary research is effective communication using a common language. Historically that common language has been mathematics. Increasingly, however, this language is becoming computation. It is my hope that the course outlined in this paper might served as a prototype for other Python-based computational methods courses for economists and other social scientists.

## References


[behnel2011] S. Behnel, et al. *Cython: The best of both worlds*, Computing in Science and Engineering, 13(2):31-39, 2011.

[cass1965] D. Cass. *Optimum growth in an aggregative model of capital accumulation,* Review of Economic Studies, 32, 233–240.

[judd1998] K. Judd. *Numerical Methods for Economists*, MIT Press, 1998.

[koopmans1965] T. Koopmans. *On the concept of optimal economic growth,* Econometric Approach to Development Planning, 225–87. North-Holland, 1965.

[mankiw2010] N.G. Mankiw. *Intermediate Macroeconomics, 7th edition*, Worth Publishers, 2010.

[mas-colell1995] A.Mas-Colell,et al. *Microeconomic Theory, 7th ediition*, Oxford University Press, 1995.

[oliphant2007] T. Oliphant. *Python for scientific computing*, Computing in Science and Engineering, 9(3):10-20, 2007.

[peterson2009] P. Peterson. *F2PY: a tool for connecting Fortran and Python programs*, International Journal of Computational Science and Engineering, 4(4):296-305, 2009.

[ramsey1928] F. Ramsey. *A mathematical theory of saving,* Economic Journal, 38(152), 543–559.

[romer2011] D. Romer. *Advanced Macroeconomics, 4th edition*, MacGraw Hill, 2011.

[sargent2014] T. Sargent and J. Stachurski. *Quantitative Economics*, 2014.

[sato1963] R. Sato. *Fiscal policy in a neo-classical growth model: An analysis of time required for equilibrating adjustment*, Review of Economic Studies, 30(1):16-23, 1963.

[solow1956] R. Solow. *A contribution to the theory of economic growth*, Quarterly Journal of Economics, 70(1):64-95, 1956.

[stachurski2009] J. Stachurski. *Economic dynamics: theory and computation*, MIT Press, 2009.

[van2011] S. Van Der Walt, et al. *The NumPy array: a structure for efficient numerical computation*, Computing in Science and Engineering, 13(2):31-39, 2011.