The Heterogeneous-Agent Computational toolKit: An Extensible Framework for Solving and Estimating Heterogeneous-Agent Models

Christopher D. Carroll and Nathan M. Palmer

June 20, 2015

Abstract

We present a modular and extensible toolkit for solving and estimating heterogeneous-agent partial- and general-equilibrium models. Heterogeneous-agent models have grown increasingly valuable for both policy and research purposes, but code to solve such models can be difficult to penetrate for researchers new to the topic. As a result it may take years of human capital development for researchers to become proficient in these methods and contribute to the literature. The goal of the HACK toolkit is to ease this burden by providing a simple and easily extensible framework in which a few common models are solved, and clear documentation, testing, and estimation examples provide guidance for new researchers to develop their own work in a robust and replicable manner. Using two examples, we outline key elements of the toolkit which ease the burden of learning, using, and contributing to the codebase. This includes a simple API for model solution and an API for estimation via simulation, as well as methods for bundling working code with interactive documentation. The foundational solution method we employ is Carroll (2012), "Solution Methods for Microeconomic Dynamic Stochastic Optimization Problems," written in a modular Python framework. We briefly discuss a number of extensions as well as tertiary projects implied by this effort.

JEL codes E21, C61, E63

1 Introduction

The Heterogeneous-Agent Computation toolKit (HACK) is a modular programming framework for solving and estimating macroeconomic and macro-financial models in which economic agents can be heterogeneous in a large number of ways. Models with extensive heterogeneity among agents can be extremely useful for policy work. For example, Carroll (2012b, 2014a, 2014b) and Carroll et al. (2014) demonstrate how aggregate consumption and output can be heavily influenced by heterogeneity. Geanakoplos' (2009) outlines how heterogeneity drives the leverage cycle, and Geanakoplos et al. (2012) applies these insights to large-scale model of the housing and mortgage markets. However the most commonly published macroeconomic and macro-finance models have very limited heterogeneity or none at all (this includes the large class of representative agent models), in large part because these are the only models which can be easily solved with existing toolkits.[1] In contrast, models with extensive heterogeneity among agents have no central toolkit and must be solved in a bespoke way. This requires a significant investment of time and human capital before a researcher can produce publishable or usable work. This results in needless code duplication, increasing the chance for error and wasting valuable research time.

The HACK project addresses these concerns by providing a set of well-documented code modules which can be composed together to solve a range of heterogeneous-agent models. Methodological advances in the computational literature allow many types of models to be solved using similar approaches – the HACK

[1] Dynare is the most popular toolkit for representative-agent models. For more details see Adjemian et al. (2011).

toolkit simply brings these together in one place. The key is identifying methodologies which are both "modular" (in a sense to be described below) as well as robust to model misspecification. These include both solution methods as well as estimation methods.

In addition to these methodological advances, the HACK project adopts a few modern practices from the field of software development to ease the burden on code review, code sharing, and programming collaboration for researchers dealing in computational methods. Researchers who must review the scientific and technical code written by others are keenly aware that the time required to review and understand another's code can easily dwarf the time required to simply re-write the code from scratch (conditional on understanding the underlying concepts). This can be particularly important when multiple researchers may need to work on parts of the same codebase, either across time or distance. This problem is not confined to scientific computing alone. Fortunately the software development community, and particularly the open-source community, has spent decades perfecting tools for programmers to quickly consume and understand code written by others, verify that it is correct, and proceed to contribute back to a large and diverse codebase without fear of introducing bugs. The tools used by these professional developers include formal code documentation, unit testing structures, modern versioning systems for automatically tracking changes to code and content, and low-cost systems of communicating ideas, such as interactive programming notebooks which combine formatted mathematics with executable code and descriptive content. These tools often operate in concert with one another, forming a powerful infrastructure which can greatly accelerate project development for both individuals and collaborative teams. These technical tools are not new – the HACK project simply aims to apply the best of them to scientific code in a structured way to increase researcher productivity, particularly when interacting with other researchers' code.

The project presented here is not an attempt to create new methodology either on the software development front or the research front (although we expect new methodological contributions to emerge from the effort). Rather the HACK project brings together many well-understood and proven methodologies to bear in an easily used and extended toolkit. The rest of this paper will first outline the useful concepts adopted from software development, with examples of each. If the reader is a practiced and experienced programmer, he or she may wish to skip directly to Section 3 to see how these ideas are applied to the specific consumer problem employed. The sections are organized as follows: Section 2 outlines key tools from professional software development. Section 3 discusses the theoretical problem which provides the framework for the HACK project and outline of the first example model under development in the HACK framework. Section 4 outlines key next steps and concludes.

2 Tools from Software Development

Before progressing to a specific example, this section provides little background about the specific software tools HACK leverages. The breadth and history of software development is extensive, and review of it is beyond the scope of this document. One of the most striking practices to emerge from this history, however, is open-source software development – the decentralized collaboration of many independent programmers on a single project, often with little or no immediate monetary reward. Aside from fascinating theoretical questions about incentive structures, the open-source movement has produced a wide array of excellent utilities which make decentralized code development robust and efficient. These utilities are closely intertwined with those from the traditional software development world; this section briefly overviews a number tools from both. The following section outlines how the solution to a basic economic problem can be developed using these utilities.

There are a number of resources which delve deeply into the topics discussed in this section. For an excellent review of many of these topics from an economists perspective, see the unparalleled set of lectures by Sargent and Stachurski (2015), which can be accessed at the time of this writing at the Quant-Econ webpage. The Python programming language is the primary language used for development of HACK; many resources related to this language can be found on the primary Python webpage. Quant-Econ is an excellent introduction to Python for economists, as is Sheppard (2014). Aruoba and Fernández-Villaverde (2014) provide a nice comparison of many programming languages for computational economics, including Python.

2.1 An Aside on Speed

Python began as an interpreted scripting language and originally was many hundreds or thousands of times slower than compiled languages such as C++. Over time as the scientific community has adopted Python more extensively, a number of projects have emerged which allow Python to be compiled many different ways. At this time, there are a number of options for accelerating Python code. This is reflected in Table 1 in Aruoba and Fernández-Villaverde (2014), which compares a number of programming languages against C++ for a loop-intensive task. When sorted by relative time against the fastest C++ implementation, Python occupies the fastest two spots which are not other C++ or FORTRAN.[2] This is not a definitive illustration of the speed capabilities of Python, as there are many caveats which must be considered in the problem setup and execution (as noted by the authors themselves). However it does serve to illustrate that Python is capable of very high speeds when compiled. Furthermore, even aside from compilation, when Python is vectorized using the major numerical libraries, NumPy and SciPy, all vectorized calculations are executed in optimized, compiled C and FORTRAN.

The one caveat to statements about Python speed is in order for complex code structures. Object-oriented programming structures in Python can prove difficult to compile easily. Extensive computations on class-based objects can impose significant speed penalties. There are a number of ways around this. The simplest is to write code which does not require class structure – simple functional libraries. This is the approach HACK takes. This allows individual functions to be accelerated via vectorization or compilation, maximizing speed potential. If a class structure cannot be avoided, the accelerated functions can be called directly by members of the class, inheriting much of the speed advantages for the compiled code.[3] Finally, the HACK library written as a functional library versus a class-based system allows easy translation of the HACK library into additional languages if desired. Julia is a promising target for such an effort; see Sargent and Stachurski (2015) and their accompanying website for more information on Julia.

2.2 Documentation

Good documentation is the key to communication between two programmers, whether between two distinct individuals in the cross section or with oneself over time. In Python, as in many scripting languages, strings written on the first line after a function declaration are automatically employed as system documentation. Two popular style guide for Python documentation are found in the Google Style Guide and the Python Enhancement Proposals (PEP) system: PEP 8 and PEP 257. HACK currently uses a slight variation on the PEP 257 style guide. We illustrate this with a trivial example of a Python documentation string for a CRRA utility function. The documentation is the section of code enclosed by the triple quotes:[4]

```
def utility(c, gamma):
    """
    Return constant relative risk aversion (CRRA) utility of consumption "c"
    given risk aversion parameter "gamma."

    Parameters
    ----------
    c: float
        Consumption value.
    gamma: float
```

[2] The first five spots in the relative ranking are occupied by different compiler implementations of C++ and FORTRAN. The 6th and 7th ranks are occupied by two of the most popular Python compilers, Cython and Numba, respectively, which are 1.41 and 1.65 times slower than the fastest C++ implementation. Notably, two C++ implementations are 1.38 times slower than the fastest, and one of the two FORTRAN implementations is 1.30 times slower than the fastest C++ implementation. That is, the fastest Python implementation is only about 3% slower than two of the three C++ implementations.

[3] Note that if the speed advantage of the individual function comes from vectorization versus compilation, the most gain may actually be achieved by simply copy-and-pasting the function contents into the class method. See Sheppard (2014), Chapter 23, for an excellent overview of Python performance and code optimization.

[4] Triple quotes in Python set off a multi-line string.

```
        Risk aversion, gamma != 1.

    Returns
    -------
    u: float
        Utility.

    Notes
    -----
    gamma cannot equal 1. This constitutes natural log utility; np.log
    should be used instead.
    """
    return( c**(1.0 - gamma) / (1.0 - gamma) )
```

This documentation is now employed by the language in the formal language help system. If we query the language help files for this function we will get these results:

```
>>> utility?
Type:        function
String form: <function utility at 0x7f96b473e6e0>
File:        ~/workspace/HACKUtilities/<ipython-input-20-2bbd1323015e>
Definition:  utility(c, gamma)
Docstring:
Return constant relative risk aversion (CRRA) utility of consumption "c"
given risk aversion parameter "gamma."

Parameters
----------
c: float
    Consumption value.
gamma: float
    Risk aversion, gamma != 1.

Returns
-------
u: float
    Utility.

Notes
-----
gamma cannot equal 1. This constitutes natural log utility; np.log
should be used instead.
```

In addition to traditional code documentation, notebook-style interfaces similar to those in Mathematica and Maple have been developed for Python and a number of other languages. Programmers, including scientific programmers, have begun using these notebooks to directly communicate the ideas behind executable code to one another via html output from these notebooks, displayed in webpages. The HACK project uses Jupyter, a language-agnostic, browser-based notebook system spun off of the the IPython project. An example from the Jupyter homepage can be seen in Figure 1.

These notebooks can embed TeX-style mathematics typesetting alongside text and code for highly expressing scientific programming vignettes.

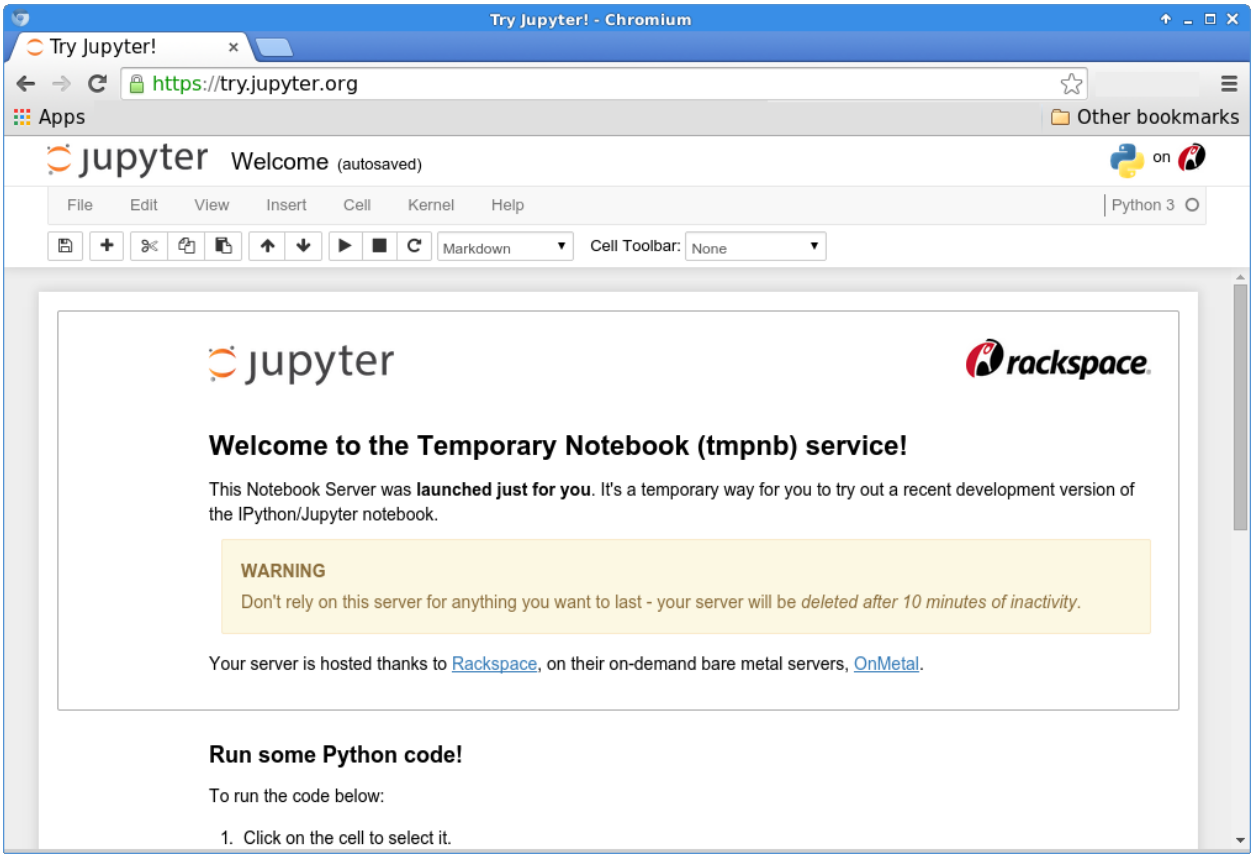

Figure 1: Jupyter Browser-Based Notebook

2.3 Unit Testing

Many programs are composed of a number of small functions which accomplish specific tasks. Testing at the individual function level is key to ensuring that the overall program executes correctly. This is all the more important for scientific computing, where a mistake deep in the code (eg. with a numerical approximation function) may be extremely difficult to track down. Unit testing is the formal practice whereby each individual function is directly bundled with a set of tests. Each test tests a specific input and output pair, examining both "success" and "failure" states. For example, a log utility function should return a specific known value for a particular risk aversion and consumption value, and should fail with a particular error if it is presented a negative consumption value. Unit tests serve multiple purposes in code: they cover a wide range of "reasonable and representative" input and output values, and also act to conceptually illustrate when a bit of code is very complex. If it is very difficult to test a "small unit" of code, that code *may* be best decomposed into smaller and more specific-purpose functions.[5]

In scientific programming, this can serve an additional purpose: peer review of code. Uncovering bugs in code, even one's own code, can be notoriously difficult. This is many times more true when one is examining the code written by another. Thus scientific peer review of code is nearly prohibitively costly, and very difficult to undertaken in a structured fashion. Unit testing can ease the burden of scientific code review in at least two ways. First, it can aid documentation in immediately outlining simple examples of code execution. Second, it can outline the pitfalls and testing procedures a reviewer may want to undertake to ensure that the code is correct. Instead of starting with a "blank page," a reviewer can take the unit tests written by the original author, run them, and then (assuming they all pass), examine the tests to see if any particular cases appear to be excluded. If so, the reviewer can use the unit tests as a template to quickly write another test case and run that as well. This can greatly accelerate both the verification of work done, as well as new testing of the code, all in a well-established and minimally costly framework.

In Python there are two built-in ways to write tests for a function: internally to the documentation, in a "doctest," and externally in a more formal unit testing framework, "unittest." Using the utility function defined earlier above, we add a doctest to the end of the function documentation (removing earlier documentation for brevity). The tests are denoted by the triple right-caret under the heading denoted "Tests." The appropriate output of the test is denoted in the line directly below the caretted line, and we will use the doctest library to run these tests. First the code definition:

```
def utility(c, gamma):
    """
    Return CRRA utility of consumption "c" given risk aversion parameter "gamma."

    ...(excluded for brevity)...

    Tests
    -----
    Test a value which should pass:
    >>> utility(1.0, 2.0)
    -1.0

    Test a value which should fail:
    >>> utility(1.0, 1.0)
    Traceback (most recent call last):
        ...
    ZeroDivisionError: float division by zero
    """
    return( c**(1.0 - gamma) / (1.0 - gamma) )
```

[5] Note that there is a tradeoff between performance and decomposition of code into smaller and smaller units. This is discussed in Sheppard (2014).

We save this code in a file called "utility.py." The code file now constitutes a Python module, and we use the doctest library to execute the tests embedded in the doctring. We execute the following code:

```
>>> import utility      # Import the new utility module
>>> import doctest      # Import built-in doctest module
>>>
>>> doctest.testmod(utility, verbose=True)  # Execute doctest on utility
Trying:
    utility(1.0, 2.0)
Expecting:
    -1.0
ok
Trying:
    utility(1.0, 1.0)
Expecting:
    Traceback (most recent call last):
        ...
    ZeroDivisionError: float division by zero
ok
1 items had no tests:
    utility
1 items passed all tests:
   2 tests in utility.utility
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

We see that the two tests passed. A contributor or a reviewer can quickly run these tests on new code, and quickly add new tests if needed. More complicated tests can be executed with the unittest framework, which is not discussed here in depth.

2.4 Language-agnostic, Human-Readable Data Serialization

Consider the following scenario: a researcher wants to replicate a computational model. After endless work and testing, the results between two codebases simply cannot be reconciled. Many hours are spent hunting for bugs until it is finally discovered that the problem is not in the code, but rather in a small mistake transcribing parameter settings. For some (most?) this can be an all-too-familiar experience.

One way to avoid this is using the exact same parameter settings file for all possible code-bases. One language-independent file is used to store all parameters and calibration settings for a model.[6] A replication of a particular model can use that single parameter file to confidently reproduce results across implementations. The parameter file should be easily readable by a human, as this is one more place mistakes may occur and the easier to double-check, the better. Calibration of course may require many different types of data objects contained together in a single setting – floating point numbers, strings, booleans, even vectors or arrays of values. Flat-file data formats such as CSV are not flexible enough to handle all these types well. Fortunately, modern software developers have already addressed this problem with a number of options. The HACK project uses JSON (JavaScript Object Notation), a data structure somewhat analogous to simplified XML. The contents of a small JSON file may look like the following. Note the ability to include vectors, strings, and boolean values in the same file:[7]

[6] Not including the data for fitting the model – this may easily be very large and is stored separately.

[7] The values used in the examples in this paper are illustrative and not used for a particular estimation exercise, unless otherwise noted.

```
{
"rho":   3.0,
"beta":  0.99,
"R":     1.03,
"liquidity_constraint": true,
"interpolation_type": "linear",
"psi_sigma": [0.001, 0.001, 0.001,
    0.28, 0.27, 0.27, 0.26, 0.26,
    0.25, 0.24, 0.23, 0.22, 0.21],
"Gamma":  [ 1.0,   1.0,   0.7,
1.01,  1.01,  1.01,  1.01,  1.01,
1.025, 1.025, 1.025, 1.025, 1.025],
}
```

The HACK project uses a single JSON file to store parameters and calibration values which can be used across multiple implementations of a model, even in multiple languages. For example, the same JSON file can be read by both a MATLAB and Python implementation of the same model.[8]

2.5 Application Programming Interface (API)

When contributing a module or a function to a larger code library, a programmer needs to know how this function or module fits into the overall framework of the codebase. A strict specification of variable inputs and outputs for a function communicates this information. This is the so-called Application Programming Interface. Any large computational project with multiple developers can benefit from such an interface. Some languages such as Java cleave to the API approach at a deep level, in the language definition itself, while others have a looser approach to API definition. In Python an API can be formally defined in a number of ways, or not at all. The HACK project forms a very simple API for the codebase, organized around the modules required to run a partial-equilibrium or general-equilibrium estimation.

Another use of APIs is to define an interface between any language and particular datasets. This second use of APIs, the database use, is just as important as its usage in organizing code. Given the vast differences in different microeconomic database structures, this is very difficult utility to create for broad use. Organizations such as the Open Economics Working Group may provide a unified approach for economic data.

2.6 Version Control

An essential tool in distributed software development is a system which can automatically archive versions of code, as well as allow the merging of changes to a document by two different programmers. Such a system is known as a version control system. The HACK project uses the Git version control system, and uses the popular online repository service Github to archive its codebase. Chacon and Straub (2014) is an excellent overview of version control in general and Git and Github in particular. Github allows code to be posted to a single online repository which tracks previous versions. A repository is copied to the computer of each contributor, and the central code source on Github may be kept private, accessible only to select users, or made widely available to the general public. Github provides a number of services on top of the pure repository service, including a simple wiki space, a space for a static website, and simple one-off repositories called "Gists," which allow the quick public or private posting of a variety of content, including Jupyter notebooks.[9]

[8] A file which reads in the parameters and sets up the environment will of course be required for each language, and the researcher must be careful to treat parameters equivalently in this setup step.

[9] See this Github blogpost, "GitHub + Jupyter Notebooks = <3", which explicitly outlines the use of Github for sharing notebooks.

2.7 Bringing It Together: Reproducible Research

Many of the tools above are used to create research which can be immediately reproduced, even entirely in a web browser. This gallery of interesting IPython Notebooks outlines a number of research projects which combine code, discussion, data visualization, and descriptive mathematics to make science as transparent and reproducible as possible. For example Ram and Hadany (2015) reproduce a section of their work in an IPython notebook, which can be found here, and excerpt of which can be seen in Figures 2, 3, 4.

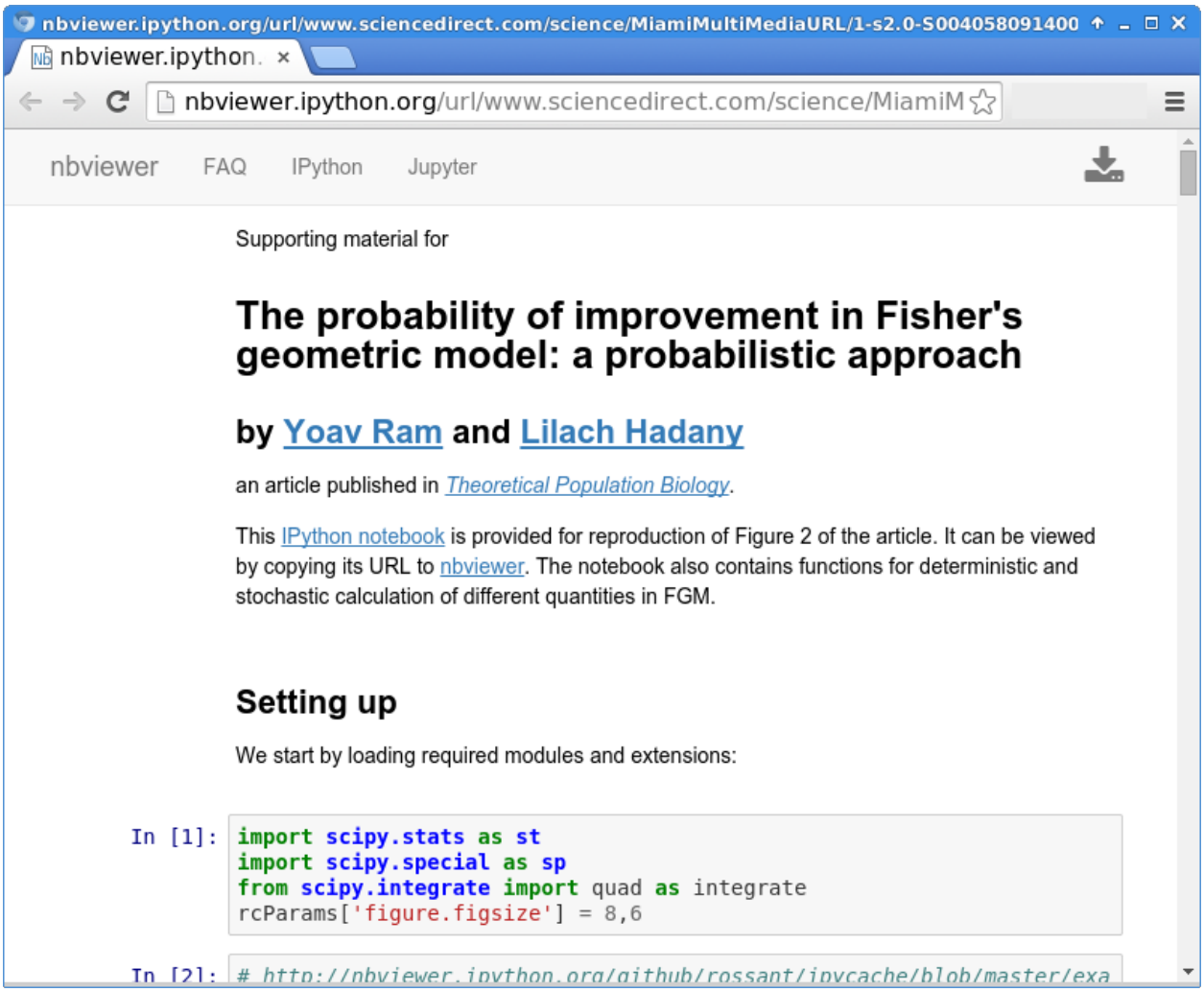

Figure 2: Ram and Hadany (2015) Notebook Excerpt 1

Many additional examples of reproducible research are available in the gallery noted above.

3 Methodological Framework

The foundational agent for the HACK toolkit is the microeconomic rational consumer. The agent's problem is stated as a dynamic stochastic optimization problem which is solved via dynamic programming. Given the solution method and appropriate data, the model is then estimated via Simulated Method of Moments (SMM), with standard errors obtained via the bootstrap. The key is to write the code such that there is a logical division between elements of the solution method. The ideal solution method decomposition should

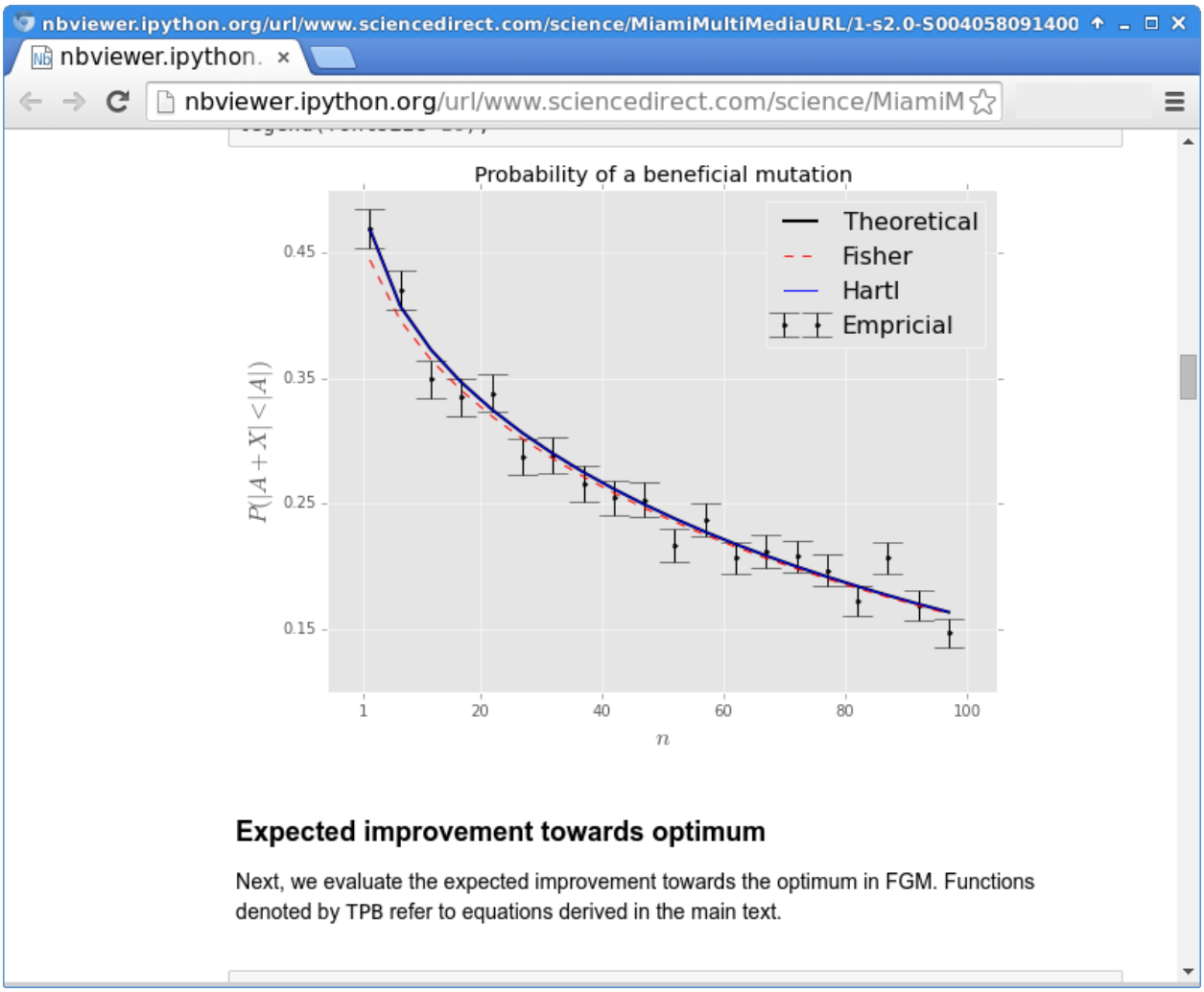

Figure 3: Ram and Hadany (2015) Notebook Excerpt 2

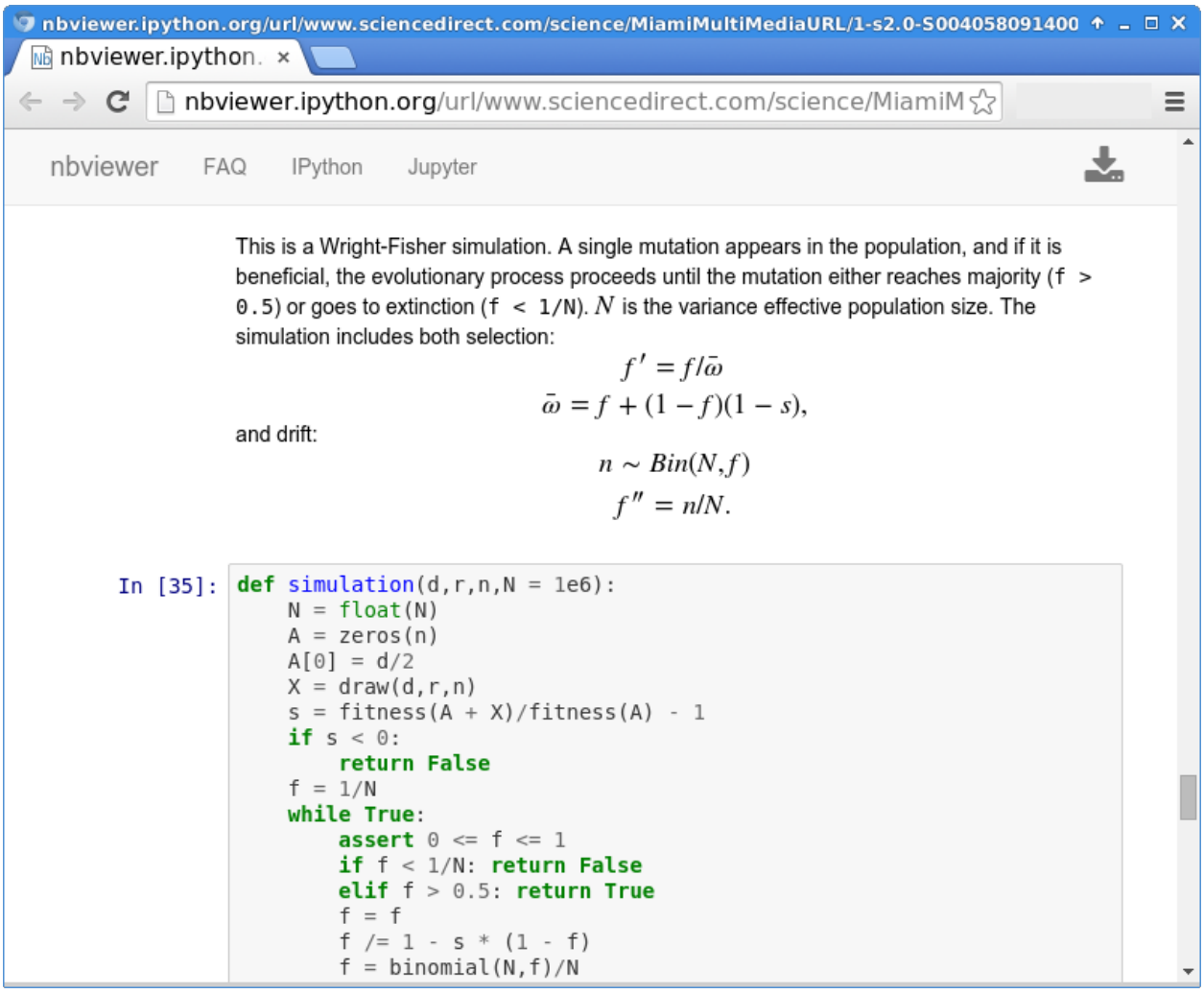

Figure 4: Ram and Hadany (2015) Notebook Excerpt 3

allow the various modules of the code to be agnostic to one another – if one module is replaced by a different module, which simply takes the same appropriate inputs and outputs, the solution works as before. To use a software term, the HACK project defines an API (Application Programming Interface) which instructs the user in how different solution modules communicate with one another, regardless of what they do internally. We call an element of the solution method "modular" is this is particularly easy to do. For example, as described further below, the Simulated Method of Moments estimation procedure can be robust in this way – under broad conditions, it can be applied to a very wide range of dynamic decision models[10].

One final note before proceeding. This modular approach aligns well with the authors' strategy of implementing non-optimal behavior as a departure from an already-established optimizing framework. If agents are to learn, the first straightforward extension is to learn in about some element of the solution method – eg. learn expectations or learn the optimal policy. If an agent makes mistakes, there are clear places to implement these mistakes in the optimizing framework: in the expectation function, in the law of motion, in following the optimal policy. In the codebase of HACK, the optimal problem is developed first as the framework upon which further extensions are hung. Implementing a non-optimizing solution thus involves augmenting or extending a bit of the baseline code.

This rest of this section outlines the basic optimization problem and solution method which forms the foundation of the HACK framework. The solution method is decomposed into a few major conceptual parts, which are implemented as modular libraries in HACK. Additional unimplemented solution methods are discussed. At each stage, the modular nature of the methods are noted.

3.1 A Basic Partial-Equilibrium Example

Consider the following finite-horizon consumption-under-uncertainty problem.[11] At time $T+1$, the consumer dies with certainty. The problem is to allocate consumption appropriately from $t=0$ to $t=T$. The full problem from Carroll (2012a) is:

$$\begin{aligned} \max_{\{\mathbf{c}_{t+j}\}_{j=0}^{\infty}} \mathbb{E}_t \left[\sum_{j=0}^{T-t} \beta^j u(\mathbf{c}_{t+j}) \right] \\ s.t. \\ \mathbf{a}_t &= \mathbf{m_t} - \mathbf{c_t} \\ \mathbf{b}_{t+1} &= \mathbf{a_t}\mathsf{R} \\ \mathbf{p}_{t+1} &= \mathbf{p_t}\Gamma\psi_{t+1} = \mathbf{p_t}\Gamma_{t+1} \\ \mathbf{m}_{t+1} &= \mathbf{b}_{t+1} + \mathbf{p}_{t+1}\xi_{t+1} \\ \mathbf{m}_0 \ given \end{aligned}$$

where

- $\mathbf{a}_t$ is end-of-period assets,
- $\mathbf{m}_t$ is beginning-of-period total market resources ("cash on hand"),
- $\mathbf{c}_t$ is consumption in period t,
- R is a constant return factor on assets, $\mathsf{R} = (1+r)$,
- $\mathbf{p}_t$ is permanent non-asset income,
- Γ is a constant permanent income growth factor,
- ψ_t is a mean-1 iid permanent shock to income, and
- ξ_t is a mean-1 iid transitory shock income, composed as

[10] The choices represented in HACK are not the only modular solution methods which may be used, but rather a baseline. If you have a favorite solution method which you believe is robust and modular as described here, you are encouraged to contribute!

[11] See Carroll (2014) for much more background and detail on this style of problem.

$$\xi_t = \begin{cases} 0 & with\ prob\ \wp_t > 0 \\ \frac{\theta_t}{\not\wp_t} & with\ prob\ \not\wp_t \equiv (1 - \wp_t) \end{cases}, \text{where}$$

- $\wp_t$ is a small probability that income will be zero
- θ_t is a mean-1 iid shock transitory to income

This setup can describe a wide range of consumer circumstances, including retirement and fixed income over final years of life.

The utility function $u(.)$ is of the Constant Relative Risk Aversion (CRRA) form with risk-aversion parameter *rho*:

$$u(c) = \frac{c^(1-\rho)}{1-\rho}.$$

As in Carroll (2012a), this problem can be normalized by permanent income $\mathbf{p}_t$ to produce a simplified version of the full problem, with a reduced number of state variable. The bold symbols used above indicate non-normalized variables, while the regular non-bold symbols used below indicate variables normalized by permanent income.[12] The normalized problem can be written in Bellman form:

$$\begin{aligned} v_t(m_t) &= \underset{c_t}{max}\ u(c_t) + \beta \mathbb{E}_t \left[\Gamma_{t+1}^{1-\rho} v_{t+1}(m_{t+1}) \right] \\ &s.t. \\ a_t &= m_t - c_t \\ b_{t+1} &= \left(\frac{\mathsf{R}}{\Gamma_{t+1}} \right) a_t = \mathcal{R}_{t+1} a_t \\ m_{t+1} &= b_{t+1} + \xi_{t+1} \\ &m_0\ given \end{aligned}$$

or simplified further:

$$\begin{aligned} v_t(m_t) &= \underset{c_t}{max}\ u(c_t) + \beta \mathbb{E}_t \left[\Gamma_{t+1}^{1-\rho} v_{t+1}(m_{t+1}) \right] \\ &s.t. \\ m_{t+1} &= \mathcal{R}_{t+1}(m_t - c_t) + \xi_{t+1} \\ &m_0\ given \end{aligned}$$

3.2 The Solution Method

The general solution method is as follows: in the final period T, the value function in the following period is $v_{T+1}(m) = 0\ \forall_m$, and the value function in period T is simply $v_T(m) = u(m)$. This makes the problem in period $T-1$ straightforward to solve numerically for both the consumption function and value functions $c^*_{T-1}(m)$ and $v^*_{T-1}(m)$:

$$c^*_{T-1}(m) = \underset{c \in [0, \bar{m}]}{argmax}\ u(c) + \beta \mathbb{E}_{T-1} \left[\Gamma_T^{1-\rho} u(\mathcal{R}_T(m-c) + \xi_{t+1}) \right]$$

and

$$v^*_{T-1}(m) = u(c^*_{T-1}(m)) + \beta \mathbb{E}_{T-1} \left[\Gamma_T^{1-\rho} u(\mathcal{R} u(c^*_{T-1}(m)) \right]$$

[12] See Carroll (2012a) for the details of this process.

where $\bar{m}$ is a self-imposed liquidity constraint.[13]

With these numerical solutions in hand, the solution method is now simply recursive: step back one more period to $T-2$ and solve for optimal consumption and value functions using $c^*_{T-1}(m)$ and $v^*_{T-1}(m)$. This process can be continued back until the first period $t=0$. This solution process is outlined in greater detail in Carroll (2012a).

3.3 The Estimation Method

Denote the behavioral parameters β, ρ, (discounting and risk aversion, respectively) as

$$\phi = \{\beta, \rho\}$$

and denote the structural problem parameters as

$$\begin{aligned} \varrho &= \{\varrho_t\}_{t=0}^{T}, \text{where} \\ \varrho_t &= \{\Gamma, \psi_t, \xi_t, \wp_t, \theta_t\}, \forall_t. \end{aligned}$$

Given an arbitrary behavioral parameter set $\phi = \{\beta, \rho\}$, and choosing the values and data-generating processes for the structural problem parameters ϱ to match consumer experiences in the PSID, we can solve for the set of consumption functions which are optimal under these conditions, $\{c^*_t(m)\}_{t=0}^{T}$.

With these consumption functions now in hand, we can use the calibrated parameters ϱ to generate N different simulated consumer experiences (vectors of income shocks) from $t = 0, 1, ..., T$. Applying the consumption functions $\{c^*_t(m)\}_{t=0}^{T}$ to this set of simulated experiences generates a N-sized distribution of simulated wealth holdings for all t. The moments of these cross-sectional distributions of wealth can then be compared to the equivalent moments in appropriately constructed empirical data from the Survey of Consumer Finance (SCF). We form the following objective function, which compares population median between empirical wealth-to-income ratio from the SCF and its simulated equivalent:

$$\varpi_\varrho(\phi) \equiv \sum_{i=1}^{N} \omega_i |\zeta_i^\tau - \mathbf{s}_\varrho^\tau(\phi)|.$$

Here $\varpi_\varrho(\phi)$ represents the objective value for the distance between medians of the two populations, the synthetic population variables represented by (s) and the empirical population variables represented by ζ (see Carroll 2012a for more discussion of the form of this objective function for population moments). The index i indicates individual observations in the empirical data, each of which has a population weight ω_i (required in the SCF due to oversampling of particular sub-populations). Each individual i in the empirical data has observations at the age-group frequency, τ. The variable $\mathbf{s}_\varrho^\tau(\phi)$ is the median of the simulated data for age group τ, under calibration ϱ, using the parameters $\phi = \{\beta, \rho\}$. Once this value has been constructed as a function of ϕ, the estimation occurs by simply finding the minimal ϕ value numerically:

$$\min_{\phi} \varpi_\varrho(\phi).$$

This is accomplished in code by simply handing the expression $\varpi_\varrho(\phi)$ to a numerical minimization process. The standard error on the resulting estimation of $\{\beta^*, \rho^*\}$ is found by bootstrapping the empirical data and repeating the above estimation process a number of times, $N_{bootstrap}$.

[13] Carroll (2012a) demonstrates the reasoning behind this derivation. In a model with positive probability of a zero-income event, $\bar{m} = m$.

3.4 Modular Solution and Estimation in HACK

The solution and estimation method described for the basic problem above can be decomposed into the following steps, each of which is written as a module in Python. Each module is documented, tested, and brought together in IPython notebook "vignettes" to demonstrate their use, and finally brought together in a simple interface to effect model solution and estimation. Extending the partial-equilibrium toolkit corresponds to writing a new version of a specific set of functions in each of the basic modules, using the exiting code and vignettes as examples and guides.

The major conceptual solution and estimation components for the partial-equilibrium problem are:

- parameter definition
- expectations formation and calculation
- value and policy formation
- simulation of population experience under a particular policy
- estimation of parameters using SMM and bootstrapping

These parts together form the basis of the partial-equilibrium portion of HACK. Code is divided into the following primary modules, corresponding to the solution method breakdown noted above:

- SetupParameters.py
- HACKUtilities.py
- SolutionLibrary.py
- SimulationLibrary.py
- Estimation.py

This section examines these basic modules and outlines the process by which the library can be expanded to include additional models. Work is underway to build out the general-equilibrium portion using the approach implemented in Carroll et al. (2015). Namely, the following additions will be made:

- price-finding via market clearing
- rational expectations via the Krusell-Smith (1998) algorithm.

The rest of this section outlines the main contents of the five modules noted above. Each is illustrated with pseudo-code headers and content as needed. For each module, the primary functions which need to be modified to extend the baseline model are identified and discussed.

3.4.1 SetupParameters.py

This module is coupled with an input JSON file, which specifies the full set of calibrated values required to solve the model. The JSON file, as noted previously, is language-independent, and can be read and used by nearly any modern programming language. The HACK project has used this in particular to validate multiple-language versions of the same model, eg. between MATLAB and Python. This setup allows calibration parameters to be specified once, in a separate, easily human-readable file. Importantly, the SetupParameters also executes a key function for the estimation step: it brings in and organizes the empirical data to be used in the estimation process,[14] and it defines a function *find_simulated_medians* which will take in simulated wealth data and organizes it to be comparable to the empirical data as stated in the SMM objective expression,

[14] This data is usually stored in a separate format than the parameters in the JSON file.

$$\sum_{i=1}^{N} \omega_i |\zeta_i^\tau - \mathbf{s}_\varrho^\tau(\phi)|.$$

When the user desires to change the estimation procedure (eg. change the empirical data or moments compared), the SetupParamters file must be changed appropriately.

This SetupParameters file is imported into any subsequent module which needs to access the parameters using the following line of code. Note that particular parameters are immediately accessible:

```
>>> import SetupParameters as param
>>>
>>> print "R =", param.R
R = 1.03
>>> print "Gamma =", param.Gamma
array([ 1.   ,  1.   ,  1.   ,  1.   ,  1.   ,  1.   ,  1.   ,  1.   ,
        0.7  ,  1.01 ,  1.01 ,  1.01 ,  1.01 ,  1.01 ,  1.01 ,  1.01 ,
        1.025,  1.025,  1.025,  1.025,  1.025,  1.025,  1.025,  1.025,
        1.025])
>>> print "Initial beta guess for SMM optimization:", param.beta_start
Initial beta guess for SMM optimization: 0.99
```

Note that the JSON file includes initial guesses for the parameters to be estimated by the Simulated Method of Moments routine. To extend the baseline HACK model, new calibrated parameter values will need to be added to this file.

3.4.2 HACKUtilities.py

This module contains a number of utilities used by the HACK framework, including the code to implement agent expectations. Agent expectations here are implemented as a discretization of the shock-space, achieved by choosing the size of discrete points to represent the distribution, $N_{discrete}$, creating an equiprobably-spaced partition over the support, and selecting the representative point in each partition as the conditional mean of values in the partition. Each resultant point is then assigned the probability $\frac{1}{N_{discrete}}$. See Carroll (2012a) for a detailed discussion of this approach.

An example of the code which implements a mean-1 lognormal shock space is as follows:

```
def calculate_mean_one_lognormal_discrete_approx(N, sigma):
    '''
    Calculate a discrete approximation to a mean-1 lognormal distribution.

    Parameters
    ----------
    N: float
        Size of discrete space vector to be returned.
    sigma: float
        standard deviation associated with underlying normal probability distribution.

    Returns
    -------
    X: np.ndarray
        Discrete points for discrete probability mass function.
    pmf: np.ndarray
```

```
    Probability associated with each point in X.

Test
----
Confirm that returns discrete mean of 1
>>> import numpy as np
>>> x, pmf = calculate_mean_one_lognormal_discrete_approx(N=5, sigma=0.2)
>>> np.dot(x,pmf)
1.0
'''
mu = -0.5*(sigma**2)
distrib = stats.lognorm(sigma, 0, np.exp(mu))

# ------ Set up discrete approx ------
pdf = distrib.pdf
invcdf = distrib.ppf
probs_cutoffs = np.arange(N+1.0)/N      # Includes 0 and 1
state_cutoffs = invcdf(probs_cutoffs)   # State cutoff values, each bin

# Set pmf:
pmf = np.repeat(1.0/N, N)

# Find the E[X|bin] values:
F = lambda x: x*pdf(x)
Ebins = []

for i, (x0, x1) in enumerate(zip(state_cutoffs[:-1], state_cutoffs[1:])):
    cond_mean1, err1 = quad(F, x0, x1, epsabs=1e-10, epsrel=1e-10, limit=200)
    # Note that the *first* to be fulfilled of epsabs and epsrel stops the
    # integration - be aware of this when the answer is close to zero.
    # Also, if you never care about one binding (eg if one would like to
    # force scipy to use the other condition to integrate), set that = 0.
    Ebins.append(cond_mean1/pmf[i])

X = np.array(Ebins)

return( [X, pmf] )
```

Note the embedded doctest, which runs "sanity checks" on the discretization process. Tests such as these identified initial numerical errors in the discretization process due to loose default integration tolerances – a key contribution of unit testing which can help avoid many hours of bug hunting in incorrect portions of the codebase.

The key modular feature of the HACKUtilities library is that it produces, finally, a single discrete representation of the probability space faced by the consumer. As long as shocks are iid, the number of dimensions of shocks does not matter – each distribution is discretized and the joint distribution is create combinatorically from the individual discrete marginal distributions. To create expectations, the HACKUtilities library finally produces a set of combinations of all discrete points as the support, and the corresponding combination of all discrete probabilities as the distribution. Expectations of a function f are then formed simply by the dot product of f applied to each point with the probabilities associated with all points.

If additional methods of expectations formation are desired, this is the correct module in which to develop them.

3.4.3 SolutionLibrary.py

The solution library contains the main definitions used in the solution method. The HACK project uses dynamic programming to determine the solution to the consumer problem, and in particular uses the endogenous gridpoints method to greatly accelerate solving for the policy function.[15] This solution method takes advantage of the *end of period* consumption and value functions,

$$\mathfrak{c}(a_t) \text{ and } \mathfrak{v}(a_t)$$

which map end-of-period wealth a_t to consumption and expected value. The endogenous gridpoints method is not required for the HACK project, but it greatly accelerates the solution method and is used in the baseline HACK model. This is one example of including concrete examples of non-trivial computational "tricks" which may greatly improve a solution method. The endogenous gridpoints method may not be easy to understand upon first encounter. Thus the HACK toolkit includes it along with an IPython notebook which quickly illustrates how the process works – an interactive and executable summary of Carroll (2012a).

The key methods in the SolutionLibrary module are the following, shown only with their function headers – documentation and code details are excluded for brevity. The main functions are:

Utility functions: The CRRA utility function is defined with its first and second derivative:

```
def utility(c, gamma):
    pass

def utilityP(c, gamma):
    pass

def utilityPP(c, gamma):
    pass
```

The end-of-period consumption function for period $T-1$ and all other $t < T-1$ These are the key functions used in the endogenous gridpoints backwards induction method:

```
def gothicC_Tm1(a, rho, uP, R, beta, Gamma, psi_support, xi_support, pmf):
    pass

def gothicC_t(a, c_prime, rho, uP, R, beta, Gamma, psi_support, xi_support, pmf):
    pass
```

Initialize the consumer problem: A consumer's problem must be set up before it can be solved: the expectations support and probability mass function must be created for each period:

```
def init_consumer_problem(R, Gamma, constrained,
                          psi_sigma, psi_N, xi_sigma, xi_N, ...):
    pass
```

Solve the consumer problem: The recursive solution method is implemented by two functions: the first solves a single period in the problem ("one step back"), while the second implements the full backwards recursion, from period $T-1$ to 0:

[15] The endogenous gridpoints method is discussed in extensive detail in Carroll (2006, 2012a)

```
def step_back_one_period(rho, beta, R, Gamma, shocks, pmf, a_grid):
    pass

def solve_total_consumption_problem(rho, beta, R, Gamma, shocks, pmf, a_grid):
    pass
```

The final function, "*solve_total_consumption_problem*," returns a list of consumption function objects, ordered in reverse chronology (index 0 is the consumption function for period T, index 1 is consumption function for $T-1$, etc.). All other functions in the SolutionLibrary module can be thought of as supporting the final *solve_total_consumption_problem* function. Thus the baseline model can be extended by overwriting the *solve_total_consumption_problem* function, as well as creating/overwriting whatever additional functions are needed to support the new implementation. For example, if a model is extended with respect to solution methods – for example, if Bayesian learning is added to the agent solution method – this module is the correct place to include that extension.

3.4.4 SimulationLibrary.py

The simulation library implements the simulation step of the estimation process – given a reverse-chronology list of consumption functions, the functions in this library will draw an appropriate panel of shocks of size $\{T, N_{simulate}\}$, where T is total periods and $N_{simulate}$ is the total number of agents for which to simulate experiences (eg. $T = 60$ and $N_{simulate} = 10,000$).

The key methods in the SimulationLibrary module are the following, again shown only with function headers for brevity:

Main two functions: The main two functions in the SimulationLibrary module first create the complete set of shocks required to simulate agent experiences and, second, apply the consumption solution from the SolutionLibrary to this set of shocks to produce the simulated wealth panel. To extend the baseline model, these are the two major functions to overwrite, as well as the requisite helper functions. The two major functions appear as follows:

```
def create_income_shocks_experience(psi_sigma, xi_sigma, Gamma, R, p_unemploy, T, N):
    pass

def find_wealth_history_matrix(policies, state0, agent_shocks_matrices):
    pass
```

Helper functions: There are a number of "helper" functions in the SimulationLibrary which support *create_income_shocks_experience* and *find_wealth_history_matrix*. The single task of creating and simulating the shocks is split across a number of helper functions to aid in both clarity and unit testing. Together they are used to create the final matrices of shocks needed to simulate consumption:

```
# ------- Generate permanent and transitory income shocks ------- #
def generate_permanent_income_draws(psi_sigma, N_simulate):
    pass

def generate_transitory_income_draws(temp_sigma, p_unemploy, N_simulate):
    pass

# ------ Create joint discrete distribution from independent marginals ----- #
def generate_all_combined_shocks(shocks1, p1, shocks2, p2):
    pass
```

```
# ------ Generate matrix of perm and transitory shocks of correct size ------- #
def create_shocks_matrix_1D(shocks, N_simulate, T, seed=None):
    pass

def create_shocks_matrix_1D_with_zero_income_event(shocks, N, T, seed=None):
    pass

# ------ Generate retirement income if needed ------ #
def generate_retire_income(psi_retire, xi_retire, Gamma, R, p_unemploy_retire):
    pass
```

3.4.5 Estimation.py

The final module in HACK is the Estimation module. This brings all the others together to solve, simulate, and estimate the preference parameters for the basic consumption-under-uncertainty problem outlined above. In the pseudo-code below, all the major functions which are necessary for the operation of the HACK project are outlined. To change the baseline model, one only needs to change the five major functions imported from other modules and used in the Estimation module. These five functions and their definitions comprise the programming API for the HACK framework:

- SimulationLibrary:
  - *create_income_shocks_experience*
  - *find_wealth_hist_matrix*
- SolutionLibrary:
  - *init_consumer_problem*
  - *solve_consumption_problem*
- SetupParameters:
  - *find_simulated_medians*

The module first imports all necessary HACK modules, executes the primary functions from each, creates the SMM objective function $\varpi_{\varrho}(\phi)$, and executes a single minimization:

$$\min_{\phi} \varpi_{\varrho}(\phi).$$

Also included in this module is a bootstrap function which repeats the minimization for a bootstrap sample of data, $N_{bootstrap}$ times.

A special note for the pseudo-code for Estimation.py that follows: the Python operator "**" acts to unpack a dictionary (a hash-table data storage object in Python) and use its key to associate the dictionary values with the appropriate function calls. Thus the definitions of the dictionaries "unpacked" by the ** symbols below act to keep the code clean and readable.

This code excerpt includes the pseudo-code for the calculation of the SMM objective function, *smm_objective_fxn*, which is almost entirely complete code:

```
import SetupParameters as param
import SolutionLibrary as solution
import SimulationLibrary as simulate

# ------ Create income shock draws ------ #
```

```
agent_shocks_matrices = simulate.create_income_shocks_experience(**param.create_income_shocks)

# ------ Initialize the consumer problem ------ #
income_distrib = solution.init_consumer_problem(**param.init_consumer_problem)

# ------ Create full collection of calibration parameters ------ #
calibrated_parameters = {"shocks":agent_shocks_matrices, "income":income_distrib}
calibrated_parameters.update(param.calibrated_parameters)

# ------ Define the SMM objective function ------ #
def smm_objective_fxn(phi, **calibrated_parameters):
    #  Solve consumption functions
    consumption = solution.solve_consumption_problem(phi, ...)
    # Run simulation and get simulated wealth holdings
    sim_m_history = simulate.find_wealth_hist_matrix(...)
    # Construct simulated medians correctly for efficient calculation...
    simulated_medians = param.find_simulated_medians(sim_m_history)
    # Return sum of absolute errors
    return np.dot(empirical_weights, np.abs(empirical_data - simulated_medians))


# ------ Execute a single minimization on the SMM objective function ------ #
def minimize_smm_objective(minimizer):
    pass

# ------ Bootstrap the minimization on the SMM objective function ------ #
def bootstrap(optimizer, param.empirical_data, param.Ndraws, **calibrated_parameters):
    pass
```

The beauty of the modular HACK structure emerges in this final Estimation module. In all previous modules, extending the baseline model may require much overwriting of the basic functions. However once that writing is done, the work needed to extend the Estimation.py module is minimal. In fact, the only changes likely necessary are to some of the function inputs and outputs in the body of the *smm_objective_fxn* function.

4 Summary and Conclusion

The HACK project is a modular code library for constructing macroeconomic and macro-financial models with heterogeneous agents solving portfolio decisions under uncertainty. Portfolio choice under uncertainty is central to nearly all academic models, including modern DSGE models (with and without financial sectors), models of asset pricing (eg. CAPM and C-CAPM), models of financial frictions (eg. Bernanke et al. 1999), and many more. Under the right assumptions many of these models can be solved by aggregating agent decision-making and employing the representative agent, with standardized computational frameworks for solving these models. However when individual agents look very different from one another - for example, different wealth levels, preferences, or exposures to different types of shocks - assumptions required for aggregation can quickly fail and a representative agent may no longer be appropriate. Code to solve these models tends to be bespoke and idiosyncratic, often reinvented by different researchers working on similar problems. This needless code duplication increases the chance for errors and wastes valuable researcher time.

Researchers should spend their valuable time producing research, not reinventing the wheel when it comes to computational tools. The goal of the HACK toolkit is to ease this burden by providing a simple and easily extensible framework in which a few common models are solved, and clear documentation, testing, and estimation frameworks provide guidance for new researchers to develop their own work in a robust and replicable manner. The final goals of the project are to create a collaborative codebase which can serve both

researchers and policymakers alike, employing the best of modern software development tools to accelerate understanding and implementation of cutting edge research tools. The solution methods employed in HACK are not the only methods available, and those who have additional methodological suggestions are strongly encouraged to contribute! Increasing returns to production is one of the few "non-dismal" possibilities in economic thought – we hope to capture this feature of code production in the HACK framework. Key next steps include finalizing the general-equilibrium HACK modules, identifying additional baseline models to replicate in HACK, and encouraging a new generation of students to learn from, use, and contribute to the collaborative construction of heterogeneous-agent models.

Bibliography

Aruoba, S. B., and Fernández-Villaverde, J. (2014). "A Comparison of Programming Languages in Economics." No. W20263. National Bureau of Economic Research.

Chacon, S. and Straub, B. (2014). Pro Git. Apress, 2014. At: http://git-scm.com/book

Carroll, C. D. (2006). "The method of endogenous gridpoints for solving dynamic stochastic optimization problems."" Economics letters, 91(3), 312-320.

Carroll, C. D. (2012a). "Solving microeconomic dynamic stochastic optimization problems." Mimeo. Johns Hopkins University Department of Economics, 2012.

Carroll, C.D. (2012b). "Implications of Wealth Heterogeneity For Macroeconomics." Johns Hopkins University Department of Economics Working Paper Number 597 May 2012.

Carroll, C.D. (2014a). "Representing Consumption and Saving Without a Representative Consumer." In Measuring Economic Sustainability and Progress Studies in Income and Wealth. National Bureau of Economic Research 2014.

Carroll, C.D. (2014b). "Heterogeneous Agent Macroeconomics: An Example and an Agenda." Presentation at IMF Workshop on Computational Macroeconomics. Washington, December 2014.

Carroll, C.D., Slacalek, J., Tokuoka, K., and White, M.N. (2015). "The Distribution of Wealth and the Marginal Propensity to Consume."" Draft Johns Hopkins University March 2015.

Geanakoplos, J. (2009). "The leverage cycle."" In NBER Macroeconomics Annual 2009, Volume 24 (pp. 1-65). University of Chicago Press.

Geanakoplos, J., Axtell, R., Farmer, D. J., Howitt, P., Conlee, B., Goldstein, J., Hendry, M., Palmer, N. M., and Yang, C. Y. (2012). "Getting at systemic risk via an agent-based model of the housing market." The American Economic Review, 102(3), 53-58.

Ram, Y., and Hadany, L. (2015). "The probability of improvement in Fisher's geometric model: A probabilistic approach."" Theoretical population biology, 99, 1-6.

Sargent, T. and Stachurski, J. (2015). "Quantitative Economics with Python." Lecture Notes. Accessed 16 May 2015. At: http://quant-econ.net/_static/pdfs/py-quant-econ.pdf

Sheppard, K. (2014). "Introduction to Python for Econometrics, Statistics and Numerical Analysis: Second Edition." Lecture Notes. Accessed 16 May 2015. At: https://www.kevinsheppard.com/Python_for_Econometrics