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 initial work on 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 and research purposes. 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. 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.

*Department of Economics, Johns Hopkins University
†Department of Computational Social Science, George Mason University, and Office of Financial Research
‡Views expressed in this paper are those of the authors and not necessarily of the Office of Financial Research or other affiliated institutions.

1 Dynare is the most popular toolkit for representative-agent models. For more details see Adjemian et al. (2011).
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 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 modern practices from the field of software development to ease the burden of 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.

There are a number of related code libraries and projects in economics, although with small exception, they do not take advantage of modern software-engineering tools the the extent intended in HACK. A significant exception to this is Sargent and Stachurski’s QuantEcon project, which makes extensive use of software development tools to ensure code integrity and replicability. In its current form, QuantEcon largely addresses representative-agent-style models, with a pedagogical purpose. QuantEcon is the closest in spirit to the HACK project. Any non-exhaustive list of computational tools for economics must include Dynare, a MATLAB/Octave-based toolkit which provides a “top-down” approach for solving macroeconomic and related models, largely of the representative-agent variety. Dolo is a recent effort to solve the same types of problems as Dynare using the Python programming language, using state-of-the-art just-in-time compilers to greatly accelerate solution time. There are a number of high-quality individual code libraries in a number of additional languages, such as for the Anderson-Moore algorithm, Miranda and Fackler’s CompEcon toolkit (MATLAB and R), and Heer and Maussner’s Gauss and Fortran libraries, to name a few. The Review of Economic Dynamics has instituted the RED code repository. Finally, there has been a recent rise in replication efforts in economics, explicitly in the recent Replication Wiki and the Replication Network projects. The number of problems addressed and solved by these toolkits is extensive, and the HACK project intends to interact with these libraries and their authors directly when possible.

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 we adopt from software development, with examples of each, and then demonstrate how these concepts are applied in turn to the key solution and estimation methods required to solve general heterogeneous-agent models. If the reader is a practiced and experienced programmer, he or she may wish to skip directly to Section 3 to see

---

2 A “top-down” approach, in our terminology, is achieved by defining a high-level pseudo-language in which a model may be fully described. The model itself is then solved and simulated in the background, with implementation details hidden from the user.

3 A number of methods exist for “wrapping” C++, FORTRAN and other code such that non-Python procedures may be called from Python. See the Python wiki entry on Integrating Python With Other Languages, at the time of this writing, for a number of these options.
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 outlines 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 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. Along with fascinating theoretical questions about incentive structures, the open-source movement has spurred the development of 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 of these tools from both. The next 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 economist’s 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 is an interpreted scripting language and at inception was many hundreds or thousands of times slower than compiled languages such as C++. As the scientific community adopts Python, a number of projects have emerged which allow Python to be compiled. At the time of this writing, there are a number of options for accelerating Python code. This is reflected in Aruoba and Fernández-Villaverde (2014), specifically their Table 1. The authors compare 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. 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 in order regarding Python speed. Object-oriented programming structures in Python may prove difficult to compile easily, and extensive computations on class-based objects may 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

---

4The 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.
the speed advantages for the compiled code. Finally, the HACK library written as a functional library versus a class-based system allows easy translation 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 the Julia language applied to economic problems.

### 2.2 Documentation

Good documentation is the key to communication between two programmers, whether between two distinct individuals 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 guides 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 special function documentation here is enclosed in triple quotes, which in Python sets off a multi-line string. This special function documentation is called the “docstring:”

```python
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
        Risk aversion, gamma != 1.
    
    Returns
    -------
    u: float
        Utility.
    
    Notes
    ----- 
    gamma cannot equal 1. This constitutes natural log utility; np.log 
should be used instead.
    """
    u = c**(1.0 - gamma) / (1.0 - gamma)  # Find the utility value of c given gamma
    return u  # Return the utility value
```

More traditional in-line code comments may of course still be employed using the hash symbol (“#”), as can be seen in the final two lines of code above. The normal in-line comments seen above will not be included in the special function documentation, as they do not occur on the first line after the function declaration.

The special function documentation in the docstring is now employed in the formal language help system. If we query the language help files for this function we will get the following results. Note that our documentation written above now appears under the label “Docstring:”

```plaintext
>>> utility?  # The question-mark queries the help file
Type: function
```

---

5Note 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.
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.

Once code has been documented correctly, automated documentation-creation systems can turn this simple docstring into a fully structured documentation product in a number of formats, including HTML, PDF and others. An example of the documentation above in HTML format, as created by the Sphinx documentation generator\(^\text{6}\), can be seen in Figure 1.

In addition to traditional code documentation described above, which is included directly in the code file, 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 2.

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

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

\(^{6}\text{See http://sphinx-doc.org/ for more information and many examples. Sargent and Stachurski’s QuantEcon project, for example, uses Sphinx for their documentation and webpages at the time of this writing.}\)
The SolutionLibrary module contains...

The utility function

SolutionLibrary. \texttt{utility}(c, gam)

Return constant relative risk aversion (CRRA) utility of consumption "c" given risk aversion parameter "gam" (gamma).

\begin{itemize}
  \item \texttt{c}: float
    Consumption value.
  \item \texttt{gam}: float
    Risk aversion, \texttt{gam} != 1.
  \item \texttt{u}: float
    Utility.
\end{itemize}

\texttt{gamma} cannot equal 1. This constitutes natural log utility; \texttt{np.log} should be used instead.

Figure 1: Documentation for utility generated in HTML by Sphinx

Figure 2: Jupyter Browser-Based Notebook
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.\footnote{\textsuperscript{7}}

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 \texttt{``doctest,''} and externally in a more formal unit testing framework, \texttt{``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 \texttt{``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:

```python
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)))
```

We save this code in a file called \texttt{``utility.py.'''} The code file now constitutes a Python \texttt{module}, and we use the doctest library to execute the tests embedded in the documentation. We execute the following code:

```bash
>>> 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:

\footnote{\textsuperscript{7}}\textsuperscript{\textsuperscript{7}}Note that there is a tradeoff between performance and decomposition of code into smaller and smaller units. This is discussed in Sheppard (2014).
-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. 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:

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

---

8Not including the data for fitting the model – this may easily be very large and is stored separately.
9The values used in the examples in this paper are illustrative and not used for a particular estimation exercise, unless
otherwise noted.
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.\textsuperscript{10}

### 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. Any large computational project with multiple developers can benefit from such a description, formally termed the Application Programming Interface (API). This is in fact simply another form of language documentation, but one that is aimed at programmers for extending or using a codebase. A clear example of a programming API for a large library of scientific code can be seen in the documentation for the Apache Math Commons Library.\textsuperscript{11} Figure 3 displays an excerpt of an example API for the Brent Solver optimization routine.

![Constructor Summary](image)

**Figure 3: Apache Math Commons API Excerpt for Brent Solver**

The API for this Java implementation of the Brent Solver serves two purposes: first, it communicates the basic requirements for the methods inputs and outputs. Second, however, it also instructs any programmers who wish to extend the code as to the structure similar code must take.\textsuperscript{12}

\textsuperscript{10}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.

\textsuperscript{11}Apache Commons Math is a library of lightweight, self-contained mathematics and statistics methods addressing common problems and solutions not available in the Java programming language. See [http://commons.apache.org/proper/commons-math/](http://commons.apache.org/proper/commons-math/) for more information.

\textsuperscript{12}For this particular example, there are more extensive details in the BaseUnivariateSolver API, here: [https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/analysis/solvers/BaseUnivariateSolver.html](https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/analysis/solvers/BaseUnivariateSolver.html)
In Python an API can be formally or informally defined in a number of ways, as determined by the needs of a project. The HACK project employs an automated documentation generation process to create a code API similar in nature to that of the Apache Commons Math API discussed above. As noted in the documentation section, as long as functions are documented thoroughly, an automated documentation tool can extract that in-code documentation and structure it as HTML or PDF output for user reference. An example of the API in progress for the HACK project created by Sphinx can be seen in Figure 4, which depicts an excerpt of the API documentation for the SolutionLibrary module.

![Figure 4: Documentation for SolutionLibrary generated in HTML by Sphinx](image)

An alternative use of APIs, aside from code documentation, is to define an interface between a programming language and a particular dataset. 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 difficult utility to create for broad use. A preliminary version is created in the SetupEmpirical HACK module. This module both organizes the empirical data against which the synthetic simulation data is to be compared, and defines a function which takes raw synthetic simulation data and organizes it to be directly comparable to the empirical 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. Chacon and Straub (2014) is an excellent overview of version control in general and Git and Github in particular.  

\footnote{Organizations such as the Open Economics Working Group may provide one possible unified approach for publicly available economic datasets.}
Git allows code to be posted to a single repository which tracks previous versions. A repository may be copied to the computer of each authorized contributor. The central code source on may be kept private, accessible only to select users, or made widely available to the general public. Git-powered repositories may provide 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 which allow the quick posting of a variety of content, including Jupyter notebooks.\footnote{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 5, 6, 7.

Figure 5: Ram and Hadany (2015) Notebook Excerpt 1

Many additional examples of reproducible research are available in the gallery noted above.
Expected improvement towards optimum

Next, we evaluate the expected improvement towards the optimum in FGM. Functions denoted by TPB refer to equations derived in the main text.

Figure 6: Ram and Hadany (2015) Notebook Excerpt 2
This is a Wright-Fisher simulation. A single mutation appears in the population, and if it is beneficial, the evolutionary process proceeds until the mutation either reaches majority \( f > 0.5 \) or goes to extinction \( f < 1/N \). \( N \) is the variance effective population size. The simulation includes both selection:

\[
    f' = f\hat{\omega}
\]

and drift:

\[
    \hat{\omega} = f + (1-f)(1-s),
\]

\[
    n \sim Bin(N,f)
\]

\[
    f'' = nlN.
\]

```python
In [35]: def simulation(d, r, n, N = 1e6):
    N = float(N)
    A = zeros(n)
    A[0] = d/2
    X = draw(d, r, n)
    s = fitness(A + X)/fitness(A) - 1
    if s < 0:
        return False
    f = 1/N
    while True:
        assert 0 <= f <= 1
        if f < 1/N: return False
        elif f > 0.5: return True
        f = s
        f /= 1 - s * (1 - f)
        f = binomial(N, f)/N
```

Figure 7: Ram and Hadany (2015) Notebook Excerpt 3
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 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” if 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\(^{15}\).

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.\(^ {16}\) 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:

\[
\max_{\{c_{t+1}\}_{j=0}^{\infty}} \mathbb{E}_t \sum_{j=0}^{T-t} \beta^j u(c_{t+j})
\]

s.t.

\[
\begin{align*}
    a_t &= m_t - c_t \\
    b_{t+1} &= a_t R \\
    p_{t+1} &= p_t \Gamma \psi_{t+1} = p_t \Gamma_{t+1} \\
    m_{t+1} &= b_{t+1} + p_{t+1} \xi_{t+1} \\
    m_0 &\text{ given}
\end{align*}
\]

where

- \(a_t\) is end-of-period assets,
- \(m_t\) is beginning-of-period total market resources (“cash on hand”),

\(^{15}\)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!

\(^{16}\)See Carroll (2014) for much more background and detail on this style of problem.
• $c_t$ is consumption in period $t$,
• $R$ is a constant return factor on assets, $R = (1 + r)$,
• $p_t$ is permanent non-asset income,
• $\Gamma$ is a constant permanent income growth factor,
• $\psi_t$ is a mean-1 iid permanent shock to income, and
• $\xi_t$ is a mean-1 iid transitory shock income, composed as

$$
\xi_t = \begin{cases} 
0 & \text{with prob } \psi_t > 0 \\
\frac{\theta_t}{\psi_t} & \text{with prob } \psi_t = (1 - \psi_t),
\end{cases}
$$

where

• $\psi_t$ is a small probability that income will be zero
• $\theta_t$ 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 $\rho$:

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

As in Carroll (2012a), this problem can be normalized by permanent income $p_t$ 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.$^{17}$ The normalized problem can be written in Bellman form:

$$v_t(m_t) = \max_{c_t} u(c_t) + \beta E_t \left[ \Gamma_{t+1}^{1-\rho} v_{t+1}(m_{t+1}) \right]$$

s.t.

$$a_t = m_t - c_t$$

$$b_{t+1} = \left( \frac{R}{\Gamma_{t+1}} \right) a_t = \mathcal{R}_{t+1} a_t$$

$$m_{t+1} = b_{t+1} + \xi_{t+1}$$

$m_0$ given

or simplified further:

$$v_t(m_t) = \max_{c_t} u(c_t) + \beta E_t \left[ \Gamma_{t+1}^{1-\rho} v_{t+1}(m_{t+1}) \right]$$

s.t.

$$m_{t+1} = \mathcal{R}_{t+1}(m_t - c_t) + \xi_{t+1}$$

$m_0$ given

$^{17}$See Carroll (2012a) for the details of this process.
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 $c^*_T(m)$ and $v^*_T(m)$:

$$c^*_T(m) = \arg\max_{c \in [0, \bar{m}]} [\Gamma^{1-\rho} u(R_T(m - c) + \xi_{t+1})]$$

and

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

where $\bar{m}$ is a self-imposed liquidity constraint.\(^{18}\)

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 $c^*_T(m)$ and $v^*_T(m)$. 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 $\beta, \rho$, (discounting and risk aversion, respectively) as

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

and denote the structural problem parameters as

$$\varrho = \{\varrho_t\}_{t=0}^T, \text{ where } \varrho_t = \{\Gamma, \psi, \xi_t, \psi_t, \theta_t\}, \forall t.$$  

Given an arbitrary behavioral parameter set $\phi = \{\beta, \rho\}$, and choosing the values and data-generating processes for the structural problem parameters $\varrho$ 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 $\varrho$ 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:

$$\varpi_\varrho(\phi) \equiv \sum_{i=1}^{N} \omega_i |z^I_i - s^*_\varrho(\phi)|.$$  

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 $\zeta$ (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 $\omega_i$.

\(^{18}\)Carroll (2012a) demonstrates the reasoning behind this derivation. In a model with positive probability of a zero-income event, $\bar{m} = m$.  

16
(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, \(\tau\). The variable \(s^*_{\tau}(\phi)\) is the median of the simulated data for age group \(\tau\), under calibration \(\phi\), using the parameters \(\phi = \{\beta, \rho\}\). Once this value has been constructed as a function of \(\phi\), the estimation occurs by simply finding the minimal \(\phi\) value numerically:

\[
\min_{\phi} \varpi_{\phi}(\phi).
\]

This is accomplished in code by simply handing the expression \(\varpi_{\phi}(\phi)\) 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, \(N_{\text{bootstrap}}\).

### 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
- setup of data and data/simulation comparison
- 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
- SetupEmpirical.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.

The module discussion begins with the final Estimation module listed above, as this module clearly outlines the specific functions and intermediate data structures which must be overwritten to extended the basic HACK framework to solve and estimate another model.
3.4 Modular Solution and Estimation in HACK

3.4.1 Estimation.py

The central 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 six functions and their definitions comprise the main code of the partial-equilibrium HACK framework:

- **SimulationLibrary**:
  - `create_income_shocks_experience`
  - `find_wealth_hist_matrix`
- **SolutionLibrary**:
  - `init_consumer_problem`
  - `solve_consumption_problem`
- **SetupParameters**:
  - `setup_simulated_medians`
- **SetupEmpirical**:
  - `find_empirical_mappings`

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

\[
\min_{\phi} \varpi(\phi).
\]

Also included in this module is a bootstrap function which repeats the minimization for a bootstrap sample of data, \( N_{\text{bootstrap}} \) 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:

```python
import SetupParameters as param
import SetupEmpirical as empirical
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)
```
3.4 Modular Solution and Estimation in HACK

3.4.1 The Estimation Operation

The beauty of the modular HACK structure emerges in this Estimation module. To start an extension of the baseline model, the user first identifies which of the functions in the above process must be overwritten or extended. Once this is established, the user simply walks through the codebase and makes the appropriate adjustments. Once that work is done, only minor changes are required for Estimation.py. Furthermore, the documentation, testing, and organization of all other methods are outlined in the baseline example, easing the process for the new user.

Figure 8 displays a screen shot from the simple command-line interface to the estimation process. The estimation module prompts the user for a selection from numerical optimization options, then asks for initial conditions. After the initial estimation, the option to bootstrap the standard errors is provided.

The brute force optimization option “c” will search a fixed grid for the optimal point, and produce a contour plot over the grid as part of the output. The surface for the particular parameterization used for this paper can be seen in Figure 9.

3.4.2 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, e.g. 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, and it defines a function setup_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:

$$\sum_{i=1}^{N} \omega_i |\zeta_i - s_i^\phi(\phi)|.$$  

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

In [1]: execfile("Estimation.py")

Please choose a solution method by typing its letter, [Enter] to use Nelder-Mead, or q to quit:
[a] Nelder-Mead [default]
[b] Powell
[c] Brute search and contour plot.
[a]

Please enter a numerical value for initial beta guess, or hit return to use the default value of 0.99

1.0

Please enter a numerical value for initial rho guess, or hit return to use the default value of: [4.0]

3.5

Starting Nelder-Mead with initial value (beta, rho) = (1.0, 3.5)
Optimization terminated successfully.
Current function value: 42600.463076
Iterations: 51
Function evaluations: 99
Output: [ 0.92637873  2.66819023]
time to solve: 0.518002335231 min
time to solve: 31.0801491138 sec

Would you like to see the bootstrap estimates of variance? Y/[N]

Figure 8: Simple Simulated Method of Moments Estimation User Interface
Figure 9: Simple Simulated Method of Moments Estimation Contour Plot
3.4 Modular Solution and Estimation in HACK

When the user desires to change the estimation procedure (e.g., change the empirical data or moments compared), the SetupParameters 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:

```python
>>> import SetupParameters as param

>>> print "R =", param.R
R = 1.03

>>> print "Gamma =", param.Gamma
array([ 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 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.

3.4.3 SetupEmpirical.py

This module sets up empirical data, along with functions which allow moments of the empirical data to be matched to moments of the simulated data. The main functions are:

```python
def find_empirical_mappings(...):
    pass

def setup_simulated_medians(...):
    pass

def bootstrap_data_and_components(...):
    pass
```

The first function constructs mapping functions between the empirical and simulated data moments. The second function uses these mappings to organize simulated data moments to map to empirical data moments, and the final function implements data resampling procedures required for the bootstrap estimates of variance.

3.4.4 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, $N_{\text{discrete}}$, 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 $\frac{1}{N_{\text{discrete}}}$. 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.

### 3.4.5 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.\(^{20}\) This solution method takes advantage of the end of period consumption and value functions,

\[
 c(a_t) \quad \text{and} \quad v(a_t)
\]

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:

```python
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:

```python
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
```

\(^{20}\)The endogenous gridpoints method is discussed in extensive detail in Carroll (2006, 2012a)
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:

```python
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:

```python
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.

### 3.4.6 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:

```python
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 generatePermanent_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

3.5 A Simple Extension Example

Extending the partial-equilibrium HACK library proceeds by reviewing the major functions which make up the library and determining which need to change. As before, the Estimation module acts as the central organizing example and provides a convenient outline for determining required changes.

We illustrate this process with a trivial example: replacing the endogenous gridpoints policy iteration with a more traditional policy iteration which does not use post-decision state variables.

The first step is reviewing each of the main functions. Recall that the following are the main functions employed in the Estimation module, abstracting from code structure:

# (1) Create income shocks experience
agent_shocks_matrices = simulate.create_income_shocks_experience(**param.create_income_shocks)

# (2) Create income distribution for expectation formation
income_distrib = solution.init_consumer_problem(**param.init_consumer_problem)

# (3) The Simulated Method of Moments objective function itself:
obj = smm_objective_fxn(phi, **calibration_superset)

# (4) - (6) Functions called in the smm_objective_fxn:
# (4) Solve for set of consumption functions for simulation
consumption = solution.solve_consumption_problem(phi, **calibration_subset)

# (5) Use behavior to find simulated data
sim_m_history = simulate.find_wealth_hist_matrix(consumption, state0, agent_shocks_matrices)

# (6) Find simulation medians in form of empirical data
simulated_medians = empirical.setup_simulated_medians(sim_m_history, **simulation_empirical_mappings)
These can be stylized:

- `simulate.create_income_shocks_experience`
- `solution.init_consumer_problem`
- `smm_objective_fxn`
  - `solution.solve_consumption_problem`
  - `simulate.find_wealth_hist_matrix`
  - `empirical.setup_simulated_medians`

To extend the model we walk down the list of main functions and determine which will need to change. In the case of switching out the endogenous gridpoints policy iteration for more traditional policy iteration, we find we only need to change the following:

- `simulate.create_income_shocks_experience`
- `solution.init_consumer_problem`
- `smm_objective_fxn`
  - `solution.solve_consumption_problem`
  - `simulate.find_wealth_hist_matrix`
  - `empirical.setup_simulated_medians`

All else can stay the same. We create an alternative solve_consumption_problem function, called “solve_consumption_problem_PI,” which takes the same inputs and outputs. The model can then be run immediately again and compared to the endogenous gridpoints output. In this case the non-endogenous gridpoints method takes ~100 times longer to run.

A more complex example may involve modeling agents such that their solution is actually influenced by their experiences. If agents learn a near-optimal solution from experience, as in Allen and Carroll (2001) and Palmer (2012), then the solution for each agent will be unique to their shock experience, even if all agents have the same behavioral parameters. This makes for a more complex extension to the HACK library.

Reviewing the list of key functions above, we see that the following must be changed:

- `simulate.create_income_shocks_experience`
- `solution.init_consumer_problem`
- `smm_objective_fxn`
  - `solution.solve_consumption_problem`
  - `simulate.find_wealth_hist_matrix`
  - `empirical.setup_simulated_medians`

As above, the solve_consumption_problem will need to be changed, somewhat more drastically than before. The object it outputs is now a matrix rather than list of consumption functions. The find_wealth_hist_matrix is changed as well, in that it is combined with the solve_consumption_problem function – consumption functions are updated “online” and thus the simulation portion is subsumed in the solve_consumption_problem function. Due to the nature of the learning, the “init_consumer_problem” can be dropped entirely, as agent expectations do not need to be constructed. Due to the large increase in computational cost, this solution method is much slower than even the non-endogenous gridpoints policy iteration above.
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


