Heterogeneous Agents Resources and toolKit (Soft Documentation for a Soft Launch)

December 13, 2015

Christopher D Carroll[1]
CFPB and Johns Hopkins University

David C Low[2]
CFPB

Nathan M Palmer[3]
OFR, United States Treasury

Matthew N White[4]
U of Delaware, CFPB

Open Computational
Economics Workshop
2015

[1] Carroll: http://econ.jhu.edu/people/ccarroll/, email: ccarroll@jhu.edu [2] Low: david.c.low@cfpb.gov [3] Palmer: nathan.m.palmer@ofr.treasury.gov [4] White: http://www.lerner.udel.edu/faculty-staff/matthew-n-white/, email: mnwecon@udel.edu

1 Introduction

If you are willing to risk some mild psychological trauma, conjure to mind your first experience of hand-coding a structural economic model. Your clunky effort probably built on legacy code provided by an adviser or colleague – which itself came from who-knows-what apocryphal sources. Efforts to combine elements from one model with those from another were likely frustrated by the "Tower of Babel" problem: Code from one source could not "speak" to code from another without your own intermediation as a translator, possibly between two unfamiliar languages and aided only by oracular comments that, at best, made sense only in the context of other (now missing) code.

After months of effort, you may have had the character-improving experience of proudly explaining to your adviser that not only had you grafted two ideas together, you also found a trick that speeded the solution by an order of magnitude, only to be told that your breathtaking insight had been understood for many years, as reflected in an appendix to a 2008 paper; or, worse, your discovery was something that "everybody knows" but did not exist at all in published form!

Learning by doing has value, but only within limits. We do not require young drivers to design an internal combustion engine before driving a car, nor must graduate students write their own matrix inversion algorithms before running an OLS regression.

In recent years, considerable progress has been made in addressing these kinds of problems in many areas of economic modeling. Macroeconomists using representative agent models can ship Dynare model files to each other; reduced form econometricians can choose from a host of econometric packages; statisticians have R. But modelers whose questions require explicit structural modeling involving nontrivial kinds of heterogeneity (that is, heterogeneity that cannot simply be aggregated away) are mostly still stuck in the bad old days.

The ultimate goal of the HARK (Heterogeneous Agents Resources and toolKit) project is to fix these problems. Specifically, our aim is to produce an open source repository of highly modular, easily interoperable code for solving, simulating, and estimating dynamic economic models with heterogeneous agents.[1] At this "soft launch" event we are presenting a nascent alpha version of HARK, focusing solely on microeconomic dynamic optimization problems in discrete time. The kernel of this early version is a unifying framework for such models, so that *any* microeconomic model that can be written as a sequence of choices by an optimizing agent can use the same "universal solver," differing only in how the "one period problem" is solved. Further, we seek to establish (with input from the community) standards for the description of objects like discrete approximations to continuous distributions and interpolated function approximations, so that numeric methods can be quickly swapped without ugly "patching."

This document serves as semi-formal "soft" documentation of the alpha version of HARK, including a walkthrough of our microeconomic agent framework, with an application to consumption-saving models. Section 2 discusses the formatting and features of the `HARKcore`module and the `AgentType` class, as well as a summary of the supporting

[1]By "heterogeneous," we mean both *ex ante* and *ex post* heterogeneity: agents differ before anything in the model has "happened;" and agents will experience different stochastic events during the model.

modules and tools. Section 3 then presents how consumption-saving models can be formulated, solved, and estimated with `ConsumptionSavingModel`, using the tools of the main HARK modules. Finally, in Section 4 we offer a preview of functionality that we hope to provide in a future beta version of HARK, when we open the toolKit to contributions from the outside world.

2 HARKcore and AgentType

The core of our microeconomic dynamic optimization framework is a flexible object-oriented representation of economic agents. The `HARKcore`module defines a superclass called `AgentType`; each model defines a subclass of `AgentType`, specifying additional model-specific features and methods while inheriting the methods of the superclass. Most importantly, the method `solve()` acts as a "universal solver" applicable to any (properly formatted) discrete time model. This section describes the format of an instance of `AgentType` as it defines a dynamic microeconomic problem;[2] it concludes with a brief overview of other "toolbox" modules used to solve, simulate, and estimate models defined with `HARKcore`.

2.1 Attributes of an AgentType

A discrete time model in our framework is characterized by a sequence of "periods" that the agent will experience. A well-formed instance of `AgentType` includes the following attributes:

- `solveAPeriod`: A function pointer, or a list of function pointers, representing the solution method for a single period of the agent's problem. The inputs passed to a `solveAPeriod` function include all data that characterize the agent's problem in that period, including the solution to the subsequent period's problem (designated as `solution_tp1`). The output of these functions is a single `solution` object, which can be passed to the solver for the previous period.

- `time_inv`: A list of strings containing all of the variable names that are passed to at least one function in `solveAPeriod` but do *not* vary across periods. Each of these variables resides in a correspondingly named attribute of the `AgentType` instance.

- `time_vary`: A list of strings naming the attributes of this instance that vary across periods. Each of these attributes is a list of period-specific values, which should be of the same length. If the solution method varies across periods, then `'solveAPeriod'` is an element of `time_vary`.[3]

[2]Each instance of `AgentType` represents an *ex ante* heterogeneous "type" of agent; *ex post* heterogeneity is achieved by simulating many agents of the same type, each of whom receives a unique sequence of shocks.

[3]`time_vary` may include attributes that are never used by a function in `solveAPeriod`. Most saliently, the attribute `solution` is time-varying but is not used to solve individual periods.

- solution_terminal: An object containing the solution to the "terminal" period of the model. This might represent a known trivial solution that does not require numeric methods, the solution to some previously solved "next phase" of the model, a scrap value function, or merely an initial guess of the solution to an infinite horizon model.
- pseudo_terminal: A Boolean flag indicating that solution_terminal is not a proper terminal period solution (rather an initial guess, "next phase" solution, or scrap value) and should not be reported as part of the model's solution.
- cycles: A non-negative integer indicating the number of times the agent will experience the sequence of periods in the problem. For example, cycles = 1 means that the sequence of periods is analogous to a lifecycle model, experienced once from beginning to end; cycles = 2 means that the agent experiences the sequence twice, with the first period in the sequence following the last. An infinite horizon problem in which the sequence of periods is experienced indefinitely is indicated with cycles = 0.
- tolerance: A positive real number indicating convergence tolerance, representing the maximum acceptable "distance" between successive cycle solutions in an infinite horizon model; it is irrelevant when cycles > 0. As the distance metric on the space of solutions is model-specific, the value of tolerance is generally dimensionless.
- time_flow: A Boolean flag indicating the direction that time is "flowing." When True, the variables listed in time_vary are listed in ordinary chronological order, with index 0 being the first period; when False, these lists are in reverse chronological order, with index 0 holding the last period.

An instance of AgentType also has the attributes listed in time_vary and time_inv, and may have other attributes that are not included in either (e.g. values not used in the model solution, but instead to construct objects used in the solution).

2.2 A Universal Solver

When an instance of AgentType invokes the method solve(), the solution to the agent's problem is stored in the attribute solution. The solution is computed by recursively solving the sequence of periods defined by the variables listed in time_vary and time_inv using the functions in solveAPeriod. The time-varying inputs are updated each period, including the successive period's solution as solution_tp1; the same values of time invariant inputs in time_inv are passed to the solver in every period. The first call to solveAPeriod uses solution_terminal as solution_tp1. In a finite horizon problem, the sequence of periods is solved cycles times over; in an infinite horizon problem, the sequence of periods is solved until the solutions of successive cycles have a "distance" of less than tolerance.

The output from a function in solveAPeriod is an instance of a model-specific solution class. The attributes of a solution to one period of a problem might include behavioral functions, (marginal) value functions, and other variables characterizing the result. This class must have a method called distance(), which returns the "distance" between itself and another instance of the same solution class, so as to define convergence as a stopping criterion. For many models, this will be the "distance" between a behavioral function in the solutions (and thus a distance() method must be defined for the class that represents behavioral functions).

Our universal solver is written in a very general way that should be applicable to any discrete time optimization problem– because Python is so flexible in defining objects, the time-varying inputs for each period can take any form. Indeed, the solver does no "real work" itself, but merely provides a structure for describing models in the HARK framework, allowing interoperability among current and future modules.

2.3 The Flow of Time and Other Methods

Because dynamic optimization problems are solved recursively in our framework, it is natural to list time-varying values in reverse chronological order– the solve() method loops over the values in each time-varying list in the same direction that a human would read them. When simulating agents after the solution has been obtained, however, it is much more convenient for time-varying parameters to be listed in ordinary chronological order– the direction in which they will be experienced by simulated agents. To allow the user to set the order in which "time is flowing" for an instance of AgentType, the HARK framework includes functionality to easily change ordering of time-varying values.

The attribute time_flow is True if variables are listed in ordinary chronological order and False otherwise. AgentType has the following methods for manipulating time:

- timeReport(): Prints to screen a description of the direction that time is flowing, for interactive convenience and as a reminder of the functionality. Returns time_flow.
- timeFlip(): Flips the direction of time. Each attribute listed in time_vary is reversed in place, and the value of time_flow is toggled.
- timeFwd(): Sets the direction of time to ordinary chronological order.
- timeRev(): Sets the direction of time to reverse chronological order.

These methods are invoked to more conveniently access time-varying objects. When a new time-varying attribute is added, its name should be appended to time_vary, particularly if its values are used in the solution of the model (or is part of the solution itself). For example, the solve() method automatically adds the string 'solution' to time_vary if it is not already present. Note that attributes listed in time_vary *must* be lists if solve() or timeFlip() are used. Some values that could be considered "time varying" but are never used to solve the model are more conveniently represented as a numpy.array object (e.g. the history of a state or control variable from a simulation);

because the `numpy.array` class does not have a `reverse()` method, these attributes should not be listed in `time_vary`.

The base `AgentType` is sparsely defined, as most "real" methods will be application-specific. Two final methods bear mentioning. First, the `__call__()` method points to `assignParameters()`, a convenience method for adding or adjusting attributes. These methods take any number of keyword arguments, so that code can be parsimoniously written as, for example, `AgentInstance(attribute1 = value1, attribute2 = value2)`. Using Python's dictionary capabilities, many attributes can be conveniently set with minimal code. Second, the method `isSameThing()` takes two single period solution objects as arguments and returns `True` if these solutions are "identical" as far as this instance is concerned– i.e. they are no more than `tolerance` apart according to their `distance()` method.

2.4 Other HARK Modules

Beyond the economic agent framework in `HARKcore`, the toolKit includes several other primary modules with tools for a wide array of models.[4] Each of these modules is fairly small at the time of the "soft launch," but are expected to grow significantly in the near future.[5]

The `HARKutilities` module carries a double meaning in its name, as it contains both utility functions (and their derivatives) in the economic modeling sense as well as utilities in the sense of general purpose tools. Utility functions included at this time are constant relative risk aversion and constant absolute risk aversion; other forms may be added. The module also includes tools for constructing discrete approximations to continuous distributions (e.g. `calculateLognormalDiscreteApprox()` to approximate a log-normal distribution) as well as manipulating these representations (e.g. appending one outcome to an existing distribution, or combining independent univariate distributions into one multivariate distribution). `HARKutilities` also contains some data manipulation tools, functions for constructing discrete state space grids, convenience functions for retrieving information about functions, and convenience plotting tools using `matplotlib.pyplot`.

The `HARKsimulation` module provides tools for generating simulated data or shocks for post-solution use of models. Currently implemented distributions include normal, lognormal, Weibull (including exponential), uniform, Bernoulli, and discrete. In the consumption-saving model described below, these tools are used to simulate permanent and transitory income shocks as well as unemployment events.

Methods for optimizing an objective function for the purposes of estimating a model can be found in `HARKestimation`. As of this writing, the implementation includes only minimization by the Nelder-Mead simplex method, minimization by a derivative-free Powell method variant, and two small tools for resampling data (i.e. for a bootstrap); the minimizers are merely convenience wrappers (with result reporting) for optimizers

[4]Many of these tools, and their application to the consumption-saving model, are described in Carroll and Palmer (2015), which served as "version zero" of the toolKit.

[5]The "taxonomy" of these modules is also in flux; the functions described here could be combined into fewer modules or further divided by purpose.

included in `scipy.optimize`. Future functionality will include more robust global search methods, including genetic algorithms, simulated annealing, and differential evolution.

Finally, `HARKinterpolation` defines object classes for representing interpolated function approximations. The classes in this module all inherit the superclass `HARKinterpolator`, a wrapper class that ensures interoperability across interpolation methods, i.e. naming conventions for calls to evaluate the function or its derivative. Moreover, each subclass of `HARKinterpolator` must have a `distance()` method that compares itself to another instance of the same class. Currently implemented interpolation methods include a slight extension of `scipy.interpolate.UnivariateSpline` for linear interpolation, a custom cubic spline interpolator with exponential decay extrapolation to a limiting linear function, and a "constrained composite" class that combines a generic function with a linear constraint of slope 1. These methods are for representations of univariate functions only; functionality for multiple inputs or outputs is a near term priority for development.

3 An Application: Consumption-Saving Model

To provide a concrete example of the concepts of the preceding section, consider the case of a standard consumption-saving model, as described below; the `ConsumerType` class is defined as a subclass of `AgentType` in the `ConsumptionSavingModel` module.

3.1 Model Outline

Consumers are risk averse expected utility maximizers with time-separable additive utility. In period t, they receive a flow of utility from consumption through utility function $\mathrm{u}(\cdot)$ and discount future utility flows by factor β; the consumer will die with probability D_t and survive to $t+1$ with probability $\cancel{\mathsf{D}}_t = 1 - \mathsf{D}_t$. From their available 'market resources' M_t (the sum of current income and accumlated wealth), the consumer must choose how much to consume and how much to retain as end-of-period assets A_t subject to some asset floor (a budget or borrowing constraint). Retained assets grow by a known interest factor R_{t+1} between periods. At the beginning of period $t+1$, the consumer receives income subject to permanent and transitory shocks:[6]

$$\begin{aligned}
\mathrm{V}_t(M_t, P_t) &= \max_{C_t} \; \mathrm{u}(C_t) + \beta \cancel{\mathsf{D}} \mathbb{E}_t \left[\mathrm{V}_{t+1}(M_{t+1}, P_{t+1}) \right] \\
A_t &= M_t - C_t, \\
A_t &\geq \underline{A}_t, \\
M_{t+1} &= \mathsf{R} A_t + Y_{t+1}, \\
Y_{t+1} &= P_{t+1} \Xi_{t+1}, \\
P_{t+1} &= \Psi_{t+1} \Gamma_{t+1} P_t,
\end{aligned}$$

[6]To reduce clutter, we henceforth suppress time subscripts on β, $\cancel{\mathsf{D}}$, and R.

where the stochastic shocks Ξ and Ψ are drawn from

$$\Xi_{t+1} \sim \mathrm{F}_{\Xi_{t+1}}(\Xi), \qquad \Psi_{t+1} \sim \mathrm{F}_{\Psi_{t+1}}(\Psi).$$

When the utility function exhibits constant relative risk aversion[7] ($\mathrm{u}(\bullet) = \bullet^{1-\rho}/(1-\rho)$), the problem can be normalized by the agent's permanent income P_t, reducing the dimensionality of the state space and the complexity of the problem to:[8]

$$\begin{aligned}
\mathrm{v}_t(m_t) &= \max_{\mathrm{c}_t} \ \mathrm{u}(\mathrm{c}_t) + \beta \cancel{D} \mathbb{E}_t \left[(\Psi_{t+1}\Gamma_{t+1})^{1-\rho} \mathrm{v}_{t+1}(m_{t+1}) \right] \\
& \text{s.t.} \\
a_t &= m_t - c_t, \\
a_t &\geq \underline{a}_t, \\
m_{t+1} &= a_t \underbrace{\mathsf{R}/(\Psi_{t+1}\Gamma_{t+1})}_{\equiv \mathcal{R}_{t+1}} + \Xi_{t+1}.
\end{aligned}$$

The value of borrowing constraint $\underline{a}_t$ is either 0 (a liquidity constraint) or the natural borrowing constraint determined by an "always pay back" condition.

3.2 Attributes of a ConsumerType

When a `ConsumerType` is instantiated, the following attributes are listed in `time_vary`, representing values that take on different values during the consumer's problem:[9]

- `income_distrib`: A `numpy.array` with three rows representing a discretization of the income processes $\mathrm{F}_{\Xi_{t+1}}(\cdot)$ and $\mathrm{F}_{\Psi_{t+1}}(\cdot)$. Each column of the array represents a discrete outcome; the first element in the column gives is the probability, the second element is the permanent shock Ψ_t, and the third element is the transitory shock Ξ_t. The first row of the array should sum to one.

- `Gamma`: The expected growth factor for permanent income; usually a number slightly greater than 1, but any positive value can be used.

- `beta`: The discount factor for future utility after this period.[10]

- `survival_prob`: The probability of surviving to $t+1$ conditional on being alive at t.

- `p_zero_income`: The probability of receiving zero income next period, as $\Xi_{t+1} = 0$. This probability can be calculated from `income_distrib`, but is included as a separate attribute to avoid repeated computation.

[7]While `HARKutilities` includes other utility functions that could in theory be used with a consumption-saving model, the normalization assumed by the current solution methods is only compatible with CRRA.

[8]See SolvingMicroDSOPsfor details.

[9]The attributes themselves contain lists. The descriptions here are for *one element* of each list, i.e. the relevant value for one period.

[10]In many consumption-saving applications, the consumer uses the same discount factor for all periods. This can be achieved in HARK either by setting `beta` to a list with the same value in every element; or by removing `'beta'` from `time_vary`, adding it to `time_inv`, and setting `beta` to a single `float`.

The following attributes are listed in `time_inv` when a `ConsumerType` is instantiated, representing time-invariant values that are used in the solution to the problem:

- `rho`: The coefficient of relative risk aversion of the utility function $\mathrm{u}(\cdot)$.
- `R`: The interest factor for assets retained between periods (a number just above 1).
- `a_grid`: A 1D `numpy.array` of discrete asset levels at which the problem is solved (the solution is interpolated in between). These represent values of a_t above the minimum in period t when the ENDG solver is used and values of m_t above the minimum when the EXOG solver is used (see below).
- `constrained`: A Boolean that is `True` if the consumer is liquidity constrained– he must end each period with non-negative assets. When `False`, the consumer instead obeys the natural borrowing constraint.
- `cubic_splines`: A Boolean that is `True` if the interpolation should use cubic splines. When `False`, linear splines are used.
- `calc_vFunc`: A Boolean that is `True` if the user would like the agent's solution to include the (normalized) value function $\mathrm{v}_t(m_t)$. The value function is not needed to solve a consumption-saving model, but may be used for other purposes.

Each instance of `ConsumerType` has a pointer to a solution method function in the `solveAPeriod` attribute. The `ConsumptionSavingModel` module includes two solution methods that are valid values for `solveAPeriod`: a method that solves the problem at a fixed grid of m_t values `consumptionSavingSolverEXOG` and a solver that uses the method of endogenous gridpoints described in Carroll (2006) ('ENDG') `consumptionSavingSolverENDG` to find both m_t and c_t values. The solution method actually used by an instance of `ConsumerType` can be set by changing its `solveAPeriod` attribute. These methods are described in the following subsection.

3.3 Solution Methods

Both of the included `solveAPeriod` functions take as arguments all of the variables listed above in `time_vary` and `time_inv`, along with an additional argument of `solution_tp1` representing the solution to the subsequent period. At the very least, the object passed as `solution_tp1` must have an attribute called `vPfunc` that points to the marginal value function; if `cubic_splines` is `True`, it must also have the attribute `vPPfunc`.[11]

The standard approach to solving a single period consumption-saving problem is to fix a grid of m_t, then seek a solution in c_t to the first order condition at each of these

[11] These attributes are generated by the `ConsumptionSavingModel` solvers themselves, so the solution to the subsequent period will surely have these attributes in a normal model. However, the structure of HARK allows for the subsequent period to be *any* sort of problem, so long as its `solution` output includes `vPfunc`.

values:

$$c_t^{-\rho} = \beta \not{D} \mathsf{R}\, \mathbb{E}_t \left[(\Gamma_{t+1}\Psi_{t+1})^{-\rho} \mathrm{v}'_{t+1}(\underbrace{\mathcal{R}_{t+1}(m_t - c_t) + \Xi_{t+1}}_{=m_{t+1}}) \right].$$

The function `consumptionSavingSolverEXOG` uses this typical approach. For each value of m_t, the solver conducts an iterative search over values of c_t until the first order condition is satisfied to some numeric precision. At each "guess" of c_t, the expectation on the RHS of the FOC is computed using the discrete approximation to the income process in `income_distrib`. If the marginal value of consumption exceeds the discounted expected value of money in $t+1$, the guess is updated upward; if the marginal value of consumption is less than the RHS, c_t is updated downward. When the search converges, marginal consumption is also calculated if `cubic_splines` is `True`.

The single period problem can be solved much faster using the method of endogenous gridpoints (ENDG) to avoid repeated computation of the expectation of future marginal value. Slightly rewrite the first order condition as:

$$c_t = \left(\beta \not{D} \mathsf{R}\, \mathbb{E}_t[(\Gamma_{t+1}\Psi_{t+1})^{-\rho} \mathrm{v}'_{t+1} \mathcal{R}_{t+1} a_t + \Xi_{t+1})] \right)^{-1/\rho}.$$

The function `consumptionSavingSolverENDG` computes the expectation of future marginal value exactly *once* for each value of end-of-period assets a_t in `a_grid`, inverting the marginal value function to find c_t; the level of money resources from which a_t was reached is then calculated as $m_t = a_t + c_t$. That is, the method of endogenous gridpoints asks what the agent must have *just consumed* for ending the period with a_t to *have been* optimal.

When `cubic_splines` is `True`, the solvers must find marginal consumption as well in order to construct a cubic interpolation of $\mathbf{c}_t(m)$. Taking the derivative of the EXOG form of the FOC with respect to a_t, we find:

$$\begin{aligned} \frac{\partial c_t}{\partial a_t} \mathrm{u}''(c_t) &= \beta \not{D} \mathsf{R}^2\, \mathbb{E}[(\Gamma_{t+1}\Psi_{t+1})^{-\rho-1} \mathrm{v}''(m_{t+1})] \Longrightarrow \\ \frac{\partial c_t}{\partial a_t} &= \beta \not{D} \mathsf{R}^2\, \mathbb{E}[(\Gamma_{t+1}\Psi_{t+1})^{-\rho-1} \mathrm{v}''(m_{t+1})] / \mathrm{u}''(c_t). \end{aligned}$$

The marginal propensity to consume can then be calculated as:

$$\kappa_t \equiv \frac{\partial c_t}{\partial m_t} = \frac{\frac{\partial c_t}{\partial a_t}}{\frac{\partial c_t}{\partial a_t} + 1}.$$

The values of c_t, m_t, and κ_t can then be used to construct a cubic spline interpolation of the consumption function $\mathbf{c}_t(m)$ (or a linear interpolation if κ_t is not computed) and stored in the attribute `cFunc` of the `solution_t` object.

The envelope condition of a consumption-saving problem is very straightforward: $\mathrm{v}'_t(m_t) = \mathrm{u}'(\mathbf{c}_t(m_t))$. The marginal value function `vPfunc` can be easily constructed as the composition of `cFunc` and the marginal utility function. When `cubic_splines` is true, the marginal marginal value function can be constructed by taking the derivative of the envelope condition: $\mathrm{v}''_t(m_t) = \mathbf{c}'_t(m_t)\mathrm{u}''(\mathbf{c}_t(m_t)) = \kappa_t \mathrm{u}''(c_t)$; a representation of this function is stored in the attribute vPPfunc of the solution_t object.

For reasons explained in Carroll (2014) (available online at SolvingMicroDSOPs), the single period solution object returned by the solvers (solution_t) also includes as attributes the bounding MPCs of the consumption function kappa_min and kappa_max, the lowest level of m_t allowable in this period m_underbar, and expected human capital gothic_h. If calc_vFunc is True, the value function is also computed and included in the vFunc attribute of solution_t.

3.4 Constructing the Income Process

Recall that the attribute income_distrib is a three-rowed array of probabilities, permanent shocks, and transitory shocks; this array may be time-varying in a lifecycle consumption-saving model. Rather than have the user tediously build each one, HARK includes tools for constructing these discrete approximations to the income process. By default, the module uses the function constructLognormalIncomeProcessUnemployment() to construct the income process in each period (using tools from HARKutilities, filling in the income_distrib attribute. As all of the inputs for this function reside as attributes of a ConsumerType, the instance itself is passed to the income process constructor).

The function assumes that the distribution of permanent shocks $\mathrm{F}_{\Psi_t}(\cdot)$ is mean one lognormally distributed, and that the transitory shock process $\mathrm{F}_{\Xi_t}(\cdot)$ is a lognormal $\mathrm{F}_{\Theta_t}(\cdot)$ that applies if the consumer is employed, augmented by a point mass representing unemployment. Time-varying permanent shock standard deviations are stored in the attribute psi_sigma, while transitory variances are listed in xi_sigma; the number of points in each discrete approximation is stored in psi_N and xi_N. The unemployment process is recorded in the attributes p_unemploy and income_unemploy, representing the unemployment probability and the income replacement rate when unemployed. If the consumer retires from the workforce in the model (ending permanent and most transitory shocks), the retirement period is recorded in T_retire; the "unemployment" process after retirement is summarized in p_unemploy_retire and income_unemploy_retire.[12]

ConsumptionSavingModel includes at least one other discrete income process constructor, constructLognormalIncomeProcessUnemploymentFailure(). This function is identical to the one above but allows for unemployment benefits to "fail" with a very small probability, resulting in zero transitory income. This is useful for accounting for transitory shocks below the smallest shock, but can cause numeric oddities. The user can also easily add a new function to approximate an alternate income process– perhaps with an age-varying unemployment process, phased retirement, or uniformly distributed income shocks.

To generate an array of shocks for simulating agents of some ConsumerType, the module also includes functions to generate arrays of shock histories. In the default setup, the function generateIncomeShockHistoryLognormalUnemployment()

[12]Retirement can be "turned off" in the income process by setting T_retire = 0.

creates `numpy.array` objects for transitory and permanent shocks using the same attributes of the `ConsumerType` as the discrete income process approximation.[13] To construct shocks for an infinite horizon consumption-saving model, the function `generateIncomeShockHistoryInfiniteSimple()` can be used to make shock draws from the same distribution of permanent and transitory shocks for many periods. A user can also write shock generators for other income processes.

3.5 Other Methods

Other methods of `ConsumerType` include the following:

- `updateIncomeProcess()`: Creates a new timepath of discrete approximations to the income process based on attributes of the instance. In the distribution, these are constructed using the function `constructLognormalIncomeProcessUnemployment()`. This function can be swapped out for another that constructs a different type of lifecycle– perhaps one with phased retirement rather than sudden retirement, or one with only two discrete transitory outcomes– perhaps using different attributes.

- `updateAssetsGrid()`: Creates a new grid of asset levels using the attributes `a_grid_size`, `a_min`, `a_max`, and `a_extra`, storing the result in the `a_grid` attribute.

- `update()`: Runs the methods `updateIncomeProcess()` and `updateAssetsGrid()`, and updates `solution_terminal`. When one or more of the attributes used to construct the asset grid or income process is changed, invoking this method ensures that the constructed objects are up to date with the instance's attributes.

- `unpack_cFunc()`: "Unpacks" the consumption functions stored in each element of the agent's `solution` attribute (in each element's `cFunc` attribute) into a new list stored in the attribute `cFunc`. This is for convenient access when simulating consumers. [14]

- `addIncomeShockPaths(perm_shocks,temp_shocks)`: Adds the attributes `perm_shocks` and `temp_shocks` to this instance, representing sets of permanent and transitory shocks that could be experienced by this consumers of this type.

- `simulate(w_init,t_first,t_last,which)`: Simulates consumers of this type starting from initial wealth-to-permanent-income ratios `w_init`, from period `t_first` to period `t_last` (utilizing the permanent and transitory shocks). The method returns `numpy.array` objects with histories of wealth-to-permanent income ratios w_t, normalized money resources m_t, consumption levels c_t, end-of-period assets a_t, or marginal propensities to consume κ_t depending on the strings in the list `which`.

[13]Additional attributes used to construct shocks include seeds for the random number generators (`temp_seed`, `perm_seed`, and `unemp_seed`) as well as the number of consumers to simulate `Nagents`.

[14]This method adds the string `'cFunc'` to the `time_vary` attribute, so that the consumption functions are reversed in order when time flips for this instance.

3.6 Consumption-Saving Example Models

The alpha version of HARK available at the soft launch includes three applications for implementing models based on `ConsumptionSavingModel`. Starting simple, the module `ConsumerExamples` does nothing but create three instances of `ConsumerType`, solve each one, and plot the consumption functions. The baseline lifecycle consumer type[15] has a 40 period working life followed by a 25 period retirement (during which he might die); the consumption functions for these phases are plotted separately. The infinite horizon consumer type experiences the same period repeatedly (`cycles=0`), from the perspective of the distribution of income shocks; the consumption and marginal consumption functions are plotted in two figures. Third, the cyclical consumer type has an infinite horizon, but experiences a "cycle" of income growth factors every four periods; this might represent a ski instructor who earns over half his income in one quarter of each year, settling for odd jobs in the other three quarters. These examples are merely illustrative of the variety of models incorporated by `ConsumptionSavingModel`.

The second application is `SolvingMicroDSOPs`, based on the set of SolvingMicroDSOPs of the same name. This module structurally estimates the coefficient of relative risk aversion ρ and time-varying discount factor[16] shifter $\beth$ that cause simulated lifecycle agents to best match the profile of median wealth holdings in the Survey of Consumer Finance over ages 25-60. Three Booleans at the beginning of the module determine the activities conducted, including estimating the model, computing standard errors by bootstrap, and creating a contour plot of the objective function.

Finally, the `cstwMPC` module estimates various specifications of the models described in Carroll, Slacalek, Tokuoka, and White (2015) including both perpetual youth and lifecycle models (but omitting general equilibrium versions with aggregate shocks). Model primitives are established by the module `SetupParamsCSTW`, which also allows the user to choose which model to use (perpetual youth vs lifecycle; β-point vs β-dist, net worth vs liquid assets) and which activities to execute (model estimation, sensitivity analysis, figure construction).

4 Looking Ahead to HARK Beta

It is both an understatement and vacuously true that the alpha version of HARK is incomplete– our vision of HARK is that it will never be "complete" but rather continuously expanded and extended by users. However, we do have concrete plans for the modules, models, and tools that we would like to have available when the toolKit enters a public beta release (with a target of roughly June 2016). This section provides an overview of the functionality we hope to offer at that time.

A top priority currently under development is convenient tools for multithreading. In keeping with the project's focus on heterogeneity, we would like users to be able to execute the same command on a list of `AgentTypes` by distributing the work across

[15] Parameters for the consumer types are defined in `SetupConsumerParams`.

[16] Consumers are assumed to have an age-varying path of β, as calibrated by Cagetti (2003) based on household size and other demographics. The parameter $\beth$ is a multiplicative shifter for the entire path of β.

multiple threads or processes (e.g. have eight instances run the `solve()` or `simulate()` method on eight threads), rather than loop over each type and use only a fraction of the CPU's power. Ideally, this would be done with minimal fuss on the front end. Python has multithreading packages in development, but we are not yet ready to demonstrate our tools in this area.[17]

The only models currently available in HARK are variations on the traditional consumption-saving setting. As such, many of the supporting tools implicitly assume a one dimensional state space and a single control variable. We are beginning to develop functionality to incorporate multiple states and/or controls in a HARK model, including ND interpolation methods. This would allow users to solve such extensions as portfolio allocation, a durable housing stock, or health investment decisions. Our plans include a framework for discrete states, whether endogenously chosen by agents or assigned through an exogenous process. A bit further in the future, we would like to include a framework for dynamic discrete choice problems, eliminating the redundant recoding of a standard environment.

As many models of interest incorporate general equilibrium features, where some macroeconomic aggregation of individual agent states and actions "feeds back" into the microeconomic problem of each agent, we are developing tools for forming and representing agent beliefs about macro dynamics. This might possibly include a "universal general equilibrium solver" as a parallel to the "universal microeconomic agent solver" of `AgentType`, creating a common framework for such models. At the very least, the beta version of HARK will implement the classic Krusell and Smith (1998) model.

Ultimately, the scope of HARK is limited only by what users are willing to build and contribute to the project. To foster these contributions, we want to provide a core set of tools so that reproducing and extending canonical models is as easy as possible for users. As the audience for our "soft launch" is composed of researchers that we believe to be a potential core user-base of HARK, we are keenly interested in the types of models that you would like to to build with it. Open source projects require a critical mass of interest to be sustainable, so the better targeted our development priorities are to suit potential users, the more likely the project is to succeed in the long run. If you have any feedback or suggestions, please do not hesitate to contact any of the project principals.

[17]The `cstwMPC` module includes a developmental version of multithreading in HARK, using the scheduler package `dill`. We leave it to users in the audience to determine how to active the dormant code.

References

CAGETTI, MARCO (2003): "Wealth Accumulation Over the Life Cycle and Precautionary Savings," *Journal of Business and Economic Statistics*, 21(3), 339–353.

CARROLL, CHRISTOPHER D. (2006): "The Method of Endogenous Gridpoints for Solving Dynamic Stochastic Optimization Problems," *Economics Letters*, pp. 312–320, http://econ.jhu.edu/people/ccarroll/EndogenousGridpoints.pdf.

——— (2014): "Solving Microeconomic Dynamic Stochastic Optimization Problems," *Archive, Johns Hopkins University*, http://econ.jhu.edu/people/ccarroll/SolvingMicroDSOPs/.

CARROLL, CHRISTOPHER D, AND NATHAN M PALMER (2015): "The Heterogeneous-Agent Computational toolKit: An Extensible Framework for Solving and Estimating Heterogeneous-Agent Models," Discussion paper, Johns Hopkins University and Office of Financial Research, Available at https://editorialexpress.com/cgi-bin/conference/download.cgi?db_name=CEF2015&paper_id=523

CARROLL, CHRISTOPHER D., JIRI SLACALEK, KIICHI TOKUOKA, AND MATTHEW N. WHITE (2015): "The Distribution of Wealth and the Marginal Propensity to Consume," Draft, Johns Hopkins University, At http://econ.jhu.edu/people/ccarroll/papers/cstwMPC (this is a revision of cstMPC).

KRUSELL, PER, AND ANTHONY A. SMITH (1998): "Income and Wealth Heterogeneity in the Macroeconomy," *Journal of Political Economy*, 106(5), 867–896.